Rechnerarchitektur

Einführung in die Informatik & Rechnerarchitektur (EIR1/EIF1)

Erik Pitzer

SE & MBI – FH Hagenberg – WS 2025/26

Maschinensprache & Assembler

Komplexität von Mikrocode

- Mikrobefehle sehr umständlich
- höhere Abstraktion gewünscht, z.B.
  - `[1]` $\rightarrow$ `R1` als ein Befehl, oder sogar
  - `[1] + [2]` $\rightarrow$ `[3]`
- Verstecken von komplexen Details wie
  - X-, Y-, Z-, RAM-Bus
  - Spezialregister (`MAR`, `MDR`, `COP`)
  - ALU Code
  - Adressrechner
  - Taktphasen

“Kochrezepte” als Instruktionen

- Zusammenfassen von Mikroinstruktionen zu Programmbausteinen, z.B.
  - Lesen aus RAM, Schreiben in RAM
  - Berechnungen mit Registern
  - Berechnungen mit Register und Speicherzugriffen
  - …
- eigentliches Program liegt im RAM und verweist auf kurze Mikroprogramme im ROM
- Prozessor kann dadurch verschiedene Programme ausführen und nicht immer nur das gleiche

Schreibweise von Assemblerbefehlen

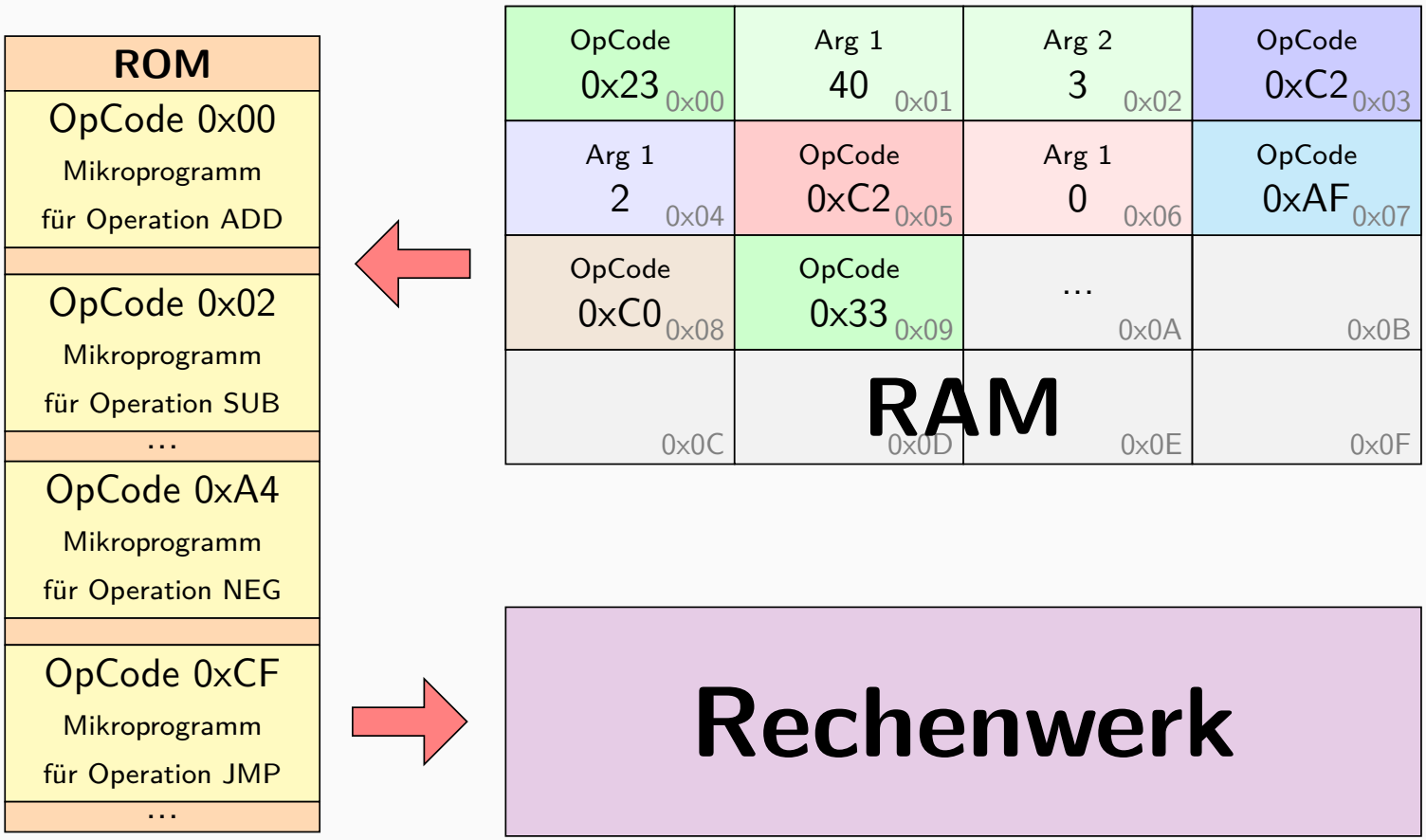
- Assembler Notation

```
MOV a, 3    ;; 3 -> a
MOV [i], a  ;; a -> [i]
LD  a1, 5   ;; 5 -> a1
```

- Reihenfolge ungewohnt:
  1. OpCode (oder Familie von Opcodes), z.B. `MOV`, `LD` wird durch 'Mnemonik' angegeben
  2. Zielregister, z.B. `a`, oder Adresse, z.B. `[i]`
  3. Quellregister, z.B. `a`, oder Wert, z.B. `3`
  4. Kommentare am Zeilenende durch Semikolon (`;;`)

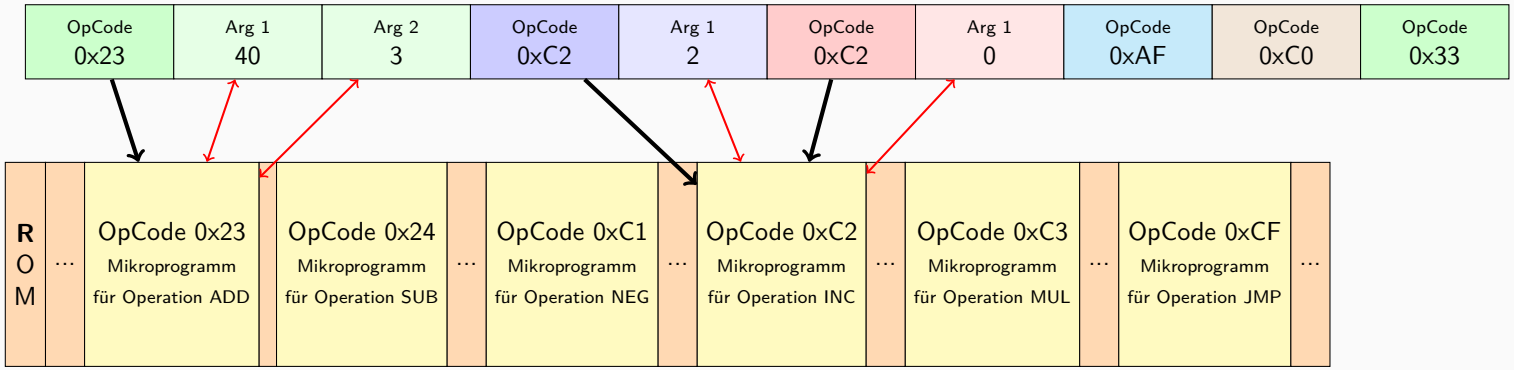
Maschinensprache (1/2)

- Programme im Hauptspeicher (RAM)
- verwenden Bausteine aus ROM
- sowohl Programm als auch Argumente liegen im RAM



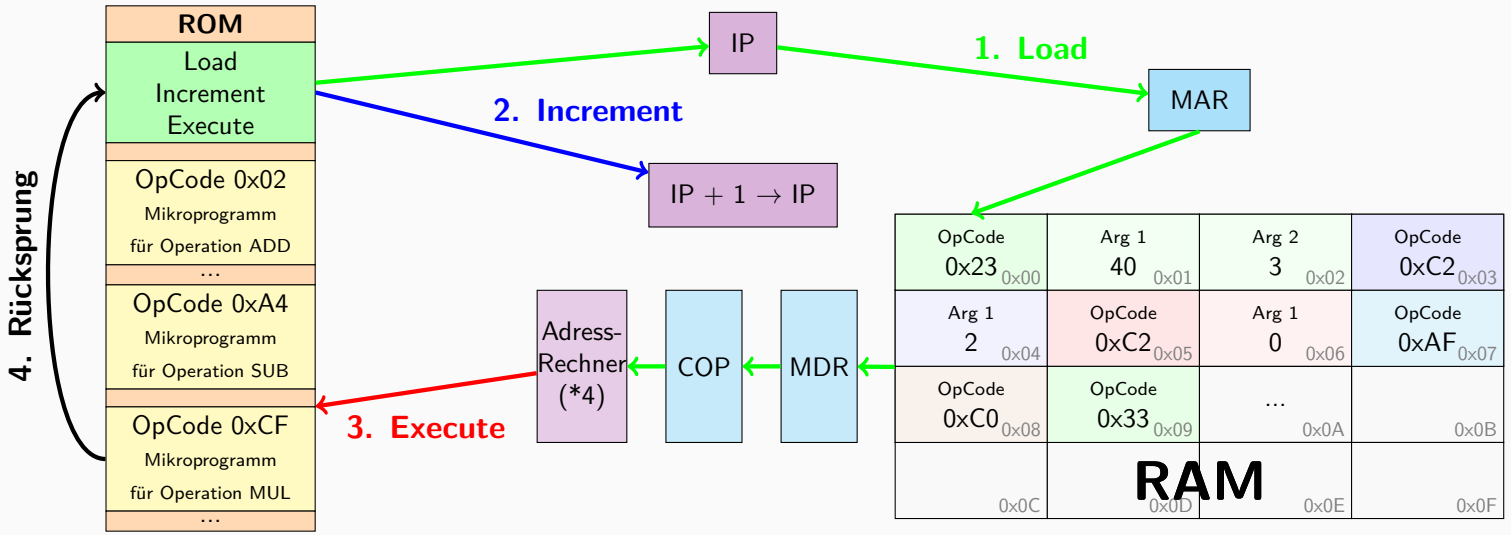
Maschinensprache (2/2)

- Mikroprogramme (Operations) holen sich ihre eigenen Argument selbst aus dem RAM



Maschinensprache Interpreter

- Instruction Pointer Register (IP, z.B. R0) zeigt auf nächste Operation
- Interpreter führt Programm im RAM aus: Load-Increment-Execute (LIE)
  1. Load: Laden des nächsten Befehls
  2. Increment: IP erhöhen
  3. Execute: Sprung im ROM zur nächsten Operation
  4. danach springt jede Operation wieder zu LIE



Load-Increment-Execute Implementierung

- Implementierung mit nur zwei Mikrobefehlen
- Befehl 1: Lade **R0** (Instruction Pointer) ins MAR

```
H   R0 → X     LOAD
R   Z := X
B   Z → MAR
```

- Befehl 2: OpCode lesen, **R0** (IP) erhöhen und auf 4·OpCode springen

```
RAM   lesen [MAR]→MDR
H     R0 → X, MDR→COP    INCREMENT
R     Z := INC X         EXECUTE
AR    CAR := 4·COP
B     Z → R0, MAR
```

Beispieloperation: Lesen in Register `MOV a, [m]` (1/2)

- Kopieren bzw. Verschieben (*move*)
- Befehl besteht aus zwei Werten im RAM
  - OpCode
  - Argument (Adresse) *m*
- Ablauf
  1. Argument/Adresse *m* aus RAM lesen (an IP+1)
  2. Wert von Adresse `[m]` aus RAM lesen
  3. in Akkumulatorregister A (z.B. R1) bringen
  4. IP erhöhen
  5. Rücksprung zu Load Increment Execute

Beispieloperation: Lesen in Register `MOV a, [m]` (2/2)

