

2.4 VLIW und EPIC-Prozessoren

Peter Marwedel
Informatik 12
TU Dortmund

Gründe für die Einführung von Parallelität (1)

- **Steigerung der Rechenleistung stößt an Komplexitätsschranken:**

Zunehmend schwieriger, die Rechenleistung bei sequentieller Ausführung zu steigern

- Hoher Anteil an *Pipelining*-Fehlern bei Silizium-*Bugs*
- **Superskalare Prozessoren (d.h. Prozessoren, die >1 Befehl pro Takt starten)** sind schwierig zu realisieren:

„the only ones in favor of superscalar machines are those who haven't built one yet“

[late Bob Rau (hp Labs)]



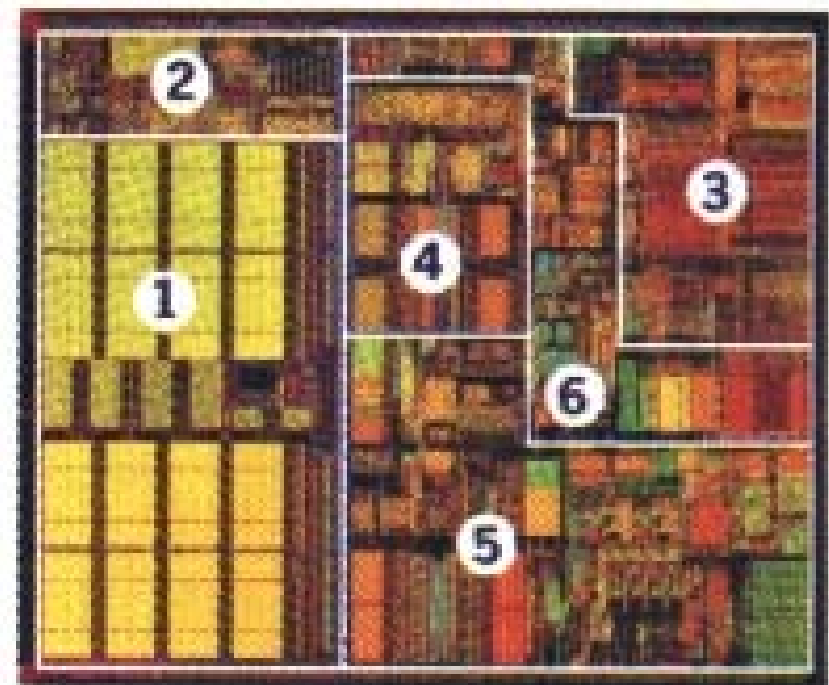
[www.trimaran.org]

Verwaltungsaufwand bei superskalaren Prozessoren

Heutige RISC-Architekturen verwenden einen Großteil der Hardware darauf, mögliche Parallelarbeit zu erkennen.

1. Cache
2. Kommunikation mit Umgebung
3. Rechenwerk
4. Cache
5. **Umsortieren (dyn. Scheduling) der Befehle**
6. **Buchführung über Speicherort von Daten**

[Bode]



Pentium III; © Intel; Bode TUM

Gründe für die Einführung von Parallelität (2)

■ Rechnen mit niedrigen Taktraten energetisch effizienter:

Basisgleichungen

Leistung:

$$P \sim V_{DD}^2,$$

Maximale Taktfrequenz:

$$f \sim V_{DD},$$

Energiebedarf für ein Programm:

$$E = P \times t, \text{ mit: } t = \text{Laufzeit (fest)}$$

Zeitbedarf für ein Programm:

$$t \sim 1/f$$

Änderungen durch Parallelverarbeitung, mit α Operationen pro Takt:

Taktfrequenz reduziert auf:

$$f' = f / \alpha,$$

Spannung kann reduziert werden auf:

$$V_{DD}' = V_{DD} / \alpha,$$

Leistung für Parallelausführung:

$$P^\circ = P / \alpha^2 \text{ pro Operation,}$$

Leistung für α Operationen pro Takt:

$$P' = \alpha \times P^\circ = P / \alpha,$$

Zeit zur Ausführung des Programms:

$$t' = t,$$

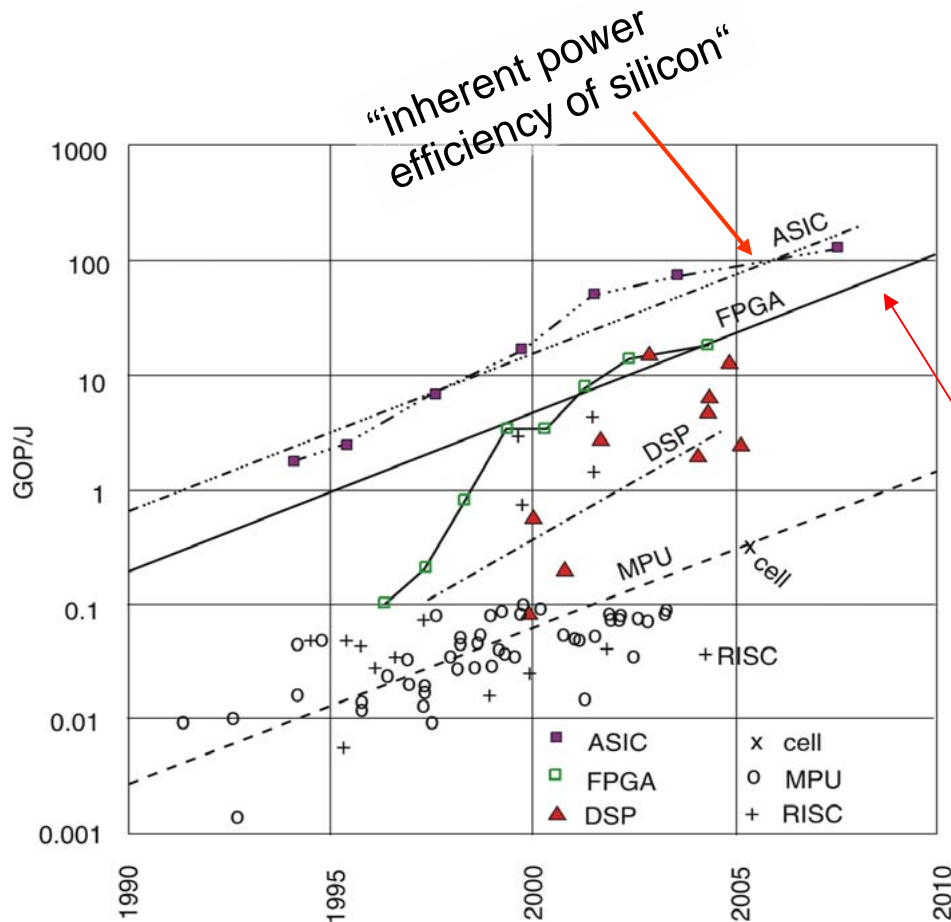
Energie zur Ausführung des Programms: $E' = P' \times t = E / \alpha$

☞ **VLIW-Prozessoren sind energieeffizient.**

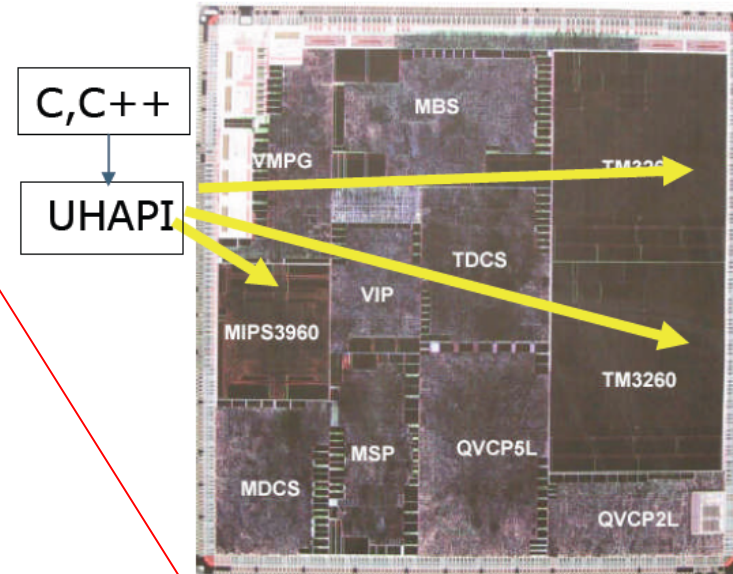
✳ Dasselbe Argument gilt auch für Multicore-Prozessoren

