Kapitel 2: Zielarchitekturen

Aufgabe 2.01: RISC/CISC

Diskutieren Sie die Vor- und Nachteile von RISC bzw. CISC bezüglich

a) Codegröße

CISC ++

Viele komplex kodierte Befehle unterschiedlicher Länge. Häufiger verwendete Befehle werden kürzer kodiert als komplexe längere Befehle. Es werden relativ wenige Befehle gebraucht um den Code abzuarbeiten und die Codegröße wird geringer.

RISC --

Wenige heterogene gleich lange Befehle (häufig 32-Bit breit). Es werden mehr „einfache“ Befehle gebraucht um den gleichen Code abzuarbeiten. Dadurch müssen häufiger Werte in Registern abgelegt werden. Als Ausgleich ist der Registersatz tendenziell größer und es werden 3-Addressbefehle verwendet, die mehr Bits in der Kodierung verwenden.

RISC +

Durch die Verwendung von komprimierten Befehlen (z.B. 16-Bit ARM Thumb Code) kann die Codegröße auch bei RISC verkleinert werden. Dies geht allerdings zu lasten der Performanz.

b) Pipelining

RISC ++

Alle RISC Befehle haben eine vergleichbare Komplexität und können daher gut in einer Pipeline umgesetzt werden, die eine hohe Auslastung der Ressourcen gewährleistet. Es muss nur in einer Stufe ein Speicherzugriff durchgeführt werden.

CISC --

Die komplexen CISC Befehle können nicht in einer Pipeline umgesetzt werden, die eine gute Auslastung der Ressourcen gewährleistet. Die Pipeline muss für komplexe Befehle ausgelegt sein und ist bei einfachen Befehlen unausgelastet. Ein Speicherzugriff kann sowohl beim Laden und Zurückschreiben der Operanden stattfinden.

CISC +

In heutigen Prozessoren (bei Intel ab Pentium Prozessor) werden die CISC Befehle intern in sog. µ-Ops umgesetzt, die dann von einem RISC Kern ausgeführt werden.

c) Steuerwerk
   RISC ++
   Durch den regelmäßigen Befehlsatz mit wenigen gleich langen Befehlen ist die Befehlsdekodierung einfacher, was Zeit und Fläche spart.
   CISC --
   Durch die vielen komplexen Befehle mit unterschiedlicher Länge ist die Befehlskodierung sehr aufwändig. Alleine die Einheit zur Befehlslängenerkennung in heutigen X86-Prozessoren hat die Größe eines PowerPCs.

d) Compilerunterstützung
   RISC +
   Für Compiler ist der Umgang mit dem großen homogenen Registersatz einfacher und es kann vergleichsweise guter Code erzeugt werden.
   CISC -
   CISC Prozessoren sind für den Menschen einfacher in Assembler zu programmieren, stellen aber den Compiler vor Herausforderungen. Die wenigen Spezialregister wirken sich negativ auf die Codeerzeugung aus. So wirkt es sich nachteilig aus, wenn die Befehlsselektion und Registerallokation unabhängig voneinander betrachtet werden.

Aufgabe 2.02: RISC/CISC

Gegeben ist folgender Assemblercode:

```
a:   word 10
b:   word 20
c:   word

_xyz:
     mov r0, [a]
     add r0, [b]
     mul r0, [b]
     mov [c], r0
```

a) Handelt es sich bei dem Prozessor, der den Assemblercode ausführen kann, um eine RISC oder CISC Maschine? Begründen Sie Ihre Antwort.
   Bei dem Code handelt es sich um eine CISC Maschine. Es gibt keine getrennten Befehle für den Speicherzugriffe (Load-Store Befehle), sondern der Speicherzugriffe wird innerhalb eines Transferbefehl oder arithmetisch-logischen Befehl durchgeführt.

b) Was macht der Code?
   Der Quellcode berechnet c = (a+b)*b, wobei die Werte a und b aus dem Speicher geladen und das Ergebnis im Speicher abgelegt wird.

c) Portieren Sie den Code auf den jeweils anderen Maschinentypen.

```
a:    word 10
b:    word 20
c:    word

_xyz:
      load r0, [a]
      load r1, [b]
      add r0, r0, r1
      mul r0, r0, r1
      store [c], r0
```

Aufgabe 2.03: Pipelining

In der Vorlesung wurde die 5-stufige DLX Pipeline (IF, ID, EX, MEM, WB) behandelt.

a) Handelt es sich um eine RISC oder CISC Pipeline?
Es handelt sich um eine RISC Pipeline.