Name	OpCode	Schalter
LIE	0x01	H: IP $\rightarrow$ X R: Z := X, CAR++ B: Z $\rightarrow$ MAR
		H: lesen, MDR $\rightarrow$ COP, IP $\rightarrow$ X R: Z := INC X, CAR := 4COP B: Z $\rightarrow$ IP, MAR
...	...	...
MOV A, [m]	0x02	H: lesen, MDR $\rightarrow$ Y R: Z := Y, CAR++ B: Z $\rightarrow$ MAR
		H: lesen, MDR $\rightarrow$ Y R: Z := Y, CAR++ B: Z $\rightarrow$ A (R1)
		H: IP (R0) $\rightarrow$ X R: Z := INC X, CAR := 4CN, CN=1 B: Z $\rightarrow$ IP (R0)

Überblick: Inhalt des ROM

- Segment 0: Init
  - Initialisierung aller Register mit 0
  - Initialisierung des Stack Zeigers auf Maximum
- Segment 1: Load-Increment-Execute (LIE)
- Segment 2-255: alle anderen Operationen
  - Viele Segmentnummern sind mögliche Einsprungpunkte
  - Manche Operationen benötigen mehrere Segmente
  - Jede Operation springt am Ende zu LIE zurück

Abbildung von Assembler auf OpCodes

Argumente von OpCodes können sein

- **i** immediate unmittelbarer, direkter Werte, z.B. `3`
- **m** memory Speicheradresse z.B. `[5]`, oder `[a]`
- manche OpCodes haben keinen direkten Parameter, sondern werden ja nach Assemblerargument auf verschiedenen OpCodes abgebildet, z.B.

```
MOV a, b ;; OpCode 10: a := b
MOV b, a ;; OpCode 11: b := a
```

Assembler Tabelle

- Tabelle mit Liste der OpCodes vom CPU Hersteller
  - verschiedene Parameter (keine “-”, Adresse “m”, oder Wert “i”)

OpCode	Arg	Mnemonik	Operation
0	-	`INIT`	A=B=0 alle Register werden mit 0 initialisiert
…	…	…	…
10	-	`ADD A, B`	A=A+B
11	-	`ADD B, A`	B=B+A Ergebnis nach B
12	-	`INC A`	A=A+1
13	-	`DEC A`	A=A-1
…	…	…	…
15	m	`MOV A, [n]`	[n]→A Wert vom RAM an Adresse n → Register A
16	m	`MOV B, [n]`	[n]→B
17	m	`MOV [n], A`	A→[n] Wert von Register A → Adresse n im RAM
18	m	`MOV [n], B`	B→[n]
19	i	`MOV A, x`	x→A Wert x (direkt im Quelltext) → Register A
…	…	…	…
25	i	`JMP m`	m→IP Sprung zu OpCode im RAM an Adresse m
…	…	…	…

Übersetzungsvorgang Assembler → Maschinensprache

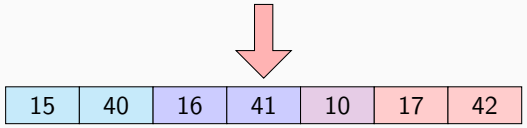
- Assembler übersetzt Quelltext in Maschinensprache
  - mit Hilfe der CPU-abhängigen Tabelle

Quelltext

```
;; z.B. Addition
;; [40] + [41] → [42]
 MOV A, [40]
 MOV B, [41]
 ADD A, B
 MOV [42], A
```

Op	Mnemonik
0	INIT
…	…
10	ADD A, B
11	ADD B, A
12	INC A
13	DEC A
…	…
15	MOV A, [n]
16	MOV B, [n]
17	MOV [n], A
18	MOV [n], B
19	MOV A, x
…	…
25	JMP m
…	…

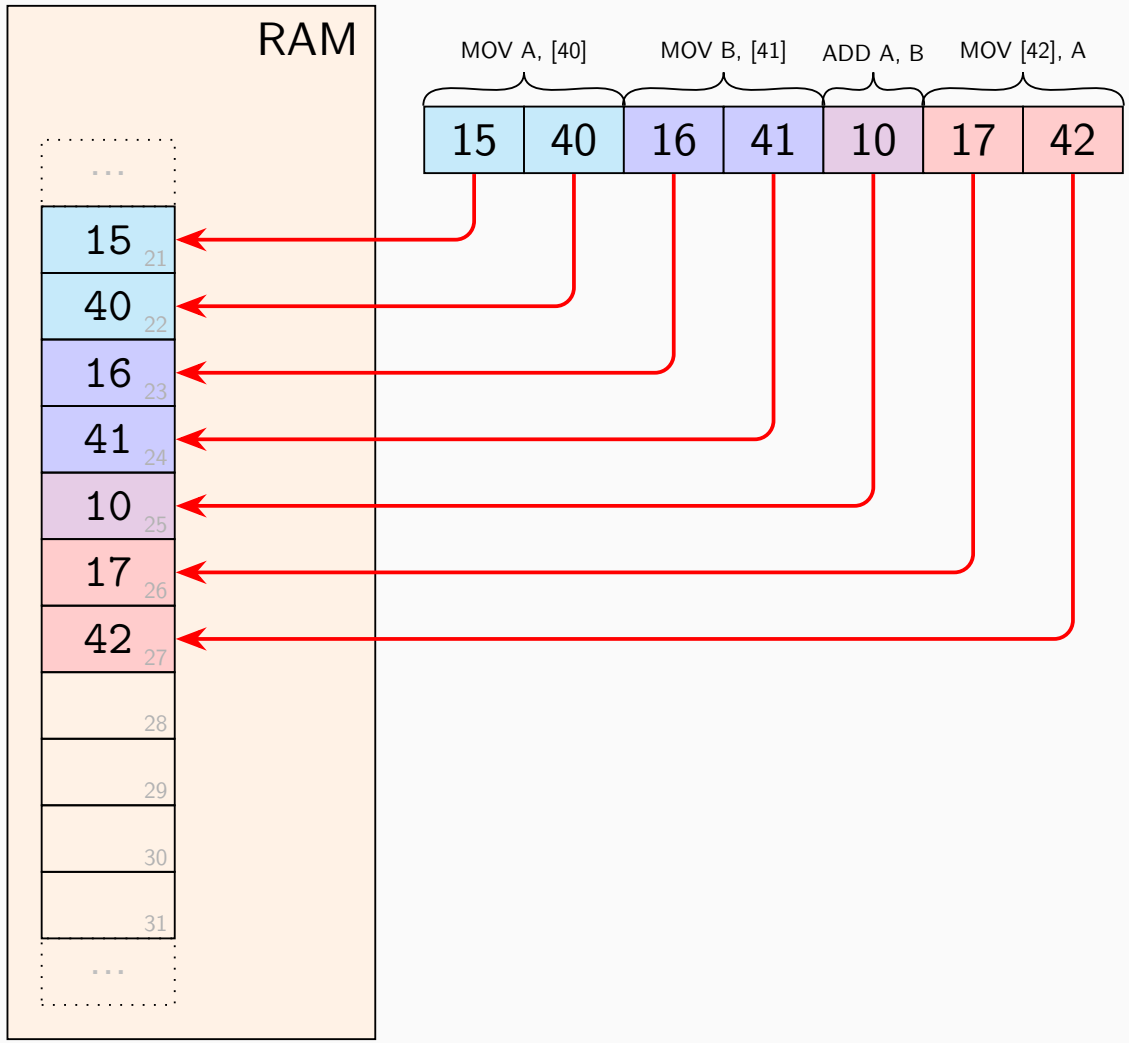
```
MOV A, [40] ;; → 15 40
MOV B, [41] ;; → 16 41
ADD A, B    ;; → 10
MOV [42], A ;; → 17 42
```



