ten weniger Spezialregister, hier x0 und y0, ist es notwendig, die benötigten Operanden zuvor durch move-Operationen in die jeweiligen Register zu laden. Dies geschieht hier parallel zur clr- bzw. mac-Operation. So wird durch x:(r0)+, x0 z.B. der durch r0 adressierte Zelleninhalt im x-Speicher (der Signalprozessor verfügt über zwei Adressräume x und y) in das Spezialregister x0 transportiert. Da gleichzeitig zwei solcher move-Operationen bearbeitet werden können, lässt sich das Programm in Bild 3.12b genauso schnell bearbeiten wie das in Bild 3.12a, vorausgesetzt, beide Signalprozessoren werden mit derselben Taktfrequenz betrieben. ◢

3.1.5 VLIW-Prozessoren

Die mit Signalprozessoren verwandten VLIW-Prozessoren (Very-Long-Instruction-Word-Prozessoren) führen ebenfalls mehrere Operationen, die in einem Befehl jedoch explizit codiert sind, parallel aus. Die Operationen enthalten jeweils einen Operationscode und die zur Ausführung benötigten Operanden oder Operandenadressen. Da dies für jede Operation innerhalb eines Befehls gilt, sind die Befehle sehr breit, was dieser Architekturform den Namen gegeben hat. Einfach aufgebaute VLIW-Prozessoren verarbeiten Befehle konstanter Breite mit einer immer gleichbleibenden Anzahl von Operationen. Jeder Operation im Befehl ist hierbei eine Verarbeitungseinheit fest zugeordnet, wobei Spezialisierungen möglich sind. Bild 3.13 zeigt eine solche auf wesentliche Merkmale reduzierte Struktur eines, in Fließbandtechnik arbeitenden VLIW-Prozessors. Sie hat Ähnlichkeiten mit der des skalar arbeitenden Prozessors entsprechend Bild 2.19, nur dass hier drei parallel arbeitende Verarbeitungseinheiten vorgesehen sind, nämlich eine ALU für arithmetisch-logische Befehle und Speicherzugriffe, eine FPU (floating point unit) für Gleitkommabefehle und eine BPU (branch processing unit) für Sprungbefehle.

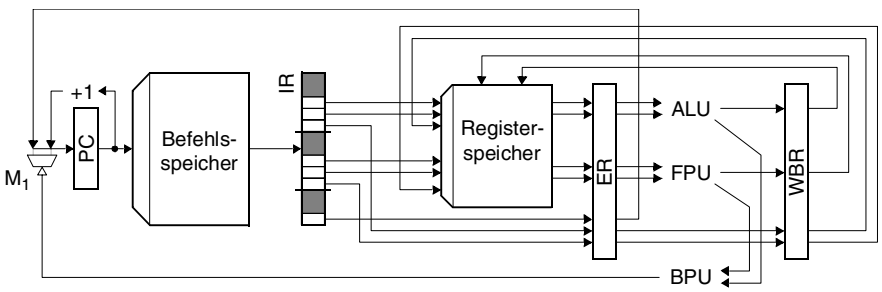

Bild 3.13. Einfacher VLIW-Prozessor ohne Datenspeicher, der in einem vierstufigen Fließband drei Operationen pro Takt bearbeitet (vgl. Bild 2.19). Es sind nur die Datenpfade dargestellt

Der im Bild dargestellte VLIW-Prozessor hat einige Nachteile, die in realen Prozessoren normalerweise vermieden werden. Zum Beispiel muss der Befehlsspeicher über einen Bus an den Prozessor gekoppelt sein, über den pro Takt ein Befehl geladen werden kann. Da dies aus Kostengründen oft nicht möglich ist, kommen in VLIW-Prozessoren normalerweise Befehlscaches zum Einsatz, die einen schmalen Bus zum Hauptspeicher und einen breiten Bus zum Prozessor besitzen. Weiter gilt für den dargestellten VLIW-Prozessor, dass die Befehle immer genauso viele Operationen enthalten müssen, wie Verarbeitungseinheiten vorhanden sind, und zwar auch

dann, wenn dies bei einer gegebenen Aufgabenstellung nicht sinnvoll ist. Gegebenenfalls müssen in den Befehlen nops codiert werden, die jedoch Platz im Befehlsspeicher bzw. -cache belegen. Daher realisiert man in nahezu allen modernen VLIW-Prozessoren Prinzipien, die eine komprimierte Speicherung von Befehlen erlauben. Dies wird im nächsten Abschnitt genauer erläutert.

◤ Bemerkung. An die *Registerspeicher* von Prozessoren, die mehrere Operationen parallel bearbeiten können, werden hohe Anforderungen gestellt. Zum Beispiel muss mit der Anzahl der parallel verarbeitbaren Operationen einerseits die Anzahl der Register vergrößert werden, damit die einzelnen gleichzeitig auszuführenden Operationen auf unterschiedlichen Registern arbeiten können und andererseits muss die Anzahl der Registerports erhöht werden, um Parallelverarbeitung überhaupt erst zu ermöglichen. Zusammengenommen steigt also mit dem Parallelitätsgrad der Realisierungsaufwand, was wiederum bewirkt, dass die Zugriffsgeschwindigkeit auf den Registerspeicher sinkt und sich dadurch möglicherweise der kritische Pfad durch den Prozessor verlängert.

In realen VLIW-Prozessoren wird der Registerspeicher deshalb oft in *Bänken* realisiert, auf die parallele Zugriffe möglich sind. Zum Beispiel kann der TMS320C62x bis zu acht Operationen gleichzeitig ausführen und benötigt daher wenigstens 24 Ports, um auf Operanden- und Ergebnisregister zuzugreifen (nicht mitgerechnet sind die für Speicherzugriffe oft zusätzlich vorhandenen Ports). Tatsächlich unterteilt sich der Prozessor jedoch in zwei sog. Datenpfade, die jeweils vier Operationen parallel bearbeiten können und getrennte 16-Port-Registerspeicher ansprechen. Zur Kommunikation zwischen den Datenpfaden, ist es außerdem möglich, pro Takt einen Operanden aus dem jeweils gegenüberliegenden Registerspeicher zu lesen. Wie für VLIW-Prozessoren typisch, geschieht dies natürlich explizit und ist somit für den Benutzer sichtbar [185].

Nachteilig an einer solchen Aufteilung des Reisterspeichers ist, dass sie bei der Programmierung explizit berücksichtigt werden muss, was vor allem den Aufwand bei Realisierung eines Hochsprachenübersetzers vergrößert. Außerdem ist die Kommunikation zwischen den Datenpfaden ein potentieller Engpass, da sich im Prinzip alle benötigten Operanden im jeweils gegenüberliegenden Registerspeicher befinden können. Aus diesem Grund werden Registeraufteilungen, wie im TMS320C62x, möglichst vermieden, und zwar entweder, indem man das Maß an Parallelität, mit dem der Prozessor arbeitet, vermindert, oder indem die Unterteilung in mehrere Registerspeicher für den Benutzer unsichtbar geschieht, wie z.B. beim Nemesis X der TU Berlin [108] bzw. beim nicht nach dem VLIW-Prinzip arbeitenden Alpha 21264 von Compaq [27].

Der Nemesis X führt pro Takt maximal drei Operationen aus und greift dabei auf einen Registerspeicher mit vier Lese- und drei Schreibports zu. Für eine Realisierung sind in dem verwendeten FPGA (field programmable gate array) hingegen nur Registerspeicher mit maximal zwei Ports verfügbar, deren Zugriffszeit jedoch ausreichend kurz ist, um innerhalb eines Fließbandtakts zwei Zugriffe zeitsequentiell bearbeiten zu können. Auf diese Weise ist es möglich, einen 4-Port-Registerspeicher aufzubauen. Der 7-Port-Registerspeicher (tatsächlich sind es acht Ports, von denen einer jedoch unbenutzt bleibt) lässt sich schließlich davon ableiten, indem man drei identische 4-Port-Registerspeicher so miteinander verschaltet, dass Schreibzugriffe auf den Registerspeichern gleichzeitig durchgeführt und die Inhalte der Registerbänke auf diese Weise jeweils synchron zueinander gehalten werden. Die Struktur ist in Bild 3.14a dargestellt.

Der Alpha 21264 ist ähnlich aufgebaut, wobei hier zwei Registerspeicher gespiegelt verwendet werden, die sechs Schreibports und vier Leseports besitzen (Bild 3.14b). Jeweils zwei Schreibports sind für die Bearbeitung ausstehender Ladebefehle vorgesehen (a). Ihre genaue Verschaltung wird im Handbuch nicht erläutert, ebenso wenig wie die zur Synchronisation der beiden Registerspeicher abzuwartende Latenzzeit von einem Takt. Die Bedeutung der durch b und c markierten Register ist dem Autor daher nicht bekannt. Eine mögliche Erklärung könnte sein, dass ohne diese Register die Last an den die Schreibports treibenden Verarbeitungseinheiten verdoppelt werden würde. Da außerdem die Registerbänke mit einer räumlichen Distanz aufgebaut sein müssen, könnte dies zusammengenommen den kritischen Pfad verlängern.

Trotz der durch die Register verursachten Verzögerung ist die Ausführungsgeschwindigkeit von Programmen nach [92] nur unwesentlich geringer als mit einem echten 14-Port-Registerspeicher. Dies liegt zum Teil auch daran, dass bei der dynamischen Umordnung der Befehle durch den superskalaren Prozessor die zusätzliche Verzögerung durch Ausführung unabhängiger Befehle überbrückt wird (siehe auch Abschnitt 3.2).

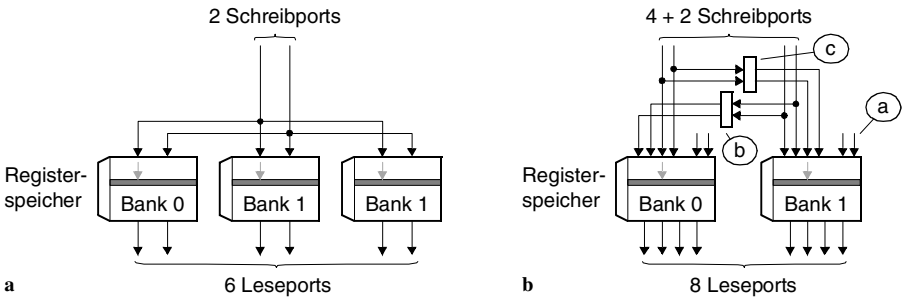

Bild 3.14. Vergrößerung der Anzahl an Leseports durch Registerspiegelung. **a** für den Nemesis X der TU Berlin. **b** für den Alpha 21264 von Compaq ◢

Befehlscodierung

Bei der bereits erwähnten Kompression von Befehlen wird einerseits die explizite Codierung von nops vermieden und andererseits die Breite der einzelnen Operationen reduziert. Bild 3.15a bis f zeigt die Befehlsformate einiger verbreiteter VLIW-Prozessoren. Zu Vergleichszwecken ist in Bild 3.15a zunächst das feste Befehlsformat des VLIW-Prozessors aus Bild 3.13 angegeben. Nops sind darin, wie beliebige andere Operationen auch, explizit und unkomprimiert zu codieren.

- *Trace 7/300.* In Bild 3.15b ist das Befehlsformat der Trace 7/300 von Multiflow, einem der ersten nach dem VLIW-Prinzip arbeitenden Rechner dargestellt [101]. Er verarbeitet aus einem 8 KWort großen Befehlscache jeweils 256 Bit breite Befehle, in denen sieben Operationen enthalten sind (ein Sprungbefehl, vier arithmetisch-logische Befehle und zwei Gleitkommabefehle). Jeder Operation ist eine Verarbeitungseinheit fest zugeordnet. Folglich sind nops, genau wie andere Operationen auch, explizit und unkomprimiert im Befehl anzugeben. Im Befehlsspeicher werden die Befehle jedoch komprimiert gehalten, indem einer Gruppe von je vier Operationen eine Maske vorangestellt ist, in der die hinzuzufügenden nops codiert sind. Dies ist beispielhaft in Bild 3.15b dargestellt: Das komprimierte Befehlswort (oben) wird zu einem unkomprimierten Befehlswort (unten) ausgeweitet. Die schraffierten Felder kennzeichnen dabei die mit Hilfe der Maske hinzugefügten nops.
- *Nemesis X.* Anders als bei der Trace 7/300 wird bei dem an der TU Berlin zur Untersuchung der dynamischen Binärübersetzung (siehe Abschnitt 4.2) entwickelten Nemesis X nicht die Codierung von nops vermieden, sondern ein sehr kompaktes Befehlsformat verwendet, in dem, wie Bild 3.15c zeigt, alternativ zwei oder drei Operationen codiert sein können [108]. Ein Befehl ist insgesamt 64 Bit, eine Operation zwischen 20 und 42 Bit breit. Die Zuordnung der Operati-

onen und Verarbeitungseinheiten ist beim drei Operationen enthaltenden Befehlsformat fest definiert. Beim zwei Operationen enthaltenden Befehlsformat geschieht sie derart, dass die 22 Bit breite Operation OP1 fest einer einzelnen Verarbeitungseinheit und die 42 Bit breite Operation OP2 alternativ von einer der beiden verbleibenden Verarbeitungseinheiten ausgeführt wird.

Neben der einfachen Realisierung ist das für VLIW-Prozessoren ausgesprochen kompakte Befehlsformat von Vorteil. Während z.B. der Nemesis X im Idealfall 12 Operationen in einem 256 Bit umfassenden Bereich codieren kann, sind dies acht Operationen beim TMS320C62x von Texas Instruments, einem Prozessor, der ebenfalls ein sehr kompaktes Befehlsformat aufweist [185]. Allerdings besteht dieser Vorteil nur, wenn es möglich ist, wenigstens zwei sinnvolle Operationen in einem Befehl zu codieren.

