Die ReAl Computerarchitektur

ReAl = *Re*ssourcen-*Al*gebra.

Vorläufige Kurzbeschreibung

Stand: 1.2 vom 11. 9. 2006

Prof. Dr. Wolfgang Matthes
Peukinger Weg 34
59423 Unna/Germany
http://www.realcomputerarchitecture.com

1.	Einführung		3
	1.1	Überblick	3
	1.2	Charakteristische Anwendungsgebiete	6
	1.3	Zum Stand der Technik	10
	1.4	Elementare Ressourcen	12
2.	Das ReAl-Verarbeitungsmodell		16
	2.1	Operatoren und Parameter	16
	2.2	Notation von ReAl-Verarbeitungsabläufen	18
	2.3	Elementare Operatoren	20
	2.4	ReAl und gegebene Programmiersprachen	22
	2.5	ReAl-Maschinen	22
	2.6	Die Plattform	27
	2.7	Verarbeitungsressourcen	29
	2.8	Verkettung	37
3.	ReAl-Systemkonfigurationen		47
	3.1	Grundlagen	47
	3.2	Ressourcen mit eigenen Speichermitteln	48
4.	ReAl-Konfigurationen in Schaltkreisen		58
5.	Plattformstrukturen		64
	5.1	Einfache Plattformen	64
	5.2	Bedingte Verzweigungen	65
	5.3	Höherentwickelte Plattformen	70
	5.4	Nachbildung der ReAl-Wirkprinzipien mittels Software (Emulation)	72
6.	Ressourcenverwaltung		75
	6.1	Ressourcenverwaltung über Tabellenstrukturen	75
	6.2	Anzahl der Ressourcen	81
	6.3	Numerierung der Ressourcen	82
	6.4	Ressourcen auf- und abbauen	83
	6.5	Ressourcenadressierung in Byte- und Maschinencodes	83
	6.6	Ressourcen als Hardware	86
	6.7	Ressourcen als Software (Emulation)	89
	6.8	Zustandsfreie und zustandsbehaftete Ressourcen	89
	6.9	Instrumentierung	91
7.	ReAl-Bytecodes		95
8.	ReAl-Maschinencodes		99
	8.1	Beispiel 1: Bytecodes	99
	8.2	Beispiel 2: 32 Bits, zwei Wirkungen	101
	8.3	Beispiel 3: 32 Bits, lange Adressen	102
	8.4	Beispiel 4: 16 Bits, zwei Wirkungen	104
	8.5	Beispiel 5: 16 Bits, längere Adressen	106
	8.6	Beispiel 6: kurze Befehle mit parallel auslösbaren Funktionen	107
	8.7	Beispiel 7: 64 Bits, viele parallel auslösbare Funktionen	108
9.	ReAl-Maschinen		111

1. Einführung

1.1 Überblick

Das Ziel:
Eine universelle Schnittstelle zwischen Hard- und Software. Die Schnittstellenspezifikation soll es ermöglichen, die Einzelheiten der Verarbeitungsvorgänge so genau zu beschreiben, daß die jeweils vorhandene Hardware in optimaler Weise ausgenutzt werden kann:

- Der den Verarbeitungsabläufen innewohnende (inhärente) Parallelismus soll in dem Maße ausgenutzt werden können, in dem tatsächlich Hardware verfügbar ist,
- es soll möglich sein, Soft- und Hardware jederzeit gegeneinander auszutauschen (z. B. ein Unterprogramm gegen einen Spezialprozessor und umgekehrt).

Die Entwicklungsgeschichte im Kurzüberblick
Herkömmlicherweise ist die Schníttstelle zwischen Hard- und Software durch eine Menge elementarer Datenstrukturen und über diese definierte Maschinenbefehle gekennzeichnet. Der einzelne Maschinenbefehl spezifiziert einen ziemlich einfachen Verarbeitungsvorgang. Befehlslisten und Datenstrukturen sind aus der Erfahrung heraus entstanden. Die ersten Computer wurden als Rechenmaschinen entwickelt. Deshalb lag es nahe, die Grundrechenarten des numerischen Rechnens als Befehlswirkungen vorzusehen. Ein Befehl veranlaßt typischerweise eine solche Rechenoperation, einen Hilfsablauf (Datentransport, Eingabe, Ausgabe usw.) oder eine Funktion der Ablaufsteuerung (Verzweigung, Unterprogrammruf usw.).

Die Architekturentwicklung wird von Erkenntnissen und Gegebenheiten verschiedener Gebiete von Forschung und Technik beeinflußt:

- Aus der Anwendungspraxis: Anforderungen des numerischen Rechnens und anderer Anwendungsgebiete,
- aus der Theorie: Grundlagen der Berechenbarkeit (welche Befehlswirkungen sind erforderlich, um eine wirklich universelle Programmierbarkeit zu gewährleisten?),
- aus der Technologie: Bedingungen und Grenzen der Realisierbarkeit.

Zu Beginn der Entwicklung war die Hardware kostbar. Deshalb konnten der Architektur nur wirklich elementare Datenstrukturen und Befehlswirkungen zugrunde gelegt werden. Die einzelnen Architekturen unterscheiden sich vor allem in den Zubringerfunktionen (Adressierung, Registermodell).

Rechnerarchitektur als Ressourcen-Algebra
Wir nehmen die Grundlagen der Berechenbarkeit und die Anforderungen der Anwendungspraxis als gegeben an. Unser Ziel besteht darin, Maschinen (= Hardware) zu bauen, die die einschlägigen Funktionen erbringen.

Jede x-beliebige Maschine ist letzten Endes aus Speichergliedern (Flipflops, Speichermatrizen) und aus kombinatorischen Schaltungen aufgebaut (Register-Transfer-Struktur).

Läuft ein beliebiges Programm auf einer Maschine beliebiger Architektur, so handelt es sich stets um die Nutzung einer Hardware, also um das Belegen von Registern, um kombinatorische Informationswandlungen, um Zustandsübergänge usw. Jedes beliebige Programm wird letzten Endes in Informationstransporte, kombinatorische Verküpfungen und Zustandsübergänge überführt, also in den Informationsfluß in einer Register-Transfer-Struktur. Das geschieht teils durch Compiler und Laufzeitumgebung (Umsetzung der Programmierabsicht in Folgen von Maschinenbefehlen), teils durch die steuernde Hardware (Ausführung der Maschinenbefehle in der jeweiligen Register-Transfer-Struktur).

Wir versuchen, von der Programmierabsicht aus direkt auf eine jeweils angemessene Register-Transfer-Struktur durchzugreifen bzw. eine solche Struktur ad hoc aufzubauen, um die Programmierabsicht zu verwirklichen.

Die grundlegene Arbeitshypothese:
Hardware spielt keine Rolle – es ist genügend da (nicht wie bei armen Leuten ...).

Unser Grundmodell:
Wenn wir irgend etwas tun wollen, so holen wir ein dazu geeignetes Stück Hardware aus einem Lager (wie einen Hammer, um einen Nagel einzuschlagen, einen Schraubenschlüssel, um eine Mutter festzuziehen usw.) und verwenden es für die auszuführende Informationswandlung. Wenn wir zwei Zahlen zueinander addieren wollen, holen wir einen Addierer, wenn wir zwei Werte miteinander vergleichen wollen, einen Vergleicher usw. Ein Stück Hardware, das seine Arbeit getan hat, wird in das Lager zurückgestellt. In Weiterführung der Analogie holen wir für unsere Arbeit soviele Werkzeuge, wie wir jeweils brauchen, z. B. 50 Hämmer, wenn 50 Nägel einzuschlagen, oder 50 Addierer, wenn 50 Zahlenpaare zueinander zu addieren sind.

Unsere Architekturdefinition umfaßt eine Menge von Ressourcen und eine Menge von Datenstrukturen. Ressourcen führen bestimmte Operationen über Daten bestimmter Strukturen aus.

Diese Abbildung konstituiert im Grunde eine algebraische Struktur. Deshalb wird die Architektur mit *ReAl* = *Re*ssourcen-*Al*gebra bezeichnet.

Das Modell einer Ressource ist eine Hardware, die bestimmte Informationswandlungen ausführt, also aus gegebenen Daten (an den Eingängen) neue Daten (an den Ausgängen) errechnet.

Allgemeine Annahmen und Entwicklungsziele:

1. Es gibt jederzeit genügend Ressourcen. Das ist zunächst eine theoretische Annahme (Hypothese vom (nahezu) unbeschränkten Ressourcenvorrat). Auf dieser Grundlage ist es möglich, Ressourcen in beliebiger Anzahl anzufordern (beispielsweise einige hundert Multiplikationswerke) und den inhärenten Parallelismus in vollem Umfang auszunutzen. Maschinenprogramme werden typischerweise so erzeugt (z. B. mittels Compiler), als ob beliebig viele Ressourcen zur Verfügung stünden. Die Anpassung an die Gegebenheiten der Praxis (jeder tatsächliche Ressourcenvorrat ist begrenzt) kann zur Compilierzeit oder zur Laufzeit erfolgen (Emulation, Virtualisierung).
2. Ob eine Ressource als Software oder als Hardware ausgeführt ist, spielt keine Rolle.
3. Das grundsätzliche Modell einer Ressource ist stets eine Hardware, also eine technische Einrichtung mit Ein- und Ausgängen.
4. Ein Verarbeitungsvorgang (Programmablauf) besteht in der Benutzung von Ressourcen im Laufe der Zeit (Ressourcen werden bei Bedarf aus dem Ressourcenvorrat entnommen und bei Nichtgebrauch zurückgegeben).
5. In Hinsicht auf ein gegebenes Anwendungsproblem ist der Universalrechner im Grunde nur eine Notlösung. Um ein wirkliches Maximum an Verarbeitungsleistung zu erhalten, wäre eine Spezialhardware erforderlich, deren Maschinenzyklen auschließlich dazu dienen, die gewünschten Ergebnisse zu berechnen. In einer solchen Maschine würden keine Taktzyklen und Speicherzugriffe dazu verwendet werden, Befehle zu holen, Zwischenwerte abzuspeichern und wieder zu laden, Funktionsaufrufe auszuführen usw. Herkömmlicherweise nehmen wir den Universalrechner, weil wir eine solche Maschine nicht ohne weiteres bauen können. Die ReAl-Architektur soll es ermöglichen, solche (fiktiven) Maschinen zur Laufzeit des Programms bedarfsweise auf- und abzubauen.
6. Die zur Steuerung der Verarbeitungsvorgänge vorgesehenen Anweisungsangaben (Operatoren) betreffen nur die grundlegenden Verfahrensschritte des Anforderns, Transportierens, Auslösens usw., nicht aber konkrete Maschinenoperationen (vollständige Maschinenunabhängigkeit).
7. Komplexere Ressourcen können rekursiv aus elementaren Ressourcen aufgebaut werden.
8. Wo die Ressourcen angesiedelt und wie sie aufgebaut sind, spielt keine Rolle. Unser Modell schließt u. a. auch die Möglichkeit ein, Ressourcen übers Internet anzufordern und auszunutzen (z.B. Spezialrechner).

Unser Grundmodell ist stets die Hardware. Die Programmierphilosophie läuft letzten Endes darauf hinaus, eine Hardware zu entwerfen, die die betreffende Verarbeitungsaufgabe ausführen kann (zunächst als Gedankenexperiment, unabhängig von der tatsächlichen praktischen Durchführbarkeit).

Wir tun so, als ob sich für jede x-beliebige Informationswandlung eine Hardware bauen ließe (und entscheiden dann fallweise, was tatsächlich als Hardware gebaut wird und was nicht).

Diese (fiktive) Hardware kann zur Laufzeit dynamisch auf-, um- und abgebaut werden. Die Maschinenbefehle der ReAl-Architektur betreffen keine bestimmten Operationen, sondern das Auf-, Um- und Abbauen von Ressourcen, die zugehörigen Datentransporte (Parameterversorgung) und das Aktivieren der betreffenden Ressourcen.

In letzter Konsequenz braucht man stets Ressourcen, die wirklich als Hardware aufgebaut sind. Daraus ergeben sich bestimmte Prinzipien der Nutzung, der Parameterversorgung usw.

Die Anwendung des ReAl-Prinzips
Der Programmierer hat grundsätzlich die Wahl. Die Wahlmöglichkeiten liegen zwischen zwei Extremen:

1. Herkömmliche Programmierweise: eine Ressource zu einer Zeit. In Analogie zu den herkömmlichen Maschinenbefehlen wird eine Ressource angefordert, mit Parametern versorgt, aktiviert und zum Schluß wieder freigegeben. Dann wird die nächste Ressource angefordert usw.
2. Die Programmierabsicht wird insgesamt in eine – womöglich gigantische – fiktive Spezialhardware umgesetzt (für jede Informationswandlung wird eine eigene Ressource angefordert, die Ressourcen werden untereinander verkettet usw.).

Die zweite Alternative findet ihre praktischen Grenzen im Speicherbedarf und im Umfang der tatsächlich verfügbaren Hardware, so daß in der Praxis stets Kompromisse gefunden werden müssen.

Wenn es keine passende Hardware gibt:
Dann ist sie durch Software zu emulieren. Programme und Hardware-Ressourcen werden gleichartig aufgerufen. Jedes Programm bzw. Unterprogramm entspricht dem Modell einer Hardware mit Registern an den Ein- und Ausgängen (Register-Transfer-Modell).

Wir bauen – fiktiverweise – stets eine Hardware, die unsere Programmierabsicht verwirklicht.

Jede Ressource ist – zumindest gedanklich (fiktiv) – eine Hardware-Software-Schnittstelle auf dem Register-Transfer-Niveau, kann also formal als Hardware beschrieben werden (Boolesche Gleichungen, Automatentabellen usw.).

ReAl-Programme werden nicht in Form einer Zeichen- oder Bitkette dokumentiert, deren Wirkungsweise sich nur durch die Ausführung ergibt, sondern in Form von Strukturen, die auch im Ruhezustand (ohne etwas auszuführen) einer eingehenden Analyse zugänglich sind.

Parameterübergabe
In letzter Konsequenz werden Parameter als Werte übergeben (Hardware braucht letzten Endes Werte, um damit rechnen zu können).

Was nicht gleich in das Schema paßt, wird passend gemacht. Hierzu sehen wir ggf. weitere Ressourcen vor (z. B. für Hilfsfunktionen, wie zum Heranschaffen und Abspeichern von Daten, aber auch zum Beschaffen und Verwalten von Ressourcen).

Elementare Parallelisierung
Was parallel ausführbar ist, kann auch als parallel ausführbar programmiert werden. Der Programmierer (oder Compiler-Autor) soll nur dann serialisieren, wenn dies aus Gründen des Verarbeitungsablaufs erforderlich ist (also nicht eine einzige Ressource anfordern und diese in 100 Durchläufen mehrfach ausnutzen, sondern 100 Ressourcen anfordern und – soweit überhaupt möglich – gleichzeitig ansprechen).

Die Parallelausführung kann durch spezielle Ressourcen unterstützt werden, die beispielsweise die parallel betriebenen Verarbeitungsressourcen mit Operanden versorgen und die Ergebnisse abtransportieren.

Wir beschreiben den Verarbeitungsablauf erst ganz ausführlich und reduzieren dann den Beschreibungsumfang nach und nach in Anpassung an die jeweils tatsächlich verfügbare Hardware (Code-Umformung).

1.2 Charakteristische Anwendungsgebiete

Das ReAl-Programmier- und Verarbeitungsprinzip kann angewendet werden:

- Zu theoretischen Betrachtungen,
- in der Programmierpraxis,
- in der Programmdokumentation,
- beim Wandeln (Konvertieren) von Programmen, die in verschiedenen Programmiersprachen formuliert sind (Ressourcen-Algebra als Metasprache),
- beim Aufbau von Systemen auf Grundlage programmierbarer Logikschaltkreise,
- zur Gestaltung von Prozessor- und Systemarchitekturen.

Theorie
Herkömmliche Programme liegen im Grunde lediglich als Texte (Zeichenketten) vor. Aus der Analyse des Programmtextes allein läßt sich das Verhalten des Programms nur ungenügend voraussagen; man muß das Programm laufen lassen, um seine Wirkungen zu erkennen. Hingegen ist es auf Grundlage des ReAl-Prinzips möglich, die Programmierabsicht in eine (fiktive) Schaltungsstruktur umzusetzen, deren Informationswandlungen bis auf die einzelnen Boolesche Gleichungen aufgelöst werden können. Hierdurch werden formale Korrektheitsbeweise vereinfacht bzw. überhaupt erst durchführbar (Anwendung der Graphentheorie, Automatentheorie, Schaltalgebra usw.). Die Tatsache, daß letzen Endes eine (fiktive) Schaltungsstruktur vorliegt, kann auch beim Testen von Programmen ausgenutzt werden; alle Verfahren – und Tricks – die sich beim Fehlersuchen in Hardware bewährt haben, werden nutzbar (Zerlegen der Gesamt-„Schaltung“ in Blöcke, die einzeln getestet werden, das Zusammenschalten von Testkonfigurationen mit Testdatenerzeugung und Testauswertung, das gezielte Einspeisen von Testmustern usw.). Ebenso wie man zum Fehlersuchen in der Hardware ggf. Signalgeneratoren, Logik-analysatoren usw. zu Hilfe nimmt, kann man in einer ReAl-Maschine entsprechende Prüfhilfen bedafsweise aus den vorhandenen Ressourcen zusammenstellen.

Programmierpraxis
Das ReAl-Prinzip kann auch mit Software verwirklicht werden. Hierzu stehen zwei Prinzipien zur Wahl:

1. Emulation einer entsprechenden (speziellen) Befehlsliste.
2. Erzeugung eines „echten“ Maschinenprogramms für die jeweilige Zielarchitektur (Compilierung).

Nutzung der Emulation:
Vorzugsweise in Embedded Systems. Entsprechende Programme werden sowohl mit üblichen Programmiersprachen als auch mit Entwurfswerkzeugen erstellt, die eine graphische Erfassung der Entwurfsabsicht unterstützen, z. B. auf Grundlage von Blockschaltbildern, Flußdiagrammen und State Machines. Solche Entwurfssysteme erzeugen typischerweise einen Zwischencode in einer gängigen Programmiersprache (C, C++ usw.), der anschließend mittels eines gängigen Compilers in ein Programm für die betreffende Zielmaschine gewandelt wird. Die Ausdrucksmittel der üblichen universellen Programmiersprachen sind aber für die hier in Rede stehenden Anwendungsprobleme nicht besonders geeignet, so daß oftmals ausgesprochen lausiger Maschinencode gebildet wird.

ReAl-Programme beschreiben im Grunde Schaltungsstrukturen, also Hardware. Deshalb können solche Programme auf naheliegende Weise aus Blockschaltbildern, Zustandsdiagrammen usw. abgeleitet werden. An die Stelle eines Zwischencodes (z. B. in C) tritt ein Code einer passend gewählten Ressourcen-Algeba, die auf das jeweilige Anwendungsgebiet hin optimiert ist (z. B. auf das Arbeiten mit Booleschen Gleichungen und Automatentabellen).

Einsatz in Anwendungbereichen mit hohen Anforderungen an die Funktionssicherheit
Der Prozessor (oder Mikrocontroller) interpretiert eine (fiktive) Schaltungsstruktur, die – anders als ein Programmtext – auch im statischen Zustand der Kontrolle und Verifizierung zugänglich ist. Somit können Mikroprozessoren, Mikrocontroller usw. auch dort angewendet werden, wo man bisher – aus Sicherheitsgründen – den Einsatz programmierbarer Einrichtungen ausgeschlossen hat. Der bisherige Zustand: es wird eine Schaltungslösung ausgearbeitet, auf Einhaltung der jeweiligen Vorschriften hin überprüft und schließlich als Hardware gebaut. Die Alternative: Die Schaltung wird mit den Ausdrucksmitteln der ReAl-Architektur beschrieben, die Beschreibung wird vom Prozessor oder Controller emuliert. Ein richtig geschriebener Emulator kann nie abstürzen, ganz gleich, welche Fehler im zu interpretierenden System vorhanden sind. Somit hat die auf ReAl-Prinzipien beruhende Software die gleiche Funktionssicherheit wie eine „echte“ Hardware-Implementierung.

Nutzung beim Erzeugen herkömmlicher Maschinenprogramme (Compilierung):
Vorzugsweise für komplexe Anwendungen, von denen eine hohe Verarbeitungsleistung erwartet wird. Ein typischer Entwicklungsgang: Formulieren der Programmierabsicht (in einer beliebigen Programmiersprache) => Wandlung in den Code einer fiktiven ReAl-Maschine => Umsetzung in den Maschinencode der Zielarchitektur => Programmausführung. Es ist üblich, den Quellcode zunächst in einen fiktiven Maschinencode umzusetzen. Solche fiktive Maschinen sind zumeist als Stackmaschinen ausgelegt. Stackmaschinen arbeiten aber inhärent sequentiell (ein Vorgang zu einer Zeit). Demgegenüber haben ReAl-Maschinen vor allem folgende Vorteile:

- Der innewohnende (inhärente) Parallelismus im Programmablauf kann besser erkannt werden,
- es ist möglich, Gelegenheiten zur Ausnutzung von SIMD-Vorkehrungen[1)] in allgemeinen Programmabläufen zu erkennen (s. folgende Anmerkung),
- Verringerung des Anteils der Organisationsabläufe (Overhead), z. B. beim Aufruf von Funktionen und beim Transportieren von Parametern.

Programmdokumentation
Ein ReAl-Programm kann die Programmierabsicht in allen wesentlichen Einzelheiten beschreiben; wenn es sein muß, bis hin zur einzelnen Booleschen Gleichung. Deshalb ist zu erwarten, daß sich solche Programme ohne weiteres in Maschinencode für künftige Systeme umsetzen lassen (Zukunftssicherheit).

Ein Sonderfall der Programmdokumentation: das Patentwesen
Es handelt sich darum, Patentansprüche zu formulieren, die herkömmlicherweise Programme (= Software) betreffen. Gemäß dem ReAl-Prinzip wird Software auf (fiktive) technische Gebilde (= Hardware) zurückgeführt. Einschlägige Patentansprüche können somit in Hinsicht auf Erfindungshöhe und Neuheit leichter beurteilt werden. (U. a. könnten erfinderische Leistungen damit begründet werden, daß im Vergleich zum Stand der Technik Ressourcen eingespart werden, daß man mit einfacheren Ressourcen auskommt usw.)

Programmkonvertierung; Metasprache
Sämtliche Programme, gleich in welcher Sprache formuliert, sind letzten Endes Steueranweisungen für Informationswandlungen, Transporte und Zustandsübergänge in Register-Transfer-Strukturen. Eine hinreichend (in bezug auf Datentypen, Operationen usw.) ausgestattete Ressourcen-Algebra eignet sich somit als universelles Compiler-Ziel bzw. – aus Sicht der Theorie – als universelle Metasprache, in der alle Ausdrücke der verschiedenen Programmiersprachen wiedergegeben werden können. Tabelle 1.1 veranschaulicht ein Beispiel.

---

1): SIMD = Single Instruction, Multiple Data. Ein einziger Maschinenbefehl veranlaßt die Ausführung mehrerer gleichartiger Operationen (vgl. die MMX- und SSE-Befehle in den Prozessoren der üblichen PCs). Solche Befehlswirkungen werden herkömmlicherweise zur Verarbeitung spezieller Datenstrukturen verwendet (z. B. für Video- und Audiodaten).

C	Basic

for (n=0; n0 ≤ ende; n++)
{
Schleifenkörper
}
	for n=0 to ende
	Schleifenkörper
	next n

ReAl	
s (Iterator, Schleifenkörper)	– Ressourcen für Iterator und Schleifenkörper anfordern
p (0 => Iterator, Ende => Iterator)	– Iterator mit Parametern (Anfangswert, Endwert) versorgen
p (... => Schleifenkörper)	– Ressourcen des Schleifenkörpers mit Parametern versorgen
p (Verzweigungsadressen => Plattform)	– Verzweigunsgadressen einstellen
y (Iterator)	– Iteratorfunktion ausführen
y (Plattform)	– auf Schleifenende testen; ggf. Schleife verlassen
l (aktuellen Iteratorwert => Schleifenkörper)	– Wert n zur Verarbeitung übergeben
y (Schleifenkörper)	– Funktionen des Schleifenkörpers ausführen
y (Plattform)	– zurück zum Iterator

Tabelle 1.1 Einfache Iterationsschleifen im Vergleich. Die ReAl-Notation wird von Kapitel 2 an eingehend erklärt

Anmerkung zu Tabelle 1.1:
Alle drei Formulierungen bewirken das gleiche. Entsprechende Compiler könnten den ReAl-Code aus Texten in C, Basic usw. erzeugen.

Typische Entwicklungsgänge:

- Programm in Programmiersprache A => ReAl-Code => Interpretation des ReAl-Cods auf einer beliebigen Zielmaschine,
- Programm in Programmiersprache A => ReAl-Code => Programm in Programmiersprache B (die Aktivierung der Ressourcen wird durch Sprachkonstrukte mit jeweils gleichartigen Wirkungen substituiert).

Systeme auf Grundlage programmierbarer Logikschaltkreise
Die Entwicklung von Anwendungslösungen auf Grundlage programmierbarer Logikschaltkreise erfordert herkömmlicherweise eine detaillierte Schaltungsentwicklung (bis zum einzelnen Gatter oder Flipflop). Um die Entwicklungsarbeiten zu erleichtern, werden fertige Schaltungen bereitgestellt, die in die eigenen Entwürfe eingebettet werden (IP Cores; IP = Intellectual Property). Hiervon gibt es zwei Arten:

1. „Weiche“ Schaltungen (Soft Cores). Sie liegen als Schaltungsbeschreibungen vor, werden in den Entwicklungsablauf eingebunden und mit den Mitteln des programmierbaren Schaltkreises (Funktionsblöcke, Makrozellen usw.) realisiert.
2. „Harte“ Schaltungen (Hard Cores). Sie sind auf dem Schaltkreis in fester Form (nicht programmierbar) vorhanden.

Die ReAl-Architektur ermöglicht es, komplexe Entwürfe unabhängig von ihrer Realisierung zu beschreiben und je nach Zweckmäßigkeit die eine oder die andere Lösung auszuwählen. Hierbei können auch Hard- und Software gegeneinander ausgetauscht werden.

Herkömmlicherweise hat man solche Probleme durch Einsatz von zwei Sprachen gelöst (Programmiersprache + Hardware-Beschreibungssprache). Dabei wird ein Universalprozessor ggf. durch zusätzliche Hardware ergänzt. Der Universalprozessor bleibt im wesentlichen gleich; es gibt lediglich die Möglichkeit, die Konfiguration in gewisen Grenzen zu ändern (das betrifft z. B. die Größe von Caches und Registersätzen, die Anordnung von Gleitkommaverarbeitungshardware usw.). Dem universellen Prozessor zugeordnete spezielle Hardware wird über Sonderbefehle angesprochen, die man zum Befehlssatz des Prozessors hinzugefügt hat.

Eine entsprechend ausgebildete ReAl-Befehlsliste ist hingegen eine einheitliche Maschinensprache, die sowohl Hardware als auch Software beschreiben kann.

Die üblichen programmierbaren Schaltkreise enthalten im Grunde nur zwei Arten von Schaltmitteln:

1. Universelle Funktionsblöcke, Makrozellen usw., die jeweils nur vergleichsweise einfache kombinatorische Vreknüpfungen ausführen und nur wenige Bits in Flipflops speichern können (Richtwert: 1...4 Flipflops je Zelle). Man kann hier bildhaft von einer „feinen Körnigkeit“ der Schaltmittel sprechen.
2. Harte IP Cores, z. B. komplette Prozessoren. Das entspricht einer „groben Körnigkeit“.

Weiche IP Cores belegen viel Siliziumfläche. Vergleicht man die harte (bis auf den Transistor optimierte) und die weiche Implementierung der gleichen Funktion, so braucht die weiche Ausführung typischerweise mehr als zehnmal soviele Transistoren als die harte. Auch die Geschwindigkeit ist entsprechend geringer (Verhältnis der Taktfrequenzen typischerweise 4:1 bis über 10:1).

Harte Implementierungen sind hingegen teuer (Entwicklungsaufwand) und nicht so flexibel. Man kann sie nicht ohne weiteres mit beliebigen Schaltungen eigener Wahl verbinden, sondern muß die vorgegebenen Schnittstellen (z. B. Bussysteme) ausnutzen. Das erfordert typischerweise zusätzlichen Adaptierungsaufwand.

Gemäß den ReAl-Wirkprinzipien kann man programmierbare Schaltkreise gestalten, die eine mittlere Körnigkeit aufweisen – der „dicke“ harte Prozessor wird gleichsam in seine Bestandteile aufgelöst, die als einzelne Bausteine bereitgestellt werden. Auch kann man die Verbindungsstrukturen in Hinsicht auf die typischen Informationstransporte optimieren. Aus den bereitgestellten Ressourcen (Rechenschaltungen, Adressierungsschaltungen usw.) kann man je nach Bedarf Universalrechner oder Spezialschaltungen aufbauen und die Konfiguration während der laufenden Arbeit ändern.

Prozessor- und Systemarchitekturen
Kennzeichnend ist die Auflösung der Prozessorstrukturen in die einzelnen Funktionseinheiten und der fließende Übergang zwischen Hard- und Software. Bei entsprechender Ausgestaltung (Standardisierung) kann man die festen – proprietären – Prozessorarchitekturen, Betriebssysteme und Anwendungsprogramme durch Ressourcen an sich beliebiger Herkunft ersetzen (aufgelöste Systemarchitektur). Sowohl System- als auch Anwendungsfunktionen werden von Ressourcen erbracht, die beliebiger Herkunft sein und wahlweise als Hardware oder als Software ausgeführt werden können. Wesentlich hierfür ist, daß die ReAl-Befehle nur das Aufrufen, Aktivieren usw. der Ressourcen beschreiben, während die eigentlichen funktionellen Wirkungen im Innern der jeweiligen Ressource erbracht werden. Damit dies in der Praxis wirksam werden kann, ist es erforderlich, Ressourcenbeschreibungen und universelle Befehlscodes (z. B. Bytecodes) umfassend zu standardisieren.

Bisherige Ansätze zur Verwirklichung solcher Vorstellungen beruhen auf höheren formalen Sprachen oder auf virtuellen Maschinen, die sich vergleichsweise einfach emulieren lassen. Verwendet man höhere formale Sprachen, so erfordert der Übergang von einer Plattform auf eine andere oder der Wechsel zwischen Hard- und Software stets eine Neucompilierung. Hierfür ist jeweils ein passender Compiler erforderlich. Da die internen Schnittstellen (z. B. der Parameterübergabe) nicht einheitlich standardisiert sind, gibt es immer wieder Kompatibilitätsprobleme. Verwendet man eine virtuelle Maschine, so lassen sich solche Schwierigkeiten weitgehend vermeiden. Herkömmliche virtuelle Maschinen sind aber vor allem unter dem Gesichtspunkt einer effektiven Compilierung von Software entwickelt worden (Beispiele: P-Code (Pascal), Forth-Maschinen, JVM (Java Virtual Machine)). Sie eignen sich deshalb kaum als universelle Schnittstellen zu komplexer Hochleistungshardware (Parallelverarbeitung, anwendungsspezifische Verarbeitungswerke usw.).

Der innewohnende (inhärente) Parallelismus wird unmittelbar aus der Programmierabsicht heraus erkannt. Somit ist es möglich, ggf. auch hunderte Rechenwerke gleichzeitig einzusetzen.

Speicher und Verarbeitungslogik können direkt miteinander verbunden werden. Im Extremfall ist die einzelne (Hardware-) Ressource eine Speicheranordnung mit Rechenwerk (Ressourcenzelle).

1.3 Zum Stand der Technik

Verarbeitungseinrichtungen (z. B. Prozessoren), die mehrere gleichzeitig nutzbare Ressourcen (z. B. Rechenwerke) enthalten, sind an sich bekannt. Die Hauptschwierigkeit besteht darin, diese Ressourcen so auszunutzen, daß der Ablauf typischer einzelner Programme beschleunigt wird. Programme für übliche Rechnerarchitekturen sind ihrer Natur nach sequentiell; das grundlegende Programmiermodell sieht vor, eine Anweisung nach der anderen auszuführen. Nicht alle Anweisungen müssen aber wirklich nacheinander ausgeführt werden. Beispiel:

1. Befehl: X := A+ B
2. Befehl: Y := C + D

Stehen zwei Rechenwerke zur Verfügung, könnte man beide Befehle gleichzeitig ausführen. Die Tatsache, daß es in üblichen Programmen Befehle und Befehlsfolgen gibt, die gleichzeitig (parallel zueinander) ausgeführt werden können, heißt innewohnender (inhärenter) Parallelismus.

Es gibt verschiedene Möglichkeiten, den innewohnenden Parallelismus zu erkennen und auszunutzen. Die entscheidende Voraussetzung ist die Verfügbarkeit mehrerer Verarbeitungswerke (Superskalarprinzip). Die wesentlichen Unterschiede bestehen darin, wie diese Hardware gesteuert wird. Im Grunde gibt es zwei Prinzipien:

1. Herkömmliche Programmschnittstelle bzw. Befehlsliste (Erkennung zur Laufzeit)
Der einzelne Befehl gibt jeweils eine einzige auszuführende Operation an und veranlaßt somit die Nutzung einer einzigen Verarbeitungsressource. Der innewohnende Parallelismus wird zur Laufzeit erkannt. Hierzu werden mehrere aufeinanderfolgende Befehle gleichzeitig geholt und decodiert. Da beim Schreiben des Programms die Parallelausführung verschiedener Vorgänge nicht berücksichtigt wurde, kann es zu Konflikten kommen. Beispiel:

1. Befehl: X := A + B
2. Befehl: Y := C + X

Werden beide Befehle gleichzeitig ausgeführt, so rechnet der zweite mit dem bisherigen Wert vonn X, liefert also ein falsches Ergebnis. Solche Konflikte werden von besonderen Schaltmitteln erkannt und durch nochmaliges Ausführen der betroffenen Befehle gelöst.

Neben den eigentlichen Verarbeitungsressourcen sind Schaltungen zum Erkennen von Gelegenheiten zur parallelen Verarbeitung und zum Erkennen und Auflösen der Konfliktfälle erforderlich. Der Aufwand hierfür ist vergleichsweise hoch (Abb.n 1.1, 1.2). Deshalb können nur wenige Befehle auf parallele Ausführbarkeit geprüft werden, und die Anzahl der Verarbeitungseinrichtungen läßt sich nicht beliebig erhöhen. Typisch sind zwei bis vier Verarbeitungswerke je Datentyp (ganze Binärzahlen, Gleitkommazahlen usw.). Eine wesentlich größere Anzahl an Verarbeitungswerken würden eine untragbaren Aufwand erfordern, um die möglichen Konfliktfälle zu erkennen.

Da im Konfliktfall die Befehlsausführung wiederholt werden muß, sinkt in der Praxis die Verarbeitungsleistung. Zudem ist es (wegen des Steuerungs- und Überwachungsaufwandes) üblich, nur elementare Befehle auf diese Weise zu unterstützen; Befehle mit komplizierteren Wirkungen werden oftmals nach wie vor nur einzeln ausgeführt.

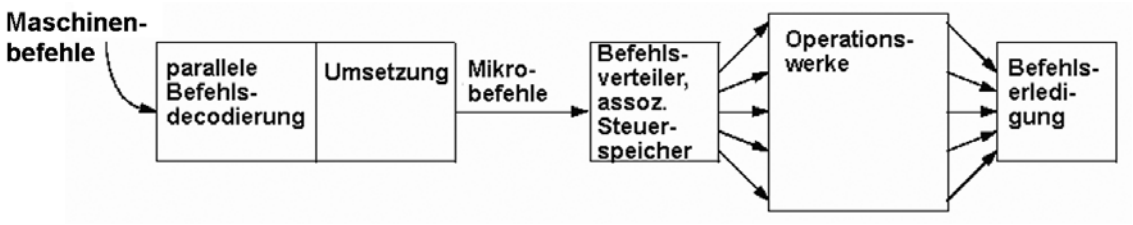
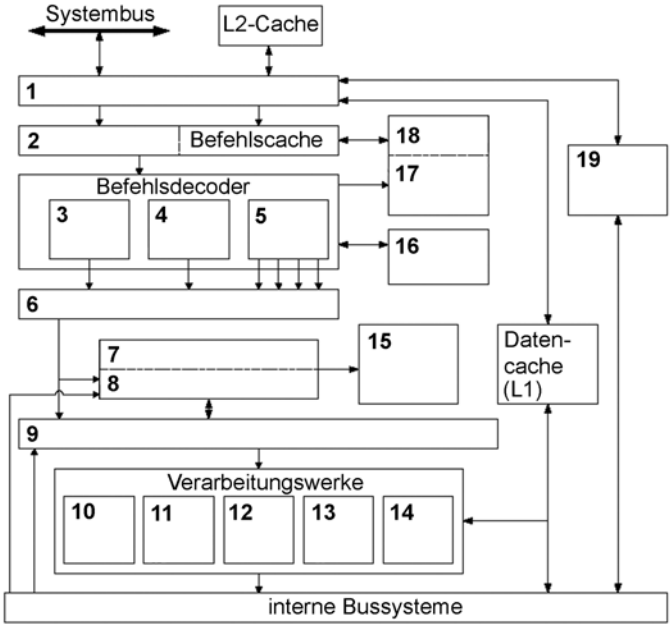


Abb. 1.1 Überblick über den Aufbau einer Superskalarmaschine

Moderne Hochleistungsprozessoren, wie sie beispielsweise in den üblichen Personalcomputern eingesetzt sind, arbeiten typischerweise folgendermaßen:

- Mehrere herkömmliche Maschinenbefehle werden gleichzeitig gelesen und decodiert,
- sie werden in Mikrobefehle umgesetzt,
- die Mikrobefehle werden in einem assoziativen Steuerspeicher (Reordering Buffer) gepuffert und von einem Verteiler zu den Operationswerken geführt,
- ein Mikrobefehl wird dann ausgeführt, wenn ein passendes Operationswerk frei ist,
- die Operationen laufen ab, ohne die ursprüngliche Verarbeitungsreihenfolge zu berücksichtigen,
- werden Konflikte erkannt, so werden die betreffenden Befehle so oft wiederholt, bis die Konflikte verschwunden sind,
- die Befehlserledigung (Instruction Retirement) stellt die Ergebnisse der konfliktfrei zu Ende gebrachten Befehle so bereit, daß die Ergebnisse der Befehlsausführung dem Programmierer gegenüber so erscheinen, wie sie sich gemäß der ursprünglichen Verarbeitungsreihenfolge ergeben hätten.

Bei dieser Form der Parallelarbeit handelt es sich also gleichsam um eine Art Probieren. Dabei ist der Bereich, in dem versucht wird, parallele Abläufe auszuführen, auf die Anzahl der Befehle beschränkt, die gleichzeitig geholt und decodiert werden können. Der Steuerungsaufwand ist vergleichsweise hoch.



1 - Busanschlußsteuerung; 2 - Befehlsleseeinheit; 3, 4 - Befehlsdecodierung für einfache Befehle; 5 - Befehlsdecodierung für komplexe Befehle; 6 - Registerzuordnungseinheit; 7 - Befehlserledigung; 8 - Mikrobefehls-Assoziativpuffer (Reordering Buffer); 9 - Mikrobefehlsabruf; 10, 11 - Gleitkommaverarbeitungswreke; 12, 13 - Verarbeitungswerke für ganze Binärzahlen; 14 - Speicherzugriffssteuerung; 15 - architekturseitiger Registersatz; 16 - herkömmliche Mikroprogrammsteuerung (steuert alles, was für die Parallelausführung zu kompliziert ist); 17 - Sprungzielpuffer; 18 - architekturseitiger Befehlszähler; 19 - Speicherzugriffspuffer.

Abb. 1.2 Blockschaltbild eines Superskalarprozessors (nach Intel)

ReAl-Maschinen können auf Grundlage solchen Prozessoren ausgeführt werden (Abb. 1.3). Dabei können die komplizierten Steuereinrichtungen entfallen. Es verbleiben die Verarbeitungswerke, die Caches und Puffer sowie die Busschnittstellen. Der Befehlsdecoder ist wesentlich einfacher. Der Universalregistersatz kann gegenüber herkömmlichen Prozessoren beträchtlich erweitert werden (z. B. auf 64 bis 256 Register). Die Verarbeitungswerke können mit dem Universalregistersatz direkt gekoppelt werden. Da die komplizierten Steuerschaltungen entfallen, könnten ggf. die Verarbeitungsfunktionen erweitert oder zusätzliche Verarbeitungswerke vorgesehen werden.

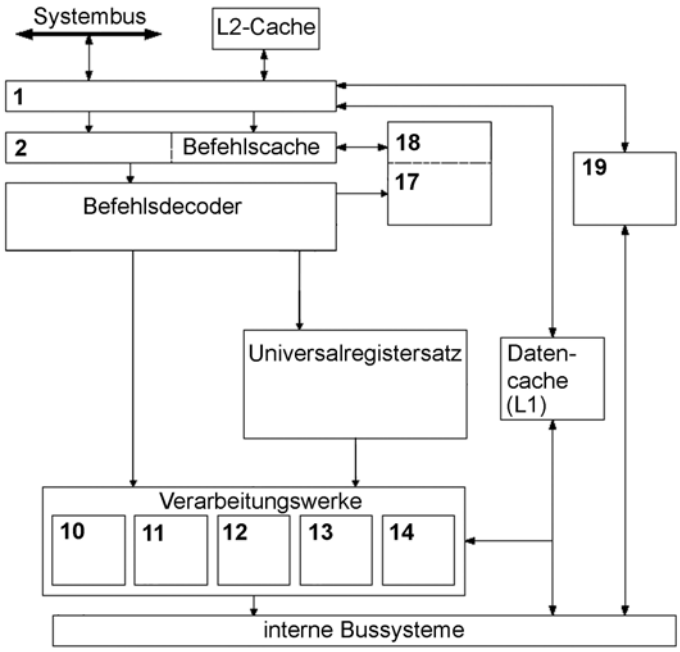


Abb. 1.3 Umwandlung eines herkömmlichen Superskalarprozessors in eine ReAl-Maschine. Die Bezugszeichen entsprechen Abb. 1.2

Auf Grundlage der ReAl-Architektur kann man die perspektivischen Möglichkeiten, die die Schaltungsintegration bietet (z. B. einige hundert Millionen Transistoren auf einem Schaltkreis), in weitem Umfang ausnutzen. Herkömmliche Hochleistungsprozessoren (z. B. ähnlich Abb. 1.2) haben rund 10...50 Millionen Transistoren. Auf einem Schaltkreis mit 200 Millionen Transistoren ließen sich z. B. vier Prozessoren zu jeweils ca. 50 Millionen Transistoren unterbringen. Das Leistungsvermögen dieser Anordnung kann aber nur dann in der Anwendungspraxis wirksam werden, wenn wenigstens vier Programme gleichzeitig auszuführen sind; das einzelnen Programm kann in sich nicht beschleunigt werden. Zerlegt man einen Prozessor der hier in Rede stehenden Art in seine Funktionseinheiten, so entsprechen die Verarbeitungseinrichtungen 4...8 universellen Rechenwerken. Caches, Steuerschaltungen usw. sollen (ihrem Umfang nach) beibehalten werden (gleicher Aufwand, nur abgewandelte Struktur). Ein Schaltkreis mit 200 Millionen Transistoren könnte also 32 universelle Rechenwerke enthalten, die gemäß den ReAl-Prinzipien als Ressourcen verwaltet werden und somit jedem einzelnen Programm zugute kommen können.

2. Befehle mit Angaben zur Steuerung der Parallelarbeit (Erkennung zur Compilierzeit)
Es gibt verschiedene Ausführungen, z. B. extrem lange Befehle mit Steuerangaben für alle Ressourcen (VLIW = Very Long Instruction Word) oder Befehle, in denen angegeben werden kann, ob nachfolgende Befehle parallel ausführbar sind oder nicht (expliziter Parallelismus). Überwachungsvorkehrungen in der Hardware sind nicht erforderlich (Schaltungsvereinfachung). Die Anzahl der so unterstützten Ressourcen ist aber beschränkt (Befehle können nicht beliebig lang werden) und typischerweise in der jeweiligen Architektur festgelegt (z. B. auf drei, vier oder acht Rechenwerke). Die tatsächliche Verarbeitungsleistung hängt wesentlich vom Compiler ab, der aus dem Quelltext heraus die Programmierabsicht erkennen und festlegen muß, wie die vorhandenen Ressourcen am besten ausgenutzt werden. Ein derart optimiertes Maschinenprogramm ist auf Hardware mit abweichender Ausstattung (z. B. acht statt drei Rechenwerke) nicht ohne Neucompilierung lauffähig.

1.4 Elementare Ressourcen

Die Register-Transfer-Struktur (Register Transfer Level RTL)
Jede digitale informationsverarbeitende Einrichtung läßt sich auf Speichermittel (Flipflops, Register, Speicherzellen) und kombinatorische Schaltungen (Funktionszuordner) zurückführen (Abb. 1.4); sie ist durch die Speichermittel und durch die Booleschen Gleichungen, die die Wirkungsweise der Funktionszuordner beschreiben, in funktioneller Hinsicht vollständig bestimmt. Die Ressourcen können deshalb

ohne Beschränkung der Allgemeinheit durch einfache Register-Transfer-Darstellungen veranschaulicht werden. Die Booleschen Gleichungen, die die in der Informatik üblichen grundlegenden Informationswandlungen beschreiben, gehören zum allgemeinen Fachwissen. Es gibt eigens formale Entwurfssprachen, um entsprechende Register-Transfer-Strukturen zu erfassen.

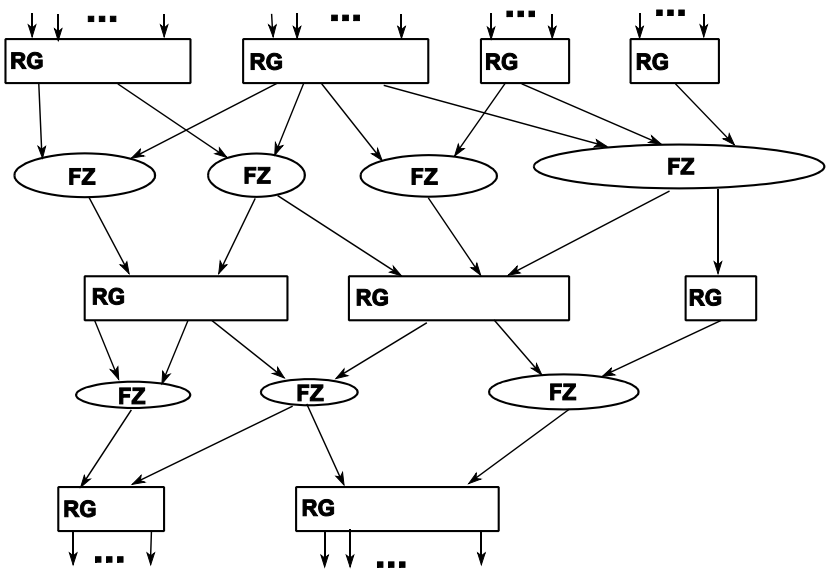


Abb. 1.4 Die Register-Transfer-Struktur. RG = Register; FZ = Funktionszuordner (kombinatorische Schaltungen)

Ressourcen als Register-Transfer-Strukturen
Einfache Ressourcen bestehen aus Registern, die die Operanden und die Ergebnisse aufnehmen, sowie aus zwischengeordneten kombinatorischen Schaltungen (Funktionszuordnern).

Elementare Ressourcen (Abb. 1.5a) bilden beispielsweise aus zwei Operanden (A, B) ein einziges Ergebnis (X):

X := A **OP** B bzw. X := **OP** (A, B)

Die meisten Verarbeitungsbefehle der typischen Universalrechner entsprechen diesem Schema (die Unterschiede liegen vor allem in der Art und Weise der Bereitstellung der Operanden und der Zuweisung des Ergebnisses). Die üblichen Rechenwerke und Arithmetik-Logik-Einheiten (ALUs) können als Beispiele solcher elementaren Ressourcen angesehen werden.

Die üblichen Universalrechner kennen nur wenige elementare Datentypen, z. B. ganze Binärzahlen, Gleitkommazahlen, Zeichen usw., wobei zumeist jeweils mehrere Formate unterstützt werden (z. B. 16, 32 und 64 Bits). Ein Rechenwerk verarbeitet meist nur Daten eines bestimmten Typs (z. B. ganze Binärzahlen oder Gleitkommazahlen). Bei elementaren Verknüpfungen haben Operanden und Ergebnis das jeweils gleiche Format.

Im allgemeinen Sinne gibt es keine solchen Einschränkungen. Eine Ressource kann aus beliebig vielen Operanden beliebig viele Ergebnisse bilden, wobei Operanden und Ergebnisse beliebigen Datentypen und -formaten entsprechen dürfen (Abb. 1.5b, 1.6). Es gibt auch keine Beschränkung auf elementare Datentypen. Die Datentypen dürfen vielmehr beliebig komplex sein (Bit- und Zeichenketten, Felder (Arrays), heterogene Strukturen (Records) usw.).

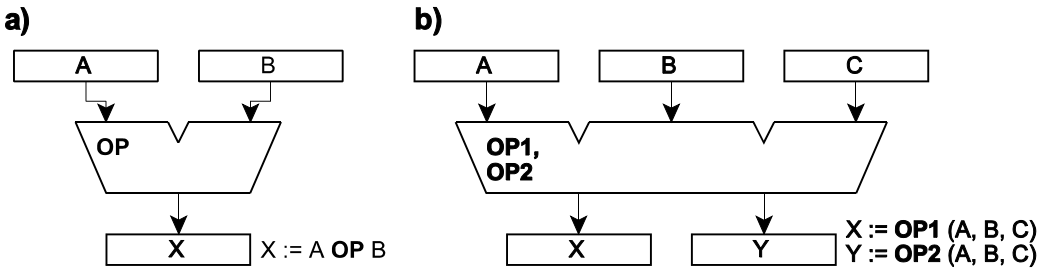


Abb. 1.5 Einfachste Modelle einer Ressource. a) aus zwei Operanden entsteht ein Ergebnis; b) aus drei Operanden werden zwei Ergebnisse gebildet

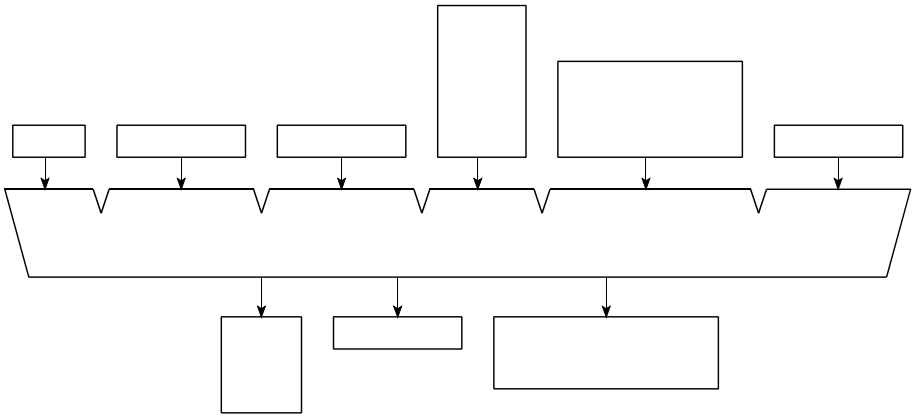


Abb. 1.6 Ressourcen können beliebig kompliziert sein (beliebig viele Operanden und Ergebnissse beliebiger Art)

Der typische herkömmliche Universalrechner führt einen Befehl zu einer Zeit aus (Einzelprozessor). Hierzu genügt eine einzige Verarbeitungsressource. Es liegt nahe, durch Anordnung mehrerer Verarbeitungsressourcen die Verarbeitungsleistung zu steigern. Mehrere Ressourcen, die voneineinander unabhängig sind (Abb. 1.7), versprechen größtmögliche Vielseitigkeit. Es bleibt aber das Problem, sie mit Operanden (Parametern) zu versorgen und die Ergebnisse abzutransportieren.

Eine Lösung besteht darin, die Ressourcen gemäß den am häufigsten vorkommenden Datenflüssen so miteinander zu verbinden, daß ein bestimmtes Ergebnis der einen Ressource unmittelbar zum Operanden anderer Ressourcen werden kann (Verkettung; Abb. 1.8).

Eine alternative Lösung sieht den Anschluß der Ressourcen an einen Speicher mit wahlfreiem Zugriff (RAM) vor (Abb. 1.9). Der einzelne Verarbeitungsablauf unterteilt sich dann in drei Zeitabschnitte:

1. Transport der Operanden zu den Ressourcen,
2. Verarbeitung innerhalb der Ressourcen (in allen gleichzeitig),
3. Transport der Ergebnisse in den Speicher.

Ressourcen als Hard- oder Software
Den Operanden- und Ergebnisregistern der Hardware entspricht ein Speicherbereich mit Speicherzellen für die Parameter und Ergebnisse, den kombinatorischen Netwerken (Funktionszuordnern) entsprechen Programmabläufe, die die jeweiligen Informationswandlungen ausführen (Abb. 1.10). Um die Operationen der Ressourcen softwareseitig nachzubilden (Emulation), sind oft zusätzliche Arbeitsbereiche erforderlich.

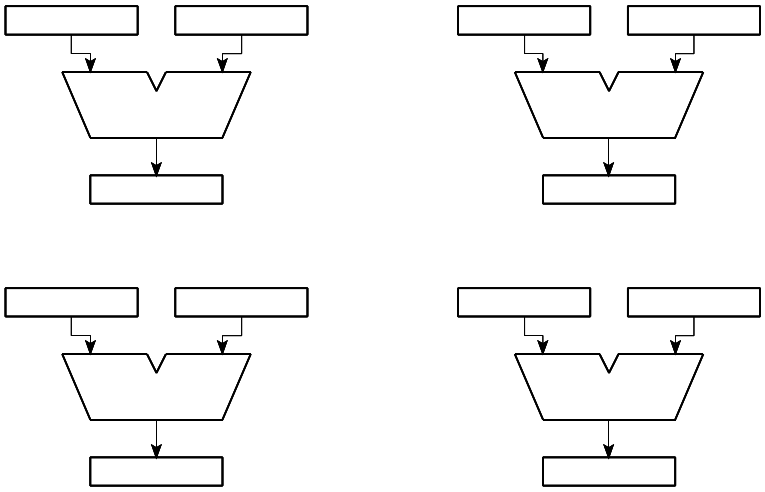


Abb. 1.7 Mehrere unabhängige Ressourcen

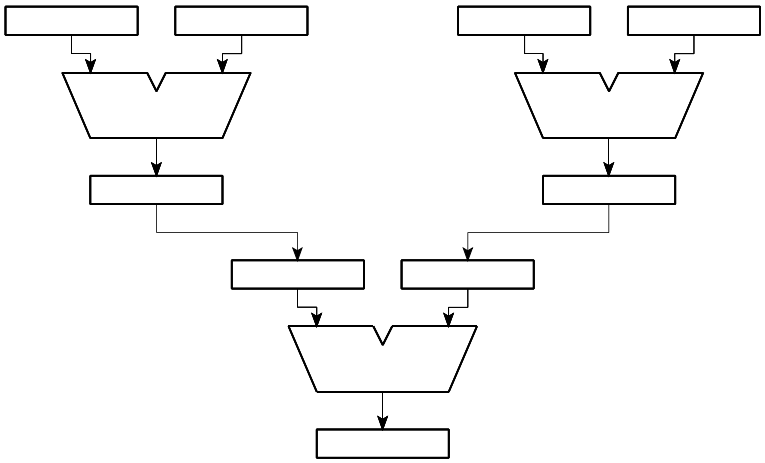


Abb. 1.8 Miteinander verbundene (verkettete) Ressourcen

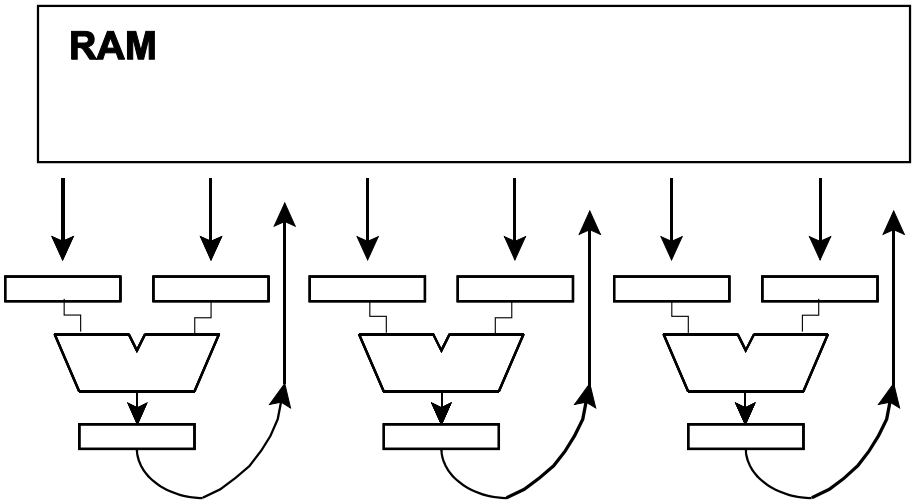


Abb. 1.9 Mehrere uanbhängige Ressourcen werden von einem Speicher aus mit Operanden versorgt. Die Ergebnisse werden in den Speicher zurückgeschrieben

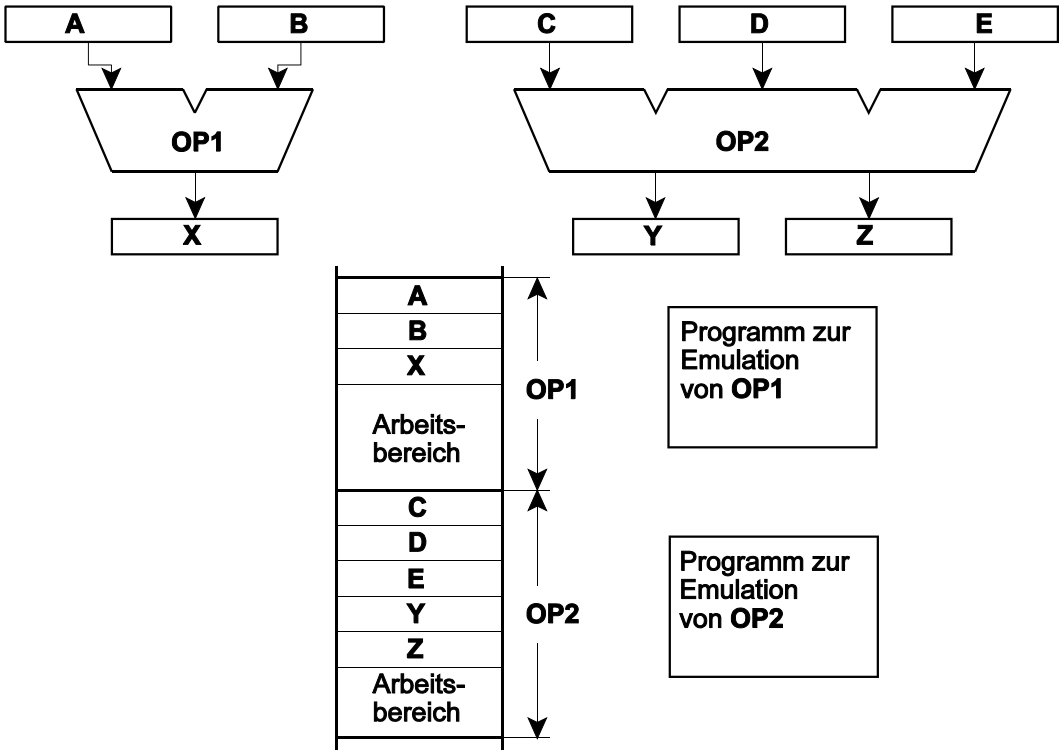


Abb. 1.10 Wechselseitiger Austausch von Hard- und Software (anhand von zwei Beispielen)

2. Das ReAl-Verarbeitungsmodell

2.1 Operatoren und Parameter

Das Verarbeitungsmodell der ReAl-Architektur ist durch folgende Abläufe gekennzeichnet (Abb. 2.1):

1. Ressourcen anfordern bzw. auswählen,
2. mit Parametern (Operanden) versorgen,
3. Operationen ausführen (Hardware oder Programm laufen lassen),
4. Ergebnisse abholen,
5. über weiteren Gang der Verarbeitung entscheiden,
6. Ressourcen wieder zurück- bzw. freigeben.

