

# Die HAM

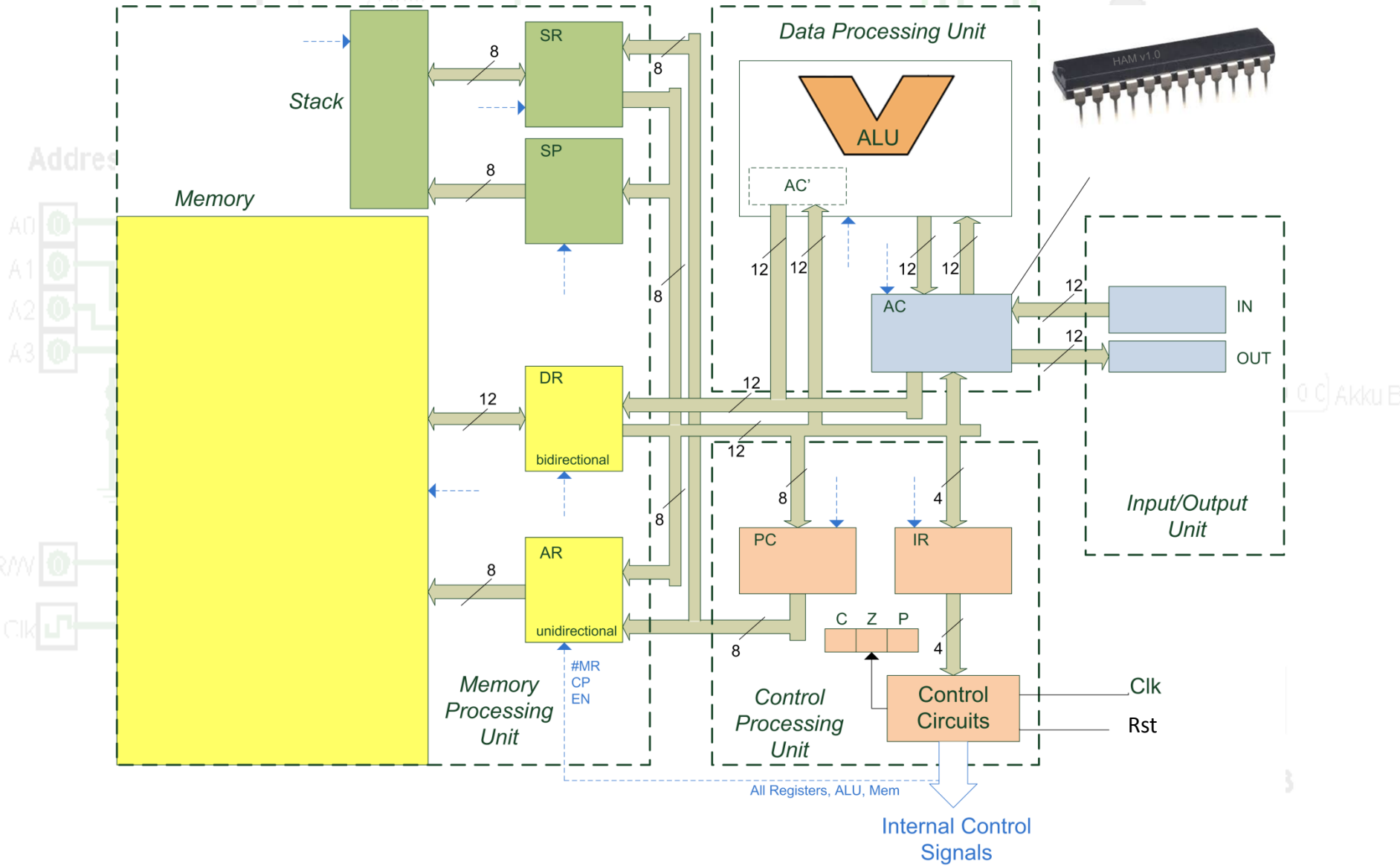
## Die Hypothetische Akku-Maschine

# Inhaltsverzeichnis

- 1 Die Ham
  - 1.1 Überblick
  - 1.2 Hardware
    - Funktion der HAM
  - 1.3 Der Assembler-Befehlssatz
    - Addition zweier Zahlen
  - 1.4 Der HAM-Editor
    - Addition zweier Zahlen
  - 1.5 Der Assemblerlauf
  - 1.6 Der HAM-Simulator
    - Addition zweier Zahlen



# 1.1 Überblick I



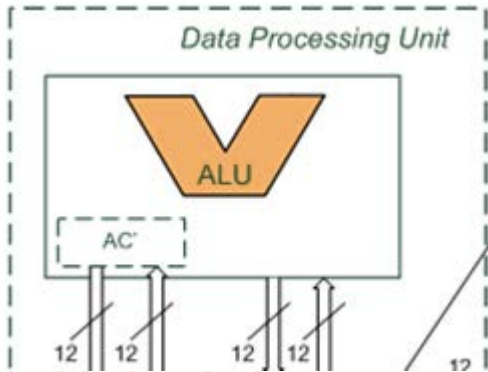
# 1.1 Überblick II

- Von-Neumann-Architektur, Zerlegung des Rechners in logische Blöcke:
  - Rechenwerk (arithmetische und logische Operationen)
  - Speicherwerk (Speicherung von Programm und Daten in einem Speicherbereich)
  - Steuerwerk (zur Steuerung des Programmflaubs)
  - Ein/Ausgabewerk (zur Kommunikation mit der Außenwelt)

# 1.1 Überblick III

- Rechenwerk:
  - einer 12-Bit-ALU inkl. AC
- Speicherwerk:
  - einem 12-Bit-Datenregister DR
  - einem 8-Bit-Adressregister AR
  - einem externen Speicher
- Steuerwerk:
  - einem 8-Bit-Programmzähler PC
  - einem 4-Bit-Instruktionsregister IR
  - einer Kontrolleinheit mit Takt und Reset
  - einem 12-Bit-Daten- und 8-Bit-Adressbus
  - Prozessorstatus mit Carry-, Zero- und Parity-Flag
- Ein/Ausgabewerk:
  - einem 12-Bit-Eingangsregister IN
  - einem 12-Bit-Ausgangsregister OUT

# 1.2 Hardware



- ALU (Arithmetic Logic Unit)
  - Rechenknecht unterschiedlicher Ausrichtung

- Arithmetische Operationen
  - Addition ohne Übertrag
- Logische Operationen
  - And-Verknüpfung, Negierung, Rotate Right
- Operationen werden von Steuerwerk abhängig vom Assemblerbefehl kodiert

→ **03 TGI Zahlensysteme.ppt**

# Flipflops

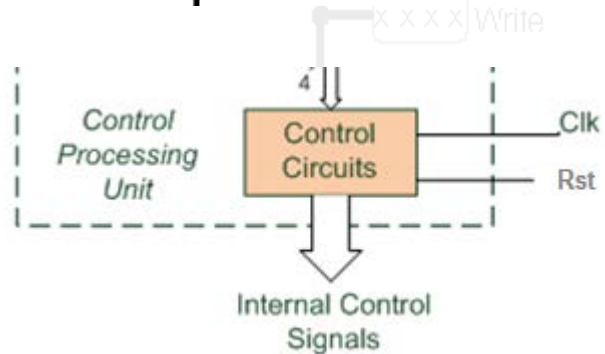
- Register (D-FF) → 04 TGI FF.ppt

- DR: Datenregister
- AR: Adressregister
- PC: Programmzähler
- IR: Instruktionsregister
- AC: Akku-Register

- External Memory → 05 TGI Speicher.ppt

- SRAM, DRAM
- Stack-Speicher (SRAM)

