Duale Hochschule Manheim DHBW

Kurs TINF21AI1

Rechnerarchitekturen I

CPU 1

Adressrechner mit Register eine Unterteilung (1/1)

Zugrunde liegt eine ALU die unsere Berechnungen durchführt. Hierbei unterscheidet man zwischen vier Möglichkeiten Daten zu behandeln.

0-Adressbefehle / Stack Rechner
Hierbei werden die Daten nur in einem Stack gehalten.

Beispiel Addition: a + b:

```
Push a ; Legt Zahl A auf den Stack
Push b ; Legt Zahl B auf den Stack
ADD    ; Addiert A und B und liefert Ergebnis (UPN)
```

Wichtig ist, dass dieser Mechanismus mit dem Stack in anderem Zusammenhang für die CPU wichtig wird.

1-Adressbefehle / Akkumulator Rechner
Verwendet Register zum Speichern der Ergebnisse.

Beispiel Addition: a + b:

```
LDA a ; Lädt a in den Akkumulator
ADD b ; Addiert b auf den Wert (a) im Akkumulator
STA a ; Speichert Wert im Akkumulator nach Adresse a
```

Adressrechner mit Register eine Unterteilung (1/2)

2-Adressbefehle klassische CISC-Rechner
Diese Art von CPUs haben einen Registersatz von 8, 16 oder 32 Registern, die für Berechnungen genutzt werden.
Beispiel Addition: a + b:

```
Load R1, a ; Lade a in R1
Load R2, b ; Lade b in R2
ADD R1, R2 ; Addiere R1 und R2
```

In der Regel liegen bei älteren Prozessoren die Ergebnisse im ersten Register (R1).

3-Adressbefehle RISC-Rechner
Diese Art von CPUs betrachtet Register als „generelle Register“. Hierbei können auch 3-Port-Speicherzugriff angewandt werden.

Beispiel Addition: a + b:

```
Load R1, a ; Lade a in R1
Load R2, b ; Lade b in R2
ADD R2, R2, R1 ; Lege das Ergbnis von R1 + R2 in R1 ab
```

Moderne CPUs haben sich in Richtung RISC entwickelt, daher ist diese Unterteilung etwas grob. Im Rahmen der Kompatibilität müssen auch die früheren Zugriffsmethoden weiter unterstützt werden.

CPU Befehlsarten

Um die CPU effizient durch Code nutzbar zu machen, haben sich vier Strömungen entwickelt:

- **Complex Instruction Set Computing (CISC)**
  Es wird ein umfangreicher Befehlssatz für den Assembler zur Verfügung gestellt. Dies können bis zu 1000 verschiedene Assembler Befehle sein.
- **Reduced Instruction Set Computing (RISC)**
  Der Umfang der Assembler Befehle wurde deutlich verkleinert und Register wurden generalisiert.
- **Very Long Instruction Word (VLIW)**
  Die Parallelität beim Ausführen der Assembler Befehle wird durch den Compiler erzeugt und optimiert. Damit werden sehr komplexe Befehle erzeugt.
- **Explicitly Parallel Instruction Computing (EPIC)**
  Mit Hilfe des Compilers wird festgelegt welche Befehle parall ausgeführt werden. Es wird auch stark in der CPU mit spekulativen Datenabfragen gearbeitet.

Es gab eine Abwendung der reinen Assembler-Programmierung, da dies im Gegensatz zu modernen Hochsprachen, weitaus aufwendiger ist. Des Weiteren haben Untersuchungen gezeigt, dass das selbstständige Belegen von Speicher sehr fehleranfällig ist, deswegen wird heutzutage auf eigne Speichermanager gesetzt. Moderne Hochsprachen folgen dem RISC Ansatz, da bspw. ein Compiler, dem Assembler viele Aufgaben abnehmen kann (Codeoptimierungen usw.).

CISC

Am Beispiel des Z80 kann man den Umfang einer CISC CPU gut verdeutlichen. Es konnte nur auf den Assembler zurückgegriffen werden, da Compiler noch teuer und technisch nicht ausreichend ausgereift waren.

