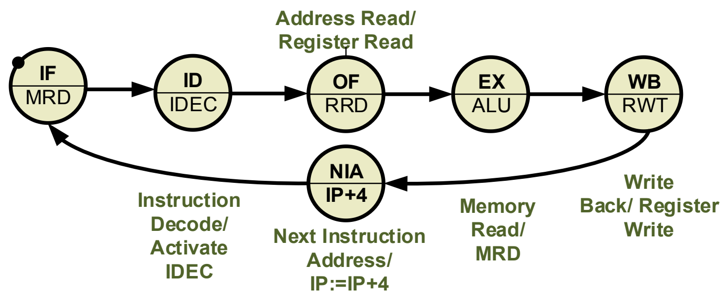
Grundlagen der Rechnerarchitektur

Wolfgang Fengler, Olga Fengler



Impressum

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Angaben sind im Internet über http://dnb.d-nb.de abrufbar.

© 2016, Technische Universität Ilmenau/Universitätsbibliothek
ilmedia

Postfach 10 05 65
98684 Ilmenau
www.tu-ilmenau.de/ilmedia

Titelbilder:

Microcombi: der (vermutlich) erste Prototyp der DDR mit PC-Architektur (Technische Hochschule Ilmenau, 1980)
© Klaus Dieter Fritz Ilmenau

MC 80-30: Zweites Serienprodukt auf Basis des Microcombi (VEB Elektronik Gera, Technische Hochschule Ilmenau, 1985)
© Wolfgang Fengler Ilmenau

URN: urn:nbn:de:gbv:ilm1-2016200264

Vorwort

Das Lehrbuch (E-Book) soll als Basis von Vorlesungen an Universitäten und Fachhochschulen zu Grundlagen von Rechnerarchitekturen dienen.

Besonderheit der hier vorliegenden Darstellungen ist, dass zur Erläuterung der Funktion an vielen Stellen Systeme paralleler Automaten zur weitgehend formalen Beschreibung verwendet werden. Weiterhin wird außer den Abschnitten zum Prozessor und Speicher großer Wert auf die Behandlung von Ein-Ausgabe-Prinzipien gelegt. Insgesamt soll für Studierende der Ingenieurwissenschaften die Grundlage zu aufbauenden Veranstaltungen (z.B. zu Microcontrollern) gelegt werden. Das dient zu einer eventuellen späteren Berufsorientierung bei der Anwendung von geräteintegrierten (Eingebetteten) Rechnern in vielen Domänen (z.B. Fahrzeugtechnik, Medizintechnik, Mobilkommunikation, usw.), als Auftraggeber oder Partner der Entwickler aus der Disziplin Technische Informatik/ Ingenieurinformatik bzw. als selbständiger Entwickler. Dazu wurden eigene Erfahrungen bei der Entwicklung von Eingebetteten Rechnern in Kooperation mit Ingenieurteams herangezogen.

Der Inhalt ist zugeschnitten auf eine Veranstaltung mit zwei Unterrichtsstunden pro Woche im Semester. Mit ausgewählten Teilen (Grundwissen) kann auch eine Vorlesung mit der Hälfte der Termine durchgeführt werden.

Der vorliegende Inhalt setzt eine vorherige Veranstaltung zu den Grundlagen digitaler Systeme voraus (vor allem: Boolesche Algebra, kombinatorische und sequentielle Logik/endliche Automaten und deren Beschreibung mit Automatengraphen, logische Grundfunktionen und -elemente und digitale Zahlendarstellungen (alles z.B. in [1] oder [2] bzw. einer entsprechenden Vorlesung [3] in den notwendigen Umfängen).

Der Inhalt des Buchs wurde gezielt beschränkt auf den dazu gehörigen Vorlesungsinhalt und basiert auf einer langjährigen Erfahrung in Lehrveranstaltungen zum Thema für Ingenieure, Ingenieurinformatiker und Informatiker. Deshalb ist auch ein Teil der enthaltenen Bilder auf Grundlage der dort verwendeten Lehrmaterialien [4] entstanden, wobei diese aber für das vorliegende Buch weitgehend überarbeitet wurden. Wir danken allen, die an diesem Material mitgearbeitet haben.

Das gewählte Layout (Querformat, große Schrift) soll es ermöglichen, den Inhalt auf einem Notebook als Vollbild so darzustellen, dass eine Arbeit damit auch in der Lehrveranstaltung möglich ist. Falls ein Ausdruck gewünscht ist, sollte er mit 2 Seiten je A4-Blatt erfolgen.

Ilmenau, den 1. Dezember 2016

Wolfgang Fengler Olga Fengler

Inhalt

1. Einführung 9
2. Parallele Automaten zur Architekturmodellierung 15
   2.1. Grundlagen 15
   2.2. Beispiele 20
3. Architekturgrundlagen 32
4. Befehlsübersicht 43
   4.1. Übersicht und Grundlagen 44
   4.2. Transportbefehle 51
   4.3. Arithmetik-Logik-Befehle 62
   4.4. Schiebe- und Rotationsbefehle 73
   4.5. Programmtransferbefehle 77
   4.6. Sonstige Befehle 91
5. Prozessor und prozessorzugeordnete Baugruppen 93
   5.1. Einordnung in die Gesamtarchitektur 93
   5.2. Prozessorgrundstruktur 99
   5.3. Befehlsabarbeitung im Prozessor 104
      5.3.1. Arithmetik-Logik-Befehle 105

5.3.2. Speicherbefehle 107
5.3.3. Sprungbefehle 110
5.3.4. Unterprogrammbefehle...112
5.3.5. Gesamtverhalten der Befehle in der Ablaufsteuerung 115
5.4. Interrupt 116
5.5. Prozessorzugriffe über externes Interface 130
6. Speicher 139
6.1. Speicherbit und Speicherwort...140
6.2. Speicher-IC...152
6.3. Speicherfunktionseinheit...155
7. Ein- und Ausgabe 162
7.1. Parallele digitale Ein-/Ausgabe 166
7.2. Programmierbare E/A mit Ein-/Ausgabe-Controller 176
7.3. Programmierbare E/A mit Steuerregistern 178
7.4. Synchronisierte parallele digitale Ein-/Ausgabe..181
7.5. Serielle digitale Ein-/Ausgabe 186
7.6. Zähler-Zeitgeber 199
7.7. Analog-Ein-/Ausgabe...205

8. Weiterführende Konzepte der Rechnerarchitektur....212
8.1. Prozessorleistung....212
8.2. Parallelisierung im Prozessor....215
8.2.1. Befehlspipelining....215
8.2.2. Superskalare Prozessorarchitektur....219
8.2.3. Very-Long-Instruction-Word-Prozessorarchitektur....221
8.2.4. Out-of-Order-Prozessorarchitektur....223
8.2.5. Simultanious-Multi-Threading-Prozessorarchitektur....224
8.2.6. Single-Instruction-Multiple-Data-Prozessorarchitektur....226
8.3. Parallelität von Prozessoren....227
8.4. Leistungssteigerung bei Speicherzugriffen....232
8.4.1. Cache-Speicher....234
8.4.2. Mikroparallelität in der Speicherarchitektur....237
Literaturverzeichnis....239
Abbildungsverzeichnis....242

1. Einführung

Der Abschnitt 1 dient zur Einordnung des Fachs in die ingenieurtechnische Umwelt und zur Motivation von Studierenden der Ingenieurwissenschaften, sich damit zu beschäftigen. Ziel ist unter anderem die Einordnung in eine mögliche spätere berufliche Tätigkeit.

Rechnerarchitektur:

- Struktur eines Rechners: Funktionsblöcke und Verbindungen, deren Semantik und
- deren Zusammenwirken beim Abarbeiten von Programmen innerhalb des Rechners, der in einer Umgebung agiert.

Ziel des Faches „Grundlagen der Rechnerarchitektur“:

Vermittlung von

- Grundlagen des o.g. Gebiets als Voraussetzung für die Anwendung, Entwicklung und/oder Mitarbeit bei der Entwicklung von digitalen Informationsverarbeitungslösungen in ingenieurtechnischen Systemen und Geräten

- und zu weiterführenden Entwicklungen, die darauf aufbauen, um sie zu verstehen und einzuordnen.

Vermittlung eines gemeinsamen Kenntnisfelds und Sprachgebrauchs zwischen Rechnerentwicklern, Rechneranwendern und Mitentwicklern aus technischer Anwendungssicht.

Dafür wird als Nebenziel notwendig: Grundverständnis für die Abarbeitung von Programmen.

Die behandelten wesentlichen 4 Hauptschwerpunkte sind:

- Prozessor,
- Speicher,
- Ein-Ausgabe und
- moderne Entwicklungen.

Warum sind diese Kenntnisse wichtig für Studierende der Ingenieurwissenschaften?

Im Berufsleben gibt es in vielen Fällen Anknüpfungspunkte (nicht erschöpfend) bei:

- Anwendung von geräteintegrierten (Eingebetteten) Rechnern in vielen Domänen (z.B. Fahrzeugtechnik, Medizintechnik, Mobilkommunikation, usw.), als Auftraggeber, Partner der Entwickler aus der Disziplin Technische Informatik/ Ingenieurinformatik bzw. als

selbständiger Entwickler. Die Größenordnung der weltweit eingesetzten Eingebetteten Rechner ist sehr viel höher als die der Rechner der PC/Notebook-Domäne.

- Entwicklung von peripheren Komponenten (z.B. Festplatten),
- technologische Entwicklung von Realisierungselementen (elektronisch, IC's, Boards, …),
- Echtzeitproblematik.

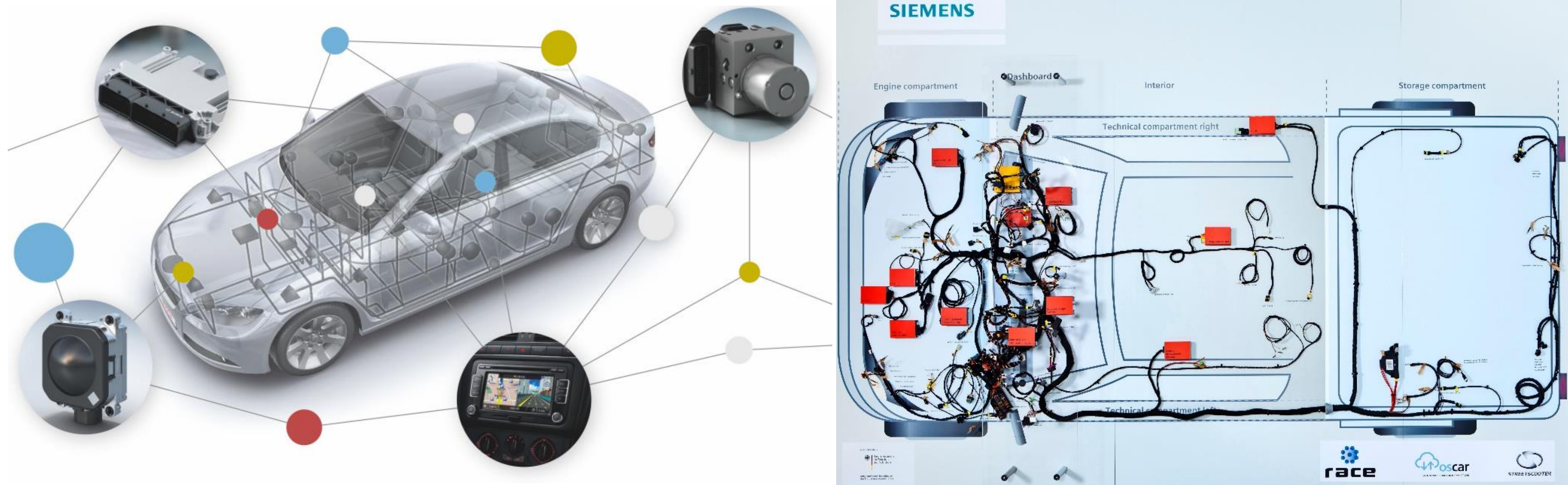


Bild 1.1: Steuergeräte im PKW [5], [6]

Anschauliches Beispiel:
Motorsteuergerät (Einordnung siehe Bild 1.1 S.11) als
Eingebetteter Rechner (siehe Bild 1.2 S.12) zur Steuerung eines Verbrennungsmotors (z.B. eines PKW), bestehend aus:

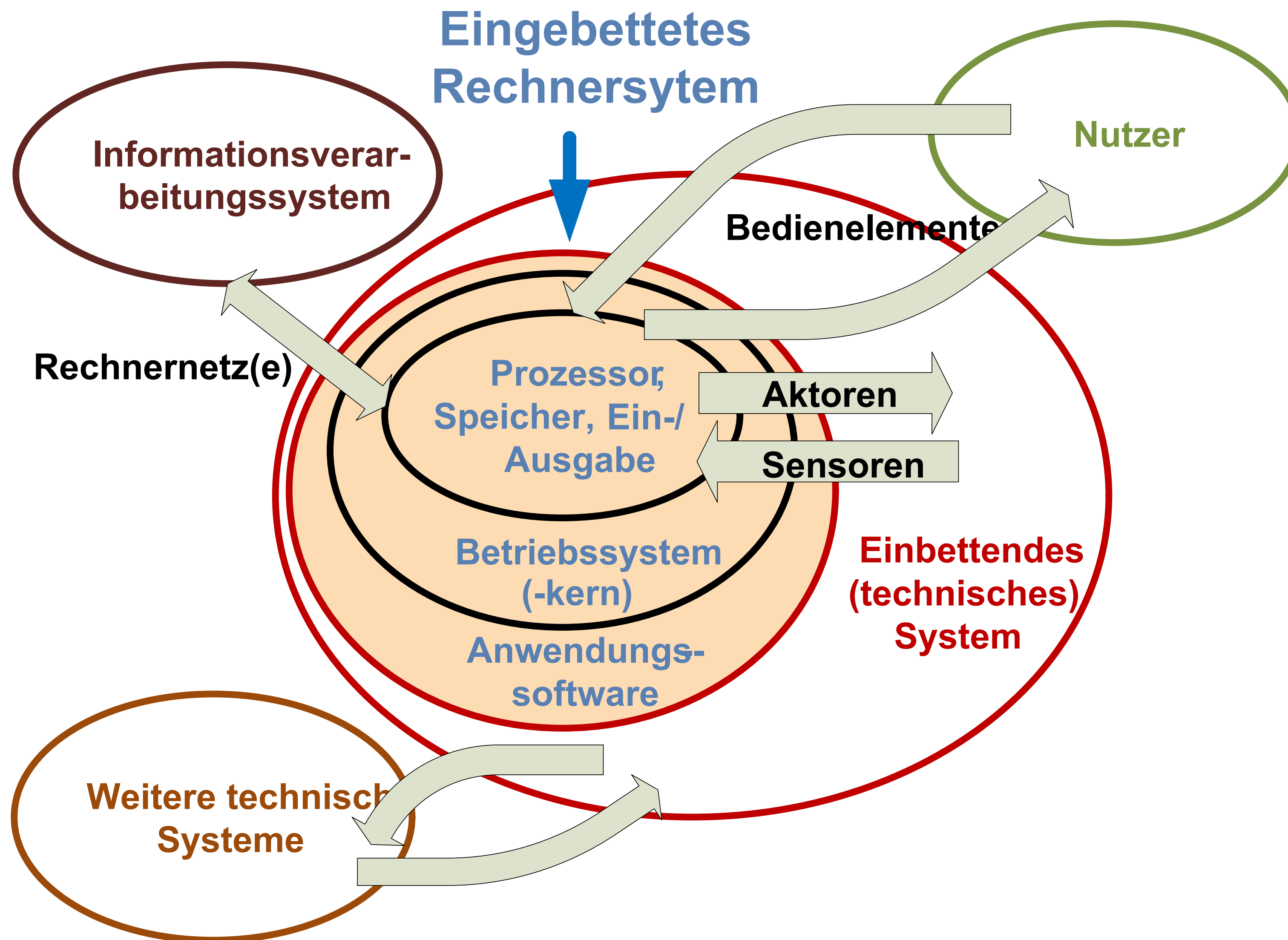


Bild 1.2: Eingebettetes Rechnersystem

- Prozessor, Speicher, Ein-/Ausgabe: Rechner zur Abarbeitung der Programme, die den Motor steuern (z.B. Benzineinspritzung) und mit der Umgebung (weitere Fahrzeugaggregate, z.B. Getriebe) und dem Nutzer (Fahrer) zusammenwirken,
- Betriebssystemkern: elementare universelle Software-(SW)-realisierte Funktionen zur Abarbeitung von Anwenderprogrammen (z.B. LINUX-Kern),
- Anwendungs-SW: Programme, die die Steuerfunktion und die Nutzerinteraktion realisieren,
- Sensoren, Aktoren: Erfassen Messwerte und geben Stellwerte von und zum technischen Prozess ein und aus (z.B. Drehwinkel der Kurbelwelle, Ventilansteuerung für Benzineinspritzung; beides benötigt der Algorithmus, der den Zeitpunkt der Einspritzung berechnet und diese auslöst),
- Einbettendes technisches System (Verbrennungsmotor),
- Bedienelemente (z.B. Gaspedal (Fahrpedal), dessen Stellung Eingangswert für die Algorithmen ist, Drehzahlanzeige),
- Rechnernetz zu anderen Steuergeräten (z.B. Getriebesteuergerät, über „Motorraumbus“, CAN-Interface (Controller Area Network)),
- weitere technische Systeme (z.B. Getriebe).

Ein Beispiel zu Steuergeräten zeigt Bild 1.3 S.14.

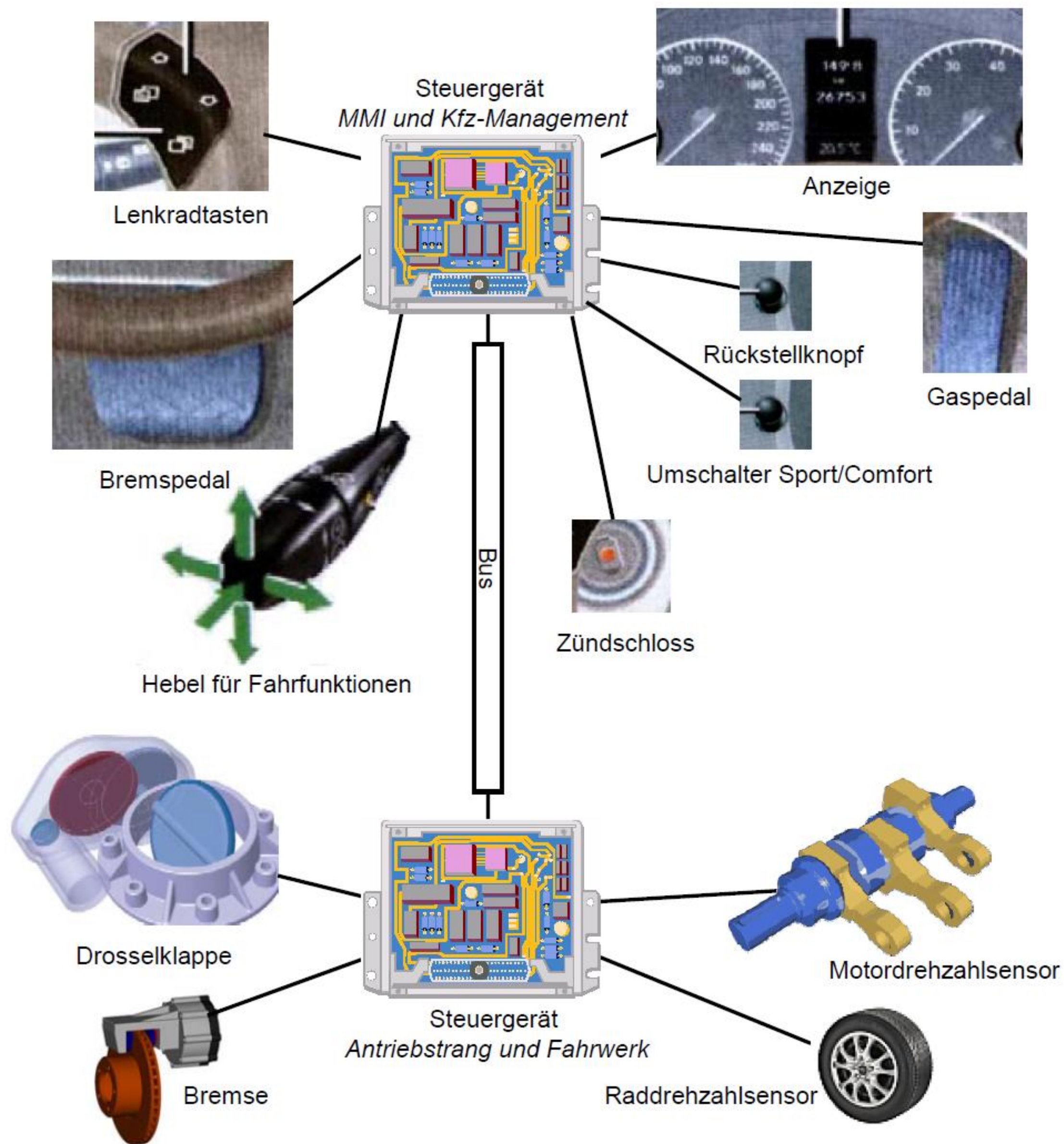


Bild 1.3: Beispiel zu Steuergeräten (aus einer Fallstudie in [7])

2. Parallele Automaten zur Architekturmodellierung

2.1. Grundlagen

Im vorliegenden Material dienen die Automatengraphen als Modellierungsmittel, d. h. die Automaten beschreiben modellhaft das formale Verhalten durch Zustandsfolgen und Ausgabesignale in Abhängigkeit der Eingangssignale. Das ermöglicht auch einen Zugang zur späteren Anwendung von verschiedenen, auf Harel basierenden State Charts in der Digitaltechnik [8], wobei diese aufgrund ihrer Komplexität hier nicht zur Anwendung kommen.

Der Abschnitt 2 hat die Aufgabe, aufbauend auf der Basis von Grundlagen zur kombinatorischen und sequentiellen Logik (siehe [1], [3]) Systeme von parallelen Automaten soweit einzuführen, dass sie zur Beschreibung von logischen Abläufen in den behandelten Komponenten und deren Zusammenwirken nutzbar sind.

Ausgangspunkt sind (einfache) endliche Automaten:

➔ sequentielle Logik (Zustände).
➔ Darstellung im Weiteren als Automatengraph, keine Kodierungen, Moore-Automaten.
Die Darstellung erfolgt hier zumeist ohne Eigenschleifen (Ergänzung über vollständig u. widerspruchsfrei formal möglich).

Parallele Automaten sind: mehrere Automaten, die gekoppelt sind (System gekoppelter Automaten).

Getakteter (synchroner) Automat:

Mögliche Zustandsübergänge erfolgen mit der nächsten steigenden Taktflanke.

Takt: 0-1-Folge mit konstanter Periode T (Bild 2.1, S.16)

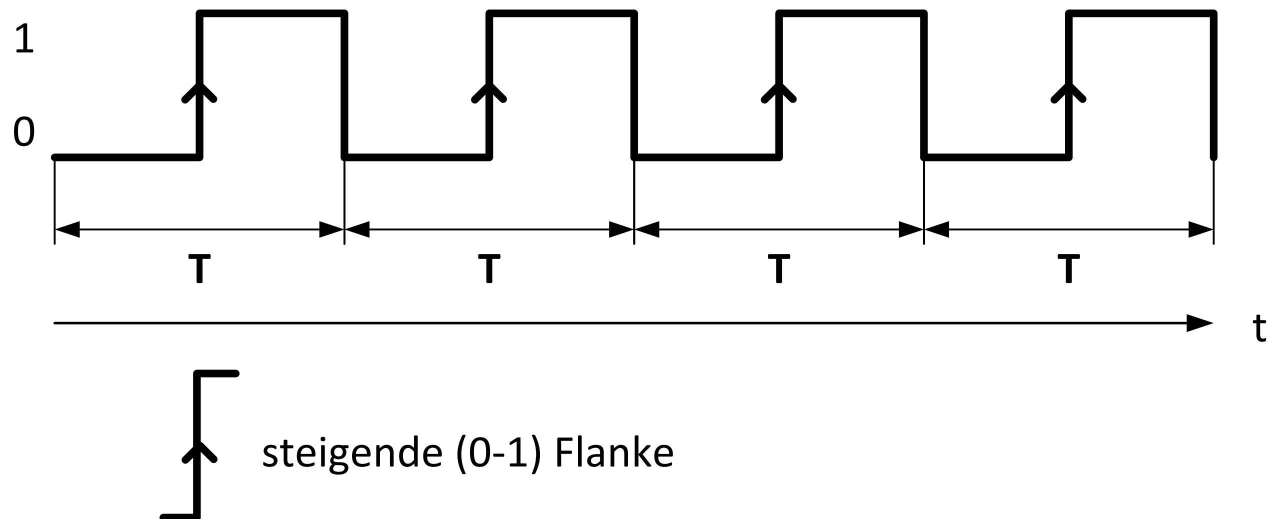
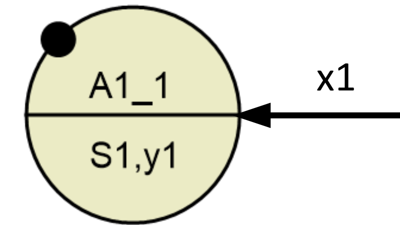


Bild 2.1: Taktverlauf eines synchronen Automaten

Im Weiteren werden als Symbolik und Schreibweise verwendet:

Zustand eines Automaten mit:



- Markierung für Initialzustand
- eindeutiger Bezeichner des Zustandes, hier: A1_1
- alle Variablen, welche in diesem Zustand wahr (logisch EINS) sind, hier: S1 und y1

Kante (zwischen Zuständen) mit Bedingung, hier: x1

Für logische Ausdrücke gilt:

	Boolesche Schreibweise	modifizierte Schreibweise
UND	$\wedge$	&
ODER	$\vee$	+
NICHT	$\overline{}$	/
Beispiel	$x1 \wedge x2 \vee \overline{x3}$	x1 & x2 + /x3
Verwendung in diesem Material		in den Formeln und an den Automaten

Weiterhin wird verwendet:

(binäre) Variable := Wert
Variable wird auf Wert (Wert kann auch 0 sein) gesetzt und behält diesen bis zum nächsten Setzen.

Bild 2.2 S.18 zeigt ein einfaches Beispiel eines Automaten (2 logische Zustände: Z0, Z1, eine Eingangsvariable x1 und 2 Ausgangsvariablen y0, y1).

Es entsteht bei simulativer Abarbeitung des Automaten mit geeigneter Variation von x1 das im unteren Teil des Bildes dargestellte Zeitdiagramm.

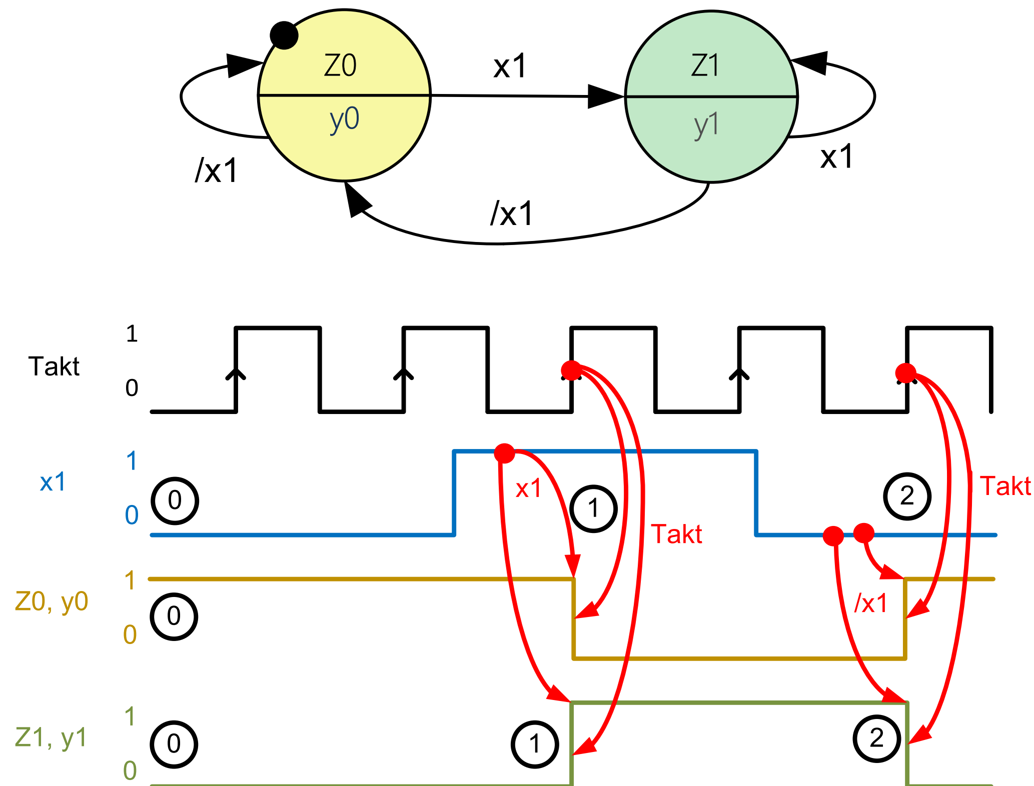


Bild 2.2: Beispiel eines einfachen Automaten (Automatengraph und Zeitdiagramm)

Der Ablauf erfolgt in den Schritten:

(0): Initialzustand, Der Zustand des Automaten ist Z0 (dargestellt durch Z0=1, alle anderen Zustände, hier Z1, = 0). x1 ist auf 0 festgelegt.

(1): Durch Variation von x1=1 erfolgt mit der nächsten steigenden Taktflanke der Wechsel des Zustandes von Z0 nach Z1 (Z0=0, Z1=1).

(2): Durch Variation von x1=0 erfolgt mit der nächsten steigenden Taktflanke der Wechsel des Zustandes von Z1 nach Z0 (Z0=1, Z1=0).

Die Ausgangsignale ergeben sich als Mooreautomat [3], [1]:

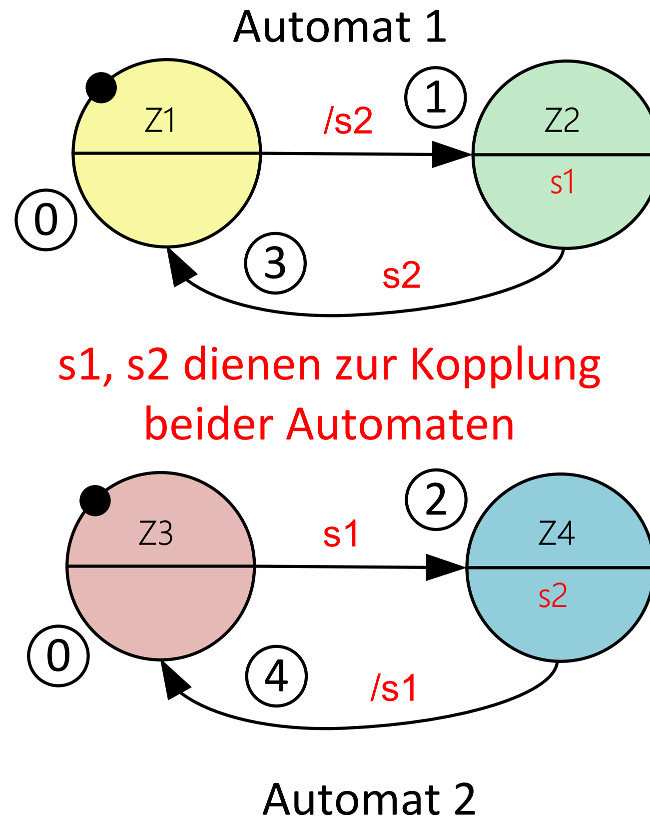
y0=Z0

y1=Z1.

Es folgt ein einfaches Beispiel zu zwei parallelen gekoppelten Automaten (Bild 2.3 S.20).

Zur Kopplung dienen die Variablen s1, s2. Dabei ist s1 beim (Moore-) Automat 1 Ausgangs- und beim Automat 2 Eingangsvariable. Umgekehrt ist s2 beim (Moore-) Automat 2 Ausgangs- und beim Automat 1 Eingangsvariable.

Beim angegebenen Automaten ergibt sich zwingend die angegebene Zustandsfolge, da keine Eingangsvariablen von außen genutzt werden.



Zustandsfolge:
(0): Initialzustand
(Z1, /s1), (Z3, /s2)
(1): durch /s2 ->
(Z2, s1), (Z3, /s2)
(2): durch s1->
(Z2, s1), (Z4, s2)
(3): durch s2 ->
(Z1, /s1), (Z4,s2)
(4): durch /s1 ->
(Z1, /s1), (Z3, /s2)
wie (0) (Initialzustand)

Bild 2.3: Beispiel eines einfachen Systems paralleler Automaten (Automatengraphen und Zustandsfolge)

2.2. Beispiele

Wozu die Modellierung mit parallelen Automaten dienen kann, soll anhand von zwei anschaulichen Beispielen erläutert werden.

Beispiel 1 („aus dem studentischen Leben“): abstrakte Wohngemeinschaft (WG) (Bild 2.4 S.22).

Gegeben sei ein Wohnraum mit einem Tisch, zwei Fachbüchern und einem Bett.
Die Bücher sind den zwei anwesenden Nutzern/Nutzerinnen lokal zugeordnet.
Das Bett wird von beiden zeitgeteilt exklusiv genutzt.
Das Bild 2.5 S.23 zeigt ein System aus parallelen Automaten, welches das Problem beschreibt und dabei die exklusive Nutzung des Betts gewährleistet. Das geschieht durch die wechselseitige Beeinflussung der beiden Automaten (je Student(in) Sti) über die Koppelsignale von Bettzugriff St1 (B1) und von Bettzugriff St2 (B2), die, wie oben schon erläutert, jeweils Ausgangsvariablen des einen und Eingangsvariablen des anderen Automaten sind.
Eine Besonderheit ist enthalten: Sollten beide Studenten(innen) exakt gleichzeitig auf das Bett zugreifen wollen, wird durch eine weitere einseitig wirkende Koppelvariable „Warten auf Bett von Student/in 1“ (WB1) der Übergang von „Warten auf Bett“ nach „Bettzugriff für Student/in 2“ gesperrt.

Durch Austausch des Betts mit einem, wiederum exklusiv nutzbaren, Drucker, auf den von den zwei Notebooks der Studenten/innen zugegriffen werden kann, kommt man schon näher an Probleme aus der Rechnerarchitektur (Bild 2.6 S.24).

Bild 2.7 S.25 und Bild 2.8 S.26 übertragen das Modell auf die moderne Rechnerarchitektur: Dual-Core (zwei Cores, Prozessorkerne) mit gemeinsamem Level-3-Cache (L3-Cache).

Erkennbar ist, dass auch hier das behandelte Modellierungsprinzip effektiv anwendbar ist. Bild 2.7 S.25 zeigt zwei Cores (Dual-Core) als Chip-Bild-Ausschnitt. In Bild 2.8 S.26 wurde das Problem mit zwei parallelen Automaten modelliert.
Core i ist das Äquivalent zu Student(in) i, L3-Cache das zu Bett.

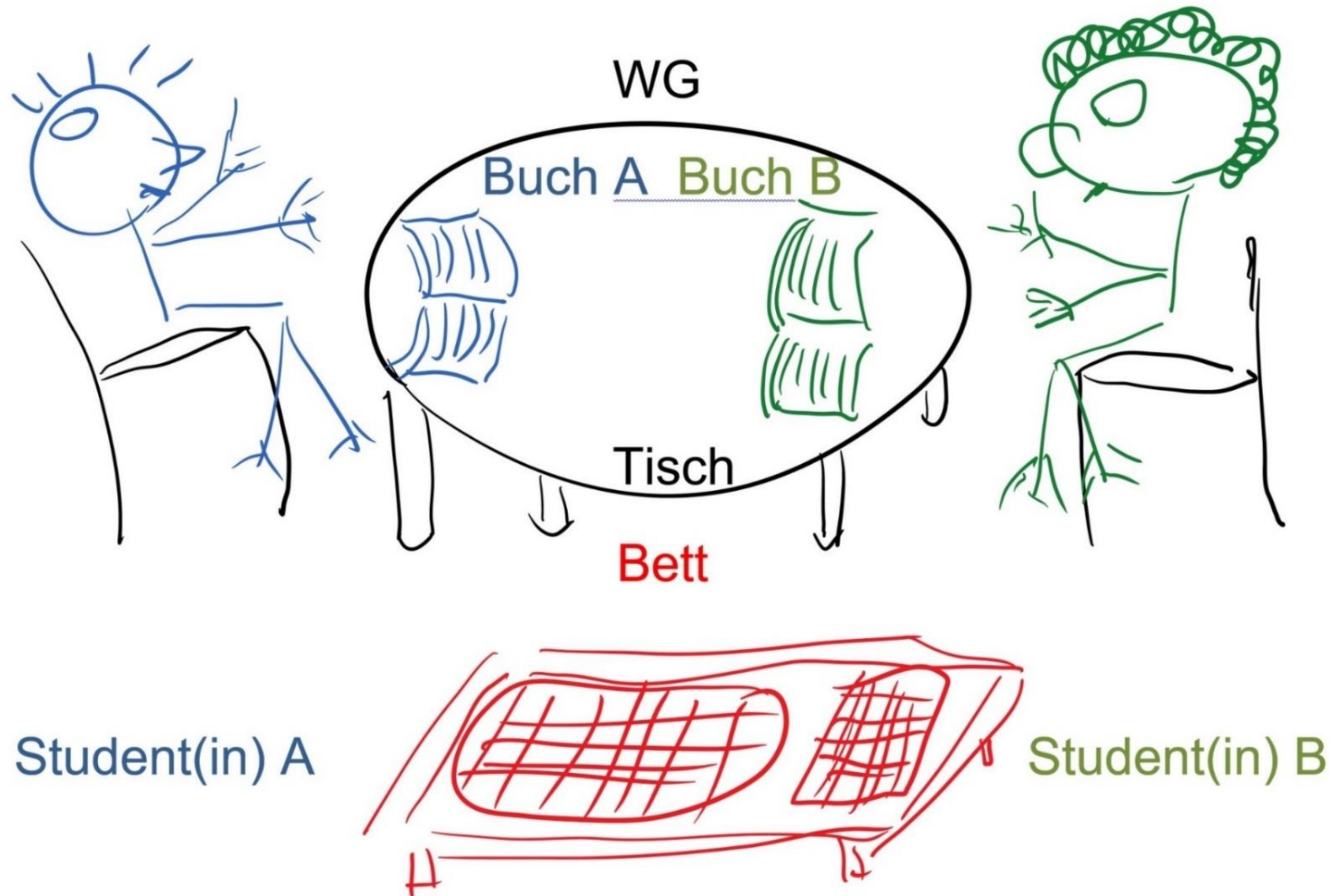
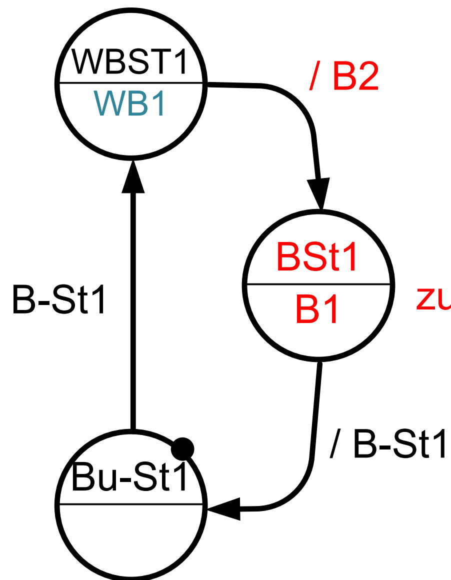


Bild 2.4: Abstrakte Wohngemeinschaft (WG), schematisch

Student/in 1

Warten auf Bett



Zugriffe auf lokales Buch

Bettzugriff frei

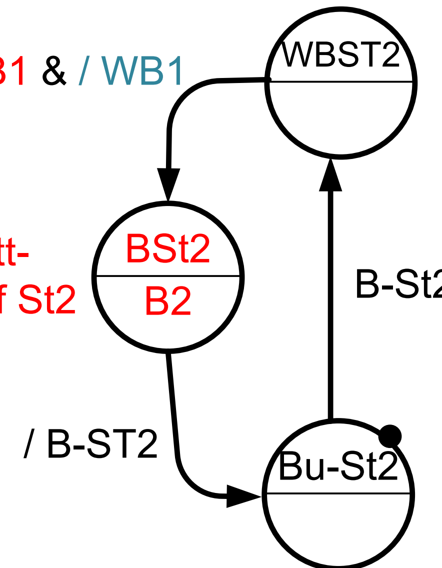
Bett-
zugriff St1

exklusiv !

Bett-
zugriff St2

Student/in 2

Warten auf Bett



Zugriffe auf lokales Buch

Automat	**Student/in 1**	**Student/in 2**	
Zustände/Koppelsignale	Bu-St1	Bu-St2	Zugr. auf lokales Buch Stud. i
	WBSt1, WB1	WBSt2	Warten auf Bett Stud. i
	BST1, B1	BST2, B2	Bettzugriff Stud. i
Eingangssignale	B-St1	B-St2	Bettwunsch Stud. i

Bild 2.5 System aus parallelen Automaten zur Modellierung des Problems WG

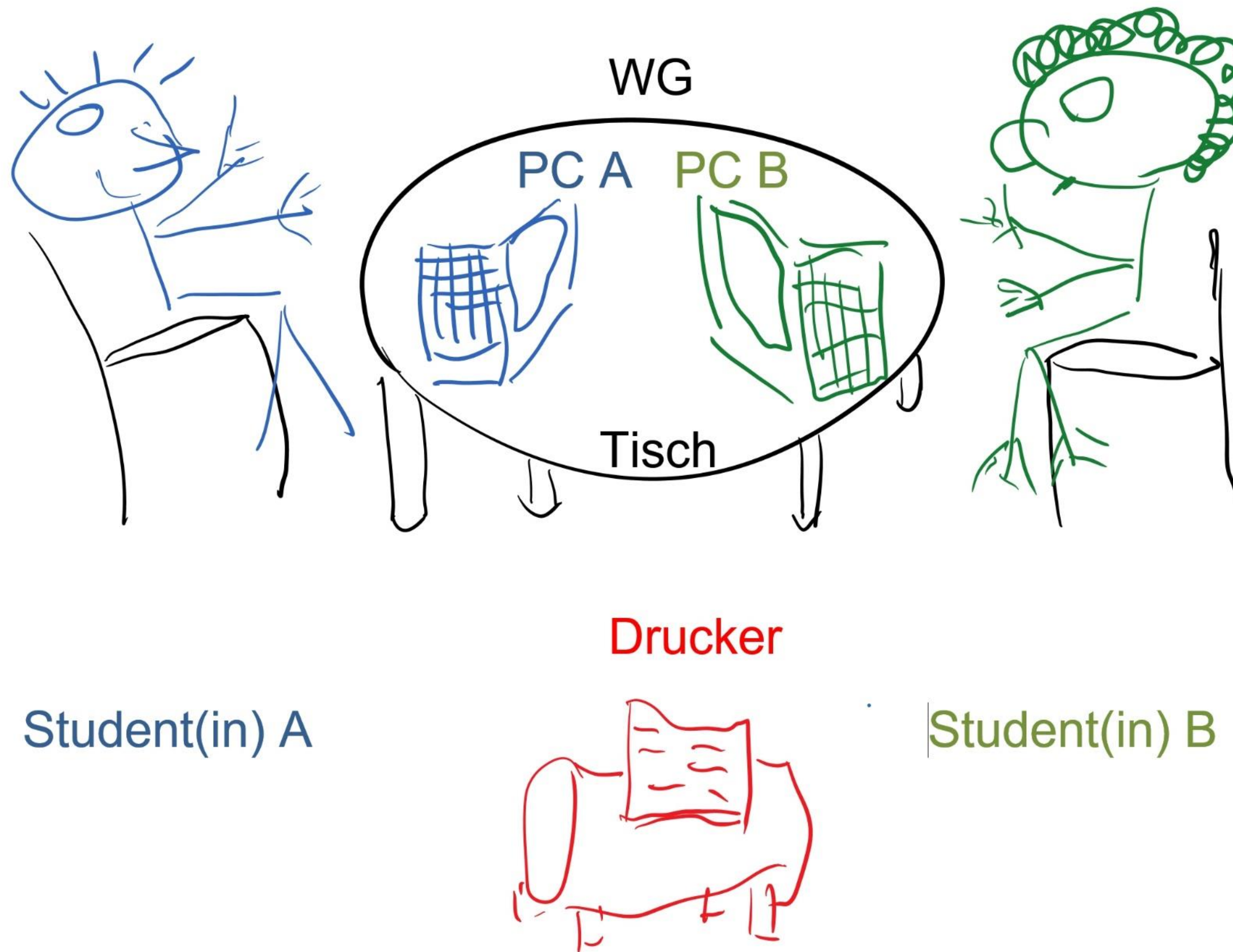


Bild 2.6: Abstrakte Wohngemeinschaft (WG), schematisch, „Rechnerarchitektur-nah“

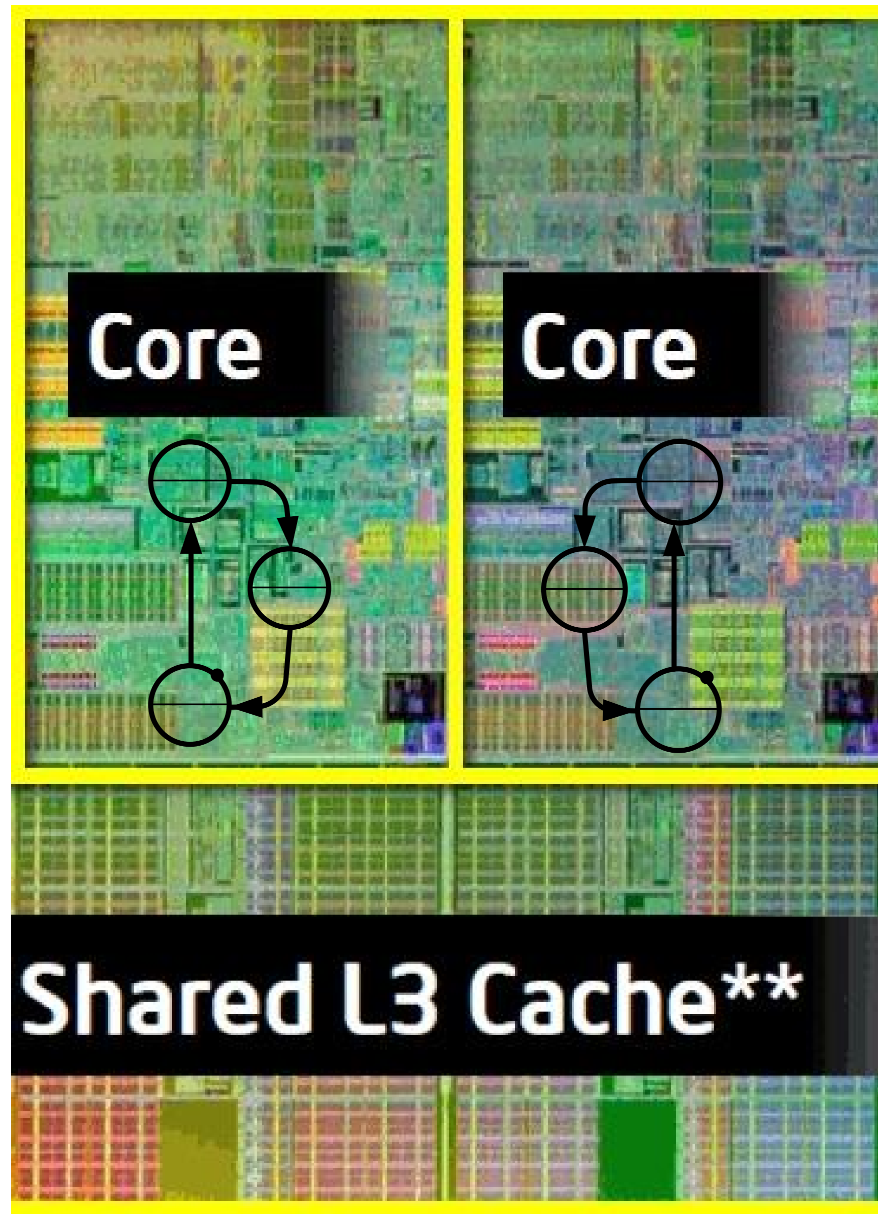
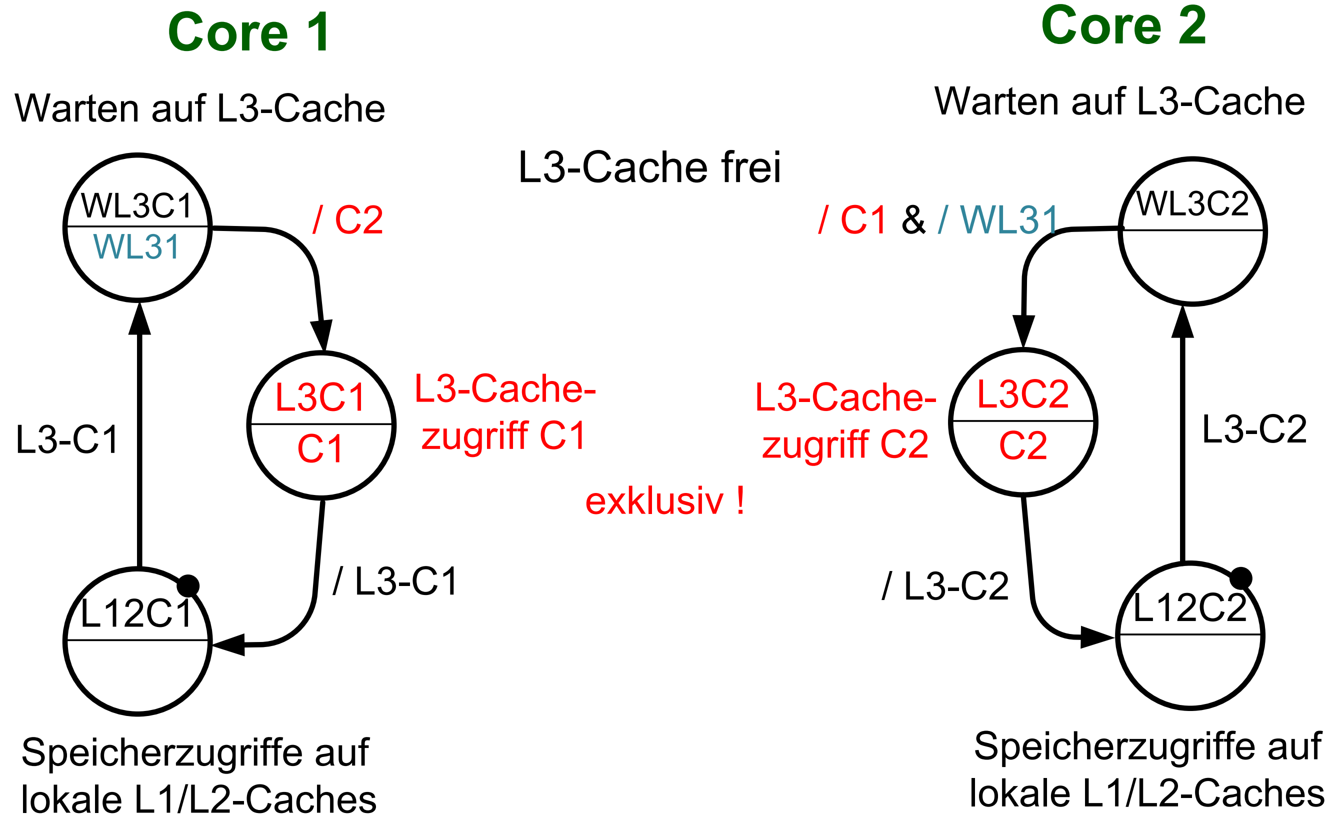


Bild 2.7: Prozessor-Chip-Ausschnitt [9], zwei Cores, ergänzt mit Automatengraphen



Automat	**Core 1**	**Core 2**	
Zustände/Koppelsignale	L12C1	L12C2	Zugr. L1,2-Cache Core i
	WL3C1, WL31	WL3C2	Warten L3-Cache Core i
	L3C1, C1	L3C2, C2	L3-Cache-Zugriff Core i
Eingangssignale	L3-C1	L3-C2	L3-Cache-Wunsch Core i

Bild 2.8: Modellierung Dual-Core mit gemeinsamem L3-Cache

Abschließend folgt das etwas komplexere Beispiel 2 zu zwei gekoppelten parallelen Automaten.

Automat 1, 2 sind zwei sequentielle Automaten (wie oben beschrieben), die gleichzeitig „arbeiten“ (d.h. Zustandsübergänge durchführen).

xi: sind binäre Eingangssignale (sie können auf je einen oder beide Automaten wirken).

yi: sind binäre Ausgangssignale, wegen der Einschränkung Moore-Automat direkt von den Zuständen abgeleitet.

Si: sind binäre Koppelvariablen. Sie werden als Signale in den Zuständen eines der 2 Automaten erzeugt und können in allen anderen Automaten in den Übergangsbedingungen verwendet werden.

Dieses System dient der Veranschaulichung der Modellerstellung und des Modellverhaltens. Es beschreibt in der Form keine konkrete Anwendung.

Die Blockstruktur enthält Bild 2.9 S.28. Die beiden parallelen Automaten zeigt Bild 2.10 S.29 und die resultierenden Zeitverläufe Bild 2.11 S.30. Die angegebenen Zahlen in beiden Bildern kennzeichnen deren Zusammenhänge.

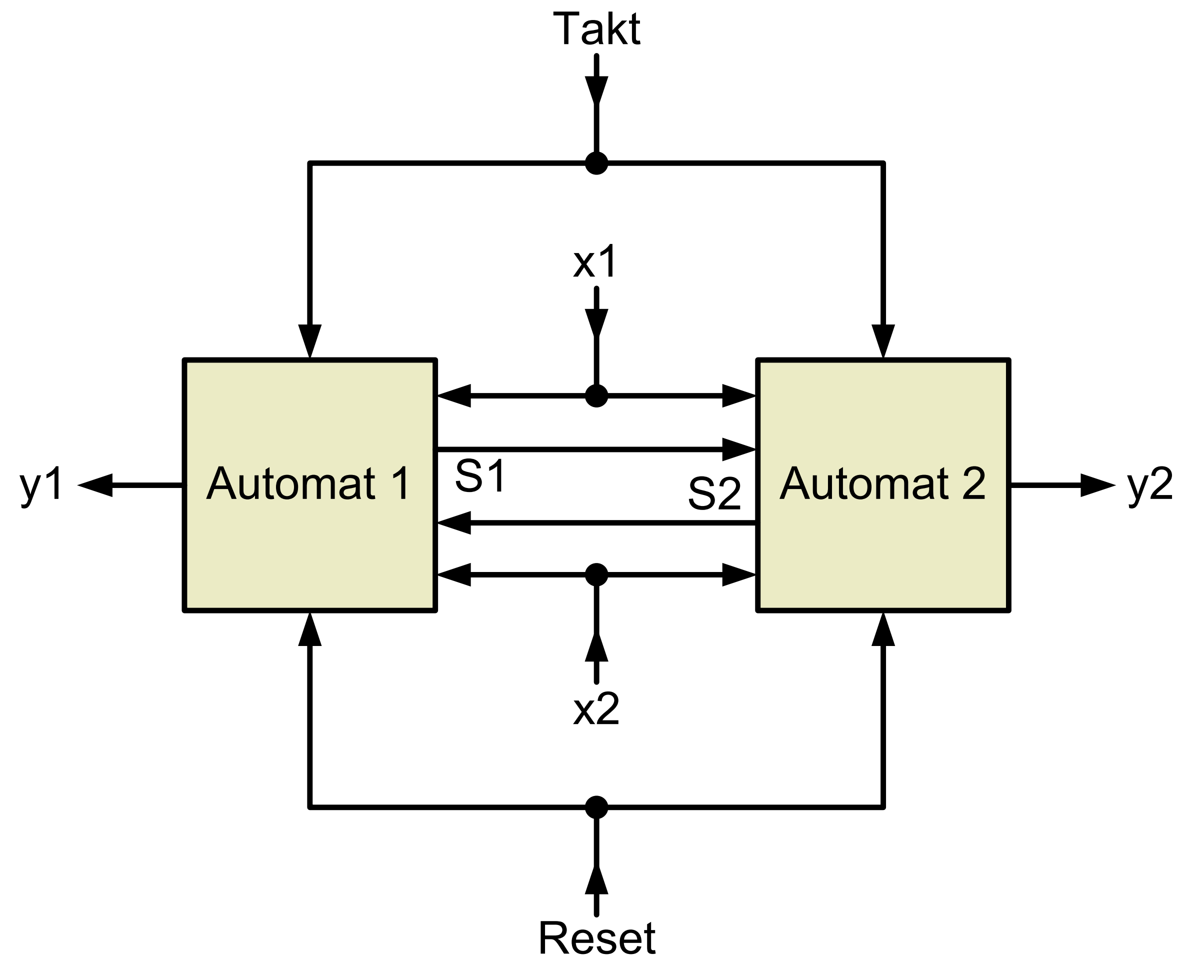


Bild 2.9: Blockstruktur zum Beispiel „Zwei gekoppelte parallele Automaten“

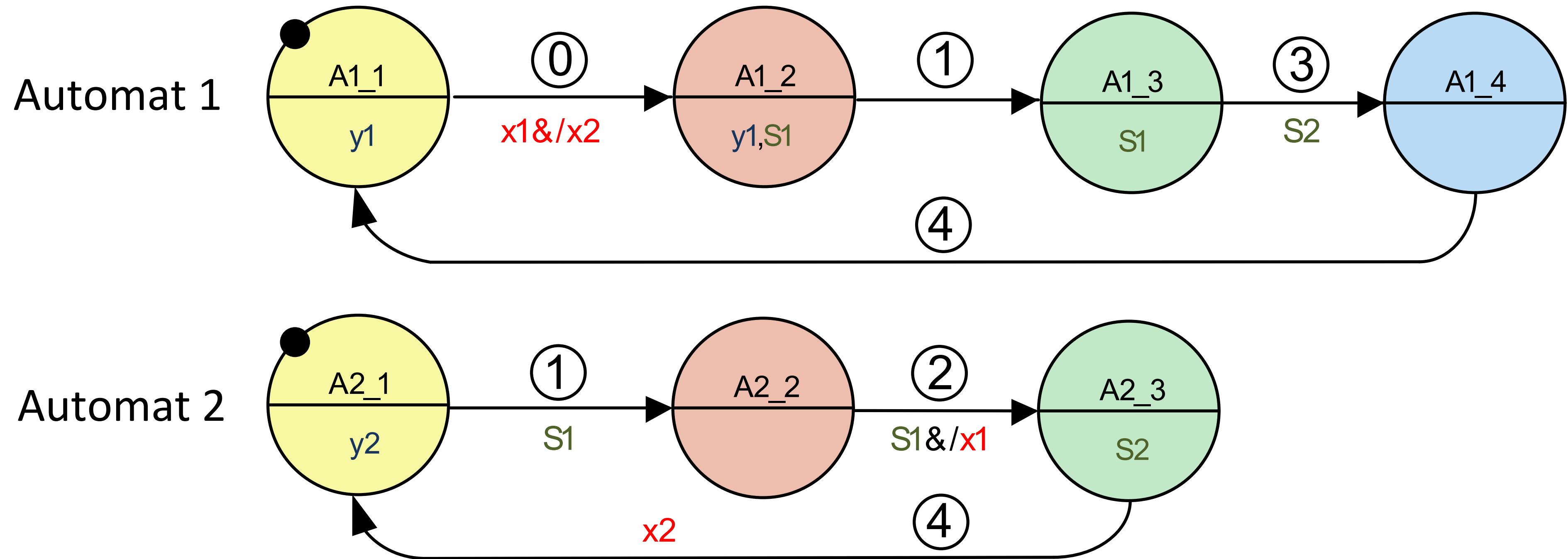


Bild 2.10: Automatengraphen zum Beispiel „Zwei gekoppelte parallele Automaten“

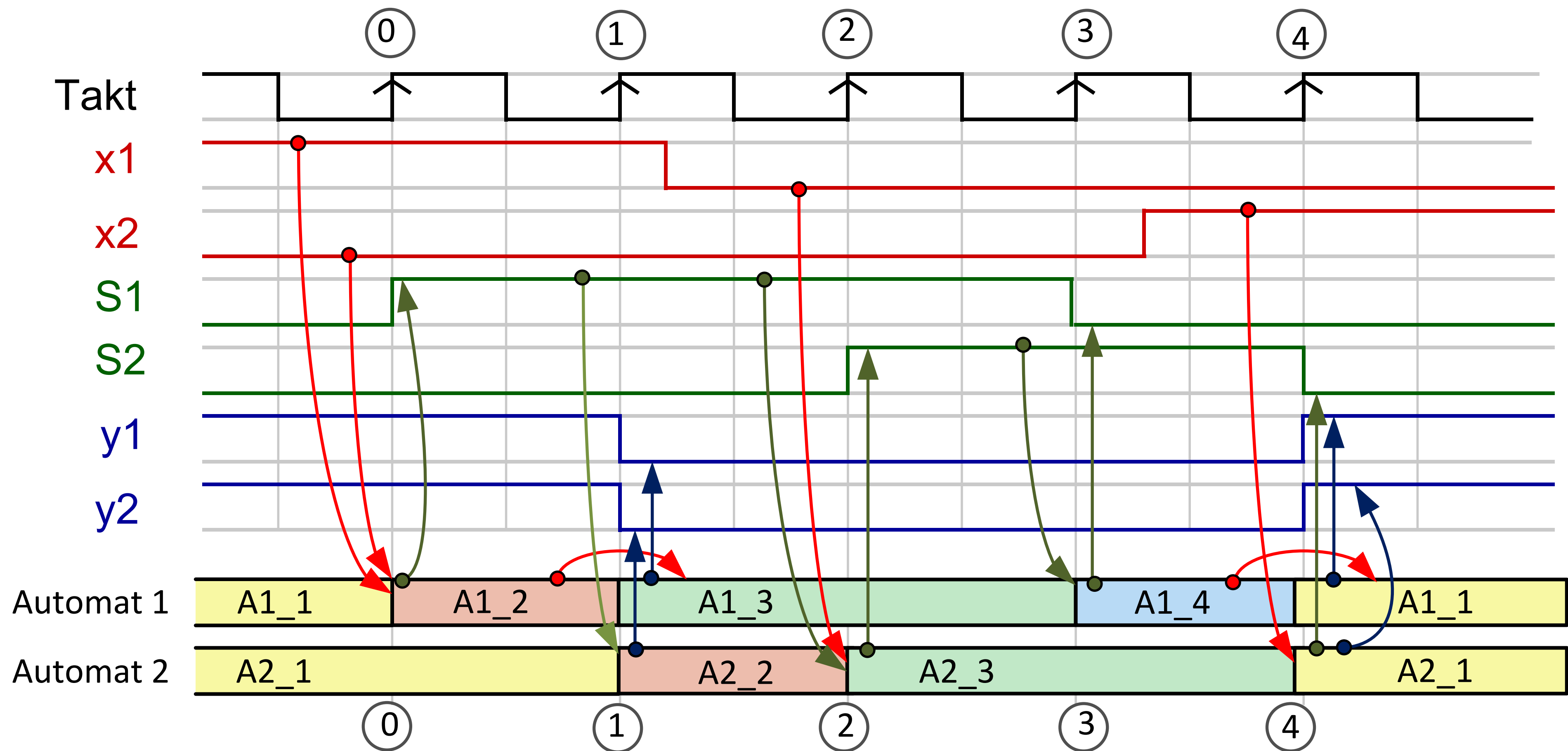


Bild 2.11: Zeitverläufe zum Beispiel „Zwei gekoppelte parallele Automaten“

Die anschließende Erläuterung beschreibt die in den Bildern 2.10 S.29 und 2.11 S.30 angegebenen Zustandsübergänge und die auslösenden Eingangs- (xi) bzw. Koppelsignale (Si).