# Kontrolleinheit



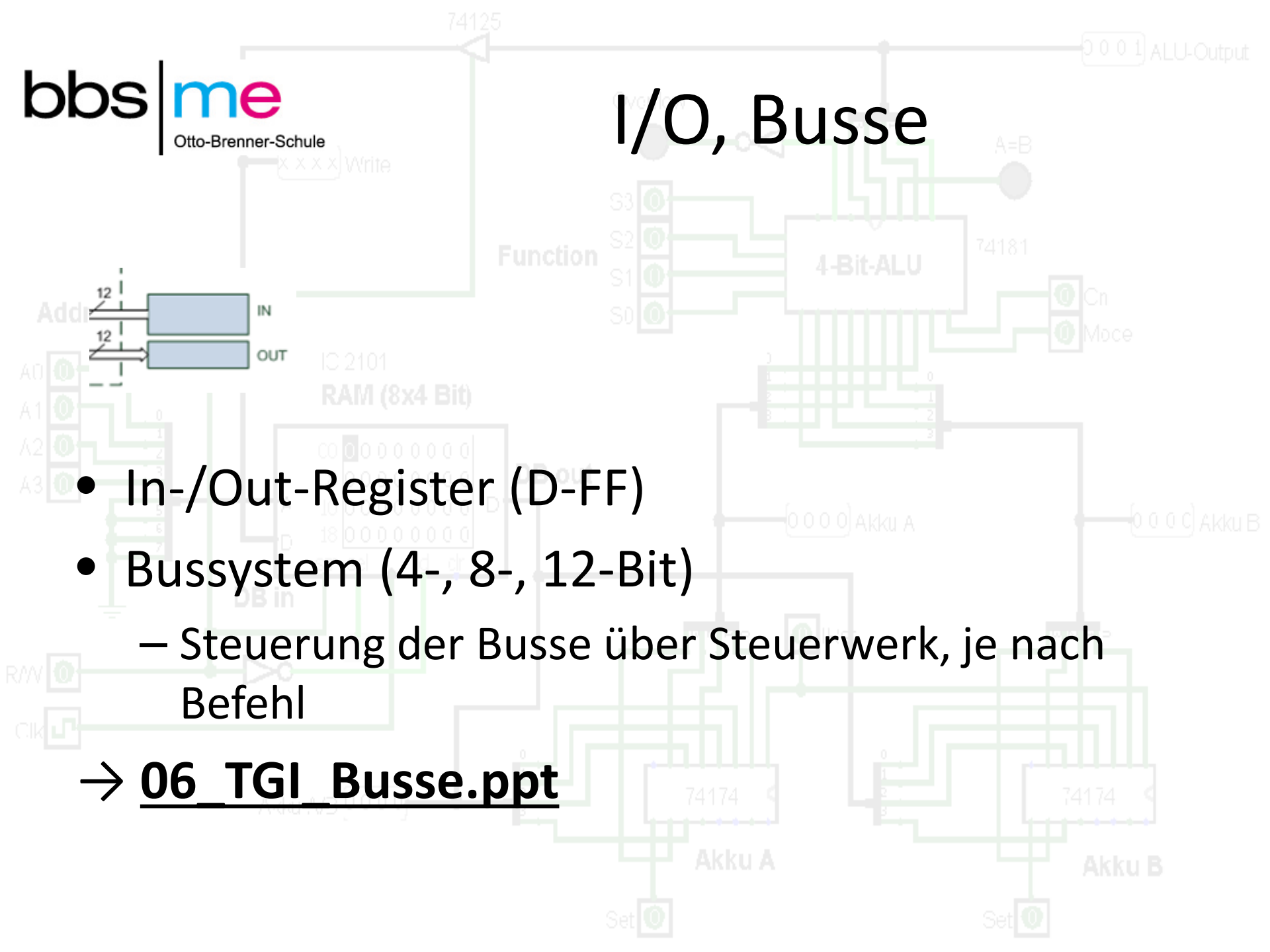
- Wird durch Instruktionsregister (4 Bit) getriggert
- Steuerung durch Micro-Programme
  - Sind unterhalb von Assembler-Programme angesiedelt
  - Implementiert ist das horizontale Micro-Befehlsformat
  - Instruktion schaltet in der Steuereinheit die entsprechenden internen Kontrollsignale
- Ein Befehl (IR, 4 Bit) steuert ein Micro-Programm
- Takt und Reset implementiert



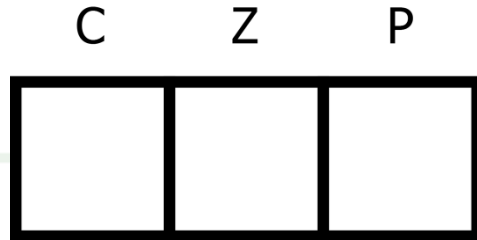
# I/O, Busse

- In-/Out-Register (D-FF)
- Bussystem (4-, 8-, 12-Bit)
  - Steuerung der Busse über Steuerwerk, je nach Befehl

→ [06\\_TGI\\_Busse.ppt](#)



# Prozessor-Flags



C: Carry-Flag

Z: Zero-Flag

P: Parity-Flag

- Prozessor-Flags geben den Status des Akkus an
  - C: 1 wenn Übertrag des Akku, sonst 0
  - Z: 1 wenn Akku = 0, sonst 0
  - P: 1 wenn Anzahl der Binäreinsen gerade, sonst 0
- Prozessor-Flags können via GPS in den Akku gelesen werden. Danach sind sie gelöscht
  - GPS: Get Processor Status

# Funktion der HAM I

```
declare register AC(11:0), DR(11:0), AR(7:0), PC(7:0), IR(3:0)
declare register SP(7:0), SR(7:0)
declare register INREG(11:0), OUTREG(11:0)
```

```
# memory Breite: 000..FFF, Tiefe: 00..FF
```

```
declare memory M(AR,DR)
```

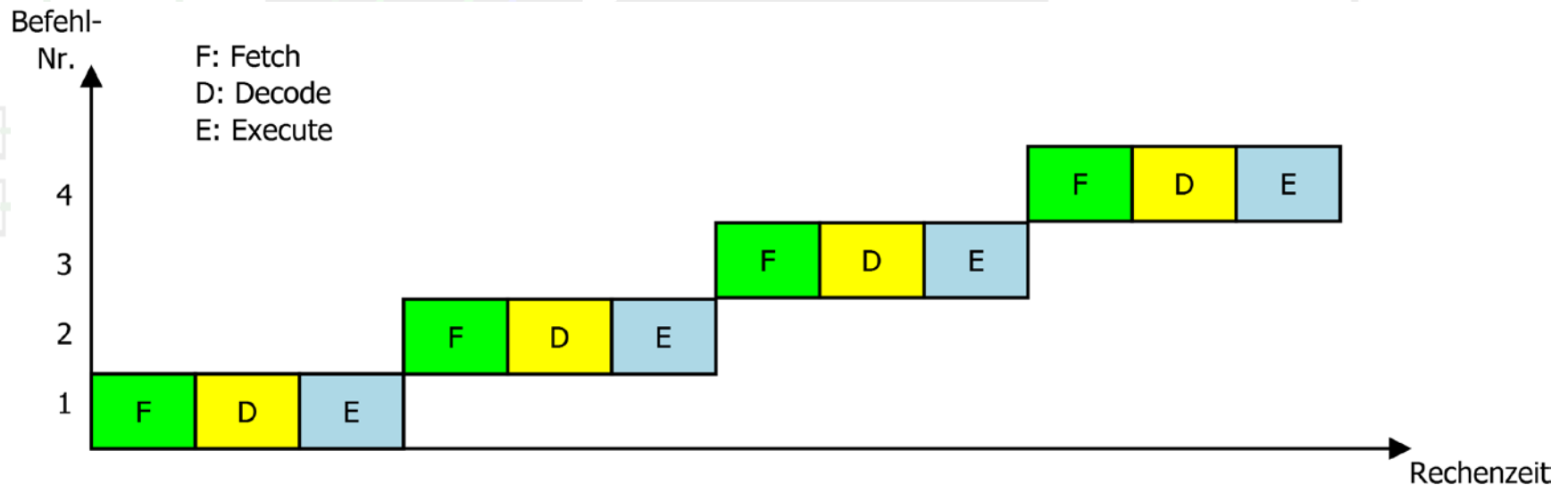
```
# stack Breite: 00..FF, Tiefe 00..FF
```

```
declare memory S(SP,SR)
```

