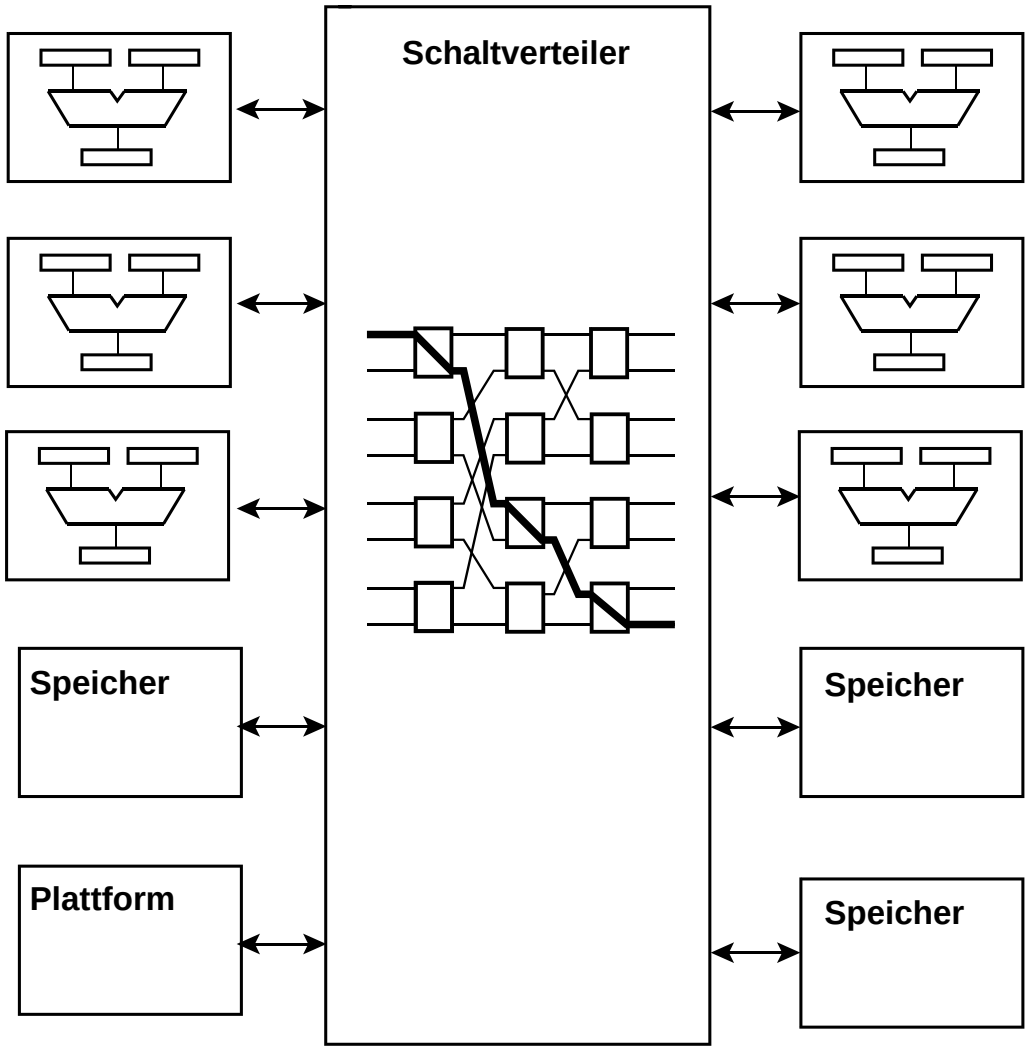
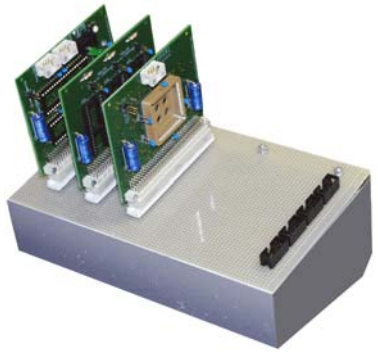
Ressourcenalgebra. Eine alternative Grundlegung der Rechnerarchitektur

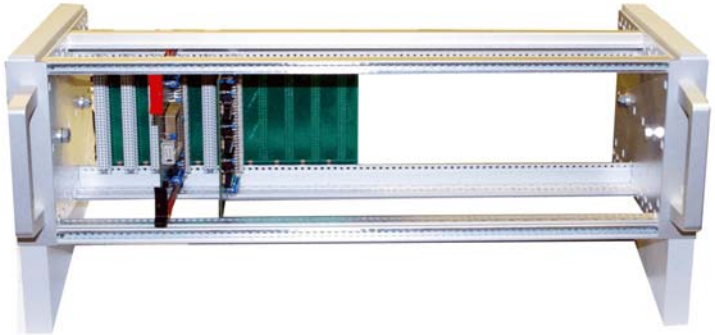
6. 2. 2014





1. Einführung
2. Wirkprinzipien
3. Leistungsbetrachtungen

Anhang, Literaturverzeichnis



Prof. Dr. Wolfgang Matthes
FH Dortmund

wolfgang.matthes@fh-dortmund.de
http://www.realcomputerarchitecture.com
http://www.controllersandpcs.de

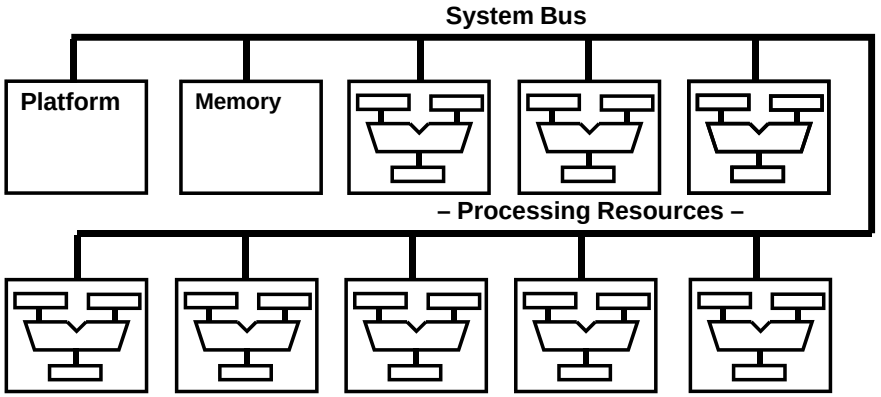
1. Einführung

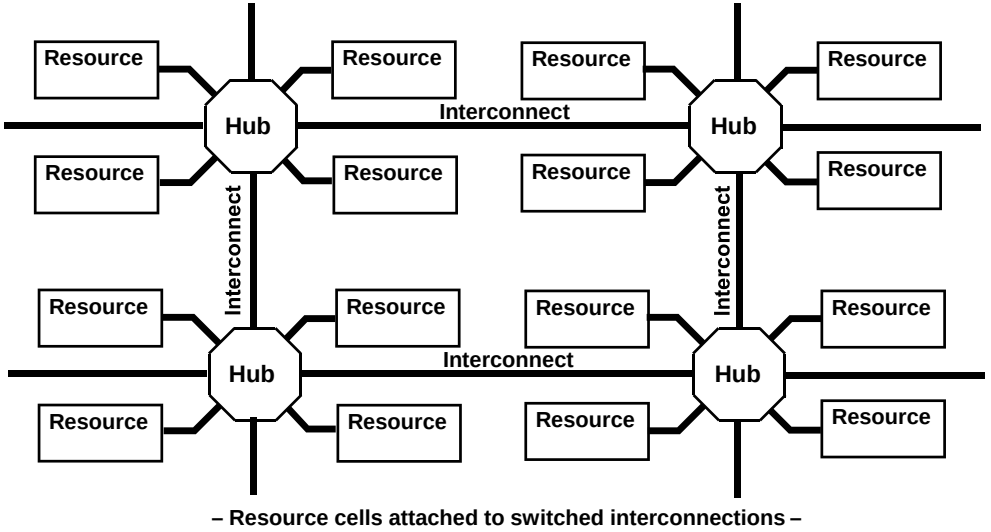
Worum geht es?
Um einen alternativen Ansatz der Rechnerarchitektur:

- Keine herkömmlichen Maschinenbefehle.
- Keine starren Prozessorkerne.

Statt dessen:

- Frei nutzbare Funktionseinheiten (Ressourcen) – und zwar – dem Grunde nach – beliebig viele.
- Eine Anwendungsprogrammschnittstelle (API) , die es ermöglicht, ein solches Sortiment von Funktionseinheiten als frei programmierbare Universalmaschine zu betreiben.





Was war der Ausgangspunkt?
Die Grundlagenforschung auf dem Gebiet der Recherarchitektur. Will man nicht sklavisch nachbauen oder gar nur kaufen oder herunterladen, ergeben sich Fragen, die es durchaus in sich haben:

- Wie kann man Architekturentscheidungen wissenschaftlich, also objektiv und verifizierbar, begründen und rechtfertigen?
- Was einbauen und was nicht?
- Weshalb ausgerechnet soundso viele Register?
- Weshalb diese Befehlswirkungen, diese Adressierungsverfahren und keine anderen?
- Usw.

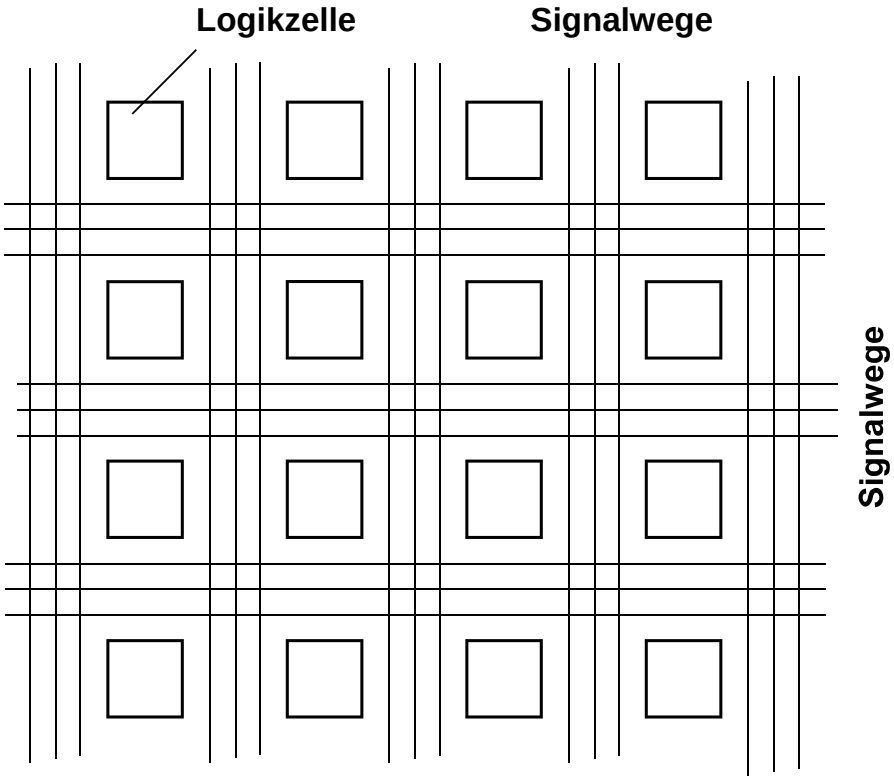
Die heutzutage vorherrschenden Architekturen sind im Grunde mehrere Jahrzehnte alt. Sie beruhen auf seinerzeitigen Erfahrungsgrundlagen, Anwendungserfordernissen und Gefühlsentscheidungen, vor allem aber auf den Grenzen, die die damals verfügbare Technologie gesetzt hat.

Heutzutage ist es nicht mehr erforderlich, bei den Transistoren und der Speicherkapazität um jeden Preis zu sparen. Deshalb können wir uns einen alternativen Ansatz leisten. Er beruht darauf, von üblichen Maschinenbefehlen und Prozessorkernen abzugehen und der Anwendungsprogrammierung eine beliebige Anzahl frei verfügbarer Ressourcen zur Verfügung zu stellen.

Endziel ist der hart verdrahtete und frei programmierbare Universalprozessor. Es versteht sich von selbst, daß man FPGAs verwenden kann, um solche Maschinen zu bauen. Es ergibt sich aber auch die Möglichkeit, neuartige Architekturen für programmierbare Logikschaltkreise zu entwickeln.

Programmierbare Logikschaltkreise
Wir wollen jetzt die theoretischen Grundlagen und die Entwicklungsgeschichte beiseite lassen und uns zunächst auf typische Probleme der programmierbaren Logikschaltkreise beschränken.

Wie sieht es in einem FPGA aus?



Die Schaltung entsteht, indem Logikzellen über die Signalwege miteinander verbunden werden.

Hart verdrahtete und programmierbare Logik

Hart verdrahtete Logik:

- Die Gatter und Schaltnetze sind bis auf den Transistor optimiert.
- Die Signalwege werden so kurz wie möglich gehalten.
- Die Taktfrequenzangaben im Datenblatt betreffen die Taktfrequenzen der Anwendungspraxis.

Programmierbare Logik:

- Alles muß mit Logikzellen aufgebaut und über vorgegebene Signalwege untereinander verbunden werden. Auch die Taktverteilung ist vorgefertigt.
- Die Taktfrequenzangaben im Datenblatt betreffen die Maximalwerte der eingebauten Taktverteilung und Taktaufbereitung.

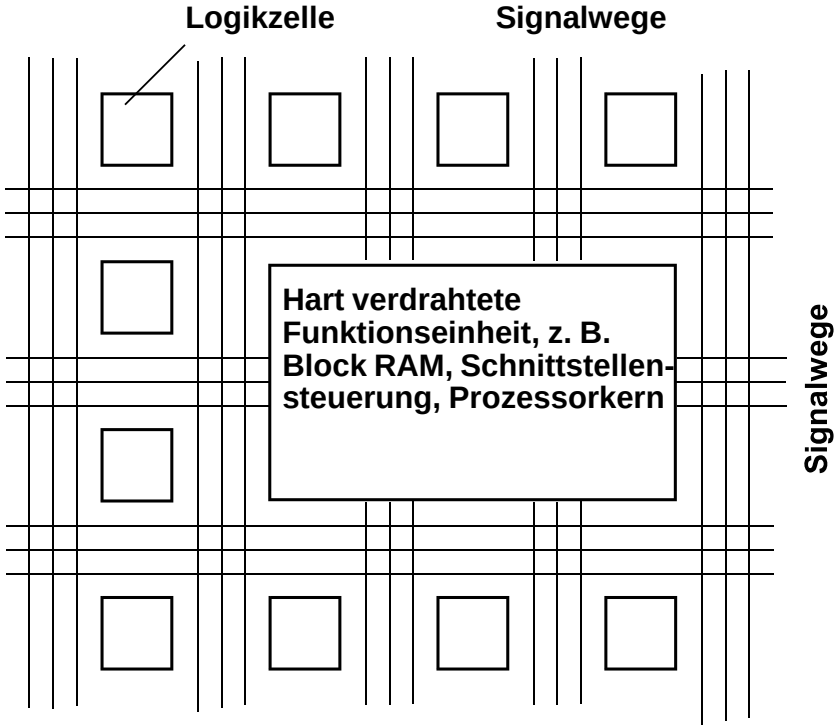
Zahlenbeispiele aus der Praxis:

- Transistorverbrauch: Ein FPGA mit 75 Millonen Transistoren ist nicht in der Lage, eine Prozessorschaltung aufzunehmen, die hart verdrahtet 7,5 Millionen Transistoren erfordert.
- Taktfrequenzen: Aus den Datenblättern der Hochleistungsprozessoren und FPGAs kann ein Verhältnis der Taktfrequenzverlangsamung (Slowdown) zwischen (Richtwerte) 10:1 und 2:1 abgelesen werden. Für pauschale Überlegungen ist eine Verlangsamung im Verhältnis 10:1 eine zweckmäßige Annahme (2 GHz Universalprozessor, 200 MHz FPGA). Je höher die Taktfrequenz im FPGA, desto größer die Entwurfsschwierigkeiten.

FPGAs sind offensichtlich brauchbare Anwendungslösungen, aber keineswegs Optimallösungen (mag doch die Werbung behaupten, was sie will...).

Was ist programmierbar, was fest?