Initialzustand:
A1_1 aktiv: S1 = 0, y1 = 1;
A2_1 aktiv: S2 = 0, y2 = 1,

(0) durch x1 & /x2 = 1 (mit nächster steigender Taktflanke) Übergang A1_1 nach A1_2: S1 = 1

(1) durch unbedingten Übergang A1_2 nach A1_3: y1 = 0;
durch S1 = 1 Übergang A2_1 nach A2_2: y2 = 0,

(2) durch S1 = 1 und x1 = 0 Übergang A2_2 nach A2_3: S2 = 1,

(3) durch S2 = 1 Übergang A1_3 nach A1_4: S1 = 0,

(4) durch unbedingten Übergang A1_4 nach A1_1: y1 = 1;
durch x2 = 1 Übergang A2_3 nach A2_1: S2 = 0, y2 = 1

-> das ist wieder der Initialzustand.

3. Architekturgrundlagen

Der vorliegende Abschnitt 3 dient zur Vermittlung von Kenntnissen zur globalen Rechnerarchitektur und zu Grundprinzipien, die in den weiteren Abschnitten an verschiedenen Stellen zur Anwendung kommen.

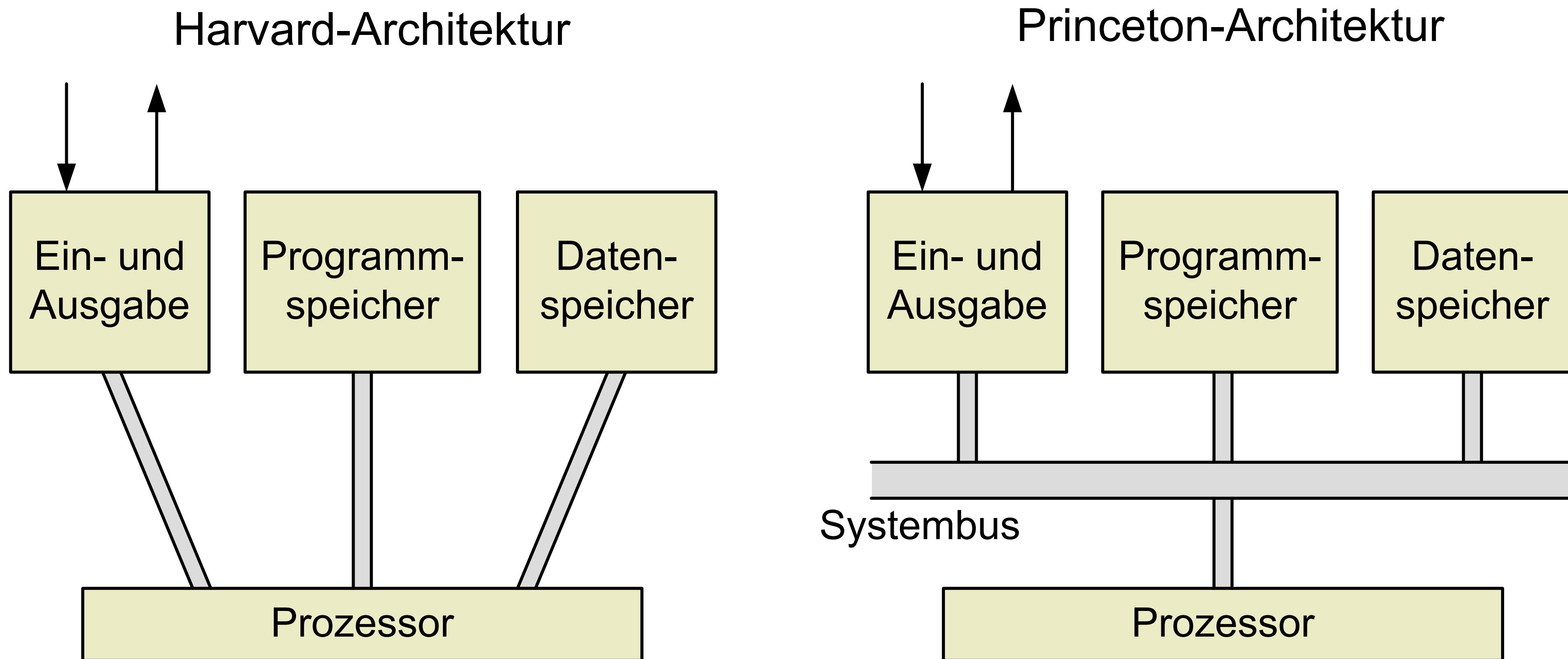


Bild 3.1: Grundarchitekturvarianten

Bild 3.1 S.32 zeigt die oberste Hierarchie eines Rechners (nicht mehrere Prozessoren, also kein Multicore, sondern nur 1 Core) in zwei Grundvarianten. Der Unterschied ist die Verbindungsstruktur (direkt: Harvard-Architektur; über gemeinsamen Systembus: Princeton-Architektur, wird i. Allg. im Weiteren benutzt).

Funktionsblöcke der Princeton-Architektur:

- Prozessor: befehlsabhängige Verarbeitung von Daten,
- Programmspeicher: enthält die Befehlsfolgen,
- Datenspeicher: enthält die zu verarbeitenden Daten und die Verarbeitungsergebnisse,
- Ein/Ausgabe: bildet Schnittstellen für Daten in und aus dem Rechner (von und zu einer Umwelt),
- Verbindungsstruktur:
  Systembus, verbindet alle 4 Funktionsblöcke untereinander.

Erläuterung der Blöcke:

Prozessor:

- Verknüpfung von Eingangsoperanden zu Ausgangsoperanden (Ergebnisoperanden) in Abhängigkeit von (Maschinen)Befehlen.
- Hilfsfunktionen:

- Steuerung der Befehlsreihenfolge und Lesen von Befehlen (Reihenfolge legt das den Algorithmus realisierende Programm fest),
- Lesen und Schreiben von Operanden.

Programmspeicher:

- Physischer Träger von Programmen in Form von Maschinenbefehlen,
- Funktionen: Bereitstellung von ausgewählten Maschinenbefehlen,
- Hilfsfunktionen: evtl. Laden von Maschinenprogrammen, Speichern der Maschinenprogramme.

Datenspeicher:

- Physischer Träger von Operanden in Maschinendatendarstellungen,
- Funktionen: Bereitstellen bzw. Übernehmen von ausgewählten Operanden,
- Speichern der Operanden.

Ein- und Ausgabe (EA):

- Physische Schnittstelle für den Operandenaustausch mit der Umwelt (z.B. technischer Prozess, Nutzer, Geräte …),
- Funktionen: Auswahl einer EA-Schnittstelle und das Lesen (Eingabe) bzw. Schreiben (Ausgabe) von Operanden in Maschinendatendarstellung. Befehle bei ladbaren Programmen (z.B. von Festplatte) werden bei EA wie Operanden behandelt.

Die folgenden Grundprinzipien werden in den Funktionsblöcken angewendet.

Auswahlprinzip: Adressierung (Bild 3.2 S.35).

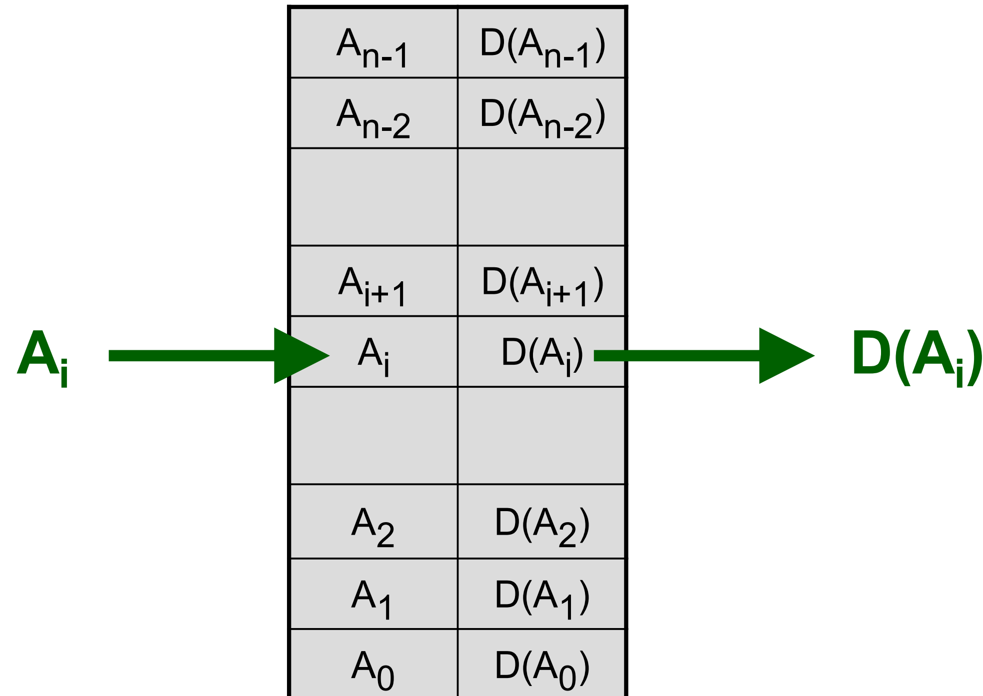


Bild 3.2: Grundprinzip Adressierung

Es ordnet einem physischen Platz im Befehlsspeicher, Datenspeicher oder im Ein-/Ausgabe- (EA-) Bereich eine logische Adresse, hier Binärzahl zu.

Dabei wird für die Auswahl von Befehlen bzw. Operanden im Speicher und die Auswahl von EA-Schnittstellen dieses Prinzip „Adressierung“ angewendet.

Funktionsweise:

Der Prozessor generiert eine Adresse als unvorzeichenbehaftete ganze Binärzahl mit m Bit:

A_i.

Dabei gilt:

$n= 2^m$

m ist typisch = 16, 32, 64

Bsp.: m=32: 2^{32} Speicherworte mit je einer unterschiedlichen Adresse i

Beim Speicherzugriff:

Ausgabe der Adresse A_i vom Prozessor zum Speicher bzw. EA und

Lesen bzw. Schreiben eines Befehls bzw. Operanden von der bzw. auf die ausgegebene Adresse A_i.

Bei Anwendung des Adressierungsprinzips auf Ein-Ausgabeschnittstellen ist

m (EA) typ. = 8, 16; Bsp.: 8

Als Adressraum eines Prozessors gilt die maximal adressierbare Anzahl Speicherworte (Bsp.: 2^{32}). Das gilt hier für Speicherworte, die aus einem Byte bestehen.

Ein Datenwort ist dann k Byte.

k ist typisch 1, 2, 4, 8.

Ein RISC-Bsp. (siehe im Weiteren): es wird nur auf Datenworte aus 4 Byte (d.h. 32 Bit) zugegriffen (siehe Struktur in Bild 3.3 S.37).

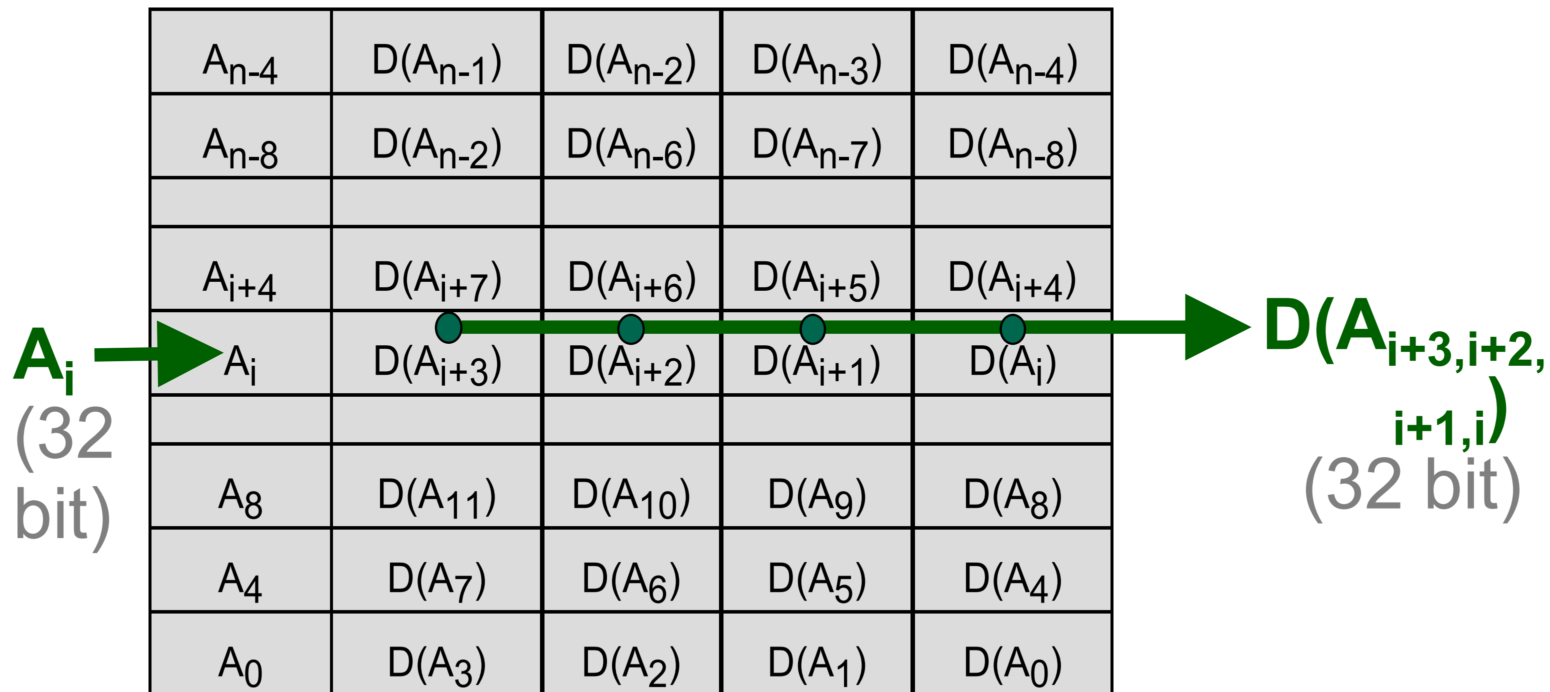


Bild 3.3: Adressierung eines Datenworts aus 4 Byte

Für eine logische Adresse von A_{n-1} bis A_0 mit $n = 2^m$:

Bsp. m=32 -> 4 GByte (4 294 967 296 Byte) adressierbar. Das entspricht der Anzahl der möglichen Adressen, die verwendet werden können.

Bei (32-Bit)-Wortadressierung werden zumeist die untersten 2 Adressbit zur Byteunterscheidung genutzt. Deshalb sind für m=32 2^{30} Datenworte (2^{m-2}) adressierbar.

Im Weiteren wird, sofern nicht anders benannt, ein Datenwort mit 4 Byte (32 Bit) verwendet (bezeichnet mit Byteadressierung, Adressabstand für zwei aufeinander folgende Datenworte sind 4 Byte, d.h. Abstand 4).

Dabei enthält ein 32-Bit- (4 Byte-) Datenwort die Adressen (Bild 3.3 S.37):

A_{i+3}, A_{i+2}, A_{i+1}, A_{i+0} (Wertigkeit der Byte in dieser Reihenfolge).

Es erfolgt das Lesen/Schreiben aller k Byte parallel (gleichzeitig).

Register im Prozessor:

- Enthalten aktuell oft benötigte Operanden und evtl. Adressen oder Steuerinformationen.
- Register-Lesen/Schreiben benötigen keinen Speicherzugriff -> Operationen mit Registern sind während der Befehlsabarbeitung deutlich schneller im Zugriff als die mit Speicherworten.

Der interne Aufbau und die realisierte Funktion eines Registers werden in Abschnitt 6.1 (u. a. in Bild 6.4 S.148) bei den Speichern behandelt. Weiteres dazu findet man auch in [1], [3].
Im Weiteren wird das Registermodell von Bild 3.4 S. 39 verwendet.

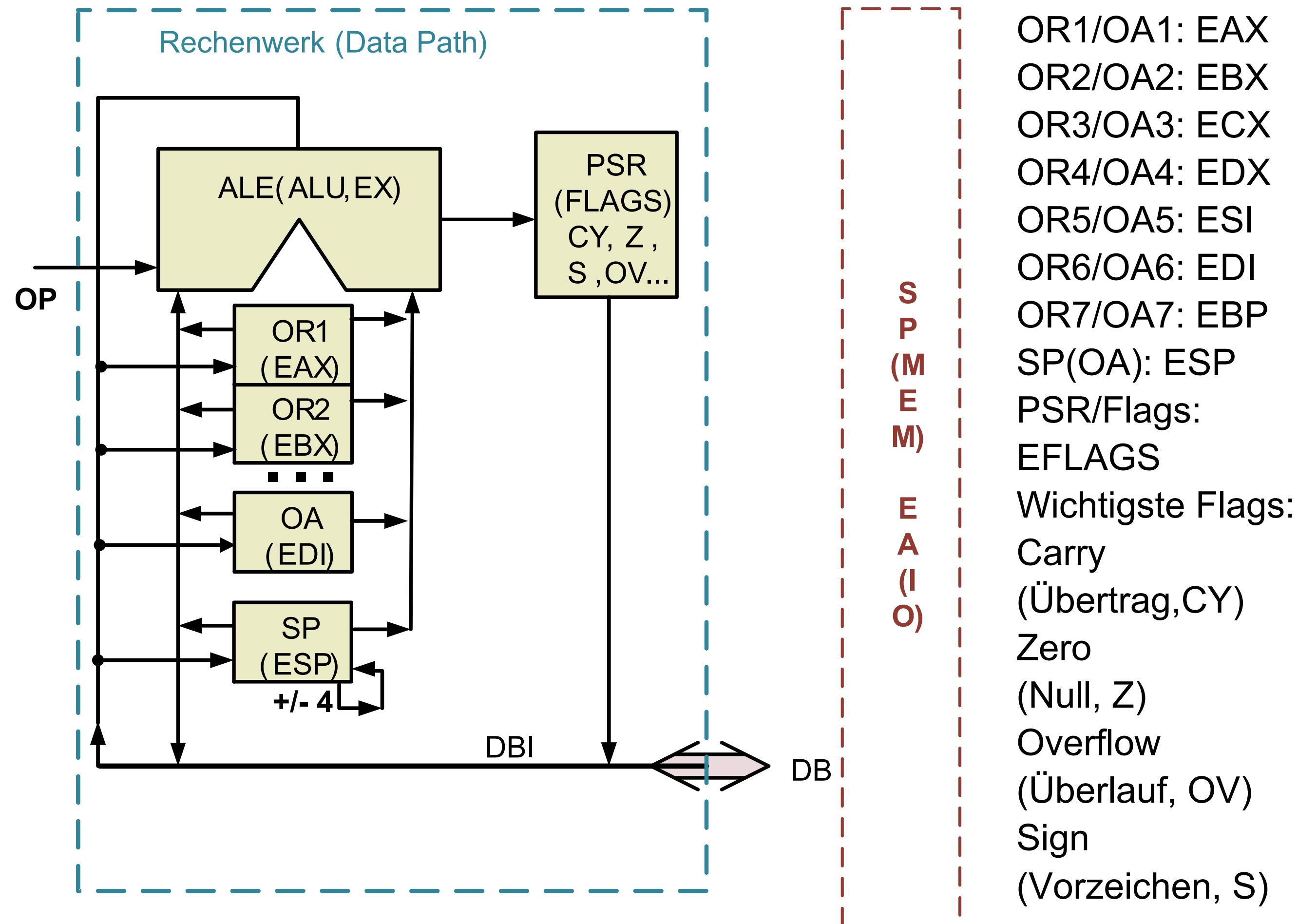


Bild 3.4: Registermodell

Dabei gilt (bezogen auf Bild 3.4 S.39, die Register sind auch bezeichnet wie im Bsp.-Prozessormodell i86-32, [4], z.B. EAX für OR1):

- Operandenregister ORi, dienen als Zwischenspeicher für Operanden bei der Befehlsausführung,
- Operandenadressregister OA, für Adressen,
- Stackpointer SP, spezielles Operandenadressregister (siehe im Weiteren bei Stack-Befehlen),
- Steuerregister PSR (Prozessorstatusregister, Flags) mit den wichtigsten Flags:

  für Zahlenbereichsproblematiken: CY, OV

  für Ungleichungen: S, Z

  - CY (Carry, auch CF): erste Variante eines Zahlenbereichsüberlaufs,
  - OV (Overflow, auch OF): zweite Variante eines Zahlenbereichsüberlaufs (für vorzeichenbehaftete Zahlen),
  - S (Sign, auch SF): Vorzeichen des Ergebnisses,
  - Z (Zero, auch ZF): Nullergebnis.
  - Genaueres zu den PSR-Bit siehe bei den Befehlen in Abschnitt 4.

➔ Die Flags sind durch bedingte Befehle auswertbar (z.B. bedingte Sprünge).

ORi/OA können die beiden Eingangsoperanden der Arithmetik-Logikeinheit ALE (auch Arithmetic Logic Unit, ALU oder Execute-Unit, EX) und der Ausgangs-/Ergebnisoperand sein (genaueres zur ALE siehe im Weiteren in den Abschnitten 4 und 5, u.a. in Bild 5.4 S.100).

Flags werden dabei entsprechend dem Ergebnis gesetzt (am Operationsende).

Alle Registerinhalte können mit dem Speicher über den Datenbus (DB) ausgetauscht werden.

Die Operation der ALE wird durch einen binären Vektor OP eingestellt, die Erzeugung dessen erfolgt wie im Abschnitt 5.2 S.99 beschrieben. Dabei führt die ALE die in den Abschnitten 4.3 und 4.4 beschriebenen Befehle als Maschinencode aus.

ALE und Register werden zumeist zusammenfassend als Rechenwerk bezeichnet.

Die Registerstruktur aus Bild 3.4 S.39 ist aus der Sicht der (Maschinencode-) Programmierung allgemein üblich, auch wenn in modernen Prozessoren kompliziertere interne Strukturen unterlegt sind.

Es können Register mit 8, 16, 32, 64 Bit existieren. Im Weiteren:

RISC-Bsp.-Prozessor: nur Register mit 32 Bit.

Beispiel-Ablauf mit Registerbenutzung:

OR1, OR2 -> OP/ALE -> OR1, PSR

Ausführen der Operation OP durch die ALE (z.B. Addition) auf die beiden Operanden in OR1 und OR2; Ergebnis (z.B. Summe) nach OR1, je nach Ergebnis werden die Flags gesetzt bzw. rückgesetzt.

Die ORi können auch als Operandenadressregister benutzt werden, d.h. ihr Inhalt stellt dann aktuell eine Adresse dar (siehe entsprechende Befehle).

4. Befehlsübersicht

Hier werden die Befehle als Übersicht, eingeordnet in die folgenden Gruppen, vorgestellt.

Für einen konkreten Prozessor werden sie in entsprechenden Arbeitsblättern, z.B. in [4] auszugsweise, beschrieben. Diese Arbeitsblätter waren auch teilweise Grundlage für den Abschnitt 4. Weiterhin ist die Verwendung von Originaldokumenten der entsprechenden Prozessorhersteller möglich. Diese beschreiben aber einen Befehlsumfang, der hier nicht benötigt wird.

- Transportbefehle verändern den Ort von Operanden (4.2).
- Arithmetik-/Logikbefehle und Bitmanipulations-/Schiebebefehle erzeugen aus Eingangsoperanden Ergebnisse (Ausgangsoperanden) (4.3/4.4).
- Programmtransferbefehle ermöglichen nicht-sequentielle Befehlsfolgen (4.5).
- Sonstige: heterogene Befehle mit andern Aufgaben als die ersten vier Gruppen (4.6).

4.1. Übersicht und Grundlagen

Die Abschnitte 4.2 bis 4.6 geben eine Übersicht der vorhandenen Befehle (Maschinenbefehle) und deren Auswirkung bei der Abarbeitung.

Programmiersprachbefehle geben die Möglichkeit, Algorithmen zu formulieren und im Prozessor als Befehlsfolgen abzuarbeiten.

Dabei gibt es als Abstraktionsebenen in der Programmierung:

- Beschreibung der Algorithmen (-> siehe im Gebiet „Algorithmen und Programmierung“),
- Beschreibung der Algorithmen in höherer Programmiersprache (-> siehe im Gebiet „Algorithmen und Programmierung“; z.B. Java, andere höhere Programmiersprachen),
- Beschreibung der Algorithmen maschinenah: Assemblersprache (-> hier verwendet, vor allem in Abschnitt 4),
- Erzeugung von abarbeitungsfähigen Maschinenprogrammen in Maschinencode (im vorliegenden Abschnitt wird nur das Grundprinzip beschrieben).

Es gilt:

Ein Befehl einer höheren Programmiersprache (z.B. C, Java) wird typischerweise in eine Folge von Maschinenbefehlen umgewandelt:

- Das ist Aufgabe eines Compilers.
- Es gibt noch die Assemblersprache: jedem Maschinenbefehl ist genau ein (symbolisch notierter) Assemblerbefehl zugeordnet.
- Z.B. Move Ziel, Quelle: MOV EBX, EDX
- Aufgabe eines Assemblers (einfaches Übersetzerprogramm): Umformung von Assembler-Text-Programmen in ein zugeordnetes Maschinenprogramm.

-> im Weiteren werden nur die wichtigsten Maschinenbefehle dargestellt (Prinzip nach RISC).

RISC: Reduced Instruction Set Computer (Rechner mit reduziertem Befehlssatz), es existieren einfache reguläre Befehle.

Es gibt moderne Prozessoren, die nur einen RISC-Befehlssatz haben. In der iA-86-/64-Architektur [4] existiert ein Befehlssatz mit RISC-Eigenschaften, aber als Untermenge aller Befehle.

Maschinenbefehl:

Jedem Befehl ist genau ein Maschinencode zugeordnet, der diesen charakterisiert und anhand dessen der Prozessor die aktuell auszuführende Operation ermittelt und durchführt (OP in Bild 3.4 S.39, Registermodell)

Verwendet wird ein n-Bit-Befehlswort (n typ. 8, 16, 32, 64, 128)

Die Binärbelegung in diesem Befehlswort entsprich einem zugeordneten Befehl.

Bsp. (fiktiv):

```
ADD     OR1, OR3
```

Wirkung des Beispielbefehls: Addition der Operanden aus OR1 und OR3, Ergebnis überschreibt OR1, der Registerinhalt von OR3 bleibt erhalten (siehe Bild 4.1, S.47).

Bild 4.1: Prinzip des Maschinencodes (fiktiv)

Bei 32-Bit-Wortadressierung (siehe Bild 3.2 S.35) wäre der Adressabstand von zwei aufeinander folgenden Maschinecodes im Programmspeicher +1, bei Byteadressierung und 32-Bit-Befehlscode wäre der Abstand +4 (siehe Bild 3.3 S.37). Diese beiden Werte haben bei den noch folgenden Programmtransferbefehlen eine Bedeutung.

Bsp. für den Abstand +4, Maschinencode fiktiv:

Programmspeicheradresse	**Maschinecode**	**Befehl**
0100 0…0 0000B / 8 … 0H	0011 0001 … 0001 0011B	i
0100 0…0 0010B / 8 … 4H	0011 0110 … 0010 0100B	i+1
0100 0…0 0100B / 8 … 8H	0100 1111 … 0000 1001B	i+2

(B: Binär, H: Hexadezimal)

In den einzelnen Befehlen im Weiteren werden Symbole verwendet, die in den Bildern 4.2 (a) S.49 und 4.2 (b) S.50 erläutert sind.

Sie enthalten die verwendeten Symbole: Argumente von Assemblerbefehlen.

Die Zeilen sind mit Bedeutung und Beispiel erklärt. Sie werden in den verschiedenen Befehlen konkret benutzt. (in [4] gibt es Erweiterungen).

Zu den einzelnen hier beschriebenen Befehlen wird auszugsweise die Syntax und die dazu gehörige Wirkungsweise entsprechend [4] angegeben. Danach folgt zumeist noch eine zusätzliche Erläuterung. Bei ähnlichen Befehlen wird teilweise auch auf syntaktische Angaben verzichtet.

Symbol	wird ersetzt durch ...	Bemerkung
reg32	ein 32-bit-Register	alle allgemeinen 32-bit-Register
const32	eine 32-bit-Konstante	ein vorzeichenbehafteter Zahlenwert im Programmcode
const8	eine 8-bit-Konstante	ein vorzeichenloser Zahlenwert im Programmcode als IO-Adresse
mem	eine Speicherreferenz (mit 32-bit-Speicheradresse)	siehe nächste Tabelle
label	ein Bezeichner, der vom Assembler durch eine Adresse ersetzt wird.	

Bild 4.2 (a): Verwendete Adressierungsarten (reduziert aus [4])

Zahlenangaben werden gekennzeichnet:

Binar: angehängtes B,

Hexadezimal: angehängtes H, Ziffern 0 … 9, A … F, vorangestellte 0,

Dezimal: keine Zusätze.

... mit Speicherzugriff

(Erläuterung für das Symbol mem)

Name der Adressierungsart	mem bedeutet hier	Beispiel
Direkt	[const32]	[03FF12FFH]
Indirekt	[reg32]	[ESI]
Index	[reg32 + const32]	[EAX - 0223H]
Basisindiziert	[reg32 + reg32[1])]	[ESP + ECX]

... ohne Speicherzugriff

Name der Adressierungsart	Operand	Beispiel
Register-Adressierung	reg32	EAX
Unmittelbare Adressierung („Direkt-Operand“)	const32	0FF004D56H

[1]) ohne ESP

Bild 4.2 (b): Verwendete Symbole (reduziert aus [4])

4.2. Transportbefehle

Die Transportbefehle verändern den Ort von Operanden.

Move in Varianten:

Kopieren: Quelle überschreibt Ziel, Quelle bleibt erhalten.

Es existieren verschiedenen Varianten von Ziel und Quelle.

Hier:

Ziel: Register, Speicherworte,

Quelle: Register, Speicherworte, Konstanten.

Bei RISC können Speicherworte nicht gleichzeitig in einem Befehl Ziel und Quelle sein.

MOV
(Move)

MOV reg32, reg32
MOV mem, reg32
MOV reg32, mem
MOV reg32, const32

Kopieren des Wertes des zweiten Operanden in den ersten Operanden

Alle Flags bleiben unverändert.

Beispiel
MOV EAX, 0FFAB011CH

Stackbefehle

Was ist ein Stack?

- Ein speziell organisiertes Speichersegment.
- Notwendig ist ein Stackpointer, SP. Er arbeitet als ein spezielles Operandenadressregister für die LIFO-Adressierung (Last In First Out).

Das Bild 4.3 S.53 soll das Stackprinzip anschaulich erläutern.

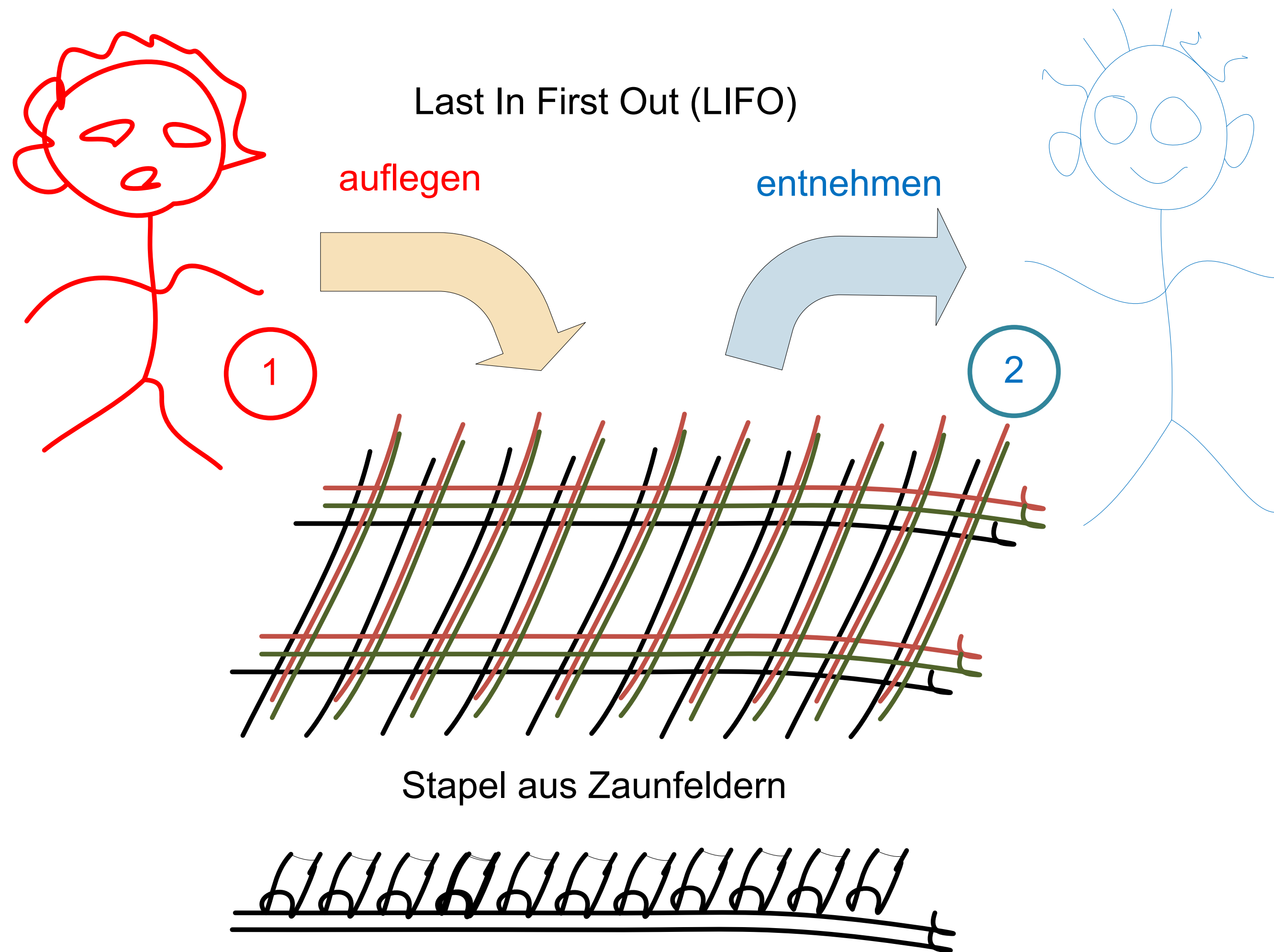


Bild 4.3: Beispiel für einen (physischen) Stapel (Stack)

Es werden Zaunfelder angeliefert und von der roten Person (1) auf dem Stapel oben aufgelegt.

Für die anschließende Montage werden die Felder von der blauen Person (2) vom Stapel entnommen. Eine Entnahme von unten wäre aufgrund der physischen Gegebenheiten nicht sinnvoll. Das bedeutet, (2) entnimmt zuerst das Feld, welches (1) als letztes aufgelegt hat (Last In First OUT, LIFO).

Das gleiche Verhalten wird für den Stackspeicherbereich angewendet. Die Erläuterung erfolgt anhand des Beispiels von Bild 4.4 S.54.

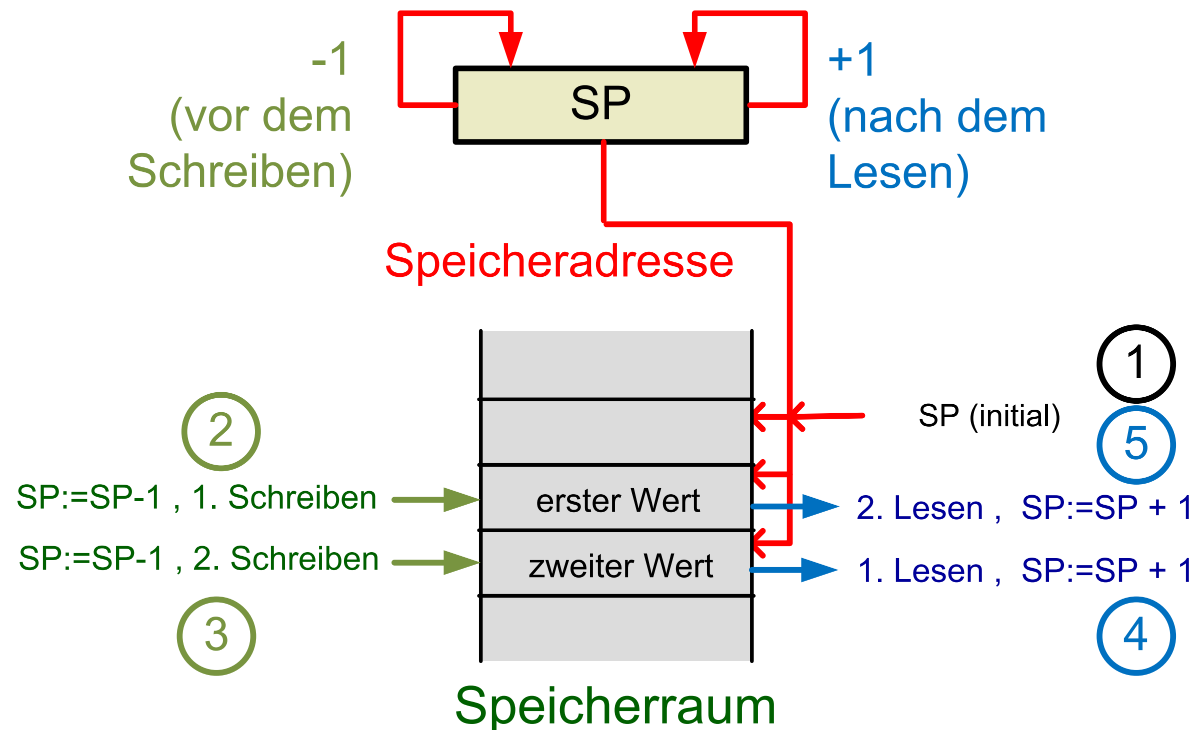


Bild 4.4: Stackbeispiel

(1) SP wird mit Endadresse + 1 des Stacksegments beschrieben (SP (initial)).

(2) 1. Schreiboperation:

Dekrementieren Wert des Stackpointers,

schreiben Datenwort (erster Wert) auf die Speicheradresse, die vom aktuellen SP adressiert wird.

(3) 2. Schreiboperation:

Dekrementieren Wert des Stackpointers,

schreiben Datenwort (zweiter Wert) auf die Speicheradresse, die vom SP adressiert wird.

(4) 1. Leseoperation:

Lesen Datenwort (zweiter Wert) von der Speicheradresse, die vom SP adressiert wird,

inkrementieren Wert des Stackpointers.

(5) 2. Leseoperation:

Lesen Datenwort (erster Wert) von der Speicheradresse, die vom SP adressiert wird,

inkrementieren Wert des Stackpointers. Danach ist der Wert des SP wieder SP (initial).

Zusammengefasster Ablauf:

Schreiben 1. Wert,

Schreiben 2. Wert,

Lesen 2. Wert,

Lesen 1. Wert.

➔ Stackoperationen müssen mit LIFO symmetrisch und zuordnungsmäßig abgestimmt angewendet werden. Im Mittel muss die Anzahl der Leseoperationen gleich der Anzahl der Schreiboperationen sein.

➔ Ab der Adresse SP (initial) und oberhalb dürfen keine Stackoperationen erfolgen.

Speicherzugriffe im Stackbereich können durch die Transportbefehle PUSH und POP erfolgen bzw. sind in die Unterprogrammbefehle (Abschnitt 4.5) und den Interruptmechanismus (Abschnitt 5.4) integriert (dort für Befehlsadressen verwendet).

PUSH

Speichern des Operanden auf dem Stack

PUSH reg32

Das Stackpointerregister ESP wird um vier verringert. Anschließend wird der Operand in der Speicherzelle, deren Adresse in ESP enthalten ist, abgelegt.

Alle Flags bleiben unverändert.

Beispiel

PUSH EAX

POP

Laden des Operanden vom Stack

POP reg32

Der Operand wird mit dem Inhalt der Speicherzelle, deren Adresse im Stackpointerregister ESP enthalten ist, geladen.
Nach dem Laden wird ESP um vier erhöht.

Alle Flags bleiben unverändert.

Beispiel

POP ECX

PUSH (Stack Schreiben)

Quelle: Register

Ziel: Speicherwort, adressiert durch den SP (ESP)

POP (Stack Lesen)

Quelle: Speicherwort, adressiert durch SP (ESP)

Ziel: Register

Lesen und Schreiben mit entsprechender SP-Veränderung:

Erhöhung bzw. Verringerung um 4, da ein 32-Bit-Register im Byte-adressierten Speicher 4 Byte (fortlaufend adressiert) belegt (Bild 3.3 S.37).

Die Wirkungsweise von PUSH und POP (je auf zwei unterschiedlichen Adressen im Stackbereich) verdeutlicht Bild 4.5 S.59.

Dabei gilt -4/+4 für Byteadressierung von 4-Byte-Datenworten (siehe oben).

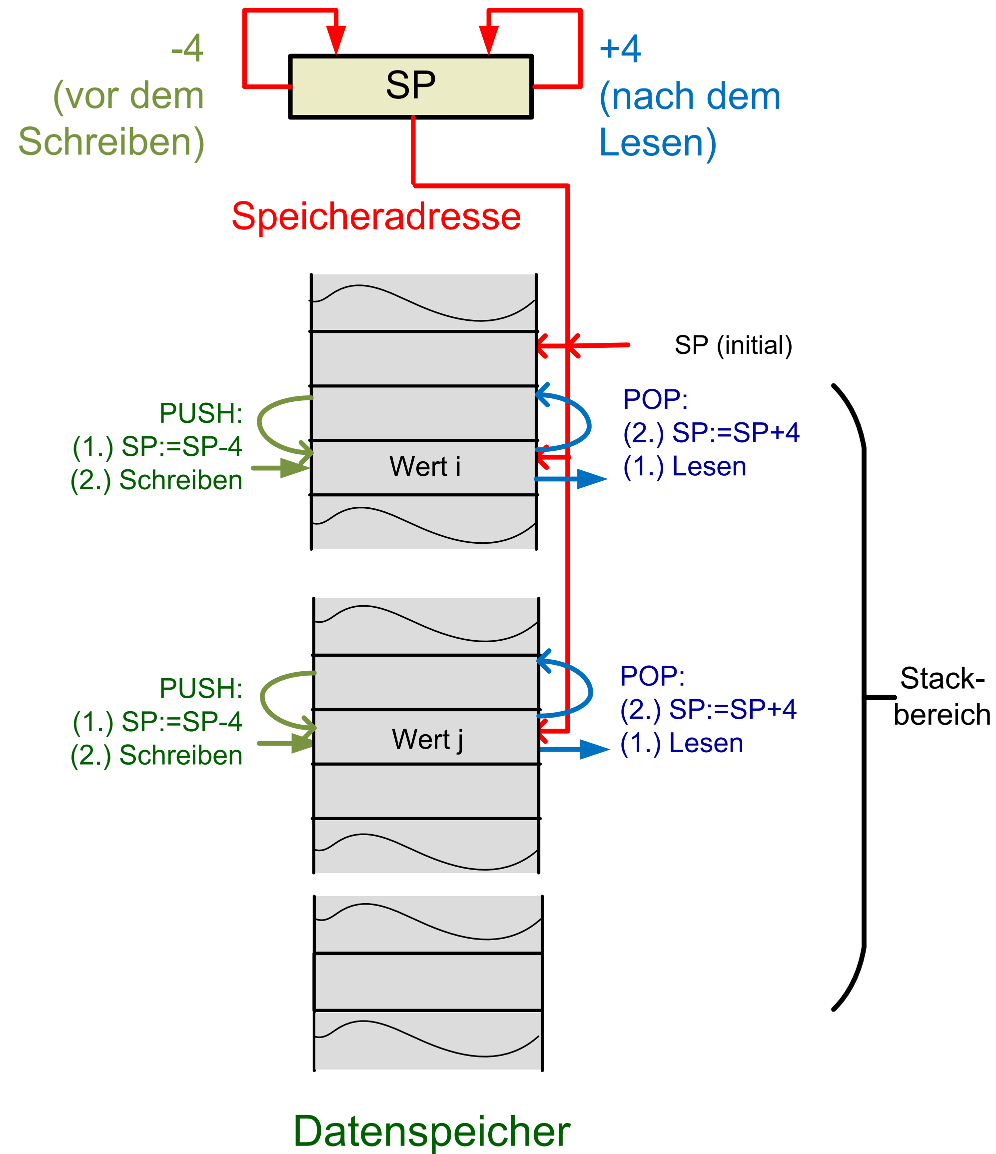


Bild 4.5: PUSH und POP im Stackspeicherbereich

Ein- und Ausgabebefehle

Sie dienen zum Datenaustausch zwischen Prozessor und EA-Schnittstellen.

IN
(Input)

Eingabe

IN EAX, const8

Eingabeoperation mit der durch den zweiten Operanden gegebenen E/A-Adresse in das im ersten Operanden angegebene Register.

Alle Flags bleiben unverändert.

Beispiel

IN EAX, 40H

OUT
(Output)

Ausgabe

OUT const8, EAX

Ausgabeoperation mit der durch den ersten Operanden gegebenen E/A-Adresse aus dem im zweiten Operanden angegebenen Register.

Alle Flags bleiben unverändert.

IN

Transport eines Werts von der EA-Schnittstelle in ein Register.

Ziel: Register (eingeschränkt),

Quelle: über 8-Bit-Konstante adressierte EA-Schnittstelle (evtl. auch Adressierung über Registerinhalt, im Weiteren nicht betrachtet).

OUT

Transport eines Werts vom Prozessor zur EA-Schnittstelle.

Ziel: über 8-Bit-Konstante adressierte EA-Schnittstelle (evtl. auch Adressierung über Registerinhalt, im Weiteren nicht betrachtet).

Quelle: Register (eingeschränkt),

Auswahlprinzip von EA-Schnittstellen:

Das Adressierungsprinzip wird wie beim Speicher verwendet (siehe Bild 3.2 S.35), aber mit kleinerem n (hier n=8, d.h. maximal 256 EA-Schnittstellen).

Für IN und OUT ist auch die Benutzung von 8-Bit-Registern als Ziel bzw. Quelle sinnvoll, da viele EA-Schnittstellen nur diese Datenbreite besitzen. Das wird hier aufgrund der getroffenen RISC-Forderung (nur 32-Bit-Register) nicht verwendet.

➔ Die geringe Anzahl der EA-Schnittstellen stellt keine Begrenzung dar, da im Normalfall nur wenige Schnittstellen benötigt werden und größere Datenmengen über Ein-/Ausgabe seriell (d.h. Bit-, Byte- oder Wort-weise zeitlich hintereinander ausgetauscht werden).

Zur Wirkung der EA-Befehle siehe auch Abschnitt 7.1.

4.3. Arithmetik-Logik-Befehle

Ziel:

Manipulation von Operanden zu Ergebnissen mit elementaren arithmetischen und logischen Operationen, dargestellt in zugelassenen Maschinendatenformaten (siehe auch Grundlagen zur Rechnerorganisation [3], [4], [2]).

Generell:

2 oder 1 Eingangsoperand

(werden mehr als 2 notwendig -> Überführen in eine Befehlsfolge aus mehreren Operationen).

1 Ausgangsoperand, evtl. keiner.

Arithmetisch-Logische Operationen setzten abhängig vom Ergebnis die Prozessorstatusregister-Bit (Flags) zur Vorbereitung von bedingten Befehlen (siehe Befehlsbeschreibungen).

Wichtigste Flags: CY (CF), OV (OF), S (SF), Z (ZF)

Mit S, Z sind alle Vergleiche möglich.

Nach Subtraktion (z.B. ohne Ergebnis schreiben, CMP):

- Z=1 -> Ergebnis Null -> Operanden waren gleich,
- Z=1 oder S=0 -> kleiner gleich,
- Z=0 und S=0 -> kleiner,
- Z=1 oder S=1 -> größer gleich,
- Z=0 und S=1 größer.

Arithmetik/Logik allgemein: komplexere Verknüpfungen sind durch Maschinenbefehlsfolgen zu erzeugen.

➔ siehe auch im Gebiet „Numerische Mathematik“.

ADD, ADC

Addition zweier Operanden

ADD reg32, reg32

Die beiden Operanden werden addiert. Das Ergebnis der Operation wird im ersten Operanden abgelegt. Bei ADC wird zusätzlich der Wert des Carry-Flags hinzuaddiert.

CF, ZF, SF und OF werden entsprechend dem Ergebnis der Operation gesetzt.

SUB, SBB

Subtraktion zweier Operanden

SUB reg32, reg32

Der zweite Operand wird vom ersten Operanden abgezogen. Das Ergebnis der Operation wird im ersten Operanden abgelegt. Bei SBB wird zusätzlich der Wert des Carry-Flags subtrahiert.

Zu den Grundrechenarten:

- Addition
- Subtraktion

Ohne und mit Übertragsberücksichtigung (Carry-Flag).

Diese Befehle beziehen sich hier auf das Datenformat ganzzahlig, Zweierkomplement und im RISC-Befehlssatz gilt: Mit Arithmetik-Logik-Verknüpfung ist kein Datentransport verbunden, Ziel und Quelle sind Register (im i86-Befehlssatz [4] sind aber auch Befehle mit Datentransport vorhanden).

INC

Inkrementieren des Operanden
Der Operand wird um Eins erhöht.

DEC

Dekrementieren des Operanden
Der Operand wird um Eins vermindert.

INC, DEC: Ziel und Quelle sind das gleiche Register.

CMP

Vergleich zweier Operanden
Der zweite Operand wird vom ersten Operanden abgezogen. Es werden nur die entsprechenden Flags gesetzt, das eigentliche Ergebnis der Subtraktion wird nicht benutzt (beide Operanden bleiben unverändert, Subtraktion ohne Ergebnisverwendung, d.h. das Ergebnis wird nicht geschrieben). In Verbindung mit den entsprechenden bedingten Sprungbefehlen lassen sich damit Vergleiche durchführen und auswerten –> dient zur Vorbereitung der bedingten Befehle.

MUL
Multiplikation von 32-Bit-Werten (vorzeichenlos, vorzeichenbehaftet)

MUL reg32

Der in Register EAX enthaltene 32-Bit-Wert wird mit dem Inhalt des angegebenen Registers multipliziert und das 64-Bit-Ergebnis im Register EDX
(Bit 63-32) und EAX (Bit 31-0) abgelegt.

Alle Flags sind nach Ausführung der Operation in undefiniertem Zustand.

Die Multiplikation erfolgt in Maschinendatenformaten [4], [3], [2].

Besonderheit bei der Registerbenutzung:

Erster Operand immer EAX,

zweiter Operand über reg32 wahlfrei angebbar.

Ergebnis ist immer EDX/EAX (63 …32/31 …0), 64 Bit, wegen der Vergrößerung des Wertebereichs bei der Multiplikation.

DIV

Division eines 64-Bit-Wertes durch einen 32-Bit-Wert (vorzeichenlos, vorzeichenbehaftet)

DIV reg32

Der 64-Bit-Dividend muss in den Registern EDX (Bit 63-32) und EAX (Bit 31-0) abgelegt sein. Dieser wird durch den im angegebenen Register enthaltenen Wert geteilt. Der Quotient wird in Register EAX abgelegt, der ganzzahlige Rest in Register EDX.

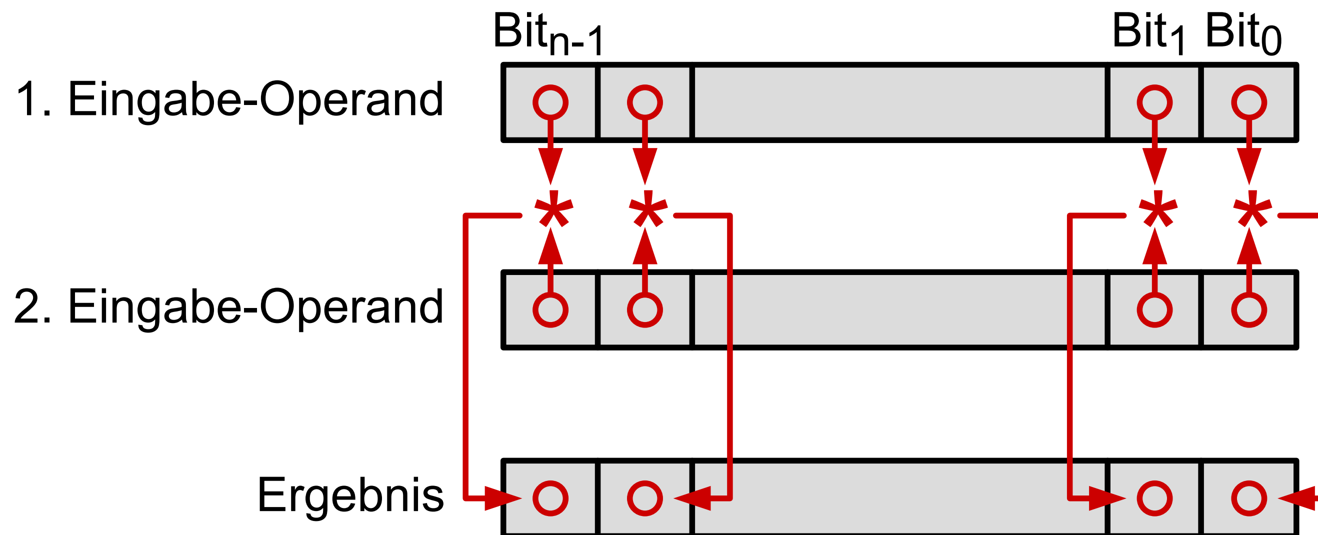
Division:

Auch hier erfolgt keine reguläre Registernutzung. Eine weitere Besonderheit ist der Rest.

Logikbefehle

- In der konventionellen Logik (z.B. Grundlagen der Rechnerorganisation [3], [1] sind logische Variablen 1 Bit. Auch in höheren Programmiersprachen gilt: Der entsprechende Datentyp entspricht einem Bit.
- Die Maschinendatendarstellung von binären Variablen im Prozessor unterscheidet sich davon:
  - Vektor von n (hier 32) binären Einzelvariablen, geordnet.
  - Operation für alle n Bit gleichzeitig: Bit-Positionen mit gleichem Index werden zu einem Ergebnisbit mit diesem Index entsprechend der angegebenen Operation verknüpft.

Die Datendarstellung wird in Bild 4.6 S.69 erläutert.



Operation * kann UND, ODER oder EXKLUSIV-ODER sein

○ Inhalt Bit_i

Bild 4.6: Datendarstellung bei Logikbefehlen

OR

Bit-weise Oder-Verknüpfung zweier Operanden

OR reg32, reg32

Die gleichwertigen Bit (siehe Bild 4.6 S.69) der beiden Operanden werden mit einem logischen Oder verknüpft und das Ergebnis an der Stelle des ersten Operanden gespeichert.

CF und OF werden gelöscht, ZF und SF werden entsprechend dem Ergebnis gesetzt.

Beispiel
OR EAX, EBX

AND
Bit-weise Und-Verknüpfung zweier Operanden

Die gleichwertigen Bit der beiden Operanden werden mit einem logischen Und verknüpft und das Ergebnis an der Stelle des ersten Operanden gespeichert.

XOR
Bit-weise Exklusiv-Oder-Verknüpfung (Antivalenz) zweier Operanden

Die gleichwertigen Bit der beiden Operanden werden mit einem logischen Exklusiv-Oder verknüpft und das Ergebnis an der Stelle des ersten Operanden gespeichert.

NOT
Bit-weise Negation des Operanden, logisches NICHT, nur ein Eingangsoperand, reg32-Ergebnis ist im gleichen Register.

Zusätzlich zu den Logikbefehlen ist für die Behandlung von 1-Bit-Variablen notwendig:

Zuordnung zu genau einer Bit-Position (typ. Bit_0). Dann ist notwendig: Bit_{n-1} … Bit_1 (alle) = 0.

Bsp.:

E-Operand1: EAX, Bit_0
E-Operand2: EBX, Bit_0
Ergebnis: EAX, Bit_0

```
AND EAX,00…01B    ; Ausblenden der oberen Bit 31 bis 1
AND EBX,00…01B    ; Ausblenden der oberen Bit 31 bis 1

OR EAX,EBX        ; eigentliche Operation Oder
```

Für eine Folge von Operationen mit den 1-Bit-Datentypen gilt: -> Ab der 2. Operation können die (AND reg32,00…01B) auf das Ergebnisregister als neuer E-Operand entfallen.

Bitbefehle

Sie dienen im Gegensatz zu den bisher behandelten Logikbefehlen zur Behandlung einzelner ausgewählter Bit im Datenwort.

BTR, BTS

BTR: Test und Rücksetzen eines Bit, BTS: Test und Setzen eines Bit

BTR reg32, const8
Die Auswahl der Bit-Position erfolgt durch const8.

BTS reg32, const8

Das Bit auf der durch den zweiten Operanden angegebenen Position wird im ersten Operanden rückgesetzt (=0; BTR) bzw. gesetzt (=1; BTS).

Bitsetzen, -rücksetzen: Angegebene Bit-Position wird 1 bzw. 0, der Wert vor der Veränderung wird in CF (Achtung, anders als bei OR, AND, EXOR, NOT) abgelegt und ist damit auswertbar (d.h. hier ist Setzen/Rücksetzen immer mit Test verbunden). Die anderen Flags sind unbestimmt.

BT (reiner Test ohne Veränderung)

BT reg32, const8

Testen eines Bit

Das Bit auf der durch den zweiten Operanden angegebenen Position wird im ersten Operanden getestet (keine Änderung).

Weiterhin gibt es auch in den Arithmetik-Logik-Befehlen:

➔ Konvertierungen (zwischen verschiedenen Maschinendatenformaten, siehe [3], [2]).

4.4. Schiebe- und Rotationsbefehle

Die Datendarstellung ist entsprechend NOT:

Vektor aus Bit-Positionen, geordnet nach Index (Bit_{31} … Bit_0); Bild 4.7 S.74: Eingabe-Operand.

SHR, SHL (Schieben rechts/links)

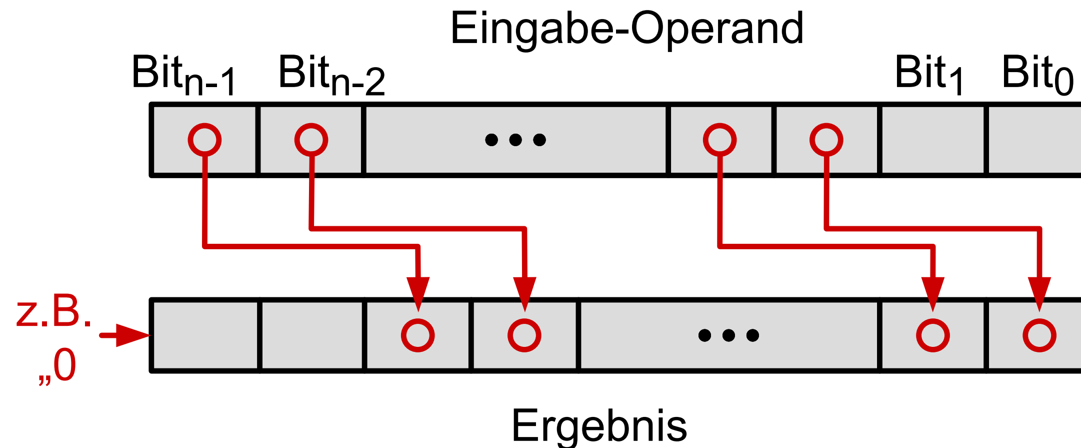
Bitverschiebung innerhalb des ersten Operanden

SHR reg32, const8

SHL reg32, const8

const8: Anzahl der Positionen

Beispiel: 2 mal schieben nach rechts



n mal schieben nach **rechts** — Division durch 2^n

n mal schieben nach **links** — Multiplikation mit 2^n

Bild 4.7: Prinzip der Schiebeoperation

Die Bit des ersten Operanden werden um die im zweiten Operanden angegebene Anzahl Stellen nach links (SHL) bzw. rechts (SHR) verschoben. Dabei ist das höchstwertige Bit ganz links. Freiwerdende Bit werden mit Null aufgefüllt. Herausgeschobene Bit werden durch das Carry-Flag geschoben.

OF ist nach der Operation undefiniert, ZF und SF sind entsprechend dem Ergebnis gesetzt. CF erhält den Wert des zuletzt herausgeschobenen Bit.

Besonderheit: Bit, die herausgeschoben werden, werden i. Allg. nicht verwendet (Ausnahmen siehe [4]).

Bit, die hineingeschoben werden: vorgebbarer Festwert (Varianten siehe [4]).

Durch Einschieben von 0 -> Arithmetische Interpretation:

Division, Multiplikation mit Zweierpotenzen (Bild 4.7 S.74 unten).

Achtung Zahlenbereich: Multiplikation: die vorderen Bit, Division: die hinteren Bit gehen verloren.

ROR, ROL

Bitrotation innerhalb des ersten Operanden

Die Bit des ersten Operanden werden um die im zweiten Operanden angegebene Anzahl Stellen nach links (ROL) bzw. rechts (ROR) rotiert. Dabei ist das höchstwertige Bit ganz links. Bit, die den Operanden verlassen, werden auf der anderen Seite wieder eingefügt (in der gleichen Reihenfolge). Gleichzeitig werden diese teilweise in das Carry-Flag kopiert.

ZF und SF bleiben unverändert. OF ist nach der Operation undefiniert. CF erhält den Wert des zuletzt herausgeschobenen Bit.