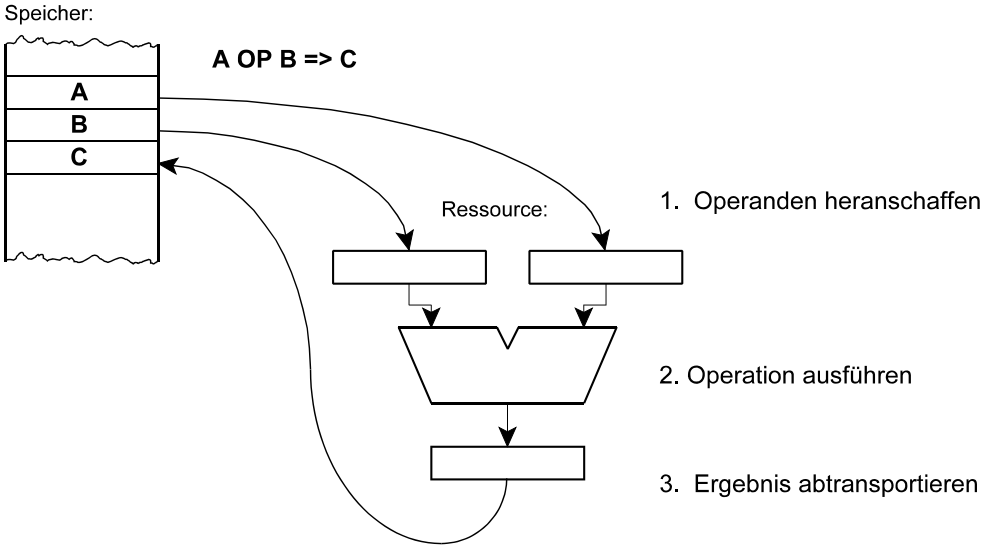


Abb. 2.1 Das grundlegende Verarbeitungsmodell: so wird eine Ressource genutzt

Betrachtet man nur eine einzige, in Parameterzahl und Wirkungsweise einfache Ressource, so gibt es keinen grundsätzlichen Unterschied zu den herkömmlichen Rechnerarchitekturen (Abb. 2.2). Es ist lediglich die Frage, ob die Vorgänge (das Heranschaffen der Operanden usw.) implizit gesteuert werden (Befehlsablaufsteuerung) oder ob sie von einzeln codierten Ablaufsteuerangaben ausgelöst werden. In diesem Sinne hat die ReAl-Achitektur eine gewisse Ähnlichkeit mit der (vertikalen) Mikroprogrammsteuerung. Die Vorteile kommen dann zur Wirkung, wenn man (1) mit mehr als einer Ressource arbeitet und wenn man (2) das Prinzip auf den gesamten Programmablauf anwendet.

Verkettung
Funktionseinheiten der Hardware können zu komplexeren Anordnungen zusammengeschaltet werden. Eine solche (spezielle) Hardware entspricht dem Datenflußschema der jeweiligen Informationswandlungen (Abb. 2.3). Sinngemäß soll es möglich sein, Ressourcen zu solchen Datenflußschemata zu verketten und die Verkettung wieder zu trennen.

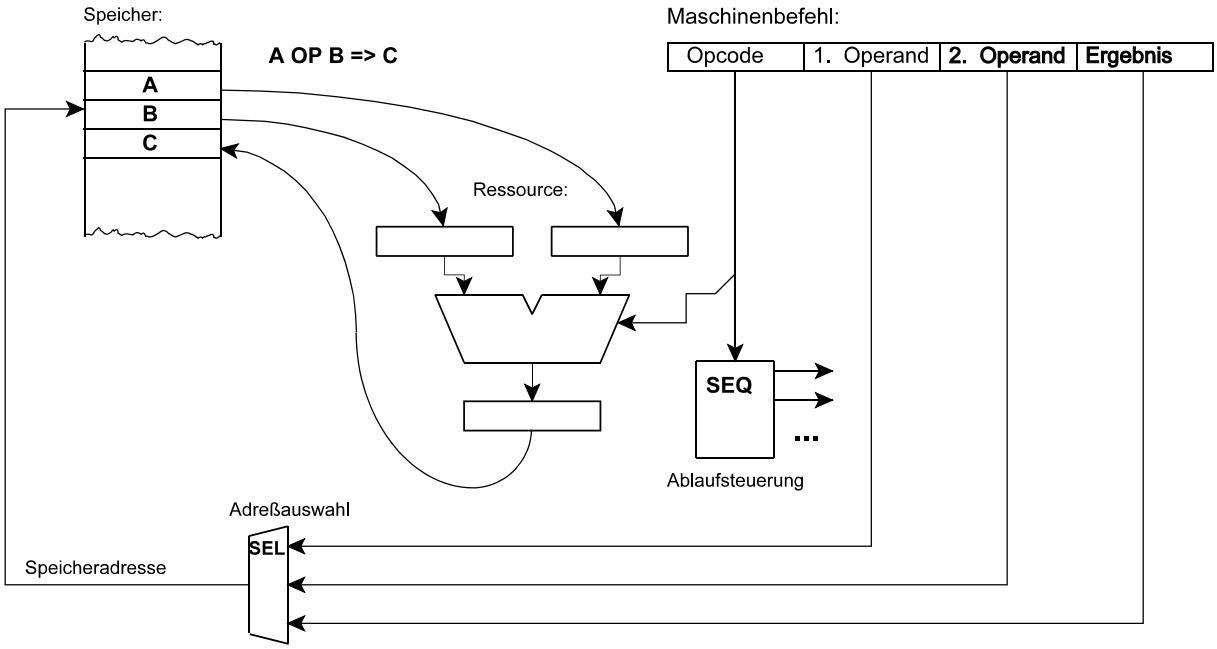


Abb. 2.2 Zum Vergleich: ein elementarer Verarbeitungsablauf in einem herkömmlichen Universalrechner (hier: einer Dreiadreßmaschine)

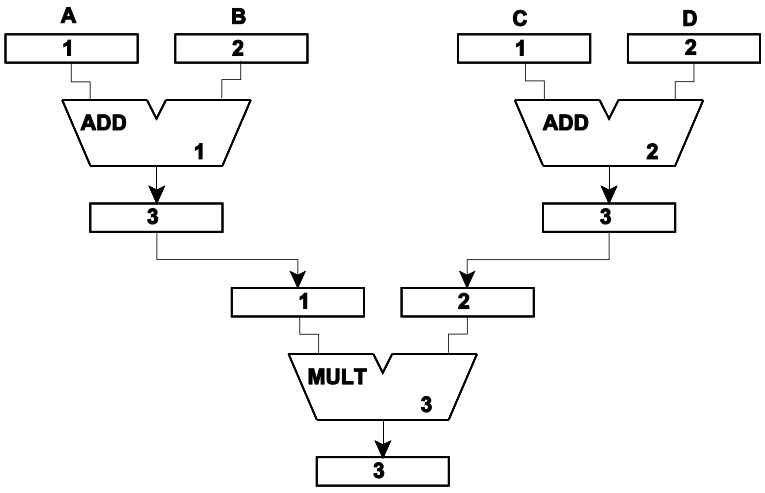


Abb. 2.3 Verkettete Ressourcenanordung zum Berechnen von (A + B) * (C + D)

Die elementaren Operatoren:

1. Ressourcen auswählen = s-Operator (select)
2. mit Parametern versorgen = p-Operator (parameter)
3. Operation(en) ausführen (laufen lassen): y-Operator (yield) = umgekehrtes Lambda
4. Ergebnisse abholen = a-Operator (assign)
5. Transportieren von Parametern zwischen Ressourcen = l-Operator (link)
6. Netzlistenbeschreibung, Verkettung, Datenflußprinzip = c-Operator (connect)
7. Verkettungen trennen = d-Operator (disconnect)
8. Ressourcen wieder zurück- bzw. freigeben = r-Operator (release)

Weitere Operatoren:

1. Hinweise = h-Operator
2. metasprachliche Angaben = m-Operator
3. Verwaltungsfunktionen = u-Operator (Utility)

Parameter:

1. Eingänge (Operanden; IN-Typ)
2. Ausgänge (Ergebnisse; OUT-Typ)
3. Ein- und Ausgänge (INOUT-Typ)

Parameterübergabe
Grundsätzlich als Wert (by Value). Ist das nicht ohne weiteres möglich, sind zusätzliche Ressourcen vorzusehen, die die Werte heranschaffen und abtransportieren.

2.2 Notation von ReAl-Verarbeitungsabläufen

Notationsweisen im Überblick:

1. umgangssprachliche Notation. Stil “Kochbuch” (“man nehme ein Addierwerk” usw.). Zu weitschweifig.
2. ReAl-Textcode (Zeichenketten). Formalisierte Notation. Die Ressourcen, Parameter usw. werden durchnumeriert (Ordinalzahlen). Alternativ dazu können symbolische Bezeichner angegeben werden.
3. ReAl-Bytecode. Eine kompakte, binär codierte Darstellung; aus dem Textcode abgeleitet.
4. ReAl-Maschinencode. Binär. Maschinenspezifisch.

Wir verwenden im folgenden den ReAl-Textcode.

Kommentare:
Mit -- eingeleitet (vgl. Programmiersprache ADA).

Kennzeichnung der Parameter im Textcode
Durch einfaches Durchnumerieren: zuerst die Eingänge, dann die Ein- und Ausgänge, dann die Ausgänge (Abb. 2.4).

Symbolische Bezeichner
Anstelle der Ordinalzahlen können symbolische Bezeichner (Eigennamen) angegeben werden.

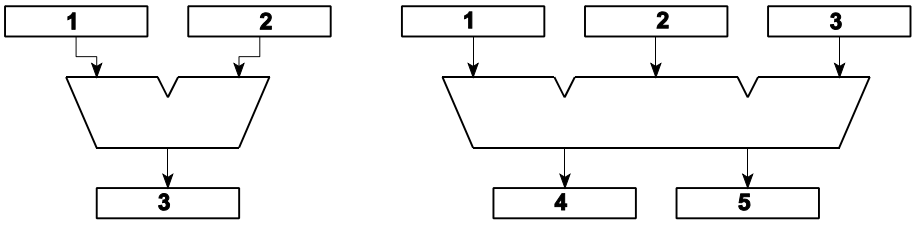


Abb. 2.4 Ressourcen mit durchnumerierten Parametern

Kurzüberblick über die hier verwendete Syntax des Textcodes:

Ressourcentyp := Ordinalzahl oder Eigenname

Ressource := Ordinalzahl oder Eigenname

Parameter := Ordinalzahl oder Eigenname

Zuweisung := => (Pfeil symbolisiert Übertragungsrichtung)

Zuordnung eines Eigennamens := Ordinalzahl : Eigenname

Parameter einer bestimmten Ressource: Ressource . Parameter

Leerzeichen können beliebig eingefügt oder weggelassen werden.

Zuordnung von Ordinalzahlen:

- Ressourcentypen: gemäß Reihenfolge in der Ressourcentyptabelle,
- Parameter: gemäß Reihenfolge in der Parameterliste des Ressourcentyps,
- Ressourcen: gemäß Reihenfolge der Ressourcentypangaben im s-Operator oder gemäß besonderen Zuweisungsangaben.

Näheres zu den Tabellenstrukturen in Kapitel 6.

Hinweis zur Numerierung der Ressourcen:
Die folgenden Beispiele beziehen sich auf eine fortlaufende Durchnumerierung (1, 2 usw.) gemäß der Reihenfolge in den s-Operatoren. Die hierbei vergebenen laufenden Nummern (Ordinalzahlen) bleiben erhalten, auch wenn zwischenzeitlich Ressourcen mit niedrigeren Nummern zurückgegeben wurden (r-Operator). Einzelheiten, Spitzfindigkeiten und Varianten in Kapitel 6.

Zuordnung von Eigennamen:

- Ressourcentypen: vorgegeben (gemäß Ressourcentyptabelle oder Systemhandbuch). Kann bedarfsweise überladen werden (u-Operatoren *resource name/rename* (rn, rrn)):

  u_rn (Ordinalzahl : Eigenname)

  u_rrn (bisheriger Eigenname : neuer Eigenname)

- Parameter: vorgegeben (gemäß Ressourcentyptabelle oder Systemhandbuch). Kann bedarfsweise überladen werden (u-Operatoren *parameter name/rename* (pn/prn)).

  u_pn (Ordinalzahl : Eigenname)

  u_prn (bisheriger Eigenname : neuer Eigenname)

- Ressourcen: im s-Operator (Ressourcentyp : Eigenname).

2.3 Elementare Operatoren

Ressourcen auswählen: s-Operator:
s (1. Ressourcentyp, 2. Ressourcentyp usw.).

Mit dem s-Operator werden Ressourcen angefordert. Bei allgemein üblichen (generischen) Ressourcen wird der jeweilige Typ angegeben, bei speziellen Ressourcen der jeweilige Bezeichner (Eigenname). Im allgemeinen Fall kommen als Bezeichner auch Internet-Adressen usw. in Betracht.

Die angeforderten Ressourcen werden fortlaufend durchnumeriert. Nachfolgende Operatoren beziehen sich dann auf die so vergebenen Ordinalzahlen bzw. Bezeichner.

In einer weiteren Auslegung können die s-Operatoren auch explizite Angaben der Bezeichner, Ordinalzahlen oder Adressen enthalten, die den angeforderten Ressourcen zugewiesen werden:

s (1. Ressourcentyp => 1. Ressourcennummer, 2. Ressourcentyp => 2. Ressourcennummer usw.).

Ressourcen mit Parametern versorgen: p-Operator:
p (1. Variable => Ressource . Parameter, 2. Variable => Ressource . Parameter usw.)

Mit p-Operatoren wird angegeben, welche Variable in welche Parameterposition welcher Ressource transportiert wird. Die Variablen werden durch Namen, Ordinalzahlen oder Adressen bezeichnet. Anstelle von Variablen können auch Direktwerte angegeben werden.

Operation(en) ausführen: y-Operator:
y (1. Ressource, 2. Ressource usw.)

Der y-Operator löst die Verarbeitung in den angegebene Ressourcen aus. Was in den Ressourcen jeweils geschieht, ergibt sich entweder direkt aus der Art der Ressource (falls diese nur eine einzige Funktion ausführen kann) oder aus Parametern (Funktionscodes), die vorher einzustellen sind (z. B. mit s- oder p-Operatoren).

Ergebnisse abholen: a-Operator:
a (1. Ressource . 1. Ergebnis => 1. Ergebnisvariable, 2. Ressource . 2. Ergebnis => 2. Ergebnisvariable usw.)

Der a-Operator weist die Inhalte der angegebenen Ergebnispositionen der angegebenen Ressourcen den angegebenen Variablen zu. Die Variablen werden durch Namen, Ordinalzahlen oder Adressen bezeichnet.

Transportieren von Parametern (vom Ausgang zum Eingang): l-Operator:
l (1. Quell-Ressource . 1. Ergebnis => 1. Ziel-Ressource . 1. Parameter, 2. Quell-Ressource . 2. Ergebnis => 2. Ziel-Ressource . 2. Parameter usw.)

Der l-Operator bewirkt den Transport von Parametern zwischen verschiedenen Ressourcen.

Verkettung herstellen (vom Ausgang zum Eingang): c-Operator:
c (1. Quell-Ressource . 1. Ergebnis => 1. Ziel-Ressource . 1. Parameter, 2. Quell-Ressource . 2. Ergebnis => 2. Ziel-Ressource . 2. Parameter usw.)
Der c-Operator trägt Verkettungssteuerangaben in die jeweiligen Ressourcen ein. In manchen Implementierungen (z. B. in FPGAs) kann er veranlassen, daß die entsprechenden physischen Verbindungen hergestellt werden. Die Ressourcen sind zu verketten, bevor die betreffenden Verarbeitungsfunktionen ausgelöst werden (y-Operator).

Verkettung trennen: d-Operator:
d (1. Quell-Ressource . 1. Ergebnis => 1. Ziel-Ressource . 1. Parameter, 2. Quell-Ressource . 2. Ergebnis => 2. Ziel-Ressource . 2. Parameter usw.)

Der d-Operator trennt bestehende Verkettungen. In manchen Implementierungen (z. B. in FPGAs) kann er veranlassen, daß die entsprechenden physischen Verbindungen aufgelöst werden. Anschließend können die Ressourcen anderweitig verkettet oder auch einzeln betrieben werden.

Ressource(n) freigeben: r-Operator:
r (1. Ressource, 2. Ressource usw.)

Die angegebenen Ressourcen werden freigegeben. Sie stehen somit für andere Verarbeitungsaufgaben zur Verfügung

Beispiel:

- Programmierabsicht: X := (A + B) * (C + D)
- nutzbare Ressourcentypen: ADD, MULT

Abb. 2.5 veranschaulicht die Ressourcenanordnung. Zur Verkettung vgl. Abb. 2.3.

Ausführliche Notation (jeder Vorgang einzeln):

s (ADD, ADD, MULT)
p (A => 1.1)
p (B => 1.2)
p (C => 2.1)
p (D => 2.2)
y (1)
y(2)
l (1.3 => 3.1)
l (2.3 => 3.2)
r (1, 2)
y (3)
a (3. 3 => X)
r (3)

Verkürzte Notation:

s (ADD, ADD, MULT)
p (A => 1.1, B => 1.2, C => 2.1, D => 2.2)
y (1, 2)
l (1.3 => 3.1, 2.3 => 3.2)
r (1, 2)
y (3)
a (3.3 => X)
r (3)

Als Datenflußschema (Verkettung):

s (ADD, ADD, MULT)
c (1.3 => 3.1, 2.3 => 3.2)
p (A => 1.1, B => 1.2, C => 2.1, D => 2.2)
y (1, 2, 3)
a (3.3 => X)
r (1, 2, 3)

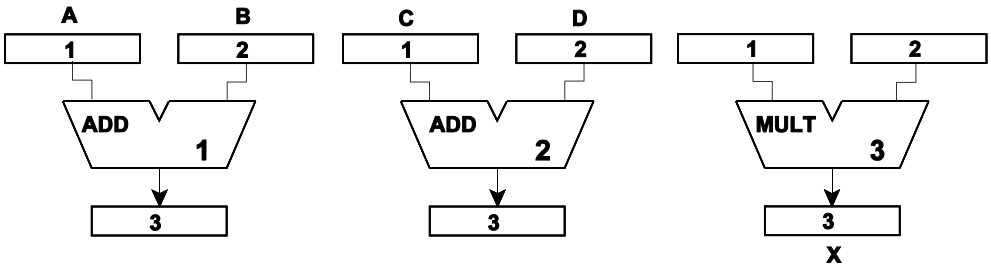


Abb. 2.5 Die Ressourcenanordnung des Beispiels

2.4 ReAl und gegebene Programmiersprachen

ReAl ist zunächst eine maschinenunabhängige Zwischensprache (ähnlich dem bekannten Java-Bytecode). Das Ziel besteht aber nicht darin, mit möglichst wenigen Bytes auszukommen (Code-Kompression), sondern darin, wesentliche Feinheiten des Verarbeitungsprozesses genau zu beschreiben. ReAl wäre in diesem Sinne beispielsweise mit PostScript zu vergleichen. Typische Ziele:

- Den innewohnenden (inhärenten) Parallelismus bereits in statu nascendi zu erfassen,
- so zu tun, als ob es immer genügend Hardware gäbe. Damit ist der ReAl-Code invariant gegen die technologische Entwicklung.
- so zu tun, als könne man ohne weiteres spezielle Hardware bauen (durch Verkettung usw.). Somit sind je nach den Kosten- und Leistungszielen und nach dem Stand der Technologie Hard- und Software gegeneinander austauschbar.

Nutzungs-Szenarien:

- herkömmliche Programmiersprache => ReAl-Beschreibung,
- neuartige (ressourcenorientierte) Programmiersprache => ReAl-Beschreibung,
- ReAl-Beschreibung => ReAl Interpreter auf gegebener Maschine,
- ReAl-Beschreibung => compilierter Code für gegebene Maschine,
- ReAl-Beschreibung => Code für ReAl-Maschine.

Die hier verwendeten Notationen sind an sich zur maschineninternen Nutzung gedacht. Deshalb wird auf Anwendungsfreundlichkeit usw. kein Wert gelegt – es geht nicht um eine neue Programmiersprache für Anwendungsprogrammierer. Der ReAl-Textcode soll vielmehr kurz sein, um entsprechende Darstellungen programmtechnisch effektiv auswerten zu können (Durchmusterung, Analyse, Wandlung, Übersetzung usw.).

ReAl-Programme sind im Grunde Herstellungsanweisungen („man nehme ... und führe folgende Schritte aus ...“). Sie eignen sich also nur wenig zum Erfassen von Programmierabsichten (Gleichnis aus dem Maschinenbau: man wird eine Drehbank oder einen Verbrennungsmotor zunächst zeichnerisch darstellen und nicht sofort – unmittelbar vom Erfindungsgedanken ausgehend – eine Fertigungsdokumentation ausarbeiten).

Die Standardisierung von Ressourcen und einschlägigen Beschreibungsmitteln ist vor allem für die Nutzung als universelle Metasprache von Bedeutung.

2.5 ReAl-Maschinen

Speichermittel, Verarbeitungsschaltungen und Steuerschaltungen werden als Ressourcen aufgefaßt. Wie die einzelnen Ressourcen ausgeführt sind, ist im Grunde gleichgültig. In diesem Sinne sind auch die herkömmlichen Maschinen ReAl-Maschinen, wenngleich mit einer oftmals bescheidenen und den Anwendungsproblemen nicht immer angemessenen Ressourcenausstattung.

Um die jeweiligen Verarbeitungsaufgaben zu lösen, sind die Ressourcen in aufeinanderfolgenden Zeitschritten auszuwählen, mit Parametern zu versorgen, zu aktivieren, wieder freizugeben usw. Diese Vorgänge werden von gespeicherten Anweisungen (dem ReAl-Maschinencode) gesteuert.

Speichermittel enthalten Speicherzellen, die Anweisungen oder Daten aufnehmen. Es gibt keine Einschränkungen hinsichtlich des Zugriffsprinzips. Typische Zugriffsprinzipien sind: (1) verschiedene Arten der Adressierung, (2) die assoziative Auswahl.

Abb. 2.6 veranschaulicht zwei Arten der herkömmlichen Adressierung. Im einfachsten Fall (Abb. 2.6a) wird eine Adreßangabe (absolute Adresse) an den Speicher angelegt, um die betreffende Speicherposition auszuwählen. Die so adressierten Angaben müssen aber stets in den gleichen Speicherpositionen untergebracht werden. Es ist nicht möglich, Programme und Datenbereiche je nach der aktuellen Speicherbelegung an anderen Positionen unterzubringen. Um diese sog. Verschieblichkeit (Relocatability) zu gewährleisten, werden Vorkehrungen zur Adreßrechnung eingeführt. Eines der einfachsten Prinzipien besteht darin, zu einer Basisadresse eine Versatzangabe (Displacement, Offset) zu addieren (Abb. 2.6b). Die Basisadresse zeigt auf den Beginn des jeweiligen Speicherbereichs, die Versatzangabe auf die einzelne Speicherposition. Die Basisadresse wird von der Systemsoftware eingestellt. Die Versatzangaben beziehen sich auf den Beginn des Speicherbereichs (dessen erste Speicherposition hat das Displacement 0, die zweite Speicherposition hat das Displacement 1 usw.).

Durch assoziative Auswahl kann man die gespeicherten Angaben über ihren Eigennamen aufrufen, gleichgültig, an welcher Stelle im Speicher sie sich befinden. Der Eigenname wird typischerweise durch Status- und Prädikatangaben ergänzt, die u. a. die Gültigkeit des betreffenden Wertes kennzeichnen (Abb. 2.7). Assoziative Speicher sind aufwendig, da zu jedem Wert auch der Eigenname sowie die Status- und Prädikatangaben gespeichert werden müssen (Assoziativteil). Um die Zugriffszeiten kurz zu halten, ist in jeder Speicherposition eine Vergleichseinrichtung erforderlich die den Inhalt des Assoziativteils mit den aktuellen Zugriffsangaben vergleicht. Deshalb kommen Assoziativspeicher nur für spezielle Zwecke in Frage (Caches, TLBs, Mikrobefehlspuffer in Superskalarmaschinen, kleinere Ressourcen-Konfigurationen).

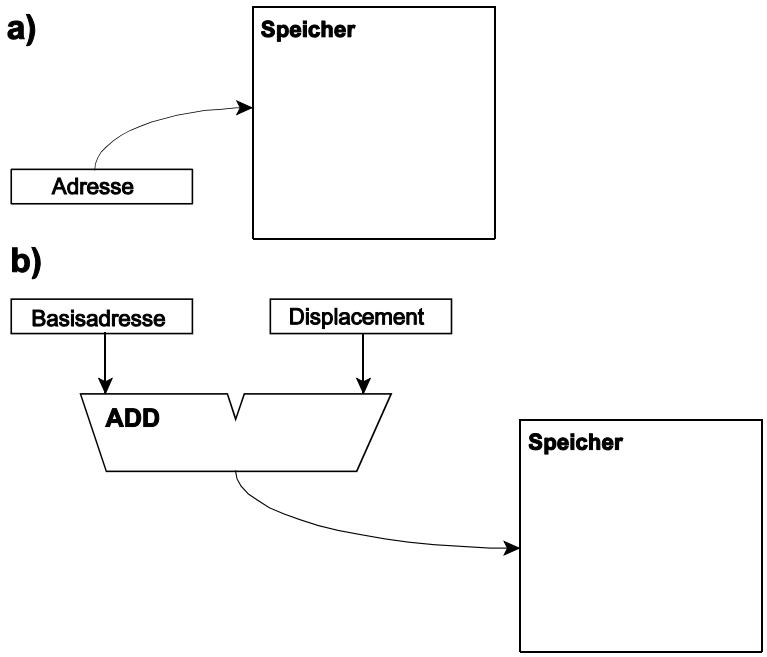


Abb. 2.6 Zugriff über Adressierung. a) mit absoluter Adresse; b) nach dem Prinzip Basis + Displacement

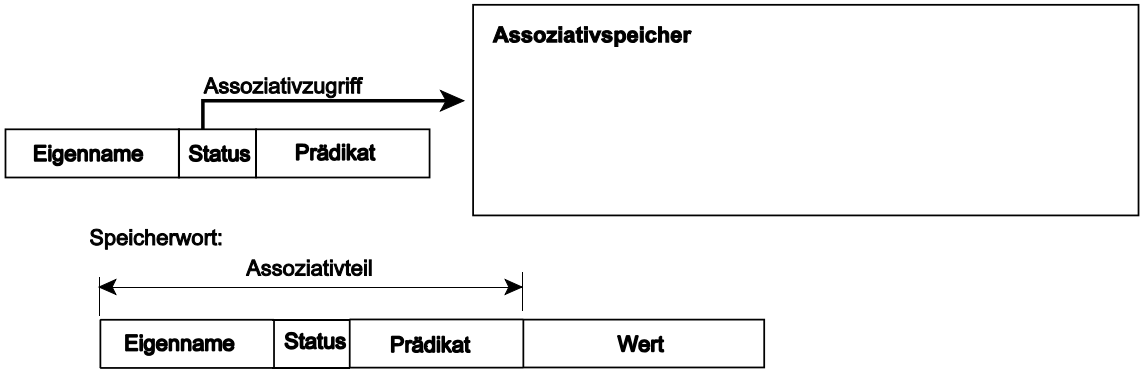


Abb. 2.7 Assoziativer Zugriff

Die Speicherung von Daten und Programme beruht vor allem auf adressierbaren Speichern. Solche Speicher haben typischerweise eine starre Zugriffsbreite von beispielsweise 8, 16, 32 oder 64 Bits. Dem gängigen Stand der Technik zufolge ist das Byte (8 Bits) die kleinste adressierbare Einheit (Byteadressierung). Die anwendungsseitigen Datenstrukturen sind aber vielgestaltig. Um zwischen der hardware- und der anwendungsseitigen Adressierung zu vermitteln, gibt es zwei Prinzipien, die sich darin unterscheiden, wann die Adreßumsetzung bzw. Adreßrechnung stattfindet:

1. Zur Compilierzeit. Die gängige und einfache Lösung. Zur Laufzeit beschänkt man sich typischerweise auf das Prinzip Basis + Displacement (vgl. Abb. 2.6b), um die Verschieblichkeit der Programme und Datenbereiche zu gewährleisten. Alle anderen Formen der Adreßrechnung werden ausprogrammiert (bei Nutzung höherer Programmiersprachen erledigt dies der Compiler).
2. Zur Laufzeit. Im Extremfall beschränkt sich der Compiler darauf, die mnemonischen Bezeichner in laufende Nummern (Ordinalzahlen) umzusetzen. Die eigentliche Adreßumsetzung findet statt, wenn das Programm ausgeführt wird. Diese Auslegung hat einige Vorteile: man kann Zugriffsrechte usw. für die einzelne Vaiable festlegen, die Variablen zur Laufzeit beliebig verschieben usw. Bei entsprechender Auslegung ist es auch möglich, ein einziges Programm für verschiedenartige Datentypen auszunutzen (z. B. einmal mit 32-Bit-Binärzahlen und ein andermal mit 64-Bit-Gleitkomazahlen zu arbeiten), ohne jedesmal neu compilieren zu müssen (objektorientierte Zugriffsweise). Der hardwareseitige Aufwand ist aber höher, und die Zugriffszeiten sind länger, da dem eigentlichen Speicherzugriff typischerweise noch Tabellenzugriffe und Rechenabläufe vorausgehen müssen (Abb. 2.8, 2.9).

Objektorientierung
Alle Programme und Datenbereiche werden als Objekte aufgefaßt. Objekte sind Behälter für Information, die jeweils als Ganzheit behandelt werden. Jedes Objekt hat eine Namen bzw. – zur Laufzeit – eine binär codierte Ordinalzahl, die das jeweilige Objekt aus der Menge aller Objekte auswählt. Die Programme beziehen sich nicht auf Adressen, sondern auf Objekte. Diese werden durch Objektdeskriptoren beschrieben, die in Objekttabellen zusammengefaßt sind:

- ist das Objekt im Arbeitsspeicher anwesend, enthält der Deskriptor einen Zeiger auf den betreffenden Speicherbereich (Anfangsadresse + Länge),
- ist das Objekt nicht anwesend, enthält der Deskriptor eine Positionsangabe für den Massenspeicher.

Die Objektverwaltung sorgt automatisch dafür, daß die jeweiligen Objekte in den Speicher geschafft oder ausgelagert werden.

Einstufiges Zugriffsschema
Die Variablen im Programm bezeichnen unmittelbar die jeweiligen Objekte (Abb. 2.8). Das entspricht einer Programmumgebung mit ausschließlich globalen Variablen.

Zweistufiges Zugriffsschema (Capability Based Addressing)
Die Variablen im Programm bezeichnen Eintrittspunkte in eine zur Laufzeit aktuelle Zugriffstabelle (Capability Table), die die Objekt-Identfier enthält, mit denen wiederum die Objekttabelle aufgesucht

wird (Abb. 2.9). Dieses Prinzip isoliert die Objektmenge von den Programmen und erlaubt eine feinfühlige Zuweisung von Schutzrechten (bis auf die einzelne Variable genau). Die Zugriffstabelle entspricht praktisch einem Stack Frame mit den Objekt-Identifiern der aktuellen lokalen Variablen.

Zum Zugriff auf die Speichermittel sind besondere Zugriffsressourcen vorgesehen. Sie können beispielsweise nach jedem der anhand der Abb.n 2.6 bis 2.9 erläuterten Prinzipien ausgelegt sein. Diese Prinzipen gehören zum allgemeinen Fachwissen, so daß eine genauere Beschreibung der verschiedenen Varianten nicht erforderlich ist. Die weiteren Beispiele betreffen deshalb vor allem die herkömmliche Auslegung (Adreßumsetzung zur Compilierzeit, Adreßrechnung nach dem Prinzip Basis + Displacement). Eine ReAl-Maschine kann aber auch höherentwickelte Zugriffsressourcen enthalten, die beispielsweise gemäß Abb. 2.9 ausgebildet sind. Es können mehrere derartige Ressourcen vorgesehen werden, und sie können als „echte“ Hardware (mit Rechenwerken, besonderen Tabellenspeichern usw.) ausgeführt werden. Zudem sind sie in vollem Maße programmseitig beeinflußbar. Hierdurch lassen sich die Leistungsschwächen der bisher bekannten Lösungen vermeiden.

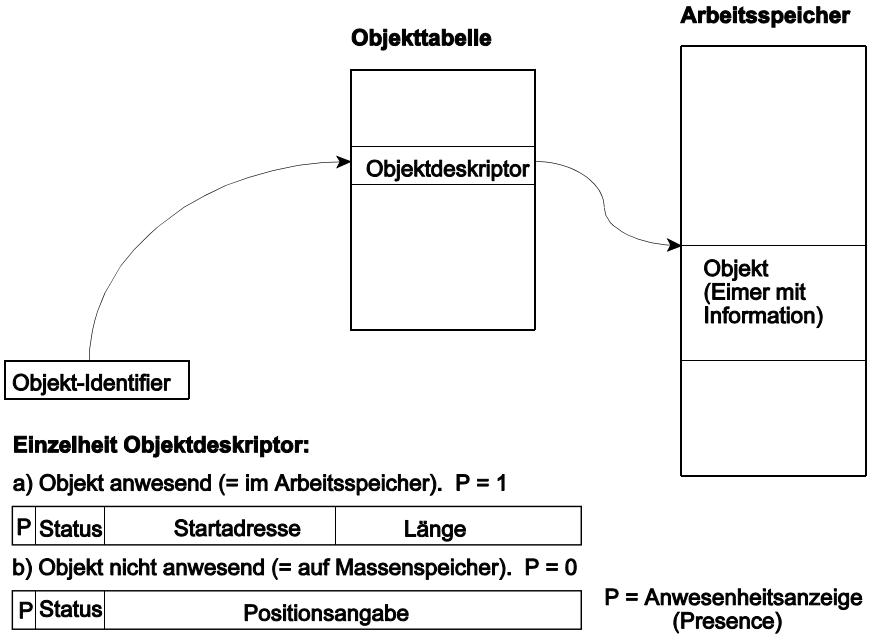


Abb. 2.8 Einstufiger objektorientierter Zugriff über Objekttabelle

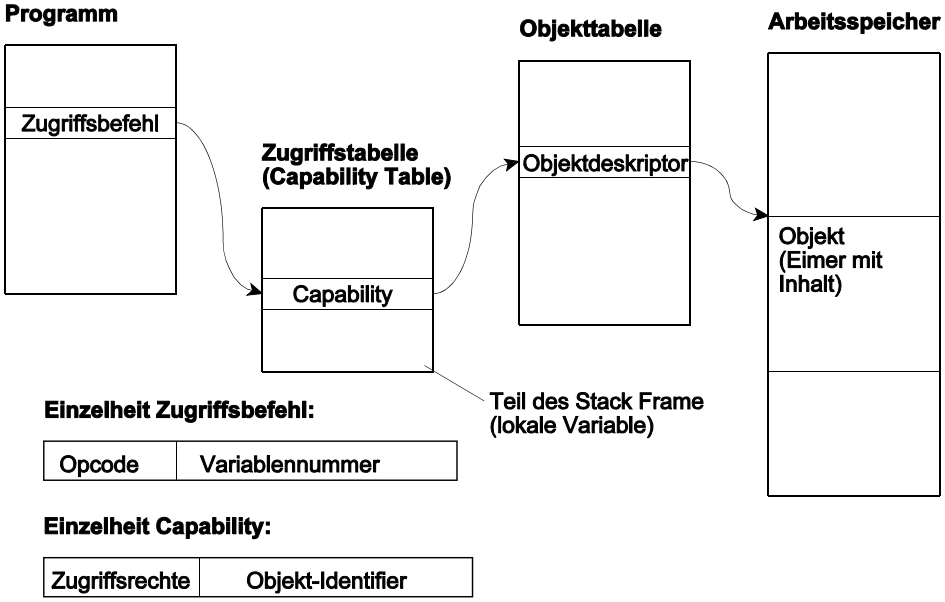
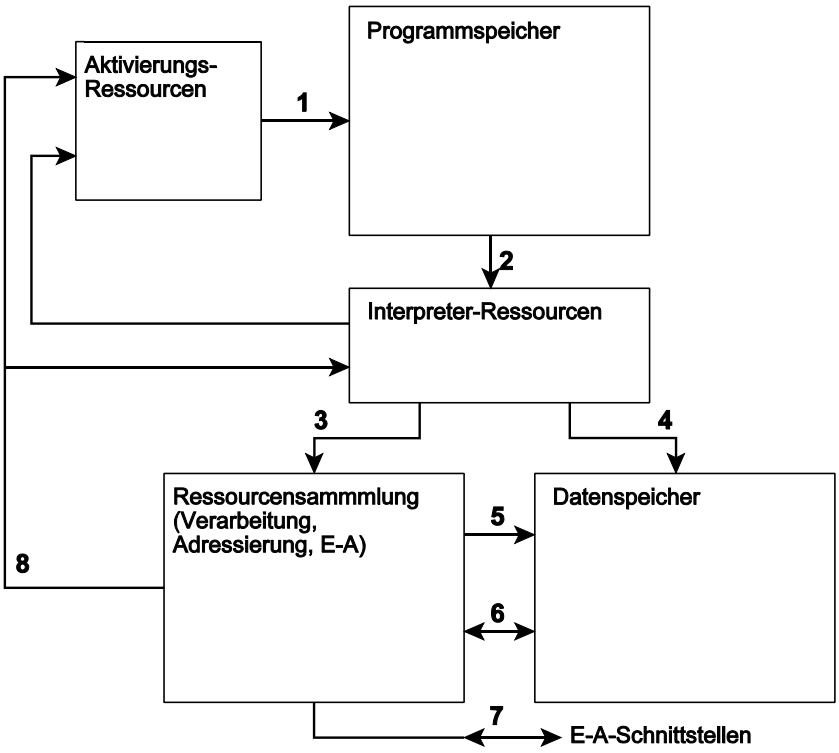


Abb. 2.9 Zweistufiger objektorientierter Zugriff (Capability Based Addressing)

Speichereinrichtungen in ReAl-Maschinen
Die Zugriffsressourcen können außerhalb der jeweiligen Speichereinrichtung angeordnet oder in diese eingebaut sein. Die Anzahl und Auslegung der Speichereinrichtungen ist im Grunde beliebig; es ist sowohl möglich, einen einzigen Speicher vorzusehen, der alle Angaben (Anweisungen und Daten) aufnimmt (v. Neumann-Prinzip), als auch mehrere jeweils nach Speicherkapazität, Zugriffsbreite, Zugriffsverfahren usw. zweckgerichtet ausgelegte Speichereinrichtungen anzuordnen (die typische Auslegung von Spezialrechnern). Abb. 2.10 veranschaulicht eine Maschine mit zwei Speichereinrichtungen, einer für die Anweisungen, einer für die Daten (Harvard-Organisation).



1 - Auswahl (Adressierung) der Operatoren bzw. Anweisungen; 2 - Angaben im ReAl-Maschinencode; 3 - Steuercodes; 4, 5 - Auswahl (Adressierung) der Parameter und Ergebnisse; 6 - Datentransport (Lesen und Schreiben); 7 - Ein- und Ausgabe; 8 - Bedingungssignale (beeinflussen die Auswahl der jeweils nachfolgenden Operatoren).

Abb. 2.10 ReAl-Maschine im Blockschaltbild. Hier in Harvard-Organisation

Daten- und Programmspeicher können zusammengefaßt werden (v.Neumann-Organisation). Im Beispiel von Abb. 2.10 können hierzu die Adreßwege 1, 4, und 5 zu einem Adreßbus und die Datenwege 2 und 6 zu einem Datenbus zusammengeschaltet werden.

2.6 Die Plattform

Damit eine ReAl-Maschine überhaupt arbeitsfähig ist, müssen folgende Ressourcen vorhanden sein:

- Speichermittel für Anweisungen und Daten,
- Aktivierungs-Ressourcen, die die jeweils auszuführenden Anweisungen im Speicher auswählen,
- Interpreter-Ressourcen, die die gelesenen Anweisungen interpretieren und ausführen,
- eine Sammlung zweckentsprechender Verarbeitungsressourcen.

Diese unbedingt erforderlichen Ressourcen bezeichnen wir im folgenden als Plattform. Es gibt zwei grundsätzliche Auslegungen:

1. Als herkömmlicher Universalrechner.
2. aAs spezielle ReAl-Hardware.

Die Hierarchie der Ressourcen
Es liegt nahe, kompliziertere Ressourcen aus einfacheren aufzubauen, sei es durch entsprechende Verschaltung (Verkettung), sei es durch programmseitig gesteuerte Ausnutzung in entsprechend vielen aufeinanderfolgenden Verarbeitungsschritten. In diesem Sinne werden die Ressourcen durch eine Ordnungszahl gekennzeichnet:

- Ordnungszahl 0: die Ressourcen der Plattform,
- Ordnungszahl 1: die Ressourcen, die von der Plattform aus unmittelbar angesprochen werden können,
- Ordnungszahl 2: Ressourcen, die aus Ressourcen der Ordnungszahlen 0 und 1 aufgebaut werden,
- Ordnungszahl 3: Ressourcen, die aus Ressourcen der Ordnungszahlen 0, 1 und 2 aufgebaut werden usw.

Die Entsprechungen in herkömmlichen Rechnerarchitekturen:

- Ordnungszahl 0: die allgemeinen Wirkprinzipien (Befehlslesen, Datenadressierung, Unterprogrammruf, Verzweigungen usw.),
- Ordnungszahl 1: die Verarbeitungsbefehle,
- Ordnungszahl 2: grundlegende Systemfunktionen, Unterprogrammbibliotheken usw.

Das Endziel der Programmentwicklung:
Alle Programmierabsichten müssen letzten Endes auf die Ausnutzung von Ressourcen der Ordnungszahlen 0 und 1 umgesetzt werden.

Umsetzung zur Laufzeit
Die Programmstruktur gemäß Programmierabsicht bleibt erhalten. Die für die aufeinanderfolgenden Programmschritte benötigten Ressourcen werden angefordert, genutzt und wieder freigegeben. Jeder Funktion bzw. Prozedur (in einer entsprechenden Programmiersprache formuliert) entspricht ein Unterprogramm, jedem Funktionsaufruf entspricht ein Aufruf des jeweiligen Unterprogramms. Wird ein Unterprogamm aufgerufen, so fordert es seinerseits Ressourcen an, versorgt sie mit Parametern, veranlaßt die Ausführung der Verarbeitungsfunktionen usw. Bei der Rückkehr zum rufenden Programm werden die angeforderten Ressourcen wieder freigegeben.

Bezogen auf die typische Auslegung der Laufzeitsysteme üblicher Programmiersprachen veranlaßt jeder Funktionsaufruf den Aufbau und den Abbau eines Stack Frame. Dabei sind die lokalen Variablen nur zeitweise vorhanden. Sie müssen bei jedem Aufruf neu bereitgestellt werden und gehen bei der Rückkehr verloren.

Vollständige Umsetzung zur Compilierzeit
Der Compiler setzt die Programmierabsicht vollständig in Ressourcen der Ordnungszahlen 0 und 1 um. Im Extremfall werden alle Ressourcen auf einmal angefordert, auch jene, die zur Ausführung von Funktionen (Unterprogrammen) benötigt werden. Sie bleiben während der gesamten Programmlaufzeit zugewiesen. Ein Funktionsaufruf beschränkt sich auf die Transporte der Operanden, auf das Aktivieren der Ressourcen und auf das Abholen der Ergebnisse. Ist es möglich, Ressourcen miteinander zu verketten, genügt es oftmals, nur die Parameter einzutragen. Alles andere läuft dann gleichsam von selbst ab, d. h. ohne weiteren Eingriff der Plattform (die Abläufe werden von den Verarbeitungsressourcen autonom gesteuert).

Dabei entfällt das Auf- und Abbauen von Stack Frames. (Ins Herkömmliche übersetzt handelt es sich um ein Verarbeitungsmodell, in dem die Stack Frames aller Funktionsaufrufe zu Anfang gleichsam vorgefertigt werden.) Alle lokalen Variablen (in den Ressourcen) werden somit praktisch zu globalen Variablen, die einfach erreichbar und ständig zugänglich sind.

Der Universalrechner als ReAl-Plattform
Es gibt zwei grundsätzliche Möglichkeiten, eine ReAl-Architektur zu unterstützen:

1. Emulation. Die Operatoren und Anweisungen der ReAl-Architektur werden von entsprechender Software aus dem Speicher geholt, die jeweiligen Wirkungen werden programmseitig nachgebildet.

2. Compilierung. Ein als ReAl-Code vorliegendes Programm (Quellprogramm) wird in Folgen herkömmlicher Maschinenbefehle umgesetzt und ausgeführt.

Typische Schritte der herkömmlichen Programmentwicklung:
Quelltext in höherer Programmiersprache => Maschinenprogamm einer (fiktiven) Stackmaschine (auf Zielmaschine durch Emulation lauffähig) => optimiertes Maschinenprogramm für Zielmaschine.

Problem: Stackmaschinen eignen sich gut zum Auflösen beliebig geschachtelter Ausdrücke, sind aber ihrer Wirkungsweise nach inhärent sequentiell.

Typische Schritte der Programmentwicklung unter Nutzung der ReAl-Prinzipien:
Quelltext in höherer Programmiersprache => Maschinenprogamm einer (fiktiven) ReAl-Maschine (auf Zielmaschine durch Emulation lauffähig) => optimiertes Maschinenprogramm für Zielmaschine.

Vorteil: die Ausnutzung der ReAl-Prinzipien erlaubt es, den innewohnenden Parallelismus des Programms nahezu vollständig zur Compilierzeit zu erkennen.

Prinzip: für jeden Programmschritt werden zunächst jeweils gesonderte Ressourcen angefordert. Diese werden soweit möglich miteinander verkettet. Somit entsteht zunächst eine fiktive Ressourcenkonfiguation in womöglich gigantischer Größenordnung (hunderttausende Rechenwerke usw.). In anschließenden Durchmusterungsläufen wird diese Konfiguration dann nach und nach auf einen praktikablen Umfang reduziert.

Ein- und Ausgabe
Sie erfolgt entweder über bestimmte Adreßbereiche (Memory Mapped I/O) oder über spezielle Ressourcen (E-A-Ports, Interfaceadapter, Impulsmustergeneratoren, Zeitgeber usw.; vgl. die Ausstattung der üblichen Mikocontroller).

Eine Eingabe-Ressource hat – im Sinne des Systems – keine Eingänge (Eingänge kommen von außen; sie sind nicht ladbar oder verkettbar).

Eine Ausgabe-Ressource hat – im Sinne des Systems – keine Ausgänge (Ausgänge gehen nach außen; sie sind nicht abholbar oder verkettbar).

Maschinencode
Es gibt mehrere Alternativen:

- ReAl-Operatoren als Bytecode (variable Länge),
- fest formatierte Maschinenbefehle, die den ReAl-Operatoren entsprechen (1 Befehl = 1 Operator),
- Anweisungen, die ganz elementare Abläufe auslösen (z. B. Informationstransporte). Die Wirkungen der ReAl-Operatoren werden mit Folgen entsprechender Anweisungen nachgebildet.
- Befehle, die den Maschinenbefehlen der herkömmlichen Architekturen ähnlich sind,
- Umsetzung der ReAl-Operatoren in Folgen herkömmlicher Maschinenbefehle (Compilierung).

In den Kapiteln 7 und 8 werden verschiedene Beispiele beschrieben.

Charakteristische Vorteile:

- Es kann eine letzten Endes beliebige Anzahl von Ressourcen angesprochen werden (keine Begrenzung der Ressourcenanzahl, wie sie beispielsweise in den sog. VLIW-Architekturen gegeben ist),
- die Ressourcenbereitstellung und -freigabe ist bis in die Einzelheiten programmseitig steuerbar; es ist möglich, aus den gegebenen Ressourcen für jede aktuelle Verarbeitungsaufgabe ad hoc eine Art virtuelle Spezialmaschine zu konfigurieren und am Ende wieder abzubauen,
- sind solche Strukturen erst einmal aufgebaut, so ist der programmseitige Verwaltungsaufwand (Overhead) deutlich geringer als in herkömmlichen Maschinen (kein Auf- und Abbau von Stack Frames, kein Speichern und Wieder-Lesen von Zwischenwerten),

- im Gegensatz zu herkömmlichen Operationsbefehlen ist die Operationsauswahl (s-Operator) von der Operationsauslösung (y-Operator) getrennt. Jede Ressource weiß somit von vornherein, wofür die übertragenen Parameter bestimmt sind. Das kann gelegentlich zur Optimierung der Hardware ausgenutzt werden. Die Auslösungsangaben sind typischerweise kürzer als die Auswahlangaben. Das ist dann von Vorteil (Codeverkürzung), wenn gleiche Funktionen immer wieder oder wenn viele Funktionen auf einmal auszulösen sind. Bei gegebener Befehlslänge kann man mehr Funktionen auf einmal auslösen, als dies üblicherweise möglich ist. Beispiel: eine der bekannten Architekturen für Hochleistungsprozessoren hat Befehlsworte von 128 Bits Länge, die drei Befehle aufnehmen. Somit können bis zu 3 Operationen gleichzeitig ausgelöst werden. Nimmt man einen Operationscode von 8 Bits an, so könnten mit 128 Bits beispielsweise ausgelöst werden:
  - wenn jeder Ressource ein Bit zugeordnet ist: bis zu 120 Verarbeitungsvorgänge,
  - wenn die Resourcenadresse 6 Bits lang ist: bis zu 20 Verarbeitungsvorgänge,
  - wenn die Ressourcenadresse 12 Bits lang ist: bis zu 10 Verarbeitungsvorgänge.

Aufruf der Befehle bzw. Anweisungen
Zum Holen und Ausführen der einzelnen ReAl-Befehle bzw. Anweisungen sind besondere Ressourcen vorgesehen (Aktivierungs-Ressourcen). Die einfachste Ressource dieser Art ist der herkömmliche Befehlszähler (fortlaufende Areßzählung). Man könnte sich aber auch ungewöhnliche bzw. neuartige Ressourcen zum Aufrufen der ReAl-Anweisungen ausdenken, beispielsweise:

- Über Prädikate, die den Anweisungen zugeordnet sind (assoziatives Prinzip),
- über den Inhalt der Anweisungen. Es werden beispielsweise jene Anweisungen aufgerufen, die bestimmte Operanden oder Ressourcen betreffen (assoziatives Prinzip).

2.7 Verarbeitungsressourcen

Lassen wir die Art der ausführbaren Informationswandlungen (Operationen) zunächst außer Betracht, so unterscheiden sich die Ressourcen (Abb. 2.11) vor allem nach

- der Anzahl der ausführbaren Operationen (eine oder mehrere),
- der Verarbeitungsbreite,
- der Art der Parameterübergabe (beispielsweise als Wert oder über Adresse (by Value / by Reference)),
- der Verkettung (einbezogene Parameter, Auslegung der Verkettungssteuerung),
- den zugeordneten Verarbeitungszuständen (zustandsfreie Auslegung oder Einbeziehung von Parametern in den Verarbeitungszustand bzw. Programmkontext),
- den Zusatzfunktionen (Überwachung von Wertebereichen, Registrieren der Nutzungshäufigkeit usw. (Instrumentierung)).

Einfache Ressourcen
Eine solche Ressource (Abb. 2.12) führt nur jeweils eine bestimmte Art von Informationswandlungen aus (Operation fest, Verarbeitungsbreite fest). Die Operanden werden als Werte übergeben, die Ergebnisse als Werte hinterlegt. Verkettung ist nicht vorgesehen. Anwendbare Operatoren: p, y, a, l.

Parameterübergabe über Adresse (by Reference)
Hierzu sind die Ressourcen mit entsprechenden Adreßregisterrn und mit Zugangswegen zum Speicher auszustatten (Abb. 2.13). Die Alternative: einer Verarbeitungsessource, die für die Wertübergabe eingerichtet ist, werden weitere Ressourcen vor- bzw. nachgeschaltet, die die Speicherzugriffe ausführen (Abb. 2.14).

Schaltmittel und Ressourcen für Adressierung und Datentransport
Entsprechende Schaltmittel sind in die Verarbeitungsressourcen eingebaut oder als gesonderte (vor- und nachzuschaltende) Ressourcen ausgebildet. Sie bestehen aus den eigentlichen Adressierungsvorkehrungen und den Mitteln der Zugriffssteuerung (deren Auslegung sich nach der jeweiligen Speicherschnittstelle (Bus, Verteiler o. dergl.) richtet).

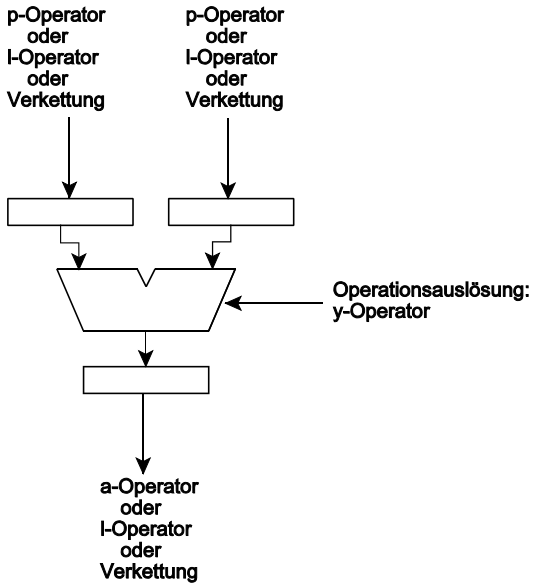


Abb. 2.11 Parametertransporte und Operationsaulösung

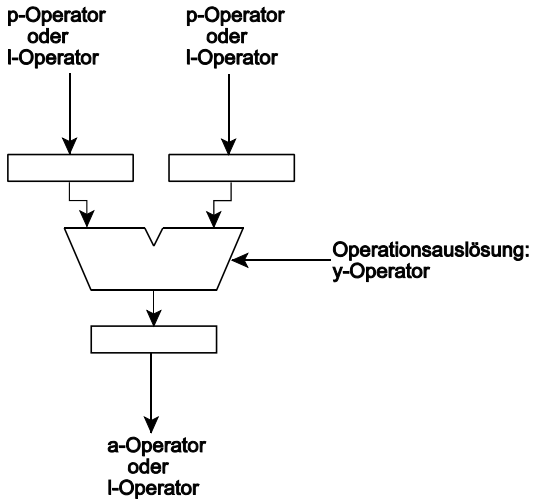


Abb. 2.12 Eine einfache Ressource (ohne Verkettungsvorkehrungen) im Blockschaltbild

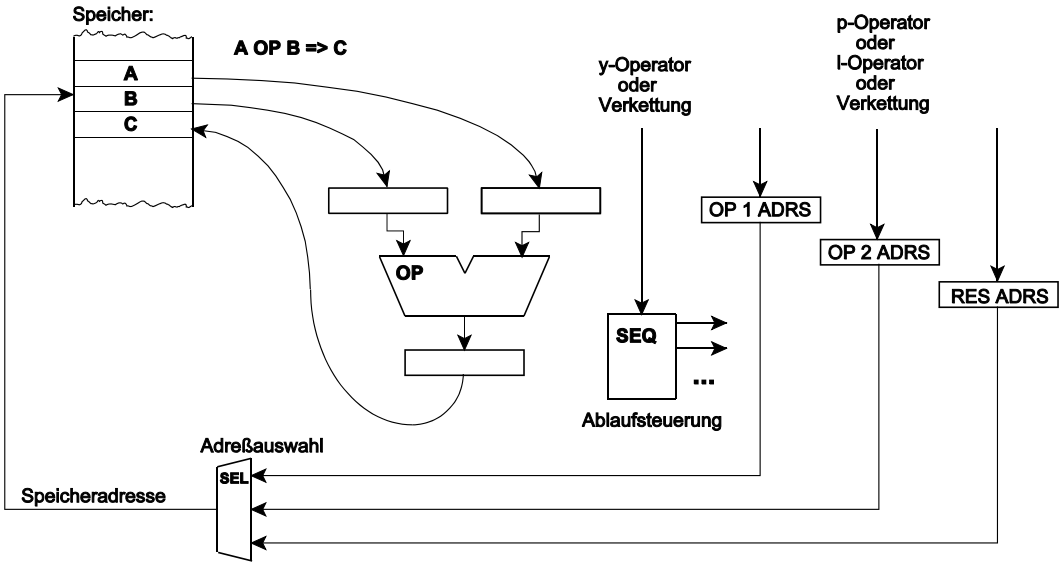
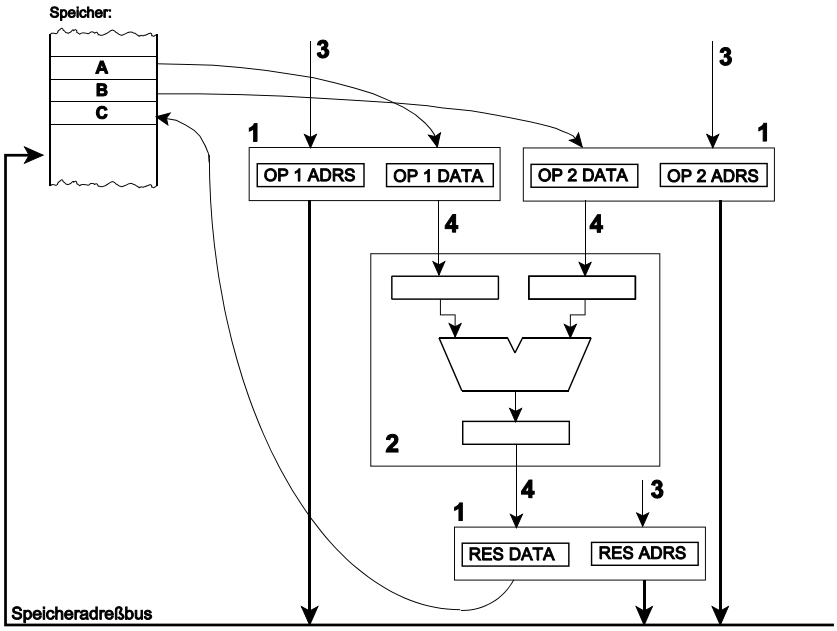


Abb. 2.13 Einfache Ressource, die die Parameterübergabe über Adresse (by Reference) unterstützt

Einfache Adressierungsvorkehrungen:

- Adreßregister,
- als Zähler ausgebildete Adreßregister (Increment, Decrement),
- Adreßaddition (z. B. Basis + Displacement; Abb. 2.15),
- Iteratoren, z. B. für die Laufvariable in typischen FOR-Schleifen (Abb. 2.16, 2.17).



