ULM

(Ulm Lecture Machine)

Michael C. Lehn

Wie funktioniert ein Computer?

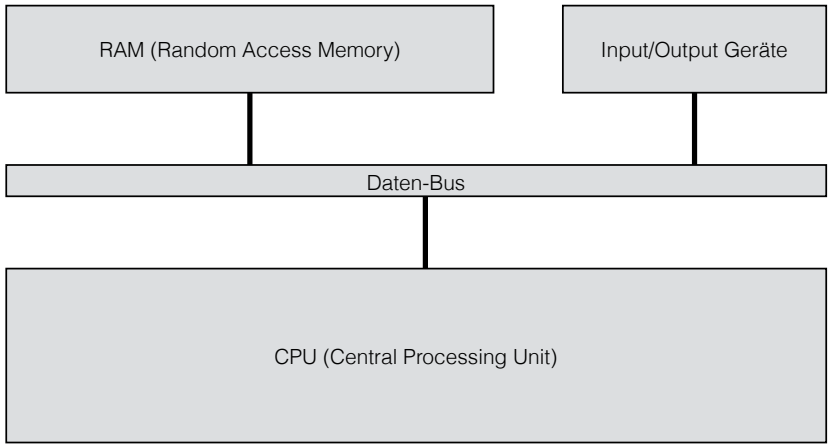
Traditionell das erste Programm in C …

ULM: Von-Neumann-Architektur

Unser erstes Computer Programm:

```
10101100000000000101000000000010001101100000000
00000001000000100110000100000010000000000000000
00011100000000001000000010000000100011011000000
00000000100000010011000010000001000000000000000
00000111000000000010000000100000001000110110000
00000000000100000010011000010000001000000000000
00000000000010000000000000000000000000100100001
10100100100001000000000
```

ULM: Von-Neumann-Architektur



Für uns relevante Zahlendarstellungen (und deren Zahlenwerte)

Binär-Darstellung	Hexadezimal-Darstellung	Zahlenwert (in Dezimal-Darstellung)	
		Vorzeichenlos (unsigned)	Vorzeichenbehaftet (signed)
$(0000)_2$	0x0	0	0
$(0001)_2$	0x1	1	1
$(0010)_2$	0x2	2	2
$(0011)_2$	0x3	3	3
$(0100)_2$	0x4	4	4
$(0101)_2$	0x5	5	5
$(0110)_2$	0x6	6	6
$(0111)_2$	0x7	7	7
$(1000)_2$	0x8	8	-8
$(1001)_2$	0x9	9	-7
$(1010)_2$	0xA	10	-6
$(1011)_2$	0xB	11	-5
$(1100)_2$	0xC	12	-4
$(1101)_2$	0xD	13	-3
$(1110)_2$	0xE	14	-2
$(1111)_2$	0xF	15	-1

Heute relevant ist nur die vorzeichenlosen Zahlendarst.

Bit-Muster mit Hex-Ziffern kompakt darstellen

Übung

Bit-Muster mit 64 Stellen:

0101	0110	1111	0000	0010	0000	0000	0001	0011	1000	1110	0001	1010	0010	0000	1100

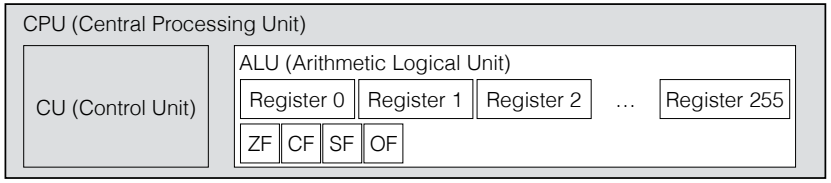
Darstellung mit 16 Hex-Ziffern:

ULM: Von-Neumann-Architektur

Angewandt auf unser erstes Programm:

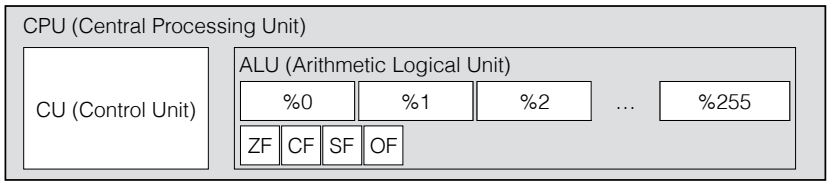
```
560028011B0001026102000038010101 1B0001026102000
0380101011B0001026102000001000000 48692100
```

ULM: Arithmetic Logical Unit (Teil der CPU)



- 256 Register. Jedes Register kann 64 bits speichern.
- Mit Registern kann man rechnen (Addition, Subtraktion, …).
- Nach einer Rechenoperation werden gewisse Statusflags gesetzt.
  - Zum Beispiel: ZF (Zero Flag) wird gesetzt, wenn beim Ergebnis alle Bits Null sind.

ULM: Arithmetic Logical Unit (Teil der CPU)

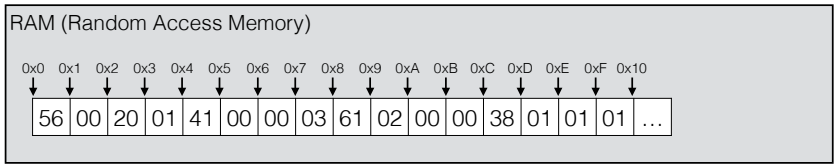


Beispiel:

%3 ← %2 - %1

- %X bezeichnet den Inhalt von Register X
- Hier: Subtrahiere %1 von %2 schreibe Ergebnis in %3.
- ZF (Zero flag) wird genau dann gesetzt, wenn %1 und %2 den gleichen Inhalt haben.