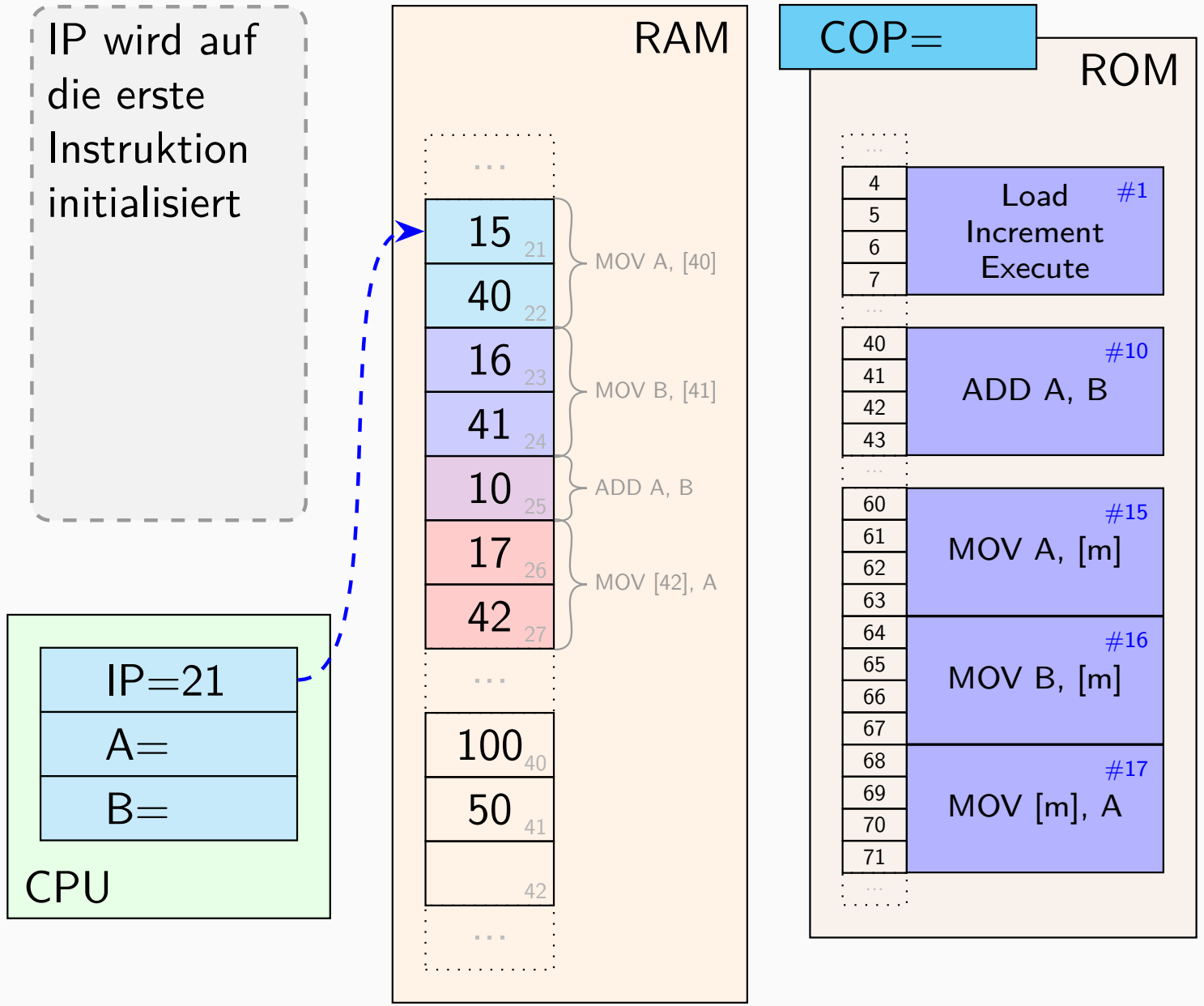
Maschinensprache

Laden des Programs

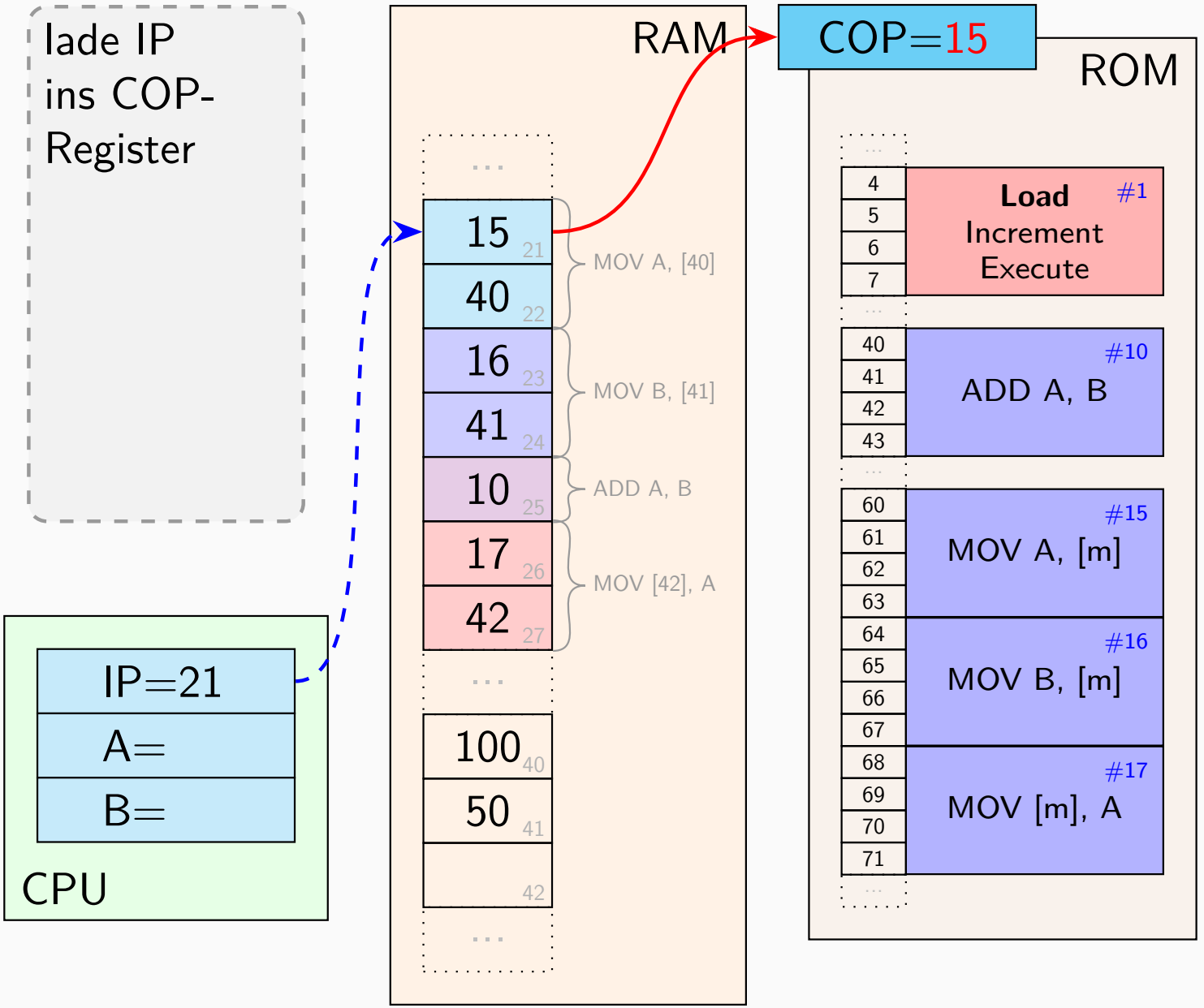
Die vom Assembler erzeugte Bytefolge wird vom Betriebs-system in den RAM geladen



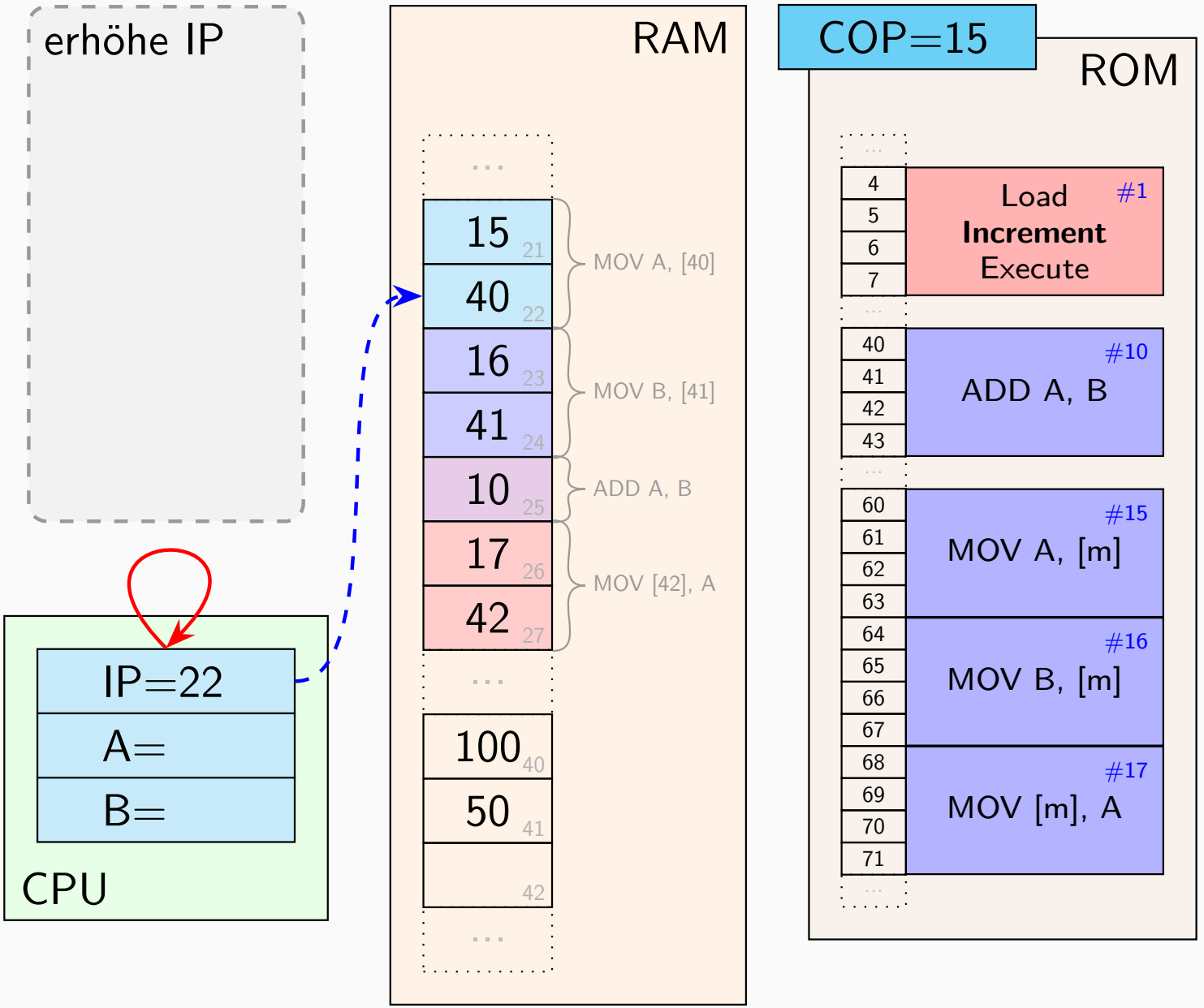
Init


IP wird auf die erste Instruktion initialisiert
RAM
15 21
40 22
16 23
41 24
10 25
17 26
42 27
100 40
50 41
42
MOV A, [40]
MOV B, [41]
ADD A, B
MOV [42], A
IP=21
A=
B=
CPU
COP=
ROM
4 5 6 7
Load
Increment
Execute
#1
40 41 42 43
ADD A, B
#10
60 61 62 63
MOV A, [m]
#15
64 65 66 67
MOV B, [m]
#16
68 69 70 71
MOV [m], A
#17

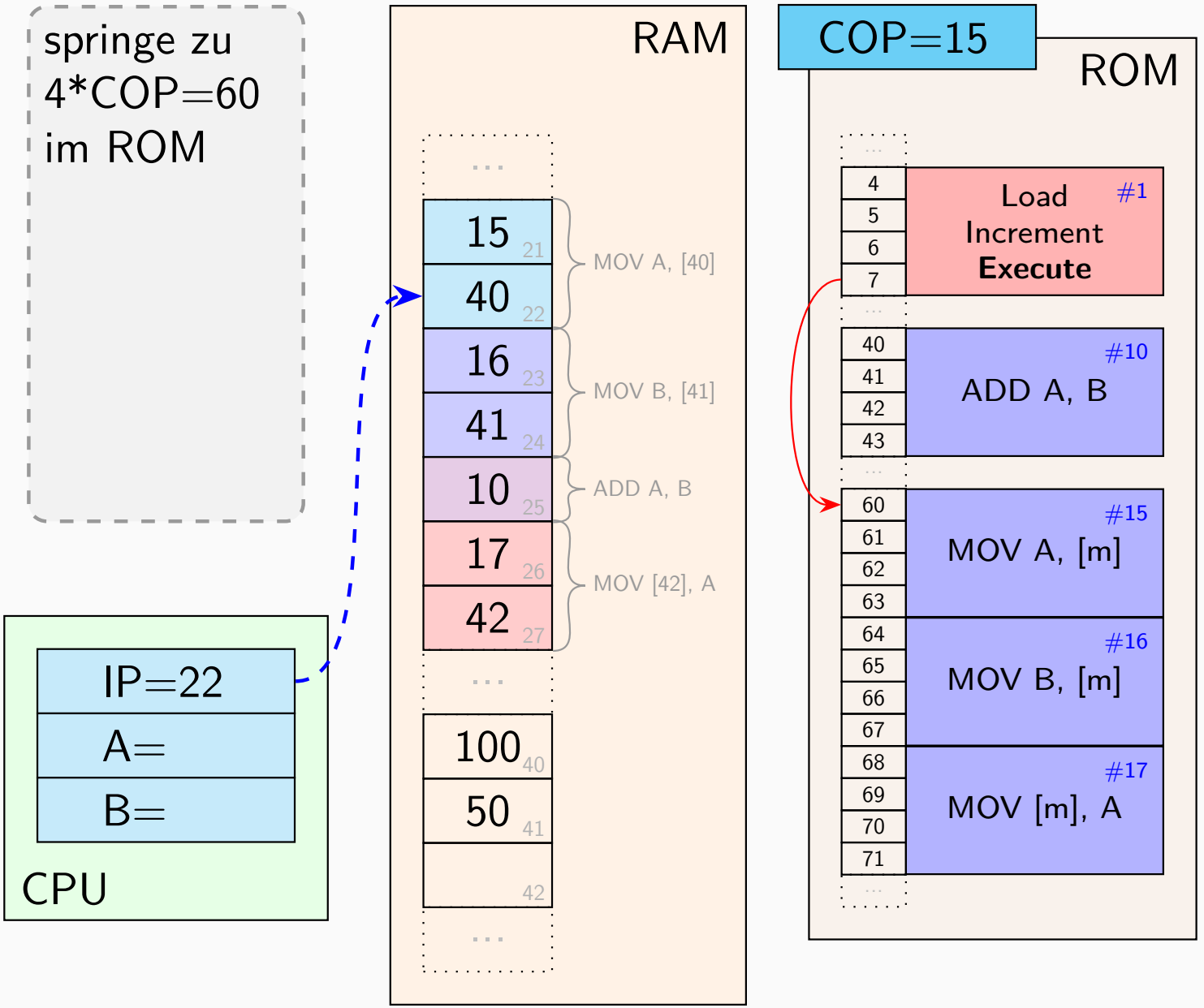
LIE Load


lade IP
ins COP-
Register
RAM
COP=15
ROM
15
21
40
22
16
23
41
24
10
25
17
26
42
27
MOV A, [40]
MOV B, [41]
ADD A, B
MOV [42], A
100
40
50
41
42
IP=21
A=
B=
CPU
4
5
6
7
Load
Increment
Execute
#1
40
41
42
43
ADD A, B
#10
60
61
62
63
MOV A, [m]
#15
64
65
66
67
MOV B, [m]
#16
68
69
70
71
MOV [m], A
#17