1 - Ressourcen für Adressierung und Datentransport; 2 - Verarbeitungsressource; 3 - Eintragen der Adreßangaben (p-Operator oder l-Operator oder Verkettung); 4 - Wertübergabe (l-Operator oder Verkettung).

Abb. 2.14 Parameteradressierung über vor- und nachgeschaltete Ressourcen

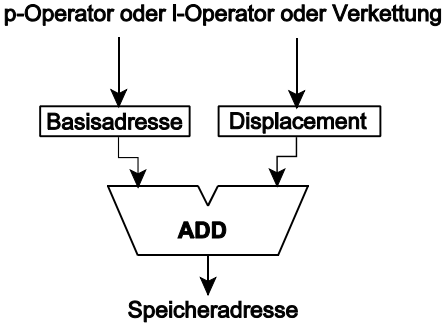


Abb. 2.15 Schaltung zur Adreßaddition (Basis + Displacement)

Ressourcen mit Operationsauswahl
Solche Ressourcen können jeweils eine von mehreren Informationswandlungen ausführen. Auswahl: über zusätzliche Parameter (Abb. 2.18, 2.19). Diese Auslegung erlaubt auch eine vom Verarbeitungsablauf abhängige Funktionswahl (die jeweils auszufühende Operation wird von vorgeordneten Ressourcen gleichsam berechnet, aus Tabellen entnommen o. dergl.). In diesem Sinne ist es zweckmäßig, jeweils einen unwirksamen Funktionscode (No Operation, NOP) vorzusehen. Ist ein solcher Funktionscode eingestellt, so verändert die Ressource keinen Parameter. Der Verarbeitungszustand (Programmkontext) bleibt erhalten; ausgangsseitige Verkettungen werden nicht ausgelöst. Hierdurch ist es möglich, die Ressource in Abhängigkeit vom Verarbeitungszustand zu umgehen (bedingte Ausführung).

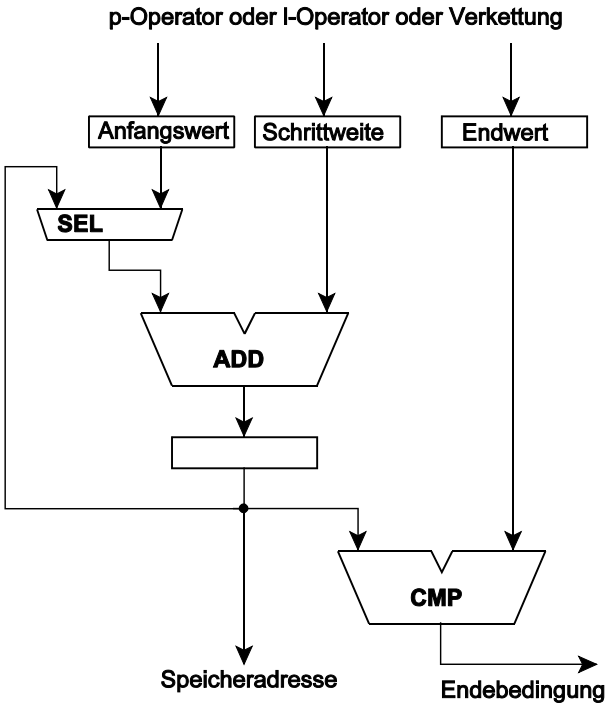


Abb. 2.16 Einfache Iteratorschaltung. Wirkungsweise: FOR Anfangswert TO Endwert STEP Schrittweite

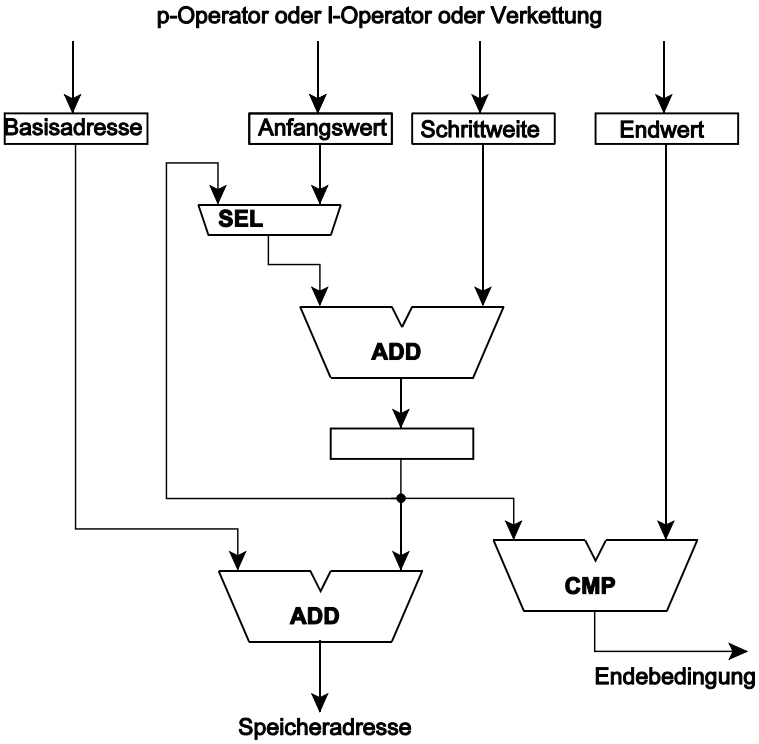


Abb. 2.17 Iterator mit nachgeordneter Adreßaddition (Kombination der Schaltmittel gem. Abb. 2.15 und 2.16)

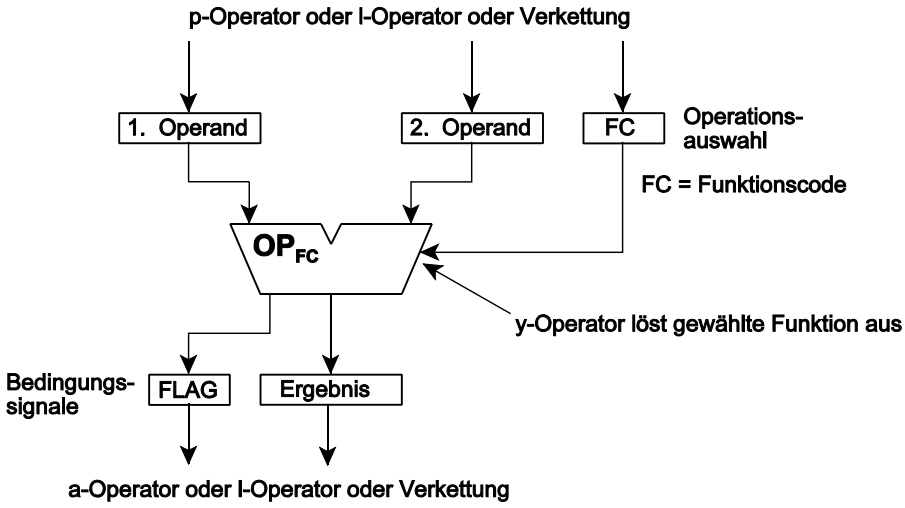


Abb. 2.18 Ressource mit Operationsauswahl

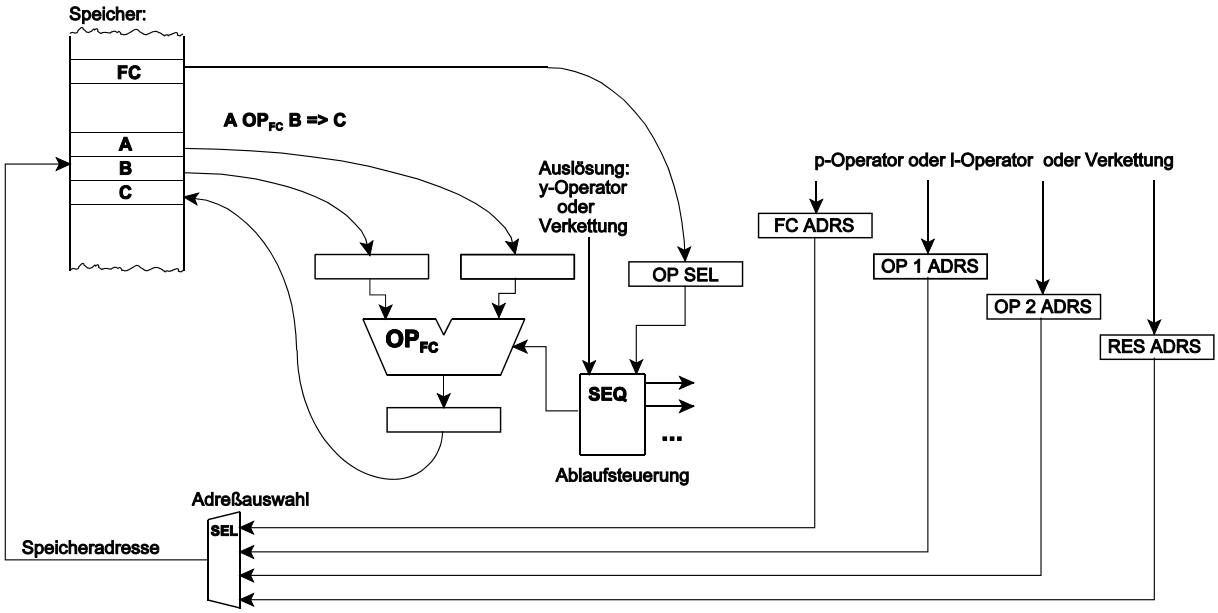


Abb. 2.19 Ressource mit Operationsauswahl über Adressierung. In diesem Beispiel werden alle Parameter durch Adressierung (by Reference) übergeben

Ressourcen mit veränderlicher Verarbeitungsbreite

Über weitere Parameter ist die Verarbeitungsbreite einstellbar (Abb. 2.20). Operanden werden nur gemäß der eingestellten Breite verarbeitet (überschüssige Bitpositionen werden ignoriert), Ergebnisse nur gemäß der eingestellten Breite abgeliefert. Die Behandlung überschüssiger Bitpositionen hängt von der Auslegung der Ressourcen und (was u. a. das Speichern betrifft) von der Plattform ab. Auslegungsbeispiele:

- Auffüllen mit Nullen (Nullerweiterung),
- Auffüllen mit der Belegung der jeweils (gemäß eingestellter Breite) höchstwertigen Bitposition (Vorzeichenerweiterung),
- überschüssige Bits ignorieren, kürzere Operanden rechtsbündig einfügen (wobei die restliche Belegung erhalten bleibt).

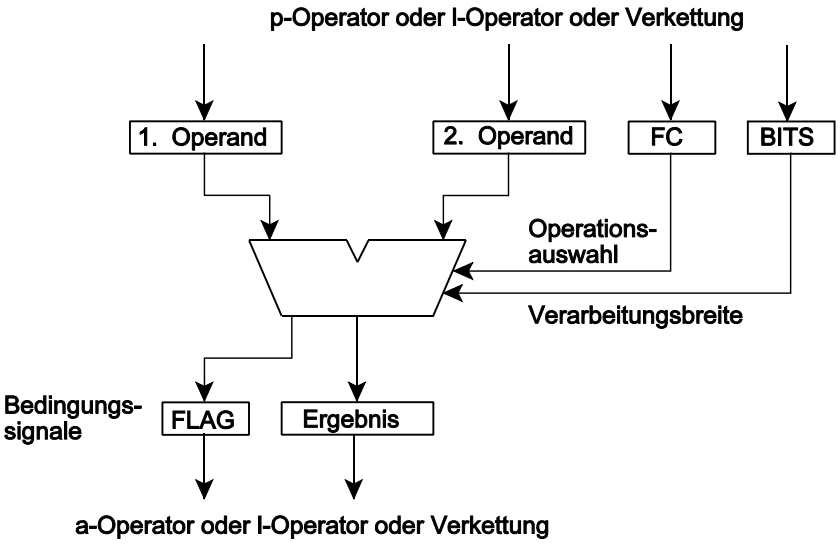
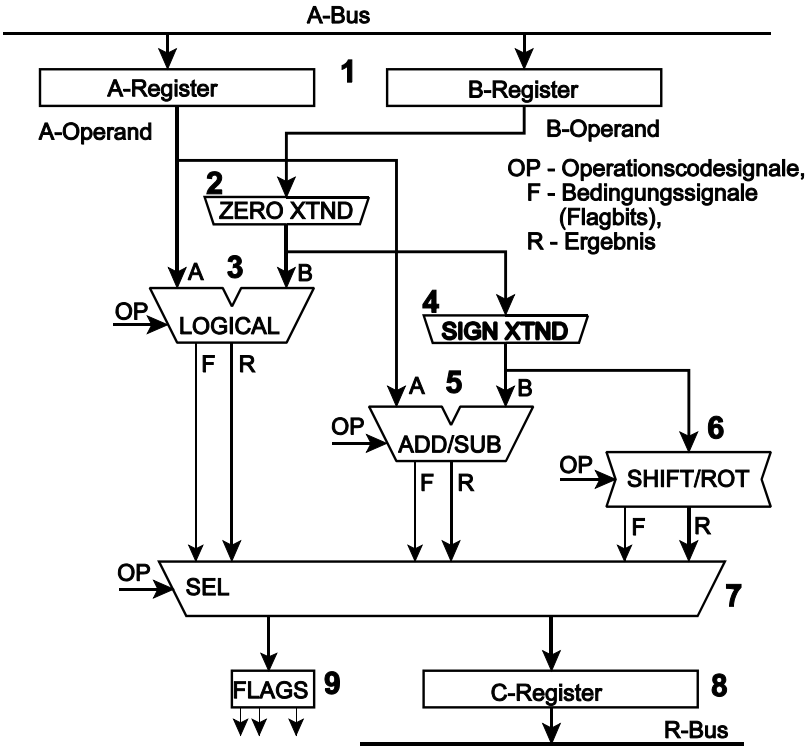


Abb. 2.20 Ressource mit veränderlicher Verarbeitungsbreite

Arithmetik-Logik-Einheiten als universelle Ressourcen
Elementare, universelle Resourcen können u. a. vorteilhaft als Arithmetik-Logik-Einheiten ausgebildet werden (Abb. 2.21), und zwar sowohl mit fester als auch mit veränderlicher Verarbeitungsbreite.



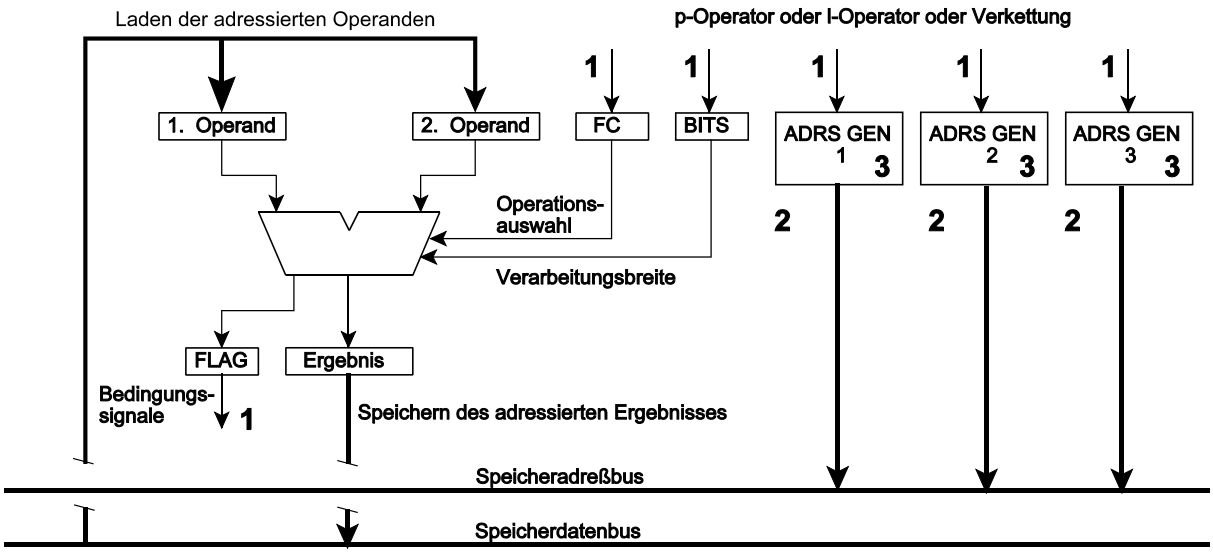
1 - Operandenregister; 2 - Nullerweiterung; 3 - logische Verknüpfungen und Transporte; 4 - Vorzeichenerweiterung; 5 - Addition und Subtraktion; 6 - Verschieben und Rotieren; 7 - Ergebnisauswahl; 8 - Ergebnisregister; 9 - Flagregister.

Abb. 2.21 Arithmetik-Logik-Einheit (ALU)

Beispiel eines Operationsvorrates:

- Addieren,
- Addieren mit Ausgangsübertrag,
- Subtrahieren,
- Subtrahieren mit Ausgangsübertrag,
- bitweise Konjunktion (AND),
- bitweise Disjunktion (OR),
- bitweise Antivalenz (XOR),
- bitweise Negation (NOT),
- Zweierkomplement,
- nach links verschieben/rotieren,
- nach rechts verschieben/rotieren,
- keine Wirkung (NOP).

Die Ressourcen können für verschiedene Formen der Parameterübergabe ausgelegt werden (Wertübergabe, Adreßübergabe usw. sowie beliebige Kombinationen; Abb. 2.22, 2.23).

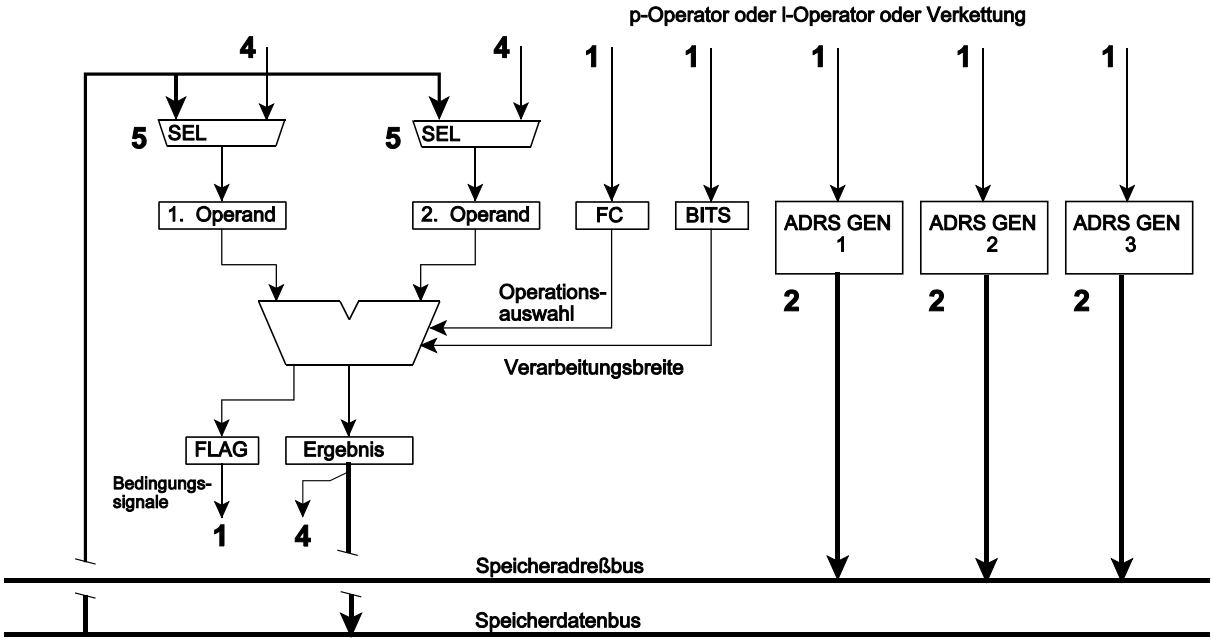


1 - Wertübergabe; 2 - Adreßübergabe; 3 - Adressierungsschaltungen (Adreßgeneratoren), die auf verschiedene Weise ausgelegt sein können (Adreßregister, Adreßaddierer, Iteratoren usw.). Die jeweilige Art der Parameterübergabe ist fest vorgesehen (= unveränderlich). Der Speicherzugangsweg ist in diesem Beispiel als Bus ausgeführt.

Abb. 2.22 Verarbeitungsressource mit verschiedenen Formen der Parameterübergabe

Varianten der Umschaltung:

1. Durch Operationsauswahl (FC-Parameter).
2. Durch Zustandssteuerung im Innern der Ressource. Die Form der letzten Übergabe vor der Operationsauslösung (y-Operator oder Verkettung) wird als gültig angesehen. 1. Beispiel: das Eintragen eines Parameters in den ersten der Adreßgeneratoren 3 führt dazu, daß für den 1. Operanden die Adreßübergabe eingestellt wird. 2. Beispiel: ein p-Operator, der den zweiten Operanden direkt einträgt, veranlaßt, daß der zweite der Adreßgeneratoren 3 außer Funktion gesetzt wird (Einstellung auf Wertübergabe).



1 - Wertübergabe; 2 - Adreßübergabe; 3 - Adressierungsschaltungen (Adreßgeneratoren); 4 - wahlweise Wertübergabe; 5 - Auswahlschaltungen. Die Parameterübergabe kann zwischen Wert- und Adreßübergabe umgeschaltet werden.

Abb. 2.23 Verarbeitungsressource mit wählbaren Formen der Parameterübergabe

Anzeige der Zugriffsbreite
Die Ressourcen und die Plattform müssen wissen, aus wievielen gültigen Bits die einzelnen Parameter bestehen.

Prinzipien:

1. Die aktuelle Zugriffsbreite wird in entsprechenden Tabellen vermerkt (vgl. Kapitel 6). Diese universelle und wenig aufwendige Lösung hat den Nachteil, daß zur Laufzeit den Operandenzugriffen jeweils entsprechende Tabellenzugriffe vorausgehen müssen, um die jeweils aktuelle Zugriffsbreite abzufragen.
2. Der Maschinencode enthält Angaben zur Zugriffsbreite. Diese Lösung ähnelt dem herkömmlichen Stand der Technik (es gibt Befehle z. B. zum Transportieren von Bytes, Worten, usw.). Soll der Aufwand nicht übermäßig ansteigen, können nur wenige Zugriffsbreiten unterstützt werden. Zudem muß die jeweilige Verarbeitungsbreite zur Compilierzeit festliegen; ein dynamisches Umschalten (zur Laufzeit) ist nicht möglich.
3. Die Speichermittel in den Ressourcen, die die Parameter aufnehmen, werden um zusätzliche Kennzeichnungsbits (TAG-Bits) erweitert, die die jeweils aktuelle Breite angeben (Abb. 2.24). Die TAG-Bits werden jeweils zusammen mit den Datenbits übertragen bzw. sind von den anderen Ressourcen abfragbar. Hierzu werden die Schnittstellen zwischen den Ressourcen (Bussysteme, Punkt-zu-Punkt-Interfaces o. dergl.) entsprechend ergänzt.

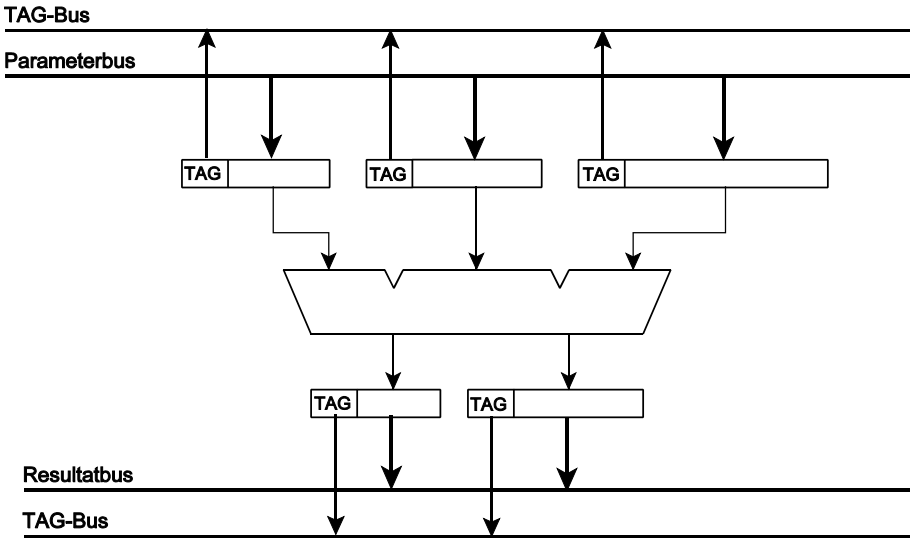


Abb. 2.24 Verarbeitungsressource mit Kennzeichnungsbits (TAG-Bits), die die jeweilige Zugriffsbreite angeben. Vor- und nachgeschaltete Ressourcen können diese Angaben über zusätzliche Busleitungen abfragen

2.8 Verkettung

Ausgänge können mit Eingängen nachgeordneter Ressourcen verkettet werden.

Herstellen einer Verkettung: mittels c-Operator.

Auflösen einer Verkettung: mittels d-Operator.

Varianten der Ausführung:

1. Die Verbindungen werden physisch generiert (z. B. bei Aufbau mit programmierbarer Hardware).
2. Die Verbindungen werden physisch geschaltet (z. B. über Bussysteme oder Schaltverteiler (Switch Fabric)).
3. Die Verbindungen werden mit Verkettungszeigern oder in Verkettungslisten beschrieben. Auf Grundlage dieser Angaben führen die Ressourcen die jeweiligen Datenübertragungen selbsttätig aus (z. B. über ein Bussystem, an dem sie als Busmaster aktiv werden, oder über einen Schaltverteiler).

Es hängt von der jeweiligen Ressource ab, welche Parameter in die Verkettung einbezogen werden können und welche nicht. Zu verkettende Ergebnisse müssen um Angaben erweitert werden, die die Ziele der Verkettung beschreiben. Handelt es sich um Hardware-Ressourcen, sind zusätzliche Schaltmittel erforderlich, um die entsprechenden Informationstransporte zu steuern (Abb. 2.25).

Technische Lösungen zur Angabe von Verkettungszielen:

- Zeiger,
- Zeigerliste,
- eine Art horizontaler Mikrobefehl, der die Signalwege steuert,
- Versenden von Datenpaketen mit entsprechenden Bestimmungsangaben.

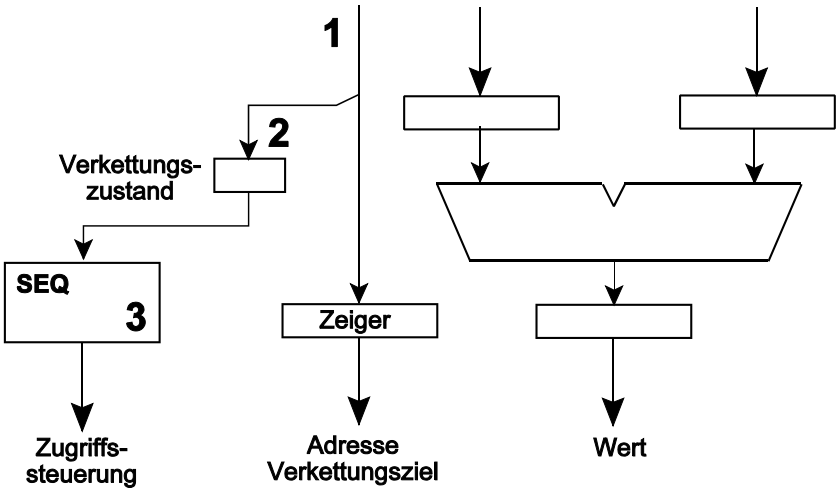


Abb. 2.25 Diese Ressource unterstützt die Verkettung des Ergebnisses. 1 - über diesen Weg wirken die c- und d-Operatoren. 2 - Zustandsbit; 3 - Zugriffssteuerung

Belegung des Zustandsbits:

- 0 = keine Verkettung,
- 1 = Verkettung aktiv.

Wirkung des c-Operators:
Eintragen der Verkettungsadresse in das Zeigerregister und Setzen des Zustandsbits 2.

Wirkung des d-Operators:
Löschen des Zustandsbits 2.

Wirkung des y-Operators:

1. Bilden des Ergebnisses.
2. Ist das Zustandsbit 2 gesetzt, so wird die Zugriffssteuerung 3 aktiv, um das Ergebnis gemäß Adreßangabe im Zeigerregister zur adressierten Ressource zu transportieren (z. B. durch Anfordern eines entsprechenden Buszugriffs).

Aktivierung der verketteten Ressourcen
Jede Ressource, die Ziel einer Verkettung ist, muß wissen, wann sie zu rechnen anfangen soll. Es gibt mehrere Möglichkeiten, solche Ressourcen zu aktivieren:

a) Auslösung über y-Operator,
b) Neuberechnung bei Änderung der Eingangsbelegungen,
c) Gültigkeitssteuerung.

Auslösung über y-Operator
In den betreffenden Ressourcen sind keine besonderen Vorkehrungen erforderlich. Die Plattform muß aber eine strikt sequentielle Ausführungsreihenfolge der Operatoren gewährleisten. Der jeweils nächste Operator darf erst wirksam werden, nachdem der vorhergehende Operator vollständig ausgeführt wurde. Das schließt auch die Datentransporte der Verkettung ein. Die Ausführung eines y-Operators für eine Ressource ähnlich Abb. 2.25 dauert so lange, bis das verkettete Ergebnis abtransportiert wurde. Erst dann wird der nächste Operator wirksam. Die Reihenfolge der y-Operatoren der verketteten Ressourcen muß der Verkettungsreihenfolge entsprechen. Vorteilhaft ist die Einfachheit. Bei der softwareseitigen Emulation und in kleineren ReAl-Maschinen (z. B. mit einem einzigen Universalbus) ergibt sich die strikte Ausführungsreihenfolge gleichsam von selbst (ein Befehl oder Datentransport zu einer Zeit). Leistungsfähigere Hardware kann allerdings nicht in vollem Maße ausgenutzt werden. Auch spart man – gegenüber Programmabläufen ohne Verkettung – nur die l-Operatoren ein, nicht aber die y-Operatoren.

Neuberechnung bei Änderung der Eingangsbelegungen
Den Operandenregisten sind Vergleichsschaltungen nachgeordnet, die den bisherigen Inhalt mit dem neuen (an den Eingängen anliegenden) Inhalt vergleichen (Abb. 2.26). Ungleichheit zeigt eine Änderung an, die eine Neuberechnung der Ergebnisse zur Folge hat. Um diese Auslösung selektiv zu steuern, kann ein zusätzlicher Parameter vorgesehen werden, der als Änderungsmaske wirkt.

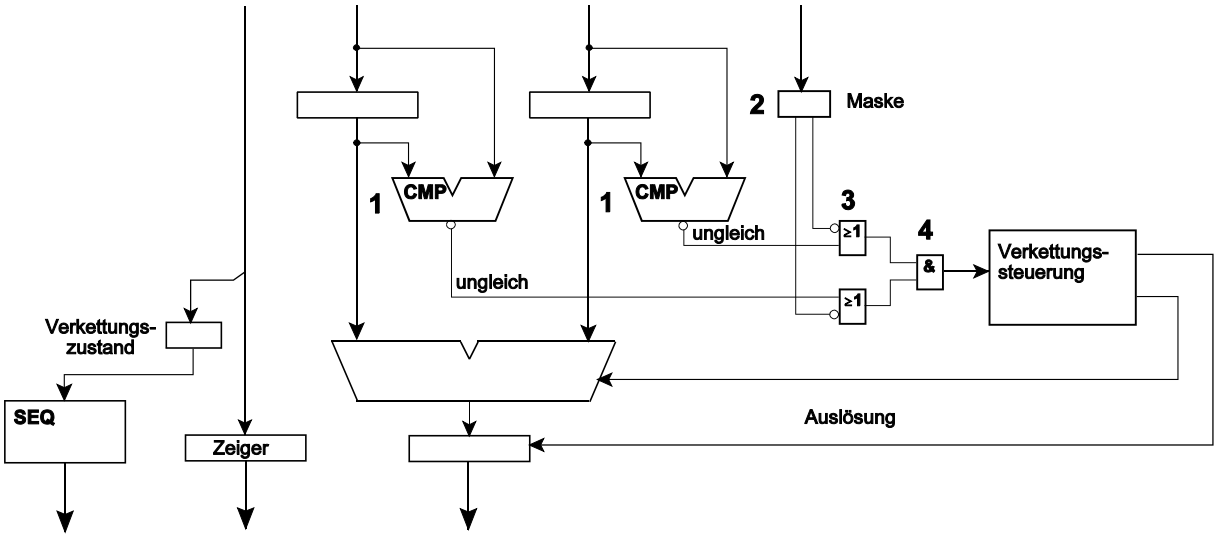


Abb. 2.26 Neuberechnung bei Änderung der Eingangsbelegungen. 1 - Vergleichsschaltungen; 2 - Maskenregister (enthält die Änderungsmaske); 3 - konjunktive Verknüpfung; 4 - disjunktive Verknüpfung

Der Operandenvergleich findet dann statt, wenn neue Operanden eingetragen werden. Unterscheidet sich der neue (ankommende) Operand vom bisherigen Inhalt des jeweiligen Operandenregisters, so wird die betreffende Vergleichsschaltung 1 aktiv. In einer einfachen Ausführung sind die Ausgänge aller Vergleichsschaltungen 1 konjunktiv verknüpft, so daß eine Neuberechnung dann ausgelöst wird, wenn alle Operanderegister mit neuen Werten versorgt worden sind (Wirkung wie y-Operator). Gelegentlich ist es aber zweckmäßig, eine Neuberechnung nur dann zu veranlassen, wenn sich bestimmte Operanden ändern (einzeln oder in Kombination). Im Beispiel sind hierzu der konjunktiven Verknüpfung 4 disjunktive Verknüpfungen 3 mit invertierten Ausgängen des Maskenregisters 2 nachgeschaltet. Jedem Operanden entspricht ein Bit im Maskenregister. Ist das Bit = 0, so wird der betreffende Operand bei der Auswertung der Änderungen nicht berücksichtigt.

Gültigkeitssteuerung
Den einzelnen Parametern (= eingangsseitigen Registern/Speicherpositionen) werden Zustände zugeordnet:

- nicht verkettet (not connected),
- frei (invalid),
- gültig (valid),
- Gültigkeit halten (stick on valid), aber frei,
- Gültigkeit halten (stick on valid) und gültig.

Diese Zustände können beispielsweise in zwei Zustandsbits codiert werden, die dem jeweiligen Operandregister zugeordnet sind (Abb. 2.27, 2.28).

Zustandsbit	STA2	STA1	STA0
Wert = 0	nicht verkettet (not connected)	Gültigkeit nicht halten	frei (invalid)
Wert = 1	verkettet (connected)	Gültigkeit halten (stick on valid)	gültig (valid)
Bit löschen	d-Operator	fest eingestellt oder spezieller p- bzw. u-Operator	c-Operator, Verkettungssteuerung, sofern stick on valid inaktiv
Bit setzen	c-Operator		ankommender Operand (Verkettung oder p-Operator oder l-Operator)

Abb. 2.27 Zustandscodierung in einem Zustandsregister

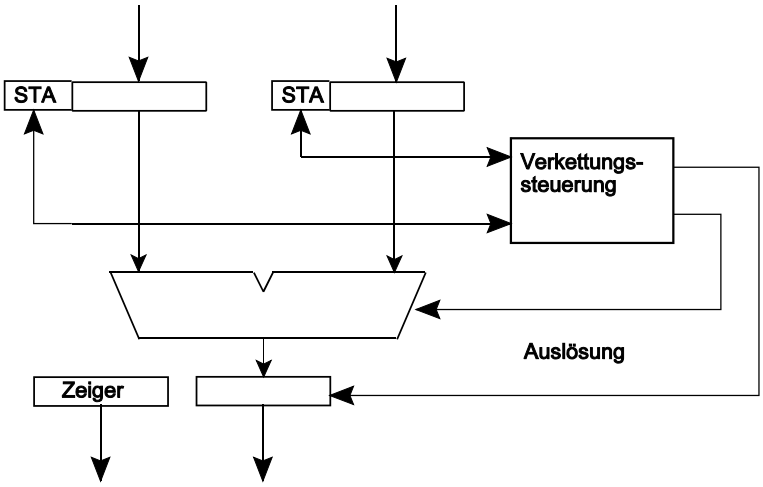


Abb. 2.28 Ressource mit Gültigkeitssteuerung

Ein c-Operator veranlaßt das Laden des Zeigers und das Aktivieren der Verkettung. Wirkung gemäß der Zustandscodierung von Abb. 2.27: STA0 = 0 (frei); STA2=1 (verkettet).

Ein d-Operator veranlaßt das Deaktivieren der Verkettung. Wirkung gemäß der Zustandscodierung von Abb. 2.27: STA2=0 (nicht verkettet). Damit werden die anderen Bitpositionen wirkungslos.

Typische Verkettungsabläufe (Abb. 2.29):

1. Die Operandenregister sind zunächst frei (STA0 = 0).
2. Die Zulieferung von Operanden macht sie gültig (STA = 1). Ein Operand wir immer dann gültig, wenn er angeliefert wird, gleichgültig auf welche Weise die Lieferung erfolgt (Verkettung, p-Operator, l-Operator).
3. Sind alle Operandenregister einer Ressource gültig, werden die Ergebnisse berechnet.
4. Stehen die Ergebnisse in ihren Registern bereit, werden die Operandenregister wieder freigemacht. Ausnahme: Eingangsregister mit Zustand „Gültigkeit halten“ (STA1=1) bleiben gültig. Diese Betriebsweise wird durch spezielle p-Operatoren oder u-Operatoren eingestellt. Nutzung: u. a. für Festwerte, die nur einmal eingetragen werden.
5. Die verketteten Ergebnisse werden an ihren Zielen abgeliefert, falls diese frei sind. Ansonsten wird gewartet (oder in einen Puffer geschrieben).

Generelle Verkettungserlaubnis
Gelegentlich ist es zweckmäßig, die Verkettungsmechanismen der Ressource insgesamt ein- und ausschalten zu können, z. B. um zu vermeiden, daß während des Einrichtens oder Umstellens der Betriebsart einzelne Steuerautomaten unberechtigt anlaufen. Hierzu können u. a. spezielle u-Operatoren vorgesehen werden. Auch ist es möglich, den y-Operator als generelle Starterlaubnis zu verwenden

(nach dem Einstellen der Betriebsweise wird die Ressource mittels y-Operator gleichsam scharfgeschaltet).

Bedingte Verkettung
Manche Ressourcen lösen Verkettungsabläufe mit bestimmten Ergebnissen nur unter bestimmten Bedingungen aus. Abb. 2.30 veranschaulicht ein Beispiel.

Verkettung zwecks Funktionsaulösung
Manche Ressourcen haben einen eingangsseitigen Parameter, der nur zur Funktionsaulösung dient und in die Verkettung einbezogen ist. Der Zweck besteht u. a. darin, die Verarbeitungsfunktionen in der Ressourcen anzustoßen oder – durch entsprechend verzögertes Auslösen der Signalisierung – zeitweise aufzuhalten.

Varianten:

1. Verkettung ohne Operanden (bloßes Auslösen).
2. Verkettung mit Operanden (Funktionscodes). Hierbei ist es zweckmäßig, folgende Belegungen zu unterscheiden:
   - gar keine Wirkung (No Op),
   - Ressource anhalten,
   - Ressource starten,
   - Eintragen eines neuen Funktionscodes. Ressource wird angehalten.
   - Eintragen eines neuen Funktionscodes. Ressource wird gestartet.

Eingangsseitige Verkettung zwecks paralleler Operandenzuführung
Die Verkettung von Operanden bewirkt praktisch eine Parallelschaltung der beteiligten Operandenregister (Abb. 2.31). Durch Verkettung ist es somit möglich, Operanden mit einem einzigen p-Operator gleichzeitig an mehrere Ressourcen zu schicken.

Bedingte Operandenverkettung
Im Ablauf von Abb. 2.29 werden ankommende Operanden sofort – noch vor der Berechnung des Ergebnisses – zum nächsten Ziel weitergeleitet. Hierdurch wird es möglich, daß alle entsprechend verketteten Ressourcen die Operanden nahezu gleichzeitig erhalten („Parallelschaltung“). Bedingte Operandenverkettung bedeutet, daß erst das Ergebnis berechnet und dann über die Weitergabe des Operanden entschieden wird (gemäß einer einstellbaren Bedingung). Nutzung: beispielsweise in datenabhängigen Schleifen oder zu Debugging-Zwecken (Abschnitt 6.9).

Gültigkeitssteuerung ohne Verkettung
Ressourcen mit Gültigkeitssteuerung ermöglichen es, auf y-Operatoren zur Funktionsauslösung zu verzichten. Die Ergebnisberechnung beginnt vielmehr stets dann, wenn alle Operanden gültig sind, gleichgültig auf welche Weise sie angeliefert wurden.

Einzelverkettung
Ein Ausgang kann lediglich mit einem einzigen Eingang (einer nachgeschalteten Ressource) verkettet werden. Diese Art der Verkettung führt auf inverse Baumstrukturen (Abb. 2.32).

Derartige Strukturen entstehen beim Auswerten geklammerter bzw. geschachtelter Ausdrücke. Herkömmlicherweise nutzt man hierzu elementare Stackmaschinen (Wandlung des Ausdrucks in eine umgekehrte polnische Notation, die direkt in Befehle der Stackmaschine umgesetzt wird). Sinngemäß lassen sich solche Ausdrücke in inverse Baumstrukturen umsetzen, wobei jeder Verknüpfungsoperation eine Ressource entspricht. Somit ist bereits einfachste Form der Verkettung geeignet, viele anwendungspraktisch wichtige Konfigurationen von Ressourcen zu unterstützen.

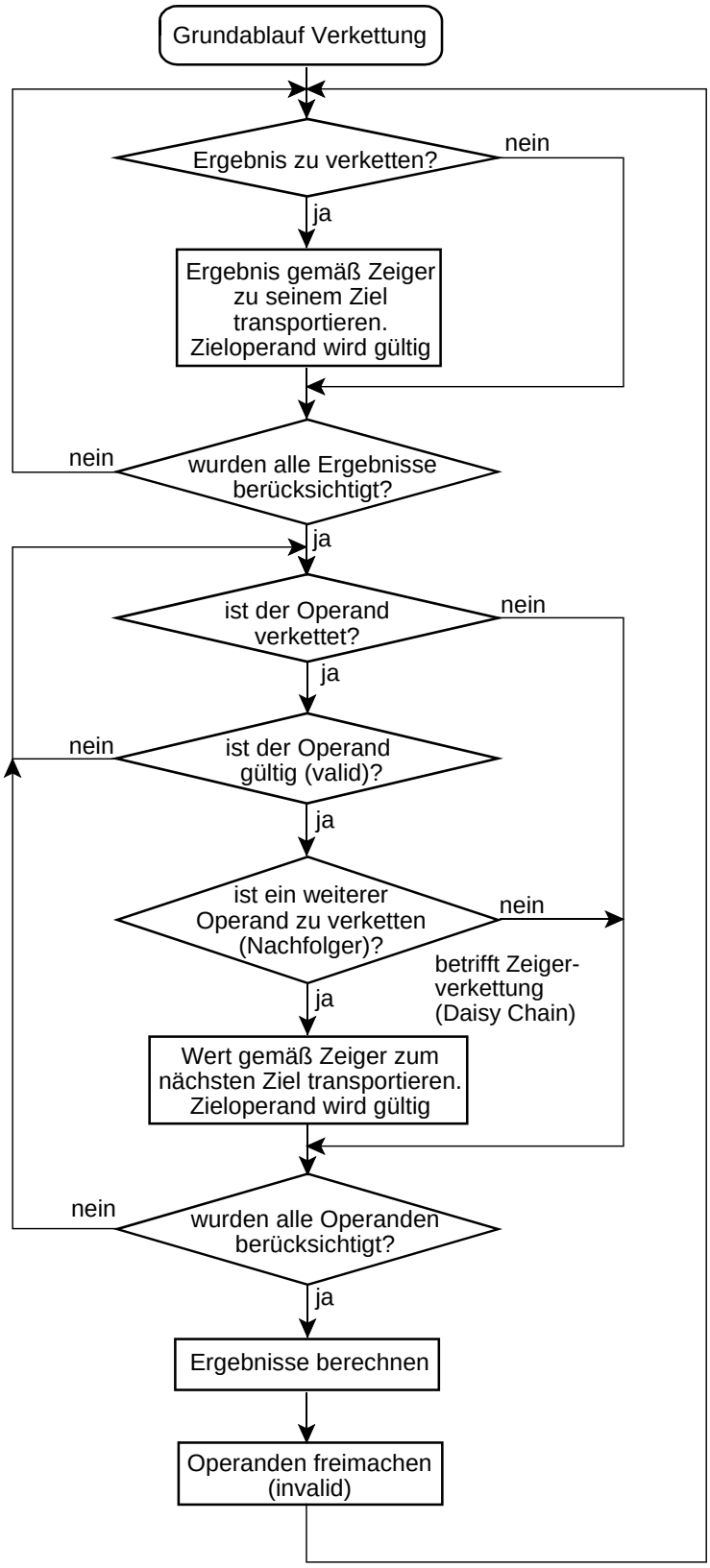
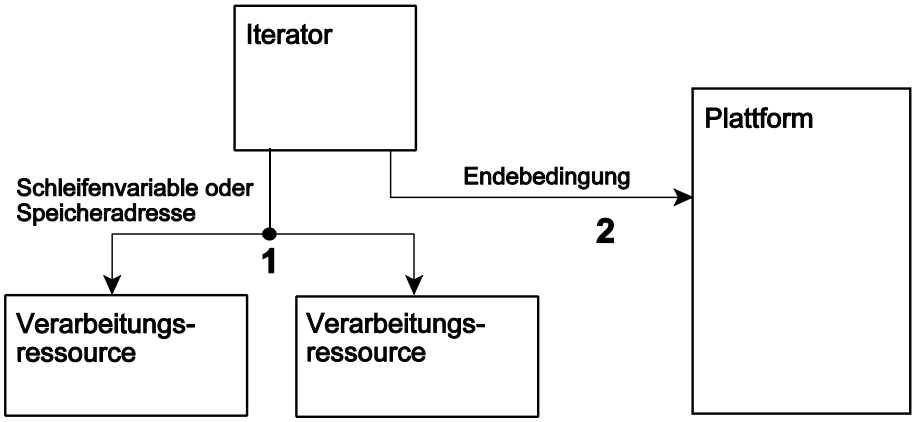


Abb. 2.29 Verkettungsablauf mit Gültigkeitssteuerung. Zur Zeigerverkettung vgl. Abb. 2.34



1 - während der Schleifendurchläufe schickt der Iterator seine aktuellen Werte zu den Verarbeitungsressourcen. 2 - ist das Schleifenende erreicht, werden die Verarbeitungsressoucen nicht mehr beliefert. Statt dessen wird die entsprechende Endebedingung zur Plattform übertragen.

Abb. 2.30 Iterator-Ressorce mit bedingter Ergebnisverkettung. Zum Aufbau des Iterators vgl. beispielsweise Abb. 2.17

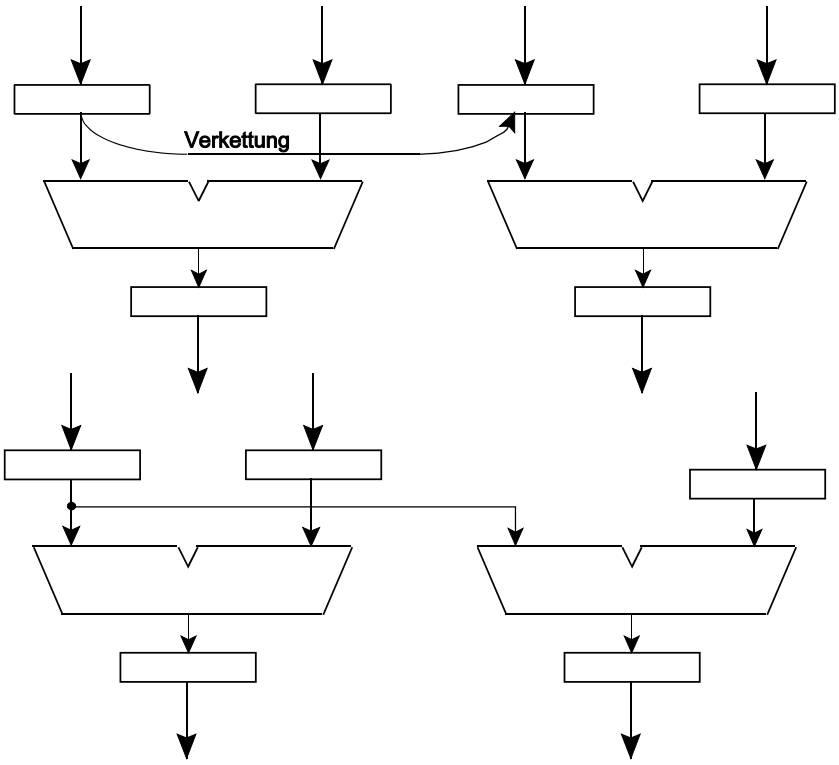


Abb. 2.31 Die eingangsseitige Verkettung (oben) entspricht einer Parallelschaltung der beteiligten Operandenregister (unten)

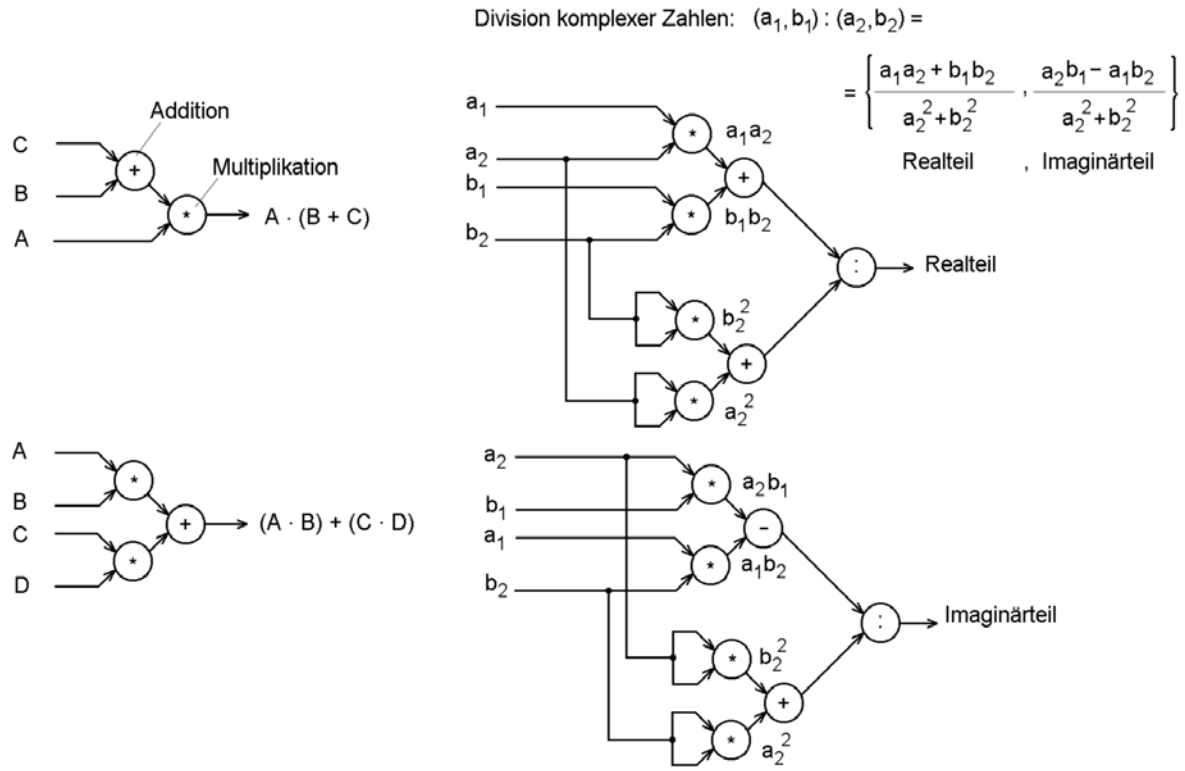


Abb. 2.32 Beispiele inverser Baumstrukturen

Mehrfachverkettung über Zeigerliste
Ausgängen, die diese Form der Verkettung unterstützen, sind Speicheranordnungen zugeordnet, die mehrere Zeigerangaben aufnehmen können. Die Zeigerangaben werden zweckmäßigerweise zu einer Zeigerliste zusammengefaßt, wobei jedem Zeiger ein Zustandsbit (Verkettungzustand) zugeordnet ist (Abb. 2.33). Hat eine solche Ressource ihre Ergebnisse berechnet, so adressiert die Zugriffsteuerung die Zeigerliste, holt alle Zeiger mit gesetztem Zustandsbit und transportiert das jeweilige Ergebnis zu den in den Zeigern angegebenen Ressourcen.

Vorteile:

- In den nachgeordneten Ressourcen sind keine besonderen Vorkehrungen erforderlich,
- bei entsprechender Auslegung der Hardware (mit mehreren Bussystemen oder über Schaltverteiler verbundenen Signalwegen) ist es möglich, ein Ergebnis gleichzeitig an mehrere verkettete Ressourcen zu senden (Parallelisierung).

Nachteil: Die Anzahl der Verkettungen ist beschränkt.

Ein Ausweg: Die Zeigerliste wird im Speicher untergebacht, so daß sie beliebig groß sein darf. Das hat aber den Nachteil, daß die Verkettungsabläufe zeitaufwendig werden.

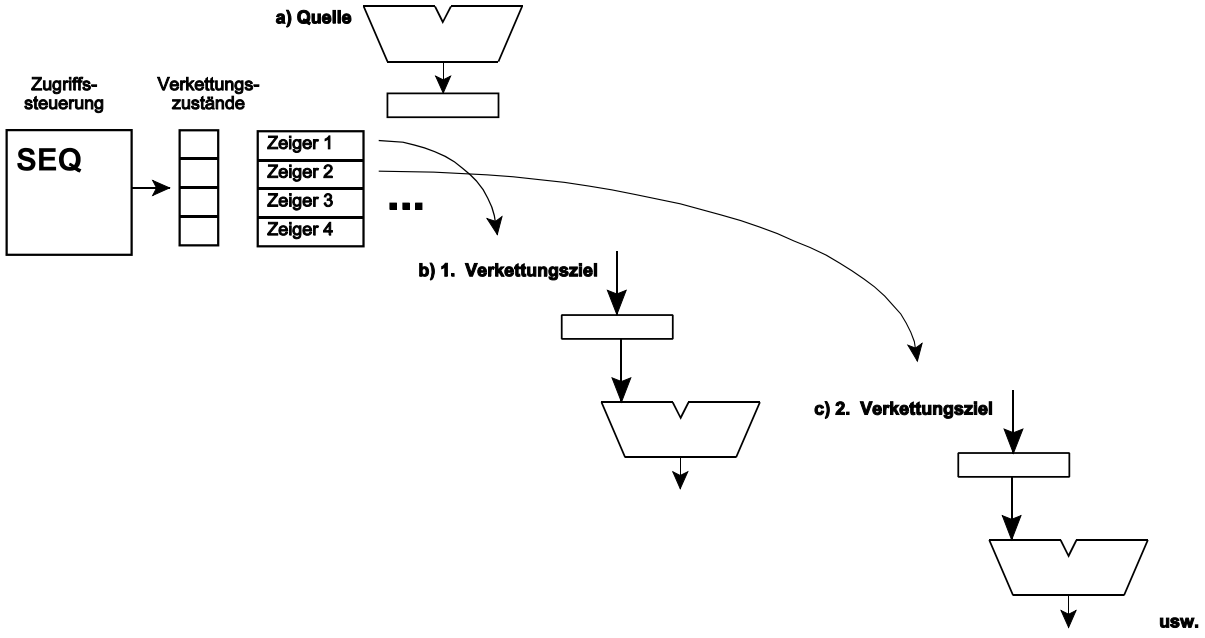


Abb. 2.33 Mehrfachverkettung über Zeigerliste

Mehrfachverkettung über Zeigerverkettung (Daisy Chain)
Hiermit kann eine Verkettung beliebig viele Ressourcen umfassen. Diese Art der Verkettung muß aber auch seitens jener Ressourcen unterstützt werden, die die verketteten Ergebnisse als Operanden empfangen. Jeder derart zu verkettende Operand braucht einen oder zwei Zeiger. Die Ressource, die das Ergebnis liefert (Quelle), hat nur einen Zeiger. Dieser verweist auf den ersten verketteten Operanden, der dort vorgesehene Zeiger auf den zweiten usw. Das Ende der Verkettung wird in der letzten Ressource durch eine entsprechende Zeigerbelegung gekennzeichnet (z. B. durch einen Nullzeiger).

Ist es nicht nötig, die Verkettung dynamisch (zur Laufzeit) umzubauen, so genügt ein einziger Zeiger je Parameter.

Um das Umbauen von Verkettungen (mit d- und c-Operatoren) zu unterstützen, können jeweils zwei Zeiger vorgesehen werden, wobei der zweite den Rückverweis auf die jeweils vorgeordnete Ressource (Vorgänger) enthält (Abb. 2.34).

Empfängt eine Ressource ein verkettetes Ergebnis, so trägt es dieses als eigenen Operanden ein. Ist ein Nachfolger angegeben, so wird das Ergebnis zur betreffenden Ressource weitergeleitet.

Ein c-Operator wirkt so, daß zunächst die bisher vorhandene Zeigerkette abgefragt wird. In der letzten Position dieser Kette wird die Endekennung (z. B. ein Nullzeiger) durch einen Zeiger auf die neu hinzugekommene Ressource ersetzt.

Ein d-Operator wirkt so, daß die Zeigerangaben der Ressource, die aus der Verkettung herausgenommen werden soll, dazu verwendet werden, um die Zeiger in der vorhergehenden und in der nachfolgenden Ressource umzustellen (Abb. 2.35):

- Die Nachfolgerangabe wird zur Nachfolgerangabe der vorhergehenden Ressource,
- die Vorgängerangabe wird zur Vorgängerangabe der nachfolgenden Ressource.

