7. Mikroprozessortechnik

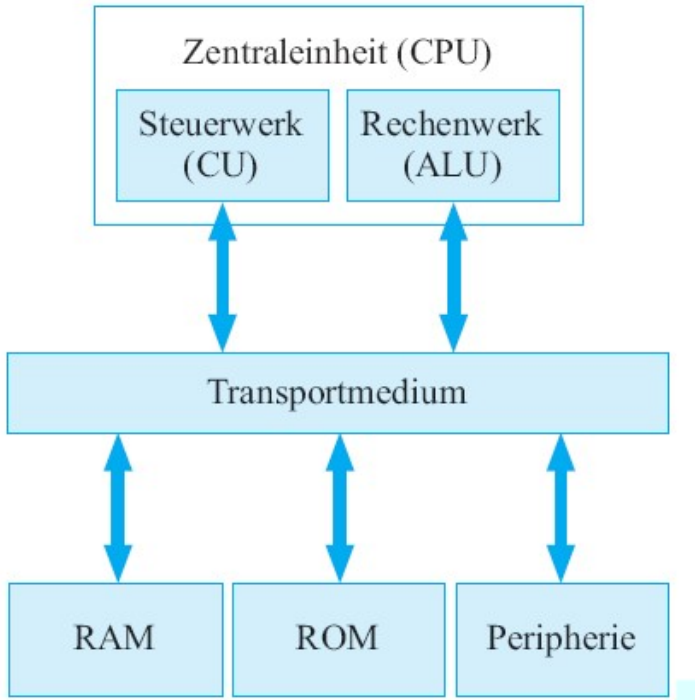
- **Rechnerarchitekturen**
  - Technische Informatik beschäftigt sich insbesondere mit den technischen Realisierungen von informationsverarbeitenden Systemen
  - mehrere exotische Möglichkeiten (oft bisher nur für spezielle Algorithmen geeignet)
    - DNA-Computing
      - Computer im Reagenzglas
      - DNA-Stränge kodieren Informationen, passende Stränge lagern sich aneinander, führen dadurch Berechnungen aus
      - 10^{23} DNA-Stränge arbeiten parallel (nur einmal Schütteln!)
    - Quantencomputer
      - rechnen mit Qubits (0 und 1 gleichzeitig, bzw. bei n Qubits wird mit allen 2^n Belegungen gleichzeitig, also in einem Schritt, gerechnet)
    - ???
  - wir werden uns hier nur mit Digitalcomputern mit der Von-Neumann-Architektur beschäftigen

Von-Neumann-Architektur

- **John von Neumann**
  - ungarischer Mathematiker, 1903-1957
  - hat viele wichtige Beiträge zu vielen Wissenschaftsgebieten geleistet
    - von der Mengen-Theorie über Wirtschafts-wissenschaften bis zur Quantenmechanik und Kernphysik (hat an der ersten Atombombe mitgebaut)
  - Von-Neumann Architektur
    - 1945 beschrieb er erstmalig eine Computerarchitektur, in der die Befehle als Programm gespeichert werden, und zwar in demselben Speicher, in dem sich auch die Daten befinden
    - diese Rechnerarchitektur hat sich bis heute im Wesentlichen gehalten

Von-Neumann-Architektur (2)

- **Eigenschaften**
  - Zentraleinheit (CPU)
    - zentrales Steuerelement
    - führt Befehle aus
      - Anwendungsprogramme
      - Betriebssystem
    - wegen seiner dominierenden Rolle Zentraleinheit (Central Processing Unit, CPU) genannt
  - Programmsteuerung
    - im Speicher (ROM und RAM) abgelegte Datenwörter werden als Maschinenbefehle interpretiert und ausgeführt
    - freie Programmierbarkeit
    - universell einsetzbar (Befehlssatz mächtig genug, um beliebige Algorithmen auszuführen, s.a. Turingmaschine aus der Theoretischen Informatik)



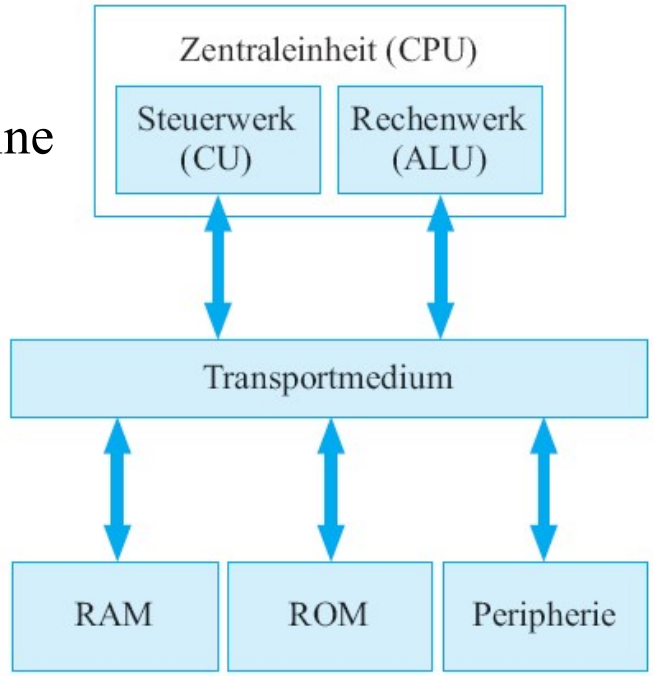
Von-Neumann-Architektur (3)

- Programme und Daten werden nicht strikt voneinander getrennt
  - beide liegen in demselben Hauptspeicher
  - der Prozessor interpretiert die gelesenen Worte entweder als Befehl, wenn er einen Befehl erwartet, oder als Daten, wenn er einen Befehl zum Lesen von Daten ausführt
  - wird häufig als zentrales Merkmal eines Von-Neumann-Rechners angesehen
    - im Unterschied zur **Harvard-Architektur**, bei der sich Daten und Befehle in verschiedenen Speichern befinden

- **heutige CPUs sind eine Mischung aus Von-Neumann- und Harvard-Architekturen**
  - Von-Neumann-Aspekt
    - Daten und Befehle befinden sich in demselben Hauptspeicher
  - Harvard-Aspekt
    - im Prozessor gibt es getrennte Daten- und Befehls-Caches (schnelle Zwischenspeicher, s.u.), auf die gleichzeitig zugegriffen werden kann

Mikroprozessortechnik (2)

- **Mikroprozessor**
  - seit 1971 kann man komplette CPUs als eine einzige integrierte Schaltung herstellen
  - diese CPUs nennt man Mikroprozessoren
- **Von-Neumann-Rechner**
  - Zentraleinheit
    - besteht aus Rechenwerk (Datenpfad) und Steuerwerk (Steuerung)
  - Speicher
    - ROM (oder Flash) für den Kaltstart nach Einschalten der Stromversorgung
    - RAM als Hauptspeicher für Daten und Programme
  - Peripherie
    - Tastatur, Grafikkarte, Festplatte, Drucker, Maus, Netzwerkkarte, …
  - Transportmedium
    - Bussystem zur Verbindung der Komponenten



Arbeitsweise

- **Programme**
  - setzen sich aus Maschinenbefehlen zusammen
- **Maschinenbefehle**
  - stehen an aufeinander folgenden Adressen im Hauptspeicher
  - werden normalerweise sequentiell abgearbeitet
  - Ausnahmen
    - Sprungbefehle (unbedingte und bedingte)
      - gewollte Verzweigungen im Programmablauf
    - Unterprogrammaufrufe
      - gewollte Verzweigungen in ein Unterprogramm, später Rückkehr an die Stelle, von der aus verzweigt wurde
    - Interrupts, Exceptions
      - Unterbrechungen, die durch Ereignisse (Tastatureingabe, Division durch 0, ….) erzwungen werden
  - Sequentieller Kontrollfluss wird dadurch geändert

Arbeitsweise (2)

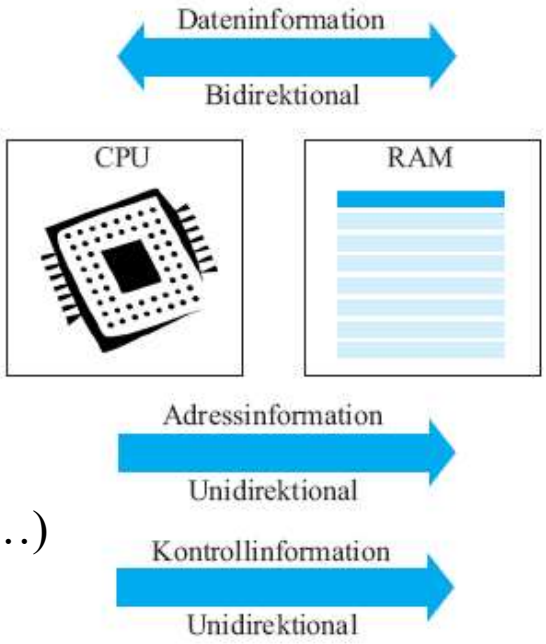
- **Beispiel**
  - Anweisung in einer Hochsprache (z.B. Java, C++, …)

```
if (x != 0) y = y + 1;
```