Beispiel Z800:

- 700 Assembler Befehle
- 400 undokumentierte Befehle, die verwendet werden mussten, da diese performanter waren

Man konnte diese Befehle in vier Gruppen einteilen:

- Transportbefehle => Lade oder speichere Daten in Register, I/O oder den Speicher
- Verarbeitungsbefehle => ADD, SUB, Schiebregister, usw.
- Befehle zum steuern eines Programablaufs => Sprünge, Unterprogrammaufrufe
- CPU Steuerung => Halt, Interrupt usw.

Nachbilden der MUL, FPU, usw. erfolgt per Subprogram oder externer Hardware durch einen Coprozessor

RISC Betrachtung

Aus den bereits gezeigt CPU-Beispielen wird ersichtlich, dass RISC Assembler Befehle sehr Komplex werden können. Das heißt auch, dass für eine Realisierung dieser Befehle, eine Menge Transistoren benötig werden. Diese wiederum können dann an anderen Baugruppen, wie z.B. der FPU, fehlen.

- IBM analysierte den Befehlssatz des IBM 370 Mainframe. Von den 200 verfügbaren Befehlen des Befehlssatzes (IBM 370 machten :
  - 10 Befehle 80 % des Programmcodes,
  - 21 Befehle 95 % des Programmcodes und
  - 30 Befehle 99 % des Programmcodes aus
- Uni Berkeley (1982, Patterson, Sequin), Basis der SPARC-Architektur (SUN)
- Uni Stanford (1981, Hennessy), Stanford MIPS, Basis der Architektur der MIPS-CPU

RISC Anforderung:

- Keine Mikroprogrammierung
- Durch wenige, einfache Befehle Verzicht auf Mikroprogramm möglich
- Steuerwerk in Hardware ausgeführt
- Meist schneller in der Ausführung
- Mittlerweile auch einfacher nutzbar als zu Anfangszeiten von CISC
- Feste Registerlänge
- 3 Register Befehle

Assembler

Im Digitalteil der Vorlesung wurde gezeigt, dass das Ansteuern mit von **?** mit 0 und 1 erfolgt. Dies kann als kompakte hexadezimaler Schreibweise dargestellt werden. Zur besseren Lesbarkeit ist eine Mnemonik eingeführt worden. Diese wird von einem Assembler in eine binäre Darstellung übersetzt. Häufig steht ADD für die Addition mit einem Register, SUB die Subtraktion usw.

Generelle Syntax:

- Operation Zielregister, Quellregister 1
  `=> ADD R1, R2`
- Operation Zielregister, Quellregister1, Quellregister 2
  `=> ADD R1, R2, R3`

Bei der Verwendung von Hochsprache werden die entsprechen Elemente in Assemblercode übersetzt und anschließend in Maschinencode umgewandelt.

```
Beispiel: Zahl im Register ablegen:
mov a, #56
```

```
Beispiel: Addieren zweier Zahlen und das
Ergebnis im Register ablegen

int a = 5;
int b = 6;
a = a + b:

mov a, #5 ; Beschreibe Register a mit 5
mov b, #6 ; Beschreibe Register b mit 6
add a, b  ; Addiere Register a und b,
wobei das Ergebnis in a zu finden ist
```

Komplexe Befehle

Wie bereits gezeigt, sind auf alten CPUs nur Additionen möglich. Um Multiplikationen auszuführen, ist auf der CPU in der Regel dedizierter Programmcode hinterlegt, welcher nach Bedarf ausgeführt wird. Wie an „Bespiel: Multiplikation“ zu sehen ist, ist eine Multiplikation aufwendiger als eine Addition. Um dies zu beschleunigen wurden Coprozessoren entwickelt, die extern nachgerüstet werden konnten. Da die Kommunikation zwischen CPU und CoCPU über einen Bus erfolgte, wurde die Kommunikation beeinträchtigt und so entsprechend verlangsamt. In modernen CPUs sind diese Elemente in der CPU integrierten. Damit hat der Bus fast nur noch die Aufgabe Daten von und ins RAM zu transportieren.