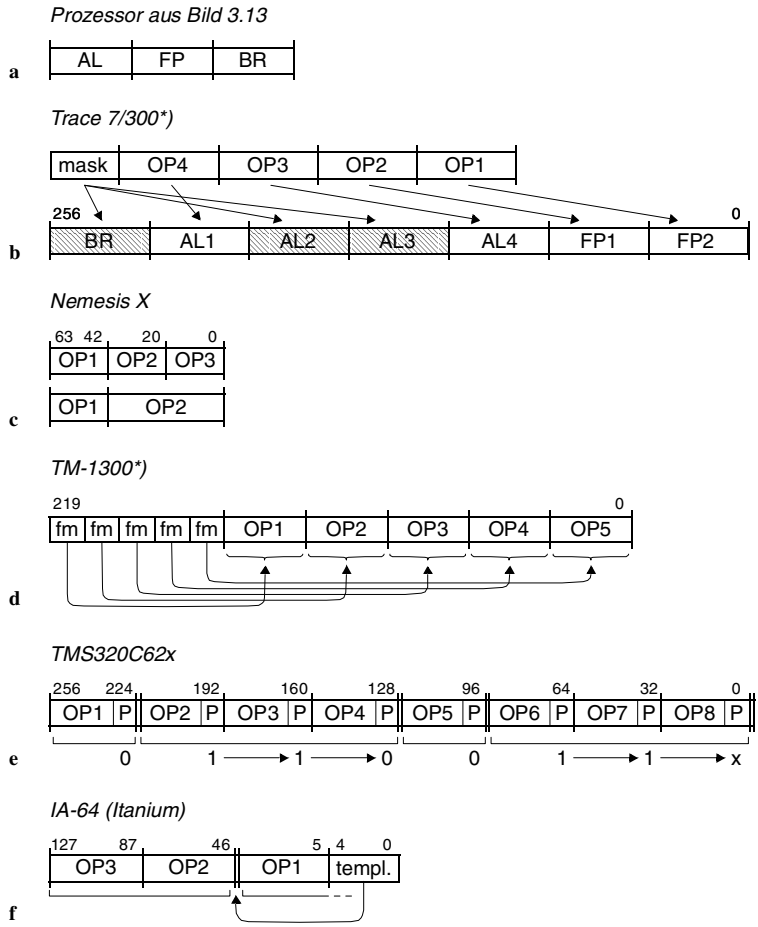

Bild 3.15. Befehlsformate gängiger VLIW-Prozessoren. *) Die bitgenauen Aufteilungen der Befehle des Trace 7/300 von Multiflow und des TM-1300 von Philips sind dem Autor unbekannt. Die Teilbilder b und d sind daher als grundsätzlich zu verstehen (vgl. [101] und [39])

- *TriMedia TM-1300*. Auch beim TriMedia TM-1300 von Philips sind die im Befehl codierten Operationen fest bestimmten Verarbeitungseinheiten, die jeweils aus mehreren Funktionseinheiten bestehen, zugeordnet [141] (siehe auch [39, 142]). Bild 3.15d stellt zu diesem Prozessor den prinzipiellen Aufbau eines bis zu 220 Bit breiten Befehlswortes dar. Jede der maximal fünf parallel ausführbaren Operationen ist um eine zwei Bit breite Formatangabe erweitert, die dem Befehlswort vorangestellt ist und durch die sich die Funktionalität der Operationen erweitern bzw. einschänken lässt.

  Zum Beispiel sind Operationen abhängig vom Inhalt eines allgemein nutzbaren Registers ausführbar. Mit Hilfe der Formatangabe kann eine Operation jedoch als unbedingt gekennzeichnet und auf diese Weise der sonst zur Codierung der Bedingung benötigte Speicherplatz eingespart werden. Insbesondere ist es dadurch möglich, eine Operation auch als unbedingt *nicht* ausführbar zu kennzeichnen, was die Codierung der entsprechenden Operation überflüssig macht. Ein nop belegt im Befehlswort des TM-1300 daher lediglich zwei Bits.

  Beachtenswert ist, dass die Formatangaben insgesamt dem Befehlswort und nicht individuell den einzelnen Operationen vorangestellt sind. Dies ist sinnvoll, weil sich dadurch die parallele Decodierung der Operationen vereinfacht. Wären nämlich die Formatangaben als Teil der Operationen gespeichert, müsste zunächst OP1 decodiert werden, bevor bekannt ist, mit welcher Bitposition OP2 beginnt. Anschließend müsste OP2 decodiert werden, um OP3 adressieren zu können usw.

- *TMS320C62x*. Noch flexibler als beim TM-1300 sind die Befehlsformate des TMS320C62x von Texas Instruments [185] und der Prozessorarchitektur IA-64 von Intel und HP bzw. deren Umsetzung Itanium und Itanium 2 von Intel. Beim TMS320C62x in Bild 3.15e enthält jede Operation ein sog. *Paketbit* (*packet bit*), in dem codiert ist, ob die „unmittelbar rechts stehende“ Operation parallel oder sequentiell ausgeführt werden soll. Ein 256 Bit breites Speicherwort kann auf diese Weise in mehrere Pakete unterteilt werden, in denen jeweils eine unterschiedliche Anzahl von Operationen parallel enthalten sind.

  Zum Beispiel geben die in Bild 3.15e unter dem Speicherwort angegebene Paketbits an, dass zunächst OP1 einzeln, anschließend OP2, OP3 und OP4 parallel, danach OP5 und schließlich OP6, OP7 und OP8 parallel ausgeführt werden sollen. Das Paketbit an Bitposition 0 jedes Speicherworts wird dabei ignoriert, so dass sich maximal acht Operationen gleichzeitig ausführen lassen. Dies vereinfacht die Realisierung, da parallel zu bearbeitende Operationen nicht in benachbarten nur zeitsequentiell lesbaren Speicherworten stehen können.

  Die Zuordnung der Operationen zu den Funktionseinheiten geschieht beim TMS320C62x nicht entsprechend der Positionen innerhalb eines Pakets, sondern ist beliebig und wird durch den Operationscode festgelegt. Da der Prozessor über zwei symetrische Datenpfade mit jeweils vier Funktionseinheiten und einen Registerspeicher verfügt, ist dabei zusätzlich zur Funktionseinheit zu definieren, in welchem Datenpfad die Operation bearbeitet werden soll. Dies geschieht mit Hilfe der in den Operationen codierten Registeradressen.

- *IA-64 (Itanium 2).* Das Befehlsformat des TMS320C62x weist Ähnlichkeiten mit dem der Prozessorarchitektur IA-64 entsprechend Bild 3.15f auf. In einem 128 Bit breiten Speicherwort sind drei Operationen codiert, die parallel oder sequentiell ausgeführt werden können, und zwar abhängig von einem als *Template* bezeichneten Feld, mit dessen Hilfe das Ende jedes Operationspakets definiert ist (die sog. stops). Zusammengenommen entspricht das Template den Paketbits des TMS320C62x. Jedoch können von einem Prozessor mit IA-64-Architektur auch Operationen gleichzeitig ausgeführt werden, die in benachbarten Speicherworten codiert sind. Daher ist das Maß an maximal erreichbarer Parallelität nicht durch das zugreifbare Speicherwort begrenzt, sondern implementierungsabhängig skalierbar (siehe hierzu den nachfolgenden Abschnitt).

Die Zuordnung der im Befehl codierten Operationen und der Funktionseinheiten geschieht i. Allg. durch eine Batterie von Multiplexern. Für die Trace 7/300 ist das entsprechende Schaltnetz in Bild 3.16 angedeutet. Die im komprimierten Befehl codierte Operation OP1 wird nur dann an die Funktionseinheit f_1 weitergeleitet, wenn in der assoziierten Maske nicht festgelegt ist, dass die Funktionseinheit ein nop ausführen soll (hier willkürlich durch eine Null codiert). Für die nächste Funktionseinheit f_2 gilt, dass sie entweder ein nop verarbeitet, wenn dies in dem entsprechenden Maskenbit codiert ist, oder eine der Operationen OP1 bzw. OP2, je nachdem, ob OP1 bereits der Funktionseinheit f_1 zugeordnet wurde oder nicht.

In ähnlicher Weise wird mit allen anderen Funktionseinheiten verfahren, wobei mit jedem zusätzlichen Multiplexer ein zusätzliches Maskenbit berücksichtigt werden muss. Ein etwas höherer Aufwand ist bei Prozessoren wie dem TM-1300, dem TMS320C62x oder dem Itanium 2, zu treiben, da die zu verarbeitenden Operationspakete an unterschiedlichen Bitpositionen innerhalb eines Speicherworts beginnen oder die Operationen unterschiedliche Breiten aufweisen können. Der grundsätzliche Aufbau einer Schaltung zur Dekompression der Befehle ist hierbei jedoch mit der in Bild 3.16 dargestellten vergleichbar.

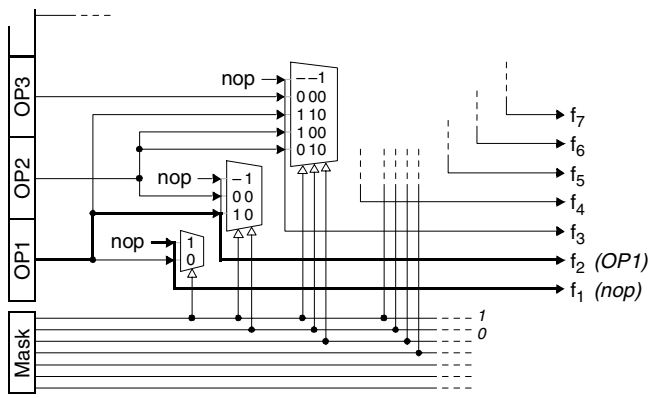

Bild 3.16. Mögliche Realisierung des Schaltnetzes zur Zuordnung der Operationen und Funktionseinheiten in der Trace 7/300 von Multiflow

Kompatibilität und Skalierbarkeit

Neben der Ausführungsgeschwindigkeit, den Kosten und dem Strombedarf gibt es zahlreiche andere Kriterien, die den Entwurf einer Prozessorarchitektur beeinflussen. Hierzu zählen die Kompatibilität und die Skalierbarkeit. Eine Prozessor B ist kompatibel (compatible) zu einem zweiten Prozessor A, wenn B sämtliche Programme ausführen kann, die sich für A codieren lassen. Falls zusätzlich A in der Lage ist, alle für B codierbaren Programme auszuführen, werden A und B als *voll kompatibel* zueinander bezeichnet.

Benutzerkompatibel (*user compatible*) sind schließlich alle Prozessoren B, deren Kompatibilität zu A auf den Benutzermodus beschränkt ist (siehe Abschnitt 1.4.3). Zwar sind sie in der Lage, die meisten für A codierten Programme auszuführen, i.Allg. gilt dies jedoch nicht für Betriebssysteme, deren Ressourcenverwaltung gewöhnlich in einem höherprivilegierten Betriebszustand bearbeitet wird (z.B. Zugriffe auf die geschützte Seitentabelle einer Speicherverwaltungseinheit). Als Konsequenz sind mit der Entwicklung eines zu einem Vorgänger A benutzerkompatiblen Prozessors B die existierenden Betriebssysteme zu portieren.

Eine Technik, mit deren Hilfe es auf einfache Weise gelingt, Kompatibilität zu erreichen, ist die in Abschnitt 2.1.7 beschriebene *Mikroprogrammierung*. Zum Beispiel ist es möglich, die Multiplikation einmal als aufwendiges, dafür jedoch mit einem Durchsatz von einer Operation pro Takt arbeitendes Schaltnetz zu realisieren und ein anderes mal mikroprogrammiert, indem die Multiplikation auf Additionen und Shift-Operationen abgebildet wird. Dies ist z.B. in den System/360-Rechnern von IBM in den 60er Jahren genutzt worden, um unterschiedliche Systeme anbieten zu können, die sich in Preis und Geschwindigkeit erheblich voneinander unterschieden, jedoch voll kompatibel zueinander waren [58].

In modernen Prozessoren hat die Mikroprogrammierung an Bedeutung verloren, da sich selbst komplizierteste Funktionseinheiten höchster Geschwindigkeit kostengünstig integrieren lassen. Kompatibilität wird heute demzufolge nicht durch Einsatz einer bestimmten Technik erreicht, sondern von Fall zu Fall auf unterschiedliche Weise. Soll z.B. ein VLIW-Prozessor realisiert werden, der zu dem in Bild 3.13 kompatibel ist, aber eine größere Operationsparallelität als sein Vorgänger aufweist, dann muss sich der neue Prozessor nach dem Einschalten zunächst genauso verhalten, als könnte er ebenfalls wie das Vorbild nur wenige Operationen gleichzeitig bearbeiten. Durch Umschalten der Interpretationsweise, z.B. indem ein Steuerbit gesetzt oder gelöscht wird, kann später die Operationsparallelität bei Aufruf entsprechend codierter Programme erhöht werden. Wesentlich hierbei ist, dass der Prozessor zunächst in einem kompatiblen Modus startet, da sich nur so existierende Anwendungen weiterhin fehlerfrei bearbeiten lassen.