**Zeigt Tendenz,
aber diverse
grobe
Näherungen!**

Existenz von VLIW-Prozessoren mit einer Energieeffizienz nahe der von ASICs



Nexperia Digital Video Platform NXP

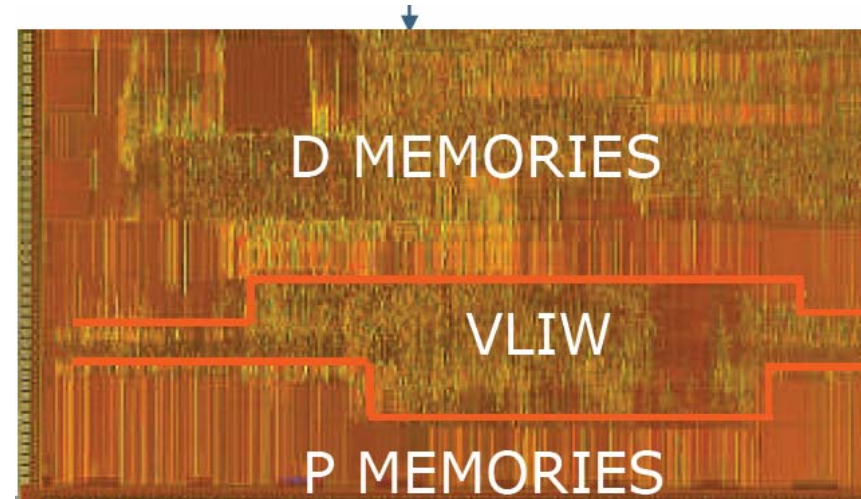
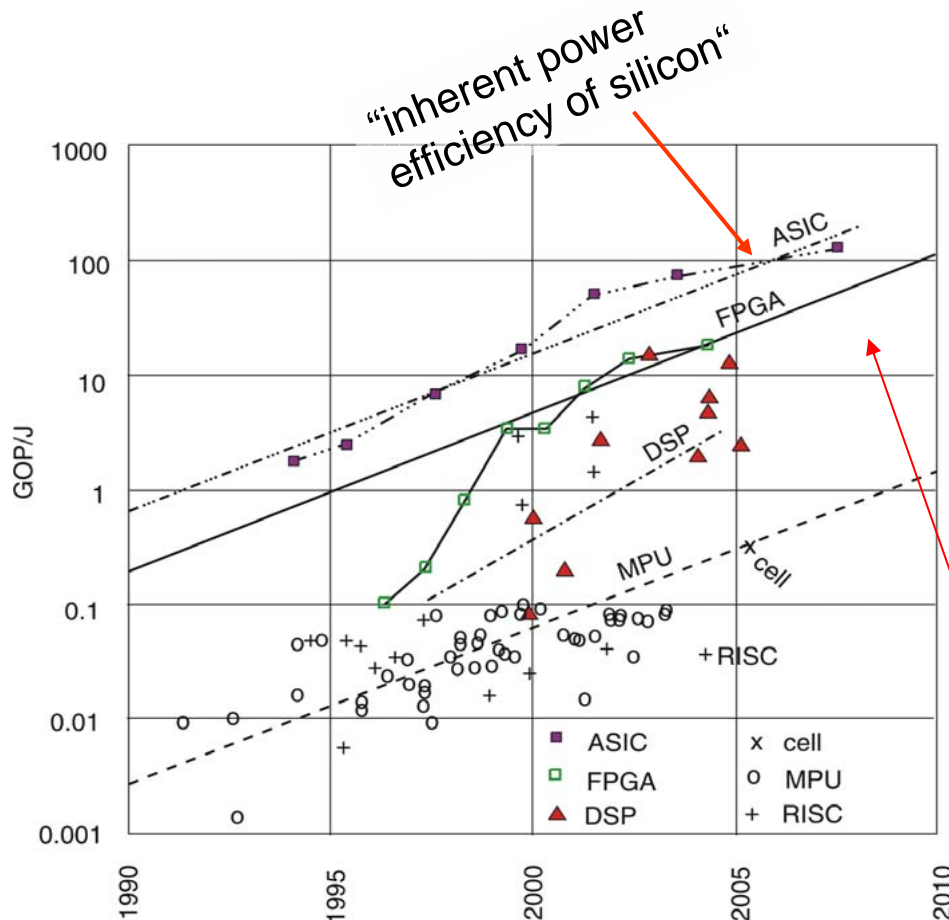


**1 MIPS, 2 Trimedia (VLIW)
60 coproc, 250 RAM's
266MHz, 1.5 watt 100 Gops**

Energieeffizienz nahe
der Grenze für Silizium

© Hugo De Man: From the Heaven of Software to the Hell of Nanoscale Physics: An Industry in Transition, *Keynote Slides*, ACACES, 2007

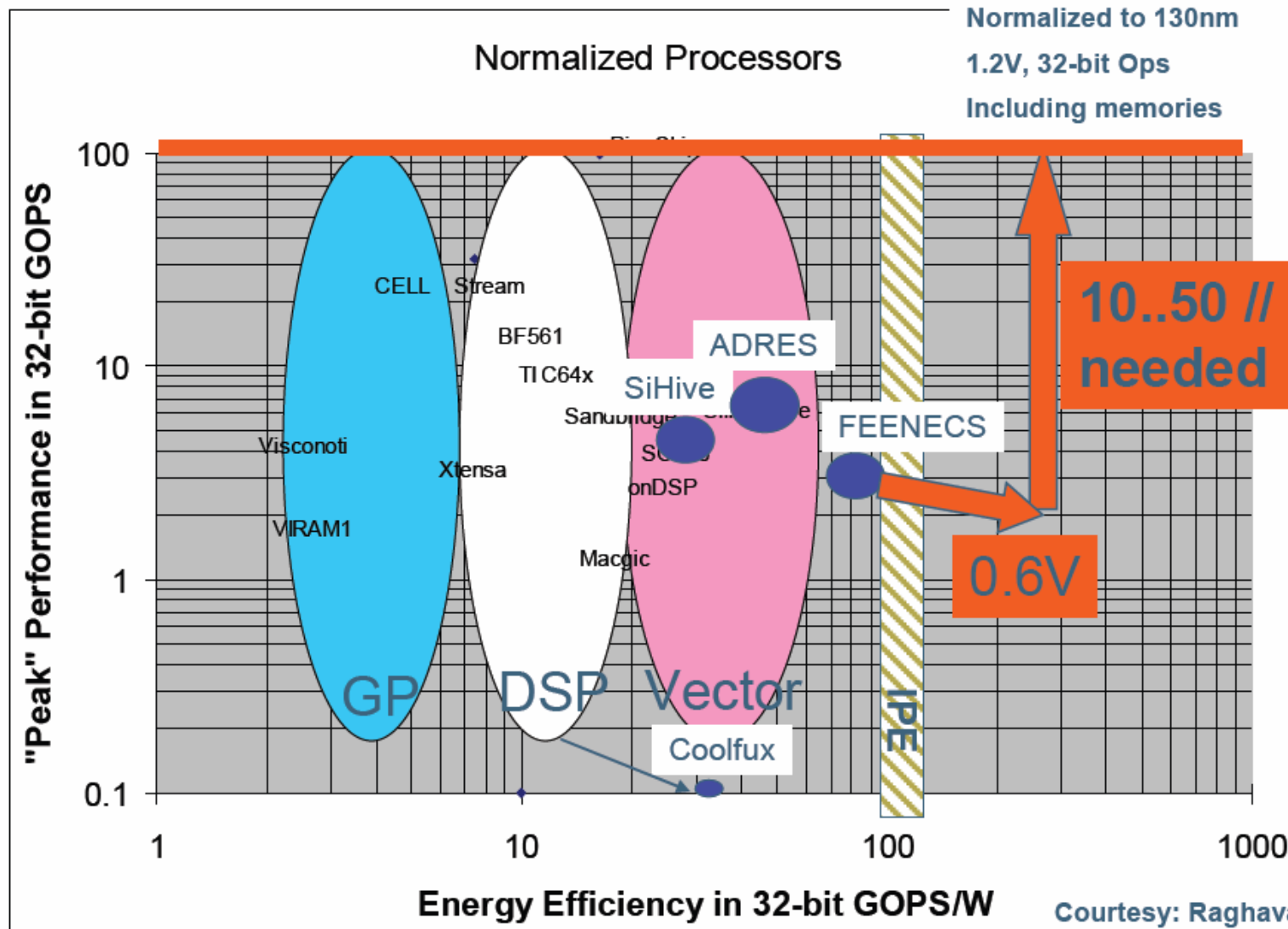
Existenz von VLIW-Prozessoren mit einer Energieeffizienz nahe der von ASICs (2)



© Silicon Hive
 41 Issue VLIW for SDR
 130 nm, 1,2 V, 6,5mm², 16 bit
 30 operations/cycle (OFDM)
 150 MHz, 190 mW (incl. SRAMs)
 24 GOPs/W, ~1/5 IPE

© Hugo De Man: From the Heaven of Software to the Hell of Nanoscale Physics: An Industry in Transition, *Keynote Slides*, ACACES, 2007

"Estimated" State of the Art Processors



Was soll parallel ausgeführt werden und wer entscheidet?

