Technische Informatik

Lernzusammenfassung

Lukas Bach
22. Februar 2017

zu den Modulen Digitaltechnik und Entwurfsverfahren und
Rechnerorganisation
am Karlsruher Institut für Technologie

22. Februar 2017

Inhaltsverzeichnis

1 Zahlensysteme, Rechnerarithmetik

1.1 Umwandlungen von Zahlensystemen

1.2 Darstellung negativer Zahlen

Als Referenz: $77_{10} = (0000\ 0100\ 1101)_2$.

Vorzeichen-Betrag-Darstellung Das erste Bit definiert das Vorzeichen (1 = negativ, 0 = positiv), der Rest den Betrag der Zahl. $-77_{10} = (1000\ 0100\ 1101)_{2,12}$

Einerkomplement-Dartsellung Positive Zahlen werden auf intuitive Weise kodiert. Negative Zahlen entsprechen immer genau ihrem invertierten Betrag. $-77_{10} = (1111\ 1011\ 0010)_{1K,12}$

Zweierkomplement-Darstellung Positive Zahlen werden auf intuitive Weise kodiert. Negative Zahlen entsprechen ihrem invertierten Betrag plus eins. Dadurch wird die doppelte Belegung der null vermieden. $-77_{10} = (1111\ 1011\ 0011)_{1K,12}$

	Darstellung mit			
Dezimal	Vorzeichen-Betrag	Einerkomplement	Zweierkomplement	Exzess-4
-4	nicht möglich		100	000
-3	111	100	101	001
-2	110	101	110	010
-1	101	110	111	011
0	100 oder 000		000	100
1	001			101
2	010			110
3	011			111

1.3 Darstellung Gleitkommazahlen mit IEEE 754

y	$y-1$	x	$x-1$	0
Vz	Charakteristik		Mantisse	

$$\text{Dezimalzahl} = (-1)^{\text{Vz}} \cdot (0,\text{Mantisse}) \cdot b^{\text{Exponent}} \quad \overset{IEEE}{=} (-1)^{\text{Vz}} \cdot (0,\text{Mantisse}) \cdot 2^{\text{Exponent}}$$

$$\text{Exponent} = \text{Charakteristik} - b^{(y-1)-x} \quad \overset{IEEE}{=} \text{Charakteristik} - 127$$

Die Mantisse ist das, was nach dem Komma kommt, die Charakteristik ist die Potenz der Zahl. Bei IEEE 754 ist $b = 2$.

1.3.1 Rechenschema: Dezimalzahl zu Gleitkommazahl

Zahlen wir $18,4_{10}$ sollen als IEEE 754 Gleitkommazahl kodiert werden.

1. Umwandlung der Dezimalzahl in eine duale Festkommazahl ohne Vorzeichen

2. Normalisieren und Bestimmen des Exponenten
    - Normalisieren: Aus $1011,001001...$ wird $1,011001001....$
    - Exponent: Charakteristik -127.
3. Vorzeichen-Bit bestimmen
4. Gleitkommazahl bilden
    - Konkateniere: 1 Bit Vorzeichen + 8 Bit Exponent + 23 Bit Mantisse

1.3.2 Rechenschema: Gleitkommazahl zu Dezimalzahl

Nach Wikipedia. Zahlen wie 0 10000011 00100110011001100110011 sollen als Dezimalzahl dargestellt werden.

1. Berechnung des Exponenten
    - Umwandlung des Exponenten in dezimal
    - 127 draufaddieren.
2. Berechnung der Mantisse
    - Prüfen: Ist die Zahl normalisiert? Falls ja, muss eine 1 vor dem Komma sein.
    - Komma um *Exponent* Stellen nach rechts verschieben
3. Umwandlung in dezimal
4. Vorzeichen ergänzen

TODO: Randfälle von IEEE 754

1.4 BCD- und Graycode

BCD-Codes sind binäre Kodierungen von Dezimalzahlen, wobei jede Dezimalzahl zu genau einem vierer Block von Binärzahlen kodiert werden.

Bei Gray-Codes unterscheidet sich bei benachtbarten Codewärtern immer nur in einer einzigen Ziffer, um Übertragungsfehler bei kontinuierlicher Übertragung zu verringern.

Die Gray-Codierung einer Zahl erhält man mit $x \bigoplus \left\lfloor \frac{x}{2} \right\rfloor$.

Dezimal	BCD	Gray
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	-	1111
11	-	1110
12	-	1010
13	-	1011
14	-	1001
15	-	1000

TODO: Unicode, Ascii TODO: Hamming-Code

1.5 Prüfbits für Einzelbit-Fehlerkorrektur

Wie viele Prüfbits sind für eine Einzelbit-Fehlerkorrektur in m-Bit Datenwörtern erforderlich? $2^k \geq m + k + 1$, dann k Prüfbits.

2 Schaltnetze

TODO: Operatorensysteme, Seite 43

2.1 Definitionen

$$y(a,b)_{DNF} = \underbrace{ab}_{\text{Minterm}} \vee \underbrace{\bar{a}\bar{b}}_{\text{Minterm}}$$

$$y(a,b)_{KNF} = \underbrace{(\bar{a} \vee b)}_{\text{Maxterm}} \wedge \underbrace{(a \vee \bar{b})}_{\text{Maxterm}}$$

$$\underbrace{a \wedge b \wedge \bar{c}}_{\text{Implikant}} = \left(\underbrace{a \wedge b \wedge \bar{c} \wedge d}_{\text{Minterm}} \right) \vee \left(a \wedge \underbrace{b}_{\text{Literal}} \wedge \bar{c} \wedge \bar{d} \right)$$

2.2 Rechenschema: Bestimmen von DNF und KNF

Von einer Schaltfunktion lassen sich $DNF = ab\bar{c} \vee \bar{a}b\bar{c}$ und $KNF = (a \vee b \vee \bar{c}) \wedge (\bar{a} \vee b \vee c)$ meist nur durch Umformungen bestimmen. Dazu können folgende Formeln verwendet werden.

Distributivgesetze	$a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$ $a \sqcup (b \sqcap c) = (a \sqcup b) \sqcap (a \sqcup c)$	Idempotenzgesetze	$a \wedge a = a = a \vee a$
Asorptionsgesetze	$a \wedge (a \vee b) = a$ $a \vee (a \wedge b) = a$	DeMorgan-Gesetze	$a \bar{\wedge} b = \bar{a} \vee \bar{b}$ $a \bar{\vee} b = \bar{a} \wedge \bar{b}$

2.3 Rechenschema: Bestimmen von MINt und MAXt aus DNF und KNF

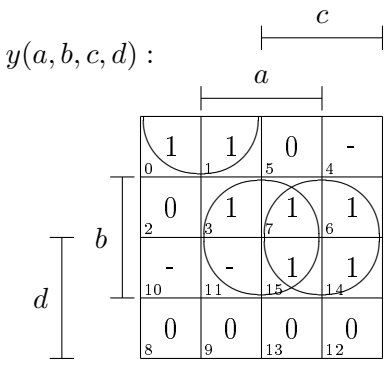
Gegeben ist $y_{DNF} = \bigvee (\bigwedge L_i)$ und $y_{KNF} = \bigwedge (\bigvee L_i)$

- Für jedes Implikant $\bigwedge L_i \in y_{DNF}$ bzw. Implikat $\bigvee L_i \in y_{KNF}$:
  - Sortiere die Literale in absteigender Reihenfolge zu $L_n L_{n-1} L_{n-2}...$, z.B. cba.
  - Kodiere die Literale zu Binärzahlen, indem Variable L_i als 1 und jede Negation $\bar{L_i}$ als 0 interpretiert wird.
  - Setze alles zu einer Binärzahl zusammen und interpretiere diese als Dezimalzahl. Sie gibt die Nummer des Minterms für Implikanten (DNF) bzw. des Maxterms für Implikaten (KNF) an.

