

Zen - 1

**Prozessor mit
kontrollflussgesteuertem
Datenfluss**

Dr.-Ing. M. Menge

2003/21

ISSN 1436-9915

Inhalt

1	Einleitung	2
2	Architektur	3
3	Befehlscodierung	5
4	Verkettung (Chaining)	6
5	Ausnahmebehandlungen	8
6	Ergebnisse	8
7	Architektur des Zen-1	13
8	Zusammenfassung und Ausblicke	16
9	Literatur	18

1 Einleitung

Mit einem klassischen datenverarbeitenden Befehl werden normalerweise Operanden verknüpft, die Ergebnisse vorangehend ausgeführter Befehle sind. Die Operanden werden hierbei meist in Registern übergeben. Da ein einzelner dyadischer Befehl zwei Operanden verarbeitet und ein Ergebnis erzeugt, muss hierbei entweder der Registerspeicher mit drei Ports realisiert sein oder die Verarbeitung eines Befehls zeitsequentiell erfolgen. Dies gilt insbesondere für skalare Fließbandprozessoren, bei denen die Lese- und Schreibzugriffe zwar zeitversetzt jedoch parallel zu je einem anderen Befehl ausgeführt werden. Der notwendige Realisierungsaufwand ist vergleichsweise gering, selbst wenn berücksichtigt wird, dass zusätzliche Maßnahmen notwendig sind, um z.B. Kontroll- oder Datenflusskonflikte zu lösen. Dies ändert sich jedoch mit den superskalaren oder VLIW-Prozessoren, die mehrere Befehle parallel bearbeiten können. Bei einer Operationsparallelität von fünf müssen z.B. wenigstens 15 Registerports und bei einem vierstufigen Fließband, 100 Bypässe realisiert werden (zur Rückführung von je zwei Operanden aus der Execute- oder Write-Back-Stufe von fünf Funktionseinheiten). Deshalb verfügt der vierfach superskalare SPARC64 V von Fujitsu z.B. über einen Registerspeicher mit 12 Ports [5], der ebenfalls vierfach superskalare Alpha 21264 von Compaq über einen Registerspeicher mit 14 Ports [4] und der fünffach parallel arbeitende VLIW-Prozessor TriMedia von Philips über einen Registerspeicher mit 20 Ports [1, 8, 9].

Neben dem hohen Aufwand ist ein weiterer Nachteil operationsparallel arbeitender Prozessoren, dass die Funktionseinheiten mit unterschiedlichen Komponenten, wie dem Decoder oder dem Registerspeicher eng verwoben sind. Trotz einer prinzipiellen Skalierbarkeit ist daher ein sehr hoher Aufwand erforderlich, um von einer bestehenden Realisierung ausgehend zu einer neuen leistungsfähigeren Prozessorvariante zu gelangen. Vor allem superskalare Prozessoren sind in dieser Beziehung sehr aufwendig zu handhaben. Um nämlich die maximal erreichbare Operationsparallelität zu verbessern, müssen neben dem Decoder auch der Registerspeicher, die Renaming-Unit, der Instruction-Buffer, die Completion-Unit usw. modifiziert werden. Bei Berücksichtigung des Umstands, dass die um einen Monat verzögerte Markteinführung eines Prozessors durch eine etwa vierprozentige Leistungssteigerung kompensiert werden muss, damit sich das Produkt am Markt amortisiert, wird die Tragweite dieser Problematik deutlich. Die genannten Nachteile werden mit einem Prozessor, der nach dem im folgenden beschriebenen Prinzip des kontrollflussgesteuerten Datenflusses (controlflow directed dataflow) arbeitet, vermieden. Anstatt die Aktionen „Operanden lesen“, „Operanden verknüpfen“ und „Ergebnis schreiben“ geschlossen in einem Befehl zu bündeln, werden die zu verknüpfenden Operanden durch Transportoperationen zu den Funktionseinheiten übertragen, die daraus autonom Ergebnisse generieren, sobald die benötigten Operanden verfügbar sind. Indem mehrere Transportoperationen parallel ausgeführt werden, wird eine zu klassischen Prozessoren vergleichbare Geschwindigkeit erreicht.

Das hier vorgestellte Verarbeitungsprinzip ähnelt dem, das mit ADARC in [10, 2] beschrieben ist. Zwar kann ADARC vom Konzept her als MIMD klassifiziert werden, es ist aber als Sonderform möglich, einen nach dem SIMD-Prinzip arbeitenden VLIW-Prozessor von der Basisarchitektur abzuleiten. Die jeweils in den Funktionseinheiten auszuführenden Dreiadressoperationen werden dabei lokal decodiert und stellen entsprechend der darin enthaltenen Operandenkennungen Anforderungen an ein als Kreuzschienenverteiler realisiertes assoziatives Kommunikationsnetz. Sobald der benötigte Operand berechnet wurde, wird er zusammen mit seiner Kennung an das Kommunikationsnetz ausgegeben und diese in sämtlichen mit der ergebniserzeugenden Funktionseinheit verbundenen Kreuzungs-

punkte des Kreuzschienenverteilers mit den dort gespeicherten Operandenkennungen verglichen. Bei Übereinstimmung wird das Ergebnis in die konsumierende Funktionseinheit weitergereicht. Die hier vorgestellte Prozessorarchitektur verwendet im Gegensatz hierzu ein passives Verbindungsnetzwerk, das allein durch die im VLIW-Befehlsword codierten Adressen geschaltet wird. Hierbei werden die auszuführenden Befehle nicht direkt sondern über das Verbindungsnetz an die Funktionseinheiten verteilt, und zwar implizit mit den verwendeten Zieladressen der Transportoperationen.

Im weiteren Verlauf dieser Ausarbeitung wird zunächst die Architektur eines Prozessors, der nach dem Prinzip des kontrollflussgesteuerten Datenflusses arbeitet, beschrieben. Daran im Anschluss wird erläutert, auf welche Weise die Transportoperationen codiert werden können. Im dritten Abschnitt wird das sog. Harte und Weiche Verketteten von Funktionseinheiten (chaining) beschrieben. Abschnitt 5 erklärt den Umgang mit Ausnahmeanforderungen. Basierend auf der allgemeinen Architektur wurden Simulationen durchgeführt, die in Abschnitt 6 diskutiert werden. Sie sind die Grundlage für den in Abschnitt 7 vorgestellten Prozessor Zen-1 – eine konkrete Implementierung des hier beschriebenen Konzepts. Die Ausarbeitung schließt einer Zusammenfassung und einem kurzen Ausblick in die Zukunft.

2 Architektur

Die vereinfachte Registertransferstruktur eines Prozessors mit kontrollflussgesteuertem Datenfluss ist in Bild 1 dargestellt. Neben dem Registerspeicher bzw. der Konstanteneinheit und der Sprungverarbeitungseinheit besitzt er drei Funktionseinheiten die je nach angestrebter Leistungsfähigkeit ggf. auch mehrfach implementiert sein können (im Bild jeweils als Schatten angedeutet). Die Verarbeitung eines Befehls beginnt damit, dass er zunächst aus dem Befehlsspeicher in das mit einem *a* markierte Instruktionsregister IR geladen wird (der Einfachheit halber ist der Befehlsspeicher ohne die normalerweise vorhandene Speicherverwaltungseinheit, den Cache, eine Sprungvorhersageeinheit u.a. dargestellt). Die in einem Befehl parallel codierten Transportoperationen enthalten jeweils zwei Adressen zur Anwahl einer Quelle und einer Senke, die bei Ausführung hier über einen Bus miteinander verbunden werden, dem eine Transportoperation fest zugeordnet ist. Zum Beispiel ist im Bild als dicke Linie dargestellt, wie der Inhalt eines Registers zum linken Eingang der ALU transportiert werden kann. Die Registernummer ist dabei in der Quelladresse der Transportoperation codiert und vom Registerspeicher auszuwerten. Umgekehrt ist in der Zieladresse ggf. codiert, welche Operation von der ALU bearbeitet werden soll. Falls wie bei der ALU mehrere Eingänge zur Verfügung stehen, wird die Operation z.B. mit dem am weitesten links stehenden Operanden festgelegt.

