

Grundlagen der Rechnerarchitektur

Felix Leitl

28. Juli 2023

Inhaltsverzeichnis

Zahlensysteme	3
Präfixe	3
Multiplikation und Division mit Zweierpotenzen	3
Rechnerarchitektur	3
Endo- vs. Befehlsarchitektur	3
Von-Neumann, URA und ISA	3
Befehlszyklus	4
Assemblertheorie	4
RiscV-Befehlssatz	4
Speicherbereiche	4
Stack	5
Lesen vom Stack	5
Schreiben auf den Stack	5
Speicher freigeben	5
Mikroprogrammierung	5
Aufbau	5
Horizontsal vs. vertikal	6
Horizontal	6
Vertikal	6
Beispiel	6
Befehlssatzarchitektur	7
Register-Register	7
Register-Memory	7
Akkumulator	7
Stack	7
Endianess and Alignment	7
Endianess	7
Alignment	8

Speicherhierarchie	8
Performance Gap	8
Räumliche und Zeitliche Lokalität	8
Zeitlich	8
Räumliche	9
Cache	9
Grundbegriffe	9
Cache Hit	9
Cache Miss	9
Platzierung und Identifikation	9
Cacheblöcke	9
Byteoffset	10
Mengen / Index	10
Tag	10
Organisationsformen	10
Vollassoziativ	10
Direktabbildend	10
N-fach assoziativ	11
Vor und Nachteile	11
Ersetzungsstrategien	11
Random	11
LRU	11
LFU	11
FIFO	11
Aktualisierungsstrategien	11
Write through	12
Write back	12
Arbeitsspeicher	12
Pipelining	12
Instruktionsparallelismus	12
Threadparallelismus	12
Grafikkarten	12
Speicherverwaltung	12

Zahlensysteme

Präfixe

Kilo	10^3	Kibi	2^{10}
Mega	10^6	Mebi	2^{20}
Giga	10^9	Gibi	2^{30}
Tera	10^{12}	Tebi	2^{40}
Peta	10^{15}	Pebi	2^{50}

Multiplikation und Division mit Zweierpotenzen

Bei Multiplikation einen shift nach links, bei Division einen shift nach rechts:

$$0xAB \cdot 2^2 = 10101011 \ll 2 = 1010101100 = 0x2AC$$

$$0xAB/2^2 = 10101011 \gg 2 = 00101010 = 0x2A$$

Rechnerarchitektur

Endo- vs. Befehlsarchitektur

Externe Sicht(Befehlsarchitektur): Was muss nach außen hin sichtbar sein, damit man den Computer programmieren kann?

Interne Sicht(Endoarchitektur): Wie werden die Funktionalitäten intern realisiert?

Von-Neumann, URA und ISA

7 Eigenschaften des URAs/von-Neumann Architektur:

1. Rechner besteht aus 4 Werken:
 - (a) Rechenwerk
 - (b) Speicherwerk
 - (c) Ein-/Ausgabewerk
 - (d) Leitwerk
2. Rechner ist programmgesteuert
3. Programme und Daten im selben Speicher
4. Hauptspeicher ist in Zellen gleicher Größe aufgeteilt, jede Zelle hat eine Adresse
5. Programm ist eine Sequenz an Befehlen
6. Abweichung von sequentieller Ausführung durch Sprünge möglich
7. Rechner verwendet Binärdarstellung

Befehlszyklus

von-Neumann-Befehlszyklus:

1. Befehl holen
2. Befehl dekodieren
3. Operanden holen
4. Befehl ausführen
5. Ergebnis zurückschreiben
6. Nächsten Befehl adressieren

Assemblertheorie

RiscV-Befehlssatz

add[i]	x1	x2	x3	
and[i]	x1	x2	x3	
or[i]	x1	x2	x3	
xor[i]	x1	x2	x3	
sll[i]	x1	x2	x3	shift left
srl[i]	x1	x2	x3	shift right
mv	x1	x2		move
neg	x1	x2		logical negation
not	x1	x2		bitwise negation
sub	x1	x2	x3	
mul	x1	x2	x3	
div	x1	x2	x3	
rem	x1	x2	x3	remainder
li	x1	Imm		
la	x1	lable		
lb	x1	Imm(x2)		load byte
lh	x1	Imm(x2)		
lw	x1	Imm(x2)		
sb	x1	Imm(x2)		store byte
sh	x1	Imm(x2)		
sw	x1	Imm(x2)		
call	lable			
ret				

