Informatik II

Dr. Henrik Brosenne
Georg-August-Universität Göttingen
Institut für Informatik

Sommersemester 2017

Inhalt

Schaltnetze

Schaltnetze

- Kombinatorische Schaltungen (*combinational circuits*).
- Nach dem Anlegen der Eingangssignale stellt sich nach endlicher Zeit (Schaltzeit, *hazard*) ein stabiler Zustand an den Ausgängen ein.
- Für die Berechnung der Ausgangswerte ist sind nur die aktuelle Eingangswerte maßgebend (keine früheren Eingangswerte).

Beispiel

Arithmetisch Logische Einheit (*arithmetic logic unit*, ALU).

- Funktionaler Kern eines Digitalrechners.
- Führt arithmetische und logische Operationen aus.
- Eingabe. Daten und Steuersignale.
- Ausgabe. Ergebnis und Statussignale.
- Meist nur für Festkomma- **oder** Gleitkomma-Arithmetik.

Schaltwerke

Schaltwerke

- Sequenzielle Schaltungen (*sequential circuits*).
- Die Ausgangswerte hängen **auch** vom aktuellen inneren Zustand ab.

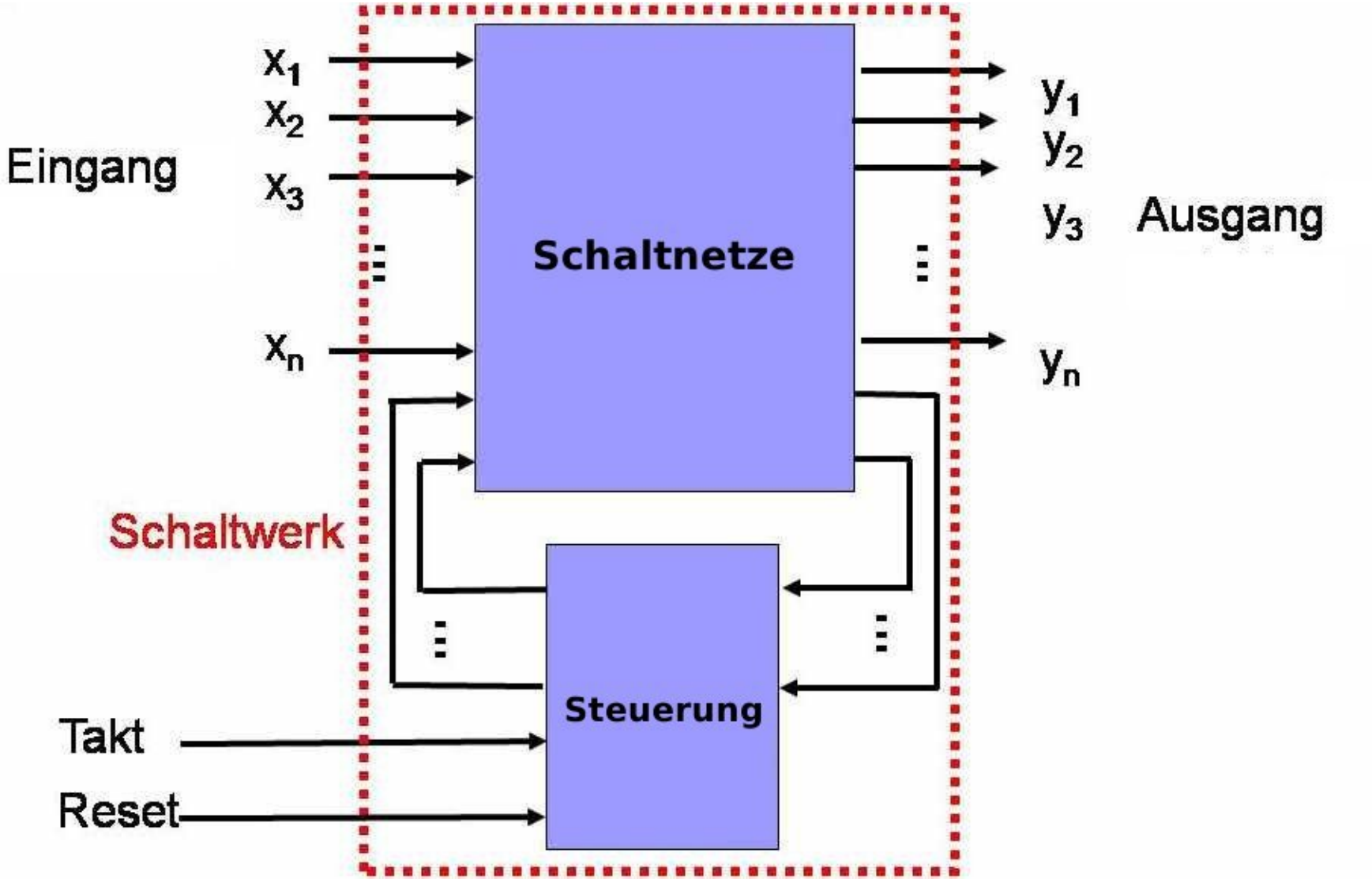
Synchrone Schaltwerke

- Alle Zustandsspeicher werden von einem (oder mehreren) zentralen Synchronisationssignal (Takt) gesteuert.

Asynchrone Schaltwerke

- Die Zustandsspeicher steuern sich gegenseitig, indem sie Synchronisationssignale an nachfolgende Zustandspeicher senden.
- Werden verwendet, weil immer schneller werdende Bausteine asynchrone Entwurfstechniken erzwingen.
- Beispiel. Der Takt für die *schnellen* Bausteine ist höher als die Signallaufzeit eines unentbehrlichen *langsamen* Bausteins.

Beispiel, Schaltwerk

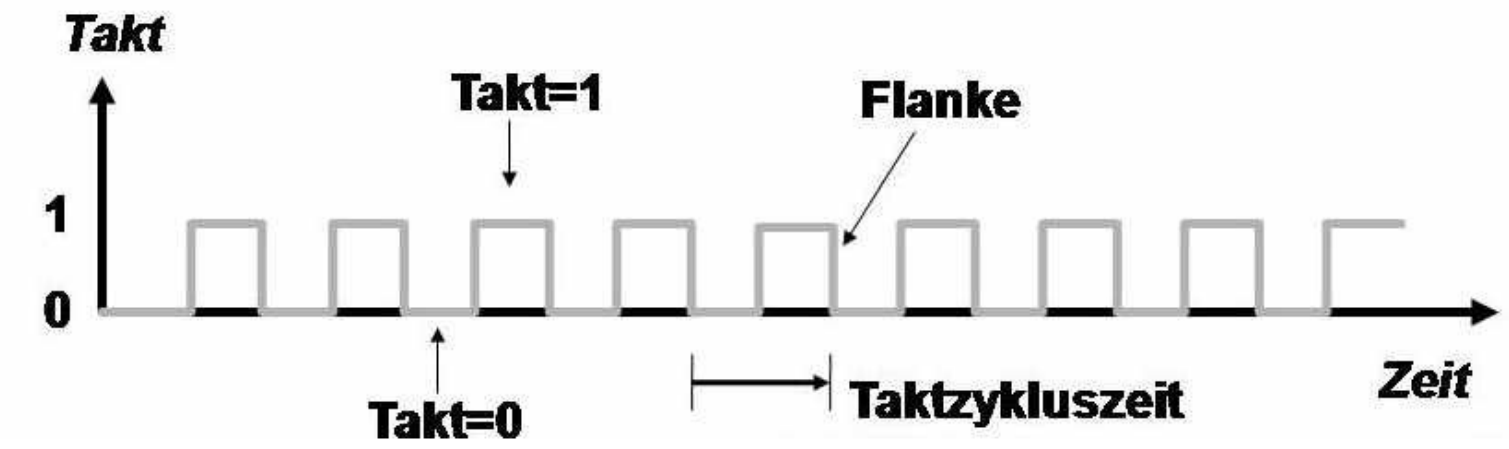


Inhalt

Takt

Taktgeber (*clock*). Ein Schaltung, die eine Reihe von Impulsen in einer präzisen Impulsbreite und in einem präzisen Intervall zwischen zwei aufeinanderfolgende Impulsen ausgibt.

Taktzykluszeit (*clock cycle time*). Intervall zwischen entsprechenden Kanten von zwei aufeinanderfolgenden Impulsen.



Taktzyklen

Taktgeber sind symmetrisch (Low- und High-Zustand sind gleichlang).

Innerhalb eines einzigen Taktzyklus können mehrere Ereignisse ablaufen durch Aufteilung des Taktzyklus in Teilzyklen.

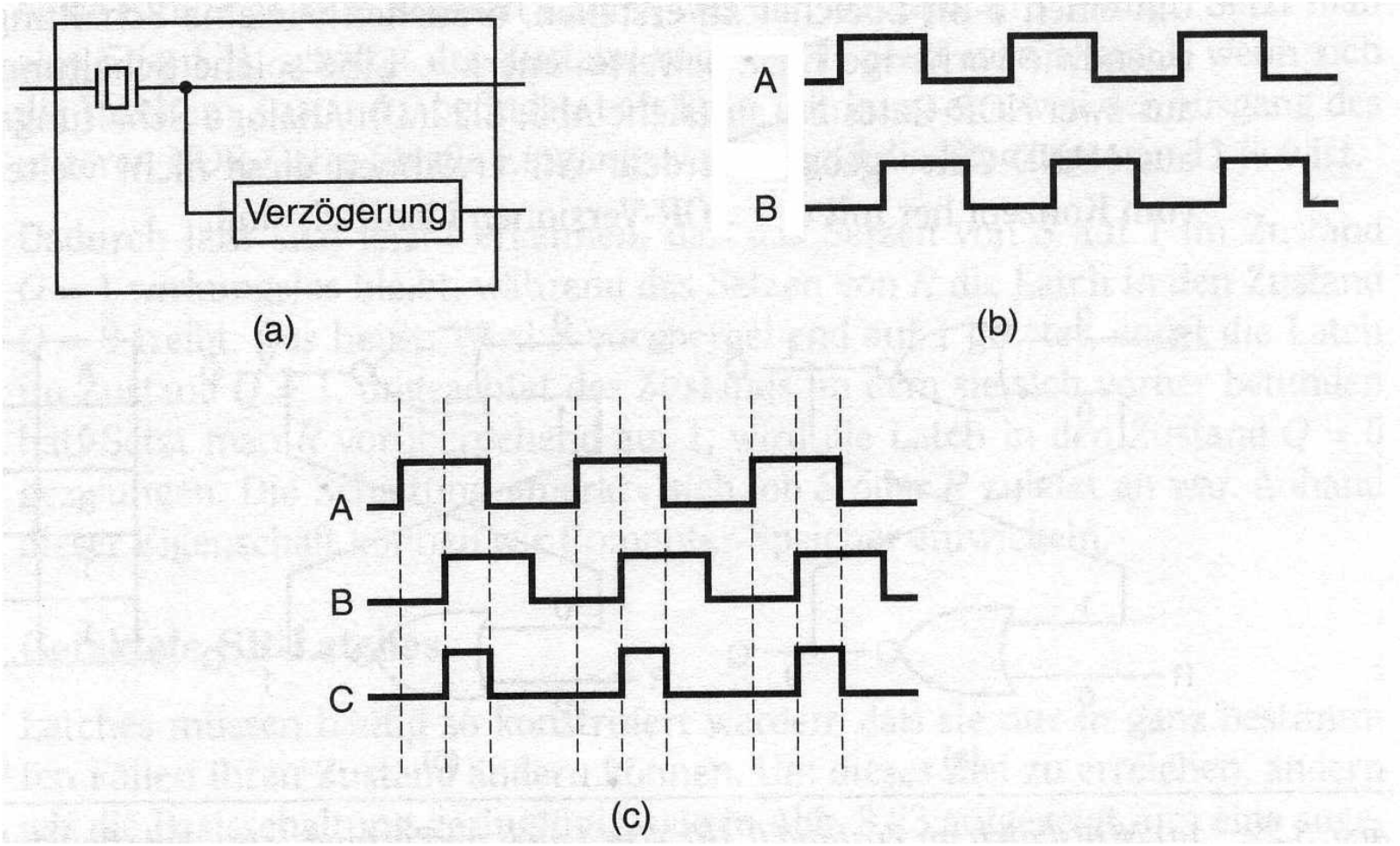
Ein übliches Verfahren, eine feinere Auflösung als der Taktgeber zu bieten, ist das Abgreifen der primären Taktleitung und das Einfügen einer Schaltung mit bekannter Verzögerung.

Ergibt. Sekundäres, zum primären Taktsignal phasenverschobenes, Taktsignal.

Verknüpfung von verschiedenen Ereignissen mit verschiedenen Flanken des primären und sekundären Taktsignals schafft mehrere Zeitverweise pro Takt.

Eine asymmetrische Impulsfolge wird durch eine AND-Verknüpfung des primären mit dem sekundären Taktsignal erreicht.

Takt, Beispiel



(a) Taktgeber
(b) Taktdiagramm für den Taktgeber.
(c) Asymmetrischer Takt, durch (a) `AND` (b).

Inhalt

Pegelsteuerung

Pegelsteuerung (*level-triggered*)

- Der Zustandsspeicher ist während einer Takthälfte **transparent** (durchgeschaltet) und während der anderen **hält** er.
- **Transparenz.** Die Eingänge wirken sich nur bei einem Wert des Taktes (z.B. 1) auf den Zustand aus.
- **Halten.** Bei dem anderen Wert des Taktes (z.B. 0) wird der aktuelle Zustand gehalten (gespeichert).
- Nachteil. Eingangssignale müssen während der gesamten transparenten Taktperiode gültig bleiben.
- Beispiel. *Latches*.

Flankensteuerung

Flankensteuerung (*edge-triggered*)

- Während des Taktwechsels, entweder bei der steigenden (positiven) Taktflanke $0 \rightarrow 1$ oder bei der fallenden (negativen) Taktflanke $1 \rightarrow 0$, wirken sich die Eingänge auf den Zustand aus.
- Vorteil. Die Eingänge müssen nur für sehr kurze Zeit gültig sein.
- Beispiel. *Flip-Flops*.

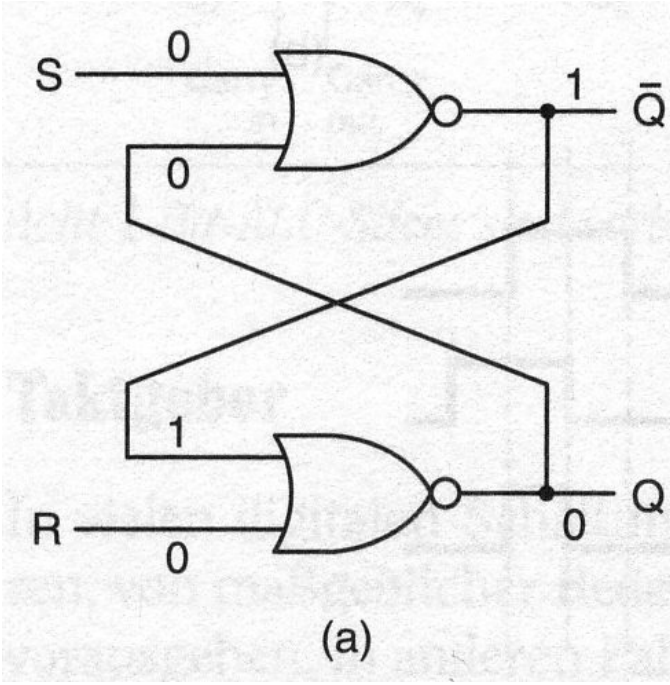
Inhalt

1-Bit Speicher

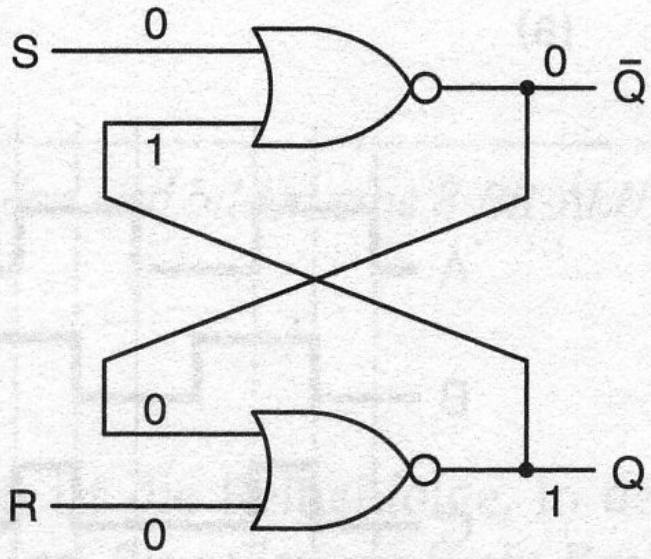
SR-Latch (Klinke)

- Halten (speichern, merken) von vorherigen Eingabewerten.
- Zwei Dateneingänge. S zum Setzen (*set*) und R zum Zurücksetzen (*reset*).
- Zwei komplementäre Datenausgänge Q und $\overline{Q} \equiv \neg Q$.
- **S=R=0.** Q bleibt unverändert, die Latch ist im stabilen Zustand Q.
- **S=1** (set). $Q = 0 \Rightarrow Q = 1$ und $Q = 1 \Rightarrow Q = 1$.
- **R=1** (reset). $Q = 0 \Rightarrow Q = 0$ und $Q = 1 \Rightarrow Q = 0$.

SR-Latch



(a)



(b)

A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0

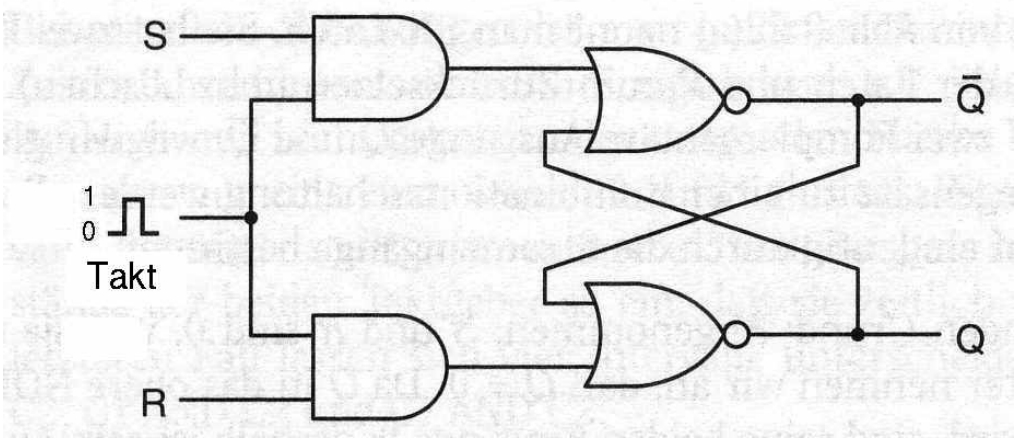
(c)

(a) SR-Latch im Zustand 0.
(b) SR-Latch im Zustand 1.

Pegelgesteuerter 1-Bit Speicher (1/2)

Die Latch soll nur in ganz bestimmten Fällen ihren Zustand ändern dürfen (pegelgesteuert).

Getaktete SR-Latch (*synchronous SR latch, clocked SR latch*)



Problem (mit und ohne Pegel). $S = R = 1$

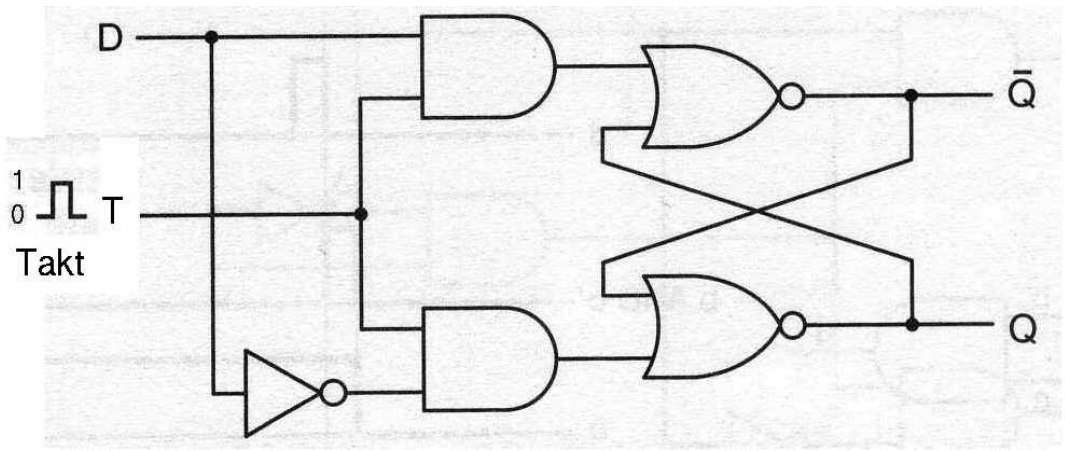
- Einziger stabiler (nicht konsistenter) Zustand ist $Q = \overline{Q} = 0$.
- Beim Pegelwechsel kehren beide Eingänge (Ausgänge der AND–Gatter) auf 0 zurück.

  Kehren beide Eingänge gleichzeitig auf 0 zurück, springt die Latch in einen ihrer stabilen Zustände, bleibt ein Eingäng länger 1 *gewinnt* dieser Eingang.