Rotieren (rechts/links, Bsp. Bild 4.8 S.76):

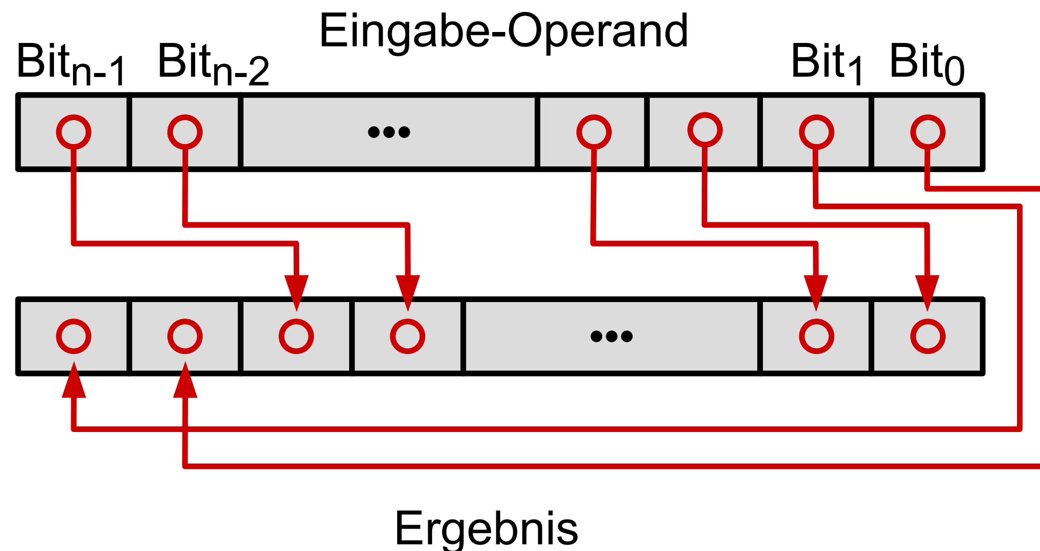


Bild 4.8: Prinzip der Operation Rotieren

Es ist keine arithmetische Interpretation sinnvoll. Aber die Befehle sind z.B. für die Vorbereitung logischer Operationen möglich.

Bsp.: EAX, Bit_2 mit EBX, Bit_1 verknüpfen, Ergebnis in Bit_{31} von EAX

```
ROL EAX, 29          ; oder ROR EAX, 3
ROL EBX, 30          ; oder ROR EBX, 2
OR EAX, EBX          ; gewünschte Beispieloperation
AND EAX, 10 … 00B ; evtl. für Nullsetzen Bit30 … Bit0 vom EAX
```

4.5. Programmtransferbefehle

Befehle aus den Abschnitten 4.2, 4.3 und 4.4 erzeugen als Folgebefehlsadresse (von $Befehl_k$) i. Allg. die des unmittelbar folgen Befehls ($Befehl_{k+1}$):

$$BA\ (Befehl_{k+1}) = BA\ (Befehl_k) +1$$

Bei 32-Bit-Wortadressierung (siehe Bild 3.2 S.35) wäre der Adressabstand von zwei aufeinander folgenden Maschinecodes im Programmspeicher +1, bei Byteadressierung und 32-Bit-Befehlscode wäre der Abstand +4 (siehe Bild 3.3 S.37):

$$BA\ (Befehl_{k+1}) = BA\ (Befehl_k) +4$$

Der Abstand +1 wird im Folgenden bei der Erläuterung der Prinzipien verwendet, der Abstand +4 bei den konkreten Beispielbefehlen.

Mit der linearen Erhöhung der Befehlsadresse mit BA := BA+1 sind in Bild 4.9 (a/b) S.78/80 nur die schwarzen Befehlsfolgen zwischen Befehl $_{i+1}$ und Befehl $_{i+n}$ bzw. zwischen Befehl $_k$ bis Befehl $_{k+m}$, rein sequentiell, möglich.

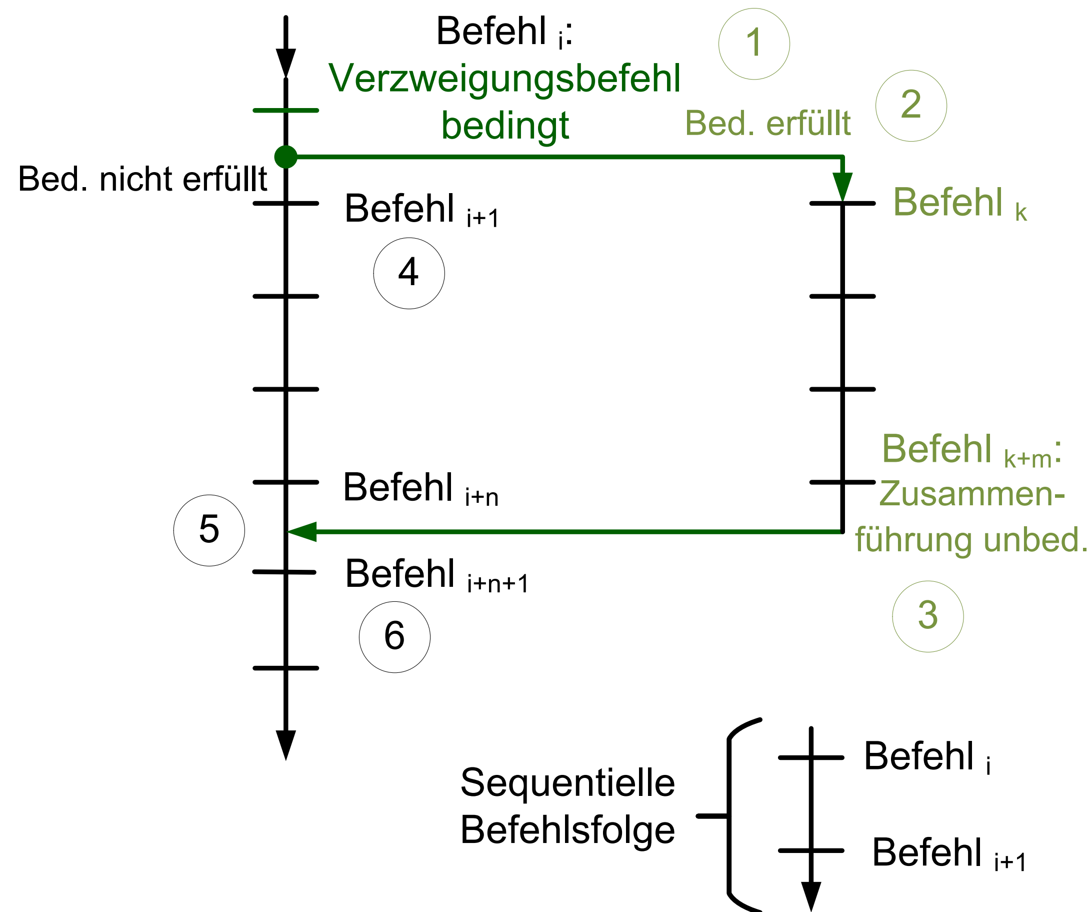


Bild 4.9 (a): Befehlsfolgen mit Verzweigung und Zusammenführung (als Programmliniendarstellung)

Bei bedingter Verzweigung (bedingter Sprung, Bilder 4.9 (a/b)) S.78/80 und unbedingter Zusammenführung (unbedingter Sprung) entsteht bei nicht erfüllter Bedingung als Ablauf:

(1) Verzweigungsbefehl (Befehl auf Adresse i), Bedingung (Bed.) nicht erfüllt.

(4) Da die Bedingung nicht erfüllt ist, folgt auf den Befehl $_i$ der nächste Befehl $_{i+1}$, der unmittelbar im Programmspeicher folgt.

(5) Der letzte Befehl des linearen Befehlsablaufs ist Befehl $_{i+n}$.

(6) Es folgt als nächster Befehl $_{i+n+1}$.

Für einigermaßen praktisch relevante Algorithmen gilt: Notwendig ist auch die Verzweigung von Befehl $_i$ zu Befehl $_k$, bedingt (in Bild 4.9 (a,b) S.78/80 der grüne Verlauf bei erfüllter Bedingung, bedingter Sprung). Ergänzt wird das durch die Zusammenführung, fest (Befehl $_{k+m}$ zu Befehl $_{i+n+1}$) -> unbedingter Sprungbefehl:

(1) Verzweigungsbefehl (Befehl auf Adresse i), Bedingung (Bed.) erfüllt.
(2) Es folgt nicht der Befehl $_{i+1}$, sondern der Befehl $_k$, der erste Befehl des alternativen Programmablaufs.
(3) Der letzte Befehl des alternativen Programmablaufs ist der Befehl $_{k+m}$, ein unbedingter Zusammenführungsbefehl.

(6) Es folgt als nächster Befehl nicht Befehl $_{k+m+1}$, sondern der Befehl $_{i+n+1}$ aus dem linearen Programmablauf.

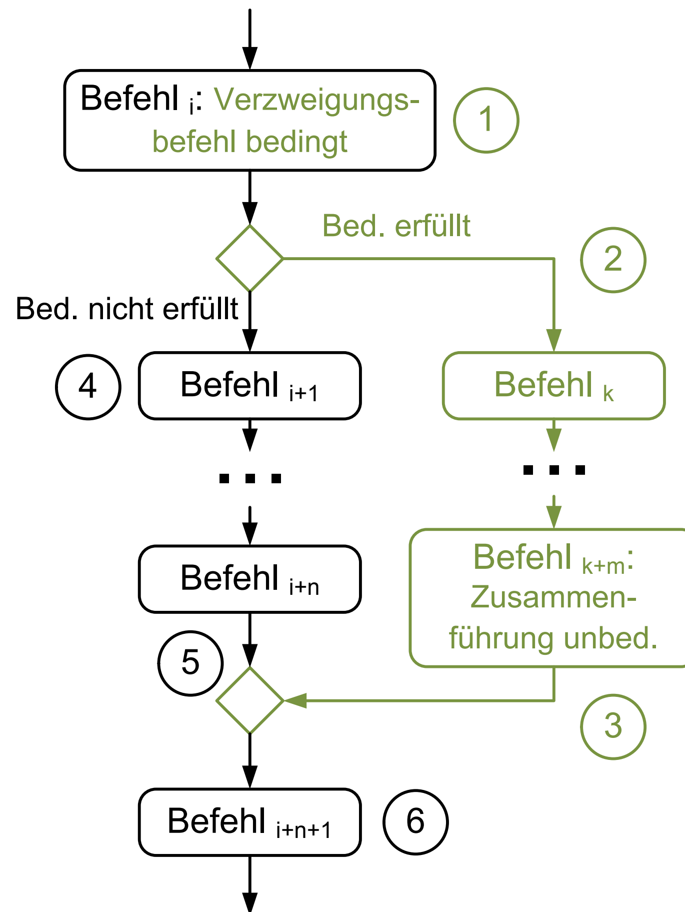


Bild 4.9 (b): Befehlsfolgen mit Verzweigung und Zusammenführung (als Aktivitätsdiagramm)

Die Struktur entspricht der Konstruktion if Bedingung then … else …, wobei das Teilprogramm nach then dem rechten Ablauf und das nach else dem linken Ablauf zuzuordnen ist.

Bild 4.9 (a) S.78 zeigt das Verhalten mittels einer Programmliniendarstellung und Bild 4.9 (b) S.80 mit einer alternativen Aktivitätsdiagrammdarstellung (UML).

Sprungbefehle

Sie realisieren die beschriebenen Programmabläufe bei Verzweigung und Zusammenführung.

Unbedingte Sprünge (Zusammenführung):
Nach Befehl $_i$ folgt immer Befehl $_k$, wobei k nicht abhängig von i ist.

Der Aufbau des Maschinencodes von Sprungbefehlen benutzt hier zweimal 4 Byte (aufgrund der hier angenommenen Byteadressierung):

Programmspeicheradresse von Befehl $_i$:	32 Bit Code (Befehl $_i$: Sprung)
Programmspeicheradresse von Befehl $_i$ +4:	32 Bit Sprungziel (Programmspeicheradresse von Befehl $_k$)

Ablauf:

(1) Befehl lesen: Maschinencode zum Prozessor -> Erkennung Sprungbefehl.

(2) Befehlsadresse erhöhen um +4.
(3) Sprungziel lesen von der aktuellen (in (2) erhöhten) Befehlsadresse.
(4) Sprungziel als nächste Befehlsadresse benutzen.

JMP

(Jump)

Unbedingter Sprung

JMP label

Die Programmausführung wird mit der durch das Label gekennzeichneten Anweisung fortgesetzt.

Alle Flags bleiben unverändert.

Beispiel

JMP m1

Sprungbefehle bedingt

Es wird bei erfüllter Bedingung wie bei unbedingt gesprungen. Sonst wird mit dem unmittelbar folgenden Befehl fortgesetzt.

Bedingungen: Flags, negiert oder unnegiert.

Bei nicht erfüllter Bedingung: weiter mit aktueller Befehlsadresse +8

(32 Bit-Maschinencode bei Byteadressierung).

Komplexere Bedingungen muss man auf elementare Befehlsfolgen zurückführen.

JNZ, JZ, JNC, JC

(Jump if not zero, …)

Bedingte Sprünge (Auswahl)

Bedingungen: NZ = Zero-Flag nicht gesetzt, Z = Zero-Flag gesetzt,

NC = Carry-Flag nicht gesetzt, C = Carry-Flag gesetzt.

Unterprogrammbefehle

- Unterprogramm: mehrfach nutzbares Teilprogramm, das von beliebigen Stellen in einem Maschinenprogramm aufgerufen werden kann.
- Sie sind z.B. für Funktionsaufrufe im Algorithmus notwendig.

Bsp.
x:= **sin** (y) (sin ist typ. kein Maschinenbefehl)

Falls **sin** mehrfach genutzt wird:
sin als Unterprogramm mit zusätzlichem, durch weitere Befehle realisiertem, Mechanismus zur Übergabe von y und Rückgabe von x.

Das Prinzip für den Unterprogrammaufruf (CALL) und die Rückkehr aus dem Unterprogramm (RETURN oder RET) zeigen die Bilder 4.10 (a/b) S.85/86. Das Unterprogramm wird ähnlich eines unbedingten Sprungs für feste Adresse aufgerufen (Sprung auf seine erste Befehlsadresse). Die Rückkehr am Unterprogrammende soll auf die Befehlsadresse erfolgen, die auf die Befehlsadresse des aktuell aufrufenden CALL folgt. Die Adresse ist variabel, d.h. abhängig, von wo das Unterprogramm aufgerufen wurde. Sie wird im Stack zwischengespeichert (siehe auch Bild 4.4 S.54).

Im Bild 4.10 (a) S.85 wird mit Programmliniendarstellung oder alternativ mit dem Aktivitätsdiagramm (UML) in Bild 4.10 (b) S.86 ein Ablauf für den zweimaligen Aufruf eines Unterprogramms von zwei verschiedenen Stellen in einem Hauptprogramm dargestellt:

(1) Der CALL-Befehl (Befehl $_i$) im Hauptprogramm speichert die Befehlsadresse von Befehl $_{i+1}$ in den Stack und springt auf den Befehl $_k$.

(2) Das Unterprogramm wird für diesen Aufruf ab Befehl $_k$ durchlaufen.

(3) Am Unterprogrammende liest der RET-Befehl die Rückkehradresse (aktuell die Befehlsadresse von Befehl $_{i+1}$) aus dem Stack und springt dadurch auf den Befehl $_{i+1}$.

(4) Das Hauptprogramm wird ab Befehl $_{i+1}$ weiter bis zum Befehl $_j$ durchlaufen.

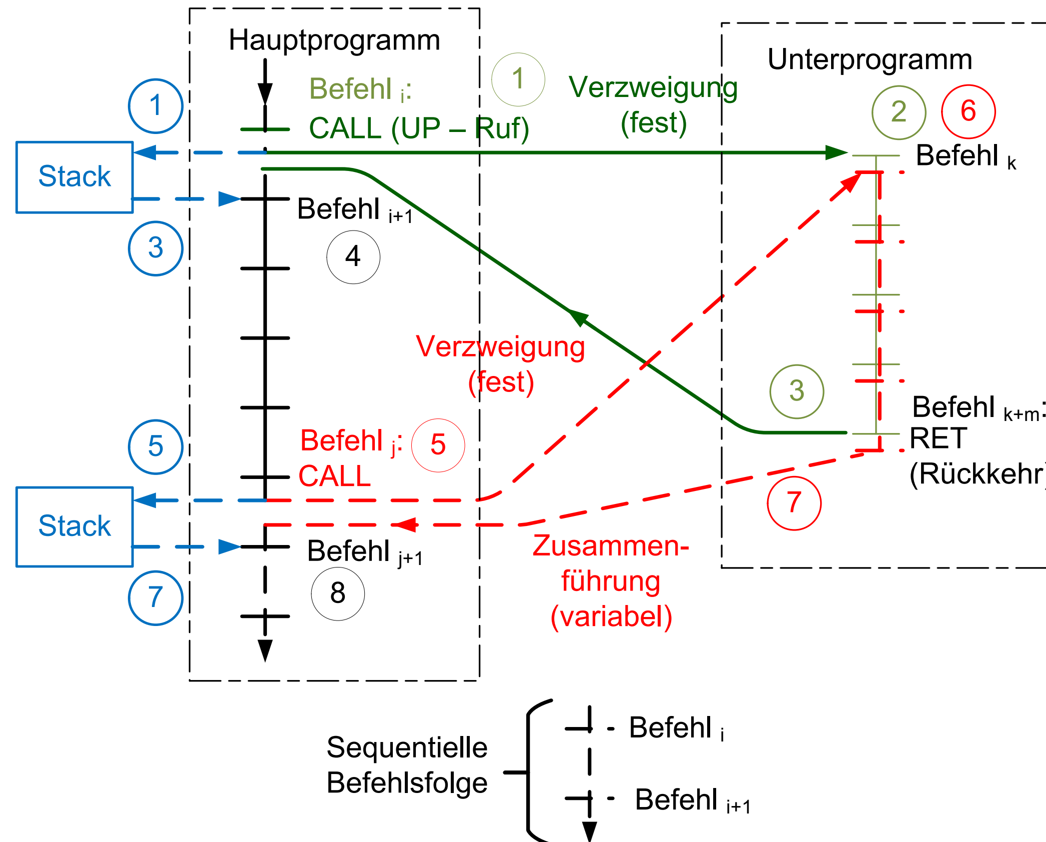


Bild 4.10 (a): Befehlsfolgen mit Unterprogrammaufruf und Rückkehr (als Programmliniendarstellung)

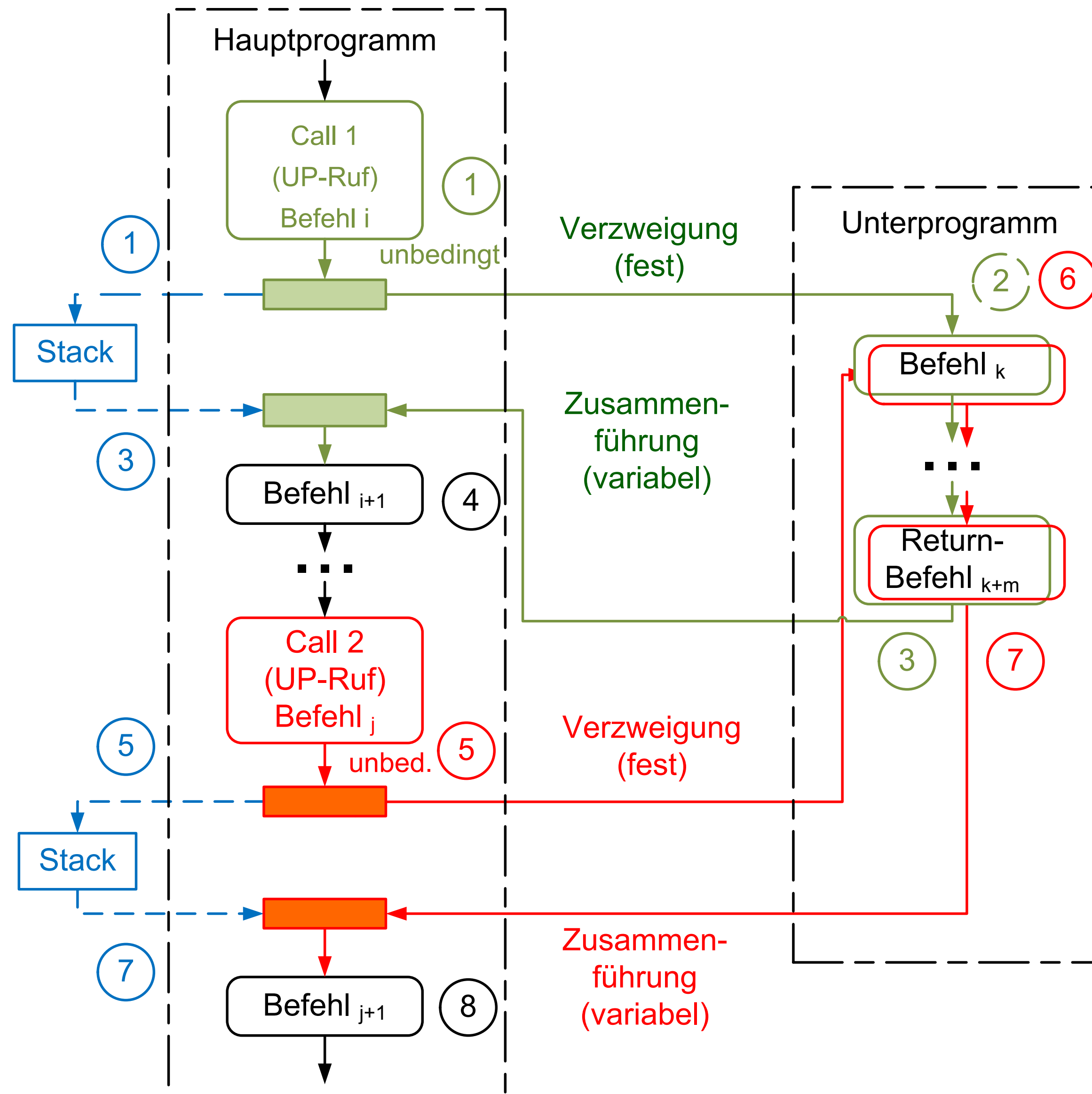


Bild 4.10 (b): Befehlsfolgen mit Unterprogrammaufruf und Rückkehr (als Aktivitätsdiagramm)

(5) Der CALL-Befehl (Befehl $_j$) im Hauptprogramm speichert die Befehlsadresse von Befehl $_{j+1}$ in den Stack und springt auf den Befehl $_k$.

(6) Das Unterprogramm wird für diesen Aufruf ab Befehl $_k$ durchlaufen.

(7) Am Unterprogrammende liest der RET-Befehl die Rückkehradresse (aktuell die Befehlsadresse von Befehl $_{j+1}$) aus dem Stack und springt dadurch auf den Befehl $_{j+1}$.

(8) Das Hauptprogramm wird ab Befehl $_{j+1}$ weiter durchlaufen.

CALL

Unterprogrammaufruf

CALL label

Die Programmausführung wird mit der durch das Label gekennzeichneten Anweisung fortgesetzt, nachdem die Adresse des dem CALL folgenden Befehls auf dem Stack abgelegt wurde (siehe PUSH). Die Rückkehr zu dieser Adresse kann später durch den Befehl RET erfolgen.

Alle Flags bleiben unverändert.

Beispiel

CALL up1

CALL (UP-Ruf) arbeitet wie ein (unbedingter) Sprung. Zusätzlich wird die Rückkehradresse (Befehlsadresse vom CALL-Befehl + 8 (Grund für +8 siehe Sprung S. 81) in den Stack mit Stackpointer-Behandlung geschrieben (siehe Stack-Schreiben bei Stack-Befehlen bzw. Bild 4.4 S. 54).

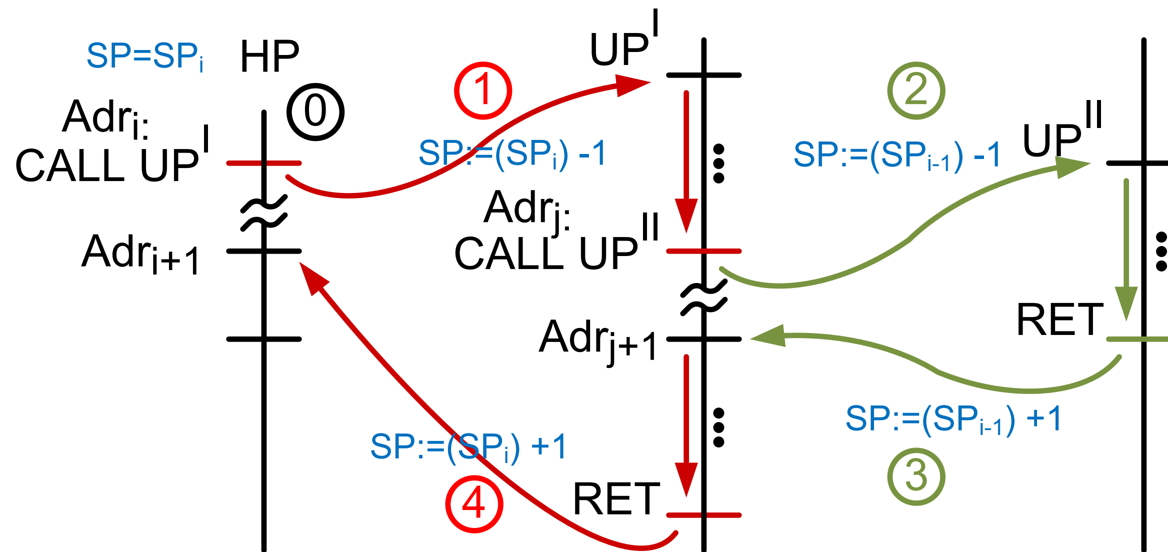
RET (RETURN, UP-Rückkehr) arbeitet wie ein (unbedingter) Sprung, aber das Sprungziel steht nicht im Maschinencode, sondern wird aus dem Stack gelesen (mit Stackpointer-Behandlung, siehe Stack-Lesen bei Stack-Befehlen bzw. Bild 4.4 S.54).

Der Stackpointer bei RET muss der Stackpointer-Wert nach dem aufrufenden CALL sein.

Im Unterprogramm sind nur Stackpointer-Werte unterhalb dieses Stackpointer-Werts (nach CALL) erlaubt).

➔ In beiden Fällen wird sonst eine ungültige Rückkehradresse verwendet!

In Unterprogrammen können wiederum Unterprogramme aufgerufen werden (verschachtelter Aufruf). Das ist möglich, weil die Rückkehradressen im Stack nach dessen Prinzip abgelegt werden. Bild 4.11 S.89 verdeutlicht das. Es beschreibt folgenden Ablauf:



≈ Abarbeitungszeit der UP, Befehlsadressabstand = +1

Stackadresse	0	1	2	3	4
Sp_i	$Wert_k$	$Wert_k$	$Wert_k$	$Wert_k$	$Wert_k$
Sp_{i-1}	*	Adr_{i+1}	Adr_{i+1}	Adr_{i+1}	*
Sp_{i-2}	*	*	Adr_{j+1}	*	*

* heißt nicht verwendbarer Inhalt, $Wert_k$ ist vom HP erzeugt

Stackpointerveränderung:

SP:	SP_i	SP_{i-1}	SP_{i-2}	SP_{i-1}	SP_i

Bild 4.11: Verschachtelter Aufruf von Unterprogrammen

(0) Vor dem Aufruf CALL UP‘ (Unterprogramm‘) hat der Stackpointer (SP) den Wert SP_i. Auf der Adresse SP_i (im Stackspeicherbereich) befindet sich ein vom Hauptprogramm (HP) erzeugter Wert $Wert_k$. Dieser ist für den weiteren Ablauf nicht relevant. Die Werte unterhalb von SP_i sind zu diesem Zeitpunkt nicht verwendbar (*).

(1) Nach dem CALL UP‘ hat der SP den Wert SP_{i-1} und auf SP_{i-1} wurde durch den CALL die Rückkehradresse Adr_{i+1} eingetragen.

(2) Nach dem CALL UP‘‘ hat der SP den Wert SP_{i-2} und auf SP_{i-2} wurde durch den CALL die Rückkehradresse Adr_{j+1} eingetragen.

(3) In dem RET in UP‘‘ wurde auf die von SP_{i-2} gelesenen Adresse Adr_{j+1} zurückgesprungen. Der Wert auf SP_{i-2} ist nicht mehr verwendbar. Der SP hat den Wert SP_{i-1}.

(4) In dem RET in UP‘ wurde auf die von SP_{i-1} gelesenen Adresse Adr_{i+1} zurückgesprungen. Der Wert auf SP_{i-1} ist nicht mehr verwendbar. Der SP hat den Wert SP_i.

4.6. Sonstige Befehle

➔ Sie sind heterogen und passen nicht in die übrigen Gruppen:

- spezielle Funktionen,
- Veränderung des Steuerzustandes des Prozessors,
- …

Beispiele:

NOP (no operation)

Befehl ohne Wirkung, aber mit Prozessorzeit-Nutzung (Befehl lesen und nächste Befehlsadresse erzeugen (+4)).

Halt/Wait

Prozessor anhalten bis zur erfüllten Wartebedingung.

Mit Befehlen zur **Modussteuerung** kann zwischen Betriebsarten des Prozessors gewechselt werden.

5. Prozessor und prozessorzugeordnete Baugruppen

5.1. Einordnung in die Gesamtarchitektur

Aufgaben des Prozessors:

- Verknüpfung von Eingangsoperanden zu Ausgangsoperanden (Ergebnisoperanden) in Abhängigkeit von (Maschinen-) Befehlen.
- Hilfsfunktionen:
  - Steuerung der Befehlsreihenfolge und Lesen von Befehlen (Reihenfolge legt das den Algorithmus realisierende Programm fest),
  - Lesen und Schreiben von Operanden.

Bild 5.1 S.94 zeigt eine Blockstruktur. Hier wird die Princeton-Architektur verwendet, d.h. alle Zugriffe des Prozessors laufen über einen einheitlichen Systembus (Bild 5.1 S.94, siehe auch Bild 3.1 S.32).

Bestandteile:

Systembus: Verbindung der einzelnen Blöcke untereinander, genauer: Verbindung zwischen dem Prozessor und den anderen Blöcken über

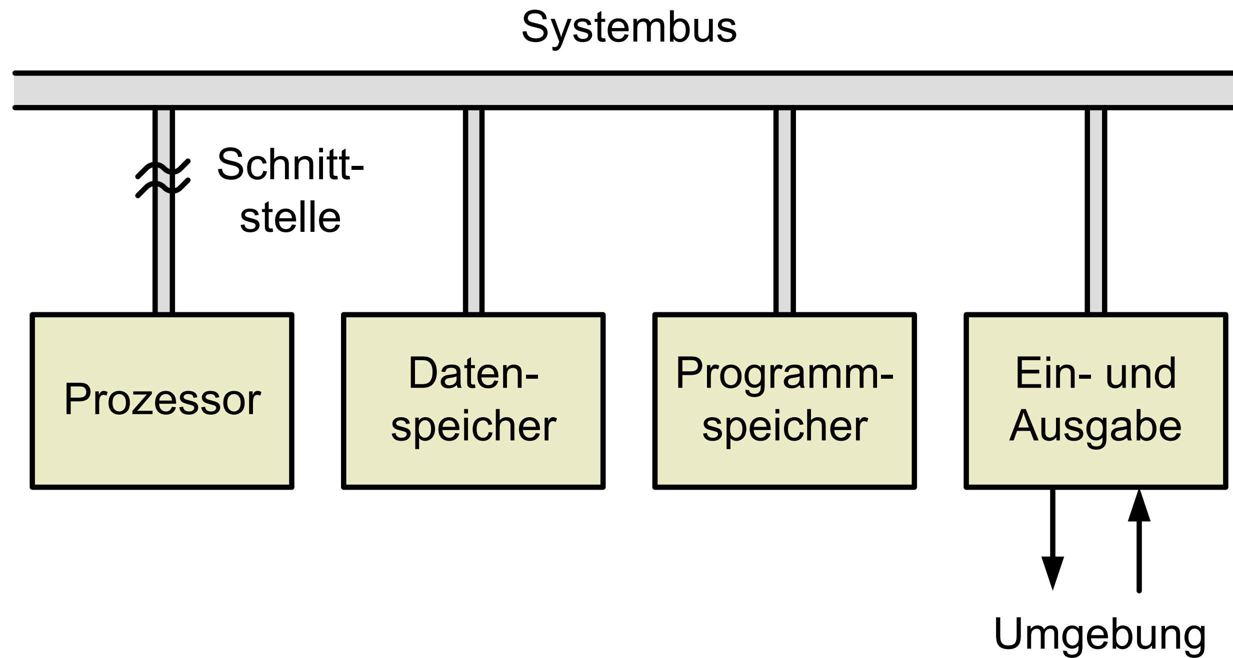


Bild 5.1: Grundarchitektur mit Systembus

- Datenbus (DB): Übertragung von Befehlscodes (vom Programmspeicher zum Prozessor) und Übertragung von Operanden zwischen dem Datenspeicher bzw. der Ein-/Ausgabe (EA) und dem Prozessor.

- Adressbus (AB): Übertragung von Adressen (vom Prozessor zu den Speichern und EA).
  Es gilt:
  ➔ Datenbustransfers benötigen immer auch Adressbustransfers.

➔ Bei Daten- und Befehlscodetransfers ist immer der Prozessor der Initiator und legt auch immer den Transferort über eine Adresse fest. Deshalb ist die Richtung (für den AB) prinzipiell vom Prozessor zum Speicher bzw. zur EA (unidirektional).

- Steuerbus (zusammengefassten Steuersignale, SB): Organisation des logisch-zeitlichen Ablaufs der Daten- bzw. Adressübertragung (und evtl. weiteres).

Warum Bus als Bezeichnung?

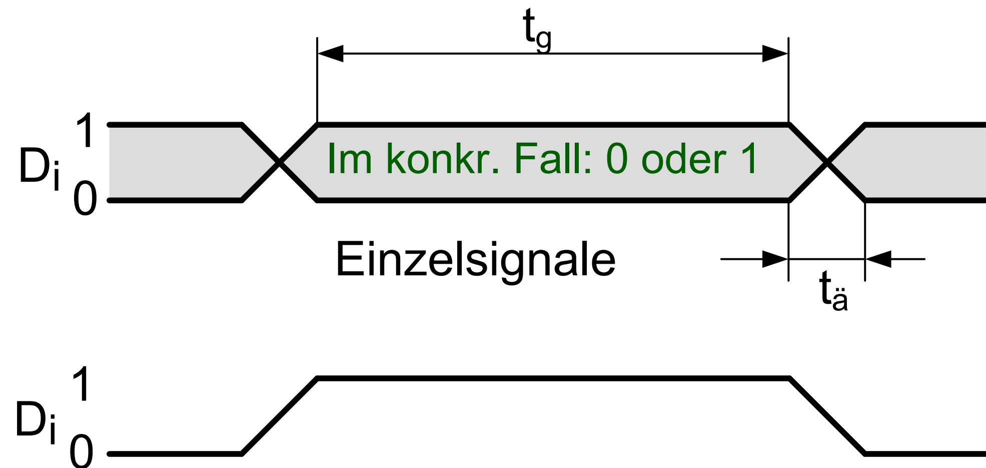
➔ Ein Bus ist im allgemeinen Sprachgebrauch ein "öffentliches Personennahverkehrsmittel".

Ein Rechnerbus kann in Anlehnung dazu als „öffentliches Datennahverkehrsmittel" bezeichnet werden:

- Öffentlich: Alle Funktionseinheiten (Prozessor, Programm- und Datenspeicher, EA-Schnittstellen) können darauf zugreifen.
- Daten: Befehlscodes, Operanden,
- Verkehrsmittel: Verbindungsstruktur für den Transport von Daten,
- Nahverkehr: innerhalb des Rechners.

Im Weiteren werden Einzelsignale oder zu (Teil-) Bussen zusammengefasste Signale wie in Bild 5.2 S.96 und Bild 5.3 S.98 dargestellt.

Einzelsignale: Sie gibt es in beide (Wirkungs-) Richtungen, jedes Einzelsignal hat nur eine (unidirektional) oder beide Richtungen (bidirektional). Erstes wird im Steuerbus verwendet, der eine Zusammenfassung von Einzelsignalen mit unterschiedlicher Wirkungsrichtung, je nach Signal, darstellt. Die Darstellungsvarianten zeigt Bild 5.2 S.96.



Einzelsignale, vorgegebene Belegung 0 oder 1

t_g: zeitlicher Gültigkeitsbereich (Datenfenster)

$t_ä$: zeitlicher Änderungsbereichbereich

Bild 5.2: Darstellungsvarianten von Einzelsignalen

Die obere Darstellung wird verwendet, wenn der konkrete Wert (0, 1) keinen direkten Einfluss auf den logisch zeitlichen Ablauf des beschriebenen Verhaltens hat. Andernfalls wird der zeitliche Ablauf von 0-1-Folgen verwendet (unterer Verlauf). Der Änderungsbereich mit kurzeitig nicht gültigem Wert entsteht durch das elektronische Verhalten der Logikelemente und Verbindungen.

Die Darstellung von Bussen (Zusammenfassung von Einzelsignalen mit gleichem logischen Charakter) zeigt Bild 5.3 S.98 am Beispiel des Datenbusses.

Oben: Datenbus-Einzelsignale mit Belegung 0 oder 1 je Signal, nicht konkretisiert, gleicher Änderungszeitpunkt aller Einzelsignale (typisch für Daten- und Adressbus).

Unten: Zusammenfassung aller Einzelsignale zum Bus, nicht konkretisiert in der Belegung.

Es gilt für Daten- und Adressbus:

- Beide Busse bestehen aus n Einzelsignalen (hier n = 32).
- Die Wertigkeit des Einzelsignals entspricht der Stellung im Bus (z.B. A_i bedeutet Wert (0 oder 1) * 2^i).
- Änderungs-/Gültigkeitszeitpunkte ($t_{kä}$, $t_{kä+1}$, t_k, t_{k+1} …) sind für alle Einzelsignale des Teilbusses gleich.

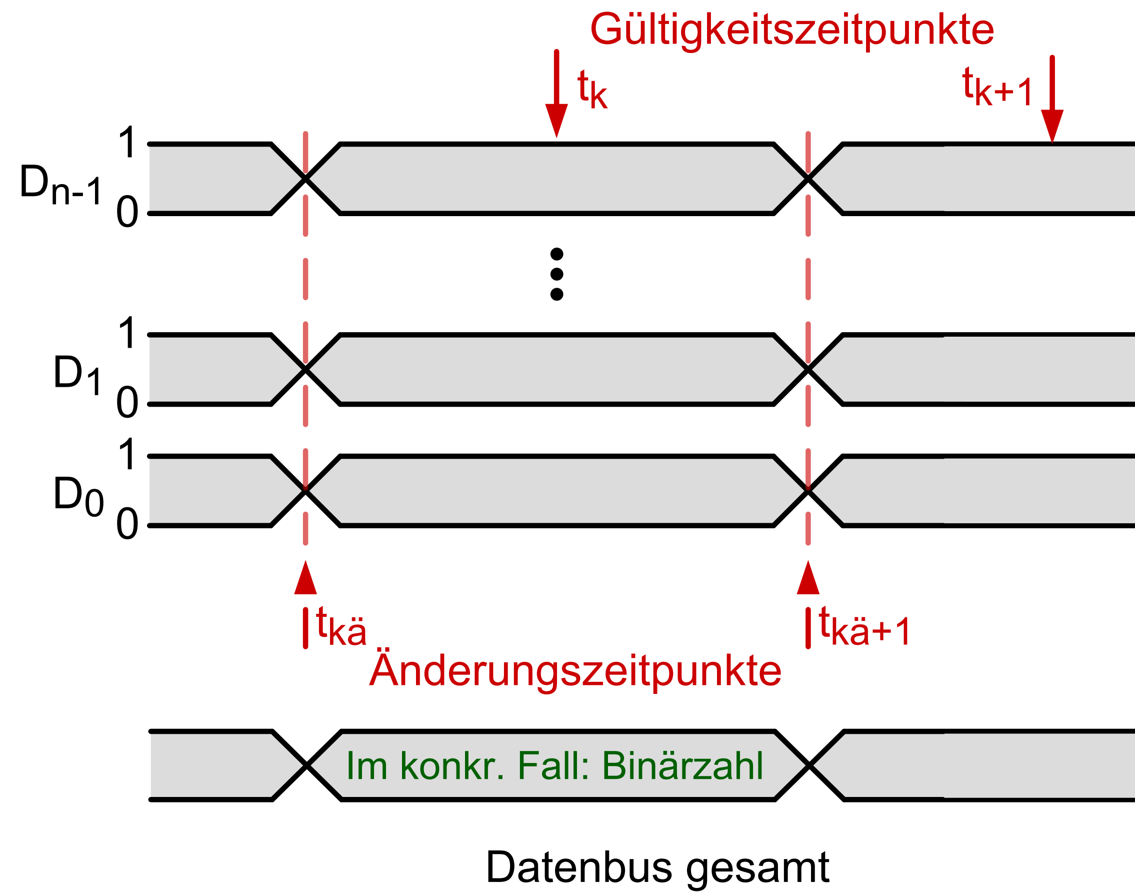


Bild 5.3: Darstellung von Bussen (Bsp. Datenbus)

5.2. Prozessorgrundstruktur

➔ Vorbemerkung: Hier erfolgt die Behandlung einer Struktur, die prinzipiell geeignet ist, Programme zur Manipulation von Operanden auszuführen.

In einfachen Einchip- (Micro-) Controllern ist diese Struktur noch weitgehend so elementar enthalten, in Hochleistungsprozessoren ist sie zwar prinzipiell vorhanden, aber mit einer größeren Anzahl Ergänzungen und Erweiterungen versehen (siehe Abschnitt 8 bzw. in der Literatur).

Die Grundstruktur zeigt Bild 5.4 S.100. Ausgangspunkt ist das (bezüglich der Register reduzierte) Rechenwerk (Data Path) von Bild 3.4 S.39. Dieses wird erweitert um ein Steuerwerk (Control Unit).

Erläuterung des Steuerwerks:

- AST (Ablaufsteuerung): steuert den logisch-zeitlichen Ablauf des Befehls im Prozessor. Sie ist ein (synchroner) digitaler Automat (siehe [4]). Varianten zum logisch-zeitlichen Ablauf der AST folgen in Abschnitt 5.3.
- BA (Befehlsadressregister, auch IP, Instruction Pointer): Enthält die Programmspeicheradresse des aktuellen Befehls. BA wird bei den Befehlsgruppen für lineare Befehlsfolge fest erhöht (bei 32-Bit-Befehlscodes und Byteadressierung jeweils um 4, siehe Abschnitt 4.5).

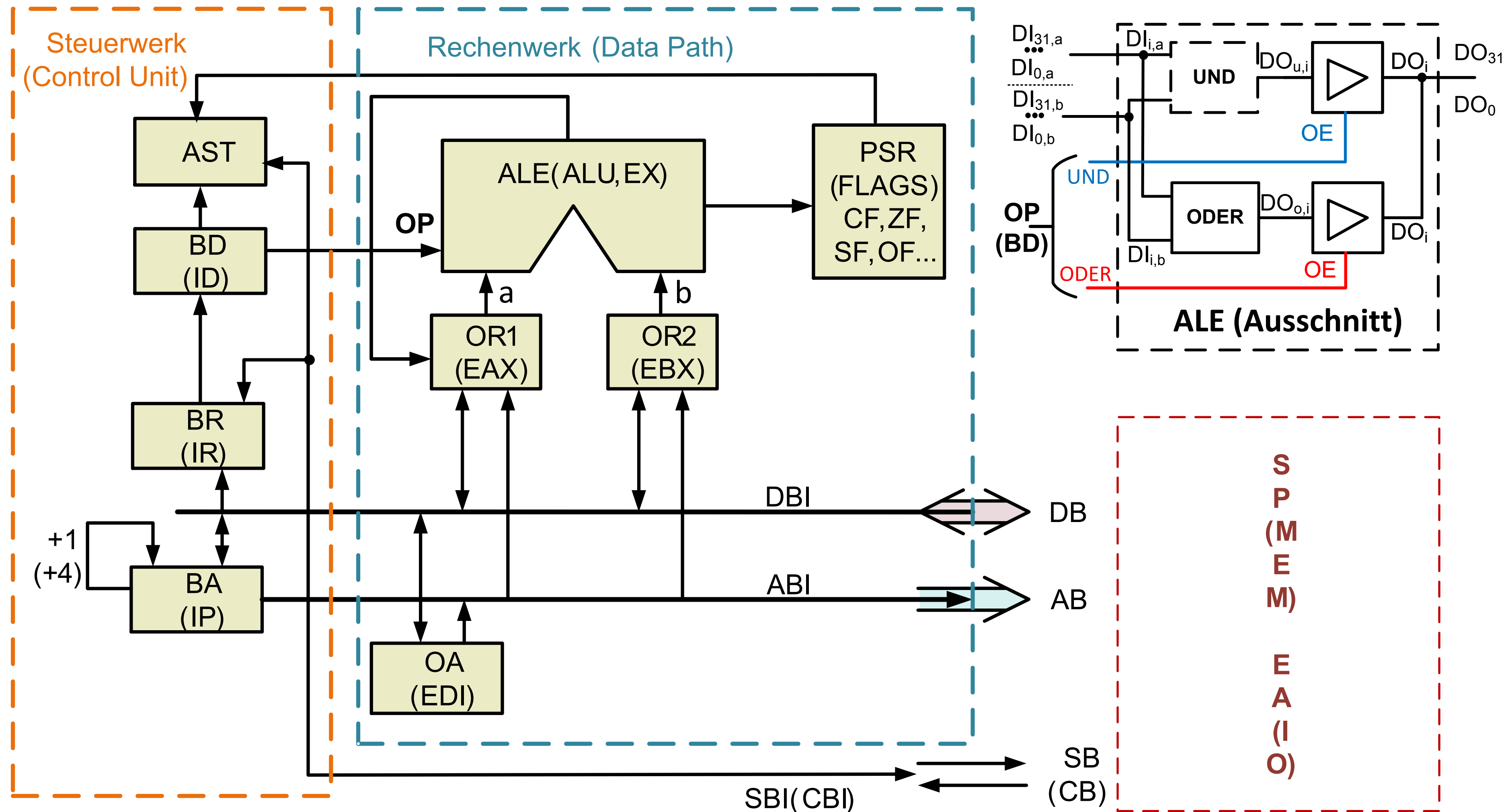


Bild 5.4: Prozessorgrundstruktur

Bei JMP und CALL wird BA auf Label (feste zugeordnete Programmspeicheradresse) gesetzt (evtl. nur bei erfüllter Bedingung), bei RET auf den aus dem Stack gelesenen Wert).

- BR (Befehlsregister, auch IR (Instruction Register), speichert den Maschinencode des aktuellen Befehls,
- BD (Befehlsdekoder, Instruction Decoder, ID) benutzt diesen Maschinencode und dekodiert ihn zu den Steuersignalen für die AST (Ablaufsteuerung) und die ALE (Arithmetik-Logik-Einheit über OP, Operation).

Zum Rechenwerk (eingeführt bereits in Abschnitt 3):

Die ALE führt die in den Abschnitten 4.3 und 4.4 beschriebenen Befehle als Maschinencode aus. Sie wird durch eine komplexe kombinatorische Logik realisiert. Dabei werden die durchzuführenden Operationen auf die beiden (oder den einen, je nach Operation) Eingangsoperanden durch den binären Steuerungsvektor OP (in Bild 5.4 S.100 bzw. 5.5 S.103) eingestellt. Das bedeutet, dass Teile der ALE aktiviert werden, die diese Operation durchführen. Die anderen Teile der Kombinatorik sind dabei passiviert. Bild 5.4 S.100 zeigt oben rechts ein sehr vereinfachtes Bild einer ALE (hier nur für die Befehle UND und ODER). Die Ausgänge ($DO_{u,i}$, $DO_{o,i}$) der mit den Eingangssignalen DI_i gleichzeitig arbeitenden 32-Bit-Logikblöcke werden wahlweise mit den OP-Teilsignalen UND oder ODER über die nachfolgenden Tristate-Treiber (Prinzip siehe Abschnitt 6.1, Bild 6.6 S.151) alternativ auf die ALE-Ausgänge DO_i geschaltet.

Für arithmetische Befehle entstehen allerdings sehr viel komplexere Strukturen.

Die Register im Prozessor enthalten aktuell oft benötigte Operanden und evtl. Adressen oder Steuerinformationen.

Letztere befinden sich im PSR (Prozessorstatusregister, Flags) mit den wichtigsten Flags:

- Für Zahlenbereichsproblematiken: CY, OV
- Für Ungleichungen: S, Z

➔ Die Flags sind durch bedingte Befehle auswertbar (z.B. bedingte Sprungbefehle (behandelt im Abschnitt 4.5).

Zur Verbindung mit den externen Komponenten Speicher (SP, Memory, MEM) und Ein-/Ausgabe (EA, Input Output, IO) dienen die bereits in Abschnitt 5.1 benannten und kurz erläuterten Adress-, Daten- und Steuerbusse. Dabei überträgt der

- Adressbus (AB, hier intern mit ABI bezeichnet) die Adresse zur Auswahl des Speicherortes (Wort oder Byte) bzw. der EA-Schnittstelle, der
- Datenbus (DB, DBI) den Maschinenbefehlscode vom Programmspeicher zum Prozessor bzw. Operanden von oder zum Datenspeicher und der
- Steuerbus (SB, SBI, Control Bus, CB, CBI) die Signale, die den logisch-zeitlichen Ablauf der Zugriffe des Prozessors nach außen realisieren.

Die Grundstruktur, erweitert auf den vollen Registersatz aus Abschnitt 3 (Registermodell, Bild 3.4 S.39) zeigt Bild 5.5 S.103. Er wurde ein temporäres Operandenadressregister (OA-

tmp) eingefügt, welches bei MOVE-Befehlen mit Ziel oder Quelle im Speicher und Angabe der Speicheradresse im Befehlscode diese während der Ausführung zwischenspeichert.

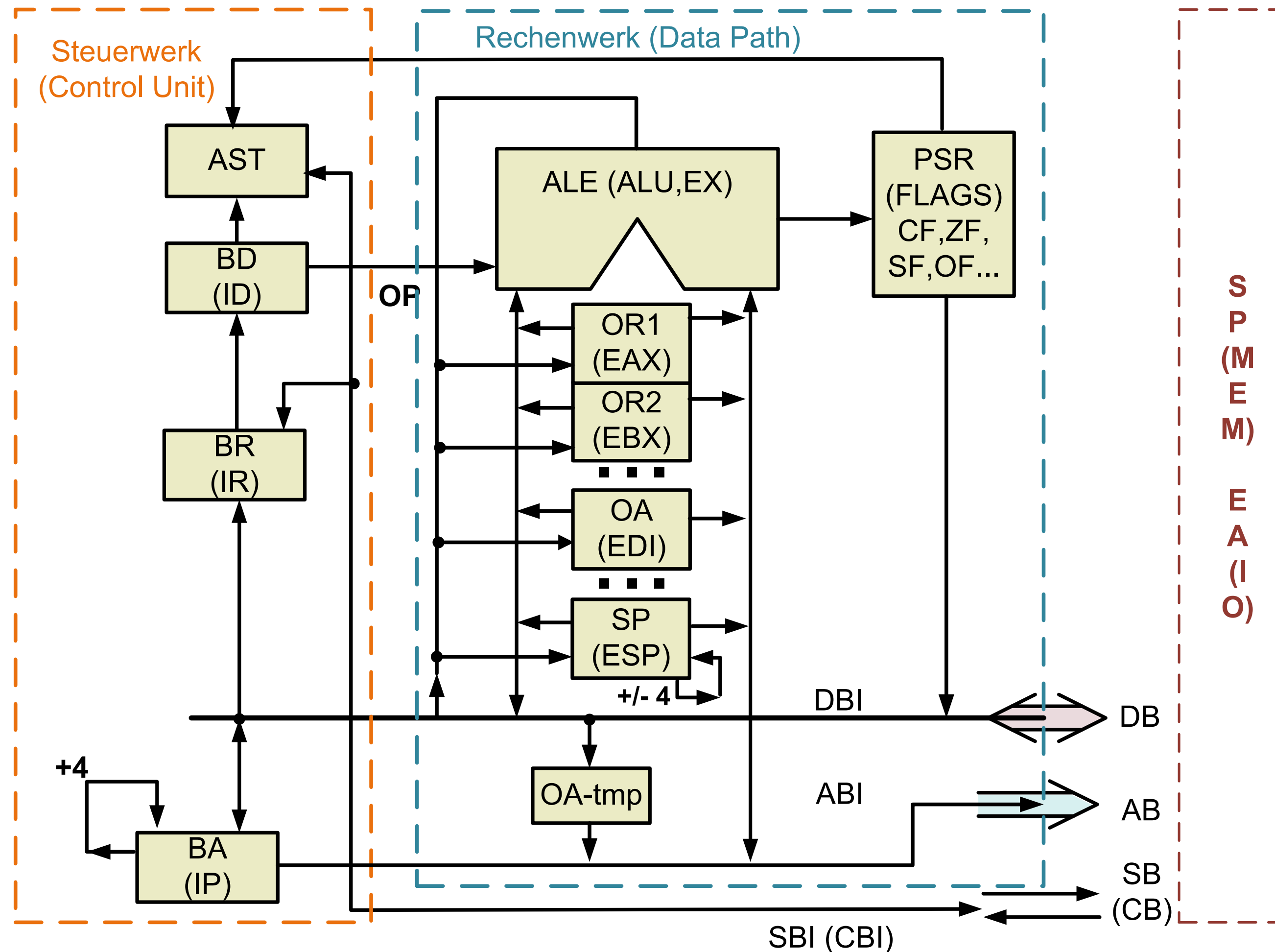


Bild 5.5: Prozessorgrundstruktur mit erweitertem Registersatz

5.3. Befehlsabarbeitung im Prozessor

Ziel dieses Abschnitts ist: Verdeutlichung des Zusammenwirkens von Teilen im Prozessor und externen Komponenten (Programm-/Datenspeicher, Ein-Ausgabe) bei der Abarbeitung von Maschinenbefehlen.

Ausgangspunkt ist die Prozessorgrundstruktur aus Bild 5.5 S.103. Der logisch-zeitliche Ablauf wird von der Ablaufsteuerung (AST) realisiert, die dabei alle weiteren Blöcke über Steuersignale beeinflusst. Die AST ist ein komplexer synchroner Automat. Die im Folgenden verwendeten Zustände modellieren die Phasen bei der Befehlsabarbeitung. Die Zustandsübergänge werden mit prozessorinternen Bedingungen bewertet (auch bei alternativen Abläufen) und die Ausgangssteuersignale resultieren aus den Zuständen (Moore-Automat). Die Automatendarstellung folgt der Darstellung in Bild 5.6 S.104.

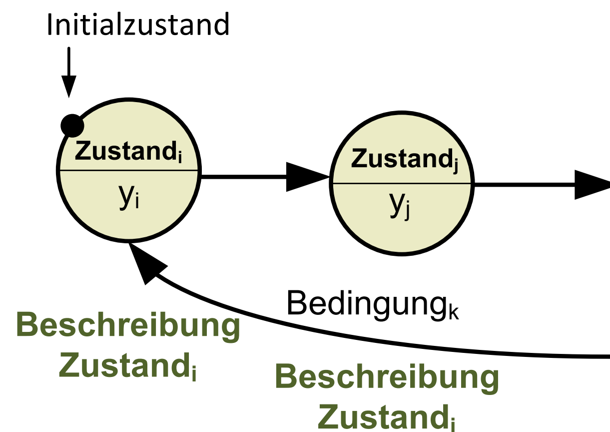


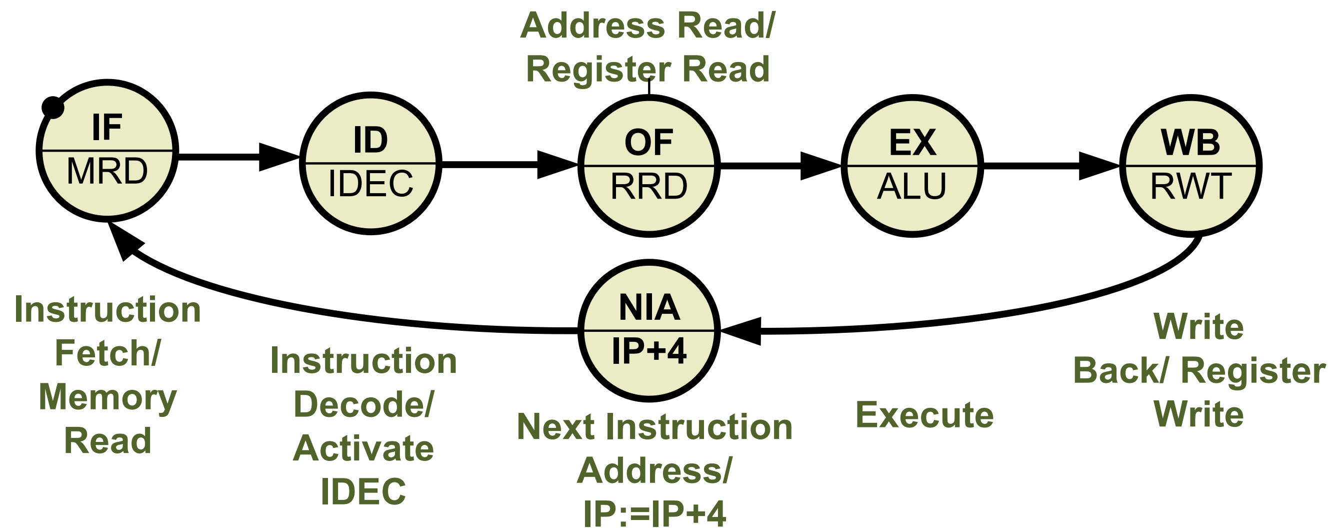
Bild 5.6: Automatendarstellung zur Befehlsabarbeitung

5.3.1. Arithmetik-Logik-Befehle

Bild 5.7 S.106 zeigt den Befehlsablauf zu Arithmetik-Logik-Befehlen (einschließlich Rotations- und Schiebebefehlen). Sie realisieren die arithmetisch-logische Verknüpfung von Operanden aus Registern (ORi) mit Ergebnisrückschreiben nach einem Register (ORj) und dem Prozessorstatusregister (PSR, Flags), d.h. es wird ein RISC-Befehlssatzzugrunde gelegt.

Der Ablauf (Durchlaufen der einzelnen Zustände) erfolgt zyklisch, hier ohne Alternativen (Verzweigungen) über folgende Zustände:

- IF (Instruction Fetch, Befehl Lesen): Adresse aus IP (Instruction Pointer, BA, Befehlsadressregister) auf AB (Adressbus), MRD (Steuersignal Memory Read) auf SB (Steuerbus, beides zum Speicher), der Speicher liefert den Maschinencode von Adresse (vom AB) zum BR (Befehlsregister, IR, Instruction Register), über DB (Datenbus),
- ID (Instruction Decode, Befehl Dekodieren): Maschinencode aus IR (BR) wird über BD umgewandelt in Steuersignale für die ALE (OP, Operation) und in die Signale der Befehlsvariante für die AST zur Steuerung des weiteren befehlsabhängigen Ablaufs.
- RRD (Register Read, Register Lesen): Lesen der beiden Eingangsoperanden aus ORi gleichzeitig parallel zu ALE,
- EX (Execute, Befehl Ausführen): Ergebnis aus den beiden E-Operanden entsprechend OP durch die ALE zum ALE-Ausgang erzeugen,



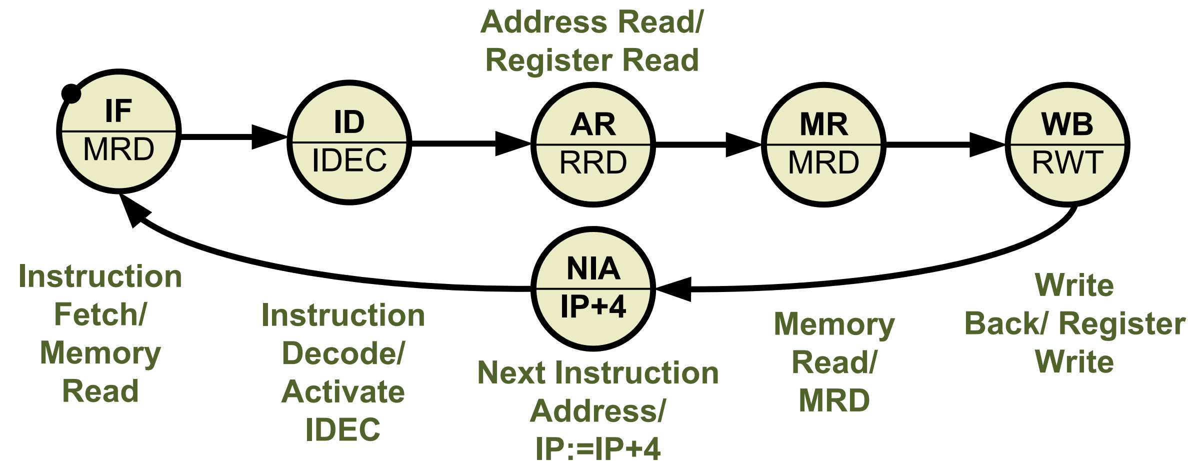
Arithmetik-Logik-Befehl **(Bsp. SUB EAX, EBX)**
Befehl Lesen (IF) – durch Speicher Lesen (MRD) des Maschinencodes
(Bsp. SUB, 0111...00010010B, fiktiv))
Befehl Dekodieren (ID) – durch Aktivierung Befehlsdekoder IDEC
1. Operand: von Register OR1 (RRD) (z.B. EAX) } **parallel**
2. Operand: von Register OR2 (RRD) (z.B. EBX)
ALU-Operation (EX) (Bsp. Subtraktion)
Ergebnis nach Register OR1 (WB, RWT) (z.B. EAX)
Erzeugen der nächsten Befehlsadresse (NIA) (z.B. IP:=IP+4)

Bild 5.7: Befehlsabarbeitung von Arithmetik-Logik-Befehlen

- RWT (Register Write, Register Schreiben): Schreiben vom ALE-Ergebnis nach einem ORj.

- NIA (Next Instruction Address, Erzeugen der nächsten Befehlsadresse): Vorbereiten der Befehlsadresse in IP (BA) für das IF des nächsten Befehls, hier durch inkrementieren (+4).

