

Eingebettete Prozessorarchitekturen

2. Very Long Instruction Word-Prozessoren

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt

Wintersemester 2010/2011

- Grundlage der meisten aktuellen Prozessoren
 - Idee
 - Mehrere Recheneinheiten in Pipeline
 - Führen mehrere Instruktionen gleichzeitig aus
 - Teilweise auch spekulativ
 - Einschränkungen durch
 - Strukturelle Hazards
 - RAW-Hazards (echte Datenabhängigkeiten)
 - WAW- und RAW-Hazards (teilweise umgehbar)
 - Control Hazards (teilweise vorhersagbar)
- ➔ Häufig durch Spezial-Hardware bearbeitet
Benötigt viel Chip-Fläche!

- Parallelität schon **vor Ausführung** herstellen
 - Beschreibe explizit parallele Abläufe
 - Manuell direkt bei Programmierung
 - Automatisch durch geeigneten Compiler
- Auswirkungen auf Hardware
 - Kann viel Chip-Fläche für Verwaltung einsparen
 - Alle Recheneinheiten müssen **parallel kontrolliert** werden
 - Sehr breite Instruktionsworte

➔ Very Long Instruction Word (VLIW)-Prozessor

Ab hier ausgewähltes Material aus dem Buch

Embedded Computing

– A VLIW Approach to Architecture, Compilers, and Tools

von

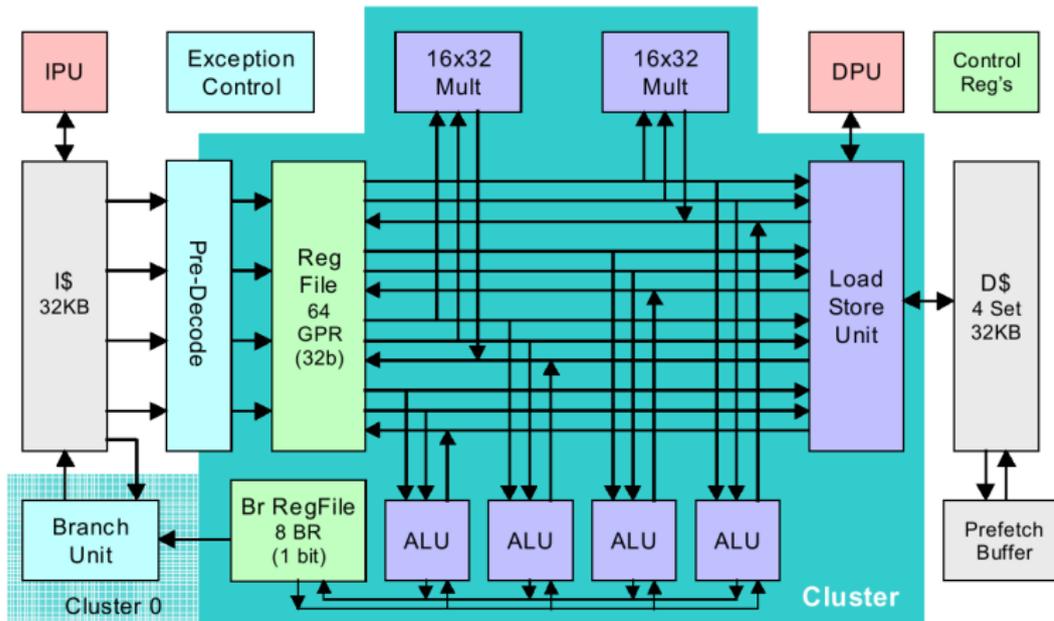
J.A. Fisher, P. Faraboschi und C. Young

- 1984 Gründung von Cydrome und Multiflow
- 1987 Erste Auslieferung der Cydra-5 und Trace/200
7 bzw. 28 parallele Operationen
- 1988 Cydrome schliesst
- 1989 Trace/300 ausgeliefert
14 parallele Operationen
- 1990 Multiflow schliesst
- 1992 Philips Trimedia TM32
5 parallele Operationen
- 1997 TI TMS320C6000 DSPs
8 parallele Operationen
- 2000 HP/STM Lx
4,8,16 parallele Operationen
- 2001 Intel Itanium-Architektur
6 parallele Operationen

