

---

# Rechnerstrukturen

Vorlesung im Sommersemester 2007

Prof. Dr. Wolfgang Karl

Universität Karlsruhe (TH)

Fakultät für Informatik

Institut für Technische Informatik



- **Kapitel 2: Parallelismus auf Befehlsebene**

## 2.2: Nebenläufigkeit, Superskalartechnik



- Parallelismus auf Befehlsebene

- Überblick

- Nebenläufigkeit

- Zu einem Zeitpunkt gleichzeitige Ausführung mehrerer Maschinenbefehle zu

- Dynamische Ansätze

- » Superskalare Mikroprozessoren

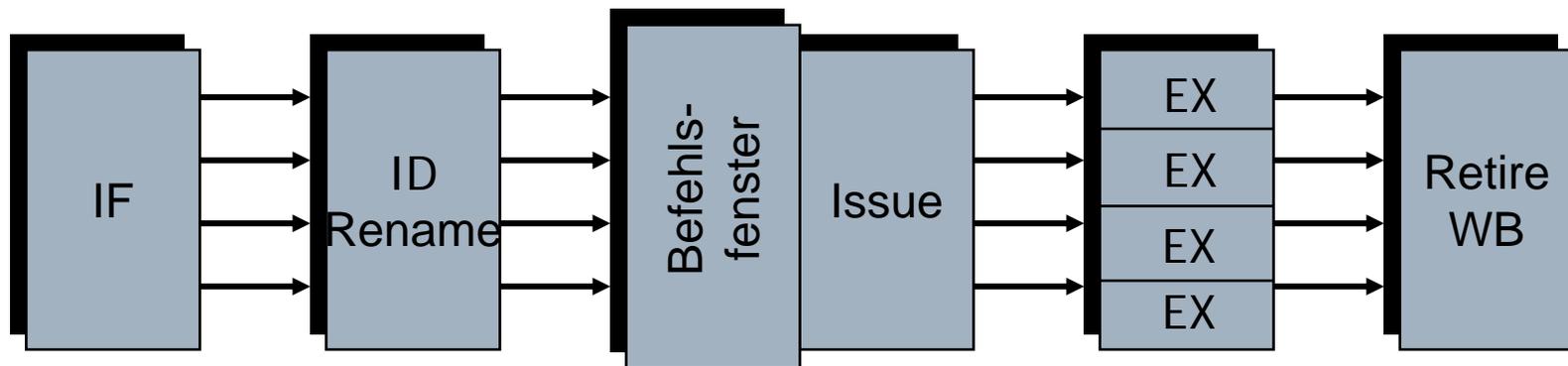
- Statische Ansätze

- » VLIW, EPIC

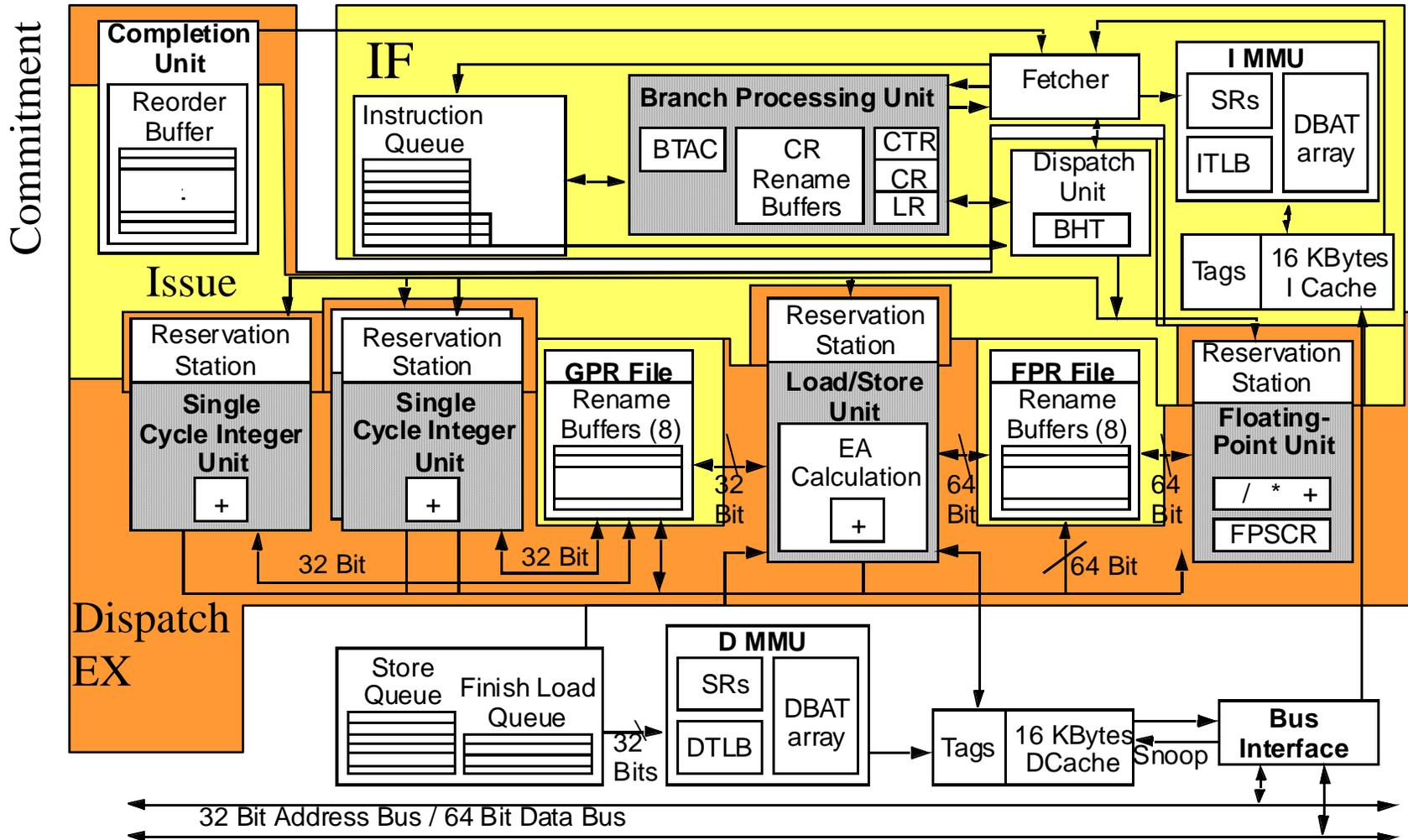


## • Superskalare Prozessor-Pipeline

- Mehrere Maschinenbefehle werden gleichzeitig geholt, dekodiert und ausgeführt; Rückschreiben der Ergebnisse
- Zusätzlich:
  - Zuordnungstufe
  - Rückordnungsstufe
  - Weitere Puffer zur Entkopplung der Pipelinestufen



- Fallstudie: Motorola PowerPC 604



- **Dynamische Methoden zur Erkennung und Auflösung von Datenkonflikten**
  - Detaillierte Betrachtung der Zuordnungsphase
  - Fallstudie: Tomasulo (IBM 360/91)
    - Konfliktauflösung und Ablaufsteuerung verteilt;
    - jede Funktionseinheit verfügt über eine Reservierungstabelle mit möglicherweise mehreren Zeilen (Einträgen);
    - Reservierungstabelle (Umordnungspuffer, Reservation Station)
      - übernimmt die Kontrolle über die Abarbeitung eines Maschinenbefehls, wenn dieser von der Decodiereinheit zur Ausführung angestoßen wird (Issue);
      - ein von einer Funktionseinheit  $i$  produziertes Ergebnis wird direkt an die Funktionseinheit  $j$  weitergegeben, wenn diese das Ergebnis als Operand benötigt;
    - Ergebnisbus:
      - alle Funktionseinheiten, die auf einen Operanden warten, werden gleichzeitig bedient;

- Fallstudie: Tomasulo (IBM 360/91)
  - Umordnungspuffer

	Quelloperand 1			Quelloperand 2			Ziel
	Vld1	Src1	RS1	Vld2	Src2	RS2	Dest
Befehl n							
Befehl n+1							
Befehl n+2							

- Fallstudie: Tomasulo (IBM 360/91)

- Umordnungspuffer

- Jeder Eintrag enthält jeweils Felder für:
  - die zwei möglichen Quelloperanden (Src1, Src2);
  - die Nummern (Namen, Tags) der Reservierungstabellen derjenigen Funktionseinheiten, welche die Quelloperanden für die auszuführende Operation liefern werden (RS1, RS2),
  - jeweils ein Flag für jeden Operanden, das anzeigt, ob ein Operand verfügbar ist (Vld1, Vld2) und
  - einen Namen (destination tag) für das Ziel (Dest).