5.3.2. Speicherbefehle



Speicher-Befehl (Lesen) **(Bsp. MOV EAX, [EDI])**
Befehl Lesen (IF) – durch Speicher Lesen (MRD) des Maschinencodes
(Bsp. MOV , 1000...00010100B, fiktiv))
Befehl Dekodieren (ID) – durch Aktivierung Befehlsdekoder IDEC
Operanden-Speicheradresse: von Register OA (AR, RRD) (z.B. EDI)
Speicheroperand lesen (MR, MRD)
Speicheroperand: nach Register OR1 (WB, RWT) (z.B. EAX)
Erzeugen der nächsten Befehlsadresse (NIA) (z.B. IP:=IP+4)

Bild 5.8: Befehlsabarbeitung von Speicher-Lesebefehlen

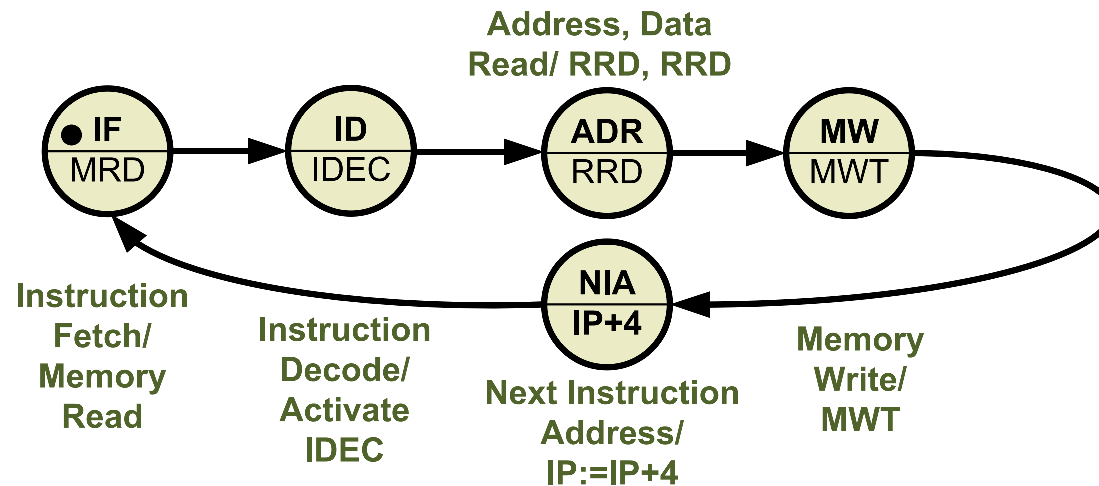
Bild 5.8 S.107 zeigt den Befehlsablauf zum Lesen eines Operanden aus dem Speicher.

Der Ablauf (Durchlaufen der einzelnen Zustände) erfolgt zyklisch, hier ohne Alternativen (Verzweigungen) über folgende Zustände:

- IF, ID wie eben erläutert.
- AR (Address Read, Adresse Lesen), das Operandenadressregister i (OAi oder ORi in anderer Funktion) oder ein spezielles OA wird auf den AB gelegt. Wenn dafür OA-tmp genutzt wird, wird vorher im aktuellen Befehl nach IP+4 (nächste Adresse eines Speicherworts im Programmspeicher) die dort stehende Speicherzieladresse mit MRD in eben beschriebenem Ablauf (analog zu IF) nach OA-tmp gelesen, ähnlich wie bei Sprung, S.81.
- MR (Memory Read): Speicher liefert den Operanden, der auf der Adresse vom AB steht, zum DB.
- WB (Write Back), Operand vom DB nach Register (ORi) schreiben.
- NIA: wie eben erläutert.

Bild 5.9 S.109 zeigt den Befehlsablauf zum Schreiben eines Operanden nach dem Speicher.

Der Ablauf ähnelt dem des Speicher Lesens (Bild 5.8 S.107) und erfolgt zyklisch, hier ohne Alternativen (Verzweigungen) über die Zustände:



Speicher-Befehl (Schreiben) (Bsp. MOV [EDI], EAX)
Befehl Lesen (IF) – durch Speicher Lesen (MRD) des Maschinencodes von Operation (Bsp. MOV , 1001...01010001B, fiktiv))
Befehl Dekodieren (ID) – durch Aktivierung Befehlsdekoder IDEC
Speicheradresse: von Register OA (ADR, RRD) (z.B. EDI) } parallel
Schreibdaten: von Register OR1 (ADR,RRD)
Speicheroperand nach Speicher schreiben (MW, MWT)
Erzeugen der nächsten Befehlsadresse (NIA) (z.B.IP:=IP+4)

Bild 5.9: Befehlsabarbeitung von Speicher-Schreibbefehlen

- IF, ID wie eben erläutert,
- ADR (AR als Address Read, RRD) wie eben erläutert, ergänzt durch ein gleichzeitiges paralleles Lesen der Daten aus einem Register ORi (Data Read, RRD),

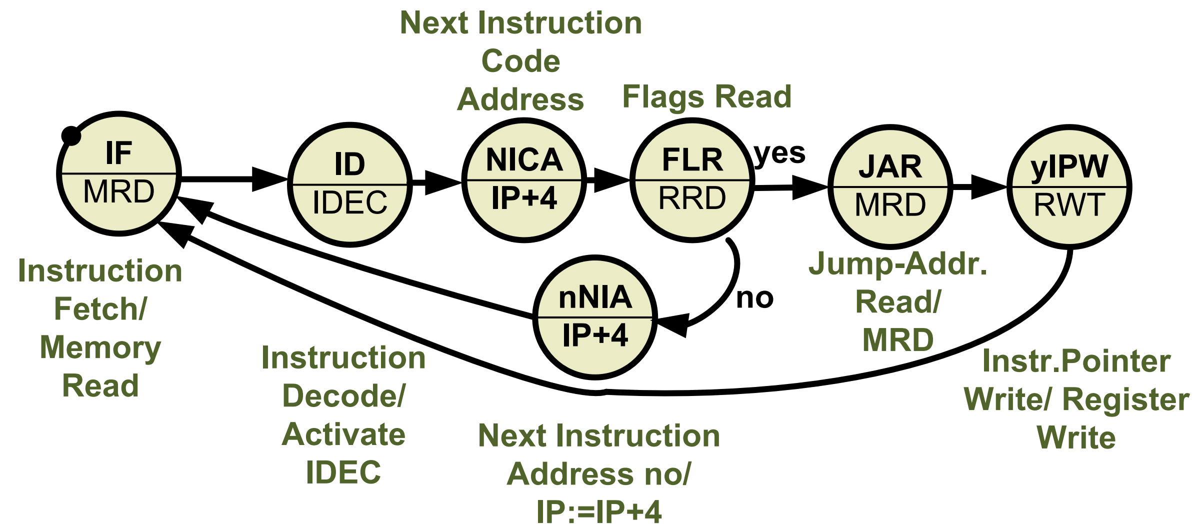
- MW Ausgabe von Adresse und Daten zum Speicher, der die Daten auf die Adresse schreibt (MWT),
- NIA: wie eben erläutert.

5.3.3. Sprungbefehle

Bild 5.10 S.111 zeigt den Befehlsablauf für einen bedingten Sprungbefehl.

- IF, ID wie eben erläutert,
- NICA (Next Instruction Code Address): IP auf Sprungadresse (im Befehlscode) setzen (siehe Sprung S.81), IP:=IP+4.
- FLR (Flag Register Read): Auslesen des Prozessorstatusregisters (PSR, Flags) zur Vorbereitung der Bedingungsauswertung (ausgewähltes Flag-Bit),
- Bedingungsauswertung des ausgewählten Flags, nicht erfüllt (no), weiter mit nNIA, erfüllt (yes), weiter mit JAR,
- nNIA: (bei not/ nicht erfüllter Bedingung) wie bei NIA, eben erläutert,
- JAR: (Jump Address Read, Sprungadresse lesen) Lesen der Sprungzieladresse aus dem Programmspeicher (aktuelle Programmspeicheradresse steht im IP (BA) -> siehe Abschnitt 4.5),
- yIPW (yes Instruction Pointer Write), Eintragen der gelesenen Sprungzieladresse in den IP (BA).

- Bei unbedingten Sprüngen entfallen die Zustände FLR und nNIA und es folgt nach NICA immer JAR und (y)IPW und damit entfällt auch der no-Übergang.



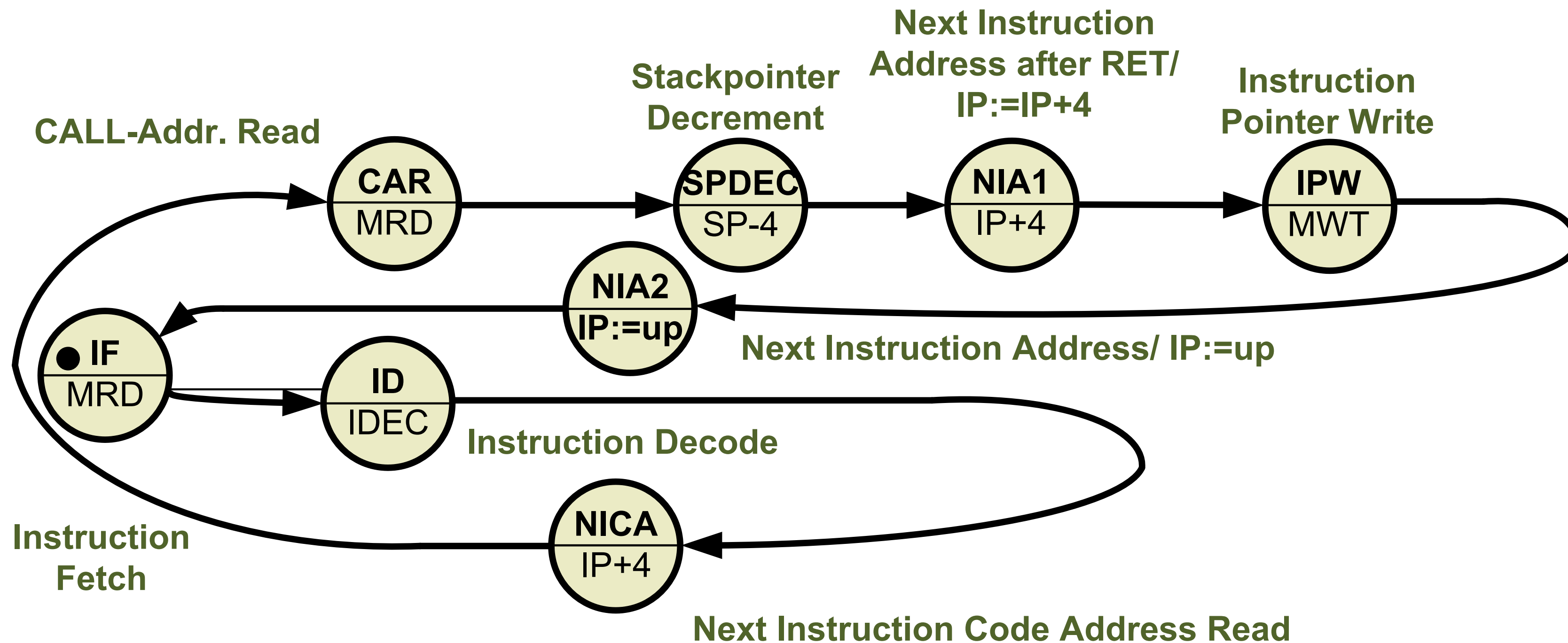
Jcond-Befehl (Bsp. JNZ m1, 10...004B])
Befehl Lesen (IF) – durch Speicher Lesen (MRD) des Maschinencodes
(Bsp. JNZ , 0000...00000010B, fiktiv))
Befehl Dekodieren (ID) – durch Aktivierung Befehlsdekoder IDEC
Erzeugen nächste Befehlscodeadr. (NICA), Adr. von Sprungadr. (z.B. IP:=IP+4)
Lesen, Auswerten Flag-Register (FLR, RRD) (Bsp. Z-Flag)
yes: (Z=0): Sprungadresse lesen (JAR, MRD) (Bsp. Adresse von m1)
yes: Sprungadresse nach Instruction Pointer (yIPW) (Bsp. Adr. v. m1)
no: (Z=1): Erzeugen nächste Befehlsadresse (nNIA) (Bsp. IP:=IP+4)

Bild 5.10: Befehlsabarbeitung von bedingten Sprungbefehlen

5.3.4. Unterprogrammbefehle

Bild 5.11 S.113 zeigt den Befehlsablauf für einen Unterprogramm-Aufrufbefehl (Call):

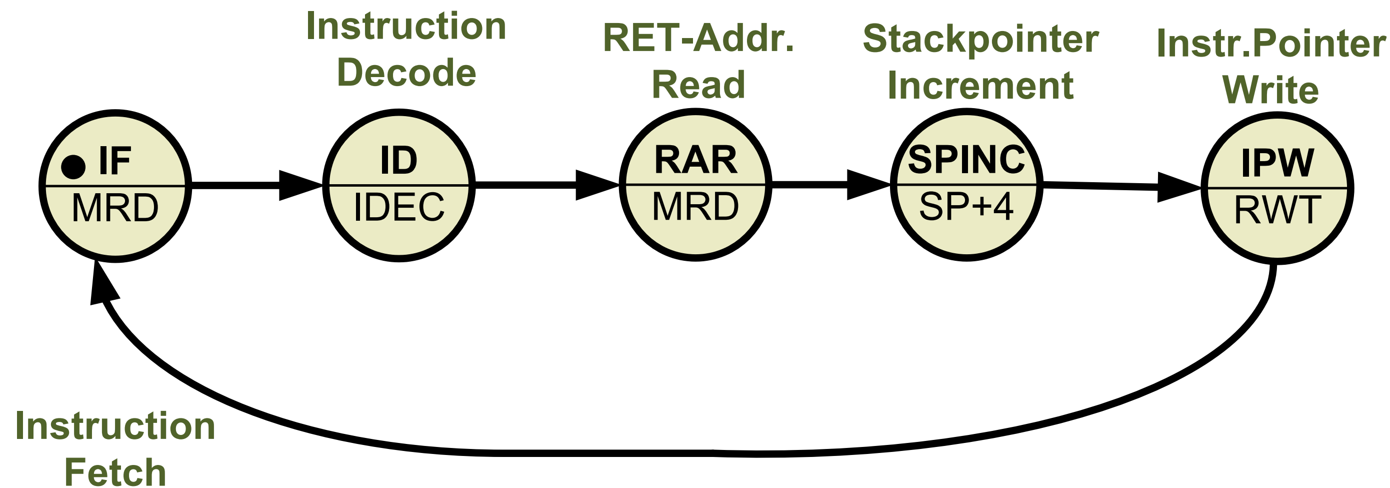
- IF, ID wie eben erläutert,
- NICA (Next Instruction Code Address, CALL Address, das nächste Speicherwort im Programmspeicher ist die CALL-Adresse, wie bei Sprung S.81): Erzeugen der nächsten Adresse im Programmspeicher (nach Befehlscode von CALL, IP:=IP+4),
- CAR (Call Address Read, Call Adresse Lesen): Lesen der Unterprogrammstartadresse vom Programmspeicher, Programmspeicheradresse aus IP (BA),
- SPDEC (Stack Pointer Decrement, Stackpointer dekrementieren): Vorbereitung für das Schreiben in den Stack (SP:=SP-4),
- NIA1 (Next Instruction Address 1, nächste Programmspeicheradresse): Erzeugen der Folgeadresse auf CALL im Programmspeicher (IP:=IP +4),
- IPW (Instruction Pointer Write, BA schreiben): IP (BA) (Rückkehradresse), in den Speicher schreiben mit Speicheradresse aus dem Stackpointer,
- NIA2 (Next Instruction Address 2, nächste Befehlsadresse): Schreiben der unter dem vorangegangenen CAR gelesenen Unterprogrammstartadresse in den IP (BA).



CALL-Befehl (Bsp. CALL 200h])
Befehl Lesen (IF) – durch Speicher Lesen (MRD) des Maschinencodes
(Bsp. von CALL , 0000...00001000B, fiktiv))
Befehl Dekodieren (ID) – durch Aktivierung Befehlsdekoder (IDEC)
Erzeugen der nächsten Befehlscodeadresse (NICA) (Bsp. IP:=IP+4)
UP-Adresse (CAR) vom Speicher lesen (Adr. aus IP, MRD) (Bsp. 200h)
Stackpointer dekrementieren (SPDEC): (Bsp. ESP:=ESP-4)
Erzeugen der nächsten Befehlsadresse, gültig nach RET (NIA1)
Instruction Pointer (IPW) nach Speicher (Adr. aus SP) schreiben (MWT)
Erzeugen der nächsten Befehlsadresse (NIA2) (Bsp. IP:=200h)

Bild 5.11: Befehlsabarbeitung von Unterprogramm-Aufrufbefehlen (Call)

Bild 5.12 S.114 zeigt den Befehlsablauf für einen Unterprogramm-Rückkehrbefehl (Return):



RETURN-Befehl (Bsp. RET)
Befehl Lesen (IF) – durch Speicher Lesen (MRD) des Maschinencodes
(Bsp. RET , 0000...00010000B, fiktiv))
Befehl Dekodieren (ID) – durch Aktivierung Befehlsdekoder IDEC
Rückkehradr. lesen (RAR) mit Speicheradr. aus Stackpointer (Bsp. ESP)
Stackpointer inkrementieren (Bsp. ESP:=ESP+4)
Rückkehradresse nach Instruction Pointer (IPW)

Bild 5.12: Befehlsabarbeitung von Unterprogramm-Rückkehrbefehlen (Call)

- IF, ID wie eben erläutert,

- RAR (Return Address Read, Rückkehradresse Lesen): Lesen der durch den zugehörigen CALL geschriebenen Rückkehradresse mit der Speicheradresse aus dem Stackpointer (SP),
- SPINC (Stack Pointer Increment, Stackpointer Inkrementieren): Stackpointer (SP) wieder auf Wert vor dem CALL setzen durch SP:= SP+4,
- IPW (Instruction Pointer Write, nach BA schreiben): in RAR gelesene Rückkehradresse in den IP (BA) schreiben.

5.3.5. Gesamtverhalten der Befehle in der Ablaufsteuerung

Alle Befehle beginnen mit den Zuständen IF, ID und erzeugen innerhalb der verschiedenen Befehlsablaufvarianten unterschiedlich die Folgeadresse für den nächsten abzuarbeitenden Befehl (z.B. Arithmetik-Logik-Befehl: IP:=IP+4, Sprungbefehl, unbedingt: IP:= gelesene Sprungzieladresse). Danach wird prinzipiell wieder (mit dem jetzt aktualisierten IP) mit dem nächste IF fortgesetzt. Dieser Ablauf wiederholt sich ständig (unendlich oft). Das prinzipielle Gesamtverhalten der Ablaufsteuerung (AST) entsteht deshalb aus der Zusammenfassung aller Befehlsabläufe in den verschiedenen Varianten (einschließlich der hier nicht näher beschriebenen Untervarianten) mit gemeinsamen Anfangszuständen IF, ID und gemeinsamem Anfangszustand IF des Folgebefehls. Bild 5.13 S.116 beschreibt dieses.

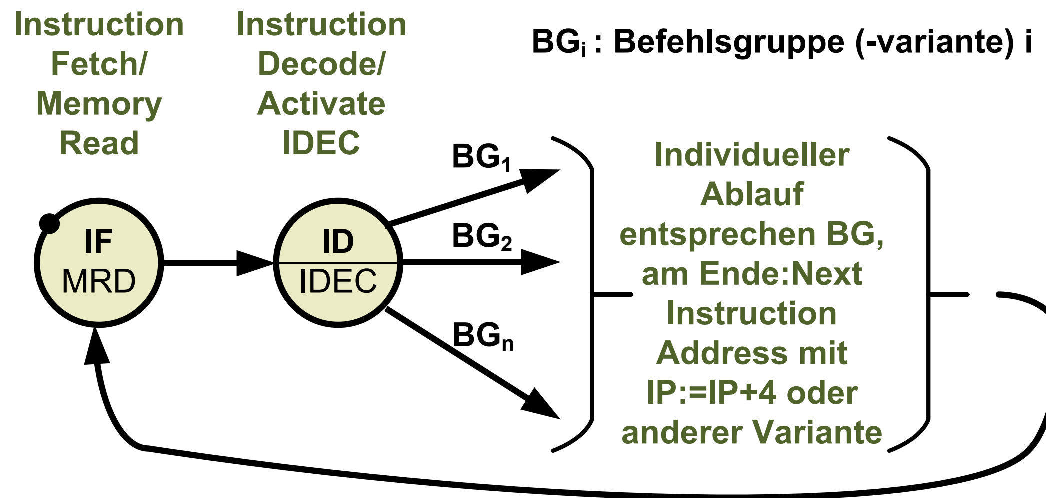


Bild 5.13: Prinzip der Befehlsabarbeitung für alle Befehlsgruppen (-varianten)

5.4. Interrupt

In vielen Fällen ist es notwendig, im Programm eine Reaktion auf ein rechnerexternes Ereignis vorzusehen. Ein Ereignis ist dabei die Änderung des Werts (z.B. von 0 nach 1) einer logischen Variablen, abgebildet auf ein binäres Logiksignal. Ein einfaches Beispiel dazu wäre eine angeschlossene externe Taste, deren Betätigen zu einer programmtechnischen Reaktion führen soll. Mit den bisher behandelten Funktionen gibt es als Möglichkeit die in Bild 5.14 (a) S.117 mit Programmliniendarstellung bzw. in Bild 5.14 (b) S.118 als Aktivitätsdiagramm (UML) dargestellte Lösung. Diese einfache Methode der zyklischen

Abfrage wird mit Polling (Abfrage) bezeichnet. Mit einem IN-Befehl wird das Bit, an dem die Taste angeschlossen ist, zyklisch abgefragt. Nach einem eventuellen Bittestbefehl kann bei nicht gedrückter Taste über einen bedingten Sprung wieder zur Abfrage übergegangen werden, sonst erfolgt die programmtechnische Reaktion. An deren Ende wird auch wieder zur Abfrage übergegangen.

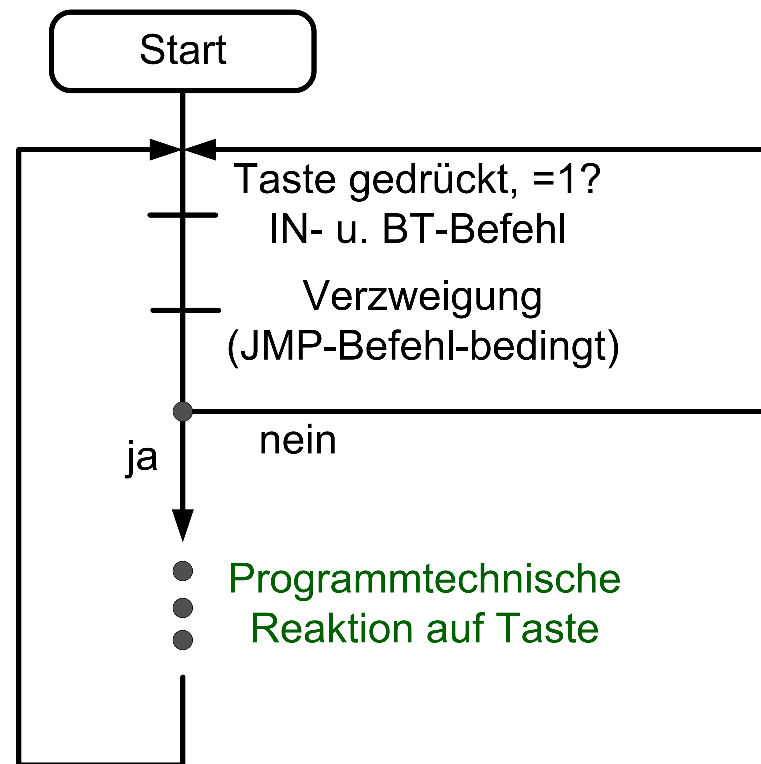


Bild 5.14 (a): Reaktion durch Teilprogramm auf ein externes Ereignis (als Programmliniendarstellung)

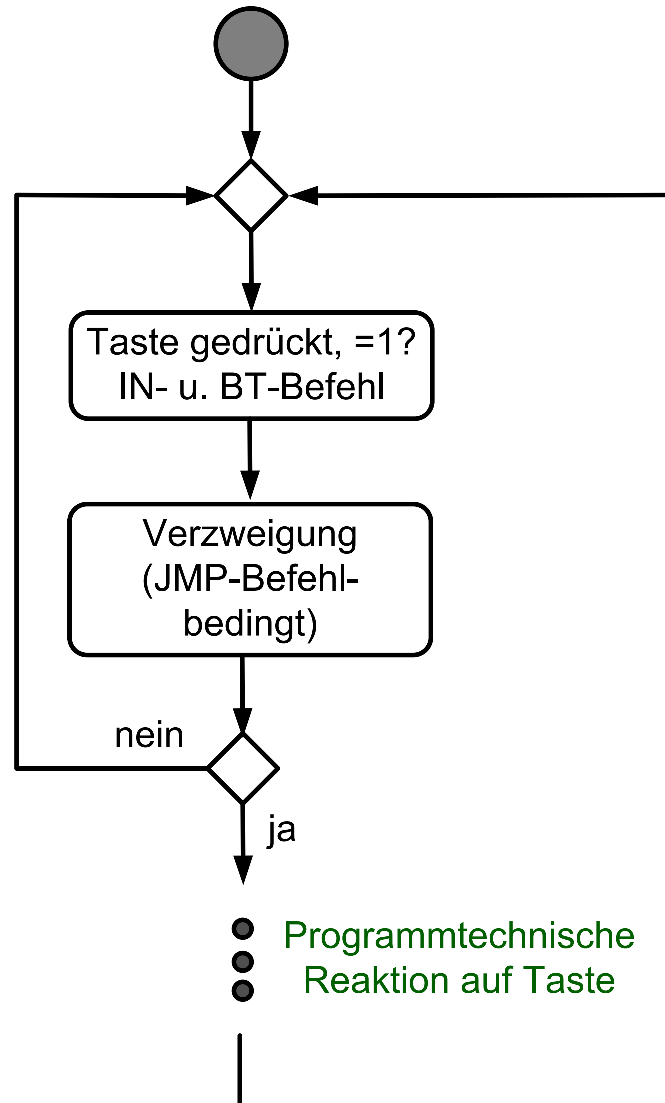


Bild 5.14 (b): Reaktion durch Teilprogramm auf ein externes Ereignis (als Aktivitätsdiagramm)

Als zeitliches Verhalten entsteht:

Tastenbetätigung: ca. 10^{-1} s

Prozessorbefehl ca. 10^{-8} … 10^{-9} s

➔ im Normalfall ist der Prozessor während der Erkennung der Taste maximal unterlastet und steht auch nicht für andere Verarbeitungsfunktionen zur Verfügung.

Um das zu vermeiden, wird die Prozessorgrundstruktur (Bilder 5.4 S.100 oder 5.5 S.103) um eine Funktion zum Interrupt (Programmunterbrechung) erweitert. Es handelt sich dabei um eine Erweiterung des synchronen Automaten der Ablaufsteuerung (AST) und einen zusätzlichen Befehl zur Rückkehr aus einem Interruptprogramm. Die Bilder 5.15 (a) S.120 bzw. (b) S.121 zeigen das Prinzip.

Als Ablauf entsteht:

Der Prozessor arbeitet mit 100% Prozessorleistung an einem Hauptprogramm (HP, evtl. ohne direkten Bezug zum externen Ereignis).

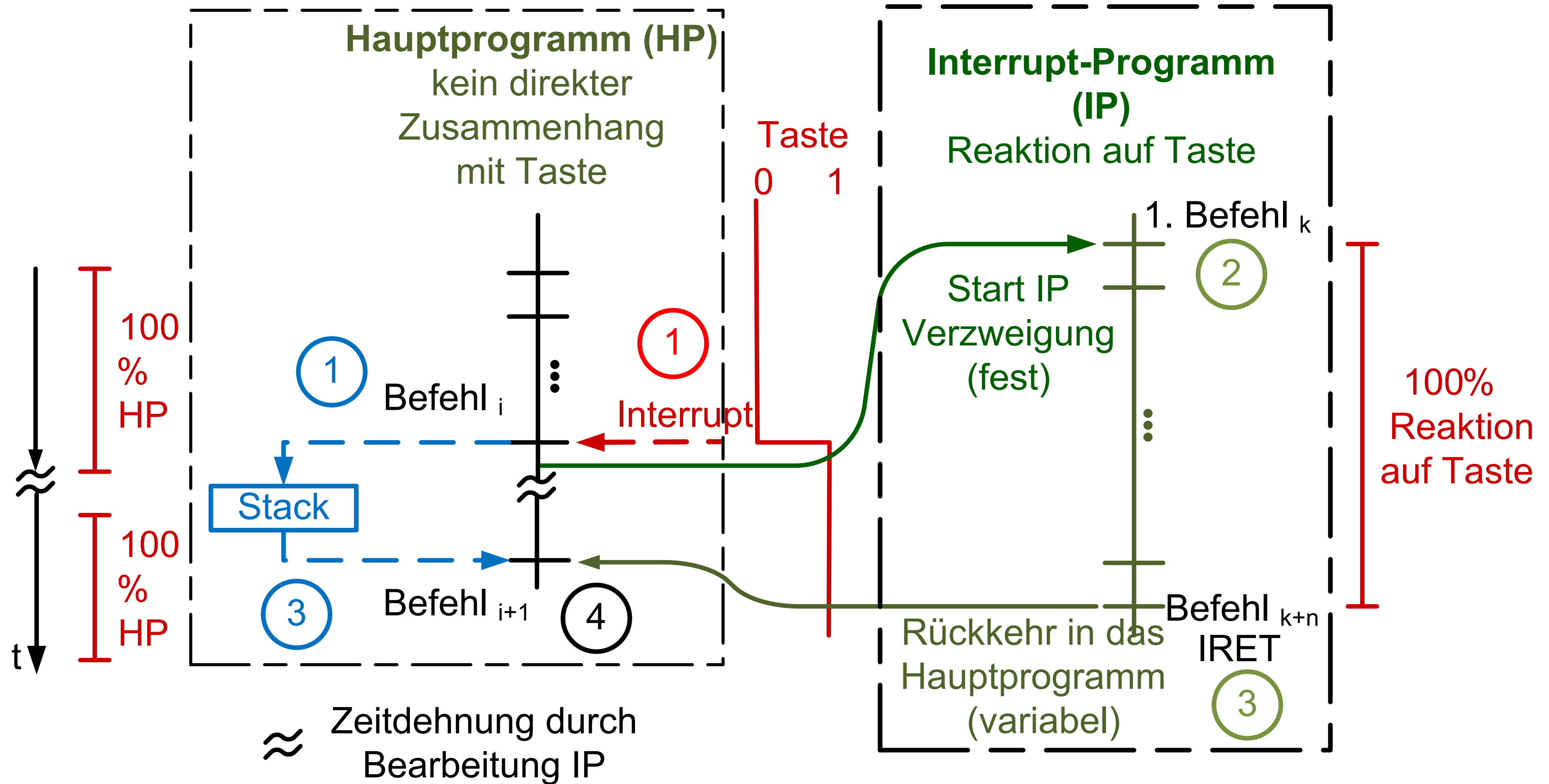


Bild 5.15 (a): Reaktion durch Interruptprogramm auf ein externes Ereignis (als Programmliniendarstellung)

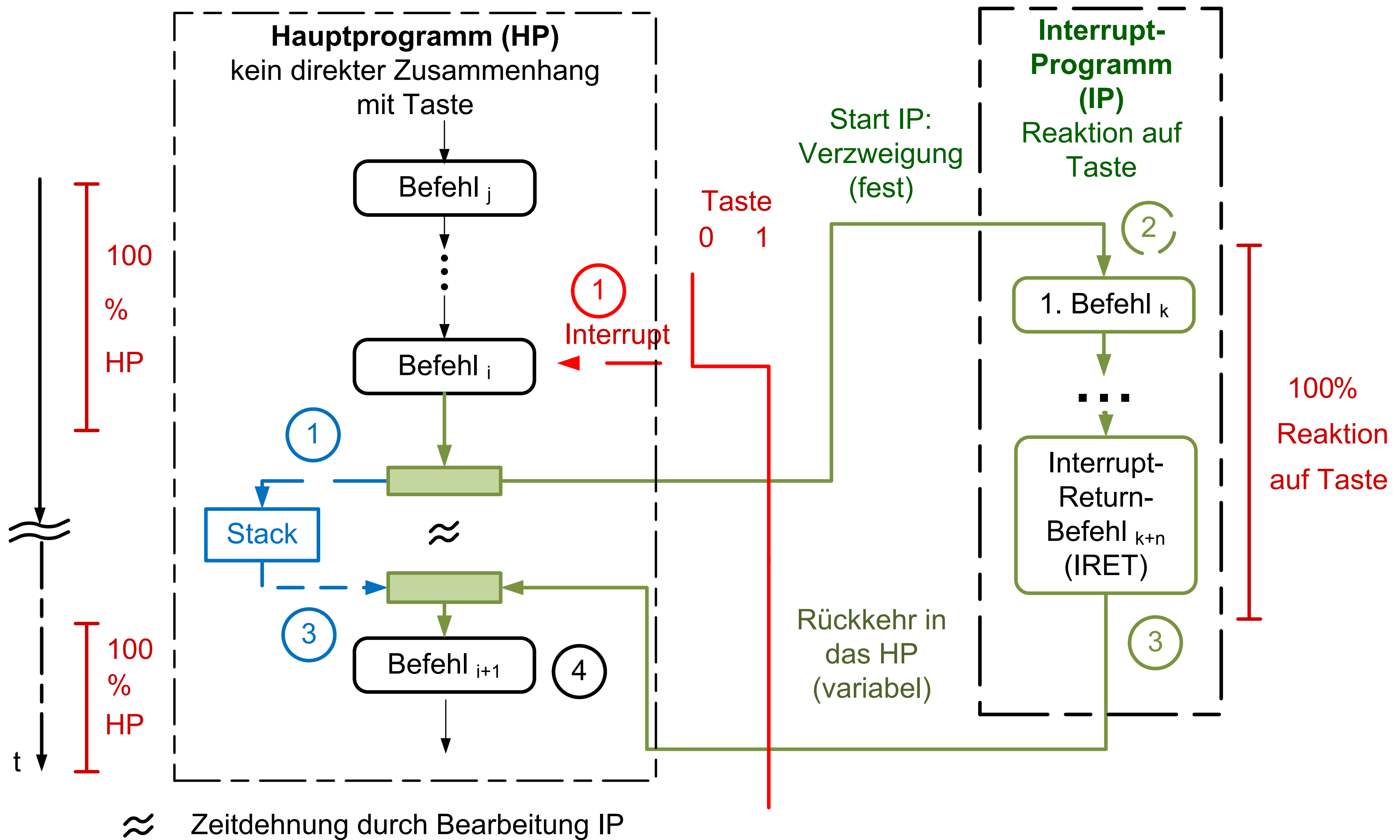


Bild 5.15 (b): Rektion durch Interruptprogramm auf ein externes Ereignis (als Aktivitätsdiagramm)

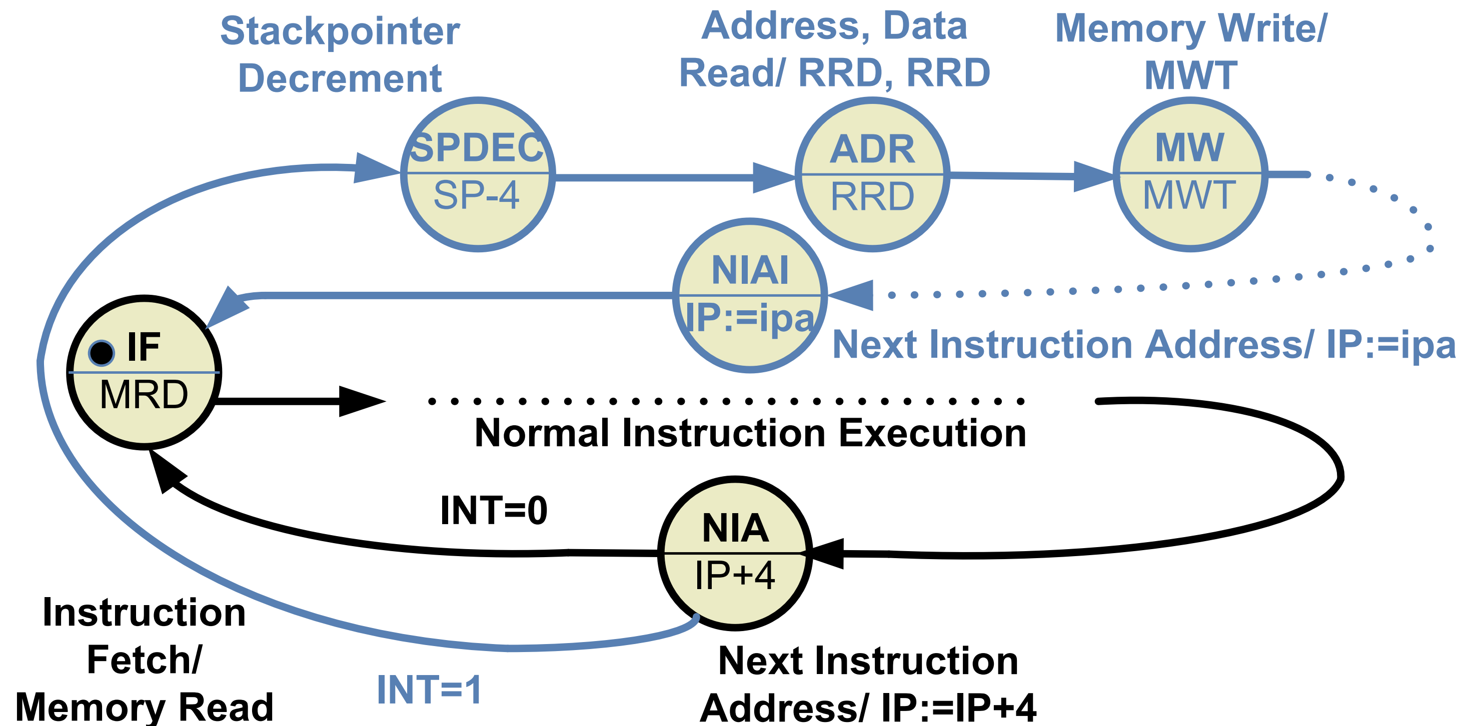
(1) Durch Betätigen der Taste (Pegel 0 nach 1) erfolgt der Start der Interrupthardware (in der AST) durch Signaländerung an dem externen Prozessorsignaleingang Interrupt (INT). Der aktuelle Befehl, der unabhängig vom externen Ereignis ist (Befehl $_i$) wird zu Ende bearbeitet. Durch die AST wird die Adresse von dem Befehl $_{i+1}$ in den Stack geschrieben (mit SP-Behandlung).
(2) Danach wird der erste Befehl des Interruptprogramms (Befehl $_k$) gestartet. Das ist ein Teilprogramm, welches auf die Taste reagieren soll und dessen erster Befehl über seine Befehlsadresse direkt dem Eingang INT zugeordnet ist.

(3) Als letzter Befehl dieses Teilprogramms wird ein Interruptrückkehrbefehl (IRET, Befehl $_{k+n}$) abgearbeitet.

(4) Dieser realisiert, dass die Adresse von Befehl $_{i+1}$ im Hauptprogramm (Folgebefehl zu Befehl $_i$) aus dem Stack gelesen wird (mit SP-Behandlung) und die Bearbeitung dort fortgesetzt wird.

Um das zu realisieren, wird der synchrone Automat der AST nach Bild 5.16 S.123 erweitert.

Der schwarz dargestellte Ablauf entsprich allen Befehlsgruppen nach Bild 5.13 S.116.



Interruptannahme (fest) (Bsp. INT=1)
INT=0, weiter normal: Befehl Lesen (IF) – usw.
INT=1, Register Stackpointer dekrementieren (SPDEC): ESP:=ESP-4
Speicheradresse lesen: von Register ESP (ADR)
Schreibdaten lesen: von Register IP (ADR)
} parallel
Schreibdaten nach Speicher schreiben (MW)
Erzeugen der nächsten Befehlsadresse (NIA) (z.B. IP:=100h)

Bild 5.16: Erweiterung der Ablaufsteuerung für die Interruptfunktion

- Alle Befehle starten mit IF (Instruction Fetch) und enden mit Next Instruction Address (Erzeugen der nächsten Befehlsadresse, hier mit IP: = IP+4, evtl. bei Sprung, CALL, usw. anders).
- Die Interruptbedingung (INT =1) wird am Ende von NIA berücksichtigt. Das führt zum Verzweigen in den Start des Interruptprogramms:
  (1) Rückkehradresse (mit NIA schon erzeugt) in den Stack schreiben (ADR, MW, einschließlich vorheriger Stackpointerbehandlung, SPDEC),
  (2) dem externen Ereignis zugeordnete Startadresse (ipa) in den Instruction Pointer (IP, Befehlsadressregister, BA) schreiben (NIAI) und weiter mit
- IF des ersten normalen Befehls des Interruptprogramms.

Für diese Interruptannahme-Variante ist der normale RETURN (siehe im Vorangegangenen, Bild 5.12 S.114) als Interruptreturn verwendbar.

Dabei kann als Problem auftreten:

Die Interruptannahme erfolgt zwischen einem Befehl, der die Flags (PSR) setzt und einem Befehl (z.B. bedingter Sprung), der aufgrund der Flags verzweigt. -> Das Interruptprogramm dazwischen verändert i. Allg. die Flags, und der bedingte Sprung hat nichts mehr mit dem Befehl davor im Hauptprogramm zu tun.

Als mögliche Behandlung kann integriert sein:

- Nach dem Schreiben der Rückkehradresse in den Stack in der Interruptannahme wird anschließen das PSR (die Flags) in den Stack geschrieben (mit SP-Behandlung).
- Im Interrupt-Returnbefehl (IRET) wird vor dem Lesen der Rückkehradresse aus dem Stack die frühere PSR-Belegung gelesen (bei der Interruptannahme geschrieben) und in das PSR geschrieben. Der nach IRET folgende Befehl findet die ursprüngliche PSR-Belegung vor.

Ein ähnliches Problem besteht bei Registerbenutzung.

Das Register ORi wird vor dem Interrupt geschrieben (z.B. mit Move).

ORi wird im Interruptprogramm benutzt. -> Das Überschreiben führt zu Überschreiben des ORi-Werts nach Move.

Nach Rückkehr folgt im Hauptprogramm ein Befehl, der ORi benutzt (z.B. ADD). Dieser würden den falschen ORi-Wert benutzen.

-> Ausweg: alle im Interruptprogramm genutzten ORi über den Stack „retten“ (mit PUSH- und POP-Befehlen) am Anfang und Ende des Interruptprogramms.

Abschließend zum Interrupt folgt als anschauliches Beispiel das „Drucken im Hintergrund“. Während einer Textbearbeitung soll nach Start eines Druckvorgangs parallel zu diesem eine Weiterarbeit in der Textbearbeitung möglich sein. Deshalb erfolgt die Zeichenausgabe zum Drucker Interrupt-gesteuert.

Bild 5.17 S.127 zeigt die Abläufe als parallele Automatengraphen. Der obere Automat (grün) modelliert das Hauptprogramm (HP, Textverarbeitung und Druckanforderung). Der untere Automat (blau) beschreibt das Interruptprogramm (IP). Der Start des Druckens eines Zeichens geschieht mit Ausgabe desselben an den Drucker (rot) (1._Z-Ausg.: Ausgabe des ersten Zeichens, N_Z-Ausg.: Ausgabe des nächsten Zeichens). Der Drucker meldet das Ende des Druckvorgangs (gegenüber der Befehlsabarbeitungszeit durch die mechanischen Funktionen um Größenordnungen langsamer) durch das Signal Interrupt_Dr, welches das Interruptprogramm startet. Die Synchronisation zwischen HP und IP erfolgt über die Variable Drucker-frei, die den Status des Druckvorgangs beinhaltet.

Der Ablauf erfolgt nach dem nachstehenden Schema:

Der Nutzer befindet sich im Hauptprogramm (HP) “Text bearbeiten“ (einschließlich einer Funktion Drucken).

Die Druckanforderung als Kommando in der Textverarbeitung verlangt das Ende des letzten Druckvorgangs (Drucker_frei = 1).

Wenn dieses gilt, erfolgt die Übergabe der Speicherstruktur mit dem auszugebenden Text (Ablegen in einem dem HP und IP zugänglichen Speicherbereich).

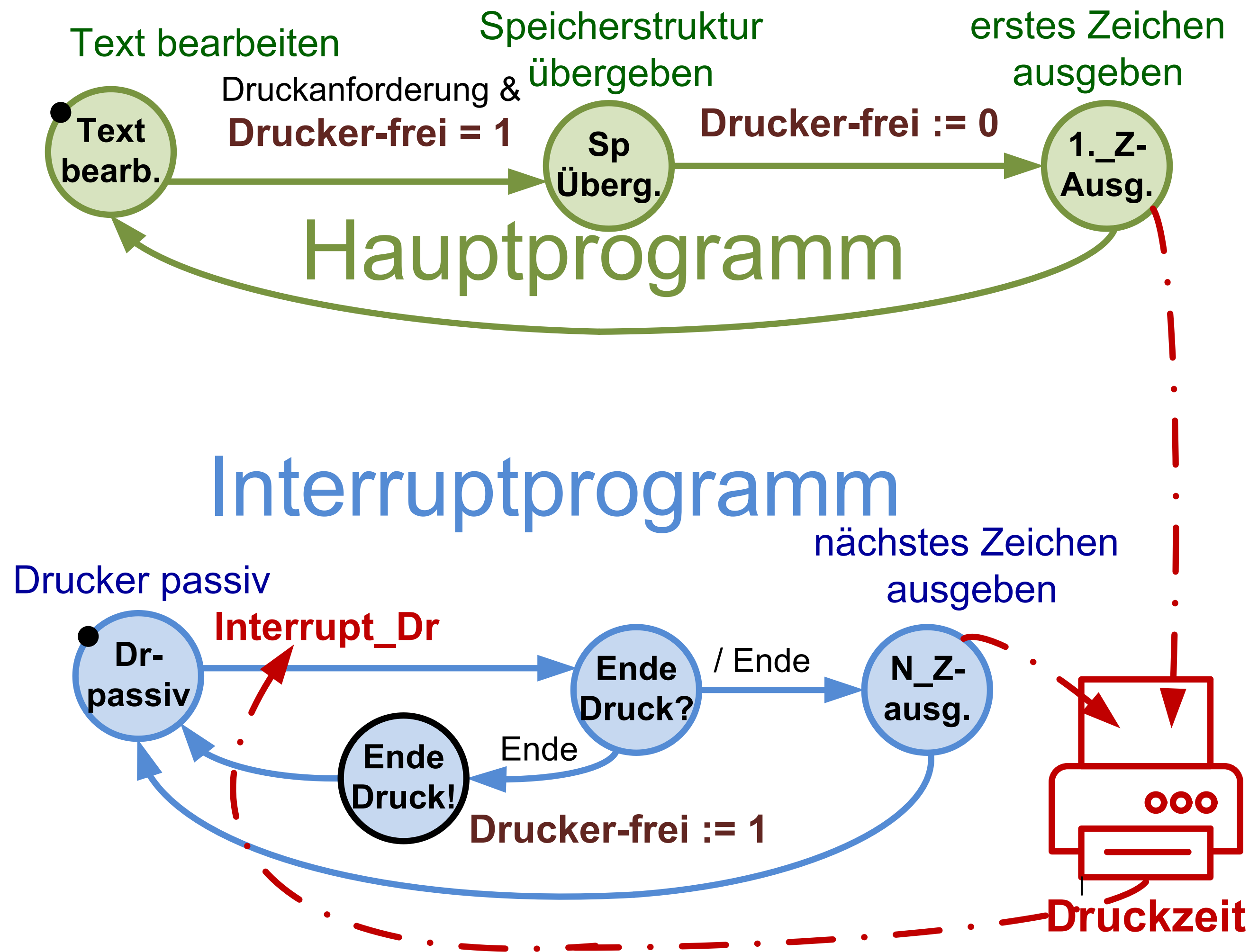


Bild 5.17: Parallele Automatengraphen zum Drucken im Hintergrund

Danach veranlasst das HP das Setzen von Drucker_frei auf 0 und die Ausgabe des ersten Druckzeichens an den Drucker.

Jetzt ist die Weiterarbeit im HP möglich, eine erneute Druckanforderung würde allerdings das Ende des letzten Druckvorgangs benötigen. Die anderen Funktionen sind uneingeschränkt möglich.

Nach der durch die Mechanik bedingten Druckzeit des aktuell ausgegebenen Zeichens generiert der Drucker ein Fertig-Signal, welches zur Erzeugung eines Interrupts benutzt wird (Interrupt-Dr) (Zustandsübergang im IP Dr-passiv nach Ende_Druck?).

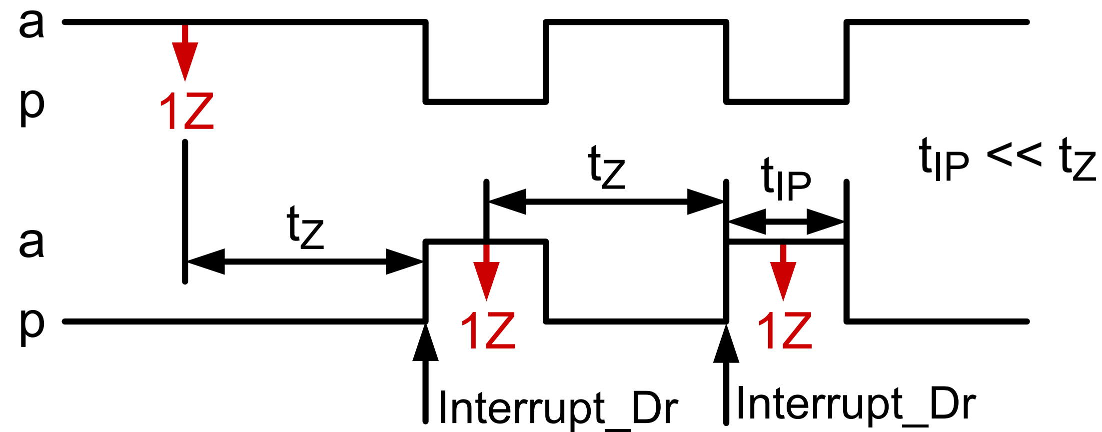
Vor der Ausgabe eines eventuellen nächsten Zeichens wird geprüft, ob bereits alle übergebenen Zeichen gedruckt wurden (Ende_Druck?) Wenn das der Fall ist wird in Ende_Druck! Drucker-frei wieder auf 1 gesetzt. Ein erneuter Druckvorgang ist möglich.

Wenn es noch nicht das letzte Zeichen war, erfolgt das Ausgeben des nächsten Zeichens und Rückkehr in die Textverarbeitung. Parallel wird auf das nächste Interrupt-Dr (ohne programmtechnische Prozessorbeteiligung) gewartet.

Den zeitlichen Ablauf dazu zeigt Bild 5.18 S.129. Das Hauptprogramm (aktuelle Textverarbeitung) wird kurzzeitig nach Interrupt durch Ende des Druckvorgangs eines Zeichens (Signal Interrupt_Dr) unterbrochen, um das nächste Zeichen auszugeben (1Z). Wenn das Interruptprogramm bezogen auf die Zeichenausgabezeit (t_z) relativ kurz ist, wird diese Unterbrechung für den Nutzer der Textverarbeitung nicht bemerkbar.

Haupt-
programm

Interrupt-
programm



a	aktiv
p	passiv
1Z (N_Z-Ausg.)	ein Zeichen ausgeben
Interrupt_Dr	Zeichenausgabe beendet -> Interrupt
t_Z	benötigte Zeit für das Drucken eines Zeichens (Druckzeit)
t_{IP}	benötigte Zeit für Interrupt-Annahme + Interruptprogramm

Bild 5.18: Zeitverläufe zum Drucken im Hintergrund

5.5. Prozessorzugriffe über externes Interface

Während der Abarbeitung der Befehle benötigt der Prozessor Zugriffe auf die weiteren Funktionseinheiten in der Gesamtarchitektur. Dabei werden notwendig:

- Bei Befehl Lesen (Instruction Fetch, IF) vom Programmspeicher: Speicher Lesen, alle Befehle (Abschnitt 4),
- bei Operand lesen/schreiben von und zum Datenspeicher: Speicher Lesen/Schreiben, Befehle für den Datentransfer von und zum Speicher (Abschnitt 4.2),
- bei Operand lesen/schreiben von und zu Ein-/Ausgabe-Schnittstellen: EA-Lesen, EA-Schreiben, Befehle für den Datentransfer von und zu Ein-/Ausgabe-Schnittstellen (Abschnitt 4.2),
- evtl. bei Interruptannahme bei mehreren Interruptquellen: Lesen einer Interruptkennung (Interruptvektor, wird im Weiteren als Spezialfall nicht betrachtet).

Als Schnittstelle nach außen dienen dabei Adressbus, Datenbus und einige Signale des Steuerbusses (Abschnitt 5.1). Als Auswahlprinzip wird Adressierung verwendet (Abschnitt 3).

Das Zusammenwirken von Prozessor und externer Funktionseinheit wird durch je zwei parallele Automaten beschrieben (links: Prozessor, rechts: externe Funktionseinheit), die durch ein oder mehrere Steuersignale (Memory Read, MRD; Memory Write, MWT; Execution

Acknowledge, XACK; Input Output Read, IOR; Input Output Write, IOW) gekoppelt sind. Bild 5.19 S.132 zeigt den Ablauf bei Speicher Lesen. Dieser beinhaltet als Schritte:

(1) Der Prozessor ist passiv bzgl. Speicher Lesen (d.h. die Ablaufsteuerung ist in anderen Zuständen, siehe im Vorangegangenen). Der Speicher ist passiv.

(2) Innerhalb des Befehlsablaufs folgt ein Zustand mit Speicher Lesen (MRD bei IF, MR).

(3) Der Prozessor legt die zu lesende Adresse auf den Adressbus (bleibt dort stabil bis zum Ende des Speicherzugriffs).

(4) Der Prozessor setzt zeitverzögert MRD auf 1.

(5) Der Prozessor wartet für t_{wait}.

(6) Durch MRD=1 verlässt der Speicher seinen passiven Zustand, übernimmt die Adresse vom Adressbus und stellt durch eine interne Speicheroperation nach $t_{Speicher}$ die Daten von Adresse auf den Datenbus ($t_{wait} \geq t_{Speicher}$).

(7) Der Prozessor übernimmt nach t_{wait} die Daten vom Datenbus in ein internes Register.

(8) Der Prozessor schaltet MRD auf 0 und wird wieder passiv (s.o.).

(9) Durch MRD=0 nimmt der Speicher die Daten vom Datenbus und wird wieder passiv (s.o.).

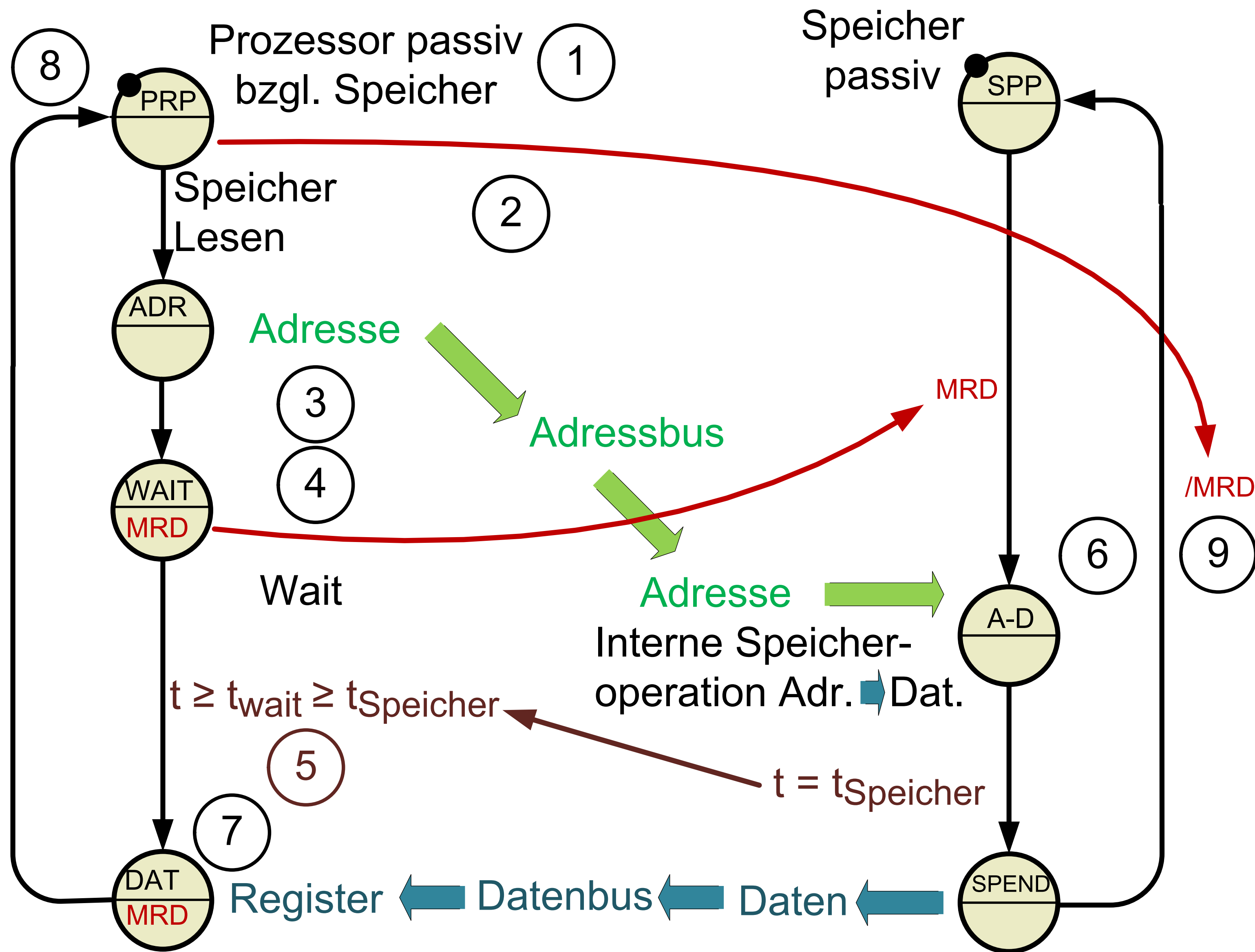


Bild 5.19: Ablauf bei Speicher Lesen

Dieser Ablauf ist auch durch ein Zeitdiagramm beschreibbar (Bild 5.20 S.133).

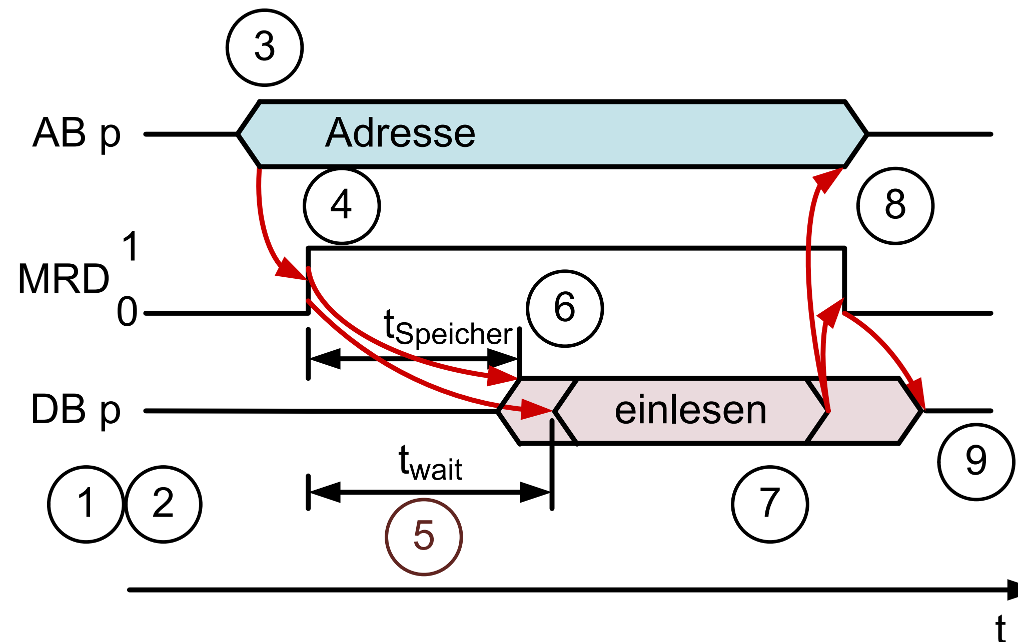


Bild 5.20: Ablauf bei Speicher Lesen als Zeitdiagramm

Der alternative Zugriff auf den Speicher ist Speicher Schreiben. Bild 5.21 S.134 zeigt den Ablauf dazu. Dieser beinhaltet als Schritte:

(1) Der Prozessor ist passiv bzgl. Speicher schreiben. Der Speicher ist passiv.
(2) Innerhalb des Befehlsablaufs erkennt die AST, dass ein Speicher Schreiben folgen soll.
(3) Der Prozessor legt die Adresse, auf die geschrieben werden soll, auf den AB und parallel die zu schreibenden Daten auf den DB.
(4) Der Prozessor setzt zeitverzögert MWT (Steuersignal Memory Write) = 1.