Pegelgesteuerter 1-Bit Speicher (2/2)

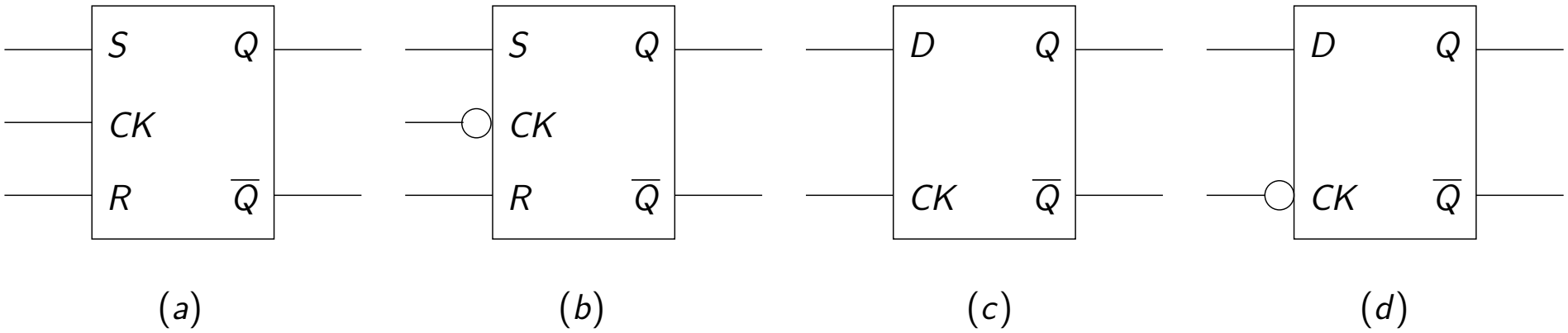
Das Problem der Zweideutigkeit wird am besten gelöst, indem man das Auftreten des Zustandes $S = R = 1$ verhindert.

Getaktete D-Latch (*data latch*, *clocked D latch*)



- **Setzen.** $T = 1$ und $D = 1$.
- **Zurücksetzen.** $T = 1$ und $D = 0$.
- **Halten.** $T = 0$ und D beliebig.

Schaltsymbole für Latches



(a) Getaktete SR-Latch, Zustandsänderung nur bei Takt (*clock*) $CK = 1$.
(b) Getaktete SR-Latch, Zustandsänderung nur bei $CK = 0$.
(c) Getaktete D-Latch, Zustandsänderung nur bei $CK = 1$.
(d) Getaktete D-Latch, Zustandsänderung nur bei $CK = 0$.

Darstellung oft auch ohne $\overline{Q}$.

Inhalt

Flip-Flops

Flankengesteuert

- Zustandübergang tritt bei steigender oder fallender Taktflanke ein.

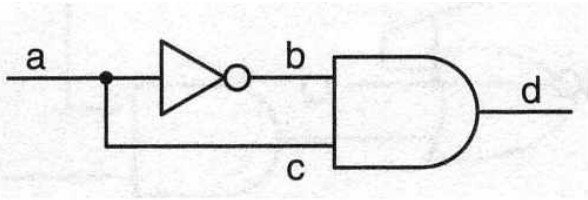
Möglicher Design Ansätze

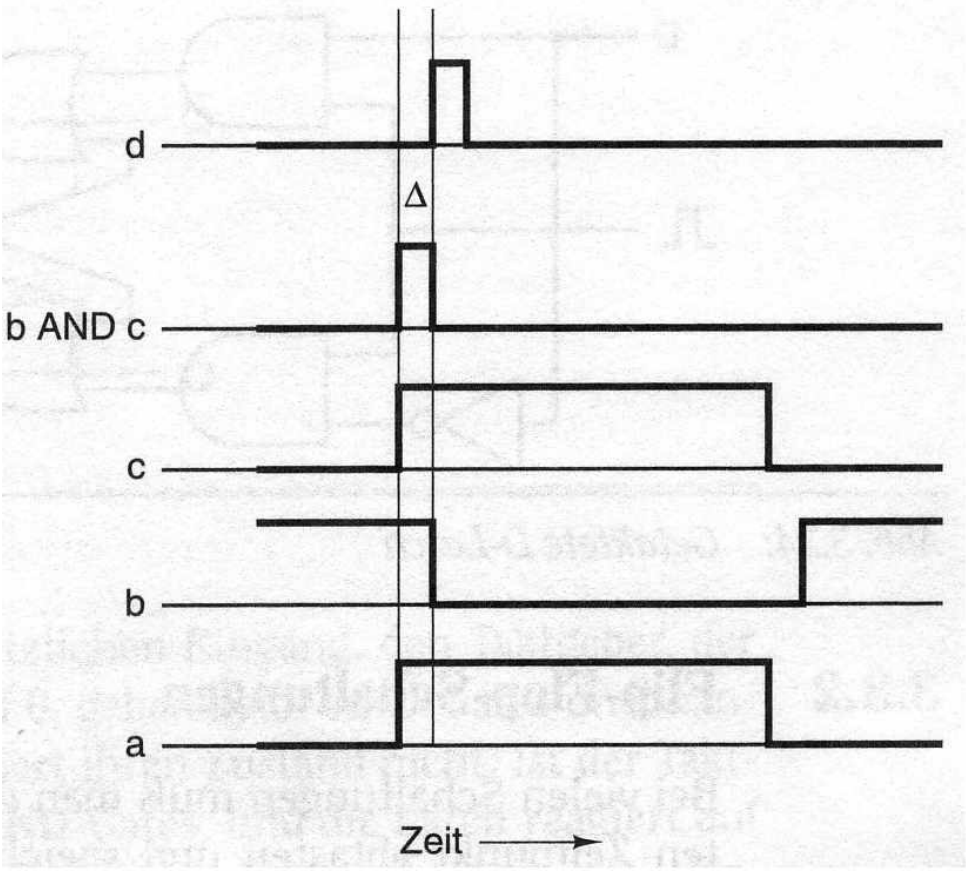
- Erzeugung eines Auslösers (*trigger*) mit kurzem Pegel auf der steigenden Taktflanke des Ausgangstakts.

  Einspeisung des Auslösers in eine getaktete Latch.
- Zweistufige (Master–Slave) Latch, die die an den Eingängen anliegenden Werte übernimmt und trotzdem die an den Ausgängen anliegenden Werte (für eine festgelegte Zeitdauer) hält.

  Realsiert durch zwei pegelgesteuerte Latches, die hintereinander geschaltet sind und im *Gegentakt* arbeiten.

Auslöser (*trigger*)





Erster Anschein. Ausgang des AND-Gatters ist immer 0.

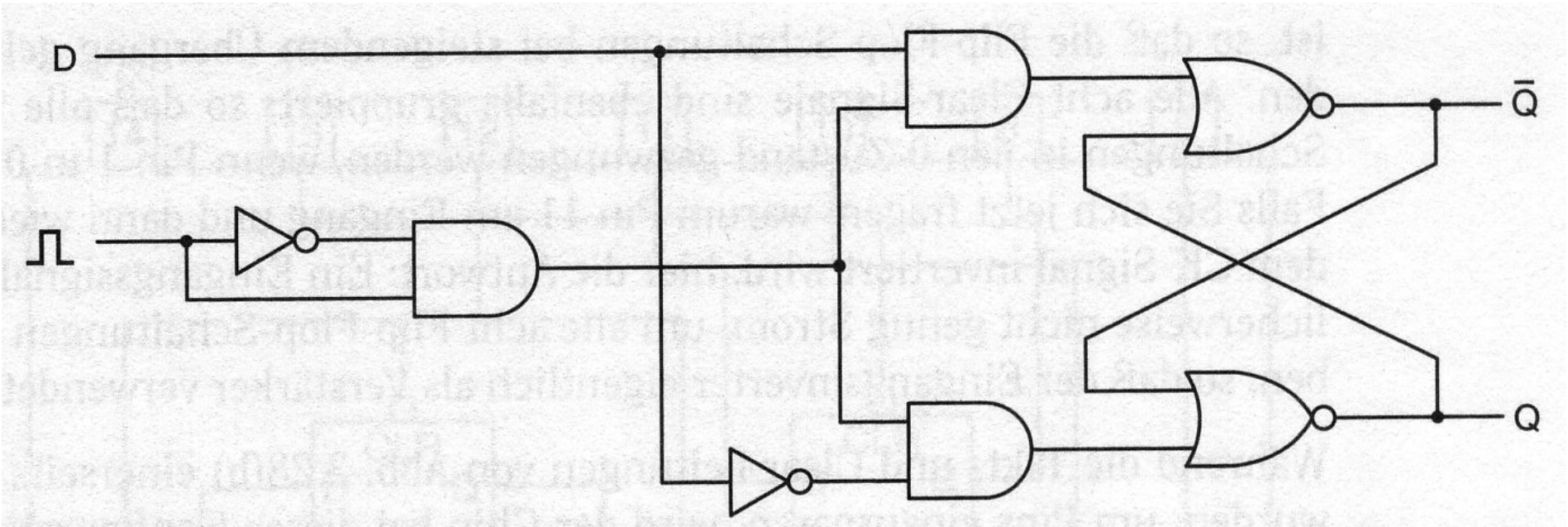
Ausbreitungsverzögerung Δ im NOT-Gatter sorgt für die Funktion der Schaltung.

Die Ausbreitungsverzögerung der Leitungen ist um mehrere Größenordnungen kleiner als die der Gatter und wird ignoriert.

Ausgang des AND-Gatters ist um die Ausbreitungsverzögerung im Gatter verschoben.

Ergebnis. Fast wie gefordert, ein kurzer Pegel kurz nach der steigenden Taktflanke.

D-Latch mit Auslöser

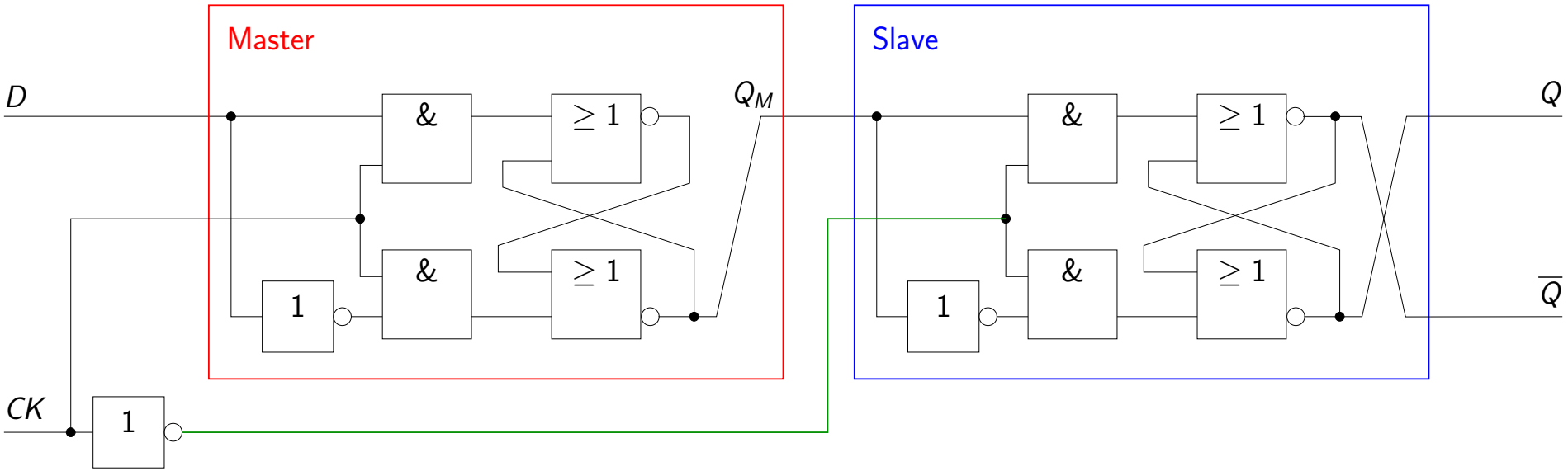


D-Latch mit Auslöser

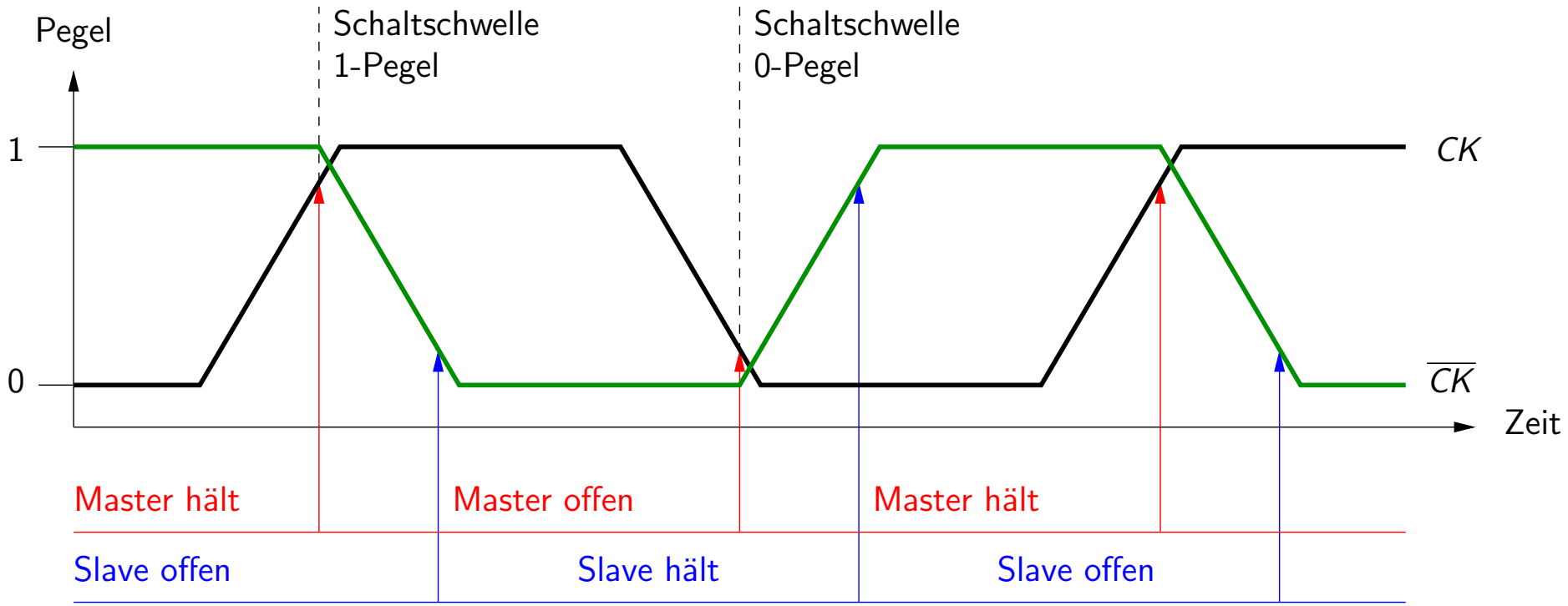
Einfaches Design des Auslöser, aber für die Praxis oft zu einfach.

Master–Slave D-Latch

Eine **Master–Slave Latch**, die an der **fallenden Taktflanke** schaltet, wird realisiert durch zwei pegelgesteuerte Latches, die hintereinander geschaltet sind und im *Gegentakt* arbeiten.



Zustandsänderungen von Master und Slave (1/2)

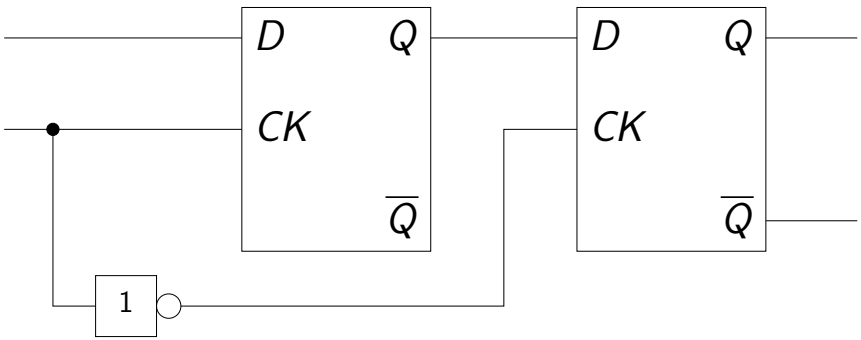


Zustandsänderungen von Master und Slave (2/2)

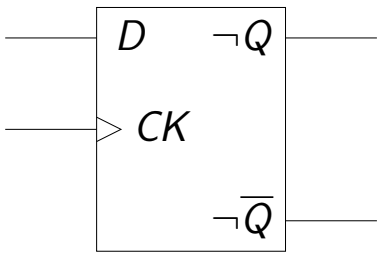
Die während der fallenden Flanke des Takts (CK) am Master anliegende Eingabe, ist - während dieses Taktzyklus - die letzte, die den Zustand des Master (Q_M) setzen kann. Nachdem sich dieser Zustand eingestellt hat, wird er vom offenen Slave übernommen und zum Zustand der Master–Slave Latch (Q).

An der fallenden Flanke des komplementären Takts ($\overline{CK}$) sind Master und Slave gleichzeitig offen. Aber die Zeitspanne vom Öffnen des Masters bis zum Wechsel des Slave auf Halten, reicht nicht aus um den Zustand des Masters (Q_M) zu ändern, d.h. der Slave bekommt vom Master weiterhin dieselbe Eingabe.

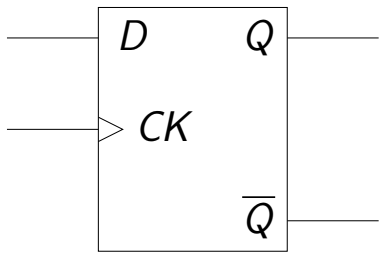
Master–Slave Latches und Flip-Flops



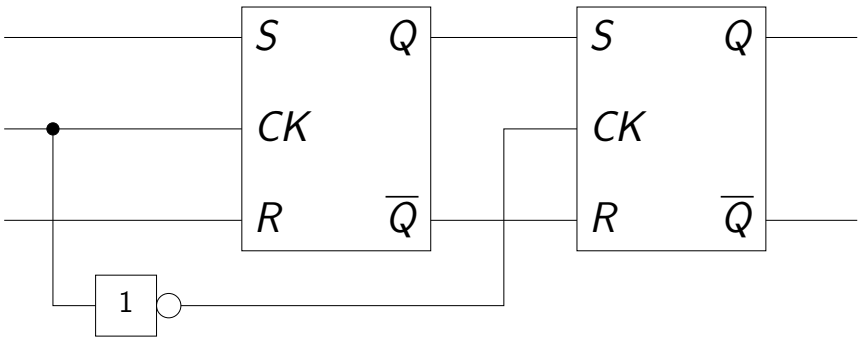
(a)



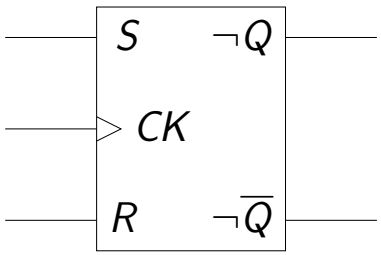
(b)



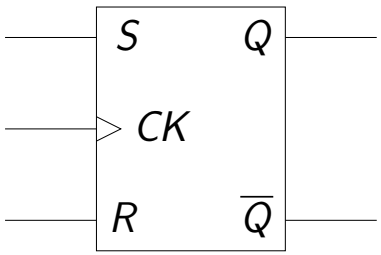
(c)



(d)



(e)



(f)

Master–Slave D-Latch (a) ist D-Flip-Flop (b), schalten an fallender Flanke ($\neg$). D-Flip-Flop (c) schaltet an steigender Taktflanke.

Master–Slave SR-Latch (d) ist SR-Flip-Flop (e), schalten an fallender Flanke ($\neg$). SR-Flip-Flop (f) schaltet an steigender Taktflanke.

Darstellung oft auch ohne $\neg\overline{Q}$ bzw. $\overline{Q}$.

Vielseitigeres Flip-Flop

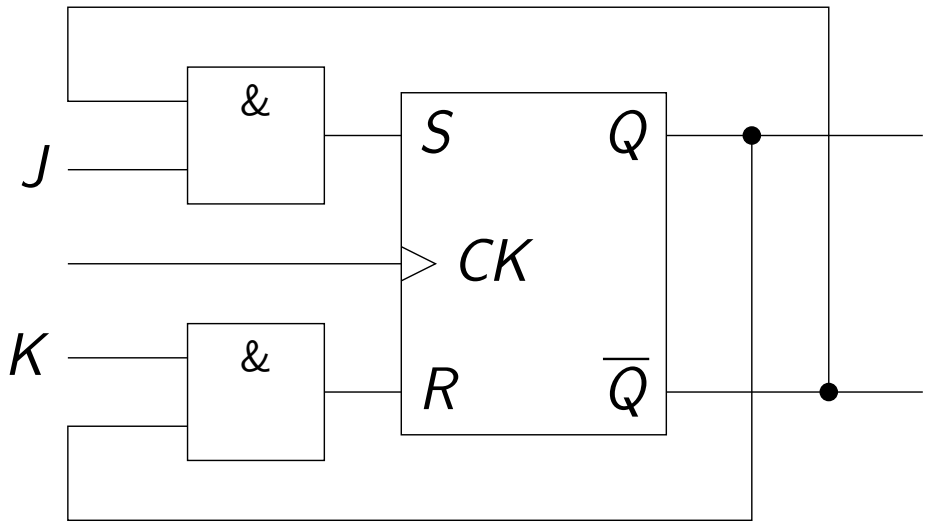
JK-Flip-Flop (*jump and kill*).