Prinzipielle „Entscheidungsträger“:

- Hardware (d.h. Prozessor)
während der (Maschinen-)Programmausführung [Kap. 3].
Kostet Hardwareaufwand und Energie.
- ➔ ■ Software (d.h. Compiler)
☞ Wie werden parallele Berechnungen repräsentiert?
Erfordert leistungsstarke Compiler.

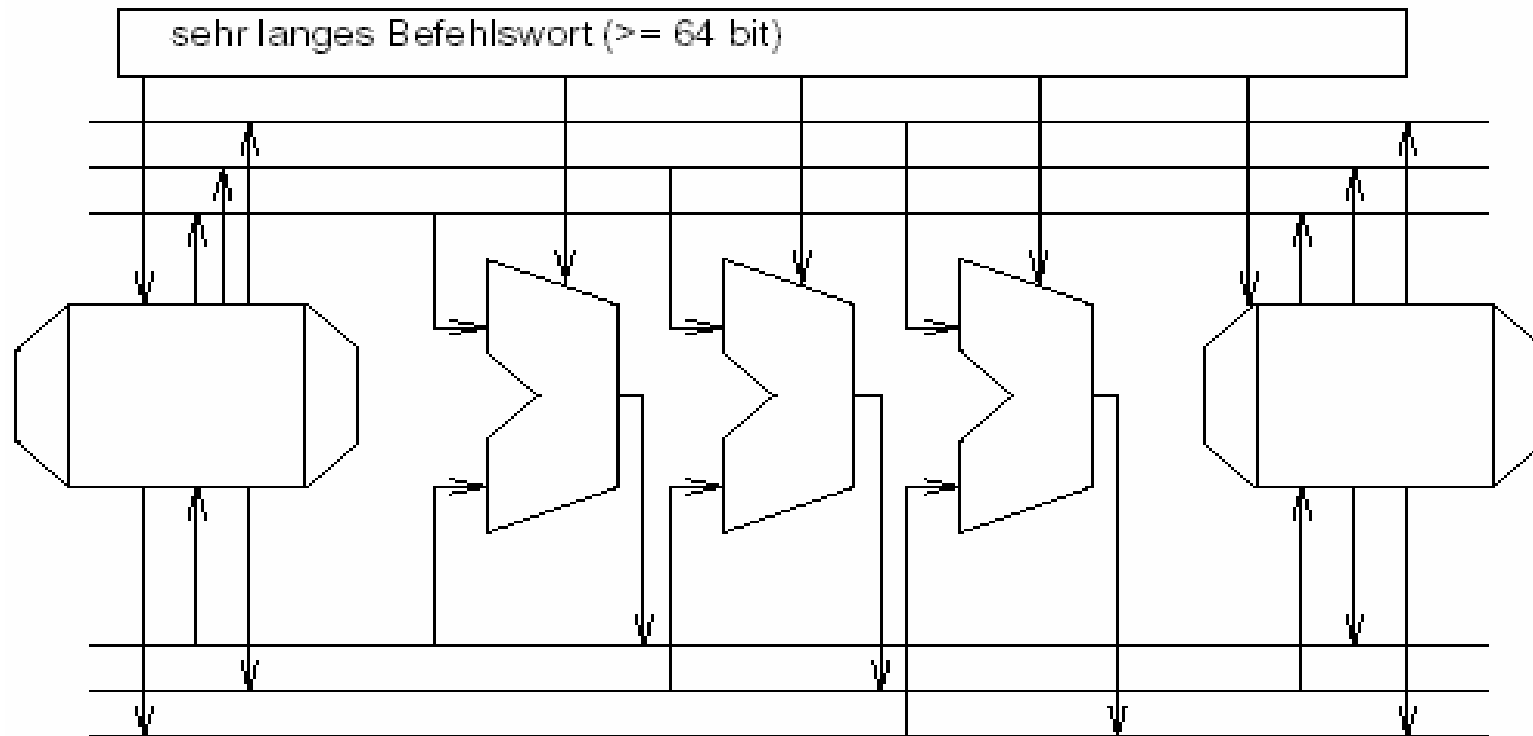
VLIW- und EPIC-Befehlssätze

Befehlssätze, die eine parallele Ausführung unterstützen:

- 1. VLIW = *very long instruction word machine***
Bildung eines Befehlspaketes konstanter Länge;
alle Befehle im Paket sind parallel auszuführen;
Compiler bestimmt, was parallel auszuführen ist.
☞ Sehr lange Befehlswörter (64, 128 Bit oder mehr)
- 2. EPIC = *Explicitly parallel instruction computing***
Mögliche parallele Ausführungen werden ebenfalls zur
Compilezeit erkannt und im Code kodiert;
Parallele Ausführung kann aber flexibler ausgedrückt
werden, nicht notwendigerweise durch Bildung eines
Befehlspaketes.

Grundidee von VLIW

Prinzip: Operationen im Befehlspaket sprechen mehrere Recheneinheiten bzw. Registerblöcke an



VLIW: Parallelisierbarkeit?

Woher kommt das Potential für Parallelisierungen?

Programme führen Instruktionen prinzipiell entlang eines Kontrollflusses aus

- Operationen manipulieren Daten sequentiell
- Arbeiten also i.d.R. auf Ergebnissen *vorheriger* Operationen

Es existieren daher häufig Datenabhängigkeiten zwischen aufeinander folgenden Operationen

- Solche Operationen können nicht parallel ausgeführt werden!
- Wo also Parallelisierungspotential finden?

In der Regel nur begrenzte Anzahl funktioneller Einheiten
☞ nur bestimmte Operationstypen parallel möglich

VLIW: Parallelisierbarkeit (2)?

Beispiel: Betrachten (auf Hochsprachenebene)

$$x = (a + b) * (a - c)$$

Realisierung $x_0 = (a + b)$ (sequentiell)

$$x_1 = (a - c)$$

$$x = x_0 + x_1$$

☞ Verschränkung der Berechnung teilweise möglich

Realisierung

[ALU Nr. 1]

$$x_0 = (a + b)$$

$$x = x_0 + x_1$$

[ALU Nr. 2]

$$x_1 = (a - c)$$

VLIW: Parallelisierbarkeit (3)?

Kontrollfluss innerhalb von Programmen häufig nicht linear!

Beispiel: `if (x > y) x = c + d;`
`y = a + b;`

- Zwar keine Datenabhängigkeiten, aber nicht parallelisierbar. Gibt's da vielleicht einen „Trick“?

Beispiel: `for (i = 0; i < n; i++)`
`x[i] += a;`

- Ex. keine Datenabhängigkeiten zwischen aufeinander folgenden Ausführungen des Schleifenrumpfes
- Prinzipiell könnten Operationen parallelisiert werden

Parallelisierung von Operationen kann nicht ohne Betrachtung des Kontrollflusses erfolgen!

VLIW: Parallelisierbarkeit von Schleifen

Wie potentielle Parallelisierbarkeit von Anweisungen in Schleifenrumpfen ausnützen?