ULM: Random Access Memory (RAM)



- Der RAM der ULM ist 2^{64} Bytes (1 Byte = 8 Bits) groß.
- Also circa. 18,45 Trillionen Bytes oder 17.179.869.184 GiB
- Jedes Byte hat eine 64-Bit-Adresse.
- Notation: $M_n(X)$ = "n Bytes ab Adresse X"
- Übung:

M_1(0xF)	0x01
M_4(0xC)	0x38 01 01 01
M_8(0x0)	0x56 00 20 01 00 00 03

ULM: Daten-Bus Operationen

- Fetch-Operation:

  Lade 1, 2, 4 oder 8 Bytes vom RAM in ein Register.
- Store-Operation:

  Schreibe 1, 2, 4 oder 8 Bytes von einem Register in den RAM.
- Input-Operation:

  Empfange vom Eingabegerät ein Byte in ein Register.
- Output-Operation:

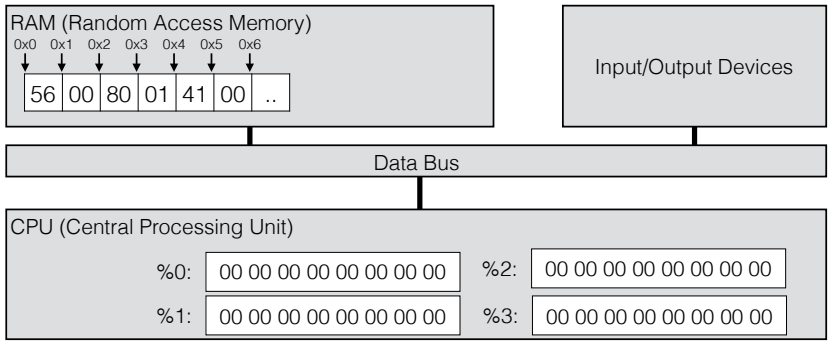
  Sende von einem Register ein Byte ans Ausgabegerät

  (dort wird der ASCII-Wert ausgegeben).

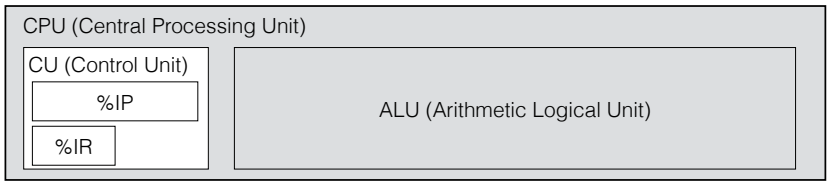
ULM: Beispiel für eine Daten-Bus Operation (Fetch-Operation)

Übung

Lade 2 Bytes von Adresse 0x2
… in das Register 3
(Erweitere die Bytes mit Nullen)

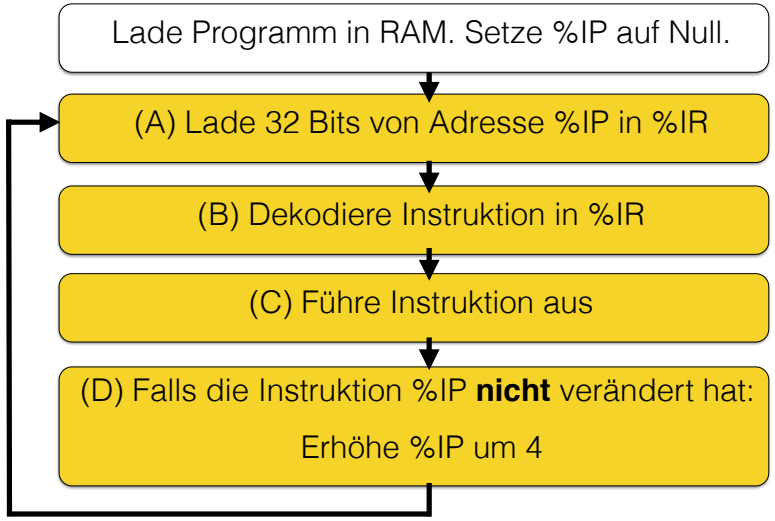


ULM: Control-Unit (Teil der CPU)



- Steuert die ALU und den Daten-Bus.
- Hat selbst wiederum zwei Register:
  - Instruction-Pointer (IP) mit 64 Bits.
  - Instruction-Register (IR) mit 32 Bits.
- Führt den Von-Neumann-Zyklus aus.

ULM: Von-Neumann-Zyklus



ULM: Befehlssatz (Auszug)

Op-Code	Beschreibung
0x01	Von-Neumann-Zyklus beenden mit Exit-Code %X
0x1B	Fetch-Operation (mit Null-Erweiterung): %Z ← M_1(X + %Y)
0x38	Addition: %Z ← %Y + X. Anschliessend ZF aktualisieren.
0x61	Sende das niedrigstwertige Byte aus Register %X an das Ausgabegerät (siehe ASCII table)

ULM: Dekorieren einer Instruktion

Dekodieren einer 32-Bit Instruktion:

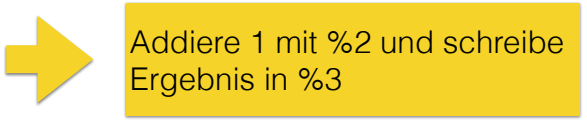
- Beispiel:

38	01	02	03

- Bezeichnung:

OP	X	Y	Z

- Suche OP im Befehlssatz (in Spalte Op-Code) und
- ersetze dann X, Y, Z in der Beschreibung.



Decimal - Binary - Octal - Hex – ASCII Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	“	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	‘	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	l
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

Wiederholung (Zahlendarstellungen) 1

- Der Computer rechnet mit Binärzahlen.
- Die ULM (und die meisten “echten” Computern) rechnen mit 64-stelligen Binärzahlen.
- Bei den Übungen rechnen wir aber mit “kürzeren” Bit-Mustern.

