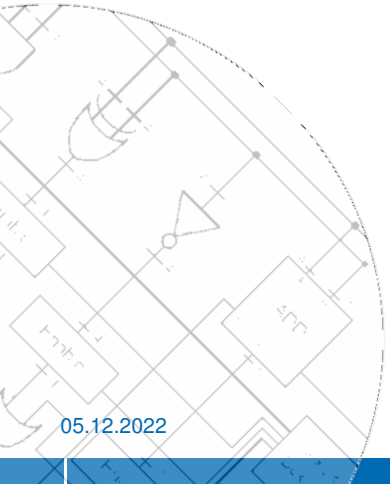
Rechnerarchitektur

Vorlesung 13: Caching mehr Details, Core i7, Kapitelabschluss, Einführung Assembler

Prof. Dr. Martin Mauve



05.12.2022

Haben Sie noch Fragen zur letzten Vorlesung?

Thema: Optimierung der Mikroarchitektur: Branch Prediction und Caching am Beispiel

Wiederholung: Caching

- schneller aber kleiner Speicher
- Teile langsamerer Speicher werden in den Cache geladen
- Identifikation dieser Teile über den TAG, manchmal auch die Zeile in der abgelegt wird
- Verschiedene Cache-Arten:
  - Fully Associativ Cache (jeder Eintrag irgendwo; Ersetzungsstrategie nötig)
  - Direct Mapped Cache (Adresse bestimmt Position im Cache)
  - n-Way Set-Associative Cache (Mischform; Ersetzungsstrategie nötig)

Fahrplan

Ersetzungsstrategien

- Für die ideale Ersetzungsstrategie müsste man in die Zukunft schauen können:
  - Welche Zeile wird lange nicht mehr gebraucht?
  - Unrealistisch . . . – also brauchen wir Heuristiken!
- Least Recently Used (LRU)
  - Wähle die Zeile, die am längsten nicht mehr in Verwendung war.
- First-In First-Out (FIFO)
  - Übersetzung: Was zuerst hineingeladen wurde, wird als erstes ersetzt.
  - Idee: Ersetze die „älteste“ Zeile, die am längsten im Cache war.
- Viele weitere, teils recht komplex: zufallsgesteuert, adaptiv, . . .
- Für alle Strategien gibt es pathologische Fälle, in denen die Anzahl der Ladevorgänge maximal wird!

Beispiel: FIFO in Fully Assoviative Cache – I

LINE	Valid	TAG	FIFO
0	0		
1	0		
2	0		
3	0		

- Speicherkapazität 16 Byte
- Zeilenlänge : 4 Byte
- Wort : 4 Byte
- Adresslänge : 8 Bit
- Aufteilung:

TAG	LINE	WORD	BYTE
7 .. 2	—	—	1 0

Zugriffe : 0, 8, 7, 13, 6, 2, 16

Beispiel: FIFO in Fully Assoviative Cache – II

LINE	Valid	TAG	FIFO
0	1	0	1
1	0		
2	0		
3	0		

- Speicherkapazität 16 Byte
- Zeilenlänge : 4 Byte
- Wort : 4 Byte
- Adresslänge : 8 Bit
- Aufteilung:

TAG	LINE	WORD	BYTE
7 .. 2	—	—	1 0

Zugriffe : 0, 8, 7, 13, 6, 2, 16

000000|00

0

Beispiel: FIFO in Fully Assoviative Cache – III

LINE	Valid	TAG	FIFO
0	1	0	1
1	1	2	2
2	0		
3	0		

- Speicherkapazität 16 Byte
- Zeilenlänge : 4 Byte
- Wort : 4 Byte
- Adresslänge : 8 Bit
- Aufteilung:

TAG	LINE	WORD	BYTE
7 .. 2	—	—	1 0

Zugriffe : 0, 8, 7, 13, 6, 2, 16

000010 00
2

Beispiel: FIFO in Fully Assoviative Cache – IV

LINE	Valid	TAG	FIFO
0	1	0	1
1	1	2	2
2	1	1	3
3	0		

- Speicherkapazität 16 Byte
- Zeilenlänge : 4 Byte
- Wort : 4 Byte
- Adresslänge : 8 Bit
- Aufteilung:

TAG	LINE	WORD	BYTE
7 … 2	—	—	1 0

Zugriffe : 0, 8, 7, 13, 6, 2, 16

000001 | 11

1

Beispiel: FIFO in Fully Assoviative Cache – V

LINE	Valid	TAG	FIFO
0	1	0	1
1	1	2	2
2	1	1	3
3	1	3	4

- Speicherkapazität 16 Byte
- Zeilenlänge : 4 Byte
- Wort : 4 Byte
- Adresslänge : 8 Bit
- Aufteilung:

TAG	LINE	WORD	BYTE
7 .. 2	—	—	1 0

Zugriffe : 0, 8, 7, 13, 6, 2, 16

000011|01

3

Beispiel: FIFO in Fully Assoviative Cache – VI

LINE	Valid	TAG	FIFO
0	1	0	1
1	1	2	2
2	1	1	3
3	1	3	4

- Speicherkapazität 16 Byte
- Zeilenlänge : 4 Byte
- Wort : 4 Byte
- Adresslänge : 8 Bit
- Aufteilung:

TAG	LINE	WORD	BYTE
7 … 2	—	—	1 0

Zugriffe : 0, 8, 7, 13, 6, 2, 16

000001 | 10

1

Beispiel: FIFO in Fully Assoviative Cache – VII

LINE	Valid	TAG	FIFO
0	1	0	1
1	1	2	2
2	1	1	3
3	1	3	4

- Speicherkapazität 16 Byte
- Zeilenlänge : 4 Byte
- Wort : 4 Byte
- Adresslänge : 8 Bit
- Aufteilung:

TAG	LINE	WORD	BYTE
7 … 2	—	—	1 0

Zugriffe : 0, 8, 7, 13, 6, 2, 16

00 0000 | 10

0

Beispiel: FIFO in Fully Assoviative Cache – VIII

LINE	Valid	TAG	FIFO
0	1	0	1
1	1	2	2
2	1	1	3
3	1	3	4

ältester Eintrag!

- Speicherkapazität 16 Byte
- Zeilenlänge : 4 Byte
- Wort : 4 Byte
- Adresslänge : 8 Bit
- Aufteilung:

TAG	LINE	WORD	BYTE
7 .. 2	—	—	1 0

Zugriffe : 0, 8, 7, 13, 6, 2, 16

000100 | 00

4

Beispiel: FIFO in Fully Assoviative Cache – IX

LINE	Valid	TAG	FIFO
0	1	4	7
1	1	2	2
2	1	1	3
3	1	3	4

- Speicherkapazität 16 Byte
- Zeilenlänge : 4 Byte
- Wort : 4 Byte
- Adresslänge : 8 Bit
- Aufteilung:

TAG	LINE	WORD	BYTE
7 .. 2	—	—	1 0

Zugriffe : 0, 8, 7, 13, 6, 2, 16

000100 | 00

4

Beispiel: LRU in Fully Assoviative Cache – I

LINE	Valid	TAG	LRU
0	1	0	1
1	1	2	2
2	1	1	5
3	1	3	4

- Speicherkapazität 16 Byte
- Zeilenlänge : 4 Byte
- Wort : 4 Byte
- Adresslänge : 8 Bit
- Aufteilung:

TAG	LINE	WORD	BYTE
7 … 2	—	—	1 0

Zugriffe : 0, 8, 7, 13, 6, 2, 16

000001|01

1

Beispiel: LRU in Fully Assoviative Cache – II

LINE	Valid	TAG	LRU
0	1	0	6
1	1	2	2
2	1	1	5
3	1	3	4

- Speicherkapazität 16 Byte
- Zeilenlänge : 4 Byte
- Wort : 4 Byte
- Adresslänge : 8 Bit
- Aufteilung:

TAG	LINE	WORD	BYTE
7 … 2	—	—	1 0

Zugriffe : 0, 8, 7, 13, 6, 2, 16

000000 | 10

0

Beispiel: LRU in Fully Assoviative Cache – III

LINE	Valid	TAG	LRU
0	1	0	6
1	1	2	2
2	1	1	5
3	1	3	4

LRU

- Speicherkapazität 16 Byte
- Zeilenlänge : 4 Byte
- Wort : 4 Byte
- Adresslänge : 8 Bit
- Aufteilung:

TAG	LINE	WORD	BYTE
7 … 2	—	—	1 0

Zugriffe : 0, 8, 7, 13, 6, 2, 16

0001 00 | 00

4

Beispiel: LRU in Fully Assoviative Cache – IV

LINE	Valid	TAG	LRU
0	1	0	6
1	1	4	7
2	1	1	5
3	1	3	4

- Speicherkapazität 16 Byte
- Zeilenlänge : 4 Byte
- Wort : 4 Byte
- Adresslänge : 8 Bit
- Aufteilung:

TAG	LINE	WORD	BYTE
7 … 2	—	—	1 0

Zugriffe : 0, 8, 7, 13, 6, 2, 16

0001 00 | 00

4

Schreiben mit Cache I

- Was passiert, wenn Daten in den Speicher geschrieben werden?
- Wenn der Inhalt der Adresse im Cache liegt:
  - In jedem Fall: Daten in den Cache schreiben
  - Alternative 1: Daten in den Speicher schreiben (write through)
    - Vorteil: Hauptspeicher hält immer die aktuellen Daten
    - Nachteil: mehr Speicherzugriffe
  - Alternative 2: Daten nur im Cache aktualisieren (write back)
    - Setzen eines „Dirty-Flags“
    - Zurückschreiben in den Speicher erst, wenn eine Cachezeile mit gesetztem Dirty-Flag aus dem Cache verdrängt wird.
    - Vorteil: Weniger Speicherzugriffe
    - Nachteile: Aufwändigeres Design, problematisch bei Mehrprozessorrechnern

Schreiben mit Cache II

- Wenn der Inhalt der Adresse *nicht* im Cache liegt:
  - In jedem Fall: Daten in den Hauptspeicher schreiben
  - Alternative 1: Daten in den Cache laden (write allocation)
    - Meist verwendet, wenn write back eingesetzt wird
    - Vorteil: Ausnutzen von räumlicher/zeitlicher Lokalität
    - Nachteil: Aufwändigeres Design
  - Alternative 2: Daten nicht in den Cache laden (no write allocation)
    - Wird meist in Zusammenspiel mit write through eingesetzt, um eine möglichst einfache Cache-Architektur zu realisieren.
    - Vor-/Nachteile: invers zu Alternative 1

Fahrplan

Sandy Bridge Mikroarchitektur

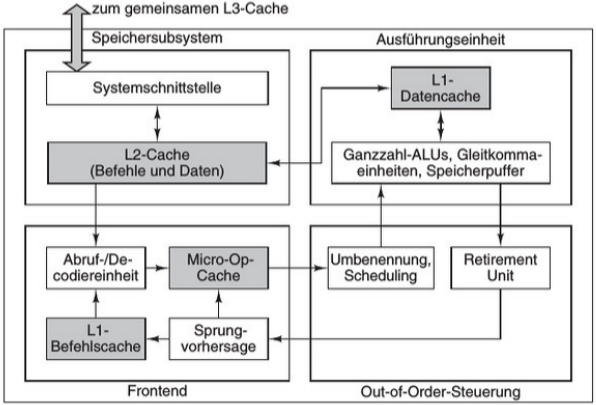


Abbildung 4.31: Blockdarstellung der Sandy-Bridge-Mikroarchitektur des Core i7

Bild: Rechnerarchitektur Von der Digitalen Logik zum Parallelrechner, 6. Auflage, Abb. 4.31