- Halten.
- Setzen (*jump*).
- Zurücksetzen (*kill*).
- Invertieren.

Ähnlich SR-Flip-Flop.
Verwendung des Problemfalls (J=K=1) zum Invertieren, sonst (S=J, R=K).

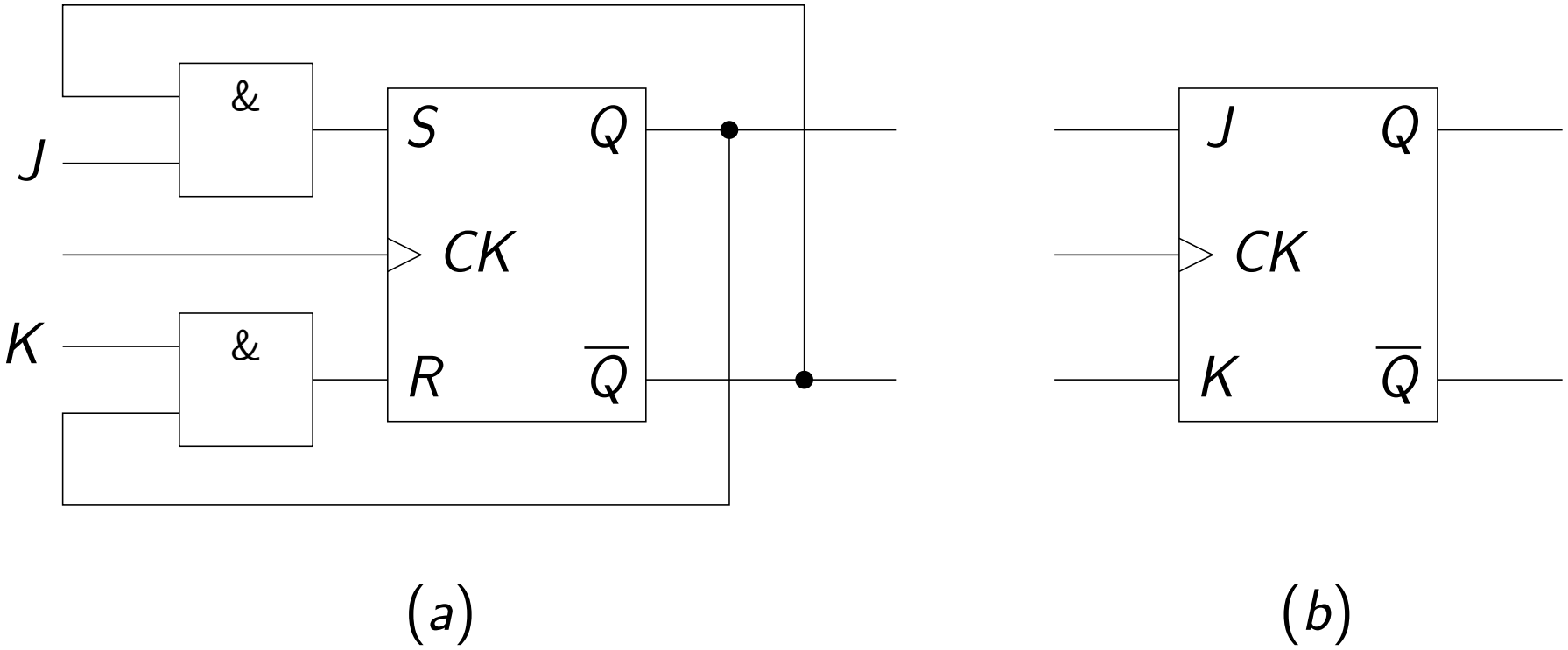
J	K	Q_{neu}	Wirkung
0	0	Q_{alt}	Halten
0	1	0	Zurücksetzen
1	0	1	Setzen
1	1	$\overline{Q_{alt}}$	Invertieren

JK-Flip-Flop



J	K	Q_{alt}	Q_{neu}	Wirkung
0	0	0	0	Halten
0	0	1	1	Halten
0	1	0	0	Zurücksetzen
0	1	1	0	Zurücksetzen
1	0	0	1	Setzen
1	0	1	1	Setzen
1	1	0	1	Invertieren
1	1	1	0	Invertieren

Schaltsymbol des JK-Flip-Flop



(a) (b)

Schaltplan (a) und Schaltsymbol (b) eines JK-Flip-Flops.

Darstellung des Schaltsymbols oft auch ohne $\overline{Q}$.

Umschalt Flip-Flop

T-Flip-Flop (*toggle*).

- Halten.
- Invertieren.

T	Q_{neu}	Wirkung
0	Q_{alt}	Halten
1	$\overline{Q_{alt}}$	Invertieren

Spezialfall des JK-Flip-Flops, (J=K=T).

(*a*) (*b*)

Schaltplan (a) und Schaltsymbol (b) eines T-Flip-Flops.

Hauptanwendungsbereich von T-Flip-Flops sind Zählerschaltungen.

Bevorrechtigte Eingänge

Gezieltes setzen des Flip-Flops auf einen bestimmten Wert

- Initialisieren (*preset*).
- Löschen (*clear*).
- Müssen nicht beide vorhanden sein, Löschen kommt häufiger vor als Initialisieren.

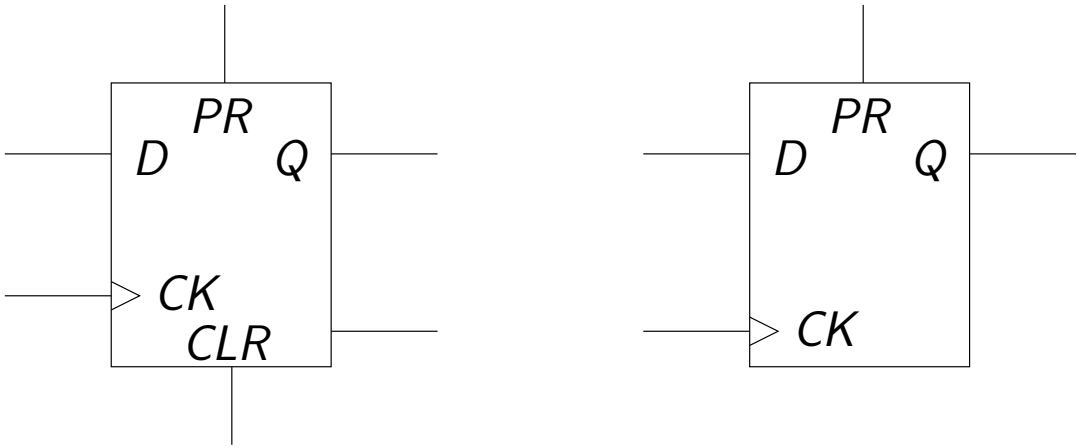
Direkte Wirkung (asynchron).

- Vorteil. Wirkt immer, auch ohne Takt.
- Nachteil. Auch kurze Impulse können Zurücksetzen auslösen (gefährlich!).
- Fazit. Geeignet für generelles Zurücksetzen, nicht geeignet für logische Funktion.

Wirkung mit der Taktflanke (synchron).

- Nicht notwenig. Initialisieren und Löschen entspricht Setzen und Zurücksetzen.

Erweiterter D-Flip-Flop

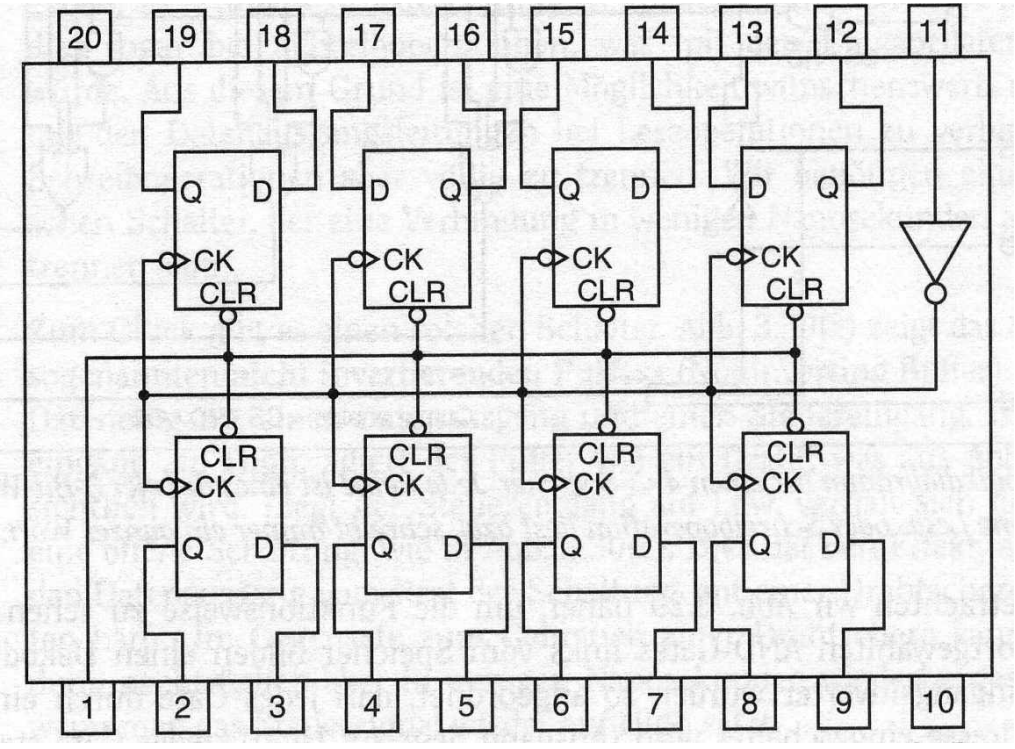


Asynchron, Wirkung unabhängig von *CK*.

- Initialisieren (*preset*) *PR*.
- Löschen (*clear*) *CLR*.

Inhalt

Register



8-Bit Register.

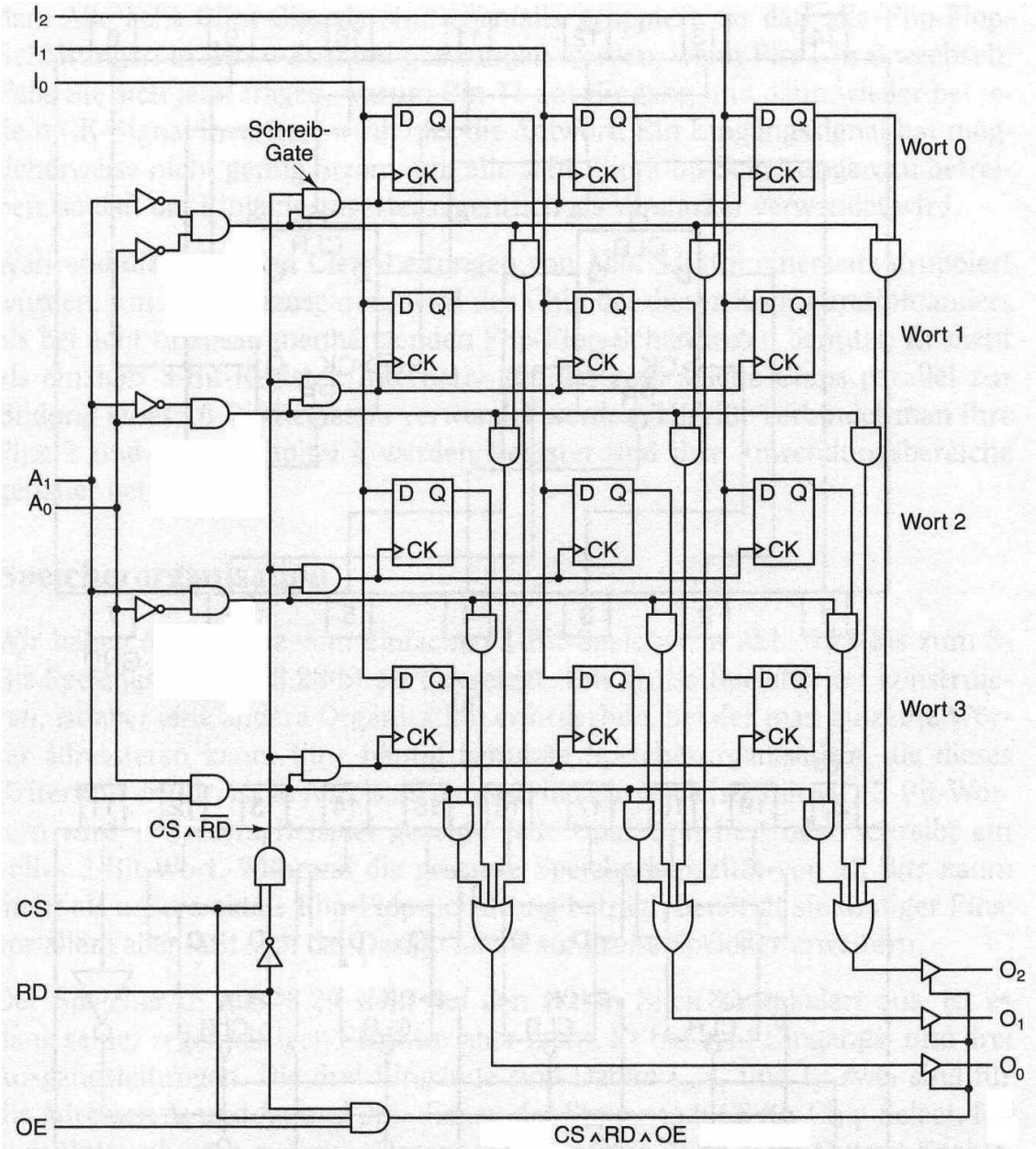
Taktleitungen sind gruppiert und werden von Pin 11 gesteuert.

Der Eingangsinverter wird an allen Flip-Flops wieder invertiert und funktioniert nur als Verstärker.

Leitungen für asynchrones Löschen sind gruppiert und werden von Pin 1 gesteuert.

Zwei solche Chips können ein 16-Bit Register bilden, indem jeweils die beiden Pins 1 und die beiden Pins 11 verbunden werden.

4x3-Speicher



Jede der 4 Reihen speichert ein 3-Bit Wort.

Eine Lese- oder Schreiboperation betrifft immer ein ganzes Wort.

- Adresse A_1, A_0.
- Eingänge I_2, I_1, I_0.
- Ausgänge O_2, O_1, O_0.
- Lesen (*read*) RD.
- Ausgeben (*output enable*) OE.
- Chip aktivieren (*chip select*) CS.
- Wenn $RD = 0$ werden an der steigenden Flanke von CS die Werte der Eingänge gespeichert.

Tri-State Schaltungselement

Eingabe und Ausgabe erreichen den Speicher über die gleichen Leitungen des Datenbusses. Damit es nicht zu Konflikten (Kurzschlüssen, Überlagerungen, etc.) kommt, müssen die Ausgabeleitungen des Speichers von Datenbus getrennt werden, wenn nicht aus dem Speicher gelesen wird.

Bemerkung

Ausgabe 0 (low) ist in der Regel nicht gleichzusetzen mit kein Signal. Abhängig vom Hardwaredesign ist z.B. low=-5V und high=+5V.

Ein **Tri-State** Schaltungselement kann neben low und high noch den Zustand hochohmig (*high impedance*) annehmen.

Schalter

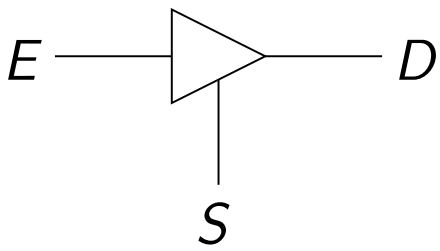
Tri-State Puffer (*tri-state buffer*)

- Dateneingang E
- Datenausgang D
- Steuereingang S

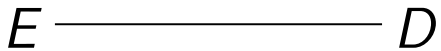
S=1. Dateneingang wird zum Datenausgang durchgeschaltet (D=E).

S=0. Datenausgang ist hochohmig (nicht verbunden, unbestimmt).

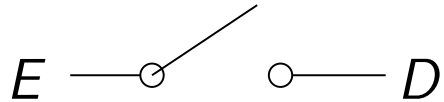
Tri-State Puffer



(*a*)



(*b*)



(*c*)

(a) Schaltsymbol für einen Tri-State Puffer.
(b) Steuereingang gesetzt (S=1).
(c) Steuereingang nicht gesetzt (S=0).

Inhalt

Von-Neumann

Im Rahmen des Baus des *Electronic Discrete Variable Automatic Computers (EDVAC)* beschrieb John von Neumann 1945 ein revolutionäres Konzept.

Die entscheidende Neuerung bestand darin, die Befehle des Programms wie die zu verarbeitenden Daten zu behandeln, sie binär zu kodieren und im internen Speicher zu verarbeiten.

Dieses Konzept wird heute als **Von-Neumann Architektur** bezeichnet.

Konrad Zuse hatte viele Ideen der von-Neumann Architektur schon 1936 ausgearbeitet, 1937 patentiert und 1938 in der Z1 Maschine mechanisch realisiert. Allerdings wird allgemein angenommen, dass von Neumann Zuses Arbeiten nicht kannte.

Die meisten der heute gebräuchlichen Computer basieren auf dem Grundprinzip der Von-Neumann Architektur.

Prinzipien

Grundprinzipien der Von-Neumann Architektur

- Programmsteuerung durch universelle Hardware
- Gemeinsamer Speicher für Daten und Programme
- Hauptspeicher besteht aus adressierbaren Zellen
- Programm besteht aus einer Folge von Befehlen
- Sprünge sind möglich (bedingte und unbedingte)
- Speicherung erfolgt binär

Inhalt

Werke und Busse

Ein Von-Neumann Rechner besteht aus folgenden Komponenten.

1. **Rechenwerk** (*Arithmetic Logic Unit, ALU*). Führt Rechenoperationen und logische Verknüpfungen durch.
2. **Steuerwerk** (*Control Unit*). Interpretiert die Anweisungen eines Programmes und steuert die Befehlsabfolge, auch Leitwerk genannt.
3. **Speicherwerk** (*Memory*). Speichert sowohl Programme als auch Daten, die für das Rechenwerk zugänglich sind.
4. **Eingabe-/Ausgabewerk** (*Input/Output Unit, I/O Unit*). Steuert die Ein- und Ausgabe von Daten, zum Anwender oder zu anderen Systemen.
5. **Bus-System.** Datenbus, Adressbus, Steuerbus. Verbindet die Komponenten des Rechners untereinander (nicht Teil des ursprünglichen Entwurfs).

Hauptprozessor

Rechenwerk.

- Häufig synonym mit ALU (*Arithmetic Logic Unit*) gebraucht.

  Eigentlich ist die ALU eine zentrale **Komponente** des Rechenwerks und ein Rechenwerk kann auch mehrere ALUs enthalten.
- Das Rechenwerk besteht zusätzlich aus einer Reihe von **Registern**.

  Register sind Speicherbereiche im Rechenwerk, die unmittelbar z.B. Operanden oder Ergebnisse von Berechnungen aufnehmen.

  Die ALU selbst enthält keine Register und ist ein reines Schaltnetz.

Steuerwerk + Rechenwerk = Hauptprozessor (*Central Processing Unit, CPU*)

Speicherwerk (1/2)

Der Arbeitsspeicher besitzt einen **eingeschränkten Adressumfang**.

Persistente Speicher wie z.B. Festplatten oder Caches, sowie Register zählt man logisch nicht zum Speicher.

Die Zugriffsgeschwindigkeit zum Arbeitsspeicher sollte der Arbeitsgeschwindigkeit des Hauptprozessors angepasst sein.

Für die verschiedenen Einsatzbereiche der Speicher werden unterschiedliche Speicherarten verwendet, die sich unterscheiden hinsichtlich

- Speichermedium und physikalischem Arbeitsprinzip,
- Organisationsform,
- Zugriffsart,
- Leistungsparameter,
- Preis.

Speicherwerk (2/2)

Haupt- bzw. Arbeitsspeicher für Programme und Daten.

- Speicher mit wahlfreiem Zugriff *(random access memory, RAM)*.

  Jede Speicherzelle kann über ihre Speicheradresse direkt angesprochen werden (wahlfreier Zugriff).

  Der Begriff RAM wird heute im Sinne von Schreib-Lese-Speicher mit wahlfreiem Zugriff (*read-write-RAM*) verwendet.
- Nur-Lese-Speicher *(read only memory, ROM)* ist in der Regel auch Speicher mit wahlfreiem Zugriff.
- Löschbarer, programmierbarer Nur-Lese-Speicher (*Erasable Programmable Read-Only-Memory, EPROM)*.

Ein-/Ausgabewerk

Das **Ein-/Ausgabewerk** steuert die Ein- und Ausgabe von Daten.