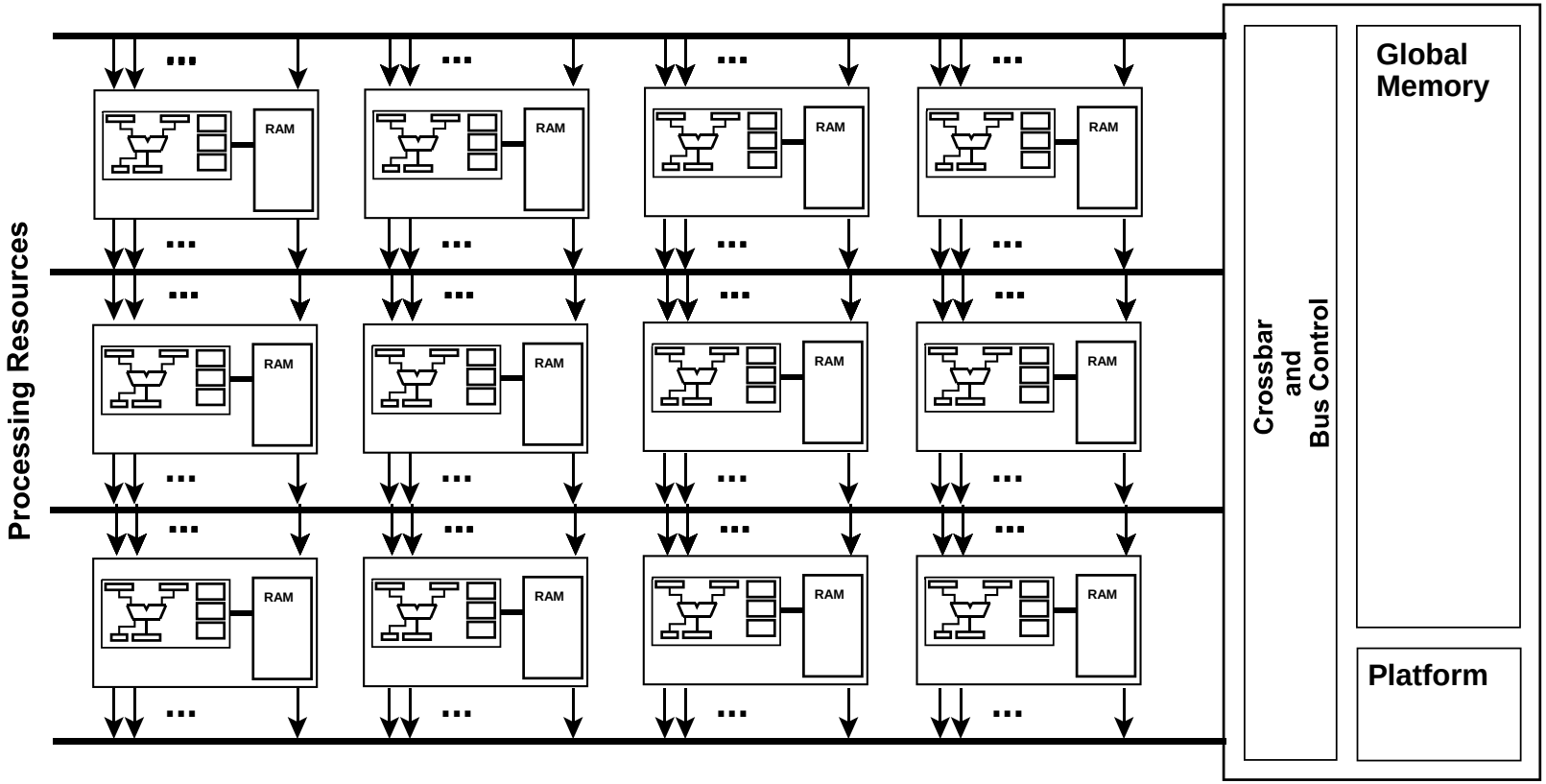
Es gehört zum Stand der Technik, mehrere bis viele hart verdrahtete Prozessorkerne auf einem Schaltkreis unterzubringen. Wie aber sollen die vielen Kerne so beschäftigt werden, daß sie auch etwas Vernünftiges leisten?



Die Schaltung ist ein Verbund von Logikzellen (für die anwendungsspezifischen Schaltungsteile) und hart verdrahteten Funktionseinheiten.

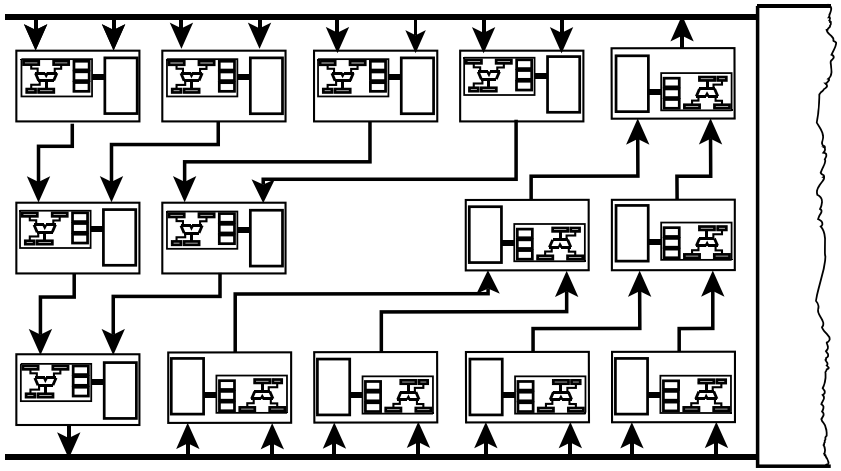
Die Struktur einer Einzweckschaltung entspricht weitgehend dem Datenflußschema des Anwendungsproblems. Sie arbeitet deshalb von Hause aus parallel. Herkömmliche Universalprozessoren hingegen müssen eigens dazu gebracht werden, sich an der Lösung eines gemeinsamen Anwendungsproblems zu beteiligen (Parallelisierung).

Eine Alternative:
Probieren wir es mit einer sozusagen mittleren Körnigkeit – weder Logikzellen noch komplette Prozessoren, sondern Funktionseinheiten, die man immer gebrauchen kann und die auf dem RTL-Niveau optimiert werden. Das sind die Ressourcen.



Die Ressourcen untereinander verbinden
Hart verdrahtete Schaltungen auf einer Siliziumfläche unterzubringen, ist eine Aufgabe, die offensichtlich beherrscht wird. Wie aber soll man sie untereinander verbinden? Wir suchen zunächst nach praxisbrauchbaren Lösungen, die sich hart implementieren lassen. Hier einige Beispiele:

- Ein gemeinsames Bussystem.
- Mehrere Bussysteme.
- Schaltverteiler und Punkt-zu-Punkt-Verbindungen.
- Schiebewege, z. B. in Ketten- oder Ringtopologie.
- Invertierter Binärbaum.



Der invertierte Binärbaum – eine Topologie, an die man nicht immer gleich denkt. Sie hat aber ihre Vorteile: Sie entspricht 1:1 dem Datenflußschema einer Stackmaschine. Die Auswertung geschachtelter Ausdrücke kann direkt auf einen invertierten Binärbaum abgebildet werden. Alle Verbindungen zwischen den Ressourcen sind kurze Punkt-zu-Punkt-Verbindungen.

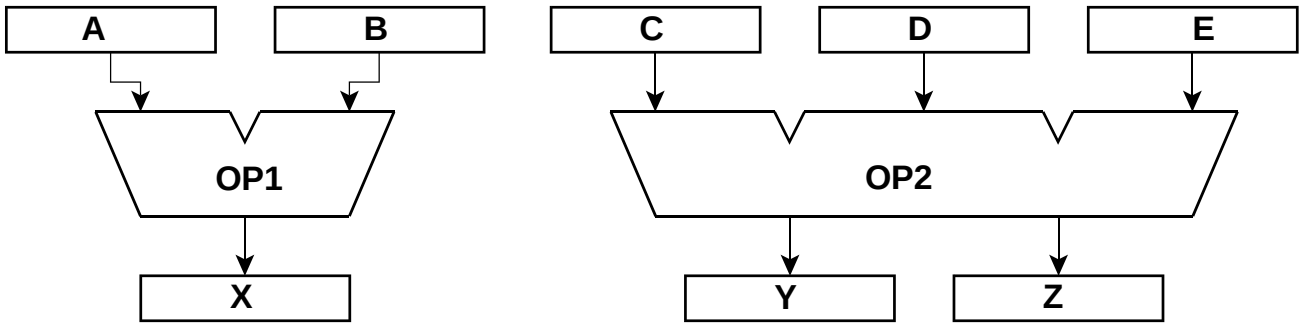
Zudem verbleibt der Rückgriff auf die programmierbaren Verbindungswege der herkömmlichen FPGAs. Es versteht sich von selbst, daß man eine Sammlung von hart verdrahteten Funktionseinheiten – durch entsprechendes Programmieren der Verbindungswege – so verdrahten kann, daß sich eine anwendungs-spezfische Spezialmaschine (Datenfluß- oder Datenstrukturmaschine) ergibt. Wir aber sind letzten Endes an einer wirklichen Universalmaschine interessiert. Hierzu brauchen wir eine Maschinenarchitektur und eine Anwendungsprogrammschnittstelle (API).

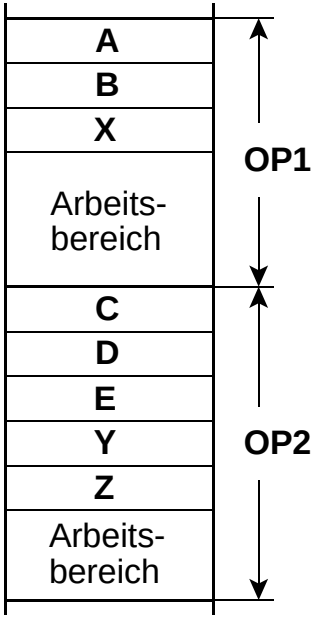
ReAl Computer Architecture

- Eine Maschinenarchitektur ist gegeben durch eine Menge von Datenstrukturen und eine Menge von Operationen.
- Um die Operationen auszuführen, braucht man Funktionseinheiten.
- Diese Funktionseinheiten bezeichnen wir als Ressourcen.
- Sie sollen bis auf die RTL-Ebene optimiert sein und (zumindest fiktiv) in unbeschränkter Anzahl zur Verfügung stehen.
- Das eigentlich Neue ist die Anwendungsprogrammschnittstelle (API), die es ermöglicht, solche Strukturen für herkömmliche Programme auszunutzen.
- Die Architekturdefinition umfaßt eine Menge von Ressourcen und eine Menge von Datenstrukturen.
- Ressourcen führen bestimmte Operationen über Daten aus, die bestimmten Datentypen entsprechen.
- Im Grunde ist es eine algebraische Struktur. Deshalb wird die Architektur mit *ReAl* = *Re*ssourcen-*Al*gebra bezeichnet.
- Im Englischen kann sie eigentlich gar nicht anders heißen als ***ReAl Computer Architecture***. Das Wortspiel ist Absicht (pun intended).
- Das Modell einer Ressource ist eine Hardware, die bestimmte Informationswandlungen ausführt, also aus gegebenen Daten (an den Eingängen) neue Daten (an den Ausgängen) errechnet.
- Jede Maschine läßt sich auf eine Ressourcenanordnung zurückführen.
- Zu den wichtigsten Zielen der Architekturentwicklung gehört es, diese Ressourcen so gut wie möglich auszunutzen.

2. Wirkprinzipien

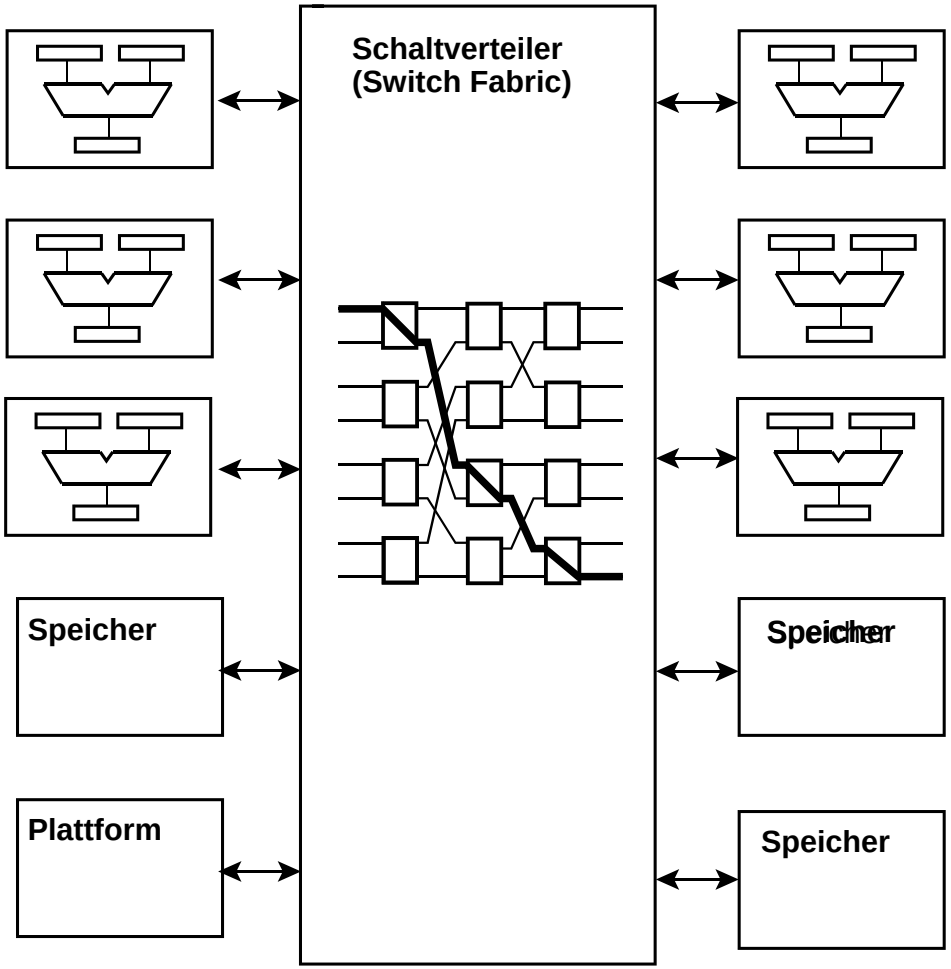
Wie sehen elementare Ressourcen aus?



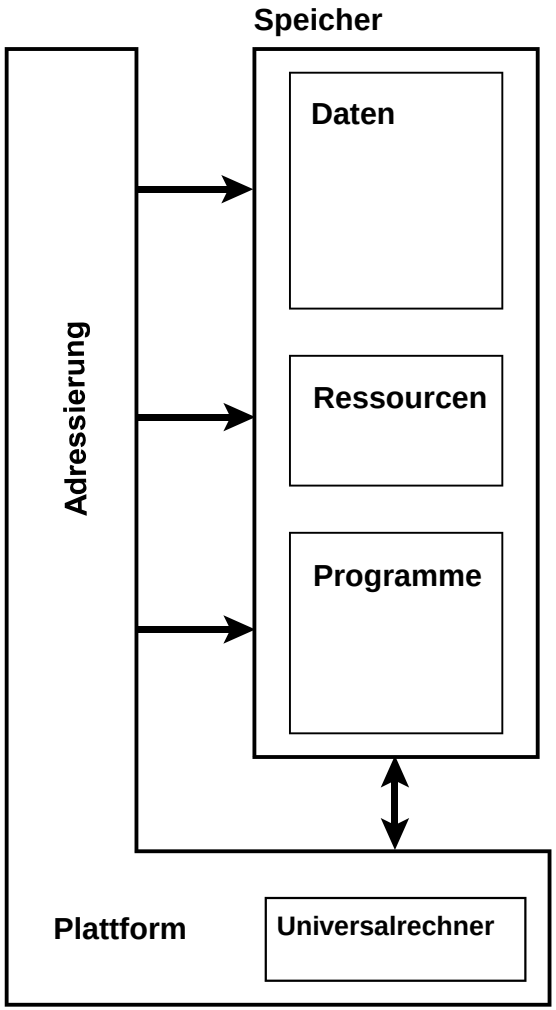


- Eine Ressource errechnet aus gegebenen Daten (an den Eingängen) neue Daten (an den Ausgängen).
- Ressourcen können als hart verdrahtete Schaltungen ausgeführt werden. Die Daten werden dann in Registern gehalten. Im Grunde sind es RTL-Strukturen.
- Nach derzeitigem Arbeitsstand ist die einzelne Ressource vorzugsweise eine RTL-Struktur ohne Pipelining (viele einfache statt wenige komplexe Ressourcen). Es liegt nahe, derartige Ressourcen über Bussysteme oder geschaltete Punkt-zu-Punkt-Signalwege miteinander zu verbinden. Hierbei sind nur nur einige wenige Kommunikationsmuster bzw. Verbindungstopologien von entscheidender Bedeutung, so daß der Aufwand in Grenzen gehalten werden kann.
- Alternativ dazu ist es möglich, die Ressourcenfunktionen zu emulieren. Die Daten befinden sich dann in entsprechenden Speicheranordnungen oder (beim herkömmlichen Universalrechner) im Arbeitsspeicher bzw. im Registersatz.

