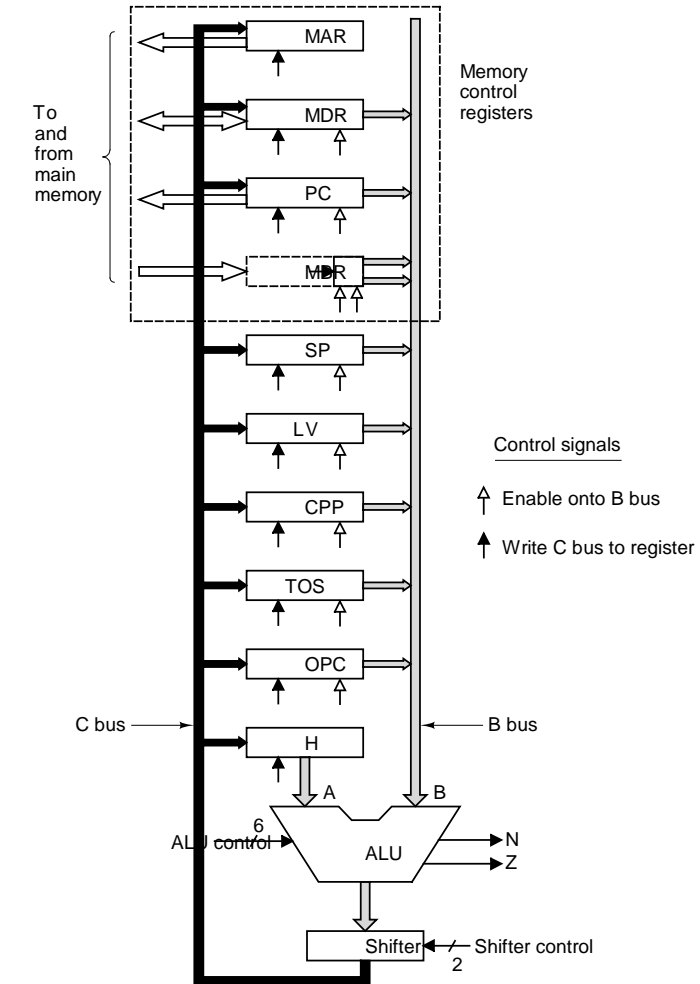


TI II: Computer Architecture Microarchitecture

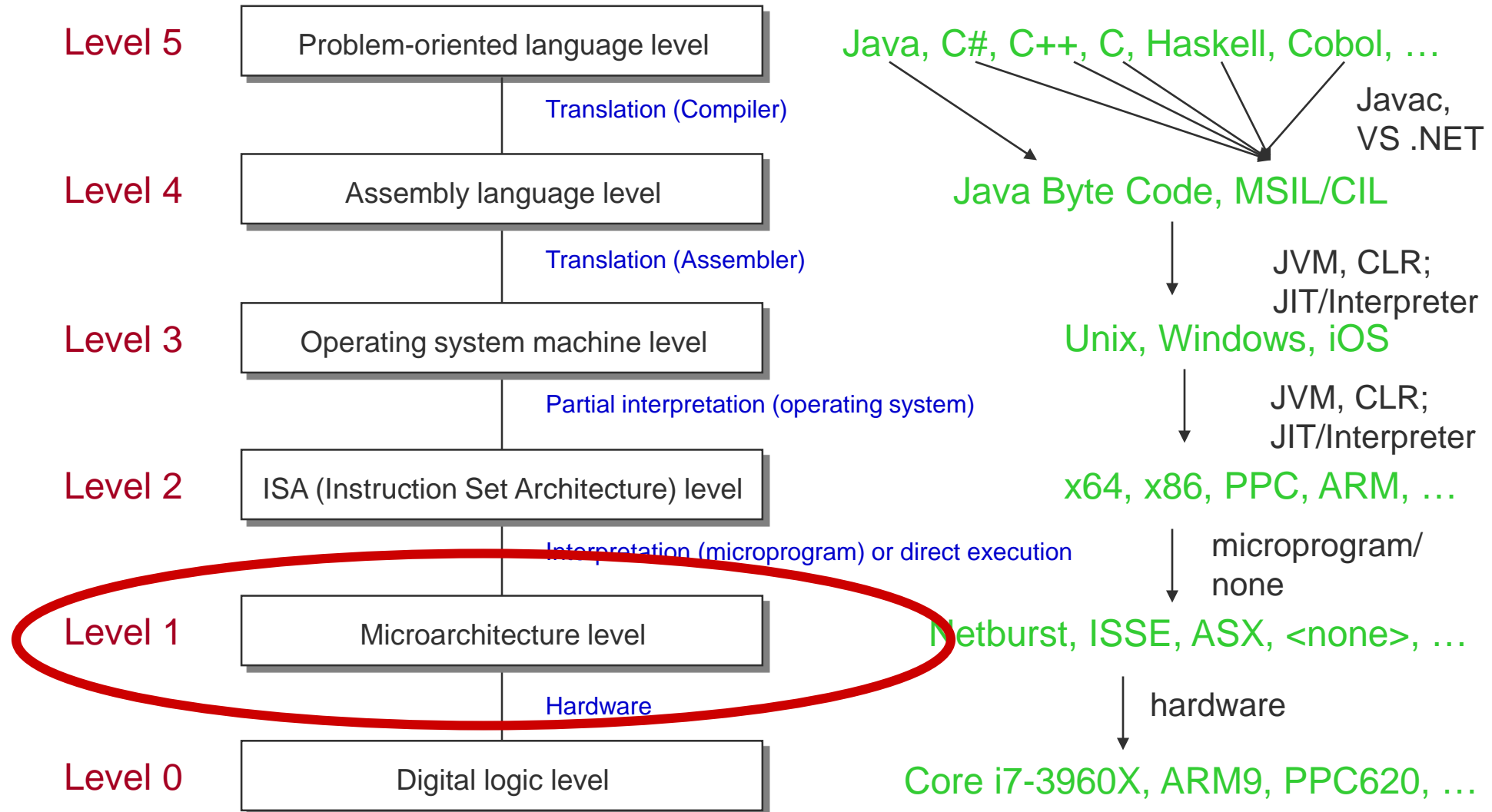
Microprocessor Architecture

