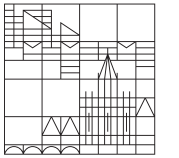
Universität Konstanz



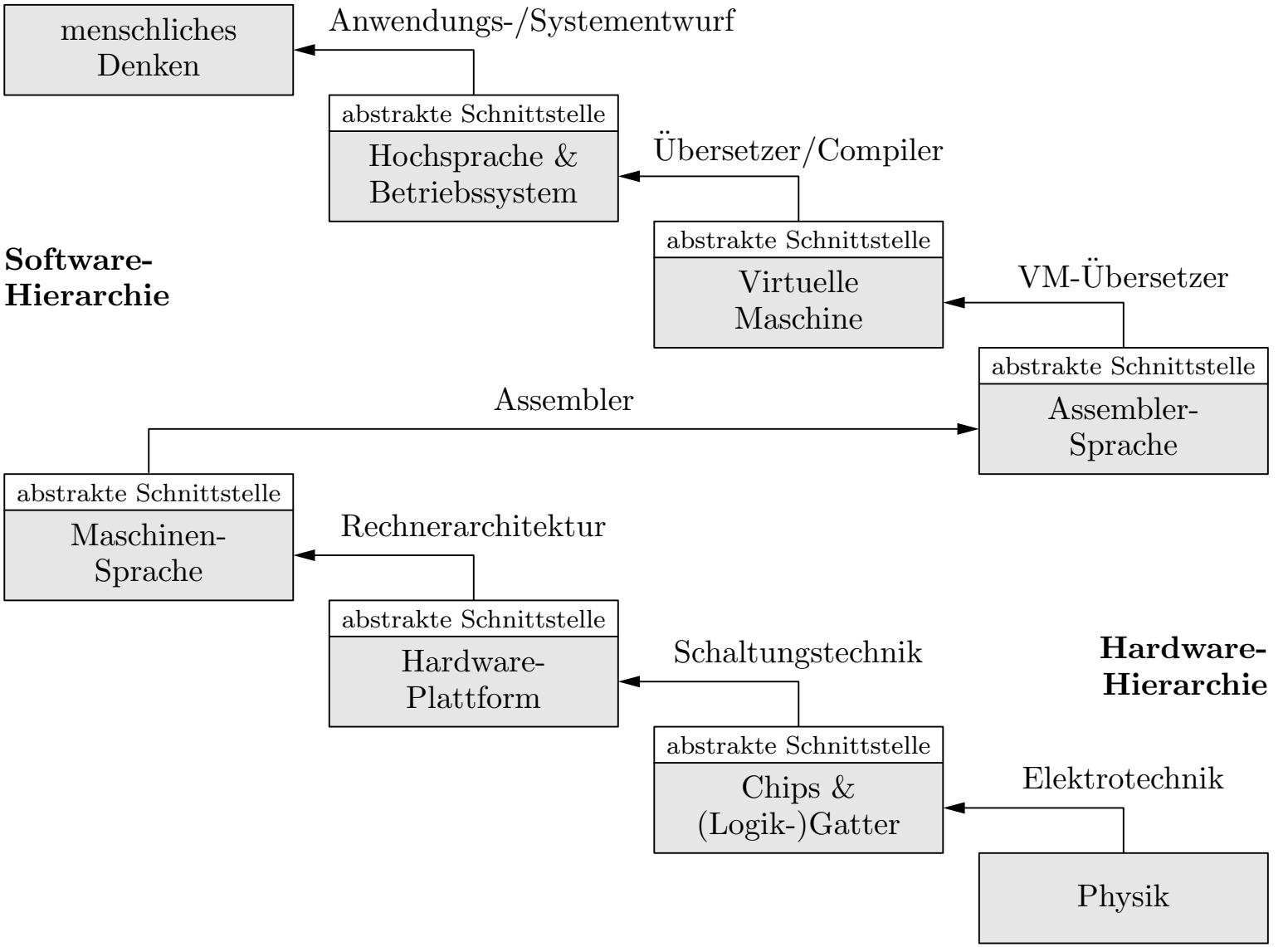
Rechnersysteme und -netze
Kapitel 6
Maschinensprache und Assembler

Bastian Goldlücke

Universität Konstanz

WS 2020/21

Rechnersysteme: Plan der Vorlesung



Erinnerung: Schaltungstechnik III

- **Arithmetik** (im wesentlichen Erinnerung an Bekanntes)
  - Zahlensysteme (Basis 10, andere Basen als 10)
  - Addition und Subtraktion, Übertrag
  - Multiplikation und Division, Stellenprodukte
  - Implementierung durch mechanische Rechner

- **Arithmetisch-logische Einheit** (Arithmetic Logic Unit, ALU)
  - Binärarithmetik (Rechnen im Zahlensystem mit Basis 2)
  - Halbaddierer und Volladdierer (1-Bit-Addierer)
  - *n*-Bit-Addierer (mit Übertragskette und Übertragsauswahl)
  - Multiplikation: Bit-Schieben, Standardalgorithmus
  - Multiplikation: negative Zahlen, Booths Algorithmus
  - Hack-Architektur (arithmetisch-logische Einheit, Prozessor)

Erinnerung: Schaltungstechnik IV

- **Sequentielle Logik**
  - Kombinatorische und sequentielle Logik
  - Schaltnetze und Schaltwerke
  - Rückkopplung (feedback) und Taktsignal (clock)
- **Bistabile Kippstufen („Flipflops“)**
  - SR-Riegel (SR latch) als Grundbaustein
  - D-, E-, T- und JK-Riegel (D, E, T, JK latch)
  - Taktpegel- und Taktflankensteuerung (latches vs. flip flops)
  - Master-Slave-Riegel und -Flipflops
- **Register, Zähler und Speicher**
  - Schieberegister, Parallel-Seriell-Wandler, Zähler
  - Speicherregister, Hauptspeicherorganisation

Erinnerung: Rechnerarchitektur

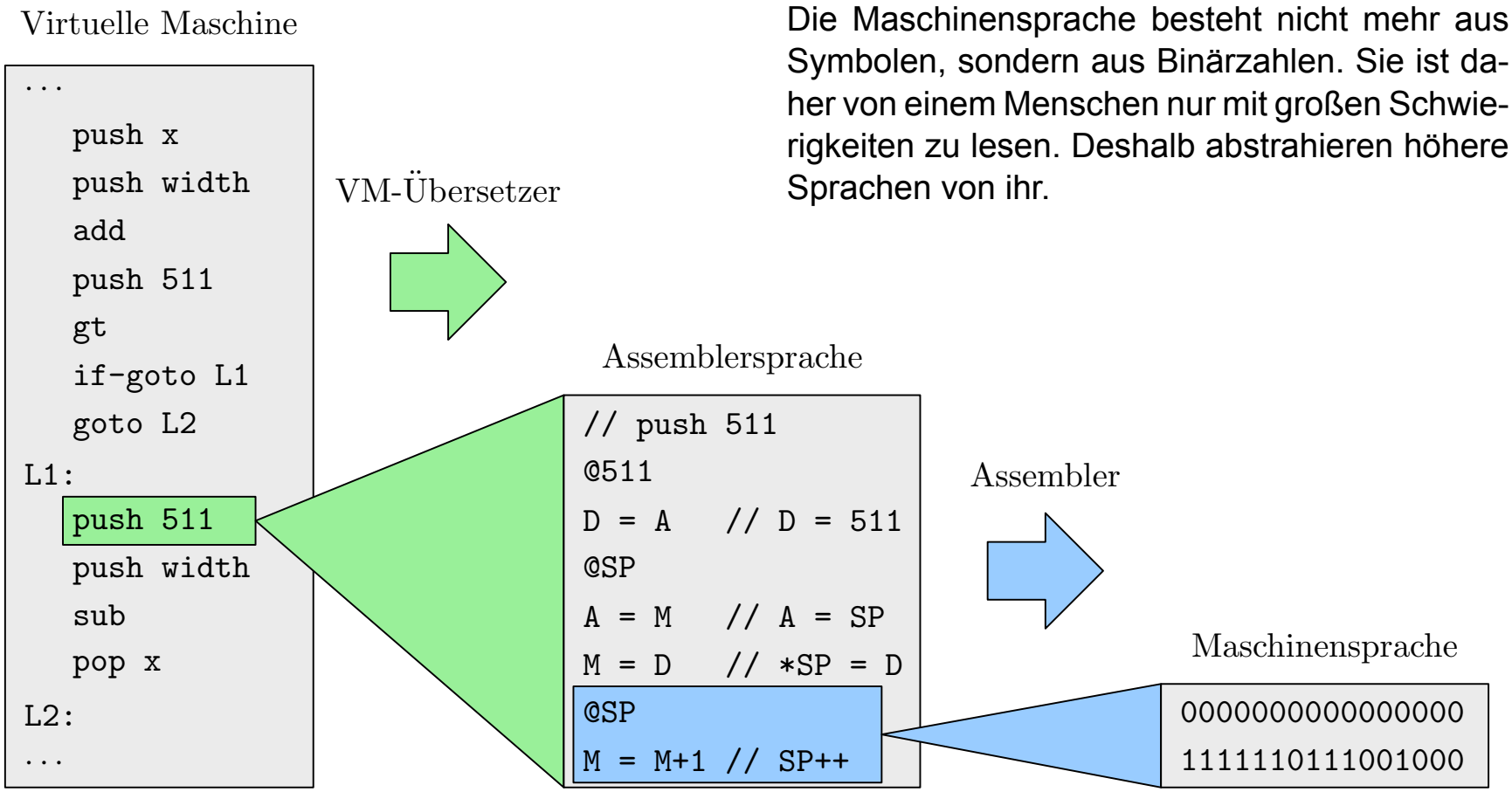
- **Speicherprogrammierung**
  - Festverdrahtete „Prozessoren“
  - Konzept der Speicherprogrammierung (stored program concept)
  - Befehlsabruf, -dekodierung und -ausführung (fetch-decode-execute cycle)
  - Rechnerarchitekturen (Harvard und von Neumann)

- **Die Hack-Plattform**
  - Befehls- und Datenspeicher (ROM32K und RAM16K)
  - Bildschirm und Bildschirmspeicher (screen)
  - Tastatur (keyboard)
  - Hauptspeicherorganisation (memory)
  - Prozessor (central processing unit, CPU)
  - Gesamtsystem (computer on a chip)

Inhalt

Inhalt

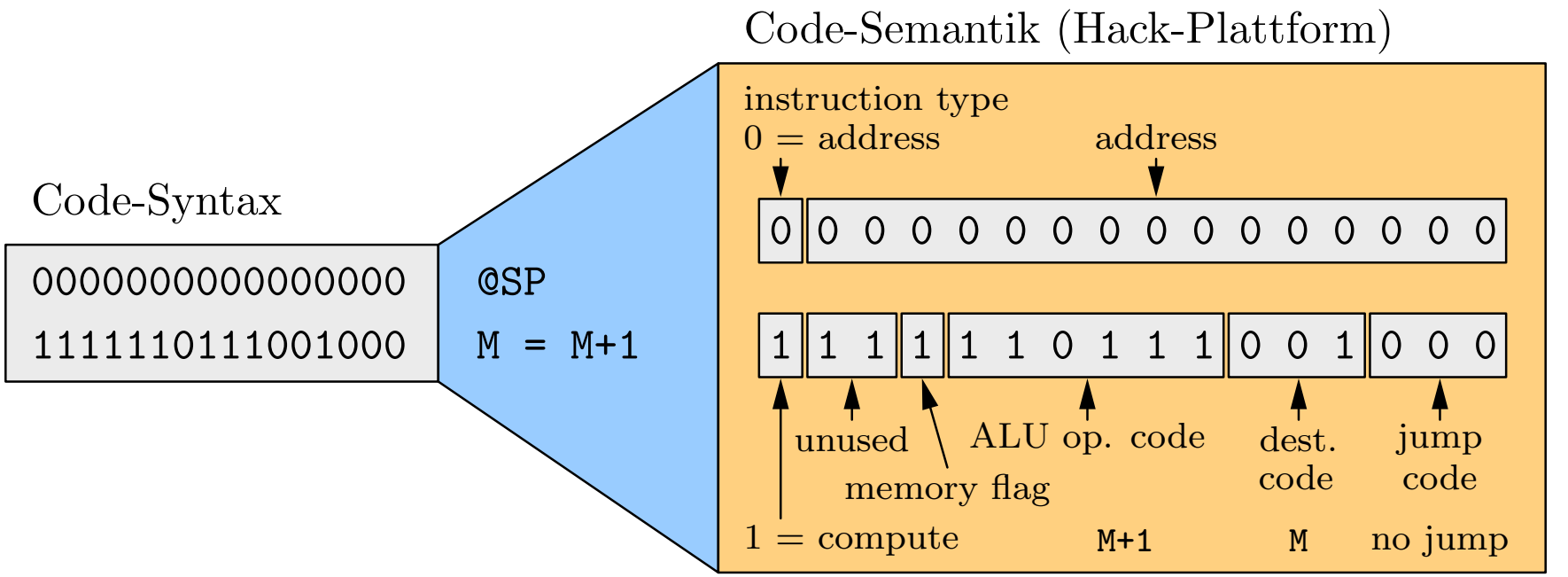
Erinnerung: Assembler und Maschinensprache



Die Maschinensprache besteht nicht mehr aus Symbolen, sondern aus Binärzahlen. Sie ist daher von einem Menschen nur mit großen Schwierigkeiten zu lesen. Deshalb abstrahieren höhere Sprachen von ihr.