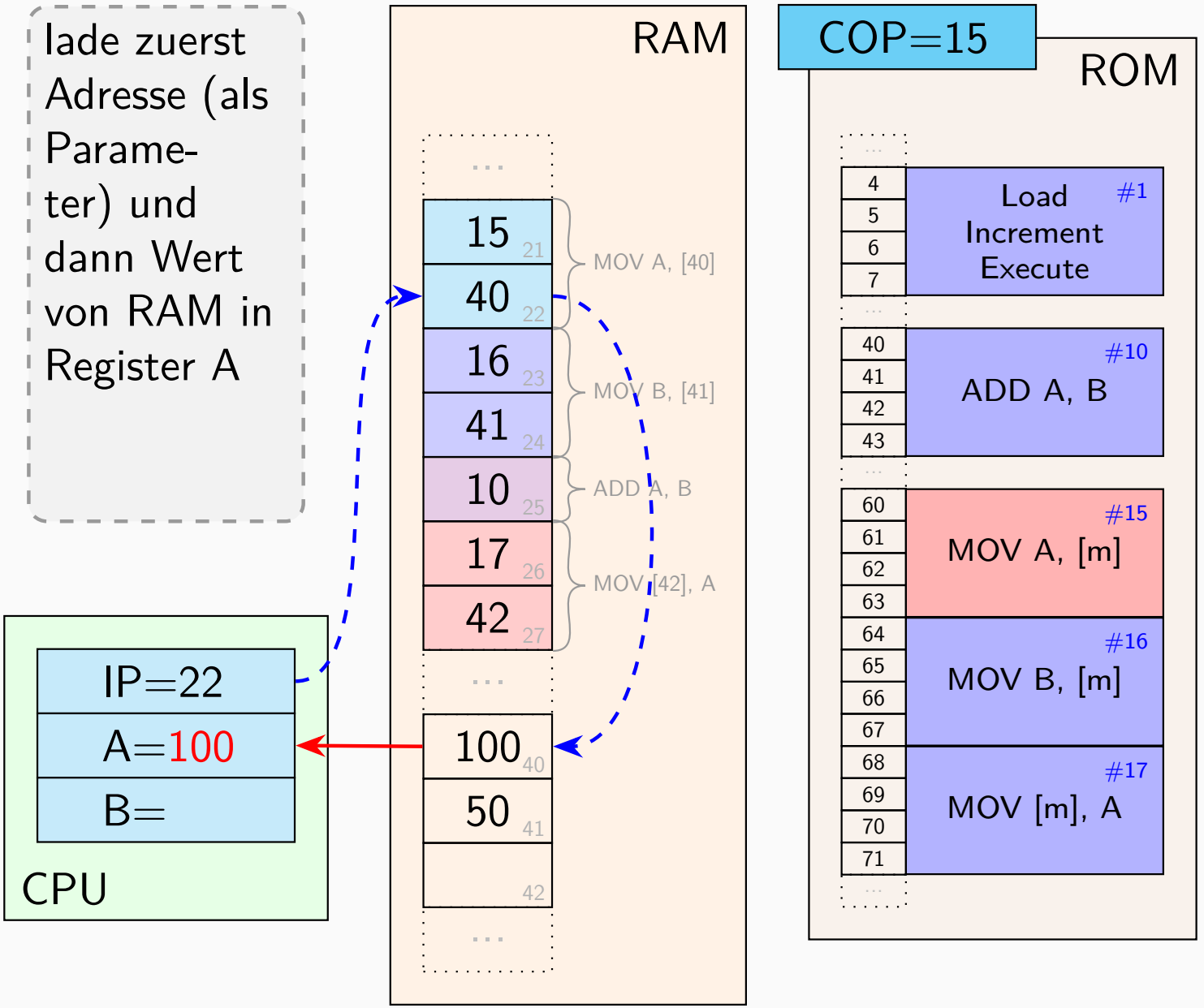
LIE Increment



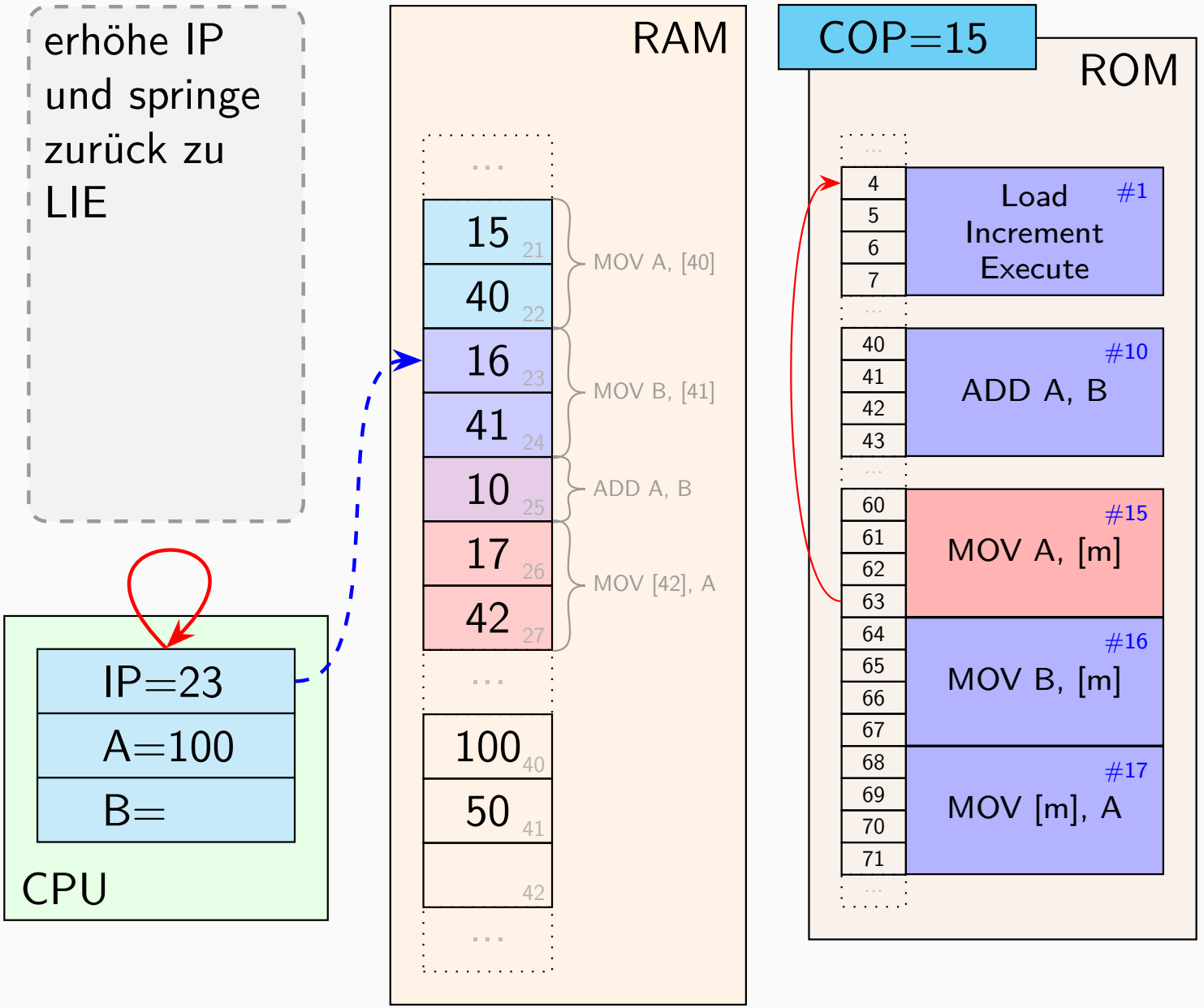
LIE Execute


springe zu
4*COP=60
im ROM
RAM
15
21
40
22
16
23
41
24
10
25
17
26
42
27
100
40
50
41
42
MOV A, [40]
MOV B, [41]
ADD A, B
MOV [42], A
IP=22
A=
B=
CPU
COP=15
ROM
4
5
6
7
Load
Increment
Execute
#1
40
41
42
43
ADD A, B
#10
60
61
62
63
MOV A, [m]
#15
64
65
66
67
MOV B, [m]
#16
68
69
70
71
MOV [m], A
#17

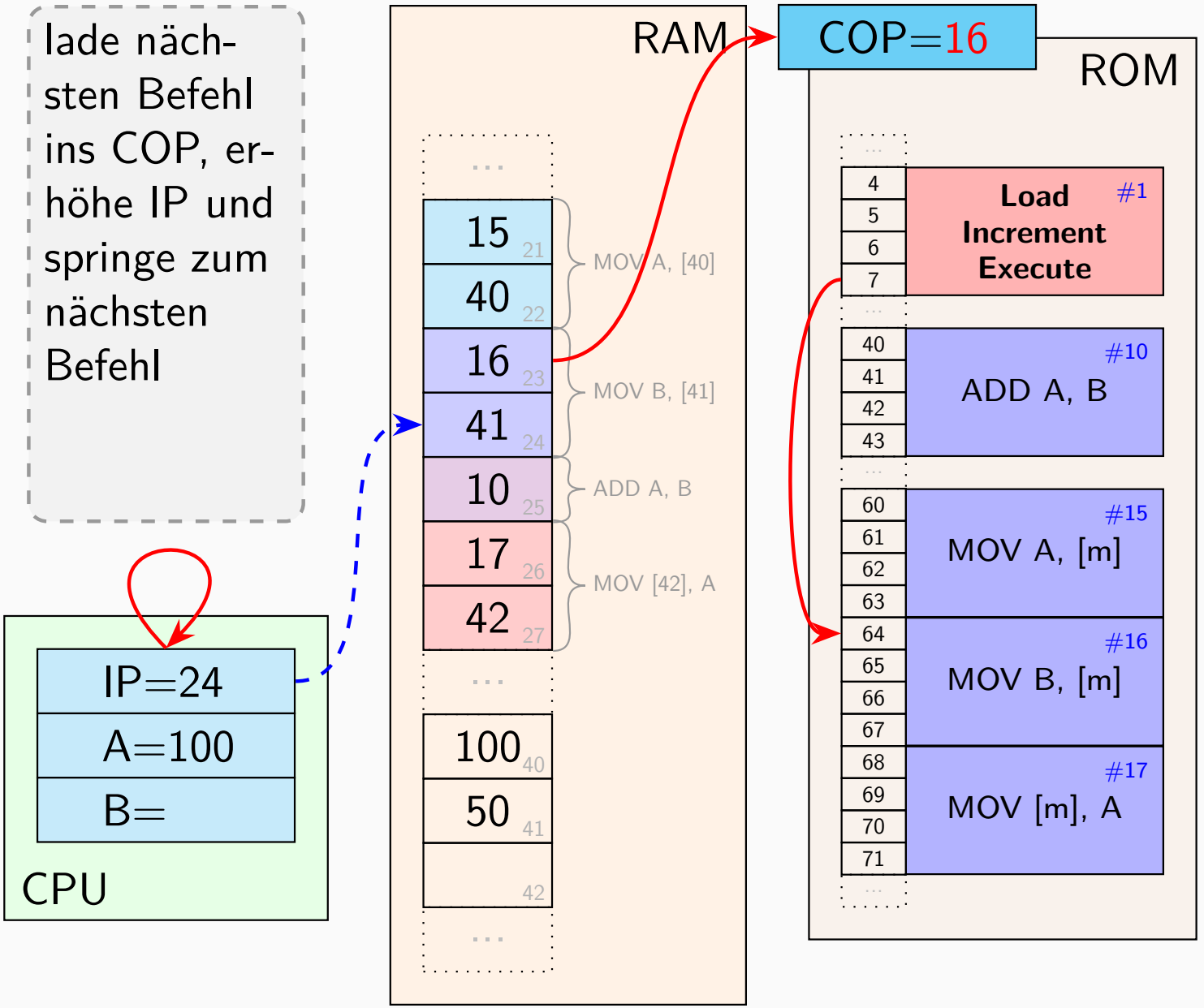
MOV A, [m]: Read


lade zuerst Adresse (als Parame-ter) und dann Wert von RAM in Register A
RAM
15 21
40 22
16 23
41 24
10 25
17 26
42 27
100 40
50 41
42
MOV A, [40]
MOV B, [41]
ADD A, B
MOV [42], A
IP=22
A=100
B=
CPU
COP=15
ROM
4 5 6 7
Load Increment Execute
#1
40 41 42 43
ADD A, B
#10
60 61 62 63
MOV A, [m]
#15
64 65 66 67
MOV B, [m]
#16
68 69 70 71
MOV [m], A
#17