HP/STMicroelectronics Lx-Kern

Aufbau

A. Koch



- Vier parallele Operationen (32b Integer ALUs)
- Zwei 16x32b Multiplizierer
- Ein Load/Store Unit
- 64x 32b Register
- 8x 1b Bedingungsregister

➡ Parameter können aber variiert werden

- Vereinfachte Version der Lx-Architektur für Lehre
- Vernachlässigt einige Details, z.B.
 - Konkrete Codierung von Instruktionen
 - Beschränkte Wortbreite von Immediate-Operanden
- Kann aber leicht auf konventionellem Prozessor simuliert werden
- Paket: C-Compiler, Assembler, Simulator, Profiler, Visualisierung
 - <http://www.hp1.hp.com/downloads/vex/>
 - Auch auf Rechnern der FG ESA installiert
- Wird auch noch in VL vorgeführt

- Allgemeine Register: $\$r0 \dots \$r63$
 - Speichern 32b Werte
- 1b-Bedingungsregister: $\$b0 \dots \$b7$
 - Speichern boolesche 1b Ergebnisse von Vergleichen
- Rücksprungregister: $\$10$
 - Speichert Rückkehradresse bei Prozeduraufruf

In der VL vereinfachte Notation!

In Wirklichkeit noch mit extra "0." : $\$r6$ ist $\$r0.6$ etc.

- `$r0` ist immer Null
- `$r1` ist der Stapelzeiger (zeigt auf oberstes Element)
- Weitere Konventionen im ABI
 - Parameterübergabe, Rückgabewert
 - Überschreibbare und erhaltene Register
 - Werden bei Bedarf eingeführt
 - Detaillierte Beschreibung in VEX Dokumentation
 - Tabelle 9
 - Liegt auch auf Web-Seite der VL

Assemblerbefehle

Arithmetische und Verschiebe-Operationen

```
add  $r13 = $r3, $r2      # r13 = r3 + r2
```

```
sub  $r12 = $r2, 1        # r12 = r2 - 1
```

```
mov  $r11 = 42            # r11 = 42
```

```
stw  0x10[$r1] = $r2      # *(r1+16) = r2
```

```
ldw  $r10 = 0x10[$r1]     # r10 = *(r1+16)
```

```
cmplt $b0 = $r0, 100      # b0 = r0 < 100
```

Assemblerbefehle

Sprungoperationen

```
goto    label1                # unbedingter Sprung
```

```
call    $t0 = myroutine       # Unterprogrammaufruf
```

```
return $r1 = $r1, 0, $t0     # Rückkehr von Unterprogramm
```

```
br      $b0, label2          # springe, falls $b0 wahr
```

```
brf     $b0, label3          # springe, falls $b0 falsch
```

```
slct  $r2 = $b0, $r3, $r4 # r2 = b0 ? r3 : r4
```

```
slctf $r2 = $b0, $r3, $r4 # r2 = !b0 ? r3 : r4
```

➡ Vollständige Übersicht in VEX Dokumentation
Tabellen 1 bis 6

Annahme: Speichersystem braucht 3 Takte beim Lesen
(Latenz 2)

A. Koch

Superskalarerer Prozessor

```
lw    $t7, 4($t2)
addi  $t6, $t7, 3
```

VLIW-Prozessor

```
ldw   $r7 = 4[$r2]
xnop  2
add   $r6 = $r7, 3
```

- Löst Datenabhängigkeiten zur Laufzeit auf
 - `addi` wird verzögert, bis Daten da sind
 - Braucht Hardware!
- Verzögerung muss **explizit** im Programm angegeben werden
 - `$r7` kann bei zu frühem Lesen alten Wert liefern
 - Was soll das?
 - Erschwert doch nur Programmierung?