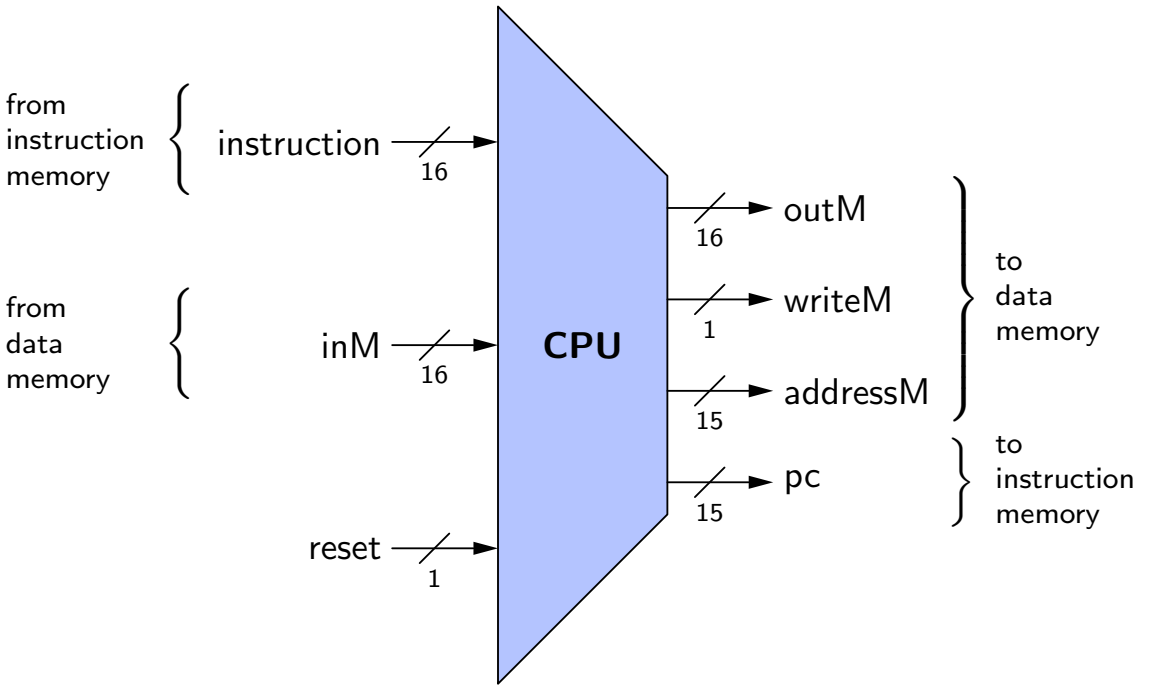
- Die **Maschinensprache** kann von einem Rechner direkt ausgeführt werden!

Erinnerung: Semantik der Maschinensprache



- Benötigt: **Hardware-Plattform**, die diese Semantik realisiert.
- Die Hardware-Plattform sollte in der Lage sein,
  - die Befehle zu interpretieren (gemäß obiger Semantik)
  - und auszuführen (entsprechende Operationen durchführen).

Hack-Architektur: Prozessor (CPU)

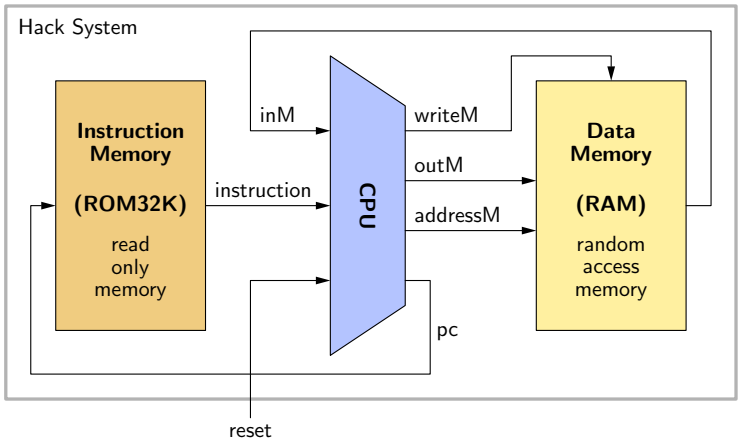


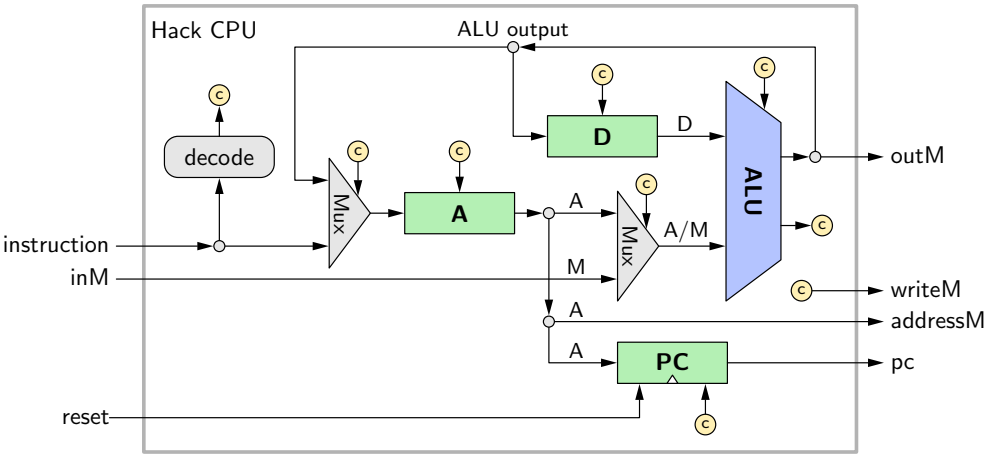
Bestandteile der CPU:

- arithmetisch-logische Einheit (ALU)
- A-Register
- D-Register
- PC-Register (programm counter)
- Steuereinheit (Befehlsdekodierung)

- Die CPU führt Anweisungen gemäß der Hack-Maschinensprache-Spezifikation aus.
  - Die Anweisung wird über `instruction`, der M-Wert über inM gelesen.
  - Die D- und A-Werte werden aus den/in die Registern gelesen/geschrieben.
  - Ggf. wird `outM` auf den M-Wert und `addressM` auf den A-Wert gesetzt.
  - Falls `reset=1`, setzte das PC-Register auf 0, sonst auf den nächsten Befehl.

Hack-System: Programmausführung



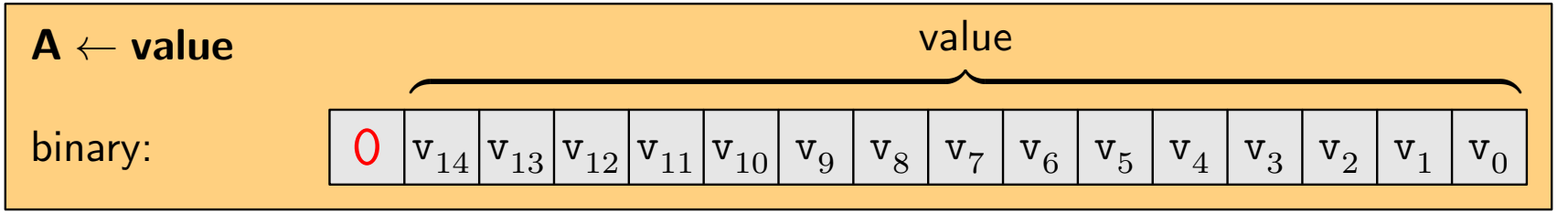


- Das `ROM` (read only memory) ist der Befehlsspeicher.
- Ein Programm ist eine Folge von 16-Bit-Zahlen, beginnend bei `ROM[0]`.
- Das `ROM` wird über das PC-Register (program counter) angesprochen.
  Das PC-Register enthält die Speicheradresse der nächsten Anweisung.
- Eine Sprunganweisung überträgt den Wert des A-Registers in das PC-Register.

  D.h., bei einem Sprung wird das Programm mit der Anweisung fortgesetzt,
  die in der Speicherzelle `ROM[PC] = ROM[A]` abgelegt ist.

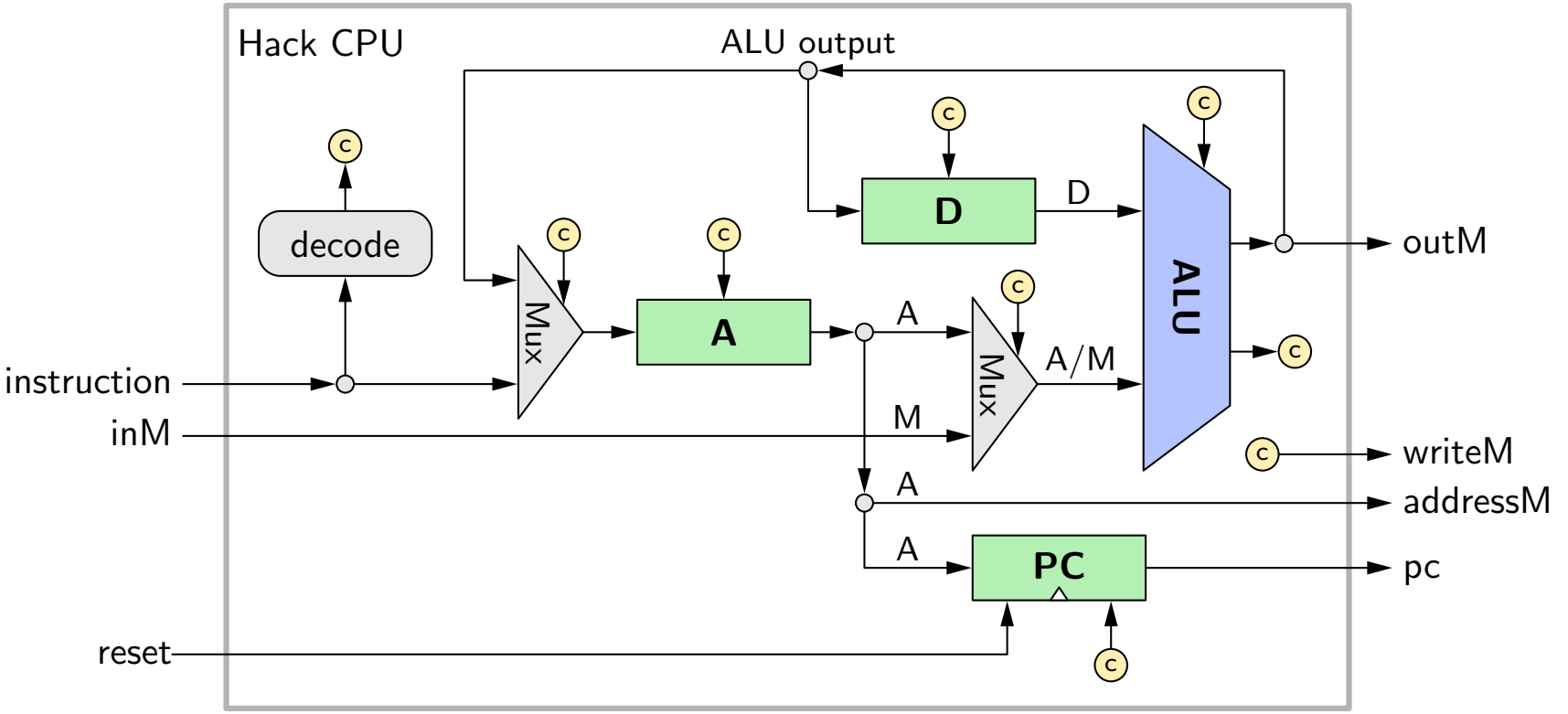
Hack-Maschinensprache: A-Anweisungen

- Die Hack-Maschinensprache besteht aus nur zwei Arten von Anweisungen:
  - **A-Anweisungen** (address instructions oder kurz A instructions)
  - **C-Anweisungen** (compute instructions oder kurz C instructions)



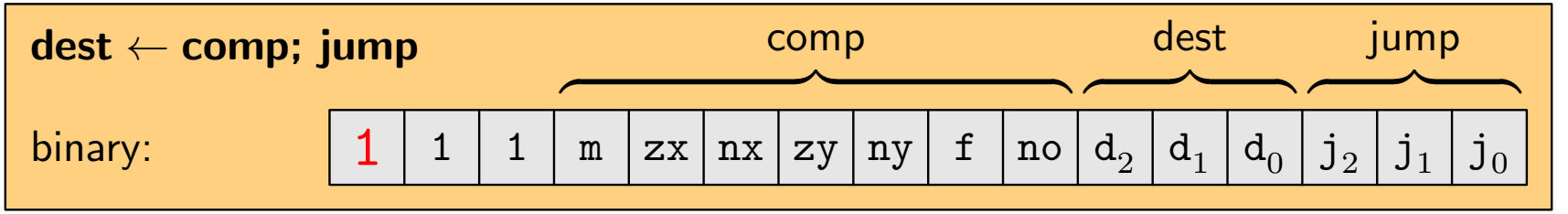
- Eine **A-Anweisung** lädt einen (konstanten) 15-Bit-Wert in das A-Register.
- Dieser Wert kann später verwendet werden als
  - Datenspeicheradresse,
  - neuer Wert des Befehlszählers (program counter, PC),
  - Wert der in eine Berechnung der arithmetisch-logischen Einheit (ALU) eingeht.

Hack-Maschinensprache: Prozessor (CPU)



- Das oberste Bit der A-Anweisung beeinflußt den Multiplexer vor dem A-Register (Auswahl der Leitungen instruction) und das Steuerbit des A-Registers (den Eingang load zur Übernahme der Adresse aus dem Befehl).