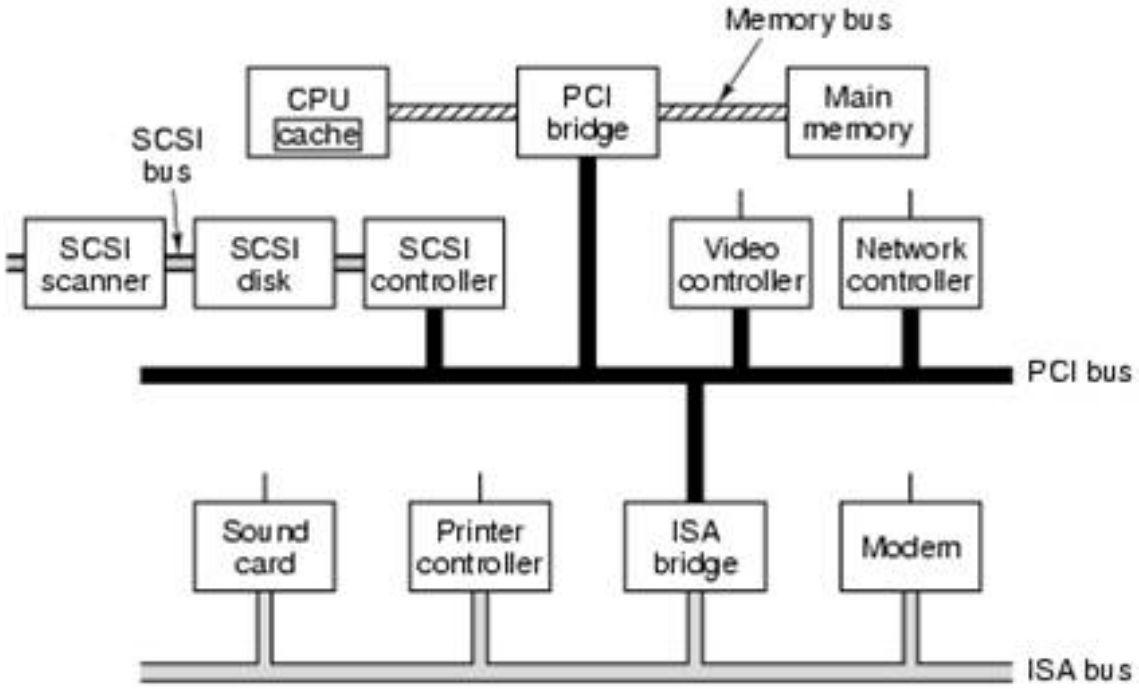
- zum Anwender.
  - Lochkarte
  - Tastatur
  - Maus
  - Scanner
  - Bildschirm
  - Drucker
- zu anderen Systemen (Schnittstellen).
  - Disketten
  - Festplatten
  - Magnetbänder
  - (Netzwerk)

Die Ein-/Ausgabe Geräte sind über das Bus-System mit Speicher und Hauptprozessor verbunden.

Bus-Systeme

Bus-Systeme

- Ein gemeinsam genutztes Medium
- Die Anzahl gleichzeitig übertragbarer Bits ist die Busbreite.
- Busse können durch Brücken hierarchisch sein.

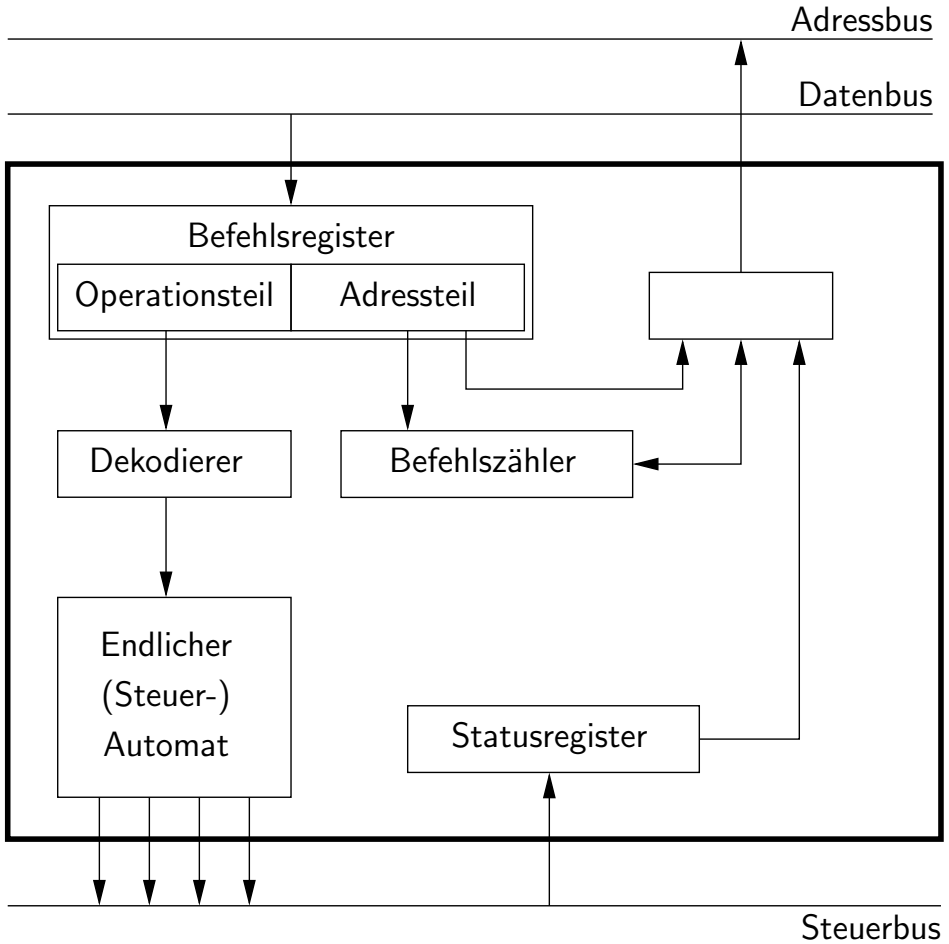


PCI (Peripheral Component Interconnect) Bus.

North Bridge (im Beispiel PCI bridge). Synchronisiert den Datentransfer von und zur CPU. Durch FSB (Front Side Bus) direkt mit der CPU verbunden

South Bridge (im Beispiel ISA bridge). Synchronisiert Datentransfer zwischen peripheren Geräten (Seriell, Audio, USB (Universal Serial Bus), Firewire). Durch PCI-Bus mit North Bridge verbunden.

Steuerwerk (1/2)



Steuerwerk (2/2)

Das Steuerwerk steuert die Arbeitsweise des Rechenwerks durch schrittweise Interpretation der Maschinenbefehle

Der Befehlszähler (*program counter, PC*) enthält die Adresse des nächsten auszuführenden Befehls.
Das Steuerwerk verwaltet den Wert des Befehlszählers.

Das Befehlsregister (*instruction register, IR*) enthält den aktuellen Befehl.

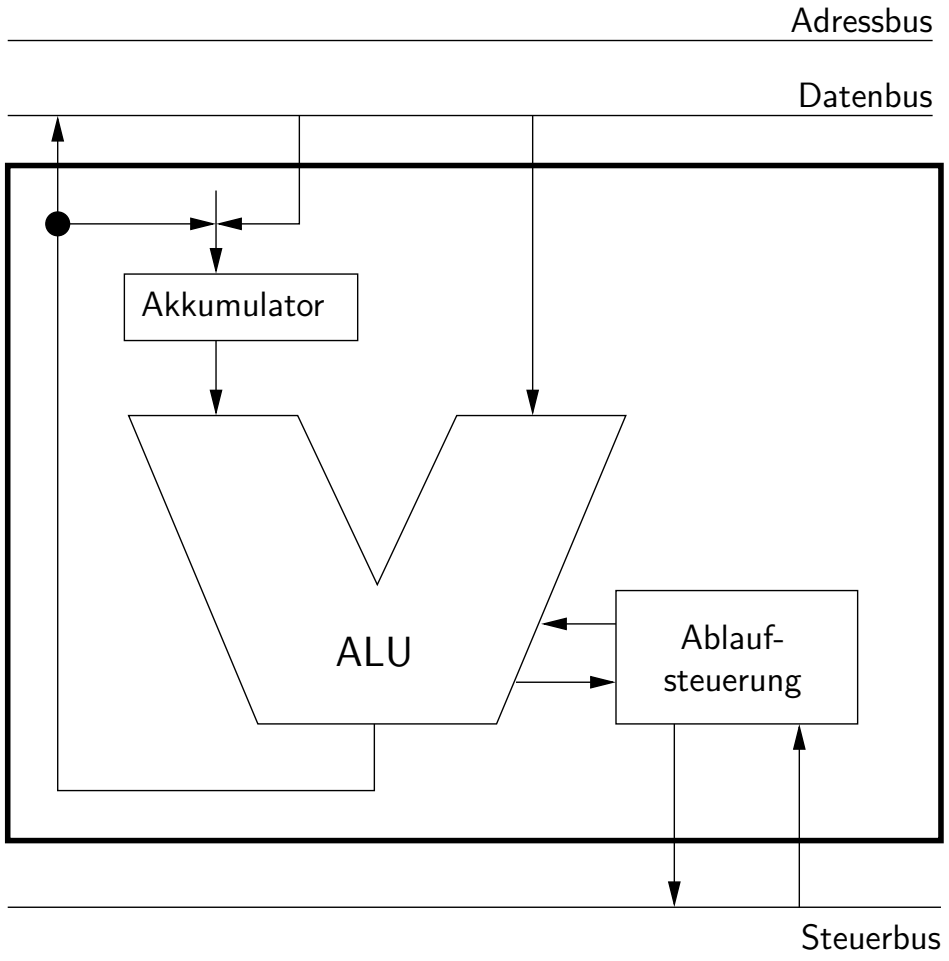
Das Statusregister (*status register, SR*) nimmt Rückmeldungen des Systems auf.

Das Steuerwerk decodiert den Befehl.

- Der Operationsteil (*operation code, opcode*) bestimmt dabei welche Operationen ausgeführt werden sollen.
- Operanden werden durch Angabe von Registern oder Speicheradressen bestimmt.
- Direktoperanden können durch Konstanten angegeben werden.
- Decodierung erfolgt in der Regel durch Mikroprogramme.

Das Steuerwerk erzeugt die nötigen Steuersignale für das Rechenwerk.

Rechenwerk (1/2)



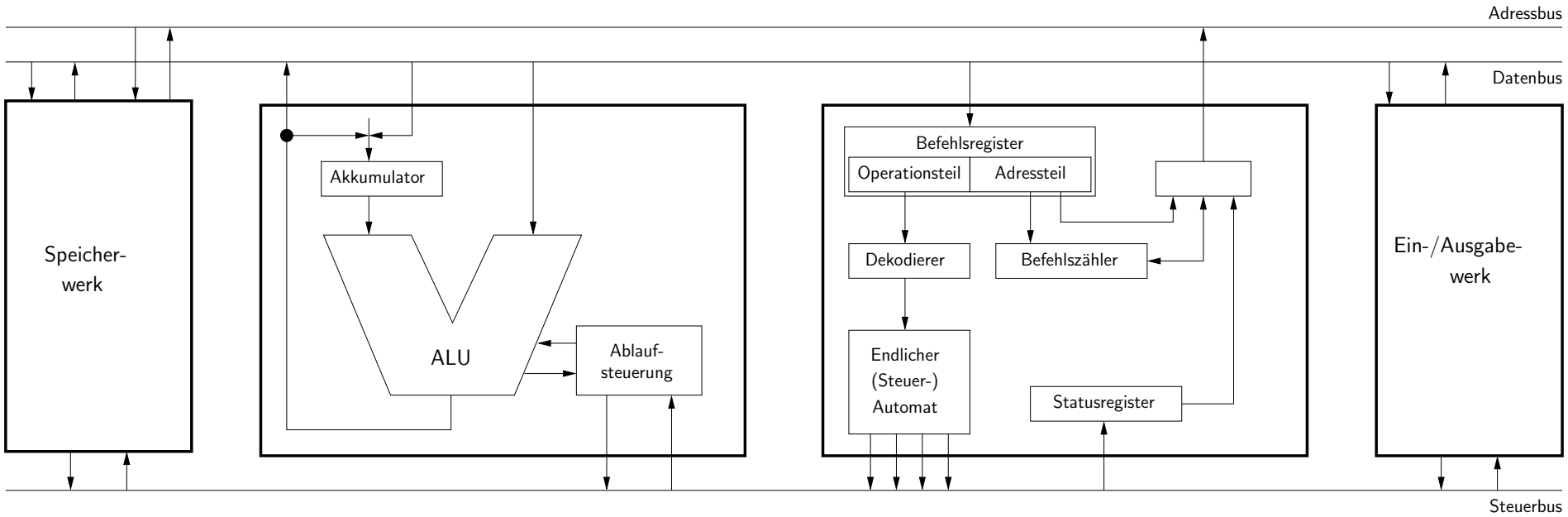
Rechenwerk (2/2)

Das Rechenwerk bildet zusammen mit dem Steuerwerk den Hauptprozessor (CPU).

Es besteht aus

- einer (oder mehreren) ALU und Registern,
- arithmetische Operationen (Addition, Subtraktion, ...),
- logische Operationen (UND, ODER, NICHT, ...),
- Verschiebe-Operationen,
- Bitmanipulation (unter Umtänden),
- Vergleichs- und Bit-Test-Operationen.

Rechneraufbau



Inhalt

Befehlszyklus (1/2)

FETCH	Befehlsholphase
DECODE	Dekodierungsphase
FETCH OPERANDS	Operanden nachladen
EXECUTE	Befehl ausführen
UPDATE PC	Befehlszähler auf den nächsten Befehl zeigen lassen

Befehlszyklus (2/2)

Die Gemeinsame Arbeitsweise von Steuerwerk und Rechenwerk wird durch den Maschinenbefehlszyklus beschrieben.

Der Befehlszyklus wird von der CPU ständig durchlaufen.

Die Befehle stehen im Speicher.

Das Steuerwerk *weiß* jederzeit, welcher Befehl als nächster auszuführen ist.

Die Adresse (= Nummer der Speicherzelle) des nächsten auszuführenden Befehls steht in einem speziellen Register des Steuerwerks, dem Befehlszähler (PC).

Üblicherweise stehen aufeinander folgende Befehle in aufeinander folgenden Speicherzellen, der zuerst auszuführende Befehl hat die niedrigste Adresse.

Zu Beginn des Programms wird der Befehlszähler mit der Startadresse des ersten Befehls geladen.

FETCH

Befehlsholphase.

- Speicherzugriff auf die vom Befehlszähler (PC) angezeigte Adresse.
- Der Befehl wird in das Befehlsregister des Steuerwerks geschrieben.
- Besteht ein Befehl aus mehreren Speicherworten, so setzt sich diese Phase auch aus mehreren Speicherzugriffen zusammen, bis der Befehl vollständig im Befehlsregister steht.
- Das Befehlsregister ist untergliedert in Operationsteil Register (OR) und Adressteil Register (AR).

DECODE, FETCH OPERANDS

Dekodierungsphase.

- Der Befehl im OR wird decodiert (Befehlsdecoder) und der Ablaufsteuerung zugeführt.
- Das Decodieren übernimmt ein Mikroprogramm oder ist hart verdrahtet.
- Die Ablaufsteuerung erzeugt die für die Befehlsausführung nötigen Steuersignale.
- Benötigt der Befehl Operanden, so wird deren Adresse aus dem Inhalt des AR ermittelt.

Operanden nachladen.

- Speicherzugriff auf die ermittelte Operandenadresse(n).

EXECUTE, UPDATE PC

Befehl ausführen.

- Die durch den Operationsteil festgelegten Operationen werden ausgeführt.

Befehlszähler auf den nächsten Befehl zeigen lassen.

- Durch Sprungbefehle oder Prozeduraufrufe kann der Inhalt des PCs verändert werden.

Inhalt

Literatur

Hinterlegt in der Stud.IP Veranstaltung *Informatik II* unter Dateien→MIPS/SPIM.

MIPS32 Architecture For Programmers

Volume I: Introduction to the MIPS32 Architecture

Volume II: The MIPS32 Instruction Set

Volume III: The MIPS32 Privileged Resource Architecture

Appendix A. Assemblers, Linkers, and the SPIM Simulator aus *David A. Patterson und John L. Hennessy.* **Computer Organization and Design: The Hardware/Software Interface.** 3te Auflage, Morgan Kaufmann, 2004.

Reinhard Nitzsche. **Einführung in die Assemblerprogrammierung mit dem MIPS-Simulator SPIM.** 1997/2001.
Gut Einführung, enthält aber kleinere Fehler.

Inhalt

Programm

Was ist ein Programm?

- Eine Reihe von **Befehlen**, die der Ausführung einer Aufgabe dient.
- Dazu wird das Programm sequentiell ausgeführt, d.h. Befehl für Befehl abgearbeitet.
- Der Prozessor arbeitet dabei **zustandsbasiert**.

Problem

- Welche Befehle kennt der Prozessor?
- Die Befehle, die der Computer versteht und ausführen kann, sind Bitfolgen, in der Regel für jeden Hautprozessor unterschiedlich.

  Ausnahme. (Aufwärts-) kompatible Rechnerfamilien, z.B. 286, 386, 486, Pentium.

Befehl

Ein Befehl muss zwei Angaben enthalten, vergleiche Befehlsregister beim der Von-Neumann Architektur.

- Die durchzuführende Operation (*operation code, op-code*). Was wird gemacht.
- Der verwendete Operand (Adresse des Operanden). Womit wird etwas gemacht.

Der Aufbau der Befehle im Detail hängt stark von der verwendeten Hardware ab.

- 8-Bit-Mikrocomputer arbeiten fast immer mit Einadress-Befehlen.

  Die Quelle ist explizit angegeben, das Ziel implizit (z.B. Akkumulator).
- 16-Bit-Hautprozessoren arbeiten oft mit Zweiadress-Befehlen.

  Quelle und Ziel sind explizit angegeben.
- Ein Befehl kann aus einem oder mehreren Speicherworten bestehen.
- Bei Computern mit großer Wortlänge können auch mehrere Befehle in einem Speicherwort stehen.

Maschinensprache und Assemblersprache

Gesamtheit aller von einem Prozessor ausführbaren Befehle (Maschinenbefehle) heißt **Maschinensprache**.

Prozessoren erhalten ihre Instruktionen als Bitfolgen (Maschinencode).

Beispiel

```
0010 0010 0000 1000 0001 0001 0001 0010
```

Bedeutet, abhängig vom Prozessor, addiere den Inhalt der Register $17 = (10001)_2$ und $18 = (10010)_2$ und speichere das Ergebnis im Register $8 = (1000)_2$ ab.

Ein **Assembler** ist ein spezieller Übersetzer, der ein in einer maschinennahen Sprache geschriebenes Programm in Maschinencode übersetzt.

Der Wortschatz eines Assembler heißt **Assemblersprache** oder auch einfach Assembler

Beispiel

```
add    $8, $17, $18
```

Assembler

Programme in Assembler sind schwer zu erstellen bzw. zu verstehen, selbst wenn nicht direkt die binären Maschinensprache benutzen wird.

- Viele Befehle stehen nicht direkt zur Verfügung.
- Die Speicherverwaltung muss selbst übernommen werden.
- Zur Ablaufsteuerung gibt es nur Sprünge.
- Verwaltung von Unterprogrammen ist nicht vorgesehen.

Zusammenfassend. Der Programmierer muss sich um **alles** selber kümmern.

Jeder Prozessor ist anders, somit auch jeder Assembler. Das bedeutet, eine Problemlösung in Assembler muss für jeden Prozessor neu programmiert werden.

Assembler und Hochsprachen

Nächster Schritt, eine **Hochsprache**.

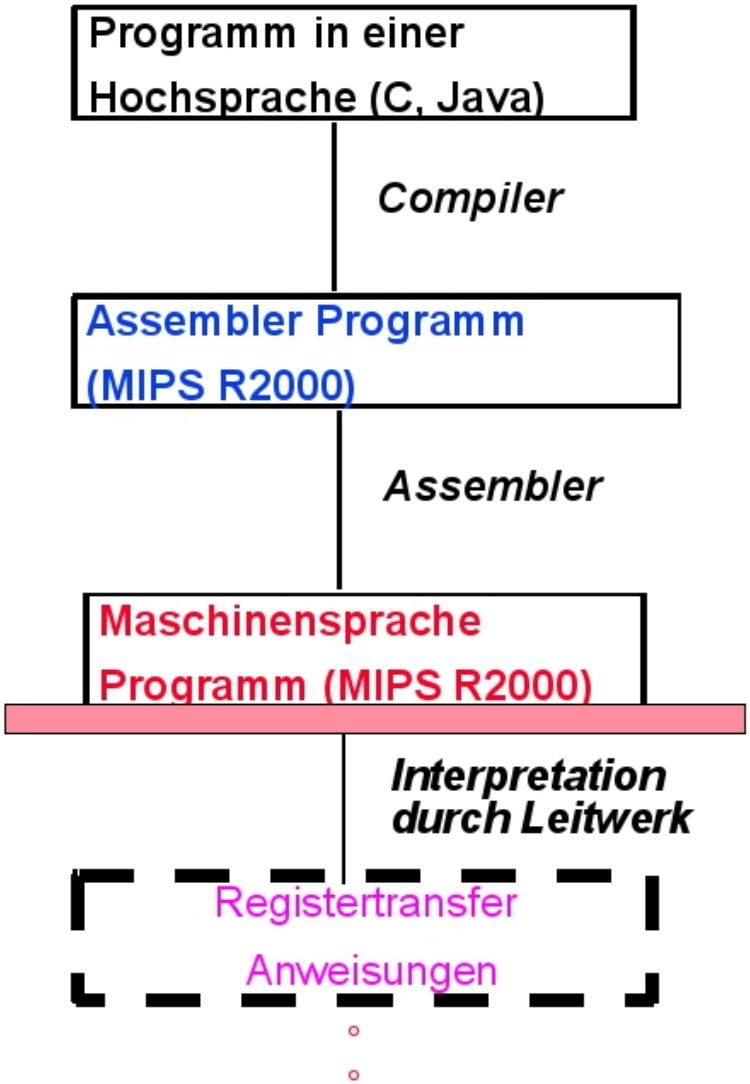
- Es werden mehr Befehle angeboten.
- Die Speicherverwaltung wird übernommen.
- Komfortable Ablaufsteuerung.
- Verwaltung von Unterprogrammen (Funktionen).
- (Viele) Befehle sind unabhängig vom Prozessor.