- Bisher haben wir Binärzahlen immer als **nicht-negative** ganze Zahlen interpretiert. Zum Beispiel

$$(1101)_2 = 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 13$$

Wiederholung (Wie addiert ein Rechner?) 2

- Im Prinzip wie beim schriftlichen Addieren.
- Aber:
  - Es wird immer mit einer festen Anzahl von Stellen gerechnet.
  - Eventuell wird der letzte Übertrag ignoriert.
  - Beispiel für 8-stelliges Rechnen:

```
  11111111
+ 00000001
  11111111   <- Zeile mit Überträgen
= 00000000
```

Das Ergebnis wird auf die letzten acht Stellen abgeschnitten!

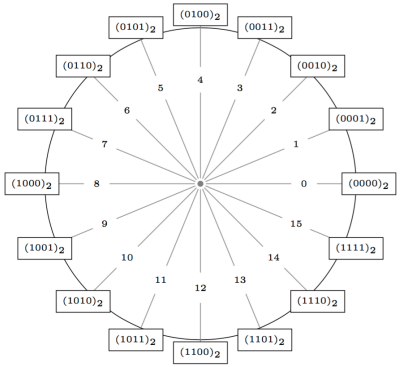
Begriffe: Unsigned-Integer und Signed-Integer 3

- **Integer:**
  - Ganze Zahl
- **Unsigned Integer:**
  - Vorzeichenlose ganze Zahl oder nicht-negative ganze Zahl
  - Also größer oder gleich Null
- **Signed Integer:**
  - Vorzeichenbehaftete ganze Zahl
  - Also auch negative ganze Zahlen

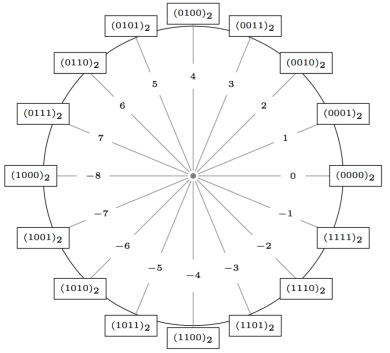
- **Wie unterscheidet sich die Addition bei Unsigned/Signed?**
  - Wir werden sehen: **Gar nicht!**

Beispiel: Interpretation einer 4-stelligen Binärzahl 4

Unsigned



Signed



- Signed-Integer werden hier mit dem sogenannten Zweier-Komplement dargestellt.
- Bei Signed-Integer erkennt man negative Zahlen so: _____
- Wie man die Addition/Subtraktion von Bit-Mustern als Addition/Subtraktion zugeordneter Winkel vorstellen kann, wird an der Tafel erklärt …

Der Rechner weiss nicht, ob er mit Signed-Integer oder Unsigned-Integer rechnet! Beim Zweier-Komplement wird “automatisch beides gerechnet”:

Das rechnet der Computer	Interpretation als Unsigned-Integer	Interpretation als Signed-Integer
1000 + 0111 = 1111	8 + 7 = 15	-8 + 7 = -1
	Korrekt	**Korrekt**

- Nicht jede Rechnung kann korrekt sein, da nur mit einer festen Anzahl von Stellen rechnen.
- Nach jeder Rechnung werden **Status-Flags** gesetzt:

- **Carry Flag (CF)**
  Gibt an, ob die Rechnung für Unsigned falsch ist.
  CF = 1 bedeutet falsch für Unsigned
  CF = 0 bedeutet korrekt für Unsigned

- **Overflow Flag (OF)**
  Gibt an, ob die Rechnung für Signed falsch ist.
  OF = 1 bedeutet falsch für Signed
  OF = 0 bedeutet korrekt für Signed

- Weitere **Status-Flags**:

- **Zero Flag (CF)**
  Gibt an, ob beim Ergebnis alle Bits Null sind.
  ZF = 1 bedeutet alle Bits sind Null
  ZF = 0 bedeutet mindestens ein Bit ist ungleich Null

- **Signed Flag (SF)**
  Gibt an, ob im Ergebnis das **höchstwertigste Bit** (das Bit ganz links) gleich Eins ist. Also bei Signed-Interpretation das Ergebnis negativ ist.
  SF = 1 bedeutet das Bit ist gesetzt
  SF = 0 bedeutet das Bit ist nicht gesetzt

Das rechnet der Computer	Interpretation als Unsigned-Integer	Interpretation als Signed-Integer
1000 + 0111 = 1111	8 + 7 = 15	-8 + 7 = -1
	Korrekt	**Korrekt**

Status-Flags nach der Rechnung:

ZF = 0	CF = 0	SF = 1	OF = 0

Aufgabe: 4-Bit-Addierer 9

Das rechnet der Computer	Interpretation als Unsigned-Integer	Interpretation als Signed-Integer
1010 + 0011 = Zeile mit Überträgen	+ =	+ =

Status-Flags nach der Rechnung:

ZF =	CF =	SF =	OF =

Aufgabe: 4-Bit-Addierer 11

Das rechnet der Computer	Interpretation als Unsigned-Integer	Interpretation als Signed-Integer
1010 + 0111 = Das Ergebnis wird auf die letzten vier Stellen abgeschnitten!	+ =	+ =

Status-Flags nach der Rechnung:

ZF =	CF =	SF =	OF =

Aufgabe: 4-Bit-Addierer 13

Das rechnet der Computer	Interpretation als Unsigned-Integer	Interpretation als Signed-Integer
0111 + 0001 =	+ =	+ =

Status-Flags nach der Rechnung:

ZF =	CF =	SF =	OF =

Aufgabe: 4-Bit-Addierer 15

Das rechnet der Computer	Interpretation als Unsigned-Integer	Interpretation als Signed-Integer
1010 + 0110 = Das Ergebnis wird auf die letzten vier Stellen abgeschnitten!	+ =	+ =

Status-Flags nach der Rechnung:

ZF =	CF =	SF =	OF =

Aufgabe: 4-Bit-Addierer

Das rechnet der Computer	Interpretation als Unsigned-Integer	Interpretation als Signed-Integer
1000 + 1111 _____ =	+ _____ =	+ _____ =

Das Ergebnis wird auf die letzten vier Stellen abgeschnitten!