  - wird z.B. so von einem Compiler in die Maschinensprache (Assembler) eines Mikroprozessors übersetzt:

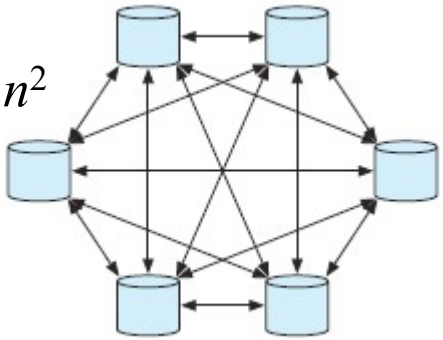
```
start:  LDA (14)     // Operand laden (x)
        BRZ weiter:  // Bedingter Sprung
        LDA (15)     // Operand laden (y)
        ADD #1       // Addition von 1
        STA (15)     // Operand speichern (y)
weiter: ...
```

  - Datenaustausch mit Speicher (LDA, STA)
    - häufige Operation
    - Adressen werden zum Speicher gesendet (14 und 15)
    - Kontrollinformationen werden gesendet (WE, …)
    - Daten werden gelesen oder geschrieben

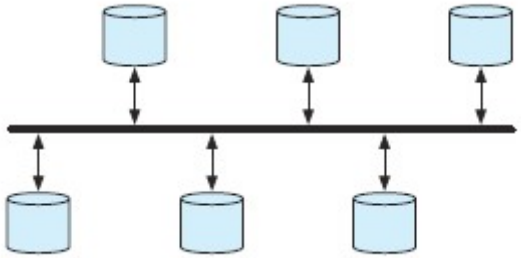


Kommunikationsarten

- **Direkte Kommunikation**
  - jede Komponente ist mit jeder anderen verbunden
  - bei n Komponenten wächst die Anzahl der Wege wie n^2
  - hoher Datendurchsatz, da viele Komponenten gleichzeitig Daten austauschen können

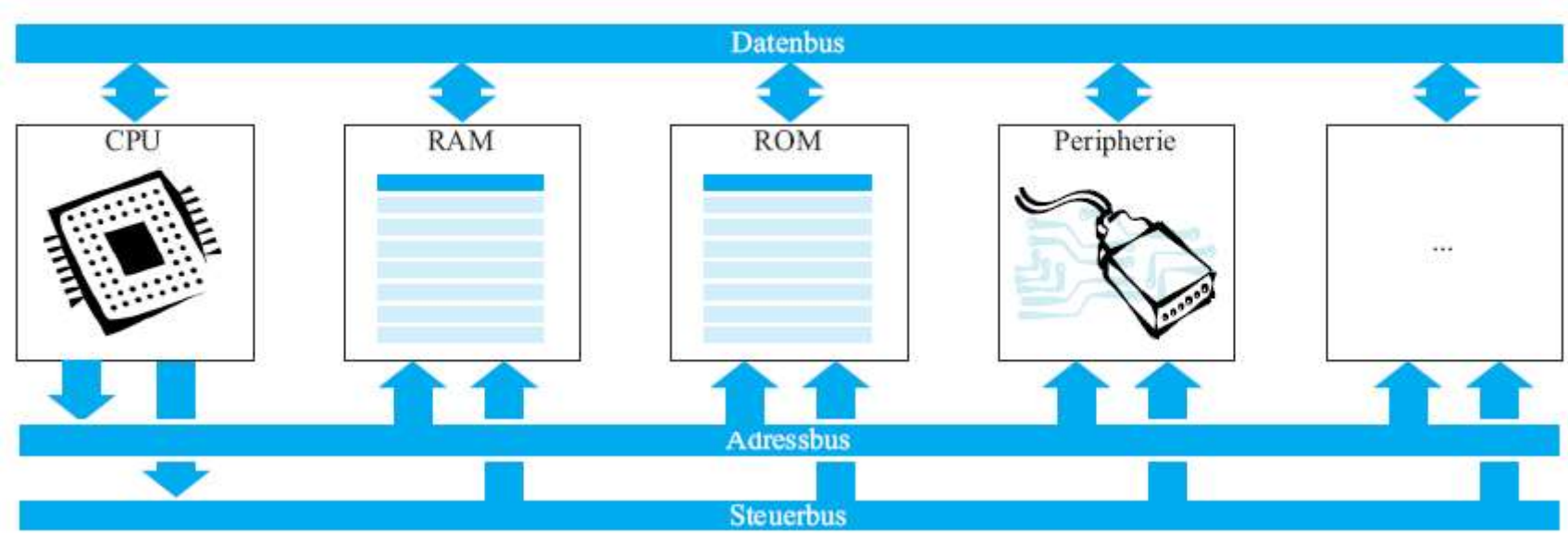


- **Indirekte Kommunikation**
  - Bus-Topologie
    - zentraler Transportweg, an den alle Komponenten angeschlossen werden
  - es darf immer nur eine Komponente schreiben, die anderen können vom Bus lesen
  - flexibel erweiterbar
  - wird aber schnell zum Flaschenhals
    - insbesondere, wenn Befehle und Daten im selben Speicher sitzen
    - "Von-Neumann-Bottleneck"



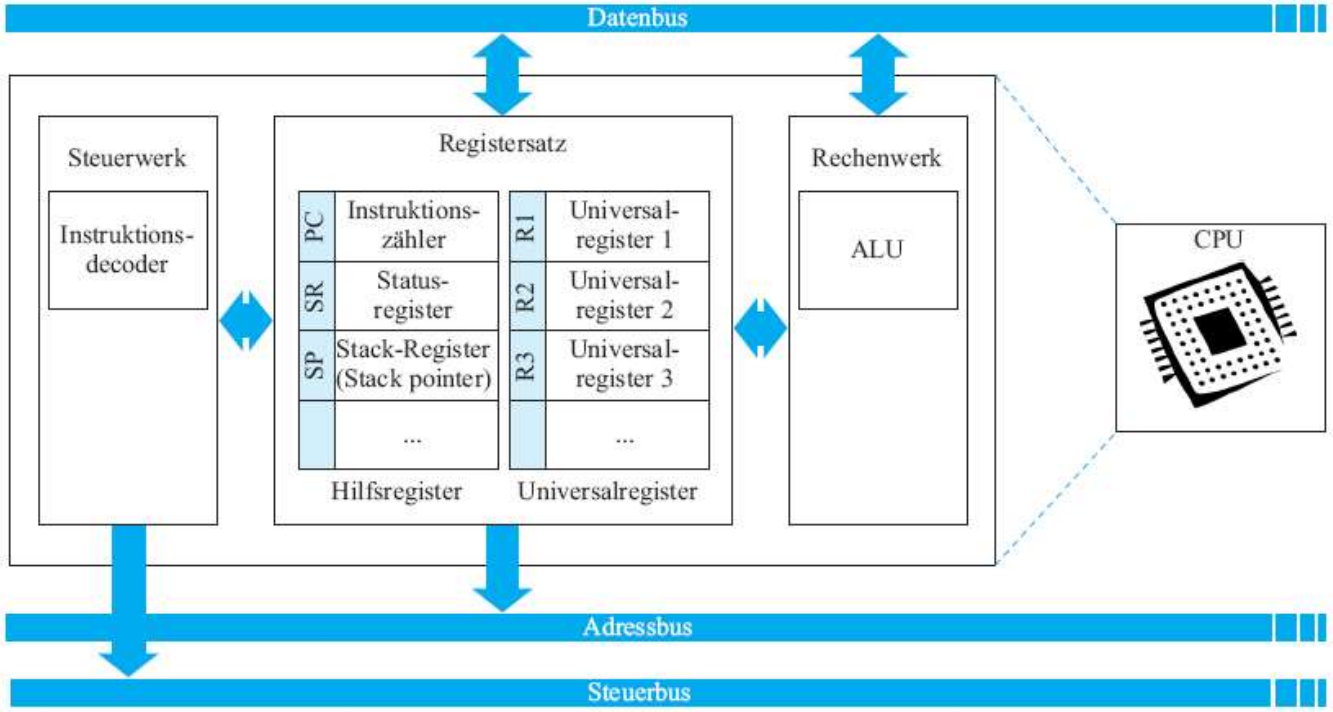
I/O-Komponenten

– auch diverse Eingabe-/Ausgabe (Input/Output oder kurz I/O) Komponenten werden von der CPU wie ein Speicher unter einer festgelegten Adresse angesprochen