Wer vermittelt zwischen Hochsprache und Maschinensprache?

- **Compiler** übersetzt das gesamte Hochsprachenprogramm in ein Assemblerprogramm (Kompilieren).

  Assembler übersetzt das Assemblerprogramm in ein Maschinensprachenprogramm (Assemblieren).
- **Interpreter** erzeugt erst zur Laufzeit Maschinencode (Interpretieren).

Visualisierung




```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw  $15,  0($2)
lw  $16,  4($2)
sw  $16,  0($2)
sw  $15,  4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

IR ← Imem[PC]; PC ← PC + 4

ALUOP[0:3] ⇐ InstReg[9:11] & MASK

Inhalt

Prozessor-Architekturen (1/2)

RISC(*Reduced Instruction Set Computing*)

- Die Befehle besitzen einen einheitlichen Befehlsaufbau, daraus folgt niedriger Dekodierungsaufwand.
- Jeder Befehl kann einfach (in einem Taktzyklus) ausgeführt werden.
- Die (meisten) Befehle sind fest verdrahtet, d.h. eine Operation wird durch tatsächliche Leiterbahnen auf dem Prozessor repräsentiert.
- Maschinencode ist eine Abfolge kurzer Befehle, d.h. der Prozessor kann mit kleiner Verzögerung reagieren, wenn das aktuelle Programm unterbrochen werden soll (Stichwort Interrupt).

Prozessor-Architekturen (2/2)

CISC (*Complex Instruction Set Computing*)

- Großer Befehlsumfang.
- Komplexe Adressierungsmöglichkeiten.
- Jeder Befehl ist ein eigenes **Mikroprogramm** im ROM des CISC-Prozessors.

  Das Mikroprogramm wird von einem Prozessor (Nanoprozessor) im Prozessor verarbeitet, der den Mikrocode in seinen komplexen Schaltkreisen ausführt (u.U. in mehreren Arbeitsschritten).

  Das Mikroprogramm wird komplett ausgeführt, solange ist keine Unterbrechung des aktuelle Programms möglich (Stichwort Interrupt).

RISC und CISC

RISC-Prozessoren

- SPARC von Sun Microsystems.
- Alpha von Digital Equipment.

CISC-Prozessoren

- 808x und 80x86 von Intel.
- 680x0 von Motorola.

Aktuellen Prozessoren werden nicht mehr als reine RISC- oder CISC-Architektur entwickelt.

Die Prozessoren 80486, Pentium und 68060 sind CISC-Prozessoren, die mit Elementen von RISC-Prozessoren ausgestattet sind.

Die PowerPC RISC-Prozessoren werden durch eine Befehlserweiterung ergänzt, die die Prozessoren mit spezielle Multimediafähigkeiten nachrüstet.

Inhalt

Pipeline

Um die Arbeitsgeschwindigkeit zu steigern wird, vorallem bei RISC-Prozessoren, eine (Befehls-) **Pipeline** (ein Fließband) verwendet.

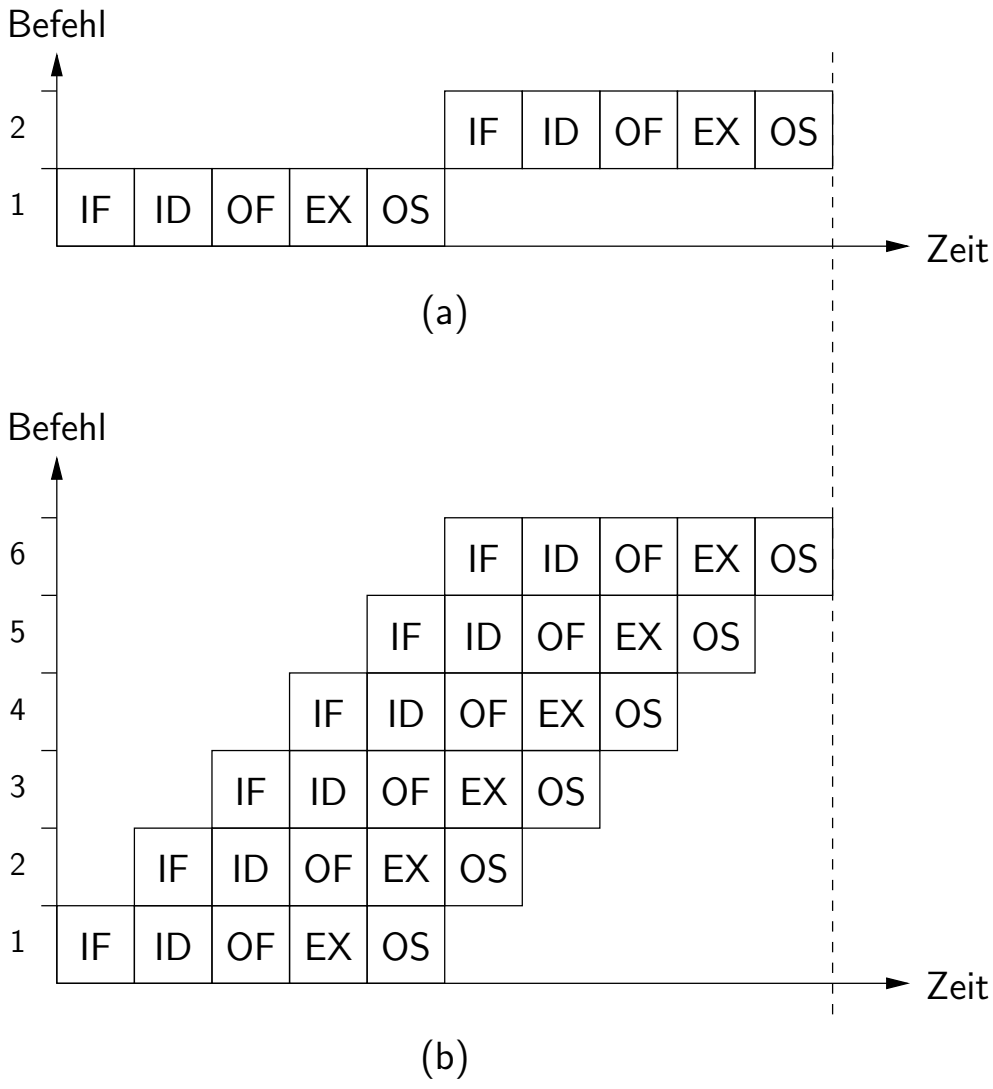
Ein Maschinenbefehl wird in mehrere Phasen zerlegt, den **Befehlszyklus**.

Verschiedene Phasen mehrerer Befehle werden von unterschiedlichen Teilen des Prozessors **parallel** verarbeitet.

Der Prozessor ist entsprechend in mehreren **Pipeline-Stufen** (*pipeline stages*) konstruiert.

Insgesamt benötigt ein einzelner Befehl nun mehrere Takte zur Ausführung, aber durch die parallele Bearbeitung mehrerer Befehle wird in jedem Taktzyklus die Abarbeitung eines Befehls abgeschlossen, somit erhöht sich der Gesamtdurchsatz durch dieses Verfahren.

Phasen-Pipelining



(a) Ohne Pipelining.
(b) Phasen-Pipelining.

Leicht veränderter Befehlszyklus gegenüber der von-Neumann Architektur.

IF Befehlsholphase (*instruction fetch*)
ID Dekodierungsphase (*instruction decode*)
OF Operanden laden (*operands fetch*)
EX Befehl ausführen (*execute operation*)
OS Operanden speichern (*operands store*)

Pipeline-Konflikte

Eine **Abhängigkeit** ist gegeben, wenn für die Bearbeitung eines Befehls in einer Stufe der Pipeline, notwendigerweise ein anderer Befehl, der sich weiter vorne in der Pipeline befindet, zuerst abgearbeitet werden muss.

Abhängigkeiten können zu Pipeline-Konflikten (*pipeline hazards*) führen.

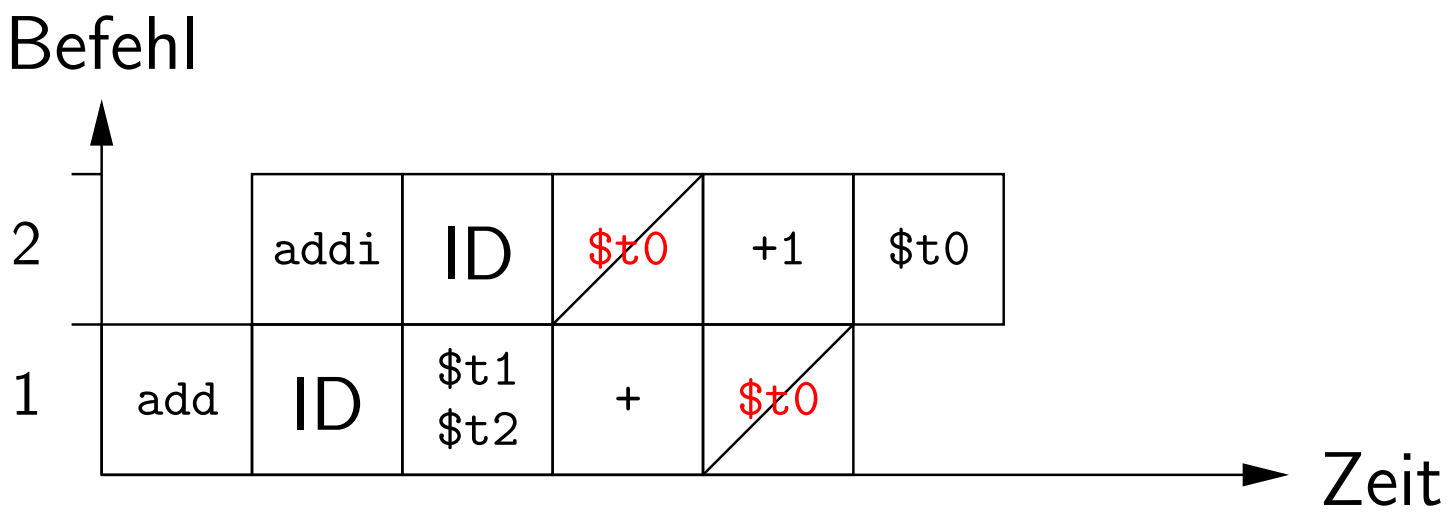
- **Ressourcenkonflikte.** Eine Stufe der Pipeline benötigt Zugriff auf eine Ressource, die bereits von einer anderen Stufe belegt ist
- **Datenkonflikte.**
  - **Befehlsebene.** Daten, die in einem Befehl benutzt werden sollen, stehen nicht zur Verfügung
  - **Transferebene.** Registerinhalte, die in einer Stufe benutzt werden sollen, stehen nicht zur Verfügung
- **Kontrollflusskonflikte.** Die Pipeline muss abwarten, ob ein bedingter Sprung ausgeführt wird oder nicht.

Beispiel

Beispiel.

Datenkonflikte auf Transferebene.

```
add  $t0, $t1, $t2        # $t0 = $t1 + $t2
addi $t0, 1               # $t0 = $t0 + 1
```



Pipeline-Konflikte auflösen

Sperren (*locks*).

Der nachfolgende Befehl wird angehalten bis die Abhängigkeit aufgelöst ist.

Weiterleiten (*forwarding*) zum Auflösen von Datenkonflikten.

Die für den folgenden Befehl benötigten Abhängigkeiten wird an diesen weitergeleitet, statt am Ende des aktuellen Befehls abgespeichert zu werden.

Verzögertes Laden (*delayed load*) zum Auflösen von Datenkonflikten.

Das Laden der Operanden findet verzögert statt (um eine oder mehrere Phasen).

Verzögerter Sprung (*delayed branch*) zum Auflösen von Kontrollflusskonflikten.

Die Verzweigung wird erst (eine oder mehrere Phasen) später wirksam (automatisch oder manuell).

Bemerkung. Wird ein verzögerter Sprung manuell vorgesehen, muss das *Loch* nicht umbedingt `nop` (*no operation*) sein, sondern kann zur Optimierung, wenn keine Abhängigkeiten vorliegen, durch einen Befehl gestopft werden, der vor dem Sprung hätte ausgeführt werden sollen. Für einen Programmierer lästig (unmöglich), aber für optimierende Compiler kein Problem.

Inhalt

Load/Store Design

Grundsätzlich ist die Ausführung von Operation (Berechnungen) auf Werten im Hauptspeicher aufwendiger als auf Werten in Registern.

Der Grund dafür ist, dass Zugriffe (lesend/schreibend) auf Register viel schneller sind als auf Speicherstellen im Hauptspeicher.

Viele (RISC-) Prozessoren benutzen ein **Load/Store Design**, dass entwickelt wurde um die aus den längeren Zugriffszeiten resultierende längere Bearbeitungszeit von Operation auf Werten im Hauptspeicher zu vermeiden.

- Es gibt viele Register.
- Hauptspeicher wird nur über Laden- und Speicherbefehle (*load/store*) angesprochen, die Werte aus dem Speicher in Register laden oder aus Registern im Speicher ablegen.
- Alle anderen Operationen werden nur auf Werten in Registern ausgeführt.

Zusätzliche Vorteile.

- Die Anzahl der Speicherzugriffe wird reduziert.
- Die Befehle werden einfacher/weniger.

Inhalt

MIPS Architektur

MIPS (*Microprocessor without Interlocked Pipeline Stages*) Architektur

- RISC-Prozessor.
- Kein Sperren der Pipeline-Stufen, Konflikte müssen anders aufgelöst werden.

  Es gibt verschieden Variationen von Befehlszyklen und Pipelines.
- Load/Store Design.
- Entwickelt an der Stanford-Universität.
- Silicon Graphics, Inc. (SGI) baute bis 2006 Hochleistungsrechner auf Basis von 64-Bit MIPS Prozessoren.

  Heute werden MIPS Prozessoren häufig in eingebetteten Systemen (Routern, Spielekonsolen, etc.) eingesetzt.

Virtuelle Maschine des MIPS Assemblers

Es ist schwierig direkt auf der MIPS Architektur zu programmieren.

Es gibt unterschiedliche Pipelines, viele Möglichkeiten für Pipeline-Konflikte und deren Auflösung (*delayed branches*, *delayed loads*, etc.).

Der MIPS Assembler kann die Komplexität der Architektur hinter einer **virtuellen Maschine** verstecken.

Programmiert man Quelltext für die virtuelle Maschine muss man sich nicht um Pipeline-Konflikte kümmern und erhält zusätzliche (Pseudo-) Befehle (*pseudo instructions*), die man wie echte Befehle benutzen kann.

Beispiel

Die MIPS Hardware stellt nur Befehle für bedingte Sprünge bereit, die in Abhängigkeit davon, ob eine Register Null ist oder nicht, ausgeführt werden.

Die virtuelle Maschine stellt auch bedingte Sprünge bereit, die in Abhängigkeit verschiedener Vergleich auf zwei Registern, ausgeführt werden.

Inhalt

SPIM Simulator

SPIM ist ein Simulator für die Assemblersprache des MIPS R2000 (MIPS32).

- `http://spimsimulator.sourceforge.net/`
- Simuliert die virtuellen Maschine des MIPS Assemblers.
- Unterliegt der BSD-Lizenz.
- `spim`, Linux Kommandozeilenversion.
- `QtSpim`, graphische Benutzerschnittstelle für Linux und Windows.

QtSpim und Linux

Unter Linux gibt es ein Problem mit QtSpim und der *locale* LC_NUMERIC, diese muss so gesetzt sein, dass als Dezimaltrennzeichen der Punkt (.) und nicht das Komma (,) benutzt wird.

Beispiel

Richtig

```
> locale
...
LC_NUMERIC="en_US.UTF-8"
...
```

Falsch

```
> locale
...
LC_NUMERIC="de_DE.UTF-8"
...
```

Hauptspeicher

Rechner mit von-Neumann Architektur haben gemeinsamen Speicher für Daten und Programme.

Probleme

- Abgelegten Daten dürfen nicht als Programmcode interpretiert werden.
- Programmcode darf nicht mit Daten überschreiben werden.
- Selbstmodifizierender Code ist möglich.
  Beispiel
  - Positiv, Evolutionäre Algorithmen.
  - Negativ, Computer Viren.

Lösung

- **Segmentierung.** Segmente für Daten, Programmcode, Stack und Betriebsystem mit festgelegten Adressbereichen und vorgegebener Dynamik.
- Segmente sind indirekt adressierbar (z.B. Beginn des Datensegments +4).
- Oft bleiben die Speicherstelle auch direkt adressierbar.

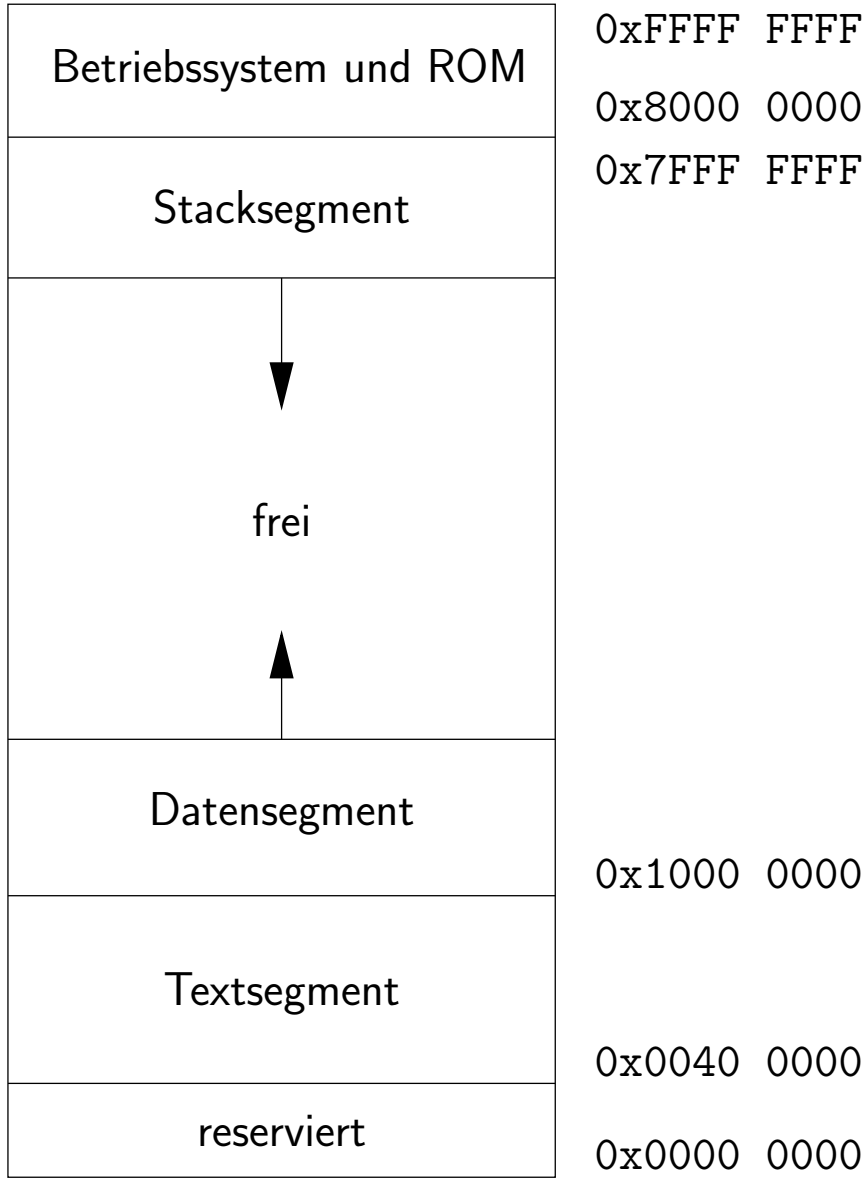
MIPS R2000 Hauptspeicher

Speicher

- 2^{32} Speicherzellen zu einem Byte.
- Mit einer 32-Bit Adresse kann jede Speicherstelle angesprochen werden.
- Datensegment wächst in Richtung größeren Adressen.
- Stacksegment wächst in Richtung kleinerer Adressen.

  Vorsicht. Die Ausrichtung der Daten ist verändert sich nicht.

  Wird z.B. ein Wort der Länge 32-Bit als erstes in das Stacksegment geschrieben, muss es an Adresse `0x7FFF FFFC` geschrieben werden.



MIPS R2000 Register (1/3)

Nr.	*Name*	*Vereinbarte Nutzung*	*Kommentar*
0	`$zero`		konstant 0
1	`$at`	Assembler-Register	reserviert für den Assembler
2	`$v0`	Rückgabewerte von	
3	`$v1`	Unterprogrammen, Zwischenergebnisse (*value*)	
4	`$a0`	Argumente für	
5	`$a1`	Unterprogramme	
6	`$a2`	(*argument*)	
7	`$a3`		
8	`$t0`	temporäre Variablen	können von Unterprogrammen
9	`$t1`	(*temporary*)	verändert werden
10	`$t2`		
11	`$t3`		
12	`$t4`		
13	`$t5`		
14	`$t6`		
15	`$t7`		

MIPS R2000 Register (2/3)

Nr.	*Name*	*Vereinbarte Nutzung*	*Kommentar*
16	`$s0`	langlebige Variablen	werden von Unterprogrammen
17	`$s1`	(*saved temporary*)	nicht verändert
18	`$s2`		
19	`$s3`		
20	`$s4`		
21	`$s5`		
22	`$s6`		
23	`$s7`		
24	`$t8`	temporäre Variablen	können von Unterprogrammen
25	`$t9`	(*temporary*)	verändert werden

MIPS R2000 Register (3/3)

Nr.	*Name*	*Vereinbarte Nutzung*	*Kommentar*
26 27	`$k0` `$k1`	Kernel-Register (*kernel*)	reserviert für das Betriebssystem, z.B. bei Unterbrechungen
28	`$gp`	Zeiger auf Datensegment (*global pointer*)	Spezielle Adresse zur direkten Adressierung von Speicherzellen im Datensegment (*static data*)
29	`$sp`	Zeiger auf den Stack (*stack pointer*)	Adresse des ersten **freien** Wortes im Stacksegment
30	`$fp`	Zeiger auf den Frame (*frame pointer*)	Adresse des ersten Wortes des aktuellen Rahmens im Stacksegment
31	`$ra`	Rücksprungadresse (*return address*)	enthält nach Aufruf z.B. des Befehls `jal` die Adresse der nachstehenden Codezeile

Assembler Direktiven

Assembler **Direktiven** sind Anweisungen im Quelltext, die festlegen wie der Assembler das Programm übersetzt.

Direktiven selbst werden **nicht** in Maschinencode übersetzt.

Direktiven beginnen immer mit einem Punkt (`.`).

In einem Programm müssen Daten und Programmcode in separaten Speicherbereichen untergebracht werden.

- Datendefinition gehören ins Datensegment (*data segment*), z.B. Reservierung von Speicherplatz.
- Befehlsfolgen gehören ins Textsegment (*text segment*).

Die Zuordnung in ein Segment erfolgt implizit über die gerade aktive Segmentangabe, die mit Hilfe von Direktiven gesetzt werden kann.

- **`.data`** setzt die aktive Segmentangabe auf Datensegment.
- **`.text`** setzt die aktive Segmentangabe auf Textsegment.

Assembler Marken

Assembler **Marken** (*label*) dienen der Verwaltung symbolischer Adressen.

Marken sind Zeichenketten gefolgt von einem Doppelpunkt (`:`).

In Markennamen sind auch Ziffern (nicht als erstes Zeichen) und der Unterstrich (_) erlaubt. **Wichtig.** Marken dürfen nicht genauso heißen wie ein Befehl.

Marken werden vom Assembler so verwaltet, dass von anderen Stellen im Programm her auf diese Marken Bezug genommen werden kann.

Marken sind äußerst nützlich bei Sprunganweisungen, bei denen der Programmierer andernfalls erst ermitteln müsste, an welche Stelle (oder über wie viele Speicherworte) gesprungen werden soll.

Variablen werden durch Marken realisiert.

Eine Programmzeile kann auch nur eine Marke enthalten, sinnvoll im Textsegment zur Gliederung des Programms.

Die meisten Assembler erlauben auch die Bezugnahme auf Marken, die gar nicht in der Datei selber definiert wurden. Dies ermöglicht die Nutzung bereits vorhandener Bibliotheken.

Assembler Befehlszeile

In jede Zeile eines Programms kann nur eine einzige Anweisung (Befehl, Direktive, etc.) geschrieben werden.