Superskalarerer Prozessor

```
lw    $t7, 4($t2)
addi  $t6, $t7, 3
subi  $t4, $t3, 2
subi  $t5, $t3, 1
```

VLIW-Prozessor

```
ldw   $r7 = 4[$r2]
sub   $r4 = $r3, 2
sub   $r5 = $r3, 1
add   $r6 = $r7, 3
```

A. Koch

- **subi** werden vor **addi** vorgezogen
 - Umsortieren von Anweisungen
 - **Dynamisches Scheduling**
 - Zur Laufzeit
 - Braucht wieder Chip-Fläche!
- Programmierer/Compiler sortiert Anweisungen
 - **Statisches Scheduling**
 - Vor Ausführung
 - Braucht keine zusätzliche Hardware

- Mehr Freiheit für Programmierer/Compiler
- Wartezeit mit nützlichen Anweisungen füllen
- Wenn keine existieren: NOPs einfügen

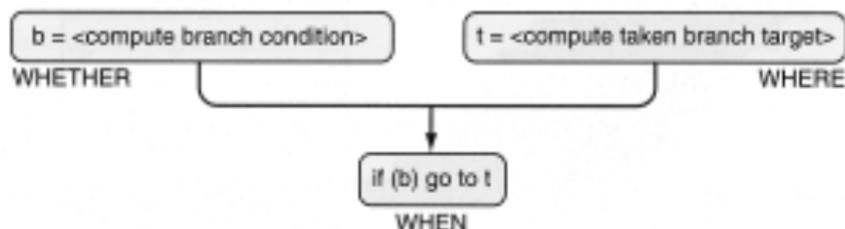
- Verzögerung nicht genau vorhersagbar
- Beispiel: Lesen mit Cache
 - Bei Cache-Hit: 2 Takte
 - Bei Cache-Miss: 18 Takte
- Eigentlich erforderlich . . .
- . . . **immer** 18 Takte warten
- Dann sind Daten sicher da
- Cache wird so aber **überhaupt nicht** genutzt
- Was tun?

- Annahme
 - Erwartete Latenz wird nur selten überschritten
- Vorgehen
 - Plane Schedule für erwartete Latenz
 - Erkenne seltene Überschreitung
 - Halte dann gesamte Maschine an
 - Unschön
 - Braucht aber viel weniger Hardware als superskalärer Ansatz
 - Dort werden gezielt einzelne Recheneinheiten anhalten

- 1 **Sprungvorhersage**
 - Versucht, richtigen Weg zu nehmen
 - Superskalarer Ansatz
 - Braucht dort wieder Chip-Fläche
 - Auch statisch möglich
 - Umstellen der Bedingungen
 - Wahrscheinlicherer Weg auf **nichtgenommenem** Zweig
- 2 **Vermeiden** von bedingten Sprüngen
 - Predication, kommt noch . . .
 - Hat aber auch Nachteile (Effizienz)
- 3 **Bedingte Sprünge effizienter** realisieren
 - Ohne übermäßigen Hardware-Aufwand

Effizientere bedingte Sprünge

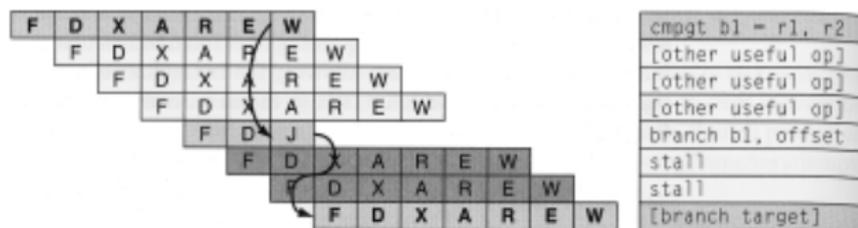
Zerlegen in Teil-Operationen



- 1 Stelle fest, ob Sprungbedingung wahr ist (*whether*)
- 2 Berechne Zieladresse des genommenen Sprungs in besonderem Register (*where*)
- 3 Führe eigentliche Sprungoperation aus
- 4 Vorteil der Zerlegung: Nun sinnvolle Operationen zwischen Schritten einschiebbar

Bedingte Sprünge

Zerlegt in Vergleich und Sprung



A. Koch

- Ergebnis des Vergleichs am Ende der Execute Stufe bekannt
- Annahme hier: Sprung selbst kann nach Decode-Stufe ausgeführt werden
- Vorteile
 - Drei **nützliche** Anweisungen nach Vergleich
 - Nur noch zwei Operationen verschwendet
- Nachteil: Mehr Instruktionen (statistisch ca. 14% mehr)
➔ Ansatz in VEX