Während sich die Kompatibilität meist auf in der Vergangenheit realisierte Prozessoren bezieht, stehen bei der *Skalierbarkeit* zukünftige Entwicklungen im Vordergrund. Komponenten oder Eigenschaften einer Prozessorarchitektur werden als skalierbar bezeichnet, wenn es möglich ist, kompatible Erweiterungen vorzunehmen, durch die sich vorhandene Programme mit höher Geschwindigkeit bearbeiten las-

sen. Entsprechend dieser freien Definition bezieht sich die Skalierbarkeit auf einzelne Merkmale und nicht auf eine Prozessorarchitektur insgesamt. Eine solche Einschränkung ist sinnvoll, weil in einer als skalierbar geltenden Prozessorarchitektur i. Allg. nicht sämtliche darin enthaltenen Komponenten bzw. verwirklichten Eigenschaften skalierbar sind. Hinzu kommt, dass die Verfahren zum Erreichen von Skalierbarkeit als von den Komponenten und Eigenschaften abhängig, besser separat betrachtet werden sollten, was im Folgenden geschieht.

Es ist möglich, Skalierbarkeit zu erreichen, ohne dies explizit beim Entwurf einer Prozessorarchitektur berücksichtigen zu müssen, nämlich dann, wenn das Vorhandensein einer Komponente bzw. Eigenschaft nicht oder nur rudimentär auf das jeweilige Programmiermodell wirkt, aus Programmierersicht also *transparent* ist. Zum Beispiel zeichnen sich Caches durch eine gute Skalierbarkeit aus, da sich allein durch Vergrößerung der jeweiligen Speicherkapazität die Geschwindigkeit, mit der ein entsprechend ausgestatteter Prozessor arbeitet, vergrößern lässt. Die auszuführenden Programme müssen hierzu weder modifiziert noch neu codiert werden.

Im Allgemeinen ist beim Entwurf der Komponenten einer Prozessorarchitektur jedoch explizit auf Skalierbarkeit zu achten. Die Umsetzung geschieht dabei mit denselben Techniken, die auch zum Erreichen von Kompatibilität verwendet werden. Zum Beispiel lässt sich die Mikroprogrammierung nutzen, um eine zeitsequentiell arbeitende skalierbare Multiplikationseinheit zu realisieren, die mit steigender Integrationsdichte durch ein einschrittig arbeitendes Multiplikationsschaltnetz ersetzt werden kann, um so die Ausführungsgeschwindigkeit von existierenden Programmen, die die Multiplikation verwenden, zu verbessern. Voraussetzung ist natürlich, dass ein Multiplikationsbefehl von Anfang an im Befehlssatz des Prozessors vorgesehen, die Skalierbarkeit also geplant wird.

Das Beispiel ist insofern unglücklich, weil die Realisierung einer zeitsequentiell arbeitenden Multiplikationseinheit oft einen anderen Sinn als eine gute Skalierbarkeit hat, nämlich den, die Programmierung des entsprechenden Prozessors zu vereinfachen. Interessanter sind Techniken, die den ausschließlichen Zweck haben, Skalierbarkeit zu erreichen. Im Zusammenhang mit VLIW-Prozessoren sind hier vor allem Verfahren zu nennen, mit denen sich das Maß an Operationsparallelität skalieren lässt. Im Idealfall sollte dabei die Ausführungsgeschwindigkeit von Programmen durch das zukünftige Hinzufügen parallel arbeitender Verarbeitungseinheiten proportional steigen.

Da in VLIW-Prozessoren die Operationsparallelität explizit codiert ist, bedeutet dies jedoch auch, dass aktuelle Programme bereits mit einem Maß an Parallelität codiert sein müssen, das erst in zukünftigen Realisierungen der Prozessorarchitektur tatsächlich nutzbar ist. Für einen aktuellen Prozessor hat dies schließlich zur Folge, dass die Semantik parallel auszuführender Operationen sequentiell nachgebildet werden muss. Die notwendigen Modifikationen sollen anhand der in Bild 3.17a dargestellten Registertransferschaltung beschrieben werden. Das Bild zeigt einen in drei Fließbandstufen unterteilten Datenpfad, der als Teil eines VLIW-Prozessors eine Operation verarbeitet. Datenflusskonflikte zu bereits erzeugten, jedoch noch nicht in den Registerspeicher geschriebenen Ergebnissen werden mit Hilfe der

Bypässe x und *w* gelöst (siehe Abschnitt 2.2.3). Dabei wurde hier der Übersichtlichkeit halber darauf verzichtet, die von anderen Datenpfaden kommenden Bypässe darzustellen.

Angenommen die Operation r0 = r1 wird vom dargestellten Datenpfad bearbeitet, dann ist das Ergebnis von r0 erst in der Rückschreibpfase im Registerspeicher verfügbar, so dass eine nachfolgende Operation r1 = r0 bei sequentieller Ausführung nur deshalb den erwarteten Operanden verarbeitet, weil er über einen Bypass verfügbar gemacht wird. Falls die beiden Operationen jedoch gleichzeitig ausgeführt werden, muss die zweite Operation nicht das unmittelbar zuvor erzeugte Ergebnis, sondern den Inhalt des noch nicht überschriebenen Registers r0 lesen. Dies lässt sich durch Deaktivierung des entsprechenden Bypasses erreichen. In der Konsequenz sind nach den beiden Operationen die Inhalte der Register r0 und r1 vertauscht. Insgesamt wird also dasselbe Ergebnis erzeugt, als würden die Operationen tatsächlich parallel ausgeführt.

Der zeitliche Verlauf der Bearbeitung eines aus mehreren Operationen bestehenden und als parallel zu interpretierenden Operationspakets veranschaulicht Bild 3.17b. Die Operation OP1 wird zunächst aus dem Befehlsspeicher gelesen und in das Instruktionsregister IR geladen. Mit dem Folgetakt wird wie gewohnt auf die zu verarbeitenden Operanden zugegriffen, und zwar entweder durch direkte Adressierung des Registerspeichers oder – falls Datenabhängigkeiten bestehen – durch Verwendung der Bypässe. Die eigentliche Ausführung der Operation beginnt zum Zeitpunkt t_3. Gleichzeitig erfolgt bereits der Zugriff auf die Operanden der Operation OP2, die sich entweder im Registerspeicher oder in der Write-Back-Stufe des Fließbands befinden, wobei ggf. der mit *w* (write back) beschriftete Bypass aktiviert wird.

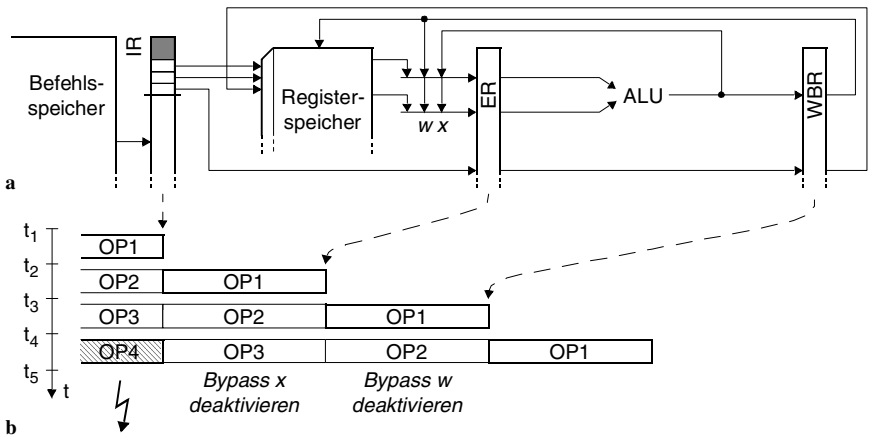

Bild 3.17. Steuerung der Bypässe, um die Semantik parallel auszuführender Operationen sequentiell nachzubilden. **a** Registertransferstruktur eines Datenpfads durch einen VLIW-Prozessor mit vierstufigem Fließband. **b** Zeitlicher Fluss eines aus vier Operationen bestehenden Befehls durch das Fließband

Da parallel auszuführende Operationen keine Datenabhängigkeiten aufweisen können, muss durch den Decoder dafür gesorgt werden, dass der mit *x* (execute)

beschriftete Bypass in dieser Verarbeitungsphase nicht schaltet. Falls also die Operationen r0 = r1 und r1 = r0 unmittelbar aufeinander folgend gestartet werden, wird mit der zweiten Operation nicht das Ergebnis der ersten Operation weiterverarbeitet, sondern der „alte“ Inhalt von r0. Im Endeffekt werden die Registerinhalte also nach Bearbeitung des Operationspakets vertauscht im Registerspeicher erscheinen. In derselben Art und Weise lässt sich mit der dritten regulär parallel auszuführenden Operation OP3 verfahren, wobei nun zusätzlich auch der mit einem *w* beschriftete Bypass deaktiviert wird.

Falls eine vierte Operation OP4 mit paralleler Semantik ausgeführt werden soll, ist dies hier nicht mehr möglich, da OP1 zum Zeitpunkt t_5 sein Ergebnis in den Registerspeicher überträgt und OP4 erst nach t_5 auf den Registerspeicher zugreift, mithin der ggf. benötigte Inhalt des überschriebenen Ergebnisregisters nicht mehr zur Verfügung steht. Tritt dieser Fall dennoch auf, kann z.B. eine Ausnahmebehandlung angestoßen und das vollständige Operationspaket durch ein Programm emuliert werden. Der damit verbundene Geschwindigkeitsverlust ist tolerierbar, wenn die Anzahl der als parallel codierten maximal bearbeitbaren Operationen so groß ist, dass Ausnahmesituationen nur mit geringer Häufigkeit auftreten. – Der in Bild 3.13 dargestellte VLIW-Prozessor, erweitert um die hier beschriebene Funktionalität, verarbeitet z.B. Operationspakete mit neun Operationen, ohne eine Ausnahmebehandlung zu verursachen. Unter realistischen Voraussetzungen sollten Operationspakete mit mehr als 30 Operationen auf diese Weise verarbeitbar sein.

Neben einer geänderten Bypasssteuerung sind noch weitere Modifikationen erforderlich, um die Semantik operationsparalleler Verarbeitung sequentiell nachahmen zu können. Zum Beispiel dürfen Operationen, bei denen Datenabhängigkeiten normalerweise nicht über Bypässe, sondern durch Sperren des Fließbands (*interlock*) gelöst werden, innerhalb eines Operationspakets gerade nicht dazu führen, dass das Fließband gesperrt wird. Des Weiteren sind Verzweigungen und zum Teil auch *Ausnahmeanforderungen* in ihrer Bearbeitung solange zu verzögern, bis das Ende eines Operationspakets erreicht ist. Falls die Operationsverarbeitung selbst die Ausnahmesituation verursacht, muss außerdem die laufende Operation in der Weise unterbrochen werden, dass der Zustand des Prozessor vor Bearbeitung des Operationspakets wieder hergestellt ist. Zumindest bezogen auf den Registerspeicher ist dies jedoch sehr einfach realisierbar, da das erste Ergebnis eines Operationspakets erst am Ende der Decodierung der letzten als parallel zu interpretierenden Operation gespeichert wird.

Die beschriebenen Erweiterungen sind im Rahmen von Untersuchungen zur Skalierbarkeit von VLIW-Prozessoren bereits Mitte der 90er Jahre an der TU Berlin entwickelt worden, aber bisher noch nicht veröffentlicht worden. Neben dem geringen Aufwand einer Realisierung ist vor allem die Erhaltung der Semantik einer parallelen Operationsverarbeitung von Vorteil. Nachteilig ist jedoch, dass die Skalierbarkeit auf einen maximalen Wert begrenzt ist, der bei der Codierung von Programmen explizit berücksichtigt werden muss (abhängig von der Fließbandtiefe). Außerdem setzt das Verfahren die Vernetzung der einzelnen Verarbeitungseinheiten mit Bypässen voraus, was nicht immer der Fall ist.

Die Nachteile werden mit einem anderen, sehr einfachen Verfahren vermieden: Hierbei wird das *Programmiermodell* derart definiert, dass es für die in einem Operationspaket enthaltenen Operationen ohne Bedeutung ist, ob sie *sequentiell* oder *parallel* bearbeitet werden. Zum Beispiel ist im Programmiermodell der Prozessorarchitektur IA-64 von Intel und HP [70] vorgegeben, dass aufeinander folgende Operationen als Teil einer sog. *Gruppe* nur dann parallel codierbar sind, wenn weder *echte Datenabhängigkeiten* bestehen, noch mehrere Operationen auf dasselbe Register schreibend zugreifen (somit lässt sich der Tausch von Registerinhalten nicht in einem Befehl codieren).

Außerdem ist festgelegt, dass ein verzweigender Sprung alle in derselben Gruppe enthaltenen nachfolgenden Operationen überspringt, unabhängig davon, ob sie tatsächlich sequentiell oder parallel ausgeführt werden. Allein anhand dieser beiden Regeln lässt sich eine unbegrenzte Skalierbarkeit der Operationsparallelität erreichen. Die zur Prozessorarchitektur unter dem Begriff „*instruction sequencing*“ beschriebenen zusätzlichen Regeln definieren Ausnahmen, die durch Hardware unterstützt den erreichbaren Grad an Operationsparallelität verbessern. Zum Beispiel dürfen *Prädikate* und die abhängigen bedingten Operationen trotz der auftretenden Datenabhängigkeit in derselben Gruppe codiert sein.

Durch geeignete Interpretation des Programmiermodells darf z.B. auch der TMS320C62x von Texas Instruments bezüglich der Operationsparallelität als skalierbar bezeichnet werden [185]. Ein Operationspaket endet grundsätzlich mit Bit 0 eines Speicherworts, unabhängig davon, ob das regulär auszuwertende P-Bit gleich Null oder gleich Eins ist. Somit lässt sich prinzipiell das letzte und das erste Operationspaket zweier aufeinander folgender Speicherworte durch das P-Bit als parallel ausführbar kennzeichnen (Bild 3.18a).