Microprogramming

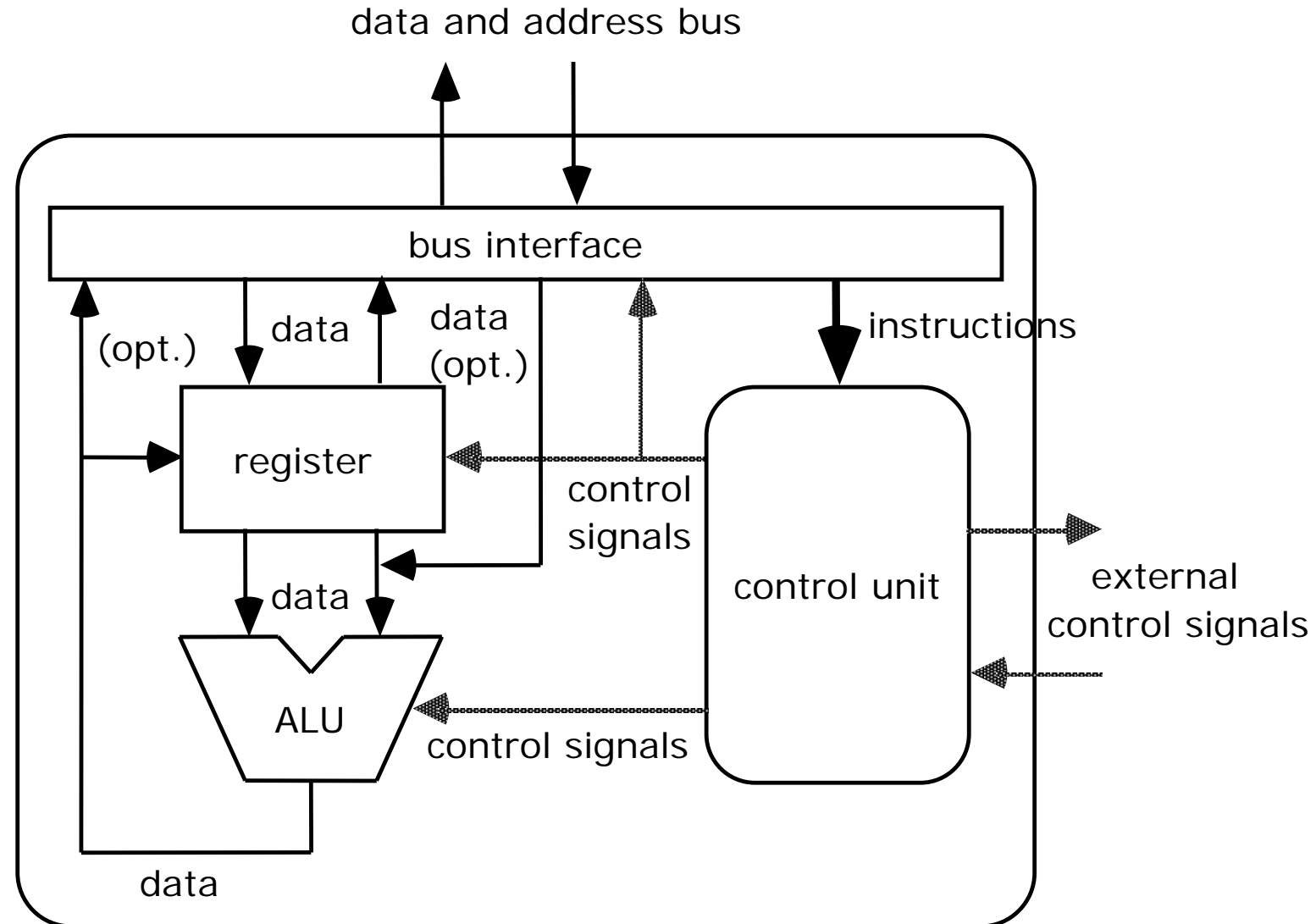
Pipelining (superscalar, multithreaded, hazards,
prediction, vector processing)