- Fallstudie: Tomasulo (IBM 360/91)

- Registerdatei

- Jedes Register einer Registerdatei enthält neben dem Wert ein Feld für
  - einen Namen (destination tag, Dest), der mit dem Ergebnis assoziiert ist, und
  - ein Bit, das anzeigt, ob der Wert für das Register gerade berechnet wird.

- Fallstudie: Tomasulo (IBM 360/91)

- Befehlsausführung

- Ein Maschinenbefehl kann zur Ausführung angestoßen werden, falls ein Eintrag in der Reservierungstabelle einer Funktionseinheit frei ist.
  - Falls alle Einträge belegt sind, dann ist ein Ressourcenkonflikt gegeben, und der Maschinenbefehl muss warten, bis ein Eintrag in der Reservierungstabelle frei ist.
  - Für einen von der Decodiereinheit zur Ausführung angestoßener Maschinenbefehl werden die Inhalte seiner Quellregister und die dazugehörigen Ready-Bits in die entsprechenden Felder des Umordnungspuffers kopiert.

- Fallstudie: Tomasulo (IBM 360/91)
  - Phasen der Pipeline (vereinfacht)
    - Issue
      - Holen der Befehle aus Instruction Queue
      - Holen der Operanden
    - Ausführung
      - Wenn die Operanden verfügbar sind, dann Anstoßen der Ausführung auf Funktionseinheit (Dispatch)
      - Beobachten des Ergebnisbusses (Überprüfen von RAW Konflikten)
      - Out-of-Order
    - Rückschreibphase
      - Schreiben der Ergebnisse auf Ergebnisbus
      - Kennzeichnen des Umordnungspuffers als verfügbar

- Fallstudie: Tomasulo (IBM 360/91)
  - Beispiel (T. Ungerer)

		registers					
R		1	2	3	4	5	6
Value		-	-	(R3)	(R4)	(R5)	-
Vld		1	1	1	1	1	1
RS		0	0	0	0	0	0

**register status**

```

mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
    
```

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2
reservation stations	$S_{add}$	1	1								
		2	1								
	$S_{mul}$	3	1								
	$S_{div}$	4	1								

**RS status**

cycle 0

token.tag  
token.data



- Fallstudie: Tomasulo (IBM 360/91)
  - Beispiel (T. Ungerer)

	registers					
R	1	2	3	4	5	6
Value	-	-	(R3)	(R4)	(R5)	-
Vld	0	1	1	1	1	1
RS	3	0	0	0	0	0

**register status**

```

mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
    
```

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2	
reservation stations	$S_{add}$	1	1									
		2	1									
	$S_{mul}$	3	0	0	mul	1	(R3)	1	0	(R5)	1	0
	$S_{div}$	4	1									

**RS status**

cycle 1

token.tag  
token.data



- Fallstudie: Tomasulo (IBM 360/91)
  - Beispiel (T. Ungerer)

		registers					
R		1	2	3	4	5	6
Value		-	-	(R3)	(R4)	(R5)	-
Vld		0	0	1	1	1	1
RS		3	1	0	0	0	0

**register status**

```

mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
    
```

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2	
reservation stations	$S_{add}$	1	0	0	sub	2	(R4)	1	0	(R3)	1	0
		2	1									
	$S_{mul}$	3	0	1	mul	1	(R3)	1	0	(R5)	1	0
	$S_{div}$	4	1									

**RS status**

cycle 2

token.tag  
token.data



- Fallstudie: Tomasulo (IBM 360/91)
  - Beispiel (T. Ungerer)

		registers					
R		1	2	3	4	5	6
Value		-	-	(R3)	(R4)	(R5)	-
Vld		0	0	1	1	1	0
RS		3	1	0	0	0	4

**register status**

```

mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
    
```

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2		
reservation stations	S <sub>add</sub>	1	0	1	sub	2	(R4)	1	0	(R3)	1	0	3 ↑ remaining cycles in FU
		2	1										
	S <sub>mul</sub>	3	0	1	mul	1	(R3)	1	0	(R5)	1	0	
		4	0	0	div	6		0	3	(R4)	1	0	

**RS status**

cycle 3

token.tag  
token.data

- Fallstudie: Tomasulo (IBM 360/91)
  - Beispiel (T. Ungerer)

	registers					
R	1	2	3	4	5	6
Value	-	(R4)-(R3)	(R3)	-	(R5)	-
Vld	0	1	1	0	1	0
RS	3	0	0	2	0	4

register status

```

mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
    
```

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2	
reservation stations	$S_{add}$	1	1	sub	2	(R4)	1	0	(R3)	1	0	2
		2	0	add	4	(R4)-(R3)	1	0	(R3)	1	0	
	$S_{mul}$	3	0	mul	1	(R3)	1	0	(R5)	1	0	
	$S_{div}$	4	0	div	6		0	3	(R4)	1	0	

RS status

cycle 4

```

token.tag      1
token.data    (R4)-(R3)
    
```



- Fallstudie: Tomasulo (IBM 360/91)
  - Beispiel (T. Ungerer)

		registers					
R		1	2	3	4	5	6
Value		-	(R4)-(R3)	(R3)	-	(R5)	-
Vld		0	1	1	0	1	0
RS		3	0	0	2	0	4

**register status**

```

mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
    
```

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2	
reservation stations	$S_{add}$	1	1	sub	2	(R4)	1	0	(R3)	1	0	
		2	0	1	add	4	(R4)-(R3)	1	0	(R3)	1	0
	$S_{mul}$	3	0	1	mul	1	(R3)	1	0	(R5)	1	0
	$S_{div}$	4	0	0	div	6		0	3	(R4)	1	0

**RS status**

cycle 5

token.tag  
token.data



- Fallstudie: Tomasulo (IBM 360/91)
  - Beispiel (T. Ungerer)

		registers					
R		1	2	3	4	5	6
Value		-	(R4)-(R3)	(R3)	(R4)- (R3)+(R3)	(R5)	-
Vld		0	1	1	1	1	0
RS		3	0	0	0	0	4

**register status**

```

mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
    
```

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2
reservation stations	$S_{add}$	1	1	sub	2	(R4)	1	0	(R3)	1	0
		2	1	add	4	(R4)-(R3)	1	0	(R3)	1	0
	$S_{mul}$	3	0	mul	1	(R3)	1	0	(R5)	1	0
	$S_{div}$	4	0	div	6		0	3	(R4)	1	0

**RS status**

cycle 6

```

token.tag      2
token.data    (R4)-(R3)+(R3)
    
```

- Fallstudie: Tomasulo (IBM 360/91)
  - Beispiel (T. Ungerer)

		registers					
R		1	2	3	4	5	6
Value		(R3)*(R5)	(R4)-(R3)	(R3)	<sup>(R4)-</sup> (R3)+(R3)	(R5)	-
Vld		1	1	1	1	1	0
RS		0	0	0	0	0	4

