Nachträge

MIPS in Chipkarten

- MIPS: **M**icroprocessor without **I**nterlocked **P**ipeline **S**tages
- seit 1981 (32-bittig), seit 1992 auch 64-bittig
- 32-Bit-MIPS in Smartcards: seit 2002, www.mips.com/media/files/white-papers/Smart_Card.pdf

Nachträge

Equivalence Checking (Zitate: Wikipedia)

- allgemein: „design specifications [are] functionally equivalent if, clock by clock, they produce exactly the same sequence of output signals for any valid sequence of input signals.“
- „Microprocessor designers use equivalence checking to compare the functions specified for the instruction set architecture (ISA) with a **register transfer level** (RTL) implementation, ensuring that any program executed on both models will cause an identical update of the main memory content.“
- In integrated circuit design, **register transfer level** is a level of abstraction used in describing the operation of a synchronous digital circuit. In RTL design, a circuit's behavior is defined in terms of the flow of signals (or transfer of data) between hardware registers, and the logical operations performed on those signals.

Nachträge

Vektorbefehle in Intel-CPUs

- SSE (Streaming SIMD Extensions)
- arbeitet mit 128 Bit breiten SSE-Registern
- Vektoraddition
  (z1,z2,z3,z4) = (x1,x2,x3,x4)+(y1,y2,y3,y4)
  statt Schleife
  z1=x1+y1; z2=x2+y2; z3=x3+y3; z4=x4+y4
- „Breite“ des Vektors aber kleiner als bei klassischen Vektorrechnern

Anmeldung im ZPA-System

Prüfen sie bitte wann der von Ihnen gewünschte Kurs frei gegeben wird, unnötige Reload belasten nur das System.

Zeitplan des ZPA Systems im Wintersemester 2010/11

Hinweis: Wenn Sie meinen, Sie sehen hier Kurse nicht, für die Sie sich anmelden wollen, klicken Sie bitte hier.
Wählen Sie das Praktikum aus, für das Sie sich anmelden wollen.
Ein Klick auf das ▲▼ Symbol sortiert die Tabelle nach der Spalte abwechselnd auf- bzw absteigend.

Anfragezeit:00:00

Kursleiter ▼▲	Kursbezeichnung ▼▲	Kursart: alle Arten ->	Status	freie Plätze
Eßer H.	Rechnerarchitektur 1. Teilgruppe IF3A	Uebung	zur Anmeld	3 i
Eßer H.	Rechnerarchitektur 1. Teilgruppe IF3B	Uebung	auflisten	0 i
Eßer H.	Rechnerarchitektur 2. Teilgruppe IF3A/IF3B/sonstige	Uebung	zur Anmeld	11 i

2. ISA: Instruction Set Architecture (Befehlssatzarchitektur)

Befehlssatzarchitektur (Instruction Set Architecture, ISA)

- Beschreibung umfasst:
  - Maschinenbefehlssatz
  - Registerstruktur
  - Adressierungsarten
  - Interruptbehandlung
- Klassisch: Unterscheidung in Ein-, Zwei- und Drei-Adress*maschinen*
- Heute üblicher: unterscheiden nach Ein-, Zwei- und Drei-Adress*befehlen*

Register und Registersätze (1/2)

- Register: die schnellsten speichernden Elemente eines Prozessors
- meist allgemein verwendbare Register (General Purpose Registers, GPR) und Spezialregister.
- Gesamtheit aus Befehlssatz und verfügbaren Registern heißt **Programmiermodell**

Register und Registersätze (2/2)

