7 Ein einfacher CISC-Prozessor

In diesem Kapitel wird ein einfacher Prozessor vorgestellt. Die Architektur, die wir implementieren, wurde von R. Bryant und D. O'Hallaron entworfen und verwendet eine Untermenge der Befehle der IA32-Architektur. Wir geben zunächst eine Architekturspezifikation in Form eines C++ Modells an. Anschließend wird die Architektur in Verilog implementiert.

7.1 Die Y86 Instruction Set Architecture

Frei programmierbare Mikroprozessoren sind sicherlich die wichtigste Anwendung der Digitaltechnik. Sie stellen die Schnittstelle zwischen der Hardware- und der Softwarewelt dar. Es gibt eine Vielzahl an Architekturen und Modellen. Im Server- und Desktopbereich hat sich die 32-Bit Intel Architektur (IA32) durchgesetzt. Entsprechende Prozessoren werden aber auch von anderen Anbietern hergestellt, etwa von AMD.
Bei der IA32-Architektur handelt es sich um eine CISC-Architektur (Complex Instruction Set Computer), was darauf hinweist, dass die Architektur eine große Anzahl von Mikroprozessorbefehlen bietet. In der Tat findet sich in den Handbüchern von Intel eine Liste von etwa 300 Befehlen mit einer Vielzahl an Optionen. Die Architektur, die wir implementieren, wurde von R. Bryant und D. O'Hallaron zu pädagogischen Zwecken entworfen und ist auf eine sehr kleine Auswahl dieser Liste beschränkt. Sie wird als „Y86"-Architektur bezeichnet, in Anlehnung daran, dass die Produktnummern der ersten entsprechenden Intel-Prozessoren auf „86" endeten.

7.1.1 Befehle, Register und Speicher

Wir gehen davon aus, dass der Leser die Grundlagen der sequenziellen Programmierung beherrscht. In höheren Programmiersprachen, wie C oder Java, werden alle Daten in Variablen abgelegt, wobei jeder Variablen eine Speicherzelle zugeordnet ist. Die Speicherzellen sind nummeriert. Die „Nummer" einer Speicherzelle wird als *Adresse* bezeichnet. Wir gehen davon aus, dass jede Speicherzelle genau ein Byte (also 8 Bit) speichert.
Wir schreiben `Mem[ea]` für den Inhalt der Speicherzelle mit der Adresse `ea` (`ea` steht für *effective address*). Sowohl die Befehle als auch die Daten, mit denen das Programm arbeitet, werden im gleichen Speicher hinterlegt (Abbildung 7.1). Eine derartige Architektur wird als *von-Neumann-Architektur* bezeichnet.
Unser Prozessor verwendet einen gemeinsamen *Datenbus*, um auf den Speicher und die I/O-Geräte zuzugreifen (siehe Abschnitt 6.4). Wir nehmen an,

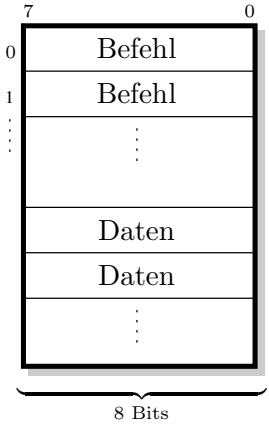

Abbildung 7.1. Schematische Darstellung des Speichers: Die Befehle und Daten werden in einem gemeinsamen Adressraum abgelegt

dass unser Prozessor die einzige Komponente ist, die diesen Bus steuert (Abbildung 7.2). Wir nennen diesen Bus *Systembus* (engl. system bus).