☞ “Abrollen” des Schleifenrumpfs

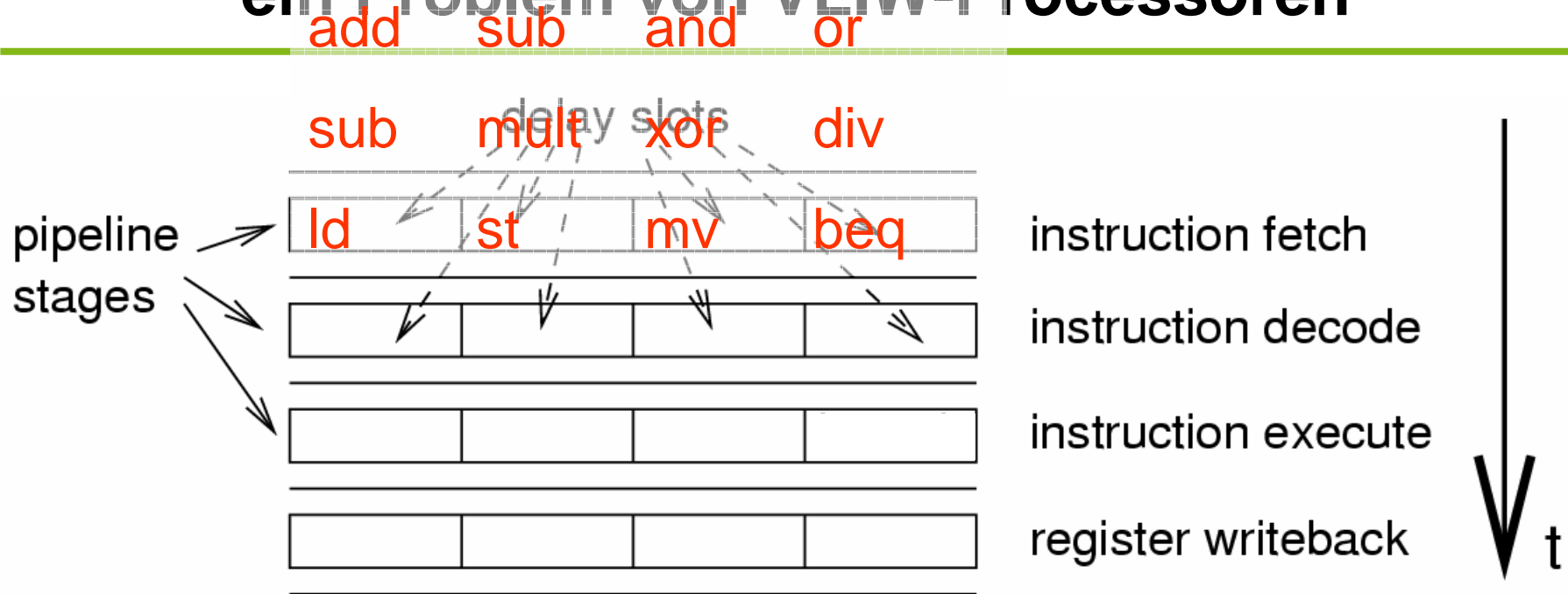
Schleife: `for (i = 0; i < n; i++)`
 `x[i] += a;`

Abgerollt (4-mal):

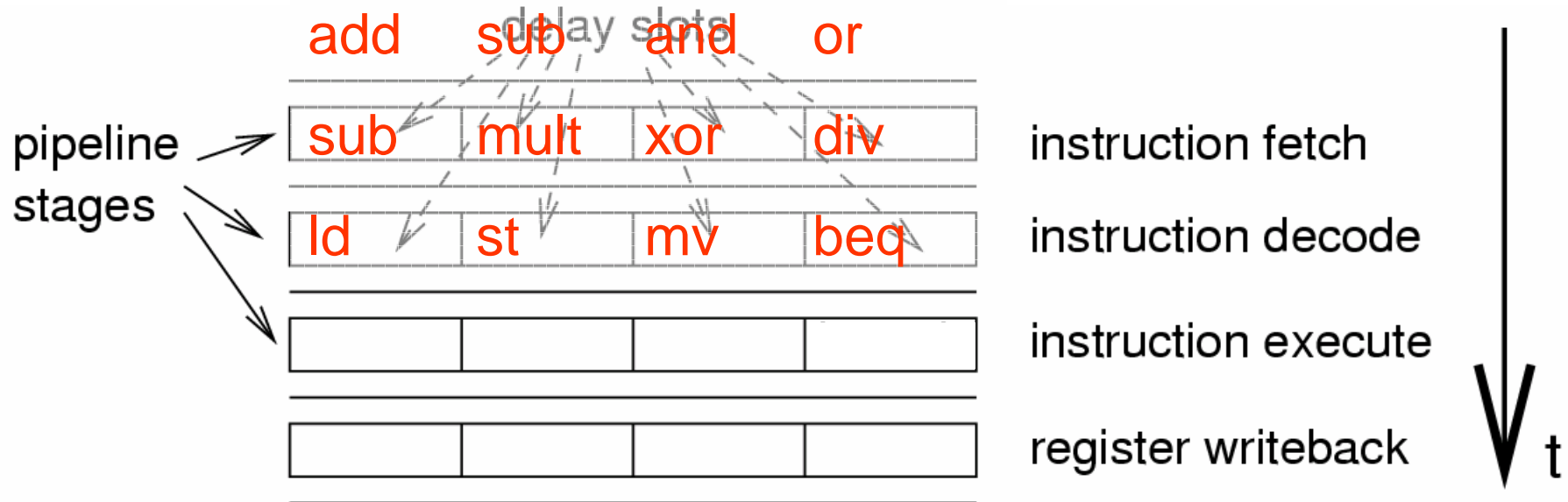
```
for (i = 0; i < n; i+=4) {  
    x[i]    += a;  
    x[i+1] += a;  
    x[i+2] += a;  
    x[i+3] += a; }  
}
```

Im Allg. (falls $n \bmod 4 \neq 0$) zusätzlicher Code erforderlich

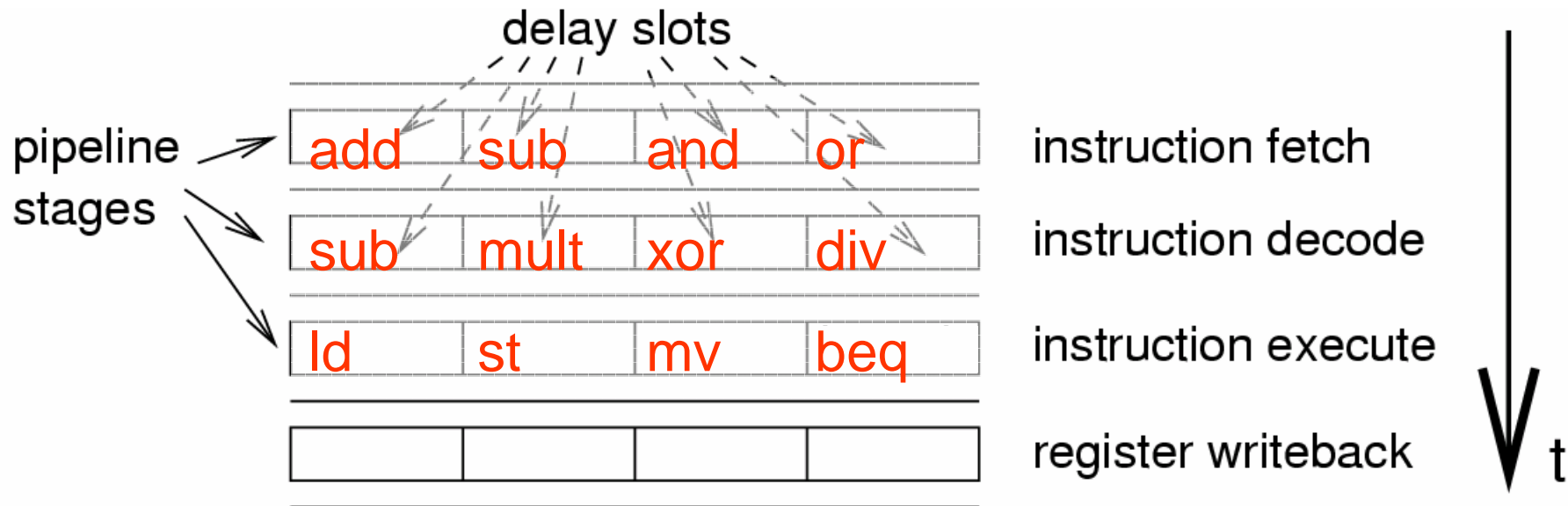
Große Zahl von *delay slots*, ein Problem von VLIW-Processoren



Große Zahl von *delay slots*, ein Problem von VLIW-Processoren



Große Zahl von *delay slots*, ein Problem von VLIW-Prozessoren



Die Ausführung von vielen Befehle wurde gestartet bevor erkannt wurde, dass ein Sprung ausgeführt werden soll.