Auch GPUs werden heutzutage, auf Grund ihrer Parallelisierungsmöglichenkeiten, für rechenintensive Operationen verwendet.

```
mul a, b # multipliziere Register a mit b

mul (a,b)
{
        add a, b ; addiere a mit b
        shift b
        add a, b ; addiere a mit b
        shift b

…
        # bis alle Bits verarbeitet
        sind
}
```

Beispiel: Multiplikation

Grundstruktur einer Assembler Abarbeitung in einer CPU

1. Das Businterface (Cache) ist für den Fetch der Daten zuständig
2. Der Instruktion Decoder ist für das Dekodieren des Befehls oder Speichern der Daten im Register zuständig
3. Die ALU oder andre Einheiten, sind dafür zuständig die Befehle ausführen
4. Wenn der Befehl ausgeführt wurde, werden die Daten zurückgeschrieben

Für Überwachung und Synchronisation der Prozesse ist die Control Unit verantwortlich

Ablauf
Fetch
Decode
Execute
Write

CPU Register

- **Data Register:** Halten die Daten, die von der CPU verarbeitet werden
- **Flag Register:** Verwaltet die aktuellen Zuständen und Informationen der CPU, (Z.B. Zero Bit, Carry Flag usw.)
- **Pointer/Adress Register:**
  - **Program/Instruktion pointer:** Hält die Zieladresse des nächsten Speicherblocks. Dieser kann für bspw. Sprünge (branches) verändert werden.
  - **Stack pointer:** Ist die Adresse an dem der Heap beginnt. Hier werden Daten nach dem FIFO Prinzip abgelegt.
  - **Index register:** CPU abhängig vorhanden. Wird als Index bei einem Andresszugriff verwendet

Übersicht der Grundlegenden Adressierungsarten

Um Daten aus dem RAM in die CPU zu laden, gibt es eine Vielzahl von Adressierungsmöglichkeiten. Diese sich unterscheiden sich von Hersteller zu Hersteller. Daher werden nur die grundlegenden Adressierungsmöglichkeiten betrachtet.

- **Immediate Adressing:** Diese Art verwendet einem konstanten Wert als Argument, daher sind keine weiteren Speicherzugriff notwendig

```
ARM:   mov R0, #22
6502:  LDA #22 ; Lade den Wert 22
```

- **Direct/Absolute Addressing:** Der Befehl enthält die absolute Speicheradresse. Damit wird das Laden mittels Speicherzugriff angestoßen. Dies wir bei RISC Prozessoren anderes gelöst.

```
6502: LDA $25 ; Lade Daten aus Adresse 0x25
```

- Extended Adressing: Hier bei wird direkt auf eine Adresse zugegriffen im LDAA 1000h.

- **Index Adressing:** Beispiel:

```
6502: LDA ($25,X) ; Lade Daten aus Adresse 0x25 + X (beliebige Hex Nummer)

X ist der Offset um den die Adresse verschoben wird
```

Übersicht der Grundlegenden Adressierungsarten

- Register Adressierung: (RISC, z.B. ARM) Lade den Inhalt des Registers mit dem eines anderen Registers
  `ARM: mov R0, R1`

- Register indirekte adressing: (RISC, z.B. ARM) Im Register ist die Adresse hinterlegt an der die eigentlichen Daten liegen.
  `ARM: LDR, R1 [R0] ; Verwende die Adresse von R0 und lade diese in Register R1.`

  Dies wird z. B. in Arrays verwendet

- Register indirekte Adressing with Offset: Hier wird zusätzlich noch ein Offset auf die Adresse gerechnet
  `ARM: LDR, R1 [R0, #10]`

Der Grund warum es diese Offsets gibt hat

Stack/Heap

- Bildet eine FIFO Struktur
- Der Stack Pointer hält die aktuelle Adresse des Stacks.
- Drei Befehles Gruppen:
  - PUSH: Schiebe ein Register, Pointer oder das komplette Flagregister in den Stack
  - POP: Hole einen Wert vom Stack. Dies kann ein Register, Pointer oder das Komplette Flagregister sein
  - PEEK: Schaue den obersten Inhalt an
- Es gibt darüber hinaus je nach CPU Hersteller noch weitere Befehle um den Stack zu verändern

Der Stack kann im Rahmen der Softwareentwicklung verwendet werden, z. B. für rekursive Aufrufe

Der Stack kann auch alle Register oder die Flag Inhalte aufnehmen. Dies ist wichtig um Interrupts zu behandeln und das Programm an der richtigen Stelle wieder einsteigen kann.