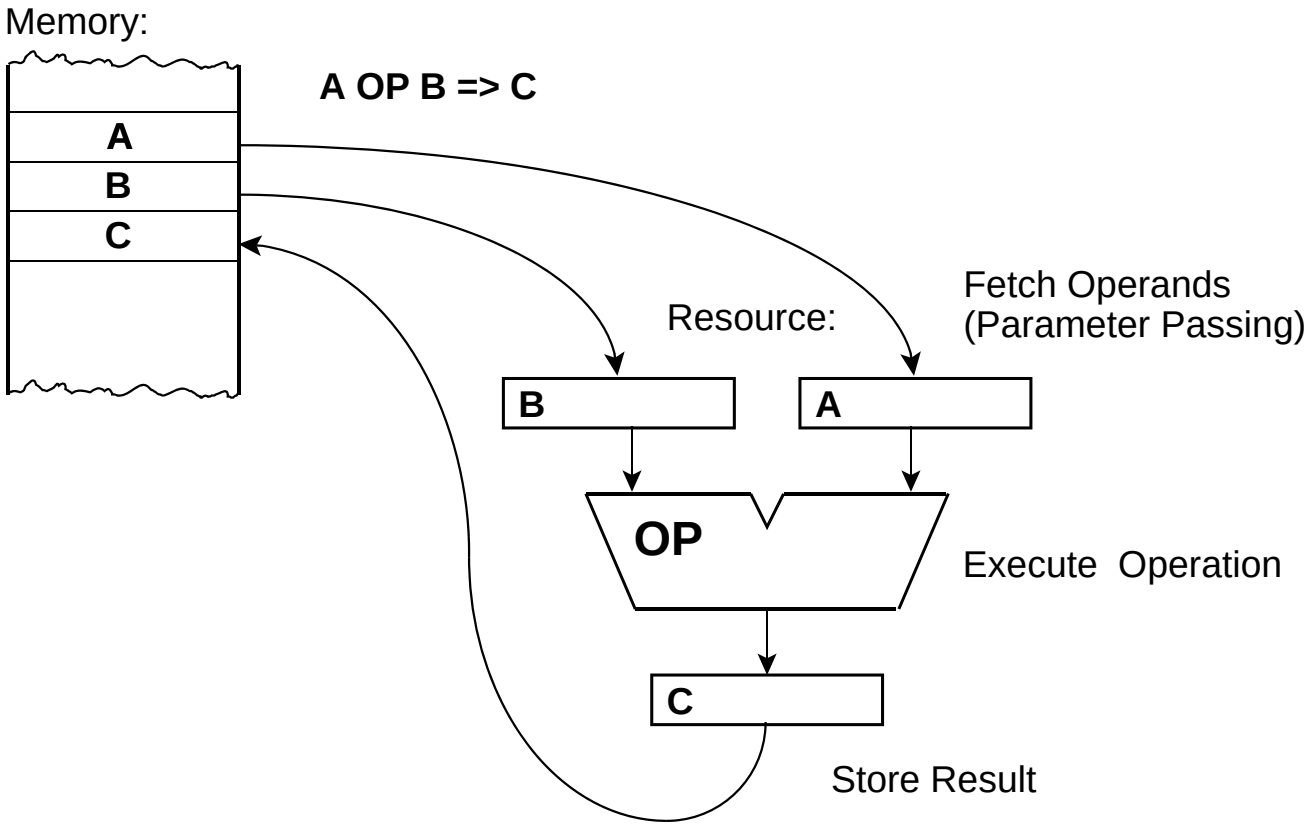
Hier sind die Ressourcen an einen Schaltverteiler angeschlossen:



Eine solche Einrichtung kann die Ressourcenfunktionen emulieren:



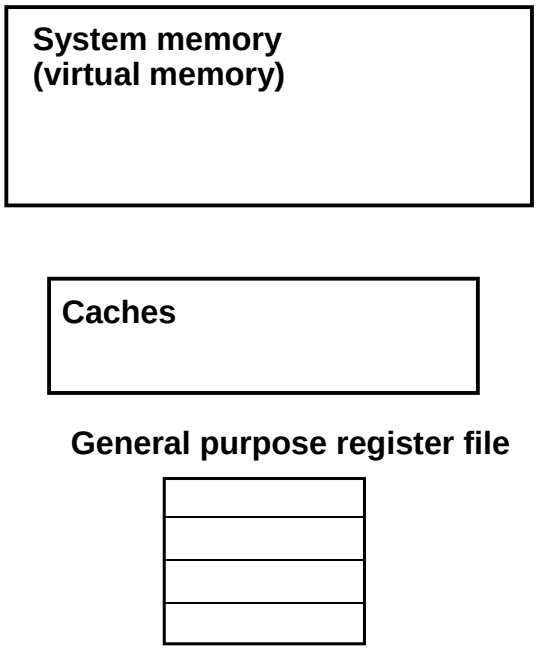
So wird eine elementare Ressource verwendet:



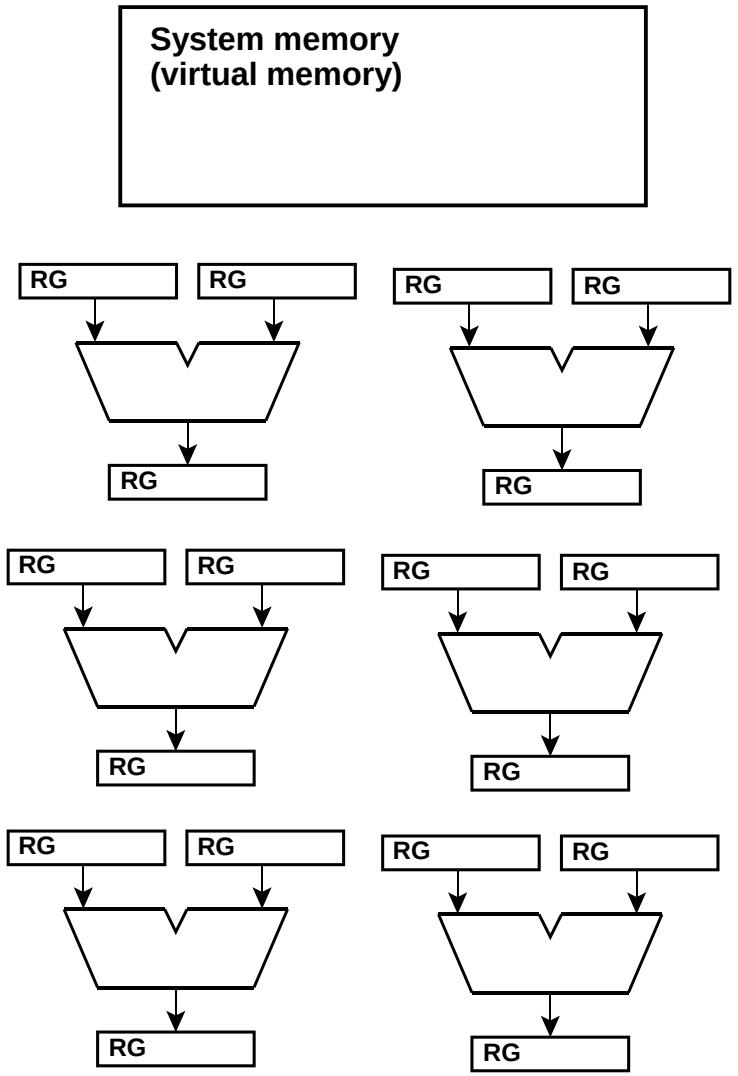
Das sieht unscheinbar aus. Es sollen aber beliebig viele Ressourcen zur Verfügung stehen...

Herkömmliche und ReAl-Maschinen unterscheiden sich in der Speicherhierarchie:

a) Conventional storage hierarchy



b) Basic storage model of a ReAl machine



Wie werden mit solchen Ressourcen Anwendungsaufgaben gelöst?

- Maschinenprogrammierung mit einer ReAl-Anwendungsprogrammschnittstelle (API) führt letzten Endes darauf, eine dem jeweiligen Anwendungsproblem angepaßte Spezialmaschine zu beschreiben, die aus sovielen Funktionseinheiten (Ressourcen) besteht, wie jeweils erforderlich sind.
- Diese fiktiven Maschinen werden während der Informationsverarbeitungsvorgänge dynamisch auf-, um- und abgebaut.
- ReAl-Maschinen können tatsächlich als Schaltung gebaut, in FPGAs synthetisiert oder emuliert werden.
- Zum Emulieren kann man ReAl-Prozessoren schaffen, die für diese Aufgabe optimiert sind, oder herkömmliche Prozessoren einsetzen.
- Sogar bei der Emulation auf herkömmlichen Prozessoren können Leistungsvorteile erwartet werden*. Das wurde experimentell nachgewiesen.

Wozu man den grundlegenden Ansatz u. a. verwenden kann:

- ReAl ist eine interne Zwischenspache, die es gestattet, den innewohnenden (inhärenten) Parallelismus des Anwendungsproblems oder der Programmierabsicht zu erkennen, zu formulieren und somit zu bewahren (beispielsweise als Bytecode).
- ReAl ist eine technische Lehre zur Auslegung universeller und spezieller Computer.
- ReAl ist eine Anwendungsprogrammschnittstelle (API) zur bestmöglichen Ausnutzung von Ressourcen.

*****: Im Vergleich zur Emulation beispielsweise von Stackmaschinen (JVM) oder Registermaschinen (Dalvik).

ReAl-Maschinenbefehle (Operatoren)
Die elementaren Befehle heißen Operatoren. Die ReAl-Anwendungsprogrammschnittstelle (API) kommt mit acht Operatortypen aus:

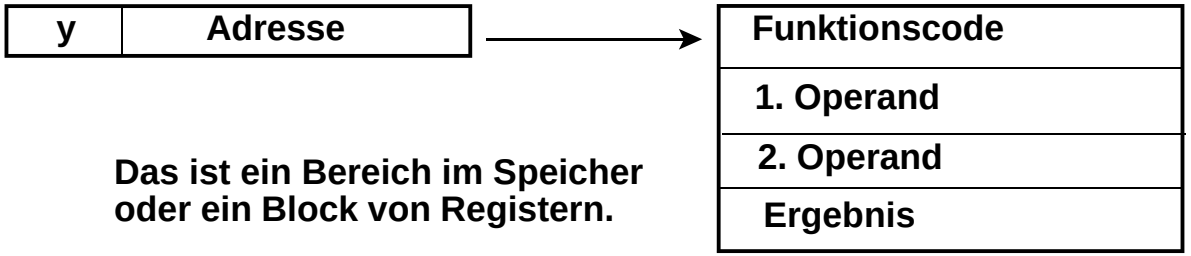
1. Ressourcen auswählen = s-Operator (select).
2. Mit Parametern versorgen = p-Operator (parameter).
3. Operation(en) ausführen (laufen lassen): y-Operator (yield) = umgekehrtes Lambda*.
4. Ergebnisse abholen = a-Operator (assign).
5. Transportieren von Parametern zwischen Ressourcen = l-Operator (link).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

- Ein Verarbeitungsvorgang (Programmablauf) besteht in der Benutzung von Ressourcen im Laufe der Zeit.
- Ressourcen werden bei Bedarf aus dem Ressourcenvorrat entnommen und bei Nichtgebrauch zurückgegeben.
- Die zur Steuerung der Verarbeitungsvorgänge vorgesehenen Anweisungsangaben (Operatoren) betreffen nur die grundlegenden Verfahrensschritte des Anforderns, Transportierens, Auslösens usw., nicht aber konkrete Maschinenoperationen.
- Komplexere Ressourcen können aus elementaren Ressourcen aufgebaut werden.

*****: ReAl-Programmierung ist im Grunde angewandter Lambda-Kalkül...

Jede Ressource weiß selbst, was sie zu tun hat*.

- Der y-Operator kann mehrere Operationen parallel anstoßen.
- Es ist egal, ob es sich um eine Operation der Hardware oder um einen Unterprogrammaufruf handelt.
- Es ist richtiger Lambda-Kalkül.
- Der Funktionscode kann beliebig kompliziert sein.
- Die Operatoren kommen mit kurzen Operationscodefeldern aus.
- Nur so kann wirkliche Parallelarbeit funktionieren.
- Man kann mit ein paar Bits im y-Operator viele Ressourcen steuern, braucht also keine extrem breiten Befehle (VLIW).
- Der Ressource ist es gleichgültig, von wem oder woher sie angestoßen wird, ob von einem y-Operator oder durch Eintreffen eines entsprechenden Parameters (p-Operator, l-Operator, Vorwärtsverkettung).
- Bei entsprechender Organisation (z. B. Vorwärtsverkettung) können die Ressoucen sofort losrechnen, wenn alle Parameter bereitstehen.
- Was die Ressourcen tun werden, ergibt sich bereits aus einer Analyse der Konfiguration im statischen Zustand, also nicht erst aus dem Befehlsaufruf zur Laufzeit. Wenn sie – wodurch/von wem auch immer – dazu veranlaßt werden sollen, etwas anderes zu tun als der Nutzer erwartet, so sollte sich das erkennen lassen...

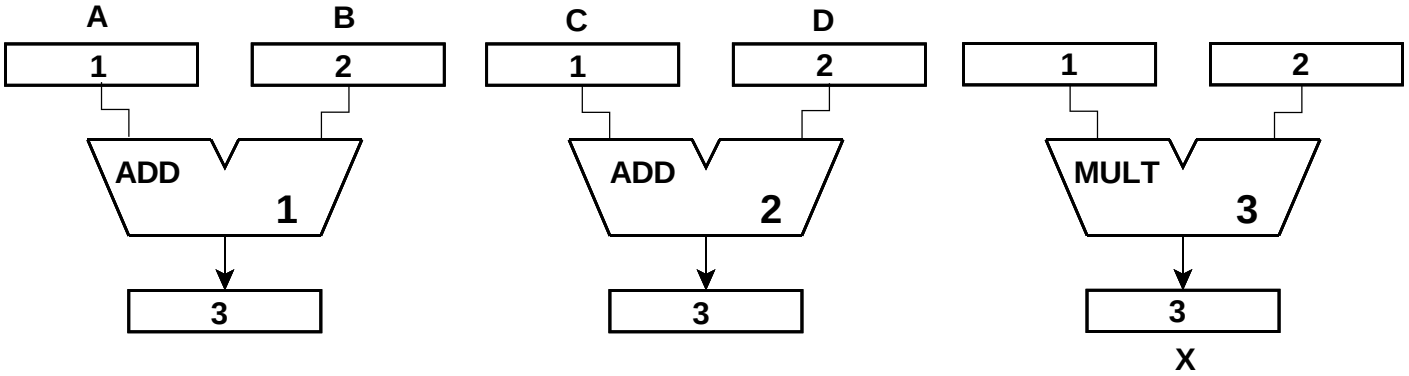


*: Das hat sich als entscheidend erwiesen. Kompromißlösungen – etwa eine Kombination des ReAl-Parametermodells mit herkömmlichen Machinenbefehlen – werden deshalb nicht weiter untersucht (wennschon, dennschon...).

Ein einfaches Beispiel:

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

Wir brauchen diese Ressourcen:



Das Programm in ausführlicher Notation:

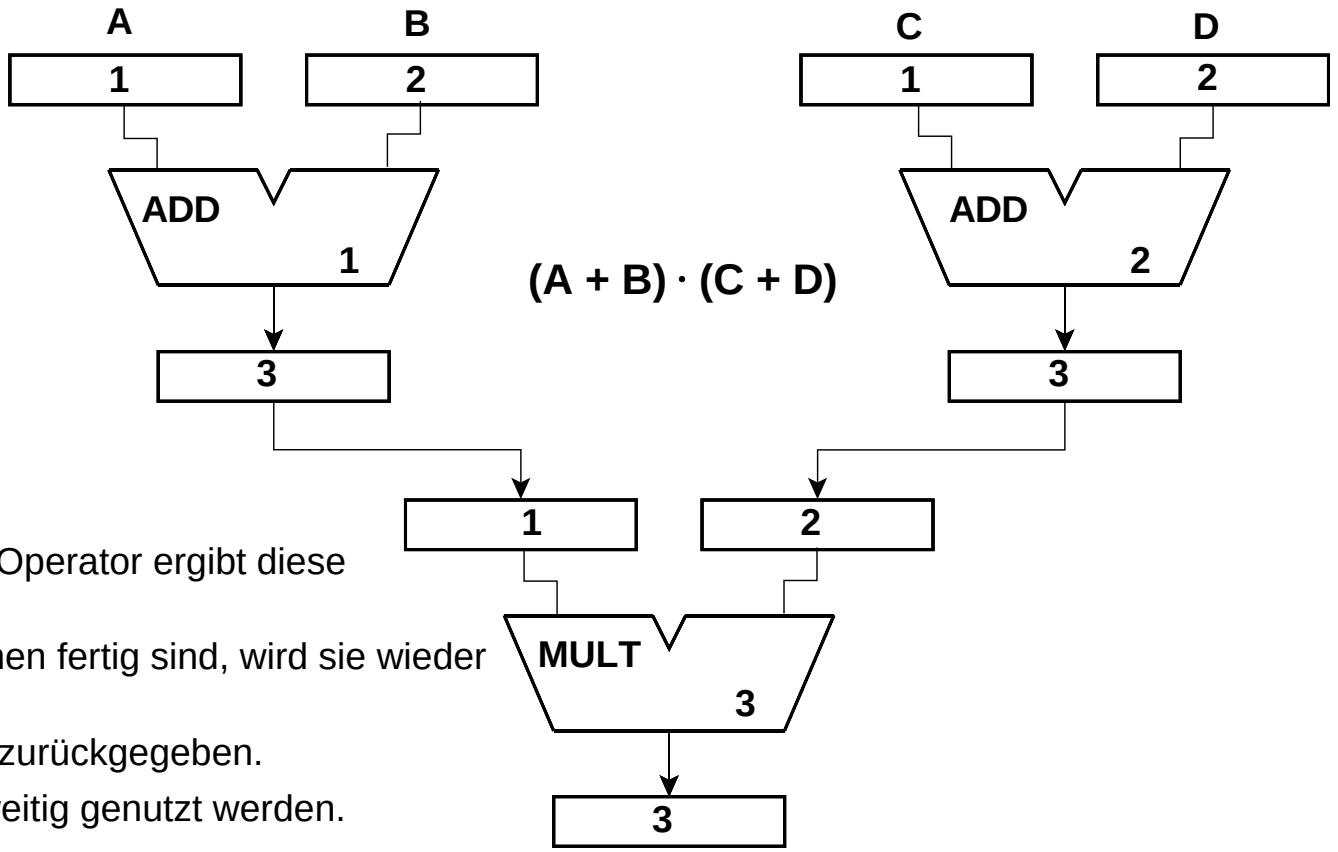
s (ADD, ADD, MULT)
p (A => 1.1)
p (B => 1.2)
p (C => 2.1)
p (D => 2.2)
y (1)
y(2)
l (1.3 => 3.1)
l (2.3 => 3.2)
r (1, 2)
y (3)
a (3. 3 => X)
r (3)

In verkürzter Notation:

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

Und jetzt als Datenflußschema (also mit Verkettung):

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



- Die Verkettung mittels c-Operator ergibt diese Spezialschaltung.
- Wenn wir mit dem Rechnen fertig sind, wird sie wieder zerlegt,
- Die Ressourcen werden zurückgegeben.
- Sie können dann anderweitig genutzt werden.

Und das soll – wem auch immer – überlegen sein?

- Es sieht nur auf den ersten Blick umständlich aus.
- Es sind aber nur Textcodes, die das Prinzip veranschaulichen sollen.
- Der Code soll möglichst alle relevanten Gesichtspunkte des Verarbeitungsablaufs selbst beschreiben, u. a. die auszuführenden Funktionen, den Datenfluß und den innewohnenden (inhärenten) Parallelismus.
- Es versteht sich von selbst, daß dies zusätzliche Bits erfordert. Dafür werden aber die Schaltungen, die diesen Code interpretieren, nicht übermäßig kompliziert. .