Aufbau der CPU

- **Steuerwerk**
  - liest den nächsten Befehl (Adresse steht im Instruktionszähler)
  - dekodiert den gelesenen Befehl
  - setzt die Steuersignale so, dass die Instruktion im Datenpfad (Rechenwerk) korrekt ausgeführt wird

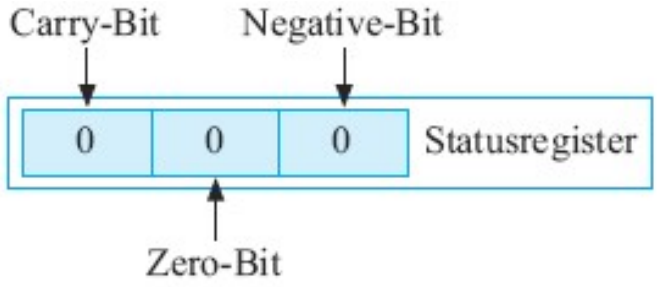


Aufbau der CPU (2)

- **Registersatz**
  - zwei Arten von Registern:
  - Universalregister
    - dienen zum Speichern von Zwischenergebnissen
    - ein Universalregister würde theoretisch ausreichen
    - mehr Register reduzieren die Notwendigkeit von Datentransfers vom/zum Hauptspeicher erheblich
    - moderne Prozessoren haben 16, 32 oder mehr Universalregister
  - Hilfsregister
    - dienen besonderen Zwecken
    - fast immer gibt es die folgenden Hilfsregister
      - Instruktionszähler (s.u.)
      - Statusregister (s.u.)
      - Stackpointer (zur einfachen Realisierung eines Stacks, hier nicht weiter besprochen)

Hilfsregister

- **Instruktionszähler (PC, Program Counter)**
  - enthält die Adresse des nächsten auszuführenden Befehls
  - Inhalt wird zur Adressierung des Hauptspeichers verwendet
  - nach Lesen von der Adresse wird Inhalt um 1 erhöht
  - bei Sprungbefehlen schreibt man einfach die Zieladresse in den PC
- **Statusregister (SR)**
  - wird vom Rechenwerk beschrieben
  - enthält Informationen über das Ergebnis der zuletzt ausgeführten arithmetischen Operation
    - z.B. Carry-Bit (C), Zero-Bit (Z), Negative-Bit (N)
  - damit können bedingte Sprünge realisiert werden
    - z.B. BRZ … **BR**anch if **Z**ero



Hilfsregister (2)

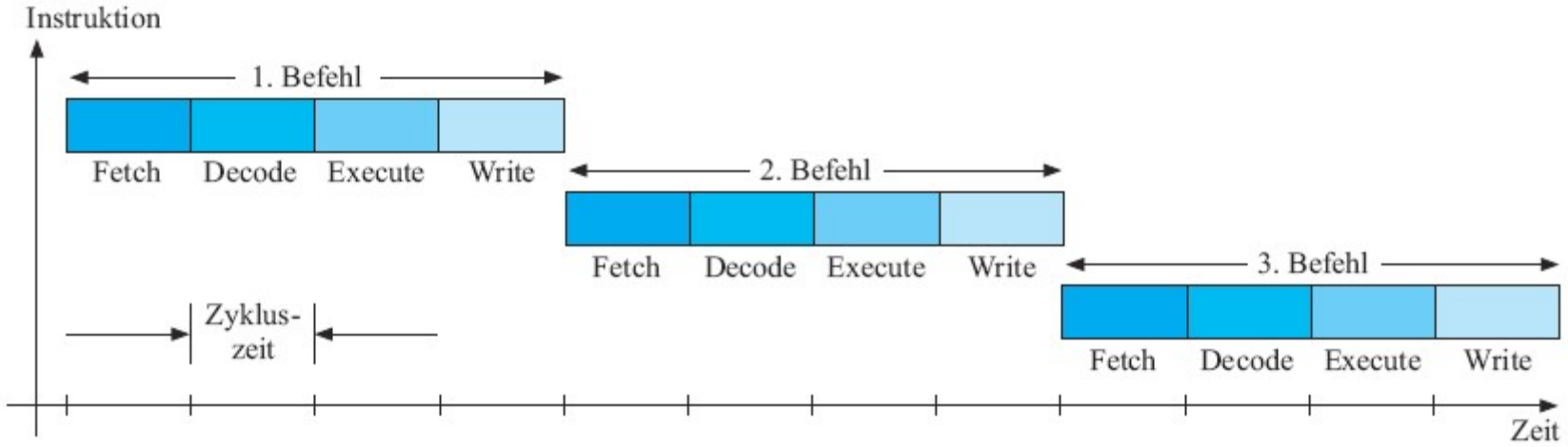
- **Instruktionsregister (IR)**
  - dient zum Zwischenspeichern des auszuführenden Befehls
    - der Befehl wird aus dem Hauptspeicher gelesen und im IR im Innern des Steuerwerkes abgelegt
  - das Steuerwerk kann nun auf jedes Bit des Befehls zugreifen und entsprechende Steuersignale für den Datenpfad erzeugen
- **Datenregister (DR)**
  - dient zum Zwischenspeichern von Daten
    - Wort wird aus dem Hauptspeicher gelesen und im DR abgelegt
  - Datenpfad benutzt dann die Daten aus dem DR

Befehlsphasen

- **Bearbeitung eines Maschinenbefehls**
  - mindestens vier Phasen (in komplexeren Prozessoren auch mehr)
    - Fetch (Holen des Befehls aus dem Hauptspeicher)
      - Inhalt des PC (Program Counter) wird auf den Adressbus gelegt
      - Daten werden vom Hauptspeicher ins Instruktionsregister (IR) geholt
      - PC wird inkrementiert (zeigt dann also auf den nächsten normalerweise zu lesenden Befehl im Speicher)
    - Decode (Dekodierphase)
      - der eigentliche Maschinenbefehl (Opcode) im IR wird dekodiert
      - evtl. werden noch die für den Befehl notwendigen Operanden gelesen
    - Execute (Ausführungsphase)
      - Rechenwerk führt den dekodierten Befehl mit den Operanden aus
      - kann bei komplexen Befehlen auch aus mehreren Phasen bestehen
    - Write (Speicherphase)
      - Ergebnis wird gespeichert (in den Hauptspeicher, in ein internes Register oder bei einem Sprungbefehl in den Program Counter)

Befehlsphasen (2)

- **Befehl nach Befehl wird abgearbeitet**
  - z.B. könnte jede Phase genau einen Takt benötigen
  - Zykluszeit (Periodendauer des Taktsignals)
    - hängt von der Verarbeitungszeit (Durchlaufzeit, Verzögerungszeit) der längsten Befehlsphase ab
  - man kann auch Prozessoren bauen, die insgesamt nur einen Takt pro Befehl benötigen

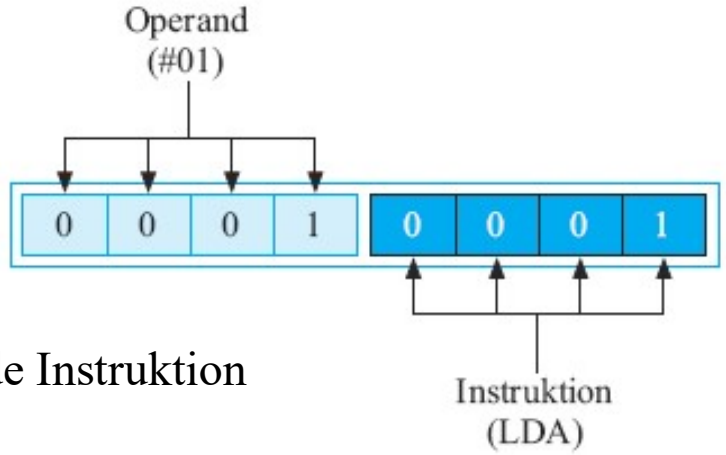


Einfacher Modellprozessor

- **Idee**
  - Konstruktion eines voll funktionsfähigen Mikroprozessors
  - extrem einfach gehalten
  - in der Leistung nicht vergleichbar mit heutigen kommerziellen Prozessoren
- **Sehr starke Vereinfachungen**
  - Registerbreite: 4 Bit
    - größere Bitbreiten ohne Änderung des Konzepts möglich
    - moderne Prozessoren haben 32 oder 64 Bit
    - preiswerte, eingebettete Prozessoren (Fahrstühle, Kfz-Steuerungen, …) haben auch schon mal nur 4 oder 8 Bit
  - Adressbus: 4 Bit
    - Busbreite gleich Registerbreite (Adressen müssen nicht gestückelt werden)
    - bei 64-Bit Prozessoren kein Problem (32 Bit entspricht 4 GiB)
    - bei 4 Bit können leider nur 16 Adressen benutzt werden