Ein Stack Overflow, ist ein Fehler des Stacks, bei welchem der Speicher vollläuft und das Programm beginnt in nicht definierte Bereiche zu schreiben.

Interrupt

Früher mussten ebenfalls Daten von externen Geräten abgefragt werden. Dazu wurde eine Art Polling betrieben, was bedeutet, dass ein bestimmtes Gerät zyklisch nach Daten gefragt wird. Dies hatte den Nachteil, dass Daten nicht in Echtzeit verarbeitet werden konnten, sondern erst zu einem späteren Zeitpunkt.

Um dem entgegen zu wirken, wurden Interrupts eingeführt.

Um einen Interrupt auszuführen, setzt ein Interrupt Controller ein Signal auf den Steuerbus und führt danach den Aufruf an das externe Geräte durch.

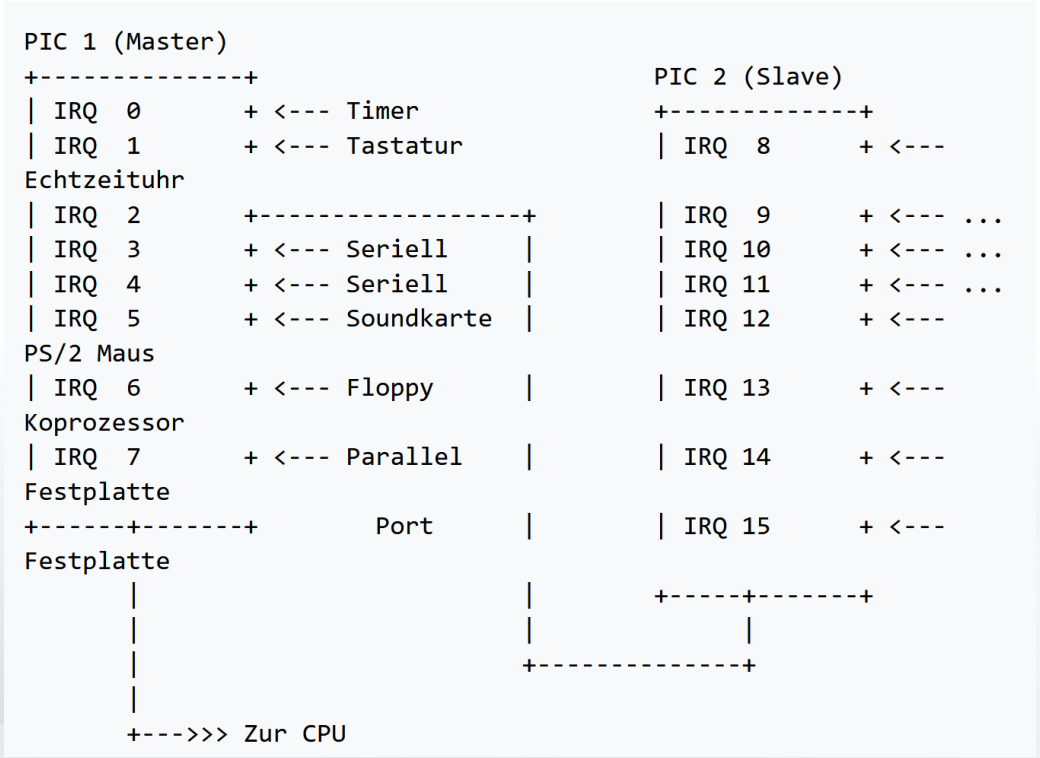
Dies hat in der PC Welt lange ein Problem dargestellt, da der Controller recht klein bemessen wurde.