Folgendes konnte nachgewiesen werden:

1. Die Wirkprinzipen des v.Neumann-Rechners lassen sich mit strukturell und aufwandsseitig plausiblen Ressourcen darstellen (u. a. einschließlich der Sprungvoraussage, der Sprungzielpufferung, der typischen Verfahren der Speicheradressierung und der in den üblichen Programmiersprachen vorgesehenen Kontrollstrukturen).
2. Es lassen sich hinreichend kompakte, praxisbrauchbare Maschinencodes angeben.
3. Das Prinzip ist leistungsseitig überlegen, weil es die verfügbaren Ressourcen besser ausnutzt. Der Effekt tritt bereits bei der Emulation auf herkömmlichen Universalrechnern auf. Die hierbei einzugehenden Kompromisse in Hinsicht auf Speicherbedarf und zusätzliche Schritte der Maschinenprogrammfertigung sollte man sich heutzutage leisten können...

Wie werden ReAl-Programme aufgesetzt?
Der grundsätzliche Ansatz besteht darin, zur Compilierzeit zunächst eine fiktive Maschine zu bauen, und zwar unter der Annahme, daß von allem genügend da ist:

- Hardware spielt keine Rolle,
- Speicherkapazität spielt keine Rolle,
- die rechentechnischen Voraussetzungen der Maschinenprogrammfertigung spielen keine Rolle.

Es gibt jederzeit genügend Ressourcen...

- Das ist zunächst eine theoretische Annahme – die Hypothese vom (nahezu) unbeschränkten (transfiniten) Ressourcenvorrat.
- Maschinenprogramme werden typischerweise so erzeugt, als ob beliebig viele Ressourcen zur Verfügung stünden.
- Dadurch, daß rücksichtslos immer neue Ressourcen angefordert werden, ergibt sich ein Programmtext, der den inhärenten Parallelismus des jeweiligen Programms in allen Einzelheiten erkennen läßt.
- Dann ist die fiktive (unbeschränkte) Ressourcenanordnung auf die praktisch verfügbare (endliche) Ressourcenkonfiguration abzubilden. Das kann zur Compilierzeit oder zur Laufzeit erfolgen.

Grundsätzliche Vorteile...

- Durch Anwendung des ReAl-Programmiermodells kann der den Verarbeitungsabläufen innewohnende (inhärente) Parallelismus in dem Maße ausgenutzt werden, in dem tatsächlich Hardware verfügbar ist.
- Schaltmittel zur Konflikterkennung, Ablaufwiederholung, Befehlsbestätigung usw. sind nicht erforderlich.
- Speicher und Verarbeitungsschaltungen können direkt miteinander verbunden werden, so daß sich kürzeste Zugriffswege ergeben.
- Diese Vereinfachungen bieten die Möglichkeit, die Taktfrequenz zu erhöhen, Pipeline-Stufen einzusparen (Verkürzung der Latenzzeiten) und auf einer gegebenen Schaltkreisfläche mehr oder leistungsfähigere Verarbeitungsschaltungen anzuordnen.
- Werden programmierbare Schaltkreise mit hart verdrahteten Ressourcen einer sozusagen mittleren Körnigkeit belegt, so wird die Siliziumfläche besser ausgenutzt. Sie können mit höheren Taktfrequenzen betrieben werden. Wegen der typischen Arbeitsweise solcher elementaren Ressourcen könnte man auch an eine asynchrone Kommunikation denken.
- Da der inhärente Parallelismus unmittelbar aus der Programmierabsicht heraus erkannt wird, ist es möglich, ggf. auch hunderte Verarbeitungswerke gleichzeitig einzusetzen, um den Ablauf des einzelnen Programms zu beschleunigen.

Es versteht sich von selbst, daß man dafür etwas tun muß... Ist das aber wirklich so schlimm?

Wie geht es eigentlich beim herkömmlichen Programmieren zu?

- Wir deklarieren alle Variablen.
- Wir schreiben z. B. C = A + B; hin.
- Der Compiler erzeugt daraus u. a. einen ADD-Befehl.
- Wenn – zur Laufzeit – die Maschine diesen Befehl liest, wirft sie ein Rechenwerk (ALU) an, um die angewiesene Operation auszuführen. Das läuft im Verborgenen ab.

Wie würde nun ein ReAl-Programm erstellt werden?
Der Anwendungsprogrammierer muß sich damit überhaupt nicht beschäftigen...

- Wir deklarieren alle Variablen.
- Wir schreiben z. B. hin: C = A + B; W = U + V; Z = X * W;
- Ein ReAl-konformer Compiler würden den Quelltext durchmustern und erkennen, daß u. a. zwei Addierer und ein Multiplizierer benötigt werden.
- Er würde die entsprechenden Ressourcen anfordern.
- Mit nicht allzuviel mehr (künstlicher) Intelligenz könnte er erkennen, daß der Multiplizierer eigentlich mit dem Ergebnis des zweiten Addierers weiterrechnet. Somit kann er beide Ressourcen miteinander verketten (c-Operator).
- Zur Laufzeit wird die Ressourcenkonfiguration in der Maschine aufgebaut (Setup) und in Betrieb gesetzt.

Ein ReAl-Programm stellt die Tatsache, daß man zu dessen Ausführung Schaltmittel braucht, explizit dar; es ist im Grunde nichts anderes als ein gewöhnliches Programm, nur muß man neben den Variablen auch alle in Anspruch genommenen Verarbeitungsfunktionen deklarieren.

Ein ReAl-Programm kann mit einer Fertigungsanweisung verglichen werden, die die zur Fertigung einer bestimmten Einrichtung nötigen Bearbeitungsvorgänge angibt und darstellt, wie die entsprechenden Maschinen zu belegen sind.

– Das Maschinenprogramm ist eine Fertigungsanweisung, die Hardware eine Werkhalle mit Fertigungsmaschinen. –

Wer so etwas bauen will,

...fängt nicht einfach an, irgend ein Stück Blech krummzubiegen und dann zu sehen, wie es wohl weitergehen könnte...

Vielmehr geht es doch wohl so:

1. Alles durchkonstruieren.

2. Dann nachsehen, was man braucht, um das Ganze wirtschaftlich fertigen zu können.

- Was brauchen wir?
- Welche Teile müssen wo durchlaufen?
- Wie ist das alles vernünftig aufzustellen?

Übertragen wir das auf das Computerwesen...

Wir wollen so etwas herstellen und laufen lassen. Auch hier wird es wohl nichts Gescheites werden, wenn wir einfach draufloshacken...

Die Programmierabsicht liegt ausformuliert als Quelltext vor.

Das Programm ist im Grunde eine Fertigungsanweisung...

Wir sehen nach, was wir brauchen, um sie auszuführen:

- Wieviele Rechenwerke? Für welche Operationen?
- Wieviele und welche Adressierungseinrichtungen?
- Was brauchen wir zur Schleifensteuerung?
- Welche Ablaufentscheidungen sind zu treffen?

  Usw.

Für alles, was jeweils zu tun ist, forden wir entsprechende Einrichtungen an. Solche Einrichtungen heißen Ressourcen.

Dabei tun wir – zumindest am Anfang – so, als ob von allem genug da wäre (transfiniter Ressourcenvorrat).

Im Ergebnis der Durchmusterung des Quellcodes (Maschinenprogrammfertigung) ist eine fiktive Maschine entstanden.

Dann fragt es sich, wie sie implementiert wird...

So schwer ist es eigentlich gar nicht...

Der Programmcode:

```
double A[n], B[n], SOP;
int x;

SOP=0;

for (x=0; x<n; x++)
```

```
SOP = SOP + A[x] * B[x];
```

Eine Durchmusterung ergibt sofort, welche Ressourcen benötigt werden:

Iterator (+1) — Wenn er nicht als komplette Ressource vorhanden ist, muß er aus zwei elementaren Ressourcen – einem Addierer und einemVergleicher – zusammengebaut werden.

Adressierung (Selektor)

Gleitkomma-Multiplizierer

Gleitkomma-Addierer, akkumulierend

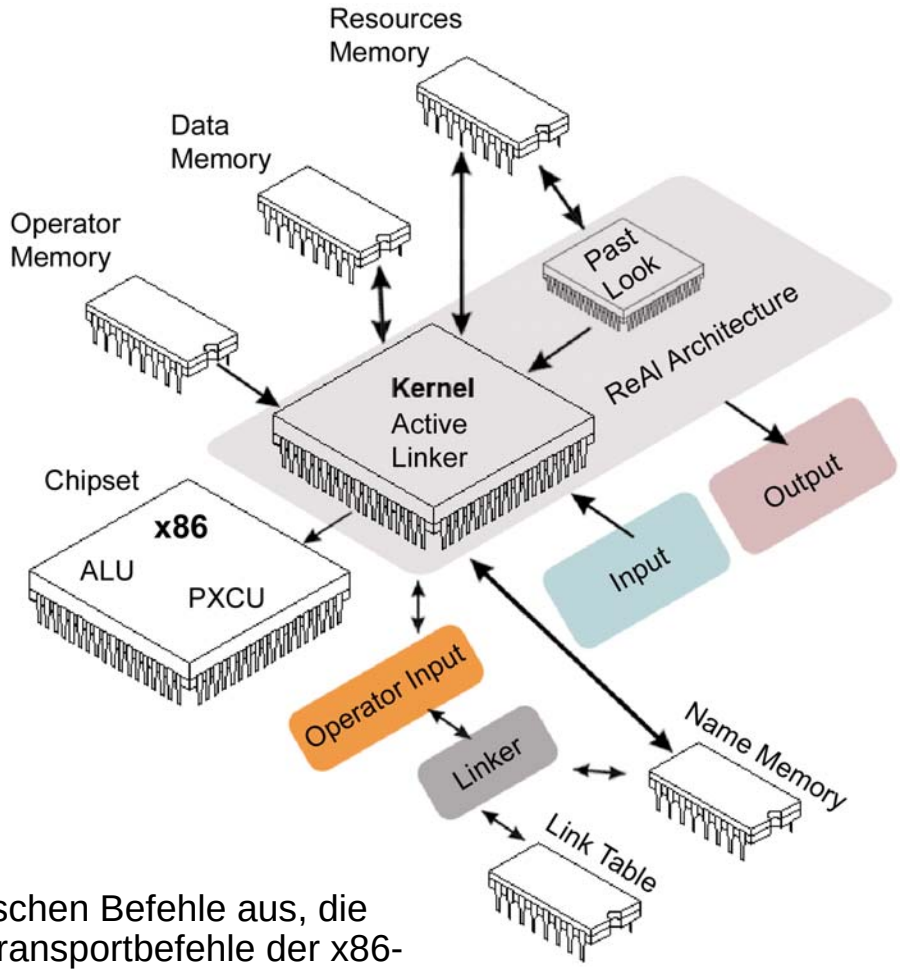
Eine weitere Analyse könnte entscheiden, ob es praktikabel ist, die Schleife aufzurollen.
Im Extremfall werden n Multiplizierer, (ld n) -1 nicht akkumulierende und 1 akkumulierender Addierer angefordert.

3. Leistungsbetrachtungen

- Es ist schwierig, eine fiktive Maschine mit einem voll funktionsfähigen Hochleistungsprozessor zu vergleichen, weil man nicht einfach die Laufzeiten messen kann, sondern die Abläufe Schritt für Schritt einzeln untersuchen muß. Als Beispiel wurde Bresenhams Algorithmus (zum Zeichnen von Geraden) verwendet.
- Das eine Programm wurde in C geschrieben und mit dem Microsoft® Optimizing Compiler Version 16.0. für x86-Prozessoren übersetzt. Der so erzeugte Assemblercode wurde manuell ausgewertet. Das andere Programm wurde unter Nutzung der ReAl API geschrieben und mit dem EmuRix-Emulator auf einem Personalcomputer ausgeführt. Die zur Auswertung erforderlichen statistischen Daten wurden vom Emulator erzeugt.
- Den Untersuchungen liegt eine fiktive Testplattform zugrunde. Sie wird (gedanklich) zweimal implementiert – als x86-Prozessor und als ReAl-Maschine. Dabei werden folgende Annahmen getroffen:

    1. Beide Implementierungen arbeiten mit der gleichen Taktfrequenz.
    2. Beide Implementierungen sind Skalarmaschinen, die in jedem Taktzyklus einen x86-Befehl oder ReAl-Operator ausführen. Die ReAl-Datenstrukturen beruhen auf 32-Bit-Wörtern. Die Verknüpfungsoperationen entsprechen den üblichen Befehlswirkungen der x86-Architektur (es gibt keine Ressourcen mit erweiterten oder komplexeren Verarbeitungsfunktionen). Unter diesen Bedingungen entsprechen die ReAl-Operatoren (in ihrer Formatgestaltung und in dem, was sie prinzipiell auslösen können) weitgehend den üblichen Maschinenbefehlen. Deshalb werden sie nachfolgend auch als Befehle bezeichnet.
    3. Beide Implementierungen bestehen aus zwei fiktiven Funktionseinheiten: aus einem Rechenwerk (Arithmetic/Logic Unit ALU) und einem Verwaltungs- und Steuerwerk (Process Execution Control Unit PXCU).

Die fiktive Testplattform.
Es gibt sie in zwei Ausführungen:
x86 und ReAl.



- Die ALU führt die arithmetischen und logischen Befehle aus, die PXCU die Programmsteuerbefehle. Die Transportbefehle der x86-Architektur (Laden, Speichern usw.) werden als Befehle der Ein- und Ausgabe betrachtet.
- Nur die ALU-Befehle erzeugen Ergebnisse der Anwendungsaufgabe. Die PXCH führt im Grunde nur Hilfsfunktionen aus.
- Die ALU ist die Werkstatt, die PXCU ist die Verwaltung.

Die statische Auswertung

- Die Effektivität wächst proportional mit Verhältnis der ALU-Befehle (Verarbeitungsbefehle) zu den PXCU-Befehlen (Verwaltungsbefehle).
- Die Befehle der Ein- und Ausgabe (Laden, Speichern usw.) ergeben den grundsätzlichen Overhead der x86-Architektur. Die ReAl-Maschine hat keine solchen Befehle, da alle Operanden in den jeweiligen Ressourcen untergebracht sind. Sie hat aber einen anfänglichen Overhead, da die Ressourcen ausgewählt und mit den anfänglichen Operanden versorgt werden müssen (Setup).
- Das Problem des Overhead wird zunächst vernachlässigt. Es wird später berücksichtigt werden.
- Der x86-Code von Bresenhams Algorithmus besteht aus 82 Maschinenbefehlen. Das EmuRix-Programm umfaßt 127 Befehle. Gegenüber dem x86-Code sind das ca. 55 % mehr.

Die Auszählung der Befehle im Maschinencode (statische Auswertung) ergibt:

x86-Maschinenbefehle:

Befehle	Anzahl	Anteil
ALU (Verarbeitung)	21	26 %
PXCU (Verwaltung)	11	13 %
Laden und Speichern	50	61%

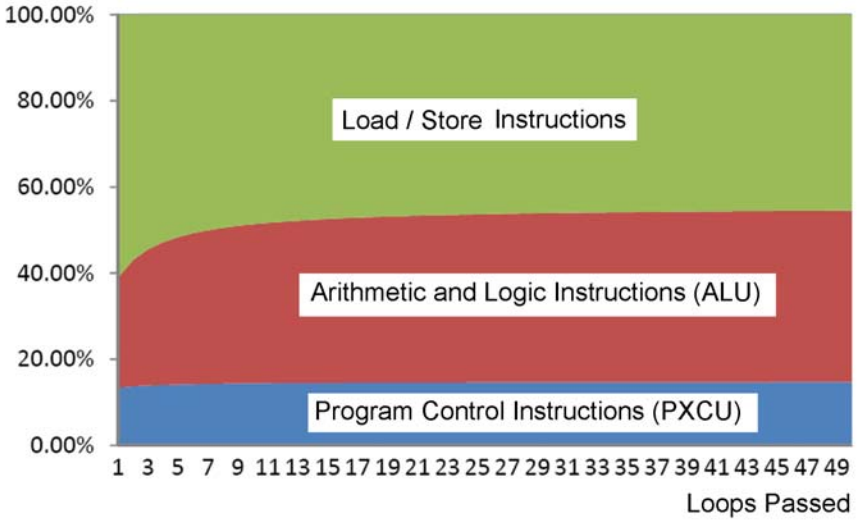
Die Anteile der ALU- und PXCU-Befehle:

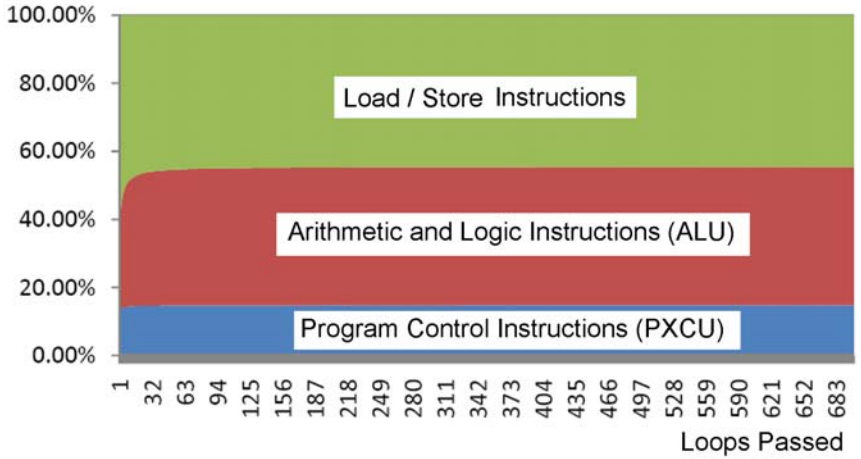
Befehle	x86	EmuRix
Anzahl ALU + PXCU	32 = 100 %	127 = 100 %
ALU (Verarbeitung)	66 %	75 %
PXCU (Verwaltung)	34 %	25 %

Diese Leistungsbetrachtungen sind eine Wiedergabe von Ergebnissen der Thesis [7].