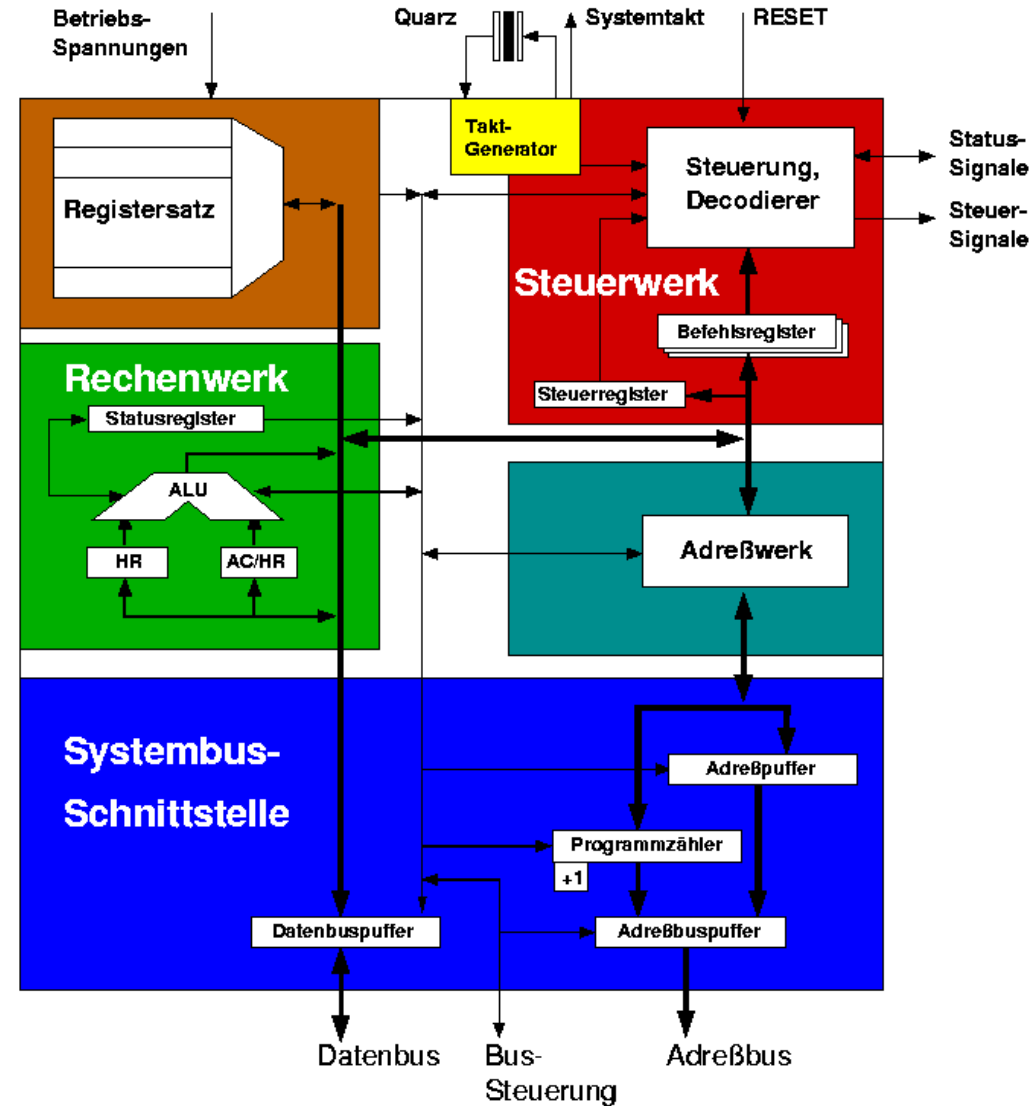
Where are we now?



