HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG (HTWG)
Fakultät Informatik
Rechner- und Kommunikationsnetze
Prof. Dr.-Ing. Michael Blaich

Übung zur Vorlesung Rechnerarchitektur AIN

Theorieübung – Hazards, Multi Cycle CPU, Cache

Bearbeitung in Zweier-Teams

Team-Mitglied 1: Tobias Latt
Team-Mitglied 2: Jannis Liebscher

Aufgabe 5.1: Hazards

In dieser Aufgabe soll ein Assembler-Code zur Berechnung des element-weisen Maximums zweier Arrays auf Hazards untersucht werden. Danach sollen die Hazards zunächst mit Forwarding und dann zusätzlich mit Code-Scheduling soweit wie möglich beseitigt werden.

Die Code-Sequenz für diese Aufgabe bestimmt elementweise das Maximum von zwei Arrays B und C und speichert das Ergebnis im Array A ab.

$$A = \max(B, C) \text{ mit } A_i = \max(B_i, C_i) \text{ für alle } i$$

	.data		
	A:	.word 0 0 0	# Array A
	B:	.word 1 2 3	# Array B
	C:	.word 3 2 1	# Array C
	n:	.word 3	# n: Elemente im Array
	.text		
		la $s0,A	# Adresse von A in $s0 laden, $s0=&A[0]
		la $s1,B	# Adresse von B in $s1 laden, $s1=&B[0]
		la $s2,C	# Adresse von C in $s2 laden. $s2=&C[0]
		lw $s3,n	# n in $s3 laden
1		sll $t3,$s3,2	# $t3=4*n
2		add $t7,$t3,$s0 DH	# $t7: Adr. des 1. Worts nach A, $t7=&A[n]
3	loop:	beq $s0,$t7,end DH	# Ende, falls $s0 über A hinausläuft
4		lw $t1,0($s1) CH	# B[i] in $t1 laden
5		lw $t2,0($s2)	# C[i] in $t2 laden
6		slt $t3,$t1,$t2 2DH	# wenn $t1<$t2
7		bne $t3,$zero,nimm2 DH	# springe zu nimm2
8		sw $t1,0($s0) CH	# A[i]=B[i]: $t1 an Speicherstelle $s0=&A[i]
9		j next	# überspringe "else"
10	nimm2:	sw $t2,0($s0)	# A[i]=C[i]: $t2 an Speicherstelle $s0=&A[i]
11	next:	addi $s0,$s0,4	# Adr. des nächsten Elem. in A, $s0=&A[i+1]
12		addi $s1,$s1,4	# Adr. des nächsten Elem. in B, $s1=&B[i+1]
13		addi $s2,$s2,4	# Adr. des nächsten Elem. in C, $s2=&C[i+1]
14		j loop	# in die nächste Schleife springen
15	end:		# Ende des Programms

Betrachten Sie den eigentlichen Algorithmus mit den nummerierten Programmzeilen.

5.1.1 Identifizieren Sie alle Control und Data Hazards. DH: 1->2,2->3,4->6,5->6
CH: 3,7

Hinweise:

- Gehen Sie davon aus, dass der jump Befehl die Adresse der nächsten Instruktion bereits in der IF-Stage berechnet.
- Gehen Sie davon aus, dass die Pipeline Sprungbedingung und Sprungziel mit zusätzlicher Hardware in der ID Stage bestimmt.

5.1.2 Wie viele Takte benötigt der Algorithmus bei der obigen Eingabe und wo entstehen Bubbles, wenn **kein Forwarding** verwendet wird und die Pipeline bei bedingten Sprüngen auf das Ergebnis wartet. 53 Takte(57/35 ohne bubbles)

5.1.3 Welche Data Hazards können durch Forwarding umgangen werden? Welche Bubbles bleiben trotz Forwarding bestehen?

5.1.4 Optimieren Sie das Programm durch Code-Scheduling.

5.1.5 Es treten immer noch Control-Hazards auf. Betrachten Sie den Fall, dass die Pipeline anstatt leer zu laufen, immer die nächste Instruktion im Speicher ausführt (Annahme: kein Sprung). Bei einer Fehlentscheidung wird die Instruktion wieder aus der Pipeline gelöscht. Wie viele „Bubbles“ durch Control Hazards gibt es im Mittel bei einer Arraygröße von 100 Elementen? 102: 2 bei Z3, 50 mal falsche Entscheidung Z7

5.1.3
Verhindert durch Forwarding:
DH: 1->2,2->3,4->6 (5->6 verkürzt auf 1 Bubble)
CH: bleiben bestehen

Aufgabe 5.2: Multi Cycle CPU - Pipeline

Betrachten Sie folgenden Assembler-Code:

	.data		
	A:	.word 0 0 0	# Array A
	B:	.word 1 2 3	# Array B
	C:	.word 3 2 1	# Array C
	n:	.word 3	# n: Elemente im Array
	.text		
		la $s0,A	# Adresse von A in $s0 laden, $s0=&A[0]
		la $s1,B	# Adresse von B in $s1 laden, $s1=&B[0]
		la $s2,C	# Adresse von C in $s2 laden. $s2=&C[0]
		lw $s3,n	# n in $s3 laden
1		sll $t3,$s3,2	# $t3=4*n
2		add $t7,$t3,$s0	# $t7: Adr. des 1. Worts nach A, $t7=&A[n]
3		lw $t1,0($s1)	# B[i] in $t1 laden
4		lw $t2,0($s2)	# C[i] in $t2 laden
5	loop:	beq $s0,$t7,end	# Ende, falls $s0 über A hinausläuft
6		slt $t3,$t1,$t2	# wenn $t1<$t2

Vollziehen Sie detailliert den Ablauf der Instruktionen von Zeile 1 bis inklusive Zeile 6 nach, indem Sie die Inhalte aller Pipeline-Register und Forwarding-Entscheidungen protokollieren. Gehen Sie davon aus, dass Sprungbedingungen mit zusätzlicher Hardware in der ID Stage evaluiert werden. Zudem sei Forwarding sowohl für die Berechnung der Sprungbedingung in der ID Stage als auch für die Berechnungen in der EX Stage unterstützt. Füllen Sie dazu die in Moodle bereitgestellte Excel-Tabelle aus. Die Instruktionen inklusive der binären Notation sind bereits eingetragen.

5.2.1 Tragen Sie zunächst in die Spaltenköpfe jeder Tabelle

a. alle Werte ein, die nach einer Stage in den Pipeline-Registern gespeichert werden. Die gespeicherten Signale des Datenpfads sollen unter Data-Path eingetragen werden. die gespeicherten Steuersignale unter EX/MEM/WB Control.
b. alle Steuersignale ein, die in der jeweiligen Pipeline-Stage verwendet werden
c. alle Steuersignale ein, die die Forwarding-Unit bestimmt

Erklärung: Die oberste Tabelle bezieht sich auf die ID Stage. In den Spaltenköpfen sollen alle Werte stehen, die am Ende der ID Stage in das ID/EX-Pipeline-Register geschrieben werden. Zudem sollen alle Steuersignale eingetragen werden, die in der ID Stage benutzt werden. Die zweite Tabelle bezieht sich auf die EX Stage und die dritte Tabelle auf die MEM Stage.

5.2.2 Füllen Sie die Tabellen aus, in dem Sie für die jeweilige Instruktion eintragen,

a. wie die Steuersignale pro Stage bei der Berechnung gesetzt waren (das bezieht sich auf die mit Control/Other beschrifteten Spalten)
b. welche Einträge (DataPath-Werte und Control-Signale) am Ende des Taktes im Pipeline-Register stehen
c. aus welchen Registern die Operanden in der ID (Vergleich der Sprungadressen) und EX Stage (ALU Operation) entnommen wurden (das bezieht sich auf die mit Control/Forwarding beschrifteten Spalten)

Hinweis: In der Vorlesung wurde nicht die Umsetzung aller Instruktionen erklärt. Wenn dies nicht der Fall ist, können Sie die entsprechenden Einträge leer lassen. Dies betrifft die Instruktion sll.

5.3 Direct Mapped Cache

Ein Prozessor verwendet eine Speicherhierarchie mit zwei Cache-Ebenen. Die erste Cache-Ebene besteht aus einem Cache mit 32 Blöcken zu je 16 Bytes. Die zweite Cache-Ebene aus 1024 Blöcken mit ebenfalls 16 Bytes. Bei beiden Caches handelt es sich um Direct-Mapped-Caches. Ein Auszug der beiden Caches ist in Tabelle 1 und Tabelle 2 dargestellt. Der Prozessor lädt ein Wort an der Speicheradresse 2404. Welcher Wert wird zurückgegeben?

Index	V	Tag	Speicherblock (Words)			
…						
19	N	14	11	4	7	13
20	Y	300	23	32	98	76
21	Y	22	98	23	67	98
22	Y	1	7	6	5	4
23	N	7	8	9	10	11
…						

Tabelle 1: Cache der ersten Ebene

0000000001000110.0000000000111100 (50)

0000000000110010.0000000000101000 (2. Ebene Index 150)

Index	V	Tag	Speicherblock			
…						
148	Y	80	123	132	198	176
149	Y	6	98	23	67	98
150	Y	0	70	60	50	40
151	N	2	8	9	10	11
152	N	14	0	0	0	0
…						

Tabelle 2: Cache der zweiten Ebene

5.4 Direct-Mapped-Cache und Set-Associative Cache

Ein Prozessor verwendet getrennte Daten- und Instruktionscaches. Der Daten-Cache wird über eine Speicherhierarchie mit zwei Cache-Ebenen realisiert. Der First-Level-Cache besteht aus einem Direct-Mapped-Cache mit 64 Blöcken zu je 8 Bytes. Ein Auszug des Caches ist in Tabelle 3 gegeben.
Der Second-Level-Cache ist ein 2-Way-Set-Associative-Cache mit einem Gesamtspeicherplatz von 2048 Bytes und einer Blockgröße von 8 Bytes. Ein Auszug dieses Caches ist in Tabelle 4Tabelle 2 gegeben. Gehen Sie weiterhin davon aus, dass eine LRU-Ersetzungsstrategie verwendet wird und die „linken“ Blöcke in der Tabelle jeweils zuletzt genutzt wurden.