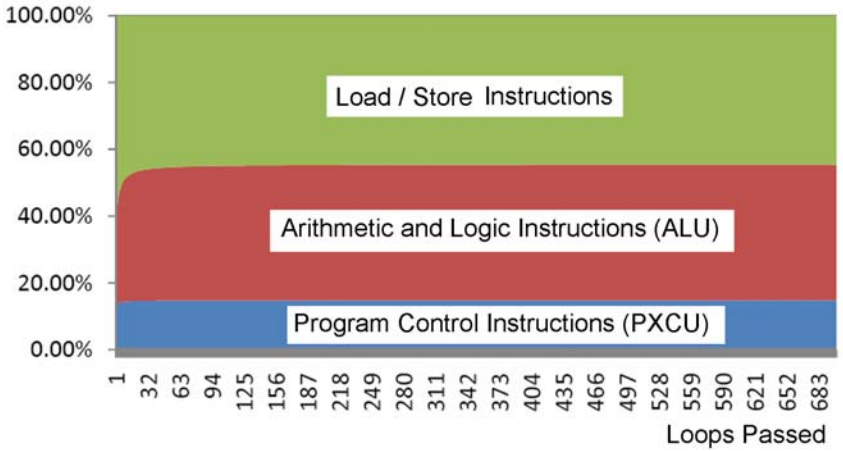
Die Prozessorauslastung während des Betriebs:

a) 50 Schleifendurchläufe

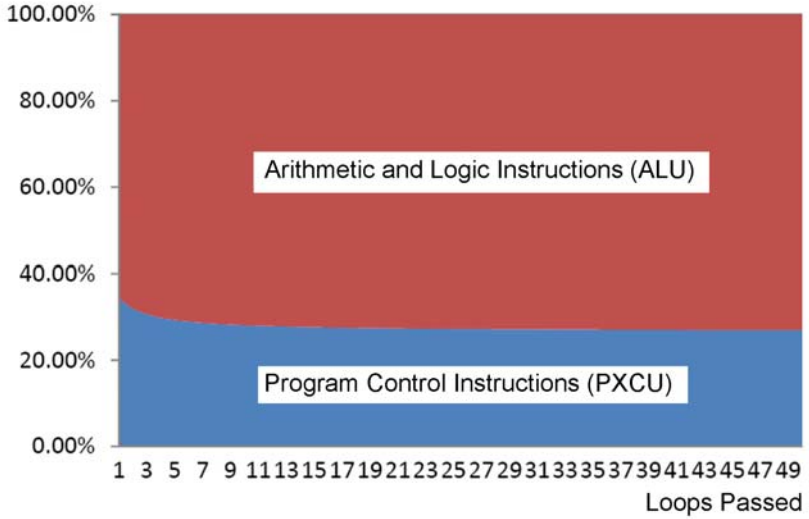




b) 700 Schleifendurchläufe



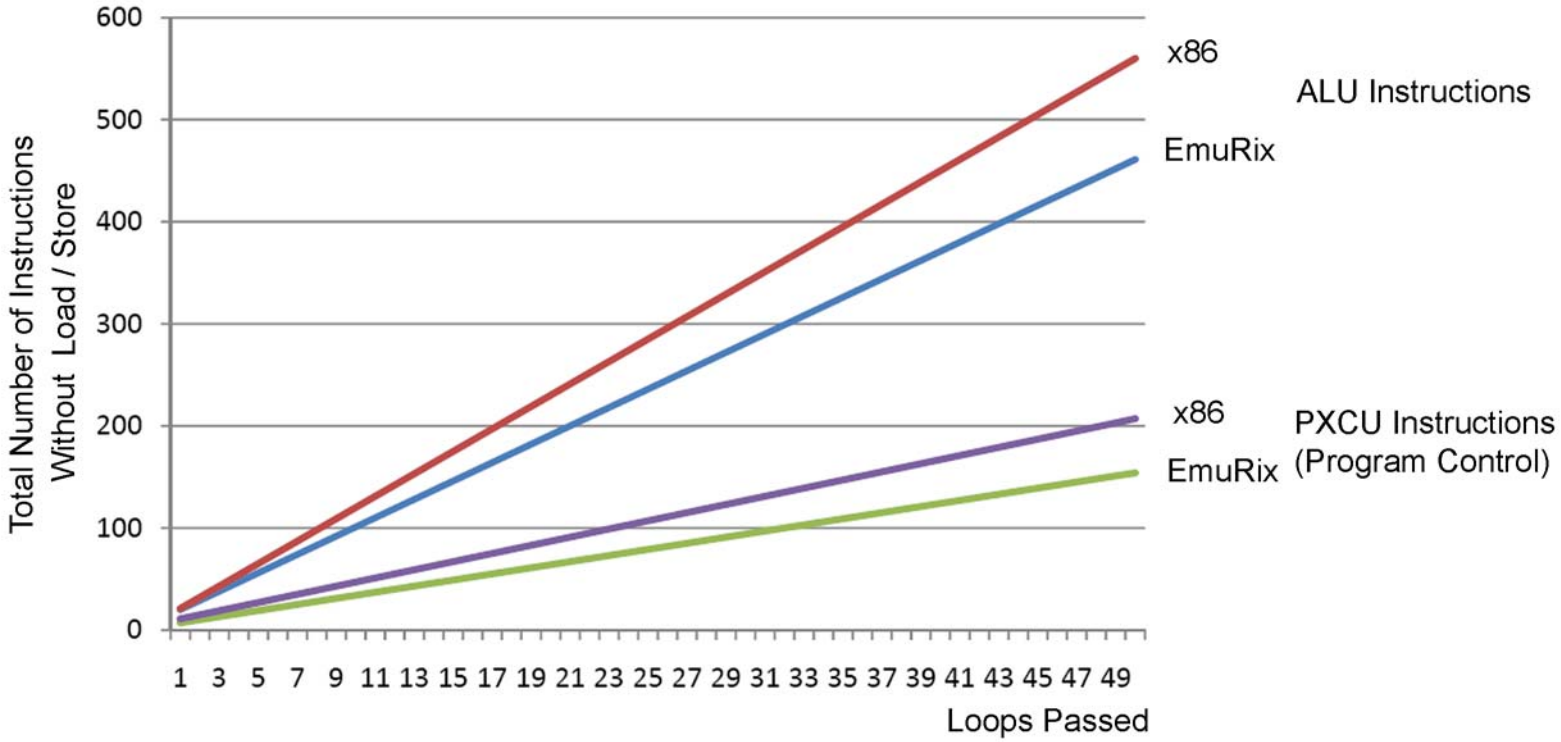
x86



ReAl

- Der Anteil der Verarbeitungsbefehle (ALU-Befehle) ist am Anfang geringer, weil zunächst der Schleifendurchlauf initialisiert werden muß. Die Anzahl der Lade- und Speicherbefehle wird im Laufe der Zeit geringer, fällt aber nie unter 40 %. Auch bei deutlich mehr Schleifendurchläufen werden die Verhältnisse nicht besser.
- Bei EmuRix gibt es keine E-A-Operationen in den Schleifen.

Die Anzahl der ALU- und PXCU-Befehle, die in 50 Schleifendurchläufen ausgeführt werden:



In den Schleifendurchläufen wird das gewünschte Resultat berechnet. Wenn man dazu weniger Befehle braucht, ist die Maschine besser.

Wieviele Befehle werden ausgeführt, welchen Anteil daran hat die Verwaltung?

Befehle	x86	EmuRix (ReAl)
ALU + PXCU	767	615
PXCU (Verwaltung)	25%	18%

Der Leistungsgewinn wird folgendermaßen berechnet:

$$\text{Performance Gain [\%]} = \frac{\text{Accumulated x86 instructions} - \text{Accumulated ReAl instructions}}{\text{Accumulated x86 instructions}} \cdot 100\ \%$$

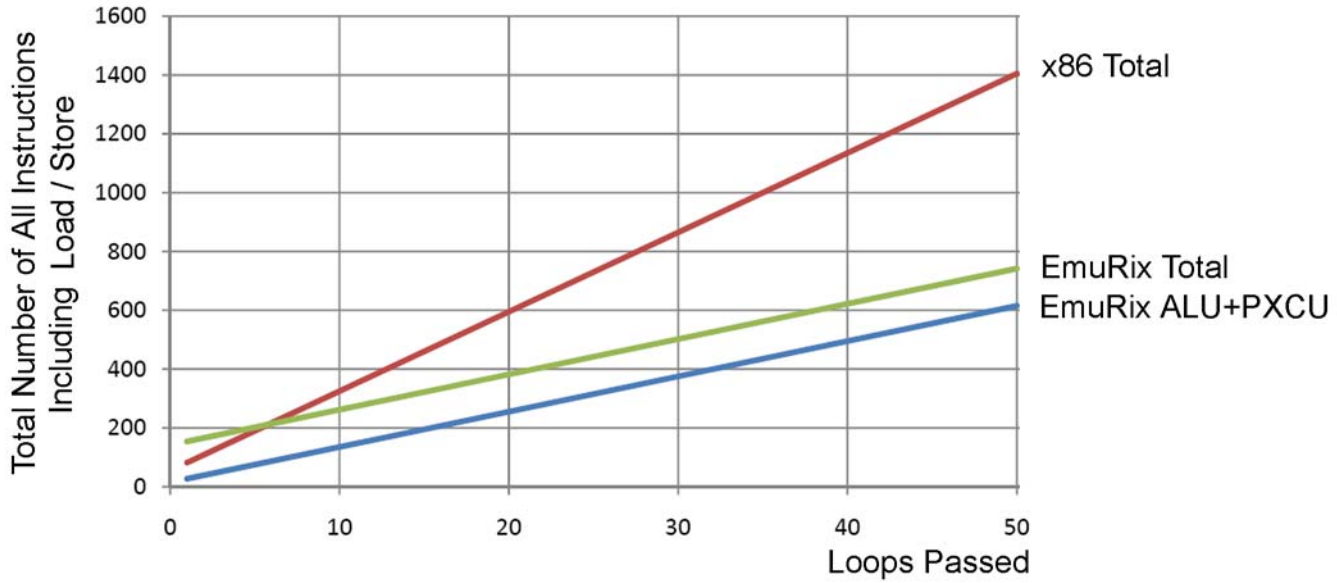
Für die ReAl-Maschine ergeben sich folgende Werte:

- ALU: 17% Mehrleistung.
- PXCU: 25 % Mehrleistung.
- ALU + PXCU: ca. 20 % Mehrleistung.

Hier ist aber der Overhead nicht eingerechnet.

- Der x86 muß Lade- und Speicherbefehle ausführen, und zwar in jedem Schleifendurchlauf erneut.
- Die ReAl-Maschine muß Befehle ausführen, um die Ressourcen einzurichten und miteinander zu verketten. Diese Befehle müssen jedoch nur einmal ausgeführt werden (Setup).
- Deshalb steigt die Mehrleistung mit der Anzahl der Schleifendurchläufe. EmuRix benötigt 127 Befehle, um die Ressourcen anzufordern und die Verknüpfungen zwischen ihnen zu organisieren.

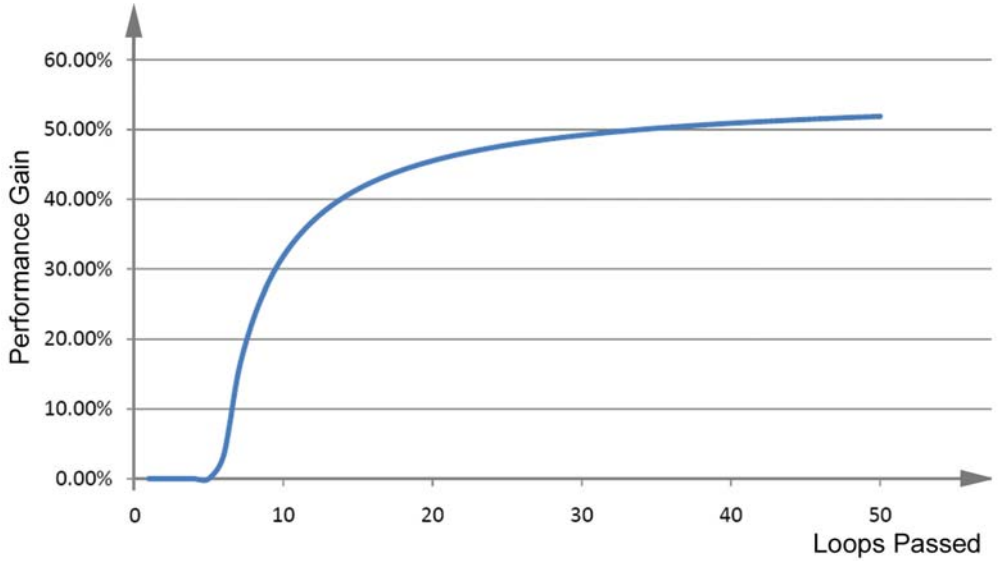
Die Anzahl aller Befehle, die in 50 Schleifendurchläufen ausgeführt werden:



Befehle	x86	EmuRix (ReAl)
E-A (Laden und Speichern)	40%	0%
ALU (Verarbeitung)	42 %	75%
PXCU (Verwaltung)	18%	25%

Es ergibt sich eine Entlastung des Prozessors um etwa 40 %, weil die Lade- und Speicherbefehle entfallen. Der Prozessor wird nicht schneller, sondern besser ausgenutzt. Dem Nutzer steht anteilig mehr Verarbeitungsleistung zur Verfügung.

Der Leistungsgewinn der ReAl-Maschine in Abhängigkeit von der Anzahl der Schleifendurchläufe:



- In der x86-Maschine muß anfänglich nur die Schleife initalisiert werden, dann läuft das Anwendungs-programm sofort los.
- In der ReAl-Maschine hingegen ist zunächst die gesamte Ressourcenkonfiguration aufzusetzen und mit anfänglichen Parametern zu versorgen (Setup).
- Deshalb ist hier erst nach dem 6. Schleifendurchlauf ein Leistungsgewinn zu erkennen (er beträgt dann 3,33 %).
- Nach 50 Durchläufen beträgt der Leistungsgewinn rund 52 %.
- Das heißt, Anwendungsprogramme, deren Laufzeit nicht von externen Einrichtungen abhängt, werden der Hälfte der Zeit ausgeführt; der Computer arbeitet aus Sicht des Anwenders doppelt so schnell.

Die Hardware ausnutzen und auch noch Strom sparen:

- Beide Absichten laufen auf das Gleiche hinaus.
- Die höchstmögliche Verarbeitungsleistung einer Schaltungsanordnung ist dann gegeben, wenn so viele Taktzyklen wie möglich zur Lösung der eigentlichen Verarbeitungsaufgabe beitragen.
- In einer idealen Maschine würden alle Taktzyklen auschließlich dazu dienen, die anwendungsseitig gewünschten Ergebnisse zu berechnen.
- Keine Taktzyklen und Speicherzugriffe würden dazu verwendet werden, Befehle zu holen, Zwischenwerte abzuspeichern und wieder zu laden, Funktionsaufrufe auszuführen usw.
- Wie weit eine Maschine dieser Idealvorstellung nahekommt, kann durch den Kennwert der Implementierungs-effizienz ausgedrückt werden.

Ausblick:

- Die Versuchsergebnisse lassen erwarten, daß die ReAl-Architektur auch dann leistungsmäßig überlegen sein wird, wenn nur Emulatoren miteinander verglichen werden. Ein naheliegendes Entwicklungsziel wäre eine virtuelle ReAl-Maschine, die die Programmiersprache Java interpretiert. Sie wäre mit den bekannten Lösungen JVM und Dalvik zu vergleichen.
- Des weiteren ist zu vermuten, daß x86-Maschinen (oder ähnliche CISC-Architekturen) die besseren Emulator-plattformen sind, weil deren Maschinenbefehle Verknüpfungsoperationen mit Speicheroperanden ausführen können. Auch wäre zu untersuchen, inwiefern Maschinen mit sehr großen Registersätzen (z. B. Itanium) als Emulatorplattformen geeignet sind. All das sind aber nur Übergangslösungen. Das Endziel sollte auf jeden Fall die echte ReAl-Maschine sein...
- Es dürfte nicht allzu schwierig sein, heutige Superskalarprozessoren in ReAl-Maschinen umzubauen und FPGAs mit hart verdrahteten Ressourcen auszurüsten Die Technologien jedenfalls sind vorhanden...

Anhang

Grundsatzlösungen bewerten

Wie vergleichen wir einen herkömmlichen Universalprozessor, eine FPGA-Lösung und eine ReAl-Maschine miteinander? Die üblichen “MIPS” sind offensichtlich bedeutungslos, das bloße Messen der Ausführungszeit ergibt keine tieferen Erkenntnisse. Probieren wir es deshalb nach folgenden Grundsätzen:

- Es ist immer ein Schaltwerk, das den Algorithmus ausführt, gleichgültig wie dessen Ablaufbeschreibung formuliert ist.
- Der Universalprozessor ist nichts weiter als ein universell nutzbares Schaltwerk.
- Für jeden Algorithmus kann man spezielle Schaltwerke bauen, die – im Rahmen beherrschbarer Aufwendungen – die Ergebnisse in einer minimalen Anzahl von Maschinenzyklen gewinnen.
- Befehle sind nur codierte, gespeicherte Beschreibungen für Zustandsübergänge. Es sind im Grunde nur Hilfsmittel, um das Schaltwerk universell zu gestalten.
- Einzweckschaltungen brauchen gar keine Befehle; ihre internen Zustandsübergänge ergeben sich allein aus ihrer Struktur und den aktuellen Daten.

Um ein allgemeines Maß für die Verarbeitungsleistung zu gewinnen, zählen wir keine Befehle, sondern die anwendungsseitigen Daten – und wir setzen sie ins Verhältnis zu allen Taktzyklen, die die Maschine benötigt, um die gewünschten Verarbeitungsergebnisse zu liefern. Die Grundfrage: Wieviele Bits werden in einem bestimmten Zeitabschnitt verarbeitet und erzeugt?