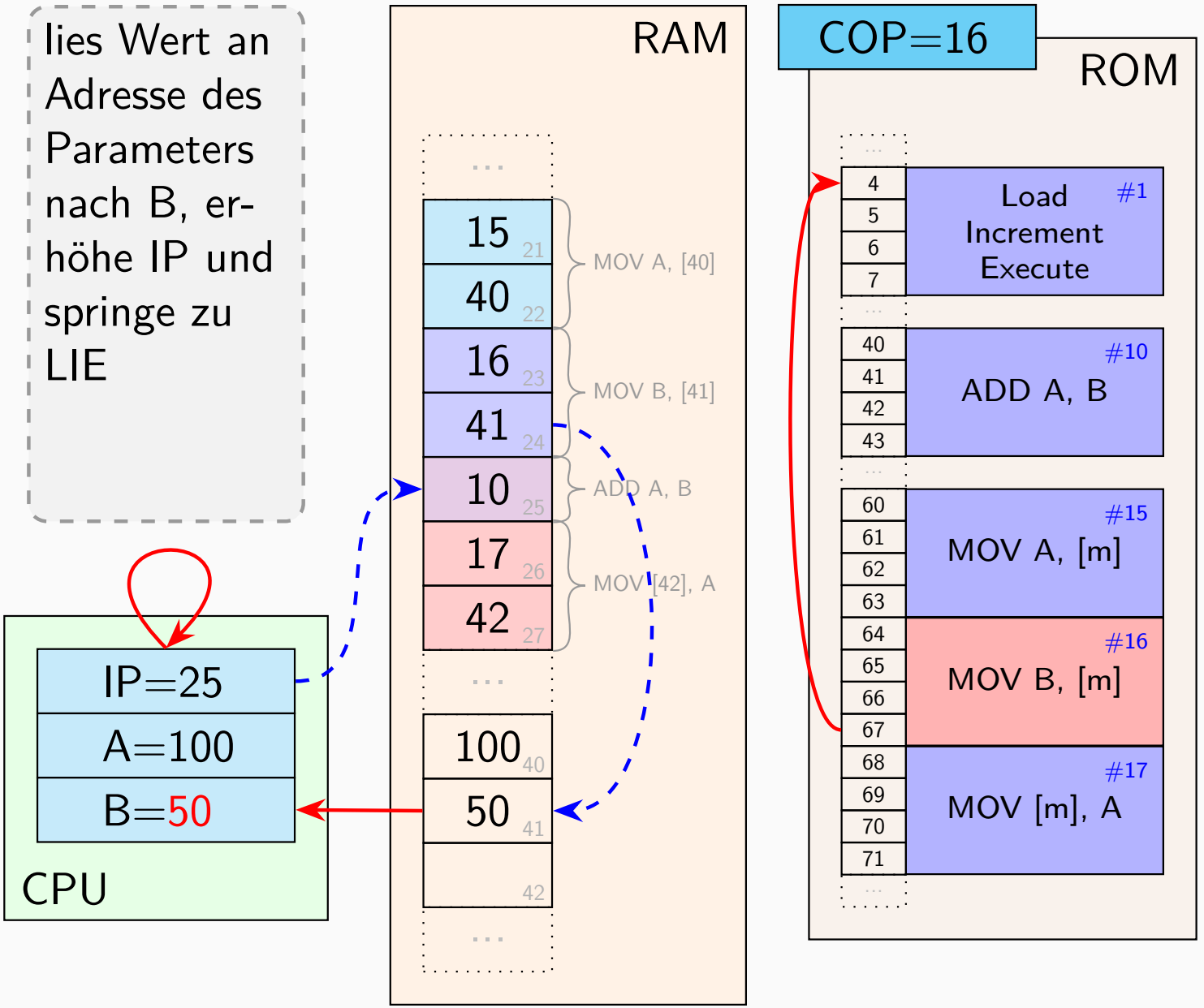
MOV A, [m]: Increment

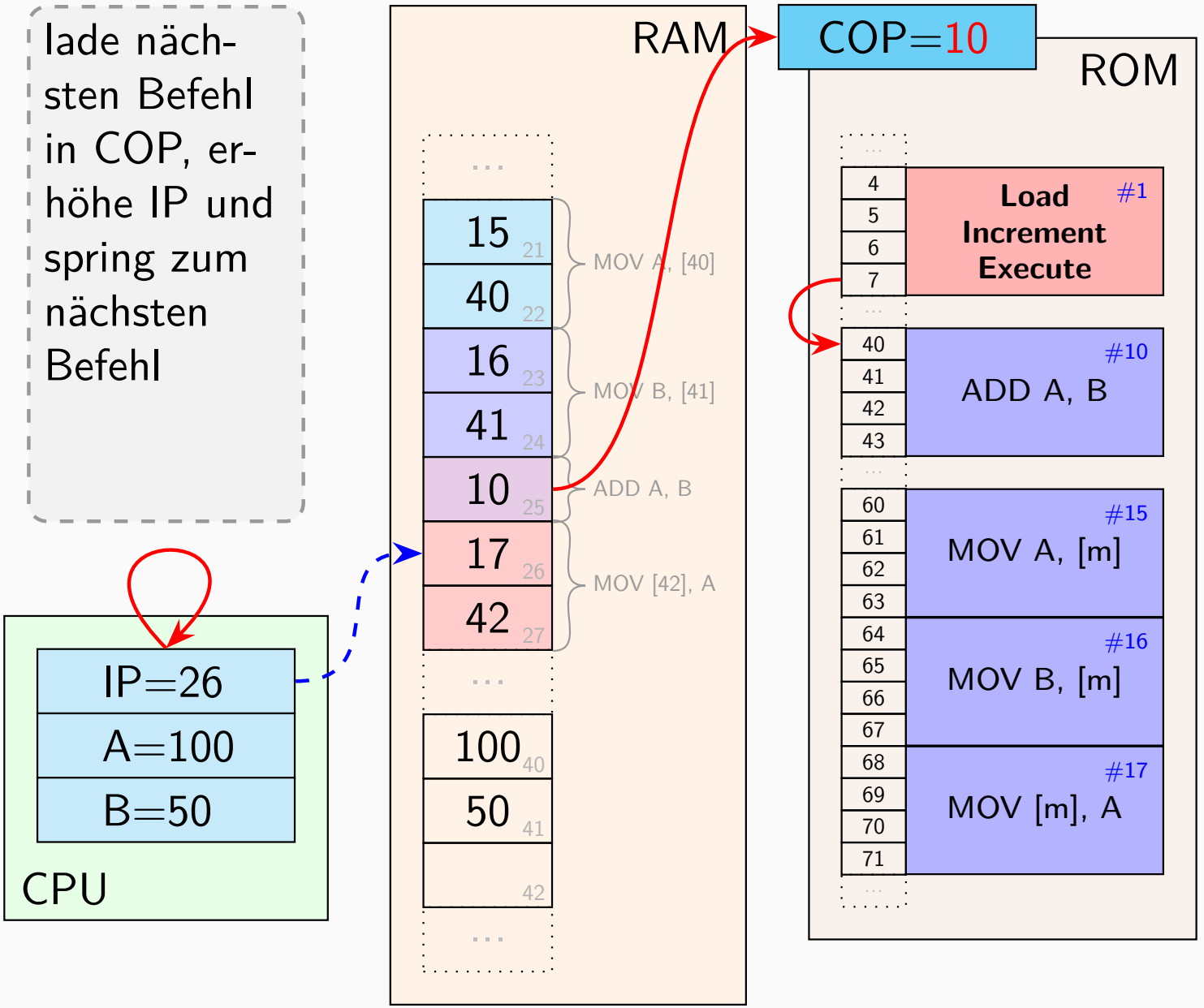


LIE


lade nächsten Befehl ins COP, erhöhe IP und springe zum nächsten Befehl
IP=24
A=100
B=
CPU
RAM
15 21
40 22
16 23
41 24
10 25
17 26
42 27
100 40
50 41
42
MOV A, [40]
MOV B, [41]
ADD A, B
MOV [42], A
COP=16
ROM
4 5 6 7
Load Increment Execute
#1
40 41 42 43
ADD A, B
#10
60 61 62 63
MOV A, [m]
#15
64 65 66 67
MOV B, [m]
#16
68 69 70 71
MOV [m], A
#17