- Deklaration aller Register mit der entsprechenden Bit-Breite (12-Bit, 8-Bit oder 4-Bit)
- Deklaration des Daten- und Programmspeichers
  - Tiefe 2 KBit, Breite 12 Bit: 2k x 12
- Deklaration des Stacks für die Subroutinen
  - Tiefe 2 KBit, Breite 8 Bit: 2k x 8

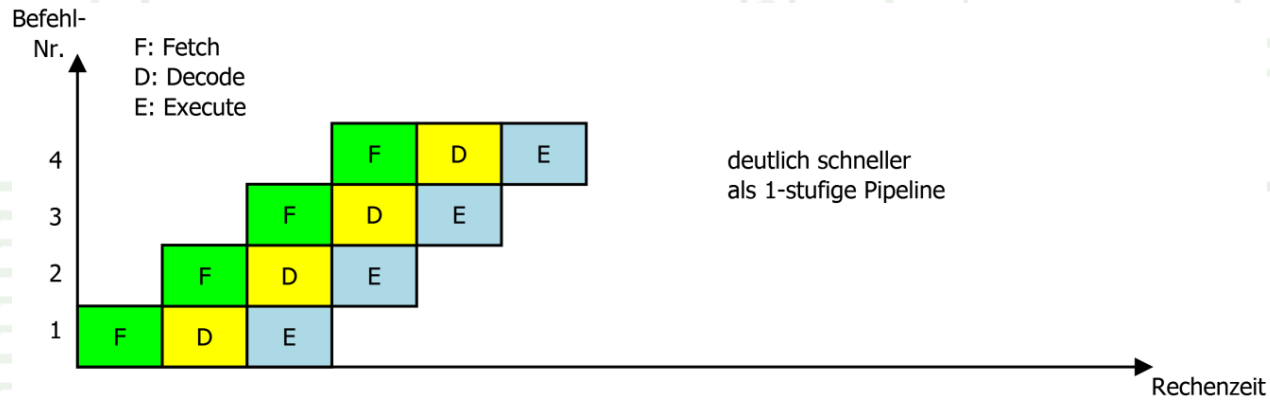
# Funktion der HAM II

- 1-stufige Pipeline nach ‚von Neumann‘ mit:
  - *Fetch*: Befehl aus dem Speicher holen
  - *Decode*: Befehl dekodieren
  - *Execute*: Befehl ausführen
- 1-stufige Pipelines laufen sequentiell ab

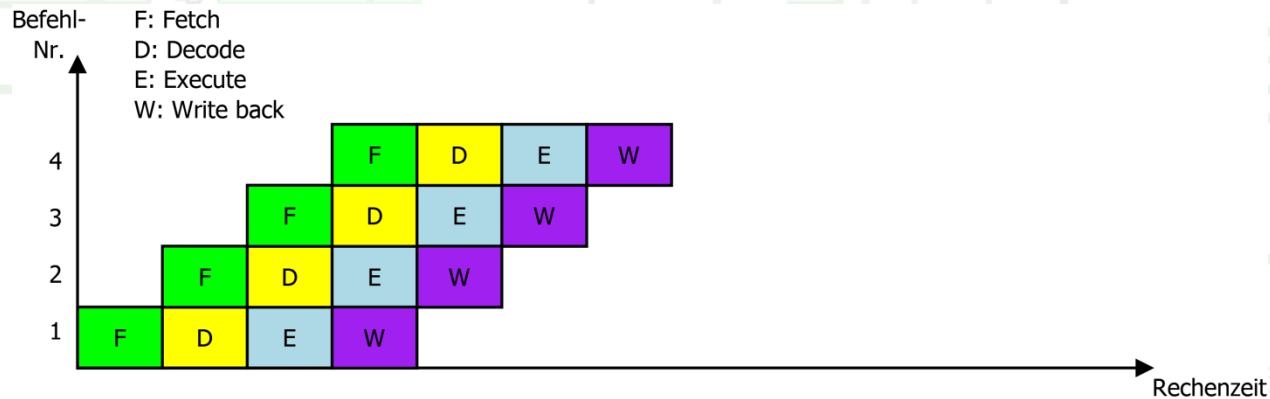


# Funktion der HAM III

- Mehrstufige Pipelines
  - 3-stufig: Fetch, Decode, Execute



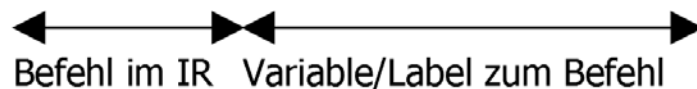
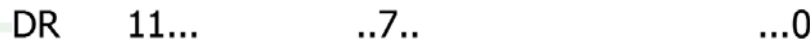
- 4-stufig: Fetch, Decode, Execute, Write back



# Fetch

```
# fetch  
FETCH:  AR<-PC;  
        read M;  
        IR<-DR(11:8), PC<-PC+1 |
```

- Programmzähler PC bestimmt die Adresse AR
- Speicheradresse lesen: ASM-Befehl + Datum, insgesamt 12 Bit
- Instruktionsregister IR dekodiert den Befehl
  - 4 Bit (MSN) → max. 16 Befehle
- Programmzähler auf nächsten Befehl
- Befehlsformat im Speicher und im DR:



# Decode

```
# decode
```

```
if IR=1 then goto LDA else  
if IR=2 then goto STA else  
if IR=3 then goto ADD else  
if IR=4 then goto AND else  
if IR=5 then goto JMP else  
if IR=6 then goto JPZ else  
if IR=7 then goto NEG else  
if IR=8 then goto RAR else  
if IR=9 then goto IN else  
if IR=10 then goto OUT else  
if IR=11 then goto LDI else  
if IR=12 then goto STI else  
if IR=13 then goto JSR else  
if IR=14 then goto RTS else  
if IR=15 then goto GPS else  
goto FETCH fi fi fi fi fi fi fi fi fi fi fi fi fi fi fi fi;
```

- Sprung in das Label abhängig von dem Inhalt des Instruktionsregisters IR
  - IR = 6, Sprung nach JPZ (Jump Zero)

# Execute

```
JPZ:    if AC<>0 then goto FETCH fi; PC<-DR(7:0) | goto FETCH;
```

- Beispiel JPZ:

- Wenn Akku != 0, dann nächster Befehl,  
sonst Programmzähler auf Label in DR(7:0)

```
LDA:    AR<-DR(7:0);  
        read M;  
        AC<-DR | goto FETCH;
```

- Beispiel LDA:

- Variable DR(7:0) ist Adresse
- Lese Speicher
- Gelesenes Datum in Akku, nächster Befehl



# 1.3 Der Assemblerbefehlssatz

Befehl	Op-Code	Takte	Beschreibung
nop	0	3	no operation
lda x	1	6	$ac \leftarrow mem(x)$
sta x	2	6	$mem(x) \leftarrow ac$
add x	3	6	$ac \leftarrow ac + mem(x)$
and x	4	6	$ac \leftarrow ac \& mem(x)$
jmp x	5	4	$pc \leftarrow x$
jpz x	6	5	if $ac = 0$ then $pc \leftarrow x$
neg	7	4	$ac \leftarrow \text{not } ac$
rar	8	4	rotate ac right
in	9	4	$ac \leftarrow in$
out	a	4	$out \leftarrow ac$
ldi x	b	8	$ac \leftarrow mem(mem(x))$
sti x	c	8	$mem(mem(x)) \leftarrow ac$
jsr x	d	6	$pc \leftarrow x$
rts	e	6	$x \leftarrow pc$
gps	f	4	$ac \leftarrow fr$



