Grundlagen der Rechnerarchitektur

Felix Leitl

31. 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

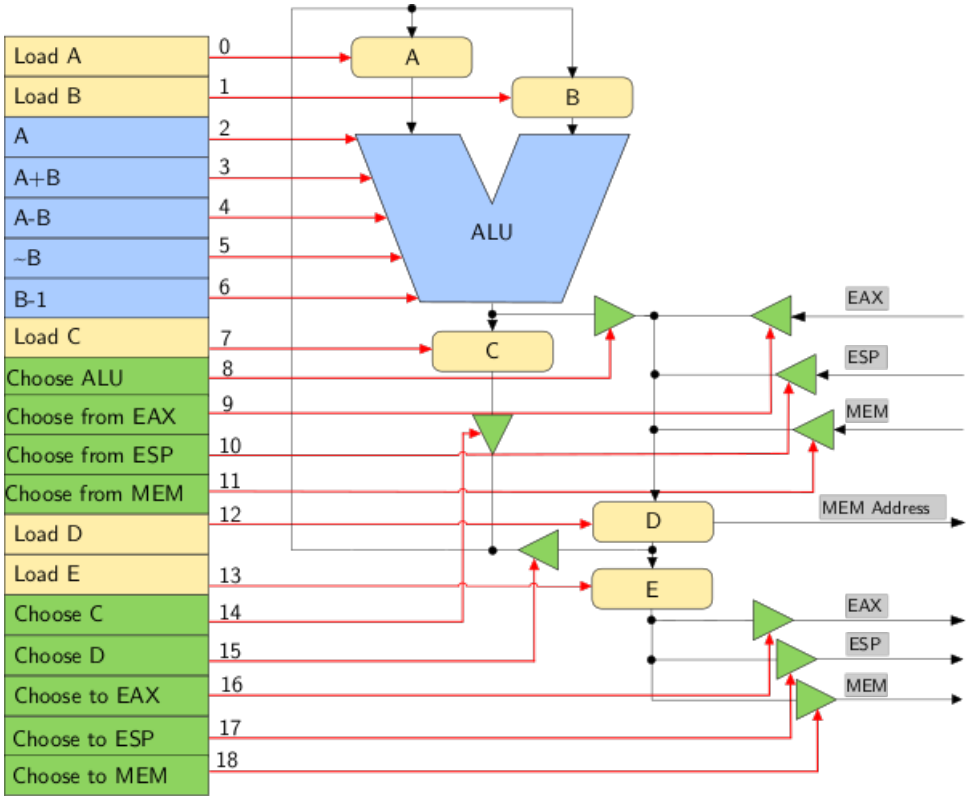
Horizontal

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:

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

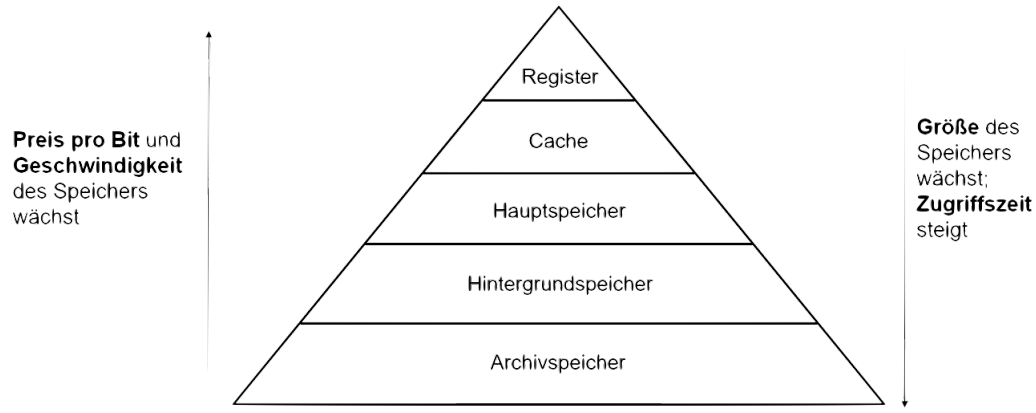
Um Daten korrekt auszurichten muss man unter Umständen Padding einfügen, also freien Platz.

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

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 addressiert!

Cache Hit

Angefragtes Datum liegt bereits im Chache 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 äligned"(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 nähste, davor liegende Adresse, die Modulo 8 == 0 ist, ist die Adresse 0x18.

Byteoffset

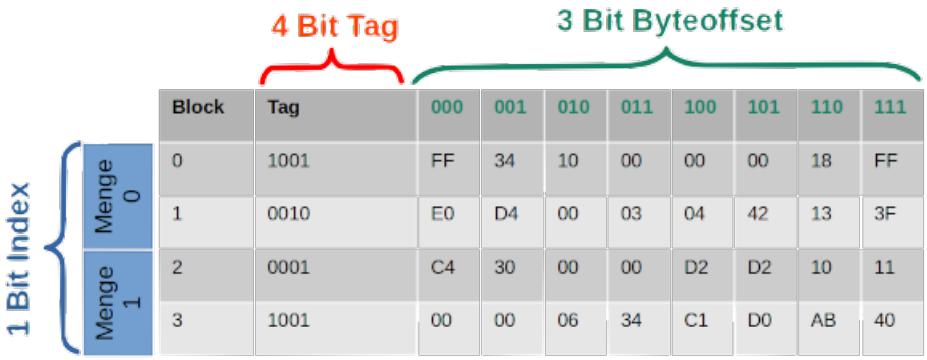
Aus dieser Erkenntnis wissen wir, dass ein Teil der Arbeitsspeicheradresse immer die Funktion eines sogenannten Byteoffsets erfüllt. Wenn wir wie im ebigen Beispiel 8 Byte große Cacheblöcke haben, sind folglich die letzten 3 Bit (wegen $2^3 = 8$) der Arbeitsspeicheradresse das Byteoffset innerhalb eines 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
0	1001	FF	34	10	00	00	00	18	FF
1	0010	E0	D4	00	03	04	42	13	3F
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 assotiativ