Einfacher Modellprozessor (2)

- **Vereinfachungen (Fortsetzung)**
  - Maschinenbefehle
    - jeder Befehl hat nur einen Operanden
      - Befehl besteht aus 8 Bit
      - Operand: höherwertige 4 Bit
      - Opcode: niederwertige 4 Bit
        - » die eigentliche auszuführende Instruktion

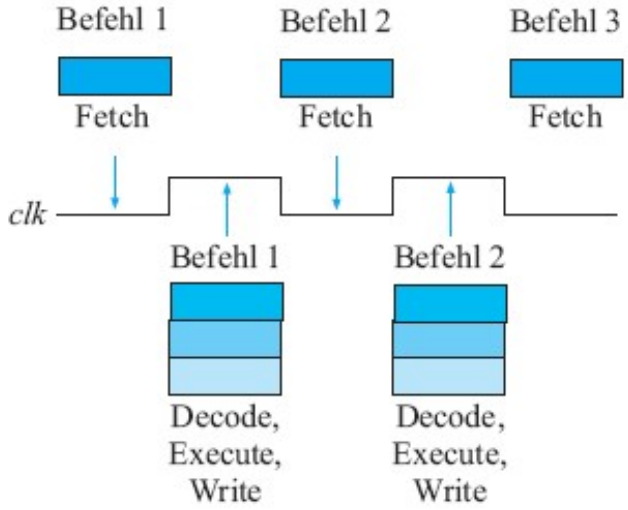


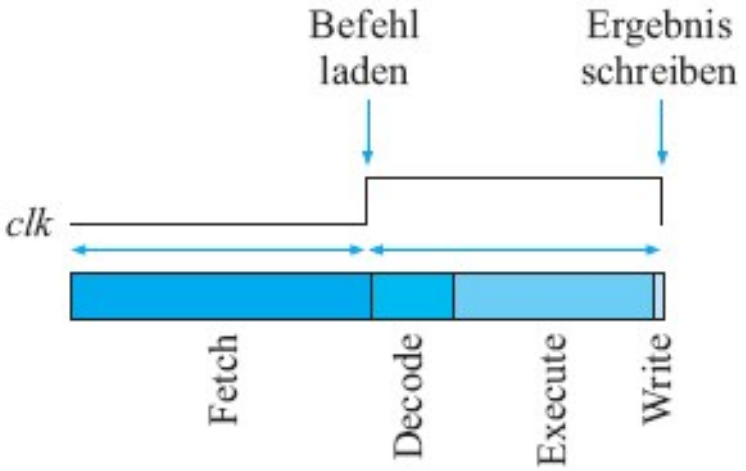
  - Daten
    - sind nur 4 Bit breit (s.o. Registerbreite)
      - daher ist der Datenspeicher anders organisiert als der Befehlsspeicher
      - wir haben also streng genommen eine Harvard-Architektur (ohne aber gleichzeitig und unabhängig auf die beiden Speicher zugreifen zu können)
  - Eingabe/Ausgabe
    - es wird hier vollständig darauf verzichtet (Extra-Hardware notwendig)
    - Eingaben müssen bereits im Speicher stehen, Ausgaben werden dorthin geschrieben

Einfacher Modellprozessor (3)

- **Vereinfachungen (Fortsetzung)**
  - Befehlsausführung in nur einem Takt *(eigentlich sind es zwei Takte, s.u.)*
    - Fetch
      - erste Takthälfte *(bzw. erster Takt)*
    - Decode, Execute, Write
      - zweite Takthälfte *(bzw. zweiter Takt)*
  - zwei Zweitpunkte, zu denen Register beschrieben werden
    - nur ein Takt, weil beide Taktflanken ausgenutzt werden

konkreter zeitlicher Ablauf





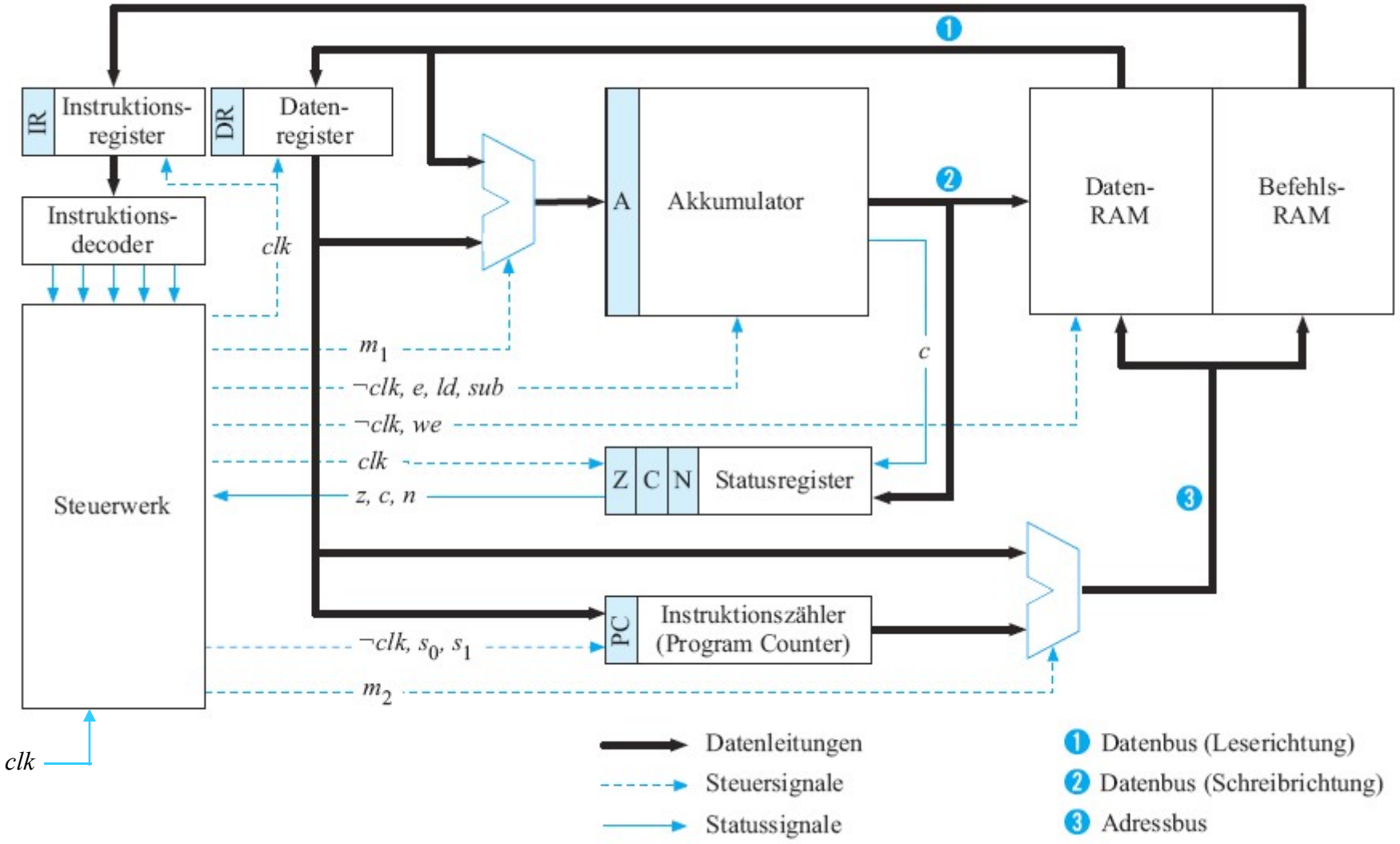
Einfacher Modellprozessor (4)

- Rechenwerk (Datenpfad)
  - besteht hier nur aus einem einzigen Akkumulator
  - nur Addition und Subtraktion unterstützt
  - komplexere ALU würde die Fähigkeiten erweitern ohne etwas an der Struktur des Prozessors zu ändern
- RAM (Hauptspeicher)
  - Befehlsspeicher wird mit nur 4 Bit adressiert (s.o.)
  - ein Befehl passt in jedes Byte
  - also nur magere 16 Befehle speicherbar (nur das Prinzip soll hier dargestellt werden)
- Befehlssatz
  - 4 Bit Opcode (Operation Code) ermöglicht nur maximal 16 verschiedene Operationen
    - es werden sogar nur 12 benutzt
    - NOP (no operation) ist ein Befehl, der nichts bewirkt (außer das Weiterzählen des program counters)