# Beispiel: Addition zweier Zahlen per Hand

Adr (hex)	M(Adr) (hex)Opc.	M(Adr) (hex)Ope.	Label	Opcode	Operand	Comments	Takte
				.ORG	\$00	Start \$00	
00	1	10	ADD:	LDA	s1	AC <- s1	6
01	3	11		ADD	s2	AC <- AC + s2 (s1+s2)	6
02	2	12		STA	sum	sum <- AC	6
03							
04							
05							
06							
07							
08							
09							
0A							
0B							
0C							
0D							
0E							
0F							
10	0	32	s1			Zahl s1	
11	0	21	s2			Zahl s2	
12	0	00	sum			Summe	

- Programm:  
LDA s1  
ADD s2  
STA sum

- Daten:  
s1 32  
s2 21  
sum 00

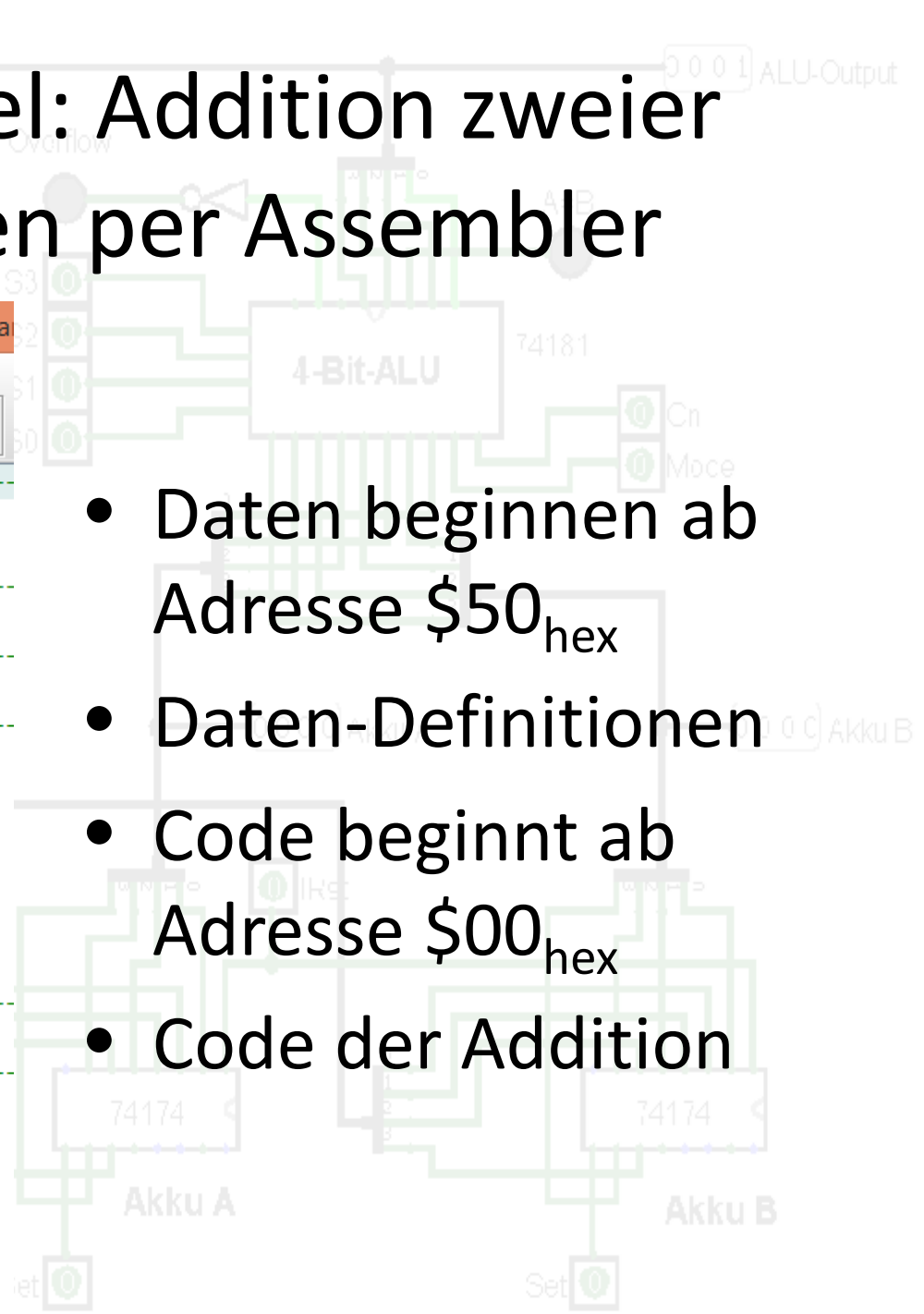
- Op-Code:  
110  
311  
212

- Takte: 18

# Beispiel: Addition zweier Zahlen per Assembler

```
C:\Kaila\BBS ME\bbs\Mikroprozessortechnik\004_Mikroprogra
File Edit Run Help
[Icons]
1 #-----
2 # ADD example
3 # (c) kai.dorau@gmx.net
4 #-----
5
6 #-----
7 # Daten
8 #-----
9 .data $50;
10
11 .def s1 $032;
12 .def s2 $021;
13 .def sum $000;
14
15
16 #-----
17 # Programm
18 #-----
19 .code $00;
20
21 main:   lda   s1;
22         add  s2;
23         sta  sum;
24
```

- Daten beginnen ab Adresse  $\$50_{hex}$
- Daten-Definitionen
- Code beginnt ab Adresse  $\$00_{hex}$
- Code der Addition



# 1.4 Der HAM-Editor I

- 1: Neue Assemblerdatei
- 2: Assemblerdatei öffnen
- 3: Assemblerdatei speichern
- 4: Zeilen ausschneiden
- 5: Zeilen kopieren
- 6: Zeilen einfügen
- 7: Schrittweise Undo
- 8: Schrittweise Redo
- 9: Keyword suchen
- 10: Code assemblieren
- 11: Programm simulieren
- 12: Editorkonfiguration
- 13: Tutorial aufrufen
- 14: Keyword eingeben
- 15: Editorfenster
- 16: Statusfenster

```

1  #
2  # ADD example
3  # (c) kai.dorau@gmx.net
4  # -----
5
6  # -----
7  # Daten
8  # -----
9  .data $50;
10
11 .def s1  $032;
12 .def s2  $021;
13 .def sum $000;
14
15
16 # -----
17 # Programm
18 # -----
19 .code $00;
20
21 main:  lda  s1;
22        add  s2;
23        sta  sum;

```

29.01.17 22:17 Simulating...  
29.01.17 22:53 Simulating...  
29.01.17 22:56 Compiling and linking...  
hasm V1.3  
Copyright (C) 2014 kai.dorau@gmx.net  
This is free software; see the source for copying conditions. There is NO

# 1.4 Der HAM-Editor II

```
16 #-----  
17 # Programm  
18 #-----  
19 .code $00;  
20  
21 main:   lda    s1;  
22         add   s2;  
23         sta   sum;
```

```
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  
ASM: Instance of ASM has been created...  
ASM: Checking syntax of C:\Kaila\BBS ME\bbs\Mikroprozessortechnik\HamToolkit\add.asm.....done  
ASM: Building Hamsim memory file C:\Kaila\BBS ME\bbs\Mikroprozessortechnik\HamToolkit\add.mem.....done  
ASM: Instance of ASM has been successfully removed...
```