Status-Flags nach der Rechnung:

ZF =	CF =	SF =	OF =

Aufgabe: 8-Bit-Addierer (Zahlen stellen wir hexadezimal dar)

Das rechnet der Computer	Interpretation als Unsigned-Integer	Interpretation als Signed-Integer
0x81 + 0x0F _____ =	+ _____ =	+ _____ =

Zeile mit Überträgen

Status-Flags nach der Rechnung:

ZF =	CF =	SF =	OF =

8-Bit-Addierer (Zahlen stellen wir hexadezimal dar)

Das rechnet der Computer	Interpretation als Unsigned-Integer	Interpretation als Signed-Integer
0x1A + 0x08 _____ =	+ _____ =	+ _____ =

Zeile mit Überträgen

Status-Flags nach der Rechnung:

ZF =	CF =	SF =	OF =

8-Bit-Addierer (Zahlen stellen wir hexadezimal dar)

Das rechnet der Computer	Interpretation als Unsigned-Integer	Interpretation als Signed-Integer
0xFF + 0x01 _____ =	+ _____ =	+ _____ =

Das Ergebnis wird auf 8 Bits abgeschnitten! (Also 2 Hex-Ziffern)

Status-Flags nach der Rechnung:

ZF =	CF =	SF =	OF =

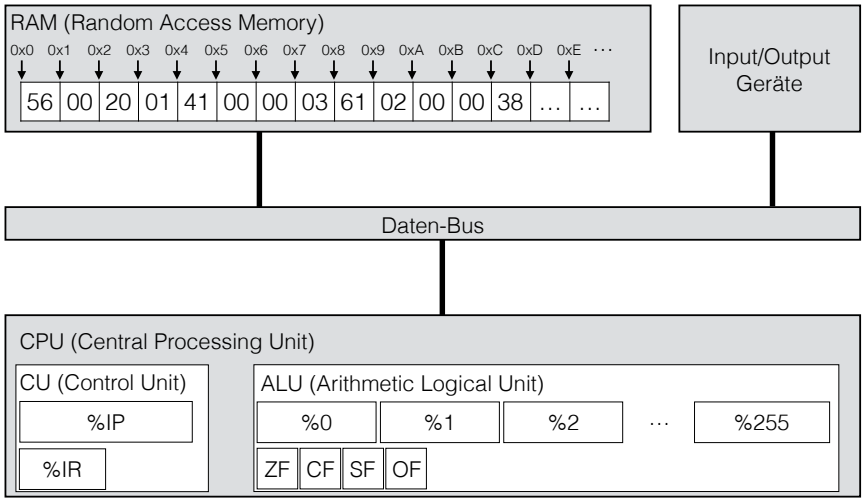
Bit-Muster auffüllen

- Die ULM rechnet immer mit 64-Bit breiten Zahlen
- Um zum Beispiel mit 16-Bit breiten Zahlen zu rechnen, müssen die Bits aufgefüllt werden.
- Hier werden zwei Varianten unterschieden:
  - **Null-Erweiterung:**
    - `0000 … 0000 1011 1010 1100 0011`
    - Der Unsigned-Wert bleibt gleich
  - **Vorzeichenerweiterung:**
    - `1111 … 1111 1011 1010 1100 0011`
    - `0000 … 0000 0011 1010 1100 0011`
    - Der Signed-Wert bleibt gleich

Bit-Muster auffüllen

- **Null-Erweiterung in Hex-Darstellung:**
  - `0x00 00 00 00 00 00 BA C3`
  - **Vorzeichenerweiterung:**
    - `0xFF FF FF FF FF FF BA C3`
    - `0x00 00 00 00 00 00 3A C3`
    - Merkregel:
      - Wenn eine Hex-Zahl links mit 0,1,..7 beginnt, dann mit 0 auffüllen.
      - Ansonsten mit “F” auffüllen.

Wiederholung (ULM: Von-Neumann-Architektur)



Wiederholung (Beschreibung des Befehlssatzes)

- **Register-Inhalte**
  - %X ist der Inhalt von Register X.
  - Zum Beispiel: %1 ist der Inhalt von Register 1.
- **Speicher-Inhalte**
  - $M_n(X)$ = "n Bytes ab Adresse X"
  - Zum Beispiel: $M_1(0x20)$ = "1 Byte ab Adresse 0x20”
- **Besonderheiten**
  - Null-Register: In %0 stehen immer Nullen!
  - Instruktion Pointer hat 64-Bits und wird mit %IP bezeichnet.
  - Instruction Register hat 32-Bits und wird mit %IR bezeichnet.

ULM: Dekorieren einer Instruktion

Dekodieren einer 32-Bit Instruktion:

- Beispiel:

38	01	02	03

- Bezeichnung:

OP	X	Y	Z

- Suche OP im Befehlssatz (in Spalte Op-Code) und
- ersetze dann X, Y, Z in der Beschreibung.