**register status**

```

mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
    
```

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2
reservation stations	<b>S<sub>add</sub></b>	1	1	sub	2	(R4)	1	0	(R3)	1	0
		2	1	add	4	(R4)-(R3)	1	0	(R3)	1	0
	<b>S<sub>mul</sub></b>	3	1	mul	1	(R3)	1	0	(R5)	1	0
	<b>S<sub>div</sub></b>	4	0	div	6	(R3)*(R5)	1	0	(R4)	1	0

**RS status**

cycle 7

token.tag 3  
 token.data (R3)\*(R5)



- Fallstudie: Tomasulo (IBM 360/91)
  - Beispiel (T. Ungerer)

		registers					
R		1	2	3	4	5	6
Value		$(R3) \cdot (R5)$	$(R4) - (R3)$	$(R3)$	$\frac{(R4)}{(R3) + (R3)}$	$(R5)$	-
Vld		1	1	1	1	1	0
RS		0	0	0	0	0	4

**register status**

```

mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
    
```

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2
reservation stations	$S_{add}$	1	1	sub	2	(R4)	1	0	(R3)	1	0
		2	1	add	4	$(R4) - (R3)$	1	0	(R3)	1	0
	$S_{mul}$	3	1	mul	1	(R3)	1	0	(R5)	1	0
	$S_{div}$	4	0	div	6	$(R3) \cdot (R5)$	1	0	(R4)	1	0

**RS status**

cycle 8

token.tag  
token.data

- Fallstudie: Tomasulo (IBM 360/91)
  - Beispiel (T. Ungerer)

		registers					
R		1	2	3	4	5	6
Value		(R3)*(R5)	(R4)-(R3)	(R3)	$\frac{(R4)}{(R3)+(R3)}$	(R5)	-
Vld		1	1	1	1	1	0
RS		0	0	0	0	0	4

**register status**

```

mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
    
```

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2
reservation stations	<b>S<sub>add</sub></b>	1	1	sub	2	(R4)	1	0	(R3)	1	0
		2	1	add	4	(R4)-(R3)	1	0	(R3)	1	0
	<b>S<sub>mul</sub></b>	3	1	mul	1	(R3)	1	0	(R5)	1	0
	<b>S<sub>div</sub></b>	4	0	div	6	(R3)*(R5)	1	0	(R4)	1	0

**RS status**

cycle 9

token.tag  
token.data



- Fallstudie: Tomasulo (IBM 360/91)
  - Beispiel (T. Ungerer)

		registers					
R		1	2	3	4	5	6
Value		(R3)*(R5)	(R4)-(R3)	(R3)	<sup>(R4)-</sup> (R3)+(R3)	(R5)	-
Vld		1	1	1	1	1	0
RS		0	0	0	0	0	4

**register status**

```

mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
    
```

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2
reservation stations	<b>S<sub>add</sub></b>	1	1	sub	2	(R4)	1	0	(R3)	1	0
		2	1	add	4	(R4)-(R3)	1	0	(R3)	1	0
	<b>S<sub>mul</sub></b>	3	1	mul	1	(R3)	1	0	(R5)	1	0
	<b>S<sub>div</sub></b>	4	0	div	6	(R3)*(R5)	1	0	(R4)	1	0

**RS status**

cycle 10

token.tag  
token.data



- Fallstudie: Tomasulo (IBM 360/91)
  - Beispiel (T. Ungerer)

	registers					
R	1	2	3	4	5	6
Value	(R3)*(R5)	(R4)-(R3)	(R3)	$\frac{(R4)-(R3)}{(R3)+(R3)}$	(R5)	-
Vld	1	1	1	1	1	0
RS	0	0	0	0	0	4

**register status**

```

mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
    
```

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2
reservation stations	<b>S<sub>add</sub></b>	1	1	sub	2	(R4)	1	0	(R3)	1	0
		2	1	add	4	$\frac{(R4)-(R3)}{(R3)+(R3)}$	1	0	(R3)	1	0
	<b>S<sub>mul</sub></b>	3	1	mul	1	(R3)	1	0	(R5)	1	0
	<b>S<sub>div</sub></b>	4	0	1	div	6	$\frac{(R3)*(R5)}{(R3)+(R3)}$	1	0	(R4)	1

**RS status**

cycle 11

token.tag  
token.data



- Fallstudie: Tomasulo (IBM 360/91)
  - Beispiel (T. Ungerer)

		registers					
R		1	2	3	4	5	6
Value		$(R3) * (R5)$	$(R4) - (R3)$	$(R3)$	$(R4) - (R3) + (R3)$	$(R5)$	$(R3) * (R5) / (R4)$
Vld		1	1	1	1	1	1
RS		0	0	0	0	0	0

register status

```

mul Reg1, Reg3, Reg5
sub Reg2, Reg4, Reg3
div Reg6, Reg1, Reg4
add Reg4, Reg2, Reg3
    
```

		Empty	InFU	Op	Dest	Src1	Vld1	RS1	Src2	Vld2	RS2
reservation stations	$S_{add}$	1	1	sub	2	(R4)	1	0	(R3)	1	0
		2	1	add	4	$(R4) - (R3)$	1	0	(R3)	1	0
	$S_{mul}$	3	1	mul	1	(R3)	1	0	(R5)	1	0
	$S_{div}$	4	1	div	6	$(R3) * (R5)$	1	0	(R4)	1	0

RS status

cycle 12

```

token.tag    4
token.data    $(R3) * (R5) / (R4)$ 
    
```



- **Zusammenfassung**

- Aus einem sequentiellen Befehlsstrom werden Befehle zur Ausführung angestoßen (zugewiesen).
- Die Zuweisung erfolgt dynamisch durch die Hardware
- Es kann mehr als ein Befehl zugewiesen werden.
- Die Anzahl der zugewiesenen Befehle pro Takt wird dynamisch von der Hardware bestimmt und liegt zwischen Null und der maximalen Zuweisungsbreite.
- Komplexe Hardware-Logik für dynamische Zuweisung notwendig.
- Mehrere von einander unabhängige Funktionsanweisungen sind verfügbar.
- Mikroarchitektur bestimmt superskalare Eigenschaft.

- Literatur:
  - Brinkschulte/Ungerer: Mikrocontroller und Mikroprozessoren. Springer-Verlag, 2002: [Kap. 6.1-6.4, Kap. 7](#)
  - Hennessy/Patterson: Computer Architecture – A Quantative Approach. 3. Auflage: [Kap. 3](#)