Interrupt

Es zwischen zwei Arten von Interrupts unterschieden:

- **Maskable:** Diese Interrupts können nicht unterdrückt werden, was bedeutet, dass das ausgeführte Programm stoppt und der Interrupt behandelt wird

- **Non Maskable:** Diese Interrupts können vom Programm unterdrückt werden und zu einem späteren Zeitpunkt behandelt werden



https://de.wikipedia.org/wiki/Intel_8259

Interrupt Handling

Folgende Schritte werden beim Interrupt Handling durchgeführt:

- Es wird ein Interrupt aufgerufen: Sichern aller Register, Pointer und Flags auf den Stack
- Interrupt Vektor wird aufgerufen: Der Interrupt Vektor ist die Programmadresse, zu welcher gesprungen, wenn ein Interrupt behandelt werden muss. Dieser Vektor ist die Interrupt Service Routine kurz ISR.
- Nach Beendigung der Interrupt Handling werden die Register, Pointer und Flags vom Heap zurück geschrieben und das Programm läuft weiter.

Auf Assembler Ebene besteht auch die Möglichkeit einen Software Interrupt auszulösen, wie bspw. `INT xxh` bei Intel.

Im Speicher ist darüber hinaus ein Speicherbereich definiert in dem Hardware die zugeordneten ISR finden kann. Dies ist heute eleganter gelöst, da das Betriebssystem und die moderner Implementierung der Hardware hier deutlich intelligenter vorgehen können.

Interrupt Vector Tabelle	
IRQ 0	ISR 0 Adresse
IRQ 1	ISR 1 Adresse
IRQ 2	ISR 2 Adresse
…	

Head Advance

Es gibt gesonderte Befehle, um den Heap direkt anzusprechen.

Subroutine

- Wenn eine Subroutine aufgerufen wird, geschieht dies mittels eines „`CALL xxh`“ . Dieser Befehl sichert den Instruktion Pointer auf dem Heap. Je nach Implementierung können auch weitere Register und/oder Flags gesichert werden.

- Mit einem Return (`6502: RTS`) Befehl wird in der Regel die Subroutine beendet. Damit wird der Instruktion Pointer und/oder die weiteren Register und Pointer vom Heap zurück geschrieben.

Subroutinen sind die früheren Funktions-Calls: Sie Gliedern das Programm in verschiedene Aufgaben auf, wie z. B. erweiterte Berechnungen. Dafür müssen sie auch auf Register und Flags zugriff haben.

Software Interupt

- Wird ein Software Interrupt aufgerufen, dann werden der Instruktion Pointer und das Flag Register auf den Heap kopiert. Es können auch weitere Register gesichert werden. Beispiel: `INT xxh;`
- Das Zurückspringen wird mit „Return From Interrupt“ ausgeführt. Dies schreibt ebenfalls den Instruktion Pointer und das Flag Register zurück. Beispiel: `IRET;`

Vorbetrachtung zu RISC und Co

Am Beispiel des Z80 wurde gezeigt, dass Assembler-Programmierung sich als aufwendig erweisen kann.

Mit RISC hat sich jedoch eine neue Technik durchgesetzt.

Ein wesentliches Element sind schnelle Speicherzugriffe, die mit Hilfe eines Caches erreicht werden.

Der zweite Schritt ist der Einsatz einer Pipeline, welche aus vier Zyklen besteht.

Diese kann man geschickt gruppieren und damit die Funktionseinheiten der CPU optimieren.

Cache Grundlagen

Zum cachen der Daten werden die Adressen in Blöcke aufgeteilt. Hierbei werden aus den letzten Bits jeweils die Word IDs gebildet. Das Problem hierbei ist die richtige Wahl der Blockung. Je größer die Blöcke werden, desto weniger unterschiedlich sind die Daten im Cache. Jedoch umso größter die Blöcke, desto schneller die Zugriffe. Der RAM ist dabei mit zu beachten.

Adresse	
1010 1010 0000	
Block ID	Word ID
1010 1010 00	00

Adresse	Daten
1010 1010 0000	9
1010 1010 0001	3
1010 1010 0010	5
1010 1010 0011	6
1010 1010 0100	A
1010 1010 0101	B
1010 1010 0110	C
1010 1010 0111	2
1010 1011 0000	1
1010 1011 0001	9
1010 1011 0010	E
1010 1011 0011	F

Fully Associative Caches

Fully Associative Caches beschreibt eine Strategie, bei welcher die Daten im Cache nach Blöcken assoziiert werden. Diese Strategie ist für den Cachecontroller simpel durchzuführen und daher schnell in der Ausführung.