Beispiel: $y_{DNF} = ... \vee (ab\bar{c})$, invers Sortiert: $\bar{c}ba \rightarrow 011 \rightarrow 3$.

2.4 Rechenschema: Bestimmen von DMF mit gegebenen MIN-Termen und Don't-Care-Stellen

Eine Funktion ist durch MIN-Terme und Don't-Care-Stellen definiert, zum Beispiel durch $y := MINt(0, 1, 3, 6, 7, 14, 15) \vee d(4, 10, 11)$. Dann füllt man das KV-Diagramm dazu aus und findet alle Primimplikanten:



Dann liest man die DMF aus den Primimplikanten ab: $y(a, b, c, d)_{DMF} = ab \vee bc \vee \bar{b}\bar{c}\bar{d}$

2.5 Rechenschema: Bestimmen der Minimalform mittels Quine-McCluskey

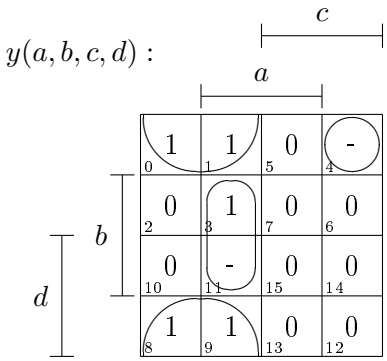
Gegeben ist eine vollständige Funktionsdefinition bei großen Funktionen mit mehr als 6 Variablen, die es zu minimieren gilt.

- 1. Quinesche Tabelle
  - 0-ter Ordnung
    - Hier werden die Minterme der Funktionstabelle aufgelistet. Man hat dann eine Spalte "Nr.", die die Minterm-Nummer angibt, und eine Spalte, in der die Variablenbelegung binär kodiert angibt.
  - n-ter Ordnung
    - Aus vorigen Ordnungen werden Zusammenstellungen von Termen gesucht, die sich als neuer Term mit Don't-Cares darstellen lassen, welcher alle Ursprungsterme überdeckt. Dabei werden die Terme, die davon überdeckt werden, gestrichen.
    - Man sucht in den Ordnungen 1 bis $n-2$ nur nach Termen, die nicht gestrichen wurden, in Ordnung $n-1$ auch nach schon gestrichenen Termen (da diese in der aktuellen Iteration erst gestrichen wurden und sich mit anderen Termen ebenfalls kombinieren lassen können).
    - Die neue Nummer des kombinierten Terms ist die Zusammenstellung der Nummern der ursprünglichen Term. Wenn Terme 3 und 5 kombiniert werden, ist die neue Nummer $3,5$. Wenn Terme $1,2,6,7$ und $4,5,8,9$ kombiniert werden, ist die neue Nummer $1,2,4,5,6,7,8,9$.
  - Man bricht ab, wenn die Terme sich nicht weiter zusammenfassen lassen. Die am Ende nicht gestrichenen Terme sind die Primimplikanten.
- 2. Quinesche Tabelle
  - Hier steht in jeder Zeile eins der zuvor gefundenen Primimplikanten, in jeder Spalte eine Nummer der Minterme $(1,2,6,7,...)$.
  - Die Überdeckungstabelle ist für eine Funktion dann vollständig definiert, wenn die Primimplikanten 2^n Minterme überdecken.
  - Reduzierung der Tabelle:
    - (Falls normale Überdeckungstabelle:) Zeilen von Implikanten, die durch andere Primimplikanten überdeckt werden, streichen (Zeilendominanz).
    - Zeilen von Primimplikanten streichen.
    - Spalten von Mintermen, die durch andere Minterme überdeckt werden, streichen (Spaltendominanz).
  - Aufstellen der Überdeckungsfunktion:
    - Für jede Spalte (Minterme) bilde die Veroderung aller diese überdeckende Primimplikanten, und verunde alles. Dabei trotzdem Primimplikanten reinnehmen.

– Die DNF ist TODO: ?

2.6 Rechenschema: Würfelminimierung mit Consensus-Verfahren

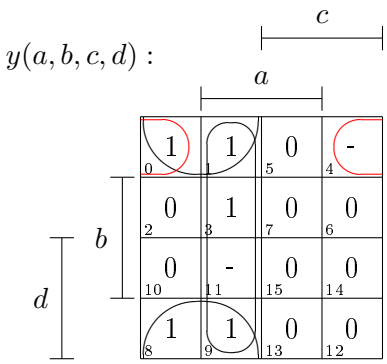
Gegeben ist eine Funktion durch ihr KV-Diagramm, es sollen minimale Würfel gebildet werden.



In einer Liste werden alle Würfel aufgelistet, und dann wiederholt versucht, einzelne Würfel durch Zusammenfassen zu streichen.

Nr.	gebildet aus	Würfel	gestrichen wegen
1		$-, 0, 0, -$	
2		$-, , 1, 1$	$\subset 4$
3		$0, 1, 0, 0$	$\subset 5$
4	$2, 1$	$-, 0, -, 1$	
5	$3, 1$	$0, -, 0, 0$	
	$5, 4$	$0, 0, 0, -$	$\subset 1$

Die nicht gestrichenen Würfel bilden eine minimale Würfelüberdeckung:



Für eine Minimalform kann der rote Würfel $(0, -, 0, 0)$ auch weggelassen werden, da die Einsstellen schon überdeckt werden.

2.7 Rechenschema: Bündelminimierung

Für mehrere KV-Diagramme mehrerer Funktionen sucht man Primkoppelterme, also Terme, die in allen Funktionen Implikante sind, um Realisierungskosten zu sparen.

