Grundlagen der Rechnerarchitektur

Felix Leitl

28. Juli 2023

Inhaltsverzeichnis

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:

$$0xAB \cdot 2^2 = 10101011 << 2 = 1010101100 = 0x2AC$$
$$0xAB/2^2 = 10101011 >> 2 = 00101010 = 0x2A$$

Rechnerarchitektur

Endo- vs. Befehlsarchitektur

Exteren 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 programmgestuerert
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 addresieren

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 Mirko-Programm besteht aus aus 4 Kernbestandteilen:

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

Horizontsal vs. vertikal

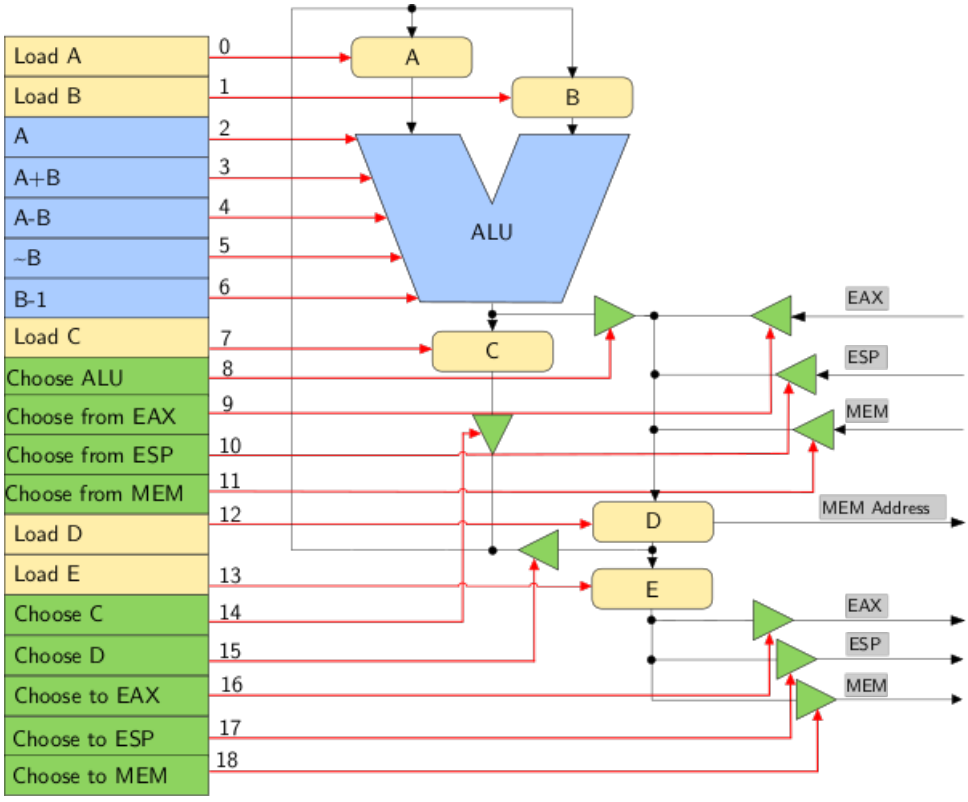
Horizontal

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

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 alignd, wenn gilt:

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

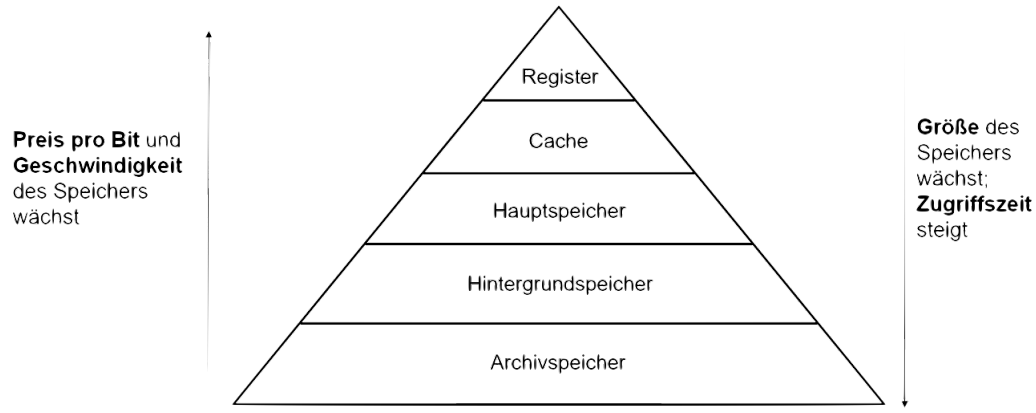
Um Daten korrekt auszurichten muss man unter Umständen Padding einfügen, also freien Platz.

Für **structs** gilt zusätzlich:

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

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

Speicherhirachie



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 zuzugriefen. Chaches 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

Arbeitsspeicher

Pipelining

Instruktionsparallelismus

Threadparallelismus

Grafikkarten

Speicherverwaltung