Diese Befehle für ungültig zu erklären würde Rechenleistung verschwenden.

- ☞ Ausführung der Befehle wird als *Feature* erklärt (statt als *bug*)
- ☞ Wie soll man all die *delay slots* füllen?
- ☞ Sprünge möglichst vermeiden!

Predicated execution: Sprungfreie Realisierung von *if-Statements*

Predicated execution “[c] I“ besteht aus:

- Bedingung c
- Befehl I

c = true  I ausgeführt
c = false  NOP

Predicated execution: Sprungfreie Realisierung von *if-Statements* (TI C6x)

```
if (c)
{ a = x + y;
  b = x + z;
}
else
{ a = x - y;
  b = x - z;
}
```

Conditional branch

```
          [c] B L1
              NOP 5
              B L2
              NOP 4
              SUB x,y,a
L1:        || SUB x,z,b
              ADD x,y,a
              || ADD x,z,b
L2:
```

Max. 12 Zyklen

Predicated execution

```
          [c] ADD x,y,a
|| [c] ADD x,z,b
|| [!c] SUB x,y,a
|| [!c] SUB x,z,b
```

1 Zyklus

VLIW-Beispiel: Transmeta Crusoe

Zielanwendung: mobile Geräte (z.B. Laptops), → Strom ↓

☞ geringer Stromverbrauch

Prinzip: x86-Befehle werden vom Prozessor in VLIW-Befehle übersetzt

Befehlskodierung:

- 2 Befehlsgrößen (64 u. 128 bit) mit 2 (bis zu 4) Ops
- 5 Typen von Operationsfeldern:
 - ALU-Operationen: RISC-typische arith.-log. Operationen (Reg/Reg, 3 Operanden, 64 bit Int.-Reg.)
 - *Compute*: zusätzliche ALU-Op (2 Einheiten!), eine FP-Op. (32 bit) oder eine Multimediaop.
 - Memory: Speichertransfer (Load/Store)
 - Branch: Sprungbefehl
 - Immediate: 32 bit Konstante für andere Op. Im Befehl

VLIW-Beispiel: Transmeta Crusoe (2)

... Befehlskodierung:

- 2 Befehlsformate (128 bit Befehlsbreite)

| | | | |
|--------|---------|-----|-----------|
| Memory | Compute | ALU | Immediate |
|--------|---------|-----|-----------|

| | | | |
|--------|---------|-----|--------|
| Memory | Compute | ALU | Branch |
|--------|---------|-----|--------|

- Befehlsübersetzung:

Crusoe-Befehl (VLIW) kann mehreren x86 Op. entsprechen

- Interpretation pro x86 Instruktion
- Bei wiederholten Codeblöcken => Caching des Crusoe-Codes

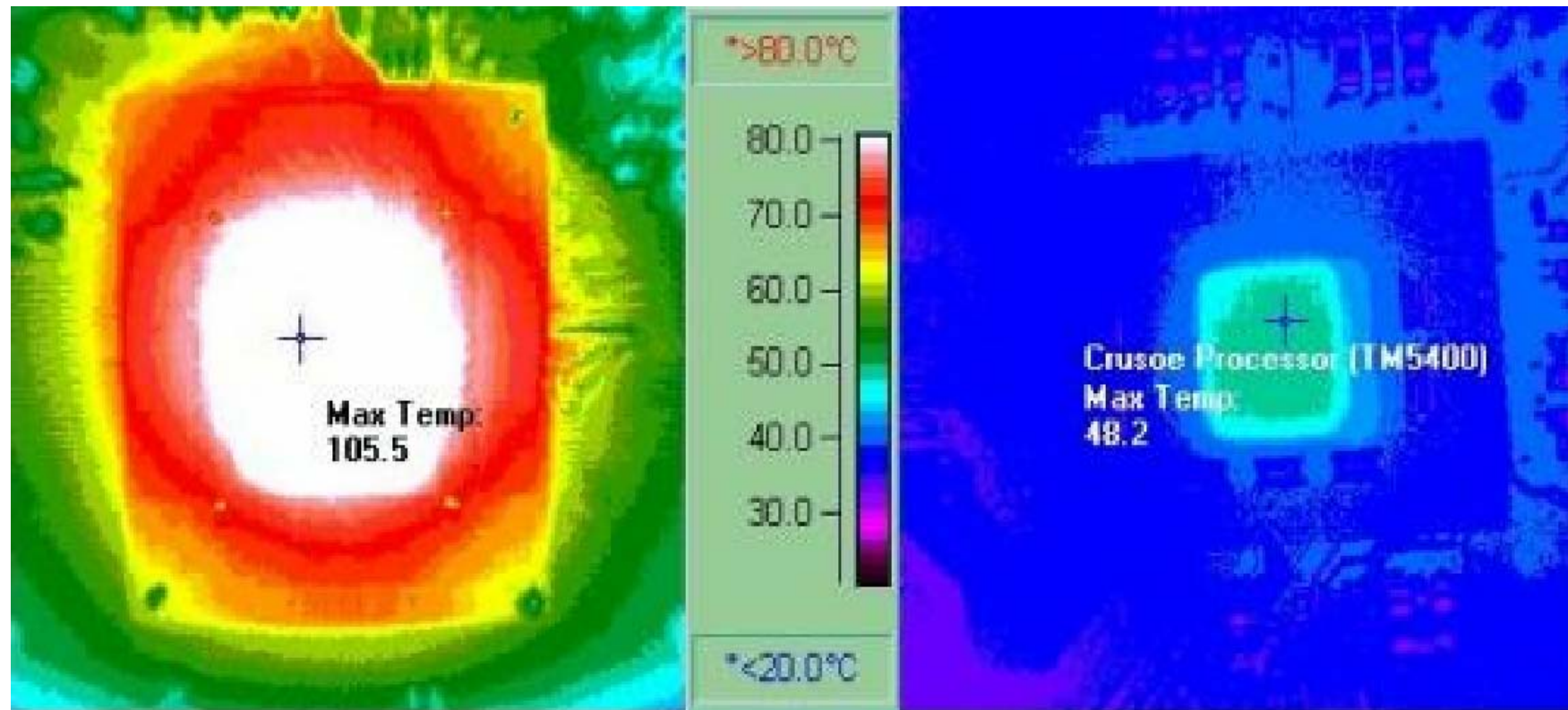
Problem: Ausnahmebehandlung

- *Crusoe verwendet "Schattenregister"*
- *Primäre Register können bei Ausnahme restauriert werden*

Temperaturvergleich

Pentium

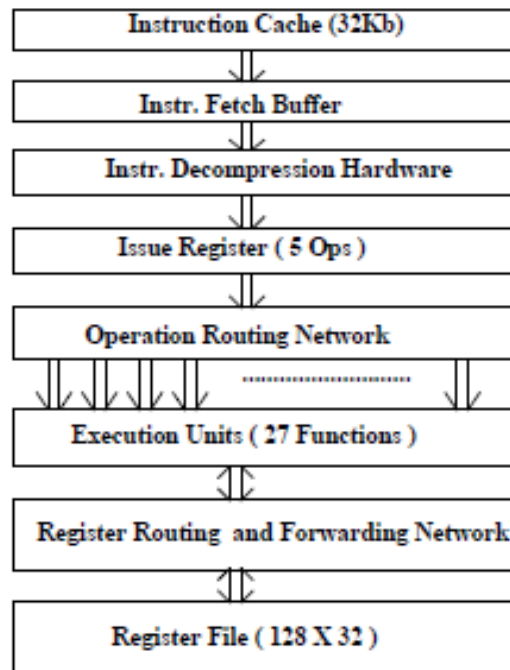
Crusoe



Bei Ausführung derselben Anwendung. Gemäß Publikation von Transmeta [www.transmeta.com] (~2003, nicht mehr verfügbar)

TriMedia

TM-1 VLIW engine



- 5 RISC operations/cycle
- 32 kByte lcache (compressed)
- dual port 16 kByte Dcache
- conditional (guarded) operations
 $R_g : R_{dest} = imul R_{src1}, R_{src2}$
- multimedia operation set