Sobald alle benötigten Operanden verfügbar sind, normalerweise also nach Ausführung des Befehls, mit dem der letzte noch benötigte Operand zur Funktionseinheit transportiert wurde, beginnt die Funktionseinheit damit ein Ergebnis zu erzeugen (c). Es wird nach Ablauf der für die Funktionseinheit notwendigen Latenzzeit in einem Ringpuffer zwischengespeichert (d) und steht darin zur Weiterverarbeitung als Quelle anderer Transportoperationen zur Verfügung. Dabei sind drei Sonderfälle zu berücksichtigen: Falls lesend auf einen leeren Ringpuffer zugegriffen wird, der die Ergebnisse einer zu diesem Zeitpunkt aktiven Funktionseinheit entgegennimmt, wird der entsprechende Befehl verzögert, bis das in Bearbeitung befindliche Ergebnis verfügbar ist. (2.) Falls lesend auf einen leeren Ringpuffer zugegriffen wird, der die Ergebnisse einer zu diesem Zeitpunkt *nicht* aktiven Funktionseinheit entgegennimmt, weist dies auf einen Fehler des Programms hin, der z.B. durch eine Ausnahmeanforderung (Exception) signalisiert werden kann. (3.) Soll ein Ergebnis in einen vollständig gefüllten Ringpuffer eingetragen werden, weist dies ebenfalls auf einen Fehler

des Programms hin, der wieder z.B. durch Stellen einer Ausnahmeanforderung quittiert werden kann. Der zweite und dritte Fall ist vermeidbar, wenn das Programm korrekt generiert wird.

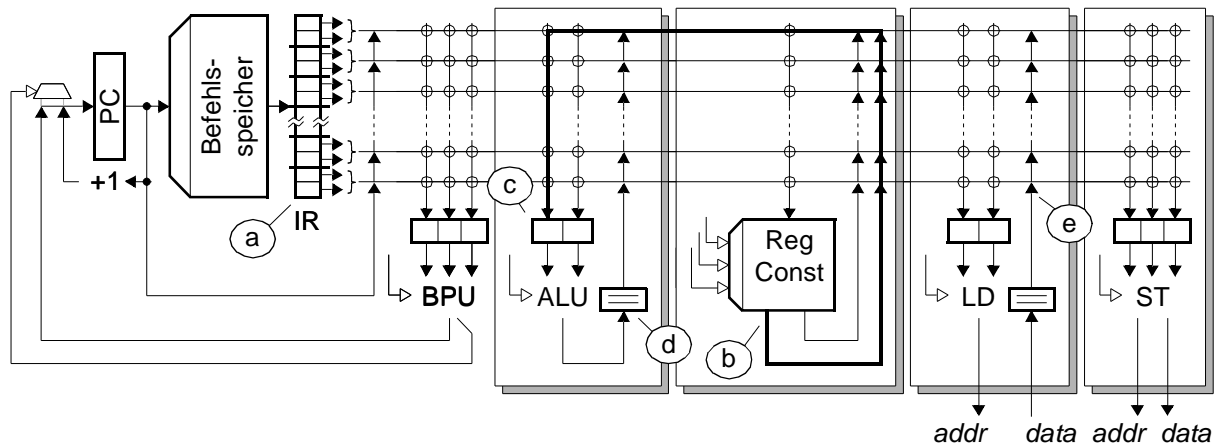


Bild 1. Allgemeine Architektur eines Prozessors mit kontrollflussgesteuertem Datenfluss.

Beispiel. Die Programmierung eines Prozessors, der entsprechend Bild 1 aufgebaut ist, eine Sprungverarbeitungseinheit BPU (branch processing unit), einen Registerspeicher, zwei ALUs, eine Multiplikationseinheit MUL, eine Ladeeinheit LD (load) und eine Speichereinheit ST (store) besitzt, soll am Beispiel des Skalarprodukts zweier 500-elementiger Vektoren beschrieben werden. Der verwendete Algorithmus ist in Bild 2a und 2b als C- bzw. Assemblerprogramm dargestellt. Nach den vereinfachenden Definitionen in den ersten beiden Zeilen des Assemblerprogramms werden in Zeile 3 die Ausgänge der mit *prod* und *k* assoziierten Funktionseinheiten *alu1* und *alu2* initialisiert. Die erste Zuweisungen in Zeile 4 beschreibt tatsächlich zwei Transportoperationen: Eine, mit der die Konstante *A* (aus dem Register-Konstanten-Speicher *Reg / Const*) und eine mit der der in *alu2* befindliche Wert *k* zur Ladeeinheit transportiert wird. Da die benötigten Operanden verfügbar sind, beginnt die Ladeeinheit unmittelbar mit dem Zugriff auf den Datenspeicher (der Einfachheit halber wird dabei angenommen, dass mit dem Zugriff eine Skalierung des Indexes entsprechend des verwendeten Formats erfolgt). Der zu *k* angegebene Stern bewirkt dabei, dass der Wert aus dem Ringpuffer nicht entfernt wird. Er steht somit für weitere Transportoperationen als Quelle zur Verfügung.

<pre> 1: unsigned prod = 0; 2: for (unsigned k = 0; k < 500; k++) { 3: prod = prod + A [k] * B [k]; 4: }</pre>	<pre> 1: #define prod alu1 2: #define k alu2 3: prod = #0 k = #0 ; 4: L: ld = (#A, k*) bne = (-, #499, L) ; 5: ld = (#B, k) k (+) = (k, #1) ; 6: mul1.l = ld bne.l = k ; 7: prod (+).r = mul1 prod (+).l = prod ; 8: r0 = prod k ;</pre>
---	---

Bild 2. Berechnung des Skalarprodukts. **a** In einem C-Programm. **b** In einem Assemblerprogramm.

Neben der Zuweisungen an die Ladeeinheit sind in dem in Zeile 4 stehenden Befehl noch Zuweisungen an die Sprungverarbeitungseinheit codiert. Da hier der linke Operand noch fehlt, wird die Ausführung des bedingten Sprungs jedoch noch nicht begonnen. Dies geschieht erst in Zeile 5 mit der Zuweisung des im Ringpuffer der *alu2* befindlichen Werts *k*

an den Eingang bne.l. Parallel dazu wird der Vektorzugriff auf das Element B [k] gestartet, das Inkrementieren des Schleifenzählers k initiiert (alu2 wird dabei als Addierer verwendet) und der erste bereits aus dem Datenspeicher gelesene Wert A [k] zum linken Eingang der Multiplikationseinheit mull1 transportiert. Der im Ringpuffer der alu2 befindliche Wert k wird dabei konsumiert. Mit k ungleich 499 wird zur Sprungmarke L verzweigt. Dies geschieht jedoch mit zwei Takten Verzögerung, weshalb die Befehle in Zeile 6 und 7 vor dem nächsten Schleifendurchlauf ebenfalls ausgeführt werden. In Zeile 6 wird die Multiplikation begonnen und deren Akkumulation in alu2 vorbereitet. In Zeile 7 wird die Summation gestartet, indem das Multiplikationsergebnis zum rechten Eingang der Funktionseinheit alu1 transportiert wird. Das Programm schließt, indem das Endergebnis in Zeile 8 in das Register r0 übertragen wird. Parallel dazu wird außerdem der noch im Ringpuffer der alu2 befindliche Wert k konsumiert.