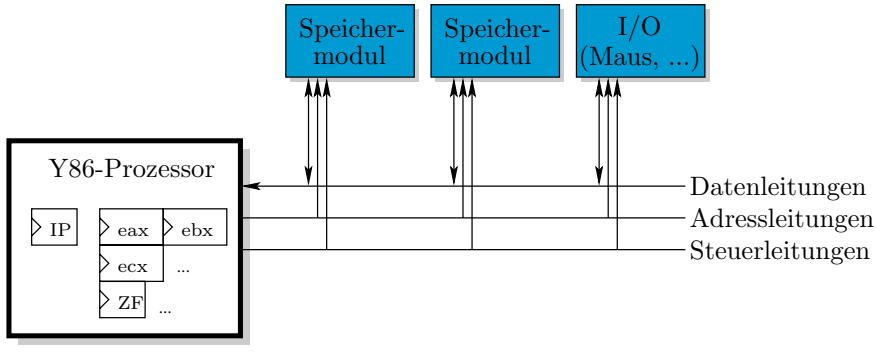

Abbildung 7.2. Der Y86-Prozessor und die anderen Systemkomponenten

Die Register

Da der Zugriff auf den Speicher über den Systembus mehrere Takte dauern kann und somit vergleichsweise langsam ist, können Datenworte im Prozessor in *Registern* zwischengespeichert werden. Diese Register sind in einer höheren Programmiersprache nicht direkt sichtbar. Es ist die Aufgabe des Compilers, die Register bei der Übersetzung des Programms effizient zu nutzen.
Die Y86-Architektur hat acht Datenregister mit jeweils 32 Bit, die wie folgt bezeichnet werden:

Index	0	1	2	3	4	5	6	7
Name	**eax**	**ecx**	**edx**	**ebx**	**esp**	**ebp**	**esi**	**edi**

Die ersten vier Register (**eax** bis **ebx**) dienen dazu, Daten zu halten.[1] Die Register **esp** und **ebp** werden im Normalfall verwendet, um einen *Stack* zu implementieren und werden daher *Stackregister* genannt. Da sie üblicherweise eine Adresse enthalten, werden sie auch *Zeigerregister* genannt. Die Register **esi** und **edi** sind für Array-Indizes gedacht.

Der Instruction Pointer (IP)

Damit der Prozessor auch weiß, wo im Speicher der nächste Befehl liegt, den er ausführen soll, gibt es ein weiteres Zeigerregister, das auf die Adresse des nächsten Befehls zeigt. Dieser Zeiger wird *Instruction Pointer* (IP) genannt. Sprunganweisungen verschieben diesen Zeiger, um das Programm an einer anderen Stelle fortzuführen. Befehle, die keine Sprunganweisung sind, ändern den IP ebenfalls: Der Zeiger wird einfach auf die Adresse der nächsten Instruktion des Programms gesetzt.
Die Befehle, die unser einfacher Prozessor ausführen kann, sind in drei Kategorien eingeteilt:

1. Die *arithmetischen und logischen Befehle* führen Berechnungen wie z. B. die Addition aus.
2. Die *Datentransfer-Befehle* ermöglichen es, Daten aus dem Prozessor über den Systembus in einem RAM-Modul abzulegen und wieder einzulesen. Diese Befehle werden auch zur Kommunikation mit den I/O-Geräten verwendet (sog. *Memory-mapped I/O*).
3. Die *Sprungbefehle* steuern den Kontrollfluss. Es handelt sich also um Befehle, die (möglicherweise bedingt) zu anderen Befehlen springen.

Zusätzlich zu den Datenregistern und dem Instruction Pointer verwendet die Y86-Architektur *Flag-Register*. Die Flag-Register speichern weitere Informationen über das Ergebnis des jeweils letzten arithmetischen oder logischen Befehls. Ein Flag-Register ist genau ein Bit breit. Wir betrachten zunächst nur ein einziges Flag-Register: Das Zero-Flag (**ZF**) wird gesetzt, wenn das Ergebnis der letzten arithmetischen Operation null (zero) war. Es wird als Bedingung für Sprungbefehle verwendet.

Die Befehle der Y86-Architektur

Bevor wir die einzelnen Befehle spezifizieren, gehen wir zuerst auf deren Kodierung ein. Jeder Befehl entspricht einer Folge von Bytes. Die Länge dieser Folge hängt von dem jeweiligen Befehl ab – einfachere Befehle benötigen weniger Platz. Dadurch soll erreicht werden, dass der Programmcode insgesamt

[1] Man beachte die etwas sonderbare, nicht-alphabetische Reihenfolge der Register.

kleiner wird.[2] Der Befehl, der ausgeführt werden soll, wird durch den Wert des ersten Bytes der Folge bestimmt. Dieses Byte wird daher auch als *Opcode* bezeichnet.
Da Zahlen schwer zu merken sind, wird der Opcode im Assembler-Programm durch ein *Mnemonic* (eine kurze Zeichenfolge) ersetzt. Ein Programm, das eine solche Textdatei wieder in Zahlen (Maschinencode) übersetzt, wird als *Assembler* bezeichnet.
Die Befehle, die wir implementieren, sind in Abbildung 7.3 zusammengefasst. Alle Befehle erhöhen den IP um die Länge des jeweiligen Befehls.