Core-i7 Datenpfad

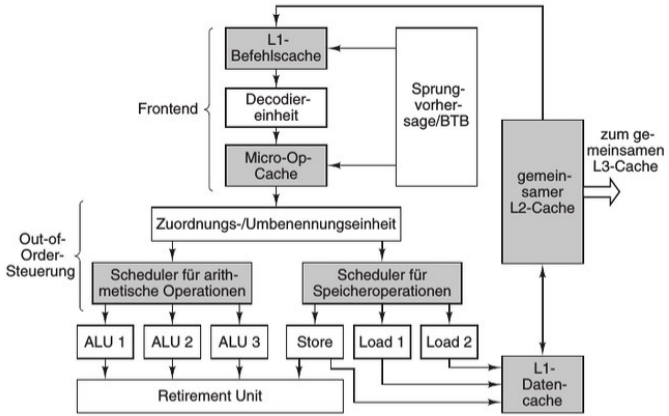


Abbildung 4.32: Vereinfachte Ansicht des Core-i7-Datenpfads

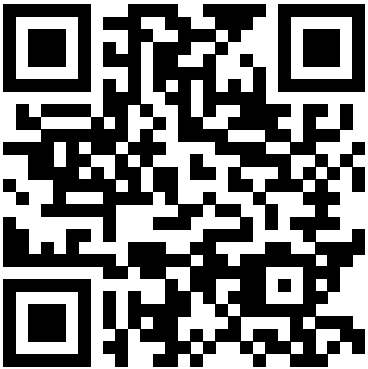
Bild: Rechnerarchitektur Von der Digitalen Logik zum Parallelrechner, 6. Auflage, Abb. 4.32

Fahrplan

Fragen

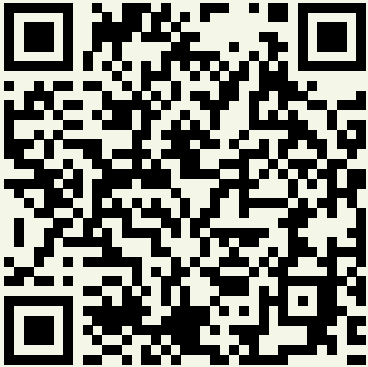
Quiz Mikroarchitektur

https://partici.fi/19125773



Anonymes Veranstaltungsfeedback: Mikroarchitektur

- Feedback zum Kapitel Mikroarchitektur
- **Ihre** Möglichkeit etwas zu loben oder zu verbessern
- Start: diese Woche Freitag, 12 Uhr
- Ende: nächste Woche Freitag, 12 Uhr
- `https://ilias.hhu.de/goto.php?target=svy_1386335&client_id=UniRZ`



Ziele dieses Kapitels

- arbeiten am Rechner
- Programme, die Sie ausführen können
- einen echten Prozessor erleben
- Funktionsaufrufe verstehen
  - auch rekrusiv

Ziele und Vorgehen

Ziele

- Das „Interface“ des Prozessors (Assemblersprache) verstehen.
- Verstehen, wie Funktionsaufrufe – auch rekursiv – realisiert werden.

Vorgehen

- Arbeiten am Rechner
- Programme, die Sie ausführen können
- einen echten Prozessor erleben

Bild von StockSnap auf Pixabay

Fahrplan

5 x86 Assembler
- Einstieg
- Grundlagen der x86 Architektur
- Programmaufbau eines Assemblerprogramms
- Arithmetische und Logische Instruktionen
- Verschiebungen und Rotationen
- Sprünge und Schleifen
- Der Stack
- Funktionen
- Von C zu Assembler
- Buffer Overflow Exploits
- Dynamische Speicherverwaltung

Was ist Assemblerprogrammierung?

- Prozessoren stellen einen so genannten Instruktionssatz zur Verfügung:
  - Das ist die Menge aller Instruktionen, die direkt vom Prozessor ausgeführt werden können
  - Beispiele: Addieren/Multiplizieren/Dividieren von zwei Zahlen, Zugriff auf den Speicher, Sprünge und Verzweigungen
- Der Instruktionssatz ist unabhängig vom Betriebssystem
- Die Instruktionen des Instruktionssatzes sind Binär- (bzw. Hexadezimal-)Zahlen:
  - Beispiel: 03 $C3_{16}$
    - IA-32 bzw. x86 (= 80x86 und Nachfolger) Befehl
    - addiert und speichert zwei Zahlen
- Damit zu arbeiten ist extrem fehleranfällig und unübersichtlich

Was ist Assemblerprogrammierung?