Der Cache in „Abbildung: Cache“ hat keine Konkurrenzprobleme.

Der Nachteil an ihm: Hohen Kosten der verbauten Chips. Somit befindet er sich im L1 Bereich.

Um eine Adresse abzufragen wird die BlockID gebildet und bestimmt ob die Daten im Cache liegen. Im Falle eines Erfolges gibt es einen Cache Hit. Im Falle eines Misserfolges einen Cache Miss. Daraufhin müssen die Daten im Speicher oder einem tieferen Cache gesucht werden.

Der gefundene Datensatz ersetzt dann eine vorhandene Line im Cache. Dies geschieht mit Hilfe einer der beschriebenen Replacement Strategien

Adresse	Daten
1010 1010 0000	9
1010 1010 0001	3
1010 1010 0010	5
1010 1010 0011	6
1010 1010 0100	A
1010 1010 0101	B
1010 1010 0110	C
1010 1010 0111	2
1010 1011 0000	1
1010 1011 0001	9
1010 1011 0010	E
1010 1011 0011	F

Abbildung: Cache

Tag	Block			
	11	10	01	00
1010 1010 00	6	5	3	9
1010 1010 01	2	C	B	A

Direct Mapped Caches

Diese Art von Cache führt wird eine weitere Gruppierung ein: die Line. Bei einer Anfrage wird die die Line bestimmt und es muss nur noch ein Tag geprüft werden.

Der Nachteil ist, dass es bei Adressen die die gleiche Line besitzen zu einer Konkurrenz Situation kommen kann, welche die Performance negative beeinflussen kann.

Daher teilt sich der Platz einer LineID mit der Anzahl Adressen in Relation zum Adresse Umfang der CPU.

Alle Adressen die an Position 12, 13 und 14 eine Null stehen haben, können in dieser Line gespeichert werden.

Wird häufig in den Logs als Trashing aufgeführt.

Adresse		
1000 1010 1010 0000		
Tag	Line ID	Word ID
1000 1010 101	000	00

Direct Mapped Caches

Kurzes Beispiel:

1111 1010 0101

Was ist hier bei der Programmierung zu beachten?

Tag	LineID	Block			
		11	10	01	00
1010 1010	00	6	5	3	9
1010 1010	01	2	C	B	A
1010 1010	10				
1010 1010	11				

Adresse	Daten
1010 1010 0000	9
1010 1010 0001	3
1010 1010 0010	5
1010 1010 0011	6
1010 1010 0100	A
1010 1010 0101	B
1010 1010 0110	C
1010 1010 0111	2
1010 1011 0000	1
1010 1011 0001	9
1010 1011 0010	E
1010 1011 0011	F

Set-Associative Caches

Dieser Cache ist eine Kreuzung zwischen Fully Associative Caches und Direct Map Caches. Im Gegensatz zum Direct Map Cache wird jedoch nicht eine Line gespeichert, sondern mehrere.

Nun sind zwei verschiedene Adressen in diesem Beispiel gegeben. Sollte eine weitere Adresse mit dem gleichen Muster erscheinen, verdrängt diese eine bestehende nach einer der vorhergehenden Strategien. Man spricht auch von k-way assoziative Cache.

In diesem Beispiel wäre das ein 2-way set assoziative Cache, für die Tags die gespeichert werden.

Adresse		
1000 1010 1010 0000		
Tag	Set ID	Word ID
1000 1010 1010	00	00

Tag	SetID	Block			
		11	10	01	00
1010 1010 00	00	6	5	3	9
1010 1010 10		4	6	7	8

Set-Associative Caches

Kurzes Beispiel:

1111 1010 0101

Was ist hier bei der Programmierung zu beachten?

Tag	SetID	Block			
		11	10	01	00
1010 1010	00	6	5	3	9
1010 1010	01				
1010 1010	10				
	11				

Adresse	Daten
1010 1010 0000	9
1111 1010 0001	7
1010 1010 1010	9
1010 1010 1100	A
0111 1111 0010	F

Flags eines Caches für die Lines

Jede Zeile im Cache hat weiterführende Flags, die der Verarbeitung und Wartung der Einträge dienen.