b) Erklären Sie den Unterschied zwischen der WB und MEM Stufe?
In der WB-Phase wird ein Schreibzugriff auf das Registerfile durchgeführt, also das Ergebnis ins Register geschrieben. In der MEM Phase wird der Datenzugriff (Lese oder Schreiben) auf den Hauptspeicher durchgeführt.

c) Wie viele Takte muss ein Befehl innerhalb der Pipeline angehalten werden, der das Ergebnis des vorherigen Befehls als Eingabe verwendet?
Der Befehl muss 2 Zyklen warten, wenn ID den Registerinhalt in der 2ten Takthälfte liest und WB bereits in der 1ten Takthälfte schreibt.

d) In welcher Pipelinestufe würde dieser Befehl angehalten werden, wenn die Pipeline über eine automatisch Konfliktbehandlung verfügt?
Innerhalb der ID-Phase.

Aufgabe 2.04: Pipelining

Auf einem Prozessor mit einer 5-stufiger DLX Pipeline wird das folgende Programm ausgeführt:

```
(1) load R2,[Var1]       ;Lade den Inhalt von Var1
(2) load R1,[Var2]       ;Lade den Inhalt von Var2
(3) add R1,R2            ;R1 = R1 + R2
(4) load R3,[Var3]       ;Lade den Inhalt von Var3
(5) load R4,[Var4]       ;Lade den Inhalt von Var4
(6) sub R4,R3            ;R4 = R4 - R3
(7) add R1,R4            ;R1 = R1 + R4
(8) store [Res],R1       ;Speichere Resultat nach Res
```

a) Markieren Sie alle Datenabhängigkeiten im ASM Code.

```
(1) load R2,[Var1]  ;
(2) load R1,[Var2]  ;
(3) add R1,R2       ; Datenabhängigkeit von (1) und (2)
(4) load R3,[Var3]  ;
(5) load R4,[Var4]  ;
(6) sub R4,R3       ; Datenabhängigkeit von (4) und (5)
(7) add R1,R4       ; Datenabhängigkeit von (3) und (6)
(8) store [Res],R1  ; Datenabhängigkeit von (7)
```

b) Skizzieren Sie den Ablauf der Pipeline. Markieren Sie evtl. auftretende Pausen. Für die Zugriffe auf Variablen können Sie davon ausgehen, dass diese über einen Cache rechtzeitig zur Verfügung stehen. Die Register werden in der Write-Back Phase in der ersten Takthälfte geschrieben und in der ID Phase in der zweiten Takthälfte geladen. Die Pipeline verfügt über einen automatischen Stall-Mechanismus, der die Ausführung so lange anhält bis der Konflikt aufgelöst ist.

#	IF	ID	EX	ME	WB
1	load R2				
2	load R1	load R2			
3	add R1, R2	load R1	load R2		
4	**load R3**	**add R1, R2**	load R1	load R2	
5	**load R3**	**add R1, R2**		load R1	load R2
6	load R3	add R1, R2			load R1
7	load R4	load R3	add R1, R2		
8	sub R4,R3	load R4	load R3	add R1, R2	
9	**add R1,R4**	**sub R4,R3**	load R4	load R3	add R1, R2
10	**add R1,R4**	**sub R4,R3**		load R4	load R3
11	add R1,R4	sub R4,R3			load R4
12	**store R1**	**add R1,R4**	sub R4,R3		
13	**store R1**	**add R1,R4**		sub R4,R3	
14	store R1	add R1,R4			sub R4,R3
15		**store R1**	add R1,R4		
16		**store R1**		add R1,R4	
17		store R1			add R1,R4
18			store R1		
19				store R1	
20					store R1

Tabelle 2.1: Programmausführung in der Pipeline

c) Gehen Sie nun davon aus, dass die Pipeline über keine automatische Konflikterkennung hat. Füllen Sie den ASM-Code mit NOP Befehlen auf, so dass das Programm korrekt ausgeführt wird.

```
load R2,[Var1]
load R1,[Var2]
nop
nop
add R1,R2
load R3,[Var3]
load R4,[Var4]
nop
nop
sub R4,R3
nop
nop
add R1,R4
nop
nop
store [Res],R1
```

d) Gehen Sie nun davon aus, dass die Pipeline über eine Forwarding-Technik verfügt, so dass ein berechneter oder geladener Wert direkt an die EX-Phase weitergeleitet werden kann. Markieren Sie in der Lösung von Teilaufgabe c) sämtliche NOPs, die damit eingespart werden können.

Die NOPs, die eingespart werden können, wurden **fett** markiert. Nach einem load ist es nicht möglich durch Forwarding sämtlichen NOPs zu entfernen, weil der Wert erst am Ende der MEM-Phase vorhanden ist, aber am Anfang der EX-Phase benötigt wird.