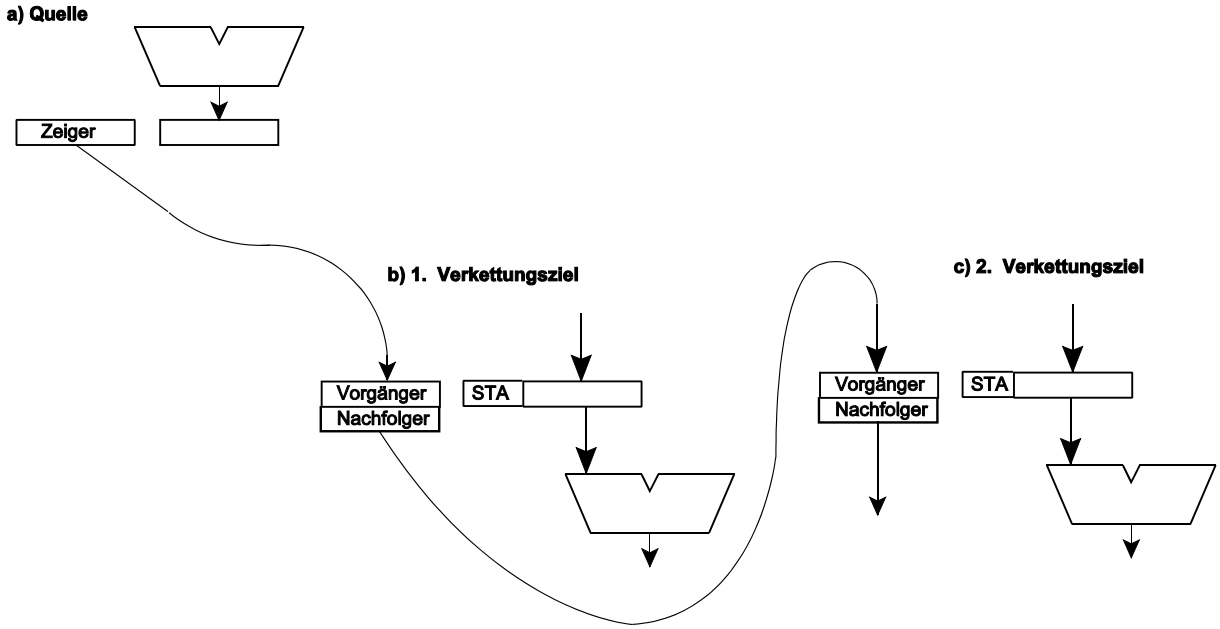
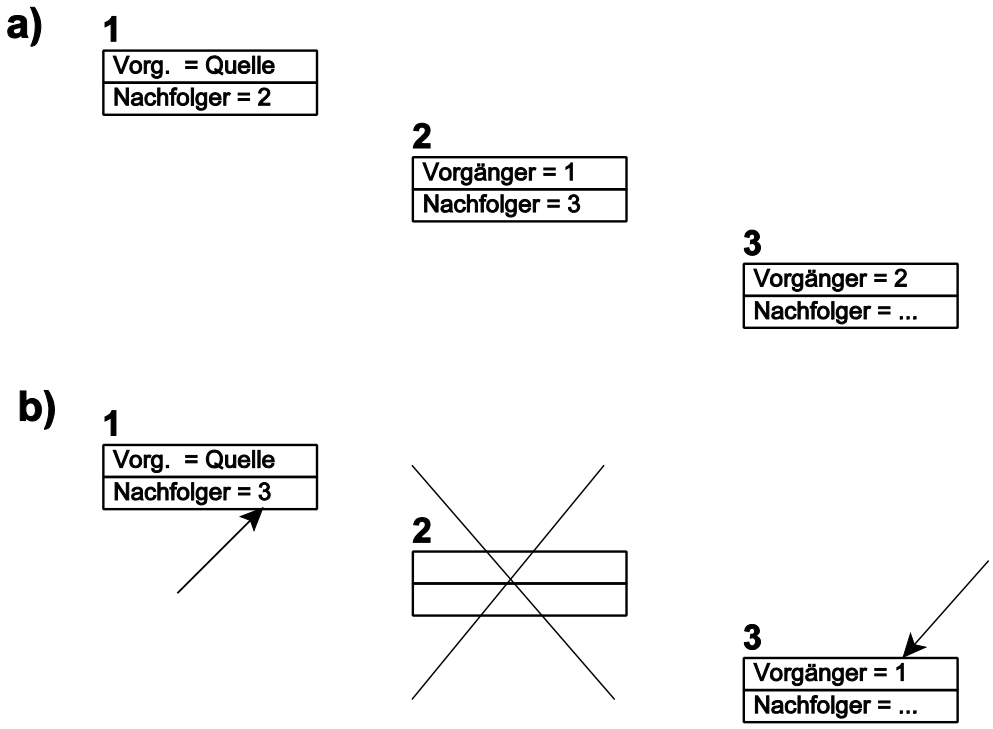


Abb. 2.34 Mehrfachverkettung über Zeigerverkettung (Daisy Chain)



a) Beispiel einer Zeigerbelegung. Der Quelle sind die Ressourcen 1, 2 und 3 nachgeschaltet.
b) die Ressource 2 wird aus der Verkettung entfernt (Operator d (Quelle => 2)). In Ressource 1 wird der Nachfolger auf Ressource 3 umgestellt, in Ressource 3 der Vorgänger auf Ressource 1. In Ressource 2 wird ggf. die Verkettungsfunktion abgeschaltet.

Abb. 2.35 So wird eine Mehrfachverkettung dynamisch umgebaut (Beispiel)

3. ReAl-Systemkonfigurationen

3.1 Grundlagen

Ein ReAl-System (Abb. 3.1) besteht grundsätzlich aus:

- Der Plattform (= den Ressourcen, die unbedingt notwendig sind, um den Betrieb des Systems einzuleiten und aufrecht zu erhalten),
- Speichermitteln für Daten, Programme, Konfigurationsangaben usw.
- den Ressourcen, die zur Erledigung der eigentlichen Verarbeitungsaufgaben vorgesehen sind.

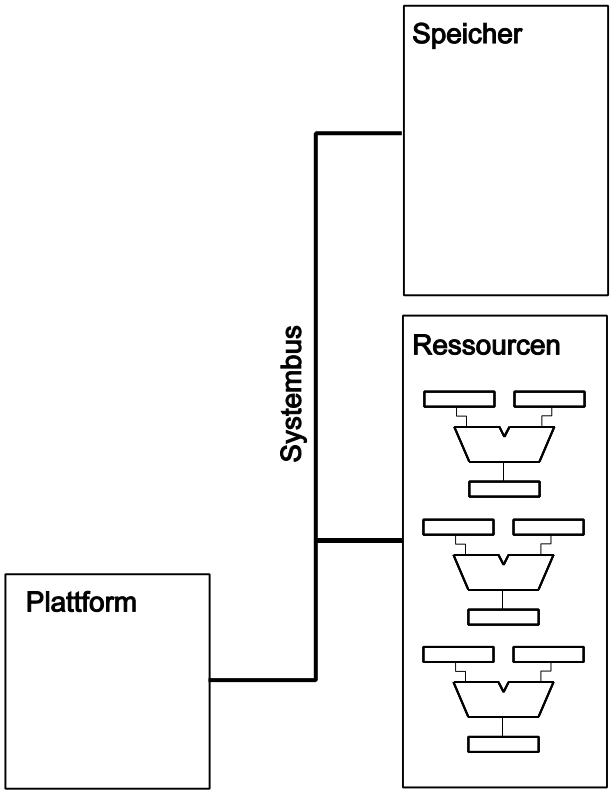


Abb. 3.1 Eine elementare ReAl-Systemkonfiguation

Elementare Konfigurationen ähneln einem herkömmlichen Universalrechner. Plattform, Speicher und Ressourcen sind beispielsweise über einen Universalbus (Systembus) untereinander verbunden. Im Gegensatz zum Universalrechner sind die Verarbeitungsressourcen aber nicht auf ein einziges Rechenwerk oder auf wenige spezielle Verarbeitungswerke beschränkt.

Herkömmliche Bussysteme haben den Nachteil, daß zu einer Zeit nur ein Übertragungsvorgang stattfinden kann. Die Alternative: schnelle Punkt-zu-Punkt-Verbindungen, die über Koppelfelder bzw. Schaltverteiler geführt sind (Abb. 3.2). Punkt-zu-Punkt-Hochgeschwindigkeitsinterfaces mit Datenraten von mehreren GBits/s und hinreichend kurzen Latenzzeiten entsprechen dem Stand der Technik. Schaltverteiler ermöglichen es, beliebige Einrichtungen untereinander zu verbinden. Dabei können viele unabhängige Übertragungsvorgänge gleichzeitig stattfinden.

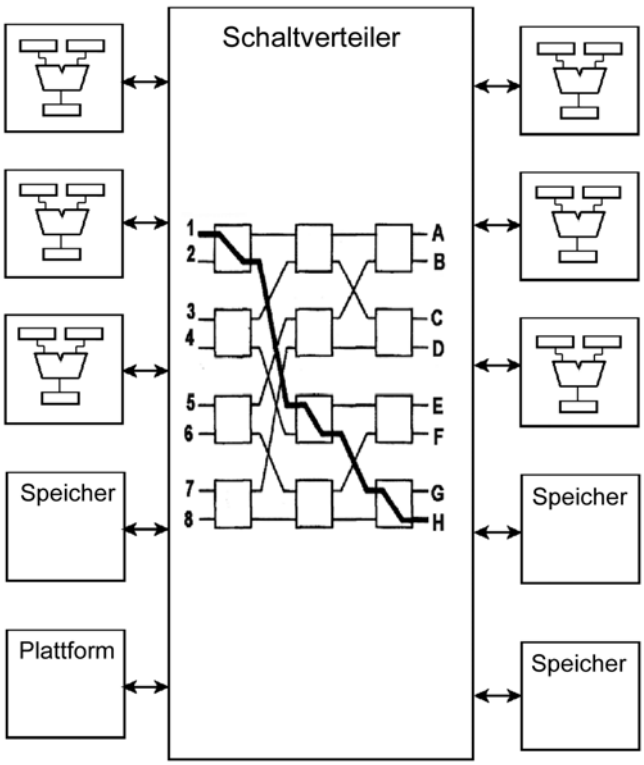


Abb. 3.2 ReAl-Systemkonfiguration mit Schaltverteiler

3.2 Ressourcen mit eigenen Speichermitteln

Der Datenaustausch zwischen Verarbeitungsressourcen und Speicher ist typischerweise zeitaufwendig. Um diese Transportzeiten zu verringern, können die Ressourcen mit eigenen Speichermitteln ausgerüstet werden.

Varianten:

- Jede einzelne Verarbeitungsressource hat eigene Speichermittel (Abb. 3.3),
- mehrere Verarbeitungsressourcen sind zu einem Bündel (Cluster) zusammengefaßt und mit gemeinsamen Speichermitteln verbunden (Abb. 3.4).

Aktive Speicheranordnungen
Mehrere (auch: sehr viele) Anordnungen aus Verarbeitungsressourcen und Speichermitteln werden zu einer sog. aktiven Speicheranordnung zusammengefaßt (Abb. 3.5). Die einzelne Ressourcenanordnung mit Speicher (z. B. gemäß Abb. 3.3) stellt praktisch eine aktive Speicherzelle dar (Ressourcenzelle). Die gesamte Speicherausstattung bildet einen einzigen zusammenhängenden Adreßraum.

In der aktiven Speicheranordnung bilden Speicher und Verarbeitungsressourcen eine Einheit. Der Aufbau großer Halbleiterspeicher (vor allem DRAMs) aus einer Vielzahl von Speichermatrizen (Banks) gehört seit längerem zum Stand der Technik, ebenso die Verbindung dieser Speicherbanks mit synchron arbeitenden (taktgesteuerten) Hochleistungsschnittstellen. Eine aktive Speicheranordnung kann beispielsweise gebildet werden, indem jeder Speicherbank und ihrer Schnittstellenhardware entsprechende Verarbeitungsschaltungen zugeordnet werden (Abb. 3.6).

Auch Caches, Registersätze, Pufferspeicher und zweckgebundene Speicherbereiche (z. B. für Objekttabellen oder für Stack Frames) können mit Verarbeitungsressourcen versehen werden.

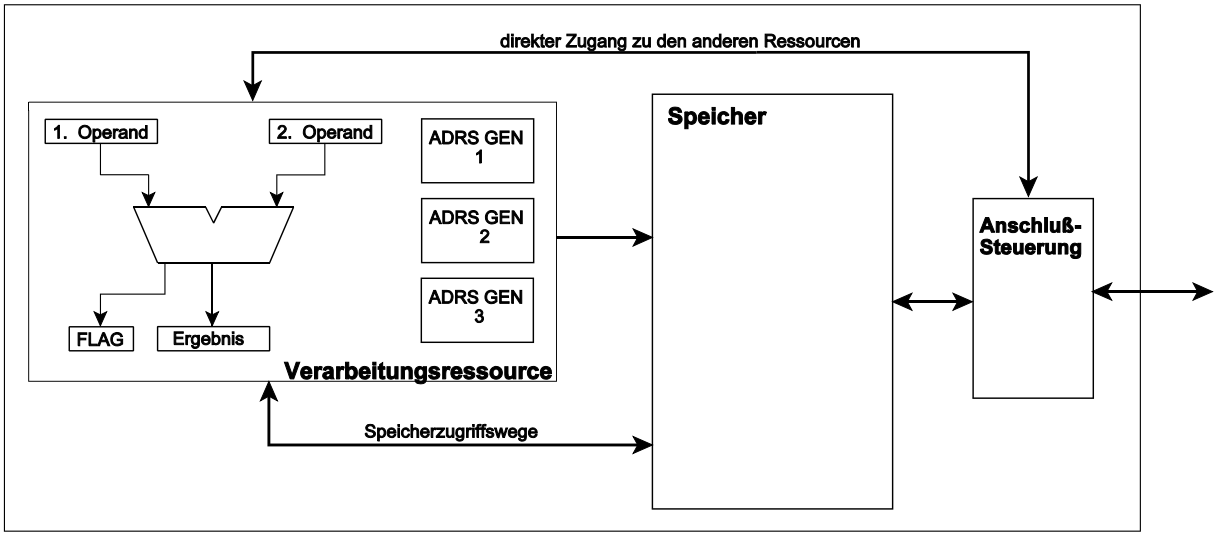


Abb. 3.3 Verarbeitungsressource mit eigenem Speicher

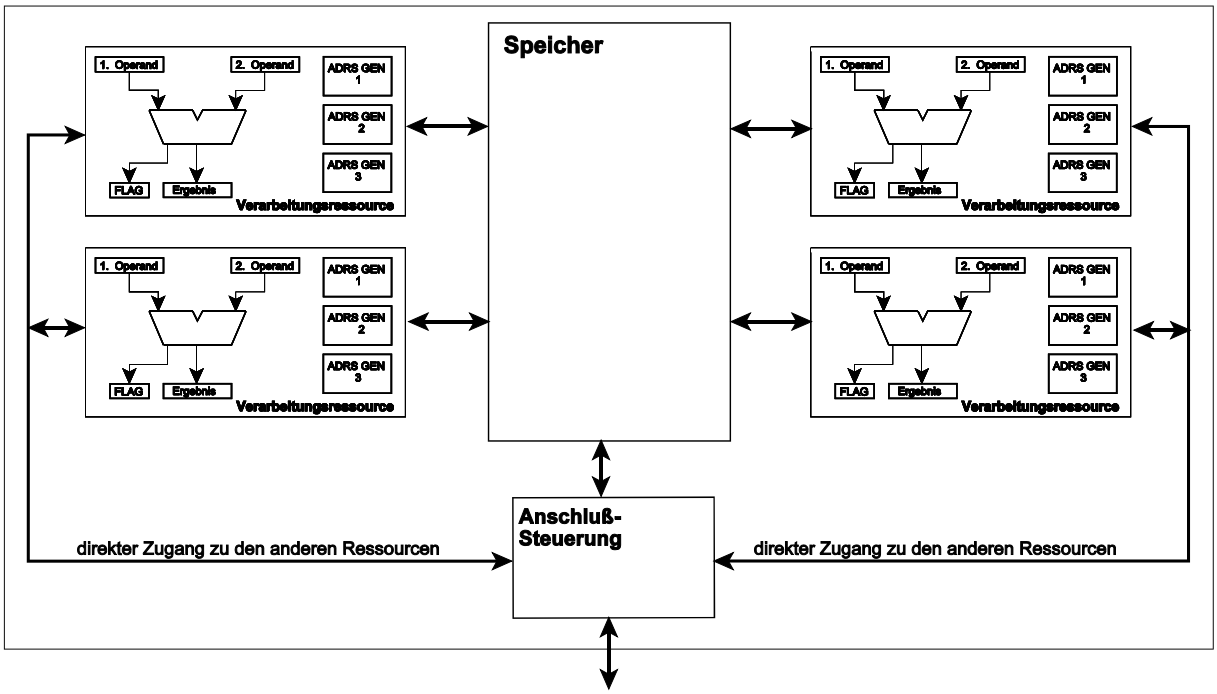


Abb. 3.4 Mehrere Verarbeitungsressourcen mit gemeinsamen Speicher (Cluster-Konfiguration)

Speicheranordnungen mit eingebauten Verarbeitungsressourcen können auf vielfältige Weise genutzt werden, beispielsweise im Rahmen der bekannten Verfahren zur Parallelverarbeitung. Zudem ist es u. a. möglich, jedem Funktionsaufruf eigene Ressourcenzellen zuzuordnen. Die Vorteile:

- Zur Laufzeit entfällt der Verwaltungsaufwand (Overhead); es sind nur die aktuellen Parameter zu transportieren, aber keine Stack-Frames auf- und abzubauen, lokale Variable zu initialisieren usw.
- alle Funktionen, deren Aufruf irgendwie parallelisierbar ist, können auch tatsächlich parallel ausgeführt werden.

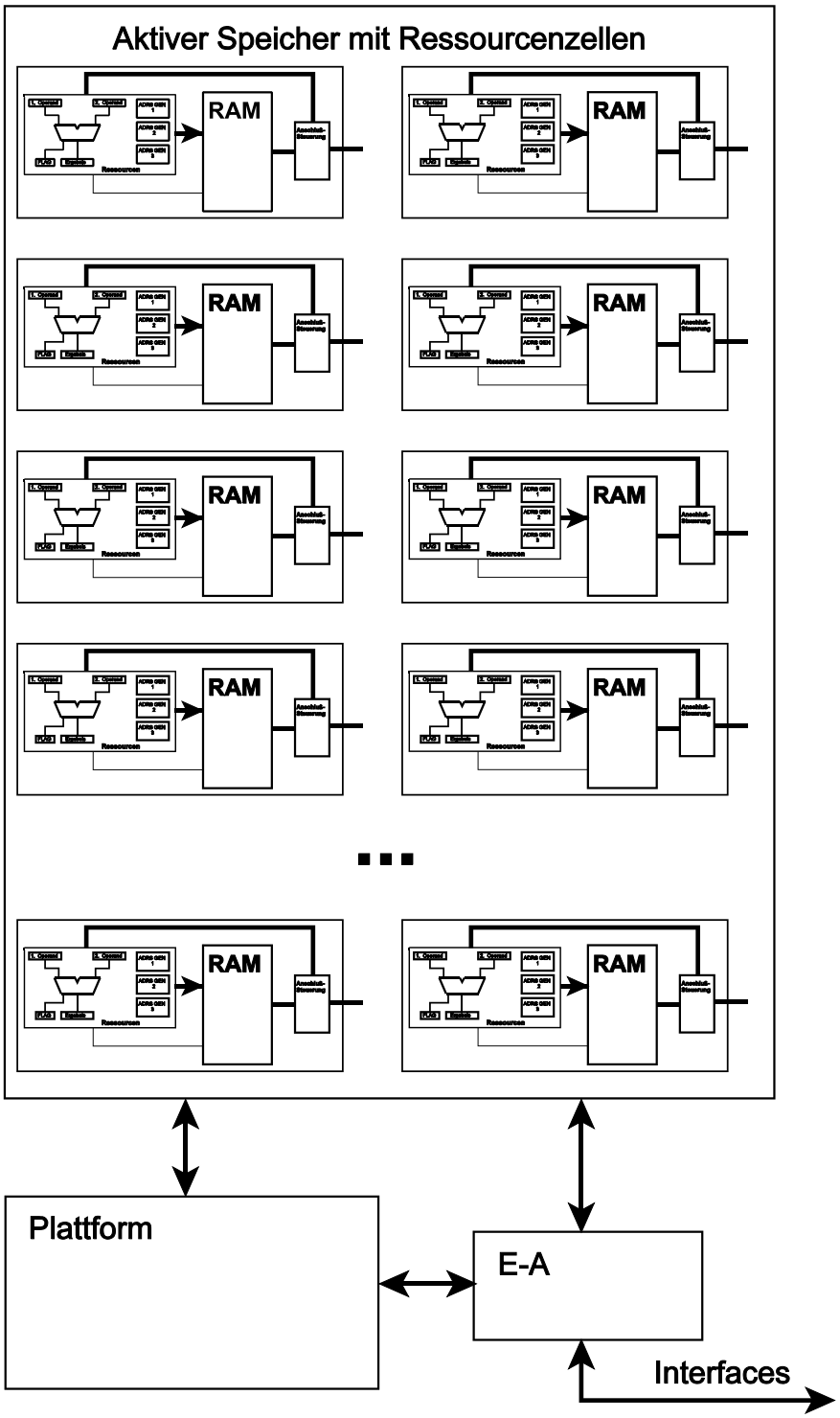


Abb. 3.5 Als aktive Speicheranordnung ausgebildetes ReAl-System

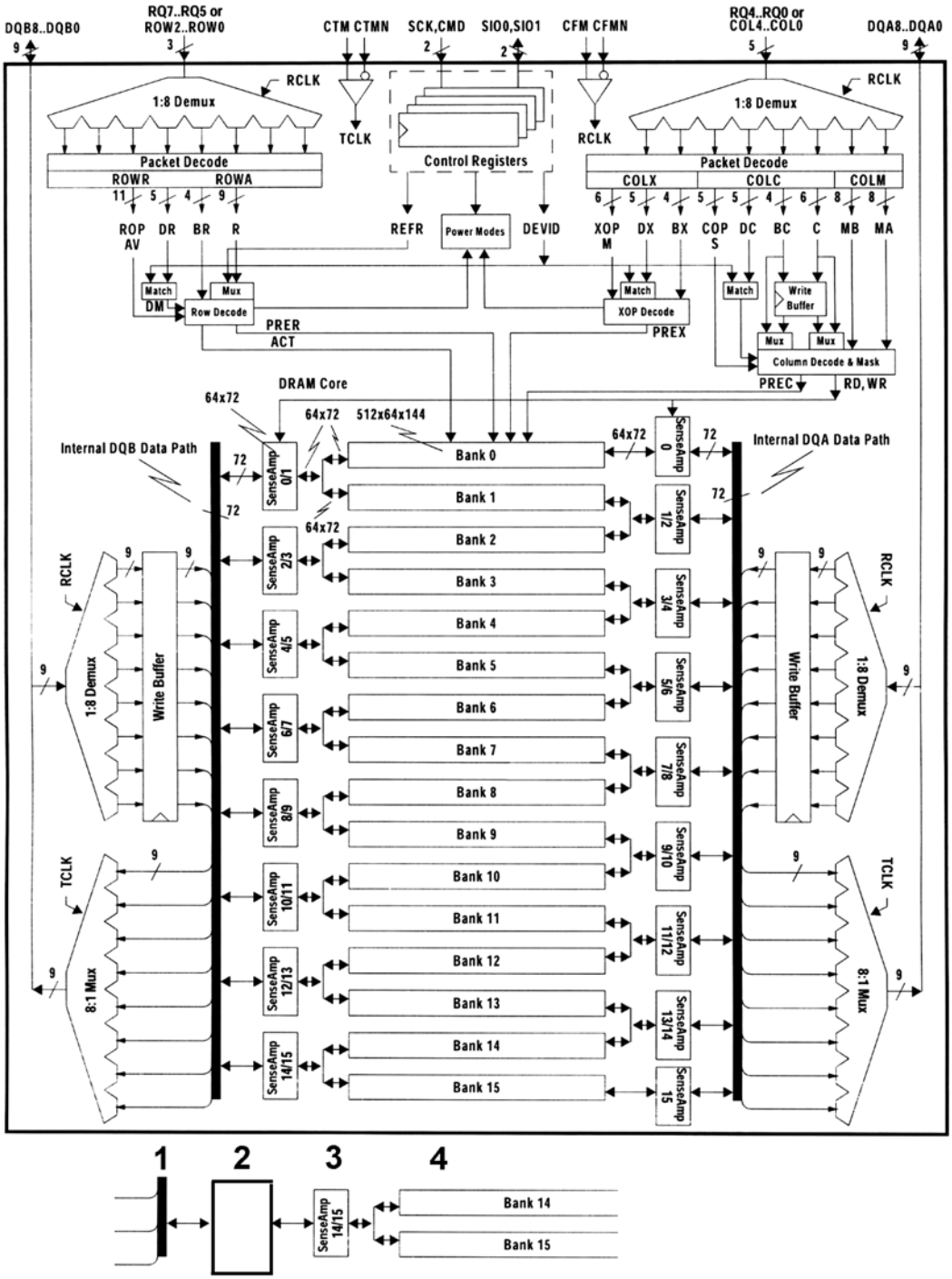


Abb. 3.6 Ein moderner Speicherschaltkreis (Beispiel: DRDRAM). Zwischen den Leseverstärkern und den internen Datenwegen können Verarbeitungsressourcen angeordnet werden. 1 - interner Datenbus; 2 - Verarbeitungsressourcen; 3 - Leseverstärker; 4 - Speicherbanks

Verarbeitungsressourcen in Caches

Caches sind typischerweise in Form von Blöcken (Cache Lines) organisiert. Zu jedem Block gehört ein Datenteil (der üblicherweise 32...256 Datenbytes aufnehmen kann) und ein Assoziativteil. Der Assoziativteil enthält die jeweilige Speicheradresse. Wird ein Speicherzugriff eingeleitet, so wird die Zugriffsadresse mit dem Inhalt des Assoziativteils verglichen. Bei Gleichheit (Cache Hit) wird kein Arbeitsspeicherzugriff ausgeführt, sondern ein Zugriff zum jeweiligen Datenteil des Caches. Hierdurch ergibt sich eine beträchtlich kürzere Zugriffszeit. Bei Ungleichheit (Cache Miss) findet hingegen der Arbeitsspeicherzugriff statt. Dabei wird der Arbeitsspeicherinhalt in den Datenteil des Blocks transportiert (so daß er beim nächsten Zugriff dort bereitsteht). Die Datenteile der einzelnen Blöcke werden mit Verarbeitungsressourcen verbunden (Abb. 3.7). Befinden sich die gewünschten Daten im Cache (Cache

Hit), so können sie von den eingebauten Ressourcen unmittelbar verarbeitet werden. Das jeweilige Trefferanzeigsignal (ADRS MATCH in Abb. 3.7) erlaubt die Ausführung der entsprechenen Operatoren. Ist ADRS MATCH inaktiv, so handelt es sich um einen Cache Miss, und die betreffenden Daten müssen zunächst auf an sich bekannte Weise (wie bei herkömmlichen Caches) aus dem Arbeitsspeicher geholt werden. Eine solche Auslegung erspart den Einbau von Verarbeitungsressourcen in DRAM-Schaltkreise, ergibt aber für übliche Programme (ohne massive Parallelverarbeitung) ähnliche Verarbeitungsleistungen (Trefferrate des Caches ≈ Verarbeitungsleistung einer aktiven Speicheranordnung, die die gesamte Arbeitsspeicherkapazität umfaßt).

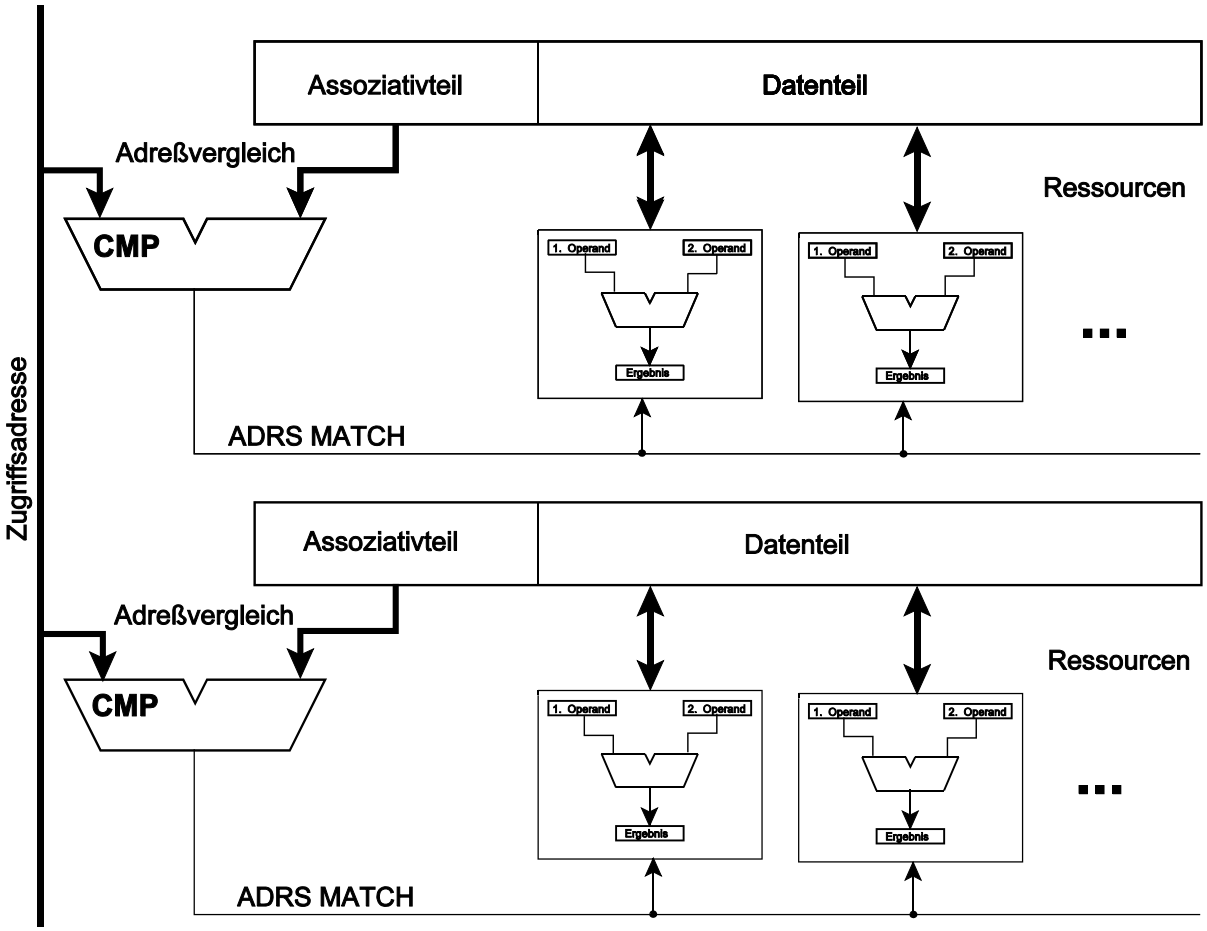


Abb. 3.7 Aktive Cache-Anordnung mit eingebauten Verarbeitungsressourcen

Verarbeitungsressourcen in Registersätzen

Große universelle Registersätze (mit typischerweise 32 bis ca. 256 Registern) sind seit längerem üblich, wenn es darum geht, besonders leistungsfähige Prozessoren zu bauen. Sie werden vor allem als Schnellzugriffsspeicher für jene Daten genutzt, mit denen gerade gerabeitet wird (z. B. für die lokalen Variablen der aktuell ausgeführten Funktion). Entsprechende Programme laufen so ab, daß zunächst die betreffenden Variablen in die Register geladen werden. Die eigentlichen Verarbeitungsbefehle beziehen sich dann nur noch auf die Register; sie führen keine Speicherzugriffe aus. Am Ende werden die Ergebnisse aus den Registern in den Arbeitsspeicher transportiert (Load-Store-Prinzip). In üblichen Prozessoren dieser Art holen die Verarbeitungsbefehle die zu verknüpfenden Daten aus den Registern und bringen die Ergebnisse dorthin zurück. Eine übliche Verknüpfung der Art X := A OP B (Dreiadreßoperation) erfordert entweder drei aufeinanderfolgende Registerzugriffe oder eine Registeranordnung mit drei Zugriffswegen. Soll mehr als eine Verknüpfung zu einer Zeit ausgeführt werden (Superskalarprinzip), sind entsprechend viele zusätzliche Zugriffswege erforderlich. Solche Registeranordnungen brauchen viel Platz auf dem Schaltkreis. Sind extrem hohe Taktfrequenzen gewünscht, so müssen mehr Stufen in der Verarbeitungspipeline vorgesehen werden (der Ablauf Lesen => Verknüpfen => Zurückschreiben ist nicht in einem einzigen Taktzyklus zu schaffen). Das hat höhere Latenzzeiten zur Folge, so daß in all den Fällen, in denen der fortlaufende Fluß durch die Pipeline unterbrochen werden muß, die Verarbeitungsleistung deutlich absinkt.

Um dies zu vermeiden, werden die Register direkt mit Verarbeitungressourcen verbunden (Abb. 3.8). Im Beispiel sind je vier Register an eine übliche Arithmetik-Logik-Einheit (ALU) angeschlossen. Es ist aber auch möglich, spezielle Verarbeitungswerke vorzusehen.

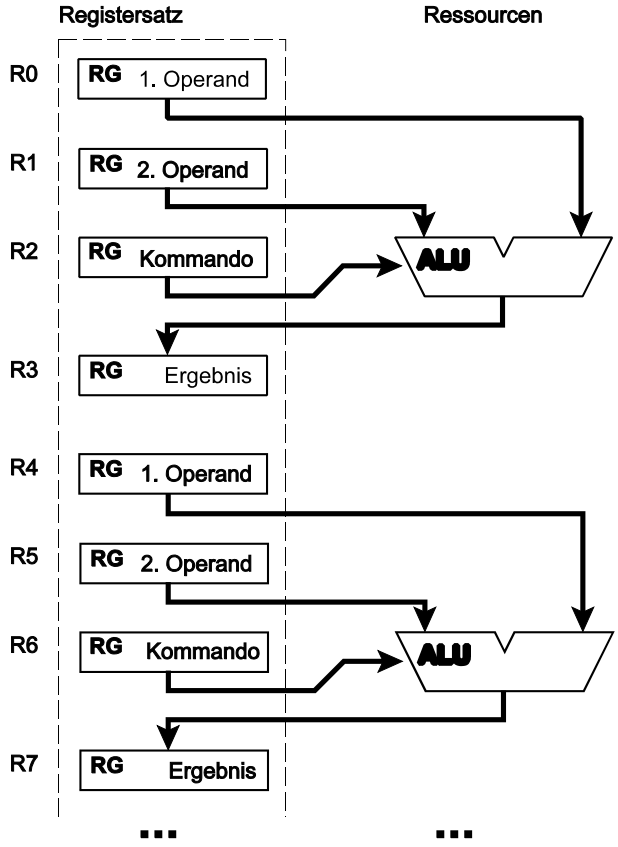


Abb. 3.8 Aktiver Registersatz mit eingebauten Verarbeitungsressourcen (1)

Zwei Register nehmen die Operanden auf (Operandenregister), eines das Ergebnis (Ergebnisregister) und eines eine Angabe, die die auszuführende Operation beschreibt (Kommandoregister).

Laden des Kommandoregisters: vorzugsweise über s-Operatoren (der s-Operator weist der an sich universellen Verarbeitungsressource (ALU) eine spezielle Funktion zu).

Eine solche Registeranordnung braucht nur zwei Zugriffswege, einen für die angeschlossenen Ressourcen und einen für allgemeine Lese- und Schreibzugriffe (Datentransport). Der zweite Zugriffsweg kann auf Schreibzugriffe zu den Operanden- und Kommandoregistern und auf Lesezugriffe zu den Ergebnisregistern beschränkt werden. Derart eingeschränkte Zugriffe können parallel zu den eigentlichen Verarbeitungszugriffen stattfinden. Die Operandenregister lasssen sich über ALU und Ergebnisregister lesen, das Ergebnisregister läßt sich über die ALU von jedem der Operandenregister aus laden.

Eine Anordnung aus beispielsweise 32 Registern könnte 8 Verarbeitungseinheiten (ALUs) enthalten, eine aus 128 Registern 32 Verarbeitungseinheiten. Die direkten Verbindungen zwischen Registern und Verarbeitungsschaltungen ermöglichen es, mit wenigen Pipeline-Stufen auszukommen (oft mit einer einzigen).

Es sind allerdings keine wahlfreien Verknüpfungen mehr möglich, z. B. <R7> := <R1> OP <R22>. Vielmehr müssen die zu verknüpfenden Operanden in die der betreffenden Ressource vorgeschalteten Register geschafft werden. Kompliziertere Verarbeitungsabäufe bestehen somit aus Folgen von Verknüpfungen und anschließenden Transportvorgängen (wodurch die Ergebnisse zu Operanden der folgenden Verknüpfungen werden).

Um einen Teil dieser Transportvorgänge einzusparen, kann man bestimmte Register kombiniert als Operanden- und Ergebnisregister verwenden, so daß das aktuelle Ergebnis als Operand an der nächsten Verknüpfung teilnehmen kann (Abb. 3.9).

Varianten:

- Das Ergebnis wird auf Operandenregister zurückgeführt, die somit als eine Art Akkumulator wirken,
- das Ergebnis wird in Operandenregister anderer Ressourcen eingetragen (elementare Form der Verkettung).

Diese Varianten können einzeln oder in Kombination vorgesehen werden. In Kombinationslösungen wird die Art der Ergebnisspeicherung über einen Operationscode gesteuert. Sind alle beschriebenen Auslegungen kombiniert, ergeben sich folgende Möglichkeiten:

1. Rückführung auf Operandenregister.
2. Speichern im Ergebnisregister.
3. Weitergabe in Operandenregister der nächsten Ressource.
4. Beliebige Kombinationen von 1. bis 3.

Da der Registersatz vergleichsweise einfach aufgebaut ist (es ist nicht nötig, mehrere parallele Zugriffswege vorzusehen), kann man auf einer gegebenen Schaltkreisfläche mehr Register unterbringen. Hierdurch wird es möglich, Variablenwerte ggf. mehrfach zu speichern und somit Transporte während der Verarbeitungsabläufe zu vermeiden.

Um die Transporte zu beschleunigen, ist es von Vorteil, die Zugriffsweg so auszulegen, daß eine Datenbelegung gleichzeitig in mehrere Zielregister geladen werden kann (Abb. 3.10).

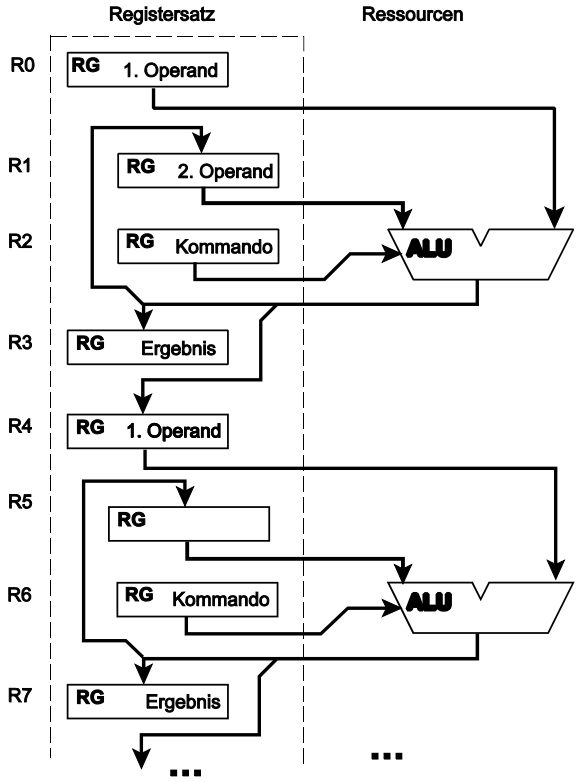


Abb. 3.9 Aktiver Registersatz mit eingebauten Verarbeitungsressourcen (2)

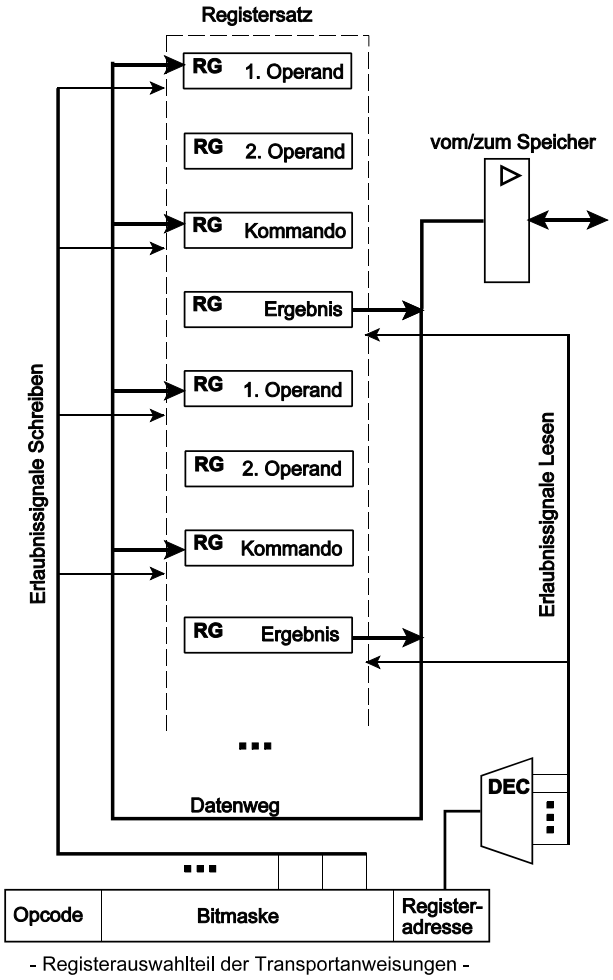


Abb. 3.10 Registersatz mit universellem Transportweg

Beispiel:
Der Registersatz besteht aus 128 Registern. Jeweils 4 Register sind einer Verarbeitungsressource zugeordnet. Es gibt insgesamt 32 Verarbeitungsressourcen. Zwei Register sind jeweils für Schreibzugriffe zugänglich, eines für Lesezugriffe. Die entsprechenden Transportanweisungen haben eine Bitmaske zum Steuern des Ladens und ein Registeradreßfeld zum Auswählen des zu lesenden Registers. Es gibt folgende Arten von Transportanweisungen:

- Laden mit Kommandocodes (entspricht s-Operator),
- Laden mit Datenwort vom Speicher (entspricht p-Operator),
- Speichern Ergebnis (entspricht a-Operator),
- Datentransport zwischen Registern (entspricht l-Operator).

Durch entsprechendes Setzen der Bitmaske in den Transportanweisungen ist es möglich, eine beliebige Kombination von Registern auf einmal zu laden. Weitere Zugriffe werden über die ALUs geführt. Hierzu sind entsprechende Kommandocodes zu laden.

Ein Kommandoregister kann mehrere Kommandocodes aufnehmen.

1. Ausführung: es ist als Schieberegister ausgebildet, wobei jeder y-Operator jeweils einen Kommandocode zur Wirkung bringt. Anschließend wird dieser Kommandocode heraus- und zum anderen Ende wieder hereingeschoben (Abb. 3.11). Aufeinanderfolgende y-Operatoren bewirken, daß zyklisch Kommando für Kommando ausgeführt wird.

2. Ausführung: die y-Operatoren haben Adreßangaben, über die jeweils eines der gespeicherten Kommandos ausgewählt wird.

3. Ausführung: nach der Aktivierung mittels y-Operator laufen die Kommandos selbsttätig ab (es handelt sich um eine Art kurze Maschinenbefehle).

In einer alternativen – eher dem Herkömmlichen angenäherten – Auslegung enthält das Kommandoregister nur einen Kommandocode, der vom jeweiligen y-Operator eingetragen und sofort zur Wirkung gebracht wird. Hierbei ist es zweckmäßig, die Kommandoregister gesondert zu adressieren (so daß sie keine Positionen des eigentlichen Registeradreßraums belegen).

Die beschriebenen Auslegungen ermöglichen es, die Parallelausführung mehrerer Operationen (im Beispiel bis zu 32) mit vergleichsweise kompakten Befehlsformaten zu unterstützen (Abb. 3.12).

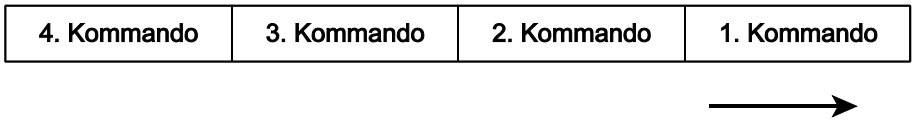


Abb. 3.11 Als Schieberegister ausgebildetes Kommandoregister (Belegungsbeispiel)

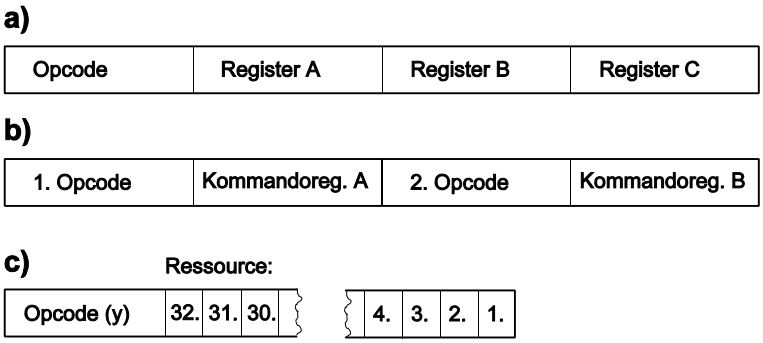


Abb. 3.12 Befehlsformate im Vergleich

a) Operationsbefehl eines marktgängigen Hochleistungsprozessors. Der Registersatz umfaßt 128 Register. Der einzelne Befehl ist 41 Bits lang. Ein 128-Bit-Wort kann drei solcher Befehle aufnehmen. Es ist möglich, alle drei Befehle parallel auszuführen.
b) modifiziertes Befehlsformat zum Steuern der zuvor beschriebenen Anordnung. Bis zu zwei Ressourcen können gleichzeitig zur Wirkung gebracht werden, indem ein Kommandocode ins jeweilige Kommandoregister geladen wird.
c) Format eines y-Operators, der mittels Bitmaske jede beliebige Kombination der 32 Verarbeitungsressourcen aktivieren kann.

Verarbeitungsressourcen in beliebigen Speicheranordnungen
Verarbeitungsressourcen können in beliebige Speicheranordnungen eingebaut werden. U. a. können Speicherbereiche, die für besondere Informationsstrukturen vorgesehen sind, eigens mit speziellen Ressourcen bestückt werden, die beispielsweise die typische Nutzung von Objekttabellen, Stack Frames, E-A-Puffern usw. in jeweils angemessener Weise unterstützen.

In den bisher beschriebenen Ausführungen sind die Verarbeitungsressourcen den einzelnen Speicherblöcken bzw. Speicherzellen fest zugeordnet. Das ergibt vergleichsweise einfache Strukturen, erfordert aber, daß sich alle von einer Ressource zu verarbeitenden Daten im jeweiligen Speicherblock befinden. Um diese Einschränkung aufzuheben, sind in einer alternativen Ausführung universelle Verbindungseinrichtungen zwischen den Ressourcen vorgesehen (Abb. 3.13). Abb. 3.14 veranschaulicht eine mit Ressourcen versehene Speicheranordnung, die zur Aufnahme von Objekttabellen vorgesehen ist.

Jede Speicherzelle nimmt Operanden auf und ist mit Verarbeitungsressourcen verbunden, z. B. mit einer universellen ALU. Ergebnisleitungen der Verarbeitungsressourcen sind auf die Speicherzelle zurückgeführt, so daß sie ggf. im Sinne eines Akkumulators genutzt werden kann. Weitere Operanden- und Ergebnisregister sind an eine universelle Verbindungseinrichtung angeschlossen, z. B. an einen Uni-

versalbus oder an einen Schaltverteiler (Switch Fabric). Somit kann jede beliebige Verknüpfung zwischen Registerinhalten ausgeführt werden. Der Aufwand ist allerdings höher, und es ergeben sich längere Transportzeiten (im Extremfall (Universalbus) kann nur ein Transportvorgang zu einer Zeit stattfinden).

Hinweis:
Im Unterschied zu den vorstehend erläuterten Auslegungen gehören hier nicht *alle* Register der Ressourcen zum Adreßraum der Speicheranordnung.

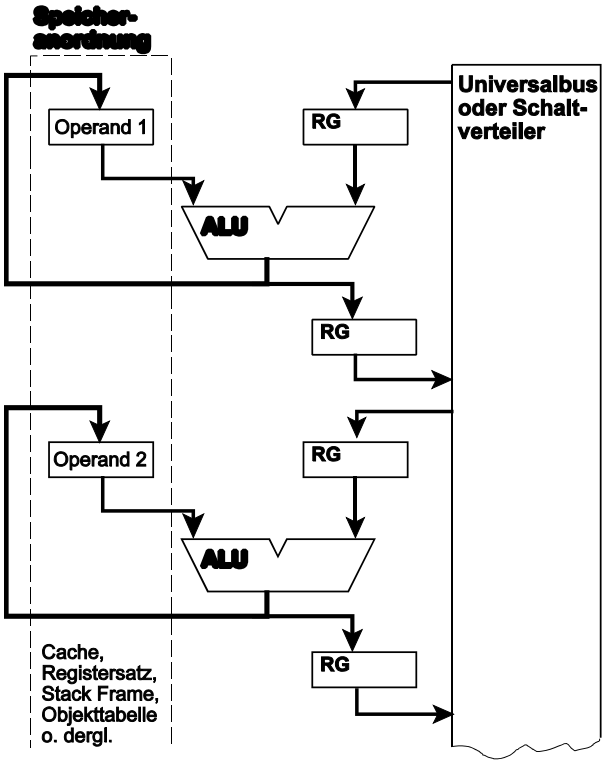
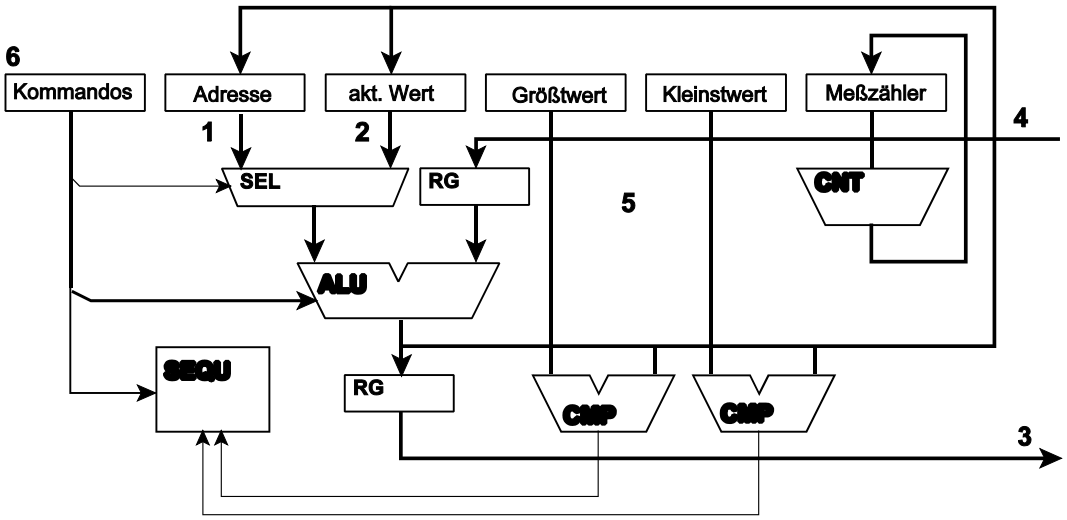


Abb. 3.13 Speicheranordnung mit Verarbeitungsressourcen



1 - Adreßrechnung; 2 - Rechnen mit dem aktuellen Wert; 3 - zu den anderen Ressourcen; 4 - von den anderen Ressourcen, 5 - Kontrolle auf Unter- oder Überschreitung des Wertebereichs; 6 - zu den Kommandos vgl. die Erläuterungen anhand der Abb. 3.10 und 3.11.

Abb. 3.14 Ressourcen in einer Position einer Objekttabelle

Im Beispiel soll die Objekttabelle die Adressen der Datenobjekte aufnehmen, aber auch das Rechnen mit

den zugehörigen Werten unterstützen. Die Objekte sind hier Zahlen, deren Wertebereich durch einen Größt- und einen Kleinstwert begrenzt ist (vgl. die Bereichsangaben (range) in der Programmiersprache Ada). Zudem ist eine Meßzählerangabe vorgesehen, um z. B. die Nutzungshäufigkeit des Objekts zu ermitteln oder um darüber Buch zu führen, wieviele Programme sich derzeit auf das Objekt beziehen (Reference Count).

4. ReAl-Konfigurationen in Schaltkreisen

Programmierbare und anwendungsspezifische Schaltkreise
Größere programmierbare Schaltkreise (Field Programmable Gate Arrays; FPGAs) enthalten herkömmlicherweise programmierbare Zellen (Logikzellen), die über ebenfalls programmierbare Verbindungswege zusammengeschaltet sind (Abb. 4.1). Die einzelnen Logikzellen sind vergleichsweise elementar. Mit einer einzigen Zelle kann man beispielsweise eine kombinatorische Verknüpfung von 4...8 Eingängen und eine Speicherstelle (= 1 Flipflop) realisieren. Der Hardwareaufwand zum Programmieren der Logikzellen und der Verbindungsnetzwerke ist beachtlich. Um einen bestimmten Funktionsumfang zu verwirklichen, braucht man oft weit mehr als die zehnfache Anzahl an Transistoren im Vergleich zum „echten“ anwendungsspezifischen Schaltkreis (auf dem die Schaltung bis auf den Transistor optimiert ist).

Deshalb enthalten manche FPGA-Schaltkreise einige besonders aufwendige und oft benötigte Schaltungskomplexe in optimierter, nicht-programmierbarer (harter) Form (das betrifft Schnittstellensteuerungen usw. bis hin zu kompletten Prozessoren). Solche Schaltkreise sind aber vergleichsweise teuer und auch weniger universell. Es kann z. B. sein, daß der eingebaute harte Prozessor für die betreffende Anwendung viel zu groß ist (Schaltkreis zu teuer) oder daß er nicht genug leistet (aufwendige Zusatzhardware erforderlich).

Eine Alternative besteht darin, Ressourcenzellen vorzusehen, die Verarbeitungsschaltungen und eine gewisse Speicherausstattung umfassen. Die Ressourcenzellen sind hart aufgebaut (sie bestehen nicht aus programmierbaren Logikzellen, sondern aus zweckgerecht verschalteten Transistoren bzw. Gattern). Einstellbar sind lediglich Funktionsweise und Verarbeitungsbreite.

Ausführungsbeispiel einer Ressourcenzelle (zum Aufbau vgl. Abb. 3.3):

- Funktionsweise: Arithmetik-Logik-Einheit (ALU),
- Verarbeitungsbreite: einstellbar; maximal: 32 Bits,
- Verkettung: wird für alle Parameter unterstützt,
- Speicherausstattung: 1 k Worte zu 32 Bits.

Die programmierbaren Verbindungswege sind für typische Nutzungsweisen der Ressourcen optimiert (sie können vergleichsweise einfach ausgelegt werden, da es nicht erforderlich ist, x-beliebige Verbindungen zwischen allen Zellen auf dem Schaltkreis zu unterstützen). Die Abb.n 4.2 bis 4.8 zeigen verschiedene Anordnungen von Ressourcenzellen. Solche Anordnungen können sowohl in programmierbaren als auch in anwendungspezifischen Schaltkreisen untergebracht werden.

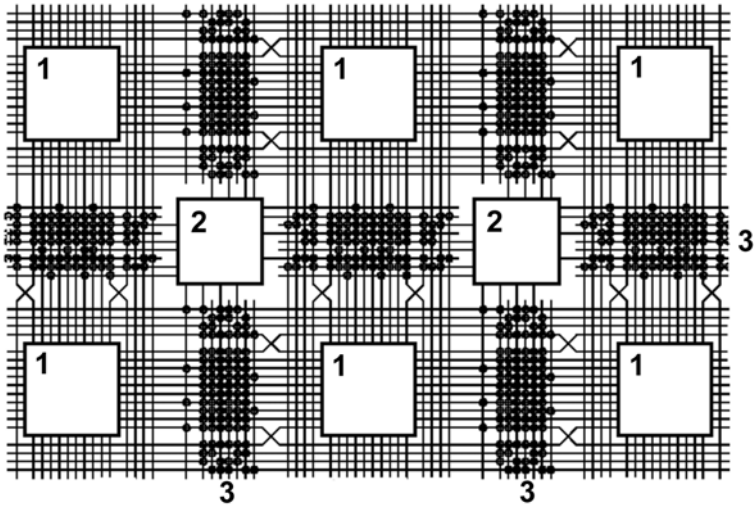


Abb. 4.1 Struktur eines herkömmlichen FPGA-Schaltkreises (nach Xilinx). 1 - programmierbare Koppelschaltungen (Verteiler); 2 - programmierbare Logikzellen; 3 - programmierbare Signalwege

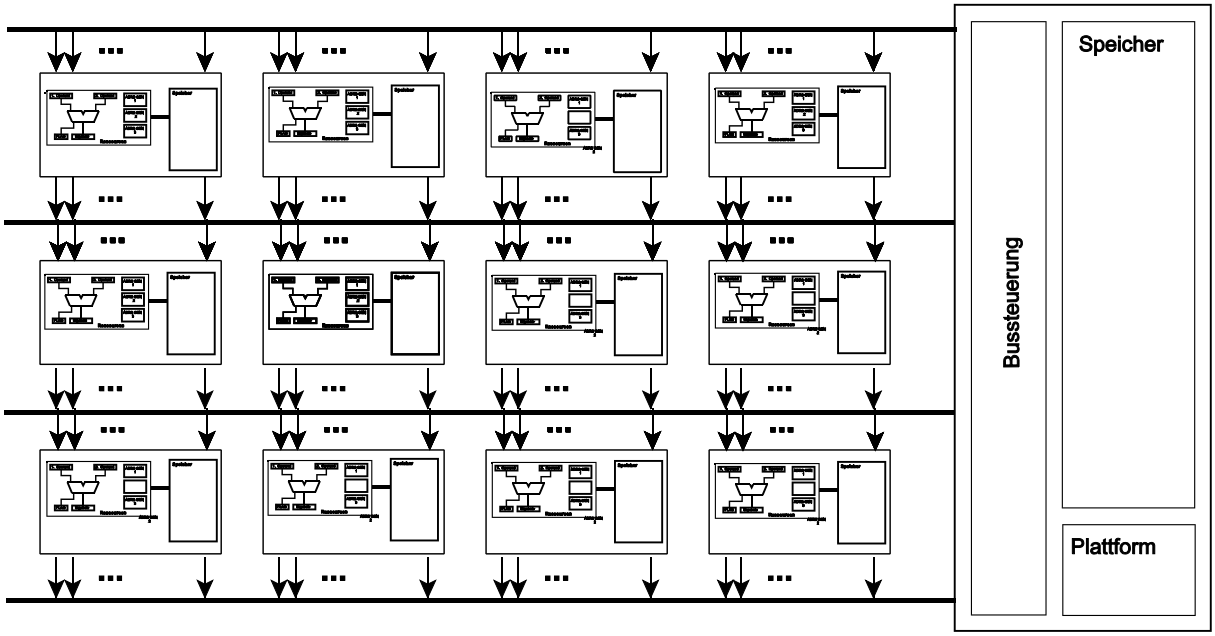


Abb. 4.2 FPGA-Schaltkreis mit Ressourcenzellen

Die Ressourcenzellen sind über Bussysteme untereinander verbunden. Jedes Bussystem ist in der Lage, eine zeilenförmige Anordnung von Ressourcenzellen mit Parametern zu versorgen und die Ergebnisse der jeweils darüberliegenden Ressourcenzelle-Zeile abzutransportieren. Die Bussteuerung stellt die Verbindung zwischen den einzelnen Bussystemen her. Auf dem Schaltkreis sind des weiteren eine gemeinsame Speicherausstattung sowie die Schaltmittel der Plattform angeordnet.