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 immerïm 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

Taktraten und DDR

Speichertakt

Der Arbeitsspeicher hat intern einen eigenen Takt. Dieser ist per se unabhängig vom CPU-Takt

Bustakt

Bei SDRAM sind Speicher- und Bustakt synchron, also ein Vielfaches voneinander

Busbreite

Beschreibt, wie viele Datenleitungen der Bus besitzt (typischer Weise zwischen 16 und 256 Bit)

Datenrate

Die Datenrate beschreibt, wie viele Daten theoretisch pro Sekunde an die CPU gesendet werden könnten

$$\texttt{Datenrate} = 2 \cdot \texttt{Busbreite} \cdot \texttt{Bustaktrate}$$

DDR-SRAM

DDR (Double Data Rate) überträgt bei steigender und fallender Flanke

Prefetch

Der Prefetch beschreibt, wie viele Datenpakete der Arbeitsspeicher pro Speichertakt theoretisch liefern kann.

DDR 1-5

	Prefetch	Bustaktrate
DDR1	2	1· Speichertaktrate
DDR2	4	2· Speichertaktrate
DDR3	8	4· Speichertaktrate
DDR4	8	4· Speichertaktrate
DDR5	16	8· Speichertaktrate

DRAM vs. SRAM

DRAM (Dynamic Random Access Memory)

Speichert Bits als Kondensatorenladung

- Vorteil: Sehr wenig Platz wird benötigt (billig)
- Beim lesen der Speicherzelle und nach einiger Zeit geht der Wert verloren und muss gesetzte werden (Refresh)

SRAM (Static Random Access Memory)

Speichert Bits durch Transistorlogik

- Vorteil: Schneller als DRAM und beim Lesen geht der Wert nicht verloren
- Nachteil: Deutlich größerer Platzverbrauch als DRAM (teuer)

(Burst-)Zugriffe

DDR-SDRAM ist intern normalerweise in zweidimensionalen Speichermatrizen organisiert, wodurch man die Adressen mit Zeilen- und Spaltennummern adressieren muss.

Ein Lesezugriff durchläuft dabei 4 Phasen:

- Precharge: DRAM wird auf Zugriff vorbereitet
- Row Addres Strobe (RAS): Zeile wird ausgelesen
- Column Adress Strobe (CAS): Spalte wird ausgelesen
- Data Read: Daten stehen zum Abholen durch den Bus bereit

Burstzugriff

Du erinnerst dich an räumliche Lokalität: Oft werden nach dem Zugriff auf eine Adresse X kurz darauf die Nachbardaten an den Adressen X+1, X+2, X+3, ... benötigt. Im Arbeitsspeicher liegen Nachbardaten normalerweise in der selben Zeile. Der Arbeitsspeicher kann nun also intern, um nacheinander die Nachbarn auszulesen, die Zeile 1x lesen (Precharge + RAS) und dann mehrmals einen Column Address Strobe (CAS) durchführen, um mehrere Spalten nacheinander auszulesen. Dadurch spart man sich viel Latenz. Dieses Vorgehen nennt sich Burstzugriff.

Pipelining

RISC vs. CISC

RISC(Reduced Instruction Set Computer)(x86)
CISC(Complex Instruction Set Computer)(RISC-V)

CISC

Vorteile:

- Geringe Speicherdichte für Programmcode erforderlich. Mit weniger Assemblerfunktionen kann man mehr Funktionalität erreichen
- Einfacher, kompakter Code

Nachteile:

- sehr große und undurchschaubare Befehlssätze
- nur kleiner Teil der Befehlssätze wurde tatsächlich verwendet

RISC

Merkmale:

- Kleine, einheitliche Machinenbefehlssätze
- Adressberechnung durch explizite Befehle durch führen
- Load-Store-Architektur: Alle Operanden liegen in Registern, Zugriff auf Speicheradressen mittels expliziter Lade- bzw. Speicherbefehle
- Große Anzahl universell nutzbarer Register
- Festverdrahtete Leitwerke: kein Mikroprogramm; jeder RISC-Befehl wird direkt in binäres Befehlsmuster dekodiert
- Konsequentes Ausnutzen von Pipelining