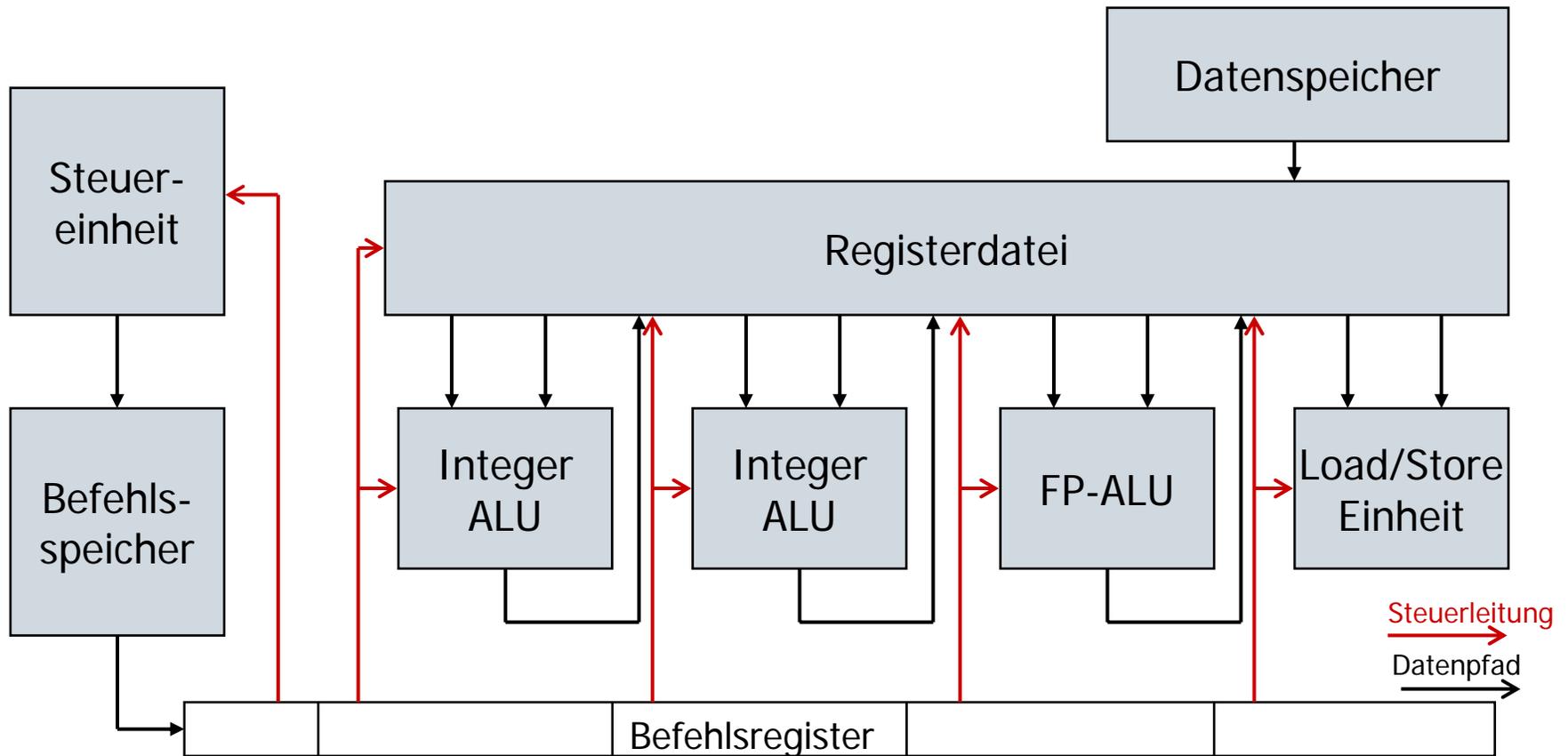
- **Kapitel 2: Parallelismus auf Befehlsebene**

## 2.3: Nebenläufigkeit, VLIW, EPIC

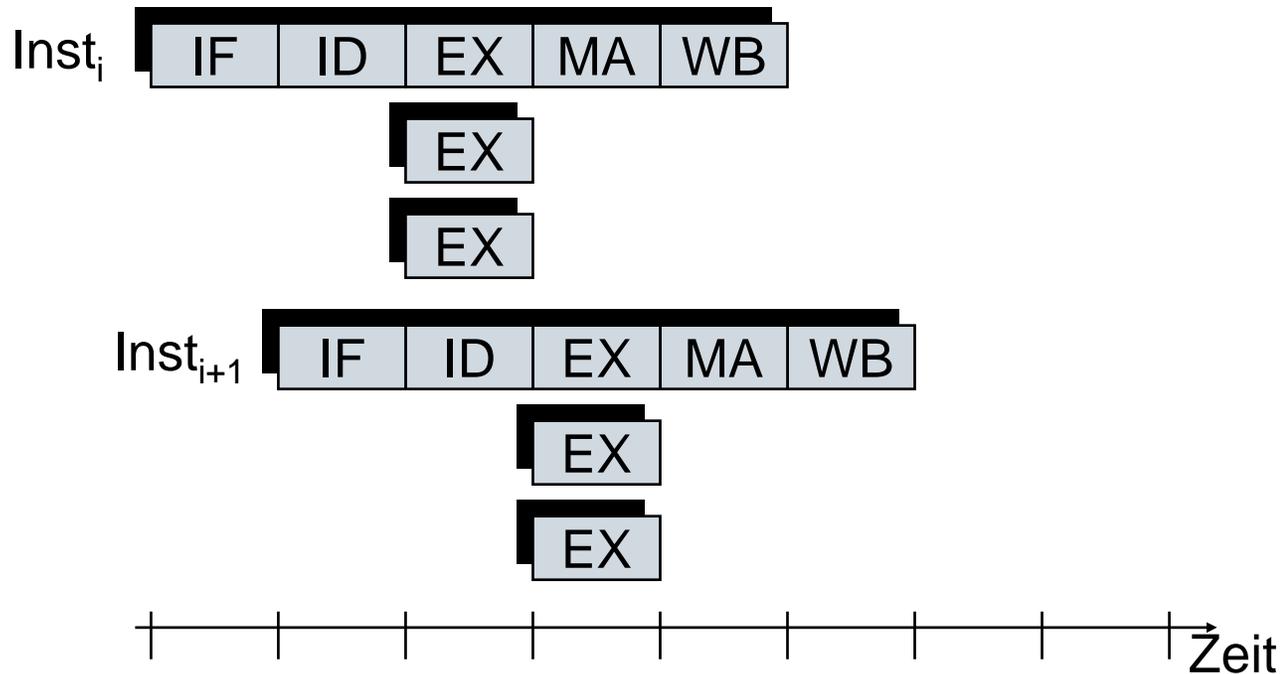


- Grundprinzip: VLIW
  - VLIW: Very Long Instruction Word
  - Eine VLIW-Architektur (Very Long Instruction Word) ist gekennzeichnet durch ein breites Befehlsformat, das in mehrere Felder aufgeteilt ist, aus denen die Funktionseinheiten gesteuert werden;
  - Eine VLIW-Architektur mit  $n$  voneinander unabhängigen Funktionseinheiten kann bis zu  $n$  Operationen gleichzeitig ausführen;
  - Operationen: RISC-Architektur
  - Steuerung der parallelen Abarbeitung zur Übersetzungszeit (automatisch parallelisierender Compiler)

- Grundprinzip VLIW  
– Prinzipieller Aufbau



- Grundprinzip VLIW
  - Prinzipielle Pipeline-Struktur



- Statische Steuerung der parallelen Abarbeitung
  - Aufgaben des Compilers
    - Frontend:
      - Lexikalische, syntaktische und semantische Analyse
    - Code-Generierung / Parallelisierung
      - Kontrollflussanalyse
      - Datenflussanalyse
      - Datenabhängigkeitsanalyse
      - Schleifenparallelisierung
        - » Loop Unrolling
        - » Software-Pipelining
    - Scheduling
      - Packen der voneinander unabhängigen Befehle in breite Befehlswörter



- Statische Steuerung der parallelen Abarbeitung

- Scheduling

- Beispiel (Quelle: K. Asanovic, MIT)

```
for (i=0;i<N;i++)  
  B[i]=A[i]+C
```

Übersetzung ↓

```
Loop:  ld f1,0(r1)  
       add r1,8  
       fadd f2,f0,f1  
       sd f2, 0(r2)  
       add r2,8  
       bne r1,r3,loop
```

- Statische Steuerung der parallelen Abarbeitung

- Scheduling (Fortsetzung Beispiel)

```
for (i=0;i<N;i++)
  B[i]=A[i]+C
```

Übersetzung ↓

```
Loop: ld f1,0(r1)
      add r1,8
      fadd f2,f0,f1
      sd f2, 0(r2)
      add r2,8
      bne r1,r3,loop
```

Scheduling →

FE Int1	FE Int2	FE Mem1	FE Mem2	FE FP+	FE FPx
add r1		ld			
				fadd	
add r2	bne	sd			

- Statische Steuerung der parallelen Abarbeitung
  - Scheduling
    - Loop Unrolling

```
for (i=0;i<N;i++)  
B[i]=A[i]+C
```