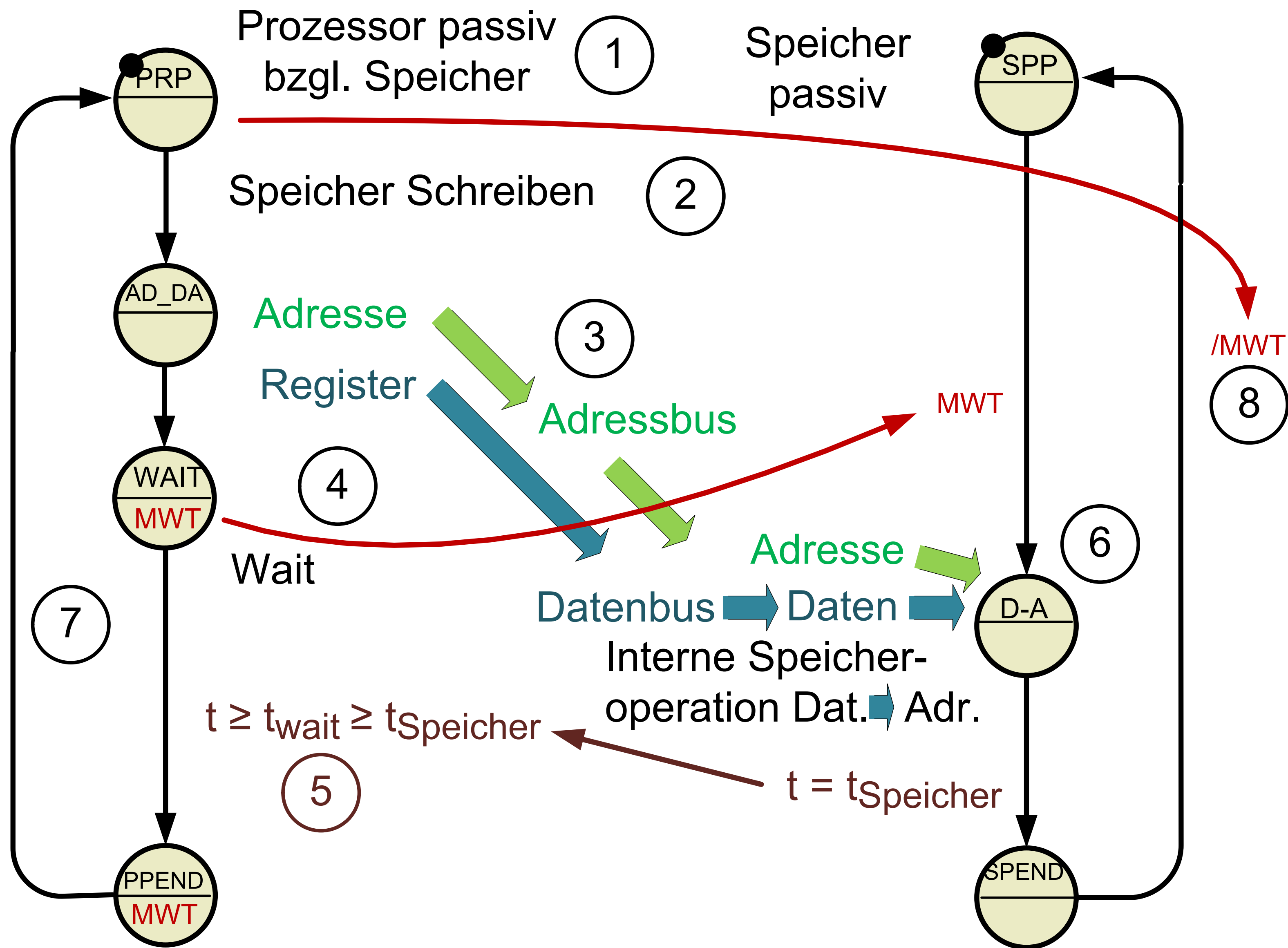


Bild 5.21: Ablauf bei Speicher Schreiben

(5) Der Prozessor wartet t_{wait} mit $t_{wait} \geq t_{Speicher}$.
(6) Aufgrund von MWT = 1 wird der Speicher aktiv und schreibt die Daten vom Datenbus auf die Adresse vom AB innerhalb von $t_{Speicher}$.
(7) Der Prozessor beendet nach t_{wait} den Speicherzugriff, setzt MWT = 0 und wird wieder passiv (bzgl. Speicher schreiben).
(8) Aufgrund von MWT = 0 wird der Speicher passiv.

Beide Zugriffe, wie bisher erläutert, arbeiten in der Variante „Synchrones Busverfahren". Das bedeutet, dass das Zeitverhalten des Prozessors (t_{wait}) auf das Zeitverhalten des Speichers ($t_{Speicher}$) mit $t_{wait} \geq t_{Speicher}$ synchronisiert sein muss.

➔ Bei exakt bekanntem Zeitverhalten und feingranular einstellbaren Zeiten ist dieses Verfahren relativ zeitoptimal.
➔ Es verlangt gleiches Zeitverhalten aller Speicher im Adressbereich, sonst muss auf den langsamsten synchronisiert werden.

Als Ausweg ist die Busvariante „asynchron" möglich, wobei die Steuersignale das Zeitverhalten synchronisieren.

Beispielhaft wird das am asynchronen Speicher Lesen dargestellt (Bild 5.22 S.136).

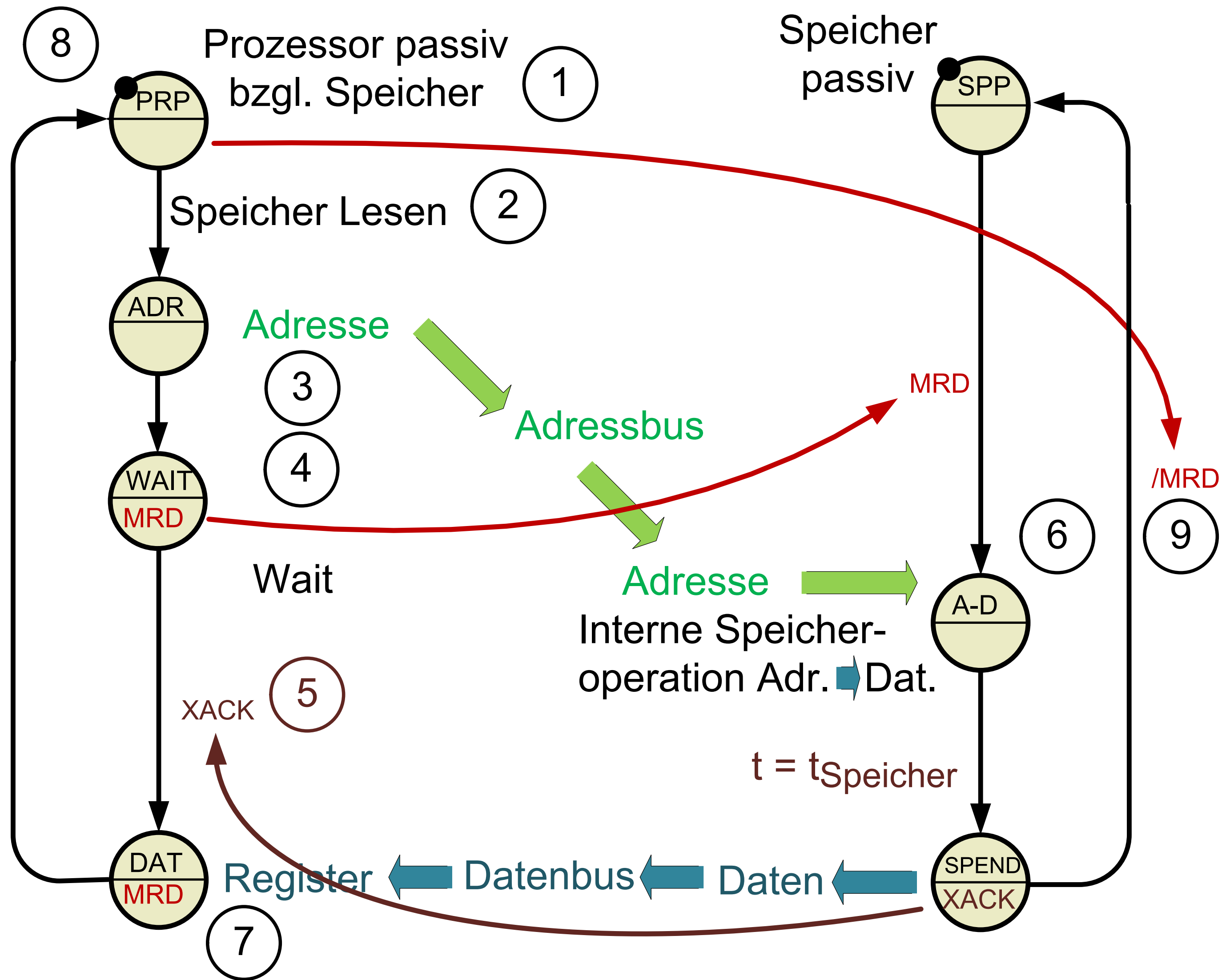


Bild 5.22: Ablauf bei asynchronem Speicher Lesen

Der Ablauf unterscheidet sich zum synchrone Speicher Lesen nur unwesentlich:

(1) bis (4) wie bei synchron.
(5) Der Prozessor wartet auf XACK = 1 (Execution Acknowledge, Ausführungsbestätigung).

(6) Durch MRD=1 verlässt der Speicher seinen passiven Zustand, übernimmt die Adresse vom Adressbus, stellt durch die interne Speicheroperation nach $t_{Speicher}$ die Daten von Adresse auf den Datenbus und setzt XACK auf 1.

(7) Aufgrund von XACK = 1 übernimmt der Prozessor die Daten vom DB in ein internes Register.
(8) Der Prozessor setzt MRD = 0 und wird passiv.
(9) Aufgrund von MRD = 0 setzt der Speicher XACK = 0 und wird passiv.

Dieses ist ein relativ grundlegendes Synchronisationsprinzip und wird später bei der synchronisierten Ein-/Ausgabe (Abschnitt 7.4) noch mal unter den dortigen Anwendungsgesichtspunkten behandelt.

Für Lesen und Schreiben auf Ein-/Ausgabe-Schnittstellen (E/A; Eingabe, Input; Ausgabe, Output) gelten prinzipiell gleiche Abläufe mit folgenden Unterschieden:

- Die Steuersignale MRD und MWT werden durch IOR (Input Output Read) bzw. IOW (Input Output Write) ersetzt.
- Zustände und Funktionen, die in den Bildern und Erläuterungen mit „Speicher“ gekennzeichnet sind, sind alternativ mit „Ein-/Ausgabe-Schnittstelle“ zu bezeichnen.

Bei Zugriffen auf Ein-/Ausgabe-Schnittstellen kann auch das synchrone oder asynchrone Prinzip angewendet werden.

Zumeist ist der Adressbereich für E/A deutlich geringer als für Speicher:

Beispiel: Speicher 32 Bit (A_{31} bis A_0), EA: 8 Bit (A_7 bis A_0)

6. Speicher

Speicher haben im Rechner entsprechend dessen Gesamtarchitektur (Bild 3.1 S.32) als Aufgaben:

Speichern von Maschinenbefehlsfolgen (Programme) und Operanden (Daten):

➔ Programmspeicher, Datenspeicher

➔ Princeton-Architektur: Sie besitzen eine gemeinsame Schnittstelle (Systembus) und befinden sich im gemeinsamen Adressraum (Abschnitt 3).

Als weitere Funktionen sind dabei notwendig:

- Lesen und Schreiben von Speicherinhalten. Hier gilt: Ein geschriebener Wert kann beliebig oft gelesen werden, bis er neu überschrieben wird.
- Im normalen Befehlsablauf gilt: Es erfolgt nur Lesen im Programmspeicherbereich und Lesen und Schreiben im Datenspeicherbereich.
- Das Laden von Maschinencode ist logisch gesehen ein Operand Speichern.
- Für Lesen und Schreiben ist eine Auswahl des Speicherplatzes/-wortes, auf den geschrieben oder gelesen wird, notwendig. Hier gilt das Prinzip Adressierung (Abschnitt 3).

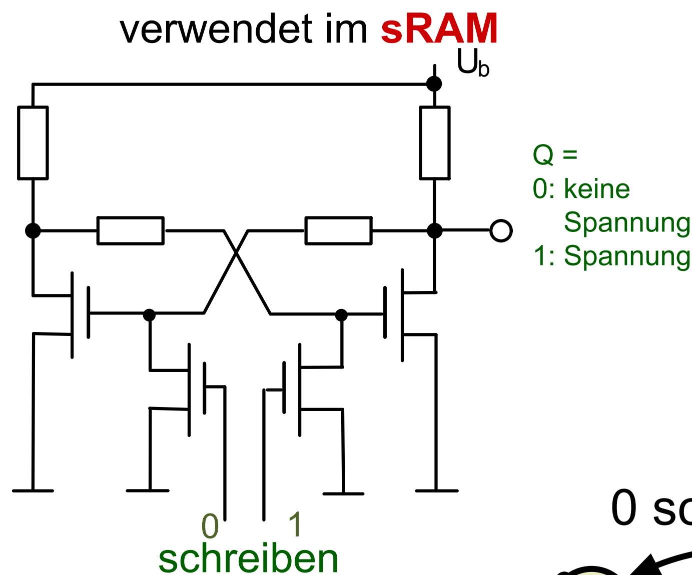
6.1. Speicherbit und Speicherwort

Als Speicherbit gilt: Speichermöglichkeit für genau einen binären variablen Wert (0 oder 1).

Ein Speicher-IC (Integrated Circuit) ist: ein Schaltkreis für m Speicherbyte/-worte (siehe Abschnitt 3).

Flip-Flop (bistabile Kippstufe)

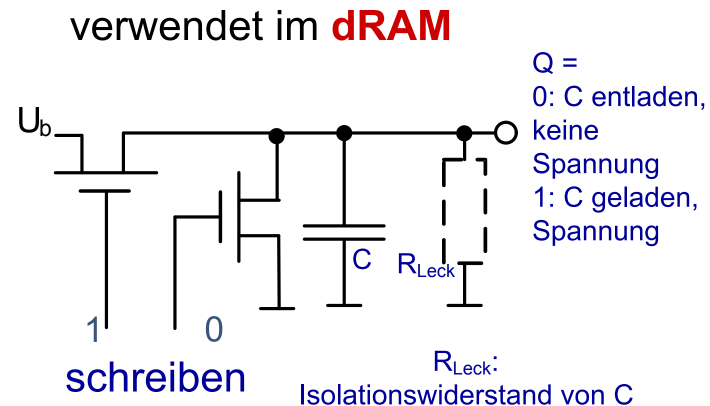
verwendet im **sRAM**



Q =
0: keine Spannung
1: Spannung

Kondensator C

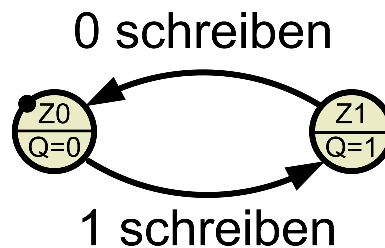
verwendet im **dRAM**



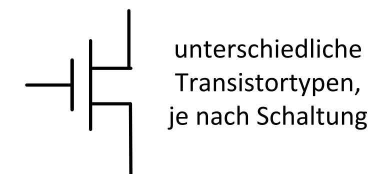
Q =
0: C entladen, keine Spannung
1: C geladen, Spannung

R_{Leck}: Isolationswiderstand von C

0 schreiben



1 schreiben



unterschiedliche Transistortypen, je nach Schaltung

Bild 6.1: Statisches und dynamischer Speicherbit (prinzipiell)

Es gibt verschiedene Varianten für Speicherbit: Bild 6.1 S.140.

Bild 6.1 S.140 links:

Flip-Flop (auch bistabile Kippstufe, siehe [3], [1]).

➔ Dafür wird je Bit eine logisch/schaltungstechnisch minimale Funktion realisiert.

➔ Der Grund ist: möglichst kleine Halbleiterfläche auf dem IC (-> möglichst viele Bit, d. h. große Speicherkapazität).

Oben, links ist eine Ersatzschaltung angedeutet (8 Bauelemente).

Unten: Ersatzmodell, als Automat mit 2 Zuständen:

Z0 Kodierung (Bit): 0

Z1 Kodierung (Bit): 1

Zustandsübergänge:

Bewertet mit zwei Variablen: „0_schreiben“, „1_schreiben“ (Eingänge),

Ausgang Q: Wert von Z1.

Es gilt: Die Speicherfunktion ist nur bei stabiler Betriebsspannung (U_b) stabil. -> Bei Unterbrechung (Ausschalten, Ausfall) -> Verlust des Speicherinhalts.

Die bistabile Kippstufe ist Basis des „statischen RAM, sRAM“.

RAM: Random Access Memory, Speicher mit wahlfreiem Zugriff,

s: statisch.

Bild 6.1 S.140 oben, rechts (Ersatzschaltbild):

Als Alternative zum sRAM-Bit ist ein dRAM-Bit (dynamischer RAM) möglich.

Speicherelement: Kondensator C (Ladungsspeicher) mit

- geladen (1) bzw.
- nicht geladen (0).

1_schreiben ist auf Spannung (U_C=U_b) aufladen, der entsprechende Schalttransistor wird durchgesteuert (kurzzeitig).

0_schreiben ist entladen (U_C=0, entsprechender Schalttransistor wird durchgesteuert (kurzzeitig).

Der Bit-Ausgang Q ist die Spannung U_C über C.

R_{Leck} ist der fertigungsbedingte nicht vermeidbare endliche Isolationswiderstand.

Aufgrund dieses Widerstands speichert C die Ladung nicht beliebig lange, sondern entlädt sich. Bild 6.2 S.144 zeigt diesen Sachverhalt. Nach schreiben von 1 (Aufladen von C auf U_b entlädt sich C nach einer e-Funktion. Ohne weitere Maßnahmen würde die Spannung über C erst den gesicherten Spannungs-Logikbereich für 1 unterschreiten und letztendlich auf den Logikpegel für 0 absinken. Um das zu vermeiden, wird die 1 zeitzyklisch nach $t_{Refresh}$ wieder mit erneutem Aufladen auf U_b „aufgefrischt“ (Refresh, Bild 6.2 S.144). Dieses muss im gesamten Speicher regelmäßig erfolgen, wobei der Zeitabstand vergleichsweise klein ist (Größenordnung 1 ms). Aufgrund dessen ist die Speicherfunktion ohne U_b nicht stabil. Es gilt wie beim sRAM: -> Bei Unterbrechung (Ausschalten, Ausfall) -> Verlust des Speicherinhalts.

Das Prinzip des Refresh ist:

(1) Adresse i Lesen und Rückschreiben
(2) nächste Adresse i+1
(3) Adresse i+1 Lesen und Rückschreiben
… für alle Adressen, dann ständig wiederholen, wenn der Speicher

eingeschaltet ist.

Der Bauelementeaufwand im dRAM je Bit ist ca. 3.

➔ Auf der gleichen IC-Fläche mit der gleichen Halbleitertechnologie ist die mögliche Speicherkapazität eines dRAM das 3-bis-4-fache eines sRAM.

Dafür sind dRAM’s i. Allg. langsamer als sRAM’s.

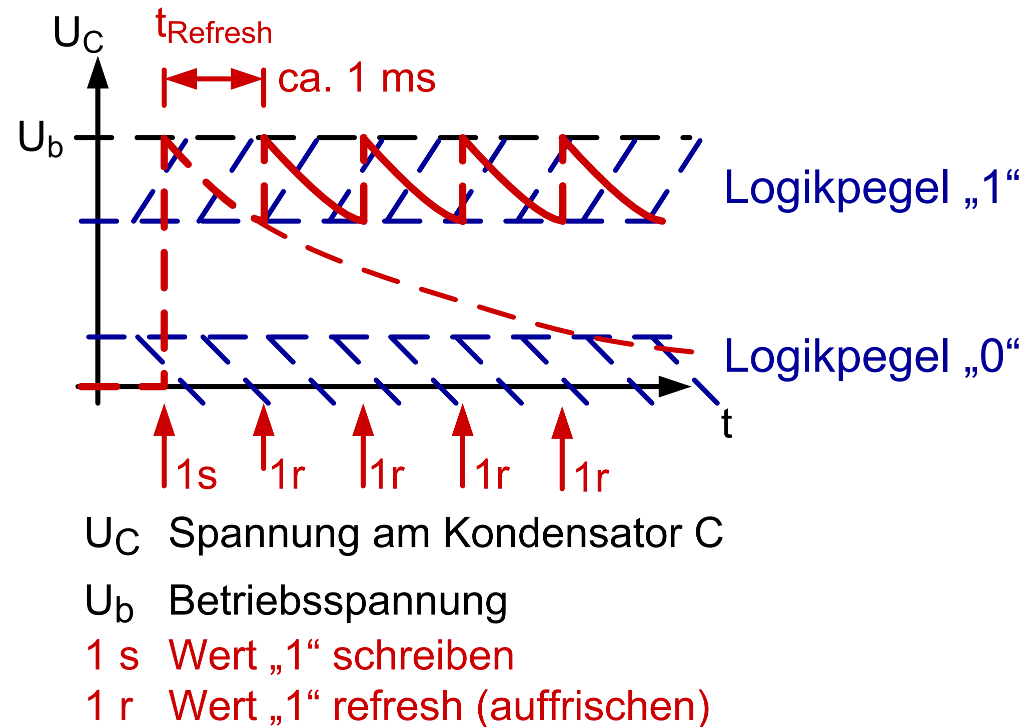


Bild 6.2: Refresh beim dRAM-Bit

Als Anwendungskriterien gelten deshalb:

- Kleine schnelle Speicher (z.B. Caches, siehe Abschnitt 8.4.1) -> sRAM,
- große etwas langsamere Speicher (z.B. Hauptspeicher) -> dRAM.

Der Prozessor benötigt zumindest beim Starten (erstes Maschinenprogramm nach Rücksetzen, Reset bzw. beim Einschalten) einen permanenten Programmspeicher, um Programme von einem Massenspeicher zu laden, der den Speicherinhalt nicht verliert (z.B. Festplatte, SSD).

Kleine Eingebettete Rechner (siehe Einführung) haben oft keinen externen Massenspeicher.

➔ Es muss der gesamte Programmspeicher auch ohne Spannung (U_b) (Betriebsspannung) erhalten bleiben.

➔ Notwendig sind Bit, die diese Eigenschaft haben.

ROM-Bit (Read Only Memory, Nur Lesespeicher)

Es gibt verschiedene Varianten, das einfachste Beispiel ist maskenprogrammierbar:

Der Speicherinhalt (Bit-Muster) wird als Verdrahtung während der Halbleiterfertigung hergestellt.

Das Bit-Muster ist nicht änderbar (z.B. keine update-Programme).

Es entsteht größerer Entwicklungsaufwand, da ein Halbleiterentwurfsschritt dafür notwendig ist.

➔ Andere Varianten:

Ein- oder mehrmalig programmierbar.

Das Schreiben (programmieren) erfolgt nicht im gleichen Modus wie übliches RAM-schreiben. Deshalb ist es langsamer.

Das Lesen geschieht im normalen Speichermodus.

Als typ. Vertreter kann der Flash-ROM genannt werden.

Im Weiteren wird zumeist von der konkreten Realisierung des Speicherbit abstrahiert und eine abstrakte Bitzelle benutzt (Bild 6.3 S.147).

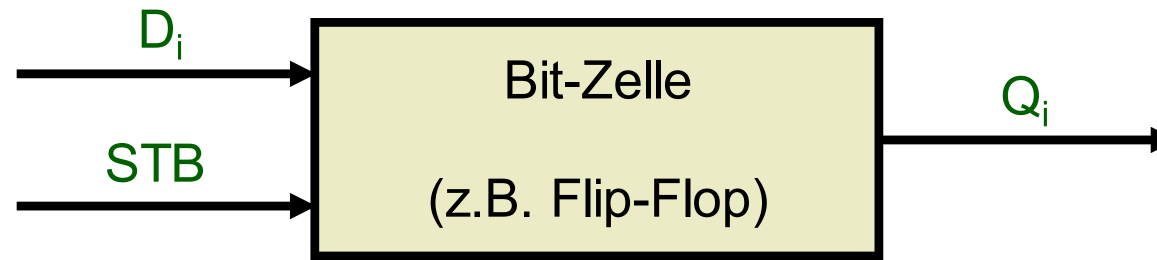
Gegenüber dem Automatengraphen in Bild 6.1 S.140 wird ersetzt:

0 schreiben durch STB & /D_i,

1 schreiben durch STB & D_i und

Q durch Q_i.

D.h., es gibt nur eine Variable D_i als Dateneingang, der Wert von D_i wird mit einem Takt STB (Strobe) übernommen (gespeichert) und Q_i hat den letzten gespeicherten Wert von D_i. Durch die Indizierung mit i wird die Unterscheidung mehrere Speicherbit im Folgenden möglich. Durch STB wird die Bitzelle zum synchronen Automaten.



D_i - Daten-Eingang
STB - Strobe (Speichertakt)
Q_i - Daten-Ausgang

	z_i	Q_i
Z0	0	0
Z1	1	1

Bild 6.3: Abstrakte Bitzelle

Ein Speicherwort/-byte besteht aus n bzw. 8 Bit mit gleichzeitigem Lesen und Schreiben. Die Basis ist ein Register aus n bzw. 8 (abstrakten) Bit-Zellen (die physische Bit-Zellen repräsentieren, Bild 6.4 S.148).

Eingang

D_{31} D STB Bit 31 Q Q_{31}

D_1 D STB Bit 1 Q Q_1

D_0 D STB Bit 0 Q Q_0

STB

Ausgang

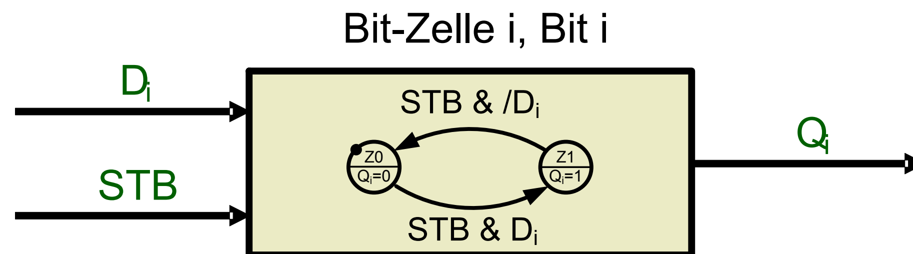


Bild 6.4: 32-Bit-Register aus abstrakten Bitzellen

Die Dateneingänge Di sind Datenausgängen Qi paarweise zugeordnet, die untereinander getrennt sind.

Das gemeinsame Schreiben wird durch verbundenes STB aller Bit-Zellen erreicht.

Ein komplettes Speicherwort entsteht aus

- n-Bit-Register,
- n-Bit Treiber,

angeordnet wie in Bild 6.5 S.150.

Die zusätzlichen Tristate-Treiber sind Elemente zum Durchschalten bzw. Abschalten von den logischen Ausgängen Q_i von den externen bidirektionalen Daten-Ein/Ausgängen D_i. Dafür besitzen Tristate-Treiber die Möglichkeit, mit einem Signal OE = 0 (Output Enable) die Ausgänge vollständig abzuschalten (offen oder hochohmig) bzw. bei OE=1 die Eingänge auf die Ausgänge durchzuschalten (DO = DI) (Bild 6.6 S.151). Da ein offener Ausgang in der binären Logik nicht definiert ist, wird von „Tristate“ (dritter Zustand) gesprochen. Es handelt sich dabei aber um keinen dritten logischen Zustand eines Bit. Für hochohmiges DO sind die Logikausgänge praktisch nicht vorhanden (d. h. weder 0 noch 1 noch undefiniert). Die angedeutete Prinzipschaltung soll in Zusammenhang mit der Wertetabelle zeigen, dass bei

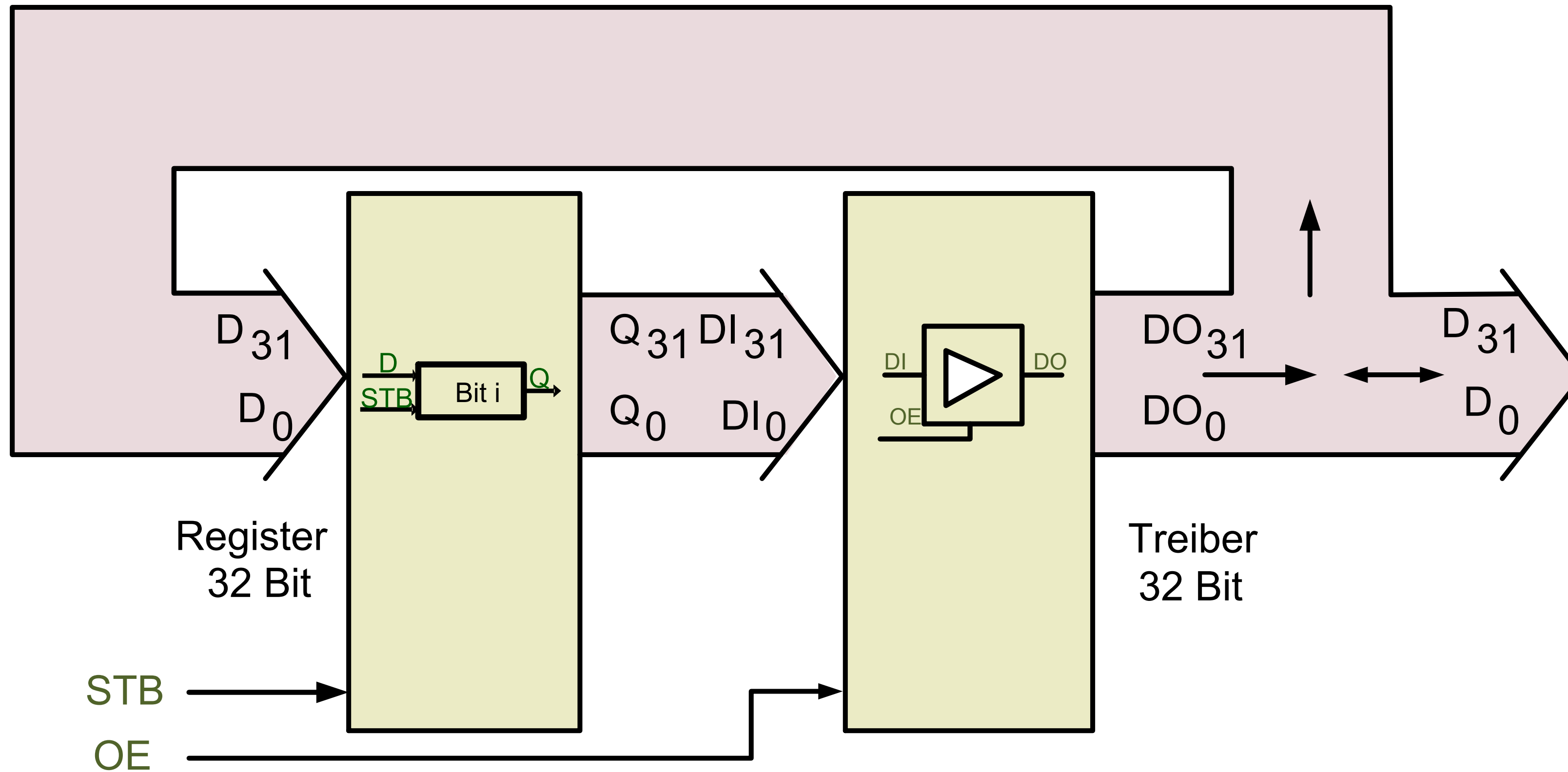
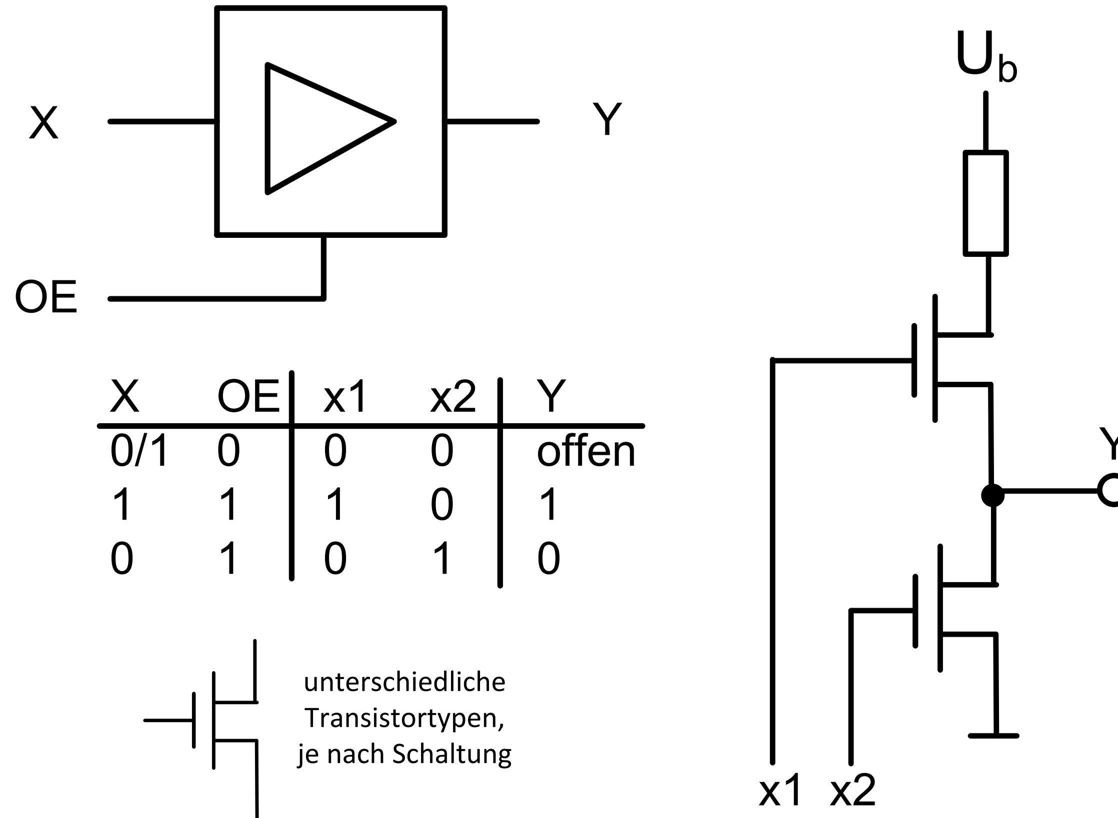


Bild 6.5: 32-Bit-Speicherwort

OE = 0 beide Transistoren gesperrt sind und damit keine Verbindung zur 0-Spannung (ca. 0 V, Spannung für 0) oder U_b (ca. Spannung für 1) existiert.

Bei OE = 1 erfolgt die Übertragung konventioneller logischer Signale mit den Werten 0 oder 1.



X	OE	x1	x2	Y
0/1	0	0	0	offen
1	1	1	0	1
0	1	0	1	0

Bild 6.6: Tristate-Treiber mit Ausgangsschaltung (prinzipiell)

Der 32-Bit-Treiber von Bild 6.5 S.150 ist dann:

32 getrennte Tristate-Treiber mit verbundenem gemeinsamem OE, d.h. alle DI sind gleichzeitig auf alle DO geschaltet (OE = 1) bzw. alle DO sind gleichzeitig offen (OE = 0), die DI_i/DO_i-Paare sind voneinander unabhängig.

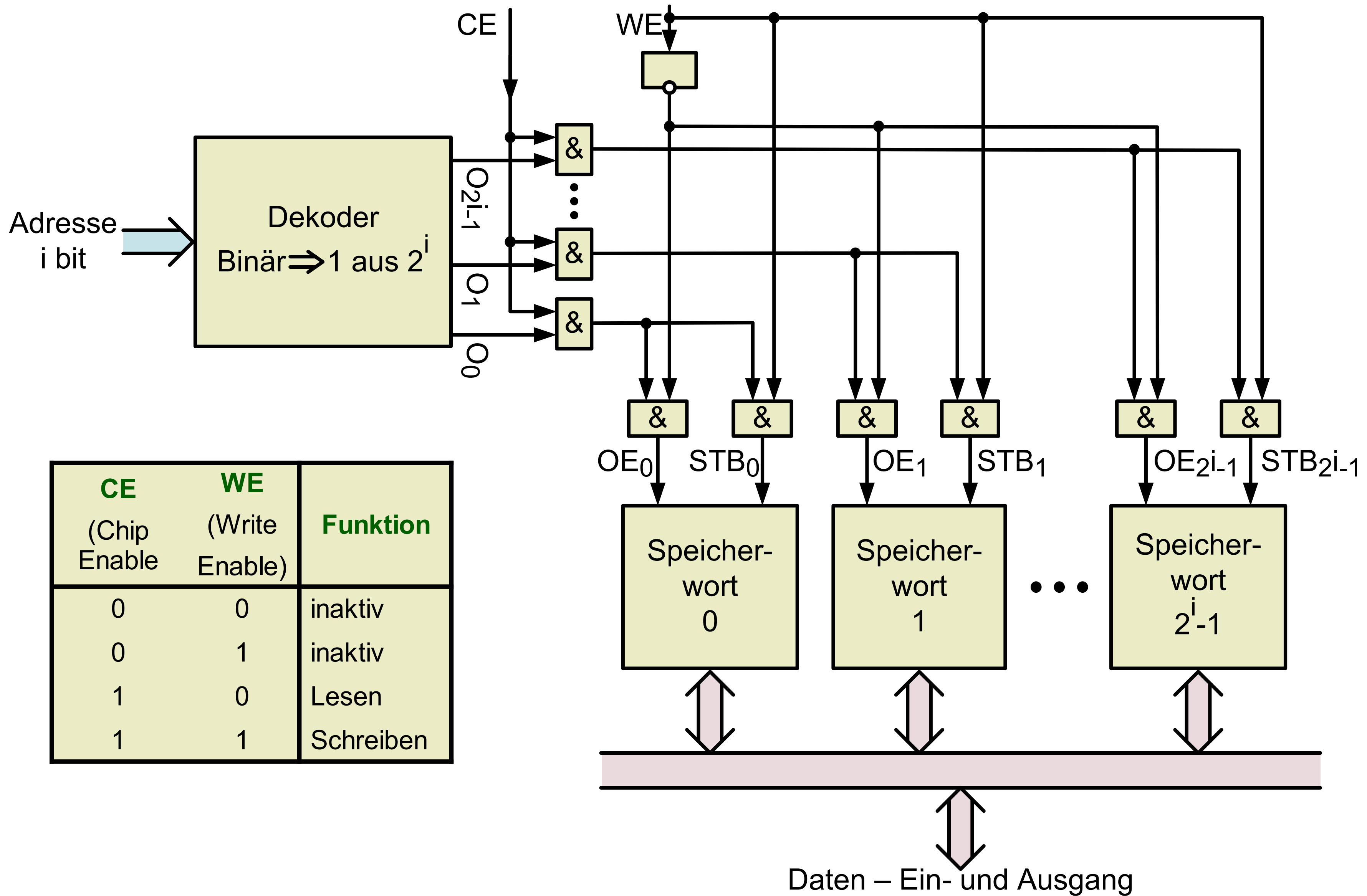
Insgesamt entstehen für das Speicherwort folgende mögliche Arbeitsweisen:

- STB=0 & OE=0 -> Speichern des Registerinhalts, d.h. die Werte bleiben erhalten,
- STB=1 & OE=0 -> Schreiben von Daten, die am bidirektionale Ein-/Ausgang (D_i) anliegen,
- STB=0 & OE=1 -> Lesen des Registerinhalts zum Ein-/Ausgang (D_i) über die durchgeschalteten Tristate-Treiber,
- STB=1 & OE=1 -> Refresh des Registerinhalts. Notwendig sind keine Daten von und zum Ein-/Ausgang (D_i) (interne Funktion, nur bei dRAM).
  In der angegebenen Variante könnten die ausgelesenen und rückgeschriebenen Registerinhalte an den D_i gelesen werden. Das Refresh der Kondensatorladung erfolgt, weil die Tristate-Treiber auch die Funktion eines elektronischen Verstärkers besitzen.

Bei einem ROM ist nur OE vorhanden und die Funktion Lesen. Die abstrakte Bit-Zelle ist entsprechend reduziert (das „Register“ besteht damit aus n Festwert-Bit).

6.2. Speicher-IC

Ein Speicher-IC (Integrated Circuit) besteht aus 2^i Speicherworten (nach dem Prinzip von Bild 6.5 S.150). Es entsteht die Struktur von Bild 6.7 S.153 als IC.



CE (Chip Enable)	**WE** (Write Enable)	**Funktion**
0	0	inaktiv
0	1	inaktiv
1	0	Lesen
1	1	Schreiben

Bild 6.7: Speicher-IC

- ➔ Mit i binären Adresssignalen sind alle Speicherworte eineindeutig unterscheidbar.
- ➔ Die Umwandlung der i Adressbit (A_k) in 2^i Auswahlsignale (O_j, Out_j) erfolgt durch einen durch 1-aus-2^i-Dekoder (Wertetabelle siehe Bild 6.8 S.154).

A_{i-1}	A_{i-2}		A_1	A_0	O_{2^i-1}	O_{2^i-2}		O_1	O_0
0	0		0	0	0	0		0	1
0	0		0	1	0	0		1	0
0	0		1	0	0	0		0	0
1	1		1	0	0	1		0	0
1	1		1	1	1	0		0	0

Bild 6.8: Wertetabelle zum 1-aus-2^i-Dekoder

Für jedes Speicherwort gilt:

$OE_j = O_j$ & CE & /WE mit: CE (Chip-Enable) ist das Auswahlsignal für den gesamten IC.

$STB_j = O_j$ & CE & WE mit: WE ist die Schreibfreigabe für den gesamten IC.

Die Daten-Ein-/Ausgänge aller Speicherworte sind (logisch) verbunden. Das ist möglich, da durch O_j immer nur genau ein Speicherwort angewählt wird.

6.3. Speicherfunktionseinheit

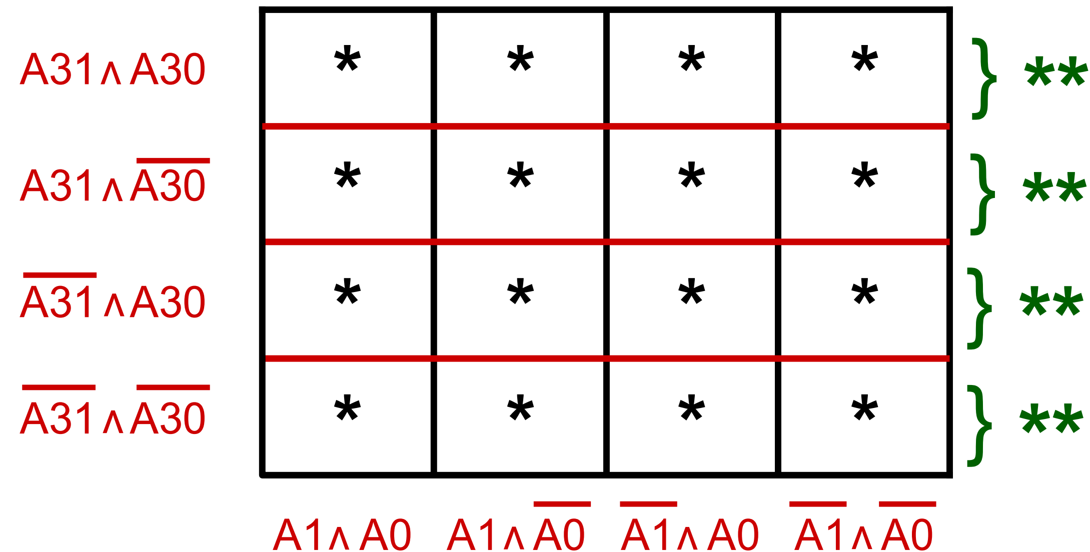
Ausgangspunkt sind Speicher-IC's nach Abschnitt 6.2. Für den Aufbau einer Speicherfunktionseinheit gilt weiterhin:

- Speicherworte (z.B. 32-Bit-Worte) werden zumeist aus Byte-Datenbreite-IC's zusammengesetzt (z.B. 4).
- Der Adressraum eines Prozessors (z.B. 2^{32}) ist zumeist größer als der Adressraum der verfügbaren Speicher (z.B. 2^{28}, d.h. der Adressraum ist 16-fach größer). Bei größerem verfügbaren Speicher gilt diese Aussage für Prozessoren mit größerem Adressraum (z.B. 2^{64}) wieder sinngemäß.

Für den Beispiel-Fall ist die Anordnung nach Bild 6.9 S.156 möglich:

➔ Vier Spalten aus IC's mit 1-Byte-Datenbreite und Unterscheidung der Spalten durch alle Binärkombinationen von A1, A0,

➔ für Speicher-IC‘s mit 28 Adresseingängen sind vier Zeilen möglich, die durch A31, A30 unterschieden werden.



* 1 Speicher mit 8 Datenanschlüssen und 28 Adresseingängen

** innerhalb der Speicher wird mit A29…A2 adressiert

Bild 6.9: Speicherstruktur mit 4x4 Speicher-IC’s

Dabei wird der Speicherbyte-IC j,i adressiert:

j wird aus A31, A30 gebildet,

i wird aus A1, A0 gebildet.

Innerhalb des IC's wird die Adresse aus A29 bis A2 gebildet.

Es wird genau eine Kombination je Byte aus

A31, A30: (/)A31 & (/)A30
A1, A0 : (/)A1 & (/)A0
A29, …A2: (/)A29 & (/)A28 & … & (/)A2

vom Adressbus benötigt.

➔ (/) für eine konkrete Belegung genau negiert oder unnegiert.

Zur Realisierung der eben beschriebenen Speicherfunktion dient die Struktur nach Bild 6.10 S.158.

Die Schnittstelle zum Prozessor (siehe auch Prozessorzugriffe über externes Interface, Abschnitt 5.5) sind:

- Der Adressbus (AB),
- der Datenbus (DB) und
- der Steuerbus (SB, davon nur MWT, MRD, Byte-Enable-Signale, BE).

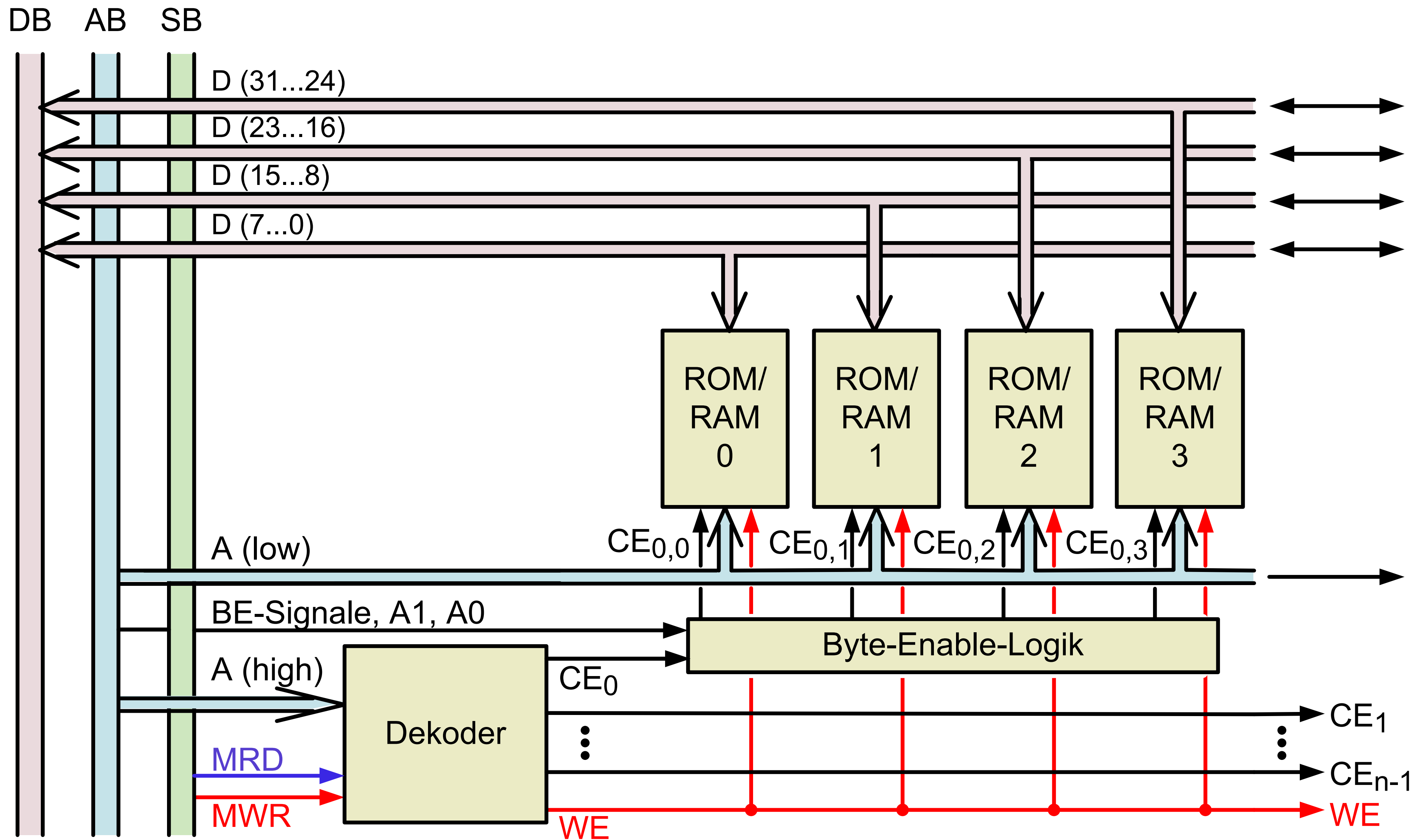


Bild 6.10: Speicherfunktionseinheit

Basis des Speichers sind Gruppen von Speicher-IC's (ROM/RAM j), die je einer Zeile in Bild 6.9 S.156 entsprechen (im Beispielfall 4 Gruppen). Die Auswahl der Gruppen erfolgt mit den Signalen CE_j (j = 0 bis 3), die aus den 4 Binärkombinationen aus A31 und A30 (Adressbussignale A(high)) gebildet werden:

CE_j = (MRD + MWT) & (/)A31 & (/)A30

Die Oderverknüpfung aus MRD (Memory Read) und MWT (Memory Write) kennzeichnet den Speicherzugriff.

Es folgt die Beschreibung der Verwendung der weiteren Adressbussignale. Wie schon zu Bild 6.9 S.156 ausgeführt, soll in der hier beschriebenen Funktionseinheit neben dem 32-Bit-Wortzugriff Byte-weiser Zugriff möglich sein (Spalten in Bild 6.9 S.156). Das wird ergänzt durch einen 16-Bit-Wortzugriff

Dazu werden zusätzliche Steuerausgänge des Prozessors (Byte-Enable-Signale, BE) notwendig:

BY (Byte), WD16 (Word 16), WD32 (Word 32).

Die CE-Signale der einzelnen IC's in der Gruppe sind dann:

$CE_{j,0}$ = CE_j & (WD32 + WD16 & /A1 + BY & /A1 & /A0)

$CE_{j,1}$ = CE_j & (WD32 + WD16 & /A1 + BY & /A1 & A0)

$CE_{j,2}$ = CE_j & (WD32 + WD16 & A1 + BY & A1 & /A0)

$CE_{j,3}$ = CE_j & (WD32 + WD16 & A1 + BY & A1 & A0)

A (low): A29 … A2 wird an die Adresseingänge aller Speicher-IC's geschaltet. Sie wählen in diesen genau ein Speicherbyte aus.

Vom Steuerbus werden nochmals MWT (Memory Write) und MRD (Memory Read) genutzt und bilden WE (Write Enable), das mit allen Speicher-IC's verbunden wird:

WE = (MWT & /MRD)

Ein Beispiel-Speicherzugriff in der Struktur von Bild 6.10 S.158 zeigt Bild 6.11 S.161. Dabei bewirken:

A (high) = /A31 & /A30 = 1 einen Zugriff in der Wortgruppe 0 (CE_0 = 1), (1)

BY = 1 und /A1 & /A0 = 1 führen zum Byte-Zugriff auf ROM/RAM 0 ($CE_{0,0}$ = 1), (2)

/A29 & … & /A3 & A2 = 1 führen zur Auswahl von Byte 1 (das vorletzte) in ROM/RAM 0 (alle Adressbit von A28 bis A3 sind negiert), (3)

MRD = 1 führt zum Lesezugriff (damit WE = 0), d. h. Datenrichtung vom ROM/RAM 0 zum Datenbus (DB) (4).

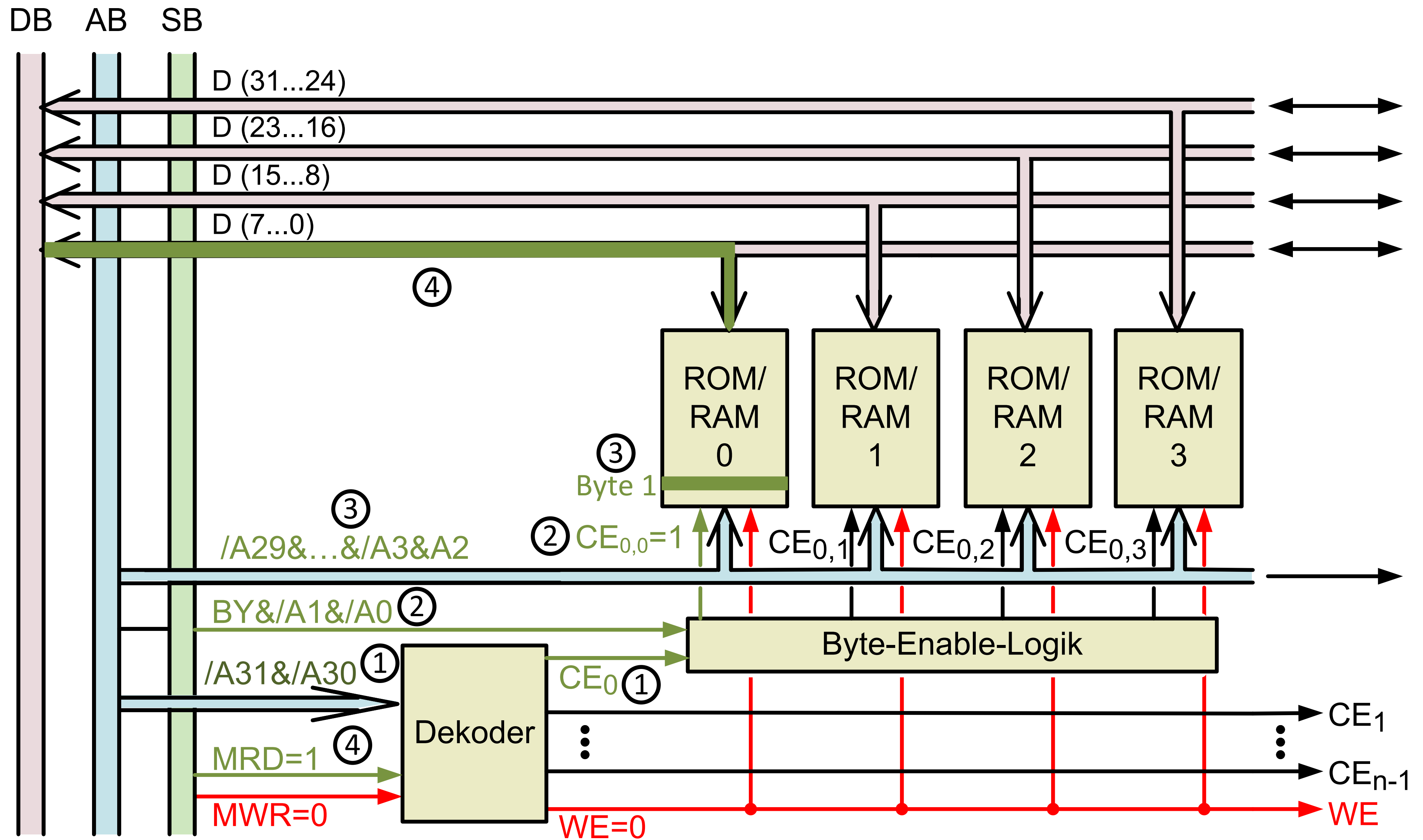


Bild 6.11: Beispielzugriff in der Speicherfunktionseinheit

7. Ein- und Ausgabe

Ein-/Ausgabeschnittstellen (E-/A-Schnittstellen, EA, Input/Output, IO) haben im Rechner als Aufgaben:

Ein- bzw. Ausgabe von Eingangs- bzw. Ausgangsoperanden (Ergebnissen) von und zur Umwelt/Peripherie.

Umwelt:

- User (Nutzer): Tastaturen, Monitore, graphische Eingabegeräte …,
- externe Gräte: Massenspeicher, wie Festplatte, SSD; Drucker, Scanner,
- Sensoren, Aktoren zu technischen bzw. technologischen Prozessen,
- Rechnernetze
- …

Der Gegenstand in der Rechnerarchitektur reicht hier bis zu einer elektrischen, evtl. standardisierten Schnittstelle (Bild 7.1 S.163).

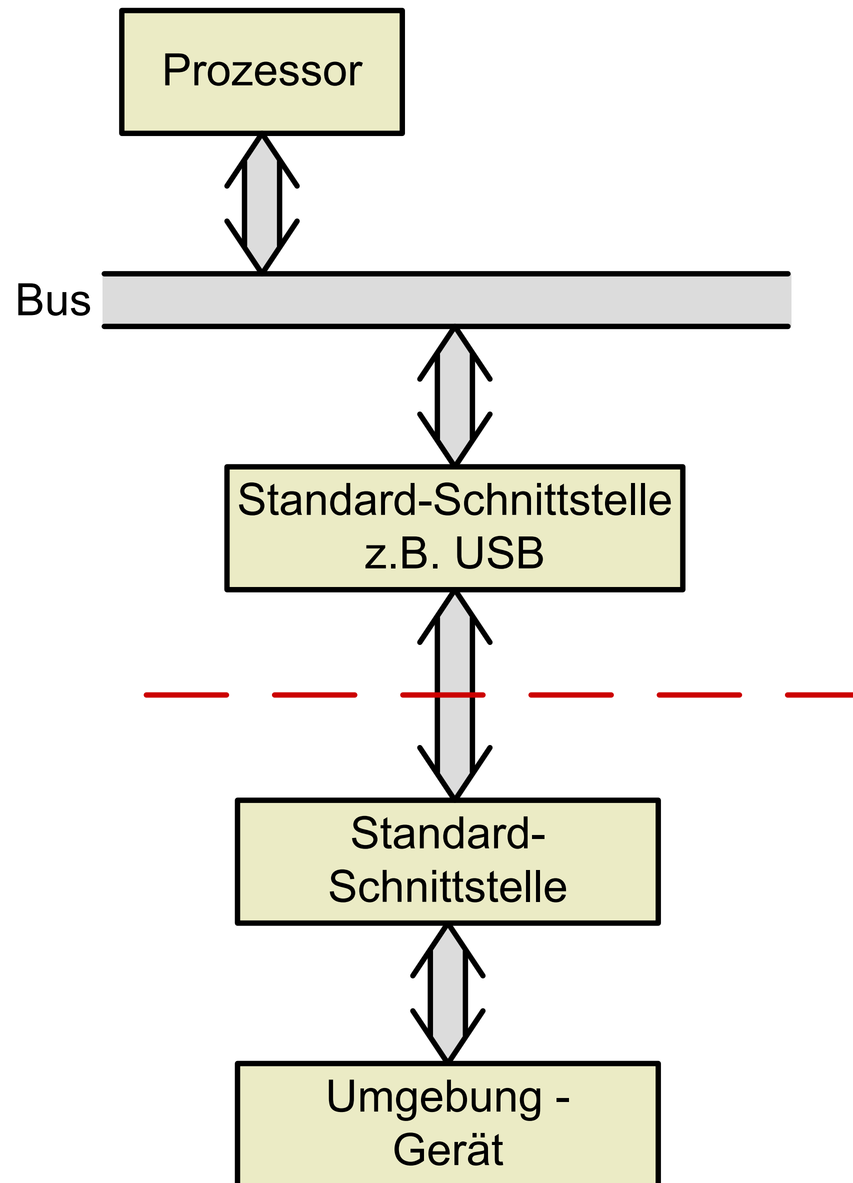


Bild 7.1: Einordnung der Ein-/Ausgabe in die Rechnerarchitektur

E-/A-Schnittstellen sind sehr heterogen aufgrund der sehr unterschiedlichen Umgebungsanforderungen:

➔ Hier werden nur Grundprinzipien beschrieben, die in vielen Schnittstellen vorkommen.

➔ Die Datendarstellung ist dabei der Ausgangspunkt.

Rechnerintern ist die Datendarstellung digital, parallel (Bild 7.2 S.165, siehe auch Bilder 5.2 S.96 und 5.3 S.98).

Es existieren n parallele Datenbit (separate binäre Signale):

- Die Bedeutung des Einzelsignals resultiert aus der Stellung des Bit in den Signalen (in der Datendarstellung ist aber keine Interpretation des Inhalts der Signale vorhanden).
- Es existiert ein Gültigkeitsbereich (Zeit, in der die Signale eine stabilen Wert besitzen).
- Zwischen zwei aufeinander folgende Gültigkeitsbereichen liegt zumeist ein Änderungsbereich mit nicht definierten Signalwerten.
- Es existiert ein Änderungszeitpunkt (Zeitpunkt mit Übergang von einem alten Signalwert zu einem neuen Signalwert (der neue Wert (0 bzw. 1) muss nicht unbedingt anders als der alte sein), $t_{kä}$.

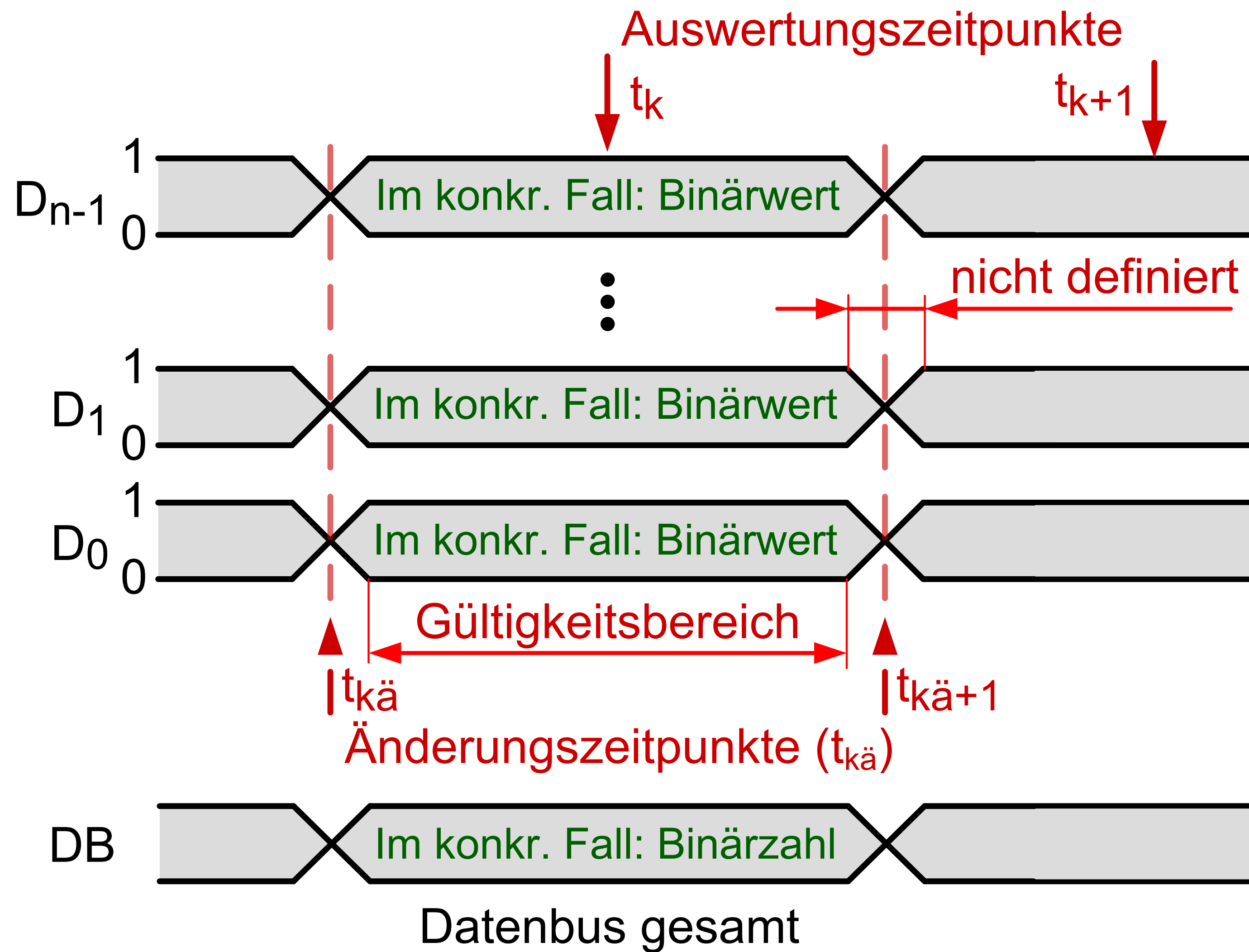


Bild 7.2: Parallele Datendarstellung (z.B. Datenbus oder parallel extern)

- Es existiert i. Allg. ein im Gültigkeitsbereich liegender Auswertungszeitpunkt t_k, t_{k+1}, …, der auch für alle Signale gleich ist und oft etwa in der Mitte liegt (in den Ausführungen: D_i (k) ist Datenbit i von (k) und D (k) ist n * Datenbit von (k)).
- Für eine Zusammenfassung der Signale zu einem zusammenhängenden Datenelement gilt: Gültigkeitsbereiche und Änderungszeitpunkte sind i. Allg. gleich.