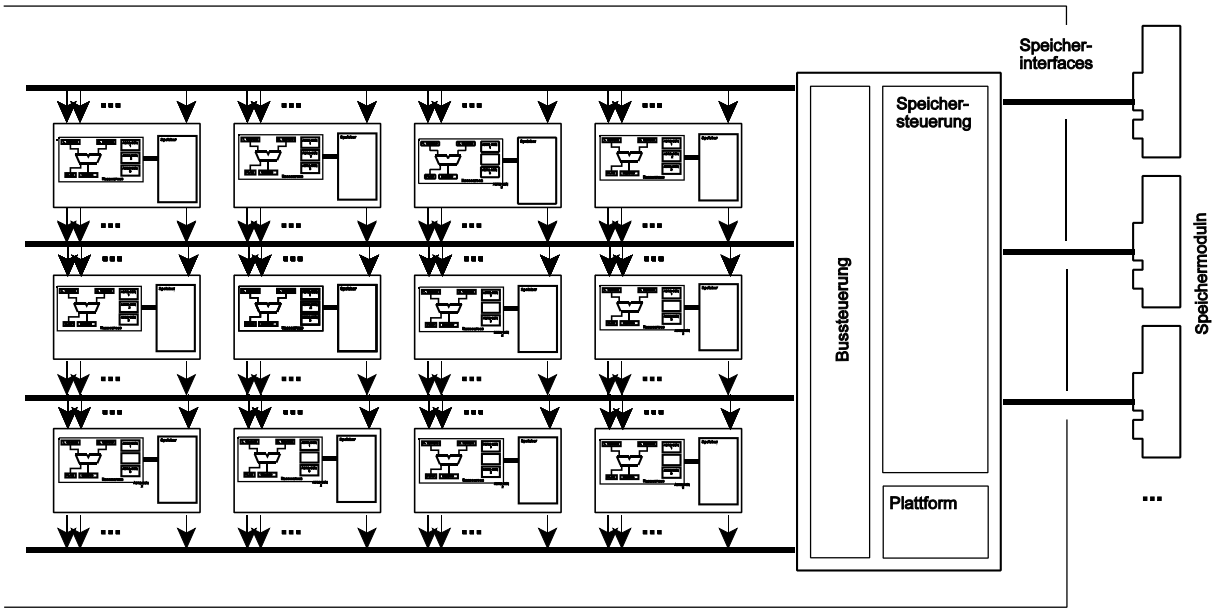


Abb. 4.3 FPGA-Schaltkreis mit Ressourcenzellen und externem Speicher

Oft braucht man sehr große Speicherkapazitäten (Mega...Gigabytes). Es liegt dann nahe, auf das Angebot des Massenmarktes zurückzugreifen und (außerhalb des FPGA-Schaltkreises) kostengünstige Speicherschaltkreise bzw. Speichermoduln einzusetzen. Die erforderliche Speichersteuerung ist hart aufgebaut.

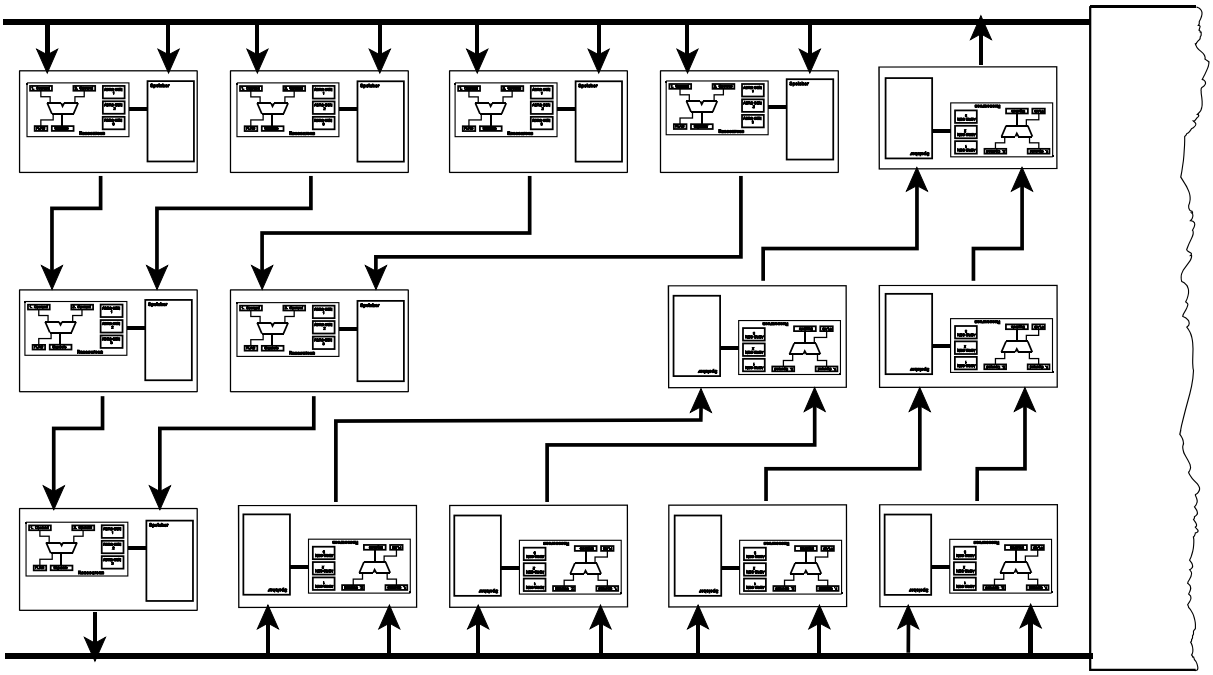


Abb. 4.4 FPGA-Schaltkreis mit Ressourcenzellen, die als invertierte Binärbäume verschaltet sind

Die Auswerung geschachtelter (geklammerter) Ausdrücke läßt sich gut auf invertierte Baumstrukturen abbilden (Abb. 4.5). Der Vorteil: die Verbindungen zwischen den Zellen der Baumstruktur sind einfache Punkt-zu-Punkt-Verbindungen. Sie sind kurz (also schnell durchlaufen) und brauchen keine Programmiervorkehrungen auf dem Schaltkreis. Im Beispiel sind zwei Baumstrukturen gegenläufig angeordnet, um die Schaltkreisfläche gut auszunutzen. Zum Einspeisen der Operanden und zum Abholen der Ergebnisse sind zwei Bussysteme vorgesehen. Infolge der gegenläufigen Anordnung wird jedes der Bussysteme in gleicher Weise zu Schreib- und Lesezugriffen ausgenutzt (gleichmäßige Lastverteilung).

Ein tiefer Binärbaum kann nicht immer sinnvoll ausgenutzt werden. Ist die Schachtelungstiefe (Nesting Depth) der Verarbeitungsvorgänge nicht allzu groß, ist eine Kombination zwischen Binärbaum und Stack von Vorteil. Die letzte Verarbeitungseinrichtung des Binärbaums bezieht einen ihrer Operanden aus dem Stack und legt auch ihre Ergebnisse im Stack ab. Um die Stackzugriffe zu beschleunigen, wird ein Stack Cache vorgesehen (Abb. 4.6). Dieser kann z. B. als herkömmliche Cache-Anordnung ausgebildet sein, die von einem Stackpointer auf bekannte Weise adressiert wird.

Durch gegenläufige Anordnung mehrerer solcher Binärbaumstrukturen kann die Schaltkreisfläche gut ausgenutzt werden. Beide Bussysteme werden gleichartig belastet. Die Freiflächen in Abb. 4.7 ergeben sich zunächst lediglich als Folge der unmaßstäblichen Darstellung. Sollten sie in der Praxis tatsächlich auftreten, können sie z. B. ausgenutzt werden, um die Stack Caches zu vergrößern.

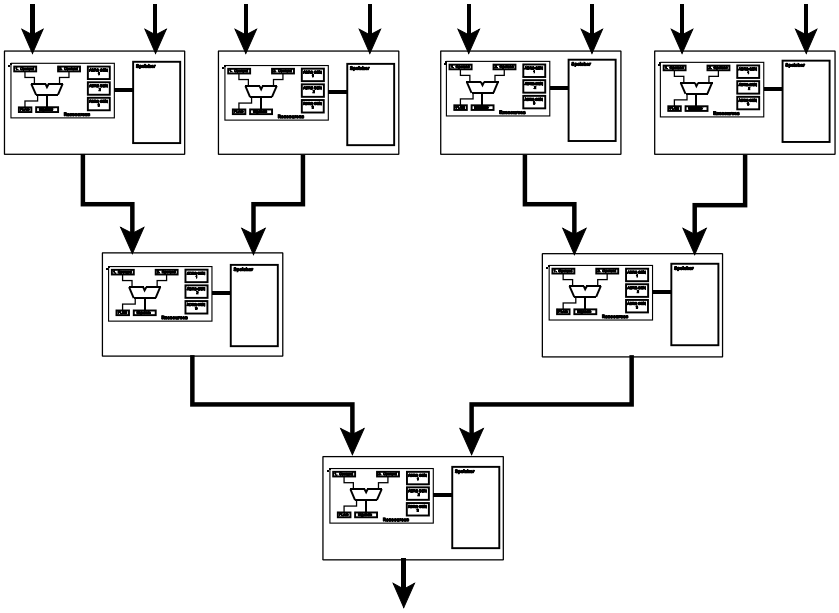


Abb. 4.5 Zu einem invertierten Binärbaum verschaltete Ressourcenzellen

Zur Nutzung derartiger Schaltkreise:

- Das Anwendungsproblem wird mit geeigneten Mitteln beschrieben (graphische Darstellungen, Zustandsdiagramme, Programmiersprachen),
- die Entwicklungsumgebung erzeugt daraus einen entsprechenden ReAl-Code (der Ressourcen anfordert, mit Parametern versorgt usw.),
- demgemäß wird ein Schaltkreis mit passender Ressourcenausstattung gewählt, und der ReAl-Code wird entsprechend abgewandelt.

Herkömmlicherweise muß der Entwickler entscheiden, welche Funktionen er mittels Software (auf einem eingebauten Prozessor) und welche er mittels Hardware (= programmierbarer Logik) löst. Hier entfällt diese Trennung. Vielmehr wird von der Entwicklungsumgebung sowie – zur Laufzeit – von der eingebauten Plattform eine jeweils zweckgerechte Ressourcen-Konfiguration erzeugt und bedarfsweise dynamisch geändert (Prozessoren entstehen bei Bedarf „on the fly“ und werden ggf. auch wieder abgebaut).

Umprogammierbare (weiche) Ressourcen
Universelle Ressourcen (z. B. ALU-Strukturen) sind für manche Anwendungsaufgaben nicht geeignet. Es liegt deshalb nahe, auf dem Schaltkreis gewisse Bereiche mit herkömmlichen programmierbaren Logikzellen vorzusehen, um bedarfsweise x-beliebige Verarbeitungs- und Steuerschaltungen aufbauen zu können (Abb. 4.9). Solche Anordnungen können aber nicht ohne weiteres mit den anderen (harten) Ressourcen zusammenwirken. Deshalb werden diese Zellenbereiche mit harten Schnittstellen umgeben, die denen der üblichen Ressourcen entsprechen (Parameterregister, Verkettungshardware, Bus-Interfaces usw.).

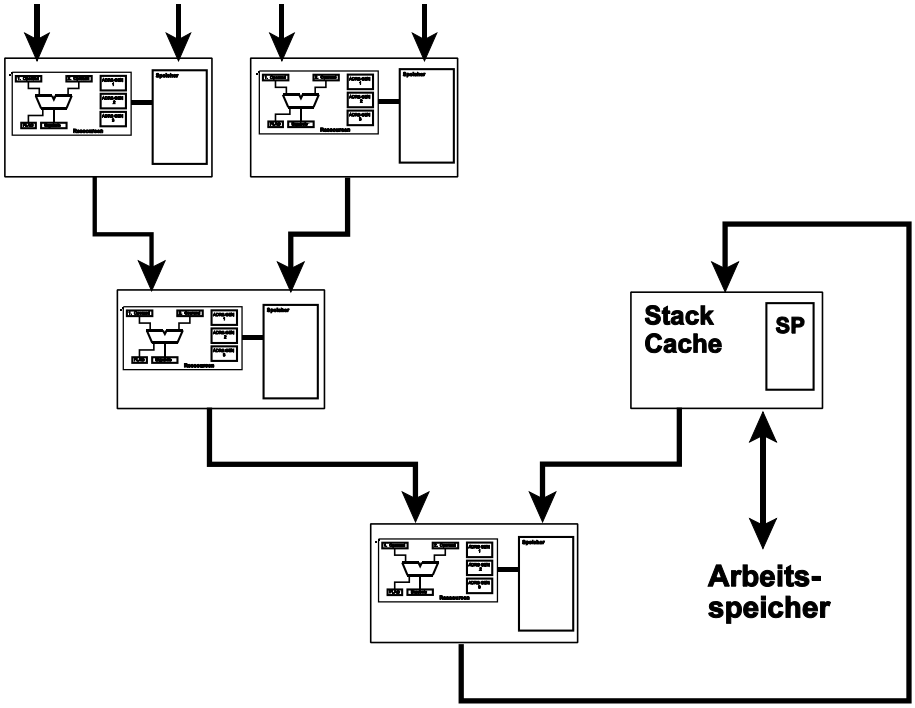


Abb. 4.6 Invertierter Binärbaum mit Stack Cache

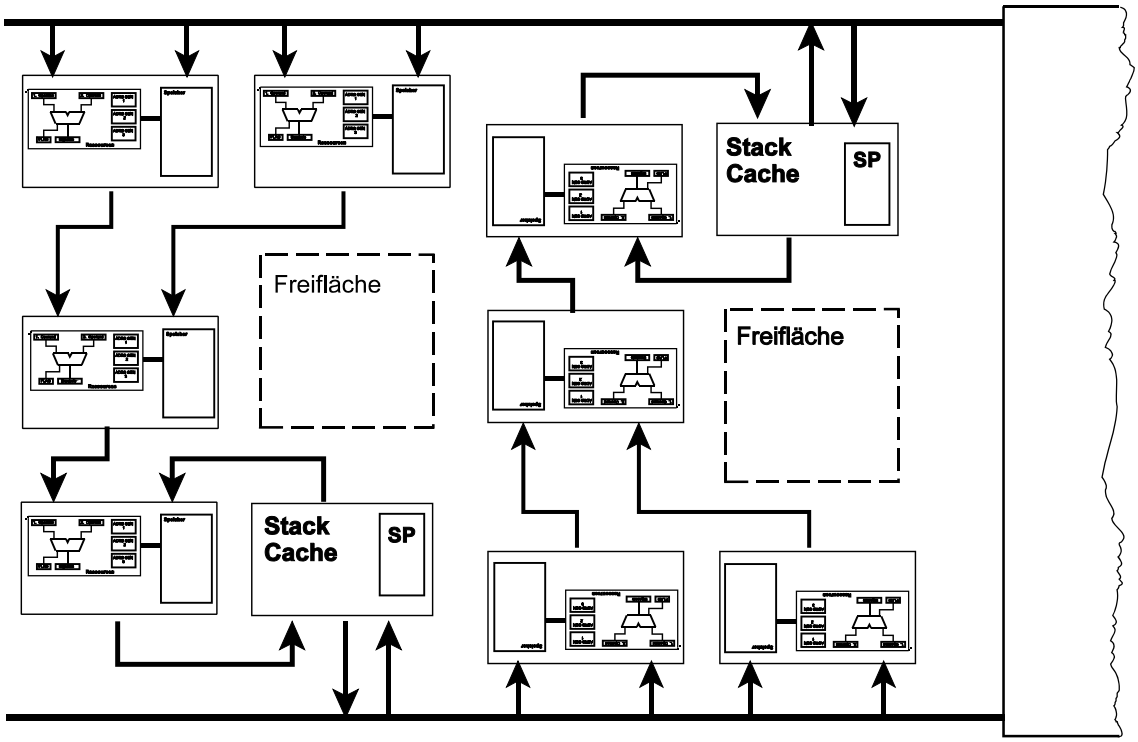


Abb. 4.7 FPGA-Schaltkreis mit Ressourcenzellen, die als invertierte Binärbäume im Verbund mit Stack Caches verschaltet sind

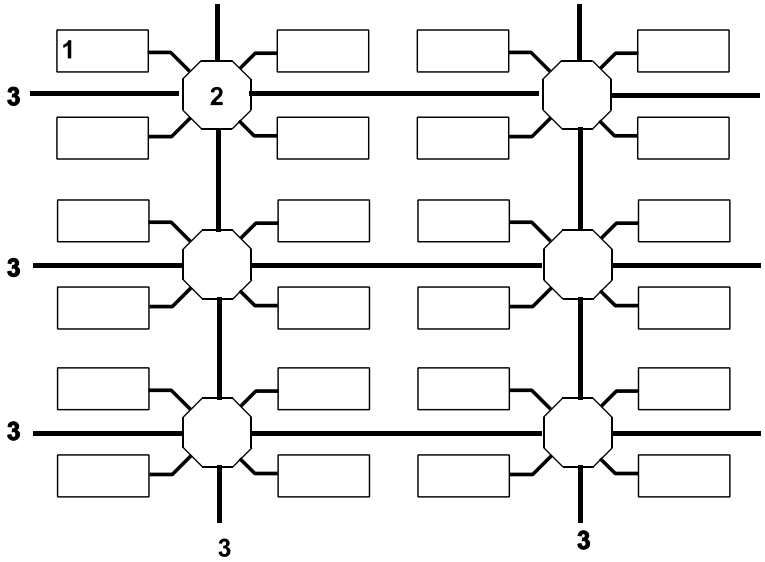
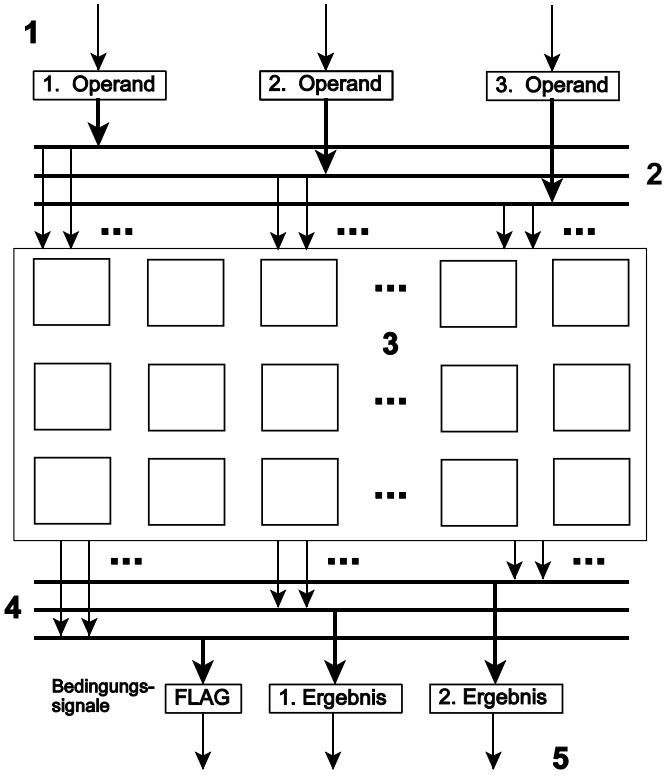


Abb. 4.8 FPGA-Schaltkreis mit Ressourcenzellen, die über Verteiler verbunden sind. Eine universelle Konfiguration auf Grundlage von Hochgeschwindigkeitsinterfaces. 1 - Ressourcenzellen; 2 - Verteiler; 3- programmierbare (globale) Signalwege



1 - eingangsseitige Ressourcenschnittstellen (hart), z. B. für drei Operanden zu 32 Bits; 2 - eingangsseitige Signalwege (programmierbar); 3 - programmierbare Logikzellen mit programmierbaren Verbindungen; 4 - ausgangsseitige Signalwege (programmierbar); 5 - ausgangsseitige Ressourcenschnittstellen (hart), z. B. für acht Bedingungssignale und zwei Ergebnisse zu 32 Bits.

Abb. 4.9 Eine programmierbare (weiche) Ressourcenzelle

Umprogrammierbare Schaltkreise

Es gibt FPGA-Schalteise, in denen die Programmierdaten der Logikzellen und Verbindungs in RAM-Strukturen gehalten werden. Sie werden herkömmlicherweise nach jedem Einschalten neu programmiert (durch Eintragen der RAM-Belegungen), und zwar vor Aufnahme des eigentlichen Betriebs. Während

des Betriebs (zur Laufzeit) ist ein Umprogrammieren nicht möglich. Mit harten Ressourcenzellen ausgerüstete Schaltkreise kann man hingegen so auslegen, daß sich auch zur Laufzeit umprogrammieren lassen (da die aus harten Zellen bestehende Struktur – z. B. gemäß Abb. 4.2 – bestehen bleibt, während die weichen Zellen umprogrammiert werden). Das Umprogrammieren kann durch s-Operatoren (Ressourcenbereitstellung), u-Operatoren (Ressourcenverwaltung) sowie c- und d-Operatoren (Verkettung) veranlaßt werden. Hierzu sind die Programmiersignalwege der betreffenden weichen Zellen an entspechende harte Strukturen anzuschließen, z. B. an eine auf dem Schaltkreis untergebrachte Plattform.

5. Plattformstrukturen

Die Plattform dient dazu, den jeweiligen ReAl-Maschinencode auszuführen. Der Maschinencode besteht aus Befehlen oder Anweisungen, die den ReAl-Operatoren direkt entsprechen oder die es ermöglichen, die Wirkungen der ReAl-Operatoren nachzubilden.

5.1 Einfache Plattformen

Sie können nur Befehle lesen (Abb. 5.1). Alle anderen Wirkungen sind von den Ressourcen aus zu erledigen (sind beispielsweise Daten aus einem Speicherbereich zu lesen, so müssen eigens entsprechende Ressourcen vorgesehen werden). Solche Plattformen sind nicht in der Lage, *alle* ReAl-Operatoren zu unterstützen.

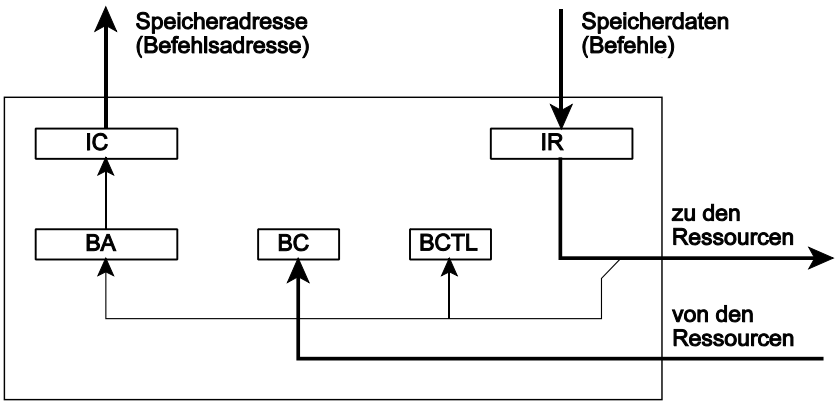


Abb. 5.1 Beispiel einer einfachen Plattform. IC - Befehlszähler; BA - Verzweigungsadresse; BC - aktuelle Bedingungsbits; BCTL - Verzweigungsbedingung; IR - Befehlsregister

Die Schaltmittel der Plattform müssen programmseitig angesprochen werden. Hierzu gibt es folgende Alternativen:

1. Über besondere Befehle oder Operatoren, der herkömmlichen Maschinenbefehlen entsprechen (Regiebefehle, u-Operatoren). Typische Funktionen: Laden von Adreßregistern, Verzweigen, Unterprogrammruf, Rückkehr, Unterbrechungssteuerung.
2. Die Einrichtungen der Plattform sind Ressourcen der Ordnungszahl 0. Sie werden wie alle anderen Ressourcen über ReAl-Operatoren (oder entsprechende Befehle bzw. Anweisungen) angesprochen.

Im folgenden wird vorzugsweise die zweite Alternative beschrieben.

In der Plattform gemäß Abb. 5.1 wirkt der Befehlszähler auf herkömmliche Weise, um aufeinanderfolgende Operatoren oder Anweisungen zu adressieren. Die Plattform hat zwei eigene Parameter, die mit p-Operatoren, l-Operatoren oder durch Verkettung geladen werden können: die Verzweigungsadresse BA und die Verzweigungsbedingung BCTL. Die Zugänglichkeit über l-Operatoren oder Verkettung ermöglicht es, mit berechneten Verzweigungsadressen und Bedingungen zu arbeiten. Die aktuelle Verzweigungsbedingung wird als dritter Parameter BC von der jeweiligen Ressource geliefert (über l-Operator oder Verkettung).

5.2 Bedingte Verzweigungen

Verzweigungsbedingungen sind besondere Ergebnisse, die von den entsprechenden Ressourcen erzeugt werden. Die einfachste Form entspricht den herkömmlichen Flagbits. Darüber hinaus sind beliebige Sonderfunktionen denkbar. Es könnte z. B. eine Addition geben, die bei Überlauf zwangsweise zu einem Behandler verzweigt.

Eine Verzweigungsressource liefert als Ausgang einen Befehlszählerinhalt. Verzweigen ist Anwendung des y-Operators auf die Verzweigungsressource.

Auch wäre prädikatgesteuerte y-Operatoren denkbar oder prädikatgesteuerte Ressourcen (y-Operator wirkt nur bei gesetztem Prädikat).

Ein typischer Verzweigungsablauf:

```
p (Adresse => BA, Bedingung => BCTL)          -- Einstellen von Adresse und Bedingung
....                                          -- weitere Operatoren
l (Ergebnis aus Verarbeitungsressource => BC) -- aktuelle Bedingung wird übergeben
```

Entspricht die aktuelle Bedingung der eingestellten, so wird die Verzweigung ausgelöst; die Verzweigungsadresse BA wird in den Befehlszähler IC transportiert.

Es ist Auslegungssache, wann die Verzweigung ausgelöst wird:

1. Sofort mit dem Eintreffen einer aktuellen Bedingung.
2. Mit einem entsprechenden y-Operator (vgl. das nachfolgende Beispiel).

Die erste Auslegung spart den y-Operator ein, erfordert aber eine strikt sequentielle Ausführungsreihenfolge (der jeweils nächste Operator darf erst dann wirksam werden, wenn der vorhergehende Operator vollständig ausgeführt wurde). Hierdurch wird gewährleistet daß die Verzweigung sofort nach dem Eintreffen der aktuellen Bedingung ausgelöst wird, ohne daß zwischenzeitlich andere Operatoren angearbeitet werden.

Die zweite Auslegung ermöglicht es, zwischen dem Eintreffen der Bedingung und dem Auslösen der Verzweigung weitere Operatoren auszuführen.

Ein ähnliches Prinzip ist in manchen der bekannten Hochleistungsprozessoren vorgesehen. Eine typische Auslegung besteht darin, nach dem Verzweigungsbefehl zunächst noch den unmittelbar nachfolgenden Befehl auszuführen und somit die Lücke in der Befehlspipeline, die sich infolge der Verzweigung zwangsläufig ergibt, noch mit nützlicher Arbeit zu füllen. Diese Art der Verzögerung (Delayed Branching) ist aber starr und beispielsweise auf den einen nachfolgenden Befehl beschränkt. Die Nutzung des y-Operators zum Auslösen der eigentlichen Verzweigung ermöglicht es hingegen, zwischen der Entscheidung über die Verzweigungsrichtung und der tatsächlichen Verzweigung beliebig viele weitere Befehle auszuführen.

Beispiel einer einfachen bedingten Verzweigung (Abb. 5.2):

Programmierabsicht:

```
C := A + B
if CARRY_OUT then goto OUT_OF_RANGE
```

Die Ressourcen:

- ADD: ein übliches Addierwerk (bzw. eine ALU). Liefert aus den Parametern 1, 2 die Summe 3 sowie die Flagbitbelegung 4.

- PGM_CTR: der Befehlszähler. Parameter: 1 - Verzweigungsadresse; 2 - Verzweigungsbedingung; 3 - auszuwertende Bedingungsbits; 4 - Ergebnis = Adresse der nächsten Anweisung. COND CTL - Bedingungssteuerung. Normalfall: Adreßerhöhung über Zählnetzwerk CT. Verzweigungsfall: Übernehmen der Verzweigungsadresse 1.

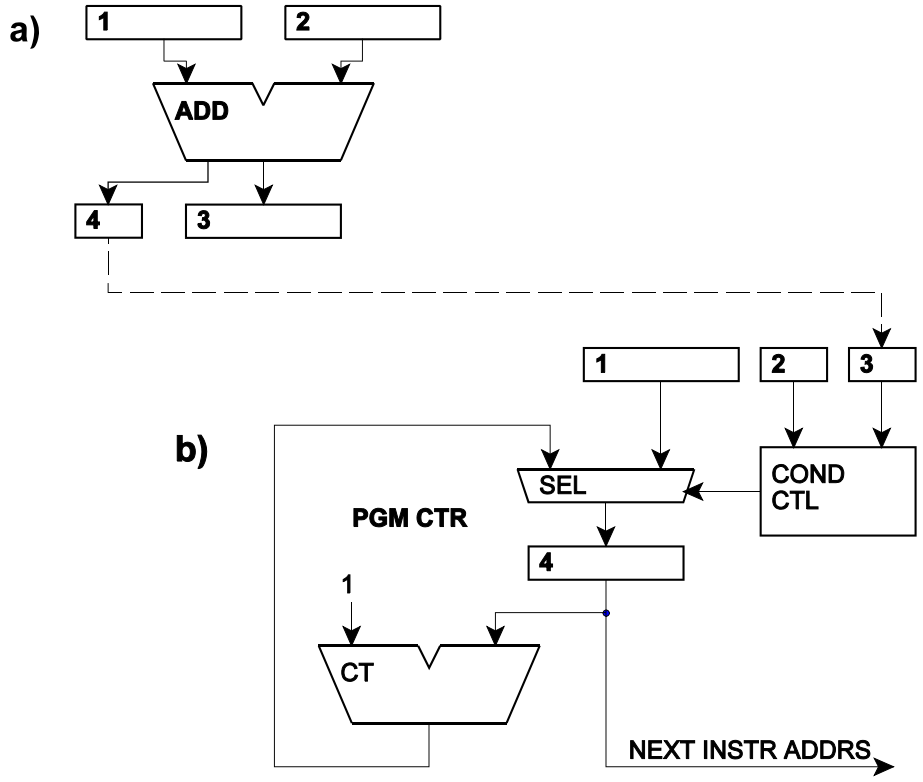


Abb. 5.2 Ressourcen, die zur Ausführung des Programmbeispiels erforderlich sind. a) Verarbeitungsressource; b) Aktivierungsressource mit Verzweigungsfunktion

```
s (ADD, PGM_CTR)
p (A => 1.1, B => 1.2, OUT_OF_RANGE => 2.1, CARRY_OUT => 2.2)
y (1)                          -- addieren
a (1.3 => C)                   -- Ergebnis zuweisen
l (1.4 => 2.3)                 -- Flagbits zum Befehlszähler
y (2)                          -- verzweigen
```

Alternative (Verkettung):

```
s (ADD, PGM_CTR)
p (A => 1.1, B => 1.2, OUT_OF_RANGE => 2.1, CARRY_OUT => 2.2)
c (1.4 => 2.3)                 -- Flagbits zu den Bedingungsbits
y (1)
a (1.3 => C)                   -- Ergebnis zuweisen
y(2)                           -- Verzweigen (darf erst ausgelöst werden, nachdem Ergebnis
                               -- zugewiesen wurde)
```

Beschleunigung der Verzweigungsabläufe

Verzweigungen bereiten besondere Schwierigkeiten in Einrichtungen, die mit Ablaufüberlappung (Befehlspipelining) arbeiten, da eine Verzweigung den Befehlsfluß in der Pipeline unterbricht und einen Neuanlauf erforderlich macht. Um den hierdurch bewirkten Geschwindigkeitsverlust abzumildern, wird herkömmlicherweise viel Aufwand getrieben. Es werden zwei Prinzipien im Verbund angewendet:

- Sprungvorhersage. Stößt die Befehlspipeline auf einen Verzweigungsbefehl, so führt sie das Befehlslesen zunächst in die Richtung weiter, die als wahrscheinlichste angesehen wird. Ein Neuanlauf der Pipeline ist nur dann notwendig, wenn diese Voraussage nicht eingetreten ist.
- Sprungzielpufferung. Die als Sprungziele adressierten Befehle werden in einem speziellen Pufferspeicher (Branch Target Cache) abgelegt, so daß sie ggf. sofort (ohne erneuten Speicherzugriff) erreichbar sind und auf schnellstem Wege in die Befehlspipeline einfließen können.

Es handelt sich herkömmlicherweise um ein autonom gesteuertes Raten und Probieren, wobei Konflikte erkannt und ggf. aufgelöst werden müssen. Die Steuer- und Überwachungsschaltungen sind entsprechend aufwendig. In einer ReAl-Maschine hingegen kann die Verzweigungsvorbereitung und Pufferung durchgängig programmseitig gesteuert werden; komplizierte Steuerschaltungen sind nicht erforderlich (die entsprechende Schaltkreisfläche könnte z. B. für größere Puffer verwendet werden).

In der Anordnung von Abb. 5.1 kann die Verzweigungsadresse früh genug (vor der eigentlichen Verzweigung) bereitgestellt werden, so daß es möglich wird, die Befehle, zu denen verzweigt werden soll (Verzweigungsziele), rechtzeitig heranzuschaffen (Abb. 5.3).

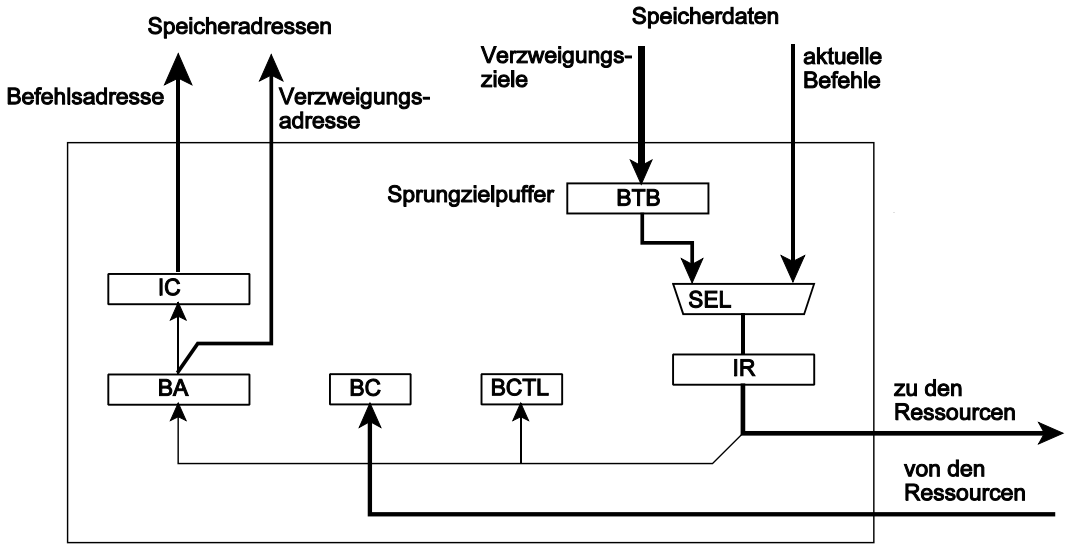


Abb. 5.3 Einfache Plattform mit Sprungzielpufferung

Mit der Verzweigungsadressse wird zum Speicher zugegriffen, um den betreffenden Befehl (das Verzweigungsziel) vorbeugend in ein Pufferregister BTB zu laden. Wird die Verzweigung wirksam, so wird der Inhalt des Pufferregisters BTB ins Befehlsregister IR übernommen.

Das Prinzip kann auf mehrere Verzweigungsziele erweitert werden (Abb. 5.4).

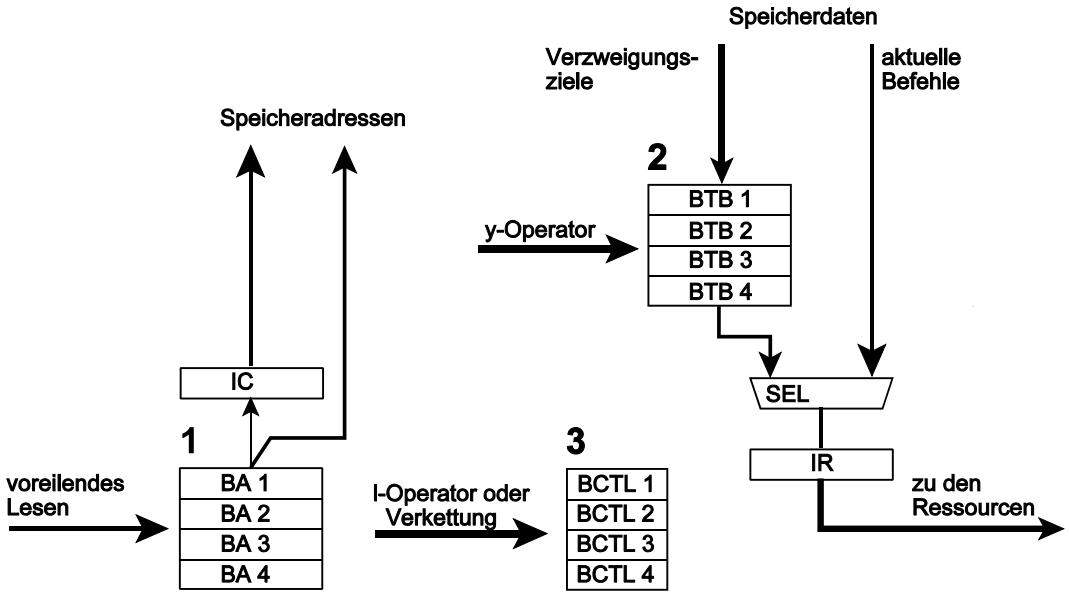


Abb. 5.4 Einfache Plattform mit erweiterter Sprungzielpufferung (es sind lediglich die wesentlichen Abwandlungen gegenüber Abb. 5.3 dargestellt). 1 - Verzweigungsadreßpuffer; 2 - Verzweigungszielpuffer; 3 - Verzweigungsbedingungspuffer.

Die Verzweigungsadressen werden in den Verzweigungsadreßpuffer 1 geladen, die zugehörigen Verzweigungsbedingungen in den Verzweigungsbedingungspuffer 3 (z. B. mittels p-Operator). Gemäß den Adressen im Verzweigungsadreßpuffer 1 werden die jeweiligen ersten Befehle der Sprungziele aus dem Speicher geholt und den Verzweigungszielpuffer 2 gebracht. Das kann von der Plattform automatisch ausgelöst werden, und zwar vorzugsweise dann, wenn keine anderen Zugriffe auszuführen sind. Dieses voreilende Holen von Befehlen entspricht dem Stand der Technik. Es ist in marktgängigen Hochleistungsprozessoren vorgesehen und muß deshalb nicht näher beschrieben werden. Eine Vereinfachung ergibt sich, wenn die Befehle, die als Verzweigungsziele in Frage kommen, direkt – also programmgesteuert – in den Verzweigungszielpuffer 2 eingetragen werden, z. B. mit p-Operatoren.

Von den Verarbeitungsressourcen signalisierte Bedingungen (l-Operatoren oder Verkettung) betreffen den Verzweigungsbedingungspuffer 3. Ist eine Bedingung erfüllt, so wird der zugehörige Eintrag im Verzweigungszielpuffer 2 aktiviert. Je nach Auslegung wird der besagte Eintrag entweder sofort oder nach Auslösen eines entsprechenden y-Operators über die Auswahlschaltung in das Befehlsregister IR transportiert. Sinngemäß gelangt die zugehörige Befehlsadresse aus dem Verzweigungsadreßzähler 1 in den Befehlszähler IC.

Unterstützung von Mehrfachverzweigungen
In einer Weiterbildung der Anordnung von Abb. 5.4 können bestimmte Bedingungssignale der Verarbeitungsressourcen verwendet werden, um Einträge im Verzweigungsadreßpuffer 1 und im Verzweigungszielpuffer 2 direkt zu adressieren. Hierdurch kann ein Verzweigen zu einem von mehreren Verzweigungszielen veranlaßt werden. Eine derart ausgelegte Plattform kann folgende Verzweigungsarten unterstützen:

1. Herkömmliche Verzweigung. Für jede derartige Verzweigung ist ein Eintrag in den Pufferspeichern von Abb. 5.4 erforderlich. Die Verzweigungsbedingung wird im Verzweigungsbedingungspuffer 3 eingestellt. Die jeweilige Verarbeitungsressource schickt ihre Bedingung zum betreffenden Eintrag. Entspricht die gesendete Bedingung der eingestellten, so wird die betreffende Verzweigung aktiv. Die Abb. 5.5 und 5.6 veranschaulichen, wieviele Verzweigungen erforderlich sind, um übliche Programmkonstrukte zu unterstützen.
2. Mehrere Einträge werden für ein Verzweigen in mehrere Richtungen benutzt. So können sich beispielsweise bei einem arithmetischen Vergleich von Zahlen folgende Bedingungen ergeben: $<$, $\leq$, $=$ $\geq$, $>$. Demgemäß können fünf Einträge belegt werden, um eine Mehrfachverzweigung gemäß dem jeweiligen Vergleichsergebnis zu unterstützen. Hierzu muß im Verzweigungbedingungspuffer 3 eine besondere Betriebsart eingestellt werden.

3. Mehrere Einträge werden für ein Verzweigen in mehrere Richtungen benutzt. Dabei werden die von den Verarbeitungsressourcen kommenden Bedingungssignale direkt als Auswahladresse verwendet.

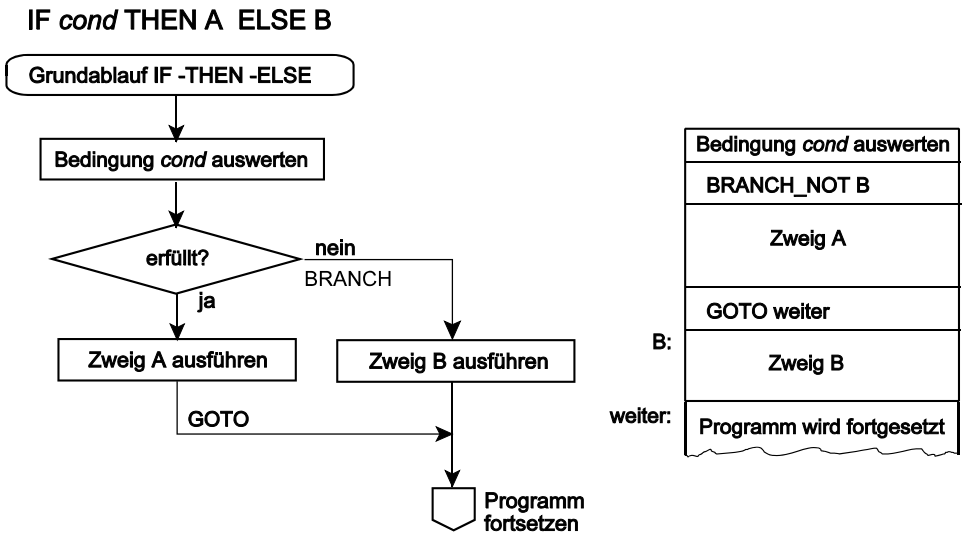


Abb. 5.5 Elementare Fallunterscheidung IF - THEN - ELSE

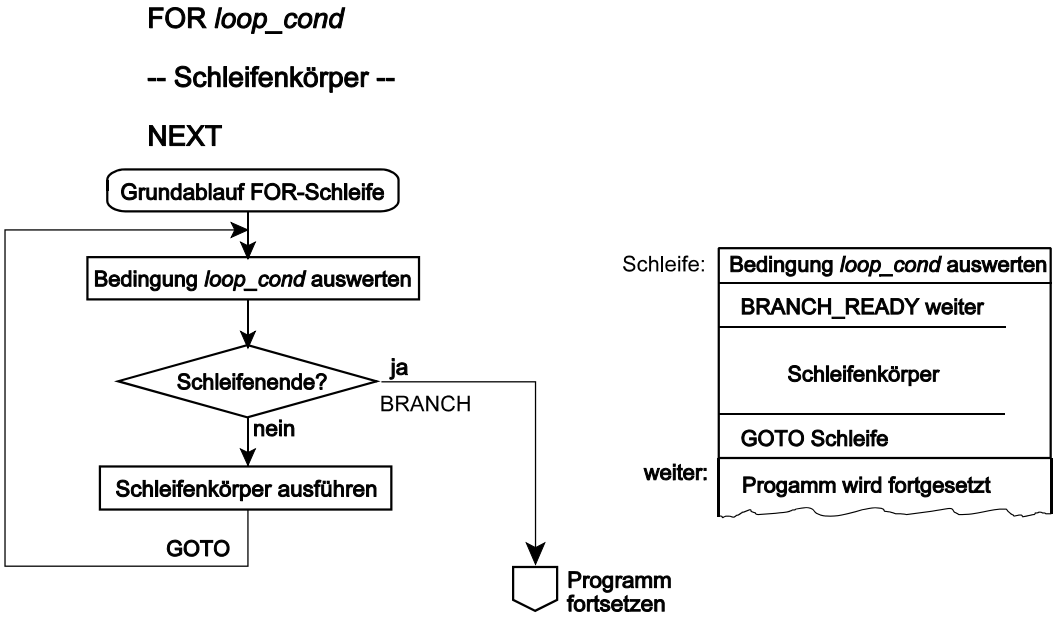


Abb. 5.6 Programmschleife am Beispiel der FOR-Schleife

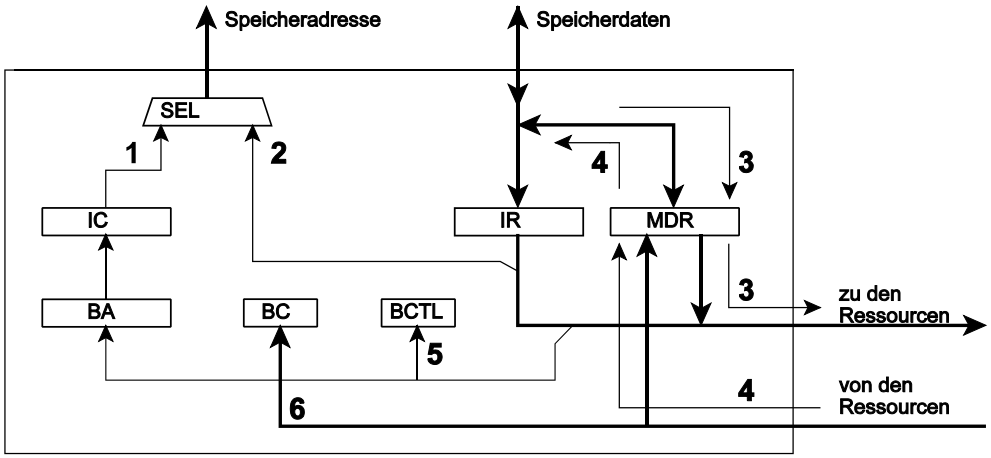
Sowohl die elementare Fallunterscheidung als auch die Programmschleife kommt mit jeweils zwei Verzweigungen aus, einer bedingten (BRANCH) und einer unbedingten (GOTO). Demgemäß sollten Puffereinrichtungen für wenigstens zwei Verzweigungen vorgesehen werden. Dann ist es möglich, Fallunterscheidungen und Schleifen komplett so vorzubereiten, daß bei jedem Durchlauf die Verzweigungen nur noch ausgelöst werden müssen (y-Operatoren oder Verkettung).

Hinweis:
In den folgenden Beschreibungen höherentwickelter Plattformstrukturen wird – aus Gründen der Überschaubarkeit – darauf verzichtet, Vorkehrungen zum Beschleunigen von Verzweigungen darzustellen.

5.3 Höherentwickelte Plattformen

Beim Erweitern der Plattform geht es im Grunde stets um einen Kompromiß zwischen Plattform-Ausstattung und Ressourcen-Ausstattung. So wird man beispielsweise die Plattform auf einem FPGA-Schaltkreis nur so aufwendig auslegen, wie dies für den jeweiligen Anwendungsfall erforderlich ist.

Um alle ReAl-Operatoren zu unterstützen, muß es möglich sein, Parameter im Speicher zu adressieren (Abb. 5.7) und Unterprogramme aufzurufen (Abb. 5.8).



IC - Befehlszähler; BA - Verzweigungsadresse; BC - aktuelle Bedingungsbits; BCTL - Verzweigungsbedingung; IR - Befehlsregister, MDR - Speicherdatenregister; 1 - Befehlsadressierung; 2 - Parameteradressierung; 3 - Datenfluß für p-Operator; 4 - Datenfluß für a-Operator; 5 - die Angaben zur Verzweigungssteuerung werden mit p-Operatoren eingetragen; 6 - auf diesem Wege gelangen die aktuellen Verzweigungbedingungen (z. B. FLAG-Bits) in die Plattform (Transport typischerweise mit l-Operatoren). Die Parameteradressen sind hier Direktwerte im Befehl (Absolutadressierung).

Abb. 5.7 Einfache Plattform, die auch Parameterzugriffe unterstützt

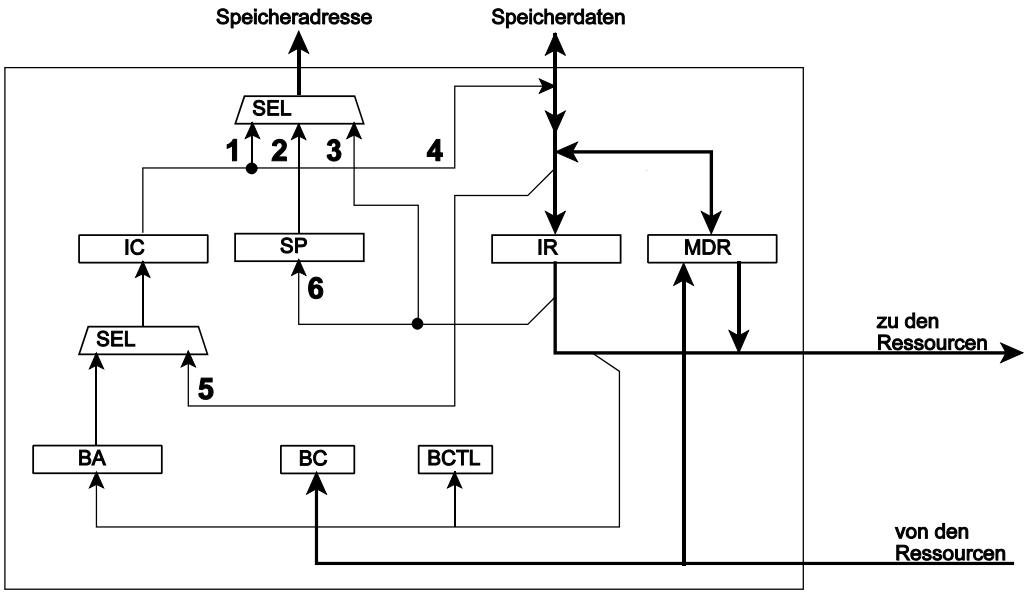
In vielen Anwendungsfällen ist es von Vorteil, wenn die Plattform jene Adressierungsweisen unterstützt, die für die Laufzeitsysteme der gängigen Programmiersprachen typisch sind (Abb.n 5.9, 5.10):

- Relativadressierung nach dem Prinzip „Basis + Displacement“,
- Verfügbarkeit mehrerer Basisregister,
- für Datenzugriffe zugänglicher Stack,
- Stack-Oganisation auf Grundlage von Stack Frames.

Die Nutzung des Stackpointers SP und des Base Pointers BP entspricht den üblichen Gepflogenheiten. Der Global Pointer GP kann verwendet werden, um beliebige andere Basisadressen zu halten (für globale Variable, für Zugriffe zum Heap usw.).

Parameteradressierung:
Zum Inhalt eines Basisregisters wird eine Displacementangabe aus dem Befehlsregister addiert. Solche Angaben sind z. B. in p- und a-Operatoren enthalten.

Stackadressierung über SP:
Der Stack Pointer SP wirkt zum einen als Basisregister. Zum anderen adressiert er den Speicher direkt, wobei sein Inhalt je nach Operation vermindet oder erhöht wird (Unterprogrammruf, Rettung, PUSH-Anweisungen, POP-Anweisungen).



IC - Befehlszähler; BA - Verzweigungsadresse; BC - Verzweigungsbedingung; BCTL - Verzweigungssteuerung; IR - Befehlsregister, MDR - Speicherdatenregister; SP - Stackpointer. 1 - Befehlsadressierung; 2 - Stackadressierung; 3 - Parameteradressierung; 4 - Datenfluß bei Unterprogrammruf; 5 - Datenfluß bei Rückkehr; 6 - Laden des Stackpointers. Es handelt sich um den allgemein bekannten Stackmechanismus zur Rettung der Rückkehradresse beim Unterprogrammruf.

Abb. 5.8 Einfache Plattform, die die Absolutadressierung von Parametern und den Unterprogrammruf unterstützt

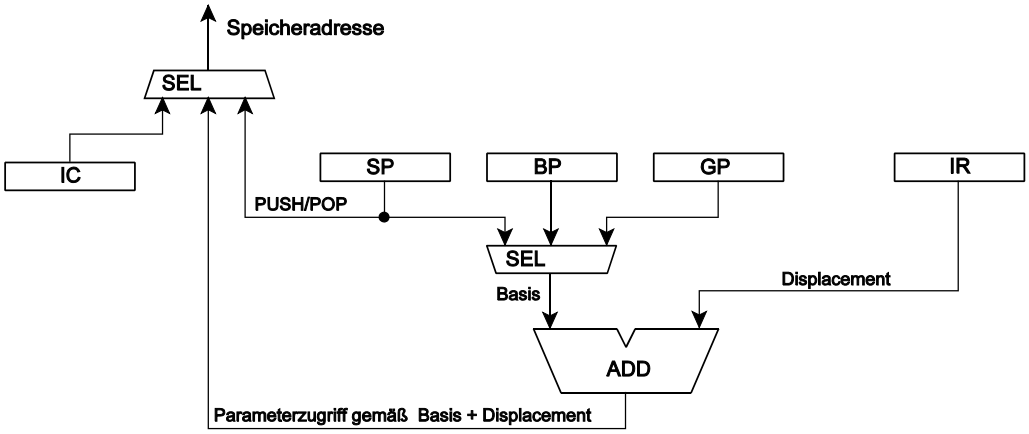


Abb. 5.9 Eine elementare Ausstattung an Basisregistern. IC - Befehlszähler; SP - Stackpointer; BP - Base Pointer; GP - Global Pointer; IR - Befehlsregister

Die Anordnung gemäß Abb. 5.10 unterstützt die typischen Abläufe des Unterprogrammablaufs und der Rückkehr. Der eigentliche Unterprogrammruf ist eine Verzweigung (Verzweigungsadresse in Register BA), dem ein Retten der in IA gehaltenen Folgeadresse (= Rückkehradresse) vorausgeht. Beim anschließenden ENTER-Ablauf (Eintritt ins Unterprogramm) wird Inhalt des Base Pointer BP auf den Stack gelegt (PUSH BP). Anschließend wird der aktuelle Stackpointer zum neuen Base Pointer (SP => BP). Beim Leave-Ablauf (Verlassen des Unterprogramms) wird der Stackpointer mit der Belegung des Base Pointer überladen (BP => SP). Anschließend wird der alte Base Pointer aus dem Stack geholt (POP BP). Der eigentliche Rückkehrbefehl lädt das Befehlsadreßregister IA mit dem Rückkehradresse aus dem Stack (POP IA). Die diversen Adreßregister (BA, SP, BP, GP) können aus dem Speicher, von den Verarbeitungsressourcen oder mit einem Direktwert aus dem Befehlsregister geladen werden.

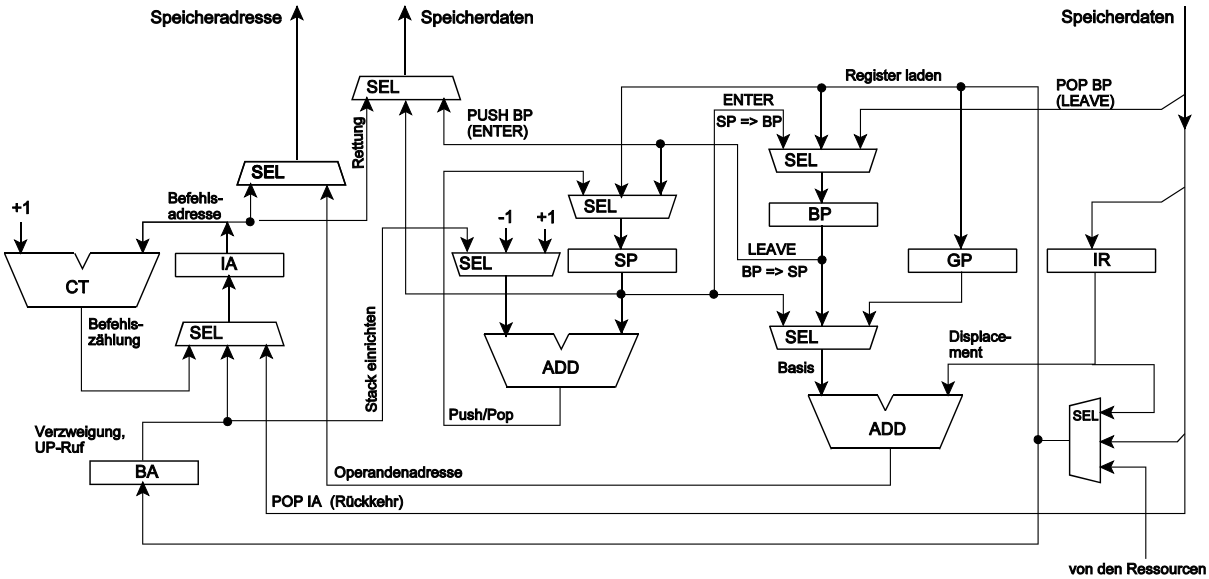


Abb. 5.10 Die Adressierungseinrichtungen einer mit Basisregistern ausgestatteten Plattform. IA - Befehlsadresse; BA - Verzweigungsadresse; SP - Stackpointer; BP - Base Pointer; GP - Global Pointer; IR - Befehlsregister

5.4 Nachbildung der der ReAl-Wirkprinzipien mittels Software (Emulation)

Alternativ zum Aufbau einer speziellen Hardware ist es möglich, einen Universalrechner als Plattform zu verwenden und die ReAl-Wirkprinzipien mit Software zu emulieren (Abb. 5.11, 5.12).

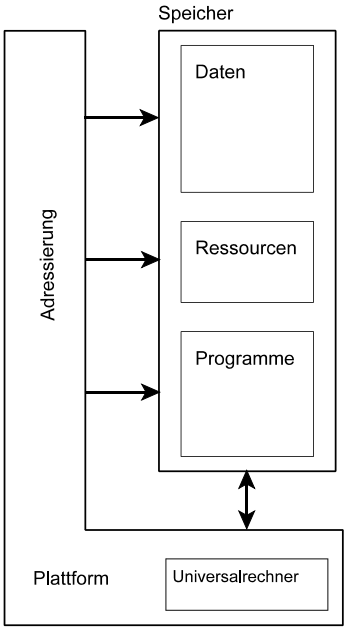


Abb. 5.11 Plattform zur softwareseitigen Emulation

Neben den Programmen und Daten sind auch die Ressourcen im Speicher unterzubringen. Eine Ressource besteht aus

- Speicherplätzen, die den Registern der Hardwarelösung entsprechen,
- zusätzlichen Speicherplätzen für Zwischenwerte usw. (Arbeitsbereiche),
- einem Programm, das die Funktion der Ressource nachbildet.