Hierzu betrachten wir einen Zeitabschnitt, der n Maschinenzyklen lang ist (Zykluszeit = t_C). Die Anzahl der anwendungsseitigen Datenbits, die in jedem Maschinenzyklus i ($1 \leq i \leq n$) über die Signalwege laufen, sei B_i. Damit ergibt sich ein Leistungsmaß PM wie folgt:

$$PM = \frac{1}{n \cdot t_C} \sum_{i=1}^{n} B_i$$

Die Maßeinheit: Nutzbits (effektive Bits) je Sekunde. Die meisten Verarbeitungsabläufe sind datenabhängig. Deshalb muß das Leistungsmaß als Mittel- oder Erwartungswert gebildet werden. Wichtig ist, nur jene Datenbits als B_i zu zählen, die wirklich zur Lösung des Anwendungsproblems beitragen, nicht aber Datenbits, die nur bewegt werden, um Eigentümlichkeiten der jeweiligen Architektur oder Systemlösung zu entsprechen. Es werden aber **alle** Maschinenzyklen gezählt, gleichgültig, was in ihnen jeweils abläuft (Befehlslesen, Retten von Registerinhalten, Kontextumschaltung usw.).

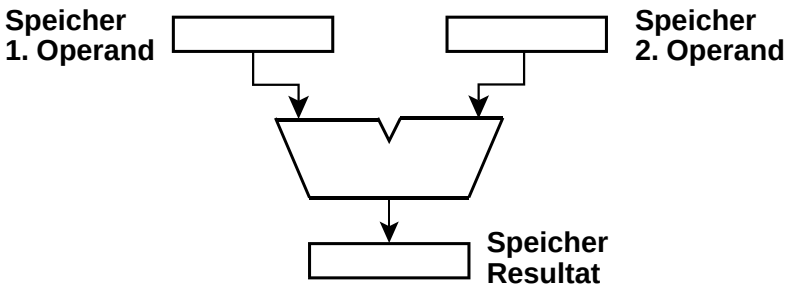
Die Implementierungseffizienz
Im Grunde sind es alles nur Schaltwerke, und sie sollen eigentlich nichts anderes tun als im Sinne der Anwendungsaufgabe etwas Vernünftiges zu leisten – und zwar am besten in jedem einzelnen Maschinenzyklus.

Daraus ergibt sich aber eine weitere Frage: Sind die anwendungsseitigen Algorithmen überhaupt so beschaffen, daß es möglich ist, in (nahezu) jedem Maschinenzyklus etwas Nützliches zur Erledigung der Anwendungsaufgabe zu tun oder gibt es auch schon von dieser Seite her einen unvermeidlichen Verwaltungsaufwand und Leerlauf (Overhead)?

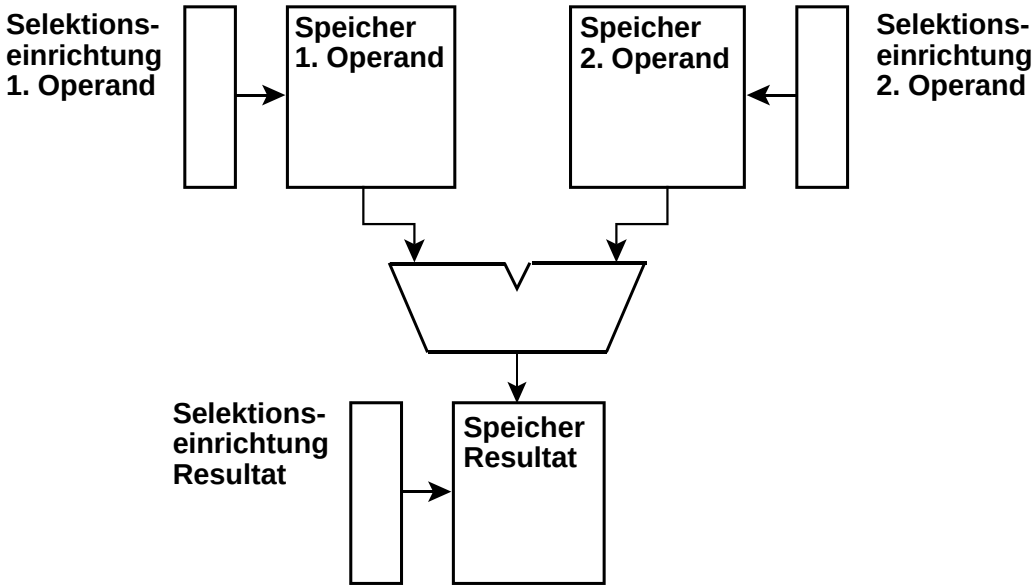
Ein Algorithmus ist ein Vorschrift, die besagt, wie aus Eingangsdaten (Operanden, Argumenten) Ausgangsdaten (Resultate) erzeugt werden. Die ideale Vergegenständlichung eines Algorithmus ist eine Anordnung, die aus den Eingangsdaten die Ausgangsdaten unmittelbar, sozusagen auf einen Schlag bildet – also durch funktionelle Zuordnung und nicht durch zeitliche Folgen aufeinander folgender Verarbeitungschritte.

Offensichtlich läßt sich so etwas nur dann bauen, wenn die Anzahl der Bits (und damit der Signalwege) nicht allzu hoch ist und wenn die Verknüpfungen nicht allzu kompliziert sind. Diese Beschränkungen können überwunden werden, indem man die Informationswandlungen in aufeinander folgenden Maschinenzyklen abschnittsweise ausführt.

Direkte Zuordnung:



Abschnittsweise Ausführung eines Algorithmus:



Eine solche Anordnung arbeitet taktgesteuert, wobei in jedem Takt aus den jeweils ausgewählten Operandenabschnitten die jeweiligen Ergebnisabschnitte gebildet werden. Die höchstmögliche Verarbeitungsleistung kann dann erreicht werden, wenn die folgenden Bedingungen erfüllt sind:

1. Um das Ergebnis zu bilden genügt es, auf jeden Operandenabschnitt genau einmal zuzugreifen.
2. In jedem Maschinenzyklus wird ein Beitrag zum Endergebnis geliefert.
3. Gelieferte Teilergebnisse werden nicht nochmals zurückgelesen oder abgeändert.
4. Die Anzahl der Ergebnisbits entspricht der Anzahl der jeweiligen Signalwege (als praktisch der Verarbeitungsbreite).

In einer derartigen Maschine werden zum Erzeugen der Ergebnisse nur so viele Maschinenzyklen benötigt, wie notwendig sind, um die Operanden- und Resultatabschnitte über die gegebenen Signalwege zu transportieren. Mit anderen Worten, jedes Datenbit muß nicht öfter als genau einmal durch die Maschine bewegt werden, und in jedem Maschinenzyklus wird ein Beitrag zum Endergebnis geliefert, der der jeweiligen Verarbeitungsbreite entspricht. Es ergibt sich somit in Maximum an Verarbeitungsleistung, das mit den jeweils vorgesehenen Aufwendungen überhaupt zu realisieren möglich ist.

Ob so etwas grundsätzlich gelingen kann, ist letzten Endes eine Eigenschaft des Algorithmus. Diese Eigenschaft läßt sich durch eine Zahlenangabe ausdrücken, die als Implementierungseffizienz e_i bezeichnet wird.

$$e_i = \frac{\sum_{i=1}^{n} \mathrm{CARDB}(A_i) + \sum_{j=1}^{m} \mathrm{CARDB}(R_j)}{z \cdot (\mathrm{ARG\_LINES} + \mathrm{RES\_LINES})}$$

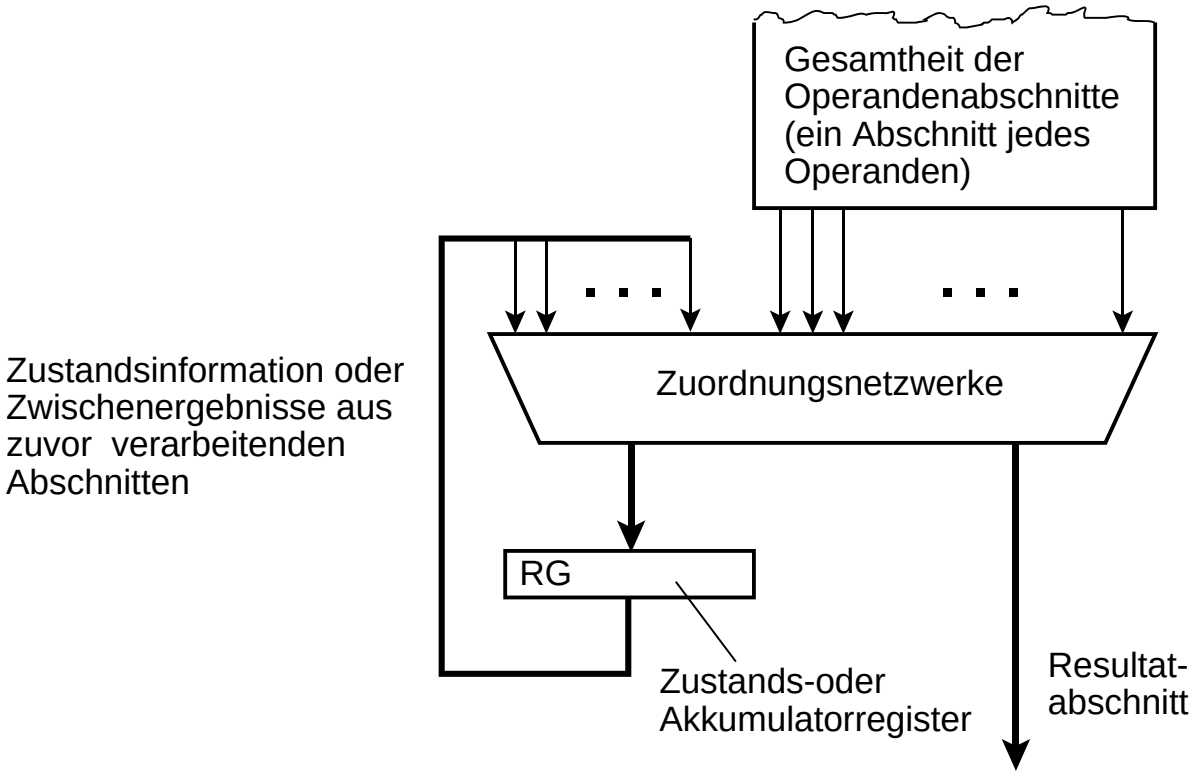
Es bedeuten:

- CARDB (A_i), CARDB (R_j) die Anzahl der Bitpositionen, in denen die Operanden und Resultate dargestellt werden.
- ARG_LINES, RES_LINES die Anzahl der Signalwege, die zum Transportieren der Operanden-und Resultatabschnitte zur Verfügung stehen (Verarbeitungsbreite).
- z die Anzahl der Maschinenzyklen, die benötigt werden, um alle Resultate zu bilden ($z \geq 1$).

Die Implementierungseffizienz e_i ist eine dimensionslose Zahl im Intervall $0 < e_i \leq 1$. e_i ist gleich 1, wenn die beschriebene zweckmäßigste Implementierung tatsächlich möglich ist, wenn es also gelingt, eine Einrichtung zu bauen, die in jedem Maschinenzyklus gemäß ihrer Verarbeitungsbreite zum Endergebnis beiträgt. Eine Implementierungseffizienz von Eins kann nur dann erreicht werden, wenn jeder Abschnitt des Resultats durch kombinatorische Zuordnung aus folgenden Angaben gebildet werden kann:

- aus den jeweils aktuellen Abschnitten der Operanden,
- bedarfsweise aus Zustandsangaben oder Zwischenwerten, die eindeutig aus den zuvor verarbeiteten Abschnitten bestimmt worden sind.

Allgemeines Schema der Ausführung eines Algorithmus mit $e_i = 1$



Im Idealfall wird jeder Operandenabschnitt nur einmal geholt und jeder Resultatabschnitt nur einmal erzeugt. Alle Nutzbits der Anwendungsaufgabe laufen nur einmal durch die Maschine. Dann gibt es keine unnützen Schaltvorgänge, die Zeit kosten und zur Stromaufnahme beitragen. Wenn eine solche Auslegung gelingt, hat man beides auf einmal: maximale Verarbeitungsleistung, also kürzeste Verarbeitungszeit, und geringste Verlustleistung.

Die Minimierung der Stromaufnahme durch Vermeiden unnötiger Schaltvorgänge ergibt sich bei voller Auslastung. Es ist kein Stromsparen, das auf dem Stillsetzen von Schaltungsteilen beruht, die gerade nichts zu tun haben.

Die Implementierungseffizienz ist kleiner als Eins, wenn

- die abschnittsweise Zuordnung technisch nicht verwirklicht werden kann (Aufwand, Kompliziertheit), so daß Folgen mehrerer Maschinenzyklen nötig sind, um jeweils einen Resultatabschnitt zu bilden,
- Resultatbits von Operandenbits abhängen, die sich in verschiedenen Abschnitten befinden können, so daß Zugriffe auf mehrere Operandenabschnitte notwendig sind, um diese Resultatbits zu bestimmen,
- Teile bereits gebildeter Resultatabschnitte geändert werden müssen (was erfordert, erneut auf diese Abschnitte zuzugreifen).

Aufwandung und Leistungen vergleichen
Um alternative Entwurfslösungen zu bewerten, genügt es oftmals nicht, nur die Kosten zu vergleichen. Vielmehr sind Kosten-Nutzen-Betrachtungen anzustellen. Im folgenden sollen einige pauschale Kennwerte genannt werden.

Die Hardware-Effizienz
Die Hardware-Effizienz HE ergibt sich als Verhältnis von Leistung und Aufwand.

$$HE = \frac{\text{Leistungsangabe}}{\text{Aufwandsangabe}}$$

Die Leistungsangabe kann beispielsweise die Ausführungszeit sein oder ein Leistungsmaß in Nutzbits/s, die Aufwandsangabe ist je nach Problemstellung zu wählen (Kosten, Strombedarf, Fertigungsstunden usw.).

Die Mehraufwandseffizienz
Diese Maß ermöglicht es, verschiedene Varianten hinsichtlich ihrer Zweckmäßigkeit untereinander zu vergleichen. Es liegt beispielsweise ein erster Entwurf vor (Variante 1). Daraus wird etwas Leistungsfähigeres entwickelt (Variante 2). Die Praxis zeigt, daß beides vorkommen kann:

- Mit vergleichsweise geringen Zusatzaufwendungen ergibt sich ein überproportional höheres Leistungsvermögen.
- Eine extreme Aufwandserhöhung bewirkt nur eine vergleichsweise geringe Leistungssteigerung.

Zur Bewertung eignet sich die Mehraufwandseffizienz $ME_{1\text{-}2}$ des Übergangs von Variante 1 zu Variante 2:

$$ME_{1-2} = \frac{HE_2}{HE_1}$$

Hierin sind HE_1, HE_2 die Werte der Hardware-Effizienz beider Varianten.

Aufwandsverhältnis und Leistungsverhältnis (Speedup)
Es ist manchmal zweckmäßig, die Verhältnisse von Aufwand und Leistung einzeln zu betrachten. Hierzu kann man ein Aufwandsverhältnis RA_{1-2} und ein Leistungsverhältnis RP_{1-2} (Geschwindigkeitszuwachs, Speedup S_{1-2}) bestimmen:

$$RA_{1-2} = \frac{\text{Aufwand 2}}{\text{Aufwand 1}} \qquad RP_{1-2} = S_{1-2} = \frac{\text{Leistung 2}}{\text{Leistung 1}}$$

Damit ergibt sich die Mehraufwandseffizienz zu

$$ME_{1-2} = \frac{RP_{1-2}}{RA_{1-2}}$$

Der Wirkungsgrad
Der Wirkungsgrad soll das Leistungsvermögen einer bestimmten Implementierung (in Hard- oder Software) in Bezug auf eine leistungsmäßig ideale Implementierung oder auf die absoluten Leistungsgrenzen kennzeichnen.

$$\eta = \frac{P_{var}}{P_{ideal}}$$

- η: Wirkungsgrad ($0 < \eta \leq 1$),
- P_{var}: Leistungsangabe der jeweils zu untersuchenden Variante,
- P_{ideal}: Leistungsangabe der idealen Implementierung.

Die (näherungsweise) ideale Implementierung ist eine kompromißlos auf Leistung ausgelegte Spezialmaschine, die gemäß dem Stand der Technik im Rahmen plausibler Aufwandsgrenzen ausführbar sein muß. Die Leistungsangaben kann man anhand von Probeentwürfen oder -programmen abschätzen. Um zu idealen Maschinen zu kommen, bieten sich folgende Möglichkeiten an:

- Probeentwürfe auf dem RTL-Ebene, wobei man sich oft auf das Verarbeitungswerk (Datenwege + Register + Verknüpfungsschaltungen) beschränken kann.
- Das Anwendungsproblem wird in einer Hardwarebeschreibungssprache formuliert. Die Schaltungssynthese ergibt die Größenordnung des Aufwands.
- Bekannte Einrichtungen (vor allem in den oberen Leistungsklassen) werden als Referenzlösungen herangezogen.

Ein Blick zurück in die Entwicklungsgeschichte
Der erste Schritt: Ressourcenalgebra als formales Beschreibungsmittel, aber feste Ressourcenauswahl. Heraus kommen nach wie vor herkömmliche Universalrechner, die aber nicht so recht zu den Marketing-Schlagworten (RISC, CISC, VLIW usw.) passen wollen.