3 Befehlskodierung

Der in Bild 1 dargestellte Prozessor mit kontrollflussgesteuertem Datenfluss kann Befehle verarbeiten, die eine einheitliche Breite besitzen und in denen eine konstante Anzahl an Transportoperationen codiert ist. In jeder Transportoperation ist eine Quell- und eine Zieladresse, jedoch kein Operationscode enthalten. Da viele Funktionseinheiten mehr Ein- als Ausgänge besitzen, kann der Zieladressraum (Eingänge) größer als der Quelladressraum (Ausgänge) und die in den Transportoperationen codierten Adressen unterschiedlich breit sein. Bei der Unterteilung des Zieladressraums kann außerdem genutzt werden, dass die in den Funktionseinheiten aufzuführenden Operationen nicht über beide Eingänge, sondern in der immer selben Weise über den linken oder rechten Eingang festlegbar sind. So müssen z.B. für eine Funktionseinheit, die zwei Eingänge besitzt und drei Operationen ausführen kann nur vier Adressen im Zieladressraum reserviert werden. Eine Vereinfachung der Operationscodierung ist außerdem erreichbar, indem bestimmte Quellen und Ziele nicht über beliebige Busse miteinander verbunden werden können, sondern nur über eine Teilmenge der verfügbaren Busse. Wenn z.B. der Ausgang der Speicherzugriffseinheit MEM fest an den im Bild untersten Bus gekoppelt wird (Markierung e), muss in einer entsprechenden Transportoperation keine Quelladressen mehr codiert sein. Wenn außerdem der Ausgang der Speicherzugriffseinheit MEM als Quelle ausschließlich über diesen Bus zugreifbar ist, kann darauf verzichtet werden, diese Quelle über andere Busse adressieren zu können, wodurch die in anderen Transportoperationen codierten Quelladressen an Breite verlieren.

Weitere Vorteile der festen Kopplung von Ausgängen und Bussen ist, dass einerseits der Aufwand vermindert wird und andererseits die kapazitiven Lasten auf den Bussen geringer werden. Letzteres wirkt sich positiv auf die maximale Taktfrequenz und den Strombedarf eines entsprechend realisierten Prozessors aus. Soll es möglich sein, ein einzelnes Ergebnis auf unterschiedliche Eingänge zu verteilen, müssen entweder mehrere explizite Busse verwendet werden (z.B. lässt sich der Ausgang der Speicherzugriffseinheit an zwei Busse koppeln), oder die Architektur in der Weise erweitert werden, dass ein Ergebnis über einen Bus an mehrere Ziele gleichzeitig übertragen werden kann. Hierzu dürfen die einzelnen Transportoperationen jedoch nicht mehr fest einem Bus zugeordnet sein, sondern müssen durch einen zwischen Befehlsspeicher und Instruktionsregister einzufügenden Vordecodierer variable den verschiedenen Bussen zugeordnet werden. Als Nebeneffekt können von einem derart realisierten Prozessor Befehle variabler Breite verarbeitet werden, was eine Voraussetzung für die Skalierbarkeit der Operationsparallelität ist. Dabei wird ähnlich verfahren wie beim Itanium von Intel [3], bei dem die Befehle so codiert werden, dass die Semantik eines Programms nicht davon beeinflusst wird, ob die einzelnen Operationen sequentiell,

zum Teil parallel oder vollständig parallel ausgeführt werden. Übertragen auf einen Prozessor mit kontrollflussgesteuertem Datenfluss bedeutet dies, dass bei der Codeerzeugung der Syntaxgraph in einer Tiefensuche durchlaufen und zu jeder nach oben führenden Kante ein Transportoperation generiert wird. Ein neuer Befehl (mit den darin codierten parallelen Transportoperationen) wird erzeugt, wenn dies aufgrund von Ressourcenkonflikten oder echten Datenabhängigkeiten erforderlich ist.

4 Verkettung (Chaining)

Die hier beschriebene Prozessorarchitektur zeichnet sich dadurch aus, dass Funktionseinheiten über Busse beliebig miteinander verkettet werden können, und zwar auf unterschiedliche Art und Weise. Bei der befehlsgesteuerten sog. weichen Verkettung (Soft-Chaining) wird durch Transportoperationen initiiert eine Verbindung über einen der expliziten Busse hergestellt und am Ende der Ausführung eines Befehls wieder aufgelöst. Neben den bisher beschriebenen expliziten Transporten ist es aber auch möglich, eine Verbindung als Seiteneffekt einer Transportoperation zu erzeugen, und zwar über zusätzliche in Bild 1 nicht dargestellte Verbindungen. Zum Beispiel kann eine am zentralen Bussystem vorbeiführende Rückkopplung einer arithmetisch logischen Einheit verwendet werden, um die häufig auftretenden akkumulierenden Berechnungen durchzuführen. Um sie zu aktivieren kann z.B. die Zieladresse des ersten zu verarbeitenden Operanden modifiziert werden. Natürlich können solche Direktverbindungen auch befehlsübergreifend „hart“ etabliert werden, um auf diese Art und Weise mehrstufige Funktionseinheiten dynamisch zu synthetisieren. Neben den bereits erwähnten Rückkopplungen sind Direktverbindungen z.B. zwischen Speicherzugriffs- und anderen Funktionseinheiten oder zwischen Multiplikations- und Additionseinheiten sinnvoll. Letzteres um die in vielen Algorithmen auftretende Produktschritte berechnen zu können, ohne einen für andere Transportoperationen nutzbaren Bus zu belegen. Die harte Verkettung einer Konstanteneinheit (in Bild 1 ist sie kombiniert mit dem Registerspeicher dargestellt) mit einer beliebigen Funktionseinheit ist sogar ohne einen zusätzlichen Bus realisierbar, und zwar, indem die jeweilige Konstante in das Eingangsregister der Zielfunktionseinheit transportiert und als permanent zu halten gekennzeichnet wird.

Welche harten Verkettungen zu aktivieren sind, kann z.B. über ein Steuerregister festgelegt werden. Bei einer harten Verkettung von Konstanten muss zusätzlich noch berücksichtigt werden, dass neben dem Zugriff auf das Steuerregister zusätzlich der Wert der Konstanten in das jeweilige Operandenregister der Zieleinheit zu übertragen ist, so dass hierbei insgesamt zwei Transportoperationen ausgeführt werden müssen. Da mit einem Zugriff auf das für harte Verkettungen benutzte Steuerregister jedoch mehr als eine Verbindung gleichzeitig geschaltet werden können, ist der programmierte Zusatzaufwand gering. Soll eine etablierte harte Verbindung wieder gelöst werden, ist dies jederzeit durch einen Zugriff auf das Steuerregister möglich. Alternativ kann dies auch implizit als Seiteneffekt einer anderen Operation geschehen, z.B. wenn das hart verkettete Operandenregister Ziel einer expliziten Transportoperation ist. Je nach Implementierung kann so eine einzelne Verbindung oder es können alle harten Verbindungen gelöst werden. Im folgenden sollen die Möglichkeiten der harten und weichen Verkettung von Funktionseinheiten an einem Beispiel erläutert werden:

Beispiel. Zwar verdeutlicht das in Bild 2b präsentierte Assemblerprogramm zur Berechnung des Skalarprodukts zweier Vektoren die Programmierung eines Prozessors mit kontrollflussgesteuertem Datenfluss, nicht jedoch dessen Fähigkeit zur Parallelverarbeitung. So werden dort innerhalb der Schleife z.B. nur 1,5 echte Operationen pro Takt ausgeführt (eine Multiplikation, zwei Additionen, ein Vergleich und zwei Ladeoperationen die in insgesamt vier Befehlen codiert sind), statt der erwarteten zwei dyadischen Operationen, die

möglich sind, wenn sechs explizite Busse zur Verfügung stehen, um gleichzeitig Operanden und Ergebnisse zu transportieren. Um das Maß an Parallelität zu steigern, kann das sog. Software-Pipelining verwendet werden. Hierzu wird der Schleifenrumpf in Stufen z.B. entsprechend Bild 3a unterteilt und parallel jeweils die Operationen aus unterschiedlichen Schleifendurchläufen bearbeitet. In der ersten Stufe werden dabei die Vektorelemente geladen und der Schleifenzähler inkrementiert, in der zweiten Stufe wird das Produkt der einzelnen Vektorelemente gebildet und in der dritten Stufe werden die einzelnen Produkte schließlich akkumuliert. Das sehr komplexe Assemblerprogramm ist in Bild 3b dargestellt.