Grundlagen

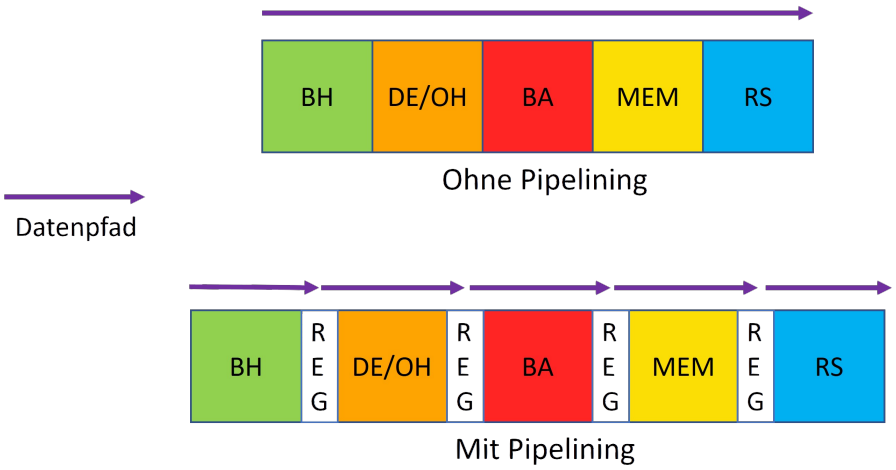
Das grundlegende Prinzip von Pipelining (Fließbandverarbeitung) ist, dass verschiedene Phasen verschiedener, unabhängiger Befehle gleichzeitig durchlaufen werden. Dies kann man auch als pseudoparallele Verarbeitung bezeichnen.

RISC-V-Befehlszyklus:

- BH: Befehl holen
- BD/OH: Befehl dekodieren und Operanden holen
- BA: Befehl ausführen
- MEM: Speicherzugriff
- RE: Rückschreibephase

Zwischen die einzelnen Stufen werden Register eingefügt. Dies ist nötig, damit die Werte unterschiedlicher Befehle voneinander entkoppelt werden und sich nicht gegenseitig beeinflussen. Hätte man keine Zwischenregister, würde das Ergebnis von Phase A in Phase B reinlaufen", während Phase B noch nicht mit der vorherigen Instruktion fertig ist.

Die Einführung der Register sorgt außerdem dafür, dass es nicht einen langen Datenpfad, sondern mehrere kürzere Datenpfade gibt. Ein Datenpfad sind die Leitungen und logischen Gatter, die zwischen zwei Speicherzellen/Registern durchlaufen werden müssen. Dadurch wird der kritische Pfad (= längster Datenpfad) ebenfalls kürzer, wodurch der Takt angehoben werden kann, da weniger Logik pro Takt ausgeführt werden muss.



Speedup

Ausführung ohne Pipelining:

$$T_{seq} = (\sum_{i-1}^{k} \tau_i) \cdot n$$

Ausführung mit Pipelining:

$$T_{par} = k \cdot \tau_{max} + (k-1) \cdot d_{Reg} + (n-1) \cdot (\tau_{max} + d_{Reg})$$

$\tau_{max} = \texttt{max}_{1 \leq i \leq k}\{\tau_i\}$ steht für die Ausführungszeit der langsamsten Stufe
$k \cdot \tau_{max} + (k-1) \cdot d_{Reg}$: Der erste Befehl läuft durch alle k Pipelinestufen

Speedup allgemein:

$$S = \frac{T_{seq}}{T_{par}} = \frac{\texttt{sequentielle Ausführung}}{\texttt{parallele Ausführung}}$$

Hazards

Datenhazards

Entstehen durch Datenabhänigkeit. Dabei wird das Ergebnis der vorherigen Instruktionen von der aktuelle Instruktion benötigt, ist aber noch nicht verfügbar

Steuerungshazards

Treten unter Umständen bei Befehlen auf, die den Befehlszähler verändern, wie bspw. bedingte Sprünge. Es gelangen Befehle in der Pipeline, die anschließend wieder verworfen werden müssen, weil an eine andere Stelle gesprungen wird

Strukturhazards

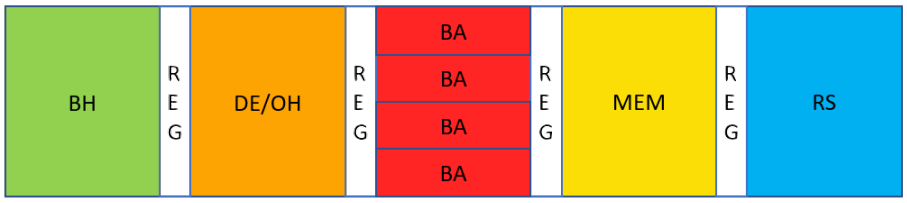
Entstehen, wenn gleichzeitig auf eine Hardwarekomponente (z.B. gleiche Recheneinheit) zugegriffen wird

Instruktionsparallelismus

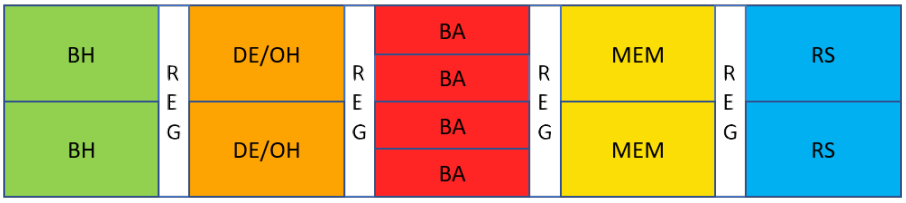
Superskalarität

Bei Superskalarität werden gleichzeitig an mehrere Recheneinheiten Befehle eines Threads geschickt, die dann parallel abgearbeitet werden. Auf Hardwareebene muss dafür die Logik der BA-Phase vervielfältigt werden:

Pipeline mit 4 Recheneinheiten in der BA-Phase



2-fach superskalare Pipeline mit 4 Recheneinheiten in der BA-Phase



Dynamische Parallelisierung

Nicht alle Befehle eines sequentiellen Befehlsstrom können parallel ausgeführt werden. Datenhazards, also Datenabhängigkeiten zwischen Registern, sind ein Grund dafür. Bei Superskalarität wird deshalb spezielle Hardware eingefügt, die diese Abhängigkeiten erkennen und auflösen kann.