```
<label>: <instruction> <arg1>  <arg2>  <arg3>  #<Kommentar>
<label>: <instruction> <arg1>, <arg2>, <arg3>  #<Kommentar>
```

Die Argumente einer Anweisung können durch Kommata, Leerzeichen oder Tabulatoren getrennt werden.

Die Anzahl der Argumente hängt vom der Anweisung ab.

Die Angabe von Marken ist optional.

Alle Zeichen vom Kommentarzeichen **#** an werden als Kommentar gewertet und überlesen.

Ganze Zahlen

Ganze Zahlen

`.word`	32-Bit-Zahlen
`.half`	16-Bit-Zahlen
`.byte`	8-Bit-Zahlen

Diese Direktiven legen die nachfolgenden Zahlen im Hauptspeicher ab.

Die Werte können dezimal oder hexadezimal angegeben werden. Hexadezimale Werte beginen mit **`0x`**.

Um auf die im Hauptspeicher abgelegten Werte komfortabel zugreifen zu können, sollte zudem eine Marke vor die Direktive gesetzt werden.

Beispiel

```
values: .word 256 0x100
```

Diese Direktive legt in den beiden nächsten unbenutzten 32-Bit-Wörter die Werte **256** und **0x100** (= **256**) ab.

Die Marke **values** enthält die Adresse der ersten Speicherspeicherstelle, in der die Werte abgelegt sind.

Zeichenketten

Zeichenketten werden mit 8-Bit pro Zeichen kodiert, d.h. es können 256 verschiedene Zeichen dargestellt werden.

Zeichenketten werden mit der Direktive **`.asciiz`** in den Speicher geschrieben und in doppelte Anführungszeichen eingeschlossen.

Beispiel

```
hello: .asciiz "Hello World"
```

Der Assembler kodiert die einzelnen Zeichen des Textes kodiert und setzt am Ende des Textes wird das Zeichen mit dem Code 0 (Nullzeichen), damit, z.B. bei der Textausgabe, das Ende des Textes erkannt werden kann.

Die Direktive **`.ascii`** hängt das Nullzeichen nicht an. Das ist hilfreich, wenn längere Texte im Assemblerprogramm auf mehrere Zeilen verteilt werden müssen.

Beispiel

```
beware: .ascii  "Beware of the doctor, whose wife sells cemetery "
        .ascii  "plots, whose brother owns a granite quarry, "
        .asciiz "and whose father deals in shovels."
```

Sonderzeichen

Um besondere Zeichen im Speicher abzulegen, können folgende Zeichenkombinationen verwendet werden.

`\n`	Neue Zeile.
`\t`	Sprung zum nächsten Tabulator.
`\"`	Doppeltes Anführungszeichen.

Daten im Datensegment

Die Direktive zum Ablegen von ganzen Zahlen und Zeichenketten sollen **nur im Datensegment** (nach der **`.data`**-Direktive) verwendet werden.

Viele Befehle arbeiten nur auf ausgerichteten Daten (*aligned data*).

Ein Datum ist ausgerichtet, wenn seine Hauptspeicheradresse ein ganzzahliges Vielfaches seiner Größe ist.

Beispiel

- Ein Halbwort **`.half`** (2 Byte Größe) ist ausgerichtet, wenn es an einer geraden Hauptspeicheradresseist liegt.
- Ein Wort **`.word`** (4 Byte Größe) ist ausgerichtet, wenn es an einer durch vier teilbaren Hauptspeicheradresse liegt.

Mit der Direktive **`.align`** kann man die Ausrichtung beeinflussen.

Glücklicherweise legen die Direktive **`.word`**, **`.half`** etc. die Daten stets richtig ausgerichtet ab.

Befehle

Befehl	Argumente	Wirkung	Kommentar
`add`	`Rd, Rs1, Rs2`	Rd = Rs1 + Rs2	addition
`lw`	`Rd, Addr`	Rd = MEM[Addr]	load word
`sw`	`Rs, Addr`	MEM[Addr] = Rs	store word
`li`	`Rd, Imm`	Rd = Imm	load immediate

Beispiel

```
1  #
2  # compute the perimeter of a triangle with edges x, y, z
3  #
4  .data
5  x:      .word  12
6  y:      .word  14
7  z:      .word   5
8  u:      .word   0
9  .text
10 main:   lw   $t0, x
11         lw   $t1, y
12         lw   $t2, z
13         add  $t0, $t0, $t1   # $t0 = x + y
14         add  $t0, $t0, $t2   # $t0 = x + y + z
15         sw   $t0, u          # u = x + y + z
16         li   $v0, 10         # EXIT
17         syscall
```

Quelle: Nitzsche, Einführung in die Assemblerprogrammierung mit dem MIPS-Simulator SPIM.

Betriebssytem

SPIM stellt ein **Betriebssystem** zur Verfügung.

Das Betriebssystem startet ein Programm, indem es die Marke **`main`** anspringt, deshalb muss diese Marke immer vorhanden sein.

Einfache Ein- und Ausgaben können mit dem Betriebssystem realisiert werden können.

Die **Betriebssystemfunktionen** (*system calls*) werden mit dem Befehl **`syscall`** aufgerufen.

Der Befehl **`syscall`** hat keine Argumente, die auszuführende Funktion wird von dem Wert im Register **`$v0`** bestimmt und von der Funktion hängt ab, welche weiteren Register ausgewertet oder belegt werden.

Ausgabe

`$v0`	Systemfunktion	Wirkung
`1`	`print_int`	Wert in `$a0` wird dezimal ausgegeben.
`2`	`print_float`	Wert in `$f12` wird als 32-Bit Gleitkommazahl ausgegeben.
`3`	`print_double`	Wert in `$f12, $f13` wird als 64-Bit Gleitkommazahl ausgegeben.
`4`	`print_string`	Die mit dem Nullzeichen terminierte Zeichenkette, die an der Adresse `($a0)` beginnt, wird ausgegeben.

Eingabe

`$v0`	Systemfunktion	Wirkung
`5`	`read_int`	Liest eine, dezimal eingegebene, ganze Zahl in `$v0` ein.
`6`	`read_float`	Liest eine, dezimal eingegebene, 32-Bit Gleitkommazahl in `$f0` ein.
`7`	`read_double`	Liest eine, dezimal eingegebene, 64-Bit Gleitkommazahl in `$f12, $f13` ein.
`8`	`read_string`	Liest eine Zeichenkette ein und schreibt sie von Adresse `($a0)` bis maximal `($a0+$a1)` in den Speicher.

Sonstiges

`$v0`	Systemfunktion	Wirkung
`9`	**`sbrk`**	Liefert in `$v0` die Anfangsadresse eines freien Blocks der Größe `$a0` Bytes.
`10`	**`exit`**	Reguläres Programmende.

Befehle

Befehl	Argumente	Wirkung	Kommentar
`syscall`		Führt die, durch den Wert in `$v0` bestimmte, Systemfunktion aus.	system call
`la`	`Rd, label`	Lade in Register `Rd` die Adresse der Marke `label`.	load address
`move`	`Rd, Rs`	Lade in Register `Rd` den Wert aus Register `Rs`.	move

Beispiel 1

```
1  #
2  # compute the perimeter of a triangle
3  # with edges x, y, z and print it
4  #
5  .data
6  x:          .word  12
7  y:          .word  14
8  z:          .word   5
9  newline:    .asciiz "\n"
10 .text
11 main:    lw    $t0, x
12          lw    $t1, y
13          lw    $t2, z
14          add   $t0, $t0, $t1    # $t0 = x + y
15          add   $a0, $t0, $t2    # $a0 = x + y + z
16          li    $v0, 1           # print_int x + y + z
17          syscall
18          la    $a0, newline
19          li    $v0, 4           # print_string newline
20          syscall
21          li    $v0, 10          # EXIT
22          syscall
```

Beispiel 2

```
1   .data
2         txt1:    .asciiz "value= "
3         txt2:    .asciiz "text= "
4         input:   .ascii  "this text will be                "
5                  .asciiz "overwriten!                       "
6   .text
7                                    # input
8   main: li       $v0, 4            # print_str
9         la       $a0, txt1         # address of text one in $a0
10        syscall
11        li       $v0, 5            # read_int
12        syscall
13        move     $s0,$v0
14                                   #
15        li       $v0, 4            # print_str
16        la       $a0, txt2         # address text two in $a0
17        syscall
18        li       $v0, 8            # read_str
19        la       $a0, input        # address to write text on
20        li       $a1, 256          # max length
21        syscall
22                                   # output
23        li       $v0, 1            # print_int
24        move     $a0, $s0
25        syscall
26        li       $v0, 4            # print_str
27        la       $a0, input
28        syscall
29        li       $v0, 10           # exit
30        syscall
```

Byte-Reihenfolge

Werte von Datentypen, deren Speicherung mehrere Speicherzellen benötigt, können in verschiedenen Reihenfolgen im Speicher abgelegt werden.

In der MIPS-Architektur beträgt die Größe einer Speicherstelle ein Byte, deshalb wird diese Speicherorganisation als **Byte-Reihenfolge** (*byte order*) bezeichnet.

Es gibt zwei mögliche Reihenfolgen.

- **little-endian**, zuerst das Byte mit den niederwertigsten Bits.
- **big-endian**, zuerst das Byte mit den höchstwertigen Bits.

MIPS-Prozessoren können mit beiden Byte-Reihenfolge umgehen.

Der Simulator SPIM benutzt die Byte-Reihenfolge des Rechners, ab dem er läuft.

- little-endian z.B auf Intel 80x86.
- big-endian z.B. auf Macintosh oder Sun SPARC.

Beispiel

```
1   .byte 1, 2
2   .half 0x3456
3   .word 0xABCDEF
```

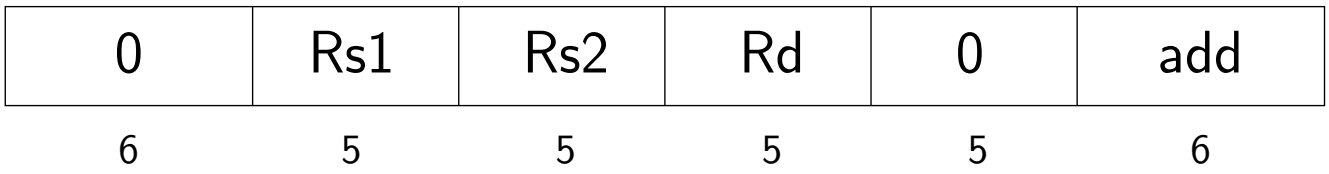
Adresse	little-endian	big-endian
0x1000 0007	0x00	0xEF
0x1000 0006	0xAB	0xCD
0x1000 0005	0xCD	0xAB
0x1000 0004	0xEF	0x00
0x1000 0003	0x34	0x56
0x1000 0002	0x56	0x34
0x1000 0001	0x02	0x02
0x1000 0000	0x01	0x01

Instruktionsformate

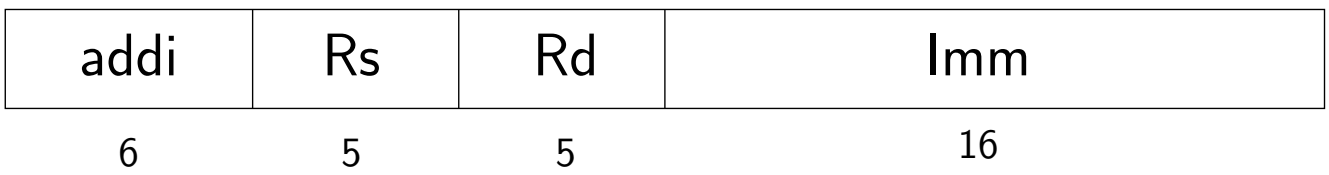
MIPS Befehle, inklusiere Argumente, werden mit 32-Bit codiert.

Beispiel

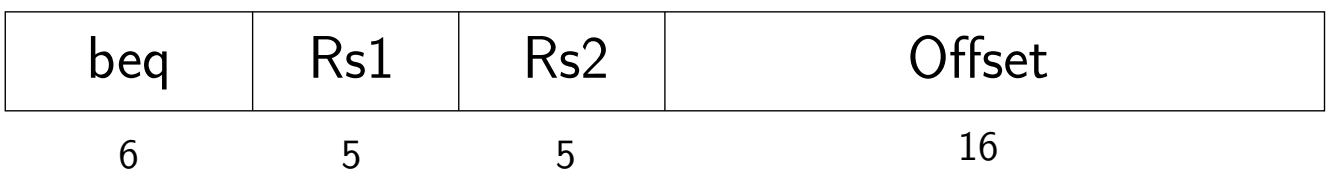
`add Rd, Rs1, Rs2`



`addi Rd, Rs, Imm`



`beq Rs1, Rs2, label`



`j label`

Adressierung

Die MIPS Hardware kennen nur eine Art der Adressierung **`Imm(Rs)`**, d.h. die Adresse ergibt sich aus der Summe des direkten Wertes **`Imm`** und dem Wert im Register **`Rs`**.

Die virtuelle Maschine kennt mehrere Arten der Adressierung.

Format	Adresse
`(Rs)`	Wert in Register **`Rs`**
`Imm`	Direkter Wert **`Imm`**
`Imm(Rs)`	Direkter Wert **`Imm`** + Wert in Register **`Rs`**
`label`	Adresse der Marke **`label`**
`label+Imm` **`label-Imm`**	Adresse der Marke **`label`** +/- direktem Wert **`Imm`**
`label+Imm(Rs)` **`label-Imm(Rs)`**	Adresse der Marke **`label`** +/- (direkter Wert **`Imm`** + Wert in Register **`Rs`**)

Beispiel

```
1   .data
2   var:     .word  20, 4, 22, 25, 7
3   .text
4   main:    lw    $t1, var            # $t1 = 20
5            lw    $t1, var+4          # $t1 = 4
6            lw    $t2, var($t1)       # $t2 = 4
7            lw    $t2, var+8($t1)     # $t2 = 25
8
9            addi $v0, $zero, 10       # exit
10           syscall
```

Ladebefehle (1/2)

Befehl	Argumente	Wirkung	Kommentar
`lb`	`Rd, Addr`	`Rd = MEM[Addr]`	load byte (8-bit)
`lbu`	`Rd, Addr`	`Rd = MEM[Addr]`	load unsigned byte (8-bit)
`lh`	`Rd, Addr`	`Rd = MEM[Addr]`	load half (16-Bit)
`lhu`	`Rd, Addr`	`Rd = MEM[Addr]`	load unsigned half (16-bit)

`ld`	`Rd, Addr`	Lädt das Doppelwort an Adresse Addr in die Register `Rd` und `Rd+1`.
		`Rd = MEM[Addr]`
		`Rd+1 = MEM[Addr+4]`
			load double word (64-Bit)

Ladebefehle (2/2)

Die Befehle **`lb`** und **`lh`** wandeln automatisch 8- bzw. 16-Bit Zahlen in 32-Bit Zahlen um.

Negative Zahlen werden im Zweikomplement dargestellt und auch entsprechend umgewandelt.

Beispiel

Aus dem Byte **`0xF0`** ($= -16$ mit 8 Bit) wird dann **`0xFFFF FFF0`** ($= -16$ mit 32 Bits).

Die Ladebefehle für Halbwörter (16 Bit) und Bytes (8 Bit) gibt es auch in einer *unsigned*-Version, wenn die gespeicherten Zahlen als vorzeichenlos betrachtet werden soll.

Beispiel

Aus dem Byte **`0xF0`** ($= 240$) wird dann **`0x0000 00F0`**.

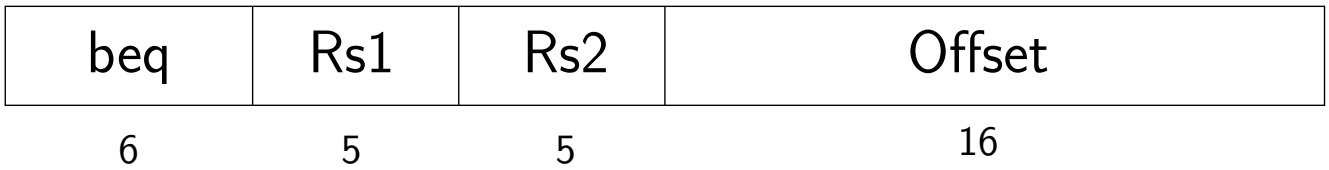
Speicherbefehle

Befehl	Argumente	Wirkung	Kommentar
sb	**Rs, Addr**	MEM[Addr] = Rs mod 256	store byte (8-bit)
sh	**Rs, Addr**	MEM[Addr] = Rs mod 2^{16}	store half (16-Bit)
sd	**Rs, Addr**	MEM[Addr] = Rs$\cdot 2^{32}$ + Rd+1	store double word (64-Bit)

unsigned-Versionen der Speicherbefehle sind nicht nötig.

Sprungbefehle *branch*