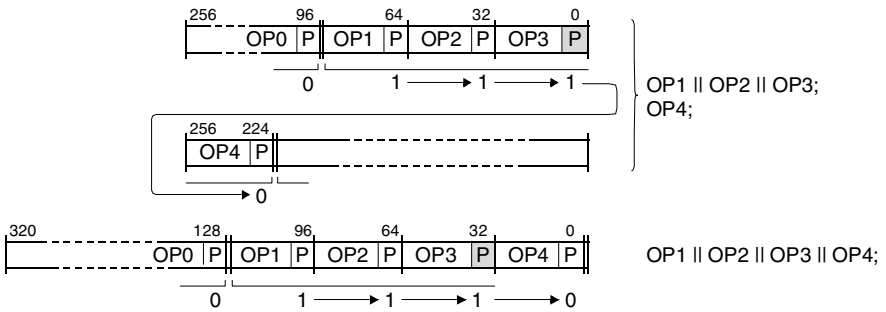

Bild 3.18. Befehle des TMS320C62x. **a** aktuelle Codierung mit maximal acht parallel verarbeitbaren Operationen. **b** Codierung eines möglichen Nachfolgeprozessors mit maximal 10 parallel verarbeitbaren Operationen

Ein Nachfolgeprozessor, der gleichzeitig bis zu 10 Operationen auszuführen vermag, kann nun möglicherweise die ursprünglich sequentiell bearbeiteten Operationspakete parallel ausführen (Bild 3.18b), nämlich dann, wenn die Operationen nach dem Umordnen nicht über zwei benachbarte Speicherworte codiert sind. Dabei ist wichtig, dass die ursprüngliche Semantik erhalten bleibt, d.h. die in Bild 3.18a und 3.18b jeweils rechts dargestellten Operationsfolgen identische Ergebnisse

erzeugen. Ausschließlich in diesem Fall darf Bit 0 des ursprünglich ersten Operationspakets gesetzt sein.

Der Nachteil eines solchen, auf freien Konventionen basierenden Verfahrens ist, dass eine von der Funktionalität her sequentiell zu bearbeitende Operationspaketfolge unerlaubter Weise als parallel codiert werden kann. Weist z.B. OP4 eine Datenabhängigkeit zu OP1, OP2 oder OP3 auf und muss daher sequentiell bearbeitet werden, wird tatsächlich das richtige Ergebnis erzeugt, wenn ein Prozessor verwendet wird, dessen Befehle entsprechend Bild 3.18a codiert sind.

Ein Prozessor, der die Befehle entsprechend Bild 3.18b parallel ausführt, generiert hingegen ein falsches Ergebnis. Der Fehler war von Anfang an in dem entsprechenden Programm codiert. Er wirkt sich jedoch erst mit Verfügbarkeit des Nachfolgeprozessors aus. Im ungünstigsten Fall geschieht dies beim Kunden, oft mit der Konsequenz, dass das Fehlverhalten dem neuen Prozessor angelastet wird. Dies kann im Extremfall von einem Konkurrenten ausgenutzt werden, um den Ruf eines Prozessorherstellers zu schädigen und so Marktanteile zu gewinnen.

Die Einhaltung von Konventionen sollte deshalb nicht freiwillig erfolgen, sondern durch geeignete Verfahren zur Laufzeit überprüft werden. Bemerkenswert ist, dass dies weder beim Itanium 2 von Intel noch beim TMS320C62x von Texas Instruments geschieht. Zwar ist die Operationsparallelität bei diesen Prozessoren beliebig skalierbar, weshalb für eine Laufzeitüberprüfung entsprechend des zugrunde liegenden Regelwerks anscheinend unbegrenzte Ressourcen benötigt werden, jedoch sind andere Prozessormerkmale nicht skalierbar und begrenzen die maximale Parallelität im konkreten Fall deutlich.

Zum Beispiel verfügt der Itanium über 128 allgemein nutzbare Integer-Register, so dass maximal 128 Integer-Register-Register-Operationen parallel ausgeführt werden können, ohne eine Datenabhängigkeit zu verursachen. Dementsprechend lässt sich das Einhalten von Datenflussunabhängigkeit innerhalb eines Operationspakets einfach mit Hilfe eines Schreibbits überprüfen, das jedem Integer-Register angeheftet ist. Die Schreibbits werden zu Beginn der Ausführung eines Operationspakets gelöscht und bei Schreibzugriffen gesetzt. Falls ein Lesezugriff auf ein Register erfolgt, dessen Schreibbit gesetzt ist, liegt somit eine Datenabhängigkeit vor, die durch eine Ausnahmeanforderung als Fehler gemeldet werden kann. Das hier angedeutete Verfahren wird ähnlich in superskalaren Prozessoren verwendet, um Operationen dynamisch zu parallelisieren. Eine detaillierte Beschreibung ist unter dem Begriff *Scoreboarding* in Abschnitt 3.2.2 nachzulesen.

Spekulative Ausführung von Operationen

Für das VLIW-Prinzip ist von wesentlicher Bedeutung, dass es einem *Übersetzer* gelingt, möglichst viele Operationen als parallel ausführbar in den Befehlen zu codieren. Da voneinander abhängige Operationen nicht parallelisierbar sind, ist die grundsätzliche Vorgehensweise hierbei die, unabhängige Operationen aufeinander zuzuschieben und auf diese Weise Gruppen zu bilden, die sich in einzelnen Befehlen

gemeinsam codieren lassen. Das Verschieben von Operationen ist jedoch nur innerhalb vorgegebener Grenzen erlaubt, die von bedingten Sprüngen (Kontrollflussaufspaltungen), Einsprungstellen (Kontrollflusszusammenführungen) oder Operationen, zu denen Datenabhängigkeiten bestehen, gebildet werden.

Trotzdem ist es möglich, Operationen auch über solche Barrieren hinweg vorzuverlegen, sofern man die bewirkte Änderung der Semantik von Programmen durch geeignete Maßnahmen kompensiert. Die verschobene Operation wird hierbei spekulativ verarbeitet und das erzeugte Ergebnis verworfen, wenn sich herausstellt, dass die Ausführung nie hätte erfolgen dürfen. – Übrigens ist die spekulative Ausführung von Operationen eine der Techniken, die sich hinter dem durch Intel und HP mit dem Akronym *EPIC*[1] (*explicitly parallel instruction computing*) umschriebenen Prinzip verbergen [151, 158].

Bei der sog. *Kontrollflussspekulation* (*control speculation*) wird eine in einem bedingt auszuführenden Kontrollpfad codierte Operation vor den zur Bedingungsauswahl benutzten Sprung verschoben und die von ihr erzeugten Ergebnisse zunächst zwischengespeichert. Stellt sich bei der Verarbeitung der Sprungoperation heraus, dass der die verschobene Operation ursprünglich enthaltende bedingte Kontrollpfad nicht zur Ausführung gelangt, sind weitere Aktionen unnötig, da das spekulativ erzeugte Ergebnis nicht endgültig gespeichert, sondern in einem temporären Register lediglich zwischengespeichert wurde. Stellt sich jedoch heraus, dass der die Operation ursprünglich enthaltende bedingte Kontrollpfad tatsächlich ausgeführt wird, ist das erzeugte Zwischenergebnis entweder dem eigentlichen Ziel zuzuweisen oder direkt für weitere Berechnungen zu verwenden (die sog. Weitergabe von Kopien). Obwohl eine Kontrollflussspekulation mit beliebigen Operationen durchführbar ist, benutzt man sie im Zusammenhang mit Register-Register-Operationen i. Allg. nicht, weil sich eine ähnliche Wirkung durch die *Prädikation* erreichen lässt. Dies wird im nächsten Abschnitt genauer erläutert.

Von besonderer Bedeutung ist die Kontrollflussspekulation jedoch für Ladeoperationen, da es auf diese Weise möglich ist, die mit Speicherzugriffen verbundene *Latenzzeiten* hinter parallel ausführbaren unabhängigen Operationen zu verbergen. Ein aus dem Handbuch der Architekturdefinition IA-64 entlehntes, leicht modifiziertes Anwendungsbeispiel ist in Bild 3.19 dargestellt. Teilbild a zeigt als Vorgabe ein Programm, in dem durch die Operation ld8 in Zeile 4 der Inhalt von r4 aus dem Hauptspeicher geladen und in Zeile 5 durch eine Addition weiterverarbeitet wird (die jeweils in einem Befehl als parallel ausführbar codierten Operationen sind durch zwei Semikolons getrennt). Da der Speicherzugriff mehrere Takte erfordert, ist es wegen der bestehenden Datenabhängigkeit notwendig, die Befehlsverarbeitung vor Ausführung der Addition kurzzeitig, nämlich bis zur Verfügbarkeit des benötigten Operanden in r4 zu stoppen (interlock). Gelingt es, den Ausführungszeit-

1. Dahinter verbirgt sich ein Architekturkonzept zur operationsparallelen Verarbeitung von durch einen Übersetzer statisch erzeugten Programmen, wobei der Grad an Parallelität durch spezielle hardware-gestützte Verfahren, wie die spekulative Operationsausführung, die bedingte Operationsausführung, das Software Pipelining usw. verbessert werden. Im Folgenden wird der Begriff nicht weiter verwendet.

punkt der Ladeoperation vorzuverlegen, lässt sich die Speicherlatenzzeit jedoch durch Ausführung paralleler Operationen verdecken. Weil hierbei die Ladeoperation vor die bedingte Sprungoperation in Zeile 3 geschoben werden muss, ist dies jedoch nur spekulativ möglich[1].

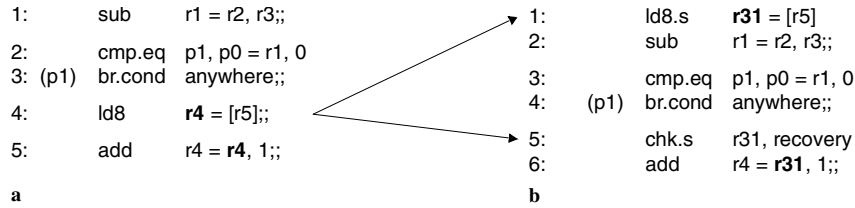

Bild 3.19. Kontrollflussspekulation beim IA-64. **a** Ursprüngliches Programm. **b** Programm nach Modifikation: Die Ladeoperation wird vor dem Sprung ausgeführt und später durch die Check-Operation (chk) überprüft

Das entsprechend abgeänderte Programm ist in Bild 3.19b aufgelistet. Die Ausführung der jetzt in Zeile 1 stehenden Ladeoperation erfolgt parallel zur Subtraktion. Dabei wird statt r4 das temporäre Register r31 geladen, um so sicherzustellen, dass beim möglichen Verzweigen zur Sprungmarke anywhere in Zeile 4 (falls r1 gleich 0 ist) der korrekte Inhalt von r4 weiterhin verfügbar bleibt. Natürlich muss der Inhalt von r31, falls die bedingte Sprungoperation nicht verzweigt, regulär nach r4 kopiert werden. Dies lässt sich jedoch vermeiden, indem man statt r4 in dem entsprechenden Programmzweig fortan r31 als Operand verwendet. Die Addition in Zeile 6 ist aus diesem Grund entsprechend abgeändert.

Ein Problem der spekulativen Ausführung von Ladebefehlen wurde bisher noch nicht erwähnt: Es kann dabei nämlich zu *Ausnahmeanforderungen* kommen, die nur bearbeitet werden dürfen, wenn feststeht, dass die Ladeoperation definitiv ausgeführt wird, wenn also in Bild 3.19b der Sprungbefehl nicht verzweigt. Durch das der Ladeoperation angeheftete Attribut „.s“ wird aus diesem Grund dafür gesorgt, dass Ausnahmeanforderungen nicht unmittelbar bearbeitet, sondern deren Auftreten zunächst nur in einem dem Zielregister zugeordneten Bit protokolliert werden (dem sog. *Not-a-Thing-*, *NaT-Bit*).

Die Auswertung des Bits geschieht durch die *Check-Operation* (chk), und zwar an der Stelle im Programm, an der ursprünglich die Ladeoperation codiert war (weil kein weiteres Ergebnis erzeugt wird, lässt sie sich im Gegensatz zur ursprünglichen Ladeoperation parallel zur nachfolgenden Addition bearbeiten). Falls das NaT-Bit des Registers r31 gelöscht, also keine Ausnahmesituation eingetreten ist, werden von der Check-Operation weitere Aktionen nicht veranlasst. Falls jedoch das NaT-Bit gesetzt ist, wird zur Sprungmarke recovery verzweigt. Dort kann z.B. die Ladeoperation ohne das Attribut „.s“ wiederholt ausgeführt und so dafür gesorgt werden, dass die nun auftretende Ausnahmeanforderung wie gewohnt bearbeitet wird.

1. Die Funktionsweise des Vergleichsbefehls cmp.eq und des bedingten Sprungbefehls br.cond wird im nächsten Abschnitt erläutert. Für den Moment reicht es aus zu wissen, dass hier zur Sprungmarke anywhere verzweigt wird, wenn r1 gleich 0 ist.

