

Technische Informatik 1 – Übung 9

Instruktionsparallelität (Rechenübung)

Marco Zimmerling
8.-9. Dezember 2011



Ziele der Übung

- **Verschiedene Arten von Instruktionsparallelität kennen lernen**
 - VLIW
 - (Super)pipelining
 - Superskalare Prozessoren
- **VLIW genauer verstehen**
 - Code umschreiben
 - Einfluss auf Codegrösse sowie Ausführzeit
 - VLIW in Kombination mit Pipelining

VLIW – Very Long Instruction Word

- ***Ein VLIW beinhaltet n Instruktionen***

- Gleichzeitiger Start und Verarbeitung

VLIW Inst. 1	Instr. 1	Instr. 2	...	Instr. n
VLIW Inst. 2	Instr. 1	Instr. 2	...	Instr. n

- ***Instruktionen müssen unabhängig sein***

- Gleichzeitig ausgeführte Instruktionen dürfen nicht die gleichen Ressourcen benutzen
- Gleichzeitig ausgeführte Instruktionen dürfen nicht das gleiche Zielregister aufweisen

- ***Es werden zusätzliche Ressourcen benötigt***

- z.B. zusätzliche ALUs

VLIW auf MIPS Architektur

- **Instruktionen werden paarweise geladen, dekodiert und ausgeführt**
 - Die 1. Instruktion ist eine ALU Op oder Verzweigung
 - Die 2. Instruktion ist ein Speicherzugriff
 - Es können NICHT gleichzeitig zwei ALU oder zwei Speicheroperationen durchgeführt werden
 - Neue Instruktion ist 64 bit breit

Instruktion 1 (32 bit)	Instruktion 2 (32 bit)
ALU Op oder Verzweigung	Datentransfer (load oder store)

- **Ausführung von nur einer Instruktion möglich**
 - *nop* Operation für die zweite Instruktion

Korrekte Anordnung der Instruktionen

■ Umschreiben für VLIW (ohne Pipelining)

- addi \$t1, \$zero, 0
- lw \$t2, 0(\$t1)
- addi \$t1, \$zero, 4
- lw \$t3, 0(\$t1)
- or \$t2, \$t2, \$t3

↗ **Achtung: Read after Write**

↖ **Achtung: Write after Read**

	Instruktion 1 (32 bit)	Instruktion 2 (32 bit)	CC
main:	addi \$t1, \$zero, 0		1
	addi \$t1, \$zero, 4	lw \$t2, 0(\$t1)	2
		lw \$t3, 0(\$t1)	3
	or \$t2, \$t2, \$t3		4

- Cycles per Instruktion: $CPI = 4/5 = 0.8$ (optimal 0.5)

Aufgabe 1.1

- **Umschreiben nach VLIW**
 - 1 . **Priorität: Codegrösse minimieren**
 - 2 . **Priorität: Laufzeit minimieren**
- **Kein Pipelining**
 - **Instruktionen werden nacheinander ausgeführt**



Label	ALU oder Verzweigung	Datentransfer	CC
main:			1
			2

Tabelle 1: Vorlage für die Darstellung von MIPS VLIW Instruktionen.

Besprechung Aufgabe 1.1

```

loop:  LW    t1, 0(s1)
       ADDI  t1, t1, 1
       SW    t1, 0(s1)
       ADDI  s1, s1, 4
       SUBI  t3, t3, 1
       BNEZ  t3, loop
  
```

 **Achtung: Read after Write**
 **Achtung: Write after Read**

	ALU oder Verzweigung	Datentransfer	CC
main:	SUBI t3, t3, 1	LW t1, 0(s1)	1
	ADDI t1, t1, 1		2
	ADDI s1, s1, 4	SW t1, 0(s1)	3
	BNEZ t3, loop		4

Lösung Aufgabe 1.1 - Codegrösse

Label	ALU oder Verzweigung	Datentransfer	CC
main:	ADDI s1, zero, 0		1
	ADDI t7, zero, 12		2
	ADDI t7, zero, 13	SW t7, 0(s1)	3
	ADDI t7, zero, 14	SW t7, 4(s1)	4
	ADDI t7, zero, 15	SW t7, 8(s1)	5
	ADDI t3, zero, 4	SW t7, 12(s1)	6
loop:	SUBI t3, t3, 1	LW t1, 0(s1)	7
	ADDI t1, t1, 1		8
	ADDI s1, s1, 4	SW t1, 0(s1)	9
	BNEZ t3, loop		10

VLIW: 10 Instruktionen an 64 bit -> 640 bit

Originalcode: 16 Instruktionen an 32 bit -> 512 bit

Es wird ein Overhead von 20% generiert.