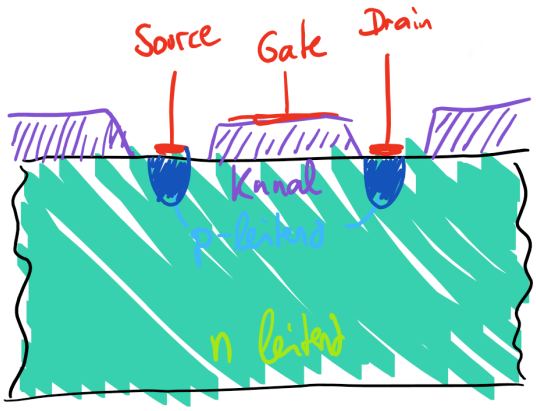
2.8 Rechenschema: NAND/NOR Konversion

- Eine Funktion in DNF (z.B. $y = \bar{a}bc \vee a\bar{b}c$) soll in NAND Form gebracht werden.
  - Doppelte Negation: $\overline{\overline{\bar{a}bc \vee a\bar{b}c}}$
  - DeMorgan: $\overline{\overline{\bar{a}bc} \wedge \overline{a\bar{b}c}}$
  - Konjunktionen zu NANDs umwandeln: $\overline{\bar{a}bc} \barwedge \overline{a\bar{b}c}$
  - In NAND Schreibweise darstellen: $\mathrm{NAND}_2(\mathrm{NAND}_3(\bar{a}bc), \mathrm{NAND}_3(a\bar{b}c))$
- Eine Funktion in KNF (z.B. $y = (\bar{a} \vee b \vee c) \wedge (a \vee b \vee \bar{c})$) soll in NAND Form gebracht werden.
  - Einfache Negation: $\overline{y} = \overline{(\bar{a} \vee b \vee c) \wedge (a \vee b \vee \bar{c})}$
  - Doppelte Negation der Terme: $\overline{y} = \overline{\overline{(\bar{a} \vee b \vee c)} \wedge \overline{(a \vee b \vee \bar{c})}}$
  - DeMorgan auf den Termen: $\overline{y} = \overline{(a \wedge \bar{b} \wedge \bar{c}) \wedge (\bar{a} \wedge \bar{b} \wedge c)}$ (Dabei werden die Literale invertiert!)
  - Konjunktionen in NANDs umwandeln: $\overline{y} = (a \barwedge \bar{b} \barwedge \bar{c}) \barwedge (\bar{a} \barwedge \bar{b} \barwedge c)$
  - In NAND-Schreibweise darstellen: $\overline{y} = \mathrm{NAND}_2(\mathrm{NAND}_3(a, \bar{b}, \bar{c}), \mathrm{NAND}_3(\bar{a}, \bar{b}, c))$
  - Invertieren: $y = \overline{y} \barwedge \overline{y}$
- Eine Funktion in DNF (z.B. $y = \bar{a}bc \vee a\bar{b}c$) soll in NOR Form gebracht werden.
  - Einfache Negation: $\overline{y} = \overline{\bar{a}bc \vee a\bar{b}c}$
  - Doppelte Negation: $\overline{y} = \overline{\overline{\bar{a}bc} \vee \overline{a\bar{b}c}}$
  - DeMorgan auf den Termen: $\overline{y} = \overline{a \vee \bar{b} \vee \bar{c} \vee \overline{\bar{a} \vee b \vee \bar{c}}}$ (Dabei werden die Literale invertiert!)
  - Disjunktionen in NORs umwandeln: $\overline{y} = (a \barvee \bar{b} \barvee \bar{c}) \barvee (\bar{a} \barvee b \barvee \bar{c})$
  - In NOR-Schreibweise darstellen. $\overline{y} = \mathrm{NOR}_2(\mathrm{NOR}_3(a, \bar{b}, \bar{c}), \mathrm{NOR}_3(\bar{a}, b, \bar{c}))$
  - Invertieren: $y = \overline{y} \barvee \overline{y}$
- Eine Funktion in KNF (z.B. $y = (\bar{a} \vee b \vee c) \wedge (a \vee b \vee \bar{c})$) soll in NOR Form gebracht werden. TODO

Bemerkung. NOR_k ist genau dann wahr, wenn alle Parameter nicht wahr sind.
NAND_k ist genau dann wahr, wenn mindestens ein Parameter nicht wahr ist.

3 MOSFET

3.1 p MOSFET

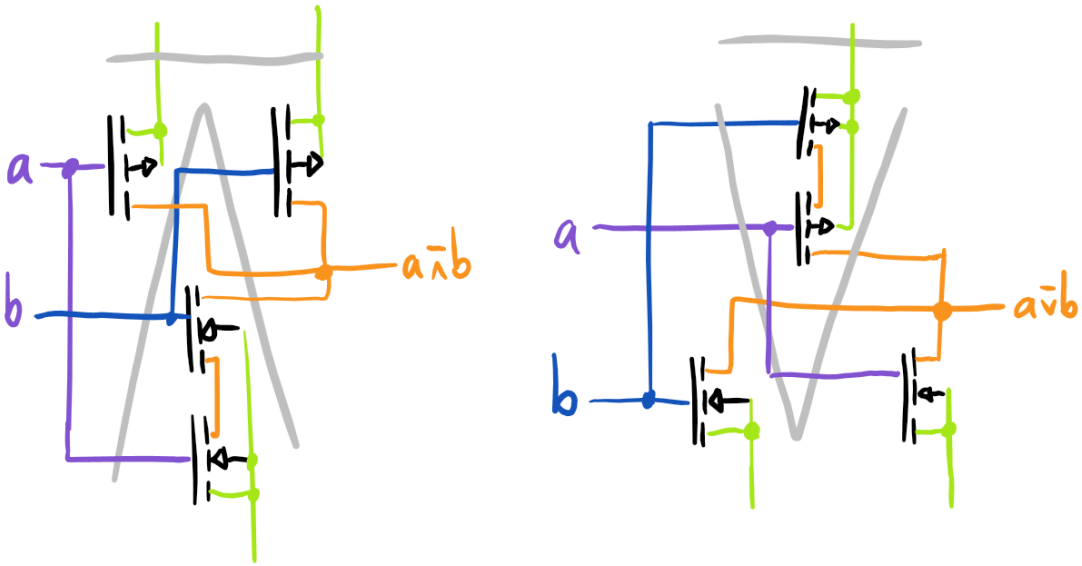


n-MOSFET ist genau umgekehrt.

3.2 cMOS

3.2.1 Grundlagen

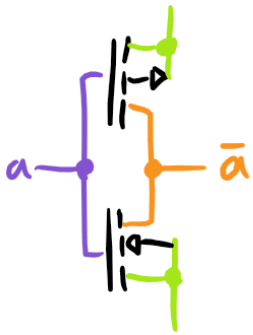
3.2.2 cMos NAND und NOR Schalter



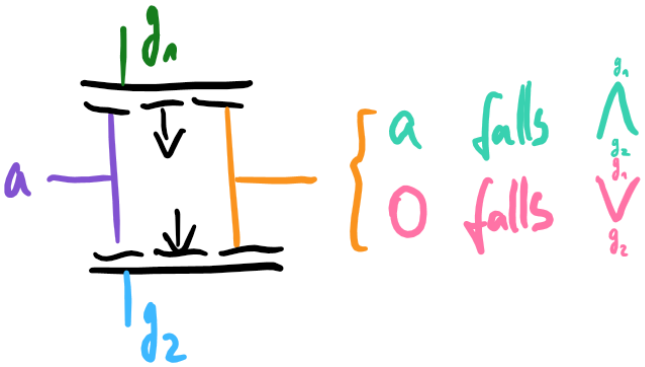
Herleitung:

1. Großes $\overline{\wedge}$ oder $\overline{\vee}$ zeichnen, Transistoren einzeichnen.
2. Erste Eingangsvariable geht an die zwei linkesten äußersten Transistoren, zweite Eingangsvariable an die verbleibenden Transistoren. (blau, lila)
3. Bei den äußersten Transistoren gehen die äußeren Source/Drain-Wege direkt zu V_{dd} bzw GND, genauso alle Pfeilausgänge (grün).
4. Die parallel geschalteten Transistoren werden zwischen den beiden durch ein Kabel verbunden. Alle verbleibenden Ausgänge gehen zusammen zum Output (orange).