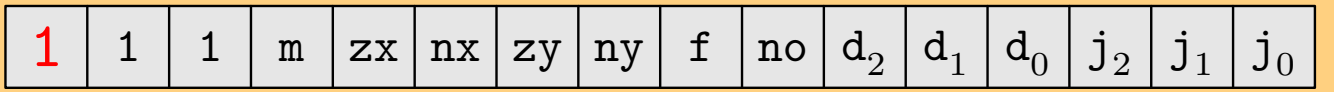
Hack-Maschinensprache: C-Anweisungen



- Die **C-Anweisung** führt eine Berechnung mit Hilfe der arithmetisch-logischen Einheit (ALU) durch.
- Die auszuführende Berechnung (computation oder kurz comp) ist in den Bits `m`, `zx`, `nx`, `zy`, `ny`, `f` und `no` kodiert.
- Die Bits d_2, d_1 und d_0 (destination oder kurz dest) bestimmen, wo das Ergebnis abgespeichert werden soll.
- Die Bits j_2, j_1 und j_0 (jump) bestimmen, ob und unter welchen Bedingungen gesprungen (verzweigt) werden soll.
- Man beachte, daß man am ersten (höchstwertigsten) Bit ablesen kann, ob es sich um eine A-Anweisung (0) oder um eine C-Anweisung (1) handelt.

Hack-Maschinensprache: C-Anweisungen

dest ← comp; 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/!D	0	0	1	1	0	1	
~A/!A	1	1	0	0	0	1	~M/!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

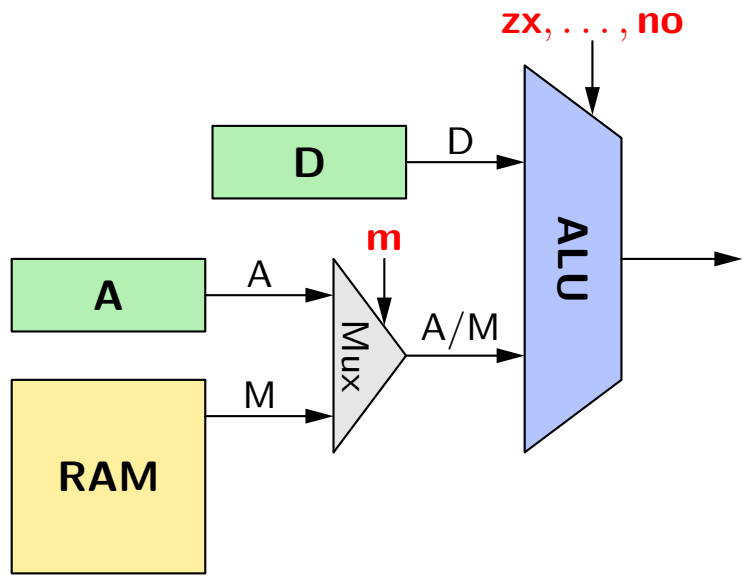
`Memory[A]` ist die durch das A-Register adressierte Speicherzelle.

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 $\neq 0$
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.

Erinnerung: ALU-Steuerung

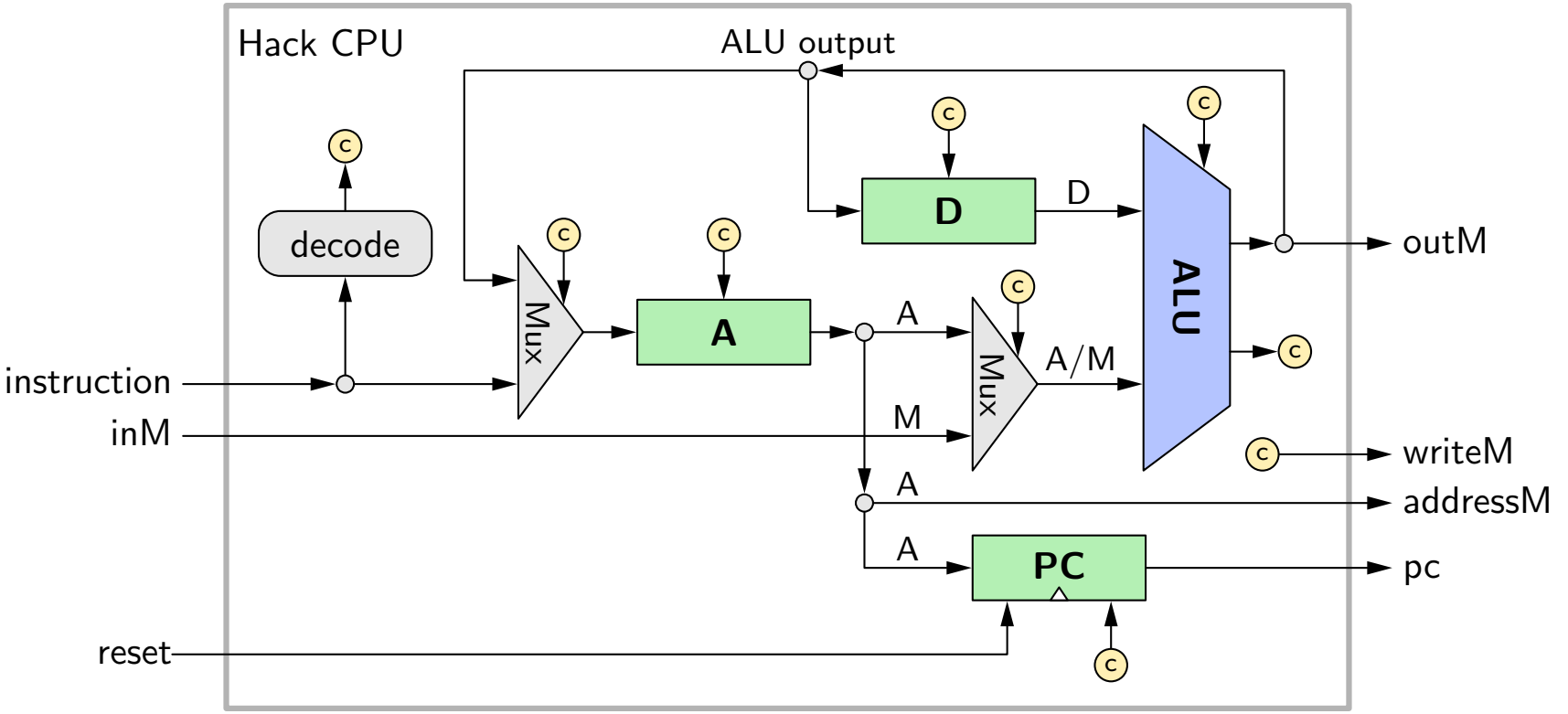
Berechnung wenn m = 0	zx	nx	zy	ny	f	no	Berechnung wenn m = 1
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
~D/!D	0	0	1	1	0	1	
~A/!A	1	1	0	0	0	1	~M/!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	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`, `~A`, `~M` (bzw. `!D`, `!A`, `!M`) berechnen das Einerkomplement der Register D, A bzw. einer Speicherzelle M.

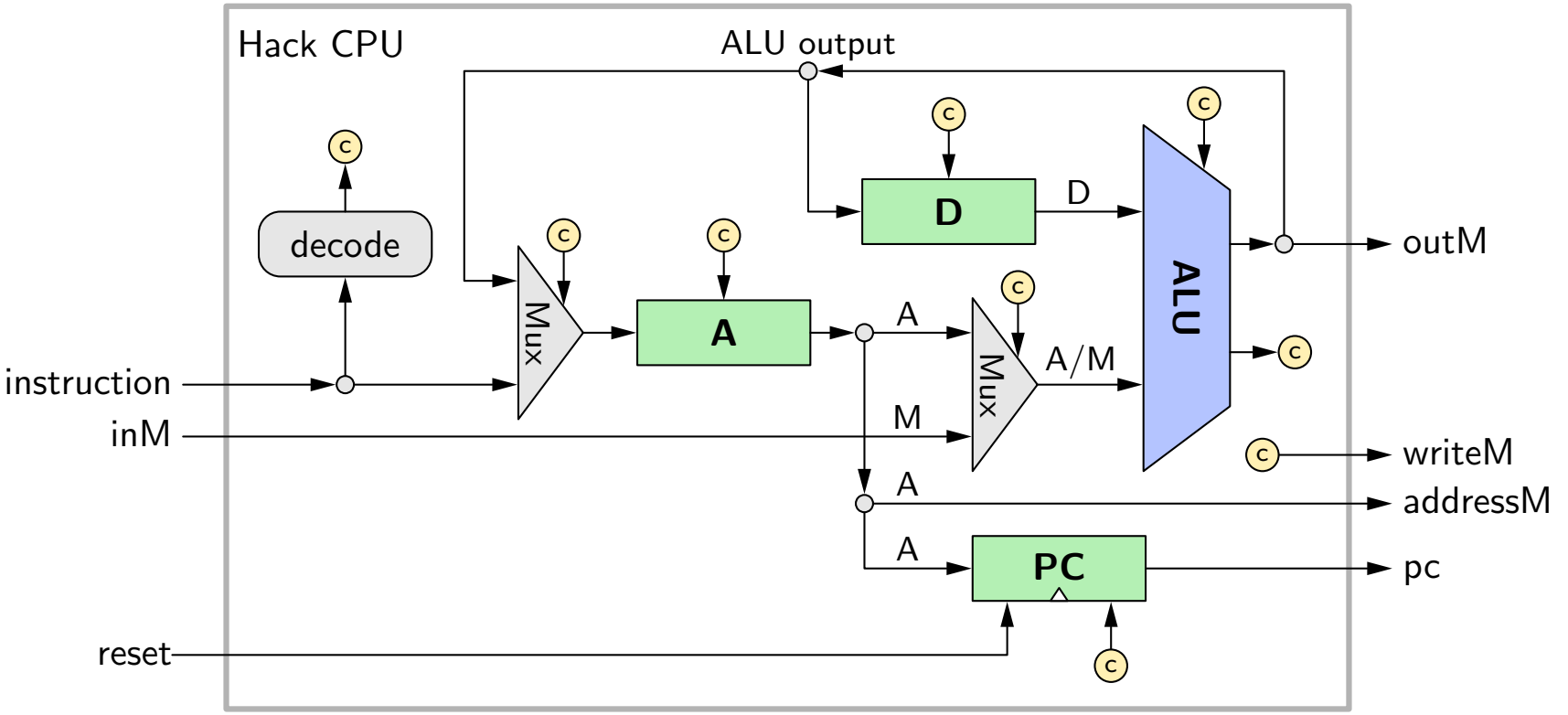
`D&A` und `D&M` bzw. `D|A` und `D|M` berechnen ein bitweises Und bzw. ein bitweises Oder des Registers D mit dem Register A bzw. einer Speicherzelle M.

Hack-Maschinensprache: Prozessor (CPU)



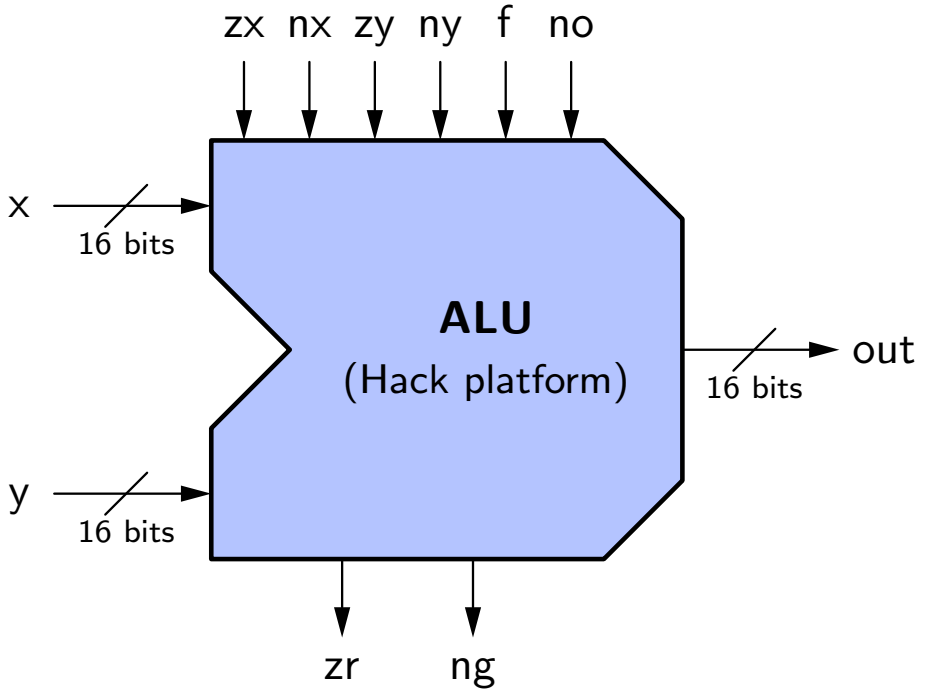
- Die dest-Bits der C-Anweisung beeinflussen die Steuerbits der D- und A-Registers, den Multiplexer vor dem A-Register und die Ausgabe writeM (write to memory). Je nach Steuerbits kann in einige oder alle dieser Ziele geschrieben werden.

Hack-Maschinensprache: Prozessor (CPU)



- Die jump-Bits einer C-Anweisung beeinflussen zusammen mit den Ausgaben zr und ng der arithmetisch-logischen Einheit (arithmetic logic unit, ALU) die Steuerbits des PC-Registers (programm counter, ggf. Adreßübernahme aus A-Register).