1. Die Befehle **add** und **sub** führen eine Addition bzw. Subtraktion durch. Im Befehl sind zwei Register kodiert, genannt RD (Destination) und RS (Source). Die Summe RD+RS (bzw. die Differenz RD-RS) wird in RD abgelegt. Falls der neue Wert von RD null ist, wird das Flag **ZF** gesetzt, und ansonsten gelöscht.
2. Der Befehl **RRmov** (kurz für Register-Register **mov**)[3] kopiert den Wert des Registers RS in das Register RD.
3. Der Befehl **RMmov** (kurz für Register-Memory **mov**) kopiert den Wert des Registers RS in den Speicher. Die Adresse (`ea`) wird dabei als Summe des Registers **esi** und des 8-Bit *Displacements* (Byte 3) berechnet:

   $$\texttt{ea} = \texttt{esi} + \text{Displacement}$$

   Beachten Sie, dass sich die Befehle **RRmov** und **RMmov** nur durch die oberen Bits des zweiten Bytes unterscheiden lassen. Dieses Feld wird als `mod` (wie Modifier) bezeichnet.
4. Der Befehl **MRmov** (kurz für Memory-Register **mov**) kopiert einen Wert aus dem Speicher in das Register RS. Die Adresse (`ea`) wird dabei genau wie beim Befehl **RMmov** berechnet.
5. Der Befehl **jnz** addiert den im Befehl angegebenen Offset (genannt *Distance*) zum Instruction Pointer, falls das Flag **ZF** nicht gesetzt ist (das Ergebnis der arithmetischen Operation also nicht null war).
6. Der Befehl **hlt** stoppt die Ausführung des Programms.

Die Kodierung und die Bedeutung dieser Befehle ist in Abbildung 7.3 zusammengefasst. Die Notation R1 ← R2 ist wie folgt zu lesen: Der Wert in Register R2 wird in das Register R1 kopiert. Die im rechten Teil der Abbil-

[2] Andere Architekturen, wie z. B. die MIPS-Architektur, verwenden Befehlsworte, die immer gleich lang sind, was die Decodierung vereinfacht.

[3] Wir verwenden eine Langform der Befehlsnamen, um die Art der Operation klarzustellen. Ein Assembler kann die Unterscheidung anhand der Operanden durchführen und generiert den passenden Opcode.

Mnemonic	Bedeutung	Opcode	7 6 3 0	
add	RD←RD+RS	01	11 RS RD	
sub	RD←RD-RS	29	11 RS RD	
jnz	`if`(¬ZF) IP←IP+Distance	75	Distance	
RRmov	RD←RS	89	11 RS RD	
RMmov	`MEM[ea]`←RS	89	01 RS 110	Displacement
MRmov	RS←`MEM[ea]`	8b	01 RS 110	Displacement
hlt		f4		

Abbildung 7.3. Die Befehle unseres Y86-Prozessors

dung dargestellten Opcodes sind in hexadezimaler Schreibweise notiert. Bei den in den Befehlen enthaltenen Offsets ist zu beachten, dass es sich um vorzeichenbehaftete Zahlen handelt (im Zweierkomplement). Im Falle des **jnz** Befehls bewirkt eine negative Zahl einen Rückwärtssprung.

Der Offset bei Sprungbefehlen bezieht sich auf den Befehl *nach* dem Sprungbefehl, da der IP ja schon um zwei erhöht worden ist.

7.1.2 Ein kleines Assembler-Beispiel

Mit den oben angeführten Befehlen ist es schon möglich, ein erstes Assembler-Programm zu schreiben, das von einem Y86-Prozessor ausgeführt werden kann. Da die Kodierung der Befehle unseres Y86-Prozessors mit denen der Intel-Prozessoren übereinstimmt, können die Programme auch auf jedem IA32-Prozessor ausgeführt werden. Umgekehrt klappt dies in der Regel nicht: Von Compilern generierte IA32-Programme verwenden viele zusätzliche Befehle, die unsere Y86-Architektur nicht kennt.
Ein Assembler-Befehl wie

sub R1, R2

ist wie folgt zu lesen: Der Inhalt des Registers R2 wird von dem Inhalt des Registers R1 abgezogen. Das Resultat wird schließlich im Zielregister R1 gespeichert.