7.1. Parallele digitale Ein-/Ausgabe

Bild 7.3 S.166 zeigt eine Blockstruktur für die parallele digitale Ausgabe.

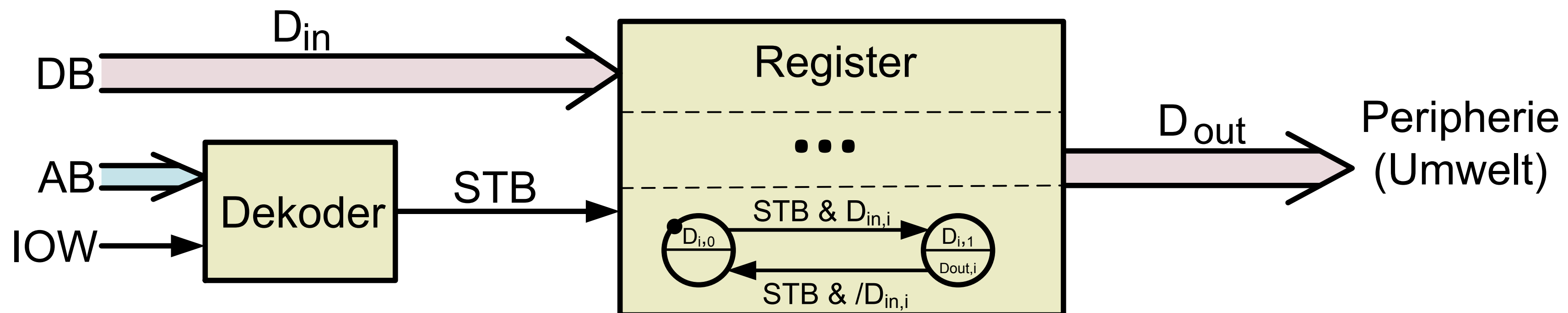


Bild 7.3: Parallele Digitale Ausgabe

Die Datendarstellung extern (zur Umwelt bzw. Peripherie) ist parallel digital nach dem Prinzip von Bild 7.2 S.165.

Sie ist intern auch parallel digital, wobei für die Daten der Datenbus relevant ist.

Erläuterungen zu den Blöcken:

Register: Es übernimmt zum Ausgabezeitpunkt (Teilphase im OUT-Befehl) mit STB=1 die Daten vom Datenbus und speichert sie bis zur nächsten Ausgabe (Gültigkeitsbereich). Die Registerausgänge stehen der Umwelt/Peripherie für den gesamten Gültigkeitsbereich zur Verfügung. (Register siehe auch Bild 6.4, S.148.)

Der Dekoder bildet aus dem Adressbus (ggf. nur einem Teil davon, z.B. 8 Adresssignale) ein Auswahlsignal aus der konkreten EA-Adresse dieser Schnittstellen und dem Steuerbussignal IOW (Input Output Write; EA-Schreiben).

Bsp.: OUT (0F8H), EAX

STB = IOW & A7 & A6 & A5 & A4 & A3 & /A2 & /A1 & /A0

(Gleichung der kombinatorischen Logikfunktion des Dekoders dieses konkreten Beispiels).

Die Phasen des OUT-Befehls (in Anlehnung an Speicher Schreiben, Bild 5.9 S.109):

IF (Instruction Fetch, Programmspeicherzugriff),

ID (Instruction Decode, intern),

RR (Register Read, intern hier Daten vom EAX),

OW (Output Write, externer Zugriff auf die Ausgabeschnittstelle mit Adresse 0F8H auf dem Adressbus und IOW=1 und Daten von EAX auf dem Datenbus),

NIA (Next Instruction Address, intern).

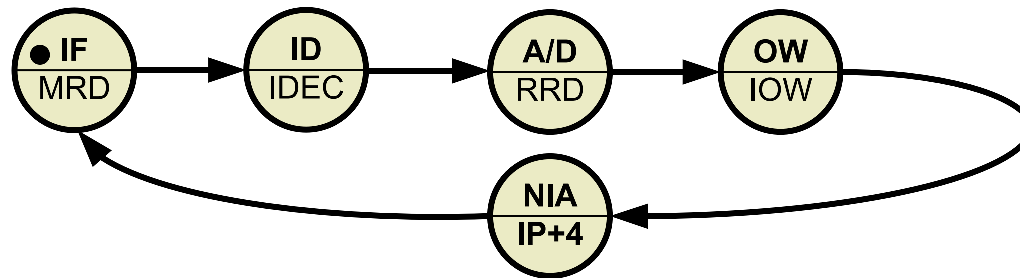


Bild 7.4: Befehlsabarbeitung eines OUT-Befehls

Der Zeitablauf (Bild 7.5 S.169) der Output-Write-Phase (OW) sieht dabei wie folgt aus:

(1) Der Prozessor legt die E/A-Adresse (Bsp. 0F8H) auf den Adressbus und die Daten vom Register (Bsp. EAX) auf den Datenbus. D_{out} (die Daten zur Umwelt) haben noch die Daten der vorherigen Ausgabe.

(2) Der Dekoder bildet STB=1 aus der Adressbusbelegung zusammen mit IOW=1.

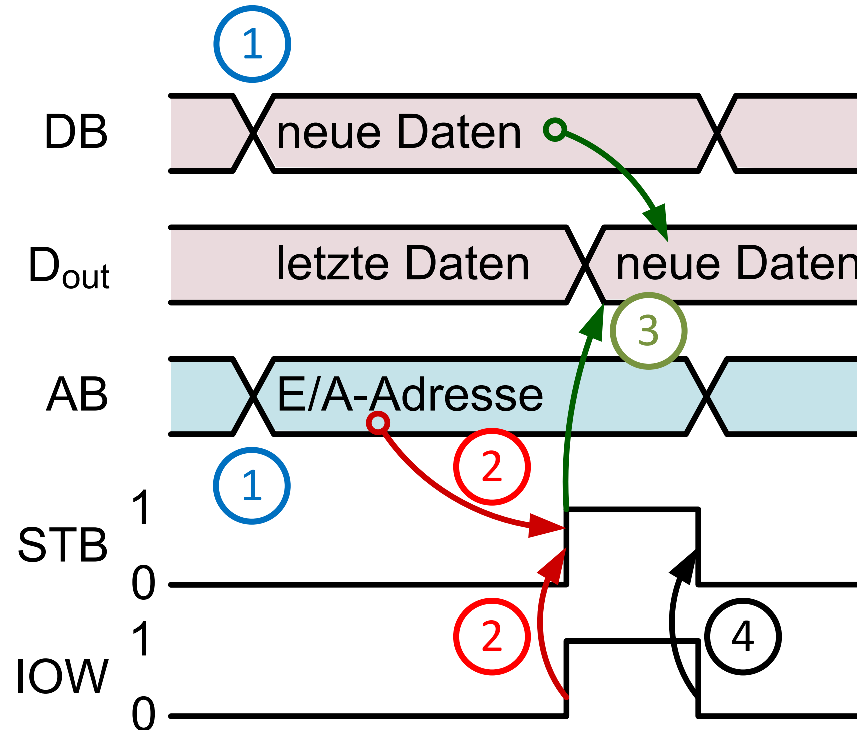


Bild 7.5: Zeitablauf der Output-Write-Phase

(3) STB=1 führ zum Einspeichern der Daten vom DB in das Register, dessen Ausgänge bis zur Folgeausgabe der Umwelt zu Verfügung stehen.

(4) Der Prozessor beendet die Phase: IOW = 0 (Prozessor) (und setzt anschließend den AB auf einen neuen Wert) -> STB = 0 (Dekoder).

Bild 7.6 S.170 zeigt eine Blockstruktur für die parallele digitale Eingabe.

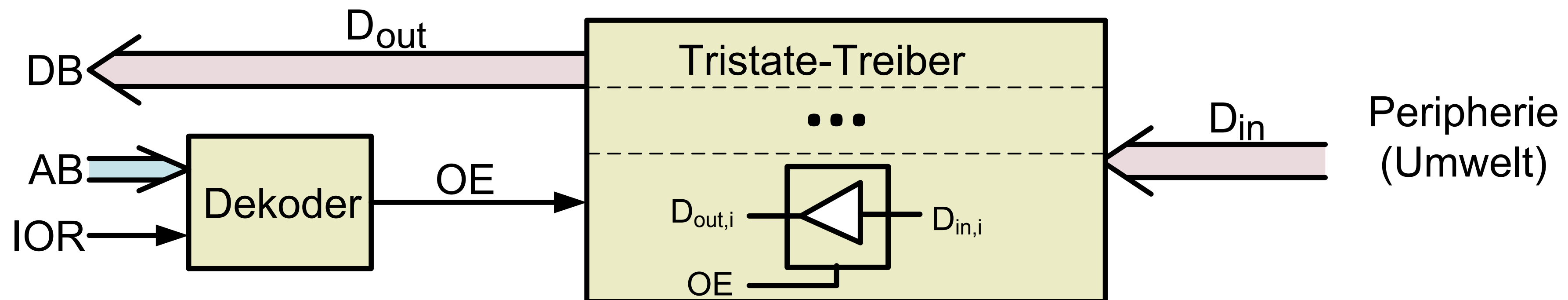


Bild 7.6: Parallele Digitale Eingabe

Die Datendarstellung extern (von der Umwelt bzw. Peripherie) ist parallel digital nach dem Prinzip von Bild 7.2 S.165.

Sie ist intern auch parallel digital, wobei für die Daten der Datenbus relevant ist.

Erläuterungen zu den Blöcken:

Tristate-Treiber: Er schaltet innerhalb des Eingabebefehls (IN) mit OE = 1 die Daten der Umwelt (Din) auf den Datenbus und ist bzgl. des DB sonst hochohmig (abgeschaltet). (Tristate-Treiber siehe auch Bild 6.5 S.150 und Bild 6.6 S.151.)

Der Dekoder bildet aus dem Adressbus (ggf. nur einem Teil davon, z.B. 8 Adresssignale) ein Auswahlsignal aus der konkreten EA-Adresse dieser Schnittstelle und dem Steuerbussignal IOR (Input Output Read; EA-Lesen).

Bsp.: IN EAX, (078H)

STB = IOR & /A7 & A6 & A5 & A4 & A3 & /A2 & /A1 & /A0

(Gleichung der kombinatorischen Logikfunktion des Dekoders dieses konkreten Beispiels).

Die Phasen des IN-Befehls (Bild 7.7 S.171) sind in Anlehnung an Speicher Lesen, Bild 5.8 S.107, ähnlich wie bei einem OUT-Befehl (Bild 7.4 S.168).

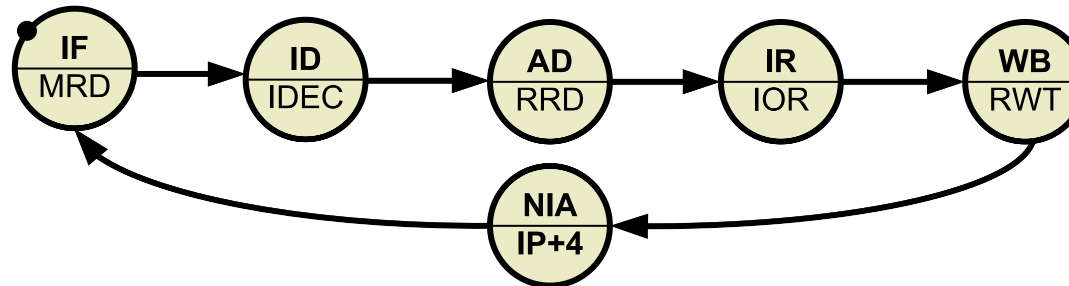


Bild 7.7: Befehlsabarbeitung eines IN-Befehls

Der Zeitablauf (Bild 7.8 S.172) der Input-Read-Phase (IR) sieht dabei wie folgt aus.

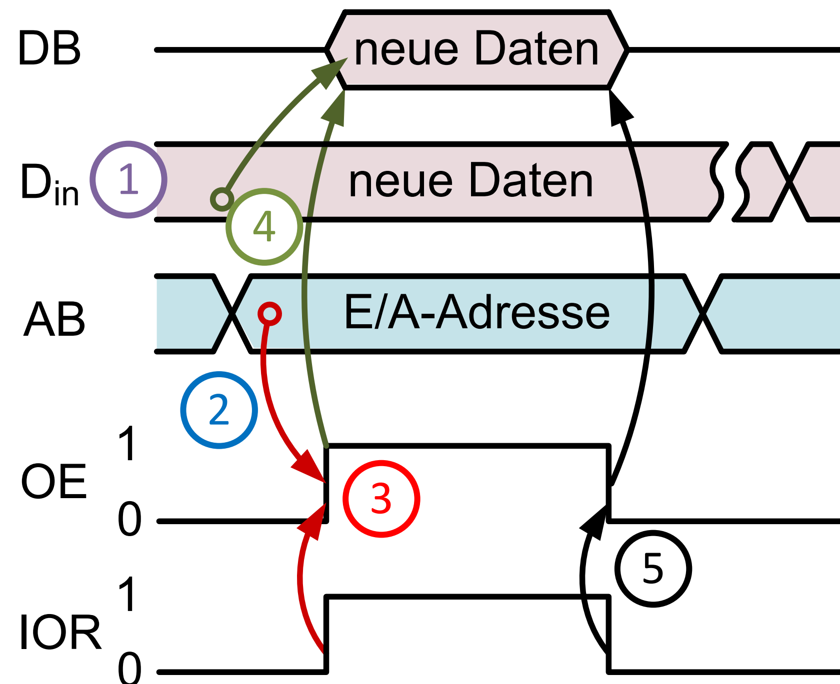


Bild 7.8: Zeitablauf der Input-Read-Phase

(1) Die Umwelt gibt ihre Daten stabil während der Eingabe auf D_{in} der Treiber.

(2) Der Prozessor gibt die EA-Adresse (Bsp. 078H) auf den AB.

(3) Der Prozessor setzt IOR = 1 -> aufgrund der gültigen Adresse und IOR = 1 -> OE=1.

(4) Aufgrund von OE = 1 wird D_{in} durch den Treiber auf den DB geschaltet -> der Prozessor liest vom DB in ein internes Register (z.B. EAX).

(5) Der Prozessor beendet die Phase mit IOR = 0 (und setzt anschließend den AB auf einen neuen Wert) -> OE = 0 (Dekoder) -> Der Treiber wird vom DB abgeschaltet und wird an den Ausgängen hochohmig.

In Anlehnung an Bild 5.19 S.132 (Speicher Lesen) und Bild 5.21 S.134 (Speicher Schreiben) kann das Zusammenwirken von mehreren EA-Schnittstellen mit den parallelen Automaten nach Bild 7.9 S.174 beschrieben werden.

Dabei modelliert der linke Automat (schwarz) den Prozessor mit den Möglichkeiten zur Aus- und Eingabe (mögliche andere externe Zugriffe, wie Speicher Lesen oder Schreiben sind nicht dargestellt). Die beiden oberen rechten Automaten (grün) modellieren zwei Eingabeeinheiten (mit unterschiedlicher eigener Adresse) und der untere rechte Automat (blau) eine Ausgabeeinheit mit einer weiteren eigenen Adresse.

Die beispielhafte Befehlsfolge

IN EAX, (078H)

OUT (0F8H), EAX

mit den Daten 012H an der Eingabeschnittstelle mit der Adresse 078H führt zu folgendem Ablauf (auf des Wesentliche reduziert):

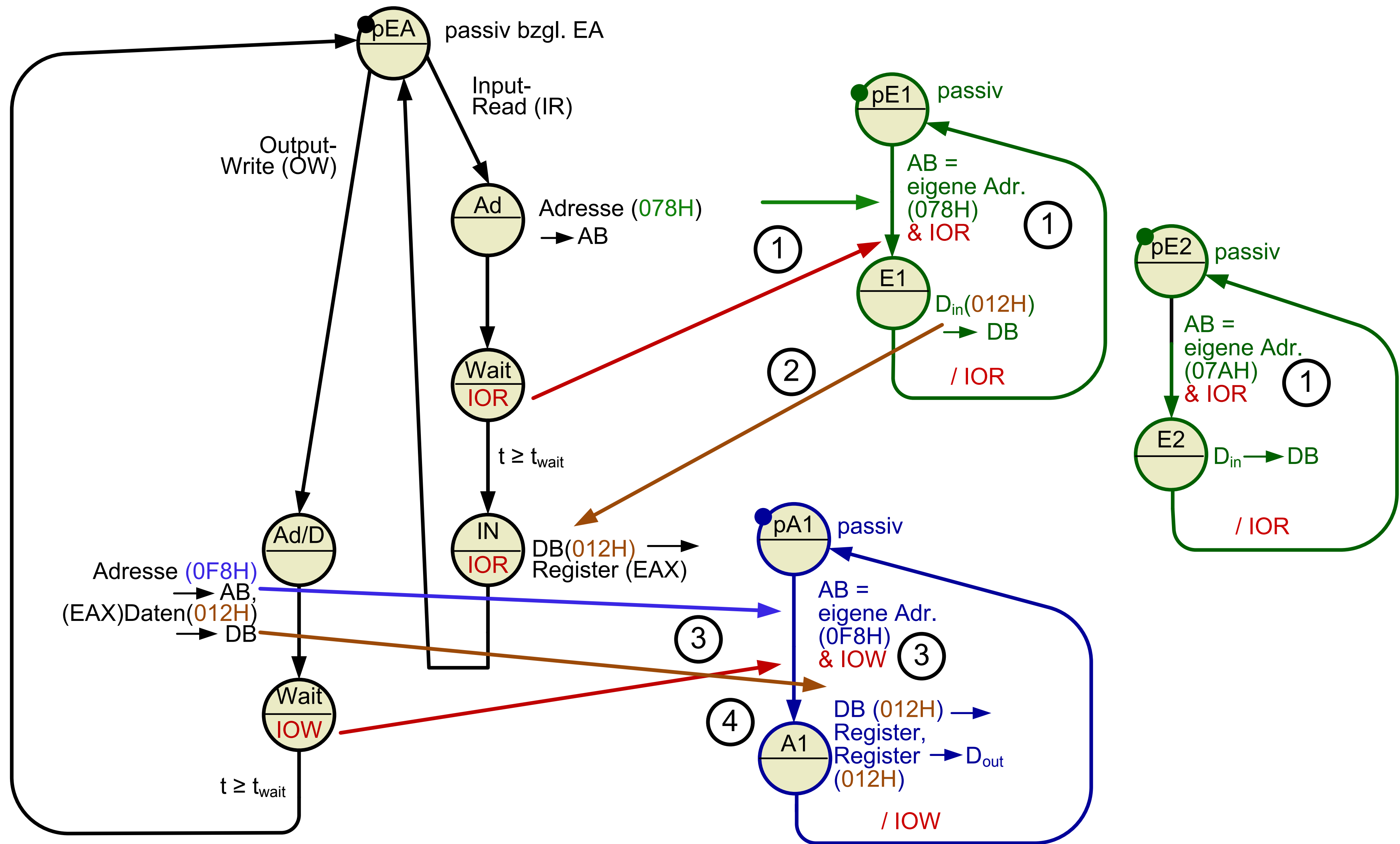


Bild 7.9: Zusammenwirken vom Prozessor mit mehreren EA-Schnittstellen

(1) Der Prozessor kommt im IN-Befehl in die Input-Read-Phase (IR, Bild 7.7 S.171), gibt die Adresse 078H auf den Adressbus und setzt das Signal IOR (Input Output Read) = 1. Die Adresse 078H führt bei der Eingabe-Schnittstelle mit dieser Adresse (oben, Mitte) zur Aktivierung, die Eingabe-Schnittstelle mit der Adresse 07AH (rechts) bleibt passiv, da eine andere Adresse auf dem AB liegt.
(2) Die aktivierte Eingabe-Schnittstelle (Adr. 078H) gibt die an ihrem D_{in} liegenden Daten (Bsp. 012H) über den DB zum Prozessor, der sie in das Register EAX speichert.
(3) Der Prozessor kommt im OUT-Befehl in die Output-Write-Phase (OW, Bild 7.4 S.168), gibt die Daten von EAX (Bsp. 012H, im direkt davor liegenden IN-Befehl von der Eingabe-Schnittstelle gelesen), auf den Datenbus, die Adresse 0F8H auf den Adressbus und setzt IOW (Input Output Write) = 1. Das führt zur Aktivierung der Ausgabeschnittstelle mit dieser Adresse.
(4) Die aktivierte Ausgabe-Schnittstelle (Adr. 0F8H) übernimmt die Daten (Bsp. 012H) vom DB und schaltet sie zur Umwelt über ihr D_{out}.

Die beiden Prozessorphasen (IR, OW) werden durch 0-Setzen des entsprechenden Signals IOR bzw. IOW beendet.

Die zeitliche Synchronisation zwischen Prozessor und EA-Schnittstellen geschieht durch die Wartezeit t_{wait} (synchrones Busverfahren, siehe auch dieses Prinzip in den Bildern 5.19 S.132, 5.21 S.134 für die entsprechenden Speicherzugriffe).

7.2. Programmierbare E/A mit Ein-/Ausgabe-Controller

Standardmäßig wickelt der Haupt-Prozessor, der in einfachen Rechnern der einzige Prozessor ist, die gesamten EA-Funktionen durch eigene Befehlsfolgen ab.

➔ Aufgrund dessen Belastung bei der programmtechnischen Realisierung seiner Hauptaufgaben und der Notwendigkeit von Echtzeit-Reaktionen des Hauptprozessors können Ein- und Ausgabefunktionen eventuell einem, zumeist weniger leistungsfähigeren, spezialisieren EA-Prozessor zugeordnet werden, der parallel zum Hauptprozessor arbeitet.

➔ Der Hauptprozessor erhält bzw. übergibt dann reine Daten ohne weitere Steuerfunktionen von und zur Umwelt.

Den Aufbau zeigt Bild 7.10 S.177.

Der Prozessor ist der Hauptprozessor für die übergeordnete Informationsverarbeitung mit direkt zugeordnetem Speicher.

Vom Systembus ist Daten-mäßig der EA-Controller erreichbar, z.B. über synchronisierte parallele digitale EA (siehe Abschnitt 7.4) oder Zwei-Tor-Speicher (siehe Bild 8.9 S.229).

Die konkrete Ablaufsteuerung der Ein- oder Ausgaben erfolgt mit speziellen Befehlsfolgen des EA-Controllers (als Firmware vorgefertigt bei Standard-EA-Funktionen oder anwenderprogrammierbar bei speziellen Funktionen).

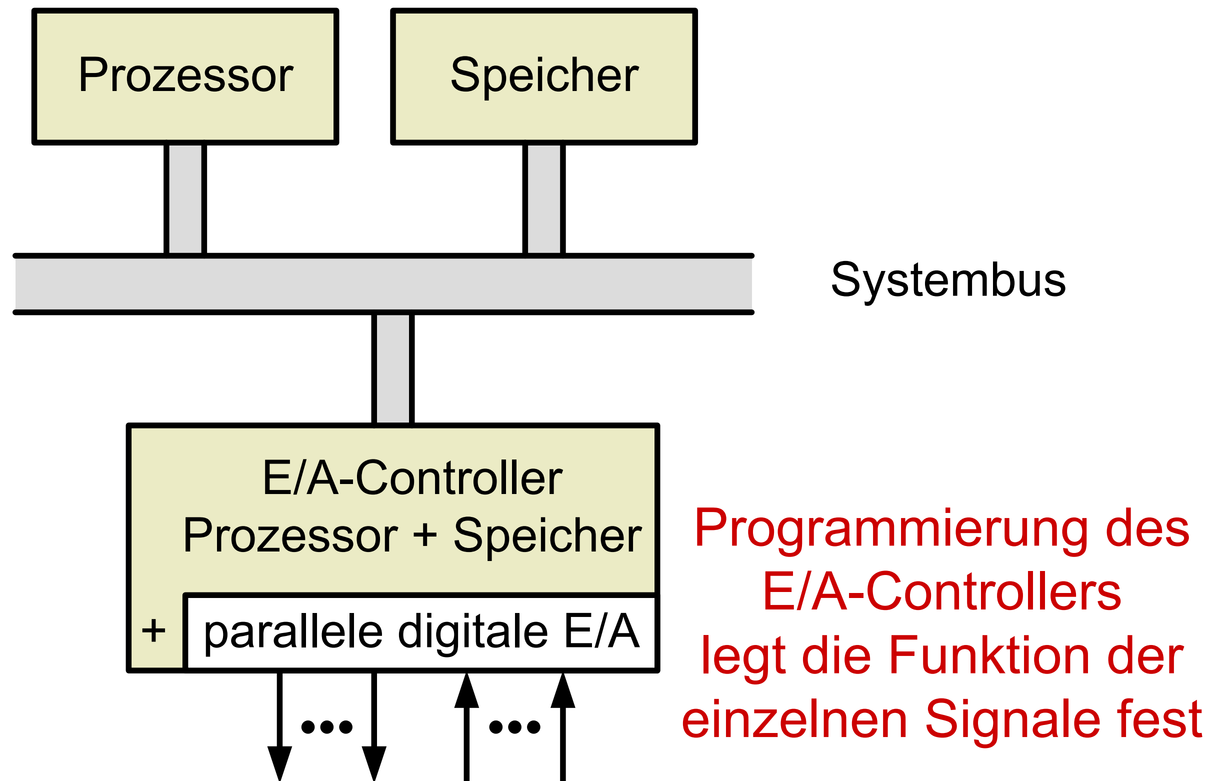


Bild 7.10: Systemeinordnung eines Ein-/Ausgabe-Controllers

Ein Beispiel für einen, allerdings leistungsfähigen, EA-Controller stellt ein Graphikcontroller mit einem Spezialprozessor für Graphikfunktionen (Graphic Processing Unit, GPU) (z.B. [9]) und mit einer Schnittstelle als Zwei-Tor-Speicher dar.

7.3. Programmierbare E/A mit Steuerregistern

Bei der programmierbaren Ein-/Ausgabe mit Steuerregistern besitzt die Schnittstelle Funktionen, die parametrisiert werden können oder Funktionen, die ausgewählt werden können oder Kombinationen von beiden.

Parametrisieren erfolgt durch Ausgabe der Parameter auf die den Funktionen zugeordneten Parameterregister (Steuerregister) mit OUT-Befehlen. Dabei werden die einzelnen Parameterregister mit unterschiedlichen EA-Adressen oder/und Sequenzen von Ausgaben angesprochen.

Die Auswahl von Funktionen erfolgt durch Setzen oder Rücksetzten von einzelnen Bit in Freigaberegistern (Steuerregistern), gleichfalls mit OUT-Befehlen (Prinzip siehe Bild 7.11 S.179).

Insgesamt gibt es dann als Übergabemöglichkeiten zwischen Prozessor und EA-Schnittstelle:

- Datenübergabe (Ein- bzw. Ausgabe über Datenregister),
- Ausgabe von Parametern über spezielle Steuerregister (Parameterübergabe),
- Ausgabe von Freigaben bzw. Nichtfreigaben von Betriebsarten in der EA-Schnittstelle.

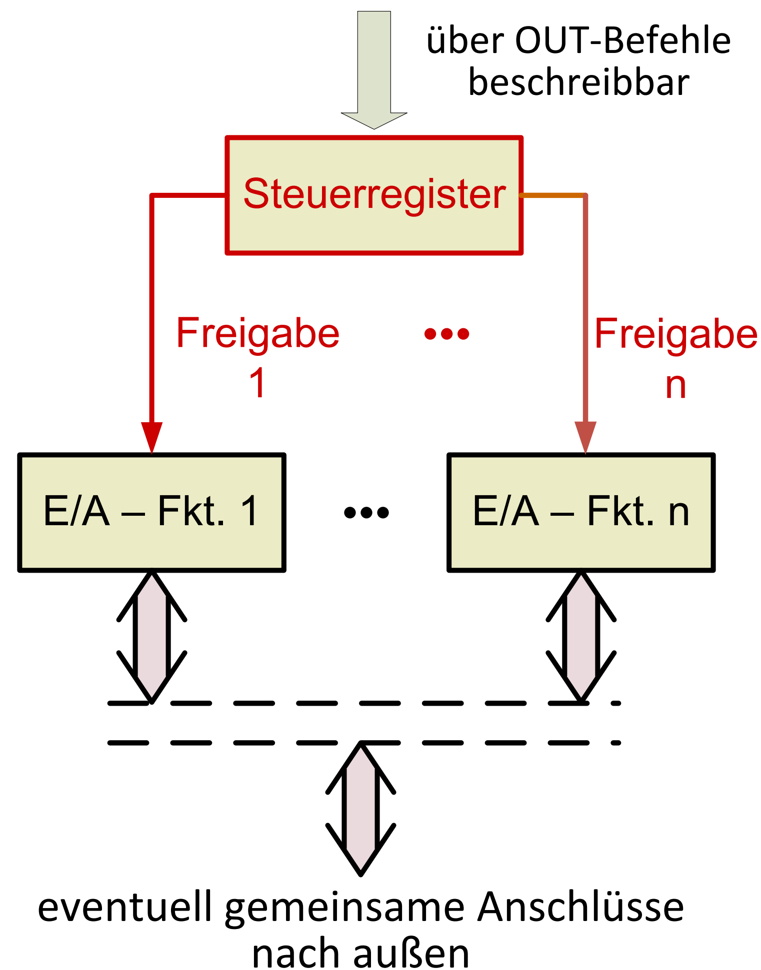


Bild 7.11 Funktionsauswahl durch Steuerregister

Bild 7.12 S.180 zeigt ein einfaches Beispiel zum Umschalten einer parallelen digitalen Ein-/Ausgabe-Schnittstelle durch ein Steuerbit, welches mit einem OUT-Befehl gesetzt oder rückgesetzt wird.

Es folgt ein Beispiel zur Benutzung:

Das Steuerbit ist Datenbit 0.

Die Adresse des Steuerbit (1-Bit-Steuerregister) ist 0FFH.

Die Adresse der Schnittstelle bzgl. Ein- und Ausgabe ist 0FEH. Die Unterscheidung von Ein- bzw. Ausgabe erfolgt durch die Verwendung eines IN- bzw. OUT-Befehls mit dieser Adresse. Vor der Verwendung der Schnittstelle muss das Steuerbit gesetzt (=1, Ausgabe) bzw. rückgesetzt (=0, Eingabe) werden.

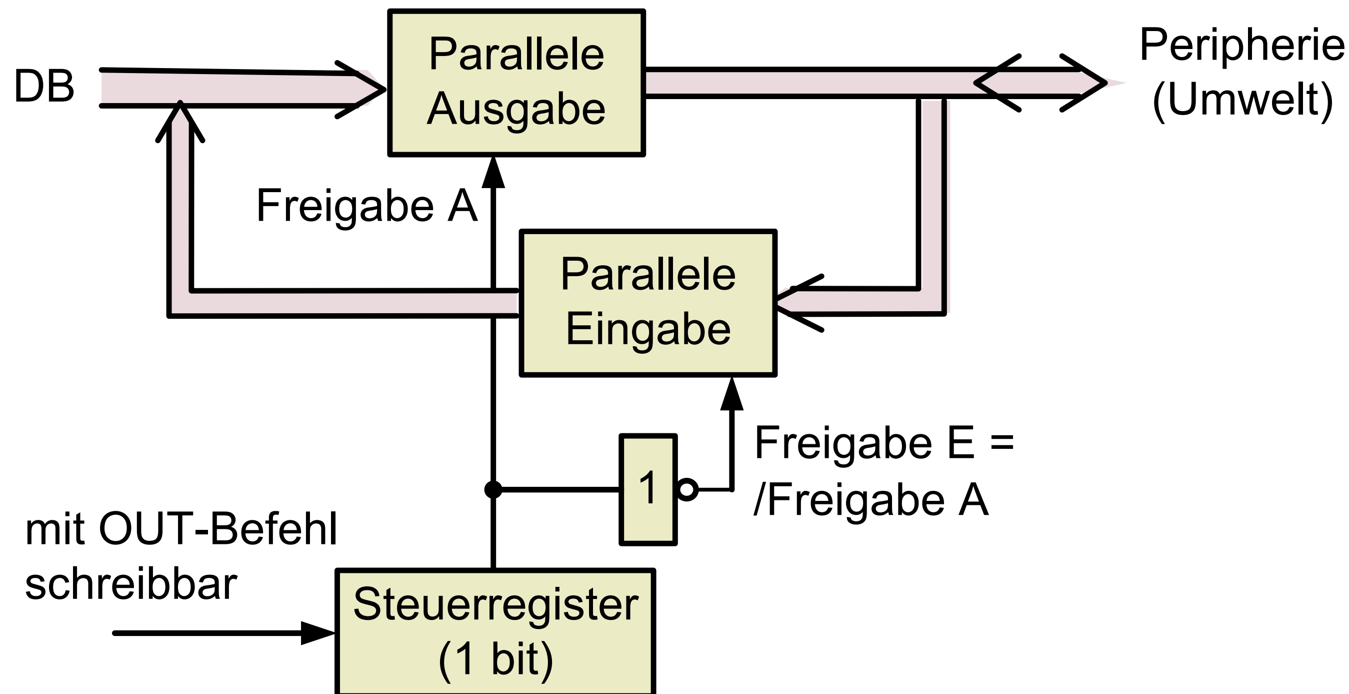


Bild 7.12 Programmierbare digitale parallele Ein-/Ausgabe

Eine beispielhafte Befehlsfolge:

```
OUT   0FFH, 01H    ; Schnittstelle auf Ausgabe schalten (Adr. 0FFH ist die
                   ; Steueradresse der Schnittstelle)
OUT   0FEH, EAX    ; EAX-Inhalt (Daten) nach Ausgabe
                   ; (0FEH) ist die Datenadresse der Schnittstelle
…                  ; weitere Programmteile
OUT   0FFH, 00H    ; Schnittstelle auf Eingabe schalten
IN    EAX, 0FEH    ; Eingabe der Schnittstellendaten nach EAX
…                  ; weitere Programmteile
```

7.4. Synchronisierte parallele digitale Ein-/Ausgabe

Bisher, d.h. bei dem Prinzip der parallelen digitalen EA nach Abschnitt 7.1, war der Zeitpunkt des Eingabe- oder Ausgabebefehls (und damit des Datentransfers von und zur Umwelt) nur vom Befehlsablauf im Prozessor abhängig.

Sinnvoll ist in viele Fällen eine zeitliche Synchronisation von ausgebender und eingebender Seite beim Datenaustausch (z.B. ein Datenblock aus n Byte wird übertragen, jedes Byte aber nur einmal, und möglichst zeitminimal).

➔ Notwendig ist das Hinzufügen von zwei Steuersignalen:

RDY (Ready for Data): Ausgang der Eingabeseite zu Signalisierung, dass sie neue Daten angeboten bekommen kann.

DAV (Data Valid oder Data Available): Ausgang der Ausgabeseite zur Signalisierung, dass neue gültige Daten angeboten werden.

Die Mitteilung der Sachverhalte „neue Daten vorhanden" (IRQ_E) bzw. „neue Daten können ausgegeben werden" (IRQ_A) zu den Prozessoren werden durch Interrupt signalisiert.

Aufgrund der zwei beteiligten Steuersignale und deren wechselseitiger Wirkung wird das Verfahren als „Zwei-Draht-Handshake" bezeichnet. Das in Abschnitt 5.5 beschriebene „Asynchrone Busverfahren" (Bild 5.22 S.136) entspricht in etwa dem hier für die Ein-/Ausgabe beschriebenen Synchronisierungsprinzip.

Zur Erläuterung dient eine Prozessorkopplung (Eingabe und Ausgabe sind in einem separaten Prozessorsystem). Es ist auch möglich, dass eine der beiden Seiten eine Umwelteinrichtung ist.

Eine Beispielanwendung wäre ein Prozessorsystem mit Bus Ausgabeseite (Hauptprozessorsystem) und Prozessorsystem mit Bus Eingabeseite (EA-Controller). Es wäre eine Einordnung in Abschnitt 7.2., Bild 7.10 S.177 möglich. Dabei ist auch zusätzlich die Gegenrichtung sinnvoll.

Die Struktur der synchronisierten EA zeigt Bild 7.13 S.183.

Die parallele Ausgabe enthält entsprechend Abschnitt 7.1. Registerausgänge (Ausgabe). Sie werden mit den Treibereingängen (Eingabe) parallel verbunden (Daten).

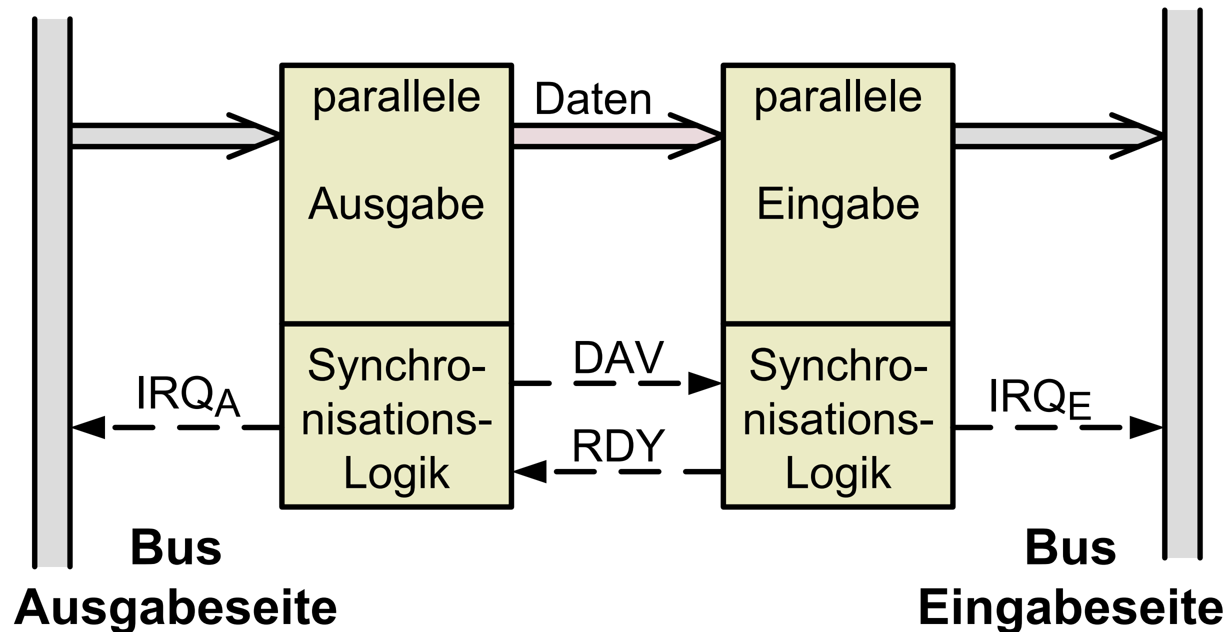


Bild 7.13: Struktur der synchronisierten parallelen digitalen Ein-/Ausgabe

➔ Sowohl die Ausgabeseite als auch die Eingabeseite besitzen eine Steuerlogik, die untereinander über die beiden Steuersignale DAV und RDY und zu den Prozessoren mit den IRQ (A bzw. E) –Signalen verbunden sind. Die Funktion dieser Steuerlogiken zeigt Bild 7.14 (a) S.184.

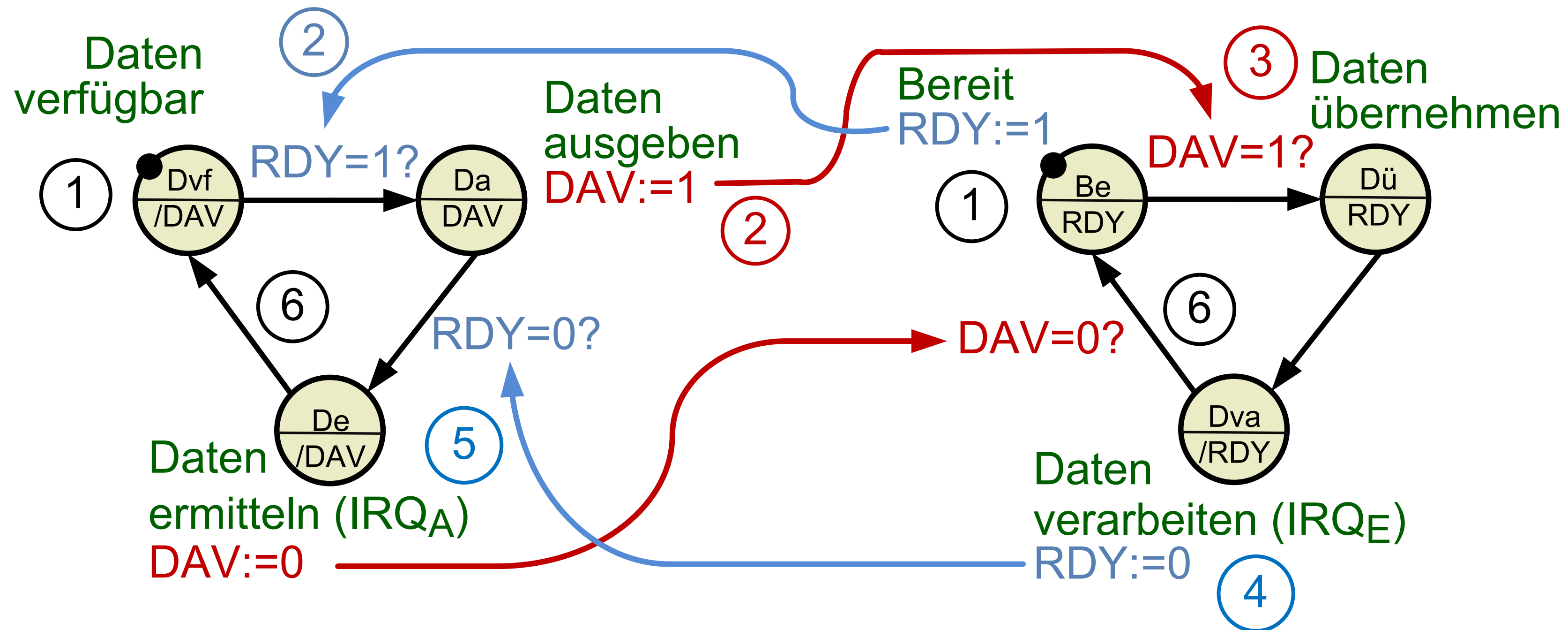


Bild 7.14 (a) Zwei-Draht-Handshake als parallele Automaten

Es entsteht als Ablauf (Zusammenwirken der beiden Automaten als Zeitverlauf, Bild 7.14 (b) S.185:

(1) initial: A (Ausgabe): Daten A verfügbar, DAV = 0; E (Eingabe): Bereit zur Datenübernahme, RDY = 1, Daten E passiv.

(2) A: RDY = 1 -> Daten ausgeben und DAV := 1,

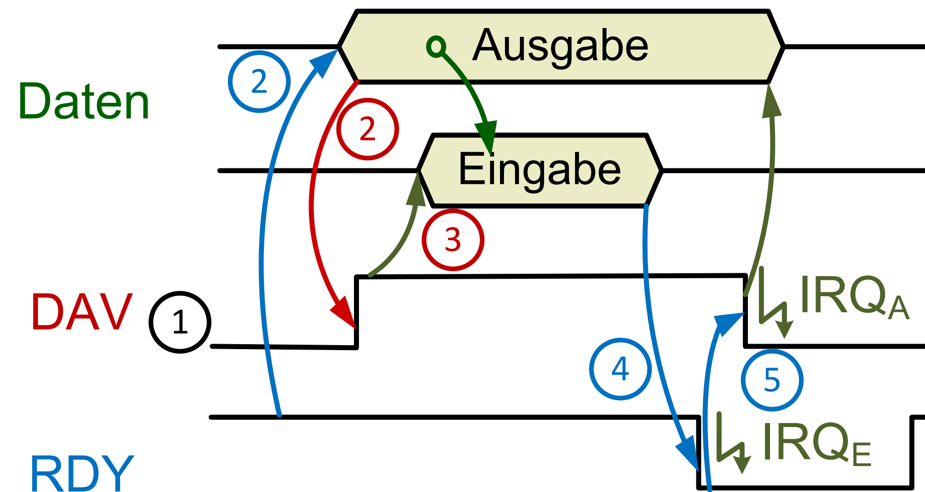


Bild 7.14 (b) Zeitverläufe zum Zwei-Draht-Handshake

(3) E: DAV = 1 -> Daten übernehmen,

(4) E: nach Datenübernahme -> RDY : = 0; IRQ_E -> Start Interruptprogramm „Übernommene Daten verarbeiten“,

(5) A: RDY = 0 -> DAV := 0; IRQ_A -> Start Interruptprogramm “Neue Daten ermitteln“),

(6) beide Interruptprogramme laufen parallel und erzeugen nach ihrem Ende jeweils den initialen Zustand nach (1).

➔ Danach ist die Wiederholung möglich (beliebig oft, solange Daten übertragen werden sollen).

➔ Das langsamere Interruptprogramm bestimmt das Zeitverhalten.

7.5. Serielle digitale Ein-/Ausgabe

Ausgangspunkt ist die serielle Datendarstellung (Bild 7.15 S.187).

Im Vergleich zur parallelen Darstellung nach Bild 7.2 S.165 (in Bild 7.15 links S.187), bei der n Binärsignale mit einem Zeitpunkt je Datenwort entstehen, ist es bei seriell (rechts):

ein Binärsignal, n Zeitpunkte je Datenwort.

Das bedeutet, die örtliche Zuordnung der Einzelsignale wird zur zeitlichen Zuordnung.

Vom Prinzip her ist dabei parallel n mal schneller bei n mal höherem Aufwand gegenüber seriell.

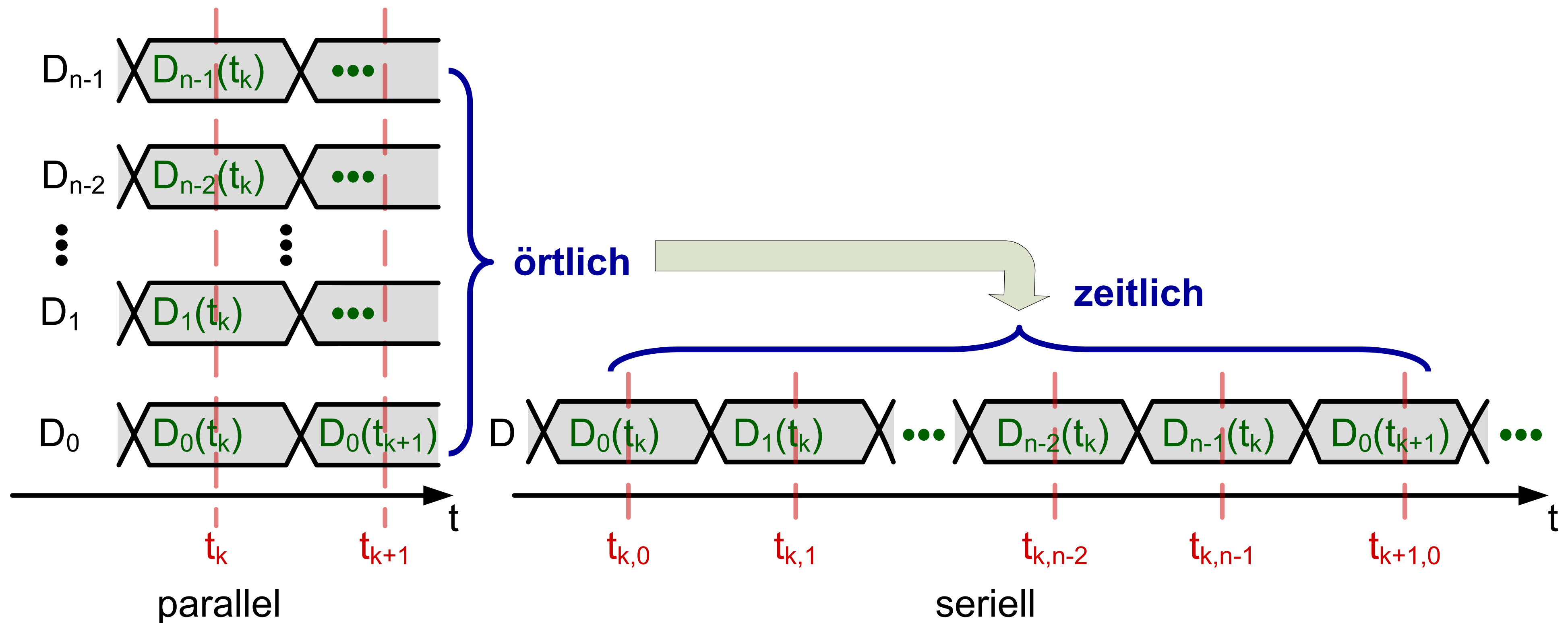


Bild 7.15: Parallele und serielle Datendarstellung

Als typische Anwendungsmöglichkeiten gilt:

parallel für kurze Entfernungen und möglichst maximale Übertragungsgeschwindigkeit (rechnerintern),

seriell für größere Entfernungen und mittlere Übertragungsgeschwindigkeiten (rechnerextern).

Für die Übertragung sind notwendig:

- Ausgabe mit Wandlung parallel nach seriell und
- Eingabe mit Wandlung seriell nach parallel.

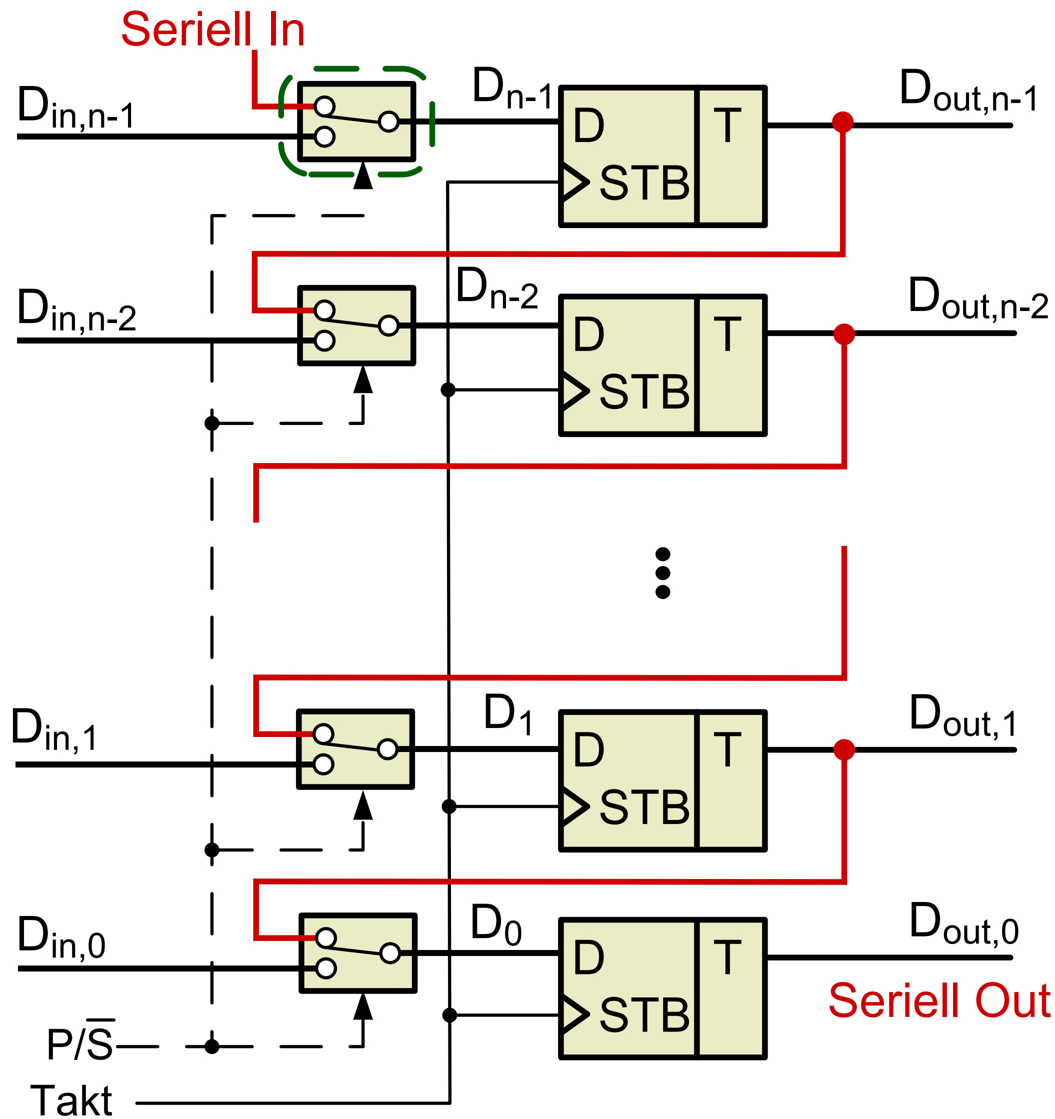
Das Grundelement für beide Wandlungen ist ein Schieberegister (siehe [4], [3], [1]). Das Prinzip zeigt Bild 7.16 (a) S.189.

Schwarz sind die Elemente für parallele Arbeitsweise. Dabei sind die Umschalter für alle Bit in der unteren Stellung. Es ist das Signal Parallel-nicht-Seriell P/S = 1. Die Schalter schalten $D_{in,i}$ auf D_i vom Flip-Flop i. Mit dem Takt (verbunden mit den STB-Eingängen aller Flip-Flops) erfolgt das Einspeichern parallel aller Di in die getakteten D-Flip-Flops.

Mit Parallel-nicht-Seriell P/S = 0: Schalten D_{n-1} = Seriell In; sonst $D_i = D_{out,(i+1)}$ (rote Verbindungen, Schalter entsprechend auf oberer Position); Seriell Out = $D_{out,0.}$ Mit dem Takt werden die Bit, beginnend mit Seriell In, hereingeschoben, bzw. über Seriell Out herausgeschoben.

Rechts im Bild gezeigt ist die logische Realisierung der Schalter.

Bild 7.16 (b) S. 190 enthält zur Erläuterung der Funktion die untersten zwei Bit des Schieberegisters als parallele Automaten.



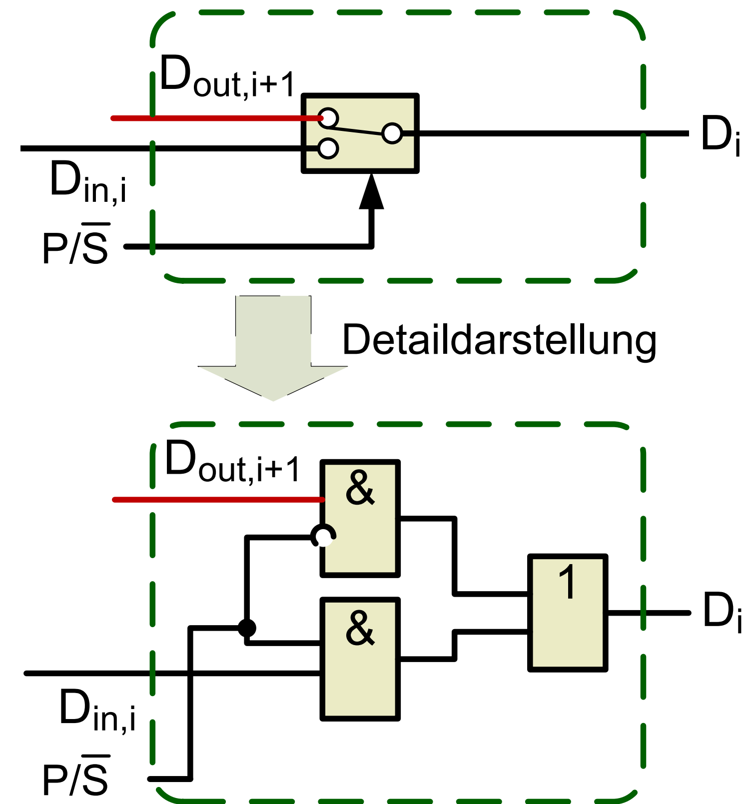


Bild 7.16 (a): Prinzip eines Schieberegisters

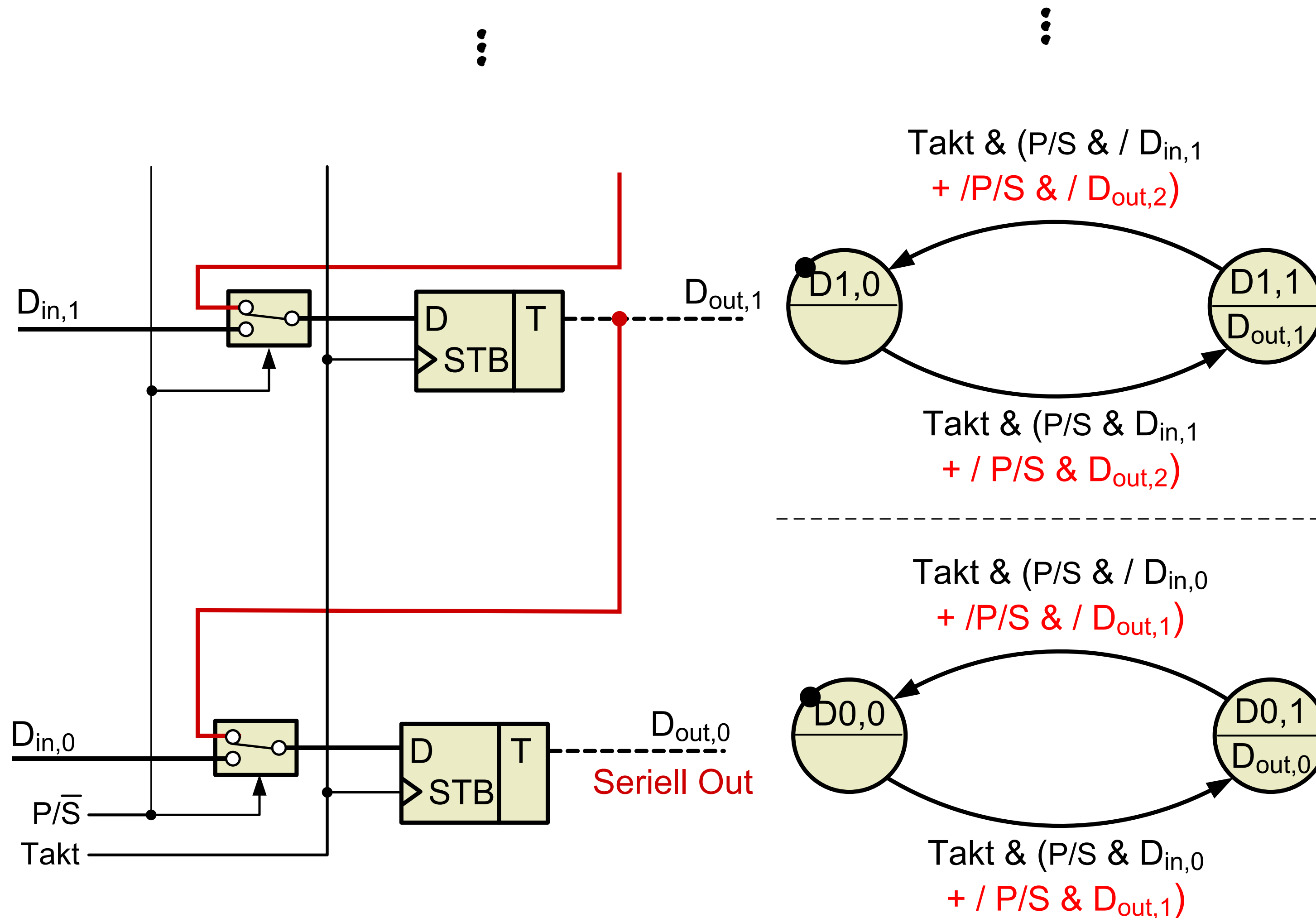


Bild 7.16 (b): Prinzip eines Schieberegisters als Automaten

Um den Abstand der Zeitpunkte $t_{k,i}$ aus der seriellen Darstellung in Bild 7.15 S.187 mit konstantem Abstand einstellen zu können, wird für die Ausgabeseite und die Eingabeseite ein Takt (mit den Taktperioden Ausgabe (TPA) und Eingabe (TPE)) notwendig. Dabei muss

- TPE etwa gleich TPA (asynchrones Verfahren) oder
- TPE = TPA (synchrones Verfahren)

sein.

Zur zeitlichen Synchronisation ist zusätzlich eine Erweiterung der seriellen Daten um serielle Steuerinformationen notwendig

Variante asynchron: Die Einzelbit werden nicht synchronisiert. Aufgrund der Tatsache, dass die Taktperioden bei Ausgabe und Eingabe leicht differieren können, verschiebt sich der Auswertungszeitpunkt der Eingabe im Gültigkeitsbereich der Ausgabebit.

Deshalb ist die Anzahl der Datenbit sehr begrenzt, zumeist auf 8 (1 Byte).

Bild 7.17 S.192 zeigt ein typisches asynchron zu übertragendes Datenwort mit den entsprechenden zugeordneten Taktperioden, Gültigkeitsbereichen, Umschalt- und Auswertungszeitpunkten. Die einzelnen übertragenen Bit haben nachstehende Bedeutung:

- Startbit: kennzeichnet den Beginn eines Datenworts zeitlich,
- Stoppbit (auch Endebit): kennzeichnet das Ende des Datenworts (muss den negierten Wert des Startbit haben),
- Prüfbit: Da die Anwendung von seriell meist rechnerextern erfolgt, sind Störungen der Datenbit wahrscheinlich. Deshalb wird optional ein redundantes Bit ergänzt, z.B. bei gerader Parität, Ergänzung auf gerade Anzahl von 1. Damit ist eine Erkennung von Ein-Bit-Fehlern möglich.