Addiere 1 mit %2 und schreibe Ergebnis in %3

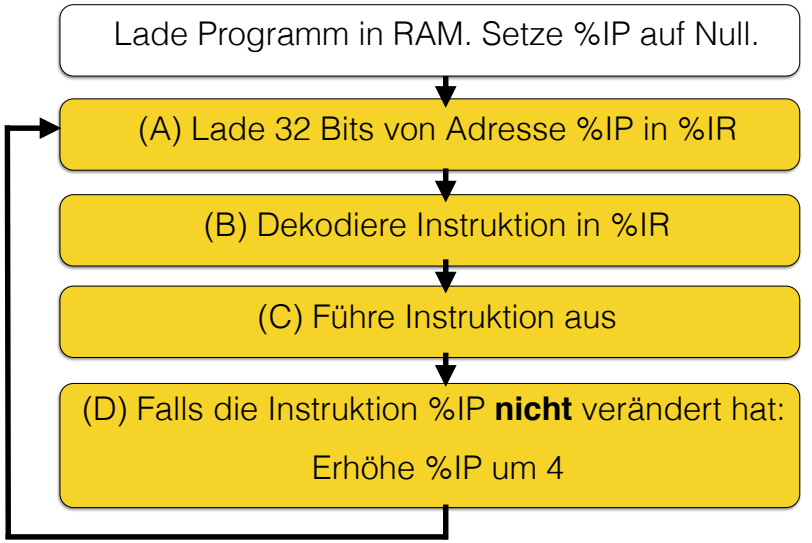
Wiederholung (ULM: Auszug aus dem Befehlssatz)

Op-Code	Beschreibung
0x01	Von-Neumann-Zyklus beenden mit Exit-Code %X
0x1B	Fetch-Operation (mit Null-Erweiterung): %Z ← M_1(X + %Y)
0x38	Addition: %Z ← %Y + X. Anschliessend ZF aktualisieren.
0x56	16-Bit Wert (mit Null-Erweiterung) in Register laden: %Z ← XY
0x61	Sende das niedrigstwertige Byte aus Register %X an das Ausgabegerät (siehe ASCII table)

Weitere Befehle

Op-Code	Beschreibung
0x39	Subtraktion: %Z ← %Y - X (mit Null-Erweiterung von X)
0x41	%IP ← %IP + 4 * XYZ (mit Vorzeichenerweiterung bei XYZ)
0x43	Falls ZF = 0, dann %IP ← %IP + 4 * XYZ (mit Vorzeichenerweiterung bei XYZ)

ULM: Von-Neumann-Zyklus



Heute geht es an den Rechner!

Plan für heute:

1. Mit dem ULM-Debugger (ulm-qui) ein Programm Schritt-für-Schritt ausführen.
2. Das Programm im ULM-Simulator auf dem Kommandozeile ausführen.
3. Das Programm so verändern, dass es einen anderen Text ausgibt.
4. Das Programm in Assembler schreiben (statt direkt in Maschinen-Code).
5. Weiteres Beispiel in Assembler.

Maschinen-Code in eine Datei schreiben

- Benutzt einen Editor, um den Code (s.u.) einzutippen.
- Speichert die Datei ab als “hello.ulm”

```
56 00 20 01
41 00 00 03
61 02 00 00
38 01 01 01
1B 00 01 02
39 00 02 00
43 FF FF FC
01 00 00 00
48 69 00
```

Bemerkungen

- Der ULM-Debugger und ULM-Simulator ignorieren Leerzeichen und Zeilenumbrüche.
- Man hätte also auch folgendes tippen können:

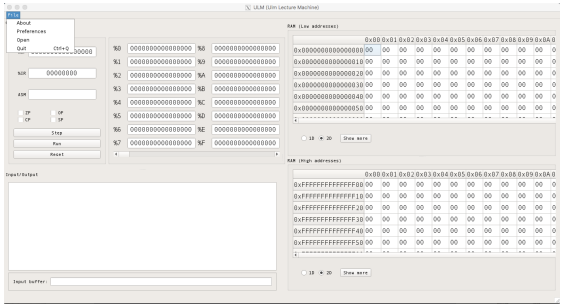
```
560020014100003610200003801010 11B000102
3900020043FFFFFC01000000486900
```

- Den Maschinen-Code kann man auch kommentieren.
- Alles nach einem Hashtag (‘#’) wird ebenfalls ignoriert.
- Also hätte man auch schreiben können:

```
56 00 20 01 # Lade 0x20 in Register 1
41 00 00 03 # Springe 3 Befehle vor
61 02 00 00 # Gibt Inhalt von Reg. 2 aus
…
```

ULM-Debugger

- Öffnet ein Terminal und startet den Debugger durch Eingabe von “**ulm-qui**” (ULM Cute Interface)
- Dadurch wird eine graphische Oberfläche gestartet:



- Ladet die Datei “hello.ulm”. Durch Klicken auf “Step” wird eine Instruktion ausgeführt.
- **Aufgabe: Beim letzten Mal haben wir das Programm auf den Folien selbst schrittweise ausgeführt. Überprüft euer Resultat mit dem Debugger.**

- Im Normalfall möchte man das Programm nicht Schritt-für-Schritt ausführen.
- Mit dem ULM-Simulator **“ulm”** kann das Programm im Terminal komplett ausgeführt werden:

```
ulm hello.ulm
Hi
```

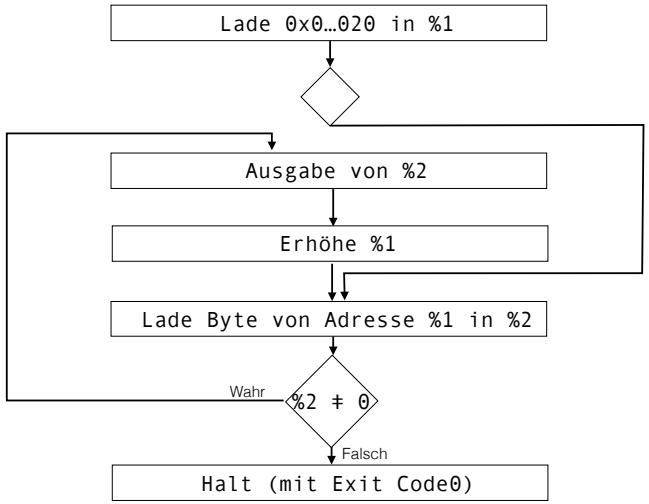
Aufgabe:

Gebt im Terminal “ulm hello.ulm” ein, um obiges Beispiel zu testen.