Speicherbereiche

		Schreibbar	Ausführbar	Dynamisch wachsend
Textsegment	Programmcode	Nein	Ja	Nein
Datensegment	Globale Variablen	Ja	Nein	Nein
Stack	Lokale Variablen	Ja	Nein	Ja
Heap	langlebige Variablen	Ja	Nein	ja

Stack

In RiscV wächst der Stack von oben nach unten. RiscV bietet ein spezielles Stackpointer-Register(sp Register), welches immer auf die Adresse des zuletzt hinzugefügten Elements zeigt

Lesen vom Stack

Letztes Element des Stacks lesen: `lw t0, (sp)`

Vorletztes Element des Stacks lesen: `lw t0, 4(sp)` (hier ein Integer (word))

Schreiben auf den Stack

Ein Element auf den Stack legen:

```
li t0, 10
addi sp, sp, -4
sw, t0, (sp)
```

Mehrere Elemente auf den Stack legen:

```
addi sp, sp, -12
sw, t0, 8(sp)
sw, t1, 4(sp)
sw, t2, (sp)
```

Speicher freigeben

```
addi sp, sp, 4
```

Mikroprogrammierung

Durch Mikroprogrammierung muss nicht zwangsläufig jeder Befehl fest verdrahtet sein, er kann auch emuliert werden. Besonders bei sehr großen Befehlssätzen (CISC(Complex Instruction Set Computer))

Mikroprogrammierung bedeutet, komplexe **Maschinenbefehle** zur Laufzeit in der CPU durch eine Reihe an noch kleineren, einfacheren Befehlen zu emulieren. **Assembler-Programme** sind demnach **Makroprogramme**

Aufbau

Ein Mikro-Programm besteht aus 4 Kernbestandteilen:

- Steuerleitungen
- ALU (Arithmetisch-logische Einheit)
- Zwischenregister
- Tri-State (Schalter zum Öffnen und Schließen der Datenleitungen)

Horizontsal vs. vertikal

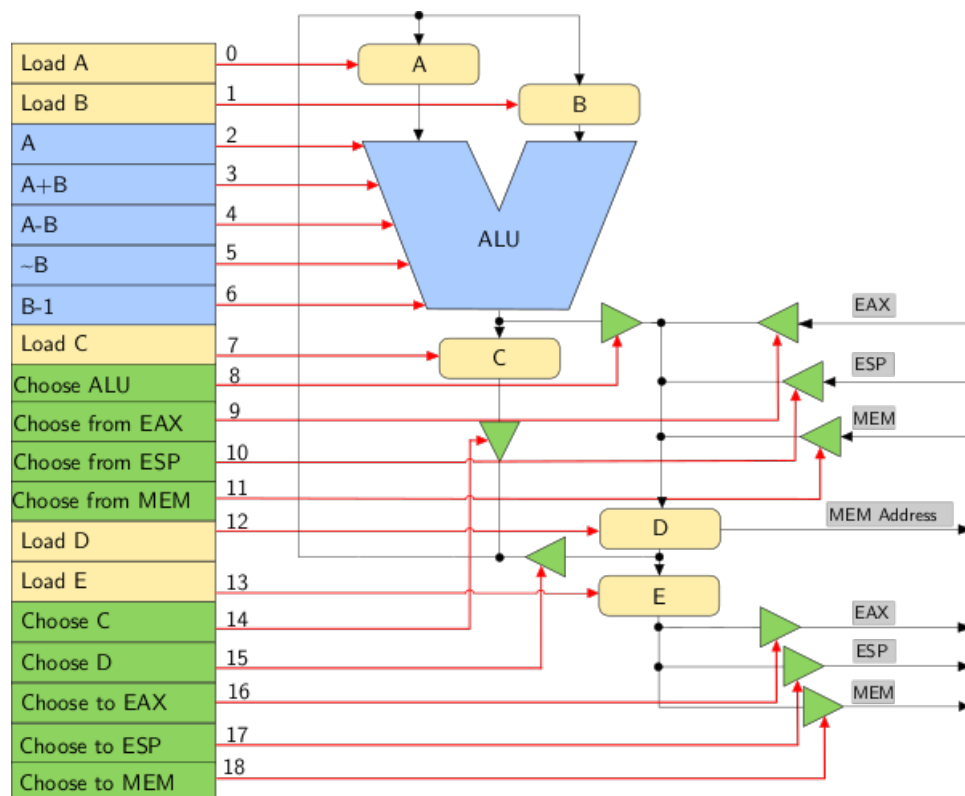
Horizontal

Für jede Steuerleitung wird ein Bit im Mikrobefehl verwendet. Vorteil, keine Dekodierung. Nachteil, Speicherverschwendung