Vollständiger Befehlssatz

Nr	Befehl	Codierung				Beschreibung
0	NOP	0	0	0	0	Wartezyklus (*No Operation*)
						Lade- und Speicherbefehle
1	LDA #n	0	0	0	1	Lädt den Akkumulator mit dem Wert *n*
2	LDA (n)	0	0	1	0	Lädt den Akkumulator mit dem Inhalt der Speicherstelle *n*
3	STA n	0	0	1	1	Überträgt den Akkumulatorinhalt in die Speicherstelle *n*
						Arithmetikbefehle
4	ADD #n	0	1	0	0	Erhöht den Akkumulatorinhalt um den Wert *n*
5	ADD (n)	0	1	0	1	Erhöht den Akkumulatorinhalt um den Inhalt der Speicherstelle *n*
6	SUB #n	0	1	1	0	Erniedrigt den Akkumulatorinhalt um den Wert *n*
7	SUB (n)	0	1	1	1	Erniedrigt den Akkumulatorinhalt um den Inhalt der Speicherstelle *n*
						Sprungbefehle
8	JMP n	1	0	0	0	Lädt den Instruktionszähler mit dem Wert *n*
9	BRZ #n	1	0	0	1	Addiert *n* auf den Instruktionszähler, falls das Zero-Bit gesetzt ist
10	BRC #n	1	0	1	0	Addiert *n* auf den Instruktionszähler, falls das Carry-Bit gesetzt ist
12	BRN #n	1	0	1	1	Addiert *n* auf den Instruktionszähler, falls das Negations-Bit gesetzt ist

Aufbau Modellprozessor



Aufbau Modellprozessor (2)

- **Instruktionszähler**
  - Steuersignale s_1 und s_0 (wie weiter oben beschrieben)
    - 00: Offset addieren, relativer Sprung
    - 01: Laden, absoluter Sprung
    - 10: Inkrementieren, nächster Befehl
    - 11: Reset, Initialisierung
  - Ausführung mit negativer Taktflanke
    - am Ende der Write-Phase, bzw. zu Beginn der Fetch-Phase
- **RAM**
  - Speicher ist in zwei Hälften geteilt
    - obere 4 Bit: Daten-RAM
    - untere 4 Bit: Befehls-RAM (für Opcode, Operand in oberen 4 Bit))
  - kein bidirektionaler Bus, sondern getrennte Lese- und Schreibleitungen
  - asynchrones Lesen
    - sobald die Adresse anliegt erscheinen die Daten auf dem Lese-Datenbus
  - synchrones Schreiben
    - wenn `we` (write enable) aktiv ist, werden die Daten mit der negativen Taktflanke am Ende der Write-Phase übernommen

Aufbau Modellprozessor (3)

- **Instruktionsregister (IR) und Datenregister (DR)**
  - übernehmen jeweils 4 Bit des adressierten Bytes mit der positiven Taktflanke am Ende der Fetch-Phase
- **Instruktionsdecoder**
  - dekodiert den Inhalt von IR
    - für jeden Maschinenbefehl wird eine andere Leitung aktiv
  - einfaches Schaltnetz
- **Statusregister**
  - Laden mit positiver Taktflanke am Anfang der Decode-Phase
    - Übernahme der Statusbits vom aktuellen Inhalt des Akkumulators (s.u.)

Aufbau Modellprozessor (4)

- **Steuerwerk**
  - hier ebenfalls ein einfaches Schaltnetz
    - da alle Befehle in einem Takt ausgeführt werden
    - das `clk`-Signal wird hier allerdings selbst als Eingang verwendet, um die beiden Taktphasen unterscheiden zu können
      - es müssen unterschiedliche Steuersignale in den beiden Phasen erzeugt werden
  - Eingänge: Instruktion (ext. Steuersignale), Statusbits von Statusregister
  - Ausgänge: sämtliche Steuersignale für den Datenpfad
- **Akkumulator**
  - kann Inhalt erhalten, neuen Inhalt laden, addieren oder subtrahieren
  - `e` (Enable), `ld` (Load), `sub` (Sub statt Add) legen die Funktion fest
  - liefert zusätzlich zu den 4 gespeicherten Bits ein Carrybit (`c`) bei Addition oder Subtraktion
  - Ergebnisübernahme im Register mit der negativen Taktflanke am Ende der Write-Phase

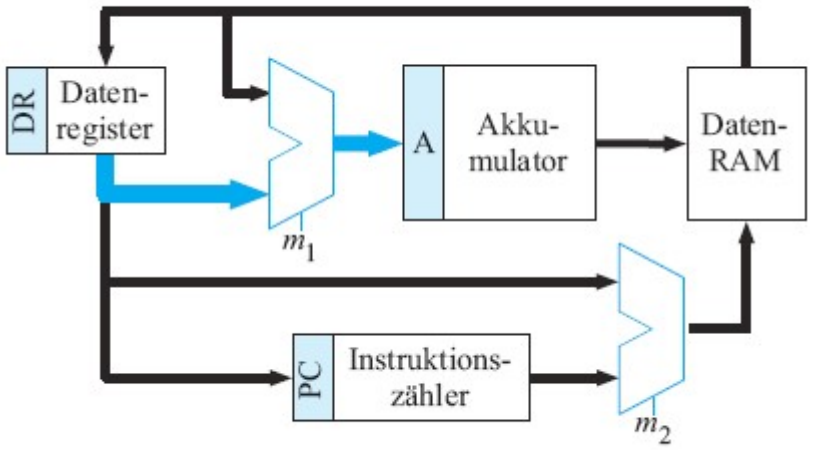
Aufbau Modellprozessor (5)

- **Datenpfade**
  - zwei Multiplexer
    - Eingang des Akkumulators
      - $m_1 = 0$: Datenbus (für Befehle mit Speicheroperanden)
      - $m_1 = 1$: Datenregister (DR, für Befehle mit immediate Operanden)
    - Adresseingang des Speichers
      - $m_2 = 0$: Datenregister (DR, für Laden oder Speichern von Daten)
      - $m_2 = 1$: Program Counter (PC, zum Laden von Befehlen)

Datenfluss für verschiedene Befehlsarten

- **LDA #n, ADD #n, SUB #n**
  - *immediate address* (unmittelbare Adressierung)
    - der Operand n ist selbst Teil des Befehls (hier steht also gar keine Adresse, sondern der Operand selbst)
    - der Operand steht daher nach dem Holen des Befehls schon im Datenregister DR
    - das RAM wird (nach dem Holen des Befehls) nicht mehr benötigt

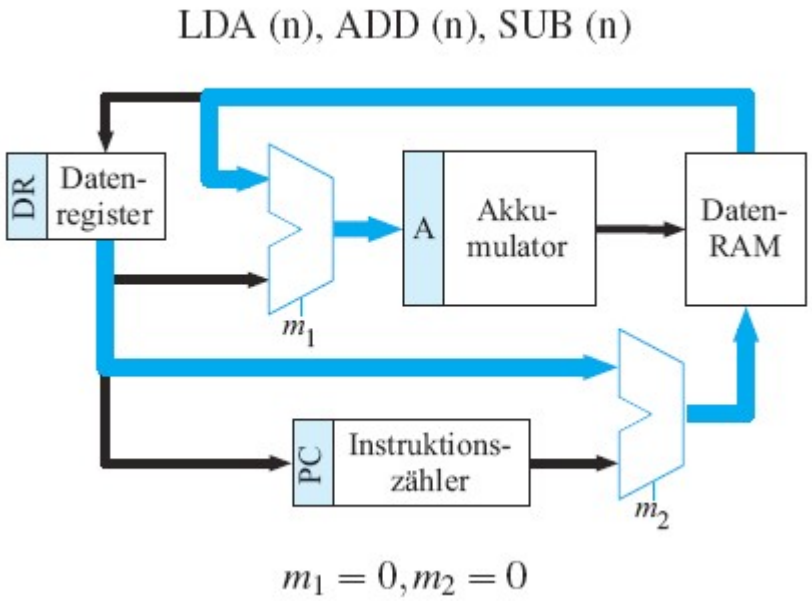
LDA #n, ADD #n, SUB #n



$m_1 = 1, m_2 = -$

Datenfluss für verschiedene Befehlsarten (2)