Die gebräuchlichsten sind:

- T = Type:
  Handelt es sich um Daten oder Instruktion?
- V = Valid:
  Sind die Daten im Cache noch gültig oder haben diese sich in der Zwischenzeit geändert? Hiermit kann auch der Cache geleert werden
- L = Lock:
  Sperrt die Daten im Cache, damit diese nicht geändert oder flushed werden können
- D = Dirty:
  Markiert Daten die geschrieben werden müssen und noch nicht mit dem RAM synchronisiert wurden

Anmerkung:

- Unified Cache:
  Data und Instruktion sind gemeinsam im Cache, daher T und D Flag zwingend notwendig
- Split Cache:
  Date und Instruktion sind in getrennten Caches untergebracht. Damit wird das T Flag nicht benötigt und das D-Flag nur im Datencache, da der Instruktion Cache nur gelesen wird

Data Structure Alignment

Wie bereits besprochen, funktioniert ein Cache am besten, wenn die Daten in einer sortierten Form vorliegen. Variablen die über den ganzen Code verteilt sind führen zu Performance-Einbußen da die Daten immer wieder nachgeladen werden müssen. Alter Code ist häufig kompakte geschrieben und kann somit einen Cache ineffizient werden lassen. Moderne Compiler berücksichtigen dies jedoch und versuch es zu optimieren.

Gegeben sei ein 16 Bit Datenwert *(siehe Blau)*, dieser liegt in zwei Cache Lines und zwei Variablen mit je einem Bytewert *(siehe grün)*. Diesee liegen in zwei Cache Lines. Dies sollte der Compiler erkennen und umsortieren.

N-Byte Werte müssen durch n teilbar sein.

Beispiel:
- 16 Bit eine teilbare Adresse mit 2
- 32 Bit eine teilbare Adresse mit 4

Adresse	Line	Daten
1010 1010 0000	A	9
1010 1010 0001	A	3
1010 1010 0010	A	5
1010 1010 0011	A	6
1010 1010 0100	B	A
1010 1010 0101	B	B
1010 1010 0110	B	C
1010 1010 0111	B	2
1010 1011 0000	C	1
1010 1011 0001	C	9
1010 1011 0010	C	E
1010 1011 0011	C	F

Pipeline

Wir haben gesehen das sich der Zyklus einen Befehls aus den vier Schritten besteht:

- Fetch
- Decode
- Excecute
- Fetch (write)

In der alten Welt konnte der Start des nächsten Befehls erst nach dem Fetch erfolgen. Wir haben auch den Umstand, das in die Funktionseinheiten der CPU nur dann genutzt werden in dem Funktionsschritt der gerade aktiv ist. Somit ist jede Einheit nur etwa den viertel der Zeit beschäftigt.

Mit einer Pipeline kann ich dies optimieren.
Wie wir im Bild gesehen haben muss die Pipeline erst einmal beladen werden um die Effizienz zu erzeugen.

Zyklus	Fetch	Decode	Execute	Fetch
1	**Befehl 1**			
2	Befehl 2	**Befehl 1**		
3	Befehl 3	Befehl 2	**Befehl 1**	
4	Befehl 4	Befehl 3	Befehl 2	**Befehl 1**

Betrachtung der Pipeline

Welche Herausforderungen stelle eine Pipeline an die Programmierung?

Die Pipeline arbeitet schnell wenn keine Lücken entstehen oder sie auf nicht Befehle warten muss. In modernen Programmiersprachen versucht der Compiler dies zu vermeiden und den Code dahingehend zu optimieren. Leider lässt sich dies nicht immer verhindern, was zur Folge hat, dass die Performance schwanken kann.

Zyklus	Fetch	Decode	Excecute	Fetch
1	Befehl 1			
2	Befehl 2	Befehl 1		
3	Befehl 3	Befehl 2	Befehl 1	
4	Befehl 4	Befehl 3	Befehl 2	Befehl 1

Unterteilung der Probleme

- **Strukturkonflikte**:
  Eine Funktionseinheit wird in mehreren Schritten gleichzeitig gebraucht
  FPU wird von Befehl 1 und 2 verwendet
- **Datenkonflikte**:
  Ein Befehl benötigt das Ergebnis eines vorhergehenden Befehls
- **Steuerungskonflikte**:
  Sprungbefehl, Interrupt usw. werden abgearbeitet. Wenn diese Befehle in den Fetch Zyklus kommen, hat die Pipeline bereits weitere Befehle angefangen zu bearbeiten. Sprungvorhersagen oder andere Mechanismen versuchen dies zu verhindern.