Abrollen der inneren Schleife

```
for (i=0;i<N-3;i+4)  
{  
B[i]=A[i]+C  
B[i+1]=A[i+1]+C  
B[i+2]=A[i+2]+C  
B[i+3]=A[i+3]+C  
}
```

- Statische Steuerung der parallelen Abarbeitung
  - Loop Unrolling

```

Loop:  ld f1,0(r1)
       ld f2,8(r1)
       ld f3,16(r1)
       ld f4,24(r1)
       add r1,32
       fadd f5,f0,f1
       fadd f6,f0,f2
       fadd f7,f0,f3
       fadd f8,f0,f4
       sd f5,0(r2)
       sd f6,8(r2)
       sd f7,16(r2)
       sd f8,24(r2)
       add r2,32
       bne r1,r3,loop
    
```

FE Int1	FE Int2	FE Mem1	FE Mem2	FE FP+	FE FPx
		ldf1			
		ldf2			
		ldf3			
add r1		ldf4		fadd f5	
				fadd f6	
				fadd f7	
				fadd f8	
		sd f5			
		sd f6			
		sd f7			
add r2	bne	sd f8			

The diagram illustrates data flow between functional elements (FE). Red arrows indicate dependencies: one arrow points from the 'ldf1' instruction in FE Mem1 to the 'fadd f5' instruction in FE FP+, and another arrow points from the 'sd f5' instruction in FE Mem1 back to the 'fadd f5' instruction in FE FP+.

- Statische Steuerung der parallelen Abarbeitung
  - Software-Pipelining

	FE Int1	FE Int2	FE Mem1	FE Mem2	FE FP+	FE FPx
Loop: <code>ld f1,0(r1)</code> <code>ld f2,8(r1)</code> <code>ld f3,16(r1)</code> <code>ld f4,24(r1)</code>			<code>ldf1</code>			
			<code>ldf2</code>			
			<code>ldf3</code>			
	<code>add r1</code>		<code>ldf4</code>			
<code>add r1,32</code>			<code>ldf1</code>		<code>fadd f5</code>	
<code>fadd f5,f0,f1</code>			<code>ldf2</code>		<code>fadd f6</code>	
<code>fadd f6,f0,f2</code>			<code>ldf3</code>		<code>fadd f7</code>	
<code>fadd f7,f0,f3</code>			<code>ldf4</code>		<code>fadd f8</code>	
<code>fadd f8,f0,f4</code>					<code>fadd f5</code>	
<code>sd f5,0(r2)</code>			<code>ldf1</code>	<code>sd f5</code>	<code>fadd f5</code>	
<code>sd f6,8(r2)</code>			<code>ldf2</code>	<code>sd f6</code>	<code>fadd f6</code>	
<code>sd f7,16(r2)</code>		<code>add r2</code>	<code>ldf3</code>	<code>sd f7</code>	<code>fadd f7</code>	
<code>add r2,32</code>	<code>add r1</code>	<code>bne</code>	<code>ldf4</code>	<code>sd f8</code>	<code>fadd f8</code>	
<code>sd f8,-8(r2)</code>				<code>sd f5</code>	<code>fadd f5</code>	
<code>bne r1,r3,loop</code>				<code>sd f6</code>	<code>fadd f6</code>	
				<code>sd f7</code>	<code>fadd f7</code>	
				<code>sd f8</code>	<code>fadd f8</code>	
				<code>sd f5</code>		

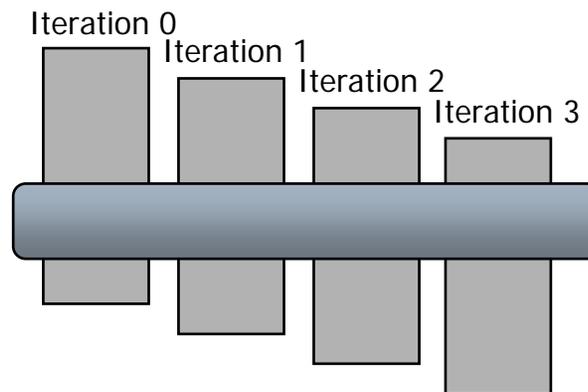
Prolog

Schleife

Epilog



- Statische Steuerung der parallelen Abarbeitung
  - Software-Pipelining
    - Technik zur Reorganisation von Schleifen
      - Jede Iteration in dem mit Software-Pipelining generierten Code enthält Befehle aus verschiedenen Iterationen der ursprünglichen Schleife



- Frühe VLIW-Rechner
  - Multiflow Trace (J. Fisher)
    - Rechnerkonfigurationen mit 7, 14 und 28 Operationen/VLIW-Instruktion
    - 28 Operationen in einem 1024 Bit Wort
    - Numerische Anwendungen
    - Trace-Scheduling
  - Cydrome Cydra-5 (B. Rau)
    - 7 Operationen/256Bit Wort
    - Rotating Register File
- Einsatz VLIW-Technik heute:
  - Allzweck-Mikroprozessoren
    - Transmeta Crusoe
  - Bereich eingebetteter Prozessoren (DSP)
    - Beispiel: Trimedia TM32 Architecture
    - Agere/Motorola Star Core



- TI TMS320C6000 Architekturmerkmale
  - 2 x 4 Funktionseinheiten (A und B Seite)
    - Jede Seite enthält 16 Register
      - Programmierer sieht 32 Register A0 – A15, B0 – B15
      - Ausgewählte Register für Boole'sche Ergebnisse von bedingten Befehlen
      - 9 32 Bit Lese- und 6 32 Bit Schreibports
      - Jeweils ein Crossover-Pfad: Beschränkter wechselseitiger Zugriff
    - Jede Seite enthält
      - Eine 40 Bit Integer-ALU (L Unit)
        - » Arithmetische und logische Operationen, Vergleiche, Normalisierung, Bit-Count,
      - 16 Bit Multiplizierer
        - » 16 x 16 → 32 Bit Multiplikation

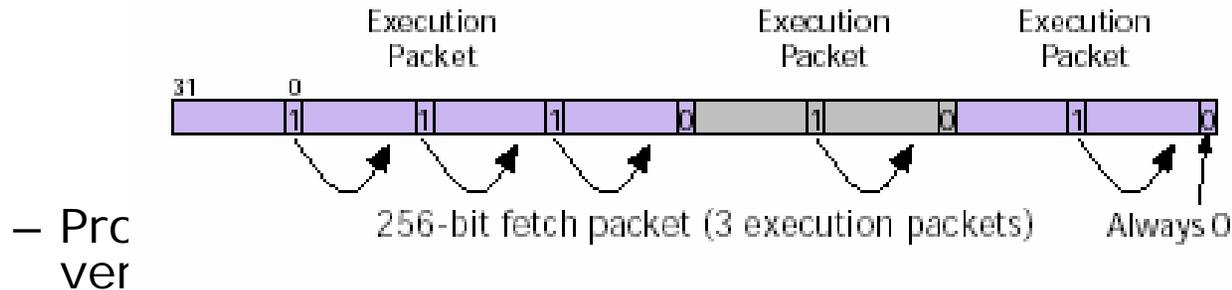
- TI TMS320C6000 Architekturmerkmale
  - 2 x 4 Funktionseinheiten (A und B Seite)
    - 40 Bit Schiebereinheit
      - Arithmetische Operationen, Verzweigung und Verzweigungsadressgenerierung
    - 32 Bit Addierer
      - Adressgenerierung