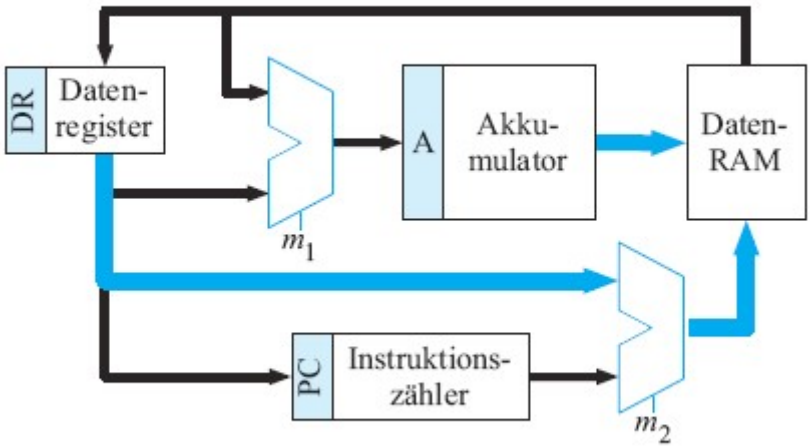
- **LDA (n), ADD (n), SUB (n)**
  - *direct address* (direkte Adressierung, im Buch steht indirekte Addressierung, die gibt es jedoch bei diesem Prozessor gar nicht)
    - die Zahl n (aus dem Maschinenbefehl in DR) bezeichnet die Hauptspeicheradresse, in der sich der Operand befindet
    - zweiter Zugriff auf den Hauptspeicher in der zweiten, positiven Taktphase, um den Operanden zu holen



Datenfluss für verschiedene Befehlsarten (3)

- **STA n**
  - Übertragen des Akkuinhaltes in den Hauptspeicher an Adresse n
    - die Zahl n (aus dem Maschinenbefehl in DR) bezeichnet die Hauptspeicheradresse, in die der Akkumulatorinhalt kopiert werden soll
    - daher wäre STA (n) die bessere Bezeichnung (ebenfalls direkte Adressierung)

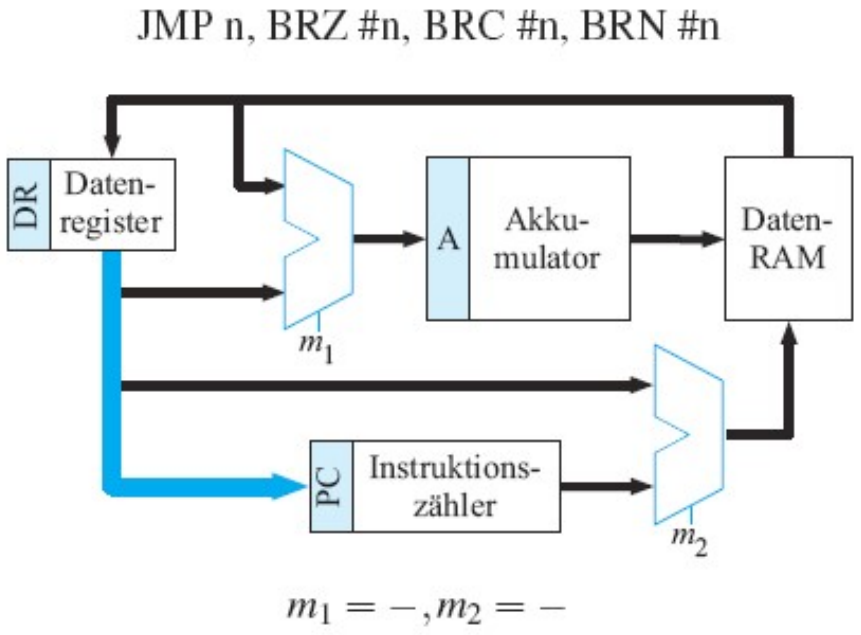
STA n



$m_1 = -, m_2 = 0$

Datenfluss für verschiedene Befehlsarten (4)

- **JMP n, BRZ #n, BRC #n, BRN #n**
  - direkter Sprung (*jump*) JMP n
    - die Zieladresse ist selbst Teil des Sprungbefehls
    - Program Counter wird mit der Adresse in DR überschrieben
  - bedingte, relative Sprünge (*branches*) BRZ #n, BRC #n, BRN #n
    - der Wert aus DR wird zur momentanen Adresse im PC hinzuaddiert, falls das Bit Z, C oder N im Statusregister gesetzt ist
    - positiver Wert in DR
      - Sprung nach vorne
    - negativer Wert in DR
      - Sprung nach hinten



Beispiel: Multiplikation

- **Annahme**
  - Speicherzellen 13 und 14 enthalten die Operanden
  - das Produkt soll in Speicherzelle 15 entstehen
- **Algorithmus**
  - der Multiplikand in 14 wird so oft zum Ergebnis in 15 (=0 am Anfang) addiert, wie der Multiplikator in 13 angibt
  - dazu wird in jedem Durchgang der Multiplikator in 13 dekrementiert
  - END ist ein MACRO, das man z.B. durch JMP 9 ersetzen könnte, um das Programm in einer Endlosschleife effektiv zu beenden

```
// Beispielprogramm
// zur Berechnung der
// Multiplikation
//
// Eingabe:
// (13) = Multiplikator
// (14) = Multiplikand
//
// Ausgabe:
// (15) = (13) * (14)

0: init:  LDA #0
1: begin: STA (15)
2:        LDA (13)
3:        BRZ #6    // end:
4:        SUB #1
5:        STA (13)
6:        LDA (15)
7:        ADD (14)
8:        JMP 1     // begin:
9: end:   END
```

Beispiel: Multiplikation (2)

Zyklus	*clk*	PC	Adressbus	Datenbus	(IR)	(DR)	(A)	(13)	(14)	(15)	
01	↑	00	~~01~~	LDA #0	LDA	00	??	02	03	??	
	↓	01	??	??	LDA	00	00	02	03	??	
02	↑	01	~~02~~	STA (15)	STA	15	00	02	03	??	
	↓	02	15	0	STA	15	00	02	03	00	
03	↑	02	~~03~~	LDA (13)	LDA	13	00	02	03	00	
	↓	03	13	02	LDA	13	02	02	03	00	
04	↑	03	~~04~~	BRZ ~~09~~	BRZ	~~09~~	02	02	03	00	06
	↓	04	??	??	BRZ	~~09~~	02	02	03	00	
05	↑	04	~~05~~	SUB #1	SUB	01	02	02	03	00	
	↓	05	??	??	SUB	01	01	02	03	00	
06	↑	05	~~06~~	STA (13)	STA	13	01	02	03	00	
	↓	06	13	01	STA	13	01	01	03	00	
07	↑	06	~~07~~	LDA (15)	LDA	15	01	01	03	00	
	↓	07	15	00	LDA	15	00	01	03	00	
08	↑	07	~~08~~	ADD (14)	ADD	14	00	01	03	00	
	↓	08	14	03	ADD	14	03	01	03	00	
09	↑	08	~~09~~	JMP 01	JMP	01	03	01	03	00	
	↓	01	??	??	JMP	01	03	01	03	00	
10	↑	01	~~02~~	STA (15)	STA	15	03	01	03	00	
	↓	02	15	03	STA	15	~~00~~	01	03	03	03
11	↑	02	~~03~~	LDA (13)	LDA	13	~~00~~	01	03	03	
	↓	03	13	01	LDA	13	01	01	03	03	

Adressbus = PC

vor Taktflanke

nach Taktflanke

Beispiel: Multiplikation (3)

09	↑	08	09	JMP 01	JMP	01	03	01	03	00
	↓	01	??	??	JMP	01	03	01	03	00
10	↑	01	02	STA (15)	STA	15	03	01	03	00
	↓	02	15	03	STA	15	00	01	03	03
11	↑	02	03	LDA (13)	LDA	13	00	01	03	03
	↓	03	13	01	LDA	13	01	01	03	03
12	↑	03	04	BRZ 09	BRZ	09	01	01	03	03
	↓	04	??	??	BRZ	09	01	01	03	03
13	↑	04	05	SUB #1	SUB	01	01	01	03	03
	↓	05	??	??	SUB	01	00	01	03	03
14	↑	05	06	STA (13)	STA	13	00	01	03	03
	↓	06	13	00	STA	13	00	00	03	03
15	↑	06	07	LDA (15)	LDA	15	00	00	03	03
	↓	07	15	03	LDA	15	03	00	03	03
16	↑	07	08	ADD (14)	ADD	14	03	00	03	03
	↓	08	14	03	ADD	14	06	00	03	03
17	↑	08	09	JMP 01	JMP	01	06	00	03	03
	↓	01	??	??	JMP	01	06	00	03	03
18	↑	01	02	STA (15)	STA	15	06	00	03	03
	↓	02	15	06	STA	15	00	00	03	06
19	↑	02	03	LDA (13)	LDA	13	00	00	03	06
	↓	03	13	00	LDA	13	00	00	03	06
20	↑	03	04	BRZ 09	BRZ	09	00	00	03	06
	↓	09	??	??	BRZ	09	00	00	03	06
21	↑	09	04	END	JMP	09	00	00	03	06
	↓	09	??	??	JMP	09	00	00	03	06

