Rechnerarchitektur Serie 4

Patrick Stöckli
16-942-468

Hoeun Yu

Aufgabe 1: Sign Extensions

Die Sign Extension bewirkt, dass ein signed byte oder ein Halbwort auf 32 bit-Länge verlängert wird. Durch die Sign Extension wird aber gleichzeitig das Vorzeichen der Zahl bewahrt. Durch die Sign Extension wird das Zweierkomplement gewahrt, was bedeutet, dass tatsächlich der Wert des ursprünglichen bytes oder Halbwort geladen wird. Bei einer Zero Extension würde indes immer nur den positiven Wert zurückgeben, was bei einem ursprünglich negativen Wert zu einer ungewünschten Wertveränderung führen würde. Diese Wertbeibehaltung ist insbesondere bei Load, Store und Branch wichtig, da diese Operationen I-Type sind und daher nur ein immediate von 16bit besitzen.

Aufgabe 2: Logische und bitweise Operationen

Operationen:	Resultate:
0x0 && 0xEF	false
0xD3 & 0x5B	0x53
0x0 || 0xEF	true
0xA3 | 0x3A	0xBB
!0xFE	false
~0xFE	0x01

Aufgabe 3: Unendliche Schleife

Das Problem liegt darin, dass nach der bne-Instruktion keine weitere Instruktion folgt. Da der Programmfluss bei MIPS einfach weiter geht, wird die foo-Funktion erneut aufgerufen (da diese direkt nach der bne-Instruktion kommt.). Dadurch wird jr($ra) aufgerufen. Da durch die vorhergehende Iterationen des loops die Return Address auf die 5. Zeile gesetzt ist, springt man zur addi-Instruktion. Nach der addi-Instruktion ist der Wert in Register $t0 -1. Da der Wert in $t0 nie mehr 0 erreicht, wird die Schleife ständig wiederholt.

Eine mögliche Lösung wäre es, nach der bne-Instruktion ein syscall einzubauen, beispielsweise exit, der das Programm terminiert und somit eine weitere Ausführung von foo verhindert. Also in etwa so:

```
6:  bne $t0, 0, loop
7:  li $v0, 10
8:  syscall
9:
10: foo:
```

Eine andere Möglichkeit wäre es, nach der bne-Instruktion einen Sprung zu vollführen, um an einer anderen Programmstelle zu landen. Dies könnte beispielsweise so aussehen:

```
6:  bne $t0, 0, loop
7:  j continue
8:
9:  foo:
10: li $t1, 0xAFFFFF04
11: sw $t0, 0($t1)
```

12: jr $ra
13:
14: continue:

Aufgabe 4: bne statt beq

(a) Für eine bne-Implementation muss ein Invertierer an der Zero-Linie angefügt werden. Ohne Invertierer ist die Zero-Linie 1, wenn das Ergebnis von A-B = 0 in der ALU ist. Die Branch-Operation wird also aktiviert, wenn A und B gleich gross ist. Mit Invertierer ist die Zero-Linie automatisch 1, ausser wenn A-B = 0 ist. Ergo solange A und B nicht gleich gross sind, wird die Branch-Operation durchgeführt.

(b) Um bne statt beq in einer multicycle-Implementation auszuführen, müssen folgende Änderungen vorgenommen werden: Von der Zero-Linie braucht es einen Abzweiger, welcher an einen Invertierer angeschlossen ist. Dann braucht es eine zusätzliche Kontrolllinie von Control aus, welche anschliessend mit einem zusätzlichen UND-Gate mit der invertierten Zero-Linie zusammengeführt wird. Das Ergebnis des UND-Gates wird dann an das bereits existierende ODER-Gate angehängt, an welchem bereits die Kontrolllinie von PCWrite dranhängt.

Aufgabe 5: Pipeline Register

Die Register zwischen den Berechnungsstufen werden benötigt, um Daten zwischen den einzelnen Zyklen zu speichern. In jedem Zyklus werden die Daten durch eine Instruktion in das nächste Register befördert. Die Register fungieren also jeweils als Quelle für die Instruktion und garantieren somit auch den Grundsatz: Eine Instruktion pro Zyklus. Hätte man keine Register, so hätte man das Problem, dass man mehrere Instruktionen hätte, die gleichzeitig lesen und schreiben, was zu einem inkonsistenten System führt.

Aufgabe 6: Pipelining Hazard

Ein Control-Hazard liegt vor, wenn eine Branch-Operation vorgenommen wird und der Computer bereits Operationen eines Branches in die Pipeline lädt, bevor klar wird, dass dieser Branch überhaupt ausgewählt wurde. Ist dies nicht der Fall, müssen die bereits geladenen Instruktionen aus der Pipeline entfernt werden.