Erinnerung: Arithmetisch-Logische Einheit



ALU der Hack-Platform
(16 Bit)

Die Ausgaben zr und ng der ALU zeigen Eigenschaften des berechneten Ergebnisses an:

Es ist zr $= 1$, wenn out $= 0$.
(zero flag)

Es ist ng $= 1$, wenn out < 0.
(negative flag)

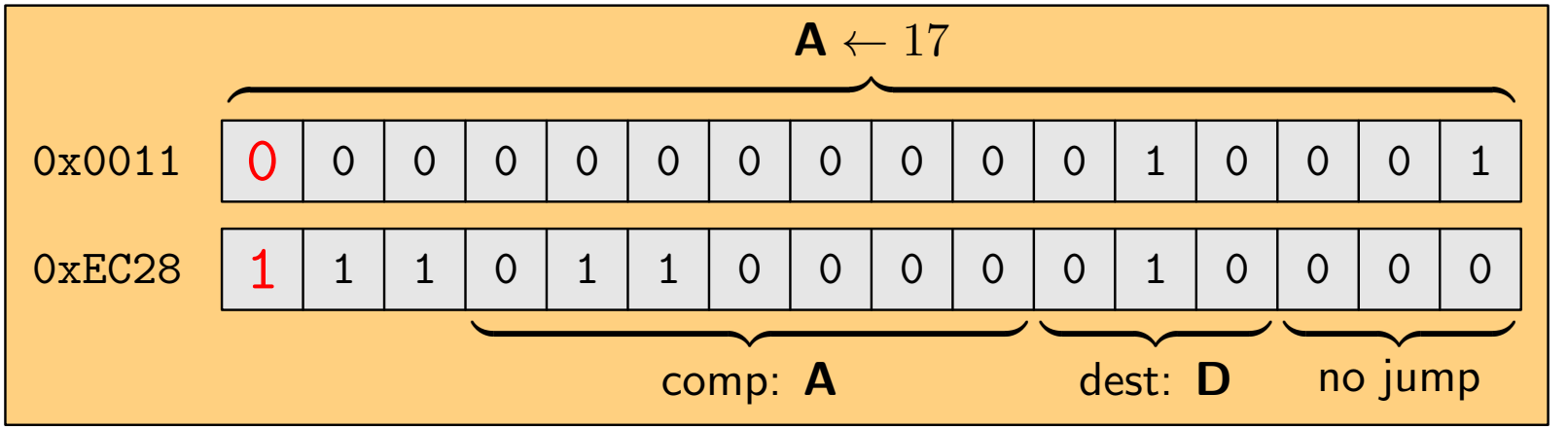
Über-/Unterlauf wird ignoriert.

Die Eingaben zx, nx, zy, ny, f und no kodieren die auszuführende Operation:

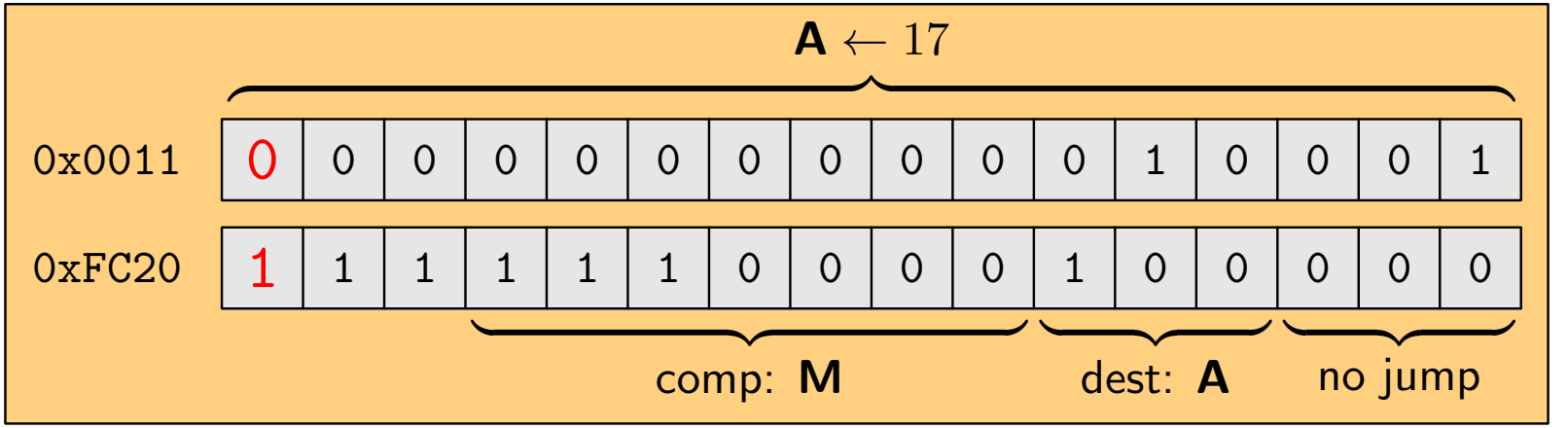
zx	Setze Eingabe x $= 0$.	nx	Bilde Einerkomplement der Eingabe x.
zy	Setze Eingabe y $= 0$.	ny	Bilde Einerkomplement der Eingabe y.
f	Wählt zwischen Addition und bitweisem Und als Operation.		
no	Bilde Einerkomplement der Ausgabe out.		

Hack-Maschinensprache: Beispiele

Lade den Wert 17 in das D-Register.

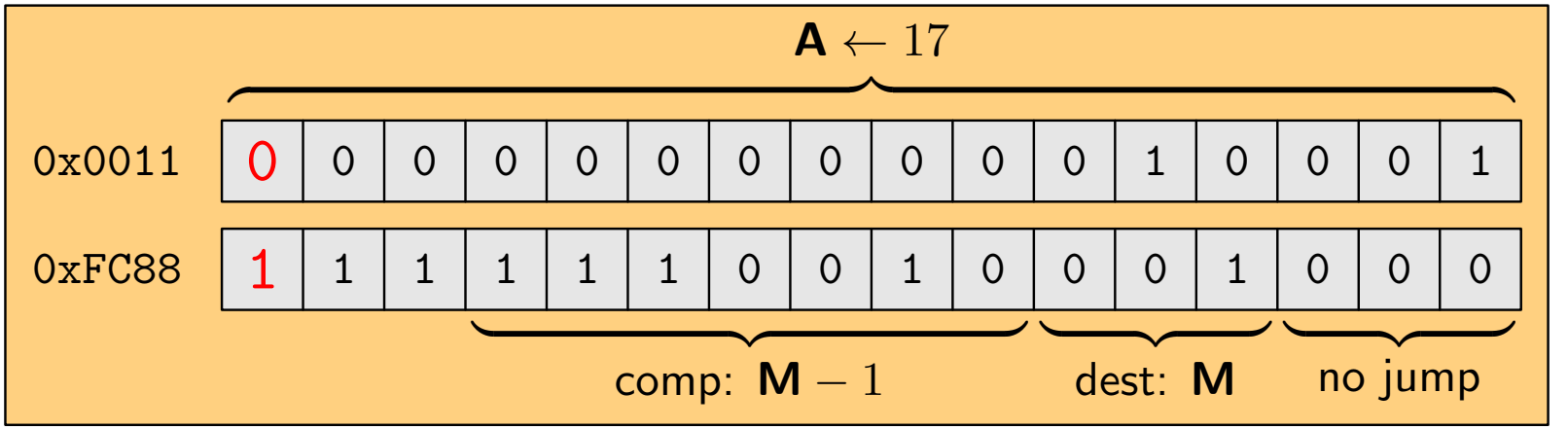


Lade einen 16-Bit-Wert aus der Speicherzelle mit Adresse 17 in das A-Register.

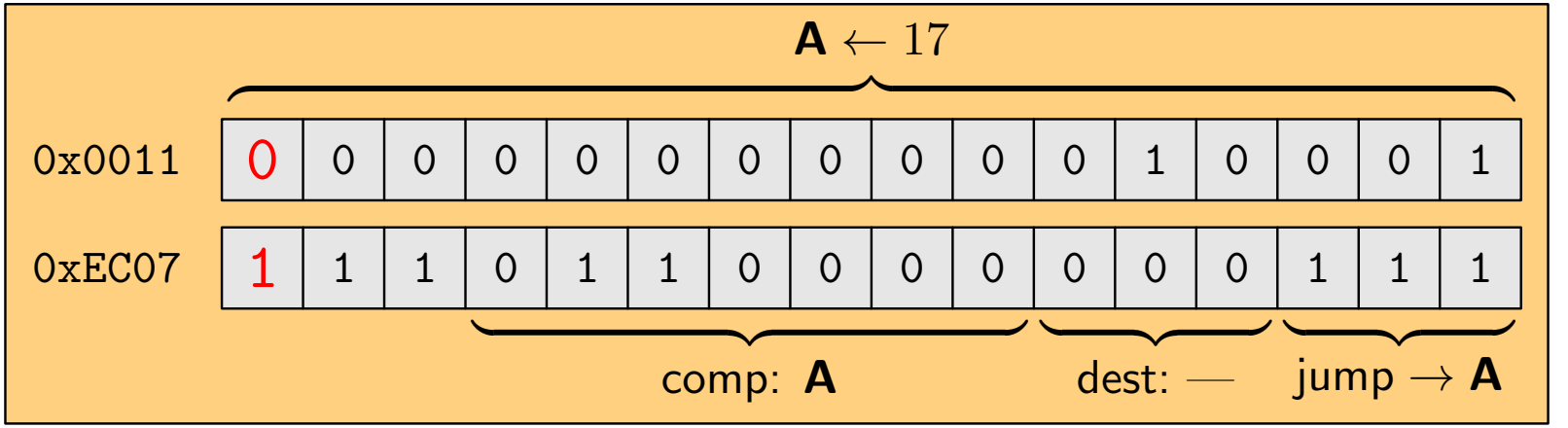


Hack-Maschinensprache: Beispiele

Erniedrige den Wert der Speicherzelle mit Adresse 17 um Eins.



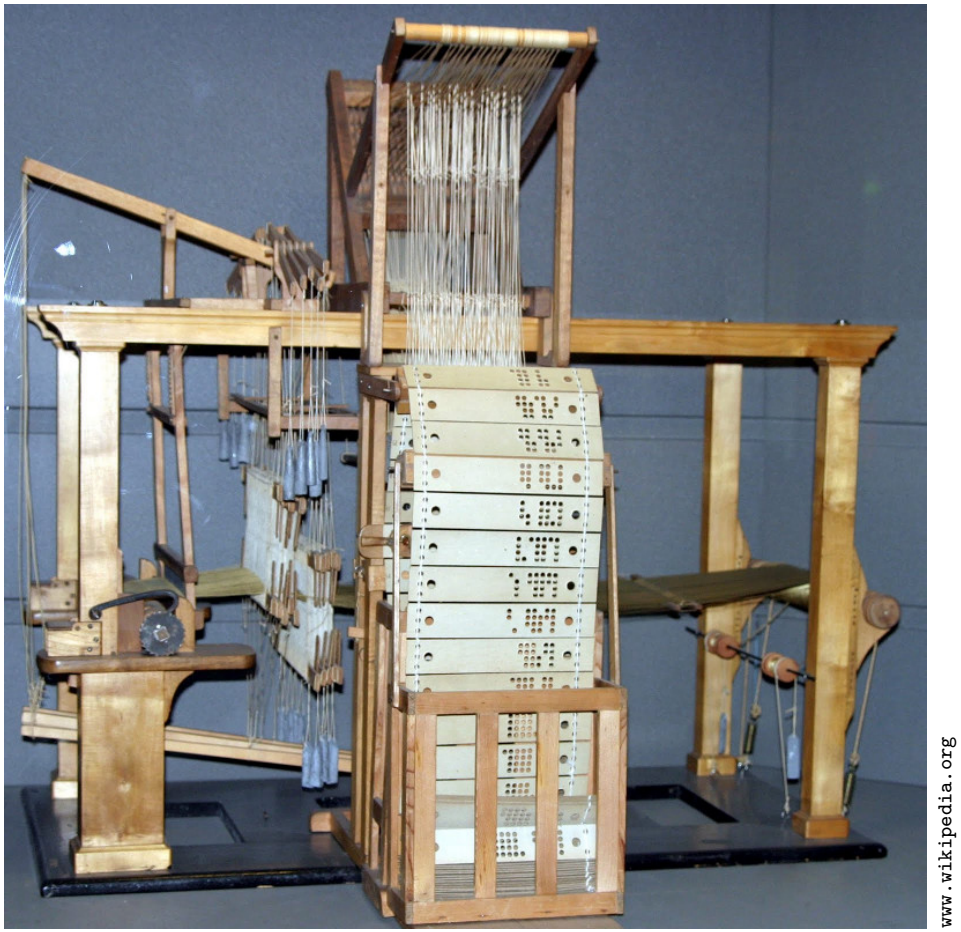
Setze die Programmausführung an Adresse 17 fort.