Lösung Aufgabe 1.1 - Laufzeit

Label	ALU oder Verzweigung	Datentransfer	CC
main:	ADDI s1, zero, 0		1
	ADDI t7, zero, 12		2
	ADDI t7, zero, 13	SW t7, 0(s1)	3
	ADDI t7, zero, 14	SW t7, 4(s1)	4
	ADDI t7, zero, 15	SW t7, 8(s1)	5
	ADDI t3, zero, 4	SW t7, 12(s1)	6
loop:	SUBI t3, t3, 1	LW t1, 0(s1)	7
	ADDI t1, t1, 1		8
	ADDI s1, s1, 4	SW t1, 0(s1)	9
	BNEZ t3, loop		10

Vor der Schleife: 10 Instruktionen in 6 Taktzyklen

In der Schleife: 6 Instruktionen in 4 Taktzyklen (4 Mal ausgeführt)

-> Total: $10 + 4 \times 6 = 34$ Instruktionen in $6 + 4 \times 4 = 22$ Taktzyklen

CPI = $22/34 = 0.647$, Ausführungszeit: $22 \times 10 \text{ ns} = 220 \text{ ns}$

Lösung Aufgabe 1.2

Label	ALU oder Verzweigung	Datentransfer	CC
main:	ADDI s1, zero, 0		1
	ADDI t7, zero, 12		2
	ADDI t7, zero, 13	SW t7, 0(s1)	3
	ADDI t7, zero, 14	SW t7, 4(s1)	4
	ADDI t7, zero, 15	SW t7, 8(s1)	5
		SW t7, 12(s1)	6
		LW t1, 0(s1)	7
	ADDI t1, t1, 1	LW t3, 4(s1)	8
	ADDI t3, t3, 1	SW t1, 0(s1)	9
		LW t1, 8(s1)	10
	ADDI t1, t1, 1	SW t3, 4(s1)	11
		LW t3, 12(s1)	12
	ADDI t3, t3, 1	SW t1, 8(s1)	13
		SW t3, 12(s1)	14

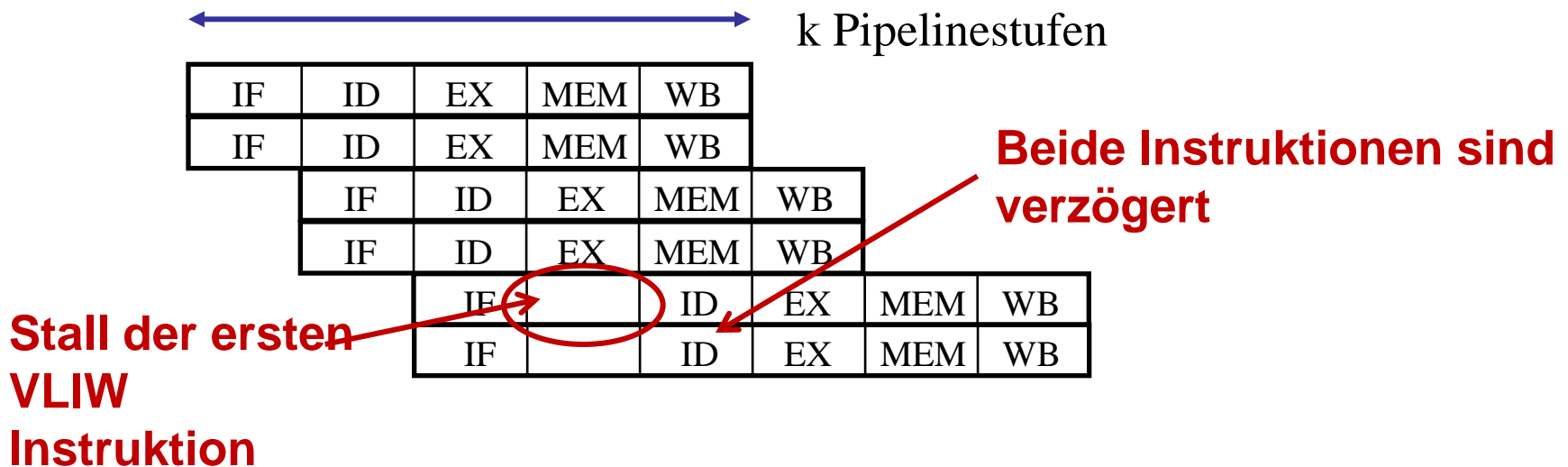
14 Instruktionen an 64 bit -> 896 bit

CPI = 14/21 = 0.666, Ausführungszeit = 140 ns

Reduktion der Ausführungszeit um 36%. Code wird 29% grösser.