Ein ähnliches Verfahren ist bereits Mitte der 80er Jahre in der Trace 7/300 von Multiflow realisiert worden. Da man dort die Ausnahmeanforderung nicht in einem separaten Bit, sondern als reguläres Ergebnis Null im Zielregister protokolliert, ist eine Ladeoperation, die *keine* Ausnahmesituation verursacht, mit der jedoch auf den Wert Null zugegriffen wird, unnötig wiederholt zu bearbeiten. Ein zweiter Nachteil dieser Implementierungsvariante ist, dass sich das spekulative Ergebnis einer Ladeoperation nicht spekulativ weiterverarbeiten lässt, wie bei der Prozessorarchitektur IA-64, wobei das NaT-Bit jeweils von den Operanden zum Ergebnis weitergereicht wird. So kann eine längere Berechnungsfolge spekulativ bearbeitet und mit nur einer einzigen Check-Operation die Gültigkeit des endgültigen Ergebnisses überprüft werden.

Das nicht spekulative Umordnen von Ladeoperationen wird auch durch Speichereoperationen begrenzt. Es kann nämlich nicht ausgeschlossen werden, dass sich aufeinander folgende Schreib- und Lesezugriffe auf den Datenspeicher jeweils auf dieselbe Speicherzelle beziehen und somit eine echte Datenabhängigkeit besteht, die für das korrekte Funktionieren eines Programms zu berücksichtigen ist. Bei der sog. *Datenflussspekulation* (*data speculation*) wird die Reihenfolge von Speichere- und Ladeoperation dennoch verändert und, um ein Fehlverhalten zu vermeiden, anschließend überprüft, ob das Umordnen erlaubt war. Gegebenenfalls muss der Zugriff wiederholt werden.

Ein ebenfalls dem Handbuch zur Prozessorarchitektur IA-64 entlehntes Beispiel hierzu ist in Bild 3.20 dargestellt [70]. Teilbild a zeigt eine Operationsfolge ohne die sog. Datenflussspekulation. Falls in r1 und r4 unterschiedliche Adressen gespeichert sind, besteht zwischen der Speichereoperation st8 in Zeile 1 und der Ladeoperation ld8 in Zeile 2 keine Datenabhängigkeit, so dass ein Umordnen erlaubt ist. Falls jedoch die Inhalte von r1 und r4 übereinstimmen, sind die beiden Operationen voneinander abhängig und lassen sich daher nicht in ihrer Reihenfolge verändern, ohne eine Änderung der Wirkungsweise des Programms zu verursachen (der Inhalt von r3 wird in diesem Fall gleich dem von r2 sein).

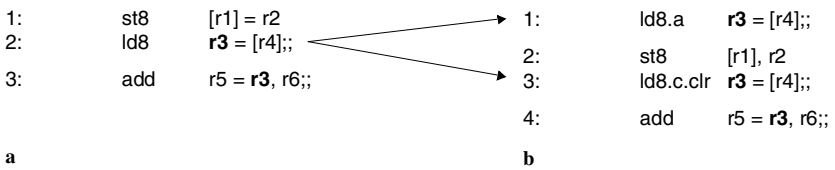

Bild 3.20. Datenflussspekulation beim IA-64. **a** Ursprüngliches Programm. **b** Programm nach Modifikation: Die Ladeoperation wird vor der Speichereoperation ausgeführt und anschließend überprüft, ob der zuvor geladene Operand dabei überschrieben wurde

Trotz der potentiellen Datenabhängigkeit kann die Ladeoperation spekulativ vor der Speichereoperation ausgeführt werden. Eine entsprechende, aus Bild 3.20a hergeleitete Operationsfolge ist in Bild 3.20b dargestellt. Wegen des der Ladeoperation ld8 in Zeile 1 angehefteten Attributs „.a“ wird zusätzlich zum Lesezugriff auf den Datenspeicher ein Eintrag in einer als Advanced Load Address Table (*ALAT*) bezeichneten cache-ähnlich organisierten Tabelle reserviert, um darin jeweils zum

verwendeten Zielregister die Datenspeicheradresse sowie das Zugriffsformat zu protokollieren[1]. Mit jedem nachfolgenden Speicherezugriff wird die ALAT entsprechend der für den Schreibzugriff verwendeten realen Adresse und des Zugriffsformats durchsucht. Falls sich dabei ein passender Eintrag findet, bedeutet dies, dass die zum Eintrag gehörende spekulativ ausgeführte Ladeoperation eine Datenabhängigkeit zur Speichereoperation aufweist, das gelesene Datum also ungültig ist. Als Folge wird der entsprechende Eintrag aus der ALAT gelöscht.

In Zeile 3 kommt es wegen der *Load-Check-Operation* (ld8.c) erneut zu einer Durchsuchung der ALAT, diesmal jedoch nach dem verwendeten Zielregister. Falls ein Eintrag gefunden wird, ist dies ein Beleg dafür, dass keine Speichereoperation ausgeführt wurde, die das spekulativ gelesene Datum hätte verändern können. Die Ladeoperation in Zeile 3 lässt sich daher, ohne erneut auf den Datenspeicher zugreifen zu müssen, abschliessen. Sollte jedoch kein passender Eintrag in der ALAT identifiziert werden, so wurde dieser entweder durch eine Speichereoperation entfernt oder der verfügbare Platz in der Tabelle hat nicht ausgereicht. In beiden Fällen wird der Lesezugriff auf den Datenspeicher wiederholt.

Bemerkt sei, dass das in Zeile 3 zusätzlich angegebene Attribut „.clr“ das Löschen des in der ALAT stehenden Eintrags bewirkt. Dies ist sinnvoll, wenn sich nicht mehrere Zugriffe auf denselben spekulativ gelandenen Operanden beziehen. Ohne dieses Attribut wird der bestehende Eintrag ggf. automatisch gelöscht, sobald die ALAT vollständig gefüllt ist und, wegen einer spekulativ auszuführenden Ladeoperation, ein neuer Eintrag reserviert werden soll. Natürlich hat dies zur Konsequenz, dass möglicherweise ein Lesezugriff auf den Datenspeicher zu wiederholen ist, auf dessen Zugriffsadresse zwischenzeitlich nicht zugegriffen wurde, für den diese Wiederholung also unnötig ist. Die Semantik des Programms bleibt dabei in jedem Fall sicher erhalten.

Sprungoperationen und bedingte Operationsausführung

Die durch Sprungoperationen in Fließbandprozessoren verursachten Probleme sind bereits in Abschnitt 2.2.4 eingehend erläutert worden. Im Folgenden seien jedoch noch einige im Zusammenhang mit VLIW-Prozessoren stehende Besonderheiten behandelt: Genau wie bei skalaren Fließbandprozessoren können Sprungoperationen Konflikte verursachen, bei deren Auftreten bereits in Bearbeitung befindliche Operationen abgebrochen werden müssen. Wegen der Parallelverarbeitung ist jedoch die Anzahl der in einem VLIW-Prozessor zu verwerfenden Operationen um ein Vielfaches höher als in einem bezüglich der Fließbandaufteilung ähnlich realisierten skalaren Prozessor.

Aus diesem Grund ist die Sprungvermeidung durch bedingte Operationsausführung, also die Prädikation, im Programmiermodell der meisten VLIW-Prozessoren definiert, so z.B. im TriMedia TM-1300 von Philips [141], im TMS320C62x von Texas

1. Tatsächlich wird im ersten Takt zunächst nur die aus der virtuellen Adresse extrahierbare Bytenummer gespeichert. Sie wird jedoch um die Rahmennummer ergänzt, sobald die Adressumsetzung der Speicherverwaltungseinheit vollständig durchlaufen wurde.

Instruments [185] und im Itanium 2 von Intel [70, 75, 78]. Als Nebeneffekt lässt sich dabei vorteilhaft nutzen, dass durch das Entfernen der Sprungoperationen Kontrollflussspekulationen unnötig werden und somit die Prädikation auch eine Technik ist, um das Maß an durchschnittlicher Operationsparallelität in den Befehlen zu erhöhen.

Bild 3.21a bis c zeigt hierzu ein Beispiel für die im Itanium 2 realisierte Prozessorarchitektur IA-64[1]. Ohne bedingte Operationsausführung lässt sich das C-Programm links zu dem in Bild 3.21b dargestellten Assemblerprogramm übersetzen. Mit r1 gleich Null wird durch die Vergleichsoperation cmp.eq das Prädikatsbit p1 gelöscht und die im then-Zweig stehenden Operationen in den Zeilen 3 und 4 ausgeführt. Mit r1 ungleich Null wird zur Sprungmarke else verzweigt und die Operationen in den Zeilen 7 und 8 bearbeitet. Die Sprungmarke end kennzeichnet das Ende des gesamten if-then-else-Konstrukts.

Eine durch Anwendung der bedingten Operationsausführung optimierte Variante des Programms ist in Bild 3.21c aufgelistet. Durch die Vergleichsoperation in Zeile 1 werden zunächst die Prädikatsbits p1 und p2 entsprechend der zu prüfenden Bedingung bzw. komplementär dazu gesetzt. Die Operationen des then- und des else-Zweigs lassen sich anschließend parallel bedingt bearbeiten. Mit r1 ungleich Null wird z.B. p1 gesetzt und p2 gelöscht, so dass die Operationen des else-Zweigs in den Zeilen 4 und 5, nicht jedoch die des then-Zweigs in den Zeilen 2 und 3 ausgeführt werden. Mit r1 gleich Null kehrt sich die Situation ins Gegenteil.

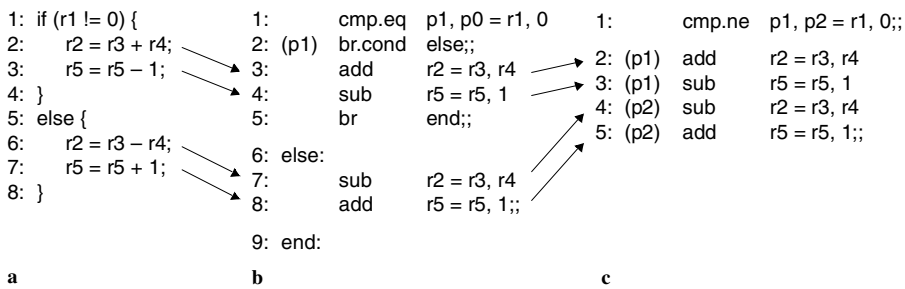

Bild 3.21. Bedingte Ausführung von Operationen beim IA-64. **a** Ursprüngliches C-Programm. **b** Assemblerprogramm ohne bedingte Operationsausführung. **c** Assemblerprogramm mit bedingter Operationsausführung

Die fünf Operationen sind hierbei in zwei sequentiell zu bearbeitenden Befehlen codiert. Dies ist deshalb erforderlich, weil die Bedingungsergebnisse in p1 und p2 zuerst erzeugt werden müssen, bevor sie sich durch die im zweiten Befehl parallel codierten Operationen auswerten lassen. Es gibt jedoch einen Sonderfall, der es ermöglicht, Operationen trotz Datenabhängigkeit als parallel ausführbar zu codie-

1. Der Itanium 2 wird hier deshalb verwendet, weil dasselbe Beispiel für den TriMedia TM-1300 bzw. den TMS320C62x wegen der großen Anzahl von zu berücksichtigenden Verzögerungsstufen (Delayslots) von Sprungoperationen sehr unübersichtlich wäre. Der TriMedia TM-1300 führt nach einer Sprungoperation die drei, der TMS320C62 sogar die fünf unmittelbar folgenden Befehle unbedingt aus.

ren, nämlich eine Kombination aus Vergleichs- und davon abhängiger Sprungoperation. Dies ist der Grund, weshalb sich die ersten beiden Operationen im Assemblerprogramm aus Bild 3.21b in einem Befehl codieren lassen.

Ein weiterer im Programmiermodell des IA-64 definierter Sonderfall ist, dass Operationen, die dasselbe Zielregister verwenden und sich aufgrund der Prädikate komplementär verhalten, dennoch parallel ausgeführt werden können (wie dies in Bild 3.21c z.B. mit den Operationen in den Zeilen 2 und 4 geschieht). Hingegen ist dies nicht möglich, wenn die sich gegenseitig ausschließenden Operationen eine gemeinsame Verarbeitungseinheit durchlaufen, und zwar deshalb nicht, weil die Zuordnung der Operationen und der Verarbeitungseinheiten zu einem Zeitpunkt geschieht, zu dem i.Allg. noch nicht bekannt ist, ob die Operationen wirklich ausgeführt werden sollen oder nicht (normalerweise nach dem Decodieren eines Befehls). Falls also im Itanium 2 nur eine Additionseinheit zur Verfügung stehen würde, könnten in Bild 3.21c lediglich die Operationen in den Zeilen 2, 3 und 4 parallel codiert werden, obwohl tatsächlich nur eine der beiden Additionen in den Zeilen 2 und 4 zur Ausführung kommt.

Ein Nachteil der Prädikation im Vergleich zur kombinierten Verwendung unbedingter Verknüpfungs- und bedingter Sprungoperationen ist, dass durch sie zwar die Anzahl der in einem Befehl codierbaren Operationen i.Allg. vergrößert, die Anzahl der effektiv parallel ausführbaren Operationen jedoch möglicherweise sogar vermindert wird. Zum Beispiel werden von den in Bild 3.21c im zweiten Befehl codierten vier Operationen in jedem Fall nur zwei Operationen tatsächlich bearbeitet. Es ist sogar möglich, dass in einem if-then-Konstrukt mit Prädikation bei nicht erfüllter Abfragebedingung die Ausführung des leeren else-Zweigs mehrere Takte Zeit erfordert, weil nämlich die bedingten Operationen des if-Zweigs, ohne Ergebnisse zu erzeugen, dennoch verarbeitet werden müssen. Stellt sich in einem solchen Fall heraus, dass die Abfragebedingung in der Mehrzahl der zur Laufzeit auftretenden Fälle nicht erfüllt ist, muss eine permanente Strafe in Kauf genommen werden. Das Programm würde somit langsamer ausgeführt als unter Verwendung bedingter Sprungoperationen, deren Verhalten sich dynamisch korrekt vorhersagen ließe.