Inhalt

Physikalische und symbolische Programmierung

Jacquard-Webstuhl
[Joseph-Marie Jacquard 1805]



www.wikipedia.org

Webmuster auf Lochkarten:

- Erlaubt einfache Herstellung groß gemusterter Gewebe.
- Eine Lochkarte je Schuß (Kettfäden: längslaufend, Schußfäden: querlaufend).
- Lochkarten bewirken Anheben oder Absenken der Kettfäden des Gewebes.
- Karten sind durch Fäden miteinander verbunden.
- Ringkette von Lochkarten: sich wiederholende Muster.
- physikalische Programmierung

Physikalische und symbolische Programmierung

Entwicklung der Programmierung

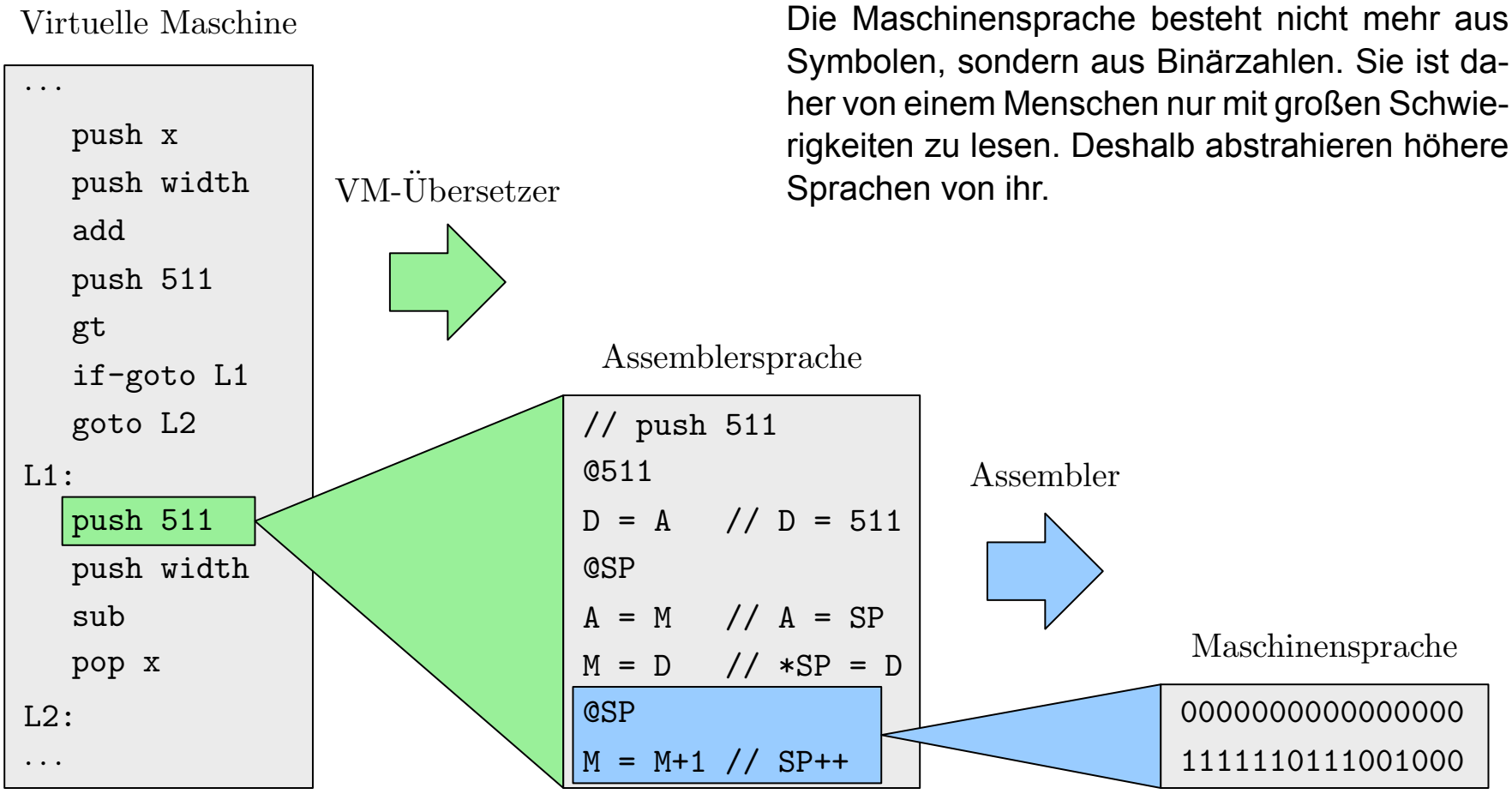
- Physikalische Programmierung
  (z.B. Lochkarten wie beim Jacquard-Webstuhl)
- Symbolische Dokumentation
  (z.B. Plan zur Berechnung der Bernoulli-Zahlen, den Ada Lovelace in ihren Notizen zu einer Beschreibung der "Analytical Engine" von Charles Babbage entwickelte, dort in Diagrammform)
- Symbolische Programmierung
  (Programmierung in symbolischer Form, (automatisches) Übersetzen der symbolischen in eine physikalische Programmform, Ausführen der physikalischen Form)
- Für die Übersetzung aus der symbolischen in die physikalische Form wird ein **Übersetzer** (translator, assembler, compiler) benötigt.

www.wikipedia.org

Augusta Ada Byron King, Countess of Lovelace [1815–1852]

Erste Programmiererin (auch wenn ihr Programm nie ausgeführt wurde, da die "Analytical Engine" nicht gebaut wurde)

Erinnerung: Assembler und Maschinensprache



Die Maschinensprache besteht nicht mehr aus Symbolen, sondern aus Binärzahlen. Sie ist daher von einem Menschen nur mit großen Schwierigkeiten zu lesen. Deshalb abstrahieren höhere Sprachen von ihr.

- Die **Maschinensprache** kann von einem Rechner direkt ausgeführt werden!
  (Sie ist eine physikalische Form eines Programms.)

Physikalische und symbolische Programmierung

Dualität von Hardware (Rechner) und Software (Programm)

- Die Maschinensprache (Befehlssatz, instruction set) kann als eine abstrakte Beschreibung (der Fähigkeiten) der Hardware-Plattform gesehen werden.
- Die Hardware (physikalischer Rechner) kann als physikalisches Mittel gesehen werden, um eine abstrakte Maschinensprache zu realisieren.

Maschinensprache und Assemblersprache

- Maschinensprache und Assemblersprache sind nur zwei Seiten der gleichen Sache.
- Die Maschinensprache ist nah am physikalischen Rechner. Programme sind Folgen von Binärzahlen, die (physikalisch) als Befehle interpretiert werden können.
- Die Assemblersprache ist eine symbolische Form der Maschinensprache, die für Menschen verständliche Symbole zur Darstellung von Programmen benutzt.
- Jeder Binärzahl der Maschinensprache entspricht ein einfacher symbolischer Ausdruck der Assemblersprache.

Assembler und Assemblersprache

- Ein **Assembler** ist ein Programm, das aus der symbolischen **Assemblersprache** in die binäre **Maschinensprache** übersetzt.

- Programmbefehle in einer Maschinensprache bestehen gewöhnlich aus einem sogenannten **Opcode** (kurz für operation code) und weiteren, je nach Befehl verschiedenen Angaben wie Adressen, in den Befehl eingebetteten Zahlen, Längenangaben etc. (vgl. die A- und C-Anweisungen der Hack-Maschinensprache).

- Da die Opcodes als Zahlenwerte oft schwer zu merken sind, verwenden Assemblersprachen sogenannte **mnemonische Symbole** (oder kurz: **Mnemonics**, nach dem griechischen $\mu\nu\acute{\eta}\mu\eta$: Gedächtnis), d.h., leicht(er) zu merkende Befehlskürzel.

- Eine Assemblersprache erlaubt es außerdem, z.B. Speicheradressen Namen („Variablennamen“) zuzuweisen, die für Menschen leichter handhabbar sind als numerische Speicheradressen.

- Beispiel: C-Anweisung der Hack-Maschinensprache

Assemblersprache	Maschinensprache (binär)	(hexadezimal)
`A = A-1`	`1110 1100 1001 0111`	`0xEC97`

Hack-Assemblersprache

C-Anweisungen (compute instructions)

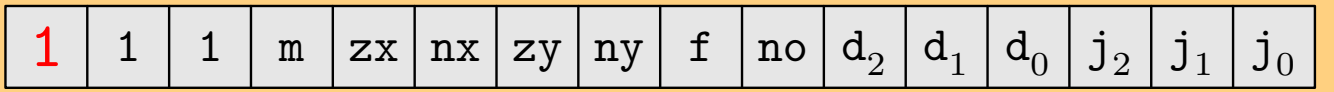
`dest` = `comp`; `jump` // `dest` ← `comp`; if `jump`: PC ← A

`dest`	`comp`	`jump`	
—	—	—	(jump never)
`M`	`0`, `1`, `-1`,	`JGT`	(jump if greater than)
`D`	`D`, `A`, `M`,	`JEQ`	(jump if equal)
`MD`	`~D`, `~A`, `~M`, bzw. `!D`, `!A`, `!M`,	`JGE`	(jump if greater or equal)
`A`	`-D`, `-A`, `-M`,	`JLT`	(jump if less than)
`AM`	`D+1`, `A+1`, `M+1`, `D-1`, `A-1`, `M-1`,	`JNE`	(jump if not equal)
`AD`	`D+A`, `D+M`, `D-A`, `D-M`, `A-D`, `M-D`,	`JLE`	(jump if less or equal)
`AMD`	`D&A`, `D&M`, `D|A`, `D|M`	`JMP`	(jump always)

- `dest`: Wo das Berechnungsergebnis gespeichert wird.
- `comp`: Welche Berechnung ausgeführt wird.
- `jump`: Ob und unter welchen Bedingungen gesprungen (verzweigt) wird.

Erinnerung: Hack-Maschinensprache

dest ← comp; 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/!D	0	0	1	1	0	1	
~A/!A	1	1	0	0	0	1	~M/!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 $\neq 0$
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.

Hack-Assemblersprache

A-Anweisungen (address instructions)

```
@value                          // A ← value
```

Hier ist `value` entweder eine Zahl oder ein Symbol, das für eine Zahl steht.

Verwendung für:

- Laden einer Konstante (A = `value`)
- Auswahl einer Datenspeicherzelle (`register` = `RAM[A]`)
- Auswahl einer Befehlsspeicherzelle (`fetch ROM[A]`)

Programmbeispiel:

```
@17       // A = 17
D=A       // D = 17
```

```
@17       // A = 17
D=M       // D = RAM[17]
```

```
@17       // A = 17
JMP       // fetch the instruction
          // stored in ROM[17]
```

Hack-Assemblersprache: Symbole

In Assemblerprogrammen werden normalerweise verschiedene **Symbole** benutzt:

- Marken (label) im Programmtext,
  die die Ziele von Sprunganweisungen angeben.
- Namen, die besondere Speicherzellen identifizieren.
- Variablennamen (Namen für Speicherzellen,
  die z.B. Zwischenergebnisse halten)

Alle möglichen Symbole gehören in eine von zwei Klassen:

- **Vordefinierte Symbole**
  (durch die Hack-Plattform festgelegt
  oder allgemein sinnvoll verwendbar)
- **Benutzerdefinierte Symbole**
  (durch einen Programmierer festgelegt)

```
    @R0
    D=M
    @END
    D;JLE
    @counter
    M=D
    @SCREEN
    D=A
    @addr
    M=D
(LOOP)
    @addr
    A=M
    M=-1
    @addr
    D=M
    @32
    D=D+A
    @addr
    M=D
    @counter
    MD=M-1
    @LOOP
    D;JGT
(END)
    @END
    0;JMP
```

Hack-Assemblersprache: Vordefinierte Symbole

- Als Teil der Übersetzung von der Assemblersprache in die Maschinensprache ordnet der Assembler vordefinierten Symbolen feste Speicheradressen zu.
- **Virtuelle Register:**
  Die Symbole `R0`, …, `R15` (Register 0 bis 15) sind vordefiniert mit den Werten 0, …, 15.
- **Ein- und Ausgabeadressen:**
  Die Symbole `SCREEN` und `KBD` sind vordefiniert mit den Werten 16384 (Basisadresse des Bildschirmspeichers) bzw. 24576 (Adresse des Tastaturregisters).
- **Vordefinierte VM-Adressen:**
  Die Symbole `SP`, `LCL`, `ARG`, `THIS`, `THAT` sind vordefiniert mit den Werten 0, …, 4.

Diese Adressen dienen der späteren einfacheren Implementierung einer virtuellen Maschine. ⇒ nächste Vorlesung

```
    @R0
    D=M
    @END
    D;JLE
    @counter
    M=D
    @SCREEN
    D=A
    @addr
    M=D
(LOOP)
    @addr
    A=M
    M=-1
    @addr
    D=M
    @32
    D=D+A
    @addr
    M=D
    @counter
    MD=M-1
    @LOOP
    D;JGT
(END)
    @END
    0;JMP
```

Hack-Assemblersprache: Benutzerdefinierte Symbole

Markersymbole (label symbols): (Großbuchstaben)

- Dienen zum Identifizieren der Ziele von Sprunganweisungen (`jump`).
- Werden durch die Pseudo-Anweisung `(XXX)` definiert: Das Symbol `XXX` wird definiert als die Befehlsspeicheradresse, die die nächste Programmanweisung enthält.

Variablensymbole (variable symbols): (Kleinbuchstaben)

- Jedes benutzerdefinierte Symbol `xxx`, das in einem Assemblerprogramm auftaucht und nicht irgendwo durch eine Pseudo-Anweisung `(xxx)` definiert ist, wird automatisch einer festen Datenspeicheradresse (random access memory, RAM) zugeordnet, beginnend bei der Speicheradresse 16.

