

Prof. L. Thiele

## Technische Informatik 1 - HS 2011

---

### Lösungsvorschläge für Übung 9

Datum: 8.12.2011

---

## 1 Instruktionsparallelität – VLIW

Gegeben sei folgendes Programm für den MIPS-Prozessor (ähnlich dem Programm in Übung 7). Es initialisiert ein Array mit den Werten [12, 13, 14, 15] und erhöht diese Werte anschliessend um 1. Das Register `s1` zeigt auf die Basisadresse des Arrays (im folgenden Programm wird die Basisadresse 0 angenommen). Das Register `zero` ist fest mit 0 verdrahtet.

```
main:  ADDI    s1, zero, 0
        ADDI    t7, zero, 12
        SW     t7, 0(s1)
        ADDI    t7, zero, 13
        SW     t7, 4(s1)
        ADDI    t7, zero, 14
        SW     t7, 8(s1)
        ADDI    t7, zero, 15
        SW     t7, 12(s1)
        ADDI    t3, zero, 4
loop:  LW     t1, 0(s1)
        ADDI    t1, t1, 1
        SW     t1, 0(s1)
        ADDI    s1, s1, 4
        SUBI    t3, t3, 1
        BNEZ   t3, loop
```

### 1.1 Minimieren der Codegrösse

Das Programm soll nun für den MIPS VLIW (Very Long Instruction Word) Instruktionssatz umgeschrieben werden. Dabei soll die Codegrösse minimiert werden. Der Prozessor ist mit 100 MHz getaktet und benutzt keine Pipeline, d.h. der Prozessor beendet eine VLIW Instruktion bevor die nächste ausgeführt wird.

- Das Programm soll für den MIPS VLIW Instruktionssatz umgeschrieben werden.
- Wie viel Platz benötigt der VLIW Code im Speicher? Vergleichen Sie mit dem Platzbedarf des ursprünglichen MIPS Codes.
- Wie viele Taktzyklen werden pro Instruktion (CPI) im Durchschnitt benötigt? Wie lange dauert die Ausführung des Programms?

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

Tabelle 1: Lösung für Aufgabe 1.1.

**Lösung:**

- (a) Siehe Tabelle 1.
- (b) Der ursprüngliche MIPS Code benötigte 16 Instruktionen an 32 Bits, also 512 Bits. Der VLIW-MIPS Code benötigt 10 Instruktion an 64 Bits, also 640 Bits. Es wird also ein Overhead von 20% generiert.
- (c) Es muss beachtet werden, dass die Schleife vier Mal ausgeführt wird. Gemäss Tabelle 1 werden vor der Schleife 10 Instruktionen in 6 Taktzyklen abgearbeitet. In jedem Schleifendurchlauf sind es 6 Instruktionen in 4 Zyklen. Insgesamt sind es also 34 Instruktionen ( $10 + 4 \cdot 6$ ) in 22 Cycles ( $6 + 4 \cdot 4$ ). Der CPI beträgt also  $22/34 = 0.65$ . Für die Ausführungszeit muss man zuerst die Dauer eines Cycles berechnen. Bei einem 100 MHz Prozessor sind dies 10 ns. Die Ausführungszeit ist also  $22 \cdot 10 = 220$  ns.

**1.2 Minimieren der Ausführungszeit (Zusatzaufgabe)**

Das Programm soll wiederum für den MIPS VLIW Instruktionssatz umgeschrieben werden. Es wird der gleiche Prozessor wie in der vorhergehenden Teilaufgabe verwendet. Dabei soll die Ausführungszeit minimiert werden. Tipp: Verwenden Sie Schleifenentfaltung (loop unrolling).

- (a) Passen Sie den Code entsprechend an. Zusätzlich soll die Anzahl der benötigten Register für die Schleifenentfaltung minimiert werden.
- (b) Wie viel Platz beansprucht der Code im Speicher?
- (c) Wie viele Taktzyklen werden pro Instruktion (CPI) im Durchschnitt benötigt? Wie lange dauert die Ausführung des Programms?
- (d) Vergleichen sie die Codegrösse, die Ausführungszeit und den CPI der beiden Versionen (Aufgabe 1.1 und 1.2) des Programmes.

**Lösung:**

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

Tabelle 2: Lösung für Aufgabe 1.2.