Bedingte Sprünge

Zerlegt in Vergleich, Adressberechnung und Sprung



A. Koch

- Ergebnis des Vergleichs am Ende der E-Stufe bekannt
- Annahmen hier
 - Sprungziel-Register mit **prb** nach E-Phase gesetzt
 - Sprung selbst nun nach F-Stufe
 - Hier keine Adressberechnung mehr nötig
- Vorteile
 - Drei nützliche Anweisungen nach Vergleich/Adressberechnung (prb)
 - Nur noch eine Operation verschwendet
- Nachteil: Nun statistisch 28% mehr Instruktionen

➡ Recht selten, Hitachi SuperH SH5 CPU

Vermeiden von bedingten Sprüngen

Prädikation

C-Programm

```
if (x > 0)
    c = a * b; // 1 Takt Latenz
else
    c = a + b; // 0 Takte Latenz
p[0] = c;
```

A. Koch

VEX Assembler

```
cmpgt    $b1 = $r5, 0
xnop     1
brf      $b1, L1
mpy      $r4 = $r2, $r3
goto     L2
L1:
add      $r4 = $r2, $r3
L2:
stw      0[$r10] = $r4
```

- Annahme: Keine weiteren nützlichen Anweisungen
- **x** in **r5**, **a** in **r2**, **b** in **r3**, **c** in **r4**, **p** in **r10**
- Latenz zwischen Vergleich und Sprung: 1 Takt

- Annahme: **Jede** Anweisung kann Bedingungsregister abfragen
 - Wahr: Anweisung normal ausführen
 - Falsch: Anweisung hat keinen Effekt
 - Z.B. Write-Stufe abschalten

```
      cmpgt    $b1 = $r5, 0
($b1) mpy     $r4 = $r2, $r3
(!$b1) add    $r4 = $r2, $r3
      stw     0[$r10] = $r4
```

- Effekt: Beide Zweige werden abgearbeitet
- Aber nur der passende hat einen Effekt
- Vorteil: Kein Bruch in Ausführungsfolge wie beim Sprung
- Pipeline läuft normal weiter

- Nicht jede Anweisung kann Bedingungsregister auswerten
- Spezialisierte Anweisungen
- In VEX *Select*: `s1ct` und `s1ctf`

```
mpy    $r4 = $r2, $r3
add    $r6 = $r2, $r3
cmpgt  $b1 = $r5, 0
s1ct   $r4 = $b1, $r4, $r6
stw    0[$r10] = $r4
```

- Auch hier: Beide Zweige abarbeiten
- Dann Ergebnisse aus passendem Zweig auswählen
- Braucht ggf. mehr Register für Zwischenergebnisse

- Volle Prädikation
 - Intel Itanium: Anweisung kann aus 64 möglichen Bedingungen auswählen
 - TI C6000 DSP: Auswahl aus 8 Bedingungen, Polarität selektierbar
- Partielle Prädikation
 - In HP/STM Lx und VEX
- Allgemeines Problem
 - Pipeline wird zwar nicht unterbrochen
 - Liest aber immer mehr sinnlose Anweisungen ein ...
 - ... die letztlich nicht ausgeführt werden
- Prädikation nur sinnvoll bei **kleinen** IF/ELSE-Blöcken