Ein Structural-Hazard liegt vor, wenn zwei verschiedene Instruktionen in der Pipeline auf die gleiche Ressource zugreifen müssen. Dann müssen diese Instruktionen seriell anstatt parallel abgearbeitet werden, was zu Verzögerungen führt.

Ein Data-Hazard passiert, wenn eine Instruktion Daten benötigt, welche aber durch die vorhergehende Stufe noch nicht berechnet wurden. Das Problem erscheint, wenn die Pipeline die Lese-/Schreibreihenfolge gegenüber der richtigen Reihenfolge ändert. Ein Beispiel: Eine Addition ist nach 5 Berechnungszyklen fertig berechnet. Die zweite Operation, eine Subtraktion, welche den Wert der Addition benötigt, liest den Wert aufgrund pipelinings bereits im 3. Zyklus der Additionsberechnung aus. Dies kann im schlimmsten Fall zu race-conditions führen.

Wie man anhand der Definitionen der Hazards sehen kann, betreffen die Fehler ganz unterschiedlichen Aspekte. Bei einem Data-Hazard ist die wechselseitige Abhängigkeit der Operationen das Problem. Ein Structural-Hazard betrifft eher den Speicher und hat nichts mit der Abhängigkeit der Operationen untereinander zu tun. Control-Hazards hat seine Ursache zwar wie Data-Hazards bei den Abhängigkeiten zwischen den Operationen. Jedoch besteht die Abhängigkeit

bei Control-Hazards darin, welche Operationen ausgeführt werden und nicht wie sie konkret ausgeführt werden.

Aufgabe 7: Stall

Der Unterschied entsteht durch die unterschiedlichen Operationen und dem Zeitpunkt, an dem die wichtigen Angaben zur Verfügung stellen. Auf der Folie 15 handelt es sich um eine add und eine sub-Operation, welche beide auf das gleiche Register zugreifen. Eine write-Instruktion findet aber so früh innerhalb eines Zyklus statt, dass eine read-Operation ebenfalls noch innerhalb des Zyklus ausgeführt werden kann. Es ist also innerhalb eines Zyklus möglich, dass die add-Operation den ausgerechneten Wert ins Register schreibt und die sub-Operation den Wert aus dem Register ausliest.
Auf Folie 19 handelt es sich um einen beq-Befehl. Der PC wird im beq-Befehl erst am Ende der DM-Instruktion geändert, nachdem die Adresse kalkuliert und kontrolliert wurde. Der benötigte Branch ist daher erst am Ende des Zyklus verfügbar. Dies lässt zu wenig Zeit, um eine IM-Instruktion durchzuführen, weshalb diese im nächsten Zyklus durchgeführt wird. Dies führt zu einem längeren Stall.

Aufgabe 8: Data Hazard

Instruktion	Register	1.Zyklus	2.Zyklus	3.Zyklus	4.Zyklus	5.Zyklus	6.Zyklus	7.Zyklus	8.Zyklus
add	$t0, $t5, $t4	IM	Reg	ALU	DM	Reg			
lw	$s2, 0($t0)		IM	Reg	ALU	DM	Reg		
sub	$s3, $t0, $s2			IM	Reg	ALU	DM	Reg	
sw	$t4, 4($s3)				IM	Reg	ALU	DM	Reg

Der erste Data Hazard ist in lw, da der Wert aus Register $t0 in Register $s2 geladen wird. Das Problem ist, dass der Wert von $t0 bereits im 3.Zyklus gelesen wird. Der richtige Wert für $t0 steht aber theoretisch erst im 5.Zyklus zur Verfügung, weshalb lw den falschen Wert liest.

Der zweite und dritte Data Hazard ist in sub. Auch hier ist das Problem, dass im 4. Zyklus der Wert des Registers $t0 und der Wert des Registers $s2 gelesen wird. Der richtige Wert für $t0 steht wie bereits beschrieben erst im 5. Zyklus zur Verfügung (wenn die add-Operation beendet ist), während der richtige Wert für Register $s2 sogar erst im 6.Zyklus zur Verfügung steht.

Der vierte und letzte Data Hazard befindet sich in sw. Hier wird der Wert des Registers $s3 im 5. Zyklus gelesen, obwohl der richtige Wert erst im 7. Zyklus zur Verfügung steht.

Den ersten und dritten Data Hazard könnte man mit forwarding lösen, da es sich add und sub-Operationen handelt und der Output aus der ALU direkt als Input für eine nächste ALU-Operation verwendet werden kann.

Der zweite Data Hazard kann indes nur via Stall gelöst werden. Dies, da das Resultat erst geladen werden kann, wenn es zur Verfügung steht und nicht früher.

Der vierte Data Hazard kann wiederum mittels forwarding gelöst werden, in dem das Resultat aus der ALU-Operation von sub übergeben wird.