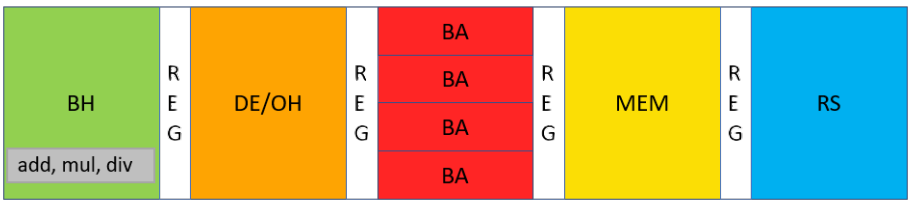
Bei superskalaren Pipelines können auch Strukturhazards auftreten. Ein solcher tritt auf, wenn eine Recheneinheit (z.B. der Dividerer) benötigt wird, jedoch gerade noch durch eine andere Instruktion belegt ist. Auch für die Auflösung von Strukturhazards muss zusätzliche Hardware eingefügt werden, die die Belegung der Recheneinheiten überwacht und ggf. Befehle verzögert.

Die Parallelisierung geschieht also zur Laufzeit durch die Hardware, was als dynamische Parallelisierung bezeichnet wird.

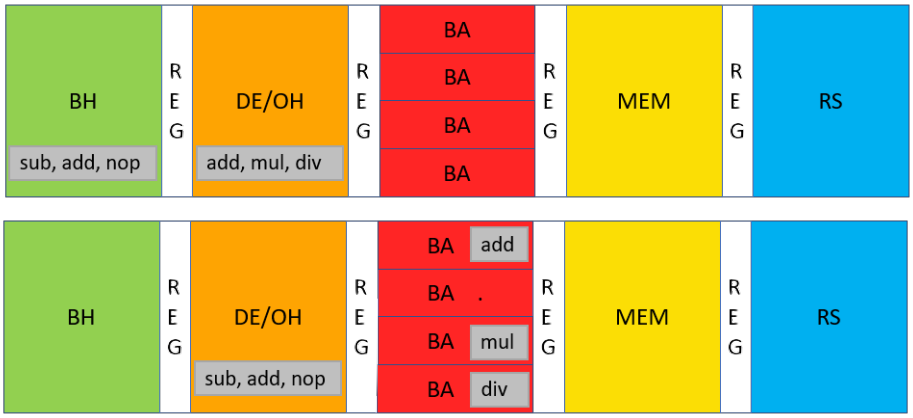
Very Long Instruction Word

Auch bei der Nutzung eines VLIW (Very Long Instruction Word) ist das Ziel die Beschleunigung der Abarbeitung von sequentiellen Programmen. Wie bei Superskalarität wird Parallelität auf Befehlsebene ausgenutzt. Im Gegensatz zur Ausführung bei Superskalarität werden die Befehle nicht dynamisch zur Laufzeit gruppiert, sondern statisch vom Compiler vor der Laufzeit. Dies wird auch als statische Parallelisierung bezeichnet

Der Compiler fügt parallel ausführbare Instruktionen (ohne Datenabhängigkeiten!) zu einem Instruktionswort zusammen. Die maximal mögliche Anzahl ist dabei abhängig von der Länge des Instruktionswortes. Dieses durchläuft dann sequentiell die Pipeline:



Wenn nicht genügend parallel ausführbare Operationen zur Verfügung stehen, muss der Compiler ein oder mehrere nop (no operation), also Platzhalter einfügen. Dadurch wird jedoch der Durchsatz verringert



Threadparallelismus

Grundlagen Multithreading

Rechner können nur eine bestimmte Anzahl an Threads echt-parallel ausführen. Um dennoch ünendlich"viele Threads unterstützen zu können und es dem Benutzer gegenüber so aussehen zu lassen, als würden diese parallel ausgeführt werden, wird in schnellen Abständen zwischen den aktiven Threads gewechselt.

Zeitscheiben Multithreading

Jeder Thread bekommt immer eine fixe Zeit, die er aktiv rechnen darf, bevor er wieder ausgewechselt wird (z.B. alle 50ms wird gewechselt). Die CPU muss hierfür einen Hardwaretimer besitzen. Das Betriebsystem setzt den Hardwaretimer, lässt den Thread an die CPU, und nach Ablauf der Zeit wechselt die CPU automatisch durch ein sogenanntes Interruptsignal (Unterbrechungssignal) zum Betriebssystem zurück