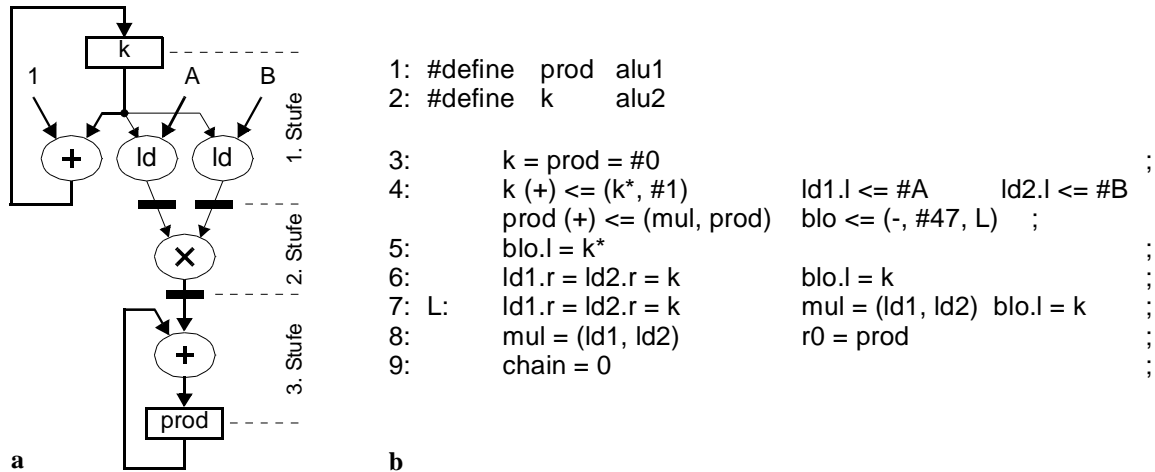


Bild 3. Skalarprodukt zweier Vektoren. **a** Datenfluss innerhalb der Schleife. **b** Assemblerprogramm.

Der besseren Lesbarkeit halber werden den Einheiten *alu1* und *alu2* in Zeile 1 und 2 zunächst Namen zugewiesen. Das eigentliche Programm beginnt in Zeile 3 mit der Initialisierung der Ausgangsringpuffer. In Zeile 4 werden die durch eine Optimierung erkannten harten Verkettungen entsprechend der in Bild 3a dick dargestellten Kanten aktiviert. Die dabei verwendete Schreibweise abstrahiert von den tatsächlich codierten Operationen. Tatsächlich sind in Zeile 4 nämlich ein Schreibzugriff auf das Steuerregister *chain* sowie Transportoperationen für die Konstanten 1, A, B und 497 codiert. In Zeile 5 wird die Schleifensprungoperation *blo* (branch lower) als vorbereitende Maßnahme und als Teil des Prologs der Software-Pipeline das erste Mal initiiert. Dies ist hier erforderlich, weil der in Bild 1 dargestellte Prozessor Verzweigungen mit zwei Takten Verzögerung ausführt. Im zweiten Schritt des Prologs wird in Zeile 6 das Laden der ersten beiden Vektorelemente angestoßen sowie der zweite Schleifendurchlauf vorbereitet. Wegen der harten Verkettung von Multiplikations- und Additionseinheit sind weitere Aktionen der Schleifeninitialisierung, z.B. das initialisieren der Multiplikationseinheit hier nicht notwendig.

In Zeile 7 ist der vollständige Schleifenkern codiert. Pro Takt werden darin weitere Operanden gelesen und die bereits geladenen Vektorelemente durch weiche Verkettung zur Multiplikationseinheit weitergereicht, um schließlich automatisch akkumuliert zu werden. Selbst wenn die Ladeoperation mehrere Takte Latenzzeit aufweist, wird pro Takt ein Ergebnis generiert, sobald die Software-Pipeline vollständig gefüllt ist. Mit dem in Zeile 8 beginnenden Epilog werden die letzten noch aus dem Hauptspeicher geladenen Operanden zur Multiplikationseinheit übertragen. Außerdem wird das endgültige Akkumulationsergebnis, mit dessen Verfügbarkeit, in das Register *r0* transportiert. Die harte Verkettung wird durch Zugriff auf das Steuerregister *chain* in Zeile 9 aufgelöst. Als Seiteneffekt werden dabei die Inhalte aller Operandenregister der beteiligten Funktionseinheiten, in denen sich noch Werte befinden, gelöscht. Es kann zusammengefasst werden, dass innerhalb der Schleife

pro Takt sechs herkömmliche dyadische Operationen und somit eine Operation pro explizit nutzbaren Bus ausgeführt werden können. Für das gesamte Programm wird dabei nur ein einzelnes Register nämlich für das Endergebnis benötigt.

5 Ausnahmebehandlungen

Ein Nachteil von Prozessoren mit kontrollflussgesteuertem Datenfluss ist, dass der Umgang mit Ausnahmeanforderungen sich sehr aufwendig gestalten kann. Anders als bei herkömmlichen Prozessoren ist der zu sichernde Kontext nämlich nicht in den Arbeitsregistern gespeichert, sondern zusätzlich in den Funktionseinheiten, d.h. den Operandenregistern und Ringpuffern. Solange Ausnahmeprogramme (Exception-Handler) selbst nicht unterbrechbar sein müssen, ist es am einfachsten, die Inhalte aller Steuer-, Operanden- und Operationsregister sowie der Ringpuffer in sog. Schattenregistern zwischenspeichern, erstere anschließend in den Initialzustand zu versetzen und schließlich die Ausnahmebearbeitung zu beginnen. Der ursprüngliche Zustand kann dabei nach Bearbeitung der Ausnahmeanforderung wieder hergestellt werden, indem die in den Schattenregistern gesicherten Inhalte in die Originalregister oder Puffer kopiert werden. Falls auch der Inhalt des Befehlszählers mit der Ausnahmeanforderung in einem Schattenregister gesichert wurde, kann der Rücksprung als Seiteneffekt des Wiederherstellens des ursprünglichen Programmzustands vor der Ausnahmeanforderung erreicht werden. Dabei wird der Inhalt des entsprechenden Schattenregisters einfach in den Befehlszähler zurückkopiert.

Das hier angedeutete in vielen herkömmlichen Prozessoren wie z.B. dem ARM9TDMI ebenfalls verwendete Verfahren ist insbesondere geeignet, korregierbare, durch die reguläre Befehlsausführung verursachte Fehler zu beheben. Zum Beispiel kann ein Seitenfehler durch ein Ausnahmeprogramm in der Weise bearbeitet werden, dass die fehlende Seite geladen und mit dem Rücksprung der ursprüngliche Zustand des unterbrochenen Zugriffs aktiviert wird, wobei der die Ausnahmeanforderung auslösende Zugriff im Ausnahmeprogramm wiederholt wird. Deutlich aufwendiger ist der Umgang mit Ausnahmeanforderungen, wenn das jeweilige Ausnahmeprogramm selbst unterbrechbar sein muss, wie z.B. bei Prozesswechseln. Die in den Schattenregistern gespeicherten Inhalte müssen hierbei explizit, d.h. Befehlsgesteuert, innerhalb des Ausnahmeprogramms in den Register- oder Hauptspeicher übertragen werden. Bei Rückkehr in das unterbrochene Programm müssen umgekehrt die ursprünglichen Inhalte der Schattenregister wieder hergestellt werden, bevor dessen Bearbeitung fortgesetzt werden kann. Dies geschieht ebenfalls explizit durch Zugriff auf den Register- oder Hauptspeicher.

6 Ergebnisse

Unter der Voraussetzung, dass ein Prozessor mit kontrollflussgesteuertem Datenfluss die selbe Anzahl an Funktionseinheiten, Registern und Registerports besitzt, wie z.B. ein VLIW-Prozessor und die Anzahl der expliziten Busse hier nicht geringer als die Anzahl der Bypässe dort ist, werden beide die selbe Ausführungsgeschwindigkeit aufweisen, wenn sie mit der selben Taktfrequenz betrieben werden. Wesentlich interessanter als ein direkter Vergleich dieser Prozessorarchitekturen ist, wie die Ausführungsgeschwindigkeit durch die Anzahl der realisierten Busse, Funktionseinheiten, Registerspeicher und Registerports beeinflusst wird. Aus diesem Grund wurden fünf unterschiedliche Benchmark-Programme, nämlich Bubblesort, FIR, Polynom, Sieve und Tonwertkorrektur für fiktive Prozessoren mit kontrollflussgesteuertem Datenfluss in Assembler programmiert, die sich in der Anzahl der realisierten Busse, Funktionseinheiten und Registerports unterscheiden und die daraus

generierten Binärprogramme mit Hilfe von jitter [6, 7] auf Befehlsebene simuliert. Die dabei generierten Ergebnisse werden im folgenden zunächst bezogen auf die einzelnen Benchmark-Programme anschließend zusammenfassend diskutiert.