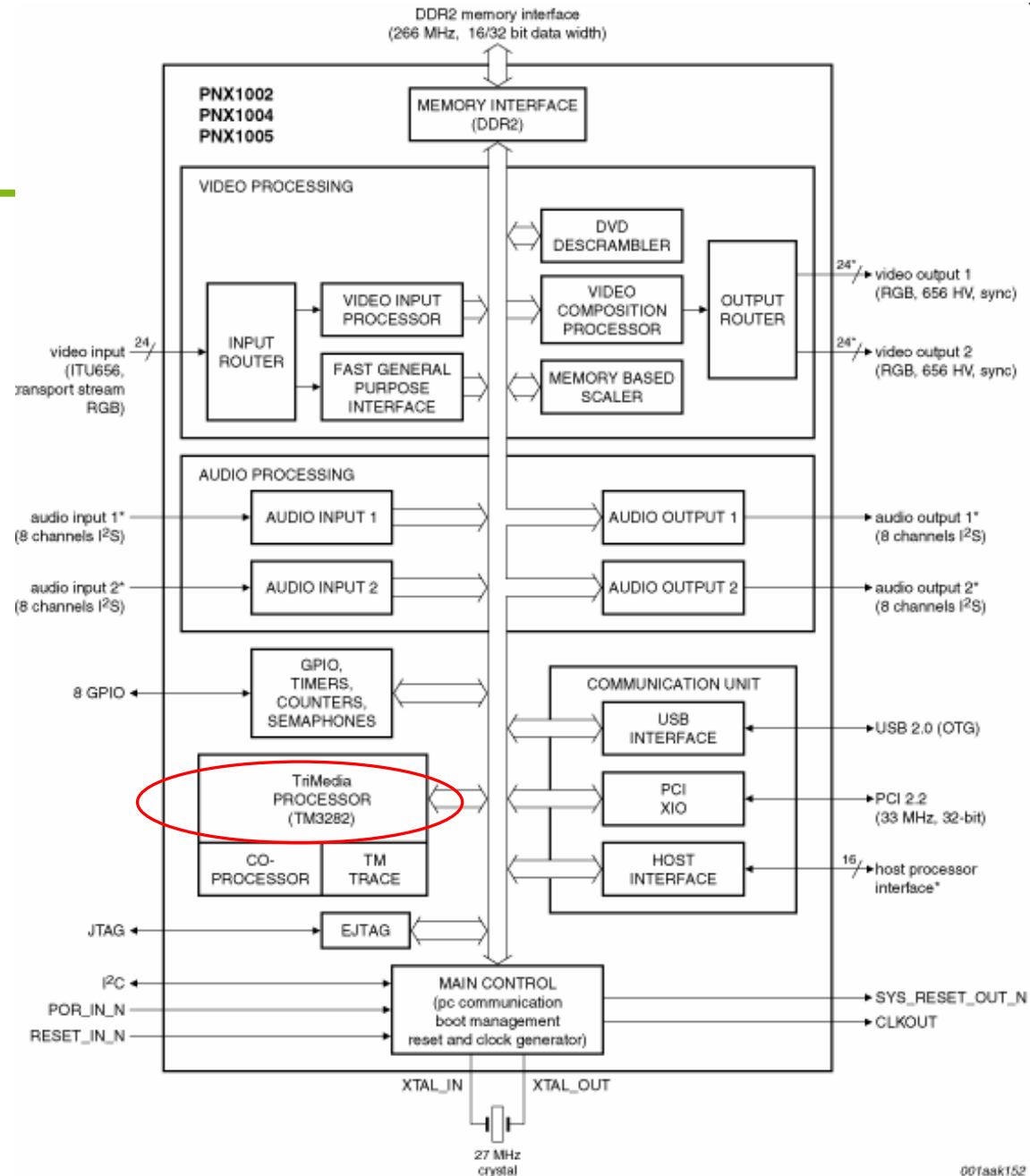
| Func Type | Qty | Latency | Recovery |
|------------|-----|---------|----------|
| Const | 5 | 1 | 1 |
| ALU | 5 | 1 | 1 |
| Memory | 2 | 3 | 1 |
| Shift | 2 | 1 | 1 |
| DspAlu | 2 | 2 | 1 |
| DspMul | 2 | 3 | 1 |
| Branch | 3 | 3 | 1 |
| Falu | 2 | 3 | 1 |
| Ifmul | 2 | 3 | 1 |
| Fcomp | 1 | 1 | 1 |
| Fdiv/Fsqrt | 1 | 17 | 16 |

2009: 45nm-Version, Bestandteil vieler Systems on a Chip (SoCs)

[http://www.hotchips.org/archives/hc8/3_Tue/HC8.S6/HC8.6.1.pdf]
 [http://ce.et.tudelft.nl/publicationfiles/1228_587_thesis_JAN_WILLEM.pdf]
 [http://en.wikipedia.org/wiki/TriMedia_%28Mediaprocessor%29]

TriMedia-Processor

Heute 5-8
Operationen
pro Zyklus
startbar



<http://www.tridentmicro.com/producttree/stb/media-processor/pnx100x/>
© Trident, 2010
(seit 4.1.2012 unter Gläubigerschutz)

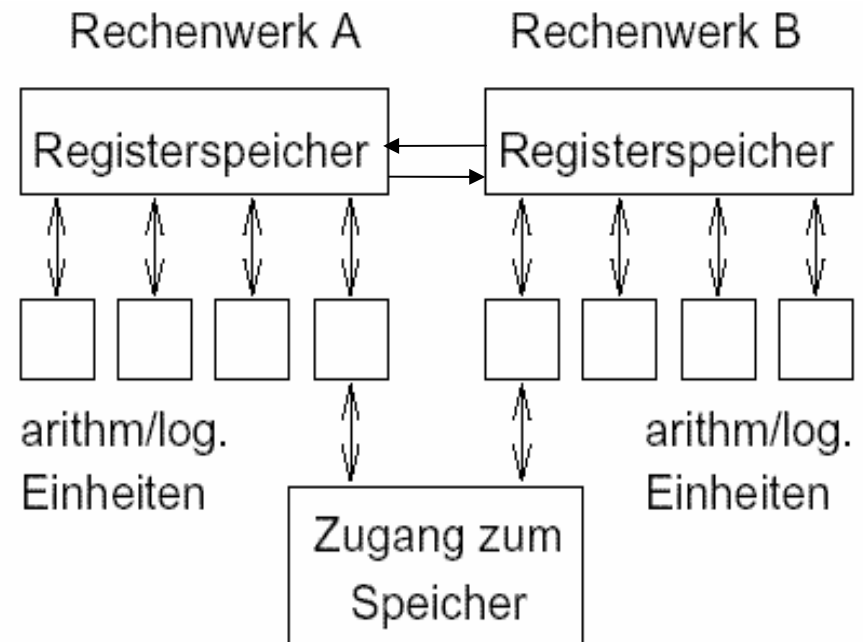
VLIW-Beispiel: TMS320C62xx Signalprozessor

Eingeführt ab 1997.

Eigenschaften:

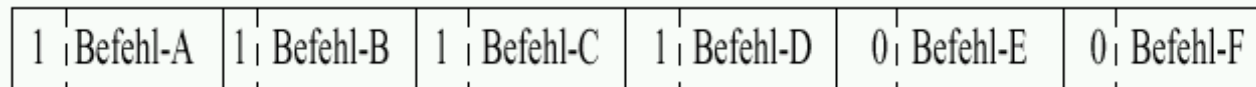
Zwei Rechenwerke mit je

- 1 16 x 16 Bit Multiplizierer
- 1 Registerspeicher mit 16 32-Bit-Registern
- 1 Adder,
- 1 Adder/Shifter,
- 1 Adder/Shifter/Normalisierer
- 1 Speicherlese- +
- 1 Speicherschreib-Pfad
- 1 Pfad zum anderen Rechenwerk



VLIW-Beispiel: TMS320C62xx Signalprozessor (2)

- Bei 200 MHz bis zu 8 32-Bit-Befehle pro Takt (5 ns) zu starten (max. 1.6 „GIPS“)
- Jeder 32-Bit-Befehl spricht funktionelle Einheit an; Compiler nimmt Zuordnung zur *Compile-Zeit* vor.
1 Bit in jedem Befehl: Nächster Befehl noch im selben Takt?
Max. 256 Befehlsbits (*instruction packet*) pro Zyklus.