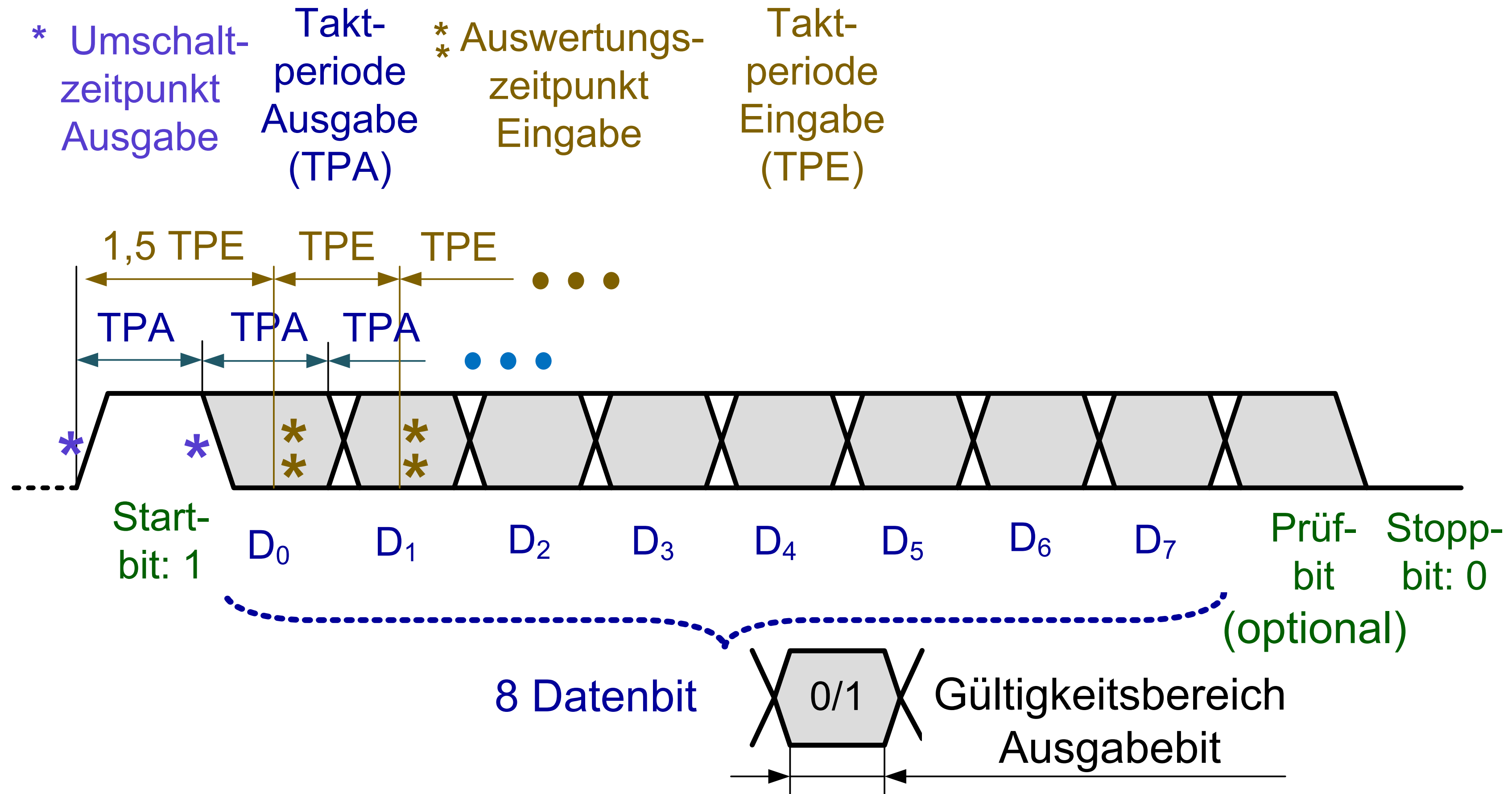


Bild 7.17: Datenformat und Zeitangaben zur asynchronen seriellen Ein- und Ausgabe

- 8 (hier als Bsp. für kleines n) Datenbit.
- Erkennung des Werts von Datenbit und Prüfbit bei der Eingabeseite: 1,5 Bit-Zeiten nach der Startbit-0-1-Flanke D_0, 2,5 Bit-Zeiten nach der Startbit-0-1-Flanke D_1 usw.

Ausgabe und Eingabe arbeiten mit ungefähr den gleichen Bit-Zeiten (TPA, TPE). Das ist notwendig wegen der Erkennung der Bit bei der Eingabeseite. Dazu benutzen Ausgabe und Eingabe für ihre Schieberegister Taktgeneratoren mit etwa der gleichen Frequenz. Genau gleich ist hier nicht möglich, da aufgrund von Bauelementetoleranzen (elektronisch) die Frequenzen immer etwas voneinander abweichen (zwei Taktgeneratoren).

Die prinzipielle Struktur einer seriellen Ausgabe/ Eingabe, im Beispiel asynchron, Kopplung von zwei Prozessorsystemen enthält Bild 7.18 S.194. Auf das optionale Prüfbit wurde verzichtet. Es kann auch jede Seite durch eine periphere Einrichtung ersetzt werden.

Dabei haben die einzelnen Blöcke folgende Bedeutung:

- Die parallele Ausgabe (synchronisierte parallele Ausgabe), gibt das Datenwort an die parallelen Eingänge des Schieberegisters A (Ausgabe).
- Zusätzlich werden als erstes Bit das Startbit („1“) und als letztes Bit das Stoppbit („0“) mit parallel übernommen (P/S = 1).
- Alle Bit werden mit dem Takt des Taktgerators A (Ausgabe, Periode TPA) aus dem Schieberegister herausgeschoben (P/S = 0).

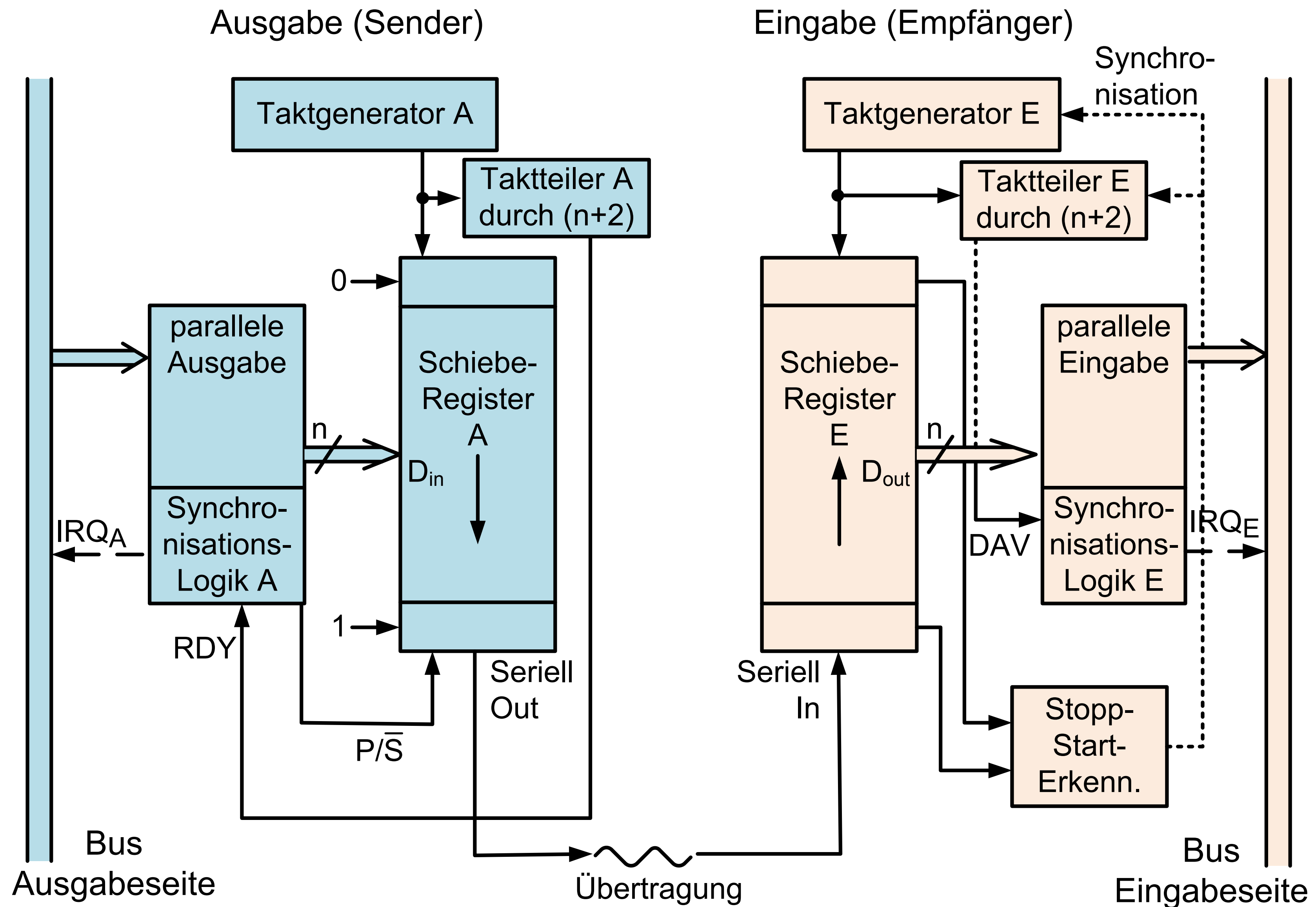


Bild 7.18: Prinzipstruktur zur asynchronen seriellen Datenaus- und Eingabe

- Nach Herausschieben von n+2 Bit (Start-, Daten- und Stoppbit) wird durch den Taktteiler A RDY für die Synchronisationslogik erzeugt, die dem Prozessor der Ausgabeseite über IRQ_A signalisiert, dass er das nächste Datenwort ausgeben kann.
- Nach der Übertragung (evtl. elektronisch verstärkt bzw. gewandelt, ohne Veränderung von Logikpegel und Zeitbezug) werden die einzelnen Bit in das Schiebregister E (Eingabe) hereingeschoben. Das erfolgt mit dem Takt des Taktgenerators E (Eingabe, Periode TPE), wobei dessen Synchronisation auf die Stopp-Startbitflanke erfolgt (Stopp-Start-Erkennung). Dabei startet die Erkennung der Bit anschließend mit 1,5 TPE, danach mit jedem weiteren TPE.
- Nach Hereinschieben von n+2 Bit (Start-, Daten- und Stoppbit) wird durch den Taktteiler E DAV für die Synchronisationslogik E erzeugt, die bei der parallelen Eingabe (synchronisierte parallele Eingabe) bewirkt, dass die parallelen Ausgänge des Schieberegisters E in dessen Eingaberegister (dieses ist einen Erweiterung gegenüber Bild 7.6 S.170 und Bild 7.13 S.183) übernommen werden.
- DAV für die Synchronisationslogik E erzeugt auch IRQ_E, welches dem Prozessor der Eingabeseite signalisiert, dass er das nächste Datenwort übernehmen kann.

Die Ausgabeseite wird auch mit Sender und die Eingabeseite mit Empfänger bezeichnet.

Eine vollständige bidirektionale Verbindung von zwei Systemen über serielle Kopplung verlangt auf jeder Seite einen Sender und einen Empfänger mit der jeweiligen Sender-Empfänger-Übertragung.

Bei synchroner serieller Datenaus- und Eingabe gilt, wie oben bereits geschrieben, TPE=TPA. Das wird dadurch erzeugt, dass nur der Sender einen Taktgenerator besitzt und die Taktinformation mit übertragen wird. Das kann als zweites Signal oder geeignet seriell im oder mit dem Datensignal im selben Signalweg erfolgen.

Dadurch können beim synchronen Prinzip theoretisch beliebig lange Datenstrukturen übertragen werden. Ein Beispiel für eine derartige Datenstruktur zeigt Bild 7.19 S.196.

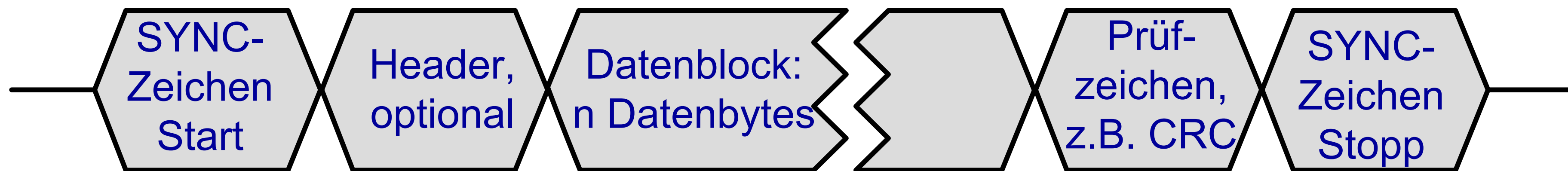


Bild 7.19: Datenformat zur synchronen seriellen Ein- und Ausgabe

Die einzelnen Zeichen, die aus mehreren Bit bestehen (z.B. 8, 16), haben als Bedeutung:

- Sync-Zeichen: Erkennung von Blockanfang (Start) und Ende (Stopp),
- Header: Zusatzinformationen zum Datenblock, optional:

- Sender- und Empfängeradresse (Identifier),
- Blocklänge, sie steuert die Anzahl der folgenden Datenbyte,
- laufende Blocknummer (bei Übertragung von mehreren Datenblöcken,
- u.ä.,

- Datenblock: Nutzinformation mit Länge n Zeichen (z.B. Zeichen 1 Byte). Dabei kann n fest oder im Header angegeben variabel sein,
- Prüfzeichen, z.B. als CRC (Cyclic Redundancy Code): komplexere Datensicherung (als Parität).

Zwei Beispiele zur Taktübertragung zusammen mit den Daten (Takt-Daten-Modulation, TDM) zeigt Bild 7.20 S.198.

Bei dem Prinzip Return-to-Zero-Code (RZ, Übertragungsdaten braun) wird innerhalb einer Taktperiode der Ausgabe (TPA) zu Beginn (Zeit t_i) immer ein Taktimpuls (als logische 1) mit Länge 0,25 TPA übertragen. Ist das Datenbit = 1, folgt ein weiterer Impuls (logisch 1) mit Zeitpunkt t_i+0,5 TPA und Länge 0,25 TPA. Ist das Datenbit 0, folgt bis zum nächsten Taktimpuls zum Zeitpunkt t_{i+1} kein Impuls.

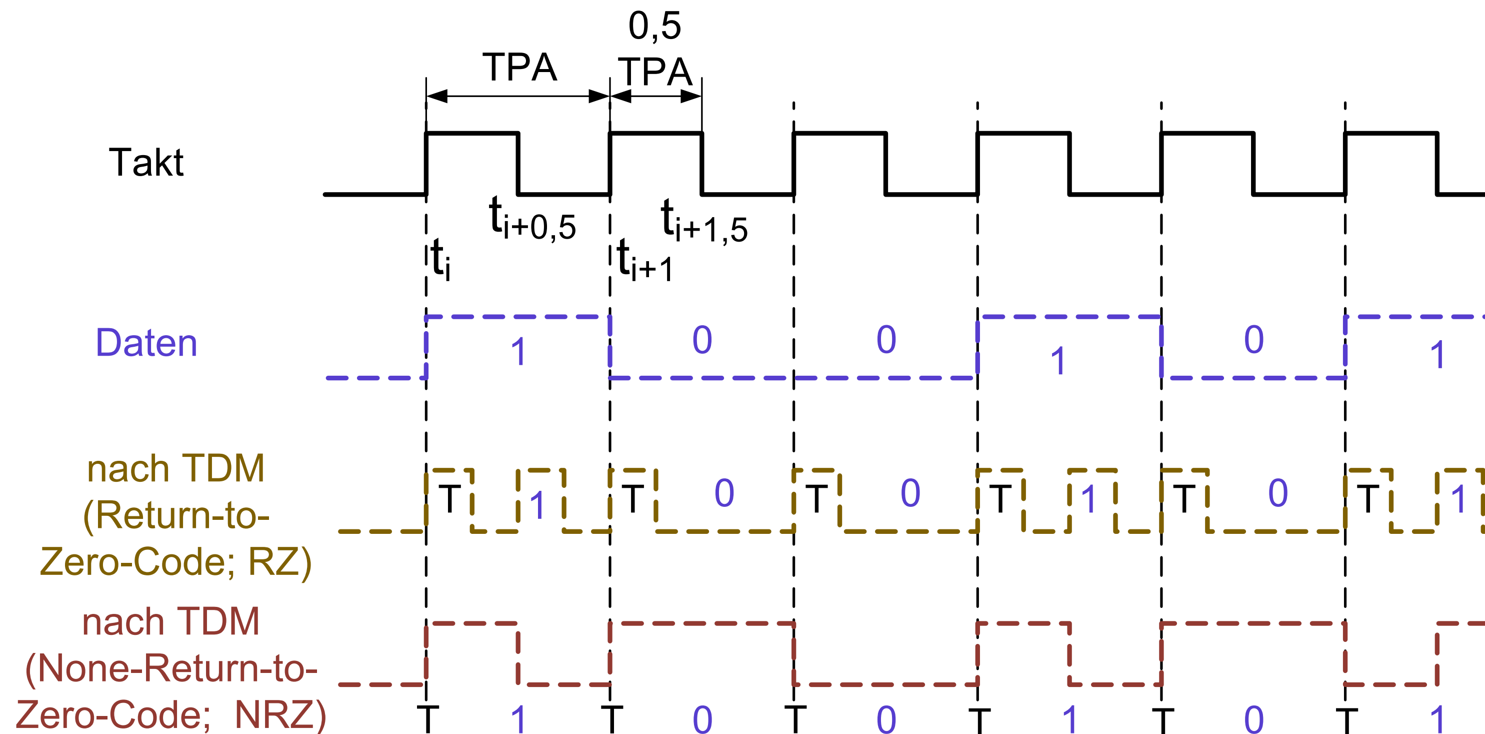


Bild 7.20: Zwei Varianten zur gemeinsamen Übertragung von Takt und Daten

Bei dem Prinzip None-Return-to-Zero-Code (NRZ, Übertragungsdaten rot) wird zu Beginn einer Taktperiode der Ausgabe (TPA) (Zeit t_i) immer ein Pegelwechsel (0-1 oder 1-0, je nachdem, welcher Pegel vorher war) übertragen. Ist das Datenbit = 1, folgt ein weiterer Pegelwechsel (0-1 oder 1-0, je nachdem, welcher Pegel vorher war) zum Zeitpunkt t_i+0,5 TPA. Ist das Datenbit 0, folgt bis zum nächsten Taktimpuls zum Zeitpunkt t_{i+1} kein Pegelwechsel.

7.6. Zähler-Zeitgeber

Bei Zähler-Zeitgeber-Funktionseinheiten werden die Datenwerte durch Impulsfolgen (0-1-0-Wechsel) über die Zeit dargestellt.

Typische Möglichkeiten sind:

Zähler: Eingabe von Impuls- (0-1-0) Folgen über einen Zählimpulseingang (zie) in einer vorgegeben Zeit T (Bild 7.21 S.199). Dabei kann sich diese Zeit periodisch, zumeist konstant, wiederholen. Der Datenwert ist die Anzahl der Impulse innerhalb der Zeit T.

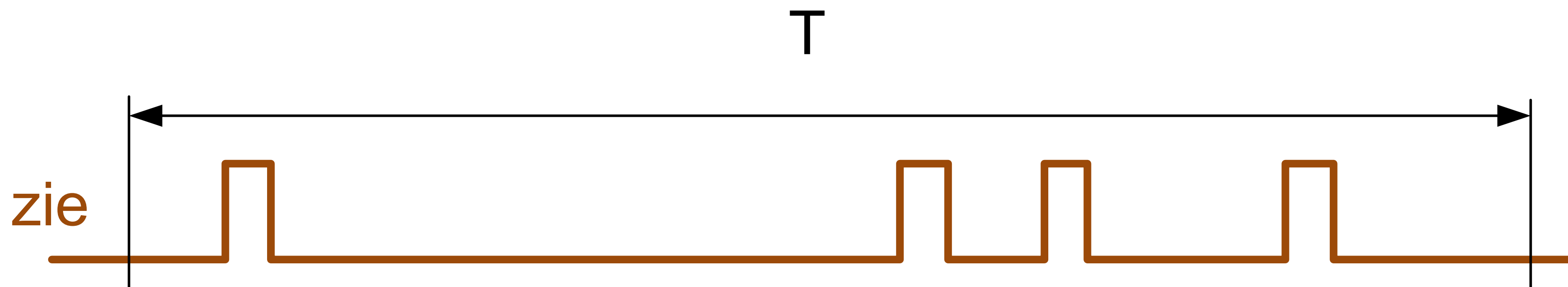


Bild 7.21: Beispielimpulsfolge zum Zählen in einer Zeit T

Impulsausgang: Impuls- (0-1-0) Folge mit einer programmtechnisch beeinflussbaren Frequenz $f_p = 1/T_p$ (Bild 7.22 S.200). Der Datenwert ist dabei f_p. Von den Impulsen können zeitzyklische Interrupts als Zeitbasis für den Prozessor abgeleitet werden.

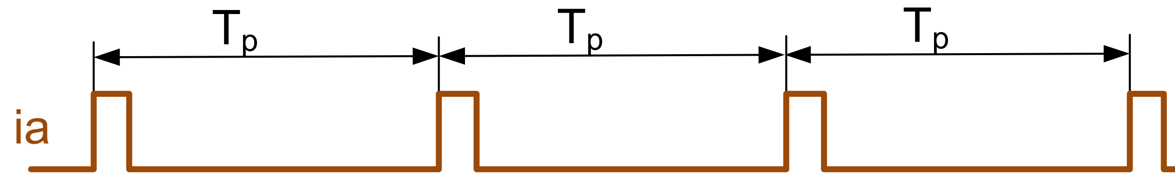


Bild 7.22: Impulsausgabe mit einer konstanten Zeitperiode T_p

Basis von Zähler-Zeitgeberfunktionen sind Binärzähler [3], [1]. Das sind sequentielle Logikschaltungen, die mit jedem Eingangsimpuls (Wechsel 0-1-0) ihren binären Wert (Zählerstand) inkrementieren (Vorwärtszähler) bzw. dekrementieren (Rückwärtszähler).

Bei Erreichen ihres Endwerts (Vorwärtszähler alle Stellen = 1, Rückwärtszähler alle Stellen = 0) beginnen sie wieder vom Anfangswert.

Ein Modell zu einem Beispiel-Binärzähler als System synchroner paralleler Automaten (d.h. Zustandsübergänge mit der nächsten steigenden Taktflanke, siehe Bilder 2.2 S.18 und 2.3 S.20) zeigt Bild 7.23 S.201.

Er beinhaltet 4 Binärstellen. Je Stelle existiert ein sequentieller Teilautomat.

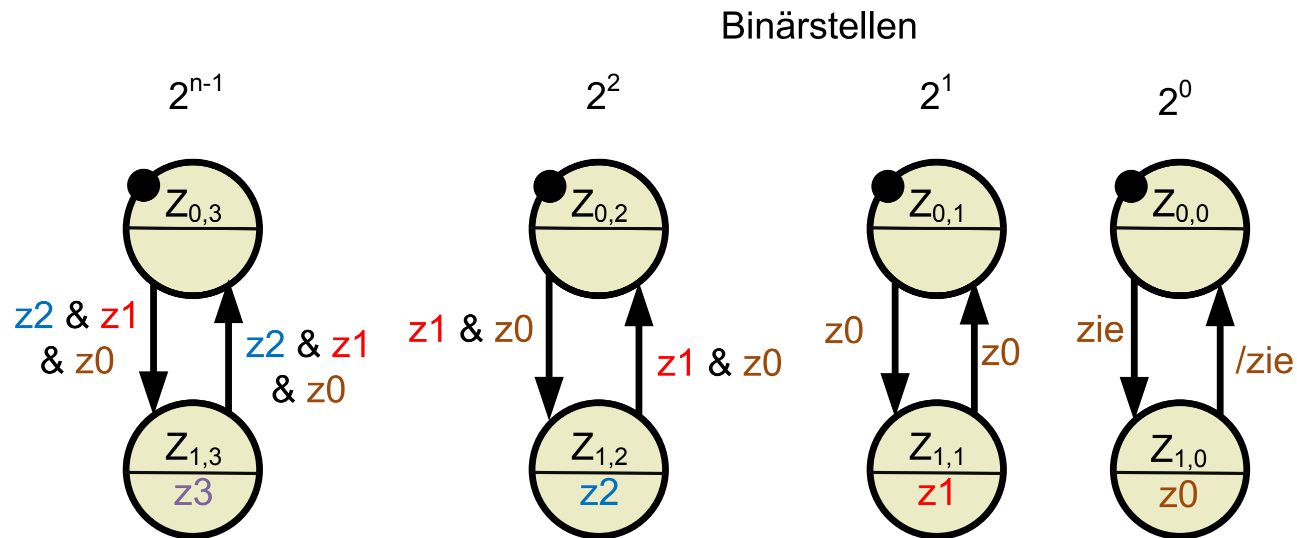


Bild 7.23: Automatenmodell eines Binärzählers

Die einzelnen Stellen sind über die Statussignale z2 bis z0 geeignet gekoppelt. zie ist der Eingangszählimpuls. Bild 7.24 S.202 zeigt den Verlauf der Logiksignale und der Zählerstände für eine zie-0-1-Folge mit konstanter Periode (Auszug: z3 bis z0). Durch Negation der Ausgänge entsteht ein Rückwärtszähler.

Für die hier vorgesehene Funktion ist es notwendig, dass der Anfangswert des Zählers voreinstellbar ist, d. h., dass ein Rückwärtszähler mit einem Startwert kleiner dem Maximalwert beginnen kann und diesen nach Zählernullwert erneut als Anfangswert benutzt. Das kann ähnlich wie beim Schieberegister durch paralleles Einspeichern (siehe Bilder 7.16 (a) S.189 und 7.16 (b) S.190) erfolgen. Unter Verwendung dieser Variante eines binären Rückwärtszählers lässt sich eine Zähler-Zeitgeber-Funktionseinheit aufbauen,

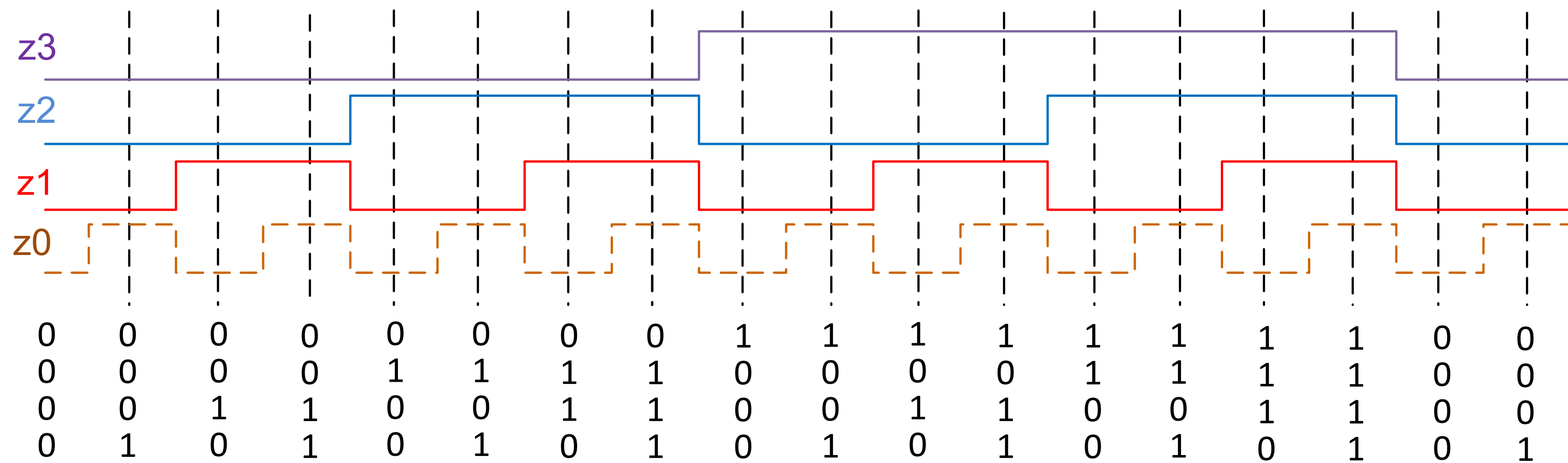


Bild 7.24: Verlauf von Logiksignalen und Zählerständen des Binärzählers von Bild 7.23 S.201

wie sie in Bild 7.25 S.203 gezeigt wird. Dabei haben die einzelnen Blöcke als Bedeutung:

Rückwärtszähler: Binärzähler, zählt die über seinen Zähleingang ankommenden Impulse, beginnend mit einem über die parallele Ausgabe programtechnisch vorgegeben Startwert (auch: „Zählkonstante“) in Richtung Null. Das wiederholt sich nach Erreichen des Zählerstands Null (Nulldurchgang). Der aktuelle Zählerstand lässt sich über die parallele Eingabe einlesen.

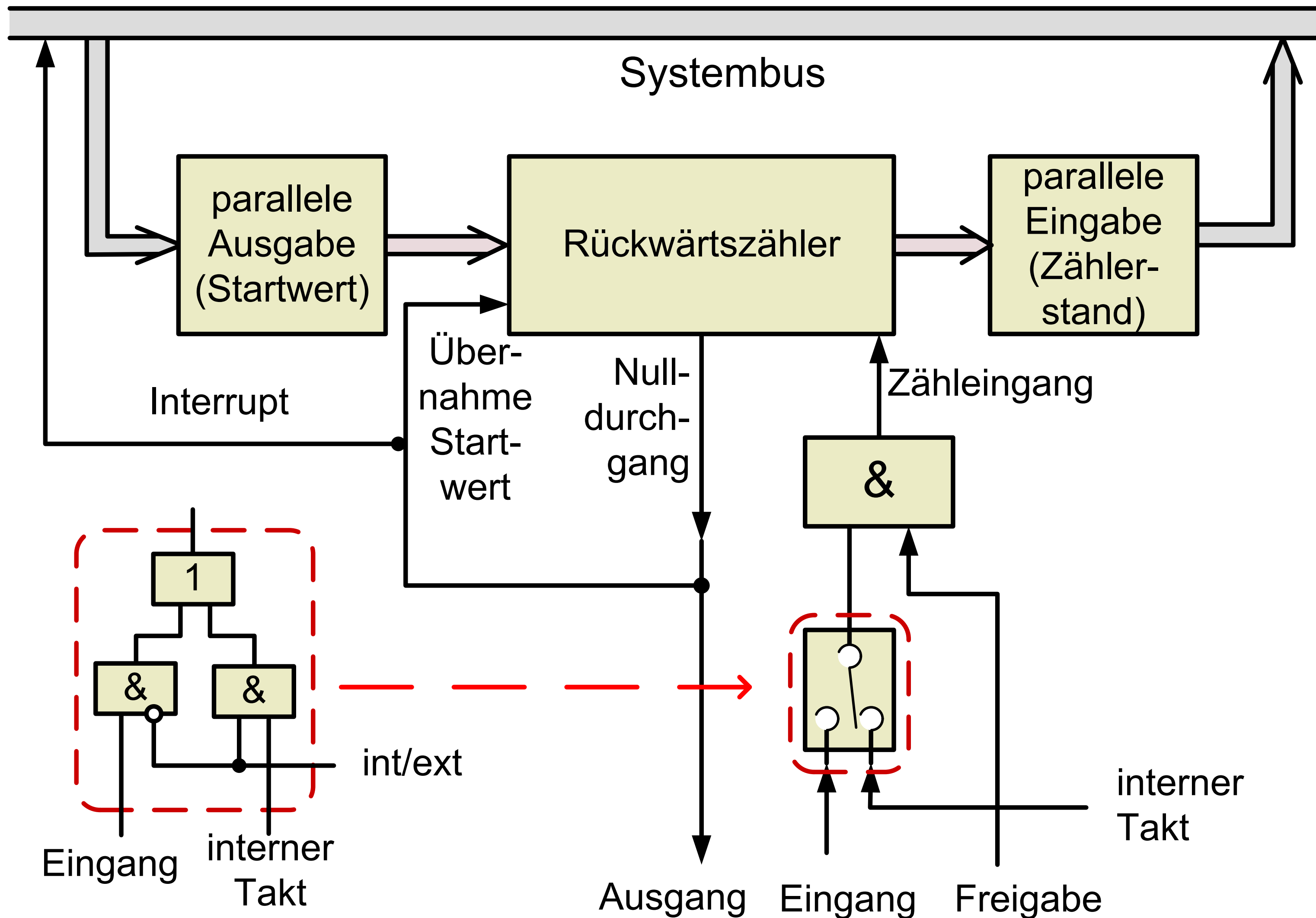


Bild 7.25: Zähler-Zeitgeber-Funktionseinheit

Bei der Betriebsart „Zähler von Impulsfolgen“, Impulse wie in Bild 7.21 S.199, kommen die Impulse von außen (Eingang -> Zähleingang, Schalterstellung durch int/ext = 0 in der

detaillierten logischen Darstellung des Umschalters). Die Zählerstandsänderung zwischen zwei Eingaben des Zählerstandes (Zeit T) über die parallele Eingabe ist der Datenwert.

Bei der Betriebsart „Impulsausgabe“ (Zeitgeber, Timer) kommen die Impulse von einem festen internen Takt, die Ausgabefrequenz des Nulldurchgangs ist gleich 1/(Startwert * TiT) mit TiT ist die Taktperiode des internen Takts (int/ext = 1).

Zumeist wird die Betriebsart Zeitgeber genutzt, um mit dem Nulldurchgang (einen programmtechnisch zeitlich festlegbaren) regelmäßige Start eines zugeordneten Interruptprogramms durchzuführen (siehe Interrupt, Abschnitt 5.4).

7.7. Analog-Ein-/Ausgabe

Die Datenwerte sind bei analoger Darstellung Spannungen als Zeitverlauf:

$$U = f(t)$$

Dabei geht es um die Eingabe physikalischer Größen, deren Verlauf auf einen Spannungsverlauf abgebildet wird (z.B. durch Sensoren).

Es entstehen die in Bild 7.26 S.205 dargestellten Sachverhalte.

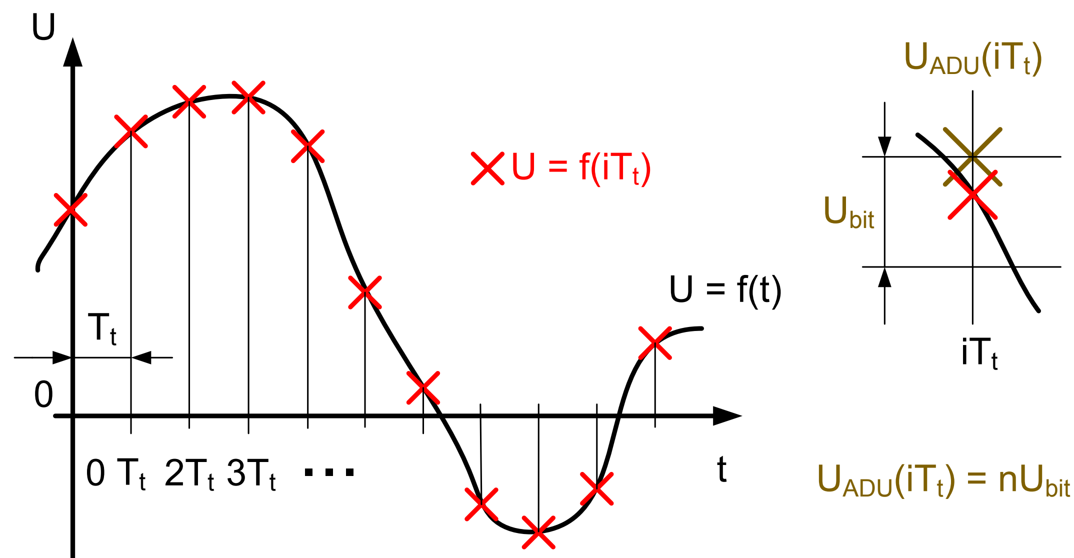


Bild 7.26: Analoge Eingabe: Spanungsverläufe und digitalisierte Werte

Für die weitere Verarbeitung im Rechner können nicht alle Funktionswerte des Funktionsverlaufs verwendet werden. Deshalb erfolgt eine Zeitquantisierung. Es werden nur die Werte $U = f(i * T_t)$ mit i ganzzahlig und konstantem T_t (Tastperiode) verwendet. Die Wahl von T_t und die möglichen Verarbeitungsalgorithmen sind Gegenstand der „digitalen Signalverarbeitung". Zusätzlich ist eine Wertquantisierung notwendig, die $U = f(i * T_t)$ auf $U_{ADU}(i * T_t) = n * U_{bit}$ mit ganzzahligem n abbildet. Dabei wird der nächste zu $U = f(i * T_t)$ liegende $n * U_{bit}$ -Wert verwendet. Beide Quantisierungen werden unter Nutzung eines Analog-Digital-Umsetzers (ADU, auch Analog Digital Converter, ADC) durchgeführt.

Es entsteht die Grundstruktur von Bild 7.27 S.207 (oberer Teil).

Dabei wandelt der ADU das anliegende Spannungssignal U_e in eine (ganze) Binärzahl, beginnend mit dem Signal Start über die Dauer von T_t. Er meldet deren Ablauf durch das Signal Ende. T_t ist sehr viel größer als die Abarbeitung eines Befehls durch den Prozessor. Deshalb wird eine Zeitsteuerung eingesetzt, die mit Ende Interrupt (IRQ_{ADU}) generiert, aufgrund dessen der Prozessor den gewandelten AD-Wert über die synchronisierte parallele Eingabe übernimmt.

Im unteren Teil von Bild 7.27 S.207 ist die Ausgabe von analogen Spannungen $U = f(i * T_t)$ dargestellt. Basis ist ein Digital-Analog-Umsetzer (DAU, auch Digital Analog Converter,

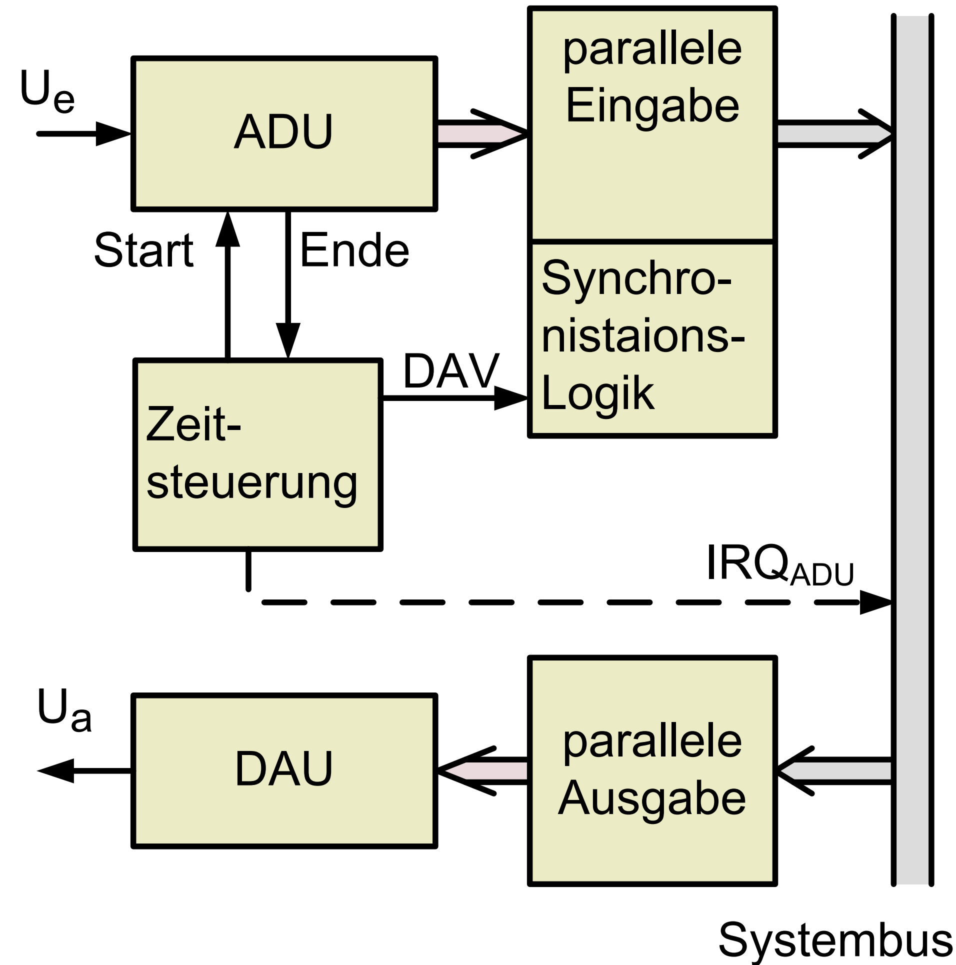


Bild 7.27: Analog-Ein- und Ausgabe

ADC), dessen Eingänge mit einer parallelen digitalen Ausgabe verbunden sind. Der auf diese ausgegebene binäre Digitalwert wird in die Ausgangsspannung nach der Formel

$$U_a = U_{bit} \times \sum_{i=0}^{k-1} b_i \, 2^i \quad \text{mit} \quad b_i = 0 \text{ oder } 1,$$ je nach Belegung in der Binärzahl,

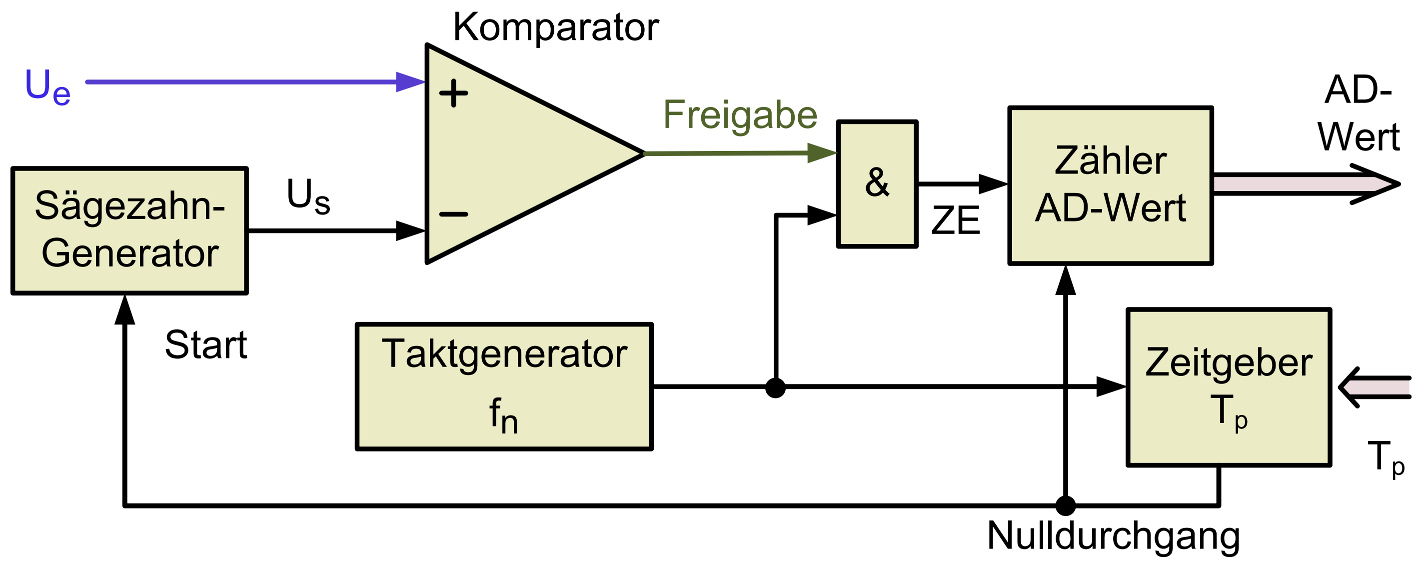
umgewandelt.

Der Ausgang ist für die Dauer T_t aufgrund der speichernden Wirkung der digitalen Ausgabe konstant. Die Notwendigkeit zur Ausgabe von Spannungen an technische Systeme nimmt ab, so dass diese Funktion in heutigen Rechnern nicht mehr häufig anzutreffen ist.

ADU's gibt es auf Basis verschiedener Verfahren. Hier soll das leicht verständliche „Sägezahn-Verfahren" (auch Single Slope Converter) erläutert werden (Bild 7.28 S.209).

Die Eingangsspannung U_e wird mit einer („Sägezahn-") Spannung U_s über einen Komparator verglichen. So lange $U_e \leq U_s$ ist, wird durch dessen Funktionsweise der Ausgang Freigabe = 1 und die Taktimpulse des Taktgenerators (mit Frequenz f_n) werden vom Zähler AD-Wert gezählt (ZE). Der Zählerstand zum Zeitpunkt $U_e = U_s$ ist der gewandelte Digitalwert und kann vom Prozessor gelesen werden. Für die Zeit $U_e > U_s$ ist Freigabe = 0, es werden keine weiteren Impulse gezählt. Der Zeitgeber T_p setzt mit seinem Nulldurchgang den Sägezahngenerator zurück, der Wandlungsvorgang startet erneut.

Ein Sägezahngenerator ist dabei eine Schaltung, für die für die Dauer von T_p gilt:



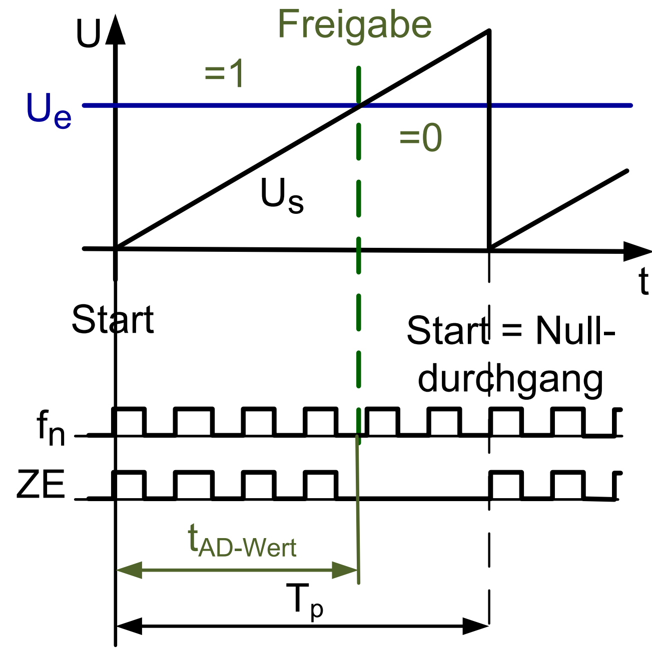


Bild 7.28: ADU nach dem Sägezahnverfahren

$$U_s = U_{ref} \times (t - nT_p)$$

wobei U_{ref} eine konstante Referenzspannung ist. n ist ganzzahlig.

Der Zähler AD-Wert und der Zeitgeber T_p können mit zwei Zähler-/Zeitgeber-Kanälen nach Abschnitt 7.6 gebildet werden.

Eine mögliche Realisierung eines DAU zeigt Bild 7.29 S.211. Sie erzeugt den Spannungsausgang U_{out} nach dieser Formel:

$$U_{out} = - U_{ref} \sum_{i=0}^{k-1} \frac{b_i}{2^{k-i}}$$

$\text{mit} \quad b_i = 0 \text{ oder } 1$, Stellen der ausgegebenen (ganzen positiven) Binärzahl.

Grundlage ist die summierende Wirkung des Operationsverstärkers in dieser Schaltung.

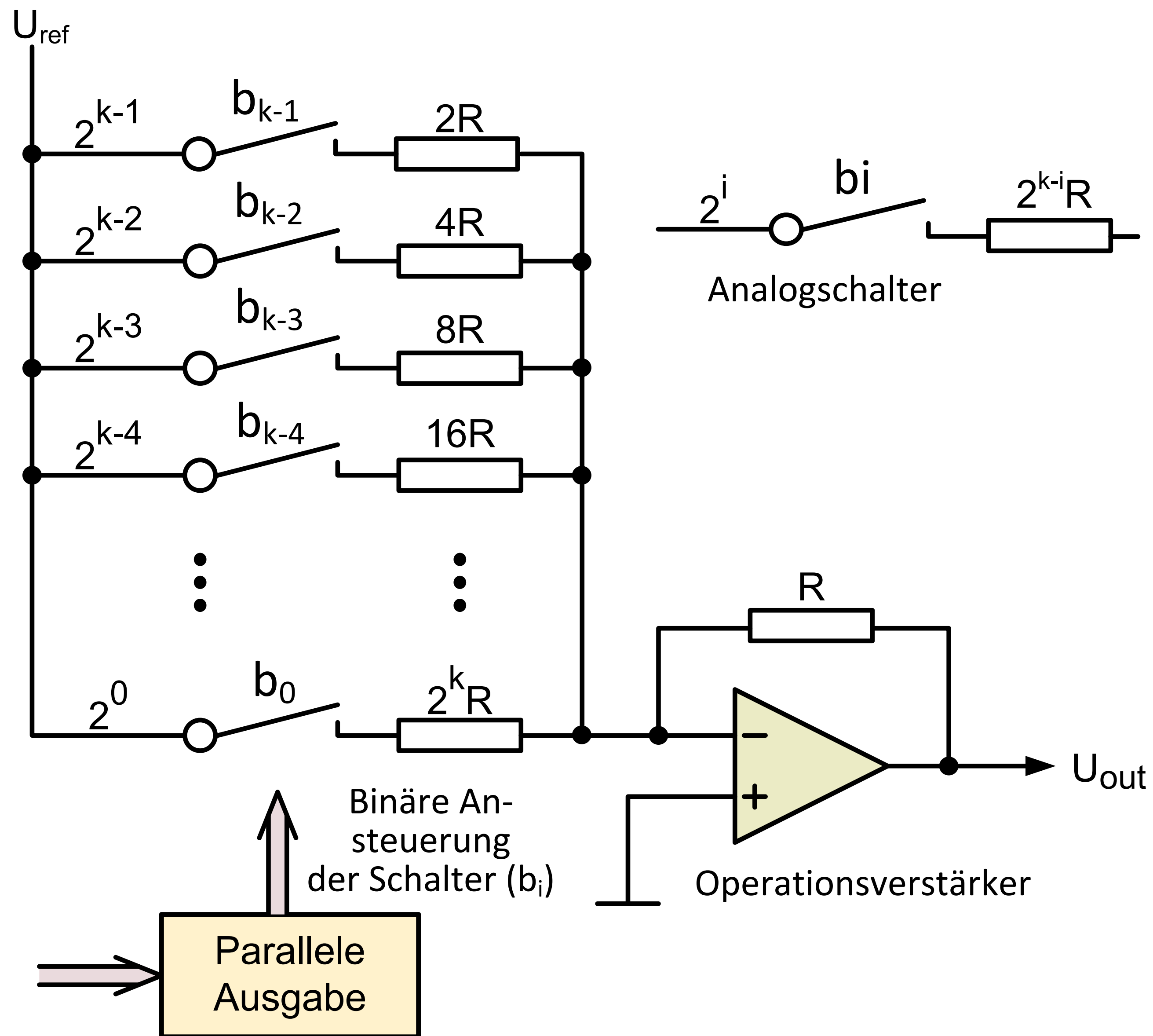


Bild 7.29: DAU mit Widerstandsnetzwerk

8. Weiterführende Konzepte der Rechnerarchitektur

8.1. Prozessorleistung

Die meisten der folgenden weiterführenden Konzepte haben als Ziel die Erhöhung der Prozessorleistung. Dabei gilt als grobe Abschätzung:

Leistung = Arbeit (Befehle) / Zeit

Dabei wird die Befehlsanzahl und die Zeit gemessen und die Messungen gelten genau für dieses Anwendungsprogramm in dem gegebenen Kontext (u.a. den aktuellen Daten).

Etwas allgemeiner kann man so vorgehen:

Leistung = Durchschnitts-Befehle / Zeit

Als Gleichungen gelten hier:

$$L = \frac{IPC}{t_{cycle}}$$

$$IPC = \frac{1}{CPI}$$

$$CPI = \sum_{i=0}^{n-1} \mathrm{t_i} \, \mathrm{p_i}$$

mit

- L: Prozessorleistung [MIPS, **M**illion **I**nstructions **P**er **S**econd]
- IPC: **I**nstructions **P**er **C**ycle (Anzahl Durchschnittsbefehle pro Taktzyklus)
- t_{cycle}: Taktzykluszeit [µs]
- CPI: **C**ycles **P**er **I**nstruction (Mittlere Anzahl Taktzyklen pro Befehl)
- t_i: Anzahl Taktzyklen des i-ten Befehls
- p_i: Relative Häufigkeit des i-ten Befehls
- n: Anzahl der Befehlstypen

Es folgt ein Beispiel mit konkreten Werten:

Taktfrequenz = 1GHz

t_{cycle} = 0,001 µs

fiktive Häufigkeitsverteilung:

1-Takt-Befehle: p_1 = 33,3%

2-Takt-befehle: p_2 = 33,3%

3-Taktbefehle: p_3 = 33,3%

$CPI = \sum_{i=0}^{2} t_i\, p_i$ = 1 × 0,333 + 2 × 0,333 + 3 × 0,333

= 2 (p_i und CPI gerundet)

$IPC = \frac{1}{CPI}$ $= \frac{1}{2}$ = 0,5

$L = \frac{IPC}{t_{cycle}}$ $= \frac{0,5}{0,001\ \mu s}$ = 500 MIPS

Die p_i sind abhängig von dem typischen Auftreten der einzelnen Befehle in der Anwendungsklasse:

- z.B. sehr viele Berechnungen -> viele ALU-Befehle,
- Behandlung großer Datenstrukturen -> viele Speicherbefehle.

Prinzipielle Möglichkeiten zur Erhöhung der Leistung sind:

➔ Parallelisierung der Befehlsabarbeitung im Prozessor,

➔ Parallelisierung von mehreren Prozessoren,

- ➔ Erhöhung der Taktfrequenz (physikalische Grenze z.Z. etwa 4 GHz, verschiedene Ursachen),
- ➔ kürzere Speicherzugriffe.

Zumeist wird versucht, eine Kombination aller vier anzustreben.

8.2. Parallelisierung im Prozessor

Diese wird im Weiteren mit Mikroparallelität bezeichnet.

8.2.1. Befehlspipelining

In Abschnitt 4.1 wurde als Prinzip eingeführt:

RISC: Reduced Instruction Set Computer (Rechner mit reduziertem Befehlssatz), es existieren einfache reguläre Befehle.

Die Befehle mit Operandenmanipulation (EX-Befehle) habe dabei einen relativ einfachen und gleichmäßigen Ablauf in 5 Phasen: IF (Instruction Fetch), ID (Instruction Decode), OF (Operand Fetch), EX (Execute), WB (Write Back) (siehe Bild 8.1 S.216). Dabei realisieren

diese den Ablauf von Bild 5.7 S.106 (Abschnitt 5.3, Befehlsabarbeitung), wobei IF und NIA (Next Instruction Address, hier durch IP:=IP+4) in einen Zustand zusammengefasst wurden.

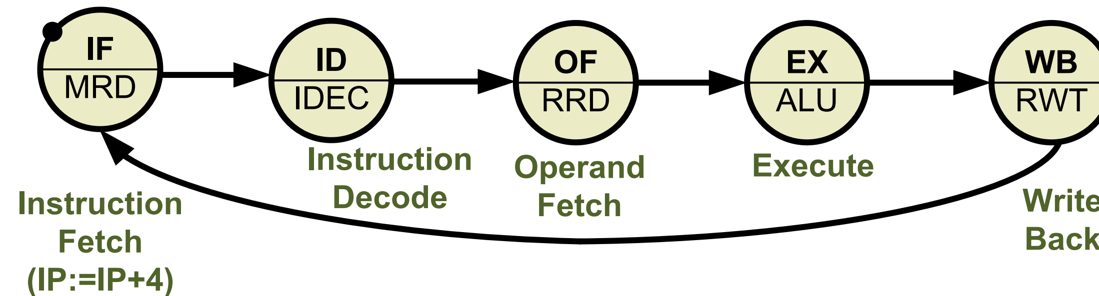


Bild 8.1: Befehlsablauf eines EX-Befehls im RISC-Prozessor

Es wird im Nachstehenden davon ausgegangen, dass diese Zustände (Befehlsphasen) in je einem Prozessortakt abgearbeitet werden können. Das bedeutet, dass ein Befehl 5 Taktzeiten (t_{cycle}) benötigt, um abgearbeitet zu werden.

Bild 8.2 S.217 (entsprechend Bild 5.4 S.100, Prozessorgrundstruktur) zeigt die Funktionsblöcke, die in den einzelnen Phasen aktiv sind. Erkennbar ist, dass die einzelnen Blöcke vorwiegend nur in einer Phase aktiv sind. Deshalb kann ohne größeren Logikaufwand im Prozessor zu einer fünfstufigen Überlappung der Befehle übergegangen werden. Bild 8.3 S.218 zeigt oben die rein sequentielle Abarbeitung der Befehle, unten die mit der um einen Takt verschobenen.

Oben (ohne Pipelining) ist dargestellt: Der nächste Befehl folgt, wenn der letzte vollständig abgearbeitet ist: Alle 5 Takte wird ein Befehl fertig.

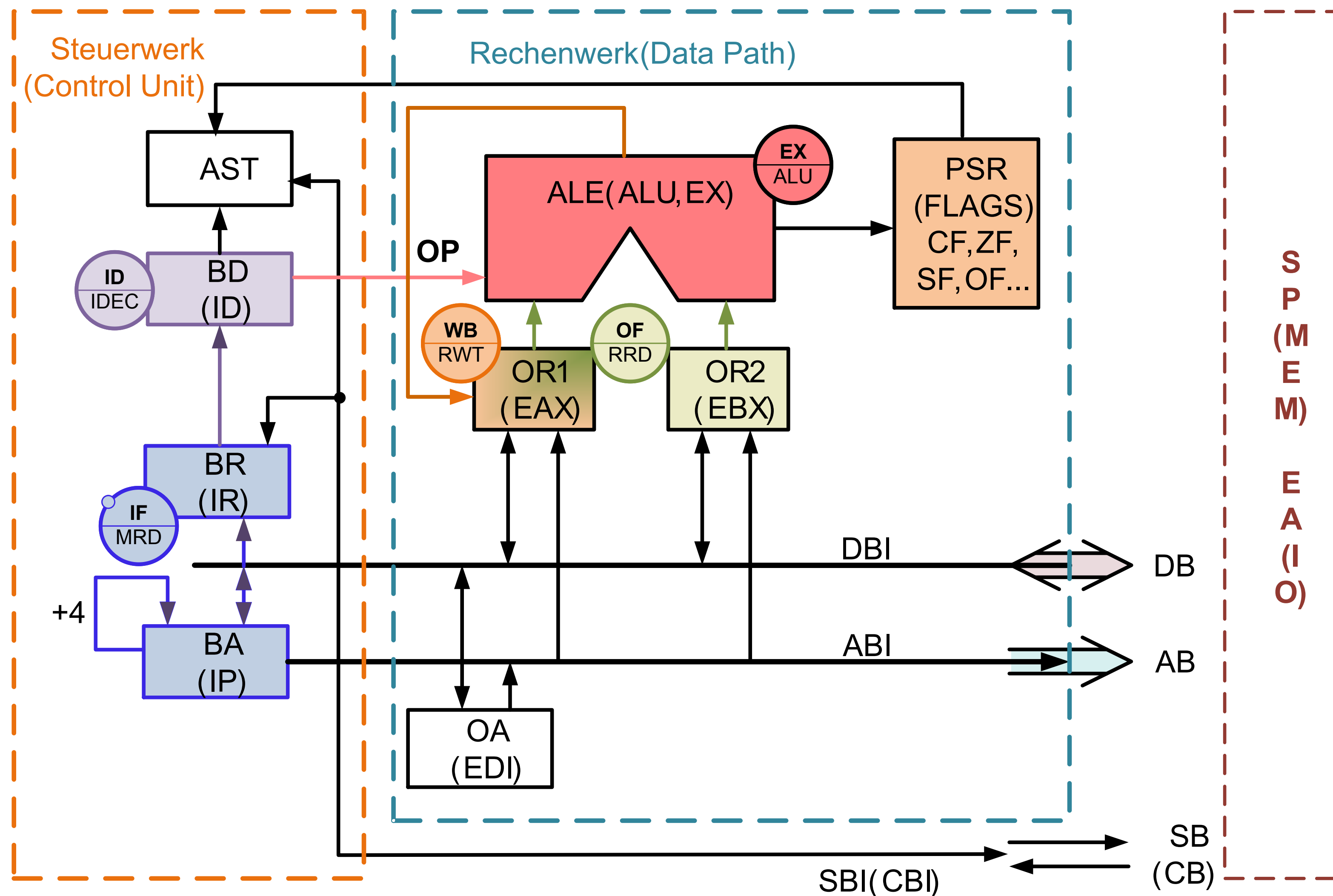
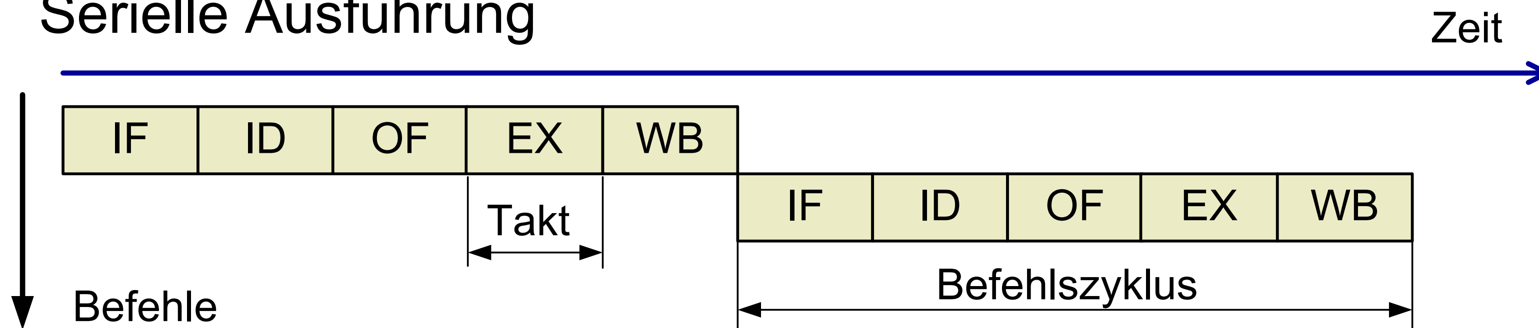
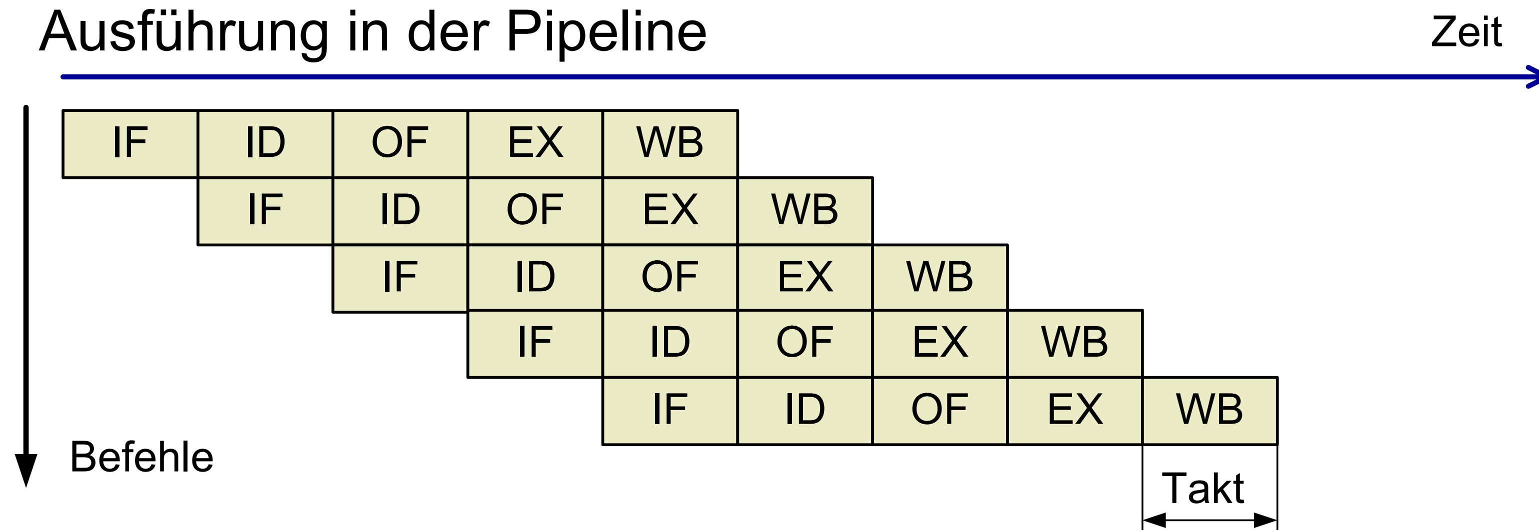


Bild 8.2: Prozessorgrundstruktur und RISC-Execute-Befehlsablauf





IF / ID / EX: Befehl lesen / dekodieren / ausführen
OF / WB: Operand lesen / schreiben (Register)

Bild 8.3: Befehlsablauf ohne (oben) und mit (unten) Pipelining

Unten (mit Pipelining): Der nächste Befehl folgt, wenn die aktuelle Phase (1 Takt) des letzten Befehls zu Ende ist. Alle Phasen arbeiten für 5 Befehle gleichzeitig um eine Phase zeitversetzt: Ein Befehl dauert auch 5 Takte, aber jeden Takt wird ein Befehl fertig.