VLIW mit Pipelining

- Die beiden VLIW Instruktionen werden auf jeder Stufe gleichzeitig bearbeitet
 - Wenn eine Instruktion gestallt werden muss, wird die zweite Instruktion des VLIW ebenfalls angehalten



Aufgabe 2: Pipelining mit VLIW

Funktion `minVec(int* A0, int* A1, int* A2, int n)`

Input: Arrays `A0[1:n]` und `A1[1:n]`

Output: `A2[i] = min(A0[i];A1[i])` für alle `i=[1:n]`

Label	ALU oder Verzweigung	Datentransfer	CC
<code>minVec:</code>	<code>addi a0,a0,4</code>	<code>lw t0,0(a0)</code>	1
	<code>addi a1,a1,4</code>	<code>lw t1,0(a1)</code>	2
	<code>slt t2,t0,t1</code>		3
	<code>bne t2,zero,l1</code>		4
	<code>addi t0,t1,0</code>		5
<code>l1:</code>	<code>addi a3,a3,-1</code>		6
	<code>addi a2,a2,4</code>	<code>sw t0,0(a2)</code>	7
	<code>bne a3,zero,minVec</code>		8
	<code>jr ra</code>		9

Hinweis: `slt t2,t0,t1` \Rightarrow Setze `t2=1` falls `t0 < t1`; ansonsten `t2=0` .

Aufgabe 2: Pipelining mit VLIW

■ Tipps

- Bestimmen sie ob gesprungen wird oder nicht
- Der Prozessor benutzt die statische Sprungvorhersage not taken
- Sprungentscheidung ist nach der MEM Stufe bekannt
- Forwarding nur von EX->EX und MEM->EX

Label	ALU oder Verzweigung	Datentransfer	CC
minVec:	addi a0,a0,4	lw t0,0(a0)	1
	addi a1,a1,4	lw t1,0(a1)	2
	slt t2,t0,t1		3
	bne t2,zero,l1		4
	addi t0,t1,0		5
l1:	addi a3,a3,-1		6
	addi a2,a2,4	sw t0,0(a2)	7
	bne a3,zero,minVec		8
	jr ra		9

Hinweis: `slt t2,t0,t1` ⇒ Setze `t2=1` falls `t0 < t1`; ansonsten `t2=0` .

Lösung Aufgabe 2.a

	1	2	3	4	5	6	7	8	9	10	11	12	13
minVec: addi a0,a0,4 lw t0,0(a0)	IF	ID	EX	MEM	WB								
	IF	ID	EX	MEM	WB								
addi a1,a1,4 lw t1,0(a1)		IF	ID	EX	MEM	WB							
		IF	ID	EX	MEM	WB							
slt t2,t0,t1			IF	-	-	ID	EX	MEM	WB				
bne t2,zero,l1						IF	-	-	ID	EX	MEM	WB	
addi t0,t1,0									IF	ID	EX	MEM	WB
l1: addi a3,a3,-1										IF	ID	EX	MEM
addi a2,a2,4 sw t0,0(a2)											IF	-	ID
											IF	-	ID
bne a3,zero,minVec													IF
jr ra													

Beide Instruktionen werden angehalten, obwohl nur die zweite blockiert ist

Lösung Aufgabe 2.b

	1	2	3	4	5	6	7	8	9	10	11
minVec: addi a0,a0,4 lw t0,0(a0)	IF	ID	EX	MEM	WB						
	IF	ID	EX	MEM	WB						
addi a1,a1,4 lw t1,0(a1)		IF	ID	EX	MEM	WB					
		IF	ID	EX	MEM↓	WB					
slt t2,t0,t1			IF	-	ID	↑EX↓	MEM	WB			
bne t2,zero,l1					IF	ID	↑EX	MEM	WB		
addi t0,t1,0						IF	ID	EX			
l1: addi a3,a3,-1							IF	ID	IF	ID	EX
addi a2,a2,4 sw t0,0(a2)								IF		IF	ID
								IF		IF	ID
bne a3,zero,minVec											IF
jr ra											