```
beq Rs1, Rs2, label
```



Branch Instruktionen benutzen einen 16-Bit Offset (*Zweierkomplement-Codierung*).

Mit Branch Instruktionen kann man $2^{15} - 1$ **Instruktionen** (nicht Bytes) vorwärts und 2^{15} Instruktionen rückwärts springen.

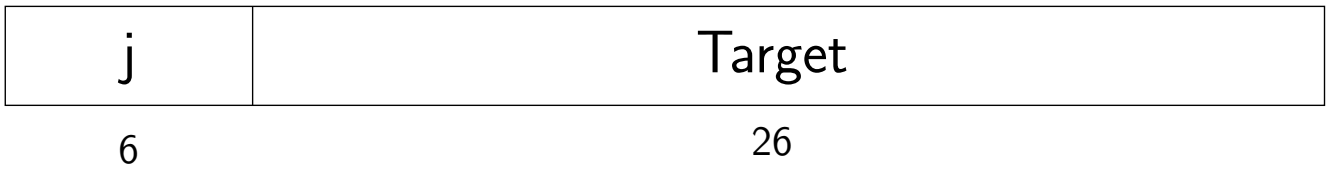
Da der Offset in Instruktionen (32-Bit) angegeben ist, kann ein Speicherbereich von -2^{17} bis $+2^{17} - 1$ Byte überdeckt werden.

Im Assemblercode kann der Offsets nicht direkt angegeben werden.

Bei einer Branch Instruktionen zu einer Marke wird der Abstand zwischen dem Branch und dem Ziel vom Assembler berechnet.

Sprungbefehl *jump*

`j label`



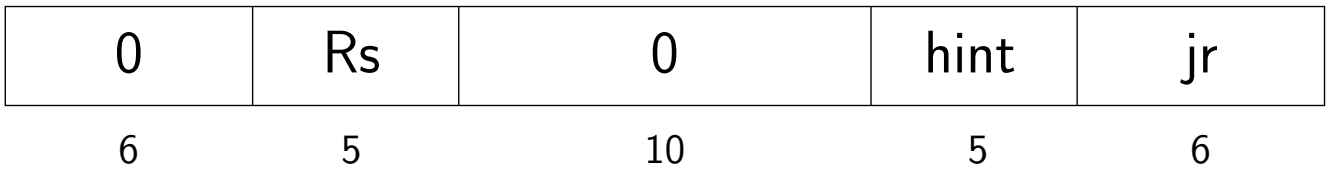
Das Ziel (2^{32}) des absoluten Sprungs wird wie folgt ermittelt.

- `Target` wird um 2 Bit nach links verschoben (2^{28}).
- Die 4 höchstwertigen Bits werden mit den 4 höchstwertigen Bits der Adresse, der Instruktion, die auf den *jump* Befehl folgt, aufgefüllt (2^{32}).

Mit *jump* können alle Adressen im der aktuellen Region der Größe
2^{28} Byte $= 256$ Megabyte erreicht werden.

Sprungbefehl *jump register*

`jr Rs`



Das `hint`-Feld kann einen Hinweis (*hint*) darauf geben, wie die Daten verwendet werden. Zur Zeit ist nur der Wert 0 für das `hint`-Feld von `jr` definiert.

Das Register kann eine 32-Bit Adresse aufnehmen, somit kann mit *jump register* **jede** Adresse erreicht werden.

Sprungbefehle

Befehl	Argumente	Wirkung	Kommentar
`beq`	`Rs1, Rs2, label`	Sprung nach `label`, falls `Rs1==Rs2`.	branch on equal
`bne`	`Rs1, Rs2, label`	Sprung nach `label`, falls `Rs1!=Rs2`.	branch on not equal
`bgez`	`Rs, label`	Sprung nach `label`, falls `Rs>=0`.	branch on greater than or equal zero
`bgtz`	`Rs, label`	Sprung nach `label`, falls `Rs>0`.	branch on greater than zero
`blez`	`Rs, label`	Sprung nach `label`, falls `Rs<=0`.	branch on less than or equal zero
`bltz`	`Rs, label`	Sprung nach `label`, falls `Rs<0`.	branch on less than zero
`j`	`label`	Sprung nach `label`.	jump
`jr`	`Rs`	Sprung an Adresse in Register `Rs`.	jump register

Beispiel

```
1   .data
2   msg1:    .asciiz  "1\n"
3   msg2:    .asciiz  "2\n"
4
5   .text
6   main:    la   $t1, label2
7            jr   $t1
8
9   label1:  li    $v0, 4
10           la    $a0, msg1
11           syscall
12
13           j finish
14
15  label2:  li    $v0, 4
16           la    $a0, msg2
17           syscall
18
19           j label1
20
21  finish:  addi $v0, $zero, 10
22           syscall
```

Addition/Subtraktion

Befehl	Argumente	Wirkung	Kommentar
`add`	**`Rd, Rs1, Rs2`**	**`Rd=Rs1+Rs2`**	addition
`addi`	**`Rd, Rs, Imm`**	**`Rd=Rs1+Imm`**	addition immediate
`sub`	**`Rd, Rs1, Rs2`**	**`Rd=Rs1-Rs2`**	substract

Bei den Befehlen **`add`** und **`sub`** können auch unmittelbare Operanden (*immediate*) verwenden werden.

- Der Assembler verwendet bei **`add`** automatisch den Befehl **`addi`**.
- Der Assembler lädt bei **`sub`** den unmittelbaren Operanden zunächst in das Register **`$at`** und führt dann die Subtraktion aus.

Diese Umwandlung geschieht auch bei den anderen arithmetischen Befehlen automatisch.

Beispiel

```
1  .data
2  var:      .word  20, 4, 28
3  newline:  .asciiz "\n"
4  .text
5  main:     lw   $t1, var
6            lw   $t2, var+4
7
8            add  $t0, $t1, $t2
9
10           li   $v0, 1              # print result
11           move $a0, $t0
12           syscall
13
14           li   $v0, 4              # print newline
15           la   $a0, newline
16           syscall
17           #-----------------------------------------------
18           add  $t0, $t0, 10
19           ...                      # print result and newline
20           #-----------------------------------------------
21           lw   $t3, var+8
22           sub  $t0, $t0, $t3
23           ...                      # print result and newline
24           #-----------------------------------------------
25           sub  $t0, $t0, 60
26           ...                      # print result and newline
```

Addition/Subtraktion ohne Überlauf (1/2)

Befehl	Argumente	Wirkung	Kommentar
`addu`	`Rd, Rs1, Rs2`	`Rd=Rs1+Rs2`	addition without overflow
`addui`	`Rd, Rs, Imm`	`Rd=Rs1+Imm`	addition immediate without overflow
`subu`	`Rd, Rs1, Rs2`	`Rd=Rs1-Rs2`	substract without overflow

Der Buchstabe `u` weist sowohl auf *unsigned* als auch auf *overflow* hin.

Durch die Vernachlässigung der Überläufe werden diese Befehle auch für vorzeichenlose Arithmetik brauchbar, trotzdem können auch bei vorzeichenlosen Zahlen Überläufe auftreten.

Addition/Subtraktion ohne Überlauf (2/2)

Beispiele. Nibble (Halbbyte) zur Demonstration.

Addiere 1 zur größten negativen Zahl (-1).

$$\begin{array}{rr} & 1111 \\ + & 0001 \\ \hline & 10000 \end{array}$$

Diese Überlaufe (in das Vorzeichen-Bit und über die maximale Anzahl an Bits hinaus) sind keine Fehler, dass Ergebnis 0 ist für vorzeichenbehaftete Zahlen korrekt.

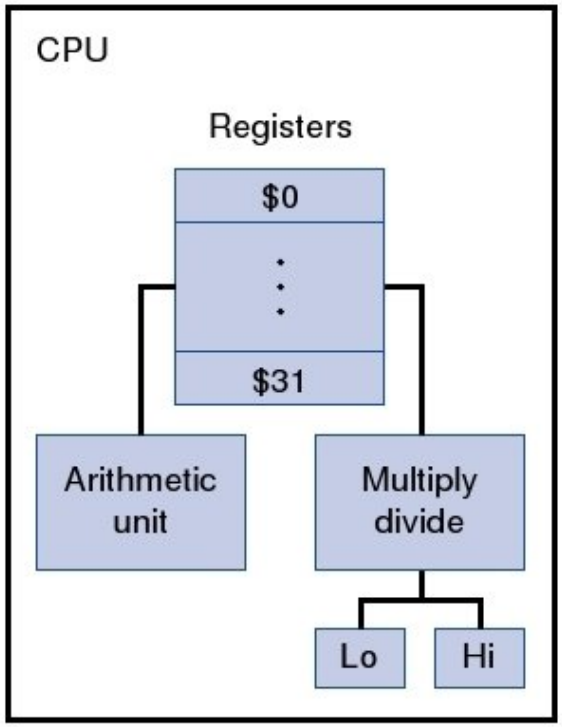
Addiere 1 zur größten positiven Zahl (7).

$$\begin{array}{rr} & 0111 \\ + & 0001 \\ \hline & 1000 \end{array}$$

Dieser Überlauf (in das Vorzeichen-Bit) ist ein Fehler, das Ergebnis -8 ist für vorzeichenbehaftete Zahlen nicht korrekt (**`add`** würde eine *exception* auslösen). Als vorzeichenlose Zahl (berechnet mit **`addu`**) kann das Ergebnis korrekt als 8 interpretiert werden.

Multiplikation und Division

Die CPU (*Central Processing Unit*) eines MIPS Prozessor besteht aus einem *Arithmetic (Logic) Unit (ALU)* und einem *Multiply Device*.



Die Befehle für das *Multiply Device* verwenden zwei Register (`lo, hi`), die **nicht** zu den direkt ansprechbaren 32 Registern gehören.

Befehle für Multiplikation und Division

Befehl	Argumente	Wirkung	Kommentar
`mult`	`Rs1, Rs2`	`hi=Rs1*Rs2 DIV` 2^{32}, `lo=Rs1*Rs2 MOD` 2^{32}	mulitply
`multu`	`Rs1, Rs2`	`hi=Rs1*Rs2 DIV` 2^{32}, `lo=Rs1*Rs2 MOD` 2^{32}	unsigned mulitply
`div`	`Rs1, Rs2`	`hi=Rs1 MOD Rs2, lo=Rs1 DIV Rs2`	divide
`divu`	`Rs1, Rs2`	`hi=Rs1 MOD Rs2, lo=Rs1 DIV Rs2`	divide without overflow

Beispiel. Nibble (Halbbyte) zur Demonstration.

Kleinste negativen Zahl -8 (1000) geteilt durch -1 (1111) ergibt bei **`div`** einen Überlauf, weil 8 nicht darstellbar ist.

Befehl	Argumente	Wirkung	Kommentar
`mfhi`	`Rd`	`Rd=hi`	move from high
`mflo`	`Rd`	`Rd=lo`	move form low
`mthi`	`Rs`	`hi=Rs`	move to high
`mtlo`	`Rs`	`lo=Rs`	move to low

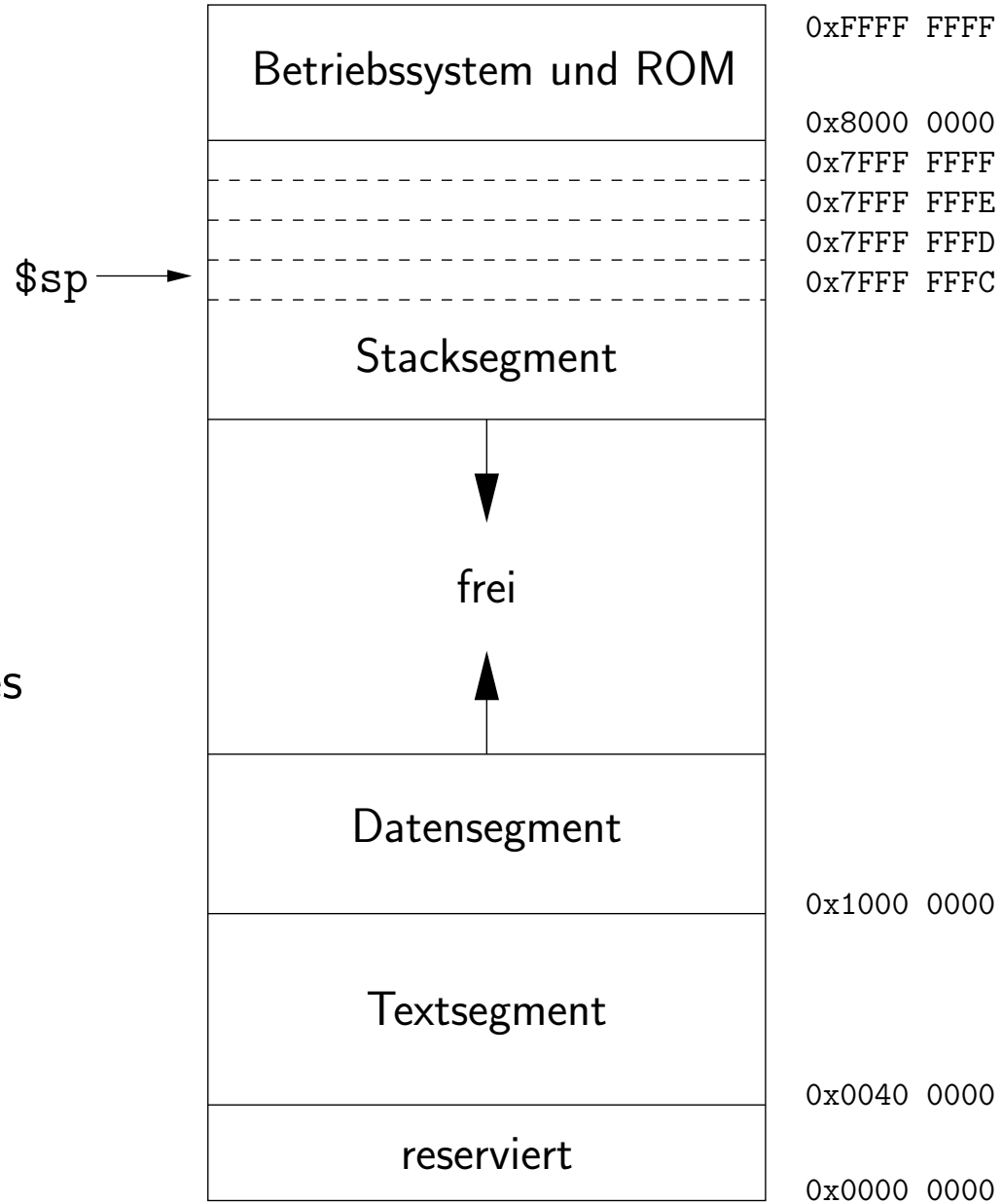
Sprungbefehle für Unterprogramme

Befehl	Argumente	Wirkung	Kommentar
`bgezal`	`Rs, label`	Sprung nach `label`, falls `Rs>=0`. Speichert Adresse der nächsten Instruktion in Register `$ra`.	branch on greater than or equal zero and link
`bltzal`	`Rs, label`	Sprung nach `label`, falls `Rs<0`. Speichert Adresse der nächsten Instruktion in Register `$ra`.	branch on less than zero and link
`jal`	`label`	Sprung nach `label`. Speichert Adresse der nächsten Instruktion in Register `$ra`.	jump and link
`jalr`	`Rs, Rd`	Sprung an Adresse in Register `Rs`. Speichert Adresse der nächsten Instruktion in Register `Rd` (Vorgabe `$ra`).	jump and link register

Stacksegment

Der Wert im Register **`$sp`** (*stackpointer*) ist das **erste freie Wort** im Stacksegment. Diese Konvention **muss** eingehalten werden.

Nach dem Laden eines Programmes zeigt der Stackpointer auf `0x7FFF FFFC`, das oberste **Wort** des Stacksegments.



Ablegen auf dem Stack

Ablegen auf dem Stack

1. Der Stackpointer wird verringert, denn der Stack wächst in Richtung kleinerer Speicherstellen.

   Da der Stackpointer auf Speicherplätze für **Wörter** zeigt, wird er jeweils um 4 Speicherstellen (Platz für ein Wort) erhöht oder verringert.
2. Das zu speicherende Wort wird an die ursprünglich Position des Stackpointers geschrieben, also 4 Speicherzellen weiter oben.

```
1       addi    $sp, -4         # decrement stackpointer
2       sw      $t0, 4($sp)     # push value in $t0 on stack
```

Die Reihenfolge von Stackpointer verringern und schreiben ist nicht beliebig, denn theoretisch kann nach jedem Befehl eine Unterbrechung (*interrupt*) auftreten.

Ein Unterbrechung wird durch eine spezielle Befehlsfolge (*interrupt handler*) behandelt, die eventuell den Stack manipuliert.

Tritt eine Unterbrechung, genau nach dem Schreiben auf den Stack und vor dem Aktualisieren des Stackpointers auf, kann der schon auf dem Stack abgelegte Wert vom *interrupt handler* überschrieben werden.

Holen vom Stack

Holen vom Stack

1. Das Wort wird vom Stack gelesen.
2. Der Stackpointer wird um 4 Speichplätze erhöht.

```
1       lw      $t0, 4($sp)     # pop value from stack in $t0
2       addi    $sp, 4          # increment stackpointer
```

Wieder sichert die Reihenfolge, dass ein Unterbrechung, nach dem Erhöhen des Stackpointers und vor dem Lesen des Wertes, nicht den Wert auf dem Stack überschreibt.

Unterprogramme

Was passiert, wenn in einem Programm ein Unterprogramm aufgerufen wird?

In Hochsprachen muss man sich mit den Details nicht belasten, denn der Compiler kümmert sich darum.

Programmiert man in Assembler muss für jedes Unterprogramm der Aufruf und auch die Rückkehr explizit implementiert werden.

Ein Großteil des Verwaltungsaufwandes, den Unterprogamme verursachen, ist die Verwaltung des Rahmens (*frame*) eines Unterprogramms.

Der Rahmen wird für Folgendes gebraucht.

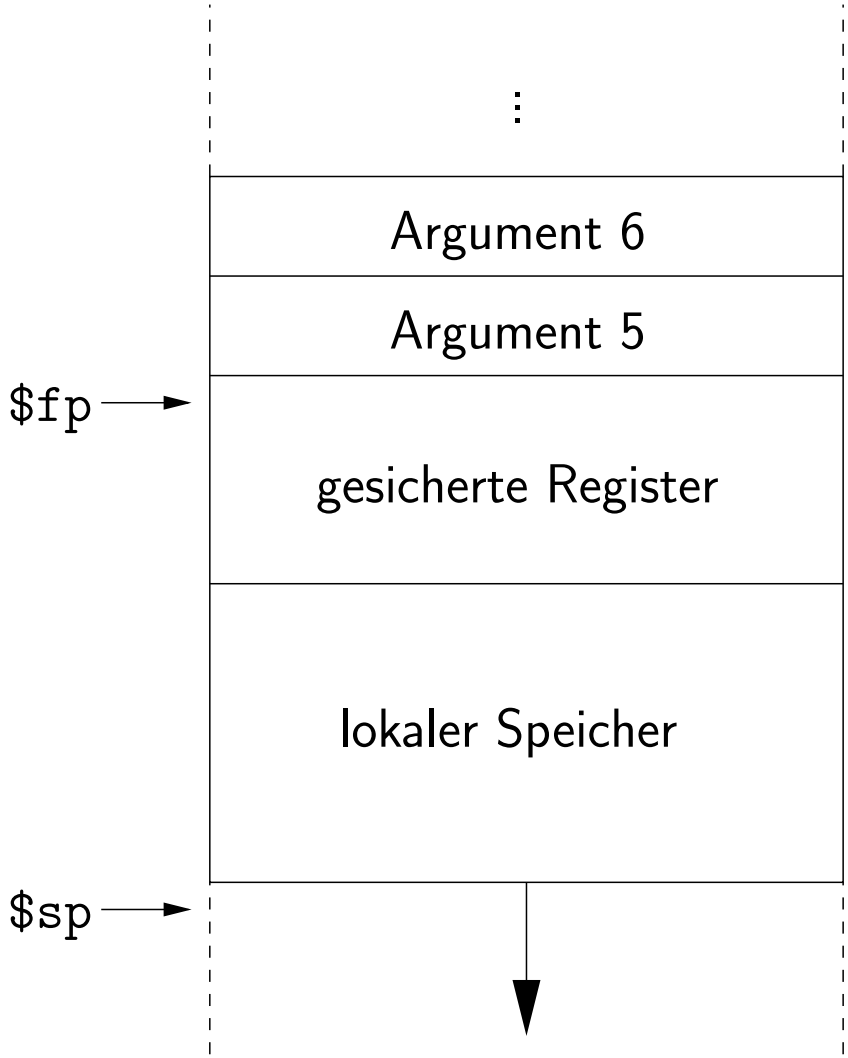
- Aufnehmen der Werte, die das Unterprogramm als Argumente übergeben bekommt.
- Sichern von Registern, die vom Unterprogramm verändert werden, aber von den der Aufrufer des Unterprogramm erwartet, dass sie sich nicht ändern.
- Bereitstellen von lokalem Speicher für das Unterprogramm.

Framepointer

Unterprogramm Aufrufe und Rückkehr folgen einer strikten *last-in, first-out (LIFO)* Reihenfolge. Damit lassen sich die Rahmen sehr gut auf dem Stack ablegen, deshalb werden diese oft Stackrahmen (*stack frames*) genannt.

Der Rahmen des aktuellen (Unter-) Programms ist der Speicher zwischen dem Framepointer (**`$fp`**), der auf das **`erste Wort`** des Rahmens zeigt und dem Stackpointer (**`$sp`**), der auf das erste freie Wort im Stacksegment zeigt.

Die Argumente können an das Unterprogram in Registern oder durch Ablegen auf dem Stack übergeben werden.



Rahmen eines Unterprogramms

Das aktuelle Unterprogramm kann den Framepointer und den Stackpointer benutzen um Speicherstellen im Rahmen des Unterprogramm zu adressieren.

Der Rahmen für ein Unterprogramm kann auf unterschiedliche Weise gebildet werden, damit das Zusammenspiel von Aufrufer (*caller*) und Unterfunktion (*callee*) funktioniert, müssen diese sich auf eine Vorgehensweise einigen.

Auf MIPS Prozessoren gibt es feste **Konvention** für die Benutzung von Unterprogramme.

Konvention für das Aufrufen von Unterprogrammen

Schritte, die der Aufrufer vor dem Aufrufen einer Unterfunktion durchführt.

1. **Sichern von Registern.** Das Unterprogramm kann die Register `$a0, ..., $a3` und `$t0, ..., $t9` verändern **ohne** ihren Wert zu sichern.

   Wenn der Aufrufer den Wert eines dieser Register nach Rückkehr des Unterprogramm weiterverwenden will, muss er dessen Wert selber sichern (*caller-saved registers*).
2. **Übergabe von Argumenten.** Die ersten vier Argumente werden an das Unterprogramm in den Registern `$a0, ..., $a3` übergeben. Weitere Argumente werden über den Stack übergeben, diese liegen direkt über dem Rahmen des Unterprogramms.
3. **Jump and Link.** Ein Sprung zur Unterfunktion wird mit einem der Sprungbefehle ausgeführt, die die Rücksprungadresse im Register `$ra` sichern.

Schritte nach der Rückkehr der Unterfunktion zum Aufrufer.

1. **Registerwerte zurückschreiben.** Alle auf dem Stack gesicherten Registerwerte werden wieder in die Register zurückgeschrieben.

Konvention für Unterprogramme (1/2)

Die ersten Aufgaben eines Unterprogramm, nachdem es aufgerufen wurde, ist seinen Rahmen einzurichten.

1. **Speicher reservieren.** Speicher für den Rahmen wird auf dem Stack reserviert, indem die Größe des Rahmen vom Stackpointer **`$sp`** abgezogen wird.
2. **Sichern von Registern.** Das Unterprogramm **muss** die Werte der Register **`$s0, ..., $s7, $fp, $ra`** sichern, bevor es deren Wert verändert (*callee-saved registers*).

   Der Aufrufer erwartet, dass die Werte dieser Register nach der Rückkehr des Unterprogramms unverändert sind.

   Register **`$fp`** wird von jedem Unterprogramm, das einen neuen Rahmen anlegt, gesichert. Register **`$ra`** braucht nur gesichert zu werden, wenn das Unterprogramm selbst wieder ein Unterprogramm aufruft. Alle anderen Register müssen gesichert werden, wenn sie verwendet werden.
3. **Setzen des Framepointers.** Der Framepointer **`$fp`** bekommt den Wert des Stackpointers **`$sp`** vor dem Reservieren von Speicher für den Rahmen des Unterprogramms.

Konvention für Unterprogramme (2/2)

Bevor eine Unterfunktion zum Aufrufer zurückkehrt muss sie ihren Rahmen wieder freigeben und die gesicherten Werte zurückschreiben.

1. **Funktionswert.** Die Unterprogramm kann Werte in **`$v0, $v1`** zurückliefern.
2. **Registerwerte zurückschreiben.** Alle auf dem Stack gesicherten Registerwerte werden wieder in die Register zurückgeschrieben.
3. **Rahmen zurücksetzen.** Framepointer **`$fp`** und Stackpointers **`$sp`** werden auf ihre Ausgangswerte zurückgesetzt.
4. **Rückkehr.** Rücksprung an die in **`$ra`** gesicherte Adresse.

Einschub. Schiebebefehle

Im Zusammenhang mit Manipulation von Frame- und Stackpointer wird oft die ganzzahlige Multiplikation oder Division mit/durch eine Zweipotzen benötigt.

Dazu muss nicht die aufwendige Multiplikation/Division verwendet werden, sondern es können **Schiebebefehle** verwendet werden.

Befehl	Argumente	Wirkung	Kommentar
`sll`	`Rd, Rs, Imm`	`Rd = Rs` $\ll_{\texttt{Imm}}$	shift left logical
`sllv`	`Rd, Rs1, Rs2`	`Rd = Rs1` $\ll_{\texttt{Rs2}}$	shift left logical variable
`srl`	`Rd, Rs, Imm`	`Rd = Rs` $\gg_{\texttt{Imm}}$	shift right logical
`srlv`	`Rd, Rs1, Rs2`	`Rd = Rs` $\gg_{\texttt{Rs2}}$	shift right logical variable
`sra`	`Rd, Rs, Imm`	`Rd = Rs / ` $2^{\texttt{Imm}}$	shift right arithmetic
`srav`	`Rd, Rs1, Rs2`	`Rd = Rs1 / ` $2^{\texttt{Rs2}}$	shift right arithmetic variable

Bei den Schiebebefehlen wird zwischen logischen und arithmetischen Befehlen differenziert. Die logischen Schiebebefehle füllen die freigewordenen Bits mit Nullen auf, während der arithmetische Schiebebefehl das höchstwertige Bit der Quelle übernimmt.

Bemerkung. Beim Schieben nach links muss das Vorzeichen nicht beachtet werden (Zweierkomplementdarstellung).

Beispiel (1/3)

Vertauschen zwei aufeinanderfolgender Listenelemente.

In einer Hochsprache.