Typische Spezialregister

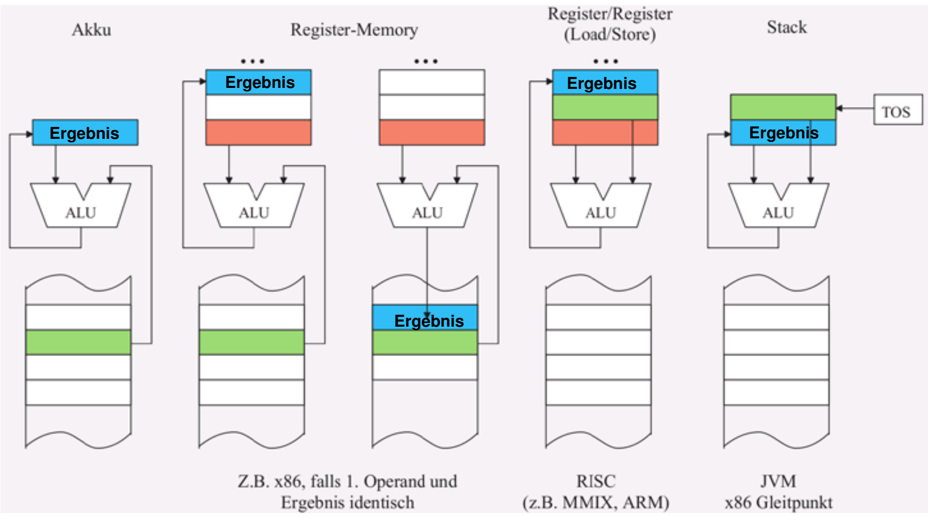
- Befehlszähler
- Stackpointer
- Statusregister (kann z. B. anzeigen, ob bei der letzten Operation ein Überlauf aufgetreten ist, oder ob das Ergebnis negativ war etc.)
- Indexregister (für Adressrechnungen)

Operanden und Ergebnis (1/6)

- ISAs unterscheiden nach Zugriff auf Register und Speicherinhalte
- nur Transferbefehle Register<->Speicher oder auch gemeinsame Operationen
- Beispielaufgabe: Werte aus zwei Speicher-zellen addieren und in dritter Zelle speichern

  C := A + B;

Operanden und Ergebnis (2/6)

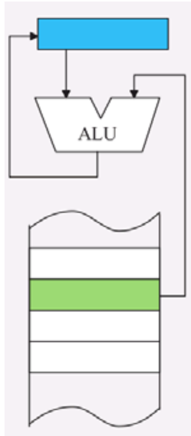


Operanden und Ergebnis (3/6)

C := A + B

mit **Akku** (ein einziges Register):

```
LOAD  A
ADD   B
STORE C
```



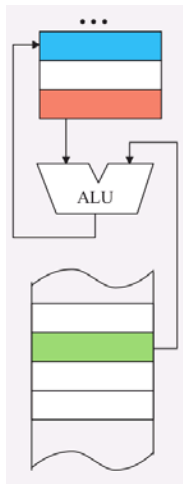
nutzt implizit den Akku:
- LOAD A = lade A in Akku
- ADD B = addiere B zum Wert in Akku (Ergebnis im Akku)
- STORE C = schreibe Akku-Inhalt nach C

Operanden und Ergebnis (4/6)

C := A + B

mit **Register-Memory**:

```
LOAD  R1,A
ADD   R3,R1,B
STORE R3,C
```



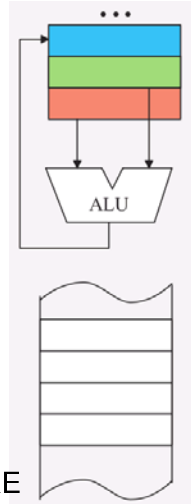
- nennt alle Register explizit (R1, R3)
- Argumente können Register oder Speicherstellen sein (z. B. R1, B)

Operanden und Ergebnis (5/6)

C := A + B

mit **Register-Register (Load/Store)**:

```
LOAD  R1,A
LOAD  R2,B
ADD   R3,R1,R2
STORE R3,C
```



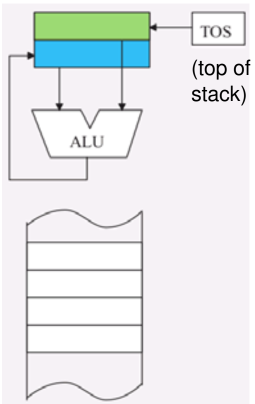
- nennt alle Register explizit (R1, R3)
- Argumente für Operationen können nur Register sein (R1, R2)
- Zugriff auf Speicher nur durch LOAD, STORE

Operanden und Ergebnis (6/6)

C := A + B

mit **Stack**:

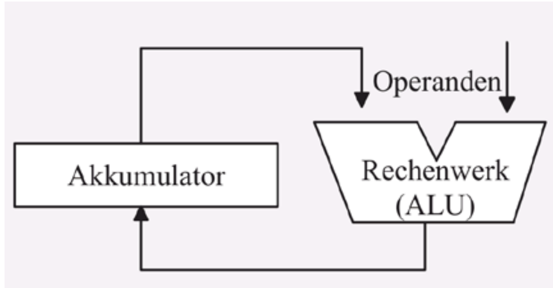
```
PUSH A
PUSH B
ADD
POP  C
```



- keine Register
- Argumente auf Stack, dann Operation
- Operation implizit (oberste Werte auf Stack)
- Zugriff auf Speicher nur durch PUSH, POP

Ein-Adress-Maschinen (1/4)

- Maschinen, deren Befehle nur einen Operanden haben, heißen **Ein-Adress-Maschinen.**
- Akku: spezielles Register, impliziter linker Operand und Zielregister für das Ergebnis („Akkumulatormaschinen“)
- rechter Operand aus Speicher



Ein-Adress-Maschinen (2/4)

- **Vorteile**
  - Ausführung der einzelnen Befehle wegen der einfachen Hardware sehr schnell
  - geringer Speicherbedarf für einen Befehl
  - Fast jeder Befehl hat einen Operanden (außer z. B. NOP, INC, DEC) $\rightarrow$ einheitliche Befehlslänge, einfaches Aktualisieren des Program Counters
- **Nachteile**
  - Programmierung in diesem Format erfordert Übung – vor allem das Auswerten von mathematischen Formeln
  - Programme wegen des häufig erforderlichen Zwischenspeicherns von Hilfsgrößen etwas „länglich“

Ein-Adress-Maschinen (3/4)

- Berechnung der Formel $y = \frac{x_1 + x_2 x_3}{x_1 - x_2}$

  mit MMIX (als Ein-Adress-Maschine; $1=Akku)

```
x1     OCTA  3
x2     OCTA  7
x3     OCTA  11
Accu   IS    $1
Main   LDO   Accu,x2
       LDO   $2,x3
       MUL   Accu,Accu,$2
       LDO   $2,x1
       ADD   Accu,Accu,$2
       STO   Accu,x3
       LDO   Accu,x1
       LDO   $2,x2
       SUB   Accu,Accu,$2
       STO   Accu,x1
       LDO   Accu,x3
       LDO   $2,x1
       DIV   Accu,Accu,$2
       TRAP  0,Halt,0
```


Ein-Adress-Maschinen (4/4)

- Gleiches Programm in „echter“ Akku-Syntax:

```
x1   OCTA  3
x2   OCTA  7
x3   OCTA  11
     LOAD  x2        Akku enthält x2
     MUL   x3        Akku enthält x2*x3
     ADD   x1        Akku enthält x2*x3+x1
     STOR  x3        x3 ← x2*x3+x1
     LOAD  x1        Akku enthält x1
     SUB   x2        Akku enthält x1-x2
     STOR  x1        x1 ← x1-x2
     LOAD  x3        Akku enthält orig. x2*x3+x1
     DIV   x1        Akku enthält Ergebnis
     TRAP  Halt...
```


Zwei-Adress-Befehle (1/3)

- Befehle mit zwei Operanden (Register oder Speicheradressen)
- linker Operand ist implizit Ziel:

  `CMD x1,x2` bedeutet x1 ← x1 ⊗ x2
- z. B. Addition

  `ADD $1,$2` bedeutet $1 ← $1 + $2
- in MMIX-Syntax:

  `ADD $1,$1,$2`

Zwei-Adress-Befehle (2/3)

- Wieder mit MMIX-Befehlen

```
x1     OCTA  3
x2     OCTA  7
x3     OCTA  11
Main   LDO   $1,x1        Startwerte in $1,
       LDO   $2,x2        $2,
       LDO   $3,x3        $3.
       MUL   $3,$3,$2     Produkt x2*x3 in $3
       ADD   $3,$3,$1     x1 + x2*x3 in $3
       SUB   $1,$1,$2     x1-x2 in $1
       DIV   $3,$3,$1     Bruch in $3, fertig
```

Zwei-Adress-Befehle (3/3)

- und mit „echter“ Zwei-Adress-Syntax

```
x1    OCTA  3
x2    OCTA  7
x3    OCTA  11
Main  LDO   $1,x1
      LDO   $2,x2
      LDO   $3,x3
      MUL   $3,$2
      ADD   $3,$1
      SUB   $1,$2
      DIV   $3,$1
```

```
x1    OCTA  3
x2    OCTA  7
x3    OCTA  11
Main  LDO   $1,x1
      LDO   $2,x2
      LDO   $3,x3
      MUL   $3,$3,$2
      ADD   $3,$3,$1
      SUB   $1,$1,$2
      DIV   $3,$3,$1
```