Typischerweise genügt jeweils ein einziges Programm, um alle gleichartigen Ressourcen zu emulieren. Ausnahme: die Plattform unterstützt die erforderliche Adreßrechnung nur unzulänglich (wie dies bei manchen Mikrocontrollern der Fall ist). Dann sind mehrere Kopien des jeweiligen Programms erforderlich, die mit unterschiedlichen Adressen arbeiten.

Beispiel einer Funktion, die auf diese Weise implementiert wird:

```
int EXAMPLE (int A, int B, double C);

{
int X, Y;
double Z;
float H, I;
...

....

return (Y);

}
```

Jetzt wird die Funktion aufgerufen:

```
OMEGA = EXAMPLE (ALPHA, BETA, GAMMA);
```

Die Ressource EXAMPLE wird als Speicherbereich definiert (Abb. 5.12).

Zeiger auf Emulationsprogramm
Steuer. und Zustandsangaben
ggf. Verkettungsangaben
ggf. interner SP
lokaler Arbeitsbereich (ggf als Stack ausgeführt)
Rückgabewert EXAMPLE
I
H
Z (low)
Z (high)
Y
X
C (low)
C (high)
B
A
ggf. Leistungsmeßangaben, Nutzungszähler usw. (Instrumentierung)

Abb. 5.12 Definition einer Ressource als Speicherbereich (Beispiel)

Bei jedem Funktionsaufruf wird ein solcher Bereich erzeugt. Jedem Funktionsaufruf im Quellprogramm entspricht ein s-Operator. Die gesamte Aufrufkette (Funktion A ruft Funktion B, diese ruft Funktion C usw.) wird zunächst – mit s-Operatoren – gleichsam vorgefertigt. Das Ausführen einer solchen Funktion wird mit einem y-Operator gestartet.

Prinzip:

- Mit s-Operatoren werden alle Funktions-Ressourcen aufgebaut,
- soll eine Funktion ausgeführt werden, wird die betreffende Ressource zunächst mit Parametern versorgt (p-Operatoren); dann wird die Programmausführung gestartet (y-Operator).
- schließlich werden die Ergebnisse zugewiesen (a-Operatoren, l-Operatoren oder Verkettung).

Alle Transporte sind Transporte auf feste Adressen; es entfällt das Auf- und Abbauen der Stack Frames. Die Bereiche können vielmehr während der gesamten Laufzeit bestehen bleiben. Somit werden praktisch alle lokalen Variablen zu globalen.

Daß das Auf- und Abbauen der Stack Frames entfällt, ist vor allem dann von Vorteil, wenn Funktionen im Innern von Programmschleifen gerufen werden.

Zudem ist es möglich, die Ausführung der einzelnen Funktionen unterschiedlichen Prozessoren zu übertragen. Die Speicherbereiche besonders leistungskritischer Funktionen können in Registerspeichern (Universalregistersätzen) untergebracht werden. Auf dieseWeise können auch sehr große Registerspeicher wirkungsvoll ausgenutzt werden (z. B. mit 256 und mehr Universalregistern). Des weiteren ist es möglich, den Universalregistern Verarbeitungsressourcen bereichsweise direkt zuzuordnen (vgl. die Abb.n 3.8 bis 3.10 nebst Erläuterungen). Derartige Anordnungen können – infolge der unmittelbaren (also kurzen) Verbindungen zwischen Registern und Verarbeitungsschaltungen – mit höheren Taktfrequenzen betrieben oder als Pipelinestrukturen mit geringerer Tiefe ausgeführt werden.

Hinweis:
Die Nutzung von Universalregistersätzen ist ein seit langem bekanntes Problem der maschinennahen Programmierung und der Programmgenerierung (in Compilern). Hierzu hat man Algorithmen entwikkelt, die herausfinden, welche Variablen am häufigsten benötigt werden und deshalb vorrangig in Registern unterzubringen sind (Register Allocation). Manche Programmiersprachen unterstützen die explizite Deklaration von Registervariablen (so daß der Anwendungsprogrammierer die Zuweisung beeinflussen kann). Sehr große Registersätze (es gibt Prozessoren mit beispielsweise 128 oder 192 Universalregistern) werden herkömmlicherweise über eine Bereichsadressierung zugänglich gemacht. Der einzelne Maschinenbefehl sieht jeweils nur einen Ausschnitt aus dem gesamten Registeradreßraum. In diesem Bereich wird der jeweils aktuelle Stack Frame untergebracht. Werden Unterprogramme (Funktionen) aufgerufen oder verlassen, so werden diese Bereiche entsprechend umgeschaltet. Dieses Prinzip ersetzt zwar das Transportieren von Parametern durch das – deutlich schnellere – Umschalten des zugänglichen Adreßbereichs, die Umschaltvorgänge kosten aber ihrerseits Zeit, und die Adressierung erfordert zusätzliche Schaltmittel, die durchlaufen werden müssen. Hierdurch ergibt sich ein längerer Taktzyklus oder die Notwendigkeit, zusätzliche Pipeline-Stufen einzuführen. Zudem hat man den Auf- und Abbau der Stack Frames zwar beschleunigt, aber nicht gänzlich abgeschafft. Nach wie vor müssen die lokalen Variablen beim Aufrufen der Funktion neu initialisiert werden und gehen beim Verlassen der Funktion verloren.

Merkmale moderner Hochleistungsprozessoren, die zur Geschwindigkeitssteigerung ausgenutzt werden können:

- mehrere Verarbeitungswerke (Superskalarprinzip),
- große Universalregistersätze,
- Multiprozessorsysteme (mehr als ein Prozessor auf einem Schaltkreis, mehrere Prozessorschaltkreise im Verbund),
- Parallelausführung mehrerer gleichartiger Operationen (SMID = Single Instruction, Multiple Data),
- Parallelausführung mehrerer verschiedenartiger Operationen, gesteuert von entsprechend langen Befehlen (VLIW = Very Long Instruction Word).

Die Maßnahmen zur Geschwindigkeitssteigerung laufen im Grunde darauf hinaus, daß im ReAl-Programm nach unabhängigen Verarbeitungsabläufen gesucht wird.

Wann sind Ressourcen voneinander unabhängig?
Zwei Ressourcen R1, R2 sind dann voneinander unabhängig, wenn im ReAl-Programm folgendes *nicht* vorkommt:

- Verkettungen zwischen R1 und R2 (in beiden Richtungen),
- l-Operatoren, die R1 und R2 betreffen (für beide Richtungen),
- Weiterverarbeitung gespeicherter Ergebnisse der jeweils anderen Ressource (p-Operatoren mit R2 beziehen sich auf Variable, die zuvor von a-Operatoren mit R1 im Speicher abgelegt wurden und umgekehrt).

Die Unabhängigkeit ist im allgemeinen eine dynamische, vom Gang der Verarbeitung abhängige Eigenschaft (beispielsweise können zwei Ressourcen eine Zeit lang unabhängig voneinander sein und später verkettet werden).

Ausnutzung des Superskalarprinzips
Ressourcen, die im jeweiligen Zeitabschnitt unabhängig voneinander sind, werden gleichzeitig aktiviert (y-Operator).

Ausnutzung von Multiprozessor-Konfigurationen
Unabhängige Ressourcen werden verschiedenen Prozessoren zugeordnet.

Ausnutzung von SIMD-Vorkehrungen
Im ReAl-Programm wird nach unabhängigen gleichartigen Ressourcen gesucht, deren Wirkungsweise mit SIMD-Befehlen nachgebildet werden kann (gleiche Operation, gleiche Verarbeitungsbreite). Derartige Operationen werden gleichsam aufgesammelt und – wenn genügend anhängig sind – gemeinsam aktiviert (y-Operator löst SMID-Befehle aus).

Ausnutzung von VLIW-Vorkehrungen
Im ReAl-Programm wird nach unabhängigen Ressourcen gesucht, deren Wirkungsweise mit VLIW-Befehlen nachgebildet werden kann. Derartige Operationen werden gleichsam aufgesammelt und – wenn genügend anhängig sind – gemeinsam aktiviert (y-Operator löst VLIW-Befehle aus).

6. Ressourcenverwaltung

6.1 Ressourcenverwaltung über Tabellenstrukturen

Um die Ressourcen verwalten zu können, braucht man Übersichten über

- Die im System verfügbaren Ressourcentypen,
- die Eigenschaften der einzelnen Ressourcentypen,
- die Zustände der vorhandenen Ressourcen (wieviele je Typ vorhanden sind, wieviele davon noch verfügbar sind usw.),
- die dem einzelnen laufenden Programm (Task, Prozeß o. dergl.) zugeordneten Ressourcen.

Es liegt nahe, diese Angaben in Tabellenstrukturen unterzubringen. Ähnlich gelagerte Aufgaben gibt es beim Verwalten eines virtuellen Speichers, in Dateisystemen, beim Compilieren von Programmen usw. Der Aufbau von Tabellenstrukturen und die entsprechenden Zugriffsverfahren gehören zum allgemeinen Fachwissen der Informatik. Deshalb genügt eine knappe Beschreibung anhand eines Beispiels.

Tabelleneinträge könne auf verschiedene Art erreicht werden:

- über einen Namen (Zeichenkette),
- über eine laufende Nummer (Ordinalzahl),
- über eine Adreßangabe.

Die folgende Beschreibung bezieht sich auf den Zugang über die laufenden Nummern (Ordinalzahlen) der Einträge. Diese Art des Zugangs ist nur geringfügig langsamer als die direkte Adressierung (es genügt eine einfache Umrechung, um aus der Ordinalzahl die Adresse zu ermitteln). Die Tabellenstrukturen können aber ohne weiteres ergänzt werden, um den Zugang über Namen zu unterstützen. Einschlägige Verfahren gehören zum allgemeinen Fachwissen der Informatik; entsprechende Funktionen sind in jedem Assembler oder Compiler vorgesehen. Sie müssen deshalb nicht näher beschrieben werden.

Es gibt drei Arten von Tabellen: die Ressourcenvorratstabelle (eine im System), die – bedarfsweise vorzusehende – Ressourcenvorratstabelle (eine je Ressourcentyp) und die Prozeßressourcentabelle (eine in jedem laufenden Programm (Prozeß)).

Ressourcentyptabelle
Die Ressourcentyptabelle enthält die beschreibenden und die Verwaltungsangaben zu den einzelnen Ressourcentypen (Abb. 6.1). Sie hat einen Eintrag je Ressourcentyp. Ein solcher Eintrag enthält:

- Eine allgemeine Typkennung,
- Angaben zu den Parametern (Operanden und Ergebnissen),
- Verwaltungsangaben.

Die Angaben zu den Parametern enthalten:

- Die Anzahl der Parameter,
- eine Beschreibung jedes einzelnen Parameters.

Die Verwaltungsangaben umfassen:

- Die Anzahl de insgesamt vorhandene Ressourcen des jeweiligen Typs,
- die Anzahl der aktuell verfügbaren Ressourcen,
- den Zustand jeder einzelnen Ressource. Im einfachsten Fall genügt ein Bit je Ressource, um zwischen den Zuständen „frei“ und „besetzt“ zu unterscheiden. Manchmal ist es von Vorteil, eine Verweisangabe auf das Programm (den Prozeß) mitzuführen, in dem die Ressource zur Zeit genutzt wird.

Die Verwaltung der Ressourcen entspricht weitgehend den Verwaltungsfunktionen, die in einem Dateisystem oder einer virtuellen Speicherorganisation erforderlich sind. Das betrifft z. B. das Auffinden eine geeigneten freien Ressource bei der Ausführung eines s-Operators. Entsprechende Prinzipien gehören zum allgemeinen Fachwissen der Informatik.

Jeder Parameter wir durch folgende Angaben beschrieben:

- Eine Typkennung,
- die Art des Parameters (Operand, Ergebnis oder beides),
- Angaben zur Verkettungssteuerung,
- die Länge in Bits.

Ggf. erforderliche Arbeitsbereiche der Ressourcen werden mit besonderen Parameterangaben beschrieben (an der Typkennung erkennbar).

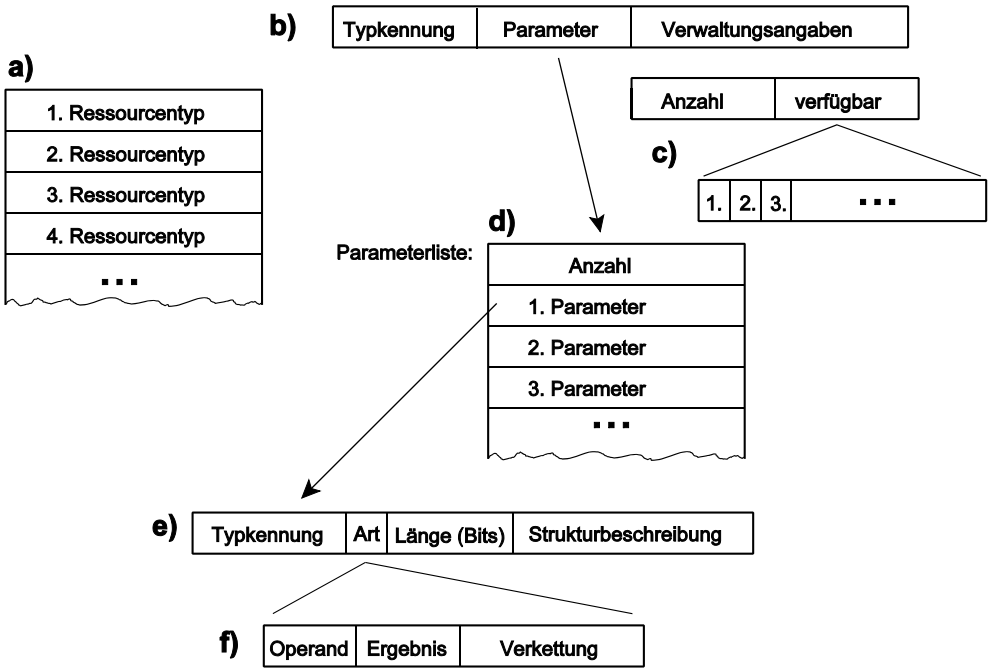


Abb. 6.1 Der Inhalt einer Ressourcentyptabelle

a) die Ressourcentyptabelle insgesamt. Jeder Ressourcentyp hat einen Eintrag.

b) der Inhalt eines Eintrags: Typkennung (eine Art Eigenname), Beschreibung der Parameter, Verwaltungsangaben.

c) elementare Verwaltungsangaben betreffen die Anzahl der vorhandenen Ressourcen dieses Typs und geben Auskunft darüber, welche dieser Ressourcen im Moment verfügbar sind. Das ist beispielsweise in einer Bitliste vermerkt (ein Bit je Ressource).

d) die Parameterbeschreibung insgesamt (Parameterliste). Jeder Parameter hat einen Eintrag.

e) der Inhalt eines Eintrags: Typkennung (elementare Typen sind z. B. Binär- und Gleitkommazahlen), Art des Parameters, Länge des Parameters, Strukturbeschreibung (bedarfsweise). Die Länge wird zunächst pauschal in Bits angegeben, unabhängig von der Struktur. Das unterstützt erste Entscheidungen zur Bereitstellung von Speicheplatz, zum Transport über Bussysteme und Interfaces usw. (alle Parameter sind letzten Endes Bitketten, die transportiert und gespeichert werden müssen). Kompliziertere Parameter haben zusätzlich eine Strukturbeschreibung.

f) so wird die Art des Parameters beschrieben: je ein Bit für die grundsätzliche Nutzungsweise (Eingang und Ausgang) sowie eine Verkettungsangabe (kennzeichnet, ob eine Verkettung überhaupt möglich ist und gibt die Art der Verkettung an).

Tabellenstrukturen

Da die Ressourcen unterschiedlich aufgebaut sind, ergeben sich verschieden lange Tabelleneinträge. In der Informatik gibt es vielfältige Lösungen des Problems, solche Tabellen in praktisch handhabbaren Datenstrukturen unterzubringen. Deshalb genügt hier die Kurzbeschreibung einer beispielhaften Ausführung.

Die Tabellen bestehen aus aus einem Kopf, der je Tabelleneintrag eine fest formatierte Angabe enthält. Diese Tabelleneinträge können direkt adressiert werden. Dem Tabellenkopf folgt ein variabler Teil, in dem die verbleibenden Angaben untergebracht werden. Jeder Eintrag im Tabellenkopf enthält einen Zeiger, der auf den zugehörigen Bereich im variablen Teil verweist. Am Anfang dieses Bereichs steht ein Rückverweis auf den Tabellenkopf, um die Verwaltung des variablen Teils zu unterstützen (Abb. 6.2, 6.3).

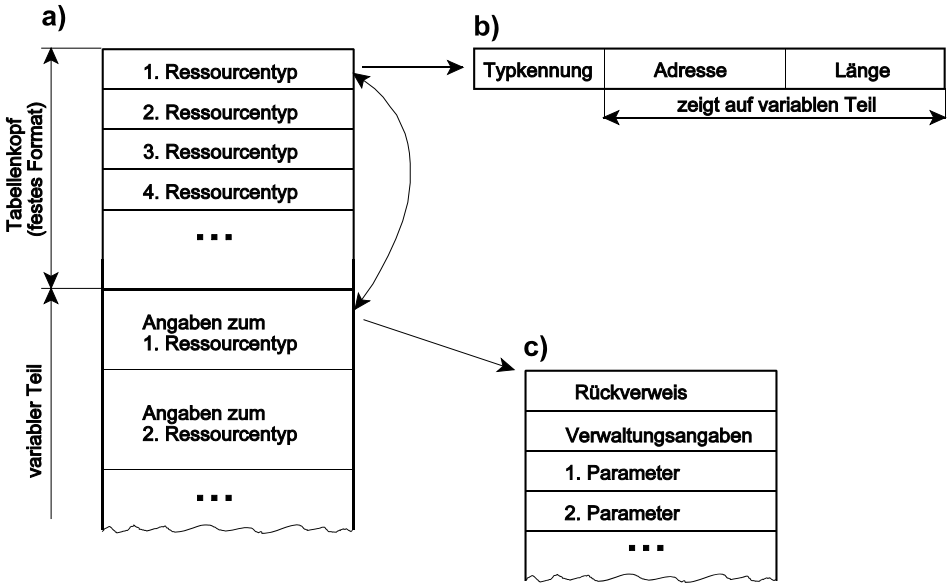


Abb. 6.2 Formatierungsbeispiel einer Ressourcentyptabelle

a) die Tabellenstruktur insgesamt,

b) ein Eintrag im Tabellenkopf. Im Beispiel enthält er lediglich einen Deskriptor, der den zugehörigen Bereich im variablen Teil beschreibt (Adreßzeiger, Längenangabe). Andere Implementierungen könnten weitere fest formatierte Angaben enthalten, z. B. zur Anzahl und zur Verfügbarkeit der Ressourcen.

c) so sieht ein Bereich im variablen Teil aus. Er enthält die weiteren Angaben zum Ressourcentyp gemäß Abb. 6.1.

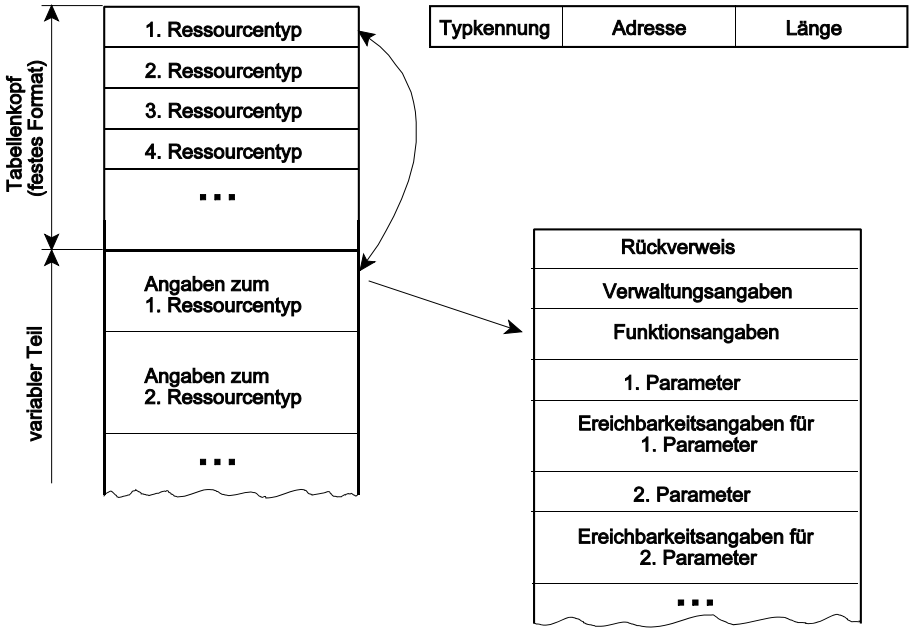


Abb. 6.3 Eine mit Funktions- und Erreichbarkeitsangaben ergänzte Ressourcentyptabelle

Die Einträge der Parameter können zusätzliche Angaben enthalten die die Ressoucenverwaltung und -nutzung unterstützen. Im folgenden werden zwei Beispiele – Funktions- und Erreichbarkeitsangaben – kurz erläutert.

Funktionsangaben
Funktionsangaben dienen dazu, universelle, für mehrere Funktionen geeignete Ressourcen auf die jeweils geforderte Funktion einzustellen.

Es gibt zwei Prinzipien der Funktionseinstellung:

1. Die universelle Ressource wird mittels s-Operator angefordert und durch Eintragen entsprechender Steuerangaben (z. B. mit p- oder mit u-Operatoren) auf die jeweilige Funktion eingestellt. Beispiel: es wird eine universelle ALU angefordert und mittels p-Operator auf 16 Bits Arbeitsbreite sowie auf Addition eingestellt.
2. Die einzelnen funktionellen Varianten werden als gesonderte Ressourcen verwaltet. Es gibt also z. B. 8-Bit-Addierwerke, 16-Bit-Addierwerke usw. als jeweils eigene Ressourcentypen. Wird mittels s-Operator z. B. ein 16-Bit-Addierwerk angefordert, so wird eine universelle ALU bereitgestellt und automatisch (als Zusatzwirkung des s-Operators) als 16-Bit-Addierwerk konfiguriert.

Im zweiten Fall ist zwischen logischen und physischen Ressourcen zu unterscheiden. Beide Arten werden in der Ressourcentyptabelle geführt. Fordert der s-Operator eine logische Ressource an (z. B. einen 32-Bit-Addierer), so trifft er auf einen Eintrag, der zur jeweiligen physischen Ressource verweist (z. B. auf eine universelle 64-Bit-ALU). Dieser Verweis ist in den Funktionsangaben untergebracht. Dort ist auch angegeben, auf welche Weise die physische Ressource zu konfigurieren ist (z. B. durch Funktionscodes, die in bestimmte Register zu laden sind).

Erreichbarkeitsangaben
Es kann sein, daß besondere Abläufe erforderlich sind, um Parameter in die Ressourcen zu übertragen oder von dort abzuholen. Die betreffenden Operatoren (p, a, l) müssen dann je nach Ressource und Parameter unterschiedliche Steuerwirkungen veranlassen (z. B. Bussysteme aktivieren, Register adressieren usw.). Das gilt sinngemäß für die Verkettungsabläufe. Solche Angaben können in der Ressourcentyptabelle mitgespeichert werden (Erreichbarkeitsangaben). Es kann sich dabei u. a. handeln um:

- Zugriffssteuerworte, Mikrobefehle oder Mikrobefehlsfolgen, die bestimmte Signalflüsse in der Hardware steuern,
- Folgen elementarer Maschinenbefehle (Transportroutinen),
- Zeiger auf entsprechende Transportroutinen,
- Adreßangaben (z. B. mit Busadressen der Hardware).

Erforderlichenfalls kann für jede Zugriffsart (p-Operatoren, a-Operatoren, l-Operatoren, Verkettung) eine besondere Angabe vorgesehen sein.

Ressourcenvorratstabelle
Es kann vorkommen, daß jede einzelne Ressource auf eigene Weise erreichbar ist (z. B. unter speziellen Adressen), daß also nicht alle Ressourcen eines Typs gleichsam pauschal behandelt werden können. Dann kann es zweckmäßig sein, je Ressourcentyp zusätzlich eine Ressourcenvorratstabelle zu führen, die die entsprechenden Angaben zu jeder einzelnen Ressource enthält (Abb. 6.4).

Prinzip:

- Die pauschalen Parameterbeschreibungen (Art, Länge usw.) stehen in der Ressourcentyptabelle,
- die Zustands- und Erreichbarkeitsangaben stehen in der Ressourcenvorratstabelle. Sie können ggf. durch zusätzliche Verwaltungsgaben ergänzt werden (die Aufschluß über die Nutzungshäufigkeit, die Anzahl der Zugriffe usw. geben).

Prozeßressourcentabelle
Die Prozeßressourcentabelle beschreibt die vom jeweiligen laufenden Programm (Prozeß, Task o. dergl.) angeforderten Ressourcen (Abb. 6.5). Sie enthält:

- Den Ressourcentyp (Rückverweis),
- die Ordinalzahl (laufende Nummer) der Ressource des betreffenden Typs,
- eine Erreichbarkeitsangabe.

Die Erreichbarkeitsangabe ist im einfachsten Fall eine Speicher- oder Hardwareadresse. Es kann sich aber auch um Mikrobefehle, Zeiger auf Transportroutinen usw. handeln. Solche Angaben werden – als Wirkung des s-Operators – aus der Ressourcentyptabelle oder aus der Ressourcenvorratstabelle übernommen, wobei sie ggf. abgewandelt werden (z. B. Ersatz relativer durch absolute Adressen).

Bei einer Softwarelösung (Emulation) ist die Erreichbarkeitsangabe typischerweise ein Zeiger in den Ressourcenemulationsbereich. Oft genügt ein einziger Zeiger, weil die Parameter fortlaufend adressiert werden. Manchmal ist aber eine gesonderte Zeigerangabe je Parameter erforderlich.

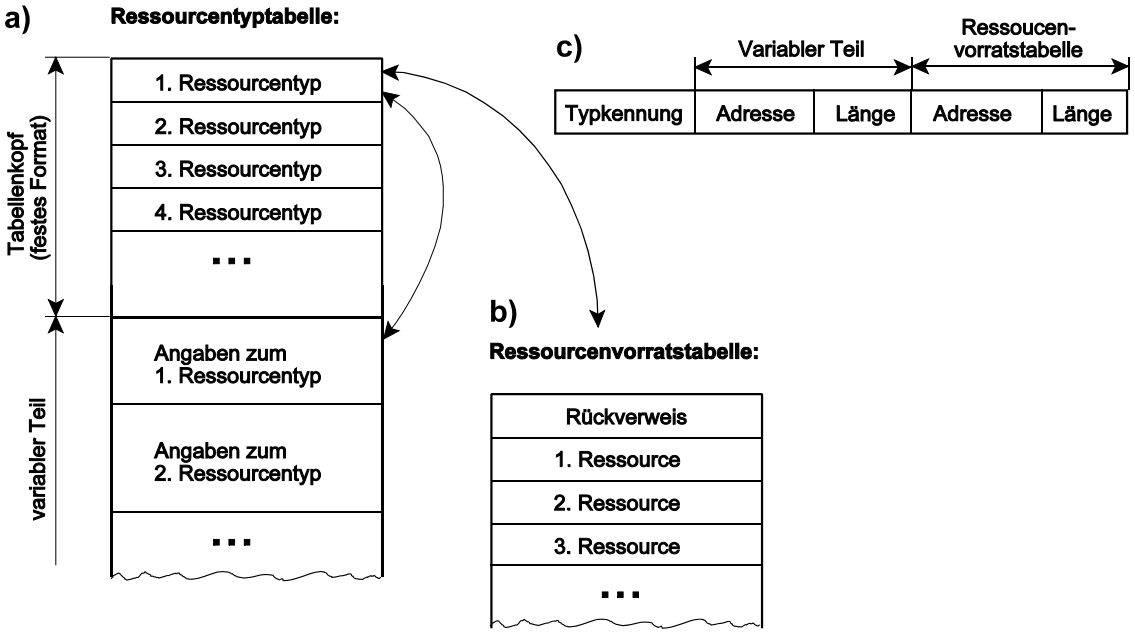


Abb. 6.4 Ressourcentyptabelle und Ressourcenvorratstabelle

a) Ressourcentyptabelle ähnlich Abb. 6.2,

b) zu verschiedenen Ressourcentypen gibt es jeweils eine Ressourcenvorratstabelle, die individuelle Angaben zu den einzelnen Ressourcen enthalten,

c) die Einträge im Kopf der Ressourcentyptabelle enthalten zwei Deskriptoren, einen für den Bereich im variablen Teil der Ressourcentyptabelle und einen für die zugehörige Ressourcenvorratstabelle.

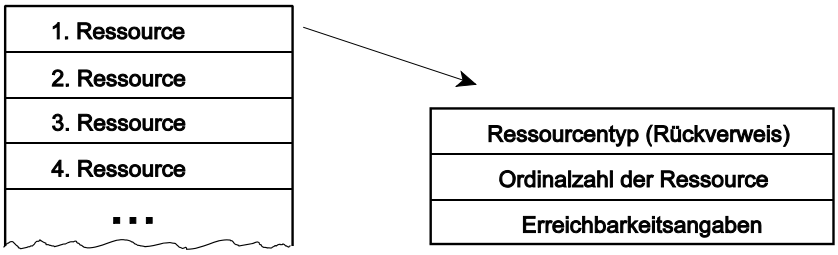


Abb. 6.5 Der Inhalt einer Prozeßressourcentabelle

Zur grundsätzlichen Nutzung der Tabellen:
Der s-Operator geht in die Ressourcentyptabelle und findet dort eine verfügbare Ressource. Dieser Ressource wird eine laufende Nummer (Ordinalzahl) gegeben. Ggf. werden verschiedene Einstellungen vorgenommen (Funktion, Verarbeitungsbreite usw.). Die für die weiteren Operatoren erforderlichen

Angaben werden in die Prozeßressourcentabelle übernommen. Die laufende Nummer dieses Eintrags ergibt eine weitere Ordinalzahl, mit der sich alle anderen Operatoren auf diese Ressource beziehen.

Beispiel:

1. Wir suchen einen 16-Bit-Addierer. Aufruf: s (ADD_16). Dieser Ressourcentyp habe die Ordinalzahl 25. Also lautet der s-Operator im symbolischen Maschinencode s(25).
2. der s-Operator findet die Ressource Nr. 6 dieses Typs als verfügbar vor,
3. die Ressource wird als belegt gekennzeichnet. Ggf. werden Funktionseinstellungen eingetragen.
4. die Ressource wird in die nächste freie Position der jeweiligen Prozeßressourcentabelle eingetragen (Ressourcentyp 25, Ressource Nummer 6 usw.). Die 11. Position der Prozeßressourcentabelle sei frei. Also erhält die Ressource die Ordinalzahl 11.
5. alle p-Operatoren, y-Operatoren usw. beziehen sich auf Ressource Nr. 11 und greifen mit diesem Wert auf die Prozeßressorcentabelle zu, um physische Adressen und andere Erreichbarkeitsangaben zu erhalten.

In der Praxis werden diese Abläufe typischerweise zur Compilier- und (teilweise) zur Ladezeit stattfinden; lauffähige Maschinenprogramme enthalten physische Ressourcenadressen und andere Erreichbarkeitsangaben; sie müssen nicht mehr auf Tabellen zugreifen.

Auflösung zur Compilierzeit bei softwareseitiger Emulation:
Der s-Operator geht in Ressourcentyptabelle und holt dort den Speicherbedarf (im Ressourcen-emulationsbereich). Die Prozeßressourcentabelle wird nur temporär beim Compilieren verwendet. Die Operation wird im Ressourcenemulationsbereich gespeichert. Zugriffe zur Laufzeit erfolgen nur über“gewöhnliche“ (= vom Compiler festgelegte) Adressen (alle Zugriffe zur Laufzeit beziehen sich auf Adressen im Ressourcenemulationsbereich). Der Ressourcenemulationsbereich kann ggf. in Hardware-Registern angelegt werden.

Hinweis:
Der s-Operator muß ggf. eine ganz bestimmte Ressource ansprechen können, z. B. Rechenwerk Nr. 22 oder Spezialprozessor MAX über IP-Adresse 123.45.67.89.

6.2 Anzahl der Ressourcen

ReAl-Systeme unterscheiden sich u. a. dadurch, wieviele Ressourcen jeweils zur Verfügung stehen. Es gibt folgende grundsätzliche Auslegungen:

1. Finit. Es handelt sich um eine feste Anzahl, die für eine bestimmte Implementierung gilt. Beispiel: ein Prozessor mit 16 Verarbeitungswerken. Somit können nicht mehr als 16 Verarbeitungsressourcen gleichzeitig vergeben werden.
2. Transfinit. Die Anzahl der Ressourcen wird lediglich beschränkt durch Grenzen des Adressierungsvermögens der beschreibenden Datenstrukturen.

Ressourcen in Software
Die Anzahl kann im Grunde transfinit sein; sie ist nur durch die Größe des Emulationsbereichs beschränkt. s- und r-Operatoren steuern lediglich die Belegung des Emulationsbereichs. Der einschlägige Verwaltungsaufwand ist Ermessenssache. Es geht hierbei um das in der Informatik wohlbekannte Problem der Freispeicherverwaltung einschließlich der sog. Garbage Collection, also darum, stückweise freigegebene Speicherbereiche wieder zur Nutzung bereitzustellen oder durch entsprechende Datentransporte einen einzigen zusammenhängenden freien Bereich zu schaffen. Eine Verwaltungsstrategie könnte darin bestehen, auf eine Freispeicherverwaltung zunächst zu verzichten, also r-Operatoren nur zu registrieren und mit jedem s-Operator bisher freien Speicher zu vergeben. Eine Garbage Collection findet nur an markanten Punkten im Programmablauf statt, z. B. dann, wenn ein bestimmter Programmzweig wirklich zu Ende gekommen ist und nicht erneut durchlaufen wird. Die Freispeicherverwaltung könnte hierbei z. B. durch entsprechende u-Operatoren unterstützt werden.

Ressourcen in Hardware
Die Anzahl ist letzten Endes finit. Jede einzelne Ressource muß verwaltet werden (zumindest ist zu konstatieren, ob sie verfügbar ist oder nicht).

Systematik der Kardinalitäten
Mit welchen Adreßlängen, Anzahlen usw. ist in der Praxis zu rechnen?

a) Analogie zur herkömmlichen Superskalarmaschine:

- 64...256 Ressourcentypen (= Hardware zum Ausführen von Maschinenbefehlen),
- maximal 4...8 Parameter. Einfache Operationen erzeugen aus 2 Operanden ein Ergebnis zuzüglich einer Zustandsmeldung (Flagbits). Manche Operationen, die man üblicherweise als elementar ansieht, brauchen einge Parameter mehr (betrifft z. B. Multiplikation und Divison von Binärzahlen).
- 16...256 aktive Ressourcen (herkömmliche Superskalarmaschinen haben typischerweise 4...16). Künftige Großschaltkreise können beispielsweise 4...16 herkömmliche Prozessoren aufnehmen, deren Ressourcenausstattung jeweils 4...16 Verarbeitungswerken entspricht.

b) Massive Parallelverarbeitung mit herkömmlichen Operationen:

- 64...256 Ressourcenarten (= Hardware zum Ausführen von Maschinenbefehlen),
- maximal 4...8 Parameter (vgl. unter a),
- Anzahl der aktiven Ressourcen: transfinit (Richtwerte: 1k...64k).

c) Emulation:

- Anzahl der Ressourcenarten: transfinit,
- Anzahl der aktiven Resourcen: transfinit,
- Parameter: verschiedene Stufen, z. B. 4, 8, 16, 32, 64, 512, 4k, transfinit. Mehr als 4k Parameter kommen in der Praxis kaum vor; die weitaus meisten Funktionen haben weniger als 64 Parameter.

Transfinit bedeutet hier, daß mit der vollständigen Ausnutzung des Wertebereichs bzw. Adreßraums gemäß Verarbeitungsbreite zu rechnen ist, also 2^{16} bei einer 16-Bit-Maschine, 2^{32} in einer 32-Bit-Maschine usw. In maschinenunabhängigen Codierungen ist die volle Bitanzahl gemäß Verarbeitungsbreite bzw. Adreßraum anzusetzen (16, 32, 64 Bits usw.). Maschinenspezifische Codes hingegen dürfen entsprechende Einschränkungen aufweisen (z. B. 40 statt 64 Adreßbits).

6.3 Numerierung der Ressourcen

Die Ressourcennummern (Ordinalzahlen) werden beim Auswählen der Ressourcen (s-Operator) zugewiesen. Alle weiteren Operatoren beziehen sich dann auf diese zugewiesenen Nummern. Typische Probleme der Zuweisung:

- Die Nutzung der Ressourcennummern als Adressen (bzw. Umwandlung dieser Nummern in Adressen oder in andere hardwareseitige Erreichbarkeitsangaben, z. B. in Zugriffssteuerworte),
- die Behandlung von Ressourcen, die zwischenzeitlich zurückgegeben werden (r-Operator).

Es sind verschiedene Verfahren denkbar:

1. Die Ressourcen werden gemäß der Auswahlreihenfolge in den s-Operatoren fortlaufend durchnumeriert.
2. Die Numerierung kann mit u-Operatoren gesteuert werden (z. B. durch Einstellen eines Anfangswertes).
3. Die Ordinalzahl- oder Adreßangaben werden in die s-Operatoren aufgenommen: s (Ressourcentyp => Ressourcennummer bzw. Adresse).

Numerierung und Freigabe (r-Operator)
Es ist typischerweise unzweckmäßig, auf jede Freigabe durch Umstellen der Numerierung zu reagieren (Verwaltungsaufwand). Die Alternative: die Durchnumerierung in den s- Operatoren wird fortgesetzt, unabhängig davon, ob zwischenzeitlich Ressourcen freigegeben werden oder nicht. Freigegebene Ressourcen können ohne weiteres neu zugewiesen werden; sie erhalten dann lediglich höhere Ordinalzahlen. Die Numerierung beginnt nur dann von neuem, nachdem die aktuelle Ressourcenzuweisung komplett aufgelöst wurde (z. B. am Programmende).

6.4 Ressourcen auf- und abbauen

Es bringt offensichtlich keine Vorteile, Ressourcen einzeln anzufordern, zu benutzen und sofort wieder freizugeben. Im Grunde wirken alle bekannten Prozessorarchitekturen auf diese Weise, nur finden die Abläufe im Innern der Hardware statt und müssen nicht programmseitig gesteuert werden. Da die entsprechenden Steuerangaben nicht im Programm erscheinen, sind die Maschinenprogramme viel kürzer.

Im Extremfall werden alle zu einem bestimmten Programmablauf benötigten Ressourcen auf einmal angefordert und – soweit möglich – parallel betrieben; das Programm beginnt mit einem s-Operator und endet mit einem r-Operator. Das ist aber nicht immer durchführbar (begrenzte Anzahl an Hardware-Ressourcen, begrenzte Speicherkapazität). Deshalb ist die Ressourcennutzung gleichsam stückweise zu organisieren. Übliche Trennstellen, an denen ein komplexes Progamm in überschaubare Abläufe zerlegt werden kann, sind u. a.:

- Einzeln compilierte Programmstücke einschließlich der darin aufgerufenen Funktionen,
- reguläre Programmkonstrukte (Fallunterscheidungen, Schleifen usw.),
- Programmblöcke (das, was in üblichen Programmiersprachen zwischen *begin* und *end* oder zwischen geschweiften Klammern steht) einschließlich der darin aufgerufenen Funktionen,
- Basisblöcke (lineare Folgen von Datentransporten und Verknüpfungen; ein Basisblock endet mit einer Verzweigung oder mit einem Funktionsaufruf). Typische Basisblöcke in herkömmlichen Maschinensprachen umfassen zumeist weniger als 10 Befehle.

Für das jeweilige Programmstück werden alle Ressourcen angefordert, genutzt und wieder freigegeben.

Ein naheliegendes Zuordnungsverfahren besteht darin, zunächst mit den Basisblöcken zu beginnen. Sind noch Ressourcen frei, nachdem der aktuelle Basisblock versorgt wurde, kann z. B. die nachfolgend aufgerufene Funktion berückichtigt werden usw.

6.5 Ressourcenadressierung in Byte- und Maschinencodes

Es entspricht dem Stand der Technik, nur zur Compilierzeit mit Ordinalzahlen, zur Laufzeit aber mit Adreßangaben zu arbeiten. In Byte- und Maschinencodes sind Ressourcen und deren Parameter zu adressieren. Es stehen zwei Auslegungen zur Wahl (Abb. 6.6):

1. Unabhängige Adreßangaben für die Ressourcen und für die Parameter innerhalb der Ressource (geteilter Ressourcenadreßraum).
2. Eine einzige Adreßangabe, die auf einen bestimmten Parameter innerhalb einer bestimmten Ressource verweist (flacher Ressourcenadreßraum).

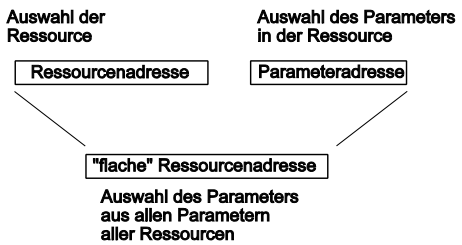


Abb. 6.6 Ressourcenadressierung: oben geteilter, unten flacher Adreßraum

Der geteilte Ressourcenadreßraum
Es gibt zwei Angaben: die eine wählt die jeweilige Ressource aus (Ressourcenadresse) und die andere den Parameter innerhalb der Ressource (Parameteradresse).

Vorteile:

- Die meisten Ressourcen haben nur wenige Parameter (Richtwert: 2...7). Wenn man die Ressourcenadresse entsprechend zwischenspeichert (z. B. in Basis- oder Halteregistern), müssen in vielen Operatoren nur die Parameteradressen mitgeführt werden (Verkürzung).
- Kürzere Adreßangaben in den y- und r-Operatoren,
- vergleichsweise geringer Aufwand zur Adreßdecodierung im Innern der Ressource (vgl. die üblichen Lösungen in den gängigen Mikrocontrollern, Peripherieschaltkreisen usw.),
- die Adressierung im Innern der Ressourcen ist unabhängig davon, welche anderen Ressourcen außerdem zu unterstützen sind.

Nachteile:

- Kompliziertere Maschinencodes (da zwei Adreßarten zu berücksichtigen sind),
- es sind eigens Basisregister, Halteregister o. dergl. erforderlich,
- Parameteradreßangaben müssen auch die Ressource mit den meisten Parametern unterstützen können. Kommen Ressourcen mit sehr vielen Parametern vor, wird die Parameteradresse lang, und zwar auch für all jene Ressourcen, die nur wenige Parameter haben. Sieht man – als Abhilfe – verschiedene Parameteradreßlängen vor, wird der Maschinencode noch komplizierter.

Die Parameteradresse im geteilten Ressourcenadreßraum
Auch hier stehen zwei Auslegungen zur Wahl:

1. Getrennte Adressierung von Operanden und Ergebnissen. In Abhängigkeit vom jeweiligen Operator betrifft die Parameteradreßangabe entweder einen Operanden oder ein Ergebnis. In p-Operatoren werden Operanden adressiert, in a-Operatoren Ergebnisse. In l-, c- und d-Operatoren betrifft die jeweils erste Parameteradreßangabe ein Ergebnis, die jeweils zweite einen Operanden.
2. Gemeinsame Adressierung von Operanden und Ergebnissen. Die Parameteradreßangabe betrifft alle Parameter.

Die getrennte Adressierung spart typischerweise ein Bit je Adreßangabe ein, erfordert aber gelegentlich etwas mehr Aufwand in der Hardware – es ist nicht nur eine Adresse zu berücksichtigen, sondern auch die Art ihrer Nutzung (die z. B. in einem Bussystem über eine besondere Leitung übermittelt werden müßte).

Beispiel: es gibt höchstens 4 Operanden und 3 Ergebnisse. Die getrennte Adressierung erfordert 2 Bits, die gemeinsame 3 Bits.

Eine weitere Möglichkeit zur Codeverkürzung: Die Ressourcen haben typischerweise mehr Operanden-Parameter als Ergebnis-Parameter. Somit kann gelegentlich die Adreßlänge für Ergebnisse kürzer ausgelegt werden als für Operanden.

Der flache Ressourcenadreßraum
Es gibt einen einzigen Adreßraum in dem jedem Parameter jeder Ressource eine bestimmte Adresse zugeordnet ist (fortlaufende Adressierung).

Vorteile:

- Einfacherer Maschinencode,
- der einheitliche lineare Adreßraum ist ein seit langem bewährtes Architekturprinzip,
- eine hinreichende Adreßlänge erlaubt es, beliebige Mischungen von Ressourcen mit beliebig vielen Parametern auf einfache Weise zu unterstützen.

Nachteile:

- Gelegentlich höherer Speicherbedarf für den Maschinencode (wegen der längeren Adreßangaben),
- gelegentlich aufwendigere Adreßdecodierung (in der Hardware), da für jeden einzelnen Parameter die Adresse in voller Länge decodiert werden muß. Sieht man eine bereichsweise Adreßdecodierung vor, um die Decodier-Hardware zu vereinfachen (vgl. PCI-Bus), ergibt sich gelegentlich eine schlechte Ausnutzung des Adreßraums (was dazu zwingen kann, die Adresse nochmals zu verlängern).
- gelegentliche Verwaltungsschwierigkeiten (Overhead), wenn Ressourcen zur Laufzeit ausgewählt und freigegeben werden (dynamische Ressourcenverwaltung).

Auslegungsbeispiele der Ressourcenadressierung:

- Nur Ressourcen mit vergleichsweise wenigen Parametern: geteilter Adreßraum,
- nur gleichartige Ressourcen: geteilter Adreßraum oder Aktivierung über Zugriffssteuerworte (vgl. den folgenden Abschnitt),
- gemischte Ressourcenbelegung (darunter Ressourcen mit vergleichsweise vielen Parametern): flacher Adreßraum.

Mit welchen Größenordnungen ist zu rechnen?
Wir können unterscheiden:

1. ReAl-Maschinen mit transfiniter Anzahl an Ressourcen (die sich ggf. in riesigen Anzahlen anfordern lassen). Vorzugsweise Nutzung: als fiktive Maschinen zur Programmentwicklung und -optimierung. Es geht darum, den innewohnenden Parallelismus soweit wie irgend möglich zu erkennen (d. h. über das gesamte Programm hinweg ohne Beschränkung durch womöglich fehlende Hardware) und herauszufinden, wie sich eine konkete Architektur (mit beschränkter Ressourcenzahl, ggf. auch mit beschränkten Zugriffsmöglichkeiten (SIMD, VLIW usw.)) zweckmäßig ausnutzen läßt.
2. ReAl-Maschinen mit – letzten Endes – finiter (und technisch realisierbarer) Hardware-Ausstattung (z. B. auf FPGA-Schaltkreisen). Die Anzahl der Register kann bestimmte Größenordnungen nicht überscheiten. Handelt es sich um feste Konfigurationen, kann man auch die Registeradressen von Anfang an festlegen (vgl. die Registeradressierung in herkömmlichen Schaltkreisen). In weitgehend programmierbaren Konfigurationen kann man den Ressourcen besondere Konfigurationsregister zuordnen, über die die jeweiligen Adressen eingestellt werden (vgl. die Adressierung an Bussystemen mit Plug-and-Play-Unterstützung).
3. ReAl-Maschinen in Software (Emulation). Die Anzahl der Ressourcen kann beträchtlich sein (Frage der Programmoptimierung).

Tabelle 6.1 nennt einige Beispiele.

Variablenadressierung
Die Variablen befinden sich in den Speichereinrichtungen der Plattform. Variable werden als Operanden in die Ressourcen transportiert bzw. mit Ergebnissen der Ressourcen belegt. Solche Transporte werden von der Plattform ausgeführt (p- und a-Operatoren), können aber auch von entsprechend ausgelegten Ressourcen erledigt werden.

Typischerweise genügt es, die Variablenadressierung seitens der Plattform so auszulegen, daß die in den Laufzeitsystemen der gängigen höheren Programmiersprachen üblichen Zugriffsprinzipen unterstützt werden (Adreßrechnungsschema Basis + Displacement, wobei als Basisadreßregister wenigstens ein Frame bzw. Base Pointer (FP/BP), ein Stackpointer (SP) und ein weiteres Zeigerregister vorgesehen sind).

Ressourcen	Parameter	Adreßraum**)	Anmerkungen
4	4 (getrennt)*)	16 4 Bits (2/2)	herkömmliche Superskalarmaschine
32	4 (getrennt)*)	128 7 Bits (5/2)	Superskalarmaschine nach ReAl-Prinzipien, Softwarelösung auf Mikrocontrollern
64	4 (getrennt)*)	256 8 Bits (6/2)	Superskalarmaschine nach ReAl-Prinzipien, FPGA, Softwarelösung auf Mikrocontrollern
256	4 (getrennt)*)	1024 10 Bits (8/2)	FPGA, Softwarelösung auf Mikrocontrollern
1024	4 (getrennt)*)	4096 12 Bits (10/2)	FPGA (sehr groß), Parallelverarbeitungssystem, Softwarelösung auf Mikrocontrollern o. dergl.
64k	16	1024k 20 Bits (16/4)	Laufzeitumgebung auf Hochleistungsrechnern; Compilierungs-Ziel z. B. für herkömmliche C-Programme
1024k	64	64M 26 Bits (20/6)	extreme Softwarelösung

*): max. 4 Operanden und 4 Egebnisse; **): in Klammern: Ressourcenadresse/Parameteradresse

Tabelle 6.1 Zur Größenordnung typischer Ressourcenadressen (Beispiele)

6.6 Ressourcen als Hardware

Jeder Parameter entspricht typischerweise einem Register. Adreßangaben sind im Grunde Ordinalzahlen (Auswahl des 1., 2., 3. Registers usw.). Es ist nicht besonders schwierig, alle Register der ausgewählten Ressourcen zu adressieren. Oftmals genügt es, den einzelnen Registern feste Adressen zuzuweisen. Die Schaltungstechnik auf Grundlage von Areßvergleichern oder -decodern ist seit langem bekannt (Abb. 6.7 bis 6.9; vgl. Mikroprozessoren mit E-A-Schaltkreisen, Steckkarten für Bussysteme usw.).

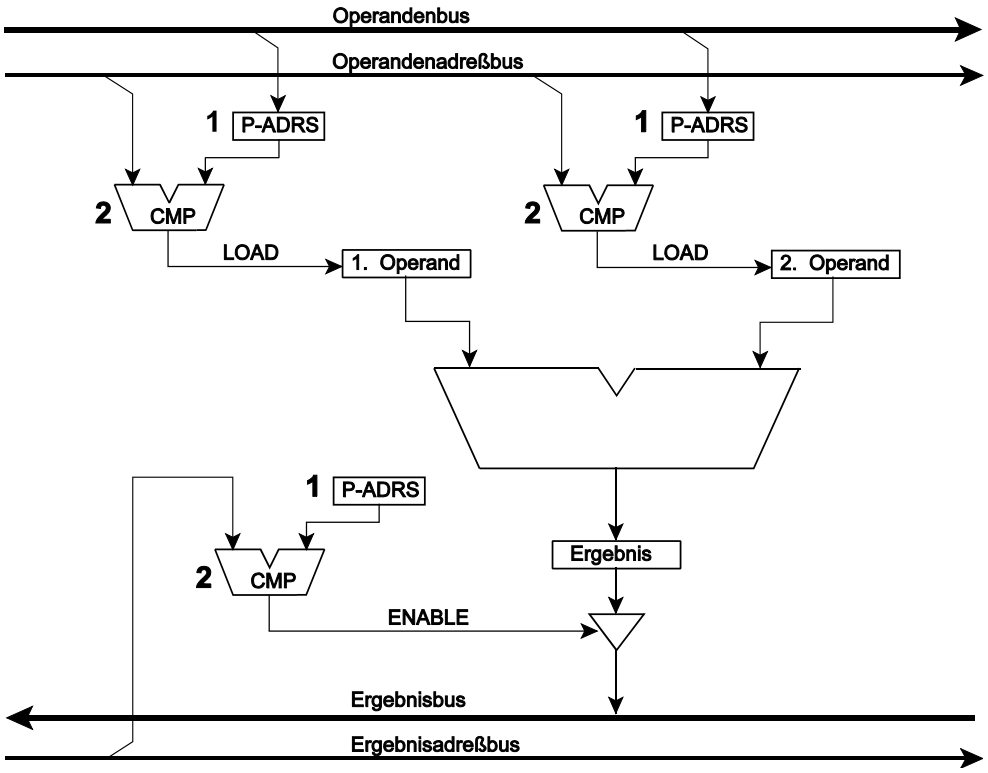


Abb. 6.7 Parameteradressierung in einer Hardware-Ressource. 1 - Adreßeinstellung (Festwerte oder im Rahmen von Konfigurationzugriffen ladbare Adreßregister); 2 - Adreßvergleicher

Die Zieladresse (des Parameters, der überschrieben werden soll) wird auf den Operandenadreßbus gelegt, die Quelladrese (des zu lesenden Parameters) auf den Ergebnisadreßbus. Erkennt einer der Adreßvergleicher 2, daß die anliegende Adresse der eingestellten Adresse 1 entspricht, so wird die betreffende Zugriffsfunktion ausgeführt (Laden eines Operandenregisters vom Operandenbus, Aufschalten eines Ergebnisses auf den Ergebnisbus).

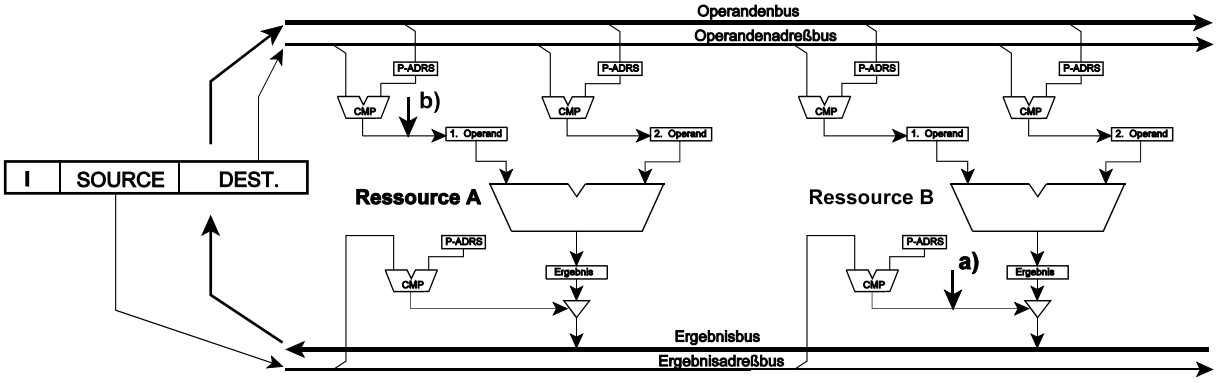


Abb. 6.8 Parametertransport zwischen zwei Ressourcen. a) Ergebnis aufschalten; b) Operanden laden

Das Ergebnis der Ressource B wird zum ersten Operanden der Ressource A (l-Operator). Hierzu wird die Zieladresse (DEST.) auf den Operandenadreßbus gelegt und die Quelladresse (SOURCE) auf den Ergebnisadreßbus. Die Datenleitungen beider Bussyteme werden (z. B. in der Plattform) miteinander verbunden (Datenweg vom Ergebnisbus zum Operandenbus).

Das Prinzip kann sinngemäß auf serielle Interfaces übertragen werden, die z. B. an Schaltverteiler angeschlossen sind. Die Wandlung herkömmlicher Busprotokolle in die bitserielle Hochgeschwindigkeitsübertragung gehört zum Stand der Technik (vgl. beispielsweise PCI Express).

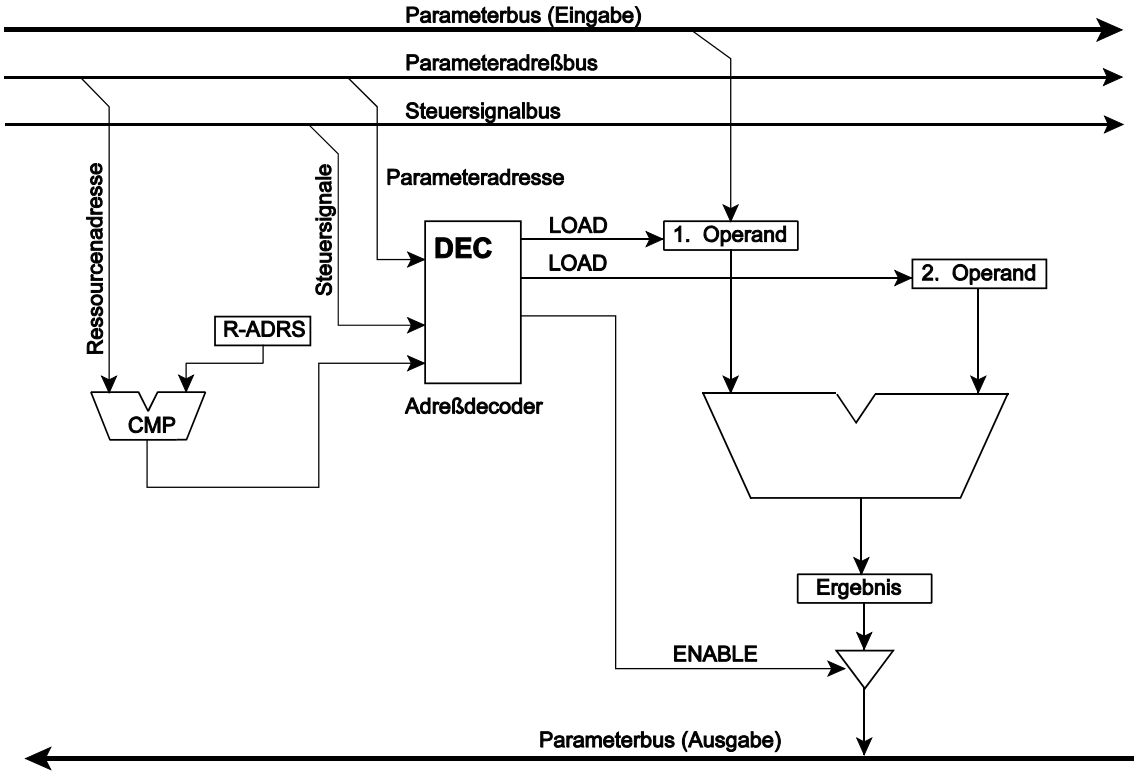


Abb. 6.9 Parameteradressierung über einen geteilten Adreßraum

Im Interesse der Überschaubarkeit ist hier eine Lösung mit gemeinsamen Adreß- und Steuerleitungen für Ein- und Ausgabe dargestellt. Die Datenwege können zu einem bidirektionalen Datenbus zusammengefaßt werden (vgl. die typischen Bussysteme der Mikroprozessoren). Die Gesamt-Ressourcenadresse ist auf einen Adreßvergleicher geführt. Die Parameteradresse wird decodiert. Der Adreßvergleicher aktiviert den Adreßdecoder.

Adressierung und Aktivierung über Zugriffssteuerworte
Eine alternative Auslegung. Die Ressourcen werden nicht über eine binäre Adresse, sondern über Zugriffssteuerworte aktiviert (Abb. 6.10). Das kann sowohl die Operationsauslösung (y-Operator, Verkettung) als auch die Parameterübertragung betreffen.