Vertikal

Steuerleitungen werden gruppiert, was zu kürzeren Mikrobefehlen führt. Vorteil, speichereffizient. Nachteil, zur Laufzeit muss der komprimierte Befehl wieder in Leitungssignale umgewandelt werden

Beispiel



Zeile	Instruktion
0	ESP → B
1	B-1 → C
2	C → B
3	B-1 → D
4	D → ESP

Befehlssatzarchitektur

Register-Register

Alle Operanden eines Assemblerbefehls müssen in einem Register stehen

```
load R1, A
load R2, B
add R3, R1, R2
store R3, C
```

Register-Memory

Operanden können sowohl in den Registern, als auch in Speicherzellen liegen

```
load R1, A
add R1, B
store R1, C
```

Akkumulator

Der erste Operand wird mittels load in den Akkumulator geladen, der zweite kommt aus dem Speicher

```
load A
add B
store C
```

Stack

Operanden werden zuerst auf den Stack gepushed. Assemblerbefehle nimmt dann die obersten Werte vom Stack und rechnet damit. Ergebnis landet ebenfalls auf dem Stack

```
push A
push B
add
pop C
```

Endianess and Alignment

Endianess

Die Endianess beschreibt, in welcher Reihenfolge die Bytes innerhalb zusammenhängender Datums abgespeichert werden

Little Endian: Least Significant Byte first

Big Endian: Most Significant Byte first

Alignment

Um sicherzustellen, dass der Zugriff auf ein Datum möglichst wenig Speicherzugriffe benötigt, ist richtiges Alignment nötig. Ein Datum ist korrekt aligned, wenn gilt:

$$\text{Adresse}(\text{Datum}) \% \text{Größe}(\text{Datum}) = 0$$

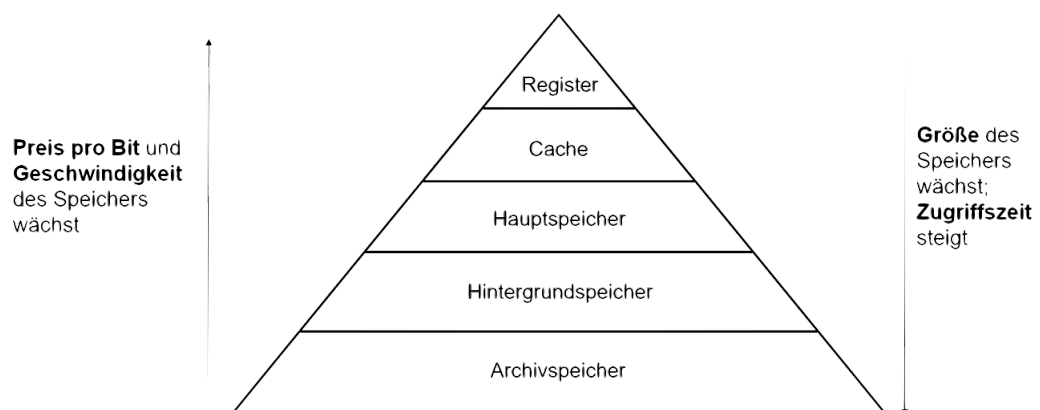
Um Daten korrekt auszurichten muss man unter Umständen Padding einfügen, also freien Platz.

Für **structs** gilt zusätzlich:

$$\text{Adresse}(\text{struct Anfang}) \% \max(\text{Größe}(\text{DatumInStruct})) = 0$$

und am Ende eines structs muss so viele Padding eingefügt werden, sodass, würde das gleiche struct noch einmal abgelegt werden, es automatisch aligned wäre

Speicherhierarchie



Performance Gap

Die Performanzsteigerung bedeutet, dass die CPU immer mehr Berechnungen in gleicher Zeit ausführen kann. Der Arbeitsspeicher kann ebenfalls mehr Daten in der selben Zeit lesen und weitergeben, jedoch fällt diese Steigerung deutlich geringer aus, was als Performanzlücke bezeichnet wird.