Drei-Adress-Befehle (1/2)

- Format von MMIX-Befehlen
- ein Ziel (erster Operand)
- zwei Quellen (2. und 3. Operand)

Beispiel

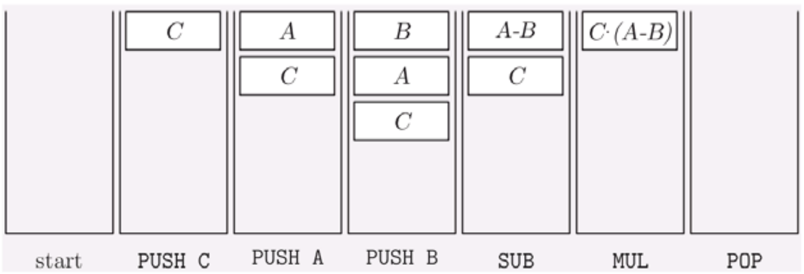
```
x1    OCTA  3
x2    OCTA  7
x3    OCTA  11
Main  LDO   $1,x1
      LDO   $2,x2
      LDO   $3,x3
      SUB   $6,$1,$2
      MUL   $7,$2,$3
      ADD   $7,$7,$1
      DIV   $7,$7,$6
```


Drei-Adress-Befehle (2/2)

- Vorteile
  - bequeme Programmierung
  - kurze Programme (Anzahl der Befehle)
- Nachteil
  - Eine Speicheradresse ist 64 Bit lang
  - drei Operanden (zunächst Register oder Speicheradresse): enorm große Befehlsbreite
    → darum keine Speicheradressen als Operanden

Null-Adress- (Stack-) Maschine

- Operanden der ALU immer oben auf Stack
- keine normalen Register
- Anwendung: Java Byte Code
- Beispiel: Berechne (a-b)*c



Stack: Java-Beispiel (1/2)

```
public class euklid {
   public static void main ( String [ ] args )
   {
      int a=1228 , b =96;
      for ( int r = a%b ; r ! = 0 ; r = a%b )
      {
          a = b;
          b = r;
      }
      System.out.println ( "GGT: " + a ) ;
   }
 }
```

erzeugt folgenden Java-Byte-Code:

Stack: Java-Beispiel (2/2)

```
0: sipush 1228         15: iload_2
3: istore_1            16: istore_1
4: bipush 96           17: iload_3
6: istore_2            18: istore_2
7: iload_1             19: iload_1
8: iload_2             20: iload_2
9: irem                21: irem
10: istore_3           22: istore_3
11: iload_3            23: goto 11
12: ifeq 26            26: getstatic #2
```

irem: berechnet a%b, Rest bei Division
iload_n: push; istore_n: pop

Länge von Befehlen

- flexibel
  - Befehlslänge hängt vom Befehl ab
  - Auslesen komplizierter (erstes Byte entscheidet über Länge), keine Ausrichtung an Wortgrenzen
  - CISC
- fest
  - Alle Befehle gleich lang
  - Leichtes Auslesen
  - Einschränkung: Operanden nur in Registern (nicht genug Platz für Adressangaben)
  - RISC

Platzbedarf

c := a+b; d := a–b; e := c*d (gesucht: nur e)

Zwei Varianten

- Register-Memory

```
ADD c,a,b        SUB d,a,b        MUL e,c,d
1    4 4 4       1    4 4 4       1    4 4 4
```

S=39

- Register-Register (Load/Store)

```
LD R1,a      LD R2,b      ADD R3,R1,R2      SUB R4,R1,R2
1     4      1     4      2                 2

MUL R5,R3,R4      STO R5,e
2                 1     4
```

S=21

Vorschau 21.10.2010

- mehr zu Adressierungsarten
- Befehlsarten
  - Gleitkommabefehle
  - Bedingte Befehle
  - SIMD-Befehle
  - Nicht-unterbrechbare Befehle
  - Spezielle Aspekte der Speicherzugriffsbefehle
- Leistungsbewertung und Leistungsmessung