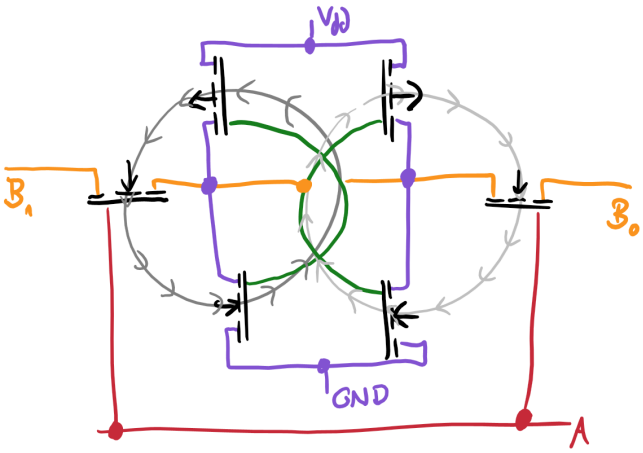
3.2.3 cMos Inverter



3.2.4 Transmission gate



3.2.5 Statische CMos Speicherzelle



Herleitung:

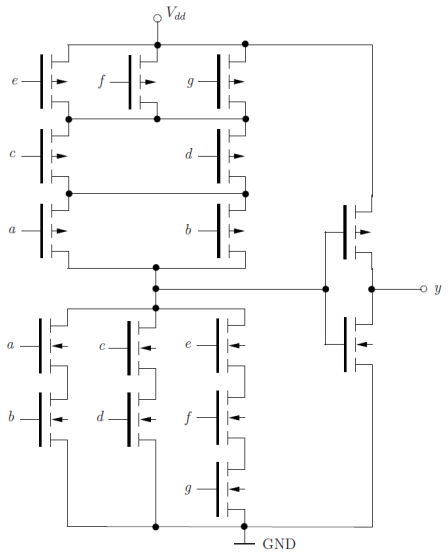
1. Kreise malen, sodass beide innen nach oben gehen.
2. 6 Transistoren in Pfeilrichtung einzeichnen.
3. Innen Leitungen entsprechend den Pfeilen zeichnen (grün).
4. Leitungen nach außen über Transistoren laufen lassen (orange).
5. Äußere Transistoren nach unten mit A verbinden (rot).
6. Viereck in der Mitte zeichnen (lila).

3.2.6 Disjunktive Funktionen

1. Im oberen P-Netz werden alle Minterme aus der disjungierten Form dargestellt, wobei alle Minterme in Reihe geschaltet sind und die Literale innerhalb der Minterme parallel geschaltet sind.

2. Im unteren N-Netz werden alle Minterme der disjungierten Form parallel geschaltet, wobei alle Literale eines Minterms in Reihe geschaltet werden.

3. Am Ende wird das ganze invertiert.



3.2.7 Konjungierte Funktionen

Das P-Netz und das N-Netz bauen sich analog zu disjunktiven Funktionen auf (ersetze Minterme im Text oben durch Maxterme), außer dass die Literale alle negiert an den Eingängen anliegen und am Ende kein Inverter kommt.

3.3 Relevante Komponenten

- Gatter (&, ≥ 1, $= 1$...)
- Multiplexer (MUX)
- Demultiplexer (DX) bzw. Decoder
- Speicher

  ROM Read Only Memory

  PROM Programmable ROM

  EPROM Erasable PROM

  RAM Random Access Memory

4 Grundlagen von C

4.1 Datentypen

char Charakter, meist 1 Byte

int Integerzahl, meist 2 oder 4 Byte

float Gleitkommazahl, meist 4 Byte

double Gleitkommazahl doppelter Genauigkeit, meist 8 Byte

4.2 Speicherklassen

auto lokale Variablen, gelten nur in einem Code-Block.

register informiert den Compiler, die lokale Variable in CPU-Registern zu behalten. Für häufig verwendete Variablen.

static lässt statischen Speicherplatz zuweisen. Ermöglicht globale Variablen mit lokaler Sichtbarkeit, um z.B. Zustand zwischen Funktionsaufrufen zu sichern.

extern deklariert globale Variablen aus anderen Programm-Modulen.

4.3 Zeiger

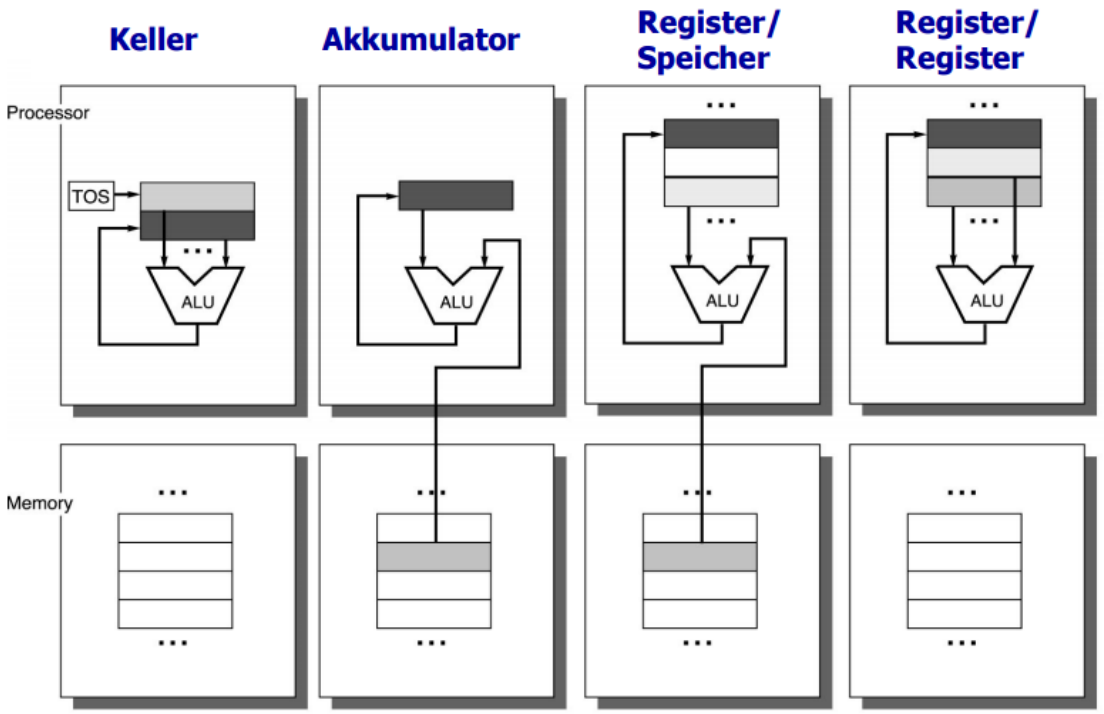
int a = 3;	Deklariert Integer a mit Wert 3.
int *p;	Deklariert Zeiger vom Typ Int.
p = &a;	Der Wert von p wird überschrieben mit &a, der Adresse von a. Damit zeigt p jetzt auf die Variable a.
int b = *p + 1;	p wird dereferenziert, also die Referenz wird aufgelöst und der Wert 2, auf den p zeigt, wird, um eins erhöht, in b gespeichert.
*p = 1;	p wird dereferenziert, und das Ziel, also a, wird mit dem Wert 1 überschrieben.

5 Befehlssatzarchitektur

von-Neumann-Architektur Daten und Code liegen in gemeinsamen Speicher

Harvard-Architektur Speicher ist nach Daten und Code getrennt

5.1 Ausführungsmodelle



5.2 ISA Datenformate

Byte 8 Bit

Halbwort 16 Bit

Wort 32 Bit

Doppelwort 64 Bit

TODO: BCD hat in gepackter Form mehrere Ziffern pro Byte, in ungepackter Form eine Ziffer pro Byte und mehr unnötigen Informationen.