Bubblesort. Dieses Benchmark-Programm sortiert ein 6000 Einträge umfassendes Feld in aufsteigender Reihenfolge, dessen Elemente zuvor absteigend sortiert waren. Da jeweils zwei benachbarte Einträge des Felds miteinander verglichen und ggf. vertauscht werden, existieren zahlreiche Abhängigkeiten, die eine effektive Parallelisierung des Programms verhindern. Die Ergebnisse der Ausführung des Benchmark-Programms sind für unterschiedlich viele Busse und Funktionseinheiten in Tabelle 1 dargestellt. Die maximale Ausführungsgeschwindigkeit wird mit fünf parallelen Transportoperationen erreicht. Da insgesamt $234 \cdot 10^6$ elementare Operationen ausgeführt werden entspricht die gemessene Laufzeit von $161 \cdot 10^6$ Takte einem Speedup von 1,44. Mit einer geringeren Anzahl explizit nutzbarer Busse sinkt dieser Wert. Er erreicht eins, wenn nur drei Busse verfügbar sind, was insofern den Erwartungen entspricht, weil eine dyadische Operation mit zwei Operanden und einem Ergebnis drei Transporte erfordert. Allerdings wird dieses Ergebnis durch die nachfolgend beschriebenen Benchmark-Programme nicht bestätigt, die i.allg. einen Speedup nahe eins erreichen, wenn zwei explizite Busse verfügbar sind. Diese geringere Anzahl reicht deshalb aus, weil Ergebnisse oft direkt in einer Funktionseinheit weiterverarbeitet werden können und ein expliziter Transport daher nicht notwendig ist.

Tabelle 1. Simulationsergebnisse zum Benchmark-Programm Bubblesort.

Busse	Funktionseinheiten			Register	Portzahl des Registerspeichers			Laufzeit [Takte]	Speedup [Op/s]
	ALU	LD	ST		Read	Write	Access		
5	1	2	2	3	2	2	2	$161 \cdot 10^6$	1,44
4	1	2	2	3	2	2	2	$198 \cdot 10^6$	1,18
3	1	2	2	3	2	2	2	$234 \cdot 10^6$	1
2	1	2	2	3	2	2	2	$306 \cdot 10^6$	0,76
1	1	2	2	3	1	1	1	$560 \cdot 10^6$	0,42
5	1	1	1	3	2	2	2	$216 \cdot 10^6$	1,08

FIR. Dieses Benchmark-Programm realisiert ein Transversalfilter 256er Ordnung und verarbeitet einen 2,5 Sekunden langen einkanaligen Audiostrom mit 40 KHz Abtastfrequenz. Insgesamt werden dabei $206 \cdot 10^6$ elementarer Operationen ausgeführt. Die Messergebnisse sind für unterschiedlich viele Busse und Funktionseinheiten in Tabelle 2 dargestellt. Da, ähnlich wie bei Bubblesort, die in den Schleifen durchgeführten Berechnungen zahlreiche Abhängigkeiten aufweisen, ist das erreichbare Maß an Parallelität deutlich begrenzt. Mit fünf explizit nutzbaren Bussen wird ein Speedup von 1,33 erreicht – ein Wert der sich nicht verbessert, wenn mehr als fünf Busse zur Verfügung stehen. Bemerkenswert ist, dass mit nur einem einzelnen expliziten Bus für eine elementare Operation nur zwei statt drei Transportoperationen benötigt werden. Dies wird hier darauf zurückgeführt, weil oft die zusätzlich vorgesehenen impliziten Busse verwendet werden können. Zum Beispiel wird für die zu berechnende Produktsumme die implizite Rückkopplung der ALU benutzt, um das Ergebnis zu akkumulieren (harte Verkettungen werden dabei nicht verwendet).

Tabelle 2. Simulationsergebnisse zum Benchmark-Programm FIR.

Busse	Funktionseinheiten			Register	Portzahl des Registerspeichers			Laufzeit [Takte]	Speedup [Op/s]
	ALU	LD	ST		Read	Write	Access		
5	4	1	1	3	2	3	3	$154 \cdot 10^6$	1,33
4	4	1	1	3	2	3	3	$154 \cdot 10^6$	1,33
3	4	1	1	3	2	2	2	$180 \cdot 10^6$	1,14
2	4	1	1	3	2	2	2	$231 \cdot 10^6$	0,89
1	4	1	1	3	1	1	1	$411 \cdot 10^6$	0,5
5	3	1	1	4	2	3	3	$180 \cdot 10^6$	1,14
5	2	1	1	5	2	4	4	$180 \cdot 10^6$	1,14
5	1	1	1	6	4	4	5	$205 \cdot 10^6$	1

Polynom. Dieses sehr einfache Benchmark-Programm berechnet ein Polynom des Grades $40 \cdot 10^6$. Insgesamt werden dabei $200 \cdot 10^6$ elementare Operationen ausgeführt. Die gemessenen Ergebnisse sind in Tabelle 3 dargestellt. Wegen der Einfachheit bieten sich kaum Ansatzmöglichkeiten für eine Parallelisierung. Der Maximale Speedup 1,25 wird erreicht, wenn drei oder mehr explizite Busse vorgesehen werden. Pro elementarer Operation werden bei einem Speedup 1,0 zwei Transporte durchgeführt.

Tabelle 3. Simulationsergebnisse zum Benchmark-Programm Polynom.

Busse	Funktionseinheiten			Register	Portzahl des Registerspeichers			Laufzeit [Takte]	Speedup [Op/s]
	ALU	LD	ST		Read	Write	Access		
4	2	1	0	3	2	2	2	$160 \cdot 10^6$	1,25
3	2	1	0	3	1	1	1	$160 \cdot 10^6$	1,25
2	2	1	0	3	1	1	1	$200 \cdot 10^6$	1
1	2	1	0	3	1	1	1	$320 \cdot 10^6$	0,625
4	1	1	0	5	3	2	3	$160 \cdot 10^6$	1,25

Sieve. Dieses Benchmark-Programm zur Primzahlenfindung nach dem von Eratosthenes beschriebenen Verfahren verwendet ein 60000-elementiges Feld mit jeweils 32 Bit breiten Einträgen. Die größte dabei berücksichtigte Zahl ist 1920000 ($32 \cdot 60000$). Insgesamt werden $51,5 \cdot 10^6$ elementare Operationen ausgeführt. Bei einer maximalen Parallelität von fünf Transportoperationen pro Takt wird ein Speedup von 1,51 erreicht. Falls zwei Busse zur Verfügung stehen, kann pro Takt eine Operation ausgeführt werden, was die Vermutung nahelegt, dass Zwischenergebnisse direkt aus den Funktionseinheiten weiterverarbeitet werden können. Dieser Schluss wird auch dadurch bestätigt, dass die Anzahl der benötigten Register deutlich steigt, wenn die Anzahl der verfügbaren Funktionseinheiten sinkt. Zwi-

schenergebnisse können hierbei nämlich nicht mehr in den Funktionseinheiten sondern müssen in Arbeitsregistern gehalten werden.

Tabelle 4. Simulationsergebnisse zum Benchmark-Programm Sieve.