Die Adressen 0 bis 15 sind für die virtuellen Register vergeben.

```
    @R0
    D=M
    @END
    D;JLE
    @counter
    M=D
    @SCREEN
    D=A
    @addr
    M=D
(LOOP)
    @addr
    A=M
    M=-1
    @addr
    D=M
    @32
    D=D+A
    @addr
    M=D
    @counter
    MD=M-1
    @LOOP
    D;JGT
(END)
    @END
    0;JMP
```

Hack-Assemblersprache: Bedingte Anweisungen

Hochsprache

```
if condition {
    code segment 1
}
else {
    code segment 2
}

// next instruction
```

Hack-Assemblersprache

```
    ⟨D ← condition⟩
    @ELSE
    D;JEQ
    code segment 1
    @END
    0;JMP
(ELSE)
    code segment 2
(END)

// next instruction
```

- Um einander widersprechende Verwendungen des A-Registers zu vermeiden, sollte in einem gut geschriebenen Hack-Programm eine C-Anweisung mit einem Verzweigungsteil (jump) keinen Verweis auf den Datenspeicher (M, memory) enthalten und umgekehrt.

Hack-Assemblersprache: Schleifen

Hochsprache

```
while condition {
    code segment 1
}

// next instruction
```

Hack-Assemblersprache

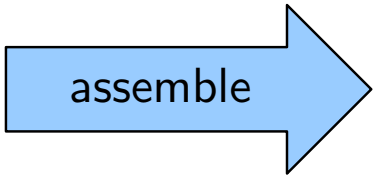
```
(LOOP)
    ⟨D ← condition⟩
    @END
    D;JEQ
    code segment 1
    @LOOP
    0;JMP
(END)

// next instruction
```

- Um einander widersprechende Verwendungen des A-Registers zu vermeiden, sollte in einem gut geschriebenen Hack-Programm eine C-Anweisung mit einem Verzweigungsteil (jump) keinen Verweis auf den Datenspeicher (M, memory) enthalten und umgekehrt.

Hack-Assemblersprache: Übersetzung

Quelltext: Assemblersprache



Zieltext: Maschinensprache

```
// Computes 1+...+RAM[0] and
// stores the sum in RAM[1]
    @i
    M=1    // i = 1
    @sum
    M=0    // sum = 0
(LOOP)
    @i     // if i > RAM[0]
    D=M    // then goto WRITE
    @R0
    D=D-M
    @WRITE
    D;JGT
    @i     // sum += i
    D=M
    @sum
    M=D+M
    @i     // i++
    M=M+1
    @LOOP  // goto LOOP
    0;JMP
(WRITE)
    @sum
    D=M
    @R1
    M=D    // RAM[1] = the sum
(END)
    @END
    0;JMP
```

Übersetzungsproblem:

- Lese den Quelltext, den Syntaxregeln der Quellsprache folgend.
- Drücke die Semantik des Quellprogramms mit den Syntaxregeln der Zielsprache aus.

Assembler:
einfacher Übersetzer

- Übersetzt jeden Assemblerbefehl in einen oder mehrere Maschinenbefehle (Hack: genau einen).
- Behandelt Symbole (`i`, `sum`, `LOOP`, `WRITE`, `END`).

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000

0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111

0000000000010001
1111110000010000
0000000000000001
1110001100001000

0000000000010110
1110101010000111
```

Hack-Assemblersprache: Übersetzung

Assemblerprogramm

```
// Computes 1+...+RAM[0] and
// stores the sum in RAM[1]
    @i
    M=1    // i = 1
    @sum
    M=0    // sum = 0
(LOOP)
    @i     // if i > RAM[0]
    D=M    // then goto WRITE
    @R0
    D=D-M
    @WRITE
    D;JGT
    @i     // sum += i
    D=M
    @sum
    M=D+M
    @i     // i++
    M=M+1
    @LOOP  // goto LOOP
    0;JMP
(WRITE)
    @sum
    D=M
    @R1
    M=D    // RAM[1] = the sum
(END)
    @END
    0;JMP
```

- Ein Assemblerprogramm ist eine Folge von Textzeilen, mit jeweils einem der folgenden Inhalte:
  - eine A-Anweisung:
    `@value`
  - eine C-Anweisung:
    `dest = comp;jump`
  - eine Symboldefinition (Sprungziel):
    `(SYMBOL)`
  - Kommentar oder Leerzeichen/Leerzeile:
    `// comment`

    (Kommentare können auch nach Anweisungen stehen.)
- Aufgabe eines **Assemblers** ist es, dieses Programm in eine Folge von durch Binärzahlen kodierte Anweisungen zu übersetzen, die von der Zielplattform (hier: der Hack-Plattform) ausgeführt werden können.

Hack-Assemblersprache: Übersetzung

Assemblerprogramm

```
// Computes 1+...+RAM[0] and
// stores the sum in RAM[1]
    @i
    M=1    // i = 1
    @sum
    M=0    // sum = 0
(LOOP)
    @i     // if i > RAM[0]
    D=M    // then goto WRITE
    @R0
    D=D-M
    @WRITE
    D;JGT
    @i     // sum += i
    D=M
    @sum
    M=D+M
    @i     // i++
    M=M+1
    @LOOP  // goto LOOP
    0;JMP
(WRITE)
    @sum
    D=M
    @R1
    M=D    // RAM[1] = the sum
(END)
    @END
    0;JMP
```

Für jede Anweisung (außer Symboldefinitionen) geschieht im Hack-Assembler folgendes:

- Die Anweisung wird zergliedert (parsed), d.h., in ihre Bestandteile zerlegt.
- A-Anweisung:
  Falls ein Symbol angegeben ist, wird es durch die zugehörige Speicheradresse, die eine Zahl ist, ersetzt (Details dazu gleich); sonst muß eine Zahl angegeben sein, die direkt verwendet wird.
- C-Anweisung:
  Für jedes Feld der Anweisung (dest, comp, jump) wird der zugehörige Binärcode erzeugt und diese Codes werden zu einem Maschinenbefehl (16-Bit-Binärzahl) zusammengefügt.
- Die erzeugte 16-Bit-Binärzahl wird in die Ausgabedatei geschrieben.

Hack-Assemblersprache: Symbolverwaltung

Assemblerprogramm

```
// Computes 1+...+RAM[0] and
// stores the sum in RAM[1]
    @i
    M=1    // i = 1
    @sum
    M=0    // sum = 0
(LOOP)
    @i     // if i > RAM[0]
    D=M    // then goto WRITE
    @R0
    D=D-M
    @WRITE
    D;JGT
    @i     // sum += i
    D=M
    @sum
    M=D+M
    @i     // i++
    M=M+1
    @LOOP  // goto LOOP
    0;JMP
(WRITE)
    @sum
    D=M
    @R1
    M=D    // RAM[1] = the sum
(END)
    @END
    0;JMP
```

Symboltabelle

R0	0
R1	1
R2	2
⋮	⋮
R15	15
SCREEN	16384
KBD	24576
SP	0
LCL	1
ARG	2
THIS	3
THAT	4
LOOP	4
WRITE	18
END	22
i	16
sum	17

Symbolverwaltung

- Diese Symboltabelle wird vom Assembler erzeugt und benutzt, um den symbolischen in den Binärcode zu übersetzen.
- Man beachte die vordefinierten Symbole am Anfang und die benutzerdefinierten Symbole am Ende der Tabelle.
- Bevor der Assembler seine Arbeit aufnimmt, enthält die Tabelle nur die vordefinierten Symbole.
- Die benutzerdefinierten Symbole werden bei der Verarbeitung des Assemblerprogramms hinzugefügt.

Hack-Assemblersprache: Übersetzung

Assemblerprogramm

```
// Computes 1+...+RAM[0] and
// stores the sum in RAM[1]
    @i
    M=1    // i = 1
    @sum
    M=0    // sum = 0
(LOOP)
    @i     // if i > RAM[0]
    D=M    // then goto WRITE
    @R0
    D=D-M
    @WRITE
    D;JGT
    @i     // sum += i
    D=M
    @sum
    M=D+M
    @i     // i++
    M=M+1
    @LOOP  // goto LOOP
    0;JMP
(WRITE)
    @sum
    D=M
    @R1
    M=D    // RAM[1] = the sum
(END)
    @END
    0;JMP
```

Ablauf der Übersetzung

- Initialisierung der Symboltabelle:
  Eine (leere) Symboltabelle wird erzeugt und
  die vordefinierten Symbole werden eingefügt.
- Erster Durchlauf (first pass):
  Der Quelltext wird vollständig durchlaufen, aber es
  werden noch keine Maschinenanweisungen erzeugt,
  sondern es werden nur die benutzerdefinierten Marken
  in die Symboltabelle eingetragen (z.B. `LOOP`).
- Zweiter Durchlauf (second pass):
  Der Quelltext wird erneut durchlaufen,
  und die Symboltabelle wird benutzt,
  um alle Anweisungen in Binärzahlen zu übersetzen.

  In diesem Durchlauf werden auch
  die Variablensymbole (`i` und `sum`) verarbeitet
  (in die Symboltabelle eingetragen).

Hack-Assemblersprache: Übersetzung

- **Initialisierung der Symboltabelle:**
  Eine Symboltabelle wird erzeugt und mit den vordefinierten Symbolen befüllt.

- **Erster Durchlauf** (first pass):
  Der Quelltext wird durchlaufen, ohne Maschinenanweisungen zu erzeugen.
  Für jede Markendefinition `(LABEL)`, die im Quelltext auftritt,
  wird ein Paar $\langle$`LABEL`, $n\rangle$ in die Symboltabelle eingetragen,
  wobei n die Anzahl bereits durchlaufener Zeilen mit Assemblerbefehlen ist.

- **Zweiter Durchlauf** (second pass):
  Der Quelltext wird erneut durchlaufen und jede Zeile wird wie folgt verarbeitet:

  - Markendefinitionen werden übersprungen (schon verarbeitet).
  - Falls die Zeile eine C-Anweisung `dest = comp;jump` enthält:
    Erzeuge den Maschinenbefehl durch Aufsuchen und Zusammensetzen
    der zugehörigen Binärcodes und gib die erhaltene Binärzahl aus.
  - Falls die Zeile eine A-Anweisung `@xxx` enthält . . .

(Fortsetzung siehe nächste Folie)

Hack-Assemblersprache: Übersetzung

- **Zweiter Durchlauf** (second pass):
  Der Quelltext wird erneut durchlaufen und jede Zeile wird wie folgt verarbeitet:
  - … (Fortsetzung von vorangehender Folie)
  - Falls die Zeile eine A-Anweisung `@xxx` enthält und `xxx` ist eine Zahl:
    Gib `xxx` als Binärzahl aus.
  - Falls die Zeile eine A-Anweisung `@xxx` enthält und `xxx` ist ein Symbol:
    Suche in der Symboltabelle nach den Symbol `xxx`.
    - Falls das Symbol gefunden wurde,
      lies die in der Symboltabelle zu `xxx` abgelegte Zahl k aus.
    - Falls das Symbol nicht gefunden wurde,
      füge der Symboltabelle das Paar $\langle$`xxx`, $k\rangle$ hinzu,
      wobei k die Adresse der nächsten freien Datenspeicherzelle ist.

    Gib den Wert k als Binärzahl aus.

- Beachte: Freie Datenspeicheradressen beginnen im Hack-System bei 16.
  Der Wert k wird einfach ausgegeben, da A-Anweisungen mit 0 anfangen.

Hack-Assemblersprache: Symbolverwaltung

Assemblerprogramm

```
// Computes 1+...+RAM[0] and
// stores the sum in RAM[1]
    @i
    M=1    // i = 1
    @sum
    M=0    // sum = 0
(LOOP)
    @i     // if i > RAM[0]
    D=M    // then goto WRITE
    @R0
    D=D-M
    @WRITE
    D;JGT
    @i     // sum += i
    D=M
    @sum
    M=D+M
    @i     // i++
    M=M+1
    @LOOP  // goto LOOP
    0;JMP
(WRITE)
    @sum
    D=M
    @R1
    M=D    // RAM[1] = the sum
(END)
    @END
    0;JMP
```

Symboltabelle

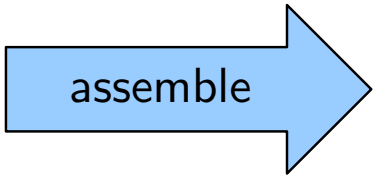
R0	0
R1	1
R2	2
⋮	⋮
R15	15
SCREEN	16384
KBD	24576
SP	0
LCL	1
ARG	2
THIS	3
THAT	4
LOOP	4
WRITE	18
END	22
i	16
sum	17

Symbolverwaltung

- Dem Symbol `LOOP` wird der Wert 4 zugeordnet, da ihm 4 Assemblerbefehle vorangehen.
- Analog werden `WRITE` und `END` die Werte 18 bzw. 22 zugeordnet.
- Das Symbol `i` erhält den Wert 16, da es das erste Variablensymbol ist, und Variablensymbolen Speicheradressen ab 16 zugewiesen werden.
  (Die Adressen 0 bis 15 sind durch die virtuellen Register belegt.)
- Dem Symbol `sum` wird der Wert 17 zugewiesen, da dies nach der Zuweisung von 16 an `i` die nächste freie Datenspeicherzelle ist.

Hack-Assemblersprache: Übersetzung

Quelltext: Assemblersprache



Zieltext: Maschinensprache

```
// Computes 1+...+RAM[0] and
// stores the sum in RAM[1]
    @i
    M=1    // i = 1
    @sum
    M=0    // sum = 0
(LOOP)
    @i     // if i > RAM[0]
    D=M    // then goto WRITE
    @R0
    D=D-M
    @WRITE
    D;JGT
    @i     // sum += i
    D=M
    @sum
    M=D+M
    @i     // i++
    M=M+1
    @LOOP  // goto LOOP
    0;JMP
(WRITE)
    @sum
    D=M
    @R1
    M=D    // RAM[1] = the sum
(END)
    @END
    0;JMP
```

Assembler:
einfacher Übersetzer

- Übersetzt jeden Assemblerbefehl in einen oder mehrere Maschinenbefehle (Hack: genau einen).
- Behandelt Symbole (`i`, `sum`, `LOOP`, `WRITE`, `END`).

Erzeugtes Maschinenprogramm:

- Ist eine Folge von Binärzahlen. (Hack: 16-Bit-Binärzahlen)
- Wird im Hack-System im ROM ab Adresse 0 abgelegt.

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000

0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111

0000000000010001
1111110000010000
0000000000000001
1110001100001000

0000000000010110
1110101010000111
```

Hack-Assembler: Implementierung

Ein Assembler für die Hack-Assemblersprache kann wie folgt implementiert werden:

Programmbausteine: (z.B. durch Klassen abzubilden)

- **Parser:** Zergliedert jeden Assemblerbefehl in seine Bestandteile.
- **Code:** Übersetzt jedes Feld in seine binäre Kodierung.
- **SymbolTable:** Verwaltet die Symboltabelle.
- **Main:** Öffnet Ein- und Ausgabedatei und steuert den Ablauf.