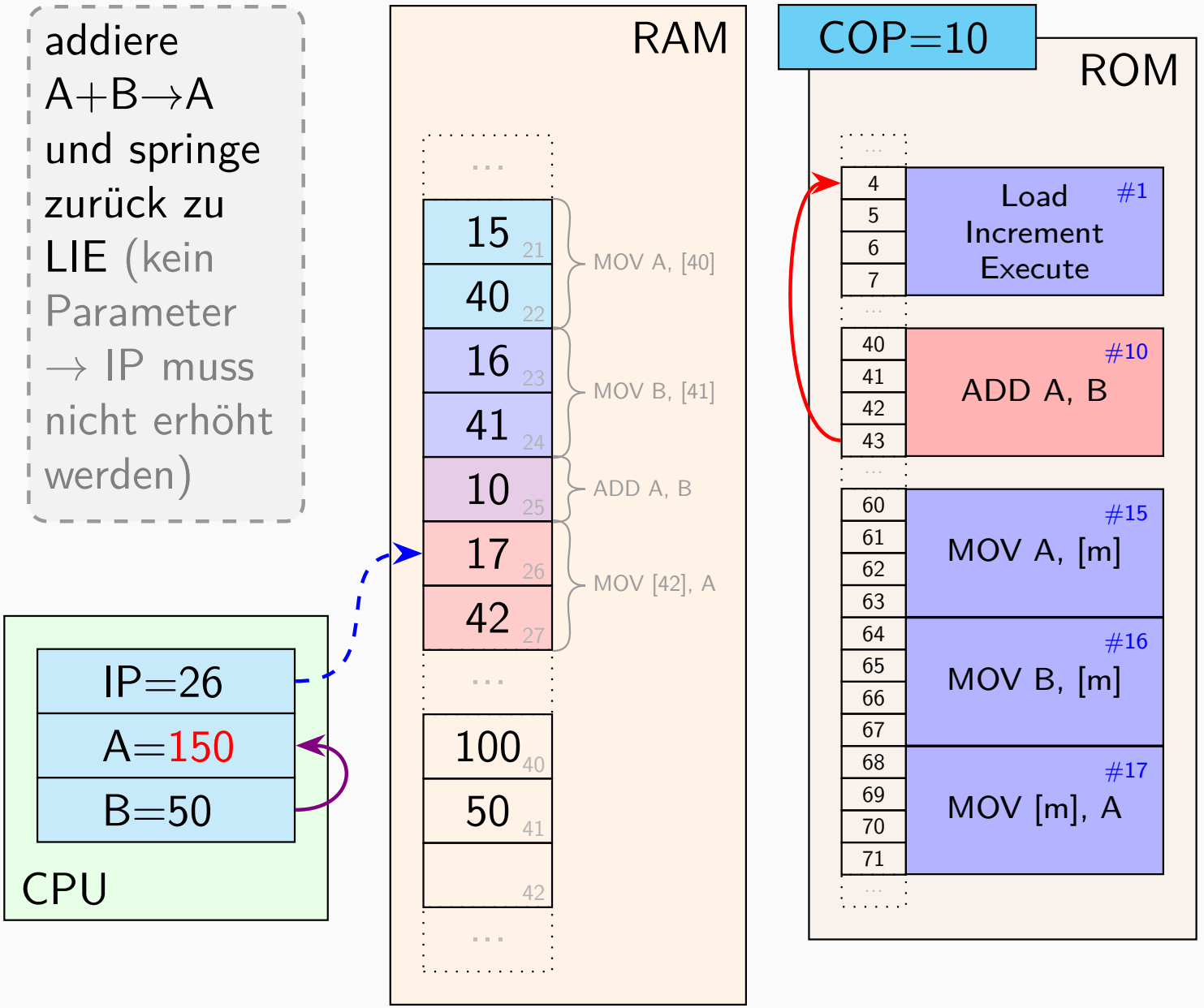
MOV B, [m]: Read & Increment



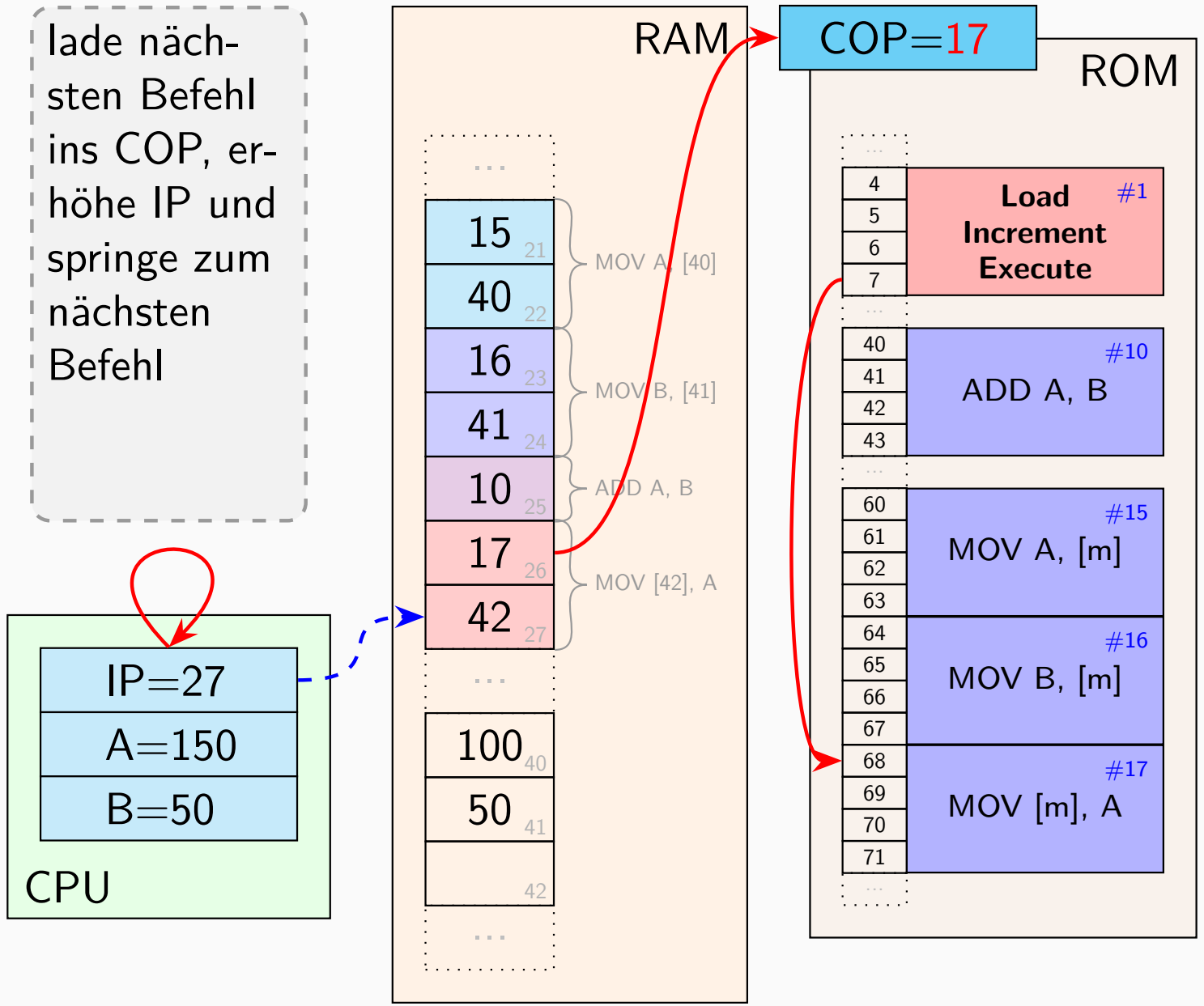

lade nächsten Befehl in COP, erhöhe IP und spring zum nächsten Befehl
IP=26
A=100
B=50
CPU
RAM
15 21
40 22
16 23
41 24
10 25
17 26
42 27
100 40
50 41
42
MOV A, [40]
MOV B, [41]
ADD A, B
MOV [42], A
COP=10
ROM
4
5
6
7
Load
Increment
Execute
#1
40
41
42
43
ADD A, B
#10
60
61
62
63
MOV A, [m]
#15
64
65
66
67
MOV B, [m]
#16
68
69
70
71
MOV [m], A
#17

ADD

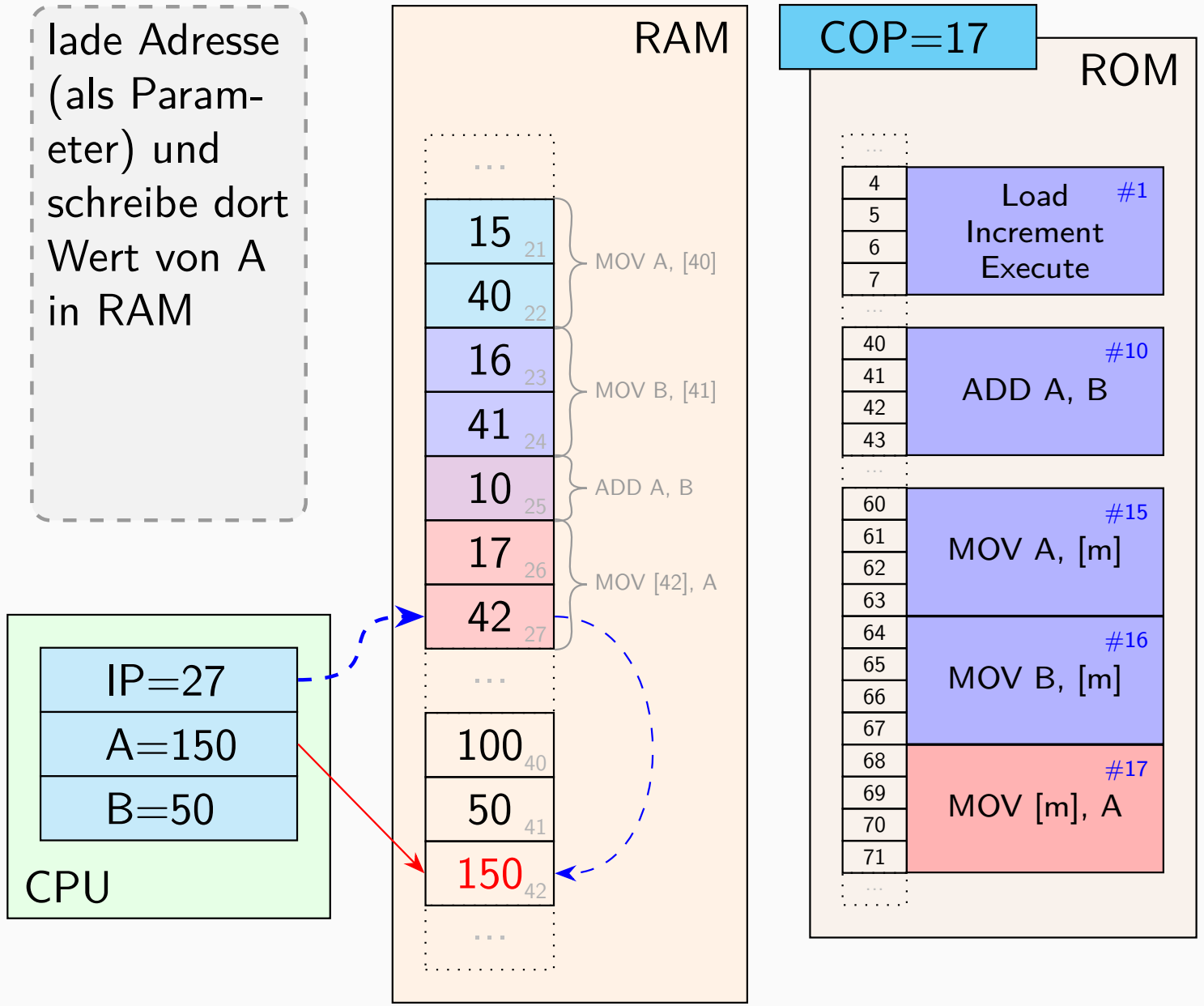
addiere A+B→A und springe zurück zu LIE (kein Parameter → IP muss nicht erhöht werden)



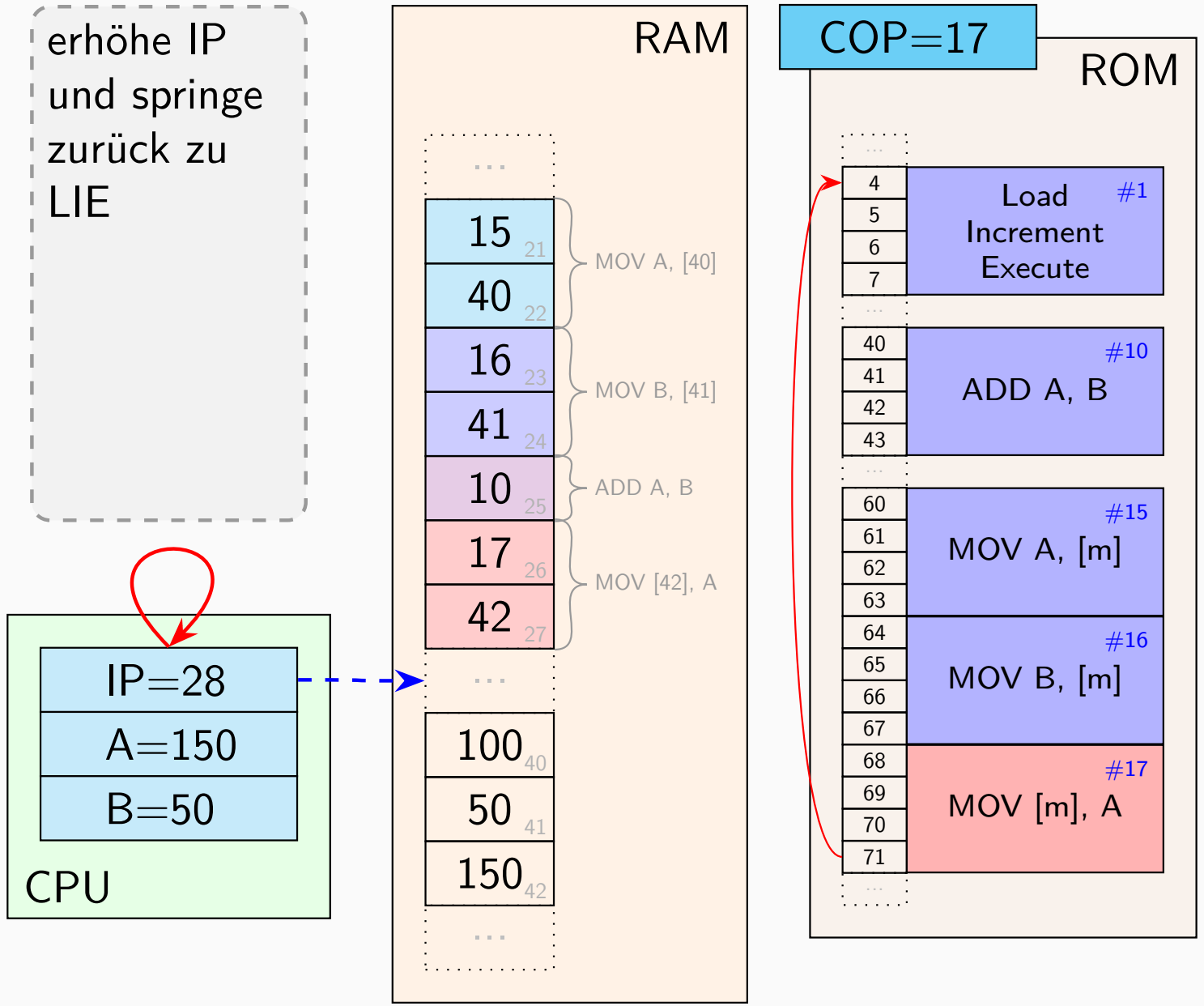
LIE


lade nächsten Befehl ins COP, erhöhe IP und springe zum nächsten Befehl
IP=27
A=150
B=50
CPU
RAM
15 21
40 22
16 23
41 24
10 25
17 26
42 27
100 40
50 41
42
MOV A, [40]
MOV B, [41]
ADD A, B
MOV [42], A
COP=17
ROM
4
5
6
7
Load
Increment
Execute
#1
40
41
42
43
ADD A, B
#10
60
61
62
63
MOV A, [m]
#15
64
65
66
67
MOV B, [m]
#16
68
69
70
71
MOV [m], A
#17