Die Leistungsverbesserung ist hier 5:1.

➔ Es existieren Ausnahmen für andere Befehlsgruppen (z.B. Sprünge, Speicherzugriffe) und spezielle Befehlsfolgen (Datenabhängigkeiten: der nächste Befehl benötigt Registerdaten des aktuellen Befehls), die spezielle Maßnahmen benötigen und dieses (prinzipielle) Verhältnis verringern.

8.2.2. Superskalare Prozessorarchitektur

Das Grundprinzip der superskalaren Architektur zeigt Bild 8.4 S.220. Sie enthält alle Funktionsblöcke von Pipelining (IF, ID, OF, EX, WB), (Abschnitt 8.2.1).

In der hier erweiterten Pipeline werden die Stufen OF-EX-WB mehrfach (hier 4-fach) parallel realisiert. Der Grund ist, dass die EX-Phasen zumeist eine längere Ausführungszeit besitzen als die anderen vier. Da EX-Einheiten jetzt mehrfach vorhanden sind, können diese zueinander parallel arbeiten, was sonst nicht möglich ist.

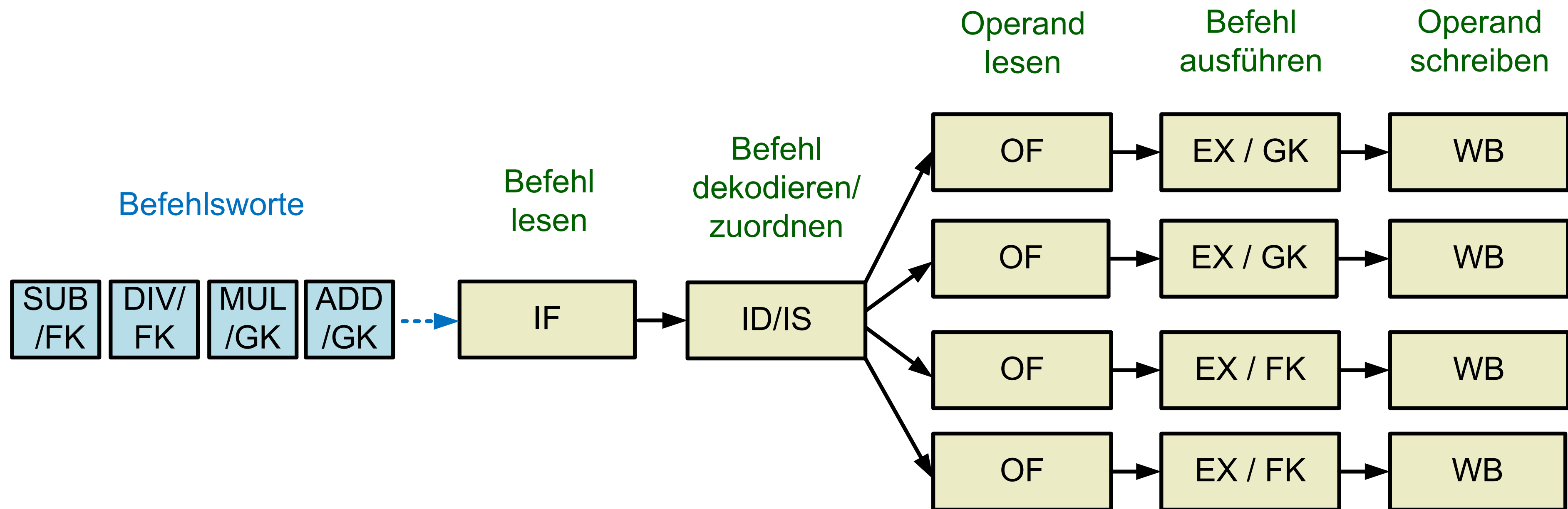


Bild 8.4: Superskalare Prozessorarchitektur

➔ Das führt zu erhöhtem Hardwareaufwand, der eventuell verringert werden kann. Das Mittel ist Spezialisierung der EX-Einheiten (nicht jede EX-Einheit kann alle Befehle ausführen, hier zwei Festkommaeinheiten (FK) und zwei Gleitkommaeinheiten (GK)). Das führt zu einer einfacheren und kleineren Logik für die einzelnen EX-Einheiten.

Der Instruction Decoder (ID) wird ergänzt durch einen Instruction Scheduler (IS). Letzterer ordnet den einzelnen OF-EX-WB-Pfaden Befehle aus dem Befehlsstrom der mit IF gelesenen Befehle zu. Die Kriterien der Zuordnung sind:

1. Der Pfad enthält eine EX-Einheit, die den Befehlstyp ausführen kann.

2. Die EX-Einheit ist mit der Ausführung des vorherigen, ihr zugeordneten Befehls fertig.

Bei vollständiger Ausnutzung aller EX-Einheiten ist die Leistungssteigerung hier theoretisch 4:1.

Die Voraussetzungen dafür sind:

1. IF, ID, OF, WB sind 4 mal schneller als EX,
2. Die Reihenfolge der mit IF gelesenen Befehle ermöglicht eine ständige Auslastung aller EX-Einheiten.

Die Forderung 2. wird zumeist nicht erfüllt, so dass die Leistungssteigerung realistisch unter dem theoretischen Maximalwert liegt.

8.2.3. Very-Long-Instruction-Word-Prozessorarchitektur

Very-Long-Instruction-Word (VLIW) steht für die Verwendung sehr langer Befehlsworte, in denen der Maschinencode mehrerer Befehle zugefasst wird.

Bild 8.5 S.222 zeigt die notwendige Struktur. Sie besteht aus vier (Anzahl ist beispielhaft) Pipelines.

Gegenüber der superskalaren Architektur (Abschnitt 8.2.2) werden die Phasen IF, ID auch parallelisiert. Vier Befehle werden gleichzeitig eingelesen und gleichzeitig dekodiert.

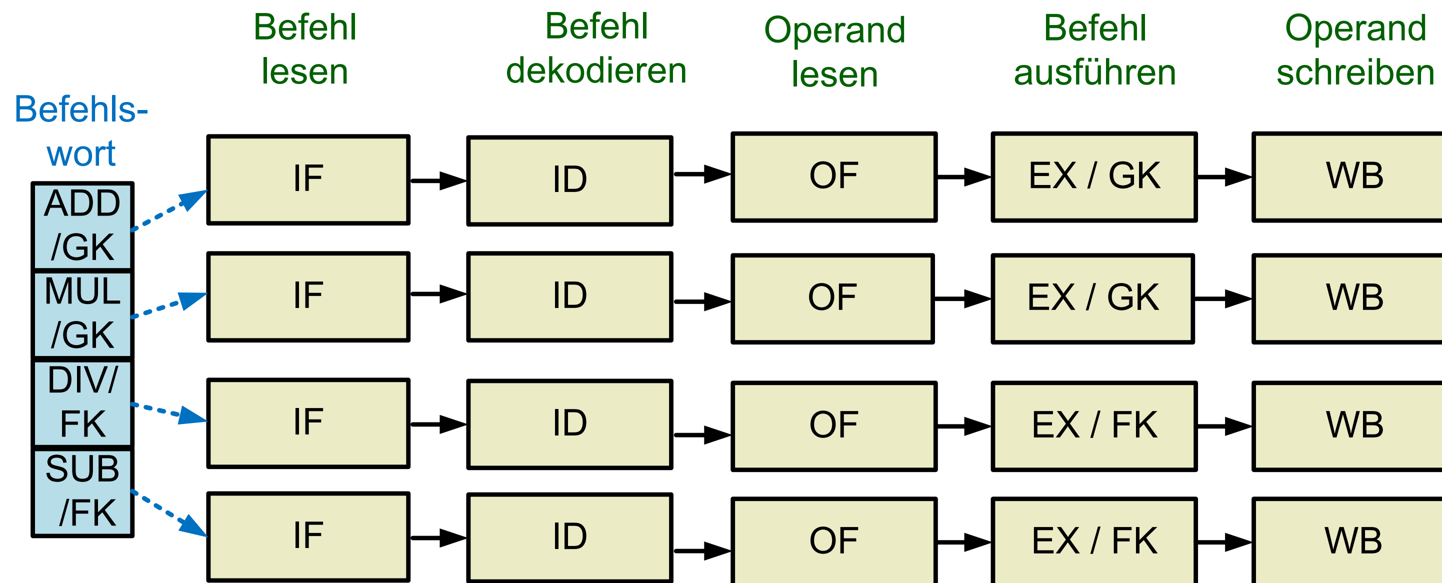


Bild 8.5: VLIW-Prozessorarchitektur

Diese Phasen müssen deshalb nicht schneller sein als die EX-Phasen. Es ist notwendig, dass die Befehle für die (spezialisierten) EX-Einheiten in der richtigen Reihenfolge im Programmspeicher abgelegt sind. Das ist eine Compileraufgabe.

Auch hier ist bei vollständiger Ausnutzung aller EX-Einheiten die Leistungssteigerung theoretisch 4:1. Für die reale Steigerung gilt auch die Aussage, dass die Reihenfolge der mit IF gelesenen Befehle eine ständige Auslastung aller EX-Einheiten ermöglichen muss.

8.2.4. Out-of-Order-Prozessorarchitektur

Die Out-of-Order-Prozessorarchitektur ermöglicht die Ausführung der Befehle abweichend von der Reihenfolge, wie sie im Speicher stehen. Das soll das Problem der superskalaren Architektur, dass die Reihenfolge der mit IF gelesenen Befehle eine ständige Auslastung aller EX-Einheiten ermöglichen muss, verringern. Bild 8.6 S.223 zeigt eine Struktur, die das realisieren kann.

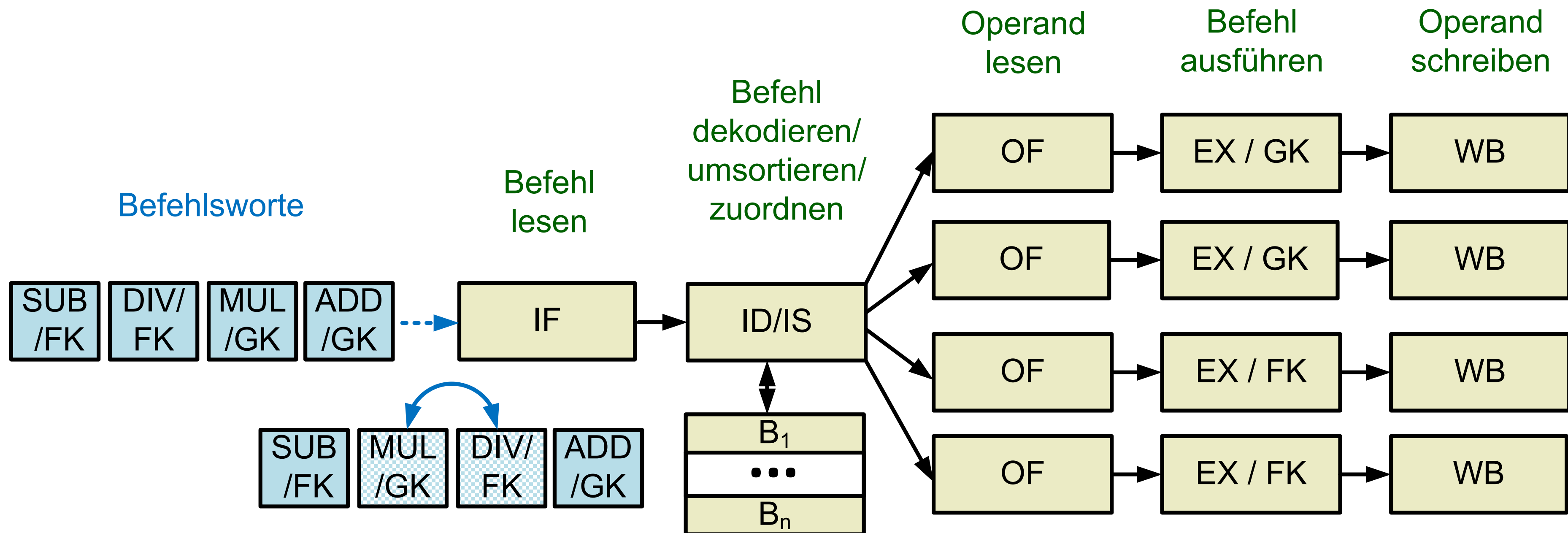


Bild 8.6: Out-of-Order-Prozessorarchitektur

Der Instruction Scheduler (IS) von Bild 8.4 S.220 wird erweitert um einen Befehlspuffer

(Instruction Pool, B_1 bis B_n). In diesem werden Befehle zwischengespeichert, die aktuell zurückgestellt werden. Dafür werden andere vorgezogen.

In Bild 8.6 S.223 ist ein beispielhafter Ablauf angedeutet. Ausgangspunkt sei, dass je eine EX/GK-Einheit und eine EX/FK-Einheit noch mit der Ausführung von vorherigen Befehlen beschäftigt sind. Dem Scheduler stehen nur je eine FK- und eine GK-Einheit aktuell zur Verfügung. Der erste, mit IF gelesene Befehl (ADD/GK) wird der freien GK-Einheit zugeordnet, die mehrere Takte Ausführungszeit besitzt. Für den folgenden Befehl (MUL/GK) steht keine GK-Einheit zur Verfügung. Er wird in dem Befehlspuffer abgelegt und wird erst zu einem späteren Zeitpunkt, wenn eine der beiden GK-Einheiten frei wird, zugeordnet. Parallel wird der Folgebefehl DIV/FK vorgezogen und der noch freien FK-Einheit zugeordnet. Nach diesem Prinzip wird fortgesetzt.

Für existierende Datenabhängigkeiten in den Befehlsfolgen (Folgebefehl(e) von aktuellen Befehlen benötigen deren Daten als Eingangsoperand) muss eine Logik vorgesehen werden, die dieses Problem löst.

8.2.5. Simultanious-Multi-Threading-Prozessorarchitektur

Auch durch Out-of-Order-Befehlsabarbeitung (Bild 8.6 S.223) lässt sich zumeist keine vollständige Auslastung aller EX-Einheiten erreichen. Durch die Simultanious-Multi-

Threading-Prozessorarchitektur (SMT) lässt sich die Auslastung weiter steigern. Dazu ist es nötig, die Programmabarbeitung auf mehrere, relativ unabhängige, parallel bearbeitbare Teilprogramme (Threads) aufzuteilen. Moderne Betriebssysteme/Compiler unterstützen dieses. Bild 8.7 S.225 zeigt, dass die Folge (IF-ID) mehrfach (hier beispielsweise zweifach) vorhanden ist und mit IF mehrere (hier 2) Befehlsströme eingelesen werden.

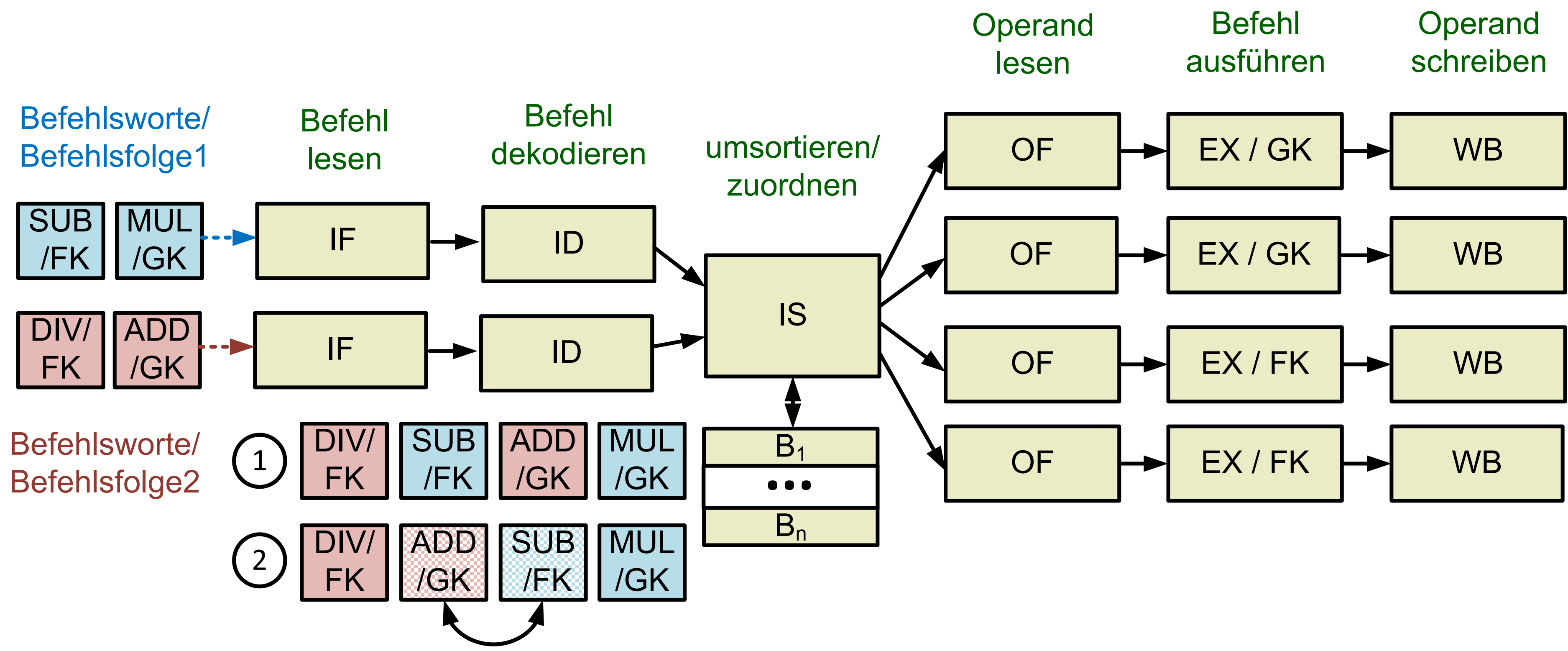


Bild 8.7: Simultanious-Multi-Threading-Prozessorarchitektur

Durch den Instruction Scheduler (IS) können jetzt Befehle aus beiden zugeordnet werden:

(1) hier je ein Befehl aus Befehlsfolge 1, dann aus 2, dann aus 1 usw.
(2) Entsteht dabei wieder das Problem, dass zu einem konkreten Zeitpunkt für den benötigten Befehlstyp keine geeignete EX-Einheit verfügbar ist, kann dieser jetzt gemischte Befehlsstrom aus beiden Befehlsfolgen wie in Out-of-Order noch umsortiert werden (Bild 8.7 S.225).

8.2.6. Single-Instruction-Multiple-Data-Prozessorarchitektur

Die Single-Instruction-Multiple-Data-Prozessorarchitektur (SISD) ist ein Spezialfall. Dabei wirkt der gleiche Befehl auf n unterschiedliche Datenelemente gleichen Typs (z.B. Datenelementgruppen mit zwei Eingangs- und einem Ausgangsoperanden) mit der gleichen Operation. Die Datenelemente müssen einem Ordnungsprinzip unterliegen, wie es z.B. Vektoren oder Matrizen aufweisen. Die Operationen können dann Vektoroperationen sein. Dieser Typ Prozessor heißt dann Vektorprozessor und wird typisch als Coprozessor eines normalen Prozessors verwendet. Die grobe Blockstruktur sieht der von Superskalar (Bild 8.4 S.220) ähnlich und ist in Bild 8.8 S.227 dargestellt.

Hier werden die (Vektor-) Befehle mit IF gelesen und dekodiert. Danach werden sie allen OF-EX-WB-Pfaden zugeordnet, so dass alle EX-Einheiten gleichzeitig die gleiche Operation

auf die mit OF gelesen unterschiedlichen Datenelemente ausführen und die Ergebnisse mit WB getrennt ablegen.

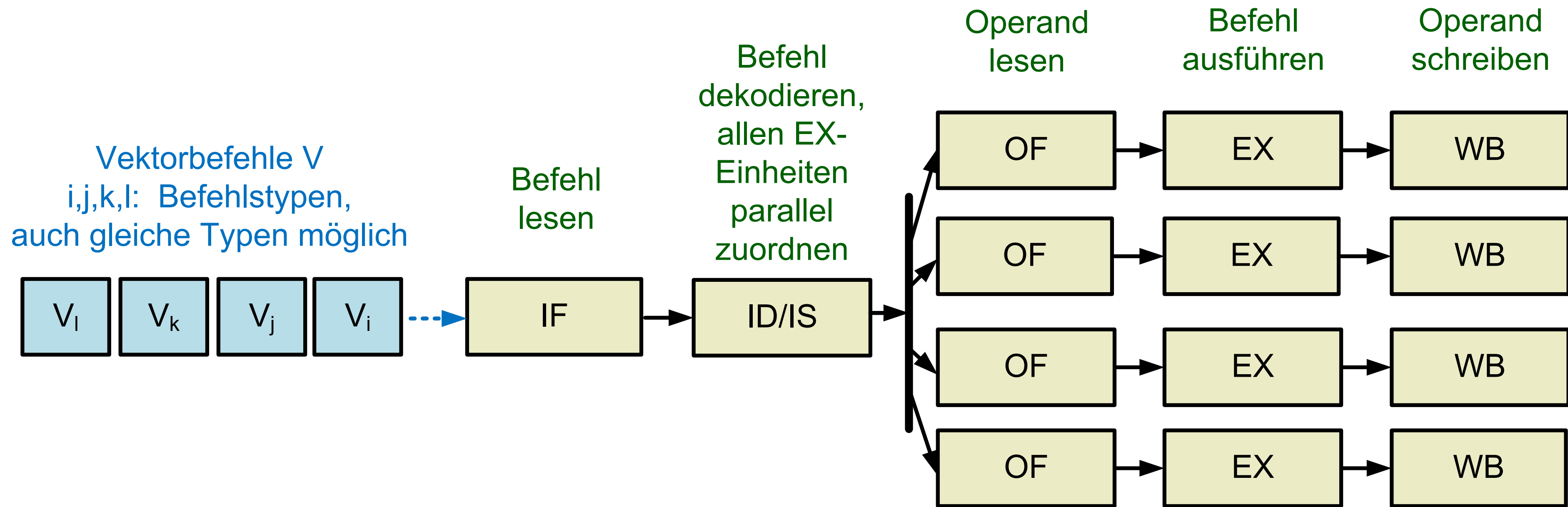


Bild 8.8: Single-Instruction-Multiple-Data-Prozessorarchitektur

8.3. Parallelität von Prozessoren

Diese wird im Weiteren mit Makroparallelität bezeichnet. Dabei sollen hier darunter Architekturen mit dem Prinzip Multiple Instruction Multiple Data (MIMD) verstanden werden.

Dabei existieren mehrere Befehlsströme, die auf unterschiedliche Datenströme wirken. Diese sind relativ unabhängig. Um von einer parallelen Architektur zu sprechen, muss es aber zumindest die Möglichkeit zur Behandlung von Abhängigkeiten über Steuerstrukturen oder Daten geben. MIMD ist bei weitem die wichtigste Form der Parallelität von Prozessoren.

Man unterscheidet zwei Arten:

- Enge Kopplung (Bild 8.9 S.229) und
- lose Kopplung (Bild 8.10 S.230).

Für die unabhängigen Teilprogramme sind in beiden Fällen die lokalen Rechnerteile (Prozessor, Speicher Ein-/Ausgabe (E/A)) zuständig. Die Abhängigkeiten untereinander werden bei enger Kopplung über einen globalen Speicher, auf den beide bzw. mehrere Prozessoren zugreifen können (Zwei/n-Tor-Speicher), realisiert. Bei loser Kopplung erfolgt die Kommunikation über einen globalen Systembus, der über E/A-Funktionseinheiten erreichbar ist. Hier wird zumeist ein serieller Bus eingesetzt, wobei bei mehr als zwei Teilnehmern in den Nachrichten Adressinformationen (Sender-Empfängerbeziehungen (siehe auch Abschnitt 7.5, Bild 7.19 S.196)) enthalten sein müssen.

Multi-Core-Architekturen kann man zur engen, Rechnernetze zur losen Kopplung rechnen.

In beiden Varianten der Kopplung ist eine Zugriffssteuerung notwendig. Bild 8.11 S.231 zeigt einen Automaten zum wechselseitigen Ausschluss von zwei Prozessoren im Zugriff auf einen globalen Speicher.

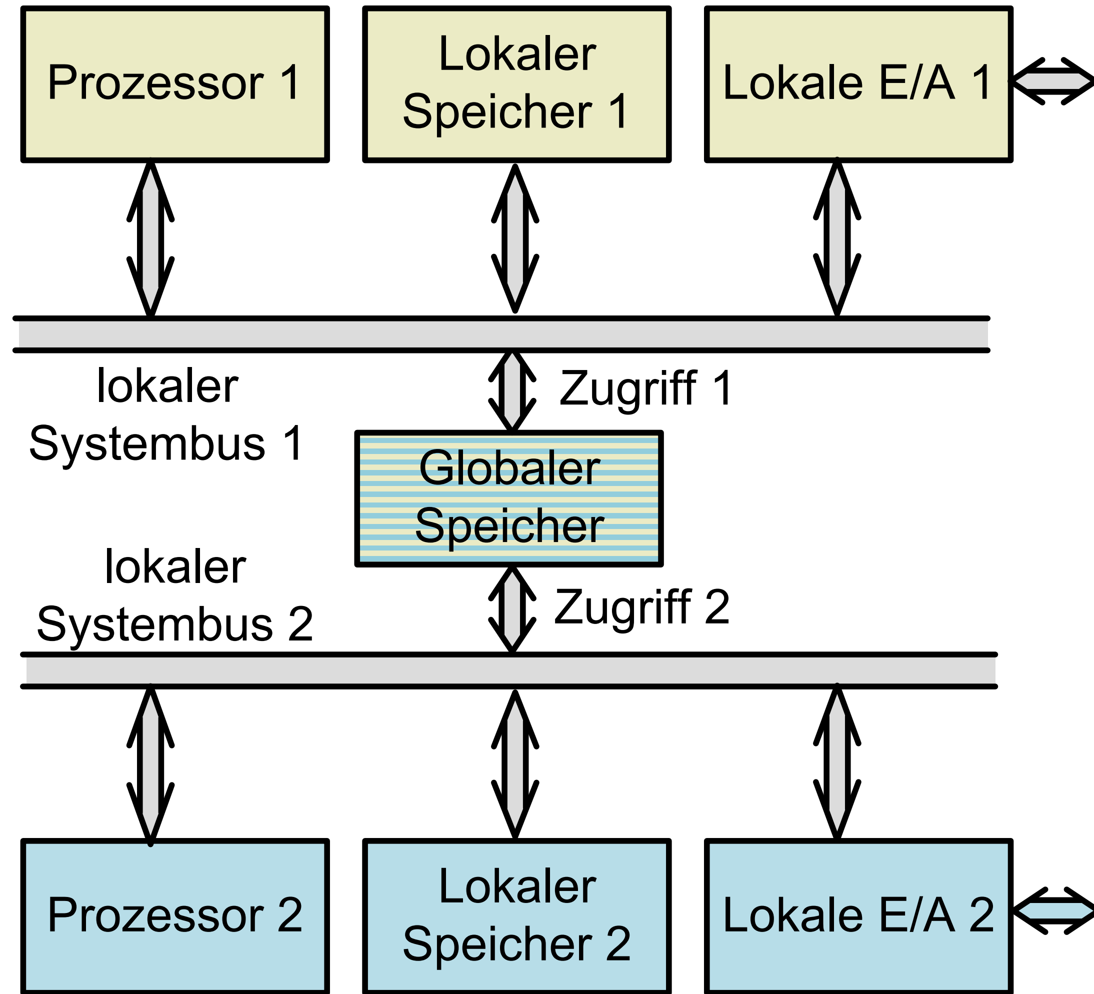


Bild 8.9: Parallele Architektur mit enger Kopplung

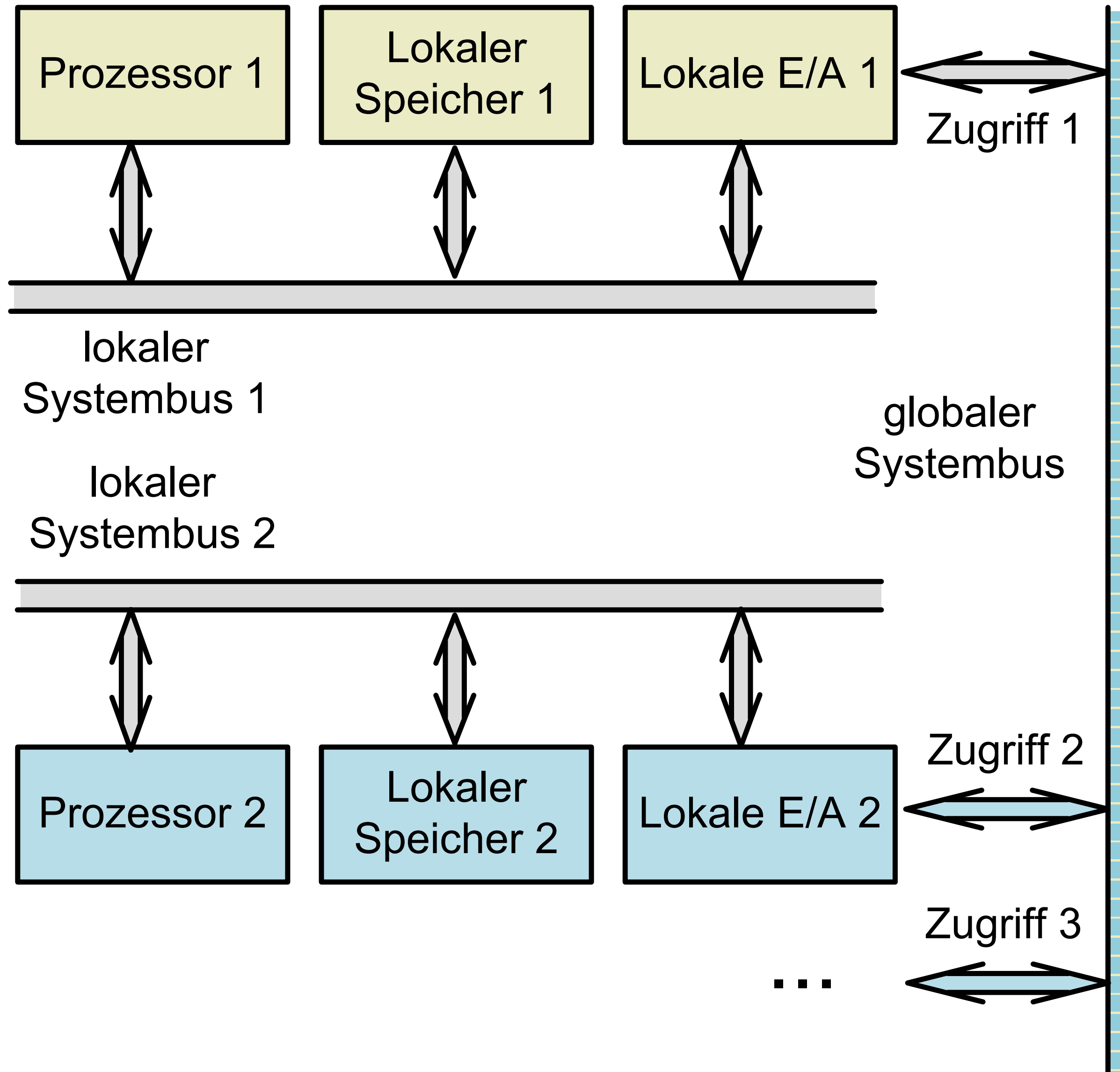


Bild 8.10: Parallele Architektur mit loser Kopplung

Zugriff i ist dabei der Zugriffswunsch des Prozessors i, MRQ_i der daraus resultierende Speicherzugriff des Prozessors i.

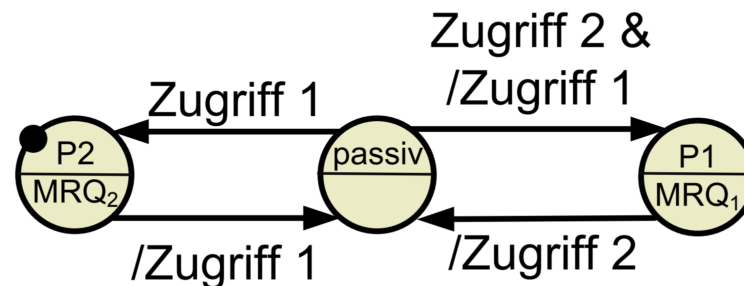


Bild 8.11 Wechselseitiger Ausschluss für einen globalen Speicher

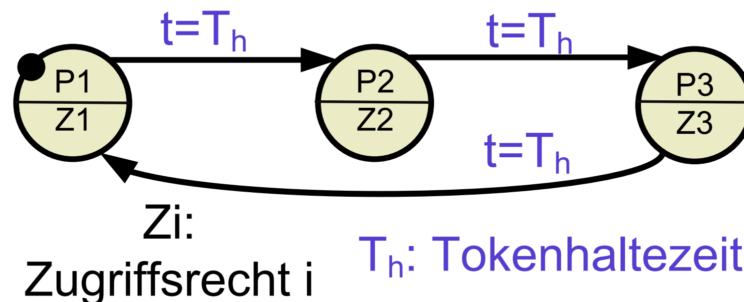


Bild 8.12 Zeitgeteilte Nutzung eines globalen Busses

Bild 8.12 S.231 zeigt einen Automaten zum zeitgeteilten (Time Triggered) Zugriff auf einen globalen Systembus von 3 Prozessoren über E/A. Dabei erhält ein Prozessor das Zugriffsrecht (Zi) für die Dauer der Tokenhaltezeit T_h (Token hier: Senderecht). Danach erhält der nächste dieses usw. Das wird zyklisch für alle Prozessoren (hier 3) wiederholt.

8.4. Leistungssteigerung bei Speicherzugriffen

In jedem (Maschinen-) Befehl ist mindestens ein Speicherzugriff (Instruction Fetch, IF) notwendig. In Befehlen für den Operandentransport kommen weiter dazu: Memory Read (MR) und Memory Write (MW). Dadurch bestimmen die Speicherzugriffe mit ihrer Dauer signifikant die Leistungsfähigkeit der Ausführung von Programmen durch den Prozessor. Letztendlich müssen alle Maßnahmen zur schnelleren Befehlsabarbeitung von Maßnahmen zum kürzeren Speicherzugriff begleitet werden.

In aktuellen Rechnerarchitekturen kann von einer hierarchisch aufeinander aufbauenden Speichernutzung ausgegangen werden (Bild 8.13 S.233). Dabei nimmt die Speichergröße mit der Prozessor-„Ferne" zu. Umgekehrt nimmt die Zugriffsgeschwindigkeit ab. Die am schnellsten zugreifbaren Daten sind in den Prozessorregistern. Deren Anzahl ist allerdings sehr klein. Deshalb haben die weiteren Hierarchieebeben eine entscheidende Bedeutung.

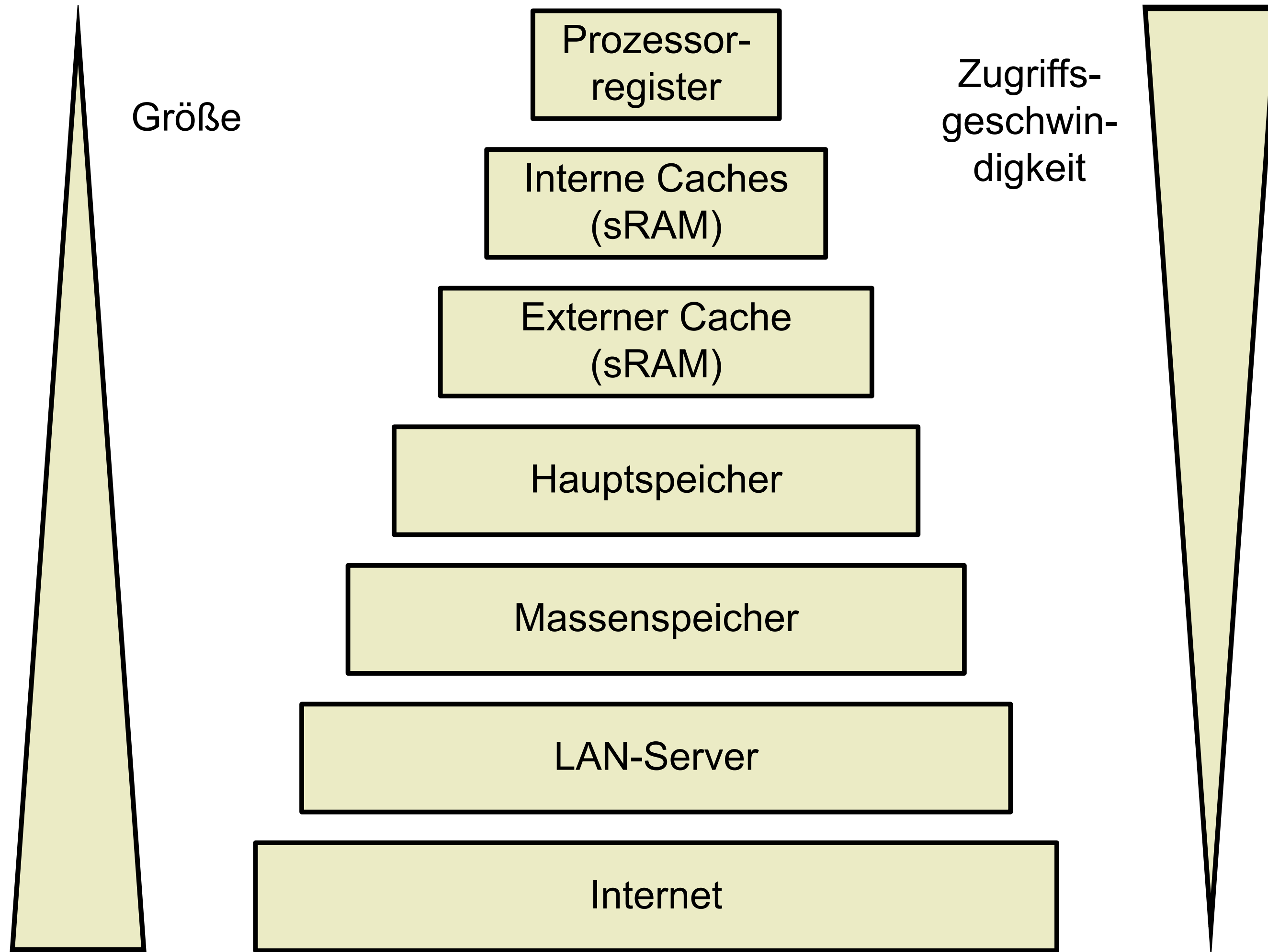


Bild 8.13: Speicherhierarchie

8.4.1. Cache-Speicher

Eine wichtige Hierarchieebene wird durch Cache-Speicher abgedeckt (die evtl. auch mehrstufig sein können). Ihre Funktion ist weitgehend unterschiedlich zu einem normalen Speicher, wie er in den Abschnitten 6.2 und 6.3 beschrieben wurde. Deshalb folgt hier eine Erläuterung ihres Grundprinzips.

Ein Cachespeicher geht von folgenden Grundideen aus:

- Kleine Speicher sind aufgrund der einfacheren Logik schneller und
- kleinere Speicher lassen sich auf den Prozessor-Schaltkreisen integrieren. Dadurch werden größer Datenbreiten im Zugriff (z.B. 256 an Stelle von 64 Bit) und kürzere Signallaufzeiten möglich.
- Kleinere Speicher können mit den schnelleren, aber Chip-Fläche-aufwendigeren sRAM-Bit (siehe Bild 6.1 S.140) realisiert werden.

Da aufgrund der reduzierten Größe nur ein Teil der im Hauptspeicher vorhandenen Programmcodes und Daten im Cache sein können, existieren dort Kopien, von denen angenommen wird, dass sie relativ häufig benutzt werden. Diese Häufigkeit hängt vom aktuellen Programmzustand ab. Eine einfache Annahme ist, dass gerade gelesene oder geschriebene Speicherworte in unmittelbarer Zukunft erneut benötigt werden.

Bild 8.14 S.235 zeigt das Cache-Speicherprinzip für eine Lösung mit vollem Adressvergleich (vollassoziativ). Aufgrund des hohen Logikaufwands und der dann damit zu langen Vergleichszeiten, werden bei realen Caches abgerüstete Varianten verwendet

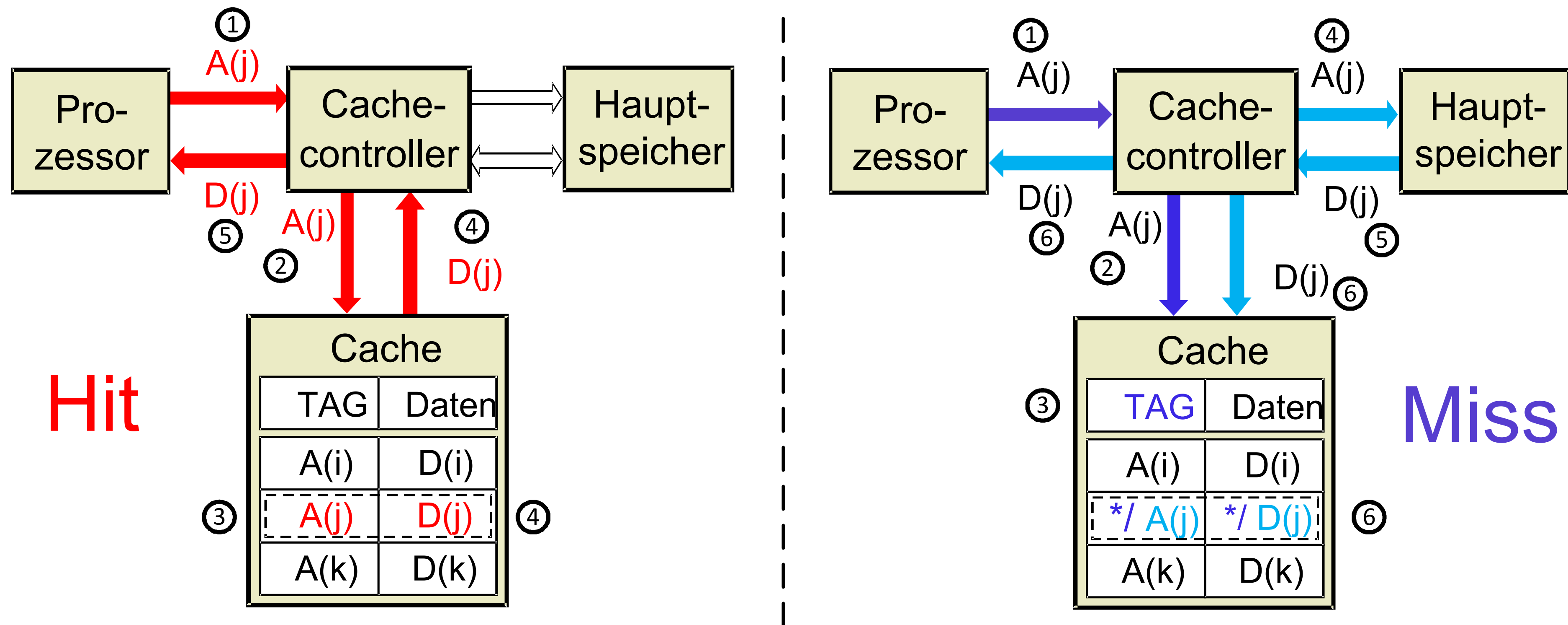


Bild 8.14: Speicherzugriffe (Lesen) mit Cache

(bis zur Reduktion auf genau einen Vergleich). Zur Veranschaulichung des Prinzips ist allerdings die Variante von Bild 8.14 S.235 sehr günstig. Im Gegensatz zu einem normalen Speicher ist im Cache ein Datenspeicherwort mit der dazu gehörigen Hauptspeicheradresse

als Kopie abgelegt. Das liegt daran, dass die Adressierung im Cache über die Hauptspeicheradresse erfolgt. Es muss für ein Datenwort sowohl die Kopie des Hauptspeicherworts als auch eine Zuordnung zu dessen Adresse abgelegt werden, nicht zwingend in der Reihenfolge, wie im Hauptspeicher.

Cachespeicherworte und dazu gehörige Hauptspeicherworte müssen konsistent gehalten werden, d. h. den gleichen Inhalt haben.

Im Bild 8.14 S.235 werden die beiden möglichen Fälle für einen Lesezugriff des Prozessors gezeigt, links für den Fall, dass die gewünschte Adresse mit den Daten bereits als Kopie des Hauptspeicherworts im Cache vorhanden ist (Hit).

Der Prozessor startet mit der Ausgabe der Hauptspeicheradresse A(j) (1). Über den Cachecontroller erfolgt ein Vergleich mit allen im Cache vorhandenen Hauptspeicheradressen (2). Bei Hit befindet sich A(j) im Cache (3) mit den dort eingetragenen Daten (D(j)). Diese werden an den Cachecontroller übertragen (4) und anschließend dem Prozessor übergeben (5). Der Hauptspeicher ist bei Hit nicht beteiligt.

Rechts in Bild 8.14 S.235 ist der Ablauf für den Fall gezeigt, dass sich aktuell keine Kopie des gewünschten Datenworts im Cache befindet (Miss). Der Prozessor gibt A(j) aus (1). Der Vergleich mit den Cache-Einträgen ist negativ (2), (3). Der Cachecontroller sendet A(j) an den Hauptspeicher (4), der das gewünschte Datenwort (D(j) liefert (5). Dieses wird

zusammen mit seiner Hauptspeicheradresse in den Cache eingetragen (6) und parallel dem Prozessor übergeben.

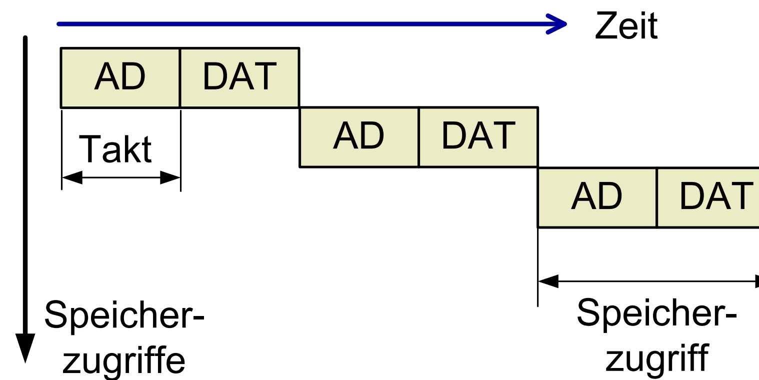
Da der Cache im Allgemeinen vollständig gefüllt ist, muss beim Schreiben D(j) ein Cache-Eintrag überschrieben werden. Dafür ist eine Strategie im Cachecontroller notwendig (z.B. werden länger nicht genutzte Hauptspeicheradressen überschrieben).

Schreiben mit Cachebenutzung erfolgt ähnlich wie bei Miss beim Lesen. Bei Hit und Miss beim Schreiben müssen sowohl im Cache als auch im Hauptspeicher geschrieben werden.

8.4.2. Mikroparallelität in der Speicherarchitektur

In Speichern mit normalem Verhalten, wie in den Abschnitten 6.2 und 6.3 beschrieben, können auch mikroparallel arbeitende Strukturen realisiert werden. Als einfaches Beispiel soll hier das Adresspipelining beschrieben werden. Hier wird der Speicherzugriff in 2 Phasen aufgeteilt (Bild 8.15 S.238):

- Adressübertragung vom Prozessor und –dekodierung im Speicher (AD) und
- Datenbereitstellung zur dekodierten Adresse im Speicher und Übertragung zum Prozessor (DAT).



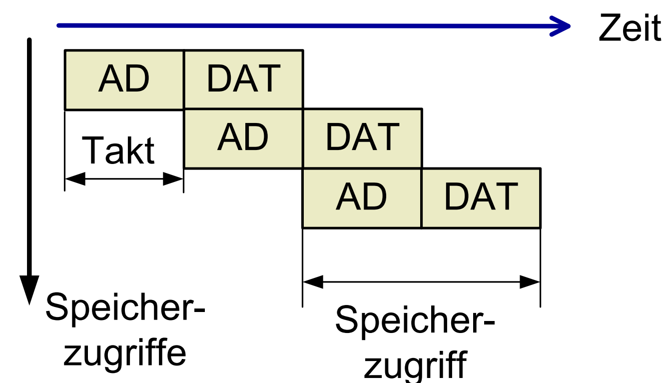


Bild 8.15: Adresspipelining im Speicher

Beide Phasen werden überlappend ausgeführt. Analog zu Pipelining im Prozessor wird der einzelne Speicherzugriff nicht kürzer, aber in der gleichen Zeit werden doppelt so viele Zugriffe möglich.

Literaturverzeichnis

[1] K. Henke und H.-D. Wuttke, Schaltsysteme. Eine automatenbasierte Einführung, Hallbergmoos: Pearson Deutschland GmbH, ISBN 978-3-8632-6543-4, 2002.

[2] D. W. Hoffmann, Grundlagen der Technischen Informatik, München: Carl Hanser Verlag, ISBN 978-3-446-44251-1, 2014.

[3] K. Henke und H.-D. Wuttke, Lehrmaterial Rechnerorganisation, Ilmenau: TU Ilmenau, FG Integrierte Kommunikationssysteme, 2016.

https://www.tu-ilmenau.de/iks/lehre/bachelor-studiengaenge/?lecture_id=29

(Stand: 20. 11. 2016)

[4] W. Fengler, A. Karg, B. Däne und A. Fleischer, Materialien zu den Lehrveranstaltungen Rechnerarchitektur, Ilmenau: Technische Universität Ilmenau, FG Rechnerarchitektur und Eingebettete Systeme, 2009-2015.

http://www.tu-ilmenau.de/ra/i/r-stu/r-lv/r-ra1/lehrmaterial/

(Stand: 20. 11. 2016)

[5] B. Bihr und H.-P. Hübner, „Automobile Vernetzung für mehr Effizienz und Sicherheit,“ in Bosch Engineering GmbH, 61. Motorpressekolloquium, Boxberg, 2013.

http://www.bosch-presse.de/pressportal/de/automobile-vernetzung-fuer-mehr-effizienz-und-sicherheit-42176.html

(Stand: 20. 11. 2016)

[6] F. Martini, „Superhirn im Auto – Siemens hat Elektroauto von StreetScooter mit neuartiger Elektronik und Software ausgerüstet,“ in siemens.com - Informationen für die Presse, München, 2014.

http://www.siemens.com/press/de/feature/2014/corporate/2014-07-race.php?content[]=CC

(Stand: 20. 11. 2016)

[7] G. Schwefer, Pattern-Oriented Transformations between Analysis and Design Models, Ilmenau: TU Ilmenau, Dissertationsschrift, 2008.

http://nbn-resolving.de/urn:nbn:de:gbv:ilm1-2008000021

(Stand: 20. 11. 2016)

[8] D. Harel, „Statecharts: A Visual Formalism For Complex Systems,“ Science of Computer Programming 8, pp. 231-274, 1987.

[9] J. S. Leitner und K. Hinum, „Test Intel HD Graphics 3000 Grafiklösung,“ NOTEBOOKCHECK, 03. 01. 2011, Wien 2011.

http://www.notebookcheck.com/Test-Intel-HD-Graphics-3000-Grafikloesung.43703.0.html

(Stand: 20. 11. 2016)

Abbildungsverzeichnis

Bild 1.1: Steuergeräte im PKW [5], [6] 11
Bild 1.2: Eingebettetes Rechnersystem 12
Bild 1.3: Beispiel zu Steuergeräten (aus einer Fallstudie in [7]) 14
Bild 2.1: Taktverlauf eines synchronen Automaten 16
Bild 2.2: Beispiel eines einfachen Automaten (Automatengraph und Zeitdiagramm) 18
Bild 2.3: Beispiel eines einfachen Systems paralleler Automaten (Automatengraphen und Zustandsfolge) 20
Bild 2.4: Abstrakte Wohngemeinschaft (WG), schematisch 22
Bild 2.5 System aus parallelen Automaten zur Modellierung des Problems WG 23
Bild 2.6: Abstrakte Wohngemeinschaft (WG), schematisch, „Rechnerarchitektur-nah“ 24
Bild 2.7: Prozessor-Chip-Ausschnitt [9], zwei Cores, ergänzt mit Automatengraphen 25
Bild 2.8: Modellierung Dual-Core mit gemeinsamem L3-Cache 26
Bild 2.9: Blockstruktur zum Beispiel „Zwei gekoppelte parallele Automaten“ 28
Bild 2.10: Automatengraphen zum Beispiel „Zwei gekoppelte parallele Automaten“ 29
Bild 2.11: Zeitverläufe zum Beispiel „Zwei gekoppelte parallele Automaten“ 30
Bild 3.1: Grundarchitekturvarianten 32
Bild 3.2: Grundprinzip Adressierung 35
Bild 3.3: Adressierung eines Datenworts aus 4 Byte 37
Bild 3.4: Registermodell 39

Bild 4.1: Prinzip des Maschinencodes (fiktiv) 47
Bild 4.2 (a): Verwendete Adressierungsarten (reduziert aus [4]) 49
Bild 4.2 (b): Verwendete Symbole (reduziert aus [4]) 50
Bild 4.3: Beispiel für einen (physischen) Stapel (Stack) 53
Bild 4.4: Stackbeispiel 54
Bild 4.5: PUSH und POP im Stackspeicherbereich 59
Bild 4.6: Datendarstellung bei Logikbefehlen 69
Bild 4.7: Prinzip der Schiebeoperation 74
Bild 4.8: Prinzip der Operation Rotieren 76
Bild 4.9 (a): Befehlsfolgen mit Verzweigung und Zusammenführung (als Programmliniendarstellung) 78
Bild 4.9 (b): Befehlsfolgen mit Verzweigung und Zusammenführung (als Aktivitätsdiagramm) 80
Bild 4.10 (a): Befehlsfolgen mit Unterprogrammaufruf und Rückkehr (als Programmliniendarstellung) 85
Bild 4.10 (b): Befehlsfolgen mit Unterprogrammaufruf und Rückkehr (als Aktivitätsdiagramm) 86
Bild 4.11: Verschachtelter Aufruf von Unterprogrammen 89
Bild 5.1: Grundarchitektur mit Systembus 94
Bild 5.2: Darstellungsvarianten von Einzelsignalen 96
Bild 5.3: Darstellung von Bussen (Bsp. Datenbus) 98
Bild 5.4: Prozessorgrundstruktur 100

Bild 5.5: Prozessorgrundstruktur mit erweitertem Registersatz 103
Bild 5.6: Automatendarstellung zur Befehlsabarbeitung 104
Bild 5.7: Befehlsabarbeitung von Arithmetik-Logik-Befehlen 106
Bild 5.8: Befehlsabarbeitung von Speicher-Lesebefehlen 107
Bild 5.9: Befehlsabarbeitung von Speicher-Schreibbefehlen 109
Bild 5.10: Befehlsabarbeitung von bedingten Sprungbefehlen 111
Bild 5.11: Befehlsabarbeitung von Unterprogramm-Aufrufbefehlen (Call) 113
Bild 5.12: Befehlsabarbeitung von Unterprogramm-Rückkehrbefehlen (Call) 114
Bild 5.13: Prinzip der Befehlsabarbeitung für alle Befehlsgruppen (-varianten) 116
Bild 5.14 (a): Reaktion durch Teilprogramm auf ein externes Ereignis (als Programmliniendarstellung) 117
Bild 5.14 (b): Reaktion durch Teilprogramm auf ein externes Ereignis (als Aktivitätsdiagramm) 118
Bild 5.15 (a): Reaktion durch Interruptprogramm auf ein externes Ereignis (als Programmliniendarstellung) 120
Bild 5.15 (b): Rektion durch Interruptprogramm auf ein externes Ereignis (als Aktivitätsdiagramm) 121
Bild 5.16: Erweiterung der Ablaufsteuerung für die Interruptfunktion 123
Bild 5.17: Parallele Automatengraphen zum Drucken im Hintergrund 127
Bild 5.18: Zeitverläufe zum Drucken im Hintergrund 129
Bild 5.19: Ablauf bei Speicher Lesen 132
Bild 5.20: Ablauf bei Speicher Lesen als Zeitdiagramm 133

Bild 5.21: Ablauf bei Speicher Schreiben 134
Bild 5.22: Ablauf bei asynchronem Speicher Lesen 136
Bild 6.1: Statisches und dynamischer Speicherbit (prinzipiell) 140
Bild 6.2: Refresh beim dRAM-Bit 144
Bild 6.3: Abstrakte Bitzelle 147
Bild 6.4: 32-Bit-Register aus abstrakten Bitzellen 148
Bild 6.5: 32-Bit-Speicherwort 150
Bild 6.6: Tristate-Treiber mit Ausgangsschaltung (prinzipiell) 151
Bild 6.7: Speicher-IC 153
Bild 6.8: Wertetabelle zum 1-aus-2^i-Dekoder 154
Bild 6.9: Speicherstruktur mit 4x4 Speicher-IC's 156
Bild 6.10: Speicherfunktionseinheit 158
Bild 6.11: Beispielzugriff in der Speicherfunktionseinheit 161
Bild 7.1: Einordnung der Ein-/Ausgabe in die Rechnerarchitektur 163
Bild 7.2: Parallele Datendarstellung (z.B. Datenbus oder parallel extern) 165
Bild 7.3: Parallele Digitale Ausgabe 166
Bild 7.4: Befehlsabarbeitung eines OUT-Befehls 168
Bild 7.5: Zeitablauf der Output-Write-Phase 169
Bild 7.6: Parallele Digitale Eingabe 170
Bild 7.7: Befehlsabarbeitung eines IN-Befehls 171
Bild 7.8: Zeitablauf der Input-Read-Phase 172
Bild 7.9: Zusammenwirken vom Prozessor mit mehreren EA-Schnittstellen 174

Bild 7.10: Systemeinordnung eines Ein-/Ausgabe-Controllers 177
Bild 7.11 Funktionsauswahl durch Steuerregister 179
Bild 7.12 Programmierbare digitale parallele Ein-/Ausgabe 180
Bild 7.13: Struktur der synchronisierten parallelen digitalen Ein-/Ausgabe 183
Bild 7.14 (a) Zwei-Draht-Handshake als parallele Automaten 184
Bild 7.14 (b) Zeitverläufe zum Zwei-Draht-Handshake 185
Bild 7.15: Parallele und serielle Datendarstellung 187
Bild 7.16 (a): Prinzip eines Schieberegisters 189
Bild 7.16 (b): Prinzip eines Schieberegisters als Automaten 190
Bild 7.17: Datenformat und Zeitangaben zur asynchronen seriellen Ein- und Ausgabe 192
Bild 7.18: Prinzipstruktur zur asynchronen seriellen Datenaus- und Eingabe 194
Bild 7.19: Datenformat zur synchronen seriellen Ein- und Ausgabe 196
Bild 7.20: Zwei Varianten zur gemeinsamen Übertragung von Takt und Daten 198
Bild 7.21: Beispielimpulsfolge zum Zählen in einer Zeit T 199
Bild 7.22: Impulsausgabe mit einer konstanten Zeitperiode T_p 200
Bild 7.23: Automatenmodell eines Binärzählers 201
Bild 7.24: Verlauf von Logiksignalen und Zählerständen des Binärzählers von Bild 7.23 202
Bild 7.25: Zähler-Zeitgeber-Funktionseinheit 203
Bild 7.26: Analoge Eingabe: Spanungsverläufe und digitalisierte Werte 205
Bild 7.27: Analog-Ein- und Ausgabe 207

Bild 7.28: ADU nach dem Sägezahnverfahren 209
Bild 7.29: DAU mit Widerstandsnetzwerk 211
Bild 8.1: Befehlsablauf eines EX-Befehls im RISC-Prozessor 216
Bild 8.2: Prozessorgrundstruktur und RISC-Execute-Befehlsablauf 217
Bild 8.3: Befehlsablauf ohne (oben) und mit (unten) Pipelining 218
Bild 8.4: Superskalare Prozessorarchitektur 220
Bild 8.5: VLIW-Prozessorarchitektur 222
Bild 8.6: Out-of-Order-Prozessorarchitektur 223
Bild 8.7: Simultanious-Multi-Threading –Prozessorarchitektur 225
Bild 8.8: Single-Instruction-Multiple-Data-Prozessorarchitektur 227
Bild 8.9: Parallele Architektur mit enger Kopplung 229
Bild 8.10: Parallele Architektur mit loser Kopplung 230
Bild 8.11 Wechselseitiger Ausschluss für einen globalen Speicher 231
Bild 8.12 Zeitgeteilte Nutzung eines globalen Busses 231
Bild 8.13: Speicherhierarchie 233
Bild 8.14: Speicherzugriffe (Lesen) mit Cache 235
Bild 8.15: Adresspipelining im Speicher 238