- Deshalb verwendet man eine symbolische Schreibweise:
  - `add eax, ebx`
  - `<mnemonic> <operanden>`
  - semantisch äquivalent zu 03 $C3_{16}$
- Ein Assembler ist ein Programm, das die symbolische Schreibweise in die entsprechenden binären Instruktionen übersetzt:
  - jede Instruktion in symbolischer Schreibweise = höchstens eine Instruktion in binärer Schreibweise (mit wenigen Ausnahmen)
  - Ergebnis des Übersetzens ist eine Objektdatei (.o), genau wie beim Übersetzen eines C-Programms
  - die Objektdatei enthält Objektcode
    - Format ist betriebssystemspezifisch

Frage:

Das ist immer noch sehr unübersichtlich, mühsam und fehleranfällig — also warum tun wir uns das eigentlich an?

Warum Assemblerprogrammierung?

- Drei zentrale Gründe:
  - Zur Optimierung von besonders kritischen und wichtigen Stellen eines Programms:
    - genaue Kontrolle über die verwendeten Instruktionen
    - Nutzen hierfür nimmt ab: Compiler werden immer besser!
    - trotzdem kann man noch deutliche Beschleunigungen realisieren
  - Zum Zugriff auf Funktionalität, die durch höhere Programmiersprachen nicht direkt angesprochen werden kann:
    - Beispiele: MMX, 3DNow!, SSE
    - werden jedoch auch zunehmend gut von Compilern unterstützt
  - Der wichtigste: Verstehen, wie ein Computer arbeitet!

Kompilieren von Hochsprachen

- Maschineninstruktionen werden auch beim Übersetzen von Hochsprachen (C, C++, Pascal, etc.) erzeugt:
  - Besonderheiten des Betriebssystems werden berücksichtigt
  - daher abhängig von Prozessor und Betriebssystem
- Aber: Hier werden die Konstrukte der Programmiersprache vom Compiler in viele Instruktionen des Befehlssatzes übersetzt
  - der Optimierungsgrad kann beim Übersetzen durch Parameter gesteuert werden
  - ebenso kann man durch Parameter bestimmen, ob die Besonderheiten eines Prozessors (MMX/SSE) verwendet werden sollen
  - man hat keine vollständige Kontrolle darüber, welche Instruktionen des Befehlssatzes erzeugt werden
- Bei der Assembler-Programmierung:
  - genaue Kontrolle darüber, welche Instruktion erzeugt wird
  - sehr viel direktere und einfachere Übersetzung als beim Kompilieren

Der Assembler

- Es gibt viele verschiedene Assembler.
- Selbst für einen Prozessor unterscheidet sich die Syntax der symbolischen Befehle von Assembler zu Assembler:
  - meist ist die Syntax jedoch zumindest ähnlich
- Auch in Bezug auf die Unterstützung durch einen Präprozessor (z.B. für Makros) gibt es große Unterschiede.
- Im Rahmen der Vorlesung verwenden wir den NASM (Netwide Assembler):
  - Open Source
  - für DOS/Windows und Linux
  - `http://www.nasm.us`

Einbetten in C

- Meist werden Programme nicht komplett in Assembler geschrieben:
  - sehr aufwändig
  - sehr fehleranfällig
- Statt dessen:
  - einzelne Funktionen werden in Assembler geschrieben
  - die Funktionen werden dann von einer höheren Programmiersprache aus aufgerufen
- Vorgehen bei C:
  - C-Programm mit Funktionsaufruf schreiben und kompilieren
  - Assembler-Programm für diese Funktion schreiben und assemblieren
  - Linken der Objektdateien