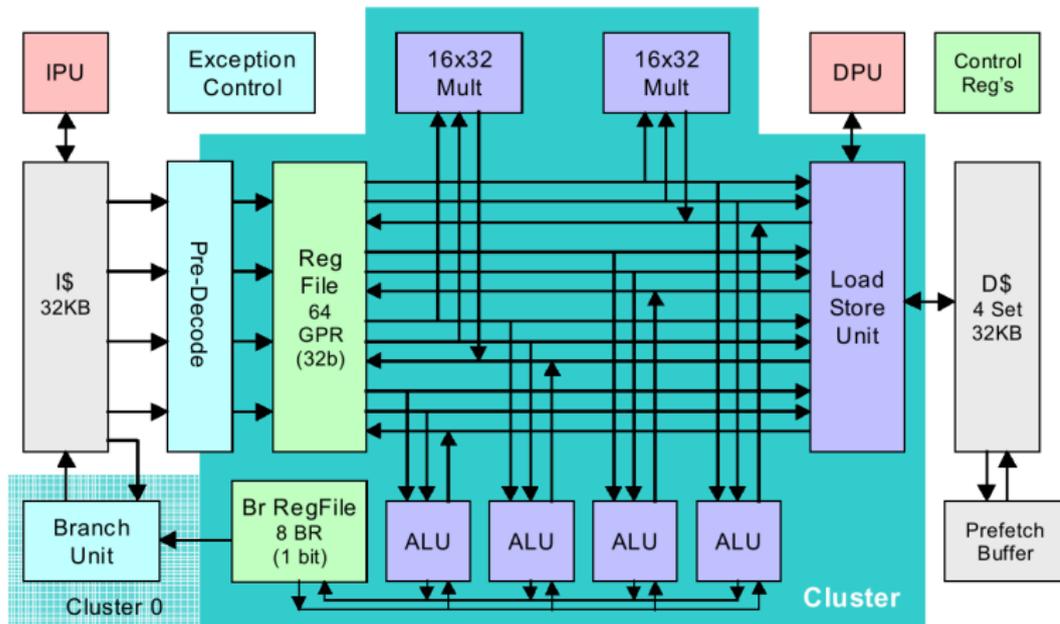
- Viele Tricks machen Prozessor **effizienter**
- Verlagere Arbeit von Chip-Fläche zur Laufzeit . . .
- . . . hin zu Compiler/Programmierer vorher
- Statische statt dynamische Lösung der Probleme

Aber wo ist die angekündigte erhöhte Parallelität??

Nochmal HP/STM Lx-Kern

Beachte: 4 ALUs und 2 Multiplizierer

A. Koch



Aber wie parallel benutzen?

- Eine **Instruktion** ...
- ... kann mehrere **Operationen** umfassen
- Alle Operationen werden **parallel** ausgeführt
- VEX (in Standardkonfiguration):
Bis zu **vier** Operationen gleichzeitig

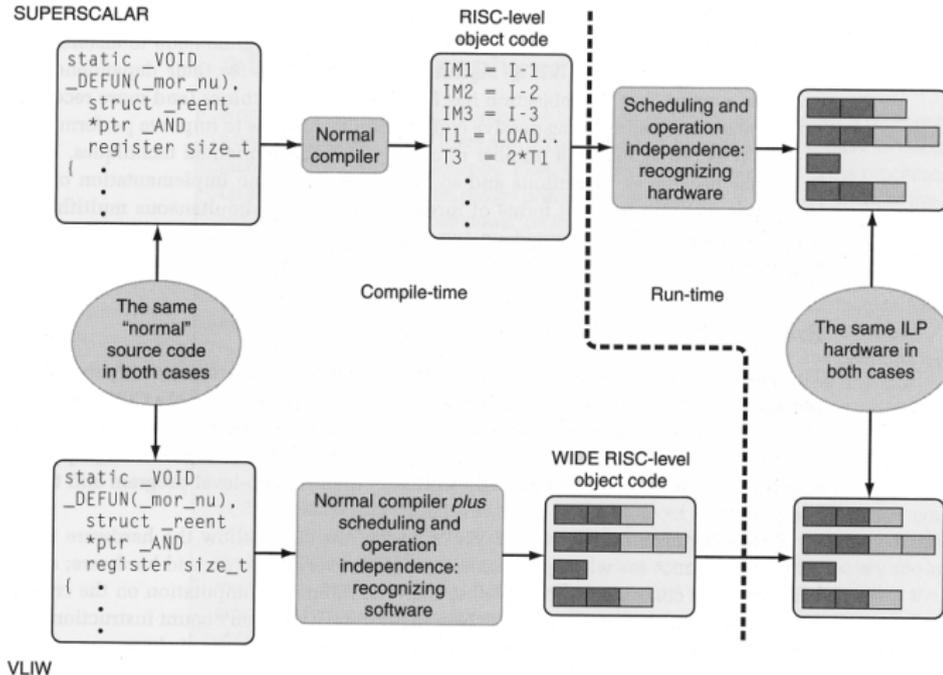
Parallele Ausführung

Das können superskalare Prozessoren auch!