Basic architecture of a simple micro processor



Interner Aufbau eines einfachen μP



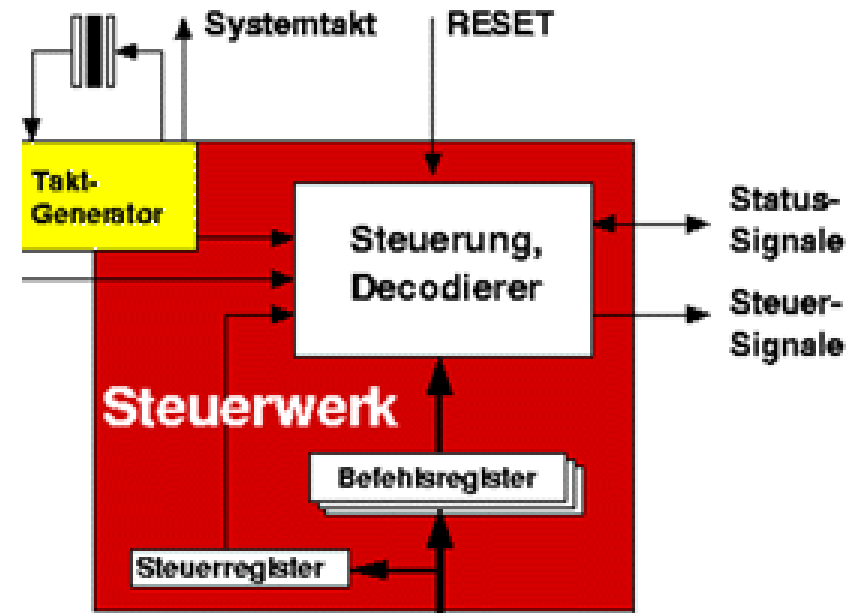
STEUERWERK

Steuert die Systemkomponenten

Befehlsregister enthält den gerade ausgeführten Befehl

Dekoder (mikroprogrammiertes Schaltwerk) wird von den Statussignalen beeinflusst und erzeugt die Steuersignale

Taktgenerator erzeugt den vom externen Quartz festgelegten Systemtakt



Steuerwerk (Control Unit)

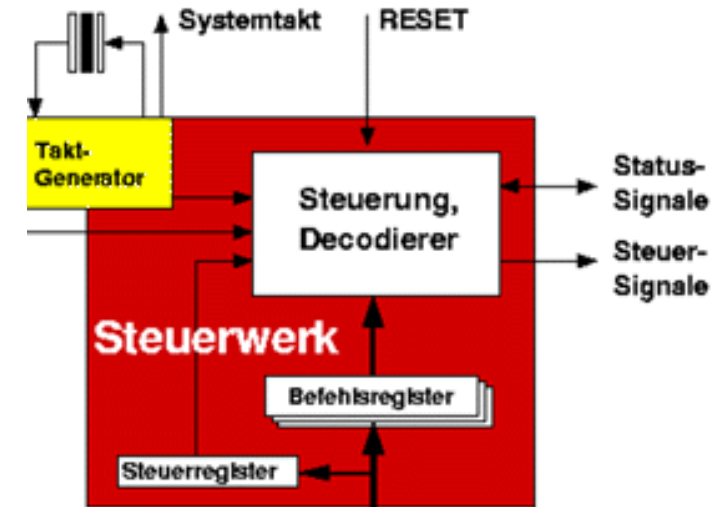
Synchrones Schaltwerk

- Meist liegt ein sog. **dynamisches Schaltwerk** vor.
- Die Zustandsinformation ist nicht in Flipflops, sondern in Kondensatoren gespeichert.