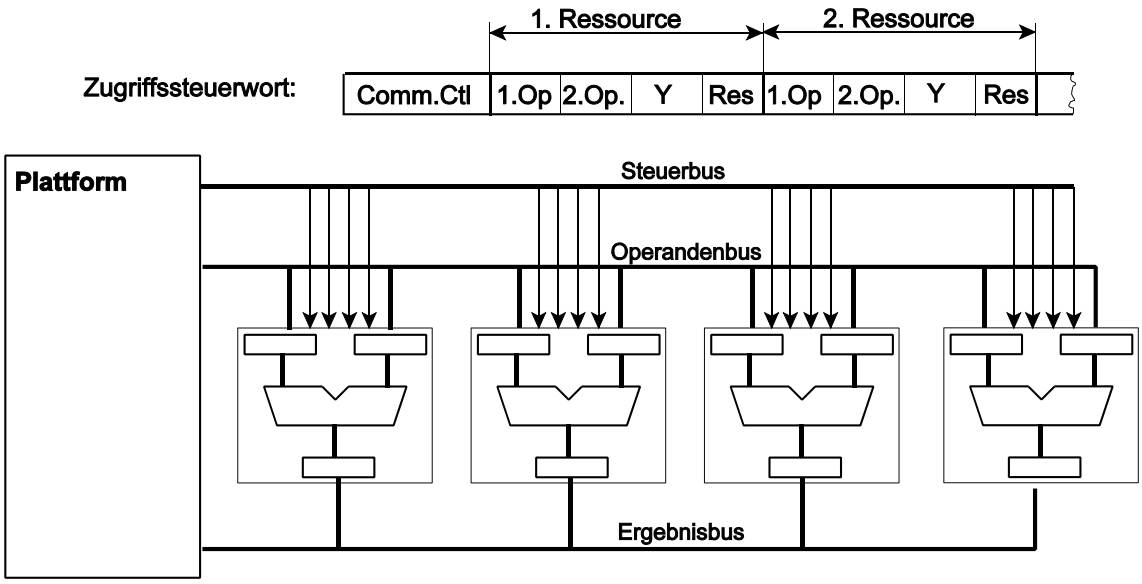


Abb. 6.10 Adressierung und Aktivierung von Ressourcen über Zugriffssteuerworte

Ein Zugriffssteuerwort betrifft mehrere Ressourcen gleichzeitig. In einem solchen Steuerwort können die Wirkungen mehrerer ReAl-Operatoren zusammengefaßt werden. Im Extremfall kan ein Steuerwort alle Transport- und Verarbeitungvorgänge auslösen, die in allen Ressourcen zum jeweiligen Zeitpunkt überhaupt ablaufen können. Im Beispiel von Abb. 6.10 hat das Zugriffssteuerwort eine Bitposition je Ressource und Vorgang.

Bitbelegung:

- 1.Op = Parameter vom Operandenbus in das erste Operandenregister der Ressource eintragen,
- 2.Op = Parameter vom Operandenbus in das zweite Operandenregister der Ressource eintragen,
- Y = Operation auslösen,
- Res = Ergebnis auf Ergebnisbus aufschalten.

Im allgemeinen Steuerfeld Comm.Ctl codierte Wirkungen (Auswahl):

- Datenwort auf Speicher auf den Operandenbus legen,
- Ergebnis speichern,
- Ergebnis auf Operandenbus zurückführen,
- Auswahl des nächsten Steuerwortes (u. a. durch Verzweigung oder Unterprogrammruf).

Abb. 6.10 zeigt eine der einfachsten Systemstrukturen (ein Bussystem mit zwei Datenwegen). Zu einer Zeit können nur ein Operand und ein Ergebnis transportiert werden. Der Operand kann dabei gleichzeitig in beliebig viele Ressourcen eingetragen werden.

Höherentwickelte Systeme können mehrere Busstrukturen oder geschaltete Punkt-zu-Punkt-Verbindungen haben. Die Steuerworte enthalten dann anstelle der Einzelbits entsprechende Adreßfelder.

6.7 Ressourcen als Software (Emulation)

Jeder Parameter entspricht typischerweise einer Speicherposition. Es liegt nahe, die Speicherpositionen der Ressourcen aufeinanderfolgend anzuordnen, so daß sich die fortlaufende Adressierung in einem flachen Adreßraum gleichsam von selbst ergibt. Ein geteilter Adreßraum erfordert mehr Aufwand zur Adreßrechnung (Abb. 6.11). Eine über das Schema Basis + Displacement hinausgehende Adreßrechnung wird aber von gängigen Prozessoren gar nicht oder nur nachlässig unterstützt (Ablaufverlangsamung).

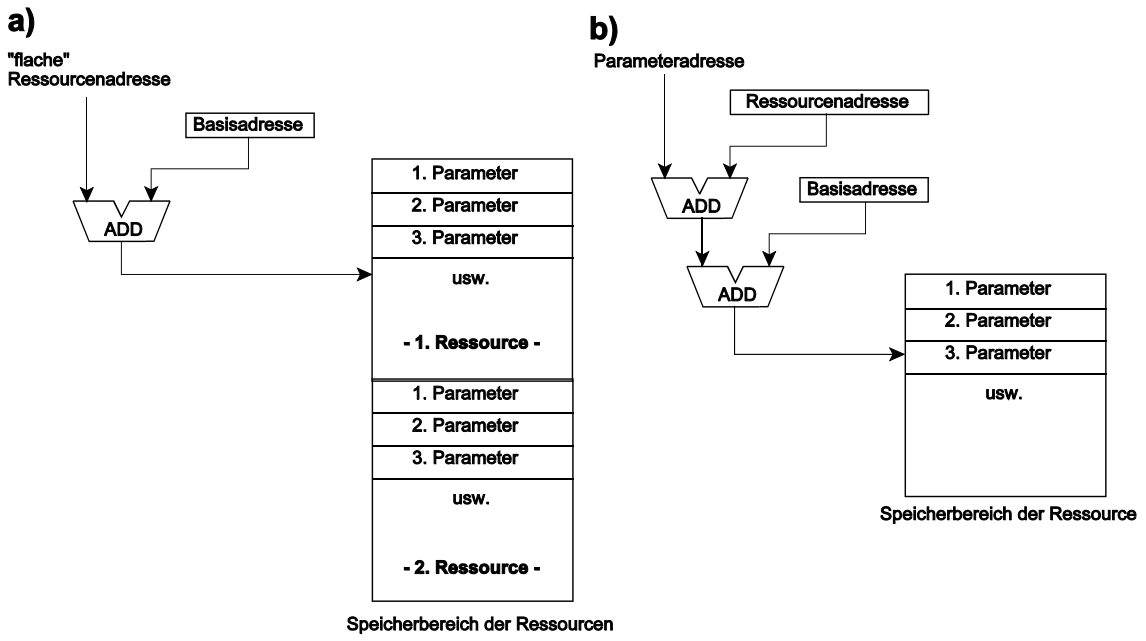


Abb. 6.11 Parameteradressierung im Speicher. a) flacher; b) geteilter Adreßraum

Es ist grundsätzlich möglich, von Ressourcen belegte Speicherbereiche in eine virtuelle Speicherorganisation einzubeziehen. Ressourcen, mit denen zur Zeit nicht gearbeitet wird, können auf den Massenspeicher ausgelagert werden.

Des weiteren ist es möglich, komplette Ressourcenbelegungen als Dateien zu speichern und zwecks Ausführung wieder zu laden. Die Ressourcenstruktur müßte so nur einmal aufgebaut werden (mit s- und c-Operatoren). Für jede weiteren Nutzung genügt ein einfacher Ladevorgang. Solche vorgefertigten Strukturen können auch vom Programmentwickler erzeugt und im Rahmen der betreffenden Software komplett ausgeliefert werden, so daß beim Anwender die entsprechenden s- und c-Operatoren gar nicht ausgeführt werden müssen.

6.8 Zustandsfreie und zustandsbehaftete Ressourcen

Der Begriff „Zustand“ ist hier im Sinne des architekturseitigen Programmiermodells zu verstehen. Ein Programm befindet sich zu jedem beliebigen Zeitpunkt in einem bestimmten Verarbeitungszustand. Im Falle von Unterbrechungen, Taskumschaltungen usw. ist dieser Zustand zu retten. Erhält das Programm später erneut Laufzeit, ist der gerettete Zustand wieder einzustellen. Bei der Auslegung einer ReAl-Maschine ergibt sich die Frage, ob die Ressourcen in den Verarbeitungszustand einzubeziehen sind oder nicht.

Zustandsbehaftete Ressourcen
Deren Belegung gehört zum Verarbeitungszustand bzw. Programm-Kontext. Wird eine Ressource in den Verarbeitungszustand einbezogen, so hat das folgende Konsequenzen:

- Ddie in der Ressource gespeicherten Angaben sind bei Unterbrechungen, Taskumschaltungen usw. zu retten und hinterher ggf. wieder einzustellen (erfordert entsprechende Zugriffswege usw. und erhöht die Latenzzeit der Kontextumschaltung),

- die Speichermittel (Register usw.) in den Ressourcen sind vollwertige Speicher im Sinne des Programmiermodells; es ist möglich, Variable, Zwischenergebnisse usw. allein in den Ressourcen zu halten,
- Ergebnisse können auf Eingänge der gleichen Ressource zurückgeführt werden (INOUT-Parameter; Abb. 6.12),
- die Verkettung ist unbeschränkt nutzbar,
- Unterbrechungen, Taskumschaltungen können jederzeit stattfinden, d. h. ohne auf den internen Verarbeitungszustand der Ressourcen Rücksicht zu nehmen (alle länger dauernden Verarbeitungsvorgänge sind jederzeit unterbrechbar).

Bei voller Ausnutzung der Verkettung in einer entspechend ausgebildeten ReAl-Maschine gibt es praktisch keine lokalen Variablen mehr, die eigens in entsprechenden Speicherbereichen (z. B. Stack Frames) gehalten werden müßten. Ebenso entfallen die betreffenden Transportbefehle zum Zwischenspeichern und Zurückholen (a- und p-Operatoren).

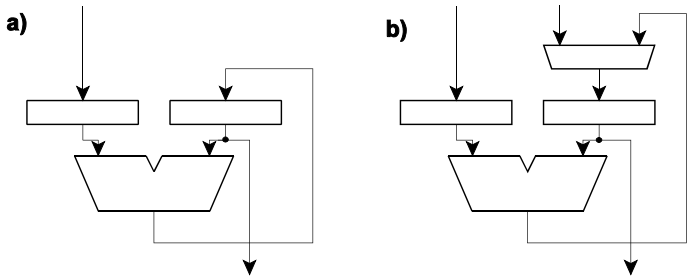


Abb. 6.12 Zustandsbehaftete Ressourcen. a) ein Ergebnis wird im nächsten Verarbeitungsgang als Operand verwendet (Rückführung); b) dieser Parameter ist sowohl Operand als auch Ergebnis (Typ INOUT)

Zustandsfreie Ressourcen
Diese Ressourcen werden nicht in den Verarbeitungszustand einbezogen. Eine Ressource heißt zustandsfrei, wenn sie ihre Parameter nicht über den jeweils aktuellen Verarbeitungsvorgang hinaus speichert, mit anderen Worten, wenn sie sich praktisch wie eine kombinatorische Schaltung verhält. Das hat folgende Konsequenzen:

- Die in der Ressource gespeicherten Angaben werden bei Unterbrechungen, Taskumschaltungen usw. nicht gerettet (entsprechende Zugriffswege sind nicht erforderlich, die Latenzzeit der Kontextumschaltung ist vergleichsweise geringer),
- alle Variablen, Zwischenergebnisse usw. sind im Arbeitsspeicher zu halten,
- es gibt keine Parameter, die zugleich Eingänge und Ausgänge sind (INOUT). Kein Eingang ist zurücklesbar, kein Ausgang durch Eintragen eines Eingangswertes (eines Operanden) überschreibbar.
- die Verkettung ist nur beschränkt nutzbar (z. B. zum Holen von Operanden und zum Abspeichern von Ergebnissen),
- vor dem Auslösen eines y-Operators sind alle Eingänge stets neu zu belegen. Das betrifft auch jene Werte, die sich gar nicht geändert haben.
- Unterbrechungen, Taskumschaltungen usw. können nur dann stattfinden, nachdem die Ergebnisse der in den Ressourcen ausgelösten Verarbeitungsvorgänge in den Arbeitsspeicher transportiert wurden. Auch alle durch Verkettung ausgelösten Abläufe müssen zu Ende gekommen sein. Kein Verarbeitungsvorgang ist in sich unterbrechbar.

Wann sind die Latenzzeiten kürzer?
Zustandsbehaftete Ressourcen brauchen Zeit zum Retten und Wiedereinstellen (Save/Restore), die Verarbeitungsvorgänge sind aber jederzeit unterbrechbar, und während der Verarbeitung fallen weniger Speicherzugriffe an.

Sind die Ressourcen zustandsfrei, entfällt das Retten und Wiedereinstellen, die Verarbeitungsvorgänge sind aber nicht unterbrechbar, und es sind insgesamt mehr Speicherzugriffe erforderlich.

Kommt es auf geringe Latenzzeiten an, so ist zu untersuchen, was länger dauert: das Retten und Wiedereinstellen oder das Zu-Ende-Bringen aller Verarbeitungsvorgänge einschließlich der zusätzlichen Speicherzugriffe zum Holen der Operanden und zum Abspeichern der Ergebnisse.

Geht es um maximale Verarbeitungsleistung, sind typischerweise zustandsbehaftete Ressourcen vorzuziehen, da es nur diese Auslegung ermöglicht, die Ressourcen uneingeschränkt zu verketten, Ergebnisse auf Eingänge zurückzuführen und die internen Speichermittel zur Datenspeicherung auszunutzen.

Um die Latenzzeiten zu verringern, können den zustandsbehafteten Ressourcen zusätzliche Speichermittel zugeordnet werden (Abb. 6.13).

Derartige Speicheranordnungen können nicht allzu groß sein (Richtwert: 4...16 Speicherpositionen), weil die Zugriffszeiten ansonsten zu lang werden (was dazu zwingt, die Taktfrequenz zu senken oder zusätzliche Pipeline-Stufen einzufügen). Ein Ausweg besteht darin, Speicherinhalte, mit denen zur Zeit nicht gearbeitet wird, über unabhängige Zugriffswege (in Abb. 6.13: Rettungsbus) in den Arbeitsspeicher zu überführen und bei Bedarf von dort aus wieder zurückzuschreiben (Save/Restore). Diese Abläufe finden parallel zu den eigentlichen Verarbeitungsvorgängen statt.

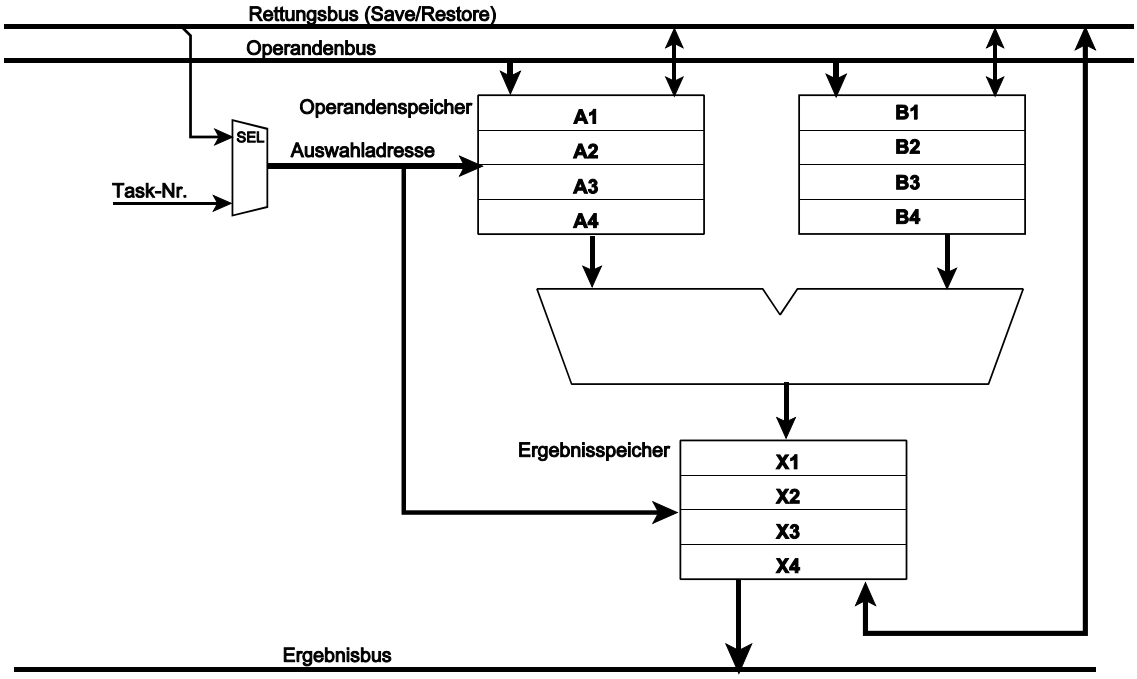


Abb. 6.13 Zustandsbehaftete Ressource mit zusätzlichen Speichermitteln

Es ist eine einfache Verarbeitungsressource dagestellt, die aus zwei Operanden ein Ergebnis bildet. Die Operanden- und Ergebnisspeicher sind aber keine einfachen Register, sondern adressierare Speicheranordnungen, die beispielsweise als Register- oder als RAM-Arrays ausgebildet sind. Jeder Task (bzw. jeder Unterbrechungsebene) ist eine Speicherposition in diesen Speichermitteln zugeordnet. Die Speicherpositionen, mit denen die Ressource arbeitet, werden über die laufende Nummer der Task bzw. Unterbrechungsebene ausgewählt. Eine Taskumschaltung bzw. Unterbrechung bedeutet lediglich ein Umschalten der besagten laufenden Nummer.

6.9 Instrumentierung

Unter „Instrumentierung“ verstehen wir hier – wie im Bereich der lokalen Netzwerke, der PCs usw. üblich – die Bestückung der Systeme mit zusätzlichen Vorkehrungen zur Systemverwaltung, zur Leistungsmessung, zum Fehlersuchen (Debugging) usw. Um solche Funktionen in ReAl-Maschinen bereitzustellen, gibt es folgende Möglichkeiten:

- die Ressourcen werden mit zusätzlichen Einrichtungen erweitert (Abb. 6.14 bis 6.16),
- es werden spezielle Ressourcen für diese Zwecke vorgesehen (Abb. 6.17),
- entsprechende Anordnungen werden durch Verkettung passender Ressourcen ad hoc erzeugt (Abb. 6.18).

1	Anfangswert
2	untere Grenze
3	obere Grenze
4	aktueller Wert
5	Vergleichswert
6	Ausnahme- und Debugging-Steuerung
7	Nutzungszähler
8	Zustandsbits
9	Verkettungszeiger vorwärts
10	Verkettungszeiger rückwärts

1- Anfangswert zwecks Initialisierung; 2, 3 - Grenzen des Wertebereichs; 4 - der aktuelle Wert; - Vergleichswert; 6 - Steuer- und Adreßangaben zur Auslösung von Programmausnahmen und zum Verhalten bei erfüllter Stopbedingung; 7 - Nutzungszähler; 8 - diverse Zustandsbits (u. a. zur Verkettungssteuerung); 9, 10 - Verkettungszeiger.

Abb. 6.14 Ein Parameter in einer Ressource – Beispiel einer extremen Auslegung

Es ist dargestellt, welche Zusatzangaben einen einfachen Parameter (z. B. eine Binärzahl) ergänzen können: Die üblichen Laufzeitsysteme vieler Programmiersprachen unterstützen nur den aktuellen Wert. Die Angaben 1, 2, 3 dienen dazu, die Implementierung einschlägiger Programmiersprachen zu unterstützen (Beispiel: Ada). Der Vergleichswert 5 ist zu Debugging-Zwecken vorgesehen. Beispiel: Anhalten der Verarbeitung und Anzeige des Verarbeitungszustandes (Vergleichsstop), wenn der Parameter einen bestimmten aktuellen Wert angenommen hat. Der Nutzungszähler 7 kann z. B. verwendet werden, um festzustellen, wie oft der Parameter in Verarbeitungsvorgängen verwendet wurde oder welche Zeit zwischen zwei Neuberechnungen vergangen ist.

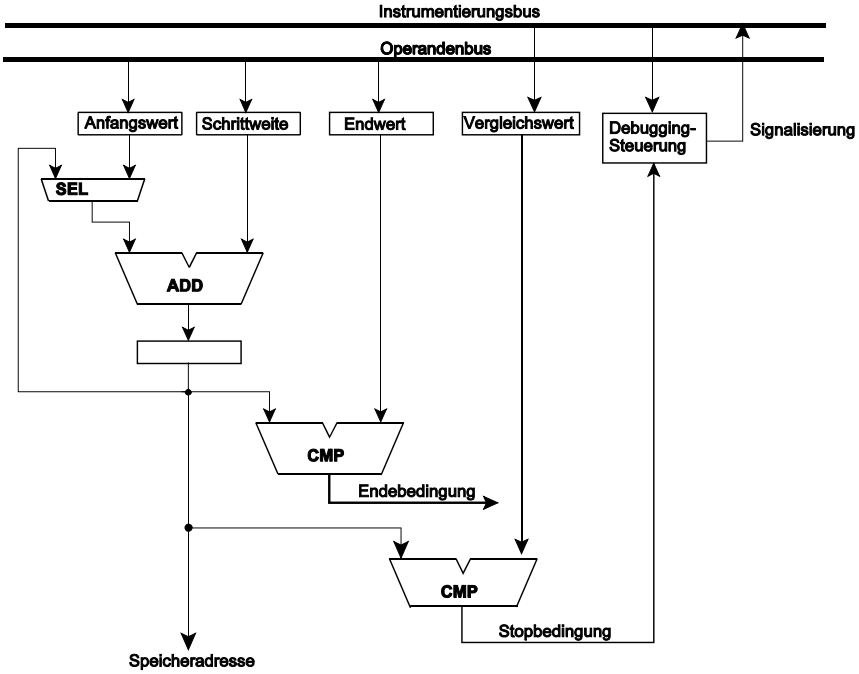


Abb. 6.15 Ressource mit eingebauten Debugging-Vorkehrungen

Ein einfacher Iterator ist um eine Vergleichseinrichtung erweitert, die die erzeugte Speicheradresse mit einem eingestellten Wert vergleicht und bei Gleichheit eine Stopbedingung signalisiert (Vergleichsstop). Sinngemäß können die Ressourcen mit Schaltmitteln zur Überwachung von Wertebereichen, mit Meßzählern usw. ausgerüstet werden.

Die Stopadressen, Bereichsangaben, Zählwerte usw. können ebenso eingestellt und abtransportiert werden wie die üblichen Operanden und Ergebnisse; sie werden einfach als zusätzliche Parameter angesehen. Das erfordert aber eine entsprechende Erweiterung des Parameteradreßraumes und somit mehr Adreßbits im Maschinencode.

Alternativ dazu können besondere Signalwege für die Instrumentierungsangaben vorgesehen werden, z. B. ein Instrumentierungsbus. Da die Transporte der Instrumentierungsangaben nicht leistungskritisch sind (solche Angaben werden nur von Zeit zu Zeit eingestellt oder abgerufen), genügt eine entsprechend einfache Auslegung (z. B. als serielles Bussystem).

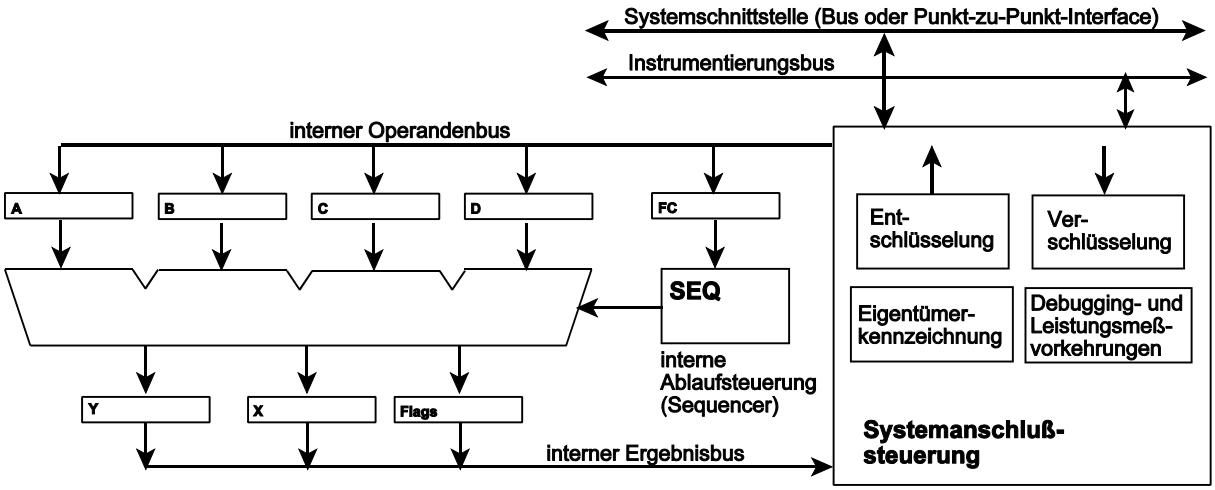


Abb. 6.16 Ressource mit eingebauter Instrumentierung

Eine etwas komplexere Verarbeitungsressource (vgl. Kapitel 9) ist mit Instrumentierungsvorkehrungen ausgerüstet. Zusätzlich zu den bereits erläuterten Debugging- und Leistungsmeßvorkehrungen sind folgende Funktionen vorgesehen:

1. Entschlüsselung der ankommenden Operanden und Verschlüsselung der abgehenden Ergebnisse. Ver- und Entschlüsselungseinrichtungen sind an sich bekannt. Hier sind sie in die Ressource eingebaut. Das hat den Vorteil, daß auf externen Bussystemen (die z. B. auf Leiterplatten verlaufen) nur verschlüsselte Daten bewegt werden.
2. Eigentümerkennzeichnung. Im einfachsten Fall handelt es sich um abfragbare Festwerte. Höher-entwickelte Ressourcen können einen eingebauten Paßwortschutz haben; sie sind nur dann nutzbar, wenn zuvor entsprechend korerekte Kennwortangaben eingetragen wurden. Der Vorteil besteht darin, daß dieser Schutz untrennbar mit der Verarbeitungshardware verbunden ist, so daß z. B. ein Kopieren der genutzten Software gar keinen Zweck hat, weil gleichartige Ressourcen in anderen Maschinen zu ihrer Nutzung andere Kennwortangaben erwarten.

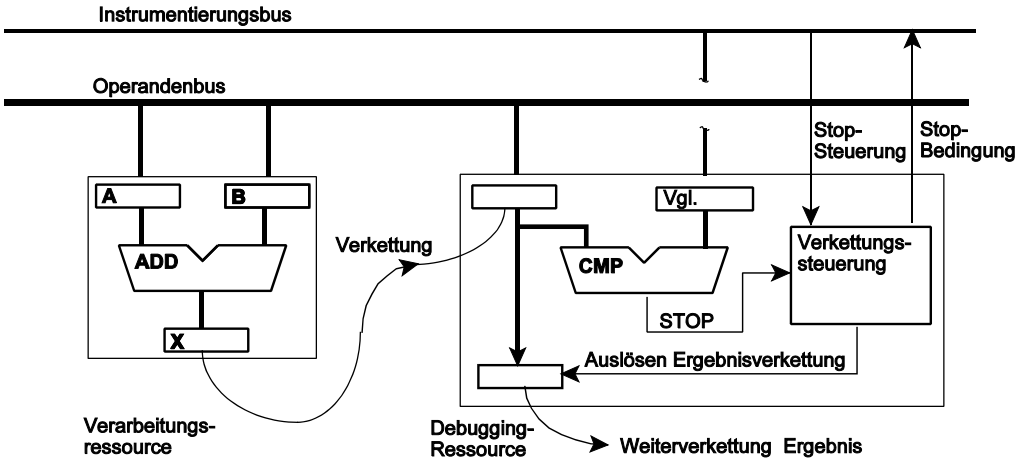


Abb. 6.17 Verarbeitungsressource mit verketteter Debuggingressource

Zusätzlich zu den Verarbeitungsressourcen werden ergänzende Instrumentierungs-Ressourcen bereitgestellt (zum Debugging, zur Leistungsmessung usw.). Diese werden bei Bedarf aufgerufen (s-Operatoren) und mit den Verarbeitungsressourcen verkettet (das eigentliche Anwendungsprogramm ändert sich hierdurch nicht). Im Beispiel ist eine einfache Verarbeitungsressource (ein Addierwerk) mit einer Debugging-Ressoucrce verkettet, die den Wertevergleich unterstützt. Ist das Ergebnis der Verarbeitungsressource gleich dem eingestellten Vergleichswert, so wird eine Stopbedingung ausgelöst. Die Debugging-Ressource ist so ausgebildet, daß sie die zu vergleichende Angabe (hier: das Ergebnis der Verarbeitungsressource) zu den eigentlichen Zielressourcen weiterverketten kann. Ist die Stopbedingung eingetreten, so wird die Weiterverkettung nicht wirksam (bedingte Verkettung). Somit wird der Verarbeitungsablauf angehalten, und es ist auf an sich bekannte Weise (durch Abholen der Registerinhalte) möglich, den aktuellen Verarbeitungszustand auszuwerten. Soll die Verarbeitung fortgesetzt werden, erhält die Verkettungssteuerung in der Debugging-Ressource ein entsprechendes Signal über den Instrumentierungsbus.

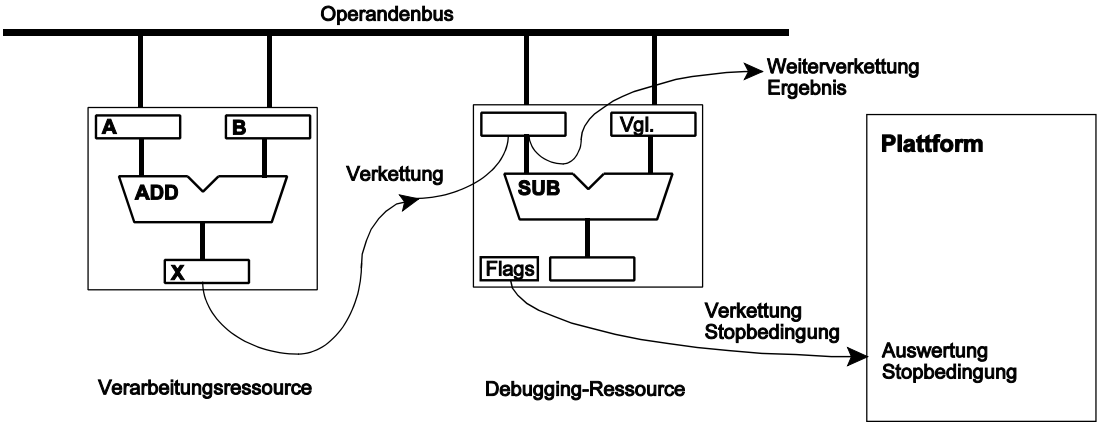


Abb. 6.18 Debugging-Konfiguration mit verketteter Ressource

Es ist möglich, die üblichen Verarbeitungsressourcen zu Instrumentierungszwecken auszunutzen. Auf diese Weise können auch Konfigurationen zum Fehlersuchen (Debugging), zur Leistungsmessung usw. bedarfsweise zusammengeschaltet werden.

Im Beispiel von Abb. 6.18 wird ein Addierwerk (Verarbeitungsressource) mit einem Subtrahierwerk (Debugging-Ressource) verkettet. Das Subtrahierwerk vergleicht das Ergebnis des Addierwerks mit einem eingestellten Wert. Zudem leitet es das ankommende Ergebnis an andere Ressourcen weiter. Die Stopbedingung wird hier durch eine Verkettung zur Plattform signalisiert (sie kann dort z. B. eine Unterbrechung auslösen). Das gewährleistet aber nicht immer ein exaktes Anhalten zum Stopzeitpunkt. Höherentwickelte universelle Ressourcen, die ihren Funktionen nach auch für Instrumentierungszwecke in Betracht kommen, können mit bedingter Operandenverkettung ausgerüstet werden.

7. ReAl-Bytecodes

Bytecodes sind binär codierte maschinenunabhängige Darstellungen von ReAl-Programmen mit unbeschränktem Adressierungsvermögen. Die Auslegung von Bytecodes gehört zum allgemeinen Fachwissen der Informatik. Deshalb kann sich die folgende Darstellung auf ein typisches Ausführungsbeispiel beschränken.

ReAl-Programme bestehen aus Folgen von Bytes. Es gibt Steuerbytes (Abb. 7.1) und numerische Angaben.

7	6	5	4	3	2	1	0
Längenangabe (0...7)			Typangabe				

Abb. 7.1 Grundformat eines Steuerbytes

Numerische Angaben
Das Steuerbyte einthält eine Längenangabe im Bereich von 1 bis 7. Die Längenangabe kennzeichnet, wie lang die nachfolgende numerische Angabe ist. Die Typangabe kennzeichnet die Art der nachfolgenden numerischen Angabe (Tabelle 7.1).

Codierte Längenwerte: 1 Byte, 2 Bytes, 3 Bytes, 4 Bytes, 6 Bytes, 8 Bytes, reserviert.

Operatoren
Die Längenangabe ist 0. Die Typangabe kennzeichnet die Art des Operators (Tabelle 7.2).

Steuerbyte 00H: wirkungslos (NOP).

Typangabe	**entsprechende Operatoren**
Ressourcentyp	s
Ressourcennummer (im Ressourcenvorat)	s
Ressourcenanzahl	s
Variablenart	p, a
Variablennummer	p, a
Bitkette (Direktwert)	p
Quellressource	a, l, c, d
Quellparameter	a, l, c, d
Zielressource	p, l, c, d, y
Zielparameter	p, l, c, d
Funktionscode	h, m, u

Tabelle 7.1 Typangaben (1): numerische Angaben (Ordinalzahlen)

Operator	Wirkung	Operator	Wirkung
NOP	keine	c	ausgewählte Ressourcen über Parameter miteinander verketten (connect)
s	Ressourcen auswählen (select)	d	Verkettung trennen (disconnect)
s_a	eine Ressource auswählen und Nummer oder Adresse zuweisen (select & assign)	l	Parameter zwischen ausgewählten Ressourcen transportieren (link)
p	Parameter aus Speichermitteln der Plattform in die ausgewählten Ressource transportieren	r	Ressourcen zu anderweitiger Verwendung freigeben (release)
p_imm	Direktwert (immediate) in die ausgewählte Ressource transportieren	h	zusätzliche Angaben (Hinweise) zur Unterstützung der Compilierung und zur Ablaufbeschleunigung. Reserviert
y	Verarbeitungsfunktionen in der ausgewählten Ressource aktivieren (yield)	m	zusätzliche metasprachliche Angaben. Reserviert
a	Parameter aus der ausgewählten Ressource in Speichermittel der Plattform transportieren (assign)	u	Dienstfunktionen (utilities). Hier werden alle maschinenspezifischen Codes eingeordnet, die zur Unterstützung der eigentlichen ReAl-Operatoren vorgesehen sind (z. B. zum Laden von Registern der Plattform)

Tabelle 7.2 Typangaben (2): Operatoren

Im Steuerbyte sind zu codieren:

- 11 Arten numerischer Angaben,
- 14 verschiedene Operatoren.

Das 5-Bit-Feld des hier beschriebenen Formats läßt somit noch eine Reserve von 21 bzw. 18 Codepositionen frei (es können noch 21 weitere Arten numerischer Angaben und 18 weitere Operatoren codiert werden).

Hinweise:

1. Manche Operatoren sind in mehreren Varianten zu codieren. Unser Beispiel beschränkt sich auf zwei derartige Operatoren, s_a und p_imm.
2. Die jeweilige Wirkung der Operatoren h, m und u wird in zusätzlichen numerischen Angaben codiert (Funktionscodes).

Die Operatoren müssen durch numerische Angaben ergänzt werden. Es gibt zwei Möglichkeiten der Reihenfolge:

1. Postfixnotation (Abb. 7.2). Erst kommen die numerischen Angaben, dann folgt der Operator. Das interpretierende System hält eine Art Zustandspuffer, der den aktuellen Wert einer jeden Angabe aufnimmt. Beim Fortschreiten von einem Operator zum nächsten sind nur jene Angaben neu einzutragen, die sich jeweils geändert haben.
2. Präfixnotation (Abb. 7.3). Erst kommt der Operator, dann folgen die numerischen Angaben. Deren Anzahl muß der jeweiligen Syntax entsprechen. Das interpretierende System muß eine Akzeptorautomaten haben, der gültige Reihen erkennt. Wurde eine solche Reihe erkannt, so wird die jeweilige Wirkung ausgelöst.

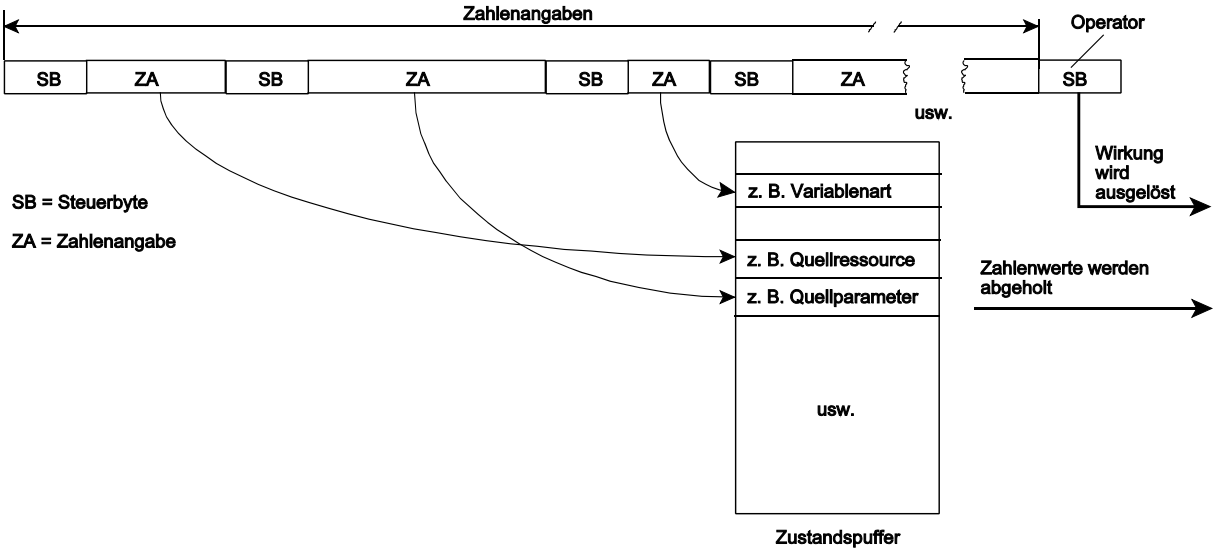


Abb. 7.2 Bytecode in Postfixnotation

Gemäß den Steuerbytes werden die Zahlenangaben in die jeweiligen Positionen des Zustandspuffers eingetragen. Tritt ein Operator auf, so wird die betreffende Wirkung ausgelöst. Dabei werden die jeweils benötigten Angaben aus dem Zustandspuffer geholt.

Im Beispiel hat der Zustandspuffer 11 Einträge, einen für jede Art der numerischen Angaben gemäß Tabelle 7.1. Werden beispielsweise bis zu 8 Bytes lange Parameter zugelassen, so muß jeder Eintrag die 8 Datenbytes aufnehmen können.

In einer ersten Ausführung wird zusätzlich die jeweilige aktuelle Längenangabe (aus dem Steuerbyte) mitgespeichert.

In einer alternativen Ausführung sind alle Parameter im Zustandspuffer beispielsweise 8 Bytes lang. Kürzere Angaben werden rechtsbündig eingefügt. Ein Parameterwert von einem Byte Länge wird in die Bitpositionen 7..0 eingetragen, einer von zwei Bytes Länge in die Bitpositionen 15...0 usw.

In einer weiteren Abwandlung ist der Zustandspuffer als Stack ausgebildet. Numerische Angaben werden auf den Stack gelegt, Operatoren entnehmen ihre Parameter vom Stack. Steuerbytes mit nachfolgenden numerischen Angaben stellen praktisch Push-Befehle dar, Steuerbytes die Operatoren codieren, Operationsbefehle (nach deren Ausführung die Operanden vom Stack entfernt werden).

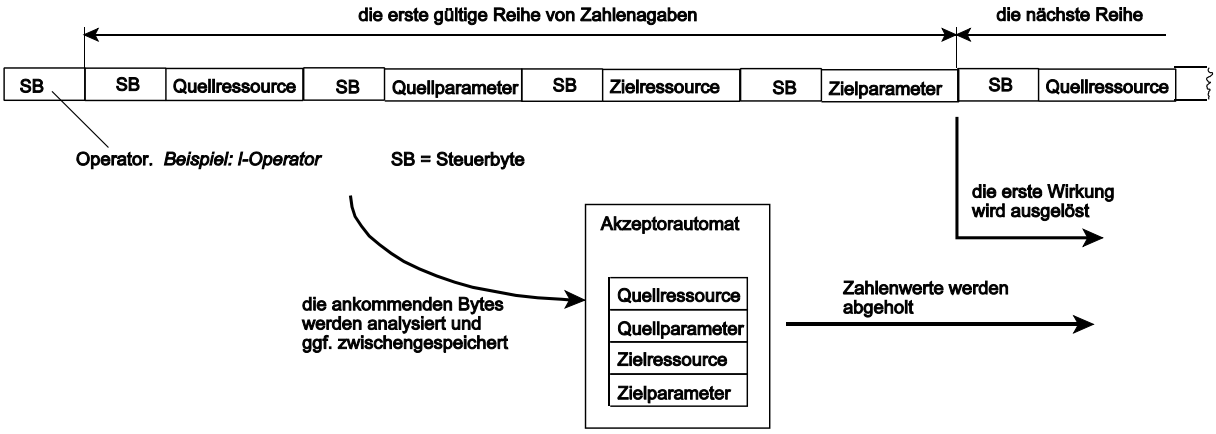


Abb. 7.3 Bytecode in Präfixnotation

Der Akzeptorautomat analysiert den Bytestrom. Jeder Operator ist durch bestimmte zulässige Folgen von Zahlenangaben gekennzeichnet (Tabellen 7.3, 7.4). Die erkannten Zahlenangaben werden zwischen-

gespeichert. Nach Erkennung einer vollständigen zulässigen Folge wird die entsprechende Wirkung ausgelöst. An eine solche Folge darf sich entweder eine weitere zulässige Folge oder ein Operator anschließen.

Ordinalzahl- oder Adreßangaben
Die soeben beschriebenen Auslegungen des Bytecodes können mit beiden Arten von Angaben arbeiten; es ist lediglich eine Frage der Interpretation.

Operator	Syntax der numerischen Angaben
s	Ressourcenanzahl - Ressourcentyp
s_a	Ressourcentyp - Ressourcennummer - Zielressource
p	Variablenart - Variablennummer - Zielressource - Zielparameter
p_imm	Direktwert - Zielressource - Zielparameter
y	Zielressource
a	Quellressource - Quellparameter - Variablenart - Variablennummer
l	Quellressource - Quellparameter - Zielressource - Zielparameter
c	Quellressource - Quellparameter - Zielressource - Zielparameter
d	Quellressource - Quellparameter - Zielressource - Zielparameter
r	Zielressource

Tabelle 7.3 Zulässige Folgen numerischer Angaben (1). Ordinalzahlangaben oder geteilter Ressourcenadreßaum

Operator	Syntax der numerischen Angaben
s	Ressourcentypadresse
s_a	Ressourcentypadresse - Ressourcenadresse
p	Variablenadresse - Zielressourcenadresse
p_imm	Direktwert - Zielressourcenadresse
y	Zielressourcenadresse
a	Quellressourcenadresse - Variablenadresse
l	Quellressourcenadresse - Zielressourcenadresse
c	Quellressourcenadresse - Zielressourcenadresse
d	Quellressourcenadresse - Zielressourcenadresse
r	Zielressourcenadresse

Tabelle 7.4 Zulässige Folgen numerischer Angaben (2). Flacher Ressourcenadreßraum

8. ReAl-Maschinencodes

Der Maschinencode kann als Bytecode (variable Länge) oder als fest formatierter Befehlscode ausgelegt sein. Die Gestaltung von Maschinenbefehlsformaten ist seit langem bekannt. Deshalb genügt es, einige Beispiele kurz zu beschreiben. Diese Beispiele veranschaulichen:

- Befehle verschiedener Länge (16, 32, 64 Bits sowie Bytecodes mit variabler Länge),
- Befehlsformate mit möglichst großem Adressierungsvermögen (= langen Adreßangaben),
- Befehlsformate, die möglichst viele gleichzeitig auslösbare Wirkungen unterstützen,
- Formate mit kurzen und langen Befehlen,
- Befehlsformate mit flachem und mit geteiltem Ressourcenadreßraum,
- die Nutzung von Halteregistern zum Übergeben von Angaben, die nicht in das jeweilige Befehlsformat passen.

Einige wichtige Gesichtspunkte:

- Möglichst lange Adreßangaben,
- ausreichend lange Ressourcentypangaben im s-Operator (8 Bits sind typischerweise die unterste Grenze),
- einfache Decodierung,
- gute Ausnutzung der Befehlslänge,
- Vorkehrungen zum Eintragen elementarer Direktwerte. Hierfür wird eine zusätzliche Variante des p-Operators vorgesehen (p_imm = p immediate).
- Reserven zum Codieren weiterer Befehle.

Ist die Befehlsliste vorzugsweise zur softwareseitigen Emulation vorgesehen, so kommt es auf lange Adreßangaben an, während die gleichzeitige Auslösung mehrerer Funktionen (Parallelarbeit) praktisch bedeutungslos ist (sie kann vom Emulator ohnehin nicht unterstützt werden). Demgegenüber geht es in Befehlslisten für spezielle Hardware (Spezialprozessoren, Verarbeitungseinrichtungen in FPGAs) vor allem um die Parallelverarbeitung. Die Ressourcenadreßangaben müssen lediglich so lang sein, daß sie die tatsächlich vorhandene Hardware unterstützen können. Für universelle Hardware (Mikrocontroller, Hochleistungsprozessoren) ist typischerweise ein Kompromiß zwischen Adreßlänge und gleichzeitig auslösbaren Funktionen zu finden.

8.1 Beispiel 1: Bytecodes

Befehlsformate mit variabler Länge sind in vielen Rechnerarchitekturen üblich. Ein solcher Befehl beginnt mit einem Operationscodebyte, das sowohl die Befehlswirkung als auch die Anzahl der nachfolgenden Bytes bestimmt. Im folgenden Beispiel (Tabelle 8.1, Abb. 8.1) hat – im Gegensatz zu den maschinenunabhängigen Bytecodes (Kapitel 7) – jeder Befehl nur eine einzige Wirkung (beispielsweise müssen, um 5 gleichartige Ressourcen auszuwählen, 5 s-Operatoren gegeben werden).

Hinweise:

1. Befehlsformate gemäß Tabelle 8.1 haben typischerweise genügend Reserven; das Adressierungsvermögen der einzelnen Angaben wird praktisch nie vollständig ausgenutzt werden.
2. Es werden zweckmäßigerweise (Speicherplatzersparnis) mehrere p_imm-Operatoren mit verschieden langen Direktwerten vorgesehen.

Operator	1. Angabe	2. Angabe	3. Angabe	4. Angabe
s	Ressourcentyp			
s_a	Ressourcentyp	Ressource		
p	Variablenadresse	Ressource	Parameter	
p_imm	Direktwert	Ressource	Parameter	
y	Ressource			
a	Ressource	Variablenadresse		
l	Ressource	Parameter	Ressource	Parameter
c	Ressource	Parameter	Ressource	Parameter
d	Ressource	Parameter	Ressource	Parameter
r	Ressource			

Tabelle 8.1 Befehlsformate im Bytecode (geteilter Ressourcenadreßraum). In Befehlen für einen flachen Ressourcenadreßraum entfallen die Parameterangaben. Zu den Befehlswirkungen vgl. Tabelle 7.2

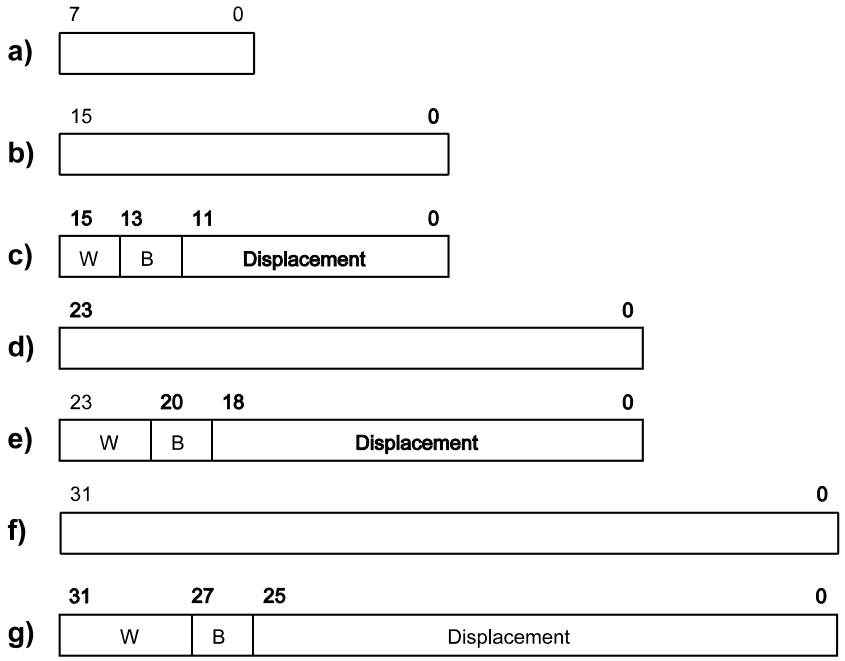


Abb. 8.1 Formate von Bytecodeangaben

a) 1 Byte. Für Operationscodes, Ressourcentypen, Ressourcenadressen, Parameteradressen und Direktwerte.

b) 2 Bytes. Für Ressourcentypen, Ressourcenadressen, Parameteradressen und Direktwerte.

c) Variablenadresse, 2 Bytes. W = Zugriffsbreite, B = Basisadreßregister (vgl. Tabelle 8.3). 4 verschiedene Zugriffsbreiten, 4 Basisadreßregister, 12 Bits Displacement.

d) 3 Bytes. Für Ressourcentypen, Ressourcenadressen, Parameteradressen und Direktwerte.

e) Variablenadresse, 3 Bytes. W = Zugriffsbreite, B = Basisadreßregister (vgl. Tabelle 8.3). 8 verschiedene Zugriffsbreiten, 4 Basisadreßregister, 19 Bits Displacement.

f) 4 Bytes. Für Ressourcentypen, Ressourcenadressen, Parameteradressen und Direktwerte.

g) Variablenadresse, 4 Bytes. W = Zugriffsbreite, B = Basisadreßregister (vgl. Tabelle 8.3). 16 verschiedene Zugriffsbreiten, 4 Basisadreßregister, 26 Bits Displacement.

Die Befehlsformate für eine bestimmte Maschine lassen sich wie aus einem Baukasten zusammenstellen (Tabelle 8.2).

System	Ressourcen-typ	Ressourcen-adresse	Direkt-wert*)	Variablen-adresse	Parameter-adresse
Hardware für Embedded Systems	1	1...2	1, 2	2	1
Hocheistungshardware	1...2	1...2	1, 2, 3, 4	4	1
Software (Emulation) für Embedded Systems	2	2 oder 3	1, 2, 3, 4	2 oder 3	-
Software (Emulation) für den oberen Leistungsbereich	3 oder 4	4	2, 4	3 oder 4	-

*): vgl. den Hinweis 2 am Anfang von Abschnitt 8.1

Tabelle 8.2 Typische Auslegungen von Befehlsformaten im Bytecode

8.2 Beispiel 2: 32 Bits, zwei Wirkungen

Das Beispiel (Abb. 8.2, Tabellen 8.3, 8.4) betrifft ein 32-Bit-Befehlswort und Adreßangaben mittlerer Länge. Jeder Befehl entspricht einem vollständigen Operator. Manche Befehle können zwei Wirkungen auslösen. Der Ressourcenadreßraum umfaßt maximal 4096 Parameter. Die 12 Adreßbits können auch geteilte Ressourcenadressen aufnehmen, z. B. für 1024 Ressuorcen mit 4 Parametern oder 512 Ressourcen mit 8 Parametern. Anwendungsbeispiele: nach ReAl-Prinzipien ausgelegte Hochleistungsprozessoren, Spezialprozessoren, umfangreiche Verarbeitungseinrichtungen in FPGAs usw.

- Befehlslänge: 32 Bits
- Ressourcenadresse: 12 Bits (flacher Adreßraum)
- Ressourcentypangabe: 12 Bits
- Direktwertlänge: 16 Bits
- Variablenadresse (Displacement): 14 Bits
- Festlegung der Zugriffsbreite: im Befehl
- Besonderheiten:
  - der y-Operator kann zwei Ressourcen gleichzeitig aktivieren,
  - der s-Operator kann zwei Ressourcen gleichzeitig auswählen.

Variante: y-Operator mit Funktionscode
Der y-Operator kann nur eine Ressource aktivieren, Die verbleibenden 12 Bits enthalten den zugehörigen Funktionscode (für Ressourcen mit wählbarer Funktion).

W = Zugriffsbreite:	B = Basisadresse:
• 1 Byte	• Stackpointer (SP)
• 2 Bytes	• Base Pointer (BP)
• 4 Bytes	• Global Pointer (GP)
• reserviert	• reserviert

Tabelle 8.3 Codierung von Zugriffsbreite und Basisadresse

Operator	31		29		27		25		23 12	11 0
y	0	0	x	x	0	0	0	x	2. Ressource (12)	1. Ressource (12)
c	0	0	x	x	0	1	0	x	2. Ressource (12)	1. Ressource (12)
d	0	0	x	x	0	1	1	x	2. Ressource (12)	1. Ressource (12)
s	0	0	x	x	1	0	0	x	2. Ressourcentyp (12)	1. Ressourcentyp (12)
r	0	0	x	x	1	0	1	x	2. Ressource (12)	1. Ressource (12)
l	0	0	W		0	0	1	x	2. Ressource (12)	1. Ressource (12)
s_a	0	0	x	x	1	1	1	1	Ressourcenadresse	Ressourcentyp (12)
a	0	1	W		B				Displacement (14)	Ressource (12)
p	1	0	W		B				Displacement (14)	Ressource (12)
p_imm	1	1	W						Immediate (16)	Ressource (12)

Abb. 8.2 ReAl-Maschinencode. 32 Bits mit zwei Wirkungen

Operator	Wirkung
y	Auslösung der Ergebnisberechnung in den angegebenen Ressourcen. Die Adreßangaben betreffen den Funktionscode-Parameter der jeweiligen Ressourcen. Adresse = 0: keine Auslösung
c	Herstellung einer Verkettung 1. Ressource => 2. Ressource. Die Adreßangaben betreffen einen Operanden- und einen Ergebnisparameter
d	Trennung der Verkettung 1. Ressource => 2. Ressource. S. auch unter c.
s	Auswählen (Anfordern) von zwei Ressourcen gemäß Typangabe. Typ = 0: keine Auswahl
r	Freigeben der angegebenen Ressourcen. Die Adreßangaben betreffen den Funktionscode-Parameter der jeweiligen Ressourcen. Adresse = 0: keine Auslösung
l	Parametertransport 1. Ressource => 2. Ressource (Zugriffsbreite W)
s_a	eine Ressource gemäß Typangabe auswählen und die angegebene Ressourcenadresse zuweisen. Die Ressourcenadresse bildet den Anfang für die Ressourcennumerierung in weiteren s-Operatoren (+1-Zählung). Typ = 0: keine Auswahl (nur Einstellen der Anfangsadresse)
a	Ergebnistransport Ressource => Parameteradresse (<Basis B + Displacement>, Zugriffsbreite W)
p	Operandentransport Parameteradresse (<Basis B + Displacement>, Zugriffsbreite W) => Ressource
p_imm	Direktwert-Parametertransport Immediate => Ressource. Direktwert wird gemäß Zugriffsbreite W vorzeichengerecht erweitert bzw. abgeschnitten

Tabelle 8.4 Befehlswirkungen

8.3 Beispiel 3: 32 Bits, lange Adressen

Das Beispiel (Abb. 8.3, Tabellen 8.5 bis 8.7) betrifft ein 32-Bit-Befehlswort und extrem lange Adreßangaben (28 Bits). Anwendung: vor allem für die softwareseitige Emulation (virtuelle ReAl-Maschinen) in oberen Leistungsbereichen. Es ist nicht möglich, zwei Angaben in einem 32-Bit-Wort unterzubringen. Deshalb werden in der Plattform vier Halteregister vorgesehen, die mit u-Operatoren geladen werden können (Tabelle 8.5). Manche Operatoren erfordern deshalb zwei Befehle. Der Ressourcenadreßraum umfaßt maximal 256 M Parameter. Die 28 Adreßbits können auch geteilte Ressourcenadressen aufnehmen, z. B. für 16 M Ressourcen mit 16 Parametern oder für 1 M Ressuorcen mit 256 Parametern.

- Befehlslänge: 32 Bits
- Ressourcenadresse: 28 Bits (flacher Adreßraum)
- Ressourcentypangabe: 26 Bits
- Direktwertlänge: 26 Bits
- Variablenadresse (Displacement): 24 Bits
- Festlegung der Zugriffsbreite: im Befehl
- Zwischenpufferung: 4 Halteregister

Halteregister	geladen mit	genutzt von
1: Variablenadresse	u_va	p
2: Direktwert	u_imm	p_imm
3: Ressourcenadresse	u_rs	l, c, d, a
4: Ressourcenadreßzähler	u_ra	s

Tabelle 8.5 Halteregister

Hinweis:
Die Anordnung von Halteregistern ermöglicht – im Gegensatz zur Verdoppelung der Befehlslänge – oftmals eine Mehrfachnutzung der eingetragenen Angaben:

- Ein Direktwert zu mehreren Ressourcen (p_imm),
- eine Variable zu mehreren Ressourcen (p),
- ein Ergebnis zu mehreren Variablen (a),
- ein Ergebnis zu mehreren Operanden (l, c, d).