- Dort: Verteilung der Operationen auf Recheneinheiten zur **Laufzeit**
 - Braucht wieder Chip-Fläche
- Bei VLIW: **Statische** Verteilung vor Ausführung
 - Parallele Strukturen werden direkt in Instruktion festgelegt
 - Gleichzeitig mehrere Recheneinheiten steuern
 - Braucht **breitere** Instruktionen

VLIW vs Superskalar

Entwicklungs- und Ausführungsfluss

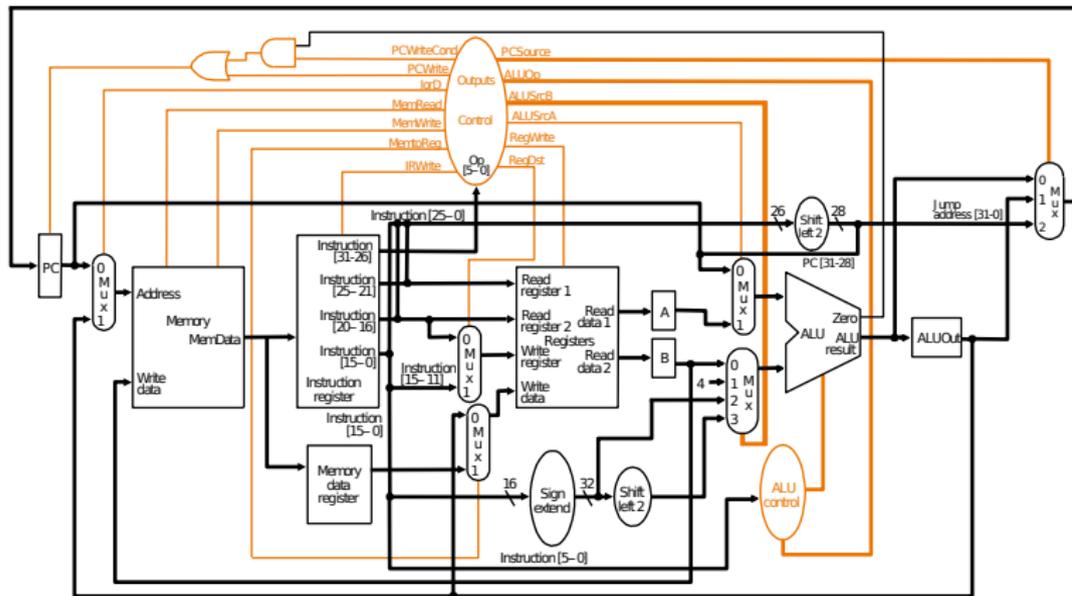


➡ Grundlegende Idee ist auch schon alt

Grundidee: Horizontaler Mikrocode

TGdI2: Multi-Takt MIPS-Prozessor

A. Koch



Grundidee: Horizontaler Mikrocode

TGdI2: Zusammenfassen von Steuerleitungen

Feldname	Wert	aktive Signale	Kommentare
ALU control	Add	ALUOp = 00	Veranlasse ALU zu addieren.
	Subt	ALUOp = 01	Veranlasse ALU zu subtrahieren; implementiert Vergleich für Verzweigungen.
SRC1	Func code	ALUOp = 10	Bestimme ALU Steuerung mit Funktionskod der Instruktion.
	PC	ALUSrcA = 0	Nimm PC als erste ALU Eingabe.
	A	ALUSrcA = 1	Register A ist erste ALU Eingabe.
SRC2	B	ALUSrcB = 00	Register B ist zweite ALU Eingabe.
	4	ALUSrcB = 01	Nimm 4 als zweite ALU Eingabe.
	Extend	ALUSrcB = 10	Nimm Ausgabe der Vorzeicheneinheit als zweite ALU Eingabe.
	Extshft	ALUSrcB = 11	Verwende Ausgabe der verschiebe-um-zwei Einheit als zweite ALU Eingabe.
Register control	Read		Lies zwei Register mit den rs und rt Feldern von IR als Registernummern und speichere die Daten in die Register A und B.
	Write ALU	RegW rite, RegDst = 1, MementoReg = 0	Schreibe in Register mit dem rd Feld des IR als Registernummer und dem Inhalt von ALUOut als Daten.
	Write MDR	RegW rite, RegDst = 0, MementoReg = 1	Schreibe in Register mit dem rt Feld des IR als Registernummer und dem Inhalt von MDR als Daten.
	Read PC	MemRead, lorD = 0	Lies den Speicher mit dem PC als Adresse; schreibe das Ergebnis ins IR (und das MDR).
Memory	Read ALU	MemRead, lorD = 1	Lies den Speicher mit der ALUOut als Adresse; schreibe das Ergebnis ins MDR.
	Write ALU	MemW rite, lorD = 1	Schreibe in den Speicher mit der ALUOut als Adresse, Inhalt von B als Daten.
	ALU	PCSource = 00 PCW rite	Schreibe die Ausgabe der ALU in den PC.
PC write control	ALUOut-cond	PCSource = 01, PCW riteCond	Bei gesetzter Zero Ausgabe der ALU schreibe den Inhalt des Registers ALUOut in den PC.
	jump address	PCSource = 10, PCW rite	Schreibe die Sprungadresse aus der Instruktion in den PC.
Sequencing	Seq	AddrCtl = 11	Wähle die nächste Mikroinstruktion sequentiell.
	Fetch	AddrCtl = 00	Gehe zur ersten Mikroinstruktion, um eine neue Instruktion zu beginnen.
	Dispatch 1	AddrCtl = 01	Verteile mittels des ROM 1.
	Dispatch 2	AddrCtl = 10	Verteile mittels des ROM 2.