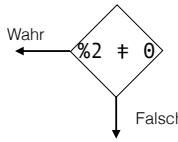
Ändert das Programm so ab, dass es den Text “Hello world!” und dann einen Zeilenumbruch ausgibt. Verwendet dazu die ASCII-Tabelle.

Hinweis: Der Zeilenumbruch wird in der Tabelle mit LF (“Linefeed”) bezeichnet und hat den Hex-Code 0x0A.

- Beginnend bei Adresse 0x0..02 stand der ASCII Code von “Hi” (gefolgt von einem Null-Byte).
- Das Programm realisiert folgende Logik:



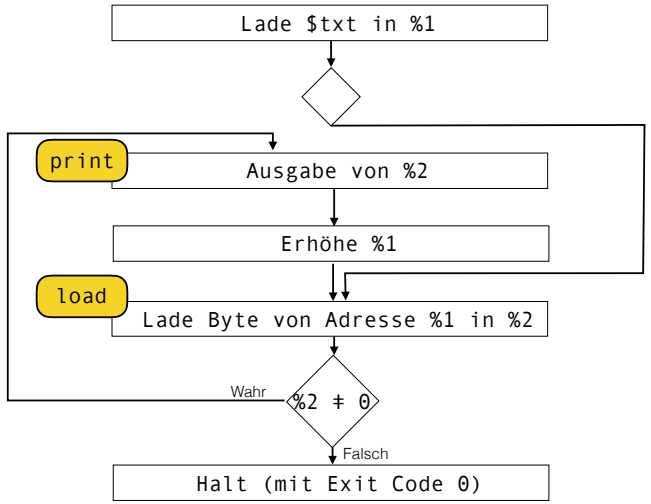
- Im Prinzip entspricht jedes Rechteck/Raute genau einem Maschinen-Befehl.
- Einzige Ausnahme: Der bedingte Sprungbefehl



Diese Befehl wurde durch zwei Befehle realisiert:

1. Der Wert Null wurde vom Register 2 abgezogen und das Ergebnis in Register 0 geschrieben.
2. Abhängig von den Status-Flags wird gesprungen: Falls das Zero-Flag nicht gesetzt ist wird gesprungen (ansonsten mit dem nächsten Befehl weitergemacht)

- Wir geben Befehle, die angesprungen werden können ein **Label** und
- wir verwenden den Platzhalter **txt** für die Adresse von “Hi”



- Textuell kann man das Diagramm und somit das Programm wie folgt beschreiben:

```
    Lade $txt in %1
    Springe zum Label load
print:
    Ausgabe von %2
    Erhöhe %1
load:
    Lade ein Byte von Adresse %1 in %2
    Subtrahiere 0 von %2
    Falls Ergebnis nicht Null springe zu print
    Halt (mit Exit Code 0)
```

- Analog kann man die nötigen Daten so beschreiben:

```
txt:
    ASCII Code für “Hi” und ein Null-Byte
```


- Textuell kann man das Diagramm und somit das Programm wie folgt beschreiben:

```
    .text
    ldzwq   $txt, %1
    jmp     load
print:
    putc    %2
    addq    $1,   %1,  %1
load:
    movzbq  (%1), %2
    subq    $0,   %2,  %0
    jnz     print
    halt    %0

    .data
txt:
    .string “Hi”
```


- Natürlich sieht das Programm auf Grund von Abkürzungen wie “ldzwq”, “movzbq”, … immer noch kryptisch aus.
- Im Vergleich zum Maschinen-Code ist es aber lesbarer.
- Die Abkürzungen sind historisch bedingt:
  - “ldzwq” steht für “Load zero extended word into quad register”.
  - Mit dem Begriff “word” bezeichnet Intel zwei Bytes (also 16 Bits) und mit “quad” vier “words” (also 64 Bit)
  - “movzbq” steht für “Move zero extended byte into quad register”.
  - “jmp” steht für “jump”.
  - “jnz” steht für “jump if not zero”.
  - “addq” und “subq” stehen jeweils für “add quad registers” und “subtract quad registers”

- Speichert das Assembler-Programm ab als “hello.s”
- Gebt im Terminal den Befehl “ulmasm hello.s” ein
- Damit wurde eine Datei “a.out” (Assembler-Output) erzeugt, die so aussehen sollte:

```
#TEXT 4
0x0000000000000000: 56 00 20 01 #  ldzwq $txt, %1
0x0000000000000004: 41 00 00 03 #  jmp load
0x0000000000000008: 61 02 00 00 #  putc %2
0x000000000000000C: 38 01 01 01 #  addq $1, %1, %1
0x0000000000000010: 1B 00 01 02 #  movzbq (%1), %2
0x0000000000000014: 39 00 02 00 #  subq $0, %2, %0
0x0000000000000018: 43 FF FF FC #  jnz print
0x000000000000001C: 01 00 00 00 #  halt %0
#DATA 1
0x0000000000000020: 48 69 00 #  .string "Hi"
#SYMTAB
#RELOCTAB
0x0000000000000000 1 2 absolute D 0x0000000000000000
```

Aufgaben:

- Führt das erzeugte Maschinen-Programm mit “ulm a.out” aus.
- Ändert “hello.s” so ab, dass “Hello world!” ausgegeben wird.