- TI TMS320C6000 Architekturmerkmale
  - VLIW-Prinzip
    - Holen von 8 32 Bit Befehlen über 256 Bit Befehlsbus (Fetch Packet)
      - Geholte Befehle müssen nicht unbedingt gleichzeitig ausgeführt werden
      - Ein Befehl in einem Fetch Packet ist nicht auf eine Ausführungseinheit beschränkt, jeder Befehl enthält Kodierung, mit der spezifiziert wird, auf welche Einheit der Befehl ausgeführt werden soll
      - Befehle sind nicht positionsabhängig
      - Programmierer / Compiler bestimmt Bindung

- TI TMS320C6000 Architekturmerkmale

- VLIW-Prinzip

- Bis zu 8 Befehle können gleichzeitig ausgeführt werden: Gruppierung von gleichzeitig ausführbaren Befehlen (Execution Packets)
- Werden im niedrigstwertigen Bit gekennzeichnet: alle nachfolgenden Befehle werden gleichzeitig ausgeführt



- Literatur:
  - Hennessy/Patterson: Computer Architecture  
A Quantative Approach. Kap. 4.1-4.4, 4.8

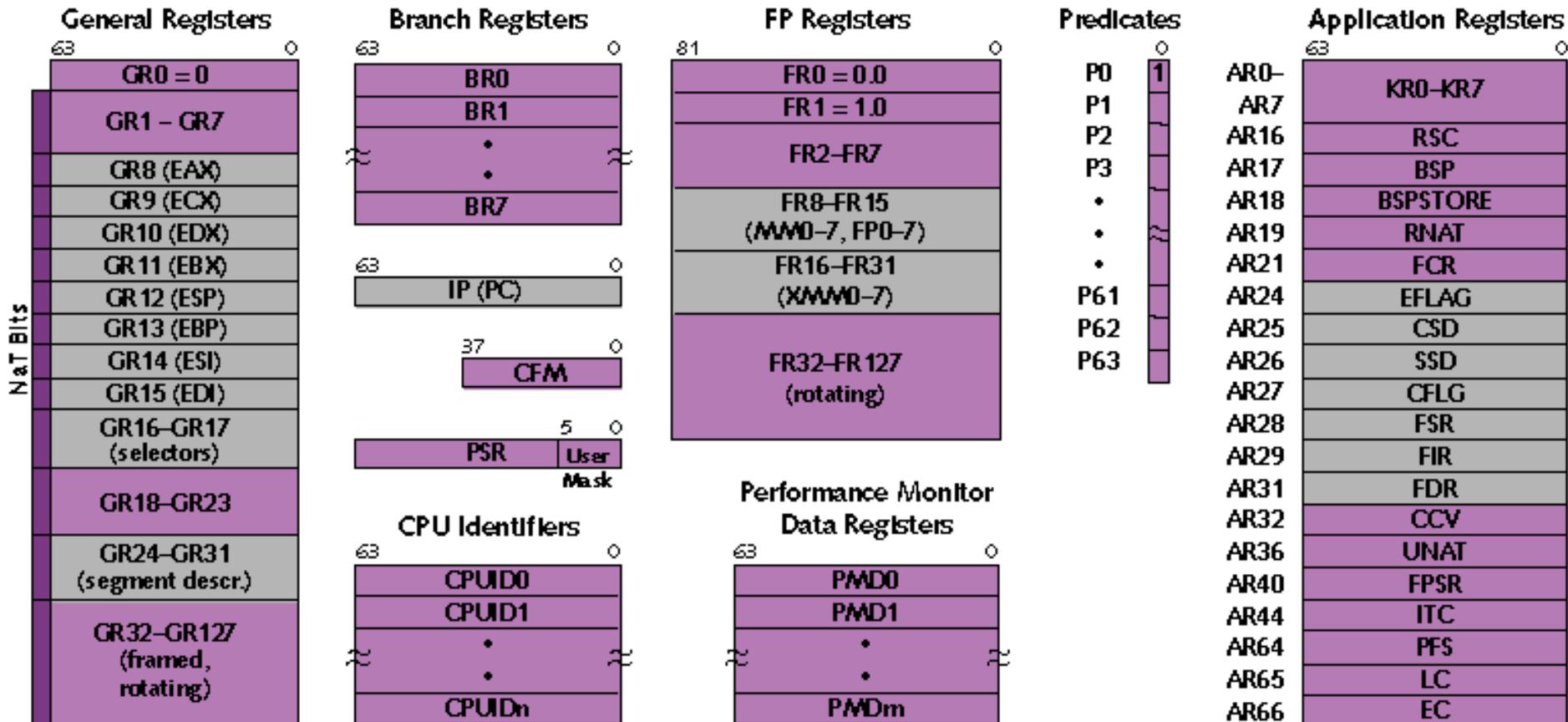


- **EPIC: Explicitly Parallel Instruction Computing**
  - Gemeinsames Projekt von Hewlett-Packard und Intel (1994 angekündigt)
  - Ziele:
    - 64 Bit Architektur: IA-64
    - Explizite Spezifikation des Parallelismus im Maschinencode: EPIC-Format, (entspricht VLIW-Prinzip)
    - Bedingte Ausführung von Befehlen (Predication)
    - Spekulative Ausführung von Ladeoperationen (Data Speculation)
    - Großer Registersatz
    - Skalierbarer Befehlssatz
    - Sinnvolles Zusammenwirken zwischen Compiler und Hardware
  - Itanium: erster Prozessor der P7-Generation (Code-Name Merced)



## • Intel IA-64 – Registersatz

Quelle: Microprocessor Report, Vol.13, Nr.7, 1999

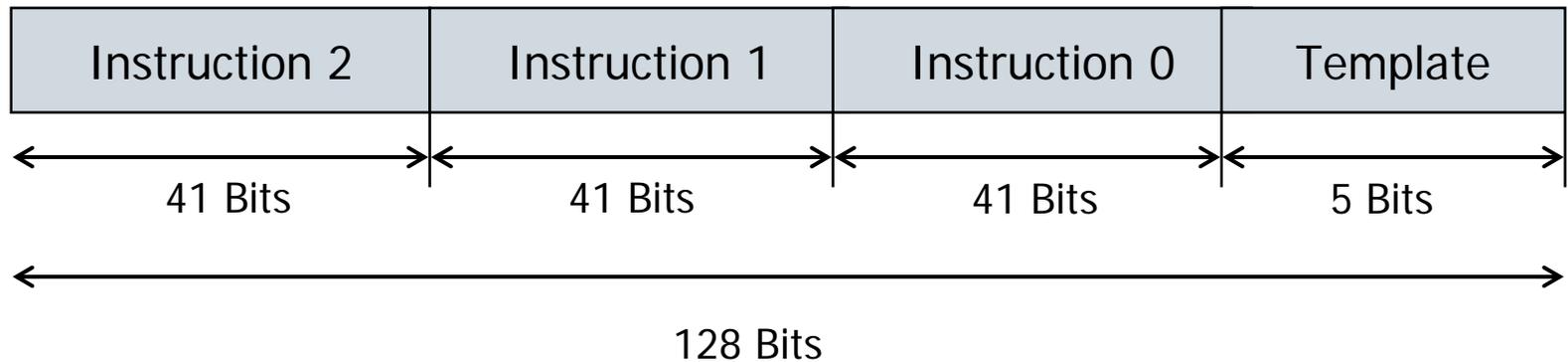


- IA-64 ISA: Befehlsformat
  - Opcode
  - Predicate
  - 2 Felder für Quelloperandenregister
  - 1 Feld für Zielregister