Ausführung:

Schritt 1 Befehle A, B, C, D, E

Schritt 2 Befehl F

De facto variable
Befehlslänge von 32
bis 256 Bit.

Sprünge „mitten in ein
Befehls-paket“ hinein.

Der TMS320C62xx Signalprozessor (3)

- **Sättigungsarithmetik,**
- **Modulo-Adressierung** (engl. *modulo addressing*),
- Normalisierungsoperation,
- integrierte E/A-Leitungen,
- kein virtueller Speicher.

VLIW: Diskussion (1)

■ Vergrößerung der Codegröße

• Gründe

- Wegen des Abrollens von Schleifen (um mehr Parallelisierungspotential zu schaffen)
- Unbenutzte funktionale Einheiten
☞ unbenutzte Teile des VLIW-Befehlswords (NOPs)

• Mögliche Gegenmaßnahmen:

- Nur ein großes Feld für Direktoperanden (für beliebige Operation im Befehlsword) ☞ z.B. Crusoe
- Komprimierung des Binärcodes
- Kein festes VLIW, sondern Kodierung parallel auszuführender Befehle mit variabler Codelänge ☞ EPIC / IA-64

VLIW: Diskussion (2)

- **Binärkompatibilität:**

Generierung der Befehlskodierung macht expliziten Gebrauch von Wissen über **interne Architektur** des Prozessors (insbes. Anzahl funktionaler Einheiten, aber auch zum *Pipelining*)

☞ Code ggf. **nicht** auf veränderter interner Architektur lauffähig

Hier: Parallelisierung durch Compiler (vs. Hardware)

Widerspricht eigentlich der Idee einer externen

Architektur (Befehlssatz) als Abstraktion von

Realisierung und Schnittstelle zum Programmierer

VLIW: Diskussion (3)

- Erzeugung ausreichender Parallelität auf Instruktionsebene (ILP = *instruction level parallelism*)

Nur innerhalb von Basisblöcken.

Abrollen von Schleifen primäre Möglichkeit.

☞ Parallele Funktionseinheiten können ggf. nicht ausreichend genutzt werden.

Problem jeder Parallelverarbeitung, auch wenn Befehle auf Funktionseinheiten [dynamisch] verteilt werden ☞ Kap. 3

EPIC-Befehlssätze

EPIC = *Explicitly Parallel Instruction Computing*

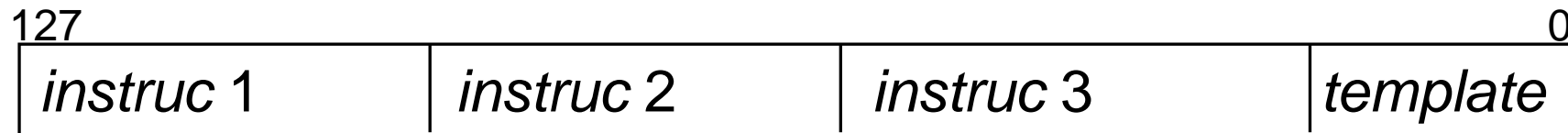
Oberklasse der VLIW-Maschinen

- Explizite Angabe der Parallelität
- Aber: Keine 1:1 Beziehung zwischen Worten im Binärkode und paralleler Ausführung
- Auch: Mikroarchitektur übernimmt Aufgaben bei der Steuerung der Parallelverarbeitung
(bei "reinem" VLIW: Kein Eingreifen der Hardware nötig)

Beispielarchitektur: Intel IA-64

Mehr Kodierungsfreiheit mit dem IA-64 Befehlssatz

3 Instruktionen pro *bundle*:



Es gibt 5 Befehlstypen:

- A: allgemeine ALU Befehle
- I: speziellere integer Befehle (z.B. *shifts*)
- M: Speicher-Befehle
- F: Gleitkomma-Befehle
- B: Sprünge

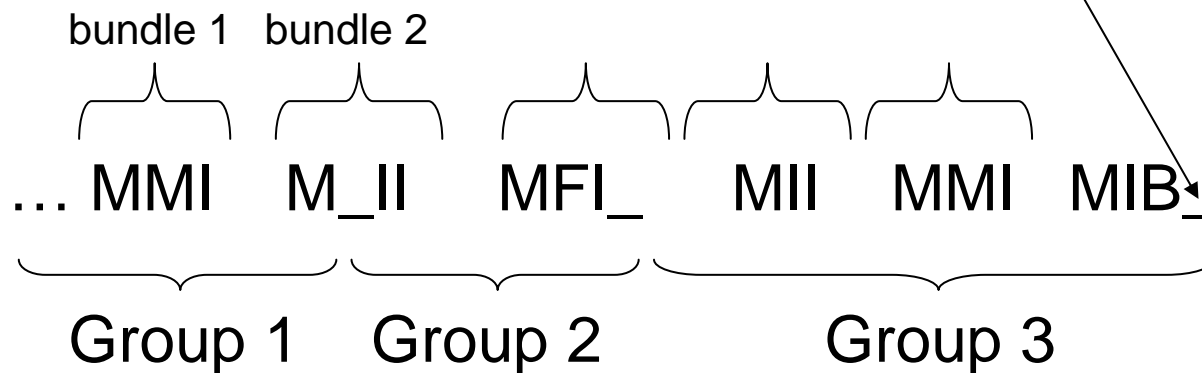
*Instruction
grouping
information*

Die folgenden Kombinationen können in *Templates* kodiert werden:

- MII, MMI, MFI, MIB, MMB, MFB, MMF, MBB, BBB, MLX
mit LX = *move 64-bit immediate* kodiert in 2 slots

Stops

Ende der parallelen Ausführung durch **stops**.
Bezeichnet durch Unterstriche.
Beispiel:

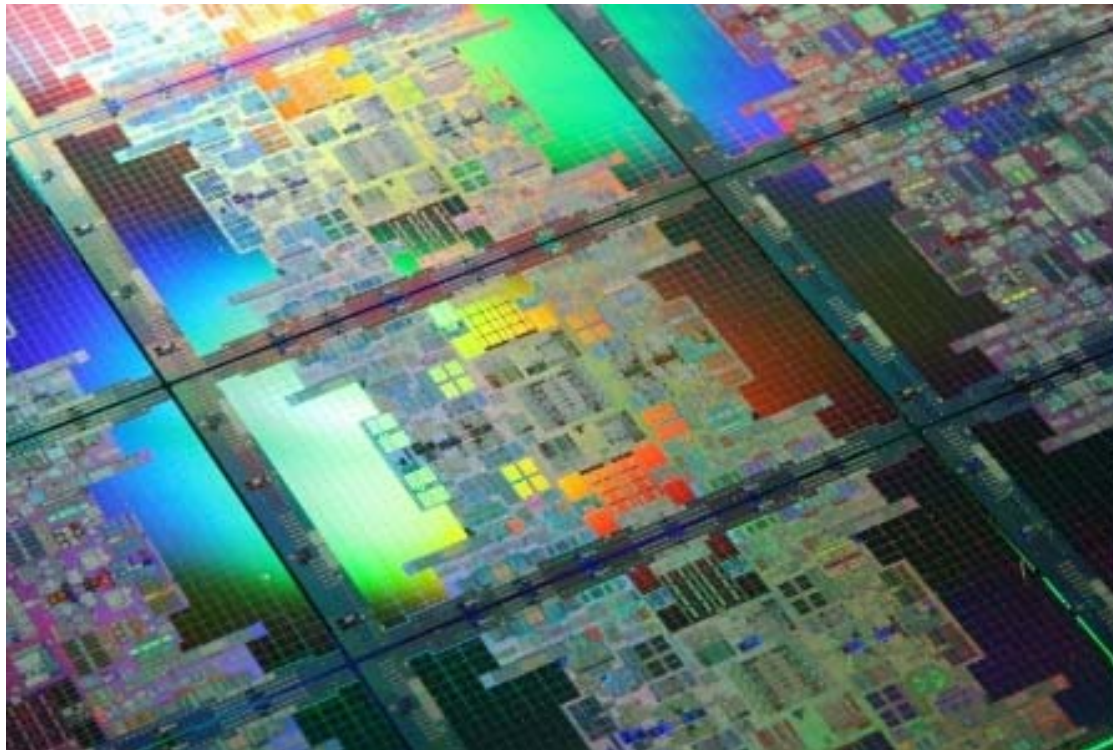


- Sehr beschränkte Platzierung von Unterstrichen in einem Bündle.
- Parallele Ausführung kann sich über mehrere Bündel erstrecken.