- Mindesttaktfrequenz ist erforderlich

- Unterhalb dieser Taktfrequenz gehen die Inhalte der aus Kondensatoren bestehenden Zustandsregister durch Leckströme bereits vor dem nächsten Taktzyklus verloren.

Taktgenerator on Chip, meist mit externem Quartz verbunden



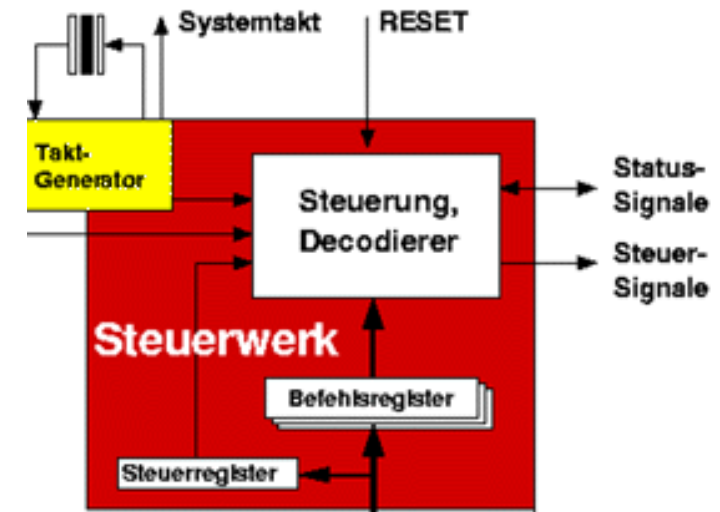
Taktgenerator

Aufgaben

- Taktfrequenz(en) herstellen
- Erzeugung eines mit dem Prozessortakt synchronisierten Rücksetzsignals

Beim Rücksetzen durchläuft das Steuerwerk eine Initialisierungsroutine

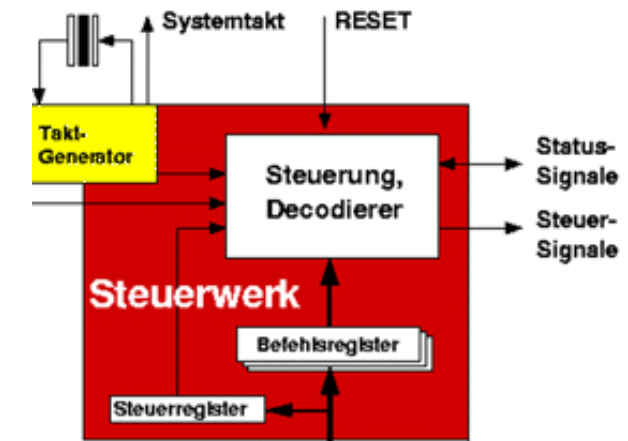
- Diese wird bei vielen Mikroprozessoren ausgeführt, während das Rücksetzsignal aktiv ist
- Deshalb muss das Rücksetzsignal genauen zeitlichen Spezifikationen genügen.



Steuerwerk

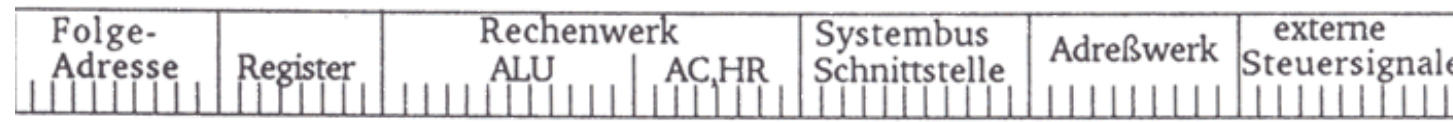
Mikroprogrammsteuerwerk

Im Festwertspeicher liegt für jeden Befehl ein Mikroprogramm



Mikroprogramm \equiv Folge von Mikrobefehlen

Aufbau eines Mikrobefehls:



Einzelne Bits eines Mikrobefehls \equiv Mikrooperationen \equiv Auswahl- und Freigabesignale für die benötigten Komponenten

Phasen der Befehlsausführung

Holphase

- den nächsten Befehl in das Befehlsregister laden

Decodierphase

- der Befehlsdecoder ermittelt die Startadresse des Mikroprogramms, welches den Befehl ausführt

Ausführungsphase

- das Mikroprogramm steuert die Befehlsausführung, indem es entsprechende Signalfolgen an die anderen Prozessorkomponenten übermittelt und Meldesignale auswertet

Steuerwerk (Control Unit)

Das Befehlsregister besteht aus mehreren Registern, da

unterschiedlich lange Befehlsformate

- verschiedene Befehle sind unterschiedlich lang (1-Wort-Befehle, 2-Wort-Befehle, 3-Wort-Befehle, ...)

Vorabladen von Befehlen (Opcode-Prefetching)

- zur Steigerung der Verarbeitungsgeschwindigkeit werden bereits mehrere folgende Befehle in das Befehlsregister geladen, während der aktuelle Befehl gerade dekodiert wird
- Opcode prefetch queue, Warteschlange, Pipelining

Steuerwerk (Control Unit)

Für jeden Befehl liegt ein Mikroprogramm im Festwertspeicher des Steuerwerks vor

Mikroprogramme können vom Benutzer nicht verändert werden

- ➔ Das Steuerwerk eines Standard-Mikroprozessors ist mikroprogrammiert, jedoch meist nicht (vom Benutzer) mikroprogrammierbar

Alternative

- Steuerwerk als festverdrahtetes Schaltwerk (reine RISC-Prozessoren)

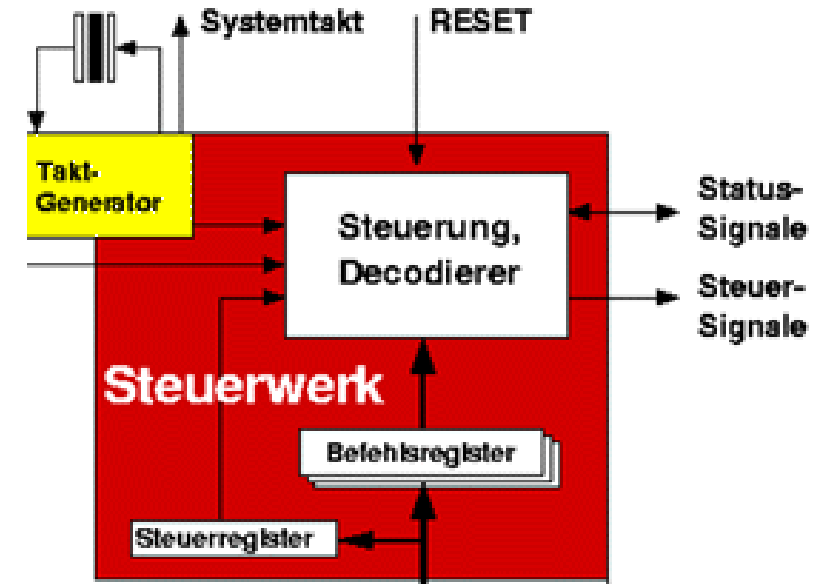
Das Steuerregister

Mit Hilfe des **Steuerregisters** kann die aktuelle Arbeitsweise des Steuerwerks beeinflusst werden

Die Bedeutung der Bits des Steuerregisters hängen vom jeweiligen Prozessor ab

Beispiele:

- Interrupt enable Bit:
 - bestimmt, ob auf eine Unterbrechungs-Anforderung am INT-Eingang reagiert wird



Das Steuerregister

Beispiele (Fortsetzung):