Räumliche und Zeitliche Lokalität

Zeitlich

Programme neigen dazu, auf die selbe Adresse innerhalb kürzester Zeit mehrmals zuzugreifen. Caches nutzen das, indem sie sich Arbeitsspeicherzugriffe merken. Wenn auf eine Adresse mehrmals zugegriffen wird, dann wird der Arbeitsspeicher nur beim ersten Zugriff besucht, anschließend der Cache

Räumliche

Programme neigen dazu, wenn sie auf eine Adresse X zugreifen, in naher Zukunft auch auf benachbarte Adresse von X zuzugreifen. Caches nutzen das, indem sie bei einem Arbeitsspeicherzugriff automatisch auch die benachbarten Daten (typischerweise etwa 64 Byte) mitladen. Wenn dann kurz darauf ein Zugriff auf eine benachbarte Adresse stattfindet, kann der Cache dieses Datum direkt liefern und muss den Arbeitsspeicher nicht noch einmal anfragen.

Cache

Grundbegriffe

Der CPU Cache ist ein schneller Zwischenspeicher, der eine Teilmenge der Daten aus dem Arbeitsspeicher hält. Bei einem Arbeitsspeicherzugriff wird zuerst im Cache nachgeschaut, ob das angefragte Datum dort bereits existiert. Der Cache ist für den Programmierer in der Regel unsichtbar, er wird nicht direkt adressiert!

Cache Hit

Angefragtes Datum liegt bereits im Cache und kann daher deutlich schneller an das Registerwerk weitergeleitet werden

Cache Miss

Angefragtes Datum liegt nicht im Cache, muss daher aus dem Arbeitsspeicher geholt werden und wird anschließend im Cache abgelegt

Platzierung und Identifikation

Cacheblöcke

Caches werden in sogenannte Blöcke gleicher Größe unterteilt. Beispielsweise könnte ein insgesamt 32 Byte großer Cache in 4 x 8 Byte große Blöcke unterteilt werden. Der Cache lädt dann bei einem Arbeitsspeicherzugriff immer 8 benachbarte Byte in einen Block.

Die zu ladenden Nachbarn sind dabei immer auf die Cacheblockgröße aligned”(vgl. Lernmodul 07): Das heißt, wenn wir zum Beispiel 8 Byte große Cacheblöcke haben, wird aus dem Arbeitsspeicher immer ein Nachbarpaket der Größe 8 Byte geholt, beginnend an einer Adresse, die Modulo 8 == 0 ist.

Das abgebildete Schaubild stellt das dar: Wir greifen auf das Byte der Adresse 0x19 (welches den Wert 0x30 hat) im Arbeitsspeicher zu, wodurch der ganze 8 Byte große Nachbarblock geladen wird. Die nächste, davor liegende Adresse, die Modulo 8 == 0 ist, ist die Adresse 0x18.

Byteoffset

Aus dieser Erkenntnis wissen wir, dass ein Teil der Arbeitsspeicheradresse immer die Funktion eines sogenannten Byteoffsets erfüllt. Wenn wir wie im obigen Beispiel 8 Byte große Cacheblöcke haben, sind folglich die letzten 3 Bit (wegen $2^3 = 8$) der Arbeitsspeicheradresse das Byteoffset innerhalb eines Blockes. Um zum Beispiel aus dem 8 Byte großen Cacheblock die Zahl an der Adresse 0x19 (= 0001 1001) zu ziehen, müssten wir das 001'ste Byte (von links) nehmen.

Mengen / Index

Mehrere Cacheblöcke können zu einer Menge gruppiert werden. Innerhalb einer Menge dürfen wir zum Ablegen unserer Daten einen beliebigen Block wählen. Die Menge, in die wir die Daten legen, hängt wiederum von der Arbeitsspeicheradresse ab. Hierfür bestimmen wir den sogenannten Index

Tag

Der Tag sind die übrigen Bits einer Arbeitsspeicheradresse, die nicht für den Index oder das Byteoffset zuständig sind. Diese werden benötigt, um eine Adresse eindeutig rekonstruieren zu können und dadurch Konflikte zwischen ähnlichen Adressen zu vermeiden. Der Tag wird als Metadaten bei den Cacheblöcken gespeichert

		Block	Tag	000	001	010	011	100	101	110	111
1 Bit Index	Menge 0	0	1001	FF	34	10	00	00	00	18	FF
		1	0010	E0	D4	00	03	04	42	13	3F
	Menge 1	2	0001	C4	30	00	00	D2	D2	10	11
		3	1001	00	00	06	34	C1	D0	AB	40

Organisationsformen

Vollassoziativ

Ein Cache mit nur einer Menge, in welcher alle Blöcke sind. Wo man Datenblöcke ablegt ist also völlig beliebig und nicht anhand einer Adresse berechenbar.