- Assembler-Code OK

# 1.4 Der HAM-Editor III

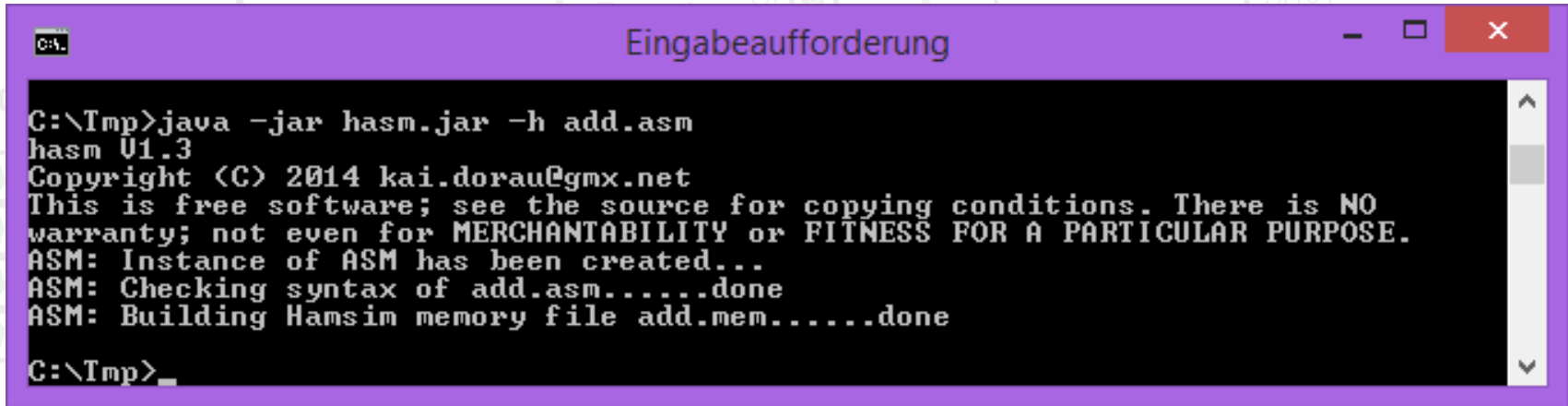
```
16 #-----  
17 # Programm  
18 #-----  
19 .code $00;  
20  
21 main:   lda   s1  
22       |     add  s2;  
23       |     sta  sum;
```

ASM: Checking syntax of C:\Kaila\BBS ME\bbs\Mikroprozessortechnik\HamToolkit\add.asm...ASM: Checking syntax failed!  
ASM: Instance of ASM has been successfully removed...

ERROR: Syntax error in line 21 semicolon?

- Assembler-Code-Fehler in Zeile 21
- Semikolon fehlt
  - main: lda s1;

# 1.5 Der Assemblerlauf

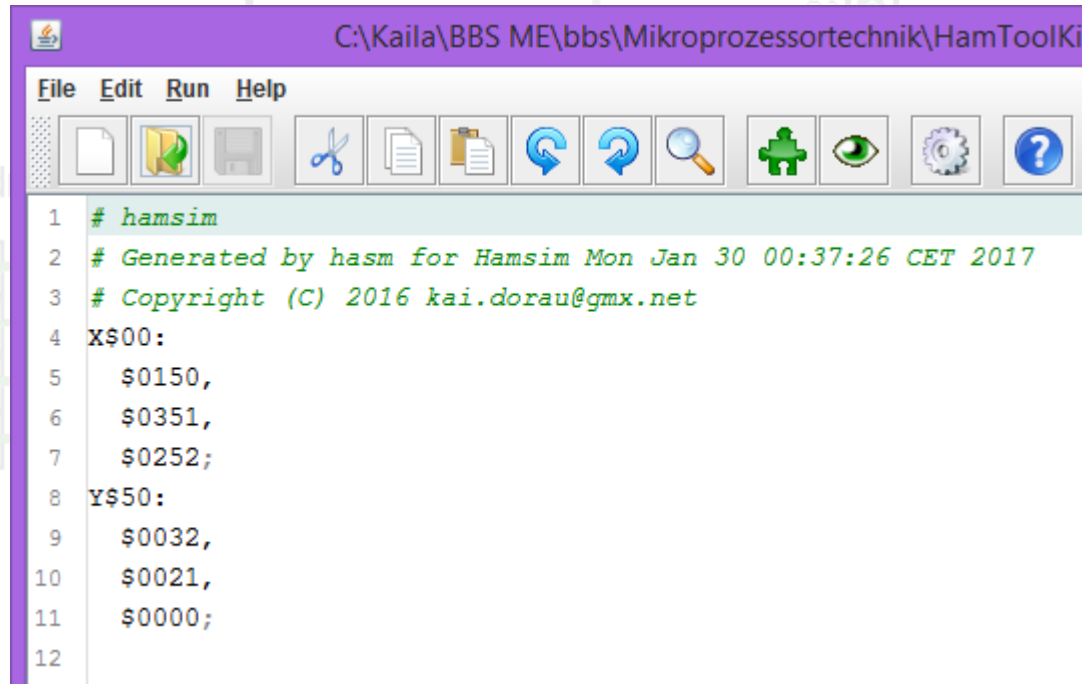


```
C:\Imp>java -jar hasm.jar -h add.asm
hasm V1.3
Copyright (C) 2014 kai.dorau@gmx.net
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
ASM: Instance of ASM has been created...
ASM: Checking syntax of add.asm.....done
ASM: Building Hamsim memory file add.mem.....done

C:\Imp>
```

- Assembler: hasm (HAM-Assembler)
- add.asm übersetzt, Maschinencode in add.mem
- Kein Syntax-Fehler aufgetreten, alles ok!

# Op-Code nach dem Assemblieren

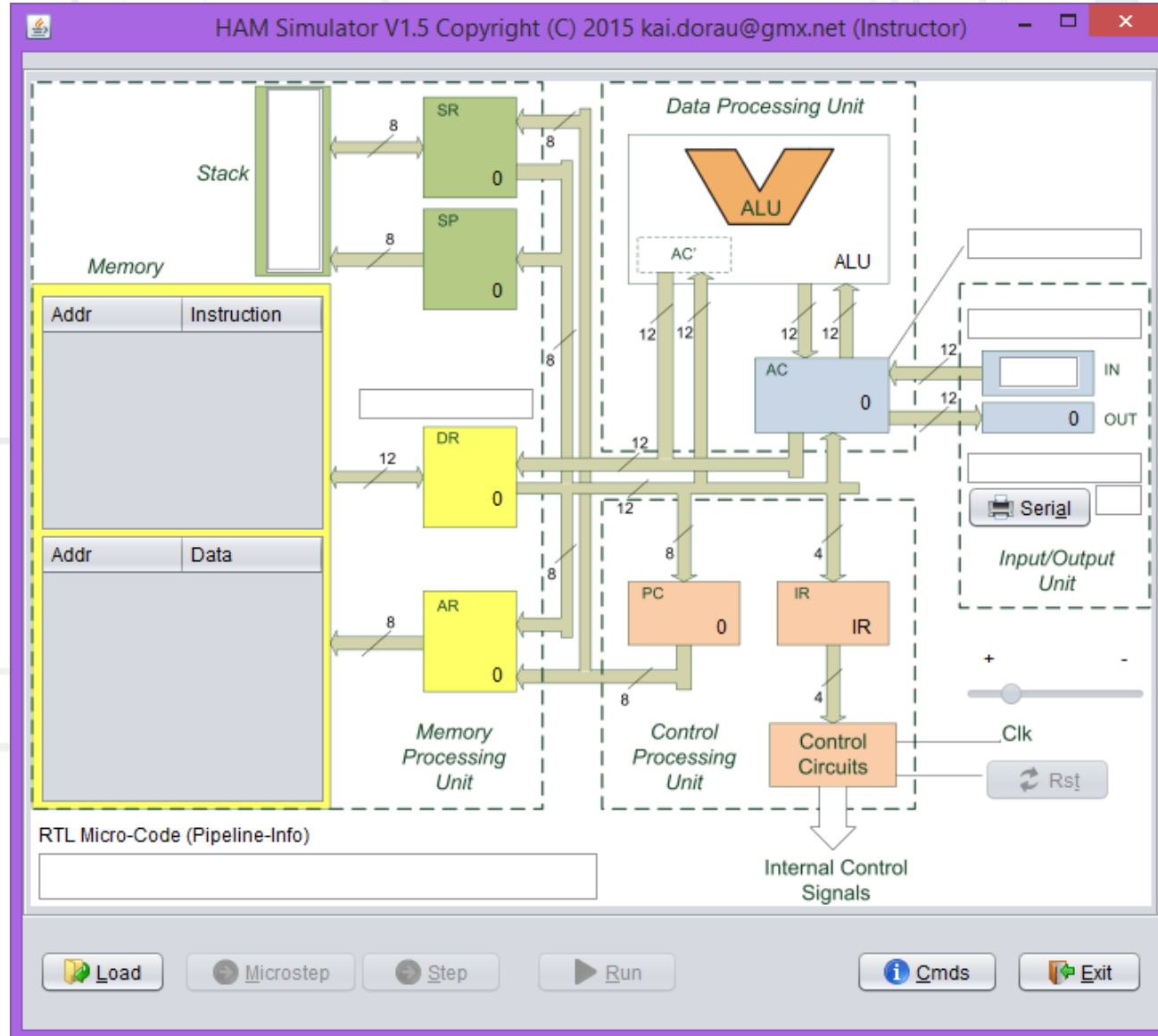


```
C:\Kaila\BBS ME\bbs\Mikroprozessortechnik\HamToolKit
File Edit Run Help
1 # hamsim
2 # Generated by hasm for Hamsim Mon Jan 30 00:37:26 CET 2017
3 # Copyright (C) 2016 kai.dorau@gmx.net
4 X$00:
5   $0150,
6   $0351,
7   $0252;
8 Y$50:
9   $0032,
10  $0021,
11  $0000;
12
```