Beide Caches verwenden eine Write-Back-Strategie, wobei beim Ersetzen der „Dirty“-Block nur auf die nächst tiefere Cache-Ebene geschrieben wird.

In Abbildung 1 ist der Assembler-Code einer Prozedur gegeben, die elementweise die Summe zweier Arrays A und B bestimmt und in einem dritten Array C abspeichert. Die Übergabeparameter der Funktion sind:

$a0: Adresse des Arrays A
$a1: Adresse des Arrays B
$a2: Adresse des Arrays C
$a3: Anzahl Elemente

Hinweis: Die Cache-Inhalte in den Tabellen sind in nicht vorzeichenbehafteten (unsigned) Worten (Word) dargestellt.

5.4.1 Bestimmen Sie für den ersten Schleifendurchlauf, wie sich die Werte im Cache verändern, wenn die Prozedur mit den Werten $a0=1000, $a1=3040, $a2=9196 und $a3=3 aufgerufen wird. Tragen Sie die Veränderungen der Cache-Inhalte in Tabelle 5 bzw. Tabelle 6 ein. Geben Sie auch die Codezeile an, die die Veränderung verursacht. Geben Sie auch die Codezeile an, die die Veränderung verursacht. Tragen Sie in der mit „D“ überschriebenen Spalte ein, ob es sich um einen „Dirty“ Block handelt.

5.4.2 Betrachten Sie nun die gesamte Schleife mit den gegebenen Argumenten.

a. Wie viele Misses treten jeweils in der ersten und zweiten Cache-Ebene auf?
b. Bestimmen Sie den CPI der Prozedur, wenn der Zugriff auf den First-Level-Cache einen Takt, der Zugriff auf den Second-Level-Cache 10 Takte und der Zugriff auf den Hauptspeicher 100 Takte dauert?

```
a:
3 Durchläufe, jeweils 2*Miss im 1. Level (Z2,Z5) = 6*Miss
Kein Miss im 2. Level


b: Instruction unclear
```

Tabelle 3: Auszug des Inhalt des First-Level-Caches

Index	V	Tag	Speicherblock (in Worten, dezimal, Byte 0 rechts)	
0	Y	17	1	0
1	Y	17	3	2
2	Y	17	5	4
…				
60	Y	5	7	6
61	Y	17	9	8
62	Y	17	11	10
63	Y	17	13	12
…				

Z3,T5,00
Z2,T1,00
Z5,T17,4

Tabelle 4: Auszug des Inhalts des Second-Level-Caches

Set Index	V	Tag	Speicherblock 1 (in Worten, dezimal, Byte 0 rechts)		V	Tag	Speicherblock 2 (in Worten, dezimal, Byte 0 rechts)	
0	Y	8	1	0	Y	5	1	0
1	Y	8	3	2	Y	5	3	2
2	Y	8	5	4	Y	5	5	4
…	Y				Y			
123	Y	8	7	6	Y	2	7	6
124	Y	8	9	8	Y	5	9	8
125	Y	8	11	10	Y	0	11	10
126	Y	8	13	12	Y	5	13	12
…								

Z5,T8,04
Z2,T0,00

Tabelle 5: Veränderungen im First-Level-Cache

Codezeile	Index	D	Tag	Speicherblock (in Worten, dezimal, Byte 0 rechts)	
2	61	0	1	9	10
3	60	0	5	7	6
5	61	1	17	16	10

(Keine Änderung)

Tabelle 6: Veränderungen im Second-Level-Cache

Codezeile	Set Index	D	Tag	Speicherblock 1 (in Worten, dezimal, Byte 0 rechts)		D	Tag	Speicherblock 2 (in Worten, dezimal, Byte 0 rechts)	
2	125	0	8	11	10	0	0	11	10
3	-								
5	125	1	8	16	10	0	0	11	10

(Keine Änderung)

Abbildung 1: Hauptprogramm

```
1     ARRSUM: beq $a3,$zero,BACK
2             lw $s0,0($a0)
3             lw $s1,0($a1)
4             add $s2,$s0,$s1
5             sw $s2,0($a2)
6             addi $a0,$a0,4
7             addi $a1,$a1,4
8             addi $a2,$a2,4
9             addi $a3,$a3,-1
10            j ARRSUM
11      BACK: jr $ra
```

10
6
16