Falsche Sprungvorhersage. Die komplette Pipeline muss geflusht werden

Lösung Aufgabe 2.c

■ Eingabedaten

- $A0 = [5, 4, 12, 7, 15]$
- $A1 = [7, 4, 9, 13, 0]$

■ Statische Sprungvorhersage: “not taken”

■ Fünf Schleifendurchläufe

- Erster Sprung: “taken” 1. und 4. Durchlauf
- Zweiter Sprung: “taken” 1., 2., 3. und 4. Durchlauf

■ Die Pipeline muss $4 + 2 = 6$ Mal geflusht werden, es gehen $6 \times 3 = 18$ Zyklen verloren

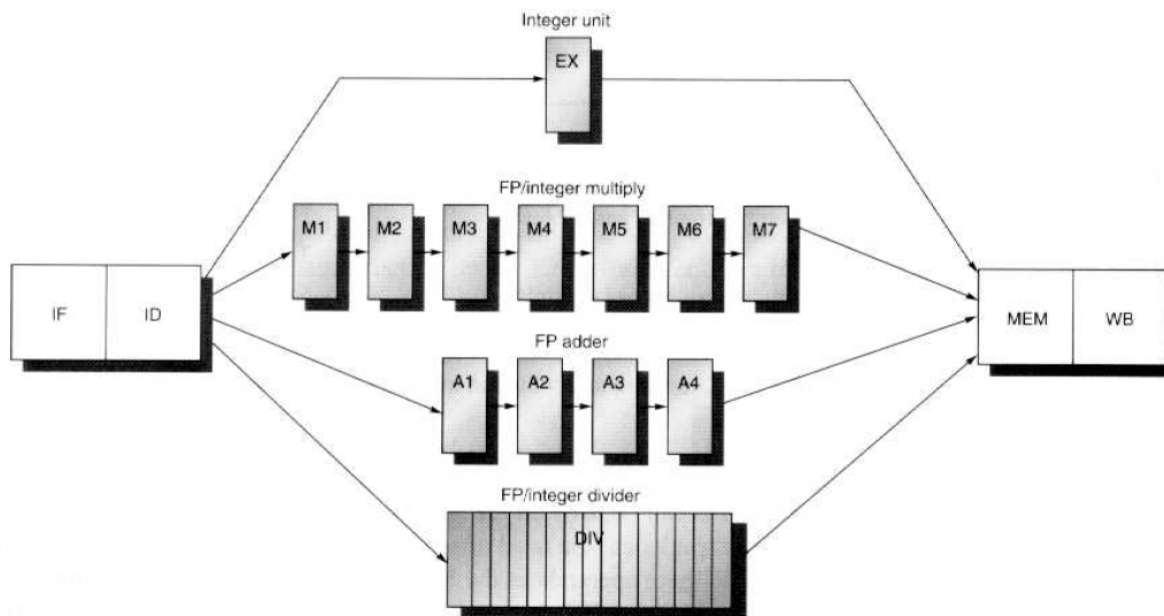


Diskussion über Instruktionsparallelität

- **In der Vorlesung wurden 4 Systeme für Instruktionsparallelität vorgestellt:**
 - Pipelining mit fester Anzahl Stufen: IF ID EX MEM WB
 - Very Long Instruction Word (VLIW)
 - Superpipelining
 - Superskalare Prozessoren