Grundidee: Horizontaler Mikrocode

TGdI2: Mikroprogramm

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						ump address	Fetch

- Bisher vereinfacht dargestellt
- 1:1 Zuordnung von Operation zu Instruktion
- Nun volle Funktionalität: Programm beschreibt exakt
 - **welche** Operationen
 - **wann** parallel ausgeführt werden
- Programmierer/Compiler muß Datenabhängigkeiten beachten
 - RAW, WAW, WAR
- Strategie in VEX
 - Alle Operationen einer Instruktion **lesen** ihre Operanden
 - Erst danach finden Schreibzugriffe statt

```
# 1. Instruktion
add  $r3 = $r4, $r5      # 1. Operation
sub  $r6 = $r4, $r5      # 2. Operation
shl  $r7 = $r8, $r5      # 3. Operation
ldw  $r9 = 4[$r1]        # 4. Operation
;;                        # Ende der 1. Instruktion
# 2. Instruktion
cmpgt $b0 = $r3, $r6     # 1. Operation
xnop 1
;;                        # Ende der 2. Instruktion
# 3. Instruktion
slct $r10, $b0, $r7, $r9 # 1. Operation
goto L1                  # 2. Operation
```

- Hier beachtet innerhalb einer Instruktion:
- Nicht mehrfach gleiches Zielregister (wäre WAW)
- Datenabhängigkeit
 - Gelesene Register haben noch alten Wert
 - I.d.R. nicht gleichzeitig Register lesen und schreiben
- Latenz `ldw/slct`: 2 Takte, `cmpgt/slct`: 1 Takt

C-Programm

A. Koch

```
x = a + b;  
if (x > 100)  
    x = 100;  
else if (x < 0)  
    x = 0;
```

VEX-Assembler

```
cmpgt $b0 = $r2, 100  
cmplt $b1 = $r2, 0  
xnop 1  
;;  
slctf $r2 = $b0, $r2, 100  
;;  
slctf $r2 = $b1, $r2, 0
```

Latenz `cmpgt/slct`: 1 Takt

Beispiel: Logische Operationen

C-Programm

```
if (x > 0 || y < 0)
    goto ERROR
```

VEX-Assembler

```
cmpgt $r10 = $r2, $r0
cmplt $r11 = $r3, $r0
;;
orl   $b0 = $r10, r11
xnop  1
;;
br    $b0, ERROR
```

Latenz `orl/br`: 1 Takt

Beispiel: Speicheroperationen

C-Programm

```
if (p)
    *p += 2
```

A. Koch

VEX-Assembler

```
cmpeq $b0 = $r2, $r0
xnop 1
;;
br    $b0, L1
;;
ldw   $r3 = 0[$r2]
xnop 2
;;
add   $r3 = $r3, 2
;;
stw   0[$r2], $r3
;;
L1:
```

- VLIW-Ansatz
- Lx und VEX-Architekturen
- Assembler
- Sichtbare und variable Latenzen
- Dynamisches und statisches Scheduling
- Bedingte Sprünge und Prädikation
- Parallele Ausführung
 - Schwerpunkt der nächsten Vorlesung