- (a) Siehe Tabelle 2. Die Befehle um einen Wert des Arrays zu lesen LW, inkrementieren ADDI und zurückschreiben SW kann nicht direkt parallelisiert werden. Es können aber verschiedene Iterationen der Schleife kombiniert werden. Dafür wird das frei gewordene Register t3 verwendet (es wird ja keine Schleife mehr gezählt).
- (b) Der entfaltete Code MIPS Code benötigte 14 Instruktionen an 64 Bits, also 896 Bits.
- (c) Es werden 21 Instruktionen in 14 Cycles ausgeführt.  $CPI = 14/21 = 0.66$ . Die Ausführungszeit ist 140 ns.
- (d) Der CPI des entfalten Codes ist minimal grösser (0.66 vs. 0.65). Die Ausführungszeit des entfalten Codes ist aber dennoch markant reduziert (14 vs. 22 Cycles). Dies entspricht einer Reduktion der Ausführungszeit von 36%. Dafür wird der Code 256 Bits (29%) grösser.

## 2 Pipelining mit VLIW

Gegeben sei ein VLIW MIPS Prozessor welcher mit einer 5-stufigen Pipeline (IF ID EX MEM WB) arbeitet. Insbesondere wird bei Verzweigungen mit der (für MIPS typischen) statischen Sprungvorhersage not taken gearbeitet. Die Sprungentscheidung ist nach der MEM Stufe bekannt. Es gibt keinen Branch Delay Slot.

Auf dem Prozessor wird die folgende Funktion minVec ausgeführt:

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

Die Funktion  $\text{minVec}(\text{int}^* A_0, \text{int}^* A_1, \text{int}^* A_2, \text{int } n)$  überführt die beiden Arrays  $A_0$  und  $A_1$  der Grösse  $n$  in ein Array  $A_2$  derselben Grösse. Für alle Elemente  $i$  des neuen Arrays gilt:  $A_2[i] = \min(A_0[i], A_1[i])$ . Die Startadressen der drei Arrays ( $A_0, A_1, A_2$ ) werden in den Registern  $a_0, a_1$  und  $a_2$  übergeben. Die Anzahl Elemente  $n$  des Arrays steht im Register  $a_3$ .

- (a) Die Pipelining-Architektur unterstützt kein Forwarding. Simulieren Sie das Pipeline-Verhalten der Funktion, wenn diese für die Arrays  $A_0 = [8]$  und  $A_1 = [5]$  ausgeführt wird (d.h.  $n = 1$ ). Markieren sie gegebenenfalls Stellen wo ein Leeren (Flush) der Pipeline wegen einer falschen Sprungvorhersage nötig ist.

**Lösung:**

Siehe Tabelle 3. Es gibt keine falschen Sprungvorhersagen; die Pipeline muss also nie geleert werden.

Insbesondere muss die `sw` Anweisung gestallt werden um auf den Inhalt von Register `t0` zu warten. Dadurch muss die zur `sw` parallel ausgeführte `addi` Anweisung ebenfalls angehalten werden: die beiden Instruktionen einer VLIW Anweisung müssen sich immer auf der gleichen Pipelinestufe befinden.

- (b) Die Pipelining-Architektur unterstützt nun Forwarding von EX nach EX sowie von MEM nach EX. Simulieren Sie das Pipeline-Verhalten der Funktion, wenn diese für die Arrays  $A_0 = [4]$  und  $A_1 = [9]$  ausgeführt wird. Markieren Sie gegebenenfalls Stellen wo ein Leeren (Flush) der Pipeline nötig ist. Kennzeichnen Sie im Diagramm wo der Forwarding-Mechanismus zum Tragen kommt, indem Sie Pfeile zwischen den entsprechenden Pipeline-Stufen der abhängigen Instruktionen eintragen.

**Lösung:**

Siehe Tabelle 4. Bei der ersten Sprunganweisung (Zeile 4) ist die Sprungvorhersage falsch. Dadurch muss Pipeline nach dem 8. Zyklus geleert werden.

Trotz des Forwardings tritt bei der `slt` Anweisung immer noch ein Stall auf. Dabei wird die Pipeline bereits nach der IF Stufe angehalten. Der Grund ist, dass die EX Stufe der `slt` Anweisung den Inhalt des Registers `t0` benötigt, und es kein Forwarding von der WB zur EX Stufe gibt. Das Register `t0` muss deshalb über die ID Stufe geladen werden.

- (c) Die Pipeline-Architektur unterstützt weiterhin Forwarding. Die Funktion wird nun mit den Werten  $A_0 = [5, 4, 12, 7, 15]$  und  $A_1 = [7, 4, 9, 13, 0]$  ausgeführt. Wie oft muss die Pipeline geleert werden? Wie viele Zyklen gehen dadurch verloren?