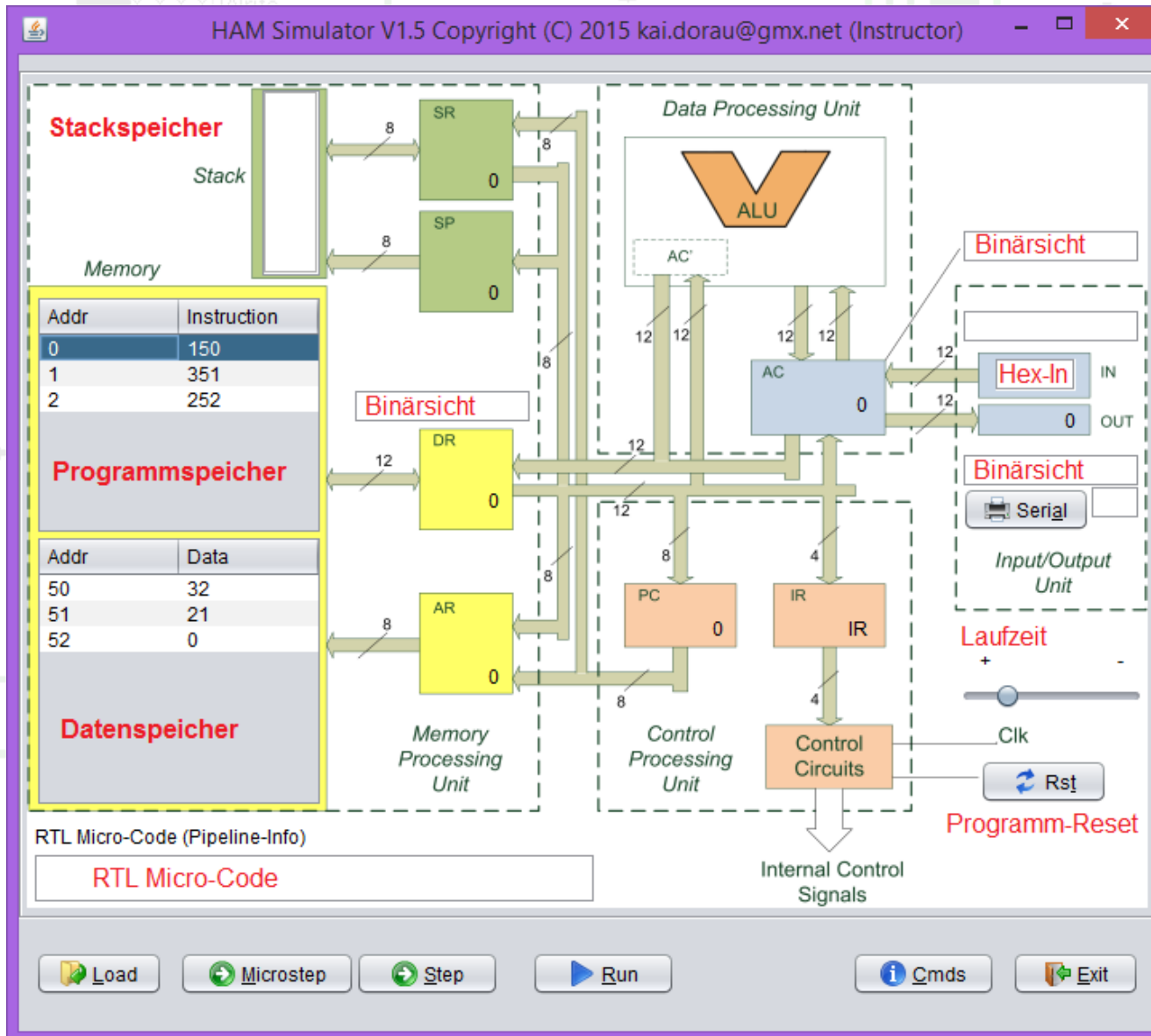
- Code: 0x00, Daten: 0x50
  - 0x150 → 1: LDA, Adresse 0x50 (Wert: 0x32)
  - 0x351 → 3: ADD, Adresse 0x51 (Wert 0x21)
  - 0x252 → 2: STA, Adresse 0x52 (Wert 0x00)