- Eine Maschine, die ihre Befehlswirkungen erbringt, indem vergleichsweise einfache Verarbeitungsschaltungen in vielen Schritten mehrfach durchlaufen werden, leistet weniger als eine Maschine, die mit besonderen Verarbeitungsschaltungen für die einzelnen Befehlswirkungen ausgerüstet ist.
- Sinngemäß wirken sich die Breite der Signalwege und die Komplexität der Steuerschaltungen auf das Leistungsvermögen aus. Ganz allgemein ausgedrückt sind es letzten Endes die Ressourcen, die die Verarbeitungsleistung bestimmen.
- Ressourcen der Befehslausführung und Befehlsablaufsteuerung sind im Grunde RTL-Strukturen. Die Speichermittel an den Ein- und Ausgängen dieser Strukturen fassen wir zum sog. **Ressourcenvektor** zusammen.
- Maschinenbefehle sind nur Mittel zum Zweck. Sie dienen dazu, die Nutzung der Ressourcen anzuweisen.
- Entscheidend ist es, die Ressourcen soweit überhaupt möglich mit nützlicher (= der Lösung des Anwendungs-problems dienender) Arbeit zu beschäftigen.
- Das läuft darauf hinaus, den Ressourcenvektor zu beeinflussen (insgesamt oder abschnittsweise).
- Es konnte gezeigt werden, daß sich die herkömmlichen Befehlsmodelle (Formate, Prinzipien der Ablaufsteuerung usw.) in diesem Rahmen darstellen lassen.
- Der erste Ansatz: welche Ressourcen als Hardware gebaut werden und wieviele davon, ist Sache der Optimierung. Es ließe sich sogar mittels linearer Optimierung erledigen, wenn die Anforderungen genau bekannt und quantitativ beschrieben wären.
- Die Maschinenbefehle werden so gestaltet, daß der jeweilige Ressourcenvektor so zweckmäßig wie möglich kommandiert werden kann.
- Die Befehlsformate ähneln jenen der herkömmlichen Mikrobefehle, aber ohne deren häßliche Einschränkungen.
- Sie sind aber nicht maschinenunabhängig; es wäre also keine richtige Architektur.
- Deshalb führen wir eine maschinenunabhängige, abstrakte Programmschnittstelle ein.
- Dieses abstrakte Modell nennen wir Ressourcen-Algebra. Sie wird in einer besonderen formalen Sprache beschrieben und codiert. Das ist das Ziel der Compiler. Das eigentliche Maschinenprogramm ergibt sich daraus durch vergleichsweise einfache Umcodierungsabläufe.

Der nächste Schritt
Das Grundkonzept der Ressourcenalgebra ist nicht schwierig. Auch ein entsprechender mathematischer Formalismus ist zu bewältigen, zumal es dazu Vorarbeiten gibt (z. B. [10]). Wie aber soll eine praktisch brauchbare Programmschnittstelle und Beschreibungssprache aussehen?

Während dieser Untersuchungen hat sich der Ansatz ergeben, der jetzt als ReAl Computer Architecture bezeichnet wird:

- Weshalb sich auf einen festen Ressourcenvorrat beschränken?
- Probieren wir es doch einmal damit, einen unbeschränkten Ressourcenvorrat anzunehmen und daraus so viele Ressourcen auszuwählen, wie wir benötigen, um das jeweilige Anwendungsproblem zu lösen...
- Es sieht auf den ersten Blick umständlich aus, ist es aber nicht wirklich. Man muß nur die Nerven behalten... In der Anfangszeit wurde viel Mühe darauf verwendet, den radikalen Ansatz zu widerlegen oder abzuwandeln

Herkömmliche Prozessoren weiterentwickeln?
Es versteht sich von selbst, daß auch auf diesem Gebiet weitere Verbesserungen möglich sind. ReAl -Maschinen können einschlägigen Empfehlungen ohne weiteres gerecht werden. Es sind sogar weitergehende Effekte zu erwarten.

Empfehlungen der Fa. Atmel ([11]), um den Durchsatz in herkömmlichen Prozessoren zu erhöhen:

- Den Anteil der Lade- und Speicherzyklen verringern (mehr als 30% der Befehle, die in einer RISC-Maschine ausgeführt werden, sind Lade- und Speicherbefehle).
- Zeitkritische Operationen, die eine Vielzahl von Daten betreffen, gleichzeitig ausführen.
- Die Nutzung der Pipeline-Ressourcen maximieren.
- Die Latenzzeiten der Verzweigungen minimieren.

Mit ReAl-Maschinen kann diesen Empfehlungen ohne weiteres entsprochen werden:

- Wenn die Ressourcen gemäß dem Datenfluß-schema konfiguriert werden, gibt es – mit Ausnahme des Holens der Anwendungsdaten und des Ablieferns der Ergebnisse – gar keine Lade-und Speicherzyklen (und auch keine Befehlslesezyklen, da die Steueranweisungen in den Ressourcen abgelegt sind).
- Kein Problem, wenn genügend Ressourcen vorhanden sind.
- Da die Verarbeitungsressourcen nicht in einer starren Hardware-Pipeline angeordnet sind, sondern frei konfiguriert werden können, gibt es die mit der Ausnutzung von Pipelines verbundenen Probleme gar nicht.
- Läßt sich durch entsprechende Auslegung der Plattformressoucen erreichen. Hierbei können auch Verzweigungen in mehr als zwei Richtungen unterstützt werden. Manche Probleme der Programmablaufsteuerung können grundsätzlich anders gelöst werden (bedingte Ausführung, Datenflußsteuerung usw.).

Ein Praxisbeispiel

Ein Algorithmus zum Berechnen der Summe absoluter Differenzen (SAD). Er wurde in der Literatur ([11]) als Beispiel verwendet, um Optimierungsprobleme herkömmlicher Prozessorarchitekturen zu veranschaulichen.

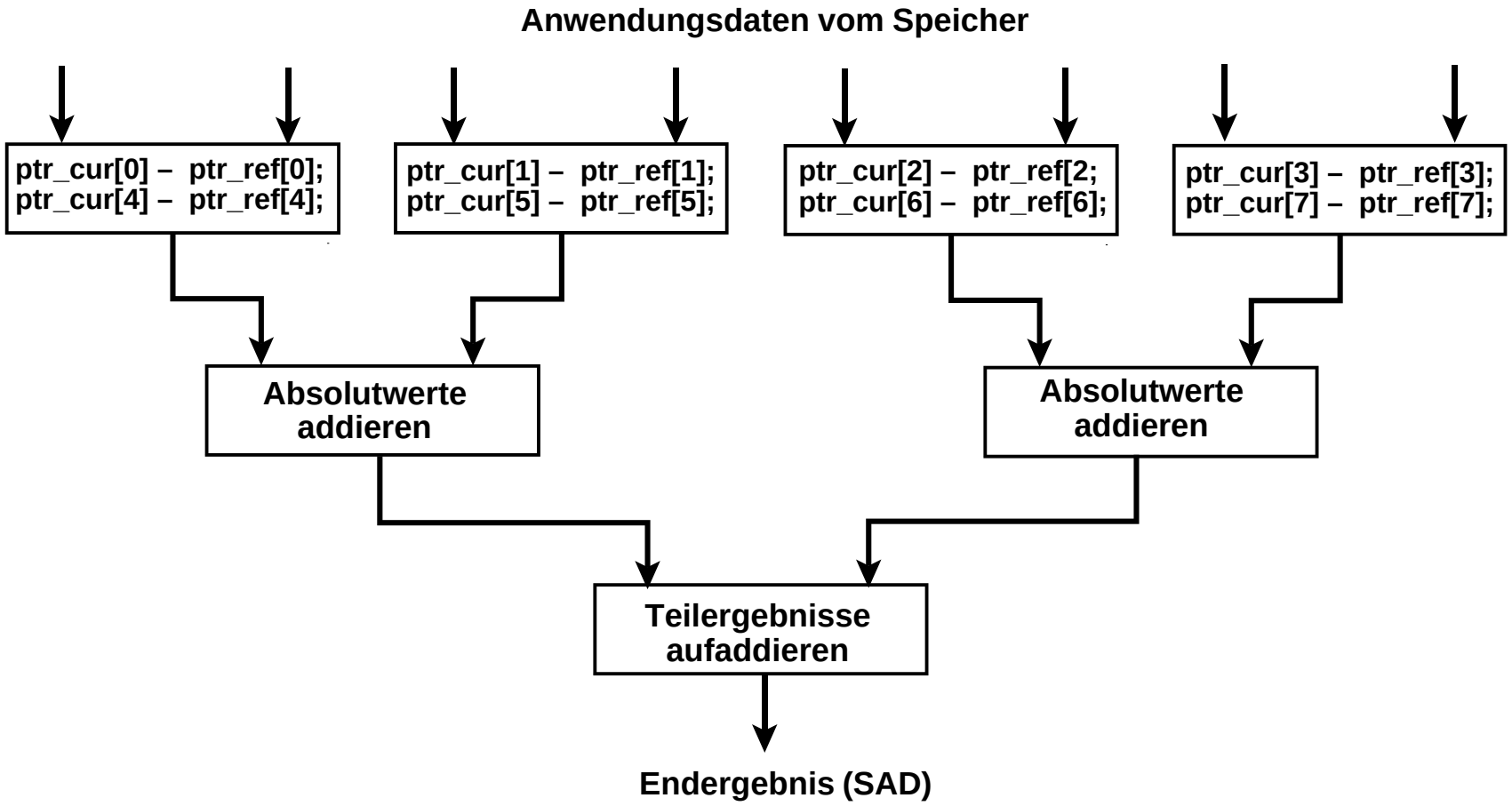
```
uint32_t
sad8_c(const uint8_t * const cur,
const uint8_t * const ref,
const uint32_t stride)
{
uint32_t sad = 0;
uint32_t j;
uint8_t const * ptr_cur = cur;
uint8_t const * ptr_ref = ref;

for (j = 0; j < 8; j++)
   {
   sad += abs (ptr_cur[0] --
ptr_ref[0];
   sad += abs (ptr_cur[1] --
   ptr_ref[1];
   sad += abs (ptr_cur[2] --
   ptr_ref[2];
   sad += abs (ptr_cur[3] --
   ptr_ref[3];
   sad += abs (ptr_cur[4] --
   ptr_ref[4];
   sad += abs (ptr_cur[5] --
   ptr_ref[5];
   sad += abs (ptr_cur[6] --
   ptr_ref[6];
   sad += abs (ptr_cur[7] --
   ptr_ref[7];
   ptr_cur += stride;
   ptr_ref += stride;
   }
return sad;
}
```

Quelle: Xvid Codec.

Der SAD-Wert kann offensichtlich in einer Programmschleife berechnet werden. Um die Geschwindigkeit zu erhöhen, wurde die innerste Schleife aufgerollt. Es sind 16 Operanden zu holen und 32 Operationen auszuführen. Jedes Laden und jede Operation erfordert wenigstens einen Maschinenbefehl, der auch noch geholt werden muß.

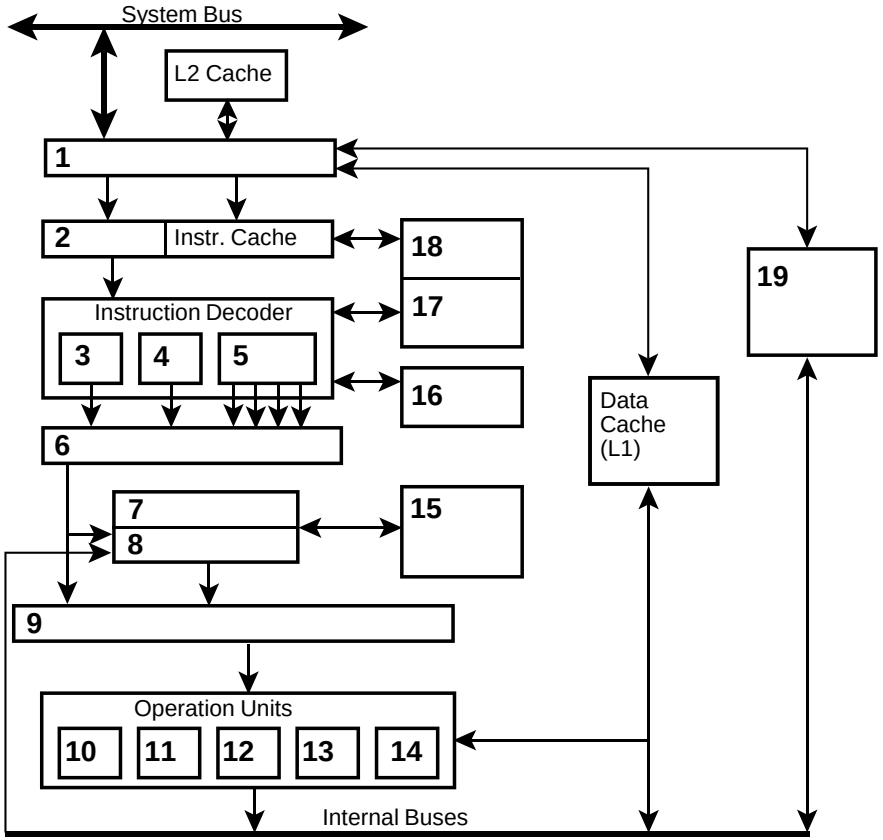
Die Alternative:
Die Verarbeitungsaufgabe wird mit einem invertierten Binärbaum aus Verarbeitungsressourcen erledigt, der durch – nicht dargestellte – Ressourcen zur Datenadressierung ergänzt wird. Ist diese Ressourcenanordnung erst einmal konfiguriert worden, ist es nicht mehr erforderlich, Maschinenbefehle zu lesen. Alle Speicherzugriffswege und damit die gesamte Speicherbandbreite stehen dann zur Verfügung, um die eigentlichen Anwendungsdaten zu transportieren.



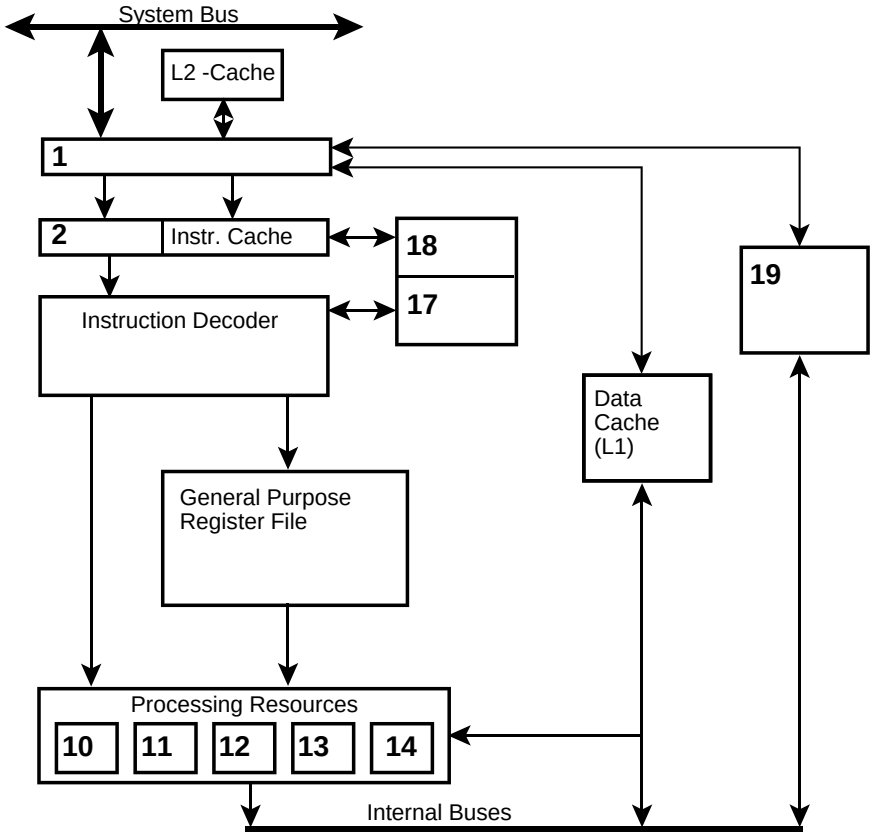
Prozessoren umbauen

Wir beziehen uns auf einen typischen Superskalarprozessor, der rund 50 Millionen Transistoren erfordert. Beim Umbau in eine ReAl-Maschine entfallen die komplizierten Steuereinrichtungen. Es lassen sich wenigstens 16 universelle Verarbeitungs-ressourcen unterbringen. Der Befehlsdecoder ist viel einfacher. Der Universalregistersatz kann beträchtlich erweitert werden (z. B. auf 64 bis 1024 Register). Die Verarbeitungsressourcen können mit dem Universalregistersatz direkt gekoppelt werden. Caches, Schnittstellensteuerungen usw. werden beibehalten. Ein Schaltkreis mit 200 Millionen Transistoren könnte also anstelle von 4 Prozessorkernen wenigstens 64 universelle Rechenwerke enthalten, die gemäß den ReAl-Wirkprinzipien als Ressourcen verwaltet werden und somit jedem einzelnen Programm zugute kommen können.