Busse	Funktionseinheiten			Register	Portzahl des Registerspeichers			Laufzeit [Takte]	Speedup [Op/s]
	ALU	LD	ST		Read	Write	Access		
5	4	1	1	4	2	3	3	$34 \cdot 10^6$	1,51
4	4	1	1	4	2	2	2	$34 \cdot 10^6$	1,51
3	4	1	1	4	1	2	3	$40 \cdot 10^6$	1,30
2	4	1	1	4	1	2	2	$52 \cdot 10^6$	1
1	4	1	1	4	1	1	1	$101 \cdot 10^6$	0,51
5	3	1	1	5	2	4	4	$34 \cdot 10^6$	1,51
5	2	1	1	6	3	4	4	$34 \cdot 10^6$	1,51
5	1	1	1	6	3	4	4	$50 \cdot 10^6$	1,02

Tonwertkorrektur. Das aufwendigste hier untersuchte Benchmark-Programm führt eine automatische Tonwertkorrektur eines 1M Bildpunkte großen Farbbildes durch. Jeder Bildpunkt ist 16 Bit breit und enthält drei jeweils 4 Bit breite Felder für die Grundfarben Rot, Grün und Blau. Das Benchmark-Programm in dem insgesamt $51,0 \cdot 10^6$ elementare Operationen ausgeführt werden, zeichnet sich durch eine gute Parallelisierbarkeit aus. Die gemessenen Ergebnisse sind in Tabelle 5 dargestellt. Der mit 14 verfügbaren Bussen maximale Speedup von 2,79 liegt nur geringfügig über dem, was mit sieben Bussen erreichbar ist. Dies wird darauf zurückgeführt, dass nur selten 14 Busse benötigt werden. Bemerkenswert ist, dass der maximale Speedup weit unter dem liegt, was mit Hilfe der realisierten Funktionseinheiten erreicht werden könnte. Es kann daher vermutet werden, dass die Funktionseinheiten nur selten parallel arbeiten. Um den Realisierungsaufwand eines Prozessors mit kontrollflussgesteuertem Datenfluss gering zu halten, könnten z.B. virtuelle Funktionseinheiten vorgesehen werden, von denen mehrere eine gemeinsame reale Funktionseinheit nutzen.

Tabelle 5. Simulationsergebnisse zum Benchmark-Programm Tonwertkorrektur.

Busse	Funktionseinheiten			Register	Portzahl des Registerspeichers			Laufzeit [Takte]	Speedup [Op/s]
	ALU	LD	ST		Read	Write	Access		
14	7	3	3	10	8	3	8	$19 \cdot 10^6$	2,79
8	7	3	3	10	6	3	6	$20 \cdot 10^6$	2,65
7	7	3	3	10	4	3	4	$20 \cdot 10^6$	2,64
6	7	3	3	10	4	3	4	$21 \cdot 10^6$	2,52
5	7	3	3	10	4	3	4	$22 \cdot 10^6$	2,41
4	7	3	3	10	3	3	3	$23 \cdot 10^6$	2,30
3	7	3	3	10	2	2	2	$28 \cdot 10^6$	1,89
2	7	3	3	10	2	2	2	$43 \cdot 10^6$	1,23
1	7	3	3	10	1	1	1	$84 \cdot 10^6$	0,63
14	6	3	3	11	2	4	4	$19 \cdot 10^6$	2,79

Tabelle 5. Simulationsergebnisse zum Benchmark-Programm Tonwertkorrektur.

Busse	Funktionseinheiten			Register	Portzahl des Registerspeichers			Laufzeit [Takte]	Speedup [Op/s]
	ALU	LD	ST		Read	Write	Access		
14	5	3	3	13	8	3	8	$21 \cdot 10^6$	2,52
14	4	3	3	14	3	4	4	$21 \cdot 10^6$	2,52
14	3	3	3	15	3	4	4	$25 \cdot 10^6$	2,2

Zusammenfassung der Benchmark-Ergebnisse. Tabelle 6 enthält eine Zusammenfassung der zuvor dargestellten Einzelergebnisse. Die in der Tabelle angegebenen Laufzeiten sind jeweils auf das Programm normiert, das am langsamsten ausgeführt wurde. Dies ist in allen Fällen die Implementierung für einen Prozessor, der nur einen einzelnen expliziten Bus besitzt und nur eine einzelne Transportoperation pro Takt ausführen kann (dabei wurden implizite Direktverbindungen nicht berücksichtigt). Die unter den normierten Laufzeiten in Klammern dargestellten Zahlen geben zusätzlich den Speedup an, wobei als Bezug jeweils die Anzahl der elementaren Operationen verwendet wird, die von den einzelnen Programmen ausgeführt werden. Da die maximale Parallelität von den unterschiedlichen Benchmark-Programmen mit einer unterschiedlichen Anzahl von expliziten Bussen erreicht wird, sind nicht alle Positionen der Tabelle besetzt. So wird die maximale Ausführungsgeschwindigkeit beim Benchmark-Programm Polynom bereits mit drei Bussen und beim gut parallelisierbaren Benchmark-Programm Tonwertkorrektur erst mit 14 Bussen erreicht (dabei wurde darauf verzichtet, Optimierungstechniken wie das Software-Pipelining anzuwenden).

Tabelle 6. Normierte Laufzeiten verschiedener Benchmark-Programme für unterschiedliche realisierte Prozessoren mit kontrollflussgesteuertem Datenfluss.

Benchmark	Normierte Laufzeit bei unterschiedlicher Buszahl (In Klammern: Operationen pro Takt)								Portzahl des Registerspeichers		
	≥ 14	7-13	6	5	4	3	2	1	Read	Write	Access
Bubblesort				0,29 (1,4)	0,36 (1,2)	0,42 (1)	0,55 (0,8)	1 (0,4)	2	2	2
FIR				0,38 (1,3)	0,38 (1,3)	0,44 (1,1)	0,56 (0,9)	1 (0,5)	2	3	3
Polynom						0,5 (1,3)	0,6 (1)	1 (0,6)	2	2	2
Sieve					0,33 (1,5)	0,39 (1,3)	0,51 (1)	1 (0,5)	2	2	2
Tonwertkorrektur	0,23 (2,8)	0,24 (2,6)	0,25 (2,5)	0,26 (2,4)	0,27 (2,3)	0,33 (1,9)	0,52 (1,2)	1 (0,6)	4	3	4

Da die Verdopplung der Buszahl von sieben auf 14 beim Benchmark-Programm Tonwertkorrektur ein Laufzeitgewinn von nur etwa 4% verursacht und der hohe Aufwand deshalb nicht gerechtfertigt erschien, wurde für weitere Betrachtungen angenommen, dass ein Prozessor mit kontrollflussgesteuertem Datenfluss verwendet wird, in dem sieben explizite Busse realisiert sind. Prinzipiell können damit pro Takt sieben Zugriffe auf den Registerspeicher ausgeführt werden. Tatsächlich wird jedoch für keines der hier betrachteten Benchmark-Programme ein Registerspeicher mit mehr als vier Schreib-Lese-Ports benötigt. Während die Anzahl der Leseports in herkömmlichen Prozessoren i.allg. größer ist, als

die Anzahl der Schreibports (es werden dyadische Operationen ausgeführt), ist dies mit einem Prozessor, wie er in diesem Beitrag beschrieben wird, nicht der Fall, weil Zwischenergebnisse nämlich direkt und nicht über den Registerspeicher weitergereicht werden können. Aus dem selben Grund kann die Kapazität des Registerspeichers auch geringer sein, als in herkömmlichen Prozessoren. Zum Beispiel werden in keinem der hier betrachteten Benchmark-Programme mehr als 15 Register benötigt, obwohl keine speziellen Maßnahmen durchgeführt wurden, um Registerinhalte in den Hauptspeicher ein- oder auszulagern.

7 Architektur des Zen-1

Auf Basis der hier präsentierten vorläufigen Untersuchungsergebnissen wird zur Zeit der in Bild 4 dargestellte Prozessor Zen-1 in VHDL für ein FPGA realisiert. Er enthält acht explizite Busse, drei arithmetisch logische Einheiten, eine Multiplikations-, eine Lade-, eine Speicher-, eine Sprungverarbeitungseinheit sowie einen 16 Register umfassenden Vierport-Registerspeicher und eine Konstanteneinheit. Die dargestellte Struktur besitzt viele der zu Bild 1 bereits beschriebenen Details. Statt eines getrennten Befehls- und Datenspeichers wird in Bild 4 jedoch ein gemeinsamer Hauptspeicher verwendet, der über ein zentrales Bus-Interface an den Prozessor angebunden ist. Damit Zugriffe auf Befehle und Daten dennoch parallel ausgeführt werden können, sind zwei getrennte Caches geringer Kapazität vorgesehen. Außerdem sind zwei Speicherverwaltungseinheiten geplant. Um die Programmierung zu vereinfachen ist außerdem eine Sprungvorhersageeinheit vorgesehen, so dass Verzweigungen verzögerungsfrei bearbeitet werden können. Der Umgang mit verzögerten Sprungbefehlen entfällt daher.