```
1  swap (int[] v, int k)
2  {
3       int temp;
4       temp = v[k];
5       v[k] = v [k+1];
6       v[k+1] = temp;
7  }
```

Beispiel (2/3)

```
1   swap:   addi    $sp, $sp, -12       # reserve space on stack
2           sw      $fp, 12($sp)        # save frame pointer on stack
3           sw      $s0, 8($sp)         # save $s0 on stack
4           sw      $s1, 4($sp)         # save $s1 on stack
5           addi    $fp, $sp, 12        # new frame pointer
6                                       # $a0 = v[], $a1 = k
7           sll     $t0, $a1, 2         # $t0 := k * 4
8           add     $t0, $a0, $t0       # $t0 := v + (k * 4)
9                                       # $t0 contains address of v[k]
10          lw      $s0, 0($t0)         # $s0 <-- v[k]
11          lw      $s1, 4($t0)         # $s1 <-- v[k+1]
12          sw      $s1, 0($t0)         # $s1 --> v[k]
13          sw      $s0, 4($t0)         # $s0 --> v[k+1]
14          lw      $s1, 4($sp)         # restore $s1 from stack
15          lw      $s0, 8($sp)         # restore $s0 from stack
16          lw      $fp, 12($sp)        # restore $fp from stack
17          addi    $sp, $sp, 12        # restore stack pointer
18          jr      $ra                 # return
```

Beispiel (3/3)

```
1   .data
2   v:        .word   1, 2, 3, 4, 5, 6, 7, 8, 9, 10
3   k:        .word   8
4   length:   .word   10
5   newline:  .asciiz "\n"
6   .text
7   #-------------------------------------------------
8   main:     lw      $s0, length       # init output
9             la      $s1, v
10
11  #-------------------------------------------------
12            la      $a0, v            # argument v
13            lw      $a1, k            # argument k
14            jal     swap              # call swap
15
16  #-------------------------------------------------
17  out:      li      $v0, 1            # output
18            lw      $a0, ($s1)
19            syscall
20
21            li      $v0, 4
22            la      $a0, newline
23            syscall
24
25            add     $s1, $s1, 4
26            sub     $s0, $s0, 1
27
28            bne     $zero, $s0, out
29
30  #-------------------------------------------------
31            li     $v0, 10            # exit
32            syscall
```

Datenhaltung, Felder

Ein Feld von mehren Wörtern kann wie folgt angelegt werden.

```
1  .data
2  feld:        .word       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Alle Feldelemente sind mit dem Wert 0 initialisiert. Es gibt keine Direktive, die das wiederholte Eintragen der Werte erleichtert.

Weiterhin gibt es die Möglichkeit mit der Direktive **`.space`** ganze Bereiche zu reservieren, allerdings **ohne** sie gleichzeitig zu initialisieren.

```
1  .data
2  feld:          .space      52
```

Der Parameter gibt die Zahl von **Bytes** an, die reserviert werden.

Bemerkung.

Bei Direktiven legen ein Feld mit 13 Wörtern an, aber **`.space`** initialisiert die Feldelemente nicht.

Beispiel

```
1   .data
2   feld:    .space 52                # int[13] feld
3
4   .text
5   main:    li     $t0, 0            # $t0 = 0
6   while:   bgt    $t0, 48, end      # while ($t0 <= 48)
7            sw     $t0, feld($t0)    #   feld[$t0/4] = $t0
8            addi   $t0, $t0, 4       #   $t0 += 4
9            b      while             # end_while
10  end:
```

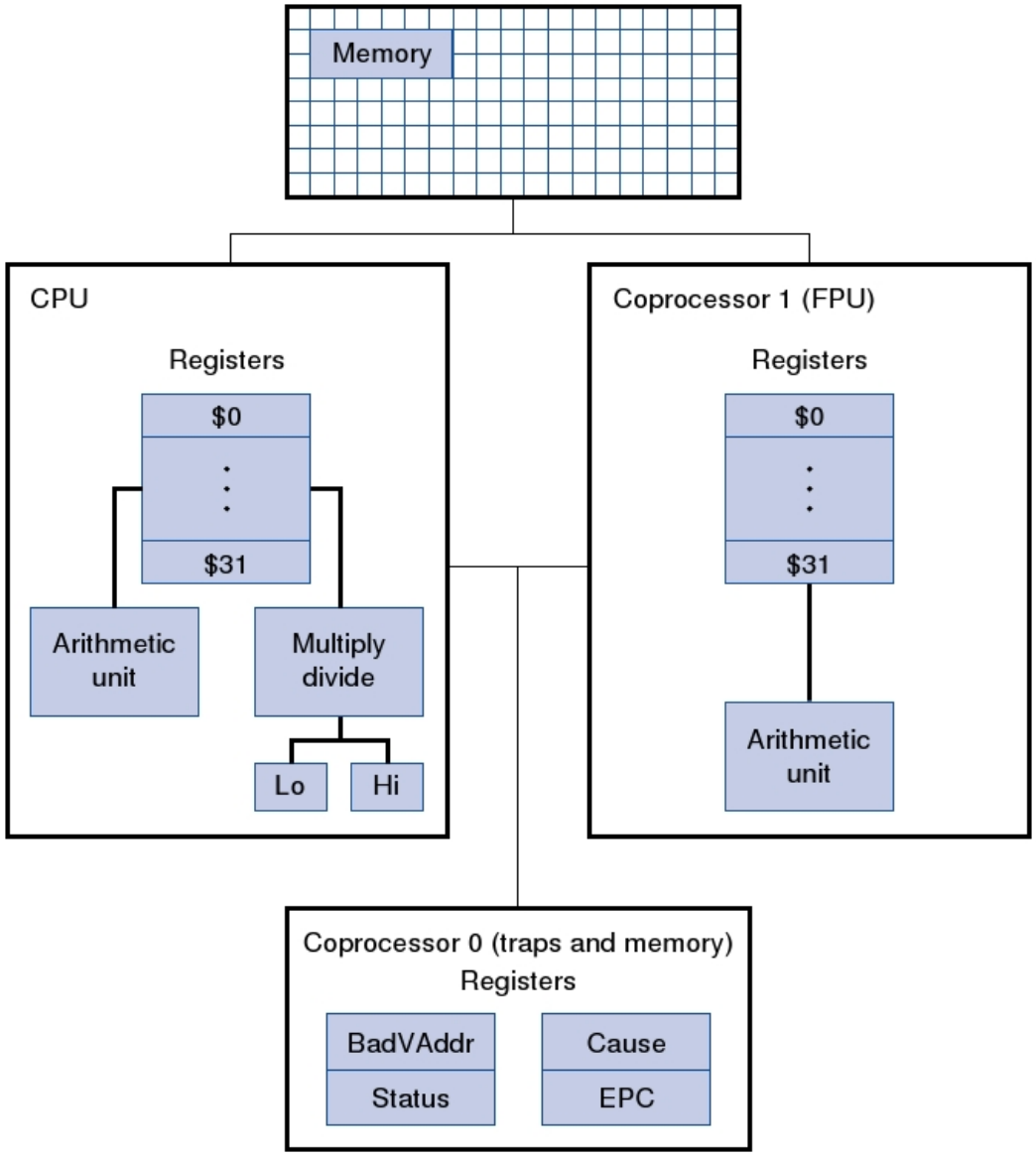
Gleitkomma-Arithmetik

Ein MIPS Prozessor besteht aus einer CPU zum Rechnen mit ganzen Zahlen und Co-Prozessoren für Rechnungen mit anderen Zahlenformaten (z.B. Gleitkommazahlen) und Hilfsaufgaben.

SPIM simuliert Folgendes.

- CPU (*Central Processing Unit*) für Ganzzahl-Arithmetik, Logik, etc.
- Co-Prozessor 0 behandelt Exceptions und Interrupts.
- Co-Prozessor 1 (*FPU, Floating Point Unit*) ist zuständig für Gleitkomma-Arithmetik.

SPIM (R2000) CPU und FPU



Datenhaltung, Gleitkommazahlen

Gleitkommazahlen werden, wie alle anderen Daten auch, mit Direktiven im Speicher abgelegt.

`.float`

Die Direktive **`.float`** legt die folgenden Werte als 32-Bit-Gleitkommazahlen mit einfacher Genauigkeit (*single precision*) ab.

`.double`

Die Direktive **`.double`** legt die folgenden Werte als 64-Bit-Gleitkommazahlen mit doppelter Genauigkeit (*double precision*) ab.

Die Werte können in nicht normalisierter Form in Exponentialschreibweise eingetragen werden (`123.45e67`, `0.0003e-9`). Das `e` muss klein geschrieben sein. Auf die Exponentialschreibweise kann auch verzichtet werden, dann muss allerdings ein Dezimalpunkt vorhanden sein (`1234567.` und `0.09999` sind erlaubt, `1234567` jedoch nicht).

Register des mathematischen Koprozessors

Der mathematische Koprozessor verfügt, wie der Hauptprozessor, über 32 Register von je 32 Bit Breite.

Die Register sind von 0 bis 31 durchnummeriert und haben die Namen **`$f0`** bis **`$f31`**.

Alle Register sind beschreibbar und für jeden beliebigen Zweck einsetzbar.

Einschränkungen

Rechnet man mit doppelt genauen Gleitkommazahlen, werden 64 Bit benötigt, die durch Zusammenfassen je zweier Register bereitgestellt werden.

Zusammengefasst werden ein Register mit einer geraden Registernummer und das folgende Register, z.B. **`$f0, $f1`** oder **`$f6, $f7`**.

Notation der Koprozessorregister.

Notation	Erläuterung	Beispiel
`Fd, Fs, Fs1, Fs2`	Registeroperanden des **mathematischen** Koprozessors	**`$f0, $f13`**

Lade-/Speicher-/Transferbefehle

Der MIPS Assembler hat für jeden Koprozessor eigene Lade-/Speicher- und Transferbefehle **xxx**<z>, wobei für <z> eine Zahl zwischen 0 und 3 zu wählen ist.

Hinweis

SPIM simuliert nur die Koprozessoren 0 und 1.

Wir betrachten die SPIM Befehle für den mathematischen Koprozessor, der mit der Nummer **1** angesprochen wird.

Befehl	Argumente	Wirkung	Kommentar
`lwc1`	`Fd, Adr`	`Fd = Mem[Adr]`	load word coprocessor 1
`swc1`	`Fs, Adr`	`Mem[Adr] = Fs`	store word coprocessor 1
`mfc1`	`Rd, Fs`	`Rd = Fs`	move from coprocessor 1
`mtc1`	`Rs, Fd`	`Fd = Rs`	move to coprocessor 1

Mit den move–Befehlen können Daten zwischen dem Haupt- und dem Koprozessor hin- und herbewegt werden.

Hinweis

Es werden jeweils nur Bitfolgen kopiert, eine Konvertierung, z.B. von ganzer Zahl nach Gleitkommazahl oder umgedreht, findet **nicht** statt.

Pseudo-Lade-/Speicher-/Transferbefehle

Der Umgang mit Gleitkommazahlen, insbesondere mit solchen mit doppelter Genauigkeit, wird durch Pseudobefehle erleichtert.

Befehl	Argumente	Wirkung	Kommentar
`l.s`	`Fd, Addr`	`Fd = Mem[Addr]`	load floating-point single
`l.d`	`Fd, Addr`	`Fd = Mem[Addr], Fd+1 = Mem[Addr+4]`	load floating-point double
`s.s`	`Fs, Addr`	`Mem[Addr] = Fs`	store floating-point single
`s.d`	`Fs, Addr`	`Mem[Addr] = Fs, Mem[Addr+4] = Fs+1`	store floating-point double
`mfc1.d`	`Rd, Fs`	`Rd = Fs, Rd+1 = Fs+1`	move double from coprocessor 1
`mov.s`	`Fd, Fs`	`Fd = Fs`	move floating-point single
`mov.d`	`Fd, Fs`	`Fd = Fs, Fd+1 = Fs+1`	move floating-point double

Addition, Subtraktion, Multiplikation, Division

Befehl	Argumente	Wirkung	Kommentar
`add.d`	`Fd, Fs1, Fs2`	`(Fd Fd+1) = (Fs1 Fs1+1) + (Fs2 Fs2+1)`	floating point addition double
`add.s`	`Fd, Fs1, Fs2`	`Fd=Fs1+Fs2`	floating point addition single
`sub.d`	`Fd, Fs1, Fs2`	`(Fd Fd+1) = (Fs1 Fs1+1) - (Fs2 Fs2+1)`	floating point substract double
`sub.s`	`Fd, Fs1, Fs2`	`Fd=Fs1-Fs2`	floating point substract single
`mul.d`	`Fd, Fs1, Fs2`	`(Fd Fd+1) = (Fs1 Fs1+1) * (Fs2 Fs2+1)`	floating point multiply double
`mul.s`	`Fd, Fs1, Fs2`	`Fd=Fs1*Fs2`	floating point multiply single
`div.d`	`Fd, Fs1, Fs2`	`(Fd Fd+1) = (Fs1 Fs1+1) / (Fs2 Fs2+1)`	floating point divide double
`div.s`	`Fd, Fs1, Fs2`	`Fd=Fs1/Fs2`	floating point divide single

Sonstige arithmetische Befehle

Befehl	Argumente	Wirkung	Kommentar
`abs.d`	`Fd, Fs1`	`(Fd Fd+1) = abs(Fs Fs+1)`	floating point absolute value double
`abs.s`	`Fd, Fs`	`Fd=abs(Fs)`	floating point absolute value single
`neg.d`	`Fd, Fs1, Fs2`	`(Fd Fd+1) = -(Fs2 Fs2+1)`	floating point negate double
`neg.s`	`Fd, Fs1, Fs2`	`Fd=-Fs2`	floating point negate single

Vegleichsbefehle (1/2)

Es gibt verschiedene Vergleichsbefehle, die jeweils zwei Gleitkommazahlen vergleichen und als Ergebnis das *condition-flag* des mathematischen Koprozessors auf 1 (wahr) oder auf 0 (falsch) setzen.

Befehl	Argumente	Wirkung	Kommentar
`c.eq.d`	`Fs1, Fs2`	Setze *condition-flag* auf wahr, wenn `(Fs1 Fs1+1)` $=$ `(Fs2 Fs2+1)`	compare equal double
`c.eq.s`	`Fs1, Fs2`	Setze *condition-flag* auf wahr, wenn `Fs1` $=$ `Fs2`	compare equal single
`c.le.d`	`Fs1, Fs2`	Setze *condition-flag* auf wahr, wenn `(Fs1 Fs1+1)` $\leq$ `(Fs2 Fs2+1)`	compare less than equal double
`c.le.s`	`Fs1, Fs2`	Setze *condition-flag* auf wahr, wenn `Fs1` $\leq$ `Fs2`	compare less than equal single
`c.lt.d`	`Fs1, Fs2`	Setze *condition-flag* auf wahr, wenn `(Fs1 Fs1+1)` $<$ `(Fs2 Fs2+1)`	compare less than double
`c.lt.s`	`Fs1, Fs2`	Setze *condition-flag* auf wahr, wenn `Fs1` $<$ `Fs2`	compare less than single

Vegleichsbefehle (2/2)

Das *condition-flag* kann als Sprungkriterium verwendet werden.

Befehl	Argumente	Wirkung	Kommentar
`bc1t`	**`label`**	Sprung nach **`label`**, wenn *condition-flag* des mathematischen Koprozessors wahr (1) ist.	branch coprocessor 1 true
`bc1f`	**`label`**	Sprung nach **`label`**, wenn *condition-flag* des mathematischen Koprozessors falsch (0) ist.	branch coprocessor 1 false

Beispiel (1/2)

```
1   .data
2   # 3x^2 + 2x + 1 ----------------------------------------
3   a_n:        .float   3. 2.
4   a_null:     .float   1.
5   newline:    .asciiz "\n"
6   .text
7   main:       li       $v0, 6                 # read float in $f0
8               syscall
9
10  #-------------------------------------------------------------
11  ...
12  #-------------------------------------------------------------
13
14              li       $v0, 2                 # print float $f12
15              syscall
16
17              la       $a0, newline
18              li       $v0, 4                 # print newline
19              syscall
20
21              li       $v0, 10                # exit
22              syscall
```

Beispiel (2/2)

```
1  a_n:        .float   3. 2.
2  a_null:     .float   1.
3  ...
4  #-------------------------------------------------------------
5                                          # $f0 = x
6             la       $a0, a_null        # $a0 = a_null
7             la       $a1, a_n           # $a1 = a_n
8
9  poly:      l.s      $f12, ($a1)        # $f12 = a_n
10
11 loop:      beq      $a1, $a0, end_loop # while ($a1 != $a0)
12                                         #   $a1 = a_i
13            addiu    $a1, $a1, 4        #   a_i = a_(i-1)
14            l.s      $f10, ($a1)        #   $f10 = a_i
15            mul.s    $f12, $f12, $f0    #   $f12 = $f12 * x
16            add.s    $f12, $f12, $f10   #   $f12 = $f12 + a_i
17            j        loop               # end_while
18 end_loop:
19 #-------------------------------------------------------------
20 ...
```