**Lösung:**

Die Schleife wird 5 mal durchlaufen. Der erste Sprung wird 2 Mal ausgeführt (beim ersten und vierten Schleifendurchlauf). Der zweite Sprung wird in den ersten 4 Durchläufen ausgeführt. Total werden also  $2 + 4 = 6$  Verzweigungen falsch vorhergesagt. Jedes Mal muss dabei die Pipeline geleert werden. Es gehen dabei  $6 \cdot 3 = 18$  Zyklen verloren.

### 3 Diskussion über Instruktionsparallelität

- (a) Um 1990 wurde der *Branch Delay Slot* eingeführt um Ablauf Hazards zu vermeiden. Wieso wird dieses damals erfolgreiche Konzept heute nicht mehr benutzt?

**Lösung:**

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.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
minVec: addi a0,a0,4	IF	ID	EX	MEM	WB													
lw t0,0(a0)	IF	ID	EX	MEM	WB													
addi a1,a1,4		IF	ID	EX	MEM	WB												
lw t1,0(a1)		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	WB				
addi a2,a2,4											IF	-	ID	EX	MEM	WB		
sw t0,0(a2)											IF	-	ID	EX	MEM	WB		
bne a3,zero,minVec													IF	ID	EX	MEM	WB	
jr ra														IF	ID	EX	MEM	WB

Tabelle 3: Lösung für VLIW Pipelining ohne forwarding.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
minVec: addi a0,a0,4	IF	ID	EX	MEM	WB											
lw t0,0(a0)	IF	ID	EX	MEM	WB											
addi a1,a1,4		IF	ID	EX	MEM	WB										
lw t1,0(a1)		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	MEM↓	WB			
addi a2,a2,4								IF		IF	ID	EX	MEM	WB		
sw t0,0(a2)								IF		IF	ID	EX	MEM	WB		
bne a3,zero,minVec										IF	ID	↑EX	MEM	WB		
jr ra											IF	ID	EX	MEM	WB	

Tabelle 4: Lösung für VLIW Pipelining mit forwarding. Zwischen Zyklus 8 und 9 muss die Pipeline geleert werden.

(b) In der Vorlesung wurden drei verschiedene Ansätze zur Optimierung der Ausführungszeit von Programmen vorgestellt: Superpipelining, VLIW Prozessoren sowie (dynamische) Superskalare Prozessoren. Was sind die Vor- und Nachteile der Ansätze im Bezug auf folgende Eigenschaften?

- Codegröße
- Compilerbau
- Logikaufbau
- Einfluss von Branch Misses
- Variable Instruktionslänge

Was für eine Architektur wird heutzutage für leistungsstarke CPUs gewählt?

**Lösung:**

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

**Compiler** Bei superskalaren Prozessoren gibt es dedizierte Hardware, welche die Instruktionen zur Laufzeit umsortiert um Stalls zu vermeiden. Es ist also weniger wichtig in welcher Reihenfolge die Befehle durch den Compiler geordnet werden. Die Hardware hat aber eine relativ beschränkte Sicht des Programmes. So ist zum Beispiel die Möglichkeit für ein Loop Unrolling in Hardware schwierig zu erkennen. Ein intelligenter Compiler erlaubt also auch bei superskalaren Systemen eine verbesserte Performance.

Die Performanz (Ausführungszeit) von Programmen auf VLIW und Superpipelining Prozessoren hängt sehr stark von der Anpassung des verwendeten Compilers an die Architektur ab. Beim Einsatz von schlecht angepassten Compilern kann es zu einer sehr schlechten Performanz kommen. So war es beispielsweise zu Beginn ein Problem die theoretisch erreichbare Performanz des Intel Itanium 2 (VLIW) Prozessors in realen Anwendungen annähernd zu erreichen, da es extrem schwierig ist, gute Compiler zu schreiben.

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

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

**Variable Instruktionslänge** Eine feste Instruktionsbreite (z.B. 32 Bit bei MIPS) vereinfacht das gleichzeitige Lesen von mehreren Befehlen bei VLIW und Superskalaren Prozessoren. Beim Superpipelining ist eine variable Instruktionsbreite einfacher zu handhaben.

**Hochleistungsarchitektur** 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. Solche superskalaren Systeme mit unterschiedlich langer Pipeline bedingen eine sehr komplexe Kontrolllogik, die nicht mehr einfach 'von Hand' entworfen werden kann.