Ein Superskalarprozessor (nach Intel):

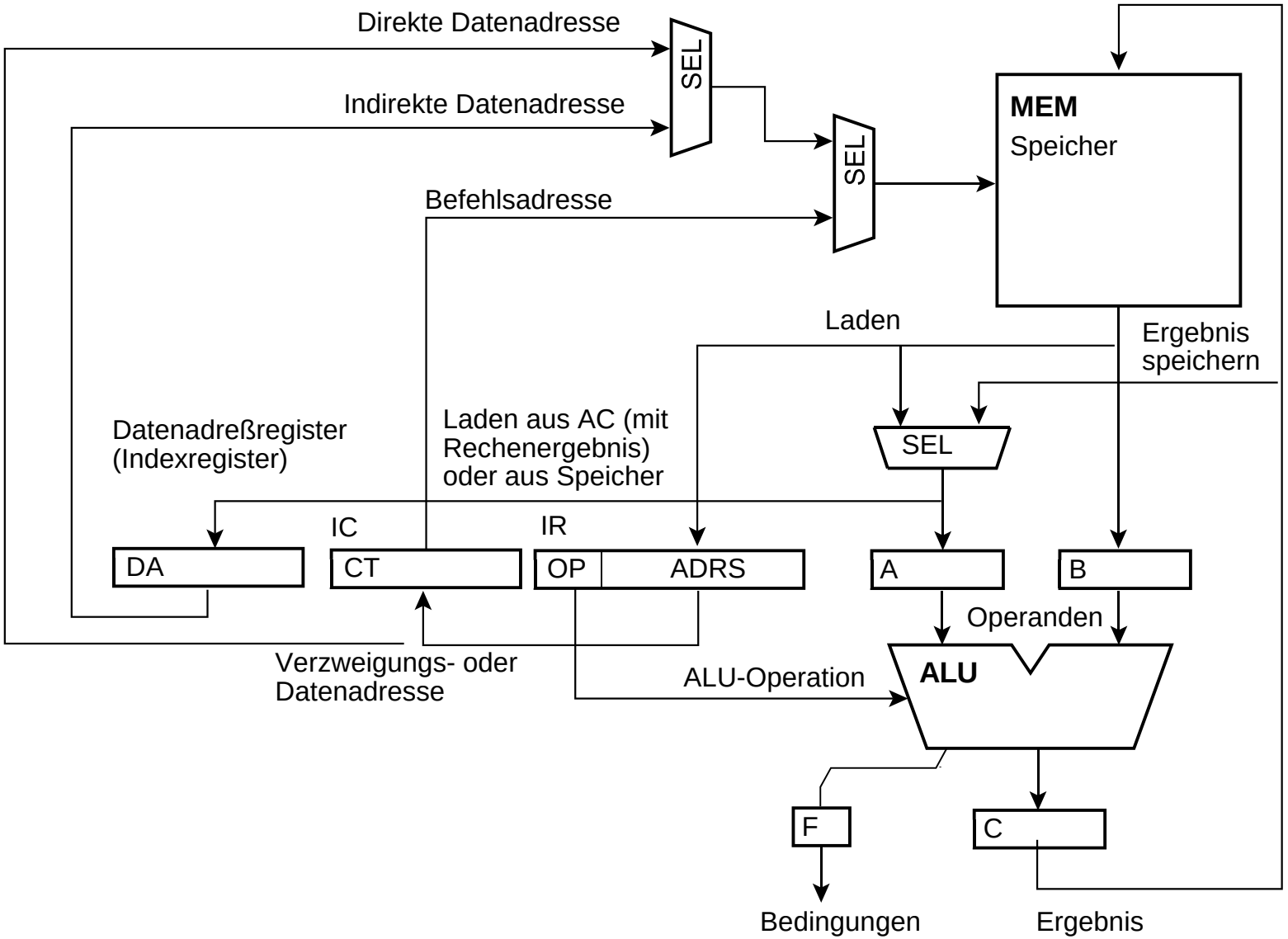


Der Umbau in eine ReAl-Maschine:

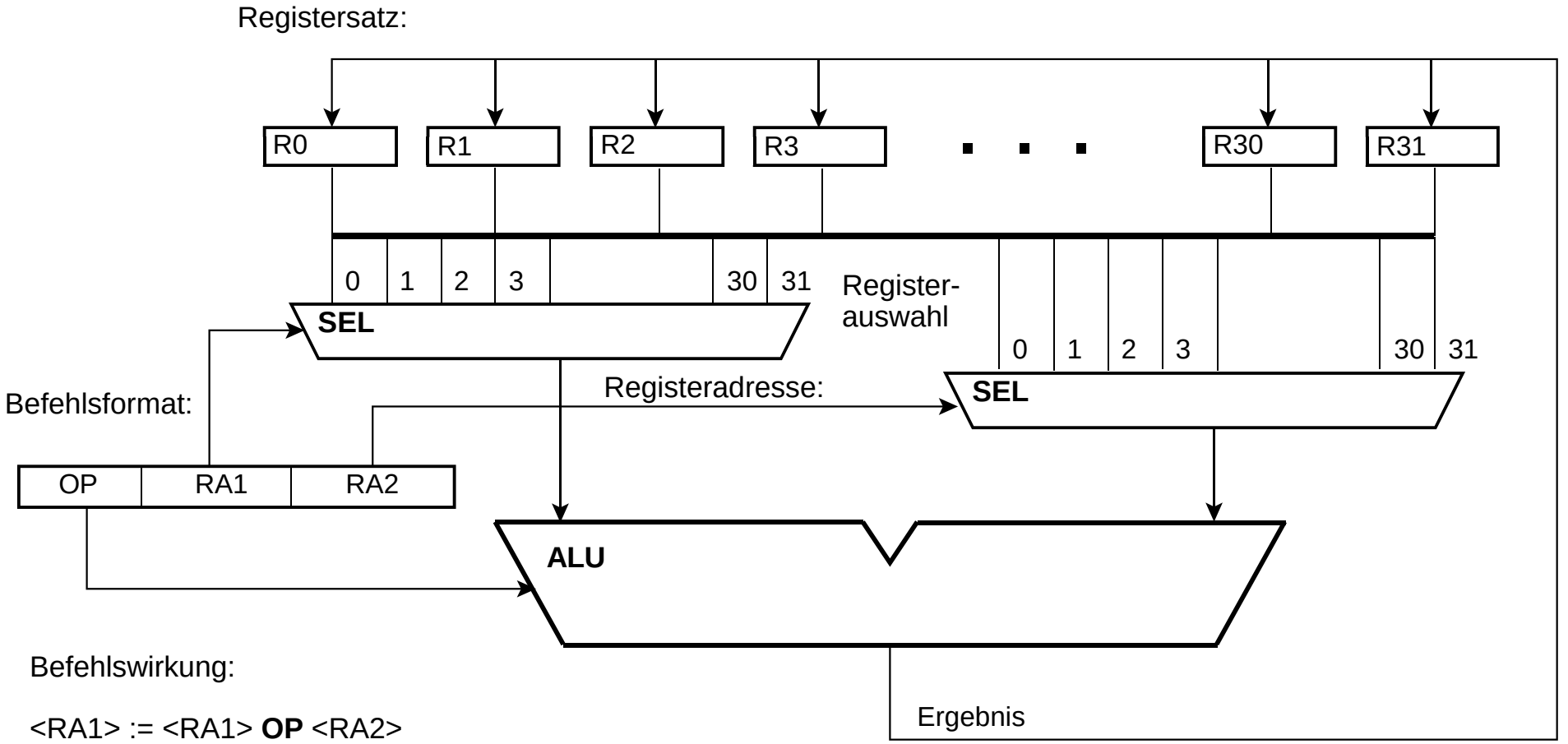


1 - Busanschlußsteuerung; 2 - Befehlsleseeinheit; 3, 4 - Befehlsdecodierung für einfache Befehle; 5 - Befehlsdecodierung für komplexe Befehle; 6 -Registerzuordnungseinheit; 7 - Befehlserledigung; 8 - Mikrobefehls-Assoziativpuffer (Reordering Buffer); 9 -Mikrobefehlsabruf; 10, 11 -Gleitkomma-verarbeitungswerke;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

Vom Universalrechner zur ReAl-Maschine
Beginnen wir mit einer der einfachsten praxisbrauchbaren Architekturen – der Akkumulatormaschine mit Indexregister. (vgl. beispielsweise Motorola MC6800).

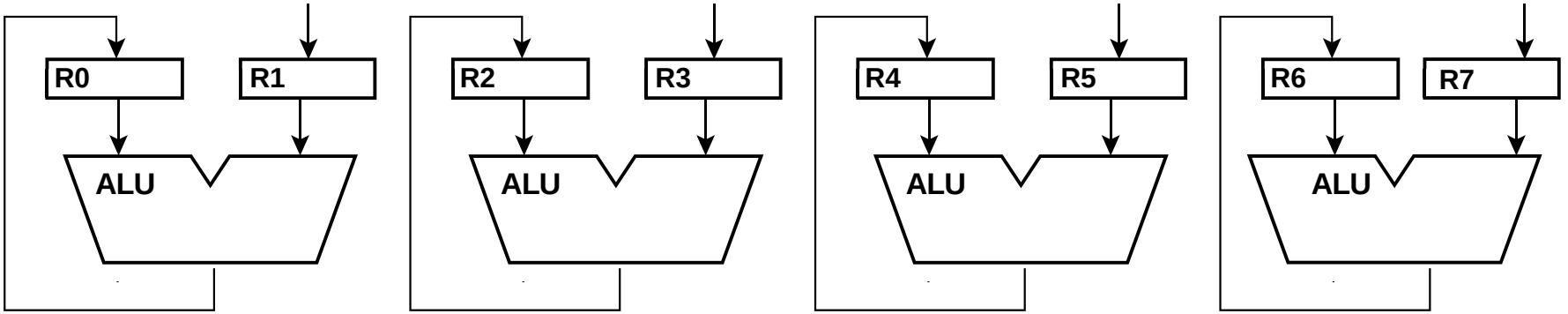


Die Universalregistermaschine
Wäre es nicht besser, mehrere Register zu haben, die wir freizügig als Akkuumulatoren oder als Adreß- bzw. Indexregister nutzen können? Beispiele gibt es genügend, vom S/360 über die PDP10 bis zu Atmel AVR usw.



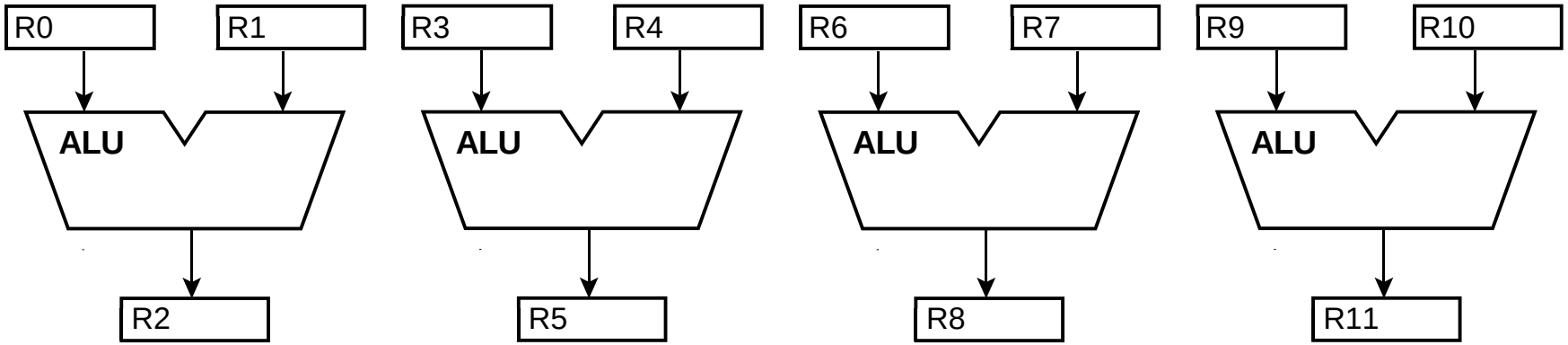
Die erste ReAl-Maschine

So weit ist der Schritt eigentlich gar nicht... Wir haben schließlich genügend Transistoren. Geben wir also jedem Registerpaar eine eigene ALU. Eines der Register ist jeweils der Akkumulator. Die Registerauswahl wird eingespart. Das vermindert die Durchlaufverzögerung. Alle ALUS können gleichzeitig arbeiten. Somit kann der innewohnende (inhärente) Parallelismus tatsächlich ausgenutzt werden.

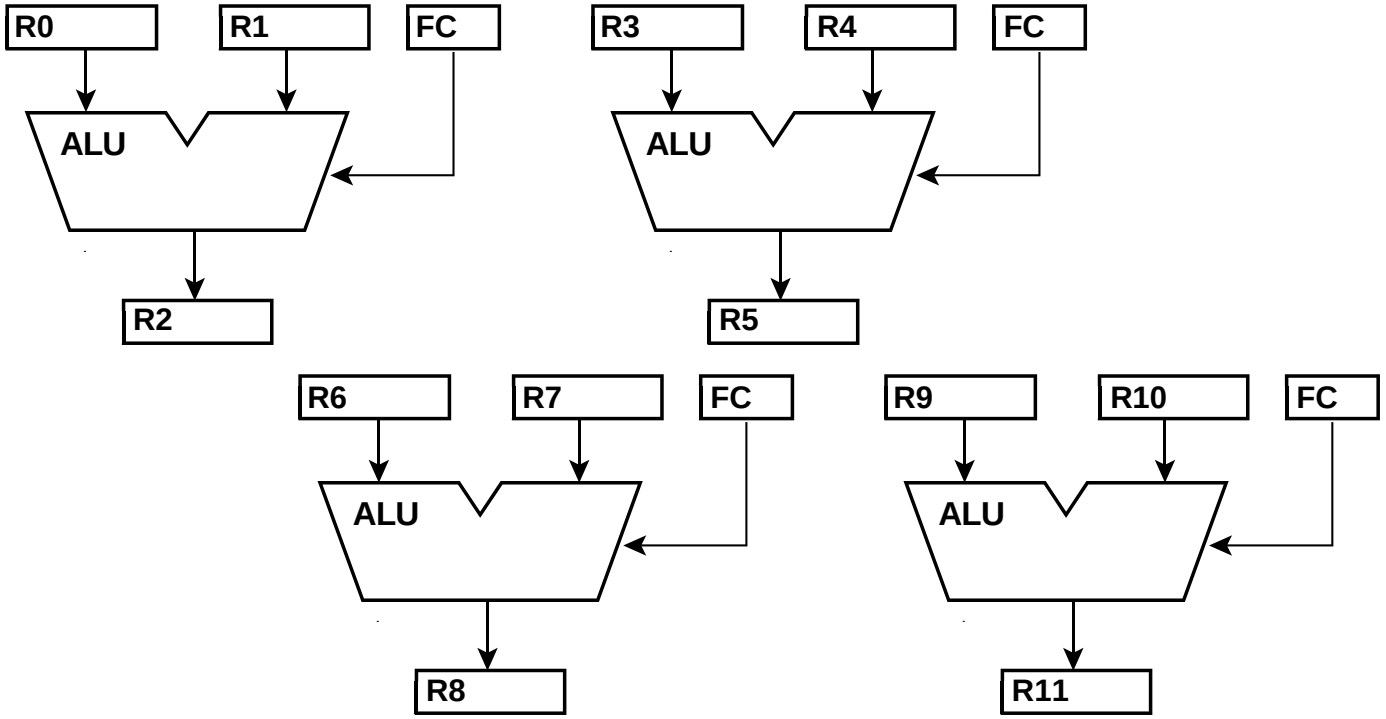


Beim Akkumulatorprinzip wird einer der Registerinhalte (= eine Variable) immer überschrieben. Was aber, wenn der bisherige Wert dieser Variable woanders noch gebraucht wird?

Also machen wir besser alle Verknüpfungen zu funktionellen Zuordnungen und halten Operanden und Ergebnisse voneinander getrennt. Es sind mehr Register, aber die können wir uns leisten. Wir müssen sie ja nicht adressieren (im Operationsbefehl) und auswählen (in der Schaltung).



Um n ALUs parallel zu nutzen, müssen n Operationscodes gleichzeitig anliegen. Somit braucht man entsprechend breite Befehle (VLIW-Prinzip). In Programmschleifen sind sie immer wieder neu zu holen. Die weitaus meisten Programmabläufe sind Schleifen. Also sagen wir besser jedem Rechenwerk von vornherein, was es zu tun hat. Jedes dieser Werke ist eine Ressource.



Wenn wir genau wissen, was im Programmablauf zu tun ist, können wir auch für jeden Verarbeitungsvorgang eine spezielles Werk bereitstellen. Die Frage, ob spezielle Werke ohne oder universelle Werke (ALUs) mit Funktionsauswahl ist eine Frage der Optimierung. Ebenso, ob die Funktion während des Verarbeitungsabaufs umgeschaltet werden soll oder nicht. Die Maschinenprogrammfertigung beginnt jedenfalls damit, für jeden Programmvorgang eine eigene Ressource vorzusehen, ganz so, als ob wir genug von allem hätten. Optimieren kann man später.

Literaturverzeichnis

a) Zur ReAl Computer Architecture:

[1] Matthes, W.: Hardware Resources: a generalizing view on computer architectures. ACM SIGARCH Computer Architecture News, Vol. 18 , Issue 2 (June 1990), pages 7-14.

[2] Matthes, W.: How many operation units are adequate? ACM SIGARCH Computer Architecture News, Vol. 19, Issue 4 (June 1991), pages 94-108.

[3] Patentanmeldungen: DE 10 2005 021 749.4 und US 11/430,824.

[4] Matthes, W.: The ReAl Computer Architecture. Proceedings IDAACS 2007, pages 249-254.

[5] Matthes, W.: Ressourcen statt Prozessorkerne? NTZ 7/8 2009, S. 12 – 16.

[6] Matthes, W.: Resources instead of Cores? ACM Sigarch Computer Architecture News, Volume 38, Number 2, May 2010, pages 49 – 63.

[7] Kuczkowicz, L.: Verfahren zur Emulation von Hochleistungsrechnern. Bachelor Thesis, Fachhochschule Dortmund, 2011.

[8] Matthes, W.: Hardware und Software. Embedded Electronics, Band 3. Elektor, 2011.

[9] Matthes, W.: Dokumentation, Aktuelles usw.:

- http://www.realcomputerarchitecture.com
- http://www.controllersandpcs.de
- http://www.controllerandpcnews.de

b) Allgemeines:

[10] Mueller-Wichards, D.: An Algebraic Approach to Performance Analysis. In: Parallel Computing in Science and Engineering. Springer, 1988 (LNCS 295), pages 159-185.

[11] J. Uthus, Oy. Strom, "MCU Architectures for Compute-Intensive Embedded Applications," Atmel White Paper. Atmel Corporation, 2006. http://www.atmel.com.

[12] Grand Challenges der Technischen Informatik. Report. Gesellschaft für Informatik e. V. / Informationstechnische Gesellschaft des VDE (2008).

[13] Xilinx Corporation. Virtex series FPGAs. http://www.xilinx.com.

[14] Intel Corporation and AMD Corporation. Informationsmaterial und Handbücher zu Hochleistungsprozessoren. http:www.intel.com, http://www.amd.com.