◤ Bemerkung. Eine andere Möglichkeit Sprungoperationen zu vermeiden besteht darin, Operationsfolgen, in denen Verzweigungen auftreten, durch neu definierte Operationen zu lösen. Zum Beispiel verfügt der an der TU Berlin entwickelte Nemesis X über eine Operation, mit der sich der Maximalwert der übergebenen Operanden ermitteln lässt. Von vielen anderen Prozessoren wäre hierzu ein Vergleich, ein bedingter Sprung und eine Zuweisung auszuführen. Ein anderes Beispiel wurde bereits in Abschnitt 1.1.7 (siehe Beispiel 1.4) für Prozessoren mit PowerPC-Architektur von Motorola bzw. IBM beschrieben. Mit Hilfe spezieller Befehle lassen sich zusammengesetzte *Bedingungen* wie z.B. a = b UND c ≠ d ohne Sprung berechnen.

Eine ähnliche Möglichkeit ist im Programmiermodell der IA-64-Architektur definiert. Es ist nämlich möglich, mehrere Vergleichsoperationen mit demselben Zielregister parallel auszuführen, wenn die Attribute „.and“ oder „.or“ an den Befehl angeheftet sind (siehe Bild 3.22). Technisch wird die logische Verknüpfung der Vergleichsergebnisse realisiert, indem man bei einer Und-Operation die Null bei einer Oder-Operation die Eins in das Ergebnisregister schreibt (vergleichbar einem wired-and bzw. wired-or). Der jeweils komplementäre Wert wird dabei ignoriert. Als Konsequenz ist jedoch in Kauf zu nehmen, dass vor Ausführung der Vergleiche das als Ziel verwendete Prädikatsbit initialisiert werden muss.

```
1:          cmp.eq      p1, p0 = r0, r0  ;;  // p1 mit 1 initialisieren
2:          cmp.eq.and  p1, p0 = r1, r2      // Falls r1 = r2 ist, p1 gesetzt lassen
3:          cmp.ne.and  p1, p0 = r3, r4      // Falls zusätzlich auch r3 ≠ r4 ist, p1 gesetzt lassen
4: (p1)     pr.cond     anywhere         ;;  // Verzweigen, wenn r1 = r2 UND r3 ≠ r4
```

Bild 3.22. Zusammengesetzte Bedingung bei einem Prozessor mit IA-64-Architektur ◢

Das in Bild 3.21b und erneut in Bild 3.23a aufgelistete Assemblerprogramm lässt sich auch ohne Sprungvermeidung optimiert in zwei Befehlen codieren und ist somit vergleichbar schnell wie das in Bild 3.21c dargestellten Assemblerprogramm ausführbar. Hierzu werden die im if-Zweig stehenden Operationen in den Zeilen 3 und 4 spekulativ vor die bedingte Sprungoperation in Zeile 2 verschoben und anschließend die beiden nun direkt aufeinander folgenden Sprungoperationen aus den Zeilen 2 und 5 zu einer einzelnen Sprungoperation zusammengefasst. Letzteres erfordert, dass die dabei ausgewertete Bedingung komplementiert wird, also r1 und Null nicht auf Gleichheit, sondern auf Ungleichheit überprüft werden.

Das Ergebnis der Optimierung zeigt Bild 3.23b. Falls r1 ungleich Null ist, kommt es zunächst zur Ausführung der Operationen des if-Zweigs in den Zeilen 2 und 3, um anschließend zum Ende des Assemblerprogramms zu verzweigen. Falls r1 umgekehrt gleich Null ist, werden die Operationen des if-Zweigs ebenfalls ausgeführt, die spekulativ erzeugten Ergebnisse jedoch durch die Operationen des else-Zweigs in den Zeilen 5 und 6 überschrieben. Das Assemblerprogramm ist in zwei VLIW-Befehlen codierbar und lässt sich bei korrekter Sprungvorhersage in einem Takt, nämlich wenn der then-Zweig durchlaufen wird, und in zwei Takten, wenn der else-Zweig durchlaufen wird, ausführen (letzteres sollte der seltener auftretende Fall sein). Zum Vergleich: Das in Bild 3.21c dargestellte Assemblerprogramm benötigt immer zwei Takte.

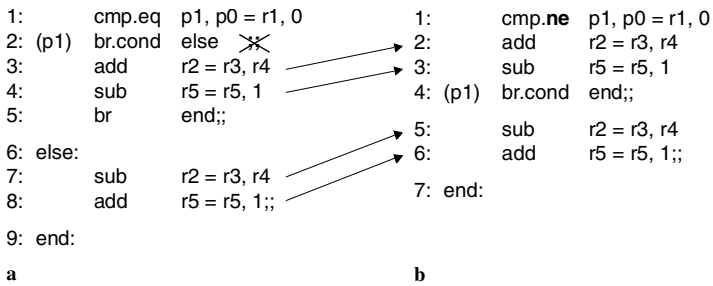

Bild 3.23. Optimierung eines if-then-else-Konstrukts, ohne Sprungoperationen zu vermeiden. **a** Nicht optimiertes Assemblerprogramm (entspricht Bild 3.21b). **b** Optimiertes Assemblerprogramm

In diesem Beispiel wirkt sich positiv aus, dass die Operationen im if- und im else-Zweig jeweils dieselben Zielregister verwenden. Wäre dies nicht der Fall, dürften die spekulativ ausgeführten Operationen die Inhalte der Register r2 und r5 nur dann verändern, wenn andere auf das if-then-else-Konstrukt folgende Operationen nicht lesend darauf zugreifen. Gegebenenfalls müssen die spekulativen Ergebnisse zuerst in temporären Registern zwischengespeichert und später, z.B. im if-Zweig, den eigentlichen Zielregistern zugewiesen werden. Die Bearbeitung eines derart modifi-

zierten Programms erfordert wegen der bestehenden Datenabhängigkeiten natürlich mehr als einen Taktzyklus Zeit und ist somit aufwendiger und ggf. auch langsamer als die des in Bild 3.23a dargestellten nicht optimierten Assemblerprogramms.

Nun ist es möglich, ohne Optimierung eine Wirkung entsprechend Bild 3.23b zu erzielen, wenn man berücksichtigt, dass die Semantik einer Operationsfolge bei Prozessoren mit IA-64-Architektur unabhängig davon ist, ob die einzelnen Operationen streng sequentiell oder teilweise parallel ausgeführt werden. Da es außerdem erlaubt ist, in einem Befehl mehrere Sprungoperationen zu codieren, impliziert dies, dass alle auf eine *verzweigende* Sprungoperation folgenden, in einem Befehl parallel codierten Operationen automatisch in ihrer Bearbeitung annulliert werden. Mit anderen Worten: Das doppelte Semikolon in Zeile 2 des Assemblerprogramms in Bild 3.23a ist nicht erforderlich. Falls die Sprungoperation in Zeile 2 nicht verzweigt, werden parallel die Operationen des if-Zweigs ausgeführt. Falls die Sprungoperation verzweigt, werden die Operationen in den Zeilen 3 bis 5 annulliert und in einem zweiten Takt die Operationen des else-Zweigs bearbeitet.

Das doppelte Semikolon am Ende von Zeile 5 in Bild 3.23a ist selbstverständlich weiterhin erforderlich, weil die Operationen in den Zeilen 3 und 7 auf dieselben Register schreibend zugreifen, wie die Operationen in den Zeilen 7 und 8. Wäre dies jedoch nicht der Fall, ließe sich sogar der else-Zweig parallel zum if-Zweig in einem Befehl codieren. Dass hierbei die Sprungmarke else innerhalb eines Befehls auftreten würde, ist kein Problem. Die einzige für die Adressen von Sprungmarken geltende Restriktion ist, dass sie an den sog. *Bundles*, in denen jeweils drei Operationen codiert sind, ausgerichtet sein müssen (ein Befehl kann mehrere Bundles enthalten).

Die hier beschriebene Definition der Semantik von Sprungoperationen ist ein herausragendes Merkmal von Prozessoren mit IA-64-Architektur, das in anderen nach dem VLIW-Prinzip arbeitenden Prozessoren nicht vorzufinden ist. Der Trace 7/300 von Multiflow oder der Nemesis X der TU Berlin kann z.B. nur eine Sprungoperation pro Befehl ausführen, was den Vorteil hat, einfach realisierbar zu sein. Der TMS320C62x von Texas Instruments oder der TriMedia TM-1300 von Philips können zwar mehrere Sprungoperationen in einem Befehl gleichzeitig bearbeiten, die parallel codierten anderen Operationen werden jedoch unabhängig davon, ob verzweigt oder nicht verzweigt wird, in jedem Fall ausgeführt. Dies gilt auch für die Sprungoperationen selbst, weshalb nur eine von ihnen wirklich verzweigen darf (das Verhalten der Prozessoren ist nicht definiert, wenn mehrere Sprungoperationen gleichzeitig verzweigen).

◤ Bemerkung. Die parallele Ausführung von Operationen, die vor und hinter einem nichtverzweigenden Sprung codiert sind, erfordert normalerweise die spekulative Vorverlegung der auf den Sprung folgenden Operationen. Wie beschrieben, ist eine solche Optimierung für Prozessoren mit IA-64-Architektur unnötig, wenn ein Prozessor mit IA-64-Architektur verwendet wird, da alle Operationen, die zusammen mit einem Sprung als parallel ausführbar codiert sind, zwar gleichzeitig gestartet, die auf einen Sprung folgenden Operationen beim Verzweigen jedoch annulliert werden.

Möchte man umgekehrt Operationen parallelisieren, die nicht auf die Sprungoperation, sondern auf das Sprungziel folgen, ist ein Umordnen von Operationen auch mit einem IA-64-konformen Prozessor nicht vermeidbar. Dabei wird durch Prädikation dafür gesorgt, dass die verschobenen Operationen tatsächlich nur zur Ausführung kommen, wenn die Sprungbedingung erfüllt ist. In Bild 3.24

ist dies an einem Beispiel dargestellt, wobei die Operationsfolge rechts durch eine optimierende Transformation aus der Operationsfolge links erzeugt wurde (das Ergebnis ist nicht optimal, wie sich durch Vergleich mit Bild 3.21c überprüfen lässt).

Eine bessere Möglichkeit des Umgangs mit einer solchen Problemstellung wurde an der TU Berlin entwickelt. Statt die Sprungoperation br.cond in Zeile 2 von Bild 3.24a zu codieren und damit das Ende eines parallel verarbeitbaren Operationspakets zu definieren, wird ein dem Programmiermodell hinzuzufügender Befehl *join.cond* (in Bild 3.24a im Kasten angedeutet) verwendet. Er sorgt dafür, dass die auf das Sprungziel folgenden Befehle ggf. parallel zu denen ausgeführt werden, die im aktuellen Befehlspaket zusammen mit dem bedingten Sprung codiert sind. Durch eine solche Modifikation lässt sich die Entscheidung, ob die Operationen des if- (in den Zeilen 3 bis 5) oder die des else-Zweigs (in den Zeilen 6 bis 8) parallel zum Vergleich bzw. dem bedingten Sprung ausgeführt werden, von der Übersetzungszeit zur Laufzeit verschieben.

Zwar ist die Befehlsfolge wegen des den if-Zweig abschließenden unbedingten Sprungs geringfügig umfangreicher (sieben statt sechs Operationen), sie ist dafür jedoch schneller bearbeitbar, wenn die Sprungentscheidung korrekt vorhergesagt wird (ein Takt statt zwei Takte). Ein wichtiger Nebeneffekt ist außerdem, dass sich der Ressourcen-Bedarf durch Verwendung der join-Operation vermindern lässt. Während nämlich die Operationsfolge in Bild 3.24b für die Additionen und Subtraktionen vier Funktionseinheiten zur Bearbeitung erfordert, sind dies in Bild 3.24a nach Modifikation nur noch zwei Funktionseinheiten. Die Einsparung ist z.B. nutzbar, um das Maß an Parallelverarbeitung in einem Programm zu verbessern.