Pipeline RAR

Read after Read (RAR)

Zwei Anweisungen lesen beide aus demselben Register.
Da der Inhalt des Registers unverändert bleibt ist dies im Moment kein Problem.

Beispiel

- `ADD .., R5 ; Addiere Register R5 zu einem anderen Wert`
- `ADD .., R5 ; Addiere Register R5 zu einem anderen Wert`

Pipeline RAW

Read after Write (RAW)

Ein Befehl verwendet einen Wert der von einem vorhergehende Befehl berechnet wird.
Hier muss der folge Befehl solange pausieren bis sein Ergebnis fertig ist.

Beispiel

- `ADD R1 … ; Addiere Werte und speichere diese in R1`
- `AND … R1 ; UND-Verknüpfe R1 mit einem anderen Wert`

Pipeline WAR

Write after Read (WAR)

Der erst Befehl liest aus einem Register welches vom nachfolgenden Befehl verändert wird.
Hierbei kann das Schreiben beendet sein bevor das Lesen erfolgte. Dies kann bei Parallelisierung mit mehreren Pipelines oder durch unterschiedlichen Komplexitäten der Befehle auftreten.

Beispiel:

- `ADD … R2   ; Lese aus R2 um etwas zu addieren.`
- `ADD … R2 … ; Schreibe mein Ergebnis in R2.`

Pipeline WAW

Write after Write (WAW)

Das Register für das Ergebnis wird durch einen Befehl beschrieben und der Folgebefehl tut dies auch.
Der zweite Befehl könnte vor dem ersten fertig sein. Dies kann bei Parallelisierung mit mehreren Pipelines oder durch unterschiedlichen Komplexitäten der Befehle auftreten.

Beispiel
ADD ...R2 ; Speichere die Addition in Register R2
ADD ...R2 ; Speichere die Addition in Register R2

Pipeline stall

Pipeline Stall bezeichnet eine Bearbeitungspause der Pipeline. Dies tritt auf wenn der Folgebefehl auf ein Ergebnis warten muss:

- `ADD R2… ; Addiere etwas und speichere es in R2`
- `ADD R2..; Addiere etwas mit R2 und speichere es in R1  < Muss warten`

- `AND …; And Befehl`
- `Sub …; Subtrahiere etwas`
- `ADD …; Addiere etwas`

Zyklus	Fetch	Decode	Execute	Store/write
1	**add**			
2	Add R2	add		
3	Add R2		add	
4	Add R2			add
5	and	add		
6	Sub	and	add	
7	add	sub	and	add
8		add	sub	and
9			add	sub
10				add

Pipeline stall 2

Pipeline Stall wird durch reorder der Befehle gelöst. Damit kann die Pipeline ohne warten arbeiten.

- `ADD R2…; Addiere etwas und speichere es in R2`
- `AND …; And Befehl`
- `Sub …; Subtrahiere etwas`
- `ADD …; Addiere etwas`
- `ADD R2..; Addiere etwas mit R2 und speichere es in R1`

Zyklus	Fetch	Decode	Execute	Store/write
1	**add**			
2	And	add		
3	Sub	And	add	
4	Add	Sub	And	add
5	Add R2	Add	Sub	And
6		Add R2	Add	Sub
7			Add R2	Add
8				Add R2
9				

Quelle

https://en.wikipedia.org/wiki/Main_Page

https://de.wikipedia.org/wiki/Wikipedia:Hauptseite

https://de.wikipedia.org/wiki/Adressrechner_(Maschinenbefehl)

https://en.wikipedia.org/wiki/Data_structure_alignment

https://de.wikipedia.org/wiki/Intel_8259

https://de.wikipedia.org/wiki/Software-Interrupt

https://techcommunity.microsoft.com/t5/core-infrastructure-and-security/windows-10-memory-protection-features/ba-p/259046