Weiteres Vorgehe

- Assembler-Programmierung in 3 Schritten:
  - jetzt: Kurzeinführung in der Vorlesung (x86/IA-32-Assembler)
  - im Selbststudium und für die praktischen Übungen:
    - Paul A Carter, „PC Assembly Language“, 2003
    - verfügbar unter `http://pacman128.github.io/pcasm/`
    - einige Hilfsmittel (Makros) aus dem Buch werden verwendet
- Verwendete Tools:
  - NASM (Assemblieren) (`http://www.nasm.us`)
  - Handbuch zu NASM (`http://www.nasm.us/docs.php`)
  - gcc (Compilieren und Linken)
    - für DOS/Windows: DJGPP (`http://www.delorie.com/djgpp/`)

Weiteres Vorgehen

- Es gibt kleinere Unterschiede zwischen der Benennung von Funktionen unter Linux und DOS/Windows:
  - unter DOS/Windows muss den Assembler-Funktionen ein _ vorausgehen (z.B. `_asm_main`)
  - unter Linux ist das nicht der Fall
- Bei der Assemblierung muss unter DOS/Windows ein anderer Parameter für das Format des Objektcodes angegeben werden als unter Linux.
- Die Abgabe der Übungsaufgaben erfolgt so, dass das Resultat unter Linux lauffähig sein muss!
  - unter DOS/Windows kann entwickelt werden, aber Sie sind dafür verantwortlich, dass der abgegebene Code unter Linux/NASM korrekt übersetzt und dann ausgeführt werden kann!

Erstes Assembler-Beispiel

```
#include "cdecl.h"

int PRE_CDECL asm_main( void ) POST_CDECL;

int main()
{
  int ret_status;
  ret_status = asm_main();
  return ret_status;
}
```

Aus: PC Assembly Language, *driver.c*

- Das C-Programm für das erste Assembler-Beispiel.
- *cdecl.h* enthält die Definitionen von `PRE_CDECL` und `POST_CDECL`
  - regelt, wie C Assembler-Funktionen aufruft
- alles andere: wie immer
  - inkl. Deklaration einer Funktion `int asm_main(void);`

Erstes Assembler-Beispiel

```
%include "asm_io.inc"

segment .text
global  asm_main

asm_main:
        enter   0,0             ; Funktionseintritt
        pusha                   ; Registersicherung

        mov     ebx, 1          ; lade eine 1 in das Register ebx
        dump_regs 1             ; gib Register aus

        popa                    ; Registerwiederherstellung
        mov     eax, 0          ; Rückgabeparameter setzen
        leave                   ; nach C zurückkehren
        ret
```

- mit `%` beginnende Zeilen: Anweisungen an den NASM-Präprozessor (analog zu `#` in C)
- *asm_io.inc* enthält Hilfsmakros, z.B. zur Ausgabe der Registerinhalte (`dump_regs 1`).
- unter DOS/Windows: `_asm_main` statt `asm_main`

Assemblieren, Kompilieren, Linken

- Assemblieren:
  - `nasm -f elf program1.asm` (Linux)
  - `nasm -f coff program1.asm` (DOS)
  - erzeugt eine Objektdatei: *program1.o*
- Kompilieren:
  - `gcc -m32 -c driver.c`
- Linken:
  - `gcc -m32 -o first driver.o program1.o asm_io.o`
  - *asm_io* beinhaltet den Objektcode für die Ausgaberoutinen, die in *asm_io.inc* deklariert wurden.
  - Ergebnis: *first.exe* (dos) oder *first* (linux)
  - auf 64-Bit-Systemen bei gcc ist der Parameter `-m32` erforderlich (wenn der Assembler 32-Bit-Programmcode generiert)

Ausgabe

```
Register Dump # 1
EAX = 00000000 EBX = 00000001 ECX = 00000001 EDX = 401570C0
ESI = 40014020 EDI = BFFFDCB4 EBP = BFFFDC58 ESP = BFFFDC38
EIP = 0804841A FLAGS = 0286       SF       PF
```

In das Register EBX wurde eine 1 geladen.

Vertiefungsübung

Was?	s/RegExp?/ez/
Wann?	Donnerstag, 08:30 Uhr
Wo?	2522.U1.55