Bemerkung: Ein Parser wird oft auch in einen **Scanner**, der den Zeichenstrom der Eingabe in einen Tokenstrom umwandelt, und den eigentlichen **Parser** zerlegt.)

Entwicklungsschritte:

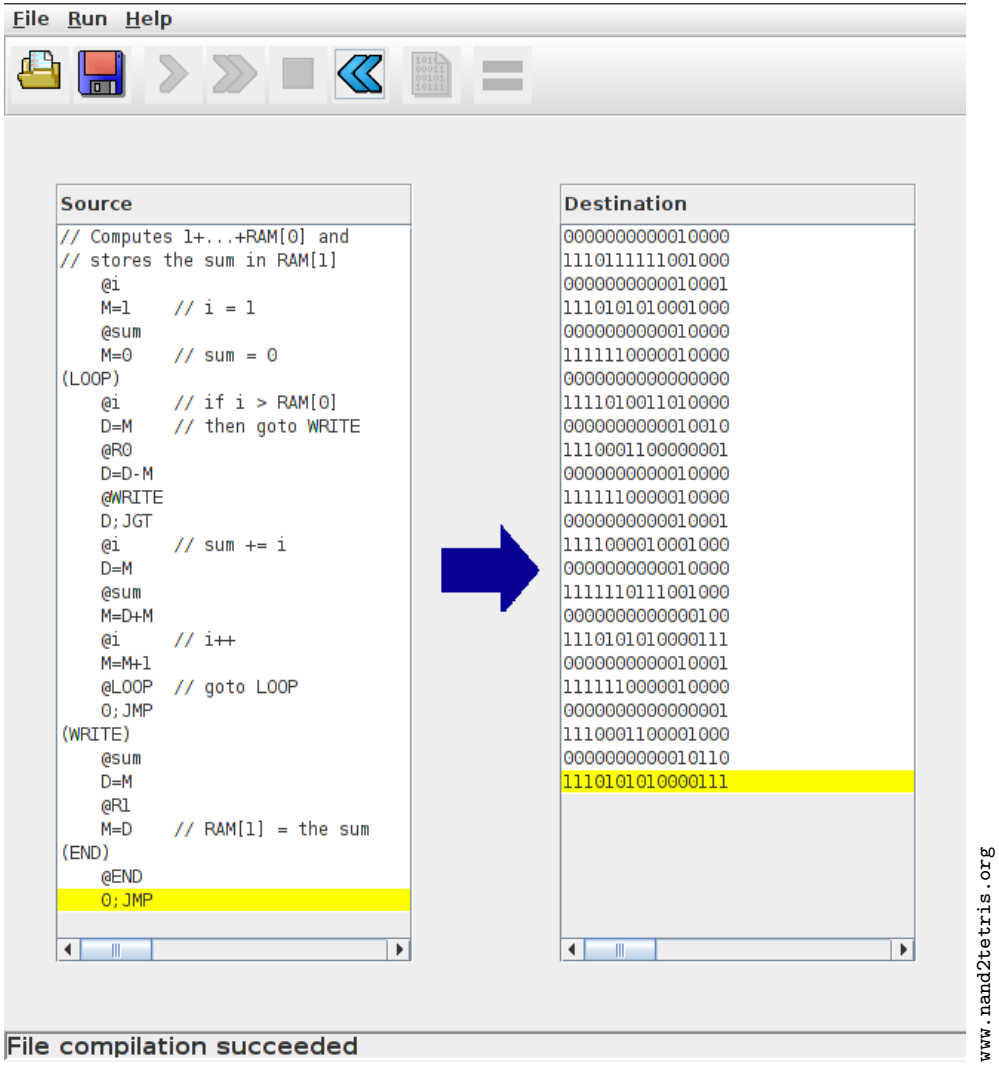
- **1. Stufe:** Basisassembler für Programme ohne Symbole.
- **2. Stufe:** Erweiterung des Basisassemblers um eine Symbolverwaltung.

Hack-Assemblersprache: Assembler

Assemblerprogramm

```
// Computes 1+...+RAM[0] and
// stores the sum in RAM[1]
    @i
    M=1    // i = 1
    @sum
    M=0    // sum = 0
(LOOP)
    @i     // if i > RAM[0]
    D=M    // then goto WRITE
    @R0
    D=D-M
    @WRITE
    D;JGT
    @i     // sum += i
    D=M
    @sum
    M=D+M
    @i     // i++
    M=M+1
    @LOOP  // goto LOOP
    0;JMP
(WRITE)
    @sum
    D=M
    @R1
    M=D    // RAM[1] = the sum
(END)
    @END
    0;JMP
```

Hack-Assembler



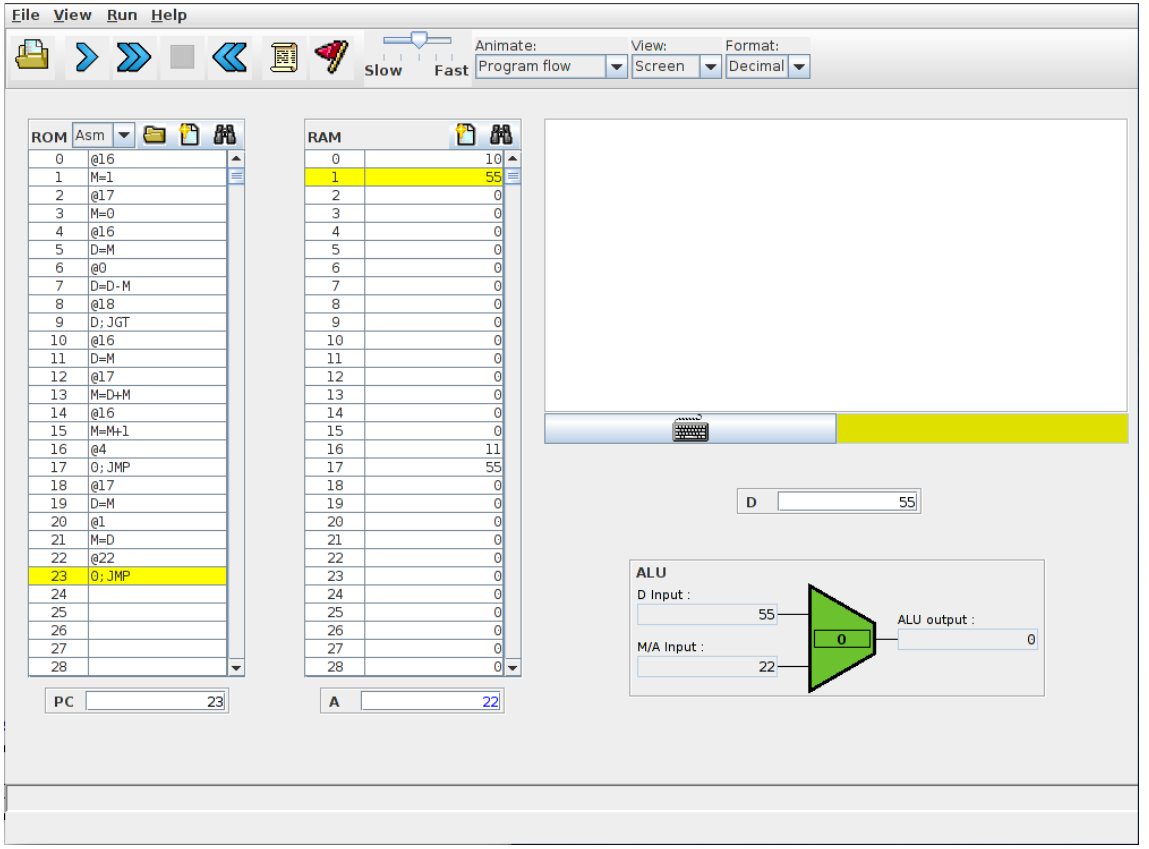
www.nand2tetris.org

Hack-Assemblersprache: CPU-Emulator

Assemblerprogramm

```
// Computes 1+...+RAM[0] and
// stores the sum in RAM[1]
    @i
    M=1    // i = 1
    @sum
    M=0    // sum = 0
(LOOP)
    @i     // if i > RAM[0]
    D=M    // then goto WRITE
    @R0
    D=D-M
    @WRITE
    D;JGT
    @i     // sum += i
    D=M
    @sum
    M=D+M
    @i     // i++
    M=M+1
    @LOOP  // goto LOOP
    0;JMP
(WRITE)
    @sum
    D=M
    @R1
    M=D    // RAM[1] = the sum
(END)
    @END
    0;JMP
```

Hack-CPU-Emulator

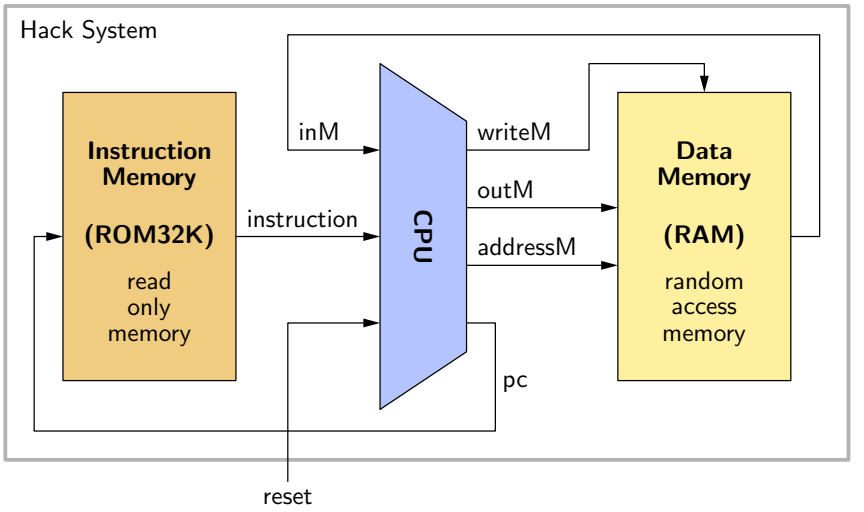


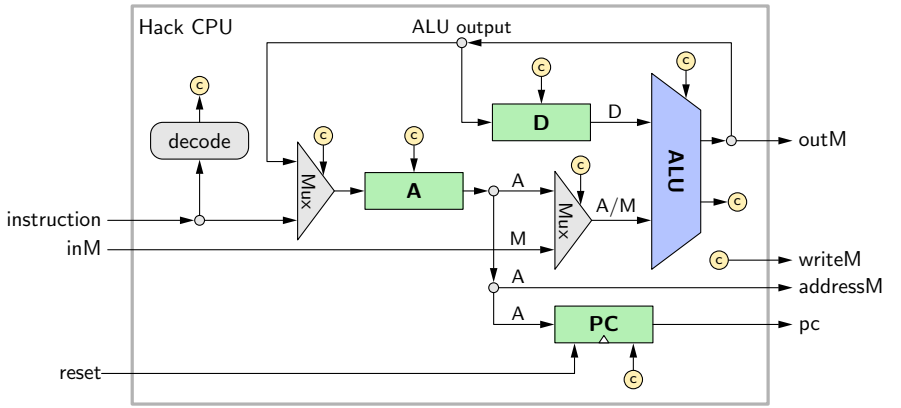
www.nand2tetris.org

Der Hack-CPU-Emulator kann Programme in Hack-Assemblersprache übersetzen und ausführen.