Operator	31		29		27	25	23 0
andere	0	0	0	0	*)		res.
s	0	0	0	0	11		Ressourcentyp (26)
r	0	0	0	1			Ressource (28)
p_imm2	0	0	1	0			Ressource (28)
y	0	0	1	1			Ressource (28)
l_2	0	1	0	0			Ressource (28)
c_2	0	1	0	1			Ressource (28)
d_2	0	1	1	0			Ressource (28)
u_rs	0	1	1	1			Ressource (28)
a_2	1	0	0	W		B	Displacement (24)
u_va	1	0	1	W		B	Displacement (24)
u_imm	1	1	0	W			Immediate (26)
p_2	1	1	1	0			Ressource (28)
u_ra	1	1	1	1			Ressourcenadresse (28)

*): Codes 00, 01, 10

Abb. 8.3 ReAl-Maschinencode. 32 Bits mit langen Adressen

W = Zugriffsbreite:	B = Basisadresse:
• 1 Byte • 2 Bytes • 4 Bytes • 8 Bytes • 16 Bytes • reserviert	• Stackpointer (SP) • Base Pointer (BP) • Global Pointer (GP) • reserviert

Tabelle 8.6 Codierung von Zugriffsbreite und Basisadresse

Operator	Wirkung
s	Auswählen (Anfordern) einer Ressourcen gemäß Typangabe. Zuweisung der Ressourcenadresse gemäß Ressourcenadreßzähler (Halteregister 4)
r	Freigeben der angegebenen Ressource
y	Auslösung der Ergebnisberechnung in der angegebenen Ressource
l_2	Parametertransport 1. Ressource => 2. Ressource (Zugriffsbreite W). 1. Parameter gemäß Halteregister 3, 2. Parameter gemäß Adreßangabe. Vollständiger l-Operator: u_rs (1. Ressource); l_2 (2. Ressource)
c_2	Herstellung einer Verkettung 1. Ressource => 2. Ressource. 1. Parameter gemäß Halteregister 3, 2. Parameter gemäß Adreßangabe. Vollständiger c-Operator: u_rs (1. Ressource); c_2 (2. Ressource)
d_2	Trennung der Verkettung 1. Ressource => 2. Ressource. 1. Parameter gemäß Halteregister 3, 2. Parameter gemäß Adreßangabe. Vollständiger d-Operator: u_rs (1. Ressource); d_2 (2. Ressource)
u_rs	Laden der Adreßangabe ins Halteregister 2 (Ressourcenadresse)
a_2	Ergebnistransport Ressource => Variablenadresse (<Basis B + Displacement>, Zugriffsbreite W). Parameter gemäß Halteregister 2. Vollständiger a-Operator: u_rs (Ressource); a_2 (Parameteradresse)
pimm_2	Direktwert-Parametertransport Immediate => Ressource. Direktwert gemäß Halteregister 2. Direktwert wird gemäß Zugriffsbreite W vorzeichengerecht erweitert. Vollständiger p_imm-Operator: u_imm (Direktwert); p_imm2 (Ressource)
u_va	Laden der Variablenadresse ins Halteregister 1
u_imm	Laden des Direktwertes ins Halteregister 2
p_2	Operandentransport Variablenadresse (<Basis B + Displacement>, Zugriffsbreite W) => Ressource. Variablenadresse gemäß Halteregister 1. Vollständiger p-Operator: u_va (Variablenadresse); p_2 (Ressource)
u_ra	Ressourcenadresse für s-Operator einstellen (Halteregister 4). Die Ressourcenadresse bildet den Anfang für die Ressourcennumerierung in nachfolgenden s-Operatoren

Tabelle 8.7 Befehlswirkungen

8.4 Beispiel 4: 16 Bits, zwei Wirkungen

Es handelt sich um eine Abwandlung des Beispiels 2 mit kurzen Befehlen (16 Bits) und zwei Ressourcenangaben im Befehl (Abb. 8.4, Tabelle 8.8). Die einzelne Ressourcenadresse ist auf 6 Bits beschränkt. Hiermit können z. B. 16 Ressourcen mit jeweils maximal 4 Parametern oder 8 Ressourcen mit jeweils 8 Parametern unterstützt werden. Anwendungsbeispiele: nach ReAl-Prinzipien ausgelegte Mikrocontroller, kleinere Spezialprozessoren, Verarbeitungseinrichtungen in FPGAs usw. Bei Beschränkung auf 16 Bits werden die Displacementangaben der Variablenadressen und die Direktwerte sehr kurz. Deshalb gibt es von den p-, p_imm- und a-Operatoren jeweils eine Langversion, die das nachfolgende 16-Bit-Wort einschließt.

- Befehlslänge: 16 bzw. 32 Bits
- Ressourcenadresse: 6 Bits (flacher Adreßraum)
- Ressourcentypangabe: 12 Bits
- Direktwertlänge: 6 oder 20 Bits
- Variablenadresse (Displacement): 4 oder 18 Bits
- Festlegung der Zugriffsbreite: in den Ressourcen (16-Bit-Befehle) oder im Befehl (32-Bit-Befehle)
- Besonderheit: der y-Operator kann zwei Ressourcen gleichzeitig aktivieren.

Operator	15		12		11 6	5	0	15	**2. 16-Bit-Wort** 0
p_l	0	0	1	0	Ressource	B	Displace-ment	W	Displacement
p_s	0	0	1	1	Ressource	Displacement			
p_imml	0	1	0	0	Ressource	Immediate		W	Immediate (14)
p_imms	0	1	0	1	Ressource	Immediate			
a_l	0	1	1	0	Ressource	B	Displace-ment	W	Displacement
a_s	0	1	1	1	Ressource	Displacement			
l	1	0	0	0	Ressource	Ressource			
c	1	0	0	1	Ressource	Ressource			
d	1	0	1	0	Ressource	Ressource			
r	1	0	1	1	Ressource	Ressource			
y	1	1	0	0	Ressource	Ressource			
s	1	1	0	1	Ressourcentyp				
s_a	1	1	1	1	Ressourcentyp			Ressource	

Abb. 8.4 ReAl-Maschinencode. 16 Bits mit zwei Wirkungen. Zu den Angaben W und B vgl. Tabelle 8.3

Operator	**Wirkung**
p_l	Operandentransport Parameteradresse (<Basis B + Displacement>, Zugriffsbreite W) => Ressource. Lange Displacementadresse (18 Bits)
p_s	Operandentransport Parameteradresse (<Basis B + Displacement>, Zugriffsbreite W) => Ressource. Kurze Displacementadresse (4 Bits)
p_imml	Direktwert-Parametertransport Immediate => Ressource. Langer Direktwert (20 Bits). Direktwert wird gemäß Zugriffsbreite W vorzeichengerecht erweitert
p_imms	Direktwert-Parametertransport Immediate => Ressource. Kurzer Direktwert (6 Bits). Direktwert wird gemäß Zugriffsbreite W vorzeichengerecht erweitert
a_l	Ergebnistransport Ressource => Parameteradresse (<Basis B + Displacement>, Zugriffsbreite W). Lange Displacementadresse (18 Bits)
a_s	Ergebnistransport Ressource => Parameteradresse (<Basis B + Displacement>, Zugriffsbreite W). Kurze Displacementadresse (4 Bits)
l	Parametertransport 1. Ressource => 2. Ressource (Zugriffsbreite W)
c	Herstellung einer Verkettung 1. Ressource => 2. Ressource. Die Adreßangaben betreffen einen Operanden- und einen Ergebnisparameter

Operator	Wirkung
d	Trennung der Verkettung 1. Ressource => 2. Ressource. S. auch unter c.
r	Freigeben der angegebenen Ressourcen. Die Adreßangaben betreffen den Funktionscode-Parameter der jeweiligen Ressourcen. Adresse = 0: keine Auslösung
y	Auslösung der Ergebnisberechnung in den angegebenen Ressourcen. Die Adreßangaben betreffen den Funktionscode-Parameter der jeweiligen Ressourcen. Adresse = 0: keine Auslösung
s	Auswählen (Anfordern) einer Ressource gemäß Typangabe
c	Herstellung einer Verkettung 1. Ressource => 2. Ressource. Die Adreßangaben betreffen einen Operanden- und einen Ergebnisparameter
d	Trennung der Verkettung 1. Ressource => 2. Ressource. S. auch unter c
s_a	eine Ressource gemäß Typangabe auswählen und die angegebene Ressourcenadresse zuweisen. Die Ressourcenadresse bildet den Anfang für die Ressourcennumerierung in weiteren s-Operatoren (+1-Zählung). Typ = 0: keine Auswahl (nur Einstellen der Anfangsadresse)

Tabelle 8.8 Beispiel 3: Befehlswirkungen

8.5 Beispiel 5: 16 Bits, längere Adressen

Es handelt sich um eine Abwandlung des Beispiels 3 mit kurzen Befehlen (16 Bits) und entsprechend kürzeren, aber durchaus brauchbare Adreßangaben (Abb. 8.5). Mit einer Ressourcenadresse von 12 Bits können z. B. 512 Ressourcen mit jeweils maximal 8 Parametern unterstützt werden. Es ist nicht möglich, zwei Adreßangaben in einem Befehl unterzubringen. Deshalb werden in der Plattform vier Halteregister vorgesehen, die mit u-Operatoren geladen werden können (vgl. Tabelle 8.5). Manche Operatoren erfordern deshalb zwei Befehle. Anwendungsbeispiele: softwareseitige Emulation (virtuelle ReAl-Maschinen), vorzugsweise im Bereich der Embedded Systems, nach ReAl-Prinzipien ausgelegte Prozessoren usw.

- Befehlslänge: 16 Bits
- Ressourcenadresse: 12 Bits (flacher Adreßraum)
- Ressourcentypangabe: 10 Bits
- Direktwertlänge: 11...13 Bits
- Variablenadresse (Displacement): 9 Bits
- Festlegung der Zugriffsbreite: im Operator
- Zwischenpufferung: 4 Halteregister mit 12 Bits

Operator	15		13		11	9	8	0
andere	0	0	0	0	*)	res.		
s					11	Ressourcentyp (10)		
r	0	0	0	1	Ressource (12)			
p_imm2	0	0	1	0	Ressource (12)			
y	0	0	1	1	Ressource (12)			
l_2	0	1	0	0	Ressource (12)			
c_2	0	1	0	1	Ressource (12)			
d_2	0	1	1	0	Ressource (12)			
u_rs	0	1	1	1	Ressource (12)			
a_2	1	0	0	W	B	Displacement (9)		
u_va	1	0	1	W	B	Displacement (9)		

Operator	15		13	11 9	8 0
u_imm	1	1	0	W	Immediate (11)
p_2	1	1	1	0	Ressource (12)
u_ra	1	1	1	1	Ressourcenadresse (12)

*): Codes 00, 01, 10

Abb. 8.5 ReAl-Maschinencode. 16 Bits mit längeren Adressen. S. weiterhin die Tabellen 8.5 bis 8.7

8.6 Beispiel 6: kurze Befehle mit parallel auslösbaren Funktionen

Dieses Beispiel betrifft kurze Befehle, in denen möglichst viele parallel ausführbare Funktionen codiert sind (Abb. 8.6, 8.7, Tabelle 8.9). Es werden bis zu 64 Ressourcen mit jeweils maximal 8 Parametern unterstützt (geteilter Adreßraum). Die Angaben für die einzelnen Operatoren werden abschnittsweise zugeführt. Um die endgültigen Angaben bereitzustellen, werden in der Plattform zwei Halteregister angeordnet. Zum Laden der Halteregister sind zusätzliche Operatoren u_rs1, u_rs2 vorgesehen. Der y-Operator kann bis zu acht Ressourcen gleichzeitig aktivieren (über Bitmaske). Anwendungsbeispiele: nach ReAl-Prinzipien ausgelegte Mikrocontroller, universelle Prozessoren, Spezialprozessoren, Verarbeitungseinrichtungen in FPGAs usw. Dieses Befehlsformat unterstützt vor allem die Parallelarbeit zwischen den Ressourcen (gleichzeitige Operationen, Parametertransporte usw.).

- Befehlslänge: 16 Bits
- Ressourcenadresse: 6 Bits (geteilter Adreßraum)
- Parameteradresse: 3 Bits
- Ressourcentypangabe: 10 Bits
- Direktwertlänge: 9 Bits
- Variablenadresse (Displacement): 9 Bits
- Festlegung der Zugriffsbreite: in den Ressourcen
- gleichzeitig aulösbar sind:
  - 2 Parametertransporte zwischen den Ressourcen
    oder
  - 2 Verkettungssteuerfunktionen
    oder
  - die Aktivierung von maximal 8 Ressourcen
    oder
  - das Eintragen von 2 Ressourcenadressen in ein Halteregister

	11 6	5 0
Halteregister 1	1. Quellressource	1. Zielressource
Halteregister 2	2. Quellressource	2. Zielressource

Abb. 8.6 Belegung der Halteregister

Operator	15		12	11 3									2 0
p	1	0	B	Displacement									Parameter
a	1	1	B	Displacement									Parameter
y	0	0	0	0	0	8.	7.	6.	5.	4.	3.	2.	1.	Ress.sel.
andere	0	0	0	0	1	0	0	res.					
s	0	0	0	0	1	0	1	Ressourcentyp (9)					
u_ra	0	0	0	0	1	1	Ressourcenadresse (6)						

Operator	15			12	11			3	2	0
l	0	0	0	1	1. Quellp.	1. Zielp.	2. Quellp.		2. Zielp.	
c	0	0	1	0	1. Quellp.	1. Zielp.	2. Quellp.		2. Zielp.	
d	0	0	1	1	1. Quellp.	1. Zielp.	2. Quellp.		2. Zielp.	
r	0	1	0	0	1. Ressource		2. Ressource			
u_rs1	0	1	0	1	1. Quellressource		1. Zielressource			
u_rs2	0	1	1	0	2. Quellressource		2. Zielressource			
p_imm	0	1	1	1	Immediate (9)				Parameter	

Abb. 8.7 ReAl-Maschinencode. Kurze Befehle mit parallel auslösbaren Funktionen. Zur Basisadreßangabe B vgl. Tabelle 8.3

Operator	**Wirkung**
p	Operandentransport Parameteradresse (<Basis B + Displacement>) => Parameter in Ressource. Die Ressourcenadresse befindet sich im 1. Halteregister (1. Zielressource)
a	Ergebnistransport Ressource => Parameteradresse (<Basis B + Displacement>). Die Ressourcenadresse befindet sich im 1. Halteregister (1. Quellressource)
y	Auslösung der Ergebnisberechnung in den angegebenen Ressourcen. Das Feld *Ress.sel.* wählt aus den ingesamt 64 unterstützen Ressourcen eine Gruppe von 8 aus, die über Einzelbits in beliebiger Kombination aktiviert werden können
s	Auswählen (Anfordern) einer Ressource gemäß Typangabe
u_ra	Ressourcenadressen für s-Operator einstellen. Die Ressourcenadresse bildet den Anfang für die Ressourcennumerierung in weiteren s-Operatoren (+1-Zählung)
l	Parametertransporte 1. Quellparameter => 1. Zielparameter, 2. Quellparameter => 2. Zielparameter. Ressourcenadressen für den ersten Transport in Halteregister 1, für den zweiten Transport in Halteregister 2. Quellparameter = 0: kein Transport
c	Herstellung zweier Verkettungen 1. Quellparameter => 1. Zielparameter, 2. Quellparameter => 2. Zielparameter. Ressourcenadressen für die erste Verkettung in Halteregister 1, für die zweite Verkettung in Halteregister 2. Quellparameter = 0: keine Verkettung
d	Trennung zweier Verkettungen 1. Quellparameter => 1. Zielparameter, 2. Quellparameter => 2. Zielparameter. Ressourcenadressen für die erste Verkettung in Halteregister 1, für die zweite Verkettung in Halteregister 2. Quellparameter = 0: keine Wirkung
r	Freigeben der angegebenen zwei Ressourcen
u_rs1	Laden des Halteregisters 1 mit zwei Ressourcenadressen
u_rs2	Laden des Halteregisters 2 mit zwei Ressourcenadressen
p_imm	Direktwert-Parametertransport Immediate => Parameteradresse. Die Ressourcenadresse befindet sich im 1. Halteregister (1. Zielressource)

Tabelle 8.9 Beispiel 5: Befehlswirkungen

8.7 Beispiel 7: 64 Bits, viele parallel auslösbare Funktionen

In diesem Beispiel (Abb.n 8.8...8.11, Tabellen 8.10, 8.11) geht es darum, sehr viele parallel ausführbare Funktionen zu codieren. Hierzu sind längere Befehle erforderlich (64 Bits). Wie in Beispiel 6 werden bis zu 64 Ressourcen mit jeweils maximal 8 Parametern unterstützt (geteilter Adreßraum). Die Angaben für die einzelnen Operatoren werden abschnittsweise zugeführt. Um die endgültigen Angaben bereitzustellen, werden in der Plattform zwei Halteregister angeordnet. Zum Laden der Halteregister sind

zusätzliche Operatoren u_rs1, u_rs2 vorgesehen. Der y-Operator kann bis zu 60 Ressourcen gleichzeitig aktivieren (über Bitmaske). Anwendungsbeispiele: Spezialprozessoren, Verarbeitungseinrichtungen in FPGAs usw.

- Befehlslänge: 64 Bits
- Ressourcenadresse: 6 Bits (geteilter Adreßraum)
- Parameteradresse: 3 Bits
- Ressourcentypangabe: 10 Bits
- Direktwertlänge: 12 Bits
- Variablenadresse (Displacement): 10 Bits
- Festlegung der Zugriffsbreite: in den Ressourcen
- gleichzeitig auslösbar sind:
  - 10 Parametertransporte zwischen den Ressourcen
    oder
  - 10 Verkettungsteuerfunktionen
    oder
  - 4 Parametertransporte zwischen Ressourcen und Plattform
    oder
  - die Aktivierung von maximal 60 Ressourcen
    oder
  - das Eintragen von 10 Ressourcenadressen in ein Halteregister

63 60	59 0
Opcode	Adreß-, Direktwert- oder Steuerangaben

Abb. 8.8 ReAl-Maschinencode. 64 Bits mit vielen parallel aulösbaren Funktionen. Das grundsätzliche Befehlsformat

Opcode	Format	Opcode	Format	Opcode	Format	Opcode	Format
0	y_1	4	res.	8	l	C	r
1	y_2	5	res.	9	p	D	s
2	u_ra	6	c	A	a	E	u_rs2
3	andere	7	d	B	p_imm	F	u_rs1

Abb. 8.9 Operationscodes

Operator	Belegung der Bits 59...0									
l, c, d	s/d1	s/d2	s/d3	s/d4	s/d5	s/d6	s/d7	s/d8	s/d9	s/d10
r	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10
u_1	s1	d1	s2	d2	s3	d3	s4	d4	s5	d5
u_2	s6	d6	s7	d7	s8	d8	s9	d9	s10	d10
y	60 Einzelbits									
p, p_imm,a	parm1		parm2		parm3		parm4			
s	rt1	rt2	rt3	rt4	rt5	rt6				
u_ra	ra 1 (6)	ra 2 (6)	ra 3 (6)	ra 4 (6)	ra 5 (6)	ra 6 (6)				

Abb. 8.10 Übersicht über die einzelnen Befehlsformate. S. weiterhin Tabelle 8.9

Operation	14 12	11 3	2 0
p, a	B	Displacement (10)	Parameter
pimm	Immediate (12)		Parameter

Abb. 8.11 Parameterangabe in den Operatoren p, pimm und a. Zur Basisadreßangabe B vgl. Tabelle 8.3

Halteregister 1	s1	d1	s2	d2	s3	d3	s4	d4	s5	d5
Halteregister 2	s6	d6	s7	d7	s8	d8	s9	d9	s10	d10

Abb. 8.12 Belegung der Halteregister

Operator	**Belegung der Befehlsbits 59...0**
p	4 Parameterangaben (parm1...4) zu 15 Bits; vgl. Abb. 8.10
a	4 Parameterangaben (parm1...4) zu 15 Bits; vgl. Abb. 8.10
y_1	60 Einzelbits zur Aktivierung der Ressourcen 59...0
y_2	60 Einzelbits zur Aktivierung der Ressourcen 63...60 und 55...0
s	6 Ressourcentypangaben (rt1...10) zu 10 Bits
l, c, d	10 Parameteradreßpaare (s/d1...s/d10) (zu 6 Bits. s = Quellparameter (3 Bits), d = Zielparameter (3 Bits)
r	10 Ressourcenadresen (r1...r10) zu 6 Bits
u_rs1	Laden des Halteregisters 1 mit 5 Ressourcenadreßpaaren (s1, d1...s5, d5) zu 12 Bits; vgl. Abb. 8.11
u_rs2	Laden des Halteregisters 2 mit 5 Ressourcenadreßpaaren (s6, d6...s10, d10) zu 12 Bits; vgl. Abb. 8.11
p_imm	4 Parameterangaben (parm1...4) zu 15 Bits; vgl. Abb. 8.10

Tabelle 8.10 Inhalte der Befehlsformate

Operator	**Wirkung**
p	4 Operandentransporte Parameteradresse (<Basis B + Displacement>) => Parameter in Ressource. Die Ressourcenadressen befinden sich im 1. Halteregister (Zielressourcen d1...d4). Parameter = 7: kein Transport
a	4 Ergebnistransporte Ressource => Parameteradresse (<Basis B + Displacement>). Die Ressourcenadressen befinden sich im 1. Halteregister (Quelllressourcen s1...s4). Parameter = 7: kein Transport
y_1	Auslösung der Ergebnisberechnung in den ersten 60 Ressourcen (59...0).
y_2	Auslösung der Ergebnisberechnung in den letzten 4 und den ersten 56 Ressourcen (63...60, 55...0).
s	Auswählen (Anfordern) von 6 Ressourcen gemäß Typangabe (rt1...6). Ressourcentyp = 0. keine Auswahl
u_ra	Ressourcenadressen für s-Operator einstellen. Die 6. Ressourcenadresse bildet den Anfang für die Ressourcennumerierung in weiteren s-Operatoren (+1-Zählung)
l	10 Parametertransporte Quellparameter => Zielparameter (s1 => d1...s10 => d10). Ressourcenadressen für die ersten 5 Transportvorgänge (1...5) in Halteregister 1, für die zweiten 5 Transportvorgänge(6...10) in Halteregister 2. Quellparameter = 0: kein Transport

Operator	Wirkung
c	Herstellen von 10 Verkettungen Quellparameter => Zielparameter (s1 => d1...s10 => d10). Ressourcenadressen für die ersten 5 Verkettungen (1...5) in Halteregister 1, für die zweiten 5 Verkettungen (6...10) in Halteregister 2. Quellparameter = 0: keine Verkettung
d	Trennung von 10 Verkettungen Quellparameter => Zielparameter (s1 => d1...s10 => d10). Ressourcenadressen für die ersten 5 Verkettungen (1...5) in Halteregister 1, für die zweiten 5 Verkettungen (6...10) in Halteregister 2. Quellparameter = 0: keine Wirkung
r	Freigeben der angegebenen 10 Ressourcen (r1...r10)
u_rs1	Laden des Halteregisters 1 mit 10 Ressourcenadressen zu 6 Bits (s1, d1...s5, d5)
u_rs2	Laden des Halteregisters 2 mit 10 Ressourcenadressen zu 6 Bits (s6, d6...s10, d10)
p_imm	4 Direktwert-Parametertransporte Immediate => Parameteradresse. Die Ressourcenadressen befinden sich im 1. Halteregister (Zielressourcen d1...d4). Parameter = 7: kein Transport

Tabelle 8.11 Beispiel 6: Befehlswirkungen

9. ReAl-Maschinen

ReAl-Systeme können mit herkömmlichen Prozessoren aufgebaut werden. Man kann z. B. einen Prozessor als Plattform einsetzen und durch weitere Prozessoren ergänzen, die als Verarbeitungsressourcen genutzt werden. Die Prozessoren werden üblicherweise über ein Bussystem oder über Punkt-zu-Punkt-Interfaces und Koppeleinrichtungen (Schaltverteiler, Hubs) miteinander verbunden (Abb.n 9.1, 9.2). Sie können auf einen gemeinsamen Speicheradreßraum zugreifen.

Die Plattform baut die Ressourcen-Arbeitsbereiche im Speicheradreßraum auf und veranlaßt die Prozessoren, die entsprechenden Funktionen auszuführen. Das kann z. B. durch Unterbrechungsauslösung bewirkt werden. Diese Lösung funktioniert mit herkömmlicher Hardware, hat aber den Nachteil vergleichsweise langer Latenzzeiten. Hierzu tragen vor allem bei:

- Die Interruptauslösung (mit der erforderlichen Kontextumschaltung),
- das Füllen der Caches (nach dem Schreiben in den Speicheradreßraum müssen die Caches der Prozessoren zunächst neu gefüllt werden).

Eine Abhilfe besteht darin, die Caches der Prozessoren für Schreibzugriffe von außen zugänglich zu machen. Das erfordert entsprechende Änderungen an den Prozessorschnittstellen. Es muß möglich sein, die Prozessoren als Targets anzusprechen, wobei die Zugriffsadressen die internen Caches und Pufferspeicher betreffen. Hierfür ist nur die Busanschlußsteuerung abzuwandeln (die anderen Schaltungen im Prozessor bleiben, wie sie sind).

Noch höhere Verarbeitungsleitungen lassen sich erreichen, wenn der Prozessor auch im Innern als ReAl-Maschine ausgelegt wird, die die ReAl-Befehle nicht nur mit herkömmlichen Maschinenbefehlen emuliert, sondern direkt ausführt. Hierzu können vorhandene Prozessorstrukturen ausgenutzt werden (vgl. die Abb.n 1.2 und 1.3). Befehlsdecodierung und Mikroprogrammsteuerung sind zu änderrn, Verarbeitungswerke, Anschlußsteuerungen, Caches, Puffer usw. bleiben im wesentlichen erhalten.

Die Alternative besteht darin, die Hardware von Grund auf aus elementaren Verarbeitungsressourcen aufzubauen. Im folgenden Beispiel ist die elementare Ressource eine Arithmetik-Logik-Einheit, die mit Registerspeichern und einem einfachen Steuerwerk (Sequencer) verbunden ist (Abb. 9.3). Der Funktionsumfang entspricht dem im Bereich der Hochleistungsprozessoren üblichen Stand der Technik (Tabelle 9.1). Die Datenstrukturen sind ganze Binärzahlen und Bitfelder. Die Unterstützung von Einzelbits, Zeichenketten, Gleitkommazahlen usw. ist aus Umfangsgründen nicht dargestellt.

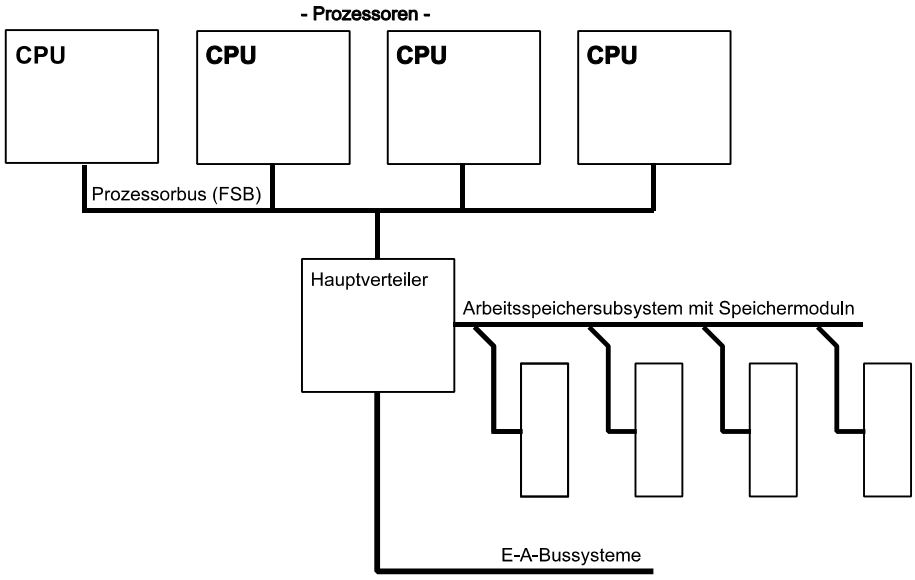


Abb. 9.1 Über ein Bussystem verbundene Prozessoren

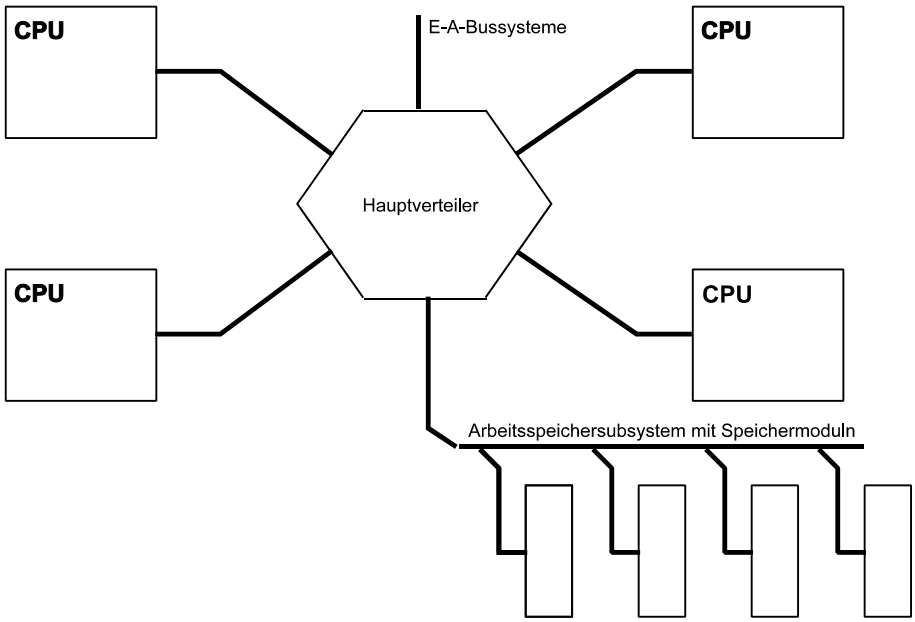


Abb. 9.2 Über Schaltverteiler verbundene Prozessoren

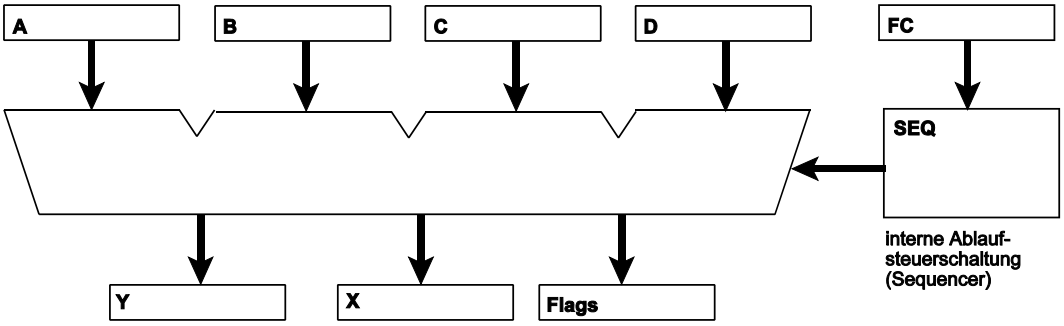


Abb. 9.3 Eine elementare Verarbeitungsressource

Ressource	Eingangsparameter		Ausgangsparameter		Funktion
ADD	2	A, B	2	X, Flags	X := A + B
SUB	2	A, B	2	X, Flags	X := A - B
MUL	2	A, B	3	X, Y, Flags	Y|X := A * B
DIV	3	A, B, C	3	X, Y, Flags	X := A|B : C, Y := A|B mod C (Rest)
NEG	1	A	2	X, Flags	X := A * (-1) (Zweierkomplement)
AND	2	A, B	2	X, Flags	X := A and B
OR	2	A, B	2	X, Flags	X := A or B
XOR	2	A, B	2	X, Flags	X := A xor B
NOT	1	A	2	X, Flags	X := not A
SHL	3	A, B, C	3	X, Y, Flags	Y|X := A nach links verschoben um C Bits, aufgefüllt mit B
SHR	3	A, B, C	3	X, Y, Flags	Y|X := A nach rechts verschoben um C Bits, aufgefüllt mit B
SHRA	2	A, B	3	X, Y, Flags	Y|X := A arithmetiesch nach rechts verschoben um B Bits
ROTL	2	A, B	2	X, Flags	X := A nach rechts rotiert um B Bits
ROTR	2	A, B	2	X, Flags	X := A nach links rotiert um B Bits
EXTRACT	3	A, B, C	2	X, Flags	X := Bitfeld aus A, beginnend an Bit B, Länge C
DEPOSIT	4	A, B, C, D	2	X, Flags	X := A mit eingefügtem Bitfeld B. Einfügung beginnend an Bit C, Länge D
FIRSTOCC	1	A	3	X, Y, Flags	X := Bitvektor mit höchstwertiger 1 aus A (sonst Nullen), Y := Bitadresse der höchstwertigen Eins
LASTOCC	1	A	3	X, Y, Flags	X := Bitvektor mit niedrigstwertiger 1 aus A (sonst Nullen), Y := Bitadresse der niedrigstwertigen Eins
NOOCCS	1	A	2	X, Flags	X := Anzahl der Einsen in A
LOAD_A	3	A, B, C	2	X, Flags	Laden X gemäß <A + B)>; B := B + C (Autoincrement)
STORE_A	4	A, B,C, D	1	Flags	Speichern D gemäß <A + B)>; B := B + C (Autoincrement)
LOAD_X	2	A, B, C	2	X, Flags	Laden X gemäß <A + (B * C)>; B := B + 1 (Autoincrement)
STORE_X	2	A, B, C, D	1	Flags	Speichern D gemäß <A + (B * C)>; B := B + 1 (Autoincrement)
FORLOOP	3	A, B, C	2	X, Flags	Anfänglich: X := A; dann X := X + B, solange X $\leq$ C

Tabelle 9.1 Der Funktionsumfang einer elementaren Verarbeitungsressource

Anmerkungen:

1. Y|X und A|B = höherwertiges Wort in Y oder A, niederwertiges Wort in X oder B.
2. Ladefunktionen (LOAD-A/X) stellen ein Ergebnis in X bereit, das an die eigentlichen Verarbeitungsressourcen geliefert wird (Verkettung).
3. Speicherfunktionen (STORE-A/X) speichern einen in D abgelegten Wert, der von den eigentlichen Verarbeitungsressourcen dort abgeliefert wurde (Verkettung).
4. LOAD_A, STORE_A = Laden und Speichern gemäß Adreßrechnung Basis + Displacement mit nachträglicher Erhöhung der Displacementangabe. Basis in A, Displacement in B, Erhöhung in C. Unterstützt Zugriffe zu aufeinanderfolgenden gleich langen Datenstrukturen der Länge C.
5. LOAD_X, STORE_X= Laden und Speichern gemäß Indexangabe in B mit nachträglicher Erhöhung des Indexwertes um 1. Unterstützt Zugriffe zu aufeinanderfolgenden gleich langen Datenstrukturen der Länge C auf Grundlage einer Indexangabe (= laufende Nummer 0, 1, 2, 3 usw.). Basisadresse in A, Länge der Datenstruktur in C.
6. FORLOOP = FOR X := A TO C STEP B.

Ersichtlicherweise kommt man mit maximal 4 Operanden und 3 Ergebnissen aus. Es genügt somit eine Parameteradresse von 3 Bits. Hiermit kann man auch die weiteren in modernen Hochleistungsprozessoren üblichen Operationen (über Gleitkommazahlen, Einzelbits usw.) unterstützen.

Jede Verarbeitungsfunktion in Tabelle 9.1 entspricht – aus Sicht der Systemarchitektur – wenigstens einem Ressourcentyp (es gibt Abwandlungen in Hinsicht auf die Operandenlängen usw.). Ein s-Operator stellt eine Hardware gemäß Abb. 9.2 auf eine bestimmte Funktion ein.

Verarbeitungs- und Speicherzugriffsfunktionen
Im Beispiel sind hierfür jeweils besondere Ressourcen vorzusehen, die ggf. entsprechend zu verketten sind. Die universellen Verarbeitungseinrichtungen gemäß Abb. 9.3 sind so auslegt, daß sie beide Aufgaben übernehmen können. Anwendungsbeispiel: eine Vektoroperation C[i] := A[i] *op* B[i] (komponentenweise Verknüpfung) erfordert fünf Ressourcen: drei zur Adressierung der Vektoren A, B, C, eine für die Operandenverknüpfung und eine zum Erkennen des Ablaufendes (FOR-Schleife).

Parallelisierung durch Aufrollen von Schleifen
Es ist bekannt, das mehrfache Durchlaufen einer Schleife durch entsprechend mehrfache Parallelausführung der Operationen des Schleifenkörpers zu ersetzen (Loop Unrolling). Ist die Durchlaufzahl (1) von Anfang an (zur Compilierzeit) bekannt und (2) nicht allzu groß, so bereitet dies in einer ReAl-Maschine keine besonderen Schwierigkeiten.

Beispiel:

FOR n = 1 to 20

– Schleifenkörper –

NEXT N

wird parallelisiert (aufgerollt), indem die zur Implementierung des Schleifenkörpers erforderliche Ressourcenausstattung 20mal angefordert und entsprechend mit Parametern versorgt wird.

Stehen nicht genügend Ressouren zur Verfügung, so ist nur eine abschnittsweise Parallelausführung möglich.

Beispiel: Im Ausdruck

FOR n = 1 TO 20

– Schleifenkörper –

NEXT N

können nur vier Schleifenkörperfunktionen parallel unterstützt werden. Hierzu ist die Schleife umzuformen:

FOR n = 1 TO 20 STEP 5

1. Schleifenkörper | 2. Schleifenkörper | 3. Schleifenkörper | 4. Schleifenkörper

NEXT N

Die Ressourcenausstattung des Schleifenkörpers wird viermal angefordert: die Schleife wird in 5 Durchläufen ausgeführt.

Ist die Durchlaufzahl zur Compilierzeit nicht bekannt (Beispiel: FOR n = 1 TO x), so ist diese einfache Form des Aufrollens nicht möglich. Ein Ausweg besteht darin, eine gewisse Anzahl an Verarbeitungsressourcen zwecks Parallelausführung pauschal bereitzustellen und deren Nutzung über entsprechend ausgebildete Iterator- und Speicherzugriffsressourcen zu steuern (Abb.n 9.4, 9.5).

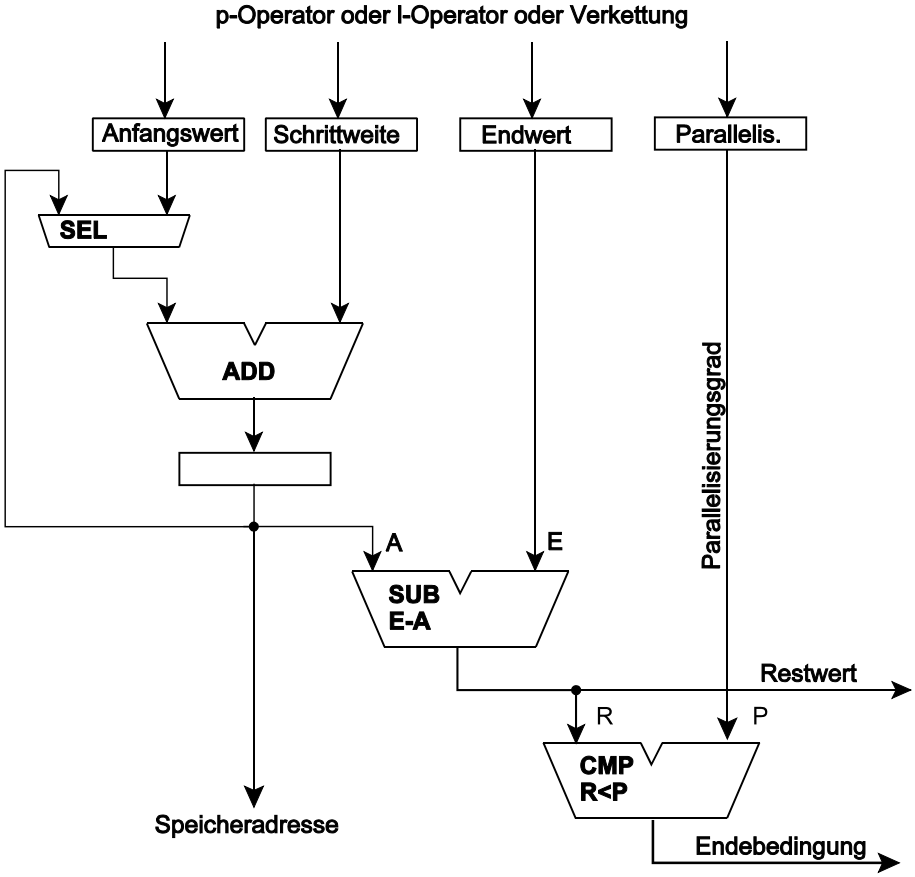


Abb. 9.4 Iterator zur Unterstützung der Parallelverarbeitung

Der Iterator entspricht im wesentlichen der Ausführung gemäß Abb. 2.16. Die Anzahl der parallel unterstützten Verarbeitungsressourcen wird als Parallelisierungsgrad P bezeichnet (P = 1: 1 Ressource, P = 2: 2 Ressourcen usw.). Die Nutzung wird üblicherweise vom Compiler gesteuert. Er ordnet z. B. einer bestimmten Schleife 4 Verarbeitungsressourcen zu (P = 4) und korrigiert die Schrittweite entsprechend:

Aus FOR n = 1 TO x wird FOR n = 1 TO x STEP P ▯ x.

Beispiel:

Aus FOR n = 1 TO 14 wird mit P = 4 FOR n = 1 TO 14 STEP 4.

Werte für n (jede Verarbeitungsresource erledigt jeweils einen dieser Werte):

im 1. Durchlauf: 1, 2, 3, 4
im 2. Durchlauf: 5, 6, 7, 8
im 3. Durchlauf: 9, 10, 11, 12
im 4. Durchlauf: 13, 14

Ersichtlicherweise sind im letzten Durchlauf nicht alle vier Verarbeitungsressourcen beschäftigt. Um den letzten Durchlauf zu erkennen, wird der aktuelle Wert vom Endwert subtrahiert: Restwert R = E - A. Ist der Restwert kleiner als der Parallelisierungsgrad (R < P), so ist das Schleifenende erreicht. Ist $R \leq 0$, so wird dei Schleife verlassen. Ist R > 0, so handelt es sich um einen letzten Durchlauf, in dem nicht alle Verarbeitungsressourcen beschäftigt sind.

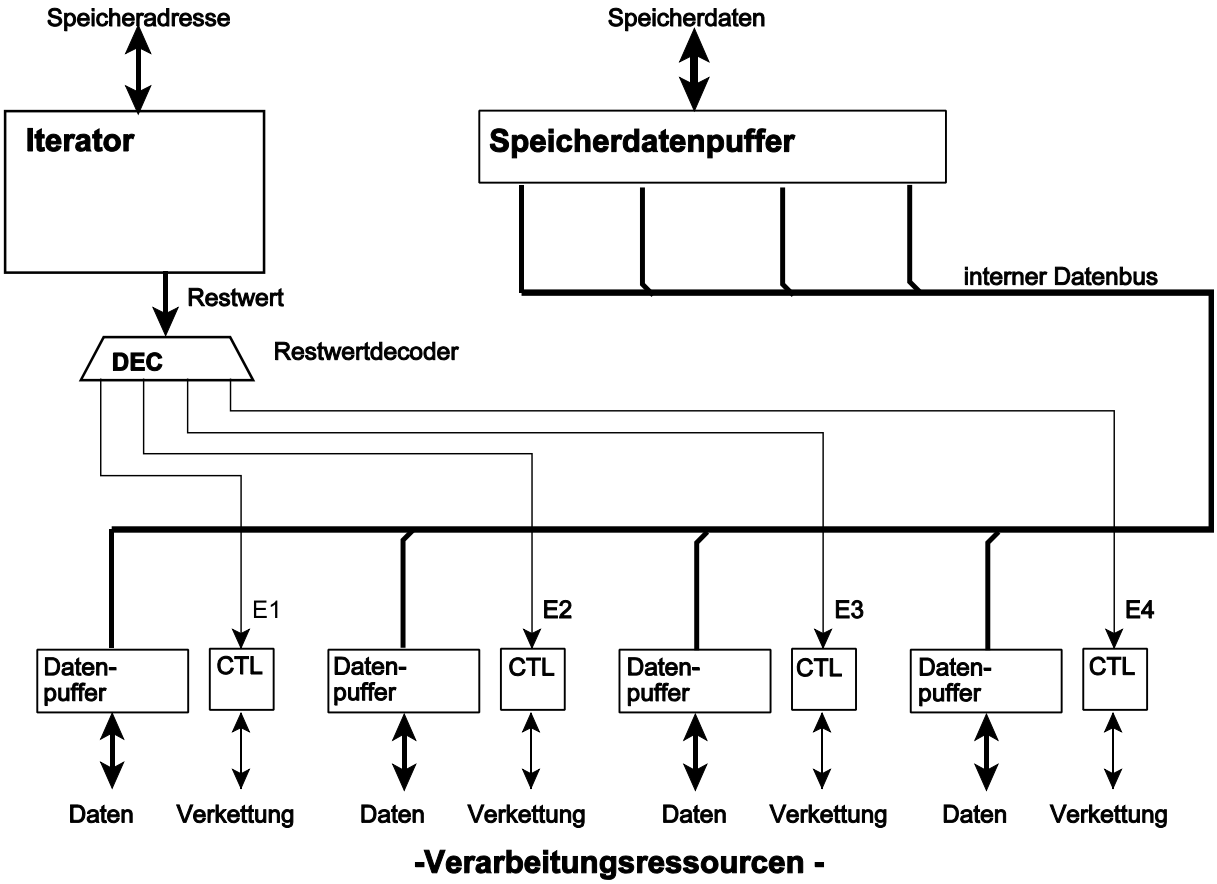


Abb. 9.5 Speicherzugriffsressource zur Unterstützung der Parallelverarbeitung

Da die Aktivierung und Versorgung der parallel arbeitenden Verarbeitungsressourcen koordiniert werden muß, ist es zweckmäßig, alle betreffenden Verarbeitungsressourcen einer einzigen entsprechend ausgelegten Speicherzugriffsressource nachzuschalten. Die Speicherzugriffsressource muß – wie auch das gesamte Speichersubsystem – die Speicherbandbreite unterstützen können, die nötig ist, um die parallel arbeitenden Verarbeitungsressourcen mit Daten zu versorgen und die anfallenden Ergebnisse abzutransportieren. Beispiel: wenn vier Ressourcen mit einer Zugriffsbreite von 64 Bits angeschlossen sind, werden die Speicherzugriffe mit einer Zugriffsbreite von 256 Bits ausgeführt. Datenpuffer, die Zugriffe mit verschiedener Breite auf verschiedenen Adressen sammeln und in Zugriffe mit größerer Breite umsetzen können, sind in modenen Hochleistungsprozessoren enthalten. Sie entsprechen somit dem Stand der Technik und müssen nicht näher beschrieben werden. Die Speicherzugriffsressource enthält einen Iterator gemäß Abb. 9.4. Jeder Verarbeitungsressource ist ein Datenpuffer und eine Verkettungssteuerung zugeordnet. Ein Verkettungsvorgang wird ausgelöst, wenn Daten bereitstehen (Lesen) oder abzuholen sind (Schreiben). Eine Verkettungssteuerung wird nur dann wirksam, wenn ihr Erlaubniseingang aktiv ist. Die Erlaubniseingänge werden vom Restwertdecoder erregt. Es handelt sich um eine kombinatorische Schaltung, die Erlaubnissignale (E1...E4) für die Verkettungssteuerungen abgibt

(Tabelle 9.2). Handelt es sich nicht um den letzten Schleifendurchlauf, so sind alle vier Verkettungssteuerungen aktiv. Im letzten Schleifendurchlauf richtet sich die Aktivierung nach dem Restwert.

Restwert	E1	E2	E3	E4
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
≤ 4	1	1	1	1

Tabelle 9.2 Zur Wirkungswweise des Restwertdecoders

ReAl-Maschinen
Eine ReAl-Maschine besteht aus:

- Der Plattform,
- Verarbeitungsressourcen,
- gemeinsamen Speichermitteln und E-A-Einrichtungen.

Diese Einrichtungen können z. B. über einen Universalbus gekoppelt sein (Abb. 9.6). In einer abgewandelten Konfiguration (Abb. 9.7) sind die gemeinsamen Speichermittel an die Plattfom angeschlossen. Der Universalbus kann in mehrere Bussysteme aufgelöst werden, z. B. in einen Operandenbus und einen Ergebnisbus.

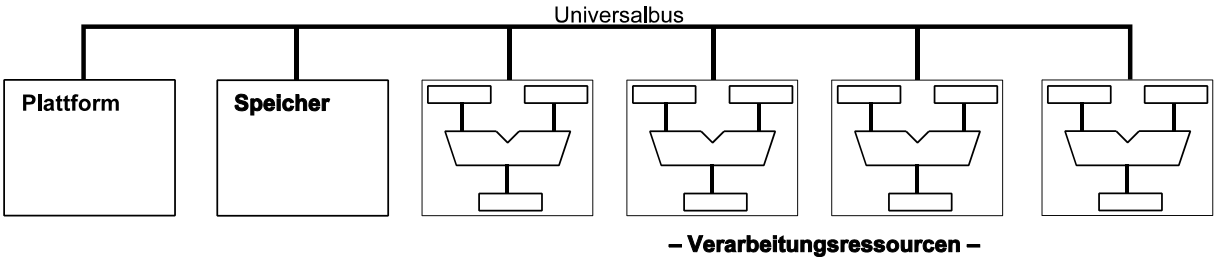


Abb. 9.6 ReAl-Machine mit Universalbus

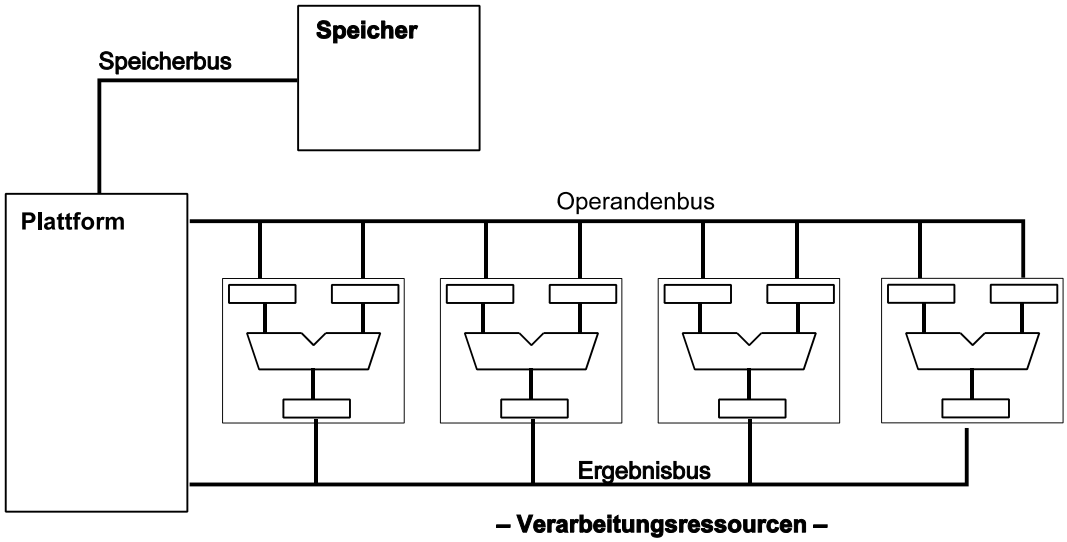


Abb. 9.7 ReAl-Maschine mit zwei Bussystemen

Ein Verarbeitungswerk kann zu einer Zeit mehr als eine Ressource verkörpern. Hierzu sind entsprechend adressierbare Speicheranordungen für die Parameter und Ergebnisse vorzusehen, die über eine Ressoucenadreßangabe adressiert werden (Abb. 9.8).

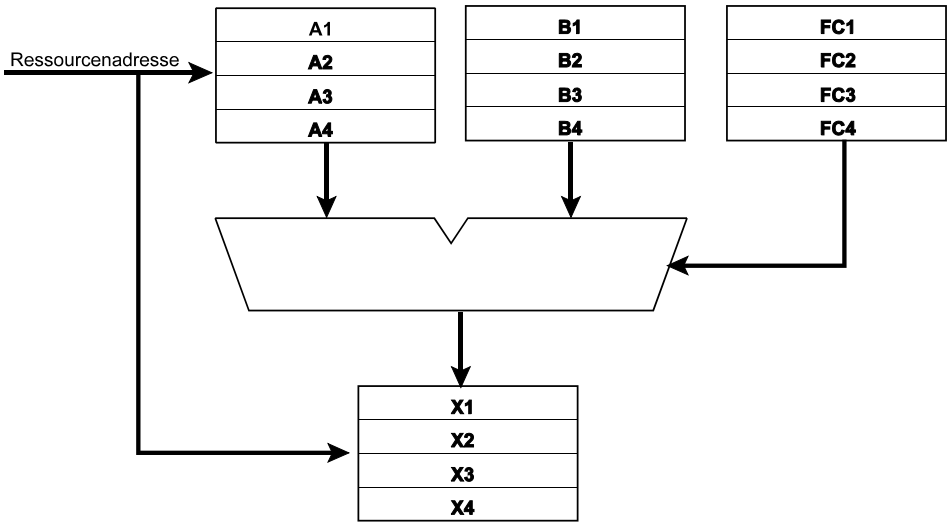


Abb. 9.8 Ein Verarbeitungswerk mit adressierbaren Speicheranordnungen kann zu einer Zeit mehr als eine Ressource verkörpern

Zentrale und autonome Steuerung
Kleinere ReAl-Maschinen können zentral gesteuert werden. Die Plattform steuert hierbei alle Speicherzugriffe und Informationstransporte. Nur die y-Operatoren werden autonom in den Verarbeitungsressourcen erledigt. Die Verkettung wird von der Plattform emuliert. Solche Maschinen können z.B. auf Grundlage von Bussystemen aufgebaut werden, an die alle Verarbeitungsressourcen als Slaves (Targets) angeschlossen sind.

Hochleistungsmaschinen erfordern eine autonome Steuerung der Speicherzugriffs- und Verkettungsfunktionen. Die Verarbeitungsressourcen gemäß Abb. 9.3 sind hierzu mit entsprechenden Anschlußsteuerschaltungen auszurüsten (Abb. 9.9) und über universelle Multimaster-Bussysteme, Schaltverteiler o. dergl. miteinander zu verbinden (vgl. die Abb.n 9.1 und 9.2).

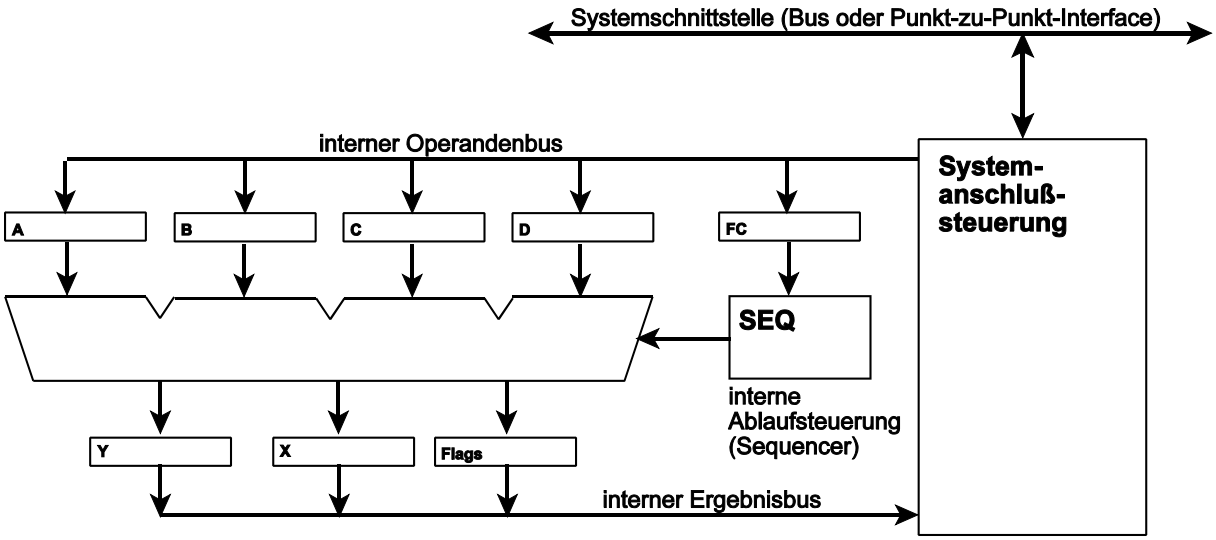


Abb. 9.9 Universelle Verarbeitungsressouce mit Systemanschlußsteuerung

Die Plattform
Es gibt verschiedene Möglichkeiten, eine Plattform aufzubauen:

- Nutzung eines herkömmlichen Prozessors,
- Auslegung als mikroprogrammiertes Steuerwerk,
- Auslegung als Sammlung elementarer Ressourcen (vgl. Abb. 5.2),
- Aufbau aus universellen Verarbeitungsressourcen mit einer anfänglichen (nach dem Einschalten bzw. Rücksetzen) festen Zusammenschaltung.

Demgemäß können die Funktionen der Plattform gesteuert werden durch:

1. Herkömmliche Maschinenbefehle,
2. Mikrobefehle,
3. elementare Maschinenbefehle für Verwaltungsaufgaben (Regiebefehle), z. B. Laden, Speichern, Verzweigen, Unterprogrammruf, Rückkehr, Unterbrechungssteuerung,
4. ReAl-Operatoren,
5. beliebige Kombinationen von 1. bis 4.

ReAl-Systeme können mehrere Plattformen enthalten.

Eine elementare Plattform als Ressourcensammlung
Hier soll untersucht werden, wieviele Resourcenadressen für eine Plattform beiseite zu setzen sind, die man sich als Zusammenfassung der in Kapitel 5 veranschaulichten Auslegungen vorstellen kann. Abb. 9.10 gibt einen Überblick über die Parameter.

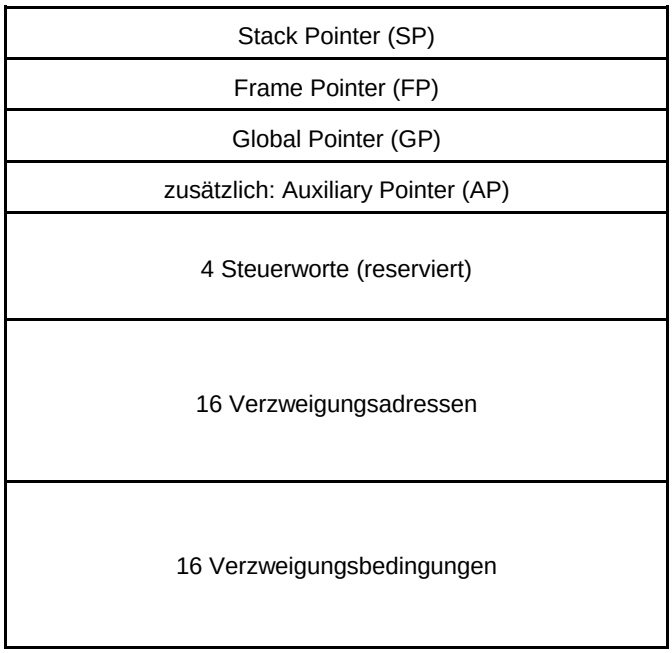


Abb. 9.10 Parameter einer elementaren Plattform. Die Verzweigungsvorkehrungen werden auch zum Unterprogrammruf verwendet

Aus Abb. 9.10 ergeben sich 40 Ziele (Registeradressen) für p-Operatoren. Mit 40 Adressen entspricht die Plattform 5 Resourcen mit je 8 Parametern oder einer flachen Ressourcenadresse von 6 Bits.

In y-Operatoren können die 40 Adressen z. B. wie folgt belegt werden:

- 16 Ziele für y-Operatoren zwecks Verzweigung,
- 16 Ziele für y-Operatoren zwecks Unterprogrammruf,
- 8 Ziele für y-Operatoren mit sonstigen Wirkungen (Rückkehr aus Unterprogramm usw.).

Aus den dargestellten Grundsatzlösungen und Varianten können ReAl-Maschinen gleichsam nach dem Baukastenprinzip konfiguriert werden. Die Auswahl ergibt sich – wie in der Rechnerarchitektur üblich – auf Grundlage von Kosten-Nutzen-Betrachtungen und Nutzungshäufigkeiten.

Beispiel:

- Eine Auslegung X bringt Vorteile bei Anwendungen der Art A,
- eine Auslegung Y bringt Vorteile bei Anwendungen der Art B.

Kommen Anwendungen der Art A häufiger vor als solche der Art B, wird man die Auslegung X wählen und umgekehrt.