5.3 Speicheranordnung

Little Endian Ordering Daten in Formaten, die größer als ein Byte sind, haben das niedrigst-wertigste Byte an der niedrigsten Stelle. Beispiel für das Wort *abcd*:

Adresse	0	1	2	3
Wert	a	b	c	d

Big Endian Ordering Daten in Formaten, die größer als ein Byte sind, haben das niedrigst-wertigste Byte an der höchsten Stelle. Beispiel für das Wort *abcd*:

Adresse	0	1	2	3
Wert	d	c	b	a

5.4 Befehlsformate von MIPS

Typ R Register-Register-Befehle (add, sub, ...)

opcode	rs	rt	rd	shiftamount	funct
31-26	25-21	20-16	15-11	10-6	5-0

Typ I Immediate-Register-Befehle (addi, lw, ...)

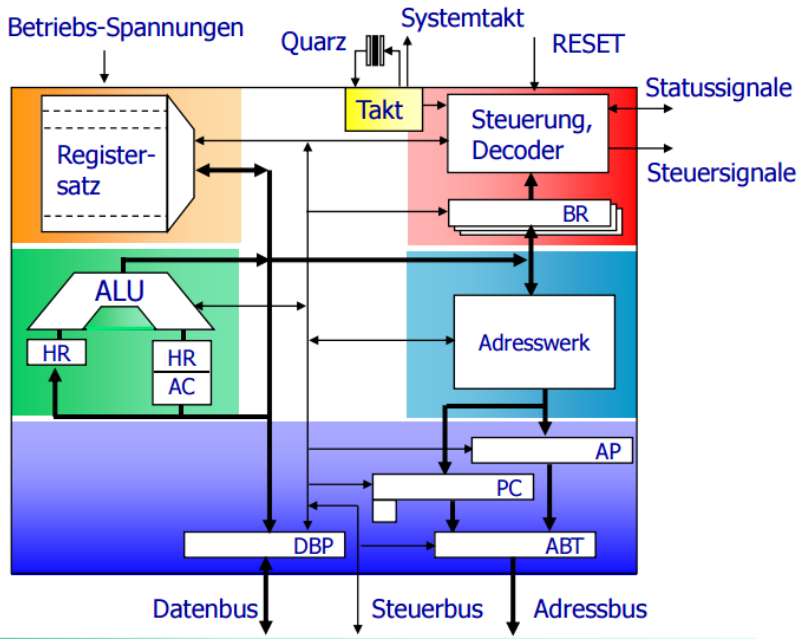
opcode	rs	rt	Immediate/Adresse
31-26	25-21	20-16	15-0

Typ J Jump (j, jal, ...)

opcode	Target Adresse
31-26	25-0

5.5 von-Neumann Bus

- HR: Hilfsregister
- AC: Accumulator
- BR: Befehlsregister
- AP: Adress Puffer
- PC: Program Counter
- ABT: Adressbus Puffer
- DBP: Datenbus Puffer



TODO: MIMA Mikroprogrammierung

5.6 RISC vs CISC

CISC	RISC
Komplexe Befehle, Ausführung in mehreren Taktzyklen	Einfache Befehle, Ausführung in einem Taktzyklen
Jeder Befehl kann auf den Speicher zugreifen	Nur Lade- und Speicherbefehle greifen auf den Speicher zu
Wenig Pipelining	Intensives Pipelining
Befehle werden von einem Mikroprogramm interpretiert	Befehle werden durch festverdrahtete Hardware ausgeführt
Befehlsformat variabler Länge	Alle Befehle mit fester Länge
Die Komplexität liegt im Mikroprogramm	Die Komplexität liegt im Compiler
Einfacher Registersatz	Mehrere Registersätze

Skalarer RISC-Prozessor Entwurfsziel: Etwa eine Befehlsausführung pro Takt.

Superskalarer RISC-Prozessor Pro Takt mehrere Befehle gleichzeitig von verschiedenen Ausführungseinheiten.

5.7 Bestandteile des Prozessors

Steuerwerk Taktgeber legt Systemtakt fest. Entweder mikroprogrammiert, interpretiert dann die Befehle, oder fest verdrahtet. Holphase, Dekodierphase, Ausführphase. Steuerregister ermöglicht Manipulationen, z.B.: Interrupt enable Bit, User/System Bit, Trace Bit (Debugging), Decimal Bit (Dual oder BCD). Befehlregister für Prefetching.

Rechenwerk Statusregister informiert Steuerwerk über Ablauf des Ergebnisses (Carry, Overflow, Zero, Sign...). HR = Hilfsregister, AC = Akkumulatoren. Meist ohne Akkumulator, stattdessen Register.

Registersatz Cachen von Operanden. Universelle- (Daten- und Adressregister (Dieses aus Basisregister für Anfangsadresse und Indexregister für Offset)) und Spezialregister (wie Befehlszähler, Steuerregister...).

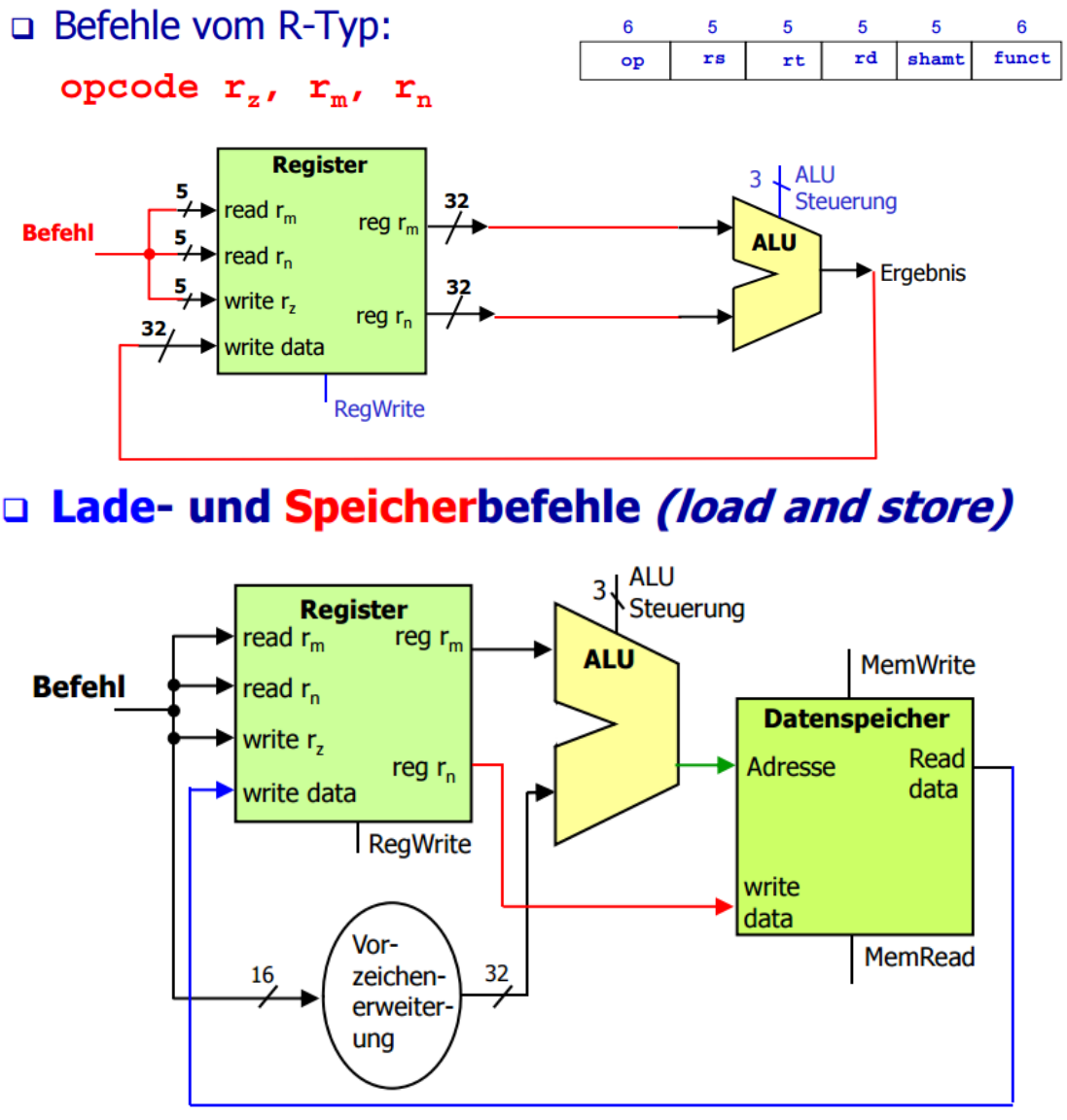
Adresswerk Berechnet entsprechend Steuerwerk Adressen von Befehlen.