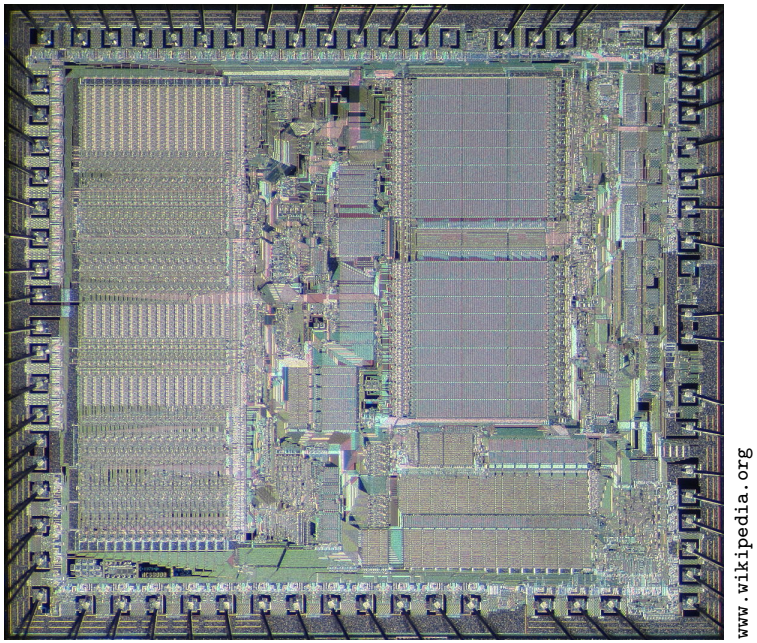
Inhalt

Hack-Prozessor und Motorola 68000

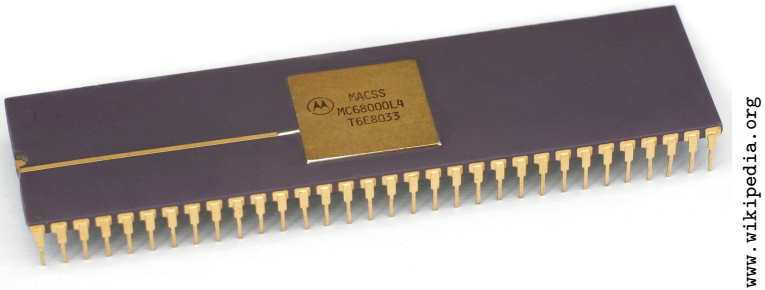




Hack-System und -Prozessor
16/16/16 Bit
Simuliertes System



www.wikipedia.org
(Pauli Rautakorpi)



www.wikipedia.org
(Konstantin Lanzet)

Motorola 68000 [1979]
16–32/16/24 Bit, 4MHz
68 000 Transistoren, 3.5μm

Hack-Prozessor und Motorola 68000

Hack-System

Hack-Prozessor (CPU)

Befehlszähler (`PC`)

D-Register (`D`)

Wertebereich: $\{-2^{15}, \ldots, 2^{15}-1\}$

A-Register (`A`)

Wertebereich: $\{\quad 0, \ldots, 2^{16}-1\}$

Hauptspeicher

`$0000` bis `$3fff`: 2^{14} Zellen RAM

`$4000` bis `$5fff`: Bildschirminhalt

`$6000`: Tastaturregister

Wertebereich: $\{-2^{15}, \ldots, 2^{15}-1\}$

Motorola-68000-System

68000 Prozessor (CPU)

Befehlszähler (`PC`)

8 Datenregister (`D0` bis `D7`)

Wertebereich: $\{\quad 0, \ldots, 2^{32}-1\}$

oder: $\{-2^{31}, \ldots, 2^{31}-1\}$

8 Adreßregister (`A0` bis `A7`)

Wertebereich: $\{\quad 0, \ldots, 2^{32}-1\}$

Hauptspeicher

`$000000` bis `$ffffff`: 2^{24} Zellen

Wertebereich: $\{\quad 0, \ldots, 2^{8}-1\}$

oder: $\{-2^{7}, \ldots, 2^{7}-1\}$

Zwei oder vier Speicherzellen können zusammengefaßt werden.

Motorola 68000: Aufbau eines Assemblerbefehls

- Ein Befehl der Motorola-68000-Assemblersprache hat die Form

  Befehlsname `.`*Verarbeitungsbreite* *Quelloperand* `,` *Zieloperand* *oder*
  Befehlsname `.`*Verarbeitungsbreite* *Zieloperand*

- Befehlsnamen können z.B. sein: `ADD` (addiere), `MOVE` (bewege), `ASR` (arithmetisches Schieben nach rechts), `CMP` (compare/vergleiche), `BGE` (branch if greater or equal/verzweige wenn größer oder gleich) etc.

- Die Verarbeitungsbreite gibt an, welcher Teil eines Registers verwendet bzw. wie viele Speicherzellen des Hauptspeichers zusammengefaßt werden sollen.

  Zulässig: `B` (byte, 8 bit), `W` (word, 2 byte, 16 bit) und `L` (long word, 4 byte, 32 bit). Allerdings sind nicht für jeden Befehl alle Verarbeitungsbreiten erlaubt.

- **Beispiel:** `ADD.W D1,D2`

  Nimm die unteren 16 Bit des Inhaltes des Datenregisters `D1`, addiere sie zu den unteren 16 Bit des Datenregisters `D2` und speichere das Ergebnis in den unteren 16 Bit des Datenregisters `D2`.

Motorola 68000: Indirekte Addressierung

- Der Motorola 68000 verfügt über recht mächtige Möglichkeiten zur indirekten Adressierung des Hauptspeichers.

- Ähnlich wie im Hack-System, bei dem nur über das A-Register Speicherzellen des Hauptspeichers ausgewählt werden können, kann nur über die Adreßregister des Prozessors indirekt adressiert werden.

- Folgende Adressierungsarten sind zulässig:

Schreibweise	Benennung	angesprochene Adresse	neuer Inhalt von `An`
`(An)`	einfach indirekt	`[An]`	
`-(An)`	indirekt mit Predecrement	`[An]`$-$`vb`	`[An]`$-$`vb`
`(An)+`	indirekt mit Postinkrement	`[An]`	`[An]`$+$`vb`
`d(An)`	indirekt mit Offset	`[An]`$+$`d`	
`d(An,Dn)`	indirekt mit Offset und Index	`[An]`$+$`d`$+$`[Dn]`	

(`[An]` bezeichnet den Inhalt des Adreßregisters `An`,
`vb` die Verarbeitungsbreite des Befehls in Byte.)

Motorola 68000: Beispiel-Assemblercode

C-Quelltext	Pseudo-Zwischencode	Assemblercode (Motorola 68000)	Maschinencode (hexadezimal)	Takte
`z = 0;` `i = 0;`	`D0 = 0` `D1 = 0`	`01: CLR.W D0` `02: CLR.W D1`	`42 40` `42 41`	4 4
	`A0 = &a` `A1 = &b`	`03: LEA.L a,A0` `04: LEA.L b,A1`	`41 f9 xx xx xx xx` `43 f9 xx xx xx xx`	12 12
`while (i < n) {`	verzweige	`05: BRA.B 13`	`60 12`	10

`  b[i] = (i+1) *a[i+1];`	`D2 = 1`
	`D2 += D1`
	`D3 = D1`
	`D3 += D3`
	`D2 *= A0[D3+2]`
	`A1[D3] = D2`
		`06: MOVEQ.L #1,D2`
		`07: ADD.W D1,D2`
		`08: MOVE.W D1,D3`
		`09: ADD.W D3,D3`
		`10: MULS.W 2(A0,D3.W),D2`
		`11: MOVE.W D2,(A1,D3.W)`
			`74 01`
			`d4 41`
			`36 01`
			`d6 43`
			`c5 f0 30 02`
			`33 82 30 00`
				4
				4
				4
				4
				$\leq$ 80
				14

`  i = i+1;`	`D1 += 1`	`12: ADDQ.W #1,D1`	`52 41`	4
`} /* while (i < n) */`	`D1 < n ?` wenn ja, verzweige	`13: CMP.W n,D1` `14: BLT.B 06`	`b2 79 xx xx xx xx` `6d e6`	16 10/8

Bedeutung einiger Assemblerbefehle:

`CLR`	clear	lösche (setze auf Null)
`LEA`	load effective address	lade Speicheradresse
`MOVE`	move	bewege
`MULS`	multiply signed	multipliziere mit Vorzeichen

`CMP`	compare	vergleiche
`BLT`	branch if less than	verzweige wenn kleiner
`BNE`	branch if not equal	verzweige wenn ungleich
`BRA`	branch always	verzweige immer

Motorola 68000: Beispiel-Assemblercode

C-Quelltext	Pseudo-Zwischencode	Assemblercode (Motorola 68000)	Maschinencode (hexadezimal)	Takte
`i = n-1;`	`D1 = -1` `D1 += n`	`15: MOVEQ.L #-1,D1` `16: ADD.W n,D1`	`72 ff` `d2 79 xx xx xx xx`	4 16

`y = b[n-1]`	`A2 = &y`
	`D2 = n`
	`D2 += D2`
	`*A2 = A1[D2-2]`
		`17: LEA.L y,A2`
		`18: MOVE.W n,D2`
		`19: ADD.W D2,D2`
		`20: MOVE.W -2(A1,D2.W),(A2)`
			`45 f9 xx xx xx xx`
			`34 39 xx xx xx xx`
			`d4 42`
			`34 b1 20 fe`
				12
				16
				4
				18

`while (i >= 1) {`	verzweige	`21: BRA.B 35`	`60 26`	10

`  y = y *x +b[i-1];`	`D2 = *A2`
	`D2 *= x`
	`D3 = D1`
	`D3 += D3`
	`D2 += A1[D3-2]`
	`*A2 = D2`
		`22: MOVE.W (A2),D2`
		`23: MULS.W x,D2`
		`24: MOVE.W D1,D3`
		`25: ADD.W D3,D3`
		`26: ADD.W -2(A1,D3.W),D2`
		`27: MOVE.W D2,(A2)`
			`34 12`
			`c5 f9 xx xx xx xx`
			`36 01`
			`d6 43`
			`d4 71 30 fe`
			`34 82`
				8
				$\leq$ 82
				4
				4
				14
				8

`  if (y == 0) {`	`D2 != 0 ?` wenn ja, verzweige	`28: TST.W D2` `29: BNE.W 33`	`4a 42` `66 08`	4 10/8

`    z = z+1;`
`    a[i] = 0;`
`  }`
	`D0 += 1`
	`A0[D3] = 0`
	verzweige
		`30: ADDQ.W #1,D0`
		`31: CLR.W (A0,D3.W)`
		`32: BRA.B 34`
			`52 40`
			`42 70 30 00`
			`60 06`
				4
				18
				10

`  else a[i] = 1;`	`A0[D3] = 1`	`33: MOVE.W #1,(A0,D3.W)`	`31 bc 00 01 30 00`	18
`  i = i-1;`	`D1 -= 1`	`34: SUBQ.W #1,D1`	`53 41`	4
`} /* while (i >= 1) */`	`D1 >= 1 ?` wenn ja, verzweige	`35: CMP.W #1,D1` `36: BGE.B 22`	`b2 7c 00 01` `6c d4`	8 10/8

Beispiel: Linux

```
        .section        .rodata
string:
        .ascii "Hello, World!\n\0"
length:
        .quad -. -string                #Dot = 'here'

        .section        .text
        .globl _start                   #Make entry point visible to linker
_start:
        movq $4, %rax                   #4=write
        movq $1, %rbx                   #1=stdout
        movq $string, %rcx
        movq length, %rdx
        int $0x80                       #Call operating system
        movq %rax, %rbx                 #Make program return syscall exit status
        movq $1, %rax                   #1=exit
        int $0x80                       #Call system again
```

Beispiel: Windows

```
            .486p                                   ; Written by Stewart Moss - May 2006
            .model      flat,STDCALL                ; Assemble using TASM 5.0 and TLINK32
include     win32.inc

extrn                   MessageBoxA:PROC
extrn                   ExitProcess:PROC

.data

HelloWorld db "Hello, World!",0
msgTitle db "Hello world program",0

.code
Start:
            push        MB_INCONQUESTION + MB_APPLMODAL + MD_OK
            push        offset msgTitle
            push        offset HelloWorld
            push        0
            call        MessageBoxA
            push        0
            call        ExitProcess
ends
end Start
```

Programmieren in Assemblersprache?

- Die Assemblersprache ist eine systemnahe, maschinenorientierte Sprache.
  Mnemonische Symbole und Symbolverwaltung machen sie für Menschen nutzbar.

- Die Assemblersprache erlaubt eine direkte Einflußnahme auf die Hardware
  (hardware-nahe Programmierung, Treiber (driver), Urlader (boot loader) etc.)

- Sie wird für bestimmte Aufgaben in der Betriebssystemprogrammierung benötigt
  (etwa Prozeßumschaltung, Semaphore, z.B. Befehl `TAS` (test and set) etc.)

- Sie erlaubt Nutzung von Sonderbefehlen der Hardware-Plattform,
  z.B. SIMD (single instruction multiple data),
  SSE (streaming SIMD extensions), Befehl `popcount` etc.

- Sie erlaubt maximale Geschwindigkeits- und Speicheroptimierung
  (dann aber für eine bestimmte Hardware-Plattform).

- Aber leider auch: Computerviren, Trojaner und andere Schadprogramme
  (da diese Schadprogramme oft sehr systemnahe Teile enthalten).

Zusammenfassung: Maschinensprache und Assembler

- **Die Hack-Maschinensprache**
  - A-Anweisungen (address instructions)
  - C-Anweisungen (compute instructions)

- **Assembler und Assemblersprache**
  - Physikalische und symbolische Programmierung
  - Maschinensprache und Assemblerprache
  - Opcodes, mnemonische Symbole (Mnemonics)
  - Die Hack-Assemblersprache
  - Typische Programmstrukturen in der Hack-Assemblersprache
  - Symbole und Symbolverwaltung
  - Programmübersetzung und Assemblerimplementierung
  - Hack-Assembler vs. “richtiger” Assembler (Motorola 68000)
  - Beispiele für Assemblerprogramme: Linux und Windows