MOV [m], A: Write

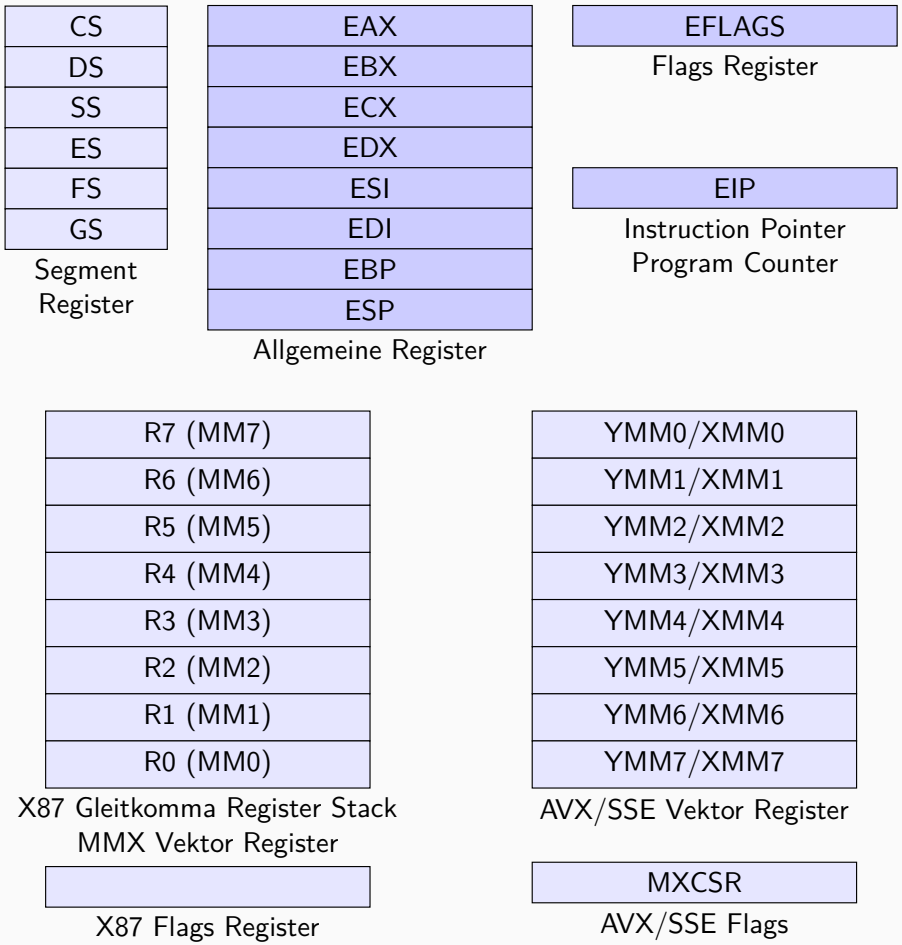

lade Adresse (als Parameter) und schreibe dort Wert von A in RAM
RAM
15 21
40 22
16 23
41 24
10 25
17 26
42 27
100 40
50 41
150 42
MOV A, [40]
MOV B, [41]
ADD A, B
MOV [42], A
IP=27
A=150
B=50
CPU
COP=17
ROM
4
5
6
7
Load
Increment
Execute
#1
40
41
42
43
ADD A, B
#10
60
61
62
63
MOV A, [m]
#15
64
65
66
67
MOV B, [m]
#16
68
69
70
71
MOV [m], A
#17

MOV [m], A: Increment



Register in der Intel x86-32 Architektur

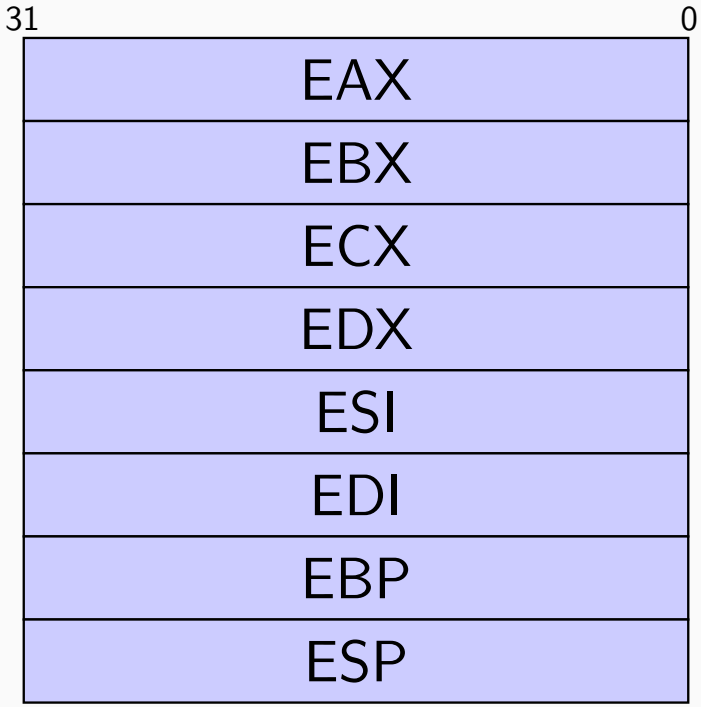
- Register haben großteils spezielle Funktion



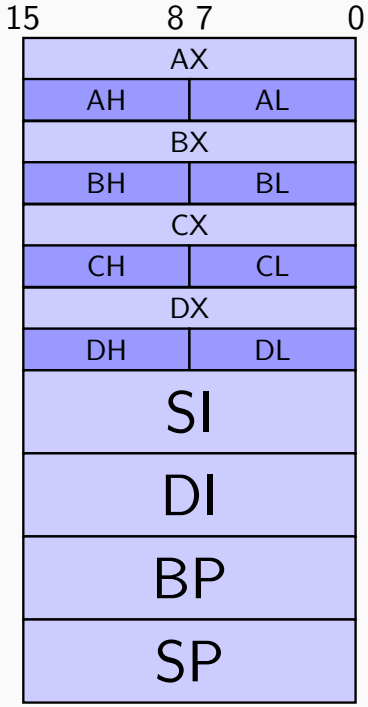
Quelle: Daniel Kusswurm, “Modern x86 Assembly Language Programming”, Apress, 2014

Intel x86-32 Subregister

- Teile der Register können direkt angesprochen werden
  - die unteren 16 Bit von **EAX** als **AX**, und die unteren 8 Bit als **AL**



32-Bit Register



16-Bit und 8-Bit Register

Quelle: Daniel Kusswurm, “Modern x86 Assembly Language Programming”, Apress, 2014

Übliche Verwendung der Register

- EAX: Akkumulator
- EBX: Zeiger, Basis-Register (Basis-Zeiger)
- ECX: Schleifen, Zähler (Counter)
- EDX: Integer Multiplikation und Division
- ESI: Quellindex (Source)
- EDI: Zielindex (Destination)
- ESP: Stack Zeiger (Stack Pointer)
- EBP: Stack-Frame Zeiger (Base Pointer)

Beispiele für Instruktionen

```
MOV eax, 0    ;; eax ← 0
MOV ebx, [x] ;; ebx ← Wert an Adresse x
INC ecx
PUSH eax      ;; Lege EAX auf den Stack
JNZ label     ;; Sprung zu label, falls
              ;; .. letzter Wert ungleich 0

XOR eax, eax ;; eax ← 0
OR eax, eax  ;; EAX durch ALU schleusen (Flags setzen)
JZ label      ;; .. dann Sprung falls 0
```

Assembler Beispiel 1

- Summe der Zahlen n bis 0

```
 MOV ecx, [0h] ;; Inhalt von Adresse 0 nach ECX (n)
 MOV eax, 0    ;; Initialisieren von EAX (Summe) mit 0
repeat:        ;; Sprungziel (Label) für Schleife
 ADD eax, ecx  ;; Addition EAX + ECX → EAX
 DEC ecx       ;; Herunterzählen von ECX (n)
 JNZ repeat    ;; Sprung zu 'repeat', falls
               ;; .. ECX noch nicht 0
```

OpCode Argumente

- Typen von Argumenten
  - Konstanten / unmittelbarer Wert (immediate)
  - Register
  - RAM Adressen
- Intel Architektur mit komplexen Kombinationen

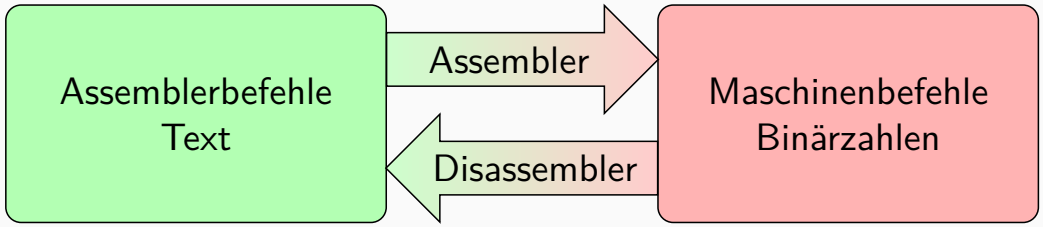
```
MOV eax, 12                ;; Konstanten / immediate
MOV eax, edx               ;; Register
MOV eax, [ebx]             ;; Adresse aus Register
MOV eax, [ebx+10]          ;; Register mit fixem Offset,
                           ;; .. z.B. Record
MOV eax, [ebx+esi]         ;; mit variablem Offset,
                           ;; .. z.B. Array Index
MOV eax, [ebx+esi*8+4]     ;; Kombination, z.B.
                           ;; .. Record in Array
```

Instruktionsformate

- OpCodes haben unterschiedliche ...
  - Länge des OpCodes selbst
  - Anzahl von Argumenten
  - Längen der Adressen
  - Gesamtlänge der Instruktion
- Designkriterien
  - Evolution des Befehlssatzes (Rückwärtskompatibilität)
  - Speicherplatz (kurzes Format für häufige Befehle)
  - Dekodiergeschwindigkeit (komprimiertes Format langsamer)
  - (zukünftige) Anzahl von Operationen
  - Adressierungsgranularität (Bits, Bytes, Wörter)

Assembler: Programm vs. Sprache

- Maschinenbefehl = Bitfolge im RAM
- Assembler (Sprache): Lesbare Darstellung
  - “merkbare” Abkürzungen, sogenannte Mnemonics



- Assemblerbefehle können durch Disassembler rekonstruiert werden
- weitere Features jedoch nicht, z.B.
  - Namen von Variablen oder Konstanten
  - Namen aller Prozeduren
  - Kommentare
  - Makros

Übersetzungsvorgang

- 1. Durchlauf
  - Syntaktische Analyse
  - Adressvergabe: Variablen, Label
- 2. Durchlauf
  - Maschinencode generieren

```
value:  dw 0xffff
name:   db "test", 0
result: resw 1

start:  mov eax, [value]
        mov esi, [name]
loop:   add eax, esi
        mov [result], eax
        jnz loop
```

```
00: dw 0xffff       ;; value → 00
04: db "test", 0    ;; name → 04
09: resw 1          ;; result → 09

13: mov eax, [00]   ;; start → 13
    mov esi, [04]
17: add eax, esi    ;; loop → 17
    mov [09], eax
    jnz 17
```