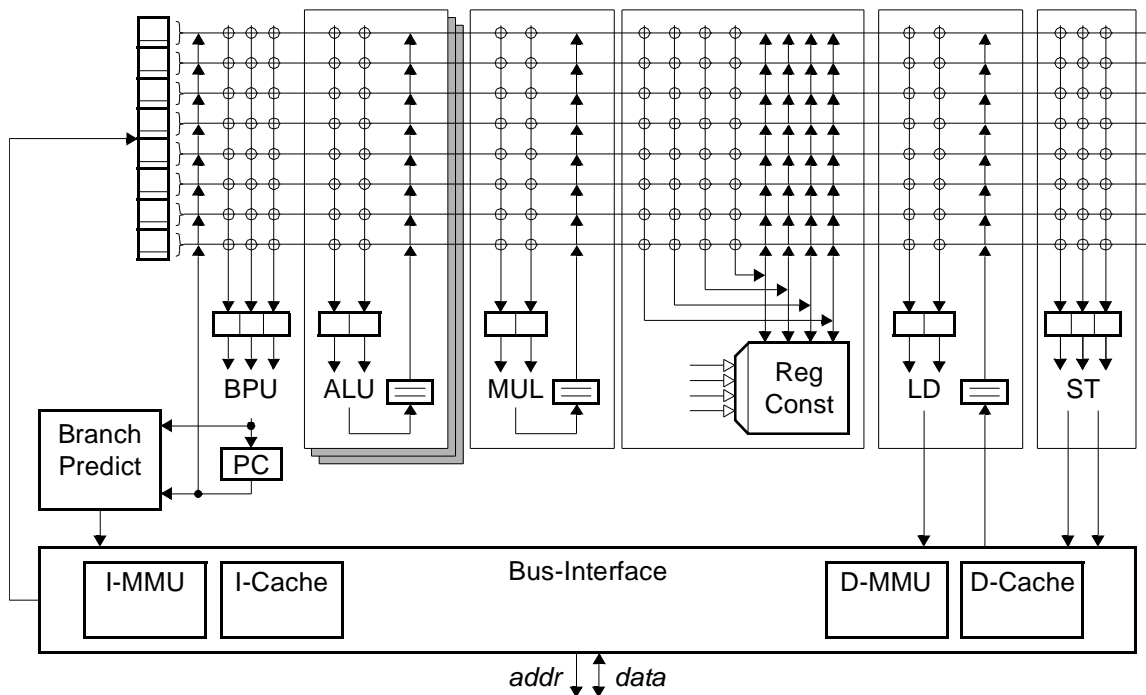


Bild 4. Geplante Architektur des Zen-1 Prozessors.

Ein Befehlsword ist 128 Bit breit. Jede der acht darin codierten Transportoperationen enthält eine 9 Bit breite Ziel- und 7 Bit breite Quelladresse. Die Aufteilung des Zieladressenraums ist in Tabelle 7 die des Quelladressenraums in Tabelle 8 angegeben. Spalte 1 in Tabelle 7 enthält jeweils eine Kennung der zu verwendenden Funktionseinheit. In den folgenden drei mit Zieladresse überschriebenen Spalten sind die Adressbits angegeben, in denen die

auszuführenden Operationen codiert sind. Die beiden letzten Spalten dienen der Modifikation der Wirkungsweise einer Operation. So können über die drei Modusbits z.B. weiche Verkettungen aktiviert, der Skalierungsfaktor bei Speicherzugriffen definiert oder die Adressinterpretation bei Sprungbefehlen festgelegt werden. Die Unterteilung der Quelladressen ist in Tabelle 8 dargestellt. Der lesende Zugriff auf die drei arithmetisch logischen Einheiten, die Multiplikationseinheit und die Ladeinheit erfordern eine sechs Bit breite Adresse. Ein zusätzliches Bit steuert, ob bei einem Zugriff das entsprechende Ergebnis konsumiert werden soll oder nicht (z). Für einen Lesezugriff auf die Sprungverarbeitungseinheit sind zwei Adressen vorgesehen: Eine über die der aktuelle Stand des Befehlszählers ermittelt werden kann und eine, über die zusätzlich ein Trap ausgelöst wird. Die Angabe, ob der gelesene Wert konsumiert oder nicht konsumiert werden soll, ist hier nicht sinnvoll, da der Befehlszähler in jedem Takt einen neuen Wert produziert, der nicht immer gelesen werden wird.

Tabelle 7. Aufteilung des Zieladressraums im Zen-1.

Einheit	Zieladresse			Modus	
	Symbol	Code	Bemerkung	Wirkung	Code
000, 001, 010	aluX (add).l	000	Addition	aluX.r = 0	000 ^a
	aluX (sub).l	001	Subtraktion	aluX.r = 1	001
	aluX (and).l	010	AND-Operation	aluX.r = 2	010
	aluX (or).l	011	OR-Operation	aluX.r = 4	011
	aluX (xor).l	100	XOR-Operation	aluX.r = -1	100
	aluX (max).l	101	Maximalwertbildung	aluX.r = this	101
	aluX (min).l	110	Minimalwertbildung	aluX.r = ldX	110
	aluX (cond).l	111	Bedingte Zuweisung	-	111
000, 001, 010	aluX.r	001	Rechter Operand	-	000
011	mul (add).l	000	Addition	mul.r = 1	000 ^b
	mul (sub).l	001	Subtraktion	mul.r = 2	001
	mul (mul).l	010	Multiplikation	mul.r = 4	010
	mul (sfr).l	011	Shift Right	mul.r = 8	011
	mul (sfl).l	100	Shift Left	mul.r = 16	100
	mul (rol).l	101	Rotate Left	mul.r = this	101
	mul (ror).l	110	Rotate Right	mul.r = ldX	110
	mul (asr).l	111	Arithmetic Shift Right	-	111
011	mul.r	001	Rechter Operand	-	000
100	ld (sb).base	000	Load Signed Byte	scale = 0	000
	ld (ub).base	001	Load Unsigned Byte	scale = 1	001
	ld (sh).base	010	Load Signed Half	scale = 2	010
	ld (uh).base	011	Load Unsigned Half	scale = 4	011
	ld (sw).base	100	Load Signed Word	scale = 8	100
	ld (uw).base	101	Load Unsigned Word	ld.index = alu0	101 ^c
	ld (l).base	110	Load Long	ld.index = alu0 * size	110
	ld (setb).base	111	Load and Set Byte		

Tabelle 7. Aufteilung des Zieladressraums im Zen-1.

Einheit	Zieladresse			Modus	
	Symbol	Code	Bemerkung	Wirkung	Code
100	ld.index	111	Index	-	101
101	st (b).base	000	Store Byte	scale = 0	000
	st (h).base	001	Store Half	scale = 1	001
	st (w).base	010	Store Word	scale = 2	010
	st (l).base	011	Store Long	scale = 4	011
				scale = 8	100
				st.index = 0, st.data = 0	101
				st.index = alu1 * size, st.data = 0	110
				st.index = alu1 * size	111
101	st.index	111	Index	-	000
101	st.data	111	Daten	-	001
110	b (lt,m).addr	000	Branch on Less Than	abs, signed, b.r = 0	000 ^d
	b (le,m).addr	001	Branch on Less Or Equal	abs, unsigned, b.r = 0	001 ^{de}
	b (gt,m).addr	010	Branch on Greater Than	rel, signed, b.r = 0	010 ^d
	b (ge,m).addr	011	Branch on Greater Or Equal	rel, unsigned, b.r = 0	011 ^{de}
	b (eq,m).addr	100	Branch on Equal	abs, signed	100
	b (ne,m).addr	101	Branch on Not Equal	abs, unsigned	101 ^{de}
	b (a,m).addr	110	Branch Always	rel, signed	110
	b (rex,m).addr	111	Return on Exception	rel, unsigned	111 ^{de}
110	b.r	111	Rechter Operand	-	000
110	b.l	111	Linker Operand	-	001
111	<i>rn</i>	00 <i>n</i>	Register	-	<i>nnn</i>
111	<i>sn</i>	01 <i>n</i>	Spezialregister	-	<i>nnn</i>

- a. Nicht erlaubt in Kombination mit aluX (sub).l.
- b. Nicht erlaubt in Kombination mit mul (sub).l.
- c. Nicht erlaubt in Kombination mit ld (setb).base.
- d. Nicht erlaubt in Kombination mit b (a,m).addr, b (rex,m).addr.
- e. Nicht erlaubt in Kombination mit b (eq,m).addr, b (ne,m).addr.