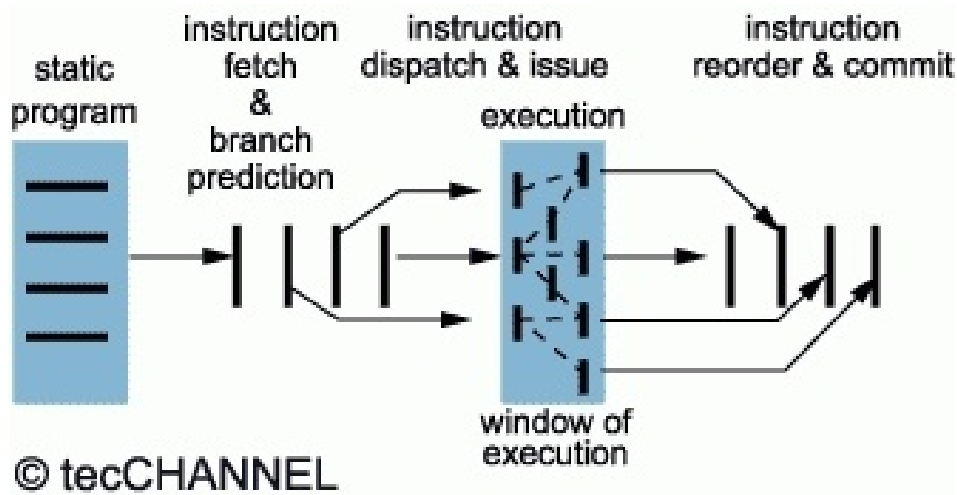
Superpipelining

- Einführung zahlreicher Pipelinestufen, vor allem bei den arithmetischen Einheiten.
 - Die Zahl der Stufen ist je nach Instruktion unterschiedlich. Führt zu „*out of order completion*“.



Superskalare Prozessoren

- **Der Prozessor bestimmt zur Laufzeit welche Instruktion ausgeführt werden soll**
 - *Eine spezielle Einheit (Commit unit) stellt dabei sicher dass die Resultate in der richtige Reihenfolge in die Register geschrieben werden*





Superpipelining vs. VLIW vs. Superskalar

■ Einfluss auf Codegrösse?

- Bei VLIW werden die Instruktionen in Blöcke von mehreren Instruktionen gefasst. Es kommt dabei öfters vor, dass nop Befehle eingeführt werden müssen.
- Superpipelining und superskalare Prozessoren benötigen keine nops



Superpipelining vs. VLIW vs. Superskalar

■ Einfluss auf Compiler?

- Bei VLIW und superpipelining ist man auf gute Compiler angewiesen um stalls zu verhindern.
- Superskalare Prozessoren 'optimieren' die Ausführung auf Hardware Ebene. Jedoch kann auch hier ein Compiler Helfen um zum Beispiel ein Loop unrolling erkennen zu können, was man Hardware nur schwer machbar ist.



Superpipelining vs. VLIW vs. Superskalar

■ Einfluss auf Logikaufwand?

- Für VLIW müssen gewisse Logikkomponenten dupliziert werden.
- Das verfeinerte Aufteilen der Pipeline für Superpipelining resultiert in einem relativ kleinen Mehraufwand (es werden zusätzliche Register für die Zwischenstufen benötigt).
- Bei Superskalaren Prozessoren ist der Zusatzaufwand (für das Controlling) enorm.



Superpipelining vs. VLIW vs. Superskalar

■ Einfluss auf Branch Misses?

- Für VLIW ist eine akkurate Branch Prediction weniger wichtig (da kurze Pipeline). Insbesondere wenn ein Branch Delay Slot eingeführt wird.
- Bei Superpipelining und Superskalaren Prozessoren muss eventuell eine Vielzahl von Instruktionen aus der Pipeline gelöscht werden.
 - Der Intel Pentium 4 besitzt 20 Stages.
 - Beim AMD Opteron X4 können bis zu 106 RISC-Operationen gleichzeitig in Verarbeitung sein.



Superpipelining vs. VLIW vs. Superskalar

- **Was wird heute eingesetzt (Hochleistungs CPUs)?**
 - Hochleistungsprozessoren sind normalerweise superskalare Systeme mit langen Pipelines.
 - Die Pipeline ist dabei je nach Befehl unterschiedlich lang: für eine ganzzahlige Addition ist die EX Stufe in viel weniger Unterstufen unterteilt als für eine Gleitkomma Multiplikation oder gar Division.
 - Superskalaren Systeme mit unterschiedlich langer Pipeline bedingen eine sehr komplexe Kontrolllogik, die nicht mehr einfach 'von Hand, entworfen werden kann.



Diskussion über Instruktionsparallelität

- Um 1990 wurde der **Branch Delay Slot** eingeführt um Ablauf Hazards zu vermeiden. Wieso wird dieses damals erfolgreiche Konzept heute nicht mehr benutzt?
 - Anno 1990 war die zusätzliche Logik für die **dynamische Branch Prediction** zu teuer. Da zwischenzeitlich die Anzahl Transistoren in einer CPU exponentiell gewachsen sind, kann man sich den Zusatzaufwand problemlos leisten.
 - Insbesondere bei langen Pipelines sowie auch bei superskalaren Systemen ist der Branch Delay Slot überflüssig geworden.