Direktabbildend

Ein Cache mit Mengen, die jeweils aus genau einem Block bestehen. Man kann also 1:1 an der Arbeitsspeicheradresse (über den Index) "berechnen", in welcher Menge und dadurch auch welchem Block die Daten liegen.

N-fach assoziativ

Ein Cache mit Mengen, die aus jeweils n Blöcken bestehen. Hybridversion/Trade-Off zwischen den ersten beiden.

Vor und Nachteile

- Je mehr Blöcke pro Menge man hat, desto schwieriger ist es die Daten wieder zu finden
- Je weniger Blöcke pro Menge man hat, desto höher ist die Wahrscheinlichkeit, dass sich zwei Adressen gegenseitig verdrängen

Ersetzungsstrategien

Innerhalb einer Menge (mit mindestens 2 Blöcken) können Daten in einen beliebigen Blöcke gelegt werden. Zu Beginn macht es Sinn, einfach einen freien Block zu nehmen. Wenn alle Blöcke voll sind, muss intelligent ein belegter Block ersetzt werden. Die eingesetzte Ersetzungsstrategie entscheidet, welcher Block für einen neuen verdrängt wird.

Random

Es wird "gewürfelt". (Nicht-vorhersagbares Cacheverhalten; dafür vergleichsweise leicht zu implementieren in Hardware)

LRU

Least Recently Used. Der Block, auf den am längsten nicht mehr zugegriffen wurde, wird verdrängt. (Liefert gute Hitraten in der Realität, aber ist schwer zu implementieren. Die Blöcke müssen sich ihre letzten Zugriffe merken, und man muss beim Verdrängen die Zeitstempel vergleichen/sortieren).

LFU

Least Frequently Used. Der Block, auf den am seltensten zugegriffen wurde, wird verdrängt. (Ähnlich bzgl. Vor-/Nachteile wie LRU. Außerdem muss man beachten, dass es dadurch passieren kann, dass ein (zeitweise) häufig benutzter Block für immer im Cache bleibt, auch wenn er nun nicht mehr betrachtet wird. Der Counter muss am Besten periodisch wieder runtergesetzt werden)

FIFO

First in, first out

Aktualisierungsstrategien

Die sogenannte Aktualisierungsstrategie eines Caches bestimmt, wie sich ein Cache verhält, wenn ein Datum geändert (aktualisiert) wird.

Write through

Alle Änderungen an Cacheinhalten werden direkt auch an die nächste Stufe der Speicherhierarchie (typischerweise der Arbeitsspeicher) weitergeleitet (durch den Cache durchgeschrieben).

- Vorteil: Kohärenz zwischen Arbeitsspeicher und Caches (beide kennen immer die neusten Daten); insbesondere bei Mehrkernsystemen relevant.
- Nachteil: Sehr langsam, da ständig etwas an den Arbeitsspeicher geschickt wird. Blockiert dadurch die Arbeitsspeicherbandbreite.

Write back

Änderungen an Cacheinhalten werden erst zurück in die nächste Stufe der Speicherhierarchie geschrieben, wenn der Cacheblock verdrängt wird. Um sich hier unnötiges Rückschreiben nicht-geänderter Daten zu sparen, wird sich bei jedem Block ein Dirty-Bit gemerkt. Dieses wird auf 1 gesetzt, wenn der Block aktualisiert wurde, was beim Verdrängen zum tatsächlichen Rückschreiben der Daten führt. Wenn es auf 0 bleibt, wird der Block einfach gelöscht und nicht zurückgeschrieben.

- Vorteil: Wenn es viele Änderungen am Cacheblock gibt, wird es nur 1x am Ende zurückgeschrieben. Das spart Zeit/Performance
- Nachteil: Zeitweise große Inkonsistenz zwischen Cache und Arbeitsspeicher

Arbeitsspeicher

Pipelining

Instruktionsparallelismus

Threadparallelismus

Grafikkarten

Speicherverwaltung