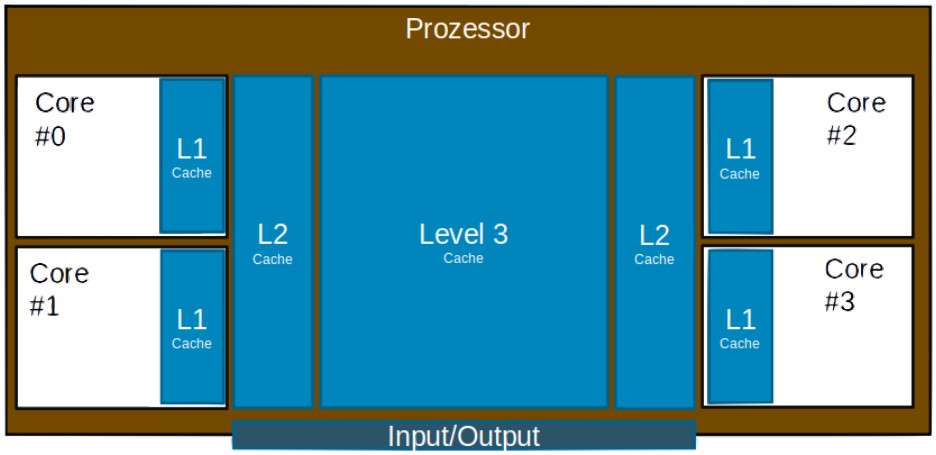
Ereignisgesteuertes Multithreading

Zwischen Threads wird dann gewechselt, wenn ein aktiver Thread beginnt, auf ein externes Ereignis zu warten (z.B. Antwort von der Festplatte oder Benutzereingabe). Derartige Events benötigen in der Regel ohnehin Kommunikation mit dem Betriebssystem (d.h. einen Sprung in den Betriebssystemcode), weshalb die CPU hier keine besondere Hardwareeigenschaften benötigt. Sobald das Betriebssystem wieder selbst ïn der CPUïst, kann es auch entscheiden, ob ein Threadwechsel stattfindet

Ein Threadwechsel ist dabei teuer: Bei jedem Wechsel müssen die Register des Threads gesichert (und wiederhergestellt) werden, und der komplette Cacheinhalt wird in der Regel nutzlos (da der nächste Thread selten die Daten des vorherigen Threads braucht). Zeitscheiben sollten daher nicht zu niedrig gesetzt werden

Cachehirachie

Bei Multicoresystemen wird der Cache in der Regel in mehrere Hierarchiestufen unterteilt. Die folgende Grafik zeigt einen beispielhaften Aufbau für eine 4-Kern-CPU (Quadcore):



Simultanes Multithreading

Bei simultanem Multithreading (SMT) nutzen wir die Tatsache aus, dass ein Thread in der Regel nicht alle Teile des Rechenwerks (und theoretisch auch nicht den ganzen Cache) gleichzeitig benötigt und erweitern den Kern daher um ein weiteres Frontend, einen weiteren Registersatz und einen weiteren Befehlszähler. Das Rechenwerk bleibt unverändert. Diese Änderungen sind deutlich günstiger, als den kompletten Kern zu verdoppeln

Dadurch ist es bei SMT möglich, zwei Thread echt-parallel im selben Kern auszuführen. Dies klappt jedoch nur, wenn die Threads unterschiedliche Operationen ausführen wollen, z.B.: Thread 1 möchte addieren, während Thread 2 dividiert

Bei einem physischen CPU Kern, der durch 2-fachem SMT bis zu 2 Threads gleichzeitig ausführen kann, spricht man dann auch davon, dass er aus 2 logischen Kernen besteht

Grafikkarten

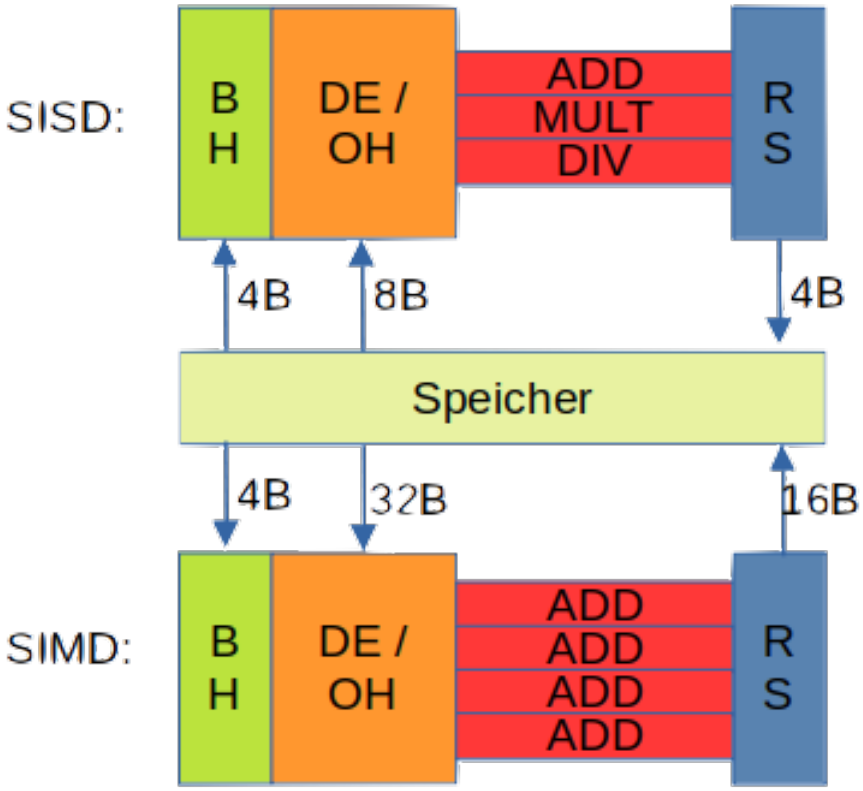
Grundlagen

SIMD Prinzip

Single Instruction, Multiple Data. Dieses Prinzip beschreibt Instruktionsparallelismus, bei dem auf eine große Datenmenge parallel die gleiche Instruktion ausgeführt werden soll. Dies unterscheidet sich von dem dir bisher bekanntem SISD (Single Instruction, Single Data), bei dem in jeder Programmzeileëine beliebige Instruktion genannt wird, die dann wiederum auf 2 Eingabezahlen arbeitet

