TECHNISCHE UNIVERSITÄT DRESDEN

VLSI-EDA

Fakultät Informatik, Institut für Technische Informatik, Professur VLSI-Entwurfssysteme, Diagnostik und Architektur

Rechnerstrukturen und -organisation (RSO)

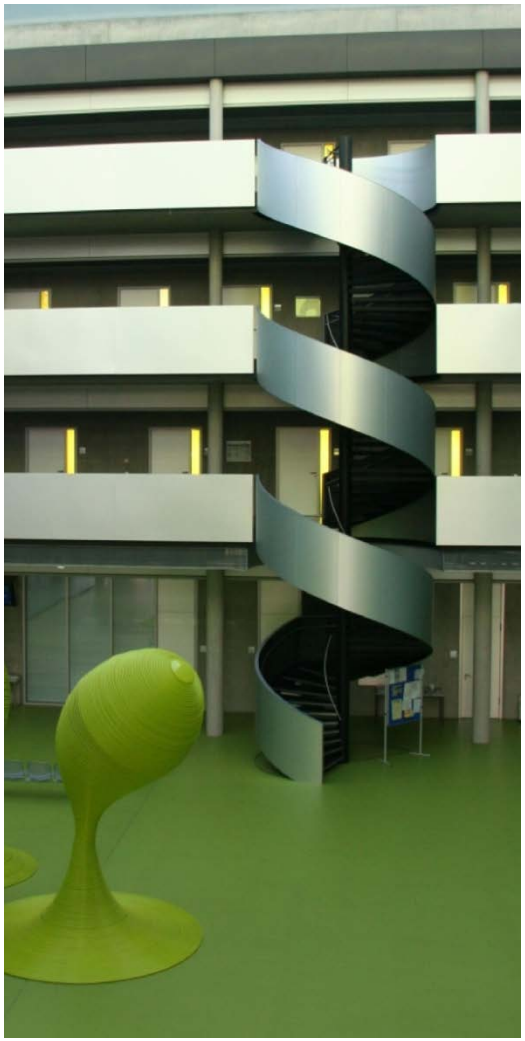
Befehlssatz

Rainer G. Spallek

TU Dresden, 13.01.2021

Rainer.Spallek@tu-dresden.de http://vlsi-eda.inf.tu-dresden.de

Gliederung



1 Zielstellung

- Erlangung eines Grundverständnisses für den Befehlssatz.
- Der Befehlssatz als Bindeglied zwischen Hardware und Software.
- Klassifizierung von Befehlssatzarchitekturen.
- Verschiedene Möglichkeiten der Befehlsstruktur.
- Unterteilung von Befehlssatzarchitekturen nach der Operandenspeicherung im Prozessor.
- Wie erfolgt die Speicheradressierung und -unterteilung.
- Kennenlernen verschiedener Adressierungsmethoden.
- Unterteilung des Operationsumfanges.
- Verständnis für die Befehlssätze verschiedener RISC / CISC Architekturen.

2 Begriffserklärung, Hauptmerkmale

Algorithmus: Die Informationsverarbeitung in einem Rechner erfordert die schrittweise Umsetzung eines definierten Algorithmus. Der Algorithmus wird als geordnete Folge von Anweisungen (Instruktionen), Befehlen dargestellt.

Befehl: Ein Befehl (Instruction) ist eine eindeutig spezifizierte Arbeitsanweisung an den Prozessor (CPU). Er ordnet eine Operation an, die in der Regel an spezifizierten Daten (Operanden) vorzunehmen ist und ein Ergebnis (Resultat) liefert. (maschinenlesbar → Maschinenbefehl)
Opcode + Operanden (Daten) oder
Opcode + Adressen (Zeiger auf Operanden)

Befehlssatz: Der Menge aller in einem Prozessor implementierten Befehle bildet den Befehlssatz. Die Architektur eines Rechners wird wesentlich durch den Befehlssatz des verwendeten Prozessors bestimmt,
ISC-Instruction Set Computer → wesentliches Architekturmerkmal.

Maschinensprache: Die Menge aller im Prozessor realisierten Maschinenbefehle definiert die Maschinensprache → Hardware-Programmiersprache. Programme in Maschinensprache → Maschinenprogramme.

Rechner einer Prozessorfamilie realisieren eine weitestgehend ähnliche Maschinensprache (Binärkompatibilität).

Befehlsvorrat: Die Menge aller verfügbaren Maschinenbefehle bildet den Befehlsvorrat. Sie werden in der Befehlsliste geordnet zusammengefasst.

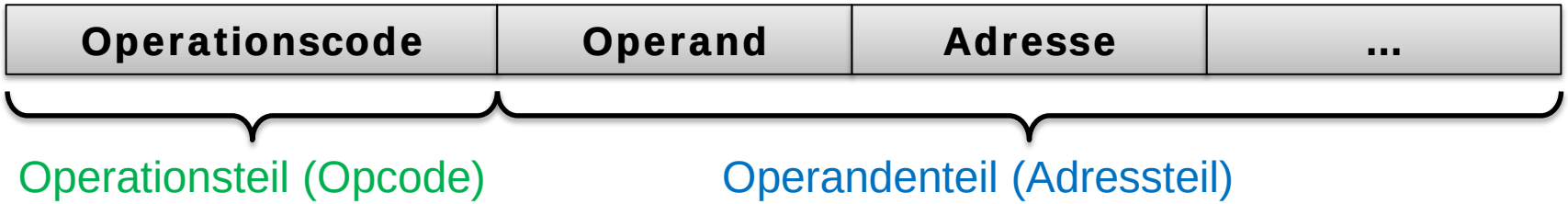
Befehlsformat: Die innere Struktur der Maschinenbefehle, dargestellt durch Binärworte, wird durch das Befehlsformat bestimmt. Entsprechend dem Befehlsformat werden die einzelnen Komponenten des Befehls binär codiert im Befehlswort zusammengefasst.

Befehlskomponenten: Die Komponenten eines Befehlssatzes sind:

die Operation, der Datentyp, die Operanden, die Adressden, die Adressierung.

Befehlsstruktur

Die Befehlskomponenten (Operation, Datentyp der Operanden, Adressierung) werden im Befehlswort strukturiert zusammengefasst und binär codiert. Die Befehlswortlänge ist allgemein nicht für jeden Befehl des Befehlssatzes einheitlich (byteweise abgestuft).



Die Positionen von Opcode und Adressteil sind nicht fest (der Opcode kann am Anfang oder Ende des Befehlswortes stehen aber auch über das Befehlswort verteilt sein).

Der Operationsteil kann je nach Befehlsformat gleichzeitig auch mehrere Operanden und oder Adressen, wie auch andere Informationen enthalten.

CISC/RISC

Je nach Umfang des im Prozessor realisierten Befehlssatzes können zwei Kategorien von Befehlssatz-Architekturen (ISA – Instruction Set Architecture) unterschieden werden:

1. **CISC** (Complex Instruction Set Computer)
   Befehlsvorrat: 400..500 Befehle/Befehlsformate
   verhältnismäßig leistungsfähige Einzelbefehle
   z.B.: DEC VAX, IBM 360, Intel x86

2. **RISC** (Reduced Instruction Set Computer)
   Befehlsvorrat: 40..50 Befehle/Befehlsformate
   hohe Ausführungsgeschwindigkeit, niedriger Decodierungsaufwand
   komplexe und lange Befehle werden konsequent vermieden
   z.B.: Sun SPARC, SGI MIPS, DEC ALPHA, HP PARISC, IBM PowerPC

Diese Unterscheidung beruht nicht primär auf Unterschieden in der Hardware-Realisierung oder dem eingesetzten Betriebssystem → Befehlssatz

CISC

- Befehlsworte und Opcode in komplexen Befehlsformaten mit variablen Befehlswortlängen und vielen komfortablen Adressierungsarten.
- Leistungsfähige, komplexe Befehle führen zu einer Verkürzung des Maschinenprogramms und damit zu einer Erhöhung der Codedichte.
- Realisierung der Maschinenbefehle durch Ausführung von Mikroprogrammen im Prozessor (der Befehlszyklus wird durch eine Mikroprogramm-Steuerung realisiert).
- Anzahl der benötigten Taktzyklen pro Befehl ist unterschiedlich (allgemein mehr als 1 Taktzyklus/Befehl).

Motivation für RISC

Bei einem komplexen Befehlssatz (CISC) werden 90% aller Operationen mit nur 10% der Befehle des Befehlssatzes durchgeführt. → 90/10 Regel

RISC

- Stark reduzierter Umfang an Befehlsformaten und Adressierungsarten (meist weniger als 4 Befehlsformate und 4 Adressierungsarten).
- Wenige einfache Basisbefehle, aus denen komplexe Operationen zusammengesetzt werden können.
- Load/Store-Architektur, ALU-Befehle realisieren keine Speicherzugriffe, Speicherzugriffe erfolgen über gesonderte Load/Store-Befehle.
- Universalregister-Architektur (meist 32 oder mehr Universalregister).
- Festverdrahtete Maschinenbefehle und fester Befehlszyklus, keine Mikroprogramm-Steuerung.
- Ausführung der meisten Befehle in nur einem Taktzyklus.

R. Chou und M. Horowitz: „The goal of any instruction format should be: (1) simple decode, (2) simple decode, (3) simple decode.“

3 Klassifikation Befehlssatz-Architekturen

Merkmale der Klassifikation:

- Operandenspeicherung innerhalb der CPU, wo und wie.
- Zahl der explizit im Befehl adressierten Operanden.
- Operandenspeicherung, Adressierung, wie spezifiziert.
- Operationen des Befehlssatzes.
- Typ und Länge der Operanden, wie spezifiziert.

4 Operandenspeicherung innerhalb der CPU

Hauptvarianten, Alternativen der Operandenspeicherung:

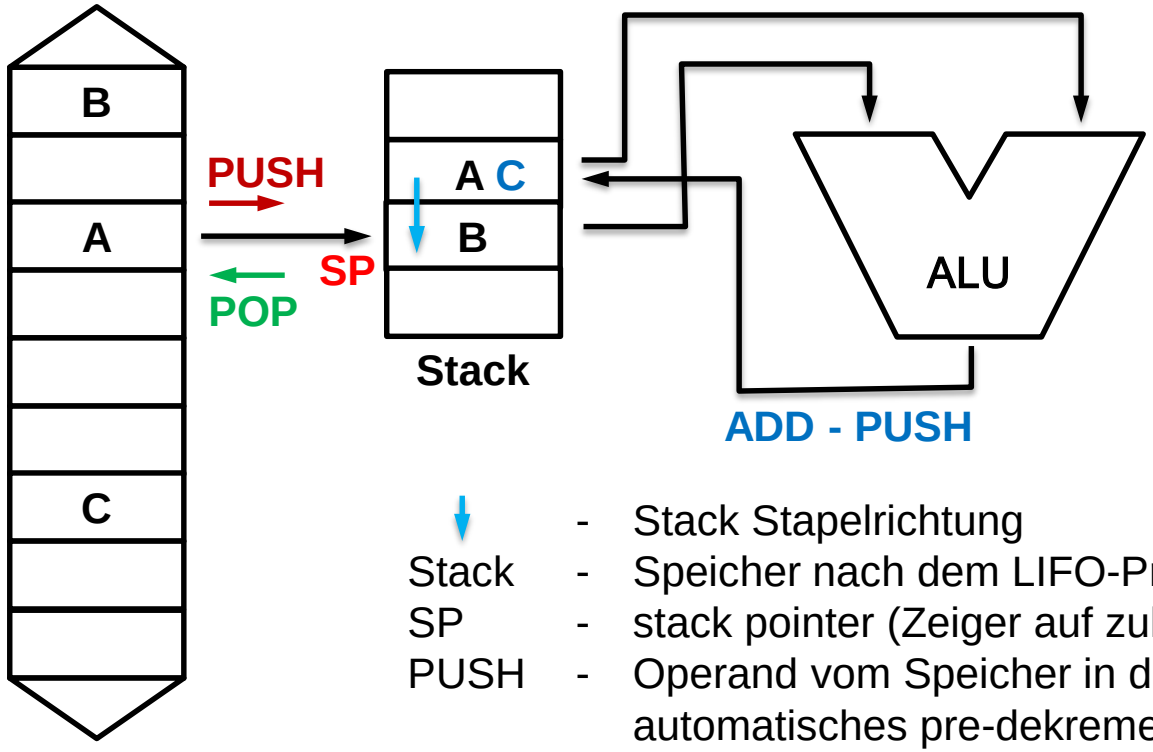
Alle bekannten Computer-Architekturen verfügen über temporären Operandenspeicher innerhalb der CPU (Register, Stack, Akkumulator)

temporärer Operandenspeicher	explizite Operanden	Quelle für Operanden	Ziel für Resultate	Zugriff auf Operanden
Stack	0	Stack	Stack	PUSH/POP auf Stack
Akkumulator (ACCU)	1	ACCU/ Speicher	ACCU	LOAD/STORE auf ACCU
Registersatz (GPR) (Universalregister)	2 oder 3	Register/ Speicher	Register/ Speicher	LOAD/STORE auf Register