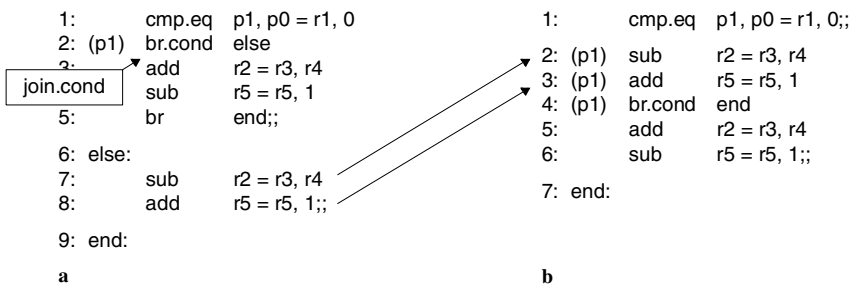

Bild 3.24. Optimierung eines if-then-else-Konstrukts. **a** Das nicht optimierte Assemblerprogramm. **b** Assemblerprogramm nach Optimierung des else-Zweigs

Die Erweiterung existierender Prozessoren um die Fähigkeit zum Umgang mit join-Operationen ist sehr einfach, sofern man berücksichtigt, dass die vor und nach dem Verzweigungspunkt codierten Operationen entsprechend der Definitionen des Programmiermodells der IA-64-Architektur auch sequentiell ausgeführt werden können und somit die Funktionalität der join-Operation dieselbe ist, wie die einer herkömmlichen Sprungoperation. Falls jedoch tatsächlich über mehrere verzweigende Sprünge hinweg parallelisiert werden soll, sind einige komplexe Architekturerweiterungen notwendig: Zum Beispiel müssen mehrere Sprungentscheidungen vorhergesagt und gleichzeitig auf unterschiedliche Adressbereiche zugegriffen werden können, was neben einer Mehrfach-Sprungvorhersageeinheit einen Multiport-Cache oder Trace-Cache erfordert (Abschnitt 2.2.4 ff. und 3.2.4). ◢

Optimierungstechniken (trace scheduling, loop unrolling, software pipelining)

Trace-Scheduling. Die Geschwindigkeit, mit der VLIW-Prozessoren arbeiten, wird wesentlich durch das Maß an Parallelität beeinflusst, welches statisch in den Befehlen codiert ist. Es überrascht daher nicht, dass die VLIW-Prozessorarchitektur durch

eine Veröffentlichung von Joseph Fisher Anfang der 80er Jahre populär geworden ist, in der der Autor ein Verfahren zur statischen Maximierung der Operationsparallelität in entsprechend realisierten Prozessoren beschrieb – nämlich das sog. Trace Scheduling [45, 101][1].

Zur Vorgehensweise: Zunächst analyisert man statisch oder dynamisch, z.B. durch Anfertigung eines sog. Profiles, das Laufzeitverhalten eines zu optimierenden Programms, und wählt entsprechend der gewonnenen Erkenntnisse einen zyklenfreien Teilpfad (also eine Befehlsfolge) aus, der möglichst häufig zur Ausführung gelangt (trace). Durch Verschieben der Sprungoperationen und Sprungmarken zu den Enden des Teilpfads wird dafür gesorgt, dass sich die verbleibenden Operationen unter Beachtung der Datenabhängigkeiten umordnen lassen. Dabei werden unabhängige Operationen so verschoben, dass sie in den Befehlen parallel codierbar sind. Wegen der im Vergleich zum ursprünglichen Programm deutlich größeren Anzahl von für die Optimierung verfügbaren Operationen lässt sich so das erreichbare Maß an Operationsparallelität steigern.

Bild 3.25a bis c verdeutlicht die Vorgehensweise beim Trace-Scheduling. Zunächst zeigt Bild 3.25a, wie sich eine bedingte Sprungoperation in drei Schritten verschieben und die in dieser Weise neu angeordneten Operationen parallelisieren lassen. Im ersten Schritt wird der zu optimierende Programmpfad A bis E (trace) *separiert* (im Bild grau unterlegt), wobei zur bedingten Sprungoperation C nur das wahrscheinlichere, z.B. durch Messungen (*profiling*) ermittelte Sprungziel, berücksichtigt wird. Anschließend werden die Operationen A und B in die auf die bedingte Sprungoperation folgenden Pfade kopiert und so die Sprungoperation C an den Anfang des separierten Programmpfads verschoben. Dies ist nur möglich, wenn C keine Abhängigkeiten zu A oder B aufweist, was hier jedoch angenommen werden soll.

Weil verzweigende Sprungoperationen komplizierter zu verarbeiten sind als nichtverzweigende Sprungoperationen, wird C außerdem in seiner Wirkung invertiert, so dass eine Verzweigung innerhalb des separierten Programmpfads nicht mehr erforderlich ist (Bild 3.25a Mitte). Im letzten Schritt werden schließlich die auf die Verzweigung folgenden Operationen paarweise vertauscht, und zwar in der Weise, dass aufeinander folgende Operationen möglichst keine Abhängigkeiten zueinander aufweisen. Die gebildeten Gruppen lassen sich schließlich in Befehlen parallel codieren. So sind nach diesem Schritt die Operationen A und D sowie B und E parallel ausführbar, was zu Beginn der Optimierung nicht möglich gewesen ist.

In Bild 3.25b ist exemplarisch dargestellt, wie sich eine Sprungmarke zum Ende eines separierten Programmpfads verschieben lässt. Dabei werden zuerst die Operationen C und D jeweils in die von A und X ausgehenden Teilpfade kopiert und der transformierte Programmpfad anschließend so umgeordnet, dass sich die hier als unabhängig angenommenen Operationen A, C und D parallel in einem Befehl codieren lassen. Im Prinzip ist es möglich, auch C' und D' in dem von X ausgehenden Programmpfad als parallel ausführbar zu codieren. Dies ist jedoch unnötig, wenn die entsprechende Operationsfolge nur selten ausgeführt wird.

1. Das Verfahren wurde zuvor bereits zur Optimierung von Mikroprogrammen verwendet [44].

Es sei noch angemerkt, dass B aus dem optimierten Programmpfad entfernt werden kann, wenn es sich hierbei um eine unbedingte Sprungoperation handelt. Der ursprünglich aus fünf Operationen bestehende, separierte Programmpfad lässt sich somit in nur zwei Befehlen codieren. – Als Ergänzung zu den vorangehenden Ausführungen ist in Bild 3.25c ein etwas komplexeres Beispiel dargestellt, in dem eine Sprungmarke und eine Sprungoperationen aneinander vorbeigeschoben werden. (Die einzelnen Schritte sollten bei strikter Anwendung der vorangehend beschriebenen Regeln leicht nachvollziehbar sein.)

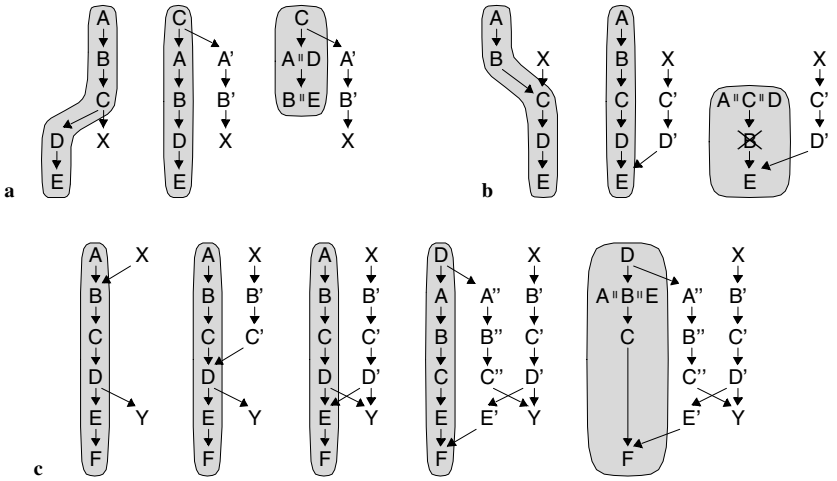

Bild 3.25. Optimierung eines sequentiellen Operationsfolge durch Trace Scheduling (in Anlehnung an [101]). **a** Verschieben einer bedingten Sprungoperation an den Anfang eines Pfads. **b** Verschieben einer Sprungmarke an das Ende eines Pfads. **c** Überlappendes Verschieben einer Sprungoperation und -marke

Schleifenabrollen (loop unrolling). Beim Trace-Scheduling endet die Separation eines Programmpfads wegen der zu erfüllenden Voraussetzung nach Zyklenfreiheit definitiv dann, wenn ein Sprung zu einer zuvor separierten Operation erkannt wird. Dies ist z.B. am Ende einer im separierten Programmpfad liegenden *Programmschleife* der Fall. Beim Trace-Scheduling ist somit das maximal erreichbare Maß an Operationsparallelität innerhalb einer Schleife durch die Anzahl der Operationen im Schleifenrumpf begrenzt.

Die Anzahl der berücksichtigbaren Operationen kann jedoch erhöht werden, indem eine Schleife teilweise oder vollständig abgerollt wird, wie in Bild 3.26a bis c angedeutet. Aus der als Vorgabe dargestellten Schleife in Bild 3.26a – die hier etwas ungewöhnlich mit einer Anweisung zur Überprüfung der Schleifenabbruchbedingung beginnt[1], in der optional auch der Schleifenzähler dekrementiert wird (durch „--“ symbolisiert) – lässt sich durch Abrollen von jeweils drei Schleifendurchläufen

1. Normalerweise würde am Ende der Schleife nicht unbedingt zur Zeile 1, sondern bedingt zur Zeile 2 gesprungen werden. Der Einfachheit halber wurde diese Optimierung hier jedoch weggelassen.

das in Bild 3.26b skizzierte Programm erzeugen. Ein für das Trace-Scheduling daraus separierter Programmpfad enthält eine deutlich höhere Anzahl parallelisierbarer Operationen als ein vor Abrollen der Schleife separierter Programmpfad.

Die mehrfache Überprüfung der Schleifenabbruchbedingung in Bild 3.26b lässt sich vermeiden, wenn die Anzahl der Schleifendurchläufe vor Eintritt in die Schleife bekannt ist. Ein entsprechend optimiertes Programm zeigt Bild 3.26c. Innerhalb des Schleifenrumpfs zwischen den Zeilen 5 und 9 wird die Schleifenabbruchbedingung ein einziges Mal überprüft. Die Semantik der in Bild 3.26a dargestellten Schleife bleibt dabei trotzdem erhalten, wenn die Anzahl der Schleifendurchläufe hier durch Drei teilbar ist, was jedoch nicht immer gilt. Aus diesem Grund wird vor dem Schleifeneintritt in den Zeilen 1 und 3 die Abbruchbedingung auf Teilbarkeit durch Drei überprüft und der ursprüngliche Schleifenrumpf ggf. ein oder zweimal außerhalb der Schleife, nämlich in Zeile 2 bzw. Zeile 4 ausgeführt.

Die Überprüfung der Teilbarkeit durch Drei vor Bearbeitung der abgerollten Schleife bezeichnet man als *Präkonditionierung* [101]. Bei der ebenfalls möglichen sog. *Postkonditionierung* werden unbearbeitete Schleifendurchgänge im Anschluss an die abgerollte Schleife ausgeführt. Es sei angemerkt, dass die durch Nutzung der Prä- oder Postkonditionierung abgerollten Schleifen nicht schneller bearbeitet werden müssen als ein Programm, das entsprechend Bild 3.26b optimiert ist, da sich die bedingten Sprungoperationen oft parallel zu Operationen des Schleifenrumpfs ausführen lassen.

```
1:      loop:  if -- goto end
2:             body
3:             goto   loop
4:      end:
```

a

```
1:      loop:  if -- goto end
2:             body
3:             if -- goto end
4:             body'
5:             if -- goto end
6:             body"
7:             goto   loop
8:      end:
```

b

```
1:             if mod -- goto loop
2:             body
3:             if mod -- goto loop
4:             body'
5:      loop:  if -- goto end
6:             body"
7:             body"'
8:             body""
9:             goto   loop
10:     end:
```

c

Bild 3.26. Optimierung einer Schleife durch Abrollen. **a** Die nichtoptimierte Schleife. **b** Die über drei Iterationsschritte einfach abgerollte Schleife. **c** Die über drei Iterationsschritte abgerollte präkonditionierte Schleife

Software-Fließbandverarbeitung (software-pipelining). Ein weiteres Verfahren zur Erhöhung der in Schleifen codierbaren Operationsparallelität ist die sog. Software-Fließbandverarbeitung, bei der man aufeinander folgende Schleifenrümpfe überlappend parallel ausführt. Bild 3.27 zeigt hierzu ein aus dem Handbuch zur Prozessorarchitektur IA-64 von Intel und HP stammendes Beispiel [70]. Die Operationen in den Zeilen 1 bis 3 von Bild 3.27a sind, wegen der über r4 und r7 bestehenden Datenabhängigkeiten, nicht parallel ausführbar. Sobald jedoch das Ergebnis der Ladeoperation des ersten Schleifendurchlaufs feststeht, kann gleichzeitig zur Addition bereits ein in einem späteren Schleifendurchlauf benötigter Operand aus dem Hauptspeicher gelesen und auch verarbeitet werden. Das in dieser Weise realisierte

Software-Fließband ist schließlich gefüllt, wenn mit dem nächsten Takt die letzte Operation des ersten Schleifendurchlaufs ausgeführt wird, wobei parallel dazu die Addition des zweiten und die Ladeoperation eines dritten Schleifendurchlaufs bearbeitet werden. Mit jedem zusätzlichen Takt wird ein Schleifendurchlauf beendet und ein weiterer begonnen. Die maximal erreichbare Operationsparallelität ist somit gleich der Anzahl der im Schleifenrumpf codierten Operationen, was einem Durchsatz von einem Schleifendurchgang pro Takt entspricht.