SIMD Instruktionen zu unterstützen erfordert natürlich auch architekturelle Veränderungen am Rechner: Es müssen einerseits mehr homogene (d.h. gleichartige) Recheneinheiten (z.B. Addierer) hinzugefügt werden, die parallel arbeiten können. Manchmal wird stattdessen (um Platz zu sparen) auf andere Recheneinheiten (z.B. Multiplizierer) gänzlich verzichtet wird. Dadurch sind die Rechenoperationen, die nun nicht mehr verbaut sind, deutlich weniger performant umzusetzen (z.B. Multiplikation durch eine Vielzahl an Additionen). SIMD Rechner sind also kein grundsätzlicher Ersatz für normale SISD Rechner.

Die Operanden-Hol-Phase und die Rückschreib-Phase benötigen außerdem (bestenfalls) eine breitere Speicheranbindung um mehr Daten pro Takt zu laden bzw. zurückzuschreiben.



Bei Grafikkarten (GPUs) wird dieses architekturelle Prinzip auf die Spitze getrieben: Hier haben wir nur ein (oder ein paar wenige) Steuerwerke, die in jedem Takt nur einen (oder ein paar wenige) Operationen holen, und dann auf einer extrem großen Anzahl an Recheneinheiten (im Kontext von GPUs auch Shader genannt) diese Operation für große Datenmengen ausführen.

CUDA Modell

GPU (Graphics Processing Unit)

Die Grafikkarte als Ganzes. Ein normaler Rechner besitzt in der Regel eine GPU (manchmal auch 2), oftmals als eigenes Gerät außerhalb der CPU. Günstige Rechner besitzen hingegen GPUs, die direkt mit in der CPU integriert sind (Integrated Graphics). Diese sind aber entsprechend sehr klein und leistungsschwach

Streaming Multiprocessor

Eine GPU besitzt ein oder mehrere Streaming Multiprocessors (SMs). Dies ist vergleichbar mit dem Konzept von Multicore bei CPUs: Jeder SM arbeitet unabhängig voneinander und kann auch tatsächlich verschiedene Instruktionen ausführen. SMs sind also relativ zueinander nicht SIMD

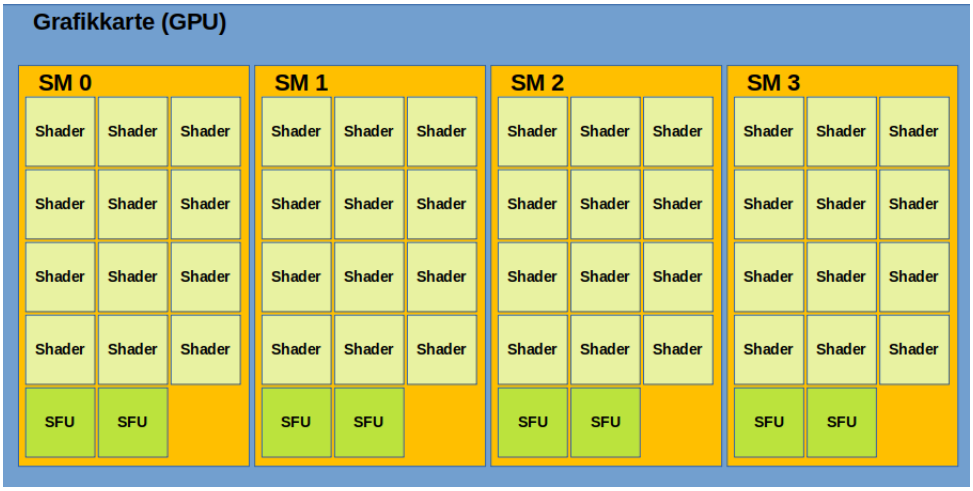
Shader

Ein Streaming Multiprocessor besteht aus einer Vielzahl an Shaders (typischerweise ca. 32). Diese Shader innerhalb eines SMs arbeiten nach dem SIMD Prinzip! Sie führen zeitgleich die exakt selbe Instruktion aus, auf unterschiedlichen Daten. Ein Shader beherrscht die absoluten Grundrechenoperationen, z.B. addieren und multiplizieren

Special Function Units

eder Streaming Multiprocessor besitzt neben den Shadern auch meistens noch 1-2 SFUs. Diese berechnen komplexere Dinge, wie z.B. Sinus oder Kosinus. Da wir davon aber nur sehr wenige haben, funktionieren sie nicht nach dem parallelen SIMD-Prinzip, sondern müssen sequentiell ausgeführt werden. Dies ist natürlich langsam - aber es ist immer noch schneller, als zu sagen, dass die GPU gar keinen Sinus/Kosinus kann und man jedes Mal diese Berechnung (die als Zwischenschritt in einem echten SIMD-Programm vorkommen könnte) an die CPU zu schicken

GPU mit 4 SMs, mit jeweils 12 Shadern und 2 FSUs



CUDA Programmieren

Genau wie die Grafikkarten hierarchisch unterteilt ist (GPU -> Streaming Multiprocessor -> Shader) müssen beim CUDA-Modell auch Programmierer*innen ihr Problem unterteilen. Das Gesamtproblem kann man dabei als die gesamte Datenmenge verstehen, die verrechnet werden soll