Die Datei “a.out” enthält zusätzliche Informationen, die vom Simulator ignoriert werden.

Relevant ist nur der nicht markierte Teil:

```
#TEXT 4
0x0000000000000000: 56 00 20 01 #  ldzwq $txt, %1
0x0000000000000004: 41 00 00 03 #  jmp load
0x0000000000000008: 61 02 00 00 #  putc %2
0x000000000000000C: 38 01 01 01 #  addq $1, %1, %1
0x0000000000000010: 1B 00 01 02 #  movzbq (%1), %2
0x0000000000000014: 39 00 02 00 #  subq $0, %2, %0
0x0000000000000018: 43 FF FF FC #  jnz print
0x000000000000001C: 01 00 00 00 #  halt %0
#DATA 1
0x0000000000000020: 48 69 00 #  .string "Hi"
#SYMTAB
#RELOCTAB
0x0000000000000000 1 2 absolute D 0x0000000000000000
```


- Schreibt das folgende Assembler-Programm ab und speichert es ab als “obscure.s”:

```
   .text

loop:
   getc   %1
   subq   $'\n',   %1,    %0
   jz     done
   addq   $1,      %1,    %1
   putc   %1
   jmp    loop

done:
   halt   %0
```

- Übersetzt das Programm in Maschinen-Code und führt es aus.
- **Aufgaben:**
  - Was macht das Programm?
  - Wie kann das Programm als Diagramm dargestellt werden?

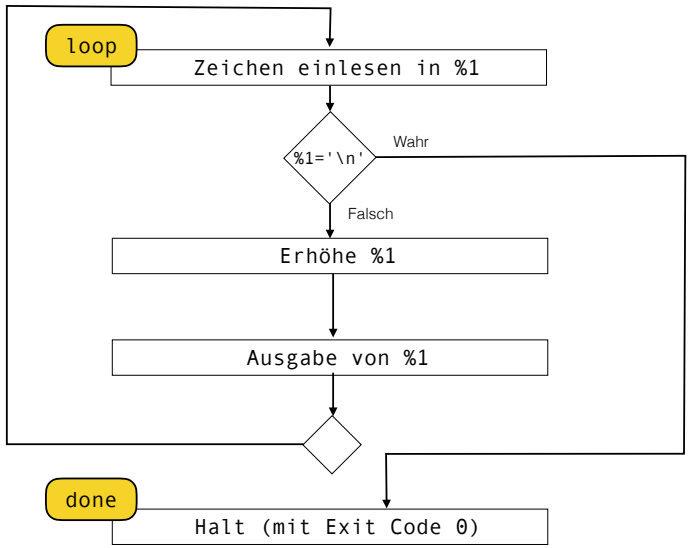
- Beim letzten Mal haben wir Programme in Assembler (statt in Maschinen-Code) geschrieben.
- Der Quellcode “obscure.s” war:

```
   .text

loop:
   getc   %1
   subq   $'\n',   %1,    %0
   jz     done
   addq   $1,      %1,    %1
   putc   %1
   jmp    loop

done:
   halt   %0
```

- Mit dem Assembler *ulmasm* konnte es in Maschinen-Code (gespeichert in „a.out“) übersetzt und ausgeführt werden:

ulmasm obscure.s
ulm a.out

- Dahinter verbirgt sich folgende Logik:

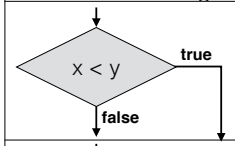


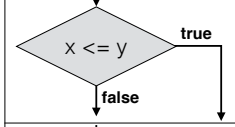
Aufgabe: Erstellt ein Ablaufdiagramm für die Cäsar-Codierung

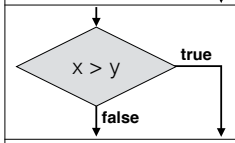
Dies soll folgende Eigenschaften haben:

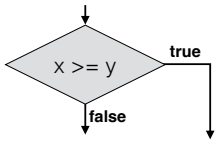
- Wenn ein Zeilenumbruch ('\n') eingegeben wird, dann wird das Programm beendet
- Die Zeichen von ‚a‘ bis ‚z‘ werden um Eins verschoben. Also ‚a‘ auf ‚b‘, ‚b‘ auf ‚c‘, …, ‚z‘ auf ‚a‘.
- Die Zeichen von ‚A‘ bis ‚Z‘ werden um Eins verschoben. Also ‚A‘ auf ‚B‘, ‚B‘ auf ‚C‘, …, ‚Z‘ auf ‚A‘.