# 1.6 Der HAM-Simulator



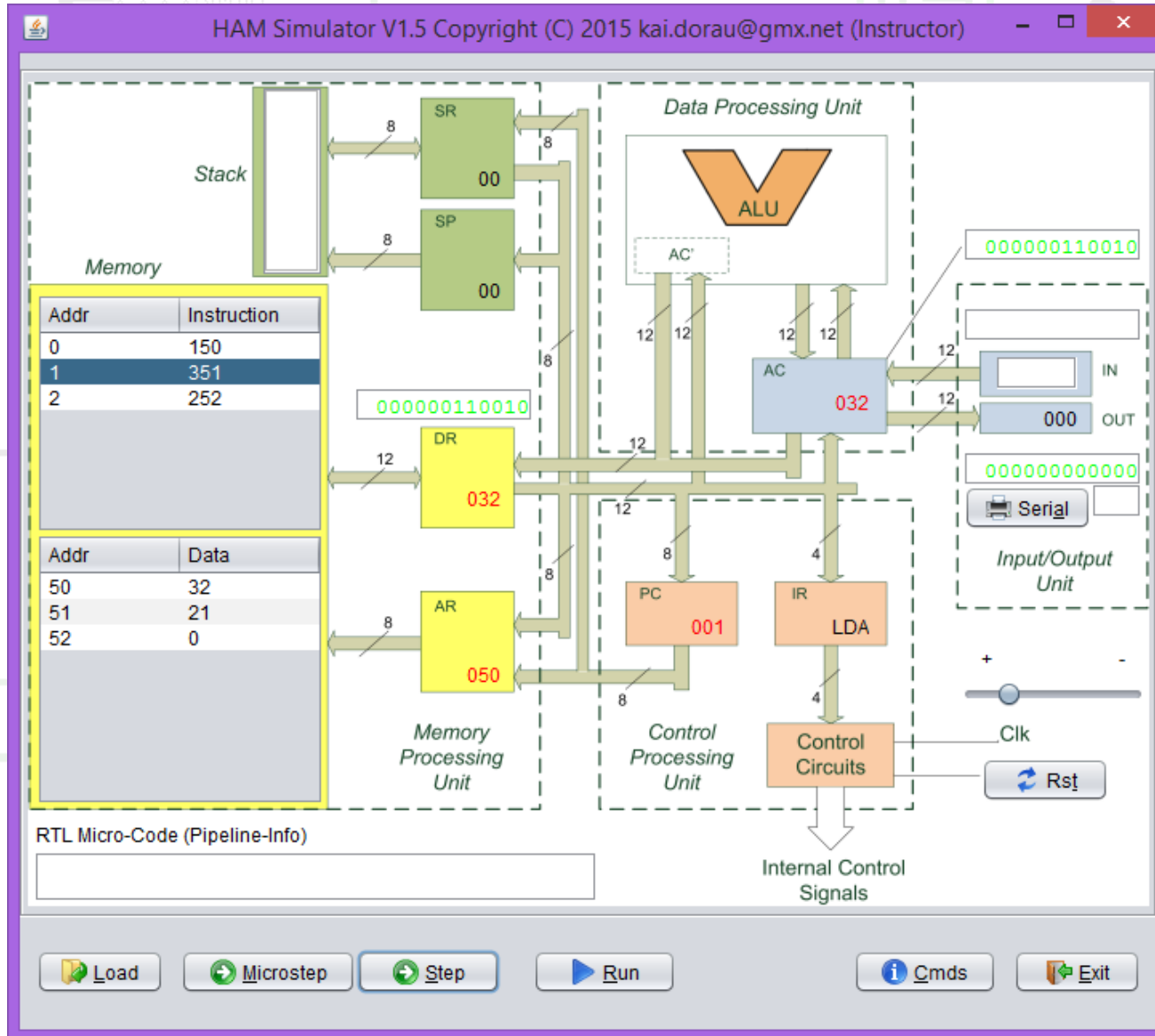
# add.mem geladen I



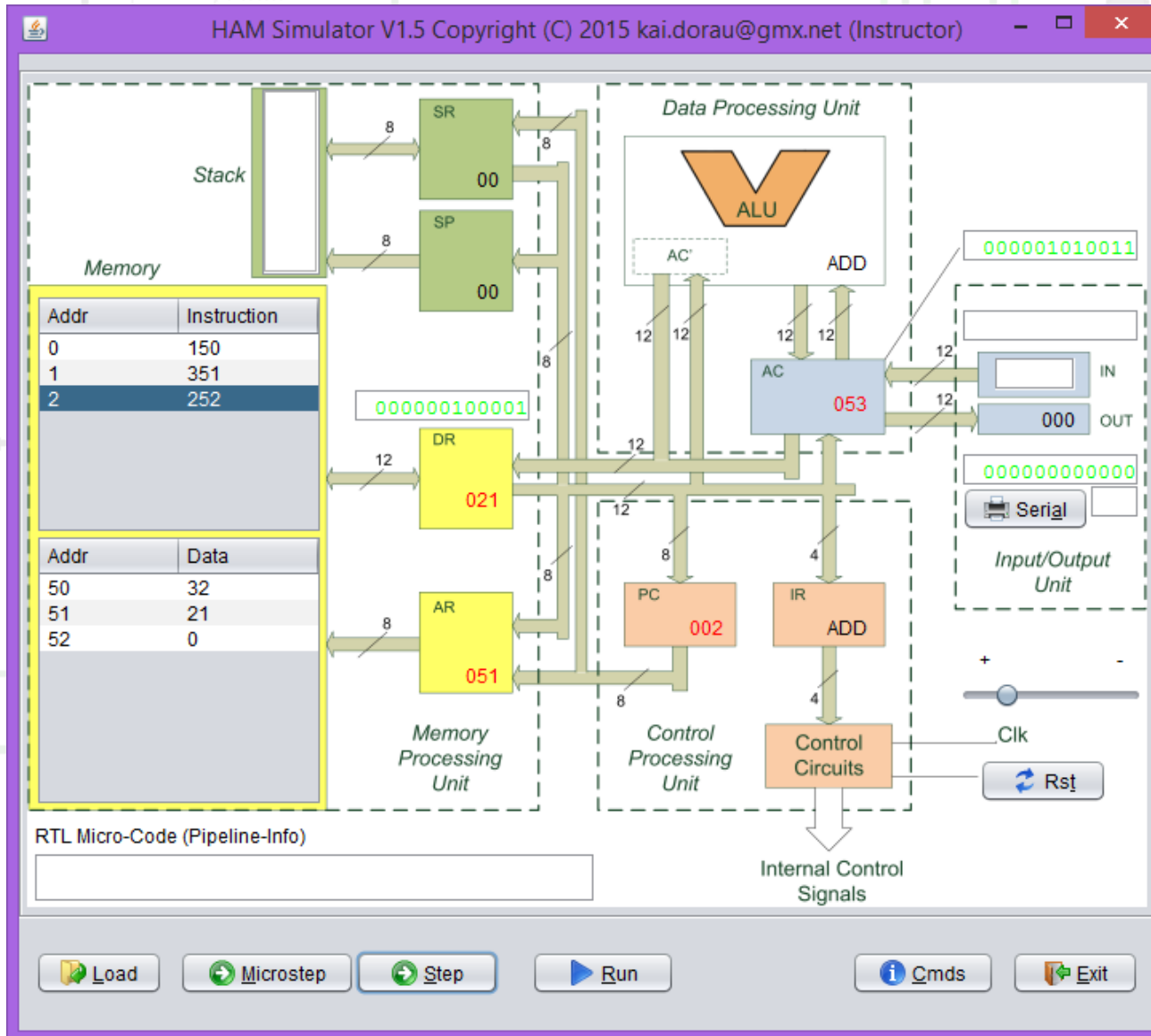
# add.mem geladen II

- Binäransicht ist 12-Bit breit
- Hex-Eingabe ist 3 Nibble (12-Bit)
- Serial ist der Strom auf den Output-Bus (8-Bit)
- Laufzeit ist die Simulationsgeschwindigkeit
- Programm-Reset für Neustart des Programms
- Microstep: Abarbeiten der einzelnen Micro-befehle der jeweiligen Assemblerbefehle
- RTL-Microcode: Anzeige der einzelnen Microsteps