6 Pipelining

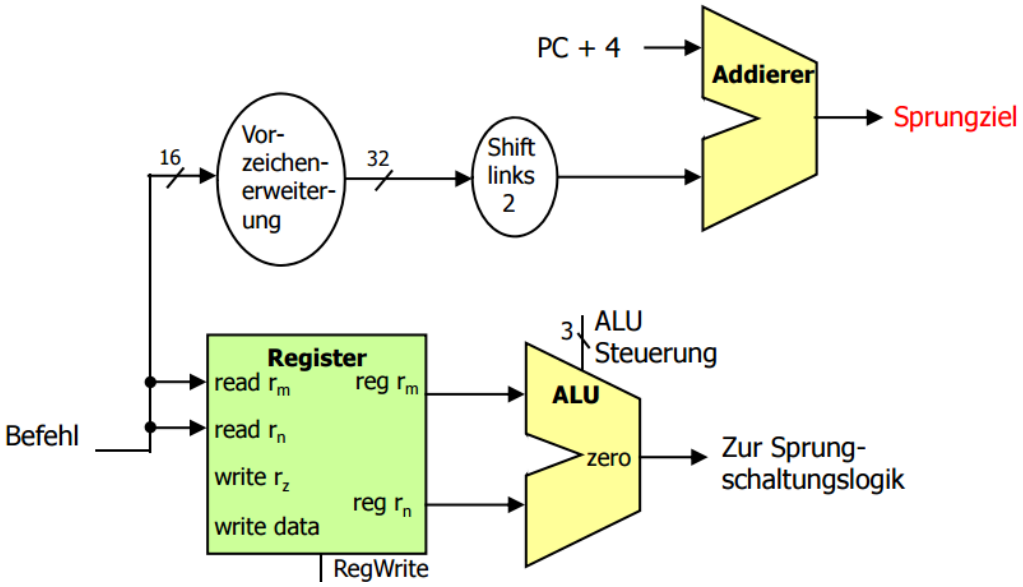
Speedup Die Leistungssteigerung durch Pipelining beträgt $s = \frac{n \cdot k}{n+(k-1)}$, wenn n Befehle in einer Pipeline mit k Stufen durchgeführt werden. Ohne Pipeline sind nk Takte notwendig, somit $s = 1$, mit Pipelining $n + (k-1)$, n für den Start aller Befehle und $k-1$ zur Fertigstellung des letzten Befehls.

Durchsatz gibt die Anzahl der Befehl pro Zeitraum an. $D = \frac{n}{(n+(k-1)) \cdot \tau} = s \cdot \frac{1}{\tau \cdot k}$ mit τ = Taktdauer.

6.1 Befehlsabarbeitung

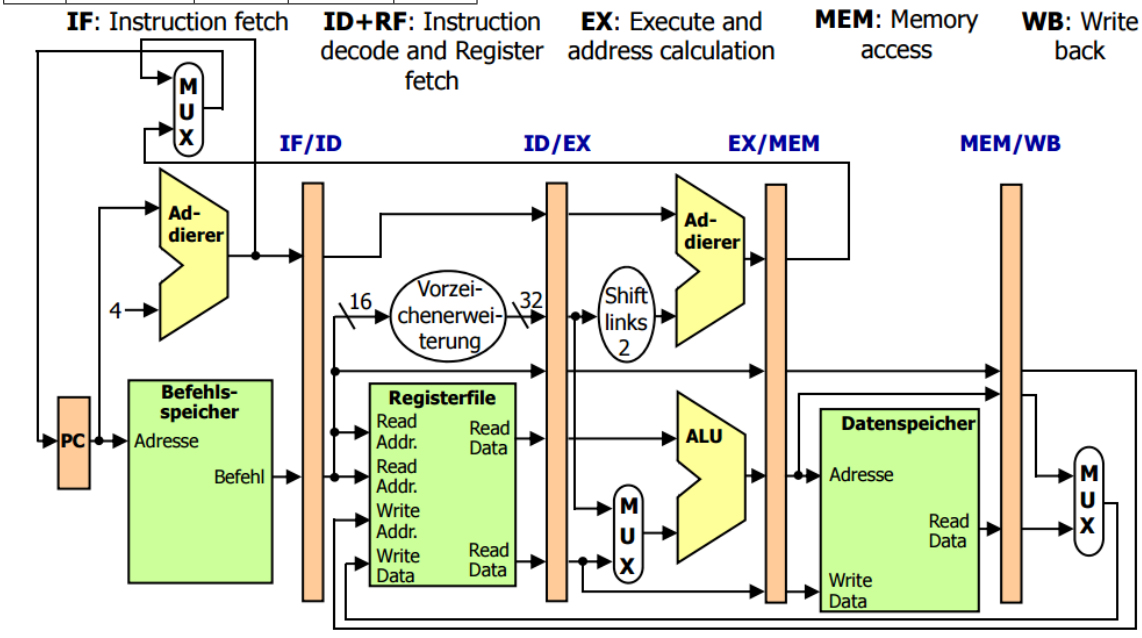


❑ Verzweigungsbefehle



6.2 DLX-Pipelinestufen

IF	ID+RF	EXE	MEM	WB



6.3 Konflikte

Datenkonflikte durch Datenabhängigkeiten

echte Datenabhängigkeit δ^t Lesen nach schreiben, Richtung: $\searrow$

- Einziger Konflikt, der bei DLX auftreten kann.
- Lösung ohne Forwarding: Zwei nachfolgende $NOPS$.
- Lösung mit Forwarding: Ein nachfolgender NOP nach einem Load-Befehl, sonst kein NOP.

Gegenabhängigkeit δ^a Schreiben nach lesen, Richtung: $\swarrow$

Ausgabeabhängigkeit δ^o Schreiben nach Schreiben, Richtung: $\downarrow$

Struktur- und Ressourcenkonflikte Zwei Pipeline-Stufen greifen auf eine Ressource (z.B. ALU) zu, die nicht parallel benutzt werden kann.

- Lösung bei DLX: $NOPS$, Übertaktung oder Ressourcenreplizierung.