- IA-64 ISA

- IA-64 Instruktionen werden vom Compiler in so genannte Bundles gepackt



---

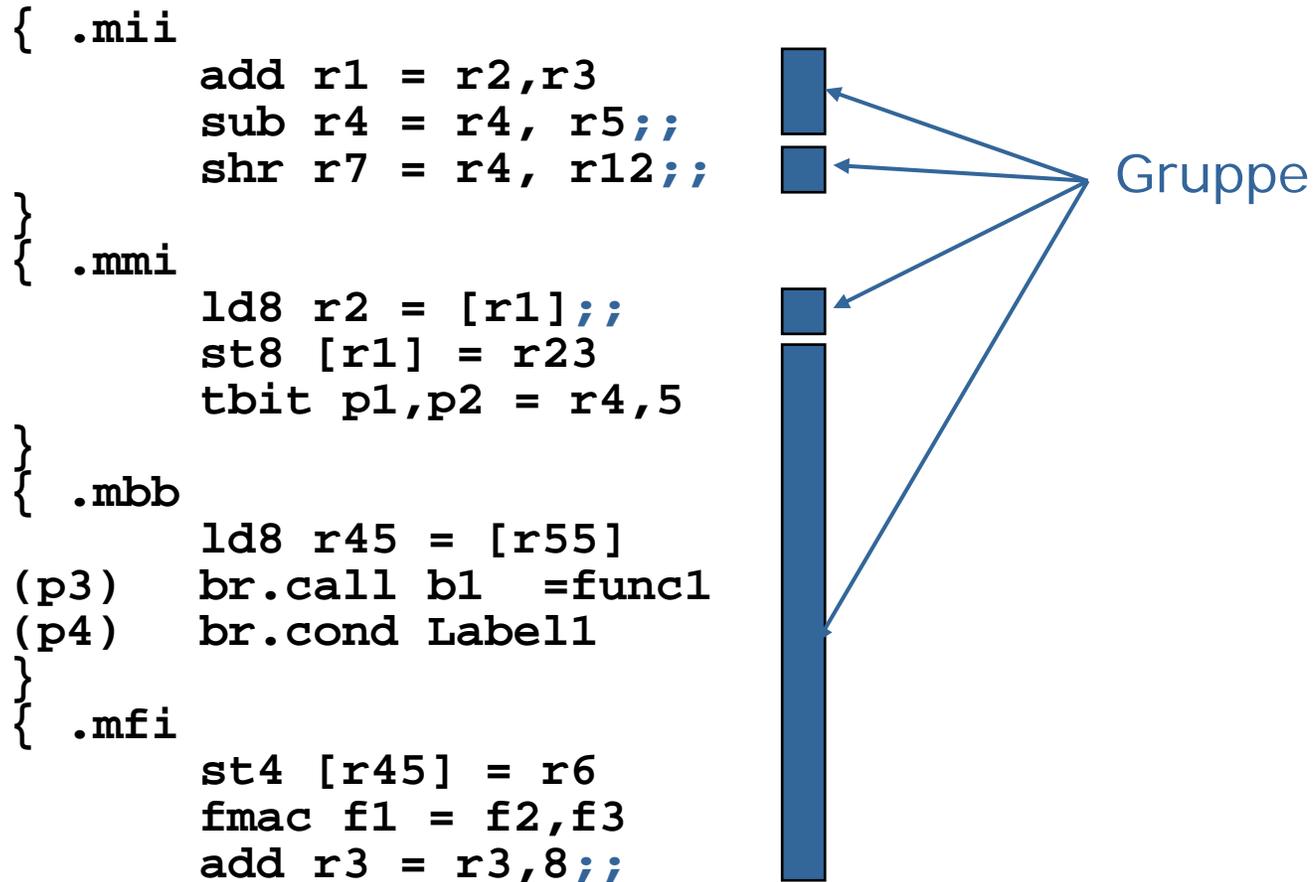
- IA-64 ISA

- Bundles:

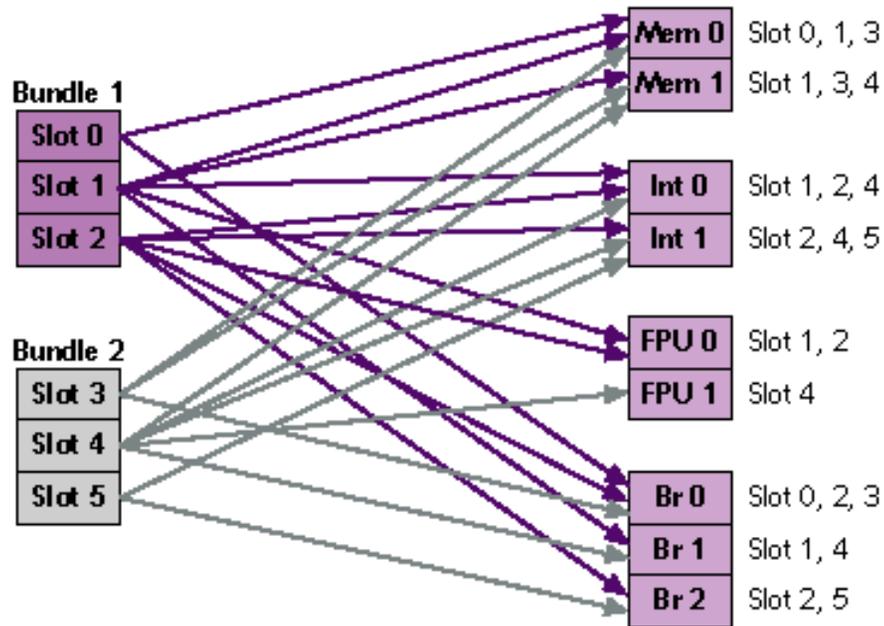
- Template: zeigt explizit an, ob
  - die Instruktionen im Bundle gleichzeitig ausgeführt werden dürfen, oder
  - eine oder mehrere Instruktionen sequentiell auszuführen sind, oder
  - benachbarte Bundles parallel ausgeführt werden können.



- Beispiele von Befehlsgruppen:



- Anstoßen der Befehle zur Ausführung
  - Beispiel: Itanium



- Es können bis zu sechs Instruktionen pro Takt zur Ausführung angestoßen werden. Die Instruktionen kommen von zwei Bundles. Jeder Bundle hat 3 Slots.

- IA-64: Skalierbarkeit

- Jedes Bundle enthält drei Instruktionen für eine Menge von drei Funktionseinheiten.
- Ein IA-64 Prozessor kann  $n$  Mengen von jeweils drei Funktionseinheiten haben, welche die Informationen im Template nutzen, dann können mehrere Bundles in ein Instruktionswort mit der Länge  $n$  Bundles gepackt werden.
- Skalierbarkeit bezüglich der Anzahl der Funktionseinheiten



- Predication, bedingte Befehlsausführung
  - Beispiel:

### Bedingte Befehlsfolge

```

if (r1 == r2)
    r9 = r10 - r11;
else
    r5 = r6 + r7;

```

(p1)	<code>cmp.eq p1,p2 = r1, r2;;</code>
(p2)	<code>sub r9 = r10, r11</code>
	<code>add r5 = r6, r7</code>

- Evaluierung der bedingten Ausdrücke mit Hilfe von Compare-Operationen.
- Jeder Befehl hat ein 6 Bit breites Predicate Feld zur Angabe des Predicate-Registers
- Elimination von Sprüngen



- **Predication (Bedingte Befehlsausführung)**
  - Zur Laufzeit werden die voneinander unabhängigen Befehle zur Ausführung angestoßen.
  - Der Prozessor führt die Befehle auf den möglichen Programmverzweigungen aus, aber speichert die Ergebnisse nicht endgültig.
  - Überprüfen der Predicate Register
    - Falls das Register eine 1 enthält, dann ist wird die Ausführung der Instruktion abgeschlossen.
    - Falls das Register eine 0 enthält, dann wird das Ergebnis verworfen.

# • IA-64 Control Speculation

- Problem: Verzweigungen schränken Code-Verschiebungen ein

```
instA
```

```
instB
```

```
...
```

```
br
```

Grenze

```
ld8 r1 =[r2]
```

```
use r1
```

Ladeoperation kann nicht über den Sprung verschoben werden, denn es könnten Alarme ausgelöst werden.