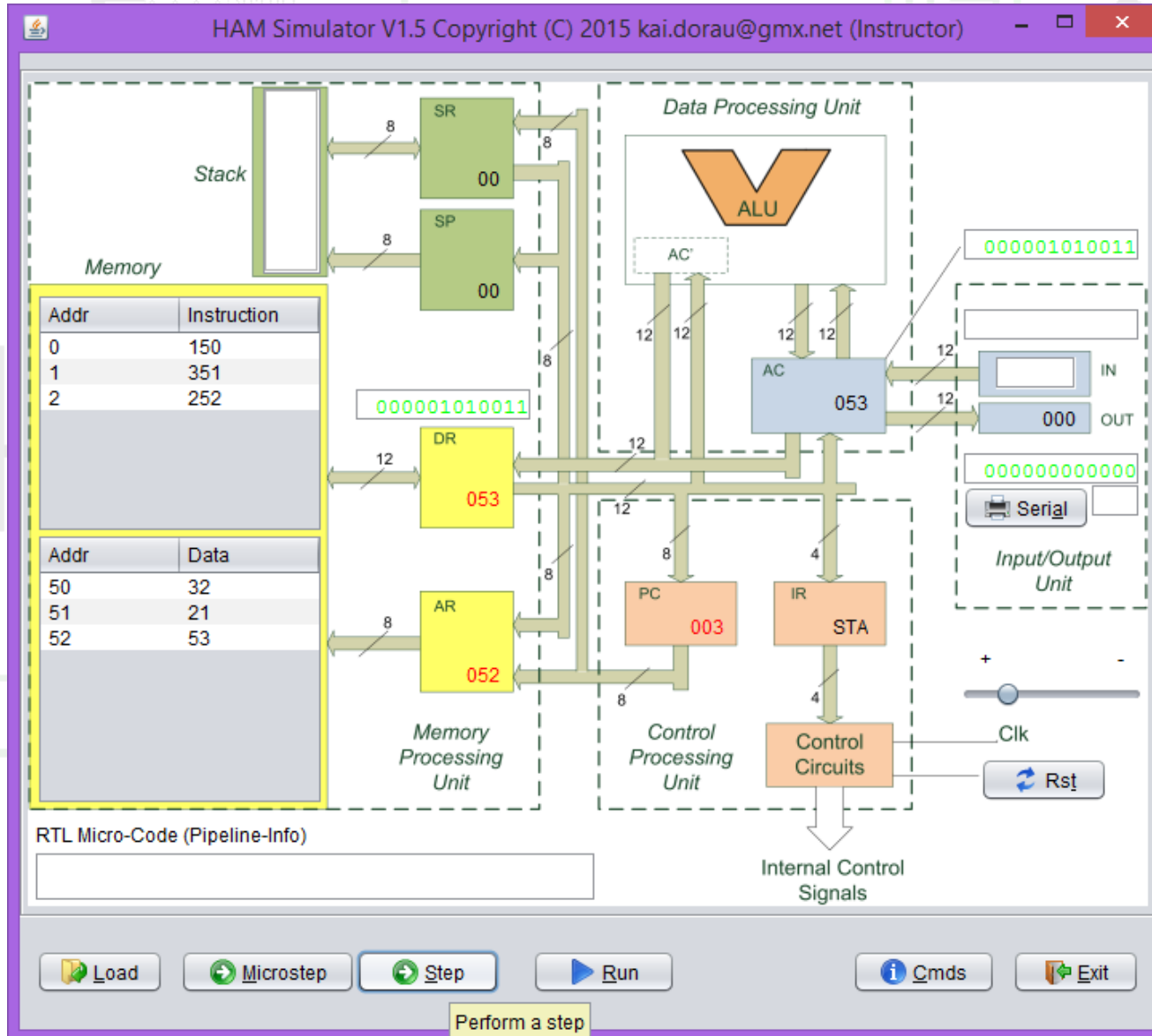
# Beispiel: LDA $s_1=B$



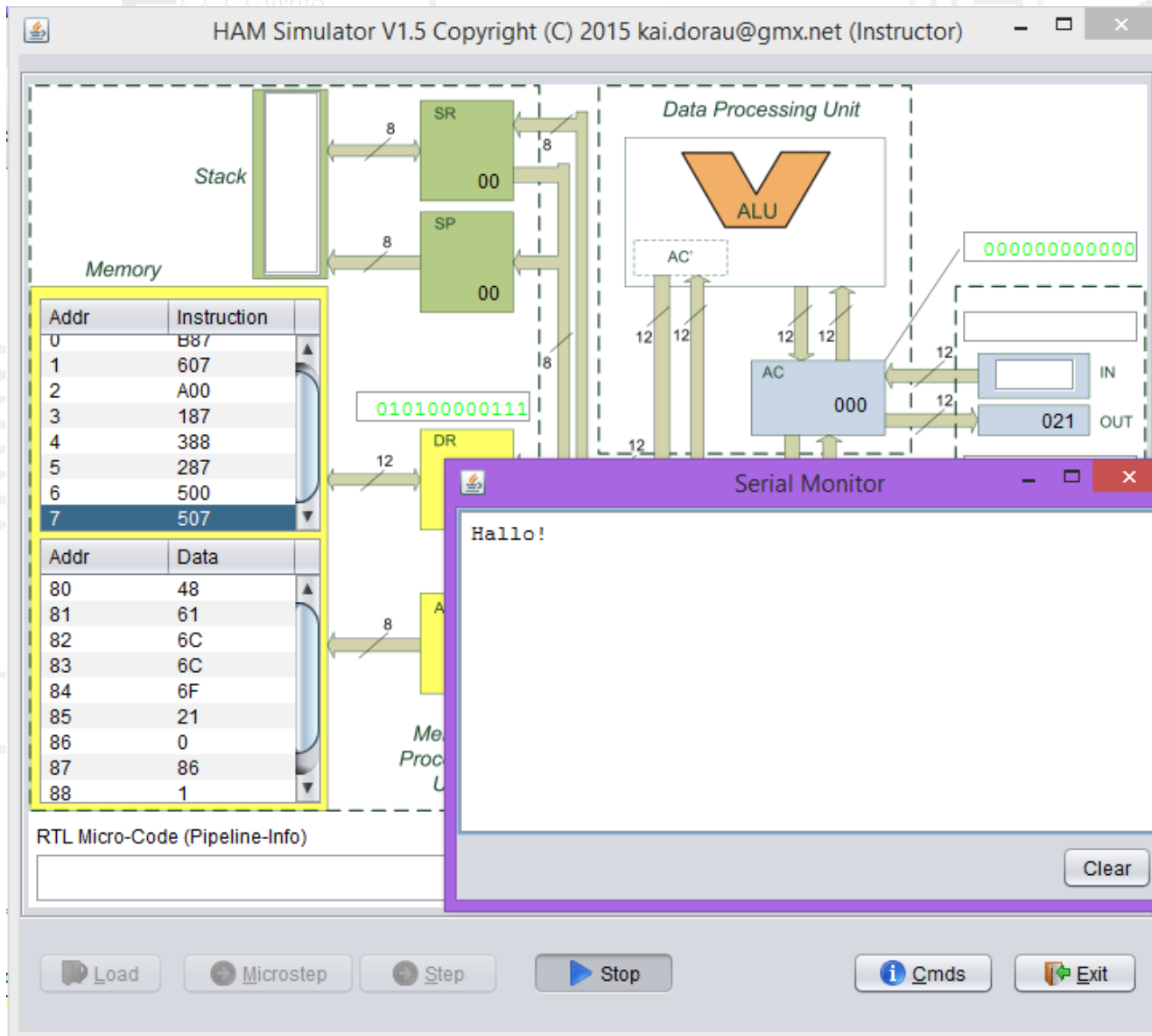
# Beispiel: ADD s<sub>2</sub>



# Beispiel: STA sum



# Bsp.: Hello!-Example



# Architekturen I

- Vorteile **Von-Neumann-Architektur**:
  - Optimale Nutzung de Speichers
  - Einfache Auslegung des Daten- und Adressbus
  - Daten können Programmcode verändern oder erweitern: *Selbstmodifizierende Programme*
- Nachteile:
  - Flaschenhals: Daten-/Adressbus, damit langsame Zugriffe auf Speicherelemente
  - Versehentliches Überschreiben des Programm-codes möglich: *Buffer Overflow/Underflow*



# Architekturen II

- Vorteile Harvard-Architektur:

- Getrennte Daten- und Programmspeicher mit getrennten Bussen sorgen für **schnelle Speicherzugriffe**
- Grundlage für sichere Betriebssysteme: Parallele Prozesse können sich nicht gegenseitig überschreiben

- Nachteile:

- Speicheraufteilung nicht optimal

# RISC (HAM)

- **Reduced Instruction Set Computing:**
  - Minimaler Befehlssatz
  - Sehr effizient auf spezieller Hardware lauffähig
  - Sehr schnelle Abarbeitung bei einfachen Lösungen möglich
  - Entwickler behalten den Überblick, wenn in Assembler programmiert wird
  - Komplexe Lösungen erfordern vielen Befehlen, die nacheinander abgearbeitet werden müssen: dadurch langsam

# CISC

- **Complex Instruction Set Computing**

- Vollständiger Befehlssatz für viele Lösungen
- Bei komplexen Lösungen kürzere Befehlsfolgen, dadurch schnellere Programme
- Befehle sind aufgrund größerer Mikroprogramme etwas länger als RISC-Befehle
- Entwickler benötigen viel Zeit bei der Entwicklung von Programmen in Assembler
- Programme meist übersichtlicher als Programme auf RISC-Maschinen

# RISC/CISC

- In der Realität sind häufig Mischformen beider Arten anzutreffen
  - Die jeweiligen Vorteile bieten optimale Prozessorausnutzung
  - Intel x86 und AMD-Prozessoren: **CISC**
  - PowerPC, Macintosh/SPARC-Prozessoren: **RISC**, obwohl viele Befehlserweiterungen
  - Mikrocontroller AVR/PIC: **RISC**