Grid

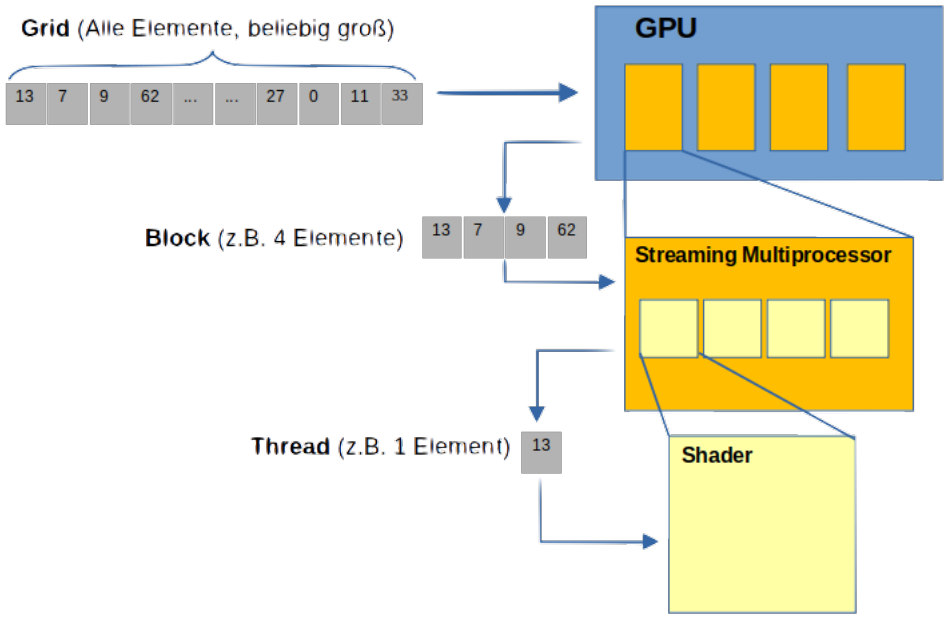
Die gesamte Datenmenge, die von einer Grafikkarte (GPU) verrechnet werden soll, wird als ”Grid”bezeichnet. Dies kann eindimensional (z.B. ein Vektor), zweidimensional (z.B. eine Matrix) oder dreidimensional (z.B. 3D Matrizen für 3D Computerspiele) sein

Block

Das Grid wird wiederum in Blöcke gleicher Größe unterteilt. Auch hier sind bis zu 3 Dimensionen möglich (sinnvollerweise identisch zur Griddimension). Ein Block wird von einem Streaming Multiprocessor bearbeitet

Thread

Ein Block besteht wiederum aus mehreren Threads. Ein Thread ist in der Regel für nur ein Datenelement zuständig (theoretisch sind aber auch mehr möglich). Ein Thread wird von einem Shader ausgeführt



Code

```
__global__ void VectorAdd(float *vectorA, float *vectorB, int size){
        int index = blockIdx.x * blockDim.x * threadIdx.x;
        if (index < size)
                targetVector[index] = vectorA[index] * vectorB[index];
        }
}
```

Die Funktion erhält als Funktionsargumente die Anfangsadressen der Vektoren sowie die Gesamtgröße der Vektoren (damit wir nicht versehentlich Werte außerhalb der Vektoren betrachten

In Zeile 3 bekommt jeder Shader mitgeteilt, für welches Element (bzw. welchen Index innerhalb des Vektors) er zuständig ist: Jeder Streaming Multiprocessor bekommt hierzu dynamisch vom GPU Scheduler einen sogenannten Block-Index (`blockIdx`) zugewiesen, und jeder Shader darin bekommt dynamisch eine sogenannte Thread-ID (`threadIdx`) zugewiesen. Da wir auch festgelegt haben, wie viele Threads es in einem Block gibt (`blockDim`), kann jeder Shader sich dadurch seinen Vektorindex ausrechnen

Anschließend wird geprüft, ob der Index überhaupt noch innerhalb des Vektors liegt. Wenn das der Fall ist, addieren wir die Werte der beiden Vektoren `vectorA[index]` und `vectorB[index]` zusammen und speichern es wiederum im Zielvektor `targetVector[index]`

Verzweigungen im CPU-Code

Bedingte Anweisungen, bei denen nur eine Teilmenge der Shader etwas rechnen soll, werden in der Regel so realisiert, dass durch die Bedingung die jeweils an-

deren Shader temporär "deaktiviert"werden. Am Ende der Verzweigung werden sie wieder äktiviert".

Speicherverwaltung

Virtuelle und physische Adressen

Physische

Jedes Byte im physischen Arbeitsspeicher (d.h. im echten, vorhandenen Arbeitsspeicher) erhält in der Regel eine eigene Adresse, die sogenannte physische Adresse. Beim Starten des Rechners wird diese festgelegt und bleibt dann im laufenden Betrieb konstant. Anhand der physischen Adresse kann dann eindeutig bestimmt werden, in welcher Rank, Bank, Row und Column (Vgl. Tafelübung zu Arbeitsspeicher) sich ein bestimmtes Byte befindet

Virtuelle

Auf vielen Rechnern arbeiten Programme jedoch nicht mit den physischen, sondern mit sogenannten virtuellen Adressen. Würde ein Programm direkt mit physischen Adressen arbeiten, dann wären Konzepte wie Auslagerung (Swapping) oder Speicherschutz nicht mehr so einfach zu realisieren