# • IA-64 Control Speculation

- Lösung: Spekulative Operationen, die keine Alarme auslösen

```
instA
instB
...
br
```

Grenze

---

```
ld8 r1 =[r2]
use r1
```

```
ld8.s r1 =[r2]
use r1
instA
instB
...
br
```

Spekulative Ladeoperation

---

```
chk.s ← Speculation Check
```



## • IA-64 Control Speculation

- Einführung von spekulativen Ladeoperationen
  - Verursachen keinen Alarm:
    - Falls zur Laufzeit die Operation einen Alarm auslöst, dann wird dieser Alarm verzögert
    - Setzen von NaT-Bit (Deferred Exception Token, Not-a-Thing Bit) in Zielregister
  - Spekulative Ladeoperation kann über Verzweigungen hinweg verschoben werden
  - Einfügen von Speculation Check Instruktion (chk.s) anstelle der Ladeoperation
    - Zur Laufzeit überprüft die Check-Operation das Zielregister, ob NaT gesetzt ist. Wenn ja, dann erfolgt eine Verzweigung zu einem speziellen Fix-up-Code.



## • IA-64 Data Speculation

- Problem: Zeiger können Compiler zu konservativen Annahmen über die Referenzen zwingen, was eine Code-Verschiebung verhindert.

```
instA  
instB  
...  
store
```

---

```
ld8 r1 =[r2]  
use r1
```

Ladeoperation kann nicht über den die Speicheroperation verschoben werden, da beide dieselbe Adresse referenzieren können.



# • IA-64 Data Speculation

– Lösung: Vorgezogene Ladeoperationen

```
instA
instB
...
store
```

---

```
ld8 r1 =[r2]
use r1
```

← Vorgezogene Ladeoperation

```
ld8.a r1 =[r2]
```

```
use r1
instA
instB
...
store
```

---

```
chk.a
```

Speculation Check



## • IA-64 Data Speculation

### – Lösung: Vorgezogene Ladeoperationen

- Verschieben von Lade-Operationen auch vor Speicher-Operationen.
  - Vorgezogene Lade-Operation (ld.a)
  - Zur Laufzeit werden Informationen (Zielregister, Speicheradresse, auf die zugegriffen wird, Zugriffsgröße) in **Advanced Load Address Table (ALAT)** festgehalten.
  - Zur Laufzeit prüft die Hardware, wenn eine Speicheroperation ausgeführt wird, ob eine Adresse in der ALAT mit der Zugriffsadresse übereinstimmt. Wenn ja, dann wird der Eintrag in ALAT gelöscht



# • IA-64 Data Speculation

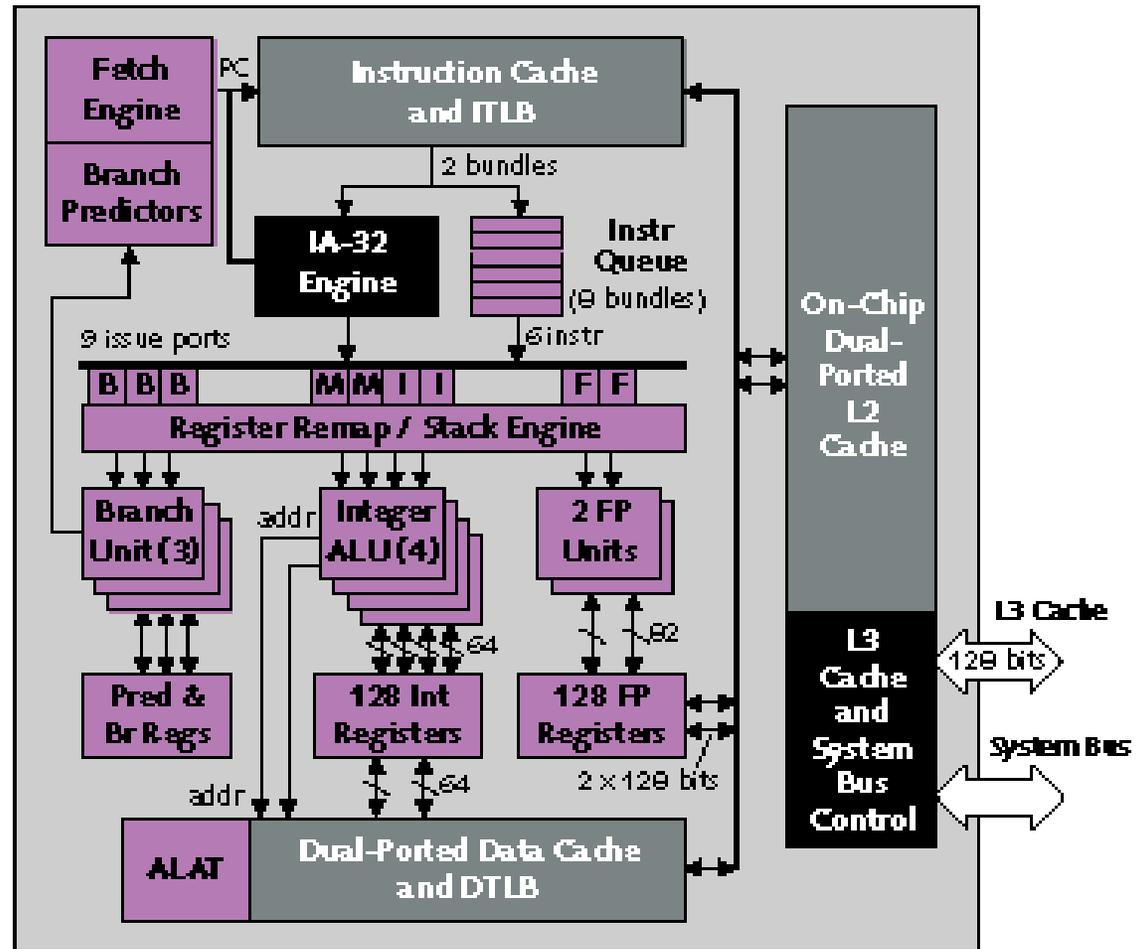
– Lösung: Vorgezogene Ladeoperationen

- Einfügen der Check-Operation (`chk.a`)

- Prüft, ob Eintrag von der entsprechenden vorgezogenen Ladeoperation in ALAT steht.
- Wenn nicht, dann hat es eine Kollision mit einer Speicheroperation gegeben und es erfolgt eine Verzweigung zu einem Fix-up-Code.



- Intel Itanium



Quelle: Microprocessor Report, Vol.13, Nr.13, October 5, 1999



- Intel Itanium

- 64-Bit Prozessor der P7 Generation
- EPIC
- 10-stufige Pipeline mit 6 Befehlen, die gleichzeitig zur Ausführung angestoßen werden können.
- 128 GP- und 128 FP-Register - keine Registerumbenennung: rotierende Registerfenster
- Bedingte Befehlsausführung: 64 1 Bit Predicate Register
- 9 Funktionseinheiten
- Misprediction Penalty: 9 Zyklen



## • Intel Itanium

- Spekulative Ladeoperationen
- 4-fach mengenassoziative L1 Befehls- und Daten-Cache-Speicher (Write-through)
- 6-fach mengenassoziativer L2 Cache (Copy-back)
- Informationen
  - Intel: <http://www.intel.com/>
    - Intel Technology Journal:  
<http://developer.intel.com/technology/itj>
  - IEEE Micro: Sonderheft im Spe./Okt. 2000
  - Henn./Patt.: Computer Architecture: Kap. 4.7



- Literatur:
  - Brinkschulte/Ungerer: Mikrocontroller und Mikroprozessoren. Springer-Verlag, 2002: [Kap. 6.1-6.4, Kap. 7](#)
  - Hennessy/Patterson: Computer Architecture – A Quantative Approach. 3. Auflage: [Kap. 3](#)