Fehler nicht korrigiert

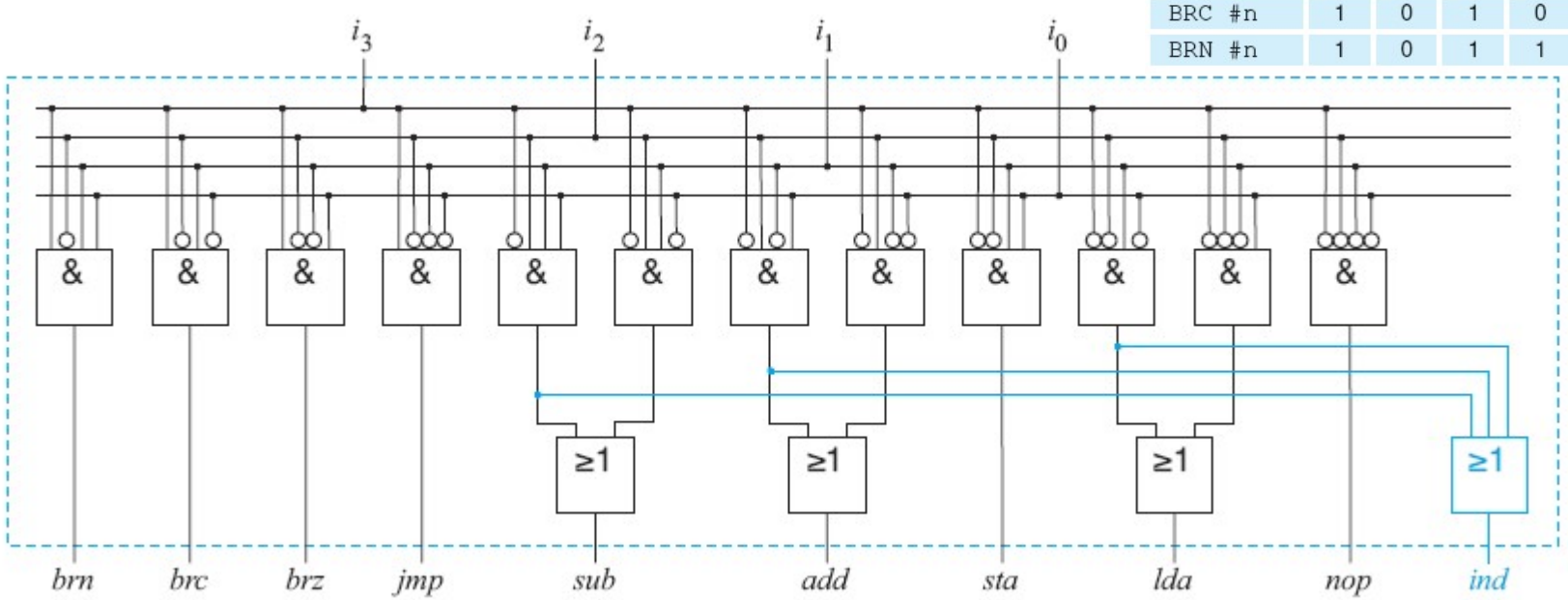
Bemerkungen zur Komplexität

- **Berechenbarkeitstheorie**
  - wichtiges Teilgebiet aus der Theoretischen Informatik
    - siehe: Turing-Maschine und Church'sche These
  - man kann zeigen, dass unser Modellprozessor prinzipiell jede berechenbare Funktion berechnen kann
    - dazu müsste nur der Hauptspeicher genügend groß sein
  - jeder denkbare Algorithmus, in einer beliebigen Programmiersprache formuliert, kann in ein Maschinenprogramm für unseren Modellprozessor übersetzt werden
    - die Theorie sagt allerdings nichts darüber aus, wie aufwändig und wie effizient das resultierende Programm ist
    - wir haben gesehen, dass selbst einfache Algorithmen nur sehr aufwändig zu realisieren sind

Implementierung Instruktionsdekoder

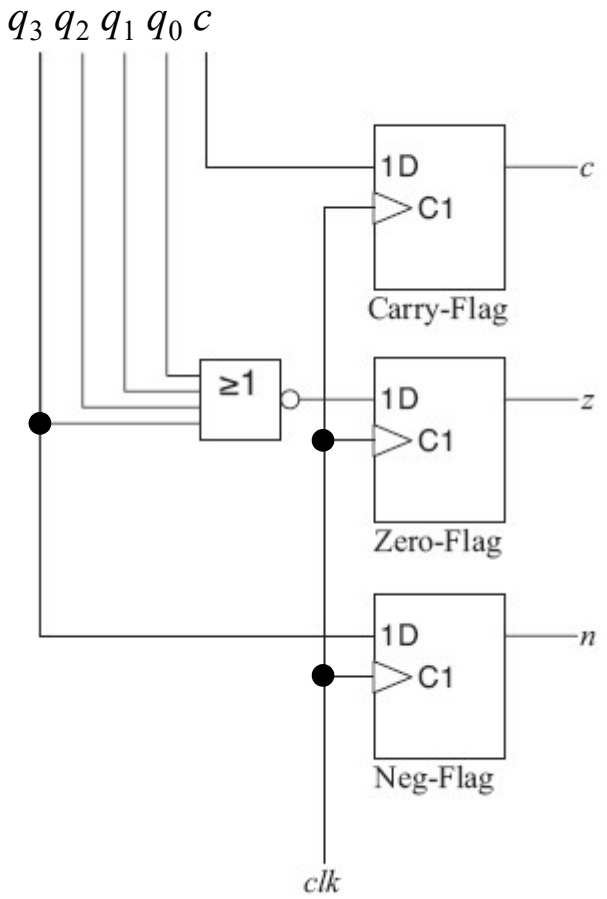
- **Schaltnetz**
  - LDA, ADD, SUB haben je zwei verschiedene Kodierungen
  - `ind` signalisiert indirekte Adressierung (sollte besser direkte Adressierung heißen)

Befehl	Codierung			
NOP	0	0	0	0
LDA #n	0	0	0	1
LDA (n)	0	0	1	0
STA n	0	0	1	1
ADD #n	0	1	0	0
ADD (n)	0	1	0	1
SUB #n	0	1	1	0
SUB (n)	0	1	1	1
JMP n	1	0	0	0
BRZ #n	1	0	0	1
BRC #n	1	0	1	0
BRN #n	1	0	1	1