Virtuelle Adressen können vom Programm quasi frei gewählt werden. Mehrere Programme können auch die gleichen virtuellen Adressen verwenden, und dann damit aber unterschiedliche physische Adressen (und daher unterschiedliche Daten) meinen

Virtuelle Adressen müssen, um physisch auf die Daten zugreifen zu können, in physische Adressen übersetzt werden. Das Festlegen von Übersetzungen verwaltet das Betriebssystem mittels sogenannter Übersetzungstabellen. Jeder Prozess bekommt eine eigene Übersetzungstabelle. Die Übersetzung - wenn eine festgelegt wurde - übernimmt in der Regel spezielle Hardware, die sogenannte Memory Management Unit (MMU). Diese liegt meistens direkt mit auf der CPU. Hierzu sagt das Betriebssystem der MMU, welcher Prozess gerade aktiv ist, indem die entsprechende Übersetzungstabelle als aktive Übersetzungstabelle festgelegt wird

Auslagerung (Swapping)

Bei der Auslagerung werden aktuell nicht benötigte Speicherbereiche vom Arbeitsspeicher auf die langsamere Festplatte verschoben. Dadurch schafft man neuen Platz im Arbeitsspeicher für aktive Prozesse, und hat somit quasi mehr Arbeitsspeicher als eigentlich physisch vorhanden. Wenn auf die ausgelagerten Daten wieder zugegriffen werden soll, müssen diese erst wieder in den Arbeitsspeicher eingelagert und ggf. ein anderer Speicherbereich ausgelagert werden (tauschen => swapping). Dieser Vorgang ist langsam und daher ist es ratsamer, mehr physischen Arbeitsspeicher in den Rechner einzubauen, als den Arbeitsspeicher um die Festplatte zu erweitern

Paging

Bei Paging wird der Arbeitsspeicher in gleich große Bereiche (z.B. 4 KiB), sogenannte Pages, unterteilt. Pro Page, die verwendet werden soll, ist immer ein Eintrag in der Übersetzungstabelle (in diesem Kontext dann Page Table genannt) nötig

Wenn ein Prozess nun Daten ablegen möchte, dann reserviert das Betriebssystem automatisch immer gleich eine ganze Page für diesen Prozess. Selbst wenn er nur 4 Byte ablegen möchte, bekäme er z.B. trotzdem direkt 4 KiB an Speicherbereich zugewiesen. Dies führt zu sogenannter interner Fragmentierung: Dieser Begriff bezeichnet ungenutzten Speicher innerhalb einer Page, der zu viel reserviert wurde

Virt. Adresse	Phys. Adresse	Valid
...	...	...
0x3CF	0x4000	1
0x3D0	0x5000	1
...	...	...
0x400	0x2000	1
0x401	0x8000	1
...	...	...
0xF00	0xF000	1
...	...	...

Byte-Offset

Bei einer 4KiB(2^{12} B) großen Page benötigt man 12 Bit um jedes Byte der Page zu adressieren. Demnach sind die letzten 3 Ziffern der virtuellen Adresse der Byte-Offset (0x 00 40 1**B 04**) und dieser muss anschließend auf die physische Adresse addiert werden (0x8**B 04**)

Segmentierung

Bei der Segmentierung wird der Arbeitsspeicher - ähnlich wie bei Paging - in zusammenhängende Speicherbereiche aufgeteilt (sogenannte Segmente). Zusammenhängend bedeutet, dass Segmente nicht durchtrennt werden können; alle darin enthaltenen Bytes liegen sequentiell aufeinanderfolgend im Speicher

Im Gegensatz zu Pages sind diese Segmente aber dynamisch groß, abhängig davon, wie viel Speicher der Prozess tatsächlich benötigt. Würde ein Prozess beispielsweise 3 KiB benötigen, dann würde er ein 3 KiB großes Segment bekommen

Dies hat jedoch diverse Nachteile:

1. Die Übersetzung (mittels der sogenannten Segment Tables) wird komplexer: Man muss bei jeder virtuellen Adresse prüfen, ob sie innerhalb eines

dynamischen Bereichs liegt (definiert durch Startadresse + Größe des Segmentes). Dadurch ist das Segment nicht mehr berechenbar, sondern man muss es quasi suchen

2. Externe Fragmentierung: Wenn Speicher reserviert wird, und wieder freigegeben wird, dann können ungleich große Lücken zwischen Segmenten entstehen. Diese Lücken sind dann gegebenenfalls in der Summe groß genug für ein neues Segment, aber jedes einzelne ist nicht ausreichend groß. Da Segmente zusammenhängend sein müssen, kann man es nicht auf mehrere Lücken aufteilen

3. Verbunden damit ist auch eine allgemein komplexe Suche nach freiem Speicher. Da man nach möglichst ideal großen Lücken suchen muss, kann das Speicherreservieren länger dauern. Bei Paging hingegen sind ohnehin alle Pages gleich groß, also würde man hier die erste freie immer nehmen

Translation-Lookaside-Buffer (TLB)

Der TLB ist ein Spezialcache, der sich die letzten ca. 4 - 32 Adressübersetzungen merkt. Bei einer Adressübersetzung wird damit zuerst geprüft, ob die physische Adresse vielleicht bereits bekannt ist; und ansonsten wird die Übersetzung eben manuell über die Übersetzungstabellen durchgeführt und sich das Resultat davon gemerkt