Stack-Architektur (stack architecture)
Null-Adress-Maschine

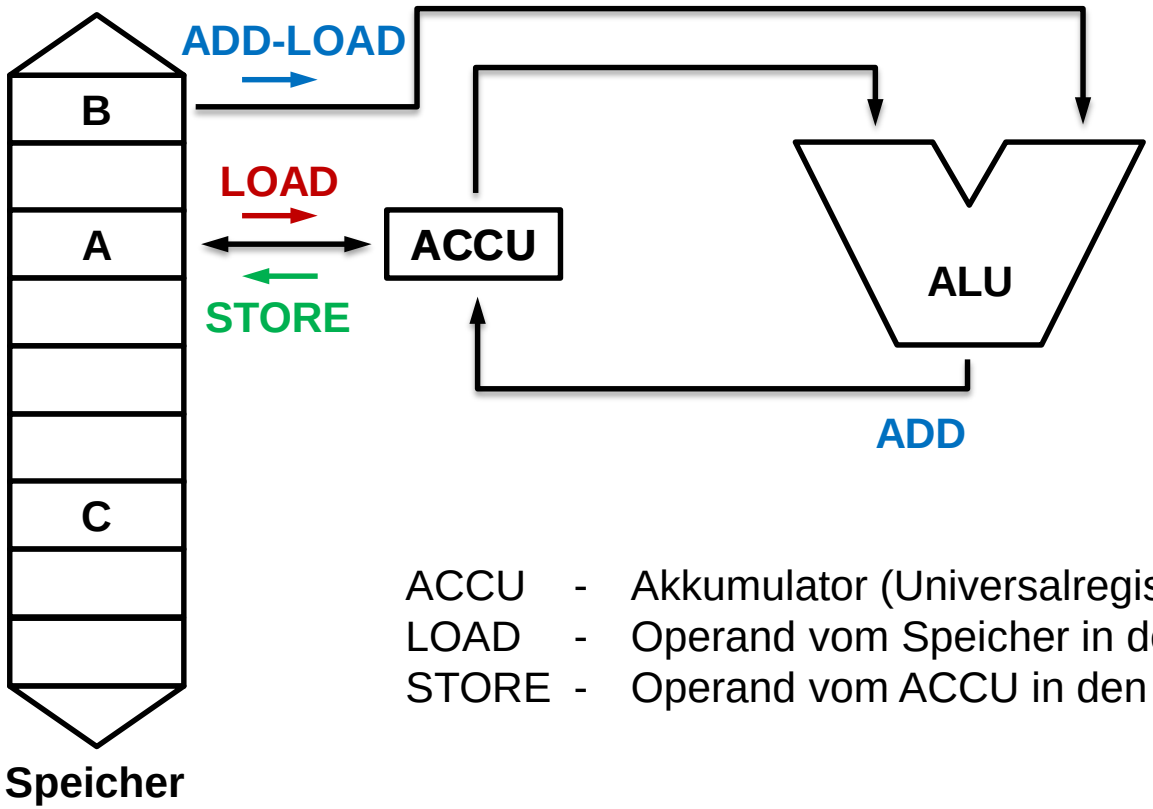
$C := A + B$

PUSH	A
PUSH	B
ADD	
POP	C



↓	-	Stack Stapelrichtung
Stack	-	Speicher nach dem LIFO-Prinzip (last in – first out)
SP	-	stack pointer (Zeiger auf zuletzt eingetragene Daten)
PUSH	-	Operand vom Speicher in den Stack transportieren automatisches pre-dekrement
POP	-	Operand vom Stack in den Speicher transportieren automatisches post-inkrement

Akkumulator-Architektur (accumulator architecture)
Ein-Adress-Maschine

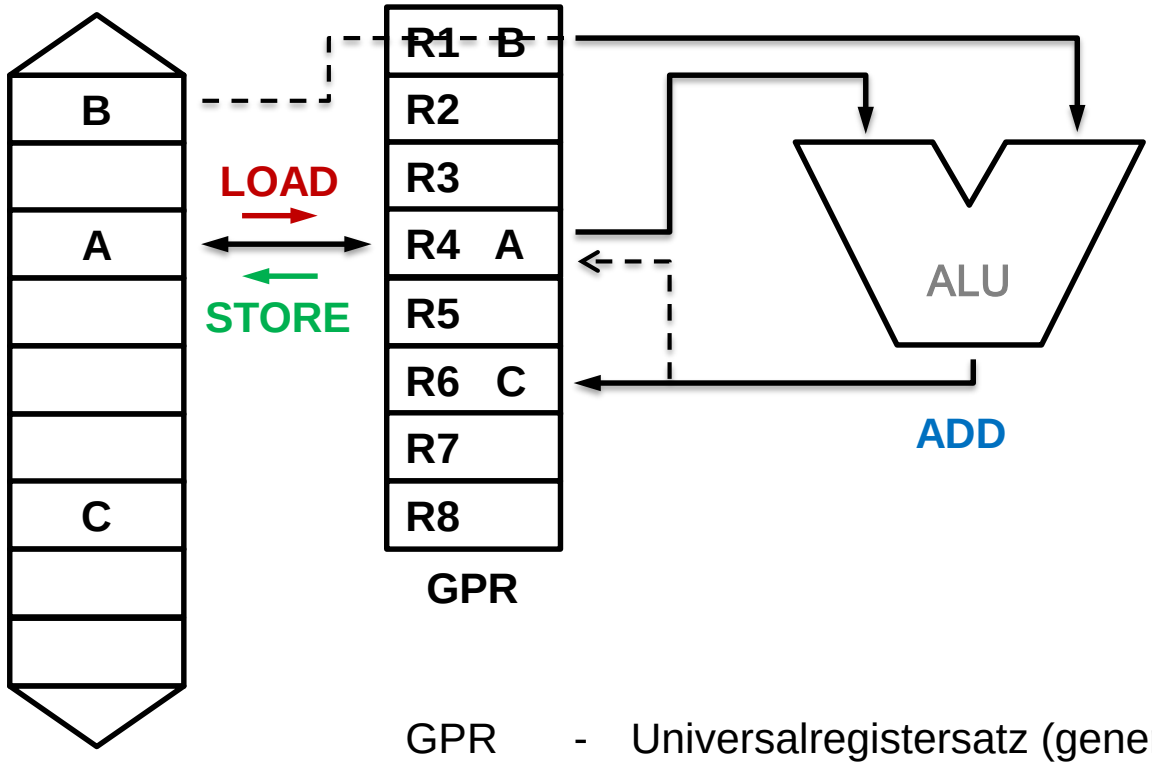


$C := A + B$

LOAD	A
ADD	B
STORE	C

ACCU - Akkumulator (Universalregister)
LOAD - Operand vom Speicher in den ACCU transportieren
STORE - Operand vom ACCU in den Speicher transportieren

Universalregistersatz-Architektur (GPR architecture)
Zwei/Drei-Adress-Maschine



Drei-Adress-Maschine

$C := A + B$

LOAD	R1, B
LOAD	R4, A
ADD	R6, R4, R1
STORE	C, R6

Zwei-Adress-Maschine

$C := A + B$

LOAD	R4, A
ADD	R4, B
STORE	C, R4

GPR - Universalregistersatz (general-purpose register)
LOAD - Operand vom Speicher in den GPR transportieren
STORE - Operand vom GPR in den Speicher transportieren

Vor- und Nachteile der Architekturtypen

Typ	Vorteile	Nachteile
Stack	einfaches Modell, gute Codedichte	kein direkter Zugriff auf Stack, nur relativ zum SP → Engpass
ACCU	kurze Befehle, minimale Hardware	ACCU ist einziger temporärer Speicher, höchster Speicherverkehr → Engpass
GRP	allgemeinstes Modell, Zwischenspeicherung der Operanden	alle Operanden explizit adressieren, lange komplexe Befehlswörter, schlechte Codedichte

Motivation für Universalregister-Architektur

- Register erlauben einen schnellen Zugriff auf die Operanden.
- Einfachere Adressierung der Register (kurze Adresslängen).
- Einbeziehung der Spezialregister in den Universalregistersatz.
- Nutzbar als zusätzliche Ebene in der Speicherhierarchie.
- Vielfältige Möglichkeiten der Zwischenspeicherung von Operanden.
- Für Compiler einfacher und effektiver nutzbar (z.B. Variablenübergabe).

5 Explizit im Befehl adressierte Operanden

monadische Operanden: (unäre, einstellige Operation) $B \coloneqq \mathrm{op}\ A$

dyadische Operanden: (binäre, zweistellige Operation) $C \coloneqq A\ \mathrm{op}\ B$

Die einstellige Operation kann auch durch die zweistellige dargestellt werden.

Für die zweistellige Operation (Verknüpfung op von zwei Operanden zu einem Resultat) sind mindestens folgende Angaben erforderlich:

- Art der Operation $\rightarrow$ op
- Adresse des 1. Operanden (1. Quelloperand) $\rightarrow A$
- Adresse des 2. Operanden (2. Quelloperand) $\rightarrow B$
- Adresse für das Resultat (Zieladresse) $\rightarrow C$

Die Befehlsformate, die sich daraus ergeben, werden entsprechend der typischen ALU-Befehle klassifiziert.

5-Adress-Befehlsformat

Die Codierung aller Angaben für einen allgemeinen dyadischen Befehl führt zu einem 5-Adress-Befehlswort. Der Operandenteil umfasst dabei die 1. und 2. Quelladresse der Operanden, die Zieladresse des Resultat und die beiden Adressen der möglichen Folgebefehle, mit und ohne Programmverzweigung.

Operationsteil $C \coloneqq A \ \mathbf{op} \ B$ **Operandenteil (Adressteil)**

Opcode	1. Quelladr.	2. Quelladr.	Zieladr.	1. Folgebef.	2. Folgebef.
op	A	B	C	$NI0$	$NI1$

Beispiel

Opcode	: 8 bit	→	256 verschiedene Befehle codierbar
Adressraum	: 32 bit	→	4 Gibyte Speicher adressierbar
Befehlswortlänge	:168 bit	→	6 32-bit-Worte pro Befehl

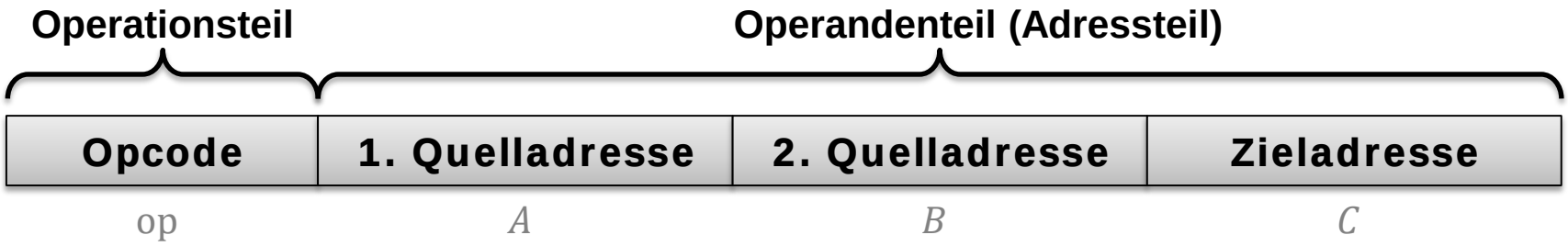
Übliche Befehlswortlängen z.B. 16…64 bit (→ 168 bit sehr ineffektiv)

Reduzierung des Operandenteils

Die beiden Adressen der möglichen Folgebefehle (mit und ohne Programmverzweigung) können durch die Verwendung eines Befehlszählers und gesonderter Verzweigungsbefehle völlig vermieden werden.

- **Befehlszähler:** enthält die Adresse für den unmittelbaren Folgebefehl
- **Verzweigungs-/Sprungbefehle:** enthalten Adressen der Folgebefehle
- **Lade-/Speicherbefehle:** Sonderbefehle für den Hauptspeichertransfer
- **Implizite Adressierung:** Quell oder Zieladressen implizit im Operationsteil
- **Überdeckte Adressierung:** gleichzeitige Nutzung als Quell-/Zieladresse
- **Direktoperanden:** Operanden werden direkt im Operationsteil codiert
- **Registeradressen:** gesonderter Adressraum, wesentlich kürzere Adressen
- **Mehrstufige Adressierung:** Umrechnung der Quell- und Zieladressen

3-Adress-Befehlsformat

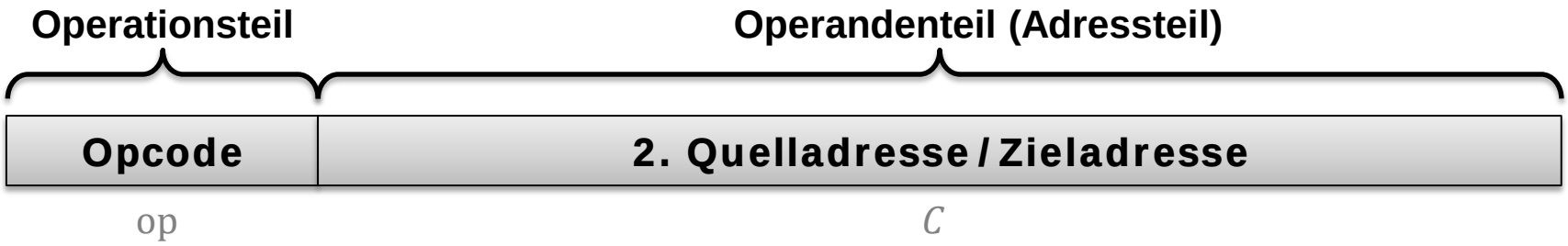


$C \coloneqq A \text{ op } B$ alle Quell- und Zieladressen explizit adressiert

2-Adress-Befehlsformat

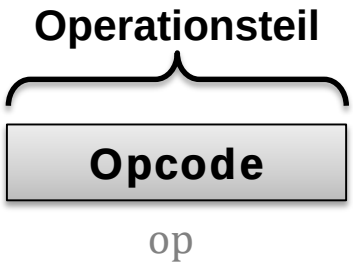
$C \coloneqq C \text{ op } A$ C – überdeckt adressiert

1-Adress-Befehlsformat



$C \coloneqq R4 \text{ op } C$ R4 – implizit, C – überdeckt adressiert

0-Adress-Befehlsformat

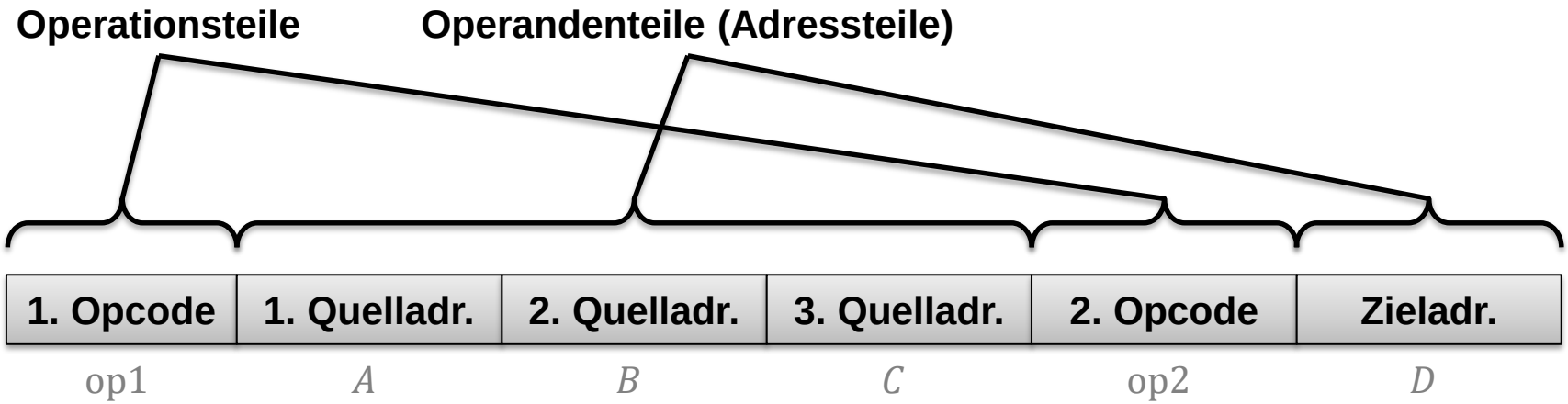


kein expliziter Operandenteil (Adressteil)

z.B. rsp #reset the stack pointer

alle Quell- und Zieladressen implizit adressiert

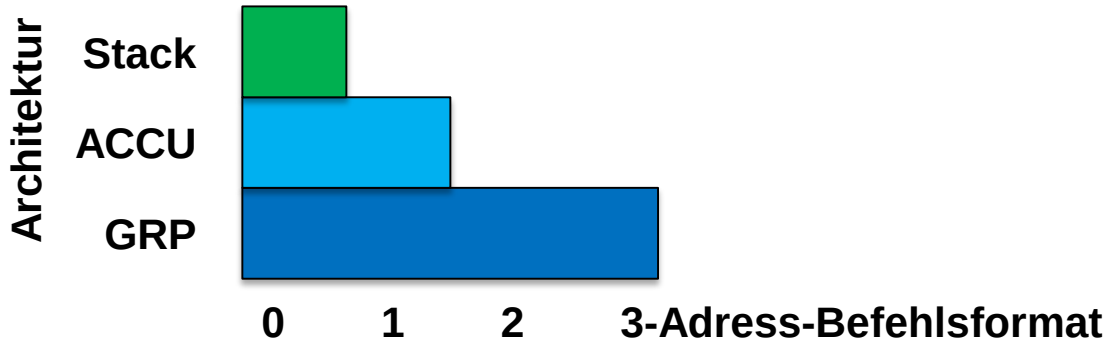
VLIW-Befehlsformat



$$D \coloneqq D + \big((A \text{ op1 } B) \text{ op2 } (A \text{ op1 } C)\big)$$

- Aus mehreren Einzelbefehlen zusammengesetzter komplexer Befehl.
- Kein festes Befehlsformat, Operation und Architektur angepasst.
- Speziell für die Nutzung von Parallelität, z.B. DSP-Architekturen.

Zuordnung Befehlsformat – Architekturtyp



Die GRP-Architektur (3-Adress-Maschine) ist bzgl. der verschiedenen Befehlsformate am universellsten, alle Formate sind prinzipiell möglich.

Bei der Stack-Architektur (0-Adress-Maschine) ist die Universalität nicht direkt sichtbar, Organisation und Funktionalität des Stacks im Zusammenhang mit der ALU sind entscheidend.

6 Operandenspeicherung, Adressierung

Folgende Betrachtungen nur für Universalregister-Architekturen!

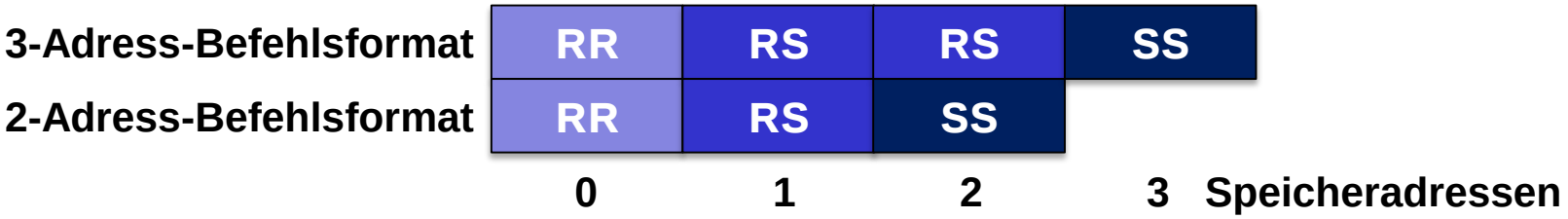
Wesentliche Befehlssatz-Charakteristiken bzgl. der Operanden:

1. Operandenzahl (für typischen ALU-Befehl)
   2-Adress-Befehlsformat (diadisch)
   3-Adress-Befehlsformat (triadisch)

2. Zahl der Speicheradressen (Speicheroperanden)
   0…3 Adressen für Speicherzugriffe

Die Adressen des Operandenteils, die nicht für einen Speicherzugriff genutzt werden, werden folglich entweder für Registeradressen oder auch für Direktoperanden verwendet.

Kombination von Register- und Speicheradressen



Klassifikation der Kombinationen Register-Speicher (Typen von Universalregister-Architekturen):

RR	Register - Register	Befehl	Load/Store-Architektur
RS	Register - Speicher	Befehl	
SS	Speicher - Speicher	Befehl	

Architekturen, die keinen Speicherzugriff für einen typischen ALU-Befehl erlauben, werden auch Load/Store-Architekturen genannt.

Vor- und Nachteile der einzelnen Typen

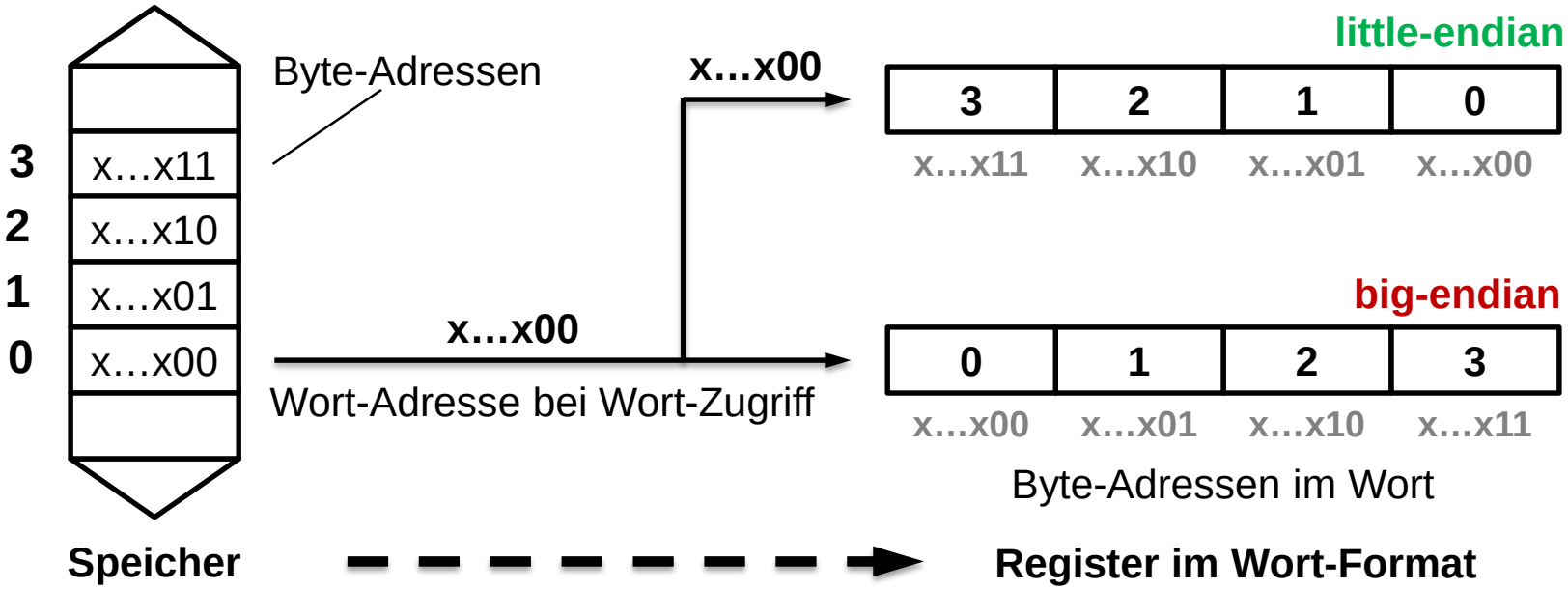
Typ	Vorteile	Nachteile
RR	einfache Befehlscodierung, feste Befehlswortlänge, einfache Codegenerierung, einfache Taktsteuerung	höhere Befehlsanzahl, gesonderte Load/Store-Befehle, relativ schlechte Codedichte
RS	Datenzugriff ohne gesonderte Load/Store-Befehle, gute Codedichte	Operanden nicht äquivalent
SS	kompakteste Form, keine Register für Zwischenwerte	große Befehlslängenunterschiede, zahlreiche Speicherzugriffe

RISC-Befehlssätze → typische Load/Store-Architekturen (nicht generell)

Speicheradressierung, Byte-Reihenfolge (byte order)

Byte-orientierter Speicherzugriff (kleinste adressierbare Einheit).
Der Zugriff auf größere Einheiten (Halbwort-, Wort-, Doppelwort-Zugriff, ...) führt zu Problemen bzgl. der internen Byte-Reihenfolge.

Interpretation von Speicheradressen (Wortzugriff)



Vereinbarungen zur Bytereihenfolge in einem Wort…

Bezeichnung	Bedeutung	Beispiel
little-endian byte order	Byteadresse x…x00 liegt bei niedrigster Wertigkeit im Wort	Intel 80x86, DEC VAX
big-endian byte order	Byteadresse x…x00 liegt bei höchster Wertigkeit im Wort	MIPS, SPARC

Für die Verarbeitung im Rechner ist die Byte-Reihenfolge meist unbedeutend. Probleme allgemein nur bei Byte-Wort-Umrechnungen.

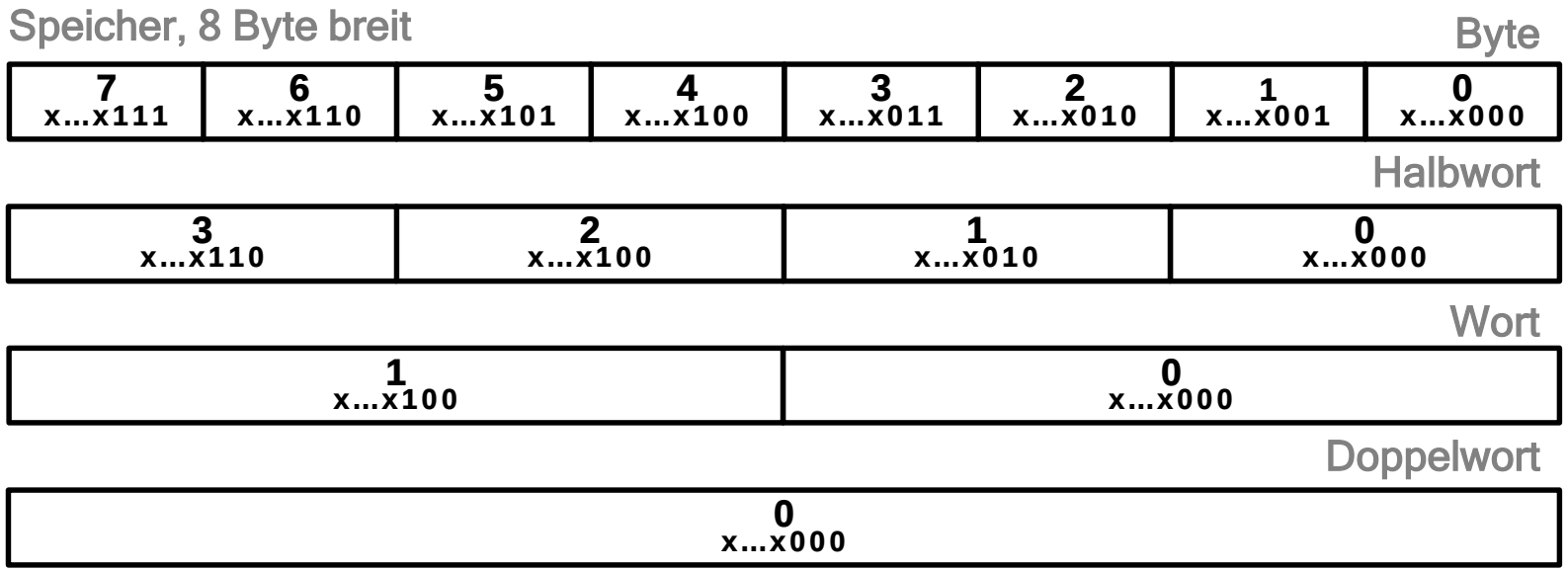
Beim Datenaustausch zwischen Rechnern unterschiedlicher Byte-Reihenfolge ist dies unbedingt zu beachten (ggf. Konvertierungen).

Im Internet wird standardmäßig big-endian byte order verwendet.

Speicherausrichtung (data-alignment)

Der Zugriff auf Einheiten, die größer als ein Byte sind, kann ausgerichtet (aligned) oder nicht ausgerichtet (misaligned) erfolgen. Ein Zugriff auf eine Einheit der Länge m Byte ab der Byte-Adresse A ist ausgerichtet, wenn $A \bmod m = 0$ gilt → natürliches Alignment.

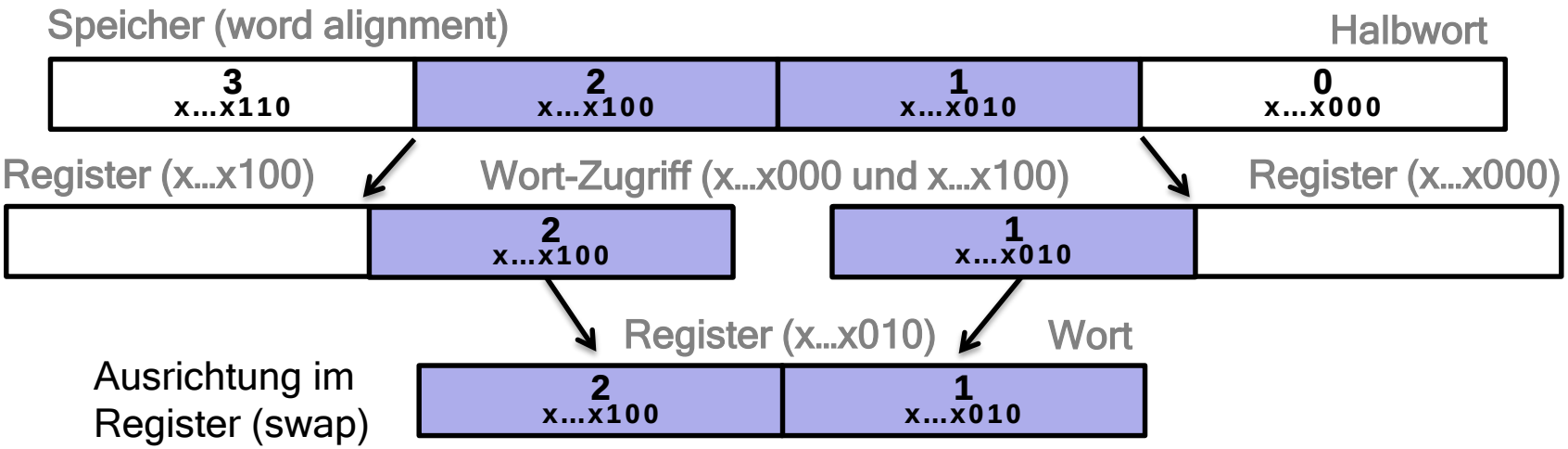
Ausgerichtete Speicherzugriffe bis Doppelwert-Grenze



Alignment-Restriktion (1)

Soll z.B. auf ein Wort auf einer nicht ausgerichteten Halbwortgrenze (x…x010) zugegriffen werden, so müssen bei ausgerichtetem Wortzugriff die beiden benachbarten Worte (x…x000 und x…x100) eingelesen werden und anschließend die darin enthaltenen Halbworte (x…x010 und x…x100) wieder zu einem Wort zusammengesetzt (ausgerichtet) werden.

Ausgerichteter Wort-Speicherzugriff auf nichtausgerichtetes Wort

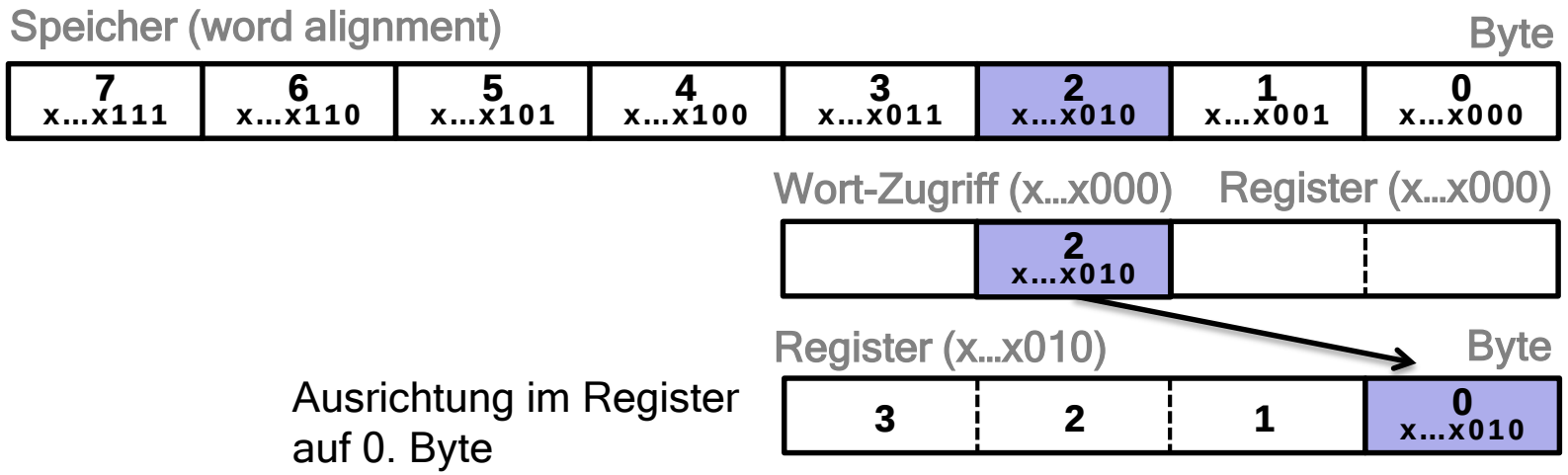


Alignment-Restriktion (2)

Ein nicht ausgerichteter Speicherzugriff erfordert im Allgemeinen mehrere ausgerichtete Zugriffe und eine anschließende Ausrichtung im Prozessor (Ausrichtungsnetzwerk, zusätzliche Befehle). Ausgerichtete Zugriffe sind allgemein schneller. Bei kürzeren Einheiten als die Registergröße ist ebenfalls ein Ausrichtungsnetzwerk erforderlich (Multiplexer-Netzwerk, Barrel-Shifter).

Byte-Zugriff ist grundsätzlich immer möglich (egal welches Byte)!

Ausgerichteter Wort-Speicherzugriff auf nichtausgerichtetes Byte



Daten-Strukturen, Füllbyte (Padding)

Aufbau einer Datenstruktur, Record aus mehreren Datenfeldern

```
record
a : byte;  // 8-Bit-Ganzzahl, vorzeichenbehaftet (2er-Komplement)
b : float; // IEEE 754: Einfache Genauigkeit (Single)
c : short; // 16-Bit-Ganzzahl, vorzeichenbehaftet (2er-Komplement)
end record;
```

Belegung hexadezimal: `.a/3A/`, `.b/C1F3 B6A0 /`, `.c/E072/`.

Speicherlayout dieses Records (Basisadresse 0x120) unter Beibehaltung der Deklarationsreihenfolge der Datenfelder, gemäß natürlichem Alignment.

XX – Füllbyte (Padding)

big-endian byte order

0x120	0x121	0x122	0x123	0x124	0x125	0x126	0x127	0x128	0x129	0x12A	0x12B
3A	XX	XX	XX	C1	F3	B6	A0	E0	72	XX	XX

Reihenfolge innerhalb Byte bleibt erhalten!

little-endian byte order

0x120	0x121	0x122	0x123	0x124	0x125	0x126	0x127	0x128	0x129	0x12A	0x12B
3A	XX	XX	XX	A0	B6	F3	C1	72	E0	XX	XX

Adressierung

Operanden, Adressen und Befehle können im Hauptspeicher oder im Registerspeicher stehen,. Sie werden über ihre Adressen angesprochen (getrennte Adressräume). Die Adressen können aus verschiedenen Komponenten zusammengesetzt sein.

Adressen

Adressen beziehen sich auf Konstanten (Direktoperand), Register (Registeradresse) oder den Hauptspeicher (Speicheradresse).

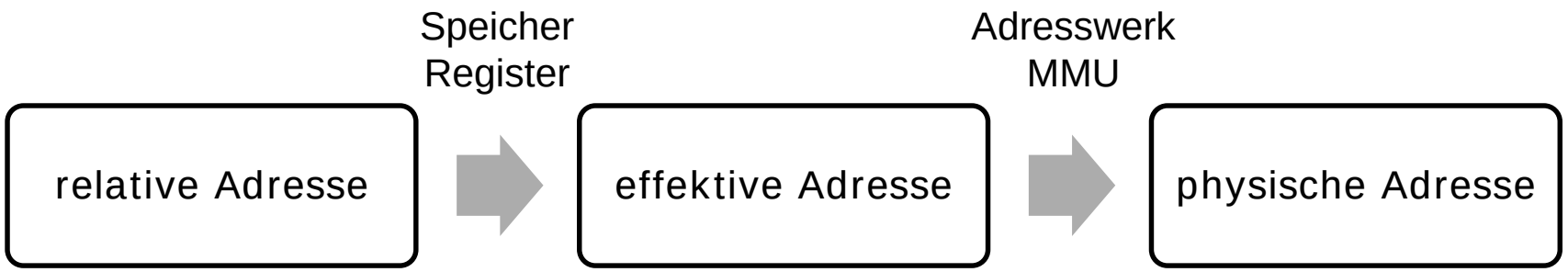
Statische, absolute Adressierung (physische Adressen):

Die physischen Adressen werden fest, statisch zur Programmierzeit angegeben. Programme und Daten sind lageabhängig.

Dynamische, relative Adressierung (effektive Adressen):

Die effektive Adresse wird erst zur Laufzeit durch eine Adressrechnung gewonnen. Im Adresswerk werden dann aus den effektiven Adressen die eigentlichen physischen Adressen für die Adressierung gebildet.

Relative Adressierung



Formale Notation der Adressrechnung

Hauptspeicheradresse	(Assambler)	:	A
Registeradresse	(Assambler)	:	RA
Direktoperand	(Assambler)	:	# Operand
Hauptspeicherinhalt von A	(Assambler)	:	(A) oder @(A) auch ((A))
Hexadezimalwert	(Assambler)	:	$ HEX
Hauptspeicherinhalt von A		:	M[A]
Registerinhalt von RA		:	RA

Adressierungsarten (1)

Adressierungsarten sind alle Möglichkeiten eines Prozessors aus relativen Adressen effektiven/physischen Adressen zu berechnen (zur Laufzeit).

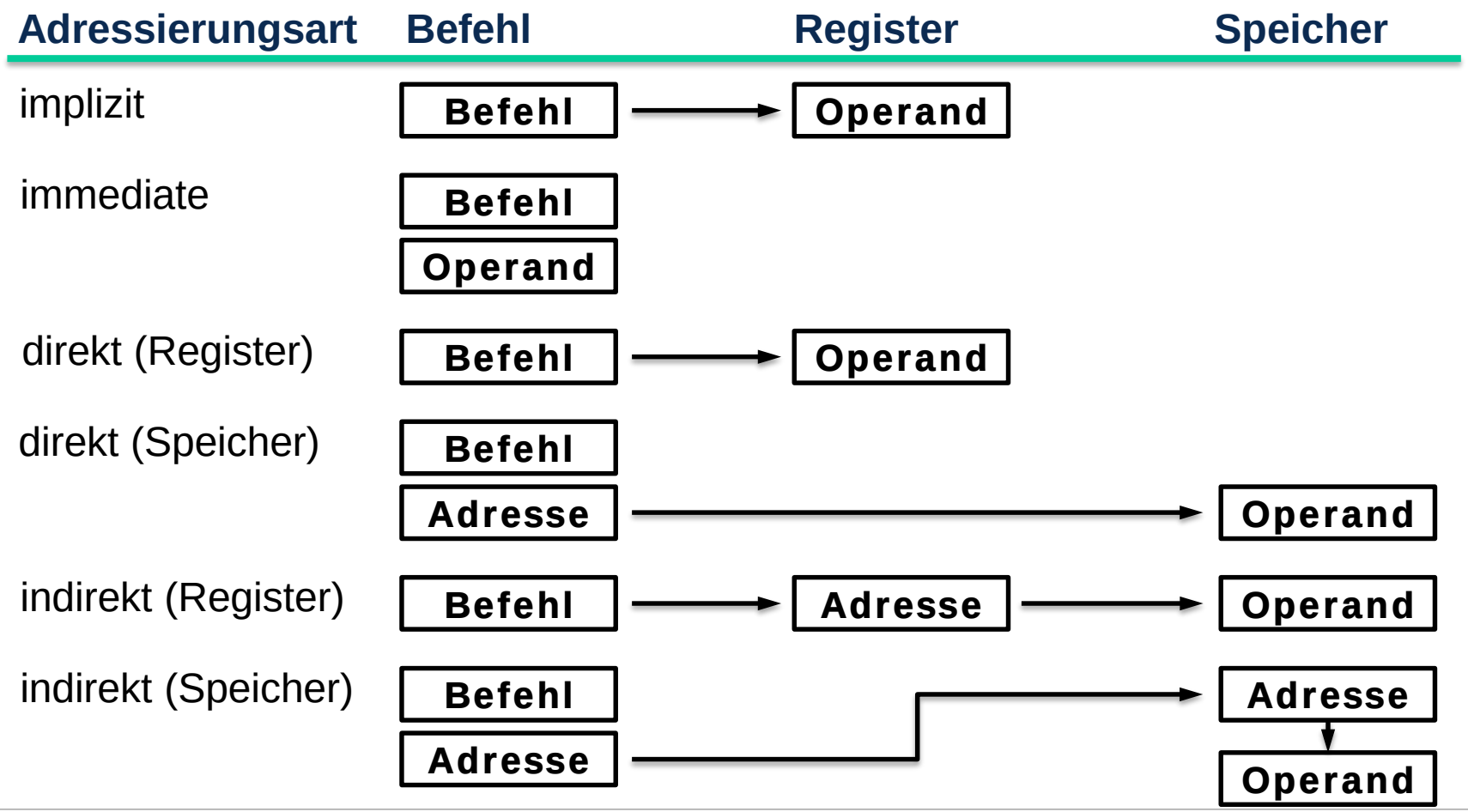
Vorteile des effektiven Einsatzes der Adressierungsarten:

- Einsparung von Hauptspeicherplatz, Rechenzeit und Programmierzeit.
- Entlastung des Programmierers von aufwendiger Adressrechnung.
- Lageunabhängigkeit der Daten und Programme (relative Adressierung).
- Wiederverwendbarkeit von Programmteilen (Unterprogrammtechnik, …).
- Ermöglichung der wiederholten Befehlsausführung auf verschiedene Daten (z.B. Tabellen, Schleifen), sowie bedingte Operationen.

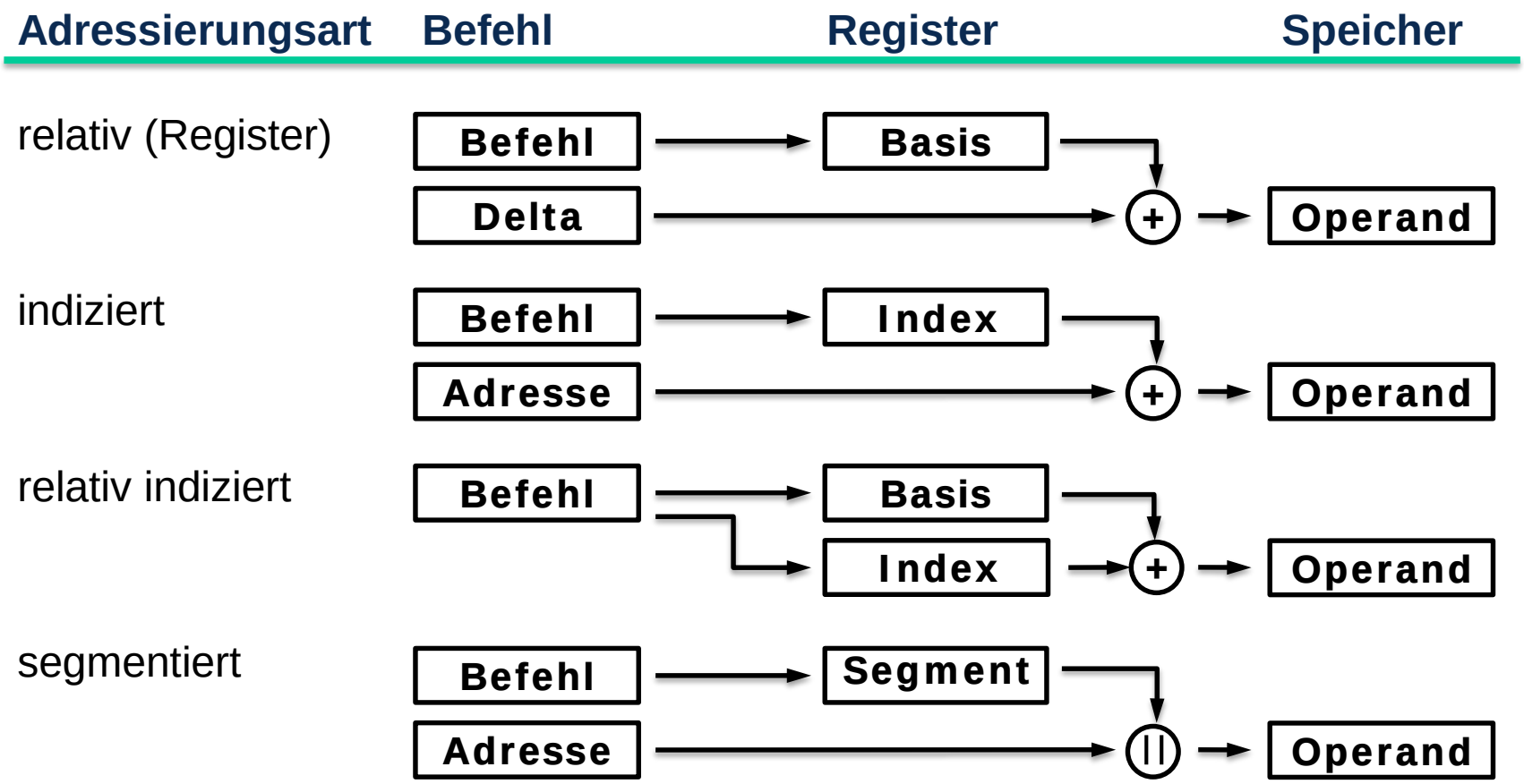
Adressierungsarten (2)

Adressierungsart	Beschreibung
implizit	Adressen, Operanden durch den Opcode implizit festgelegt.
immediate	Operand wird im Befehl explizit mitgeführt (Direktoperand).
direkt	Adresse wird im Befehl explizit mitgeführt (Direktadresse).
indirekt	Befehl enthält die Adresse des Speicherplatzes, in dem sich die eigentliche Adresse befindet (Adresse von Adresse).
relativ (based)	Befehl enthält einen Offset (Verschiebung), mit dem die Adresse relativ zu einer Basisadresse (Basisregister) gebildet wird.
indiziert	Befehl enthält eine Basisadresse, mit der die Adresse durch Addition eines Index (Indexregister) gebildet wird.
segmentiert	Adresse wird an den Inhalt eines Segmentregisters angehängt (concatenate, Seitenadressierung).
virtuell	Umsetzung einer virtuellen Adresse in eine physische Adresse.

Übersicht Befehl – Register – Speicher (1)



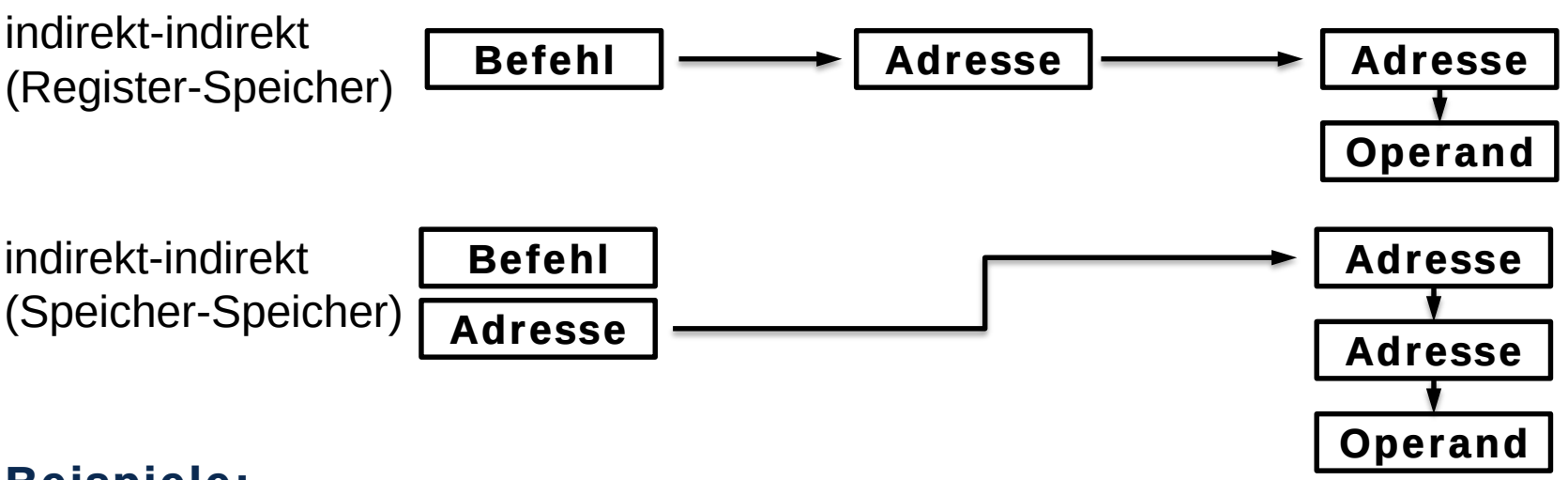
Übersicht Befehl – Register – Speicher (2)



Delta → Verschiebung, Displacement

Übersicht Befehl – Register – Speicher (3)

Adressierungsart	Befehl	Register	Speicher



Beispiele:

relative Adressierung: Verzweigungsbefehle, PC-relativ

Indizierte Adressierung: Adressierung von Feldern

Beispiele Adressierung

Adressierungsart	Befehl (Assamber)	Ergebnis (Zuweisung)
immediate	ADD R2, #7	$R2 \coloneqq R2 + 7$
direkt	ADD R2, R3	$R2 \coloneqq R2 + R3$
indirekt (Register)	ADD R2, (R3)	$R2 \coloneqq R2 + M[R3]$
indirekt (Speicher)	ADD R2, @(R3)	$R2 \coloneqq R2 + M[M[R3]]$
relativ (Displacement)	ADD R2, 10(R3)	$R2 \coloneqq R2 + M[10 + R3]$
relativ-indiziert	ADD R2, (R3+R4)	$R2 \coloneqq R2 + M[R3 + R4]$

7 Operationen des Befehlssatzes

Bezüglich der Funktionalität sind vier Klassen von Befehlen unterscheidbar:

Operationstyp	Beispiel
Datenübertragung	Register-Register, Register-Speicher Speicher-Speicher Register-E/A
Datenmanipulation	arithmetische Operationen logische Operationen Schiebe-/Rotationsoperationen
Verzweigung (bedingt, unbedingt)	Sprung- und Verzweigungsoperationen Prozeduraufrufe und -rückkehr Traps, Exceptions
Systemsteuerung	Betriebssystemaufrufe Speicherverwaltung Interrupts, Traps, Exceptions

Datenübertragung (Beispiele)

Befehl	Bedeutung	
LD	Laden eines Registers	load
ST	Speichern des Inhaltes eines Registers	store
MOVE	Übertragung (beliebige Richtung)	move
EXC	Vertauschen der Inhalte	exchange
TFR	Übertragen eines Registers in ein anderes	transfer
PUSH	Ablegen eines oder mehrerer Register in den Stack	push
POP	Laden eines oder mehrerer Register aus dem Stack	pop
READ	Lesen des Prozessor-Statusregisters	read
WRITE	Schreiben des Prozessor-Statusregisters	write
IN	Laden eines Registers aus einem Peripheriebaustein	input
OUT	Übertragen eines Registers in einen Peripheriebaustein	output

Datenmanipulation, arithmentisch (Beispiele)

Befehl	Bedeutung	
ABS	Absolutbetrag	absolute
ADD	Addition	addition
SUB	Subtraktion	substraction
MUL	Multiplikation	multiply
DIV	Division	divide
COM	Einerkomplement	complement
NEG	Vorzeichenwechsel, Zweierkomplement	negate
CLR	Löschen	clear
CMP	Operandenvergleich	compare
DAA	Dual-Dezimal-Umwandlung	decimal adjust
DEC	Dekrement	decrement
INC	Inkrement	increment

Datenmanipulation, boolesch (Beispiele)

Befehl	Bedeutung	
AND	AND-Verknüpfung	and
OR	OR-Verknüpfung	or
EOR	XOR-Verknüpfung	exclusive or
NOT	NOT-Verknüpfung	not

Datenmanipulation, Flag und Bit (Beispiele)

Befehl	Bedeutung	
SEF	Setzen eines Bedingungs-Flags	flag set
CLF	Löschen eines Bedingungs-Flags	flag clear
TST	Prüfen eines Flags oder Bits	test
BSET	Setzen eines Bit	bit set
BCLR	Rücksetzen eines Bit	bit clear
BCHG	Invertieren eines Bit	bit change
BFCLR	Rücksetzen der Bits eines Bitfeldes	clear bits
BFSET	Setzen der Bits eines Bitfeldes	set bits
BFFFO	Finden der ersten 1 im Bitfeld	find first one
BFEXT	Lesen eines Bitfeldes	extract bits
BFINS	Einfügen eines Bitfeldes	insert bits

Datenmanipulation, String oder Block (Beispiele)

Befehl	Bedeutung	
MOVS	Transferieren eines Blocks	move string
INS	Einlesen eines Blocks von der Peripherie	input string
OUTS	Ausgabe eines Blocks an die Peripherie	output string
CMPS	Vergleich zweier Blöcke	compare string
COPS	Kopieren eines Blockes	copy string
SCAS	Suchen eines Zeichens in einem Block	scan string

Datenmanipulation, Schiebe und Rotation (Beispiele)

Befehl	Bedeutung	
SHF	Verschieben eines Registerinhaltes	shift
ASL	arithmetische Links-Verschiebung	arith. shift left
ASR	arithmetische Rechts-Verschiebung	arith. shift right
LSL	logische Links-Verschiebung	shift left
LSR	logische Rechts-Verschiebung	shift right
ROT	Rotation eines Registerinhaltes	rotate
ROL	Rotation nach links	rotate left
ROR	Rotation nach rechts	rotate right
SWAP	Vertauschen der beiden Hälften eines Registers	swap

Verzweigungen (Beispiele)

Befehl	Bedeutung	
JMP	unbedingter Sprung zu einer Adresse	jump
BCC	Verzweigung, falls Bedingung cc erfüllt	branch
BRA	Verzweigung ohne Bedingungsabfrage	branch always
CALL, JSR	Sprung in ein Unterprogramm	jump to subroutine
BSRCC	JSR, wenn Bedingung cc erfüllt	branch to subr.
RTS	Rücksprung aus einem Unterprogramm	return from subr.
TRAP, INT	Sprung in Unterbrechungsroutine	software interrupt
RTI, RTE	Rücksprung aus Unterbrechungsroutine	return from int.

cc-Bedingungen für Verzweigung (Beispiele) (1)

cc	Bedingung	Bedeutung
CS	CF=1	branch on carry set
CC	CF=0	branch on carry clear
VS	OF=1	branch on overflow
VC	OF=0	branch on not overflow
EQ	ZF=1	branch on zero/equal
NE	ZF=0	branch on not zero/equal
MI	SF=1	branch on minus
PL	SF=0	branch on plus
PA	PF=1	branch on parity/parity even
NP	PF=0	branch on not parity/parity even

cc-Bedingungen für Verzweigung (Beispiele) (2)

cc	Bedingung	Bedeutung
vorzeichenlose Operanden		
LO	CF=1	branch on lower than
HS	CF=0	branch on higher or same
LS	CF v ZF=1	branch on lower or same
HI	CF v ZF=0	branch on higher than
vorzeichenbehaftete Operanden		
LT	SF ≠ OF=1	branch on less than
GE	SF ≠ OF=0	branch on greater or equal
LE	ZF v (SF ≠ OF)=1	branch on less or equal
GT	ZF v (SF ≠ OF)=0	branch on greater than

Systemsteuerung (Beispiele)

Befehl	Bedeutung	
NOP	keine Operation	no operation
WAIT	Warten auf spezielles Eingangssignal	wait
SYNC	Warten auf einen Interrupt	sync
HALT, STOP	Anhalten eines Prozessors	stop
RESET	Rücksetzsignal für Peripherie	reset
SVC	Betriebssystem-Aufruf	supervisor call

8 Typ und Länge der Operanden

Operandentypen (Datentypen)

Der Typ der im Befehl adressierten Operanden wird allgemein im Befehl selbst festgelegt. Die Codierung erfolgt dabei im Opcode zusammen mit der durchzuführenden Operation.

Alternativ wird auch die Abspeicherung von Typerkennungen (Tag) zusammen mit den Daten verwendet (Datenflussmaschinen, ...). Sogenannte „Tagged“ Architekturen sind jedoch eher die Ausnahme.

Numerische, nichtnumerische Daten

Numerische Daten

- Vorzeichenlose ganze Zahlen (unsigned integer)
- vorzeichenbehaftete ganze Zahlen (signed integer)
- binär codierte Dezimalzahlen (binary coded decimal integer, BCD)
- Gleitkommazahlen (floating point)

Nichtnumerische Daten

- alphanumerische Zeichen (characters, ASCII)
- Zeichenketten (character strings)
- Boolesche Werte (boolean values)
- Bitfelder (bit map)
- Zeiger, Adressen (pointer)

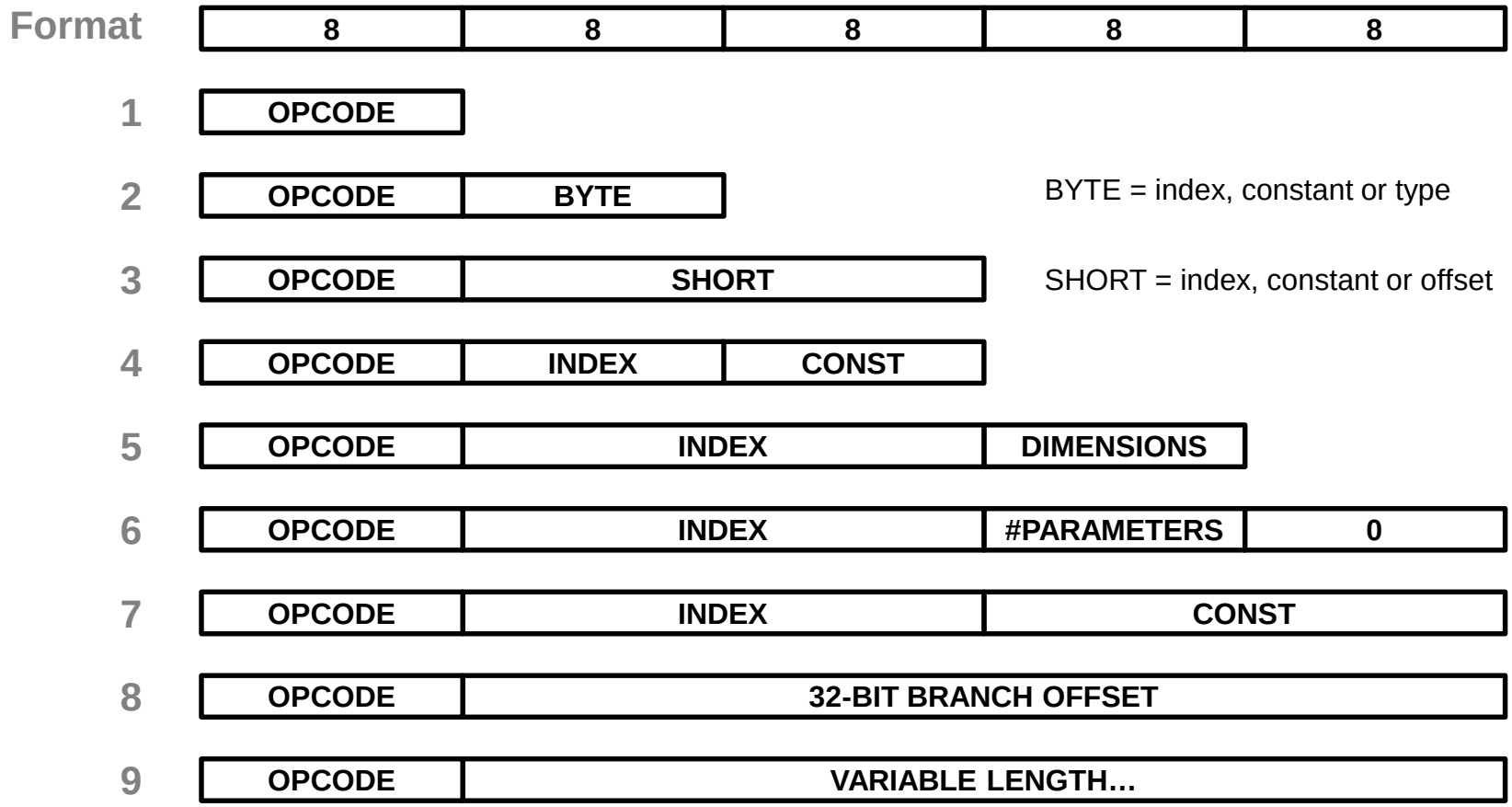
Operandenlänge (Datenformat)

Die Operandenlänge ist allgemein durch den Operandentyp gegeben, wobei pro Typ auch unterschiedliche Längen möglich sind (im Befehl codiert).

Zuordnungsbeispiel Operandentyp-Operandenlänge:

Typ	8 Bit	16 Bit	32 Bit	64 Bit	128 Bit
signed integer					
unsigned integer					
BCD					
floating point					
characters					
boolean					
bitmap					
pointer					

9 Beispiele zu Befehlssatz-Architekturen
9.1 JVM (STACK) – Instruction Formats



JVM (STACK) – Instruction Set (1)

Load/Store	
typeLOAD ind8	Push local variable onto stack
typeALOAD	Push array element on stack
BALOAD	Push byte from any array on stack
SALOAD	Push short from any array on stack
CALOAD	Push char from any array on stack
AALOAD	Push pointer from any array on stack
typeSTORE ind8	Pop value and store in local var
typeASTORE	Pop value and store in array
BASTORE	Pop byte and store in array
SASTORE	Pop short and store in array
CASTORE	Pop char and store in array
AASTORE	Pop pointer and store in array

Push	
BIPUSH con8	Push a small constant on stack
SIPUSH con16	Push 16-bit constant on stack
LDC ind8	Push constant from const pool
typeCONST_#	Push immediate constant
ACONST_NULL	Push an null pointer on stack

JVM (STACK) – Instruction Set (2)

Arithmetic		Boolean/Shift	
typeADD	Add	iIAND	Boolean AND
typeSUB	Subtract	iIOR	Boolean OR
typeMUL	Multiply	iIXOR	Boolean Exclusive OR
typeDIV	Divide	iISHL	Shift left
typeREM	Remainder	iISHR	Shift right
typeNEG	Negate	iIUSHER	Unsigned shift right

Conversion		Stack Management	
x2y	Convert x to y	DUPxx	Six instructions for duping
i2c	Convert integer to char	POP	Pop an int from stk and discard
i2b	Convert integer to byte	POP2	Pop two ints from stk and discard
		SWAP	Swap top two ints on stack

JVM (STACK) – Instruction Set (3)

Comparison	
IF_ICMPrel offset16	Conditional branch
IF_ACMPEQ offset16	Branch if two ptrs equal
IF_ACMPNE offset16	Branch if ptrs unequal
Ifrel offset16	Test 1 value and branch
IFNULL offset16	Branch if ptr is null
IFNONNULL offset16	Branch if ptr is not nonnull
LCMP	Compare two longs
FCMPL	Compare 2 fbats for <
FCMPG	Compare 2 fbats for >
DCMPL	Compare doubles for <
DCMPG	Compare doubles for >

Transfer of Control	
INVOKEVIRTUAL ind16	Method invokation
INVOKESTATIC ind16	Method invokation
INVOKEINTERFACE…	Method invokation
INVOKESPECIAL ind16	Method invokation
JSR offset16	Invoke finally clause
typeRETURN	Return value
ARETURN	Return pointer
RETURN	Return void
RET ind8	Return from finally
GOTO offset16	Unconditional brance

JVM (STACK) – Instruction Set (4)

Miscellaneous (1)	
IINC ind8, con8	Increment local variable
WIDE	Wide prefix x
NOP	No operation
GETFIELD ind16	Read field from object
PUTFIELD ind16	Write field to object
GETSTATIC ind16	Get static field from class
NEW ind16	Create new object
INSTANCEOF offset16	Determine type of obj
CHECKCAST ind16	Check object type
ATHROW	Throw exception
LOOKUPSWITCH…	Sparse multiway branch
TABLESWITCH…	Dense multiway branch

Miscellaneous (2)	
MONITORENTER	Enter a monitor
MONITOREXIT	Leave a monitor

Arrays	
ANEWARRAY ind16	Create array of ptrs
NEWARRAY atype	Create array of atype
MULTINEWARRAY ind16,d	Create multidim array
ARRAYLENGTH	Get array length

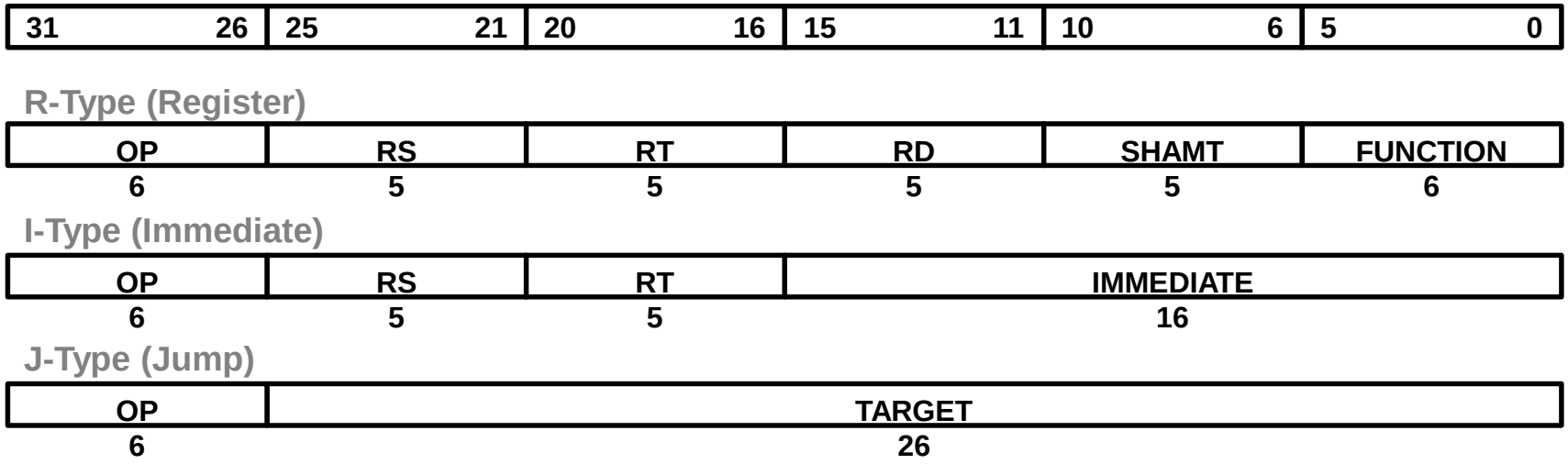
ind8/16 = index of local variable
con8/16, d, atype = constant
type, x, y = I, L, F, D offset16 for branche

JVM (STACK) – Numeric Data Types/Addressing Modes

Type	8 Bits	16 Bits	32 Bits	64 Bits	128 Bits
Signed integer	X	X	X	X	
Unsigned integer					
Binary coded decimal integer					
Floating point			X	X	

Addressing Mode	
Immediate	X
Direct	
Register	
Register indirect	
Indexed	X
Based-Indexed	
Stack	X

9.2 R2000 (MIPS) – Instruction Formats



OP	- operation code
RS	- source register specifier
RT	- target (source/destination) register or branche condition
RD	- destination register specifier
SHAMT	- shift amount
FUNCTION	- function field
IMMEDIATE	- immediate, branche displacement or address displacement
TARGET	- jump target address

R2000 (MIPS) – Instruction Set (1)

Load/Store	
LB rt, offset(base)	Load byte
LBU rt, offset(base)	Load byte unsigned
LH rt, offset(base)	Load halfword
LHU rt, offset(base)	Load halfword unsigned
LW rt, offset(base)	Load word
LWL rt, offset(base)	Load word left
LWR rt, offset(base)	Load word right
SB rt, offset(base)	Store word
SH rt, offset(base)	Store halfword
SW rt, offset(base)	Store word
SWL rt, offset(base)	Store word left
SWR rt, offset(base)	Store word right

Arithmetic (I-Type)	
ADDI rt, rs, immediate	Add immediate
ADDIU rt, rs, immediate	Add immediate unsigned
SLTI rt, rs, immediate	Set on less than immediate
SLTIU rt, rs, immediate	Set on less than immediate unsigned
ANDI rt, rs, immediate	AND immediate
ORI rt, rs, immediate	OR immediate
XORI rt, rs, immediate	Exclusive OR immediate
LUI rt, immediate	Load upper immediate

R2000 (MIPS) – Instruction Set (2)

Arithmetic (R-Type)	
ADD rd, rs, rt	Add
ADDU rd, rs, rt	Add unsigned
SUB rd, rs, rt	Substract
SUBU rd, rs, rt	Substract unsigned
SLT rd, rs, rt	Set on less than
SLTU rd, rs, rt	Set on less than unsigned
AND rd, rs, rt	AND
OR rd, rs, rt	OR
XOR rd, rs, rt	Exclusive OR
NOR rd, rs, rt	OR-NOT

Shift	
SLL rd, rt, shamt	Shift left logical
SRL rd, rt, shamt	Shift right logical
SRA rd, rt, shamt	Shift right arithmetic
SLLV rd, rt, rs	Shift left logical variable
SRLV rd, rt, rs	Shift right logical variable
SRAV rd, rt, rs	Shift right arithmetic variable

Special	
SYSCALL	System call
BREAK	Breakpoint

R2000 (MIPS) – Instruction Set (3)

Multiply/Divide	
MULT rs, rt	Multiply
MULTU rs, rt	Multiply unsigned
DIV rs, rt	Divide
DIVU rs, rt	Divide unsigned
MFHI rd	Move from HI
MFLO rd	Move from LO
MTHI rd	Move to HI
MTLO rd	Move to LO

Jump/Branch	
J target	Jump
JAL target	Jump and link
JR rs	Jump register
JALR rs, rd	Jump and link register
BEQ rs, rt, offset	Branch on equal
BNE rs, rt, offset	Branch on not equal
BLEZ rs, offset	Branch on ≤ zero
BGTZ rs, offset	Branch on > zero
BLTZ rs, offset	Branch on < zero
BGEZ rs, offset	Branch on ≥ zero
BLTZAL rs, offset	Branch on < zero and link
BGEZAL rs, offset	Branch on ≥ zero and link

R2000 (MIPS) – Instruction Set (4)

Coprocessor	
LWCz rt, offset(base)	Load word to coprocessor
SWCz rt, offset(base)	Store word from coprocessor
MTCz rt, rd	Move to coprocessor
MFCz rt, rd	Move from coprocessor
CTCz rt, rd	Move control to coprocessor
CFCz rt, rd	Move control from coprocessor
COPz cofun	Coprocessor operation
BCzT offset	Branche on coprocessor z true
BCzF offset	Branche on coprocessor z false

System Control Coprocessor (CP0)	
MTC0 rt, rd	Move to CP0
MFC0 rt, rd	Move from CP0
TLBR	Read indexed TLB entry
TLBWI	Write indexed TLB entry
TLBWR	Write random TLB entry
TLBP	Probe TLB for matching entry
RFE	Restore from exception

R2000 (MIPS) – Numeric Data Types/Addressing Modes

Type	8 Bits	16 Bits	32 Bits	64 Bits	128 Bits
Signed integer	X	X	X	X	
Unsigned integer	X	X			
Binary coded decimal integer					
Floating point			(X)[1]	(X)[1]	

Addressing Mode	
Immediate	X
Direct	
Register	X
Register indirect	X
Indexed	
Based-Indexed	
Stack	

[1] – only with FPU R2010

R2000 (MIPS) – Register

Number	Register Name	Usage
0	Zero	Constant 0
1	At	Reserved for assambler
2 – 3	v0 – v1	Expression evaluation and result of a function
4 – 7	a0 – a3	Argument
8 – 15	t0 – t7	Temporary
16 – 23	s0 – s7	Saved temporary
24 – 25	t8 – t9	Temporary
26 – 27	k0 – k1	Reserved for OS kernel
28	gp	Pointer to global area
29	sp	Stack pointer
30	fp	Frame pointer
31	ra	Return address

9.3 ALPHA AXP – Instruction Formats

31 26	25 21	20 16	15 13	12	11 5	4 0

OPERATE (INTEGER, LITERAL)

OP	RA	LITERAL	1	FUNCTION	RC
6	5	8	1	7	5

OPERATE (INTEGER, REGISTER)

OP	RA	RB	/ / /	0	FUNCTION	RC
6	5	5	3	1	7	5

OPERATE (FLOATING POINT)

OP	RA	RB	FUNCTION	RC
6	5	5	11	5

MEMORY

OP	RA	RB	DISPLACEMENT
6	5	5	16

BRANCH

OP	RA	DISPLACEMENT
6	5	21

CALL_PAL

OP	FUNCTION
6	26

ALPHA AXP – Instruction Set (1)

Load/Store, Byte Manipulation (1)	
LDA	Load address
LDAH	Load address high
LDL	Load sign-extended longword
LDQ	Load quadword
LDQ_U	Load unaligned quadword
LDL_L	Load sign-extended longword, locked
LDQ_L	Load quadword, locked
STL_C	Store longword, conditional
STQ_C	Store quadword, conditional
STL	Store longword
STQ	Store quadword
STQ_U	Store unaligned quadword

Load/Store, Byte Manipulation (2)	
EXTBL	Extract byte low
EXTWL	Extract word low
EXTLL	Extract longword low
EXTQL	Extract quadword low
EXTWH	Extract word high
EXTLH	Extract longword h
EXTQH	Extract quadword high
INSBL	Insert byte low
INSWL	Insert word low
INSLL	Insert longword low
INSQL	Insert quadword low

ALPHA AXP – Instruction Set (2)

Load/Store, Byte Manipulation (3)	
INSWH	Insert word high
INSLH	Insert longword high
INSQH	Insert quadword high
MSKBL	Mask byte low
MSKWL	Mask word low
MSKLL	Mask longword low
MSKQL	Mask quadword low
MSKWH	Mask word high
MSKLH	Mask longword high
MSKQH	Mask quadword high

Floating Point Load/Store	
LDF	Load F format (VAX single)
LDG	Load G format (VAX double)
LDS	Load S format (IEEE single)
LDT	Load T format (IEEE double)
STF	Store F format (VAX single)
STG	Store G format (VAX double)
STS	Store S format (IEEE single)
STT	Store T format (IEEE double)

ALPHA AXP – Instruction Set (3)

Integer Computation and ... Move (1)		Integer Computation and … (2)	
ADDL	Add longword	MULQ	Multiply quadword
S4ADDL	Add longword, scale by 4	UMULH	Multiply quadword high, unsigned
S8ADDL	Add longword, scale by 8	SUBL	Subtract longword
ADDQ	Add quadword	S4SUBL	Subtract longword, scale by 4
S4ADDQ	Add quadword, scale by 4	S8SUBL	Subtract longword, scale by 8
S8ADDQ	Add quadword, scale by 8	SUBQ	Subtract quadword
CMPEQ	Compare signed quadword =	S4SUBQ	Subtract quadword, scale by 4
CMPLT	Compare signed quadword <	S8SUBQ	Subtract quadword, scale by 8
CMPLE	Compare signed quadword ≤	AND	AND logical
CMPULT	Compare unsigned quadword <	BIS	OR logical
CMPULE	Compare unsigned quadword ≤	XOR	Exclusive-OR logical
MULL	Multiply longword	BIC	AND-NOT logical

ALPHA AXP – Instruction Set (4)

Integer Computation and … (3)	
ORNOT	OR-NOT logical
EQV	Exclusive-OR-NOT logical
SLL	Shift left, logical
SRL	Shift right, logical
SRA	Shift right, arithmetic
CMOVEQ	Conditional move if reg = 0
CMOVNE	Conditional move if reg 6 ≠ 0
CMOVLT	Conditional move if reg < 0
CMOVLE	Conditional move if reg ≤ 0
CMOVGT	Conditional move if reg > 0
CMOVGE	Conditional move if reg ≥ 0
CMOVLBC	Conditional move if reg, low bit clear

Integer Computation and … (4)	
CMOVLBS	Conditional move if reg, low bit set
CMPBGE	Compare bytes, unsigned
ZAP	Clear selected bytes
ZAPNOT	Clear inselected bytes

ALPHA AXP – Instruction Set (5)

Integer Branch (1)	
BEQ	Branch if reg = 0
BNE	Branch if reg ≠ 0
BLT	Branch if reg < 0
BLE	Branch if reg ≤ 0
BGT	Branch if reg > 0
BGE	Branch if reg ≥ 0
BLBC	Branch if low bit clear
BLBS	Branch if low bit set
BR	Branch
BSR	Branch to subroutine
JMP	Jump
JSP	Jump to subroutine

Integer Branch (2)	
RET	Return from subroutine
JSR_COROUTINE	Jump to subroutine, return

Floating Point Bransch	
FBEQ	FP Branch if = 0
FBNE	FP Branch if ≠ 0
FBLT	FP Branch if < 0
FBLE	FP Branch if ≤ 0
FBGT	FP Branch if > 0
FBGE	FP Branch if ≥ 0

ALPHA AXP – Instruction Set (6)

FP Computation and Conditional	
CPYS	Copy sign
CPYSN	Copy sign, negate
CPYSE	Copy sign and exponent
CVTQL	Convert quadword to longword
CVTLQ	Convert longword to quadword
FCMOVEQ	FP conditional move if reg = 0
FCMOVNE	FP conditional move if reg 6 ≠ 0
FCMOVLT	FP conditional move if reg < 0
FCMOVLE	FP conditional move if reg ≤ 0
FCMOVGT	FP conditional move if reg > 0
FCMOVGE	FP conditional move if reg ≥ 0
MF_FPCR	Move from FP control register

FP Computation and … Move (2)	
MT_FPCR	Move to FP control register
ADDF	Add F format (VAX single)
ADDG	Add G format (VAX double)
ADDS	Add S format (IEEE single)
ADDT	Add T format (IEEE double)
CMPGEQ	Compare G format = (VAX double)
CMPGLT	Compare G format < (VAX double)
CMPGLE	Compare G format ≤ (VAX double)
CMPTEQ	Compare T format = (IEEE double)
CMPTLT	Compare T format < (IEEE double)
CMPTLE	Compare T format ≤ (IEEE double)

ALPHA AXP – Instruction Set (7)

FP Computation and … (3)	
CMPTUN	Compare T format unordered (IEEE double)
CVTGQ	Convert G format to quadword (VAX double)
CVTQF	Convert quadword to F format (VAX single)
CVTQG	Convert quadword to G format (VAX double)
CVTDG	Convert D to G format (VAX double/double)
CVTGD	Convert G to D format (VAX double/double)
CVTGF	Convert G to F format (VAX double/single)
CVTTQ	Convert T format to quadword (IEEE double)
CVTQS	Convert quadword to S format (IEEE single)
CVTQT	Convert quadword to T format (IEEE double)
CVTTS	Convert T to S format (IEEE double/single)
CVTST	Convert S to T format (IEEE single/double)

FP Computation and … (4)	
DIVF	Divide F format (VAX single)
DIVG	Divide G format (VAX double)
DIVS	Divide S format (IEEE single)
DIVT	Divide T format (IEEE double)
MULF	Multiply F format (VAX single)
MULG	Multiply G format (VAX double)
MULS	Multiply S format (IEEE single)
MULT	Multiply T format (IEEE double)
SUBF	Subtract F format (VAX single)
SUBG	Subtract G format (VAX double)
SUBS	Subtract S format (IEEE single)
SUBT	Subtract T format (IEEE double)

ALPHA AXP – Instruction Set (8)

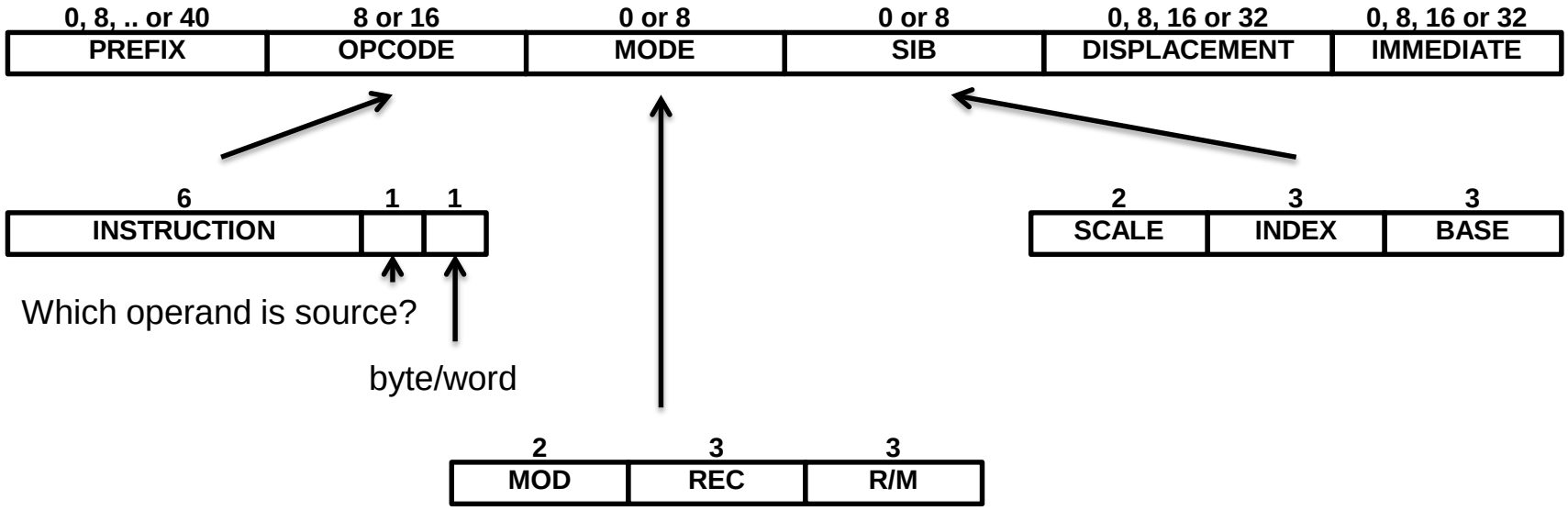
System (1)	
CALL_PAL	Call privileged architecture library
TRAPB	Trap barrier (precise exception)
FETCH	Prefetch (cache) data hint
FETCH_M	Prefetch (cache) data, modify hint
MB	Memory barrier (serialize)
WMB	Memory barrier (serialize) write
RPCC	Read process cycle counter
RC	Read and clear
RS	Read and set

System (2)	
PALRES0	PALcode reserved opcode 0
PALRES1	PALcode reserved opcode 1
PALRES2	PALcode reserved opcode 2
PALRES3	PALcode reserved opcode 3
PALRES4	PALcode reserved opcode 4

ALPHA AXP– Numeric Data Types

Type	8 Bits	16 Bits	32 Bits	64 Bits	128 Bits
Signed integer			X	X	
Unsigned integer					
Binary coded decimal integer					
Floating point			X	X	

9.4 Intel Pentium II – Instruction Formats



MODE	- specifier for the opcode
SIB	- (scale, index, base) special modes for further specification
DISPLACEMENT	- specifying a memory address (displacement)
IMMEDIATE	- containing a constant (immediate operand)

Intel Pentium II – Instruction Set (1)

A selection of integer instructions

Move	
MOV dst,src	Move SRC to DST
PUSH src	Push SRC onto the stack
POP dst	Pop a word from the stack to DST
XCHG ds1,ds2	Exchange DS1 and DS2
LEA dst,src	Load effective addr of SRC into DST
CMOV dst,src	Conditional move

Arithmetic	
ADD dst,src	Add SRC to DST
SUB dst,src	Subtract DST from SRC
MUL src	Multiply EAX by SRC (unsigned)
IMUL src	Multiply EAX by SRC (signed)
DIV src	Divide EDX:EAX by SRC (unsigned)
IDIV src	Divide EDX:EAX by SRC (signed)
ADC dst,src	Add SRC to DST, then add carry bit
SBB dst,src	Subtract DST & carry from SRC
INC dst	Add 1 to DST
DEC dst	Subtract 1 from DST
NEG dst	Negate DST (subtract it from 0)

Intel Pentium II – Instruction Set (2)

Binary coded decimal	
DAA	Decimal adjust
DAS	Decimal adjust for subtraction
AAA	ASCII adjust for addition
AAS	ASCII adjust for subtraction
AAM	ASCII adjust for multiplication
AAD	ASCII adjust for division

Boolean	
AND dst,src	Boolean AND SRC into DST
OR dst,src	Boolean OR SRC into DST
XOR dst,src	Boolean Exclusive-OR SRC to DST
NOT dst	Replace DST with 1s complement

Shift/Rotate	
SAL/SAR dst,#	Shift DST left/right # bits
SHL/SHR dst,#	Logical shift DST left/right # bits
ROL/ROR dst,#	Rotate DST left/right # bits
RCL/RCR dst,#	Rotate DST through carry # bits

Test/Compare	
TST src1,src2	Boolean AND operands, set flags
CMP src1,src2	Set flags based on SRC1 _SRC2

Intel Pentium II – Instruction Set (3)

Transfer of Control	
JMP addr	Jump to ADDR
Jxx addr	Conditional jumps based on flags
Call addr	Call procedure at ADDR
RET	Return from procedure
IRET	Return from interrupt
LOOPxx	Loop until condition met
INT addr	Initiate a software interrupt
INTO	Interrupt if overflow bit is set

Strings			
LODS	Load string	STOS	Store string
MOVS	Move string	SCAS	Scan strings
CMPS	Compare two strings		

Condition Codes	
STC	Set carry bit in EFLAGS register
CLC	Clear carry bit in EFLAGS register
CMC	Complement carry bit in EFLAGS reg
STD	Set direction bit in EFLAGS register
CLD	Clear direction bit in EFLAGS register
STI	Set interrupt bit in EFLAGS register
CLI	Set interrupt bit in EFLAGS register
PUSHFD	Push EFLAGS register onto stack
POPFD	Pop EFLAGS register from stack
LAHF	Load AH from EFLAGS register
SAHF	Store AH in EFLAGS register

Intel Pentium II – Instruction Set (4)

Transfer of Control	
SWAP dst	Change endianness of DST
CWQ	Extend EAX to EDX:EAX for division
CWDE	Extend 16-bit number in AX to EAX
ENTER size,lv	Create stack frame with SIZE bytes
LEAVE	Undo stack frame built by ENTER
NOP	No operation
HLT	Halt
IN al,port	Input a byte from PORT to AL
OUT port,al	Output a byte from AL to PORT
WAIT	Wait for an interrupt

src = source
dst = destination
# = shift/rotate count
lv = # locals

Intel Pentium II – Numeric Data Types/Addressing Modes

Type	8 Bits	16 Bits	32 Bits	64 Bits	128 Bits
Signed integer	X	X	X		
Unsigned integer	X	X	X		
Binary coded decimal integer	X				
Floating point			X	X	

Addressing Mode	
Immediate	X
Direct	X
Register	X
Register indirect	X
Indexed	X
Based-Indexed	
Stack	

9.5 UltraSPARC II – Instruction Formats

31 30	29	28 25	24 22	21 19	18 14	13	12 5	4 0

REMAINING (3 REGISTER)

	DEST	OPCODE	SRC1	0	FP-OP	SRC2
2	5	6	5	1	8	5

REMAINING (IMMEDIATE)

	DEST	OPCODE	SRC1	1	IMMEDIATE CONSTANT
2	5	6	5	1	13

SETHI

	DEST	OP	IMMEDIATE CONSTANT
2	5	3	22

BRANCH

	A	COND	OP	PC-RELATIVE DISPLACEMENT
2	1	4	3	22

CALL

	PC-RELATIVE DISPLACEMENT
2	30

UltraSPARC II – Instruction Set (1)

The primary integer instructions

Load/Store	
LDSB addr,dst	Load signed byte
LDUB addr,dst	Load unsigned byte
LDSH addr,dst	Load signed halfword
LDUH addr,dst	Load unsigned halfword
LDSW addr,dst	Load signed word
LDUW addr,dst	Load unsigned word
LDX addr,dst	Load extended (64 bits)
STB src,addr	Store byte
STH src,addr	Store halfword
STW src,addr	Store word
STX src,addr	Store extended

Arithmetic	
ADD r1,s2,dst	Add
ADDCC r1,s2,dst	Add and set icc
ADDC r1,s2,dst	Add with carry
ADDCCC r1,s2,dst	Add with carry and set icc
SUB r1,s2,dst	Subtract
SUBCC r1,s2,dst	Subtract and set icc
SUBC r1,s2,dst	Subtract with carry
SUBCCC r1,s2,dst	Sub with carry and set icc
MULX r1,s2,dst	Multiply
SDIVX r1,s2,dst	Signed divide
UDIVX r1,s2,dst	Unsigned divide
TADCC r1,s2,dst	Tagged add

UltraSPARC II – Instruction Set (2)

The primary integer instructions

Shift/Rotate	
SLL r1,s2,dst	Shift left logical (32 bits)
SLLX r1,s2,dst	Shift left logical extended (64 bits)
SRL r1,s2,dst	Shift right logical
SRLX r1,s2,dst	Shift right logical extended
SRA r1,s2,dst	Shift right arithmetic
SRAX r1,s2,dst	Shift right arithmetic extended

Boolean	
AND r1,s2,dst	Boolean AND
ANDCC r1,s2,dst	Boolean AND and set icc
ANDN r1,s2,dst	Boolean AND-NOT
ANDNCC r1,s2,dst	Boolean AND-NOT and set icc
OR r1,s2,dst	Boolean OR
ORCC r1,s2,dst	Boolean OR and set icc
ORN r1,s2,dst	Boolean OR-NOT
ORNCC r1,s2,dst	Boolean OR-NOT and set icc
XOR r1,s2,dst	Boolean Exclusive-OR
XORCC r1,s2,dst	Boolean Exclusive-OR and set icc
XNOR r1,s2,dst	Boolean Exclusive-OR-NOT
XNORCC r1,s2,dst	Boolean XOR-NOT and set icc

UltraSPARC II – Instruction Set (3)

The primary integer instructions

Transfer of Control	
BPcc addr	Branch with prediction
BPr scr,addr	Branch on register
CALL addr	Call procedure
RETURN addr	Return from procedure
JMPL addr,dst	Jump and link
SAVE r1,s2,dst	Advance register windows
RESTORE r1,s2,dst	Restore register windows
Tcc cc,trap#	Trap on condition
PREFETCH fcn	Prefetch data from memory
LDSTUB addr,r	Atomic load/store
MEMBAR mask	Memory barrier

Miscellaneous	
SETHI con,dst	Set bits 10 to 31
MOVcc cc,s2,dst	Move on condition
MOVr r1,s2,dst	Move on register
NOP	No operation
POPC r1,dst	Population count
RDCCR v,dst	Read condition code register
WRCCR r1,s2,v	Write condition code register
RDPC v,dst	Read program counter

UltraSPARC II – Instruction Set (4)

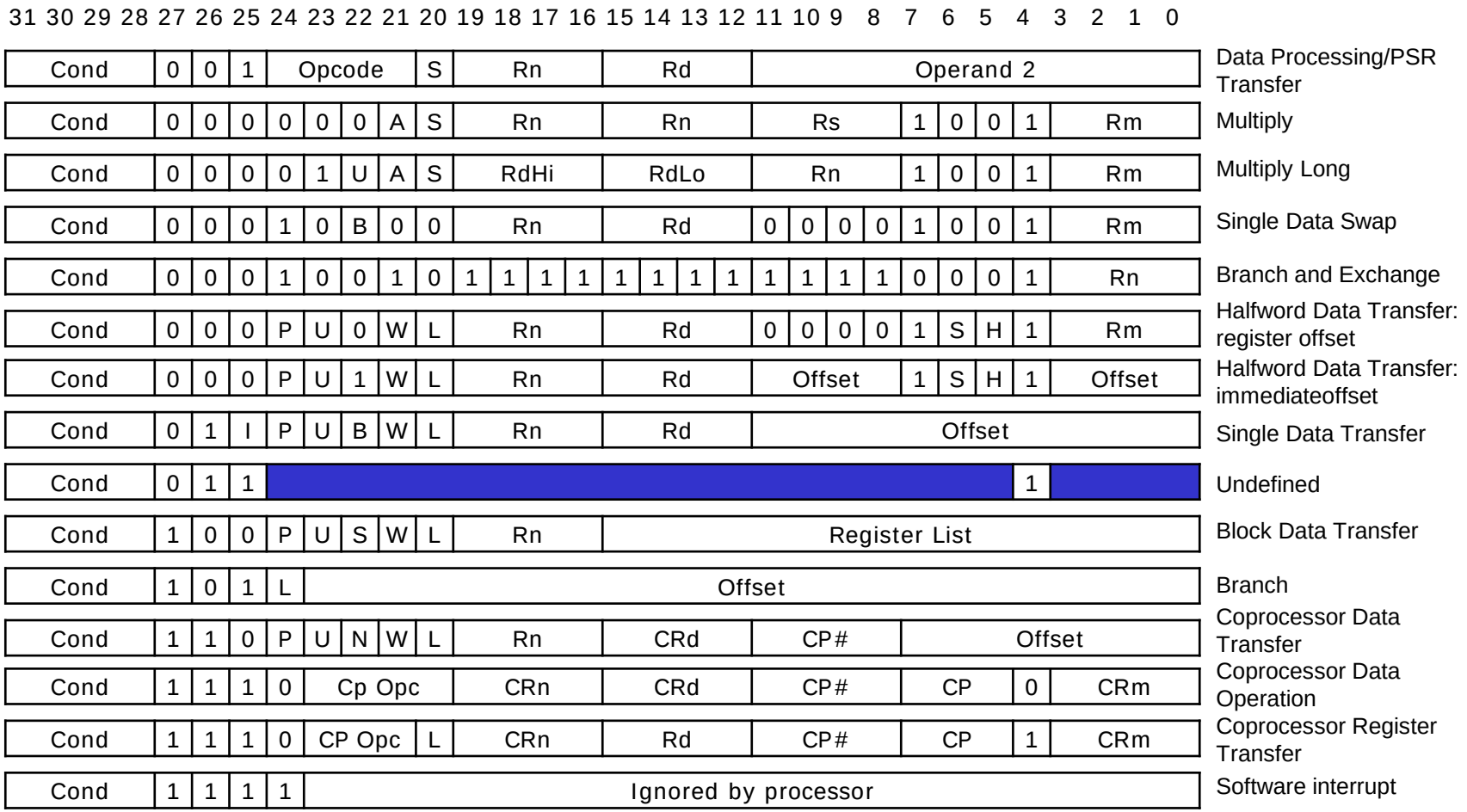
scr	= source register	mask	= operation type
dst	= destination register	con	= constant
r1	= source register	v	= register designator
s2	= Source: register or immediate	cc	= contition code
addr	= memory address	r	= destination register
trap#	= trap number	…cc	= condition
fcn	= function code	…r	= LZ, LEZ, Z, NZ, GZ, GEZ

UltraSPARC II – Numeric Data Types/Addressing Modes

Type	8 Bits	16 Bits	32 Bits	64 Bits	128 Bits
Signed integer	X	X	X	X	
Unsigned integer	X	X	X	X	
Binary coded decimal integer					
Floating point			X	X	X

Addressing Mode	
Immediate	X
Direct	
Register	X
Register indirect	
Indexed	X
Based-Indexed	X
Stack	

9.6 AT91 ARM – Instruction Formats



AT91 ARM – Instruction Set (1)

Load/Store/Move	
LDR	Load register from memory
LDM	Load multiple registers (Pop from stack)
LDC	Load coprocessor from memory
STR	Store register to memory
STM	Store multiple (Push into stack)
STC	Store coprocessor register to memory
MOV	Move register or constant
MCR	Move CPU register to coprocessor register
MRC	Move from coprocessor to CPU register
MRS	Move PSR status/flags to register
MSR	Move register to PSR status/flags
MVN	Move negative register

Arithmetic	
ADC	Add with carry
ADD	Add
RSB	Reverse Subtract
RSC	Reverse Subtract with Carry
SBC	Subtract with Carry
SUB	Subtract
MLA	Multiply Accumulate
MUL	Multiply

AT91 ARM – Instruction Set (2)

Boolean	
AND	AND
EOR	Exclusive OR
ORR	OR

Branch	
B	Branch
BL	Branch with Link
BX	Branch and Exchange

Miscellaneous	
CMP	Compare
CMN	Compare Negative
TST	Test bits
TEQ	Test bitwise equality
BIC	Bit Clear
SWP	Swap register with memory
SWI	Software Interrupt
CDP	Coprocessor Data Processing

9.7 AMD Athlon (x86) – Instruction Set (1)

Integer Instructions (1)				
AAA	BT	CMOVAE/CMOVNB/CMOVNC	CMOVO	DAS
AAD	BTC	CMOVB/CMOVC/CMOVNAE	CMOVP/CMOVPE	DEC
AAM	BTR	CMOVBE/CMOVNA	CMOVS	DIV
AAS	BTS	CMOVE/CMOVZ	CMP	ENTER
ADC	CALL	CMOVG/CMOVNLE	CMPSB	IDIV
ADD	CBW/CWDE	CMOVGE/CMOVNL	CMPSW	IMUL
AND	CLC	CMOVL/CMOVNGE	CMPSD	IN
ARPL	CLD	CMOVLE/CMOVNG	CMPXCHG	INC
BOUND	CLI	CMOVNE/CMOVNZ	CMPXCHG8B	INVD
BSF	CLTS	CMOVNO	CPUID	INVLPG
BSR	CMC	CMOVNP/CMOVPO	CWD/CDQ	JO
BSWAP	CMOVA/CMOVNBE	CMOVNS	DAA	JNO

AMD Athlon (x86) – Instruction Set (2)

Integer Instructions (2)

JB/JNAE/JC	JLE/JNG	LGS	LTR	OR	RDTSC
JNB/JAE/JNC	JNLE/JG	LIDT	MOV	OUT	RET
JZ/JE	JCXZ/JEC	LLDT	MOVSB	POP	ROL
JNZ/JNE	JMP	LMSW	MOVSD	POPA/POPAD	ROR
JBE/JNA	LAHF	LODSB AL	MOVSW	POPF/POPFD	SAHF
JNBE/JA	LAR	LODSW AX	MOVSX	PUSH	SAR
JS	LDS	LODSD EAX	MOVZX	PUSHA/PUSHAD	SBB
JNS	LEA	LOOP	MUL	PUSHF/PUSHFD	SCASB
JP/JPE	LEAVE	LOOPE/LOOPZ	MULEAX	RCL	SCASW
JNP/JPO	LES	LOOPNE/LOOPNZ	NEG	RCR	SCASD
JL/JNGE	LFS	LSL	NOP	RDMSR	SETO
JNL/JGE	LGDT	LSS	NOT	RDPMC	SETNO

AMD Athlon (x86) – Instruction Set (3)

Integer Instructions (3)			
SETB/SETC/SETNAE	SETLE/SETNG	STI	VERW
SETAE/SETNB/SETNC	SETG/SETNLE	STOSB	WAIT
SETE/SETZ	SGDT	STOSW	WBINVD
SETNE/SETNZ	SIDT	STOSD	WRMSR
SETBE/SETNA	SHL/SAL	STR	XADD
SETA/SETNBE	SHR	SUB	XCHG
SETS	SHLD	SYSCALL	XLAT
SETNS	SHRD	SYSENTER	XOR
SETP/SETPE	SLDT	SYSEXIT	
SETNP/SETPO	SMSW	SYSRET	
SETL/SETNGE	STC	TEST	
SETGE/SETNL	STD	VERR	

AMD Athlon (x86) – Instruction Set (4)

MMX Instructions			
EMMS	PADDW	POR	PSUBSW
MOVD	PAND	PSLLD	PSUBUSB
MOVQ	PANDN	PSLLQ	PSUBUSW
PACKSSDW	PCMPEQB	PSLLW	PSUBW
PACKSSWB	PCMPEQD	PSRAW	PUNPCKHBW
PACKUSWB	PCMPEQW	PSRAD	PUNPCKHDQ
PADDB	PCMPGTB	PSRLD	PUNPCKHWD
PADDD	PCMPGTD	PSRLQ	PUNPCKLBW
PADDSB	PCMPGTW	PSRLW	PUNPCKLDQ
PADDSW	PMADDWD	PSUBB	PUNPCKLWD
PADDUSB	PMULHW	PSUBD	
PADDUSW	PMULLW	PSUBSB	

10 Zusammenfassung

- Grundbegriffe zum Befehlssatz.
- Unterteilung der Befehlssatzarchitekturen in CISC und RISC Computer.
- Klassifizierung der Befehlssatzarchitekturen nach verschiedenen Kriterien.
- 1-, 2-, und 3-Adressmaschinen: Stack-, ACCU- und GRP-Architektur.
- Verschiedene Zugriffsmethoden auf den Hauptspeicher: Byte-Reichenfolge und Speicher-Ausrichtung.
- 1-, 2-, und 3-Adressbefehle: überdeckt, implizit.
- Verschiedene Adressierungsmethoden: implizit, direkt, indirekt, relativ, indiziert, segmentiert, virtuell.
- Unterteilung der Befehlstypen: Datenübertragung, Datenmanipulation, Verzweigungsbefehle, Systembefehle.