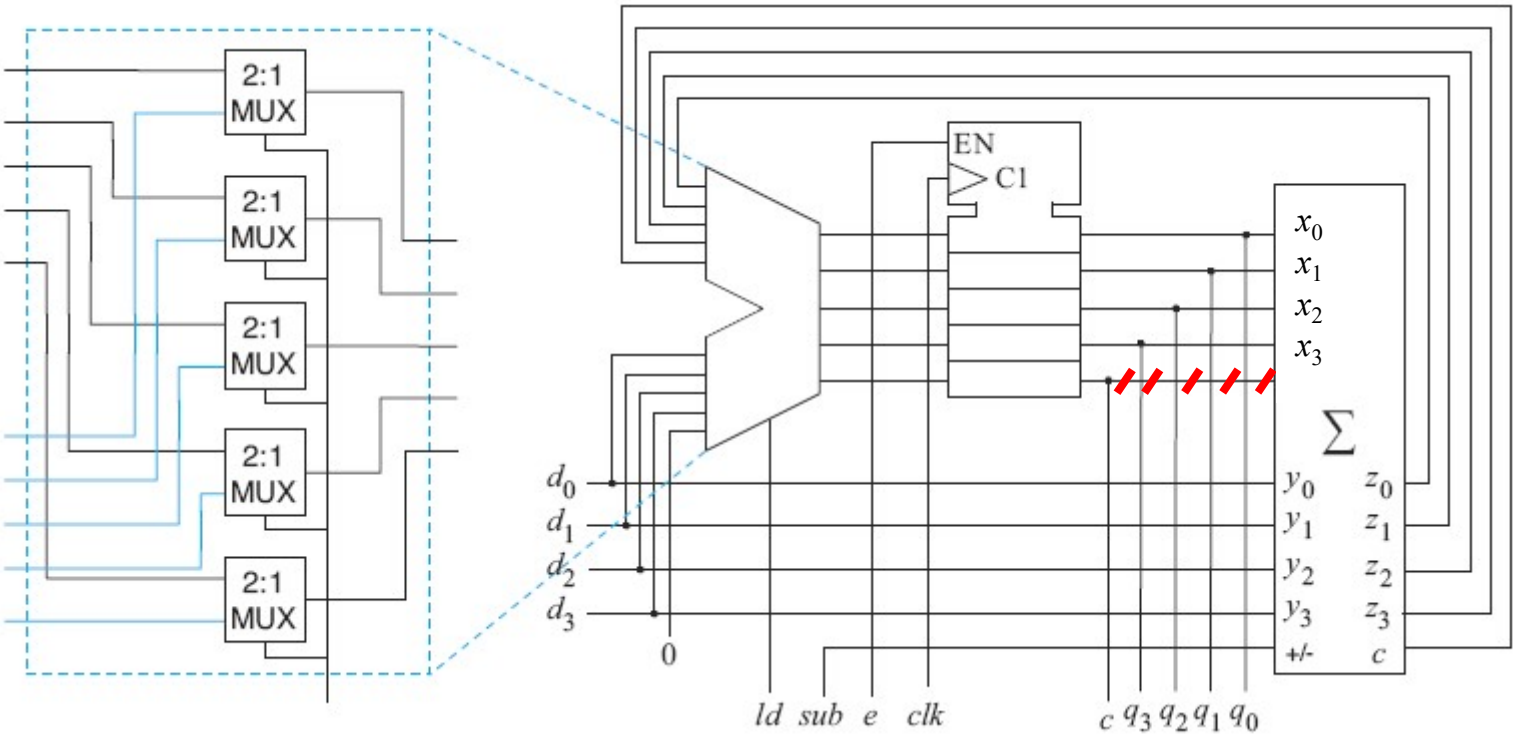
Implementierung Statusregister

- ist positiv flankengetriggert
- reagiert in der Mitte des Taktes, am Anfang der Dekodierphase



Implementierung Akkumulator

- **Aufbau im Wesentlichen wie bereits erläutert**
  - Carry-Bit wird zusammen mit den Daten abgespeichert, damit das Statusregister einen halben Takt später darauf zugreifen kann
  - `clk`-Eingang wird mit ¬`clk` verbunden, damit das Register am Ende der positiven Taktphase reagiert



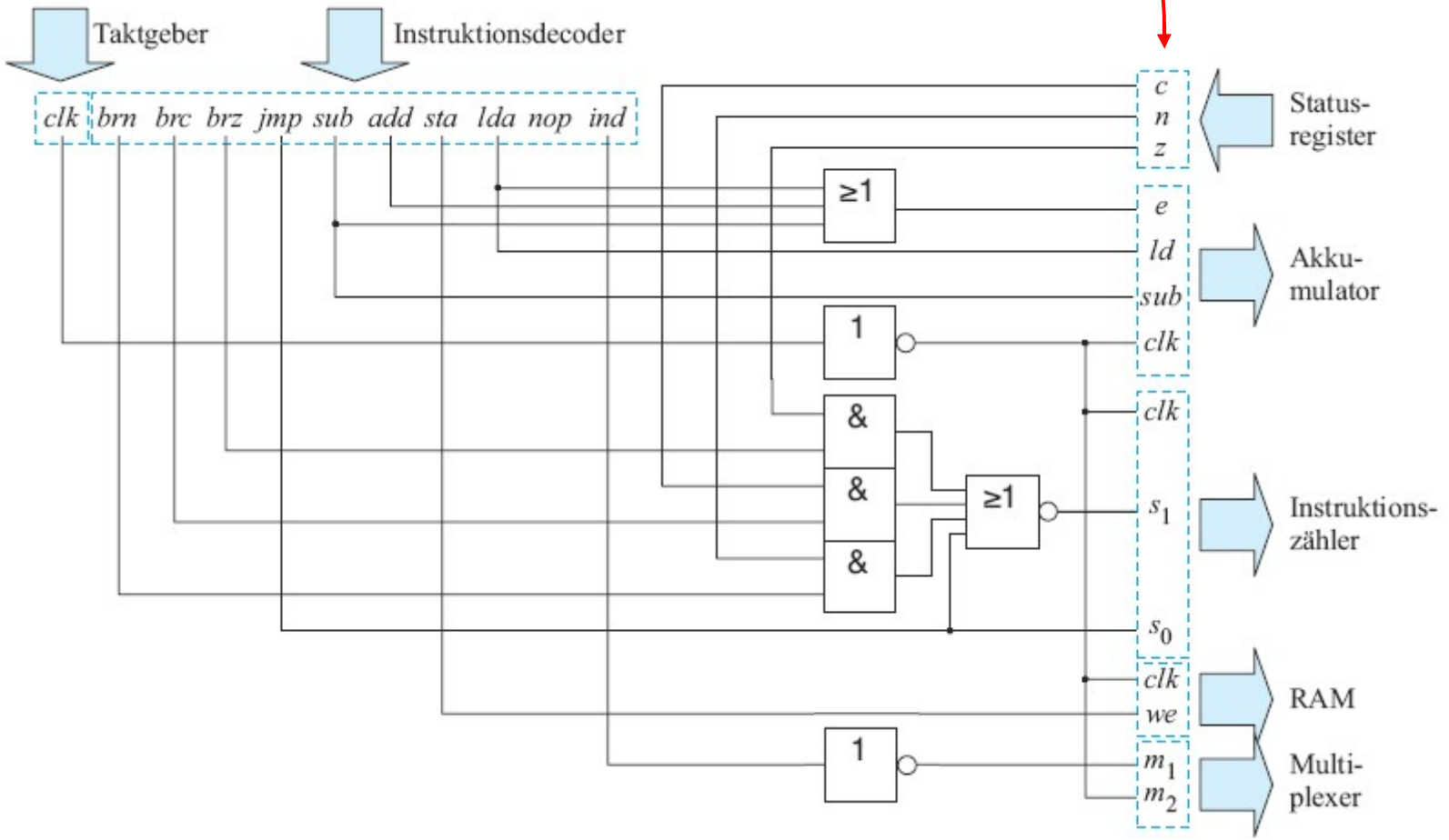
Implementierung Steuerwerk

- **für *clk* = 0 ist nur m_2=1 relevant**
  - Akumulator, PC und RAM reagieren nicht auf steigende Taktflanke

	Statusvariablen			Akkumulator			PC		RAM	Multiplexer	
Befehl	*z*	*c*	*n*	*e*	*ld*	*sub*	s_1	s_0	*we*	m_1	m_2
Ladephase (Fetch): $clk = 0$											
–	–	–	–	–	–	–	–	–	–	–	1
Ausführungsphase (Decode + Execute + Write): $clk = 1$											
NOP	–	–	–	0	–	–	1	0	0	–	–
LDA	–	–	–	1	1	–	1	0	0	¬*ind*	0
STA	–	–	–	0	–	–	1	0	1	–	0
ADD	–	–	–	1	0	0	1	0	0	¬*ind*	0
SUB	–	–	–	1	0	1	1	0	0	¬*ind*	0
JMP	–	–	–	0	–	–	0	1	0	–	–
BRZ	0	–	–	0	–	–	1	0	0	–	–
BRZ	1	–	–	0	–	–	0	0	0	–	–
BRC	–	0	–	0	–	–	1	0	0	–	–
BRC	–	1	–	0	–	–	0	0	0	–	–
BRN	–	–	0	0	–	–	1	0	0	–	–
BRN	–	–	1	0	–	–	0	0	0	–	–

Implementierung Steuerwerk (2)

- **Schaltbild (gegenüber Buch korrigiert)**
  - alle `clk` auf der rechten Seite sind `¬clk`



Bemerkungen

- **Einfachheit**
  - bedingt durch extrem einfachen Befehlssatz
- **nur Akkumulator**
  - normalerweise haben Mikroprozessoren mehrere Register und eine von den Registern getrennte ALU
- **ALU**
  - ist manchmal auch für Adressberechnungen zuständig
    - hier gibt es einen eigenen Addierer im Program Counter

Bemerkungen (2)

- **Ausnutzung der zwei Taktphasen**
  - macht das Verständnis der Architektur nur schwieriger
  - man kann sich den Ablauf auch mit einem normalen Taktsignal doppelt so hoher Frequenz klarmachen, dann gilt:
    - nur positive Taktflanken sind relevant
    - Bearbeitung eines Befehls benötigt zwei Takte
    - ein Zustandsbit im Steuerwerk muss sich merken, in welcher Bearbeitungsphase man ist (z.B. Toggle-Flipflop)