Bedingung	Umsetzung in Assembler

	`# %x - %y < 0`
	`# jump below`
	`subq %y, %x, %0`
	`jb label`

	`# %x - %y <= 0`
	`# jump below equal`
	`subq %y, %x, %0`
	`jbe label`

	`# %x - %y > 0`
	`# jump above`
	`subq %y, %x, %0`
	`ja label`

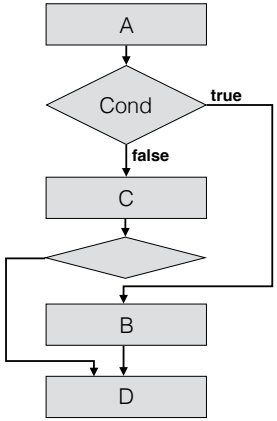
	`# %x - %y >= 0`
	`# jump above equal`
	`subq %y, %x, %0`
	`jae label`

Darstellung in einer höheren Sprache: If-then-else (Variante 1)

Pseudo code

```
A;
if (Cond) {
    B;
}else {
    C;
}
D;
```

Flow chart



Assembly code structure

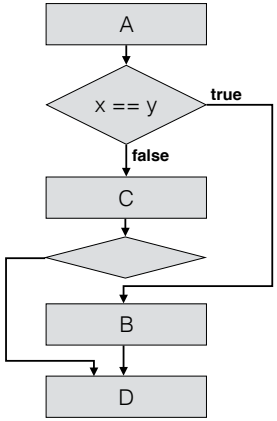
```
    A
    if Cond jmp label_B

    C
    jmp label_D
label_B:
    B
label_D:
    D
```

Darstellung in einer höheren Sprache: If-then-else (Beispiel für Variante 1)

Pseudo code

```
A;
if (x==y) {
    B;
}else {
    C;
}
D;
```

Flow chart



Assembly code structure

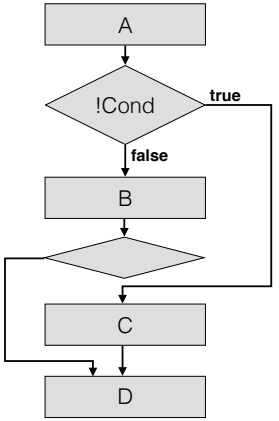
```
    A
    sub %y, %x, %0
    jz label_B
    C
    jmp label_D
label_B:
    B
label_D:
    D
```

Darstellung in einer höheren Sprache: If-then-else (Variante 2)

Pseudo code

```
A;
if (Cond) {
    B;
}else {
    C;
}
D;
```

Flow chart



Assembly code structure

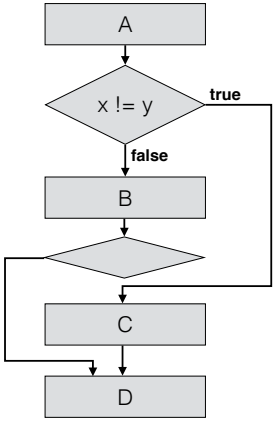
```
    A
    if !Cond jmp label_C

    B
    jmp label_D
label_C:
    C
label_D:
    D
```

Darstellung in einer höheren Sprache: If-then-else (Beispiel für Variante 2)

Pseudo code

```
A;
if (x==y) {
    B;
}else {
    C;
}
D;
```

Flow chart



Assembly code structure

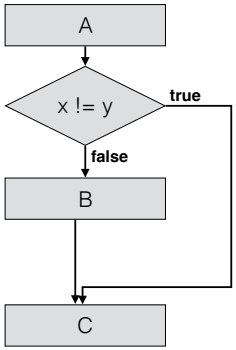
```
    A
    sub %y, %x, %0
    jnz label_C
    B
    jmp label_D
label_C:
    C
label_D:
    D
```

Darstellung in einer höheren Sprache: If-then (Beispiel)

Pseudo code

```
A;
if (x==y) {
    B;
}
C;
```

Flow chart



Assembly code structure

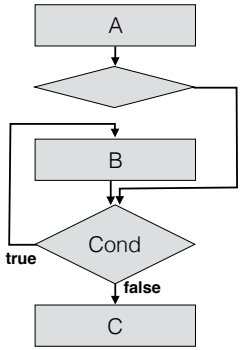
```
    A
    sub %y, %x, %0
    jnz label_C
    B
label_C:
    C
```

Darstellung in einer höheren Sprache: While-Schleife

Pseudo code

```
A;
while (Cond){
    B;
}
C;
```

Flow chart



Assembly code structure

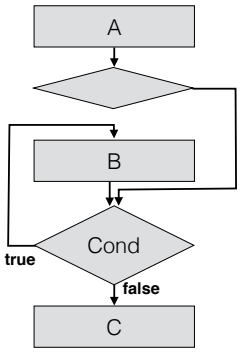
```
    A
    jmp label_Cond
label_B:
    B;
label_Cond:
    if Cond jmp label_B

    C;
```

Darstellung in einer höheren Sprache: While-Schleife

Pseudo code

```
A;
while (x==y){
    B;
}
C;
```

Flow chart



Assembly code structure

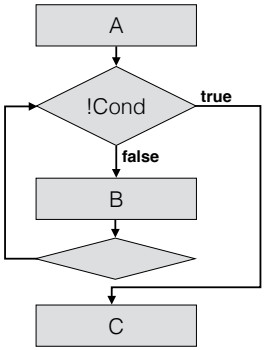
```
    A
    jmp label_Cond
label_B:
    B;
label_Cond:
    subq %y, %x, %0
    jz label_B
    C;
```

Darstellung in einer höheren Sprache: While-Schleife

Pseudo code

```
A;
while (Cond){
    B;
}
C;
```

Flow chart



Assembly code structure

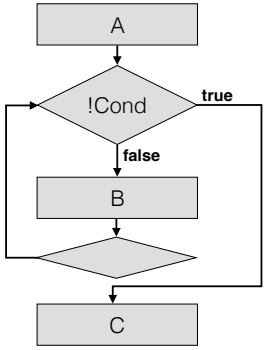
```
    A
label_Cond:
    if !Cond jmp label_C

    B;
    jmp label_Cond
label_C:
    C;
```

Darstellung in einer höheren Sprache: While-Schleife

Pseudo code

```
A;
while (x==y){
    B;
}
C;
```

Flow chart



Assembly code structure

```
    A
label_Cond:
    subq %y, %x, %0
    jnz label_C
    B;
    jmp label_Cond
label_C:
    C;
```