Steuerflusskonflikte wenn bei bedingten Sprüngen die Bedingung noch nicht aufgelöst wurde oder Zieladresse des nächsten Befehls noch nicht berechnet ist.

- Sicheres Auflösung des Konflikts bei DLX: Drei nachfolgende $NOPS$. (Verzögerte Sprungtechnik) TODO: Einer falls mit Forwarding? Eher nicht.
- Alternativ:
  - Statische Befehlsanordnung: Befehle, die vor dem Sprung ausgeführt werden, aber keinen Einfluss auf die Sprungrichtung haben, werden in die Verzögerungszeitschlitze geschoben.
  - Pipeline-Leerlauf (einfach, aber ineffizient): Hardware erkennt Verzweigungsbefehle in ID-Stufe und lädt danach keine weiteren Befehle in die Pipeline bis zur Entscheidung. Der bereits geladene Befehl muss gelöscht werden.
  - Spekulation: Mache Vorhersage, bei falscher Vorhersage müssen geladene Befehle annuliert werden. Vorhersagetechniken: "always not taken", "always taken" (statisch, einfach aber unflexibel), Branch Target Buffer (dynamisch, komplexe Hardware, betrachtung von zeitlicher und räumlicher Korrelation).

7 Caches

Hit-Rate $= \frac{\#Treffer}{\#Zugriffe}$.

Mittlere Zugriffszeit $t_{Access} = (Hit - Rate) \cdot t_{Hit} + (1 - Hit - Rate) \cdot t_{Miss}$ mit $t_{Hit} =$ Cachezugriffszeit und $t_{Miss} =$ RAM-Zugriffszeit.

7.1 Schreibzugriffsverfahren

Durchschreibverfahren (write trough policy) geänderte Daten werden in Cache und RAM überschrieben.

Gepuffertes Durchschreibverfahren (buffered write trough policy) geänderte Daten werden in Cache und Schreibpuffer überschrieben, Letzterer aktualisiert (asynchron den RAM).

Rückschreibverfahren (write back policy) geänderte Daten werden nur im Cache überschrieben und durch Dirty-Bit markiert. RAM wird bei Verdrängung angepasst.

7.2 Arten von Caches

Voll-assoziativ Vollparalleler Vergleich aller Adressen des Caches in einem Taktzyklus. Optimale Ausnutzung und Laufzeit, aber hoher Hardware-Aufwand. (max-fach Satzassoziativ)

Direct Mapped Jede Stelle des RAMs erhält einen eindeutigen Cache-Platz. Nur ein Vergleicher, dafür häufige Kollisionen von Blöcken $0, 64, ...$ (1-fach Satzassoziativ)

n-fach Satz-assoziativ Mehrere Cachezeilen werden zu Sätzen zusammengefasst, die mehrere Cache-Plätze haben.

7.3 Ersetzungsstrategien

Ersetzungsstrategien geben an, welcher Teil des Caches nach einem Miss durch neue Daten überschrieben werden.

Bélády's Algorithmus Ersetze den Eintrag, der am längsten nicht mehr gebraucht werden wird (praktisch nicht anwendbar)

FIFO, LIFO .

LRU Least recently used (MRU: Most recently used)

Pseudo-LRU Approximation von LRU, einfachere Implementierung.

LFU least frequently used (MFU)

Random description

7.4 Ursachen für Fehlzugriffe

Compulsory Miss Erster Zugriff auf den Cache-Block, der sich noch nicht im Cache befindet.

Capacity Miss Cache-Speicher voll, Verdrängung notwendig.

Conflict Miss Mehrere Daten bildet auf dasselbe Speichersegment ab.

Coherence Miss Andere CPU hat eine Adresse überschrieben und den Cache-Eintrag ungültig gemacht.

7.5 Hauptspeicher-Adresse

Die Hauptspeicheradresse habe eine Länge von n Bits, der Cache habe $k = \frac{\#Cachegroesse}{\#Blockgroesse}$ Cachezeilen.

Falls der Cache m-fach satzassoziativ ist, existieren $\frac{k}{m}$ Sätze (vollassoziativ: $m = k \Rightarrow 1$ Satz, direct mapped: $m = 1 \Rightarrow k$ Sätze). Dann sieht die Hauptspeicher-Adresse wie folgt aus:

Tag	Satzindex	Wortadresse im Block	Byteadresse im Wort
$\leftarrow Rest \rightarrow$	$\leftarrow \log_2 \#\text{Sätze} \rightarrow$	$\leftarrow \log_2 \#\text{Blockgröße} \rightarrow$	$\leftarrow 2 \rightarrow$

Anzahl der Vergleicher: # Anzahl Sätze.

7.6 Rechenschema: Wohin wird ein Block abgebildet?

$$\begin{aligned} \text{Blocknummer} &= \text{Hauptspeicheradresse } \textit{mod} \text{ Blockgröße} \\ \text{Zeilennummer} &= \text{Blocknummer } \textit{mod} \text{ Zeilenanzahl} \\ \text{Satznummer} &= \text{Blocknummer } \textit{mod} \text{ Satzanzahl} \end{aligned}$$

8 Speicherverwaltung

8.1 Ersetzungsstrategien

FIFO .

LIFO .

LRU Least recently used

LFU Least frequently used

LRD Least reference density, niedrigste $\frac{\#\text{Zugriffe}}{\#\text{Vergangene Zeit seit Einlagerung}}$.

8.2 Segmentbasierte Speicherverwaltung

Virtuelle Adresse wird zweigeteilt, linker Teil der Länge n gibt die Kennung des Segments an, rechter Teil der Länge m den Byte-Abstand zum Segmentanfang.

Damit sind maximal 2^n virtelle Segmente möglich, jeweils maximal der Größe 2^m.

9 MIPS

Achtung: MARS verwendet Little Endian. dh. def : $.word$ $0x1234abcd$ sorgt für $def \rightarrow cd, def + 1 \rightarrow ab, ...$, MIPS kann aber beides.

9.1 Direktiven

.align n Richtet folgende Daten an Größe n aus.

.space n Allokiert n Bytes im aktuellen Datensegment.

.ascii(z) string Allokiert Speicher und legt *string* dort ab, bei *.asciiz* mit angehängtem Null-Byte.

.byte ..., .half ..., .word ..., .float ..., .double ... Allokiert Speicher und legt ... dem Datentyp entsprechend ab.

.globl symbol Deklariert die Marke *symbol* als global, sodass sie von anderen Dateien referenziert werden kann.

.extern symbol size TODO: ..

TODO: Weitere Direktiven...

9.2 Arrays

Elemente von Arrays haben immer eine gewisse Breite. Bei einer Breite von 4 muss man, wenn in Register $\$t0$ der Start des Arrays steht, z.B. $lw \$t1, 8(\$t0)$ benutzen, um in $\$t1$ das zweite Array-Element zu laden.