Für Registerinhalte ist ein z-Bit ebenfalls nicht vorgesehen, obwohl dies zur Steuerung des Datenflusses sinnvoll sein kann (eine entsprechende Abwandlung ist für einen Nachfolgeprozessor des Zen-1 geplant). Im vorliegenden Fall ist in einer Quelladresse neben einer Basiskennung (001) daher nur die Registernummer enthalten, auf die sich der Lesezugriff bezieht. Ganz ähnlich wird auch auf Spezialregister zugegriffen. Neben je einem Bedingungs- bzw. Statusregister pro Einheit wird u.a. das in Abschnitt 4 beschriebene chain-Register auf diese Weise adressiert. Die obere Hälfte des Quelladressraums wird verwendet, um auf die Konstanten des Wertebereich -32 bis +31 zuzugreifen. Um auch breitere Konstanten verarbeiten zu können, ist es als Sonderfall möglich, zwei oder drei Transportoperationen zu einer Einheit zusammenzufassen und auf diese Weise 16 oder 32 Bit Konstanten zu erzeugen.

Tabelle 8. Aufteilung des Quelladressraums im Zen-1.

Symbol	Quelladresse Codierung	Bemerkung
aluX	000z000 ^a	Ergebnis der arithmetisch logische Einheit Nr. 0 lesen
	000z001	Ergebnis der arithmetisch logische Einheit Nr. 1 lesen
	000z010	Ergebnis der arithmetisch logische Einheit Nr. 2 lesen
mul	000z011	Ergebnis der Multiplikationseinheit lesen
ld	000z100	Geladenes Dateum lesen
pc	0101110	Befehlszähler lesen (Sprungeinheit)
trappc	0101111	Befehlszähler lesen und Trap auslösen (Sprungeinheit)
rn	001nnnn	Registerinhalt lesen
sn	010nnnn	Spezialregisterinhalt lesen
#c5	1cccccc	5 Bit Konstante lesen
#c16	0000110	16 Bit Konstante lesen
#c32	0000111	32 Bit Konstante lesen

a. Das z-Bit steuert, ob das Ergebnis mit dem Zugriff konsumiert werden soll oder nicht.

8 Zusammenfassung und Ausblicke

In dieser Ausarbeitung wird eine Prozessorarchitektur beschrieben, die nach dem hier als kontrollflussgesteuerten Datenfluss bezeichneten Prinzip arbeitet. Dabei werden Operationen im Befehl nicht als Einheit codiert, sondern, indem die Operanden mit Hilfe von Transportoperationen zu den entsprechenden Einheiten übertragen und die erzeugten Ergebnisse nach Abwarten der Latenzzeiten abgeholt werden. Eine zu herkömmlichen Prozessoren vergleichbare Arbeitsgeschwindigkeit wird erreicht, indem mehrere Transportoperationen parallel ausgeführt werden. Dabei begrenzt u.a. die Anzahl der verfügbaren Busse das Maß an erreichbarer Parallelität, wobei zusätzlich zu den expliziten Bussen noch Direktverbindungen zwischen Funktionseinheiten bestehen können. Vorteil eines Prozessors mit kontrollflussgesteuerten Datenfluss ist neben dessen einfachen Aufbau vor allem die gute Erweiterbarkeit. So können Funktionseinheiten hinzugefügt werden, ohne die Struktur eines bestehenden Prozessors umfassend verändern zu müssen, und zwar vor allem deshalb, weil die Decodierung der Operationen dezentral in den Funktionseinheiten geschieht.

Obwohl die Realisierung des Zen-1 noch nicht abgeschlossen ist, sind schon jetzt einige Änderungen für den geplanten Nachfolger Zen-2 festgelegt. So sollen in einem 128 Bit Befehlswort statt acht nur sieben Transportoperationen codiert werden, um auf diese Weise pro Transportoperation zwei weitere Bits zur Verfügung zu haben. Sie werden für zusätzliche zum Teil virtuelle Funktionseinheiten benötigt. Außerdem ist geplant, jedes Register als Ringpuffer zu realisieren, um ein Verhalten ähnlich den Funktionseinheiten zu erreichen. Gegebenenfalls werden die Speicherzellen als einstufige Ringpuffer organisiert, indem Synchronisationsbits hinzugefügt werden. Der Registerspeicher sowie die Ausgangsringpuffer der Funktionseinheiten sollen eine direkte Anbindung an den Hauptspeicher erhalten, über den ein Kontextwechsel ohne explizites Zutun des Benutzers möglich wird. Für eine bessere Skalierbarkeit der Operationsparallelität soll außerdem der Decoder erweitert werden, damit variabel breite Befehle ausgeführt werden können. Falls mehrere Transportoperationen die selbe Quelle adressieren, soll hierfür nur ein einzelner expliziter

Bus erforderlich sein. Im Zen-1 wie auch im Zen-2 schließt ein Sprungbefehl grundsätzlich ein Paket von parallelen Transportoperationen ab. Für die fernere Zukunft soll dies nicht mehr gelten. Pro Befehl können dann mehrere Sprungbefehle, ähnlich wie im Itanium 2 von Intel, ausgeführt werden.

9 Literatur

- [1] Ernst, R.; Embedded System Architecture; Hardware/Software Co-Design: Principles and Practice; Kluwer Academic Publishers; Dordrecht, Norwell, New York, London, 1997.
- [2] Henritzi, F.; Bleck, A.; Moore, R.; Klauer, B.; Waldschmidt, K.; ADARC: A New Multi-Instruction Issue Approach; International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '96); Sunnyvale, CA, USA; August 1996.
- [3] Intel Corporation; Intel Itanium™ Architecture Software Developer's Manual; Rev. 2.0; Santa Clara; Dec. 2001.
- [4] Kessler, R. E.; McLellan, E. J.; Webb, D. A.; The Alpha 21264 Microprocessor Architecture; International Conference on Computer Design; Austin, Texas, Oct. 1998.
- [5] Krewell, K.; Fujitsu's SPARC64 V is Real Deal; Microprocessor Report, Scottsdale, Arizona; Oct. 2002.
- [6] Menge, M.; Beschreibung eines Übersetzer Übersetzer zur lexikalischen Analyse von Bitmustern; Technischer Bericht – Rote Reihe 2001/17; ISSN 1436-9915; Berlin 2001.
- [7] Menge, M.; Schoppa, I.; Hardwaresynthese von Programmiermodellen; GI/ITG/GMM Workshop Tübingen: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen; Tübingen 2002.
- [8] Philips Semiconductor; Data Book - TM1300 Media Processor; Philips Electronics North America Corporation, USA; May 2000.
- [9] Philips Semiconductor; TriMedia - TM1000 Preliminary Data Book; Philips Electronics North America Corporation, USA; 1997.
- [10] Strohschneider, J.; Waldschmidt, K.; ADARC: A Fine Grain Dataflow Architecture with Associative Communication Network; Euromicro 94; Liverpool; September 1994.
- [11] Sudharsanan, S.; MAJC-5200: A High Performance Microprocessor for Multimedia Computing; Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia (PDIVM'2000); Cancun, Mexico; May 2000.
- [12] Sun Microsystems, Inc.; MAJC™ Architecture Tutorial; White Paper; Palo Alto, USA; September 1999.
- [13] Texas Instruments Incorporated; TMS320C62x/C67x CPU and Instruction Set Reference Guide; Literature Number: SPRU189C; Dallas, Texas; March 1998.
- [14] Transmeta Corporation; Crusoe™ Processor Model TM5800 Product Brief; Santa Clara; 2001.
- [15] Tremblay, M.; Chan, J.; Chaudhry, S.; Conigliaro, A. W.; Tse, S. S.; The MAJC Architecture: A Synthesis of Parallelism and Scalability; IEEE Micro; Vol. 20 No. 6; November-December 2000.