Speicherorganisation

- Früher
  - segmentierter Speicher
  - Kleiner Adressraum (<1MB) mit 16 Bit Adressen
  - 2^{16} Bytes = 64KiB mit Zeiger adressierbar
  - daher Zusammensetzung eines Zeigers aus Segment+Adresse
  - `SEGMENT:ADRESSE` (4+16 = 20 Bit, 2^{20} Byte = 1MiB)
  - Segmentregister: CS, DS, ES, FS, GS
  - teilweise überlappende Adressen
  - komplex für Compiler, Programmierer und Hardware
- Heute
  - flaches Speichermodell
  - 2^{32} / 2^{64} Byte = 4 GiB / 16 EiB pro Zeiger adressierbar
  - Segmentregister fast bedeutungslos

Befehle: Überblick (1/6)

- Häufig großer Umfang an verschiedenen Befehlen
- Intel x86-64
  - ca. 1000 Operationen mit ca. 3500 OpCodes
  - z.B. Mnemonic `MOV` je nach Operanden verschiedene OpCodes
- Intel x86 Software Developer's Manual mit ca. 5000 Seiten

Befehle: Verschiedene (2/6)

- Datentransfer
  - `mov`, `push`, `xchg`, ...
  - `lod`, `sto`, ...
- Datenkonvertierung
  - `cbw`, `bswap`, `xlatb`, ...
- Datenvergleich (`cmp`, ...)
- Bit- & Bytefelder (`setne`, ...)
- Abfrage und setzen von Flags (`cld`, `stc`, `pushfd`, ...)
- Andere (`popcnt`)
- Erweiterungen
  - Floating Point Stack (FPU, x87)
  - Single Instruction Multiple Data (SIMD), Vektorrechnung
    - MMX, SSE, AVG
  - Scalar Floating Point (ohne Stack)
    - SSE, AVX

Befehle: Arithmetik & Logik (3/6)

```
ADD eax, ebx   ;; Addition
ADD ax, bx     ;; Addition (16 Bit)
ADD ah, bh     ;; Addition (8 Bit)
SUB eax, ebx   ;; Subtraktion
INC eax        ;; Increment
DEC eax        ;; Decrement
NEG eax        ;; Negation (2K)
MUL ebx        ;; edx:eax = eax * ebx
DIV ebx        ;; edx:eax / ebx → eax,
               ;; edx:eax % ebx → edx

AND eax, ebx     ;; bitweises UND
TEST eax, 1010b  ;; Bitmaske
NOT eax          ;; Bitflip
```

Befehle: Bitverschiebungen (4/6)

- Bitshift (ohne oder mit Sign-Extension = Arithmetic Shift)
- Rotate, verschobene Bits werden auf der anderen Seite wieder eingefügt

```
SHL eax, 1 ;; shift left 0010₂ → 0100₂ (2₁₀ → 4₁₀)
SAR eax, 1 ;; shift arith. right 1010₂ → 1101₂ (-6₁₀ → -3₁₀)
ROL eax, 1 ;; rotate left 1100010 → 1000101
RCR eax, 1 ;; rotate with carry 010011 0 → 001001 1
```

Befehle: Zeichenketten (5/6)

- operieren auf Source Index (SI) und Destination Index (DI) Registern
- berücksichtigen Direction Flag (DF)
- erhöhen/verringern SI/DI automatisch
- keine expliziten Argumente

```
LODSB ;; load string byte DS:ESI → AL
STOSB ;; store AL → ES:EDI
SCASB ;; scan (compare) AL ↔ ES:EDI
CMPSB ;; compare DS:ESI ↔ ES:EDI
MOVSB ;; move byte DS:ESI ↔ ES:EDI
CLD   ;; clear direction flag (left to right)
REP   ;; repeat next instruction ECX times
REPNZ ;; repeat while not zero, up to ECX times
```

Befehle: Ablaufsteuerung (6/6)

- Sprünge & Verzweigungen
  - bedingte und unbedingte Sprünge zu Label
  - unter Berücksichtigung verschiedener Flags

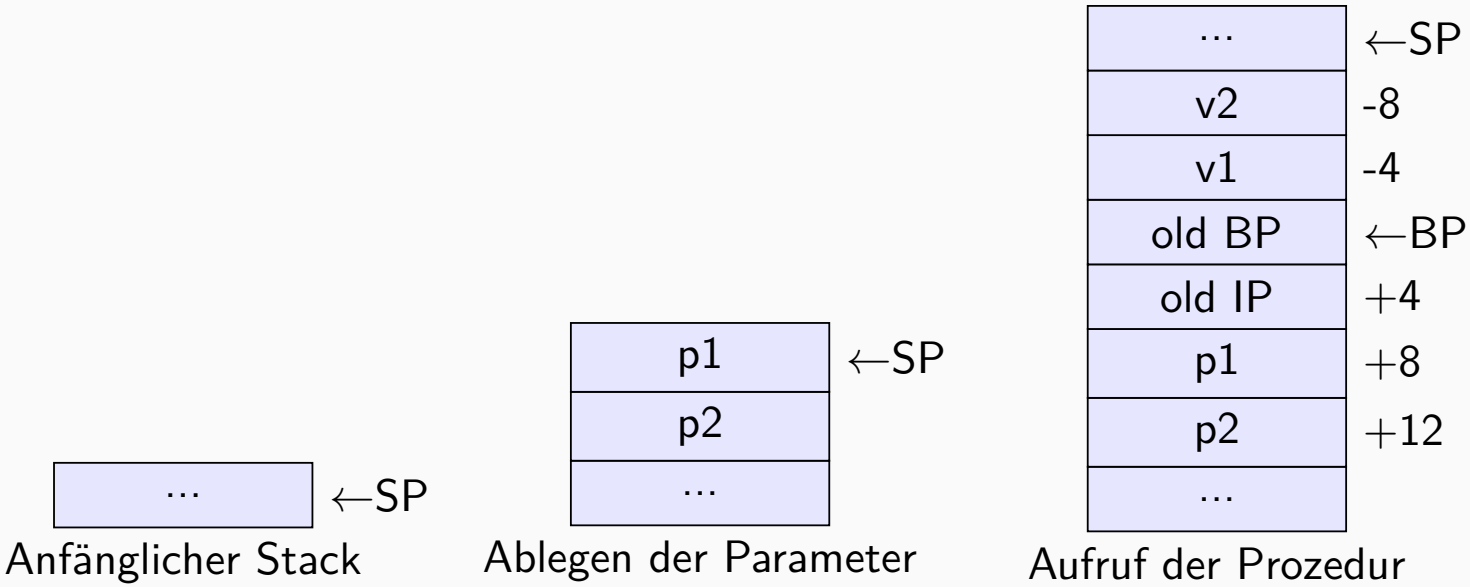
```
 JMP end ;; unbedingter Sprung
 JNZ end ;; falls kein Zero Flag gesetzt
 JAE end ;; falls Above or Equal (≥ 0)
 JGE end ;; falls Greater or Equal (2K)
 JPE end ;; falls Parity Even (gerade Parität)
end:     ;; Sprungziel
```

- Subroutinen

```
 CALL proc ;; Aufruf der Prozedur proc
           ;; push IP, JMP proc
proc:
  ;; Ausführen der Prozedur
  RET ;; Rücksprung (return)
```

Parameterübergabe bei Subroutinen

- verschiedene Konventionen, z.B. über Register, Stack oder Zeiger auf Register oder Stack
- Base-Pointer (BP) zeigt Stack-Frame
- Stack-Pointer (SP) wächst Richtung niedrigere Adressen
- Beispiel Prozedur mit zwei Parametern (p1, p2) und zwei lokalen Variablen (v1, v2)
  - Adressierung relativ zum Base Pointer



Linux (Dialekt “nasm”)

- Deklaration von Variablen unterschiedlichen Typs
- Zugriff auf benannte Adressen über Variablennamen

```
;; constants
section .const

;; global variables
section .data
aByte:   DB 'a'
aWord:   DW 0xffff
aDWord:  DD 0xffffffff
aConst:  EQU 0xac0ffee
anArray: DB 'hello', 0
result:  RESW 1
```

```
;; program code
section .text
global _start
_start:
 MOV eax, [aByte]
 ADD eax, [aWord]
 ADD eax, aConst
 SUB eax, [aDWord]
 MOV [result], eax
```

Windows (Dialekt “masm32”)

- Beispiel für Prozedur zur Bestimmung von Fibonacci Zahlen
- Zahlen werden im Voraus in ein Feld geschrieben

```
/* äquivalenter C Code */
const int fib_vals[] = {0, 1, 1, 2, 3, 5, 8, 13};
const int n_fib_vals = 8;

int fibonacci(int i, int* result) {
  int ret = 0;
  if (i < 0 or i >= n_fib_vals)
    goto invalid_index;
  *result = fib_vals[i];
  ret = 1;
invalid_index:
  return ret;
}
```

Fibonacci Prozedur: Header (1/3)

```
  .model flat,c ;; Einstellungen
  .const        ;; Globale Variablen
fib_vals
  DWORD 0, 1, 1, 2, 3, 5, 8, 13
  DWORD 21, 34, 55, 89, 144, 233, 377, 610
n_fib_vals
  DWORD ($ - fib_vals) / sizeof word ; length
  .code
fibonacci_ proc
  PUSH ebp
  MOV ebp, esp ;; Stack pointer Sichern
  ;; Register sichern
  PUSH ebx
  PUSH esi
  PUSH edi
  ;; Rückgabewert initialisieren
  XOR eax, eax
```

Fibonacci Prozedur: Indexprüfung & Feldzugriff(2/3)

```
;; Indexprüfung
MOV ecx, [ebp+8]        ;; ecx <- i
CMP ecx, 0
JL invalid_index        ;; Sprung falls i < 0
CMP ecx, [n_fib_vals]
JGE invalid_index       ;; Sprung falls i >= n_fib_vals

;; *result = fib_vals[i];
MOV ebx, offset fib_vals ;; ebx <- &fib_vals
MOV esi, [ebp+8]         ;; esi <- i
MOV eax, [ebx+esi*4]     ;; eax <- fib_vals[i]
MOV edi, [ebp+12]        ;; edi <- result
MOV [edi], eax           ;; *result <- edi
```

Fibonacci Prozedur: Ende (3/3)

```
  MOV eax, 1 ;; Setze Rückgabewert: erfolgreich

invalid_index:
  ;; Register wieder herstellen
  POP edi
  POP esi
  POP ebx
  POP ebp
  RET

fibonacci_ endp
  END
```

Einbettung von Assembler Code in C-Code (gcc)

- `DIV ebx` ergibt

  Quotient $(d:a)/b \rightarrow a$ (EDX:EAX / EBX → EAX)

  Rest $(d:a)\%b \rightarrow d$ (EDX:EAX % EBX → EDX)

```c
int main() {
  int x = 14; // -> a
  int y = 3;  // -> any register
  int quot;   // <- a
  int rem;    // <- d
  asm("DIV %3"
      // outputs
      : "=a" (quot), // eax -> quot (%0)
        "=d" (rem)   // edx -> rem (%1)
      // inputs
      : "a" (x),  // x -> eax (%2)
        "r" (y),  // y -> any register (%3)
        "d" (0)); // 0 -> edx (%4)
  printf("%d/%d = %d rest %d\n", x, y, quot, rem);
  return 0;
}
```

```
14/3 = 4 rest 2
```

Literatur

- Daniel Kusswurm, “Modern x86 Assembly Language Programming”, Apress, 2014 (verfügbar als E-Book)
- Randall Hyde, “The art of assembly language”, No Starch Press, 2003
- Karen Miller, “An assembly language introduction to computer architecture using the Intel Pentium”, Oxford, 1999
- Intel® 64 and IA-32 Architecture Software Developer’s Manual, Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3D, April 2022 (verfügbar als Download)