Common MIPS instructions

Notes: *op, funct, rd, rs, rt, imm, address, shamt* refer to fields in the instruction format
PC is assumed to point to the next instruction, **Mem** is the byte addressed main memory

Assembly Instruction	Instr Format	op op/funct	Meaning	Comments
add *$rd, $rs, $rt*	R	0/32	*$rd = $rs + $rt*	Add contents of two registers
sub *$rd, $rs, $rt*	R	0/34	*$rd = $rs - $rt*	Subtract contents of two registers
addi *$rt, $rs, imm*	I	8	*$rt = $rs + imm*	Add signed constant
addu *$rd, $rs, $rt*	R	0/33	*$rd = $rs + $rt*	Unsigned, no overflow
subu *$rd, $rs, $rt*	R	0/35	*$rd = $rs - $rt*	Unsigned, no overflow
addiu *$rt, $rs, imm*	I	9	*$rt = $rs + imm*	Unsigned, no overflow
mfc0 *$rt, $rd*	R	16	*$rt = $rd*	*rd* = coprocessor register (e.g. epc, cause, status)
mult *$rs, $rt*	R	0/24	Hi, Lo *= $rs * $rt*	64 bit signed product in Hi and Lo
multu *$rs, $rt*	R	0/25	Hi, Lo *= $rs * $rt*	64 bit unsigned product in Hi and Lo
div *$rs, $rt*	R	0/26	Lo *= $rs / $rt,* Hi *= $rs* mod *$rt*	
divu *$rs, $rt*	R	0/27	Lo *= $rs / $rt,* Hi *= $rs* mod *$rt* (unsigned)	
mfhi *$rd*	R	0/16	*$rd =* Hi	Get value of Hi
mflo *$rd*	R	0/18	*$rd =* Lo	Get value of Lo
and *$rd, $rs, $rt*	R	0/36	*$rd = $rs & $rt*	Logical AND
or *$rd, $rs, $rt*	R	0/37	*$rd = $rs | $rt*	Logical OR
andi *$rt, $rs, imm*	I	12	*$rt = $rs & imm*	Logical AND, unsigned constant
ori *$rt, $rs, imm*	I	13	*$rt = $rs | imm*	Logical OR, unsigned constant
sll *$rd, $rs, shamt*	R	0/0	*$rd = $rs << shamt*	Shift left logical (shift in zeros)
srl *$rd, $rs, shamt*	R	0/2	*$rd = $rs >> shamt*	Shift right logical (shift in zeros)
lw *$rt, imm($rs)*	I	35	*$rt = Mem[$rs + imm]*	Load word from memory
sw *$rt, imm($rs)*	I	43	*Mem[$rs + imm] = $rt*	Store word in memory
lbu *$rt, imm($rs)*	I	37	*$rt = Mem[$rs + imm]*	Load a single byte, set bits 8-31 of *$rt* to zero
sb *$rt, imm($rs)*	I	41	*Mem[$rs + imm] = $rt*	Store byte (bits 0-7 of *$rt*) in memory
lui *$rt, imm*	I	15	*$rt = imm * 2^{16}*	Load constant in bits 16-31 of register *$rt*
beq *$rs, $rt, imm*	I	4	if*($rs==$rt)* PC *=* PC *+ imm* (PC always points to next instruction)	
bne *$rs, $rt, imm*	I	5	if*($rs!=$rt)* PC *=* PC *+ imm* (PC always points to next instruction)	
slt *$rd, $rs, $rt*	R	0/42	if*($rs<$rt) $rd = 1;* else *$rd = 0*	
slti *$rt, $rs, imm*	I	10	if*($rs<imm) $rt = 1;* else *$rt = 0*	
sltu *$rd, $rs, $rt*	R	0/43	if*($rs<$rt) $rd = 1;* else *$rd = 0* (unsigned numbers)	
sltiu *$rt, $rs, imm*	I	11	if*($rs<$rt) $rd = 1;* else *$rd = 0* (unsigned numbers)	
j *destination*	J	2	PC *= address*4*	Jump to *destination*, *address = destination/4*
jal *destination*	J	3	*$ra =* PC*;* PC *= address*4* (Jump and link, *address = destination/4*)	
jr *$rs*	R	0/8	PC *= $rs*	Jump to address stored in register *$rs*
beqz $rs, label	Pseudo		If(*$rs==0*) then goto label	See also: *bnez, bgez, bgtz, blez, bltz*
bge $rs, $rt, label	Pseudo		If(*$rs≥$rt)* then goto label	See also: *bgt, ble, blt* (add *u* for *unsigned*,eg *bgeu*)
la $rd, label	Pseudo		*$rd* = label	Assign the address of the label to register *$rd*

MIPS Instruction formats

Format	Bits 31-26	Bits 25-21	Bits 20-16	Bits 15-11	Bits 10-6	Bits 5-0
R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	imm		
J	op	address				

MIPS registers

register		usage	writable	caller-	callee-
-name	-nr			saved	
$zero	0	stores the value 0, **do not write to it!**	yes	-	-
$at	1	reserved for assembler	no	-	-
$v0 - $v1	2 - 3	stores function results, more than 2 results → stack	yes	x	-
$a0 - $a3	4 - 7	function arguments, if more than 4 arguments are needed → stack	yes	x	-
$t0 - $t9	8 -15, 24-25	temporary variables	yes	x	-
$s0 - $s7	16 - 23	long-living variables	yes	-	x
$k0 - $k1	26 - 27	reserved for kernel	no	-	-
$gp	28	points to middle of a 64K block in the data segm.	no	-	-
$sp	29	stack pointer (top of stack)	yes	-	-
$fp	30	frame pointer (beginning of current frame)	yes	-	x
$ra	31	return address	yes	-	x
internal, not directly accessible, registers					
Hi, Lo		stores the result of mult/div operations (use *mflo*, *mfhi* to access these registers)			
PC		contains the address of the next instruction to be fetched			
status		register 12 in coprocessor 0, stores interrupt mask and enable bits (use *mfc0*)			
cause		register 13 in coprocessor 0, stores exception type and pending interrupt bits (use *mfc0*)			
epc		register 14 in coprocessor 0, stores address of instruction causing exception (use *mfc0*)			

operating system functions

function	code in $v0	arguments	result
print_int	1	$a0	
print_float	2	$f12	
print_double	3	$f12/13	
print_string	4	$a0 contains the start address of the string	
read_int	5		$v0
read_float	6		$f0
read_double	7		$f0/1
read_string	8	$a0 contains the destination address of the string, $a1 ist maximum length	string starting at $a0
sbrk	9	$a0 contains needed size	$v0 contains start address of the memory area
exit	10		

MIPS Assembler Syntax

```
                                    # This is a comment
        .data                       # Store following data in the data segment

items:                              # This is a label connected to the next address in the
                                    # current segment
        .word 1, 2                  # Stores the values 1 and 2 in next two words

servus: .ascii “servus!!”           # Stores a not terminated string in memory
hello:  .asciiz “hello”             # Stores ‘\0’ terminated string in memory (hello+ \0’)
                                    # Note that printing the label servus will give you
                                    # the text “servus!!hello”
.text                               # Store following instructions in the text segment
.globl main                         # the label main is the entry point of our program
main:   la $t0, servus              # An instruction connected to a label (e.g. for loops)
        addi $t0, $zero, ‘a’        # assigns the ascii value of ‘a’ to $t0
```

10 Y-Diagramm

Herleitung:

Wie Wo Was

System
Architektur
Register-Transfer
Logik
Elektronik

SARLE

WO

Systempartitionierung
Cluster
Floorplan
Platzierte Zellen
Symbolisches Layout

SiCFiPS

WAS

Leistungsanforderungen
Algorithmen
Datenfluss, Steuerfluss
Boolsche Gleichungen
Differentialgleichungen

LeADaBoDi

WIE

Netz Blockbuster im RTL über logische Elektronik

Netzwerk
Blockschaltbild
RT - Diagramm
Logiknetzliste
elektrisches Schaltbild