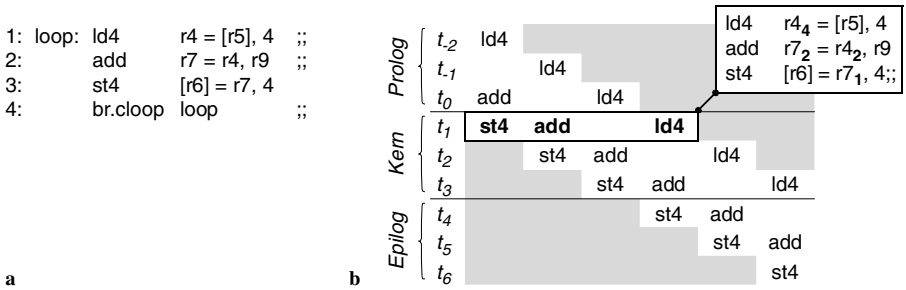

Bild 3.27. Optimierung einer Schleife durch Software-Fließbandverarbeitung. **a** Die zu optimierende Schleife. **b** Zeitliche Abfolge der einzelnen Operationen. Zu jedem Zeitpunkt werden parallel Operationen aus unterschiedlichen Iterationsschritten der Schleife ausgeführt

Der zeitliche Verlauf bei Verarbeitung der in Bild 3.27a dargestellten Schleife ist für sechs Wiederholungen in Bild 3.27b angedeutet, wobei vorausgesetzt wird, dass zwei gleichzeitige Zugriffe auf den Datenspeicher bzw. -cache möglich sind und die Ladeoperation eine Latenzzeit von zwei Takten aufweist. Der erste Schleifendurchlauf wird zwischen t_{-2} und t_1, alle weitere Schleifendurchläufe werden jeweils um einen Takt zeitversetzt, ausgeführt. Die Bearbeitung der gesamten Schleife lässt sich in drei grundsätzliche Phasen unterteilen:

- In der ersten, als *Prolog* bezeichneten Phase wird das Software-Fließband mit Operationen gefüllt. Sie endet, sobald die Ausführung der letzten Operation des ersten Schleifendurchlaufs beginnt (in Bild 3.27b zum Zeitpunkt t_1).
- In der sich anschließenden zweiten Phase, dem *Kern* (*kernel*), erreicht man die maximale Operationsparallelität, wobei in jedem Takt ein Schleifendurchlauf beendet und ein neuer begonnen wird.
- In der letzten Phase, dem *Epilog*, werden schließlich die gestarteten Schleifendurchläufe abgeschlossen, jedoch keine weiteren gestartet. Die in den Befehlen codierbare Operationsparallelität nimmt in dieser Phase wieder ab, bis schließlich die letzte Operation des letzten Schleifendurchlaufs ausgeführt worden ist.

Da die parallel codierten Operationen unterschiedlichen Schleifendurchläufen zuzuordnen sind, müssen überall dort, wo im ursprünglichen Programm *Datenabhängigkeiten* bestehen, unterschiedliche Register benutzt werden. In Bild 3.27b ist dies für den vierten Schleifendurchlauf in dem Kasten oben rechts angedeutet. So bezieht sich die Ladeoperation auf den vierten und die Addition auf den zweiten Schleifen-

durchlauf, weshalb die Register $r4_4$ und $r4_2$ verwendet werden. Sie sind bei einer Codierung durch reale Register zu ersetzen.

Weil der Kern normalerweise als Schleife programmiert ist, muss nach einem überlappenden Schleifendurchlauf selbstverständlich dafür gesorgt werden, dass die Quelloperanden in den jeweils verwendeten Registern verfügbar sind. Hierzu können entweder die Register umkopiert (z.B. bekommt $r7_2$ den Inhalt von $r7_1$ zugewiesen, so dass mit dem nächsten Schleifendurchlauf das Ergebnis der aktuellen Addition in den Speicher geschrieben wird) oder die verwendeten Indizes verändert werden. Letzteres erfordert, dass der Prozessor die Modifikation von Registeradressen in Hardware unterstützt, was leicht durch Registerverwaltungseinheiten, ähnlich denen, die in Abschnitt 2.1.5 beschrieben sind, erreichbar ist.

Das Prinzip der sog. *Registerrotation* veranschaulicht Bild 3.28a. Mit Ausführung der Pseudooperation rotate_reg werden die Indizes aller beteiligen Register um Eins dekrementiert. Der unmittelbar zuvor in das Register $r4_4$ geladene Operand befindet sich nach der Registerrotation dementsprechend in $r4_3$, der in einem vorangehenden Schleifendurchlauf geladene Operand $r4_3$ in $r4_2$ usw. Das Verfahren ist z.B. in Prozessoren mit IA-64-Architektur realisiert, wobei sich eine Registerrotation als optionaler Seiteneffekt einer Sprungoperation ausführen lässt. Neben einem in Grenzen frei programmierbaren Bereich des Integerregisterspeichers sind die *Prädikatsregister* p16 bis p63 und die Gleitkommaregister f32 bis f127 rotierbar. Dabei werden die Registeradressen jeweils inkrementiert bzw. bei einem Überlauf auf das erste rotierbare Register zurückgesetzt. Der Inhalt des Gleitkommaregisters f32 ist daher nach einer Registerrotation über f33 und der von f127 über f32 zugreifbar. Wie sich das in Bild 3.28a dargestellte Programm für einen Prozessor mit IA-64-Architektur umsetzen lässt, zeigt Bild 3.28b.

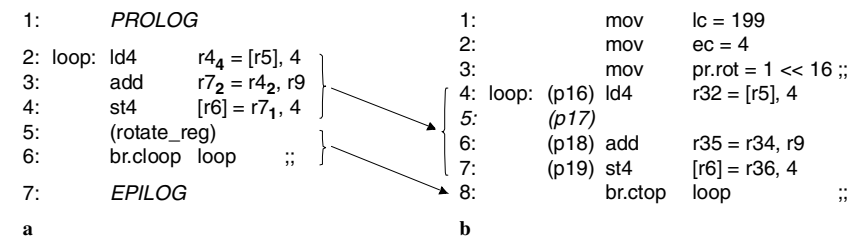

Bild 3.28. Registerrotation. **a** Prinzipielle Verwendung für Software-Fließbänder. **b** Programm für die Prozessorarchitektur IA-64

Der Kern der Operationsfolge zwischen den Zeilen 4 und 8 ist direkt von dem in Bild 3.28a skizierten Programm ableitbar, indem die mit Indizes versehenen Registernamen ersetzt werden und statt der beiden Operationen rotate_reg und br.cloop die spezialisierte Sprungoperation br.ctop verwendet wird. Die zusätzlich in den Zeilen 4 bis 7 angegebenen Prädikatsregister sind für den Prolog und den Epilog erforderlich. Durch die Initialisierung der Prädikatsregister in Zeile 3, bei der p16 gleich Eins und p17 bis p63 gleich Null gesetzt werden, wird nämlich dafür gesorgt, dass mit dem ersten Schleifendurchlauf zunächst nur die Lade- und Sprungoperation, nicht jedoch die Addition und Speichereoperation ausgeführt werden. Dies ent-

spricht dem, was im ersten Schritt des Prologs zu bearbeiten ist (siehe Tabelle 3.1; vgl. auch Bild 3.27b zum Zeitpunkt t_{-2}).

Durch die Sprungoperation br.ctop werden als Seiteneffekt die Registerinhalte rotiert und zum Anfang der Schleife verzweigt, und zwar, solange der im Spezialregister lc (loop counter) befindliche Schleifenzähler einen Wert ungleich Null enthält. Insbesondere übernimmt dabei r33 den Inhalt von r32 und p17 den Inhalt von p16. Außerdem wird p16 implizit gesetzt und auf diese Weise dafür gesorgt, dass im zweiten Schleifendurchlauf die in den Zeilen 4 und 5 stehenden Operationen sowie die Sprungoperation ausgeführt werden (siehe den zweiten Zyklus in Tabelle 3.1 und vgl. Bild 3.27b zum Zeitpunkt t_{-1}). Die Leeroperation in Zeile 5 ist dabei nur der besseren Verständlichkeit halber angegeben und in einem realen Programm unnötig. Mit dem dritten Schleifendurchlauf wird in gleicher Weise dafür gesorgt, dass p16 bis p18 gesetzt sind und deshalb neben der Lade- und Sprungoperation noch die Addition ausgeführt wird. Sie verarbeitet mit r34 den Inhalt, des zuvor in r33 befindlichen, im ersten Schleifendurchlauf aus dem Hauptspeicher geladenen Operanden (in Tabelle 3.1 wird angenommen, dass in r9 der Wert 10 gespeichert ist). Mit dem nächsten Sprung, der zum Anfang der Schleife führt, endet der Prolog.

Tabelle 3.1. Zeitlicher Ablauf bei Bearbeitung der in Bild 3.28b dargestellten Operationsfolge. Der Übersichtlichkeit halber werden gelöschte Prädikatsregister durch leere Felder repräsentiert

Zyklus	*Befehle*	*Zustand vor Ausführung von br.ctop*										
		r32	r33	r34	**r35**	r36	p16	p17	p18	p19	lc	ec
1	ld4 br.ctop	0	-	-	-	-	1				199	4
2	ld4 br.ctop	1	0	-	-	-	1	1			198	4
3	ld4 ‖ add br.ctop	2	1	0	10	-	1	1	1		197	4
4	ld4 ‖ add ‖ st4 br.ctop	3	2	1	11	10	1	1	1	1	198	4
5	ld4 ‖ add ‖ st4 br.ctop	4	3	2	12	11	1	1	1	1	197	4
...	...			...					...		...	...
200	ld4 ‖ add ‖ st4 br.ctop	199	198	197	207	206	1	1	1	1	0	4
201	add ‖ st4 br.ctop	-	199	198	208	207		1	1	1	0	3
202	add ‖ st4 br.ctop	-	-	199	209	208			1	1	0	2
203	st4 br.ctop	-	-	-	-	209				1	0	1
...		-	-	-	-	-					0	0

In der sich anschließenden Kernphase werden mit jedem Zyklus alle in Bild 3.28b zwischen den Zeilen 4 bis 8 dargestellten Operationen parallel bearbeitet, und zwar so lange, bis der Inhalt des Schleifenzählregister lc den Zählwert Null erreicht und die Sprungoperation br.ctop erneut ausgeführt wird. Dies kennzeichnet den Beginn des Epilogs, der sich vom Prolog insbesondere dadurch unterscheidet, dass nach der Registerrotation das Prädikatsregister p16 nicht mehr gesetzt, sondern gelöscht wird. Demzufolge bearbeitet man mit weiteren Schleifendurchläufen nach und nach eine immer geringere Anzahl von Operationen – das Fließband läuft leer. Das Bear-

beitungsende ist schließlich erreicht, wenn der in der Epilogphase durch die Sprungoperation br.ctop dekrementierte Epilog-Zähler ec (epilog counter) Null erreicht.

Eine abschließende Anmerkung: Die Bearbeitung einer sog. while-Schleife unterscheidet sich von der hier beschriebenen Zählschleife darin, dass das Ende der Kernphase nicht durch Erreichen eines Zählwerts Null, sondern durch das explizite Löschen eines Prädikatsregisters angezeigt wird. Im Programm ist dies zu berücksichtigen, indem man statt der Sprungoperation br.ctop die Sprungoperation br.wtop, verwendet.

3.1.6 Prozessoren mit kontrollflussgesteuertem Datenfluss

Die bis hierher beschriebenen Prozessoren verarbeiten Operanden, die i. Allg. Ergebnisse vorangehend ausgeführter Operationen sind und in Registern oder Speicherzellen übergeben werden. Für eine einschrittige Operationsverarbeitung ist es daher notwendig, gleichzeitig auf die zu verknüpfenden Operanden und das zu schreibende Ergebnis zuzugreifen, weshalb z.B. die Registerspeicher mit wenigstens drei unabhängigen Ports ausgestattet sein müssen. Dies gilt für streng sequentiell arbeitende Prozessoren, insbesondere aber auch für skalare Fließbandprozessoren, bei denen die Lese- und Schreibzugriffe zwar zeitversetzt, jedoch parallel zu je einer anderen Operation ausgeführt werden. Insgesamt ist der hierfür erforderliche Realisierungsaufwand jedoch gering, selbst wenn man berücksichtigt, dass weitere Maßnahmen notwendig sind, um z.B. Kontroll- oder Datenflusskonflikte zu lösen.

Das ändert sich, wenn Operationen parallel verarbeitet werden sollen. Mit einer statischen Operationsparallelität von Vier müssen z.B. wenigstens 12 Registerports und bei einer einfachen Fließbandstruktur, wie sie in Bild 3.17a dargestellt ist, 64 Bypässe realisiert werden (zur Rückführung von je zwei Operanden aus der Execute- oder Write-Back-Stufe von vier separaten Verarbeitungseinheiten) – ein Aufwand, der insbesondere deshalb erforderlich ist, weil Zwischenergebnisse über temporäre Register weitergereicht werden, anstatt sie direkt in die zuständigen Verarbeitungseinheiten zu übertragen.

An der TU Berlin wurde bereits Anfang der 90er Jahre ein alternatives Verarbeitungsprinzip entwickelt, dass aktuell in Form des Zen-1 als FPGA verwirklicht wird [113]. Anders als bei herkömmlichen Prozessoren werden die Aktionen „Operanden lesen“, „Operanden verknüpfen“ und „Ergebnis schreiben“ hier nicht geschlossen in den einzelnen Operationen codiert, sondern die zu verknüpfenden Operanden durch *Transportoperationen* zu den Verarbeitungseinheiten übertragen, die daraus autonom Ergebnisse generieren, sobald die benötigten Operanden verfügbar sind. Eine zu herkömmlichen Prozessoren vergleichbare Geschwindigkeit lässt sich erreichen, indem mehrere Transportoperationen parallel ausgeführt werden.

Weil die statisch explizit in einem Befehl codierten Operationen den Datenfluss eines Teilproblems beschreiben, die Ausführung der Befehle jedoch nach dem Kontrollflussprinzip geschieht, spricht man von kontrollflussgesteuertem Datenfluss[1]. Die Registertransferschaltung eines in dieser Weise arbeitenden Prozessors, der pro Takt sechs Transportoperationen verarbeiten kann, zeigt Bild 3.29. Neben dem