Intel IA-64 Vertreter: Itanium

- Erste Implementierung der IA-64 Architektur
- Prozessorkern erlaubt bis zu 6 Befehlsausführungen pro Taktzyklus (inkl. bis zu 3x *Branch*, 2x Speichertransfer)
- Speicherhierarchie mit 3 Cache-Ebenen
(Daten- und Instruktions-Cache getrennt auf 1. Ebene)
- 9 funktionale Einheiten:
2x *I-unit* (ALU+*shift*), 2x *M-unit* (ALU+Load/Store),
3x *B-unit* (*Branches* etc.), 2x *F-unit* (Gleitkomma)
- Dekodierungs-“Fenster” umfasst 2 *bundles*
- Allokation von Instruktionen auf funktionale Einheiten
 - Falls erforderliche Einheit bereits belegt → Bündel teilen
 - ∃ spezielle Restriktionen (z.B. M-M-F bundles nur isoliert)

Itanium® 9300 (Tukwila), 2010



- 2 G Transistoren
- 4 Kerne
- 8-fach Hyper-threading/Kern
- 1.5 GHz bei 1.3V

Bröckelnder Support für die Plattform ...

[<http://www.intel.com/cd/corporate/pressroom/emea/deu/442093.htm>]

Die TRIMARAN-Compiler-Infrastruktur



“An Infrastructure for Research in Backend Compilation and Architecture Exploration
Trimaran is an integrated compilation and performance monitoring infrastructure. The architecture space that Trimaran covers is characterized by HPL-PD, a parameterized processor architecture supporting novel features such as predication, control and data speculation and compiler controlled management of the memory hierarchy. Trimaran also consists of a full suite of analysis and optimization modules, ... Trimaran is intended for researchers and educators interested in the following topics:”

[www.trimaran.org]

Buch

Joseph A. Fisher, Paolo Faraboschi, Cliff Young: *Embedded Computing - A VLIW Approach to Architecture, Compilers and Tools*, Morgan-Kaufmann, 2004 (nicht wirklich stark auf Eingebettete Systeme ausgerichtet)

Zusammenfassung

- Gründe für die Abkehr vom Prinzip immer schnellerer superskalarer Prozessoren
 - Zunehmende Komplexität der Hardware
 - Fehlende Energieeffizienz
- ☞ Idee der Verlagerung der Parallelitätserkennung in den Compiler
 - VLIW: Festes Befehlsformat:
nicht kompakt (wegen NOPs), Binärkompatibilität?
 - EPIC: Größere Flexibilität der Kodierung
Hardware darf in begrenztem Umfang über Parallelität entscheiden
- Zunächst zurückhaltende kommerzielle Nutzung, bei *general purpose computing* wieder nachlassende Unterstützung

Reserve

Beispiel

Schleife: `for (i = n-1; i >= 0; i--) x[i] += a;`

In (MIPS-artigem) Assembler

```
loop: lw $8, 0($4)
      add $8, $8, $6
      sw $8, 0($4)
      add $4, $4, -4
      bge $4, $5, loop
```

Annahmen: $\$4 = \&x$, $\$5 = \&x[0]$, $\$6 = a$

In IA-64 *bundles*

| <i>template</i> | <i>slot 0</i> | <i>slot 1</i> | <i>slot 2</i> |
|-----------------|-----------------|-----------------|------------------|
| M-M-I+S | lw \$8, 0(\$4) | --- | --- |
| M-M-I+S | add \$8,\$8,\$6 | --- | --- |
| M-M-B+S | sw \$8, 0(\$4) | add \$4,\$4,#-4 | bne \$4,\$5,loop |

Annahme: Indexrechnung + Branch parallel!

Man erhält (in dieser Version) Folge von 3 EPIC-Befehlen

Beispiel 2

Abgerollte Schleife (4-mal):

```
for (i = n-4; i >= 0; i-=4) {  
    x[i]    += a;  
    x[i+1] += a;  
    x[i+2] += a;  
    x[i+3] += a; }  
}
```

In (MIPS-artigem) Assembler

```
loop: lw $8, 0($4)  
      lw $9, 4($4)  
      lw $10, 8($4)  
      lw $11, 12($4)  
      add $8, $8, $6  
      ... # add für $9...$11  
      sw $8, 0($4)  
      ... # sw für $9...$11  
      add $4, $4, -16 # Indexrechnung  
      bge $4, $5, loop
```

Beispiel 2 (2)

In IA-64 bundles (mögliche abgerollte Versionen)

| template | slot 0 | slot 1 | slot 2 |
|----------|-----------------|-----------------|--|
| M-M-I+S | lw \$8,0(\$4) | lw \$9,4(\$4) | --- |
| M-M-I | lw \$10,8(\$4) | lw \$11,12(\$4) | add \$8,\$8,\$6 kein M-M+S-I möglich! |
| M+S-M-I | add \$9,\$9,\$6 | add \$10,... | add \$11,... |
| M-M-I+S | sw \$8,0(\$4) | sw \$9,4(\$4) | --- |
| M-M-I | sw \$10,... | sw \$11,... | |
| M-I-B+S | --- | add \$4,\$4,-16 | bne \$4,\$5,loop |

Beispiel 2 (3)

Mit mehr *bundles*

| <i>template</i> | <i>slot 0</i> | <i>slot 1</i> | <i>slot 2</i> |
|-----------------|-----------------|-----------------|------------------|
| M-M-I | lw \$8,0(\$4) | lw \$9,4(\$4) | --- |
| M-M-I+S | lw \$10,8(\$4) | lw \$11,12(\$4) | --- |
| M-M-I | add \$8,\$8,\$6 | add \$9,\$9,\$6 | add \$10,... |
| M-M-I+S | --- | --- | add \$11,... |
| M-M-I | sw \$8,0(\$4) | sw \$9,4(\$4) | --- |
| M-M-I | sw \$10,... | sw \$11,... | --- |
| M-I-B+S | --- | add \$4,\$4,-16 | bne \$4,\$5,loop |

IA-64: Beispiel auf Itanium

In IA-64 bundles (mögliche abgerollte Versionen)

| template | slot 0 | slot 1 | slot 2 | Phase |
|----------|-----------------|--|-----------------------|-------|
| M-M-I+S | lw \$8,0(\$4) | lw \$9,4(\$4) | --- | 1 |
| M-M-I | lw \$10,8(\$4) | lw \$11,12(\$4) | add \$8,\$8,\$6 | 2 |
| | | | beide M-units belegt! | |
| M+S-M-I | add \$9,\$9,\$6 | | | 3 |
| | | add \$10,... | add \$11,... | (4) |
| M-M-I+S | sw \$8,0(\$2) | | | 4 |
| | | sw \$9,4(\$2) | --- | 5 |
| | | beide M-units belegt -> bundle wird geteilt! | | |
| M-M-I | sw \$10,... | | | (6) |
| | | sw \$11,... | --- | (6) |
| M-I-B+S | --- | add \$4,\$4,-16 | bne ... | 6 |

IA-64: Beispiel auf Itanium (2)

Mit mehr bundles

| template | slot 0 | slot 1 | slot 2 | Phase |
|----------|----------------|-----------------|--------|-------|
| M-M-I | lw \$8,0(\$4) | lw \$9,4(\$4) | --- | 1 |
| M-M-I+S | lw \$10,8(\$4) | lw \$11,12(\$4) | --- | 2 |

Nur 2 Speichertransfereinheiten

| | | | | |
|---------|-----------------|-----------------|--------------|-----|
| M-M-I | add \$8,\$8,\$6 | add \$9,\$9,\$6 | add \$10,... | (3) |
| M-M-I+S | --- | --- | add \$11,... | 3 |

2 M- und 2 I-units vorhanden, aber Slotbelegung beachten!

| | | | | |
|---------|---------------|-----------------|---------|-----|
| M-M-I | sw \$8,0(\$4) | sw \$9,4(\$4) | --- | 4 |
| M-M-I | sw \$10,... | sw \$11,... | --- | (5) |
| M-I-B+S | --- | add \$4,\$4,-16 | bne ... | 5 |