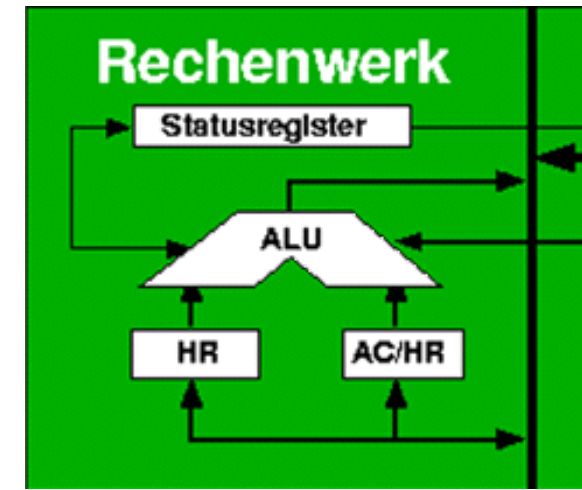
- User/System Bit
 - bestimmt ob der Prozessor im User-Modus (nur beschränkter Teil des Befehlsvorrats nutzbar) oder im Systemmodus (alle Befehle verfügbar, i.Allg. für das Betriebssystem reserviert) arbeitet
- Trace Bit
 - erlaubt Befehlsabarbeitung im Einzelschritt (Single Step Mode), d.h. nach jeder Befehlsausführung wird eine Unterbrechungsroutine gestartet → Debugging
- Decimal Bit
 - entscheidet, ob Dual oder BCD gerechnet wird

RECHENWERK

Operationswerk (Rechenwerk)

Führt die vom Steuerwerk verlangten logischen und arithmetischen Operationen aus

Statusregister informiert das Steuerwerk über den Ablauf des Ergebnisses (z.B. Carry, Overflow, Zero, Sign, ...)

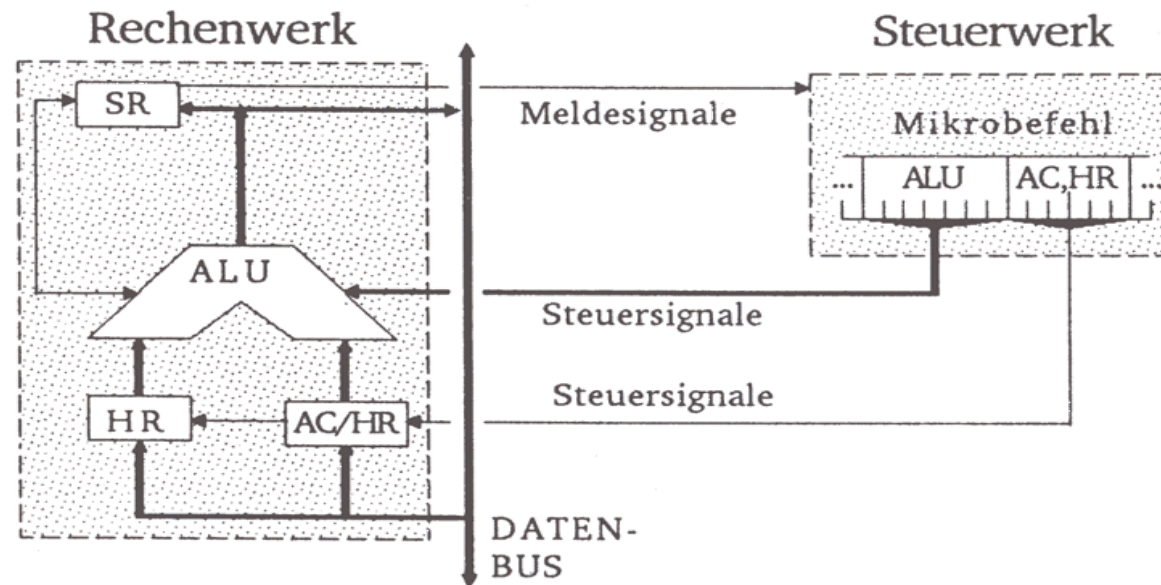


Zum Zwischenspeichern von Operanden und Ergebnissen sind **Hilfsregister** und **Akkumulatoren** vorhanden

Rechenwerk (Operationswerk, Execution Unit)

Kernstück: ALU

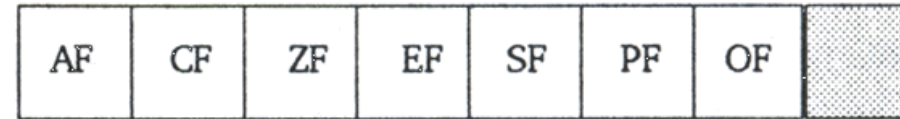
AC: Akkumulator
HR: Hilfsregister
SR: Statusregister



Das Rechenwerk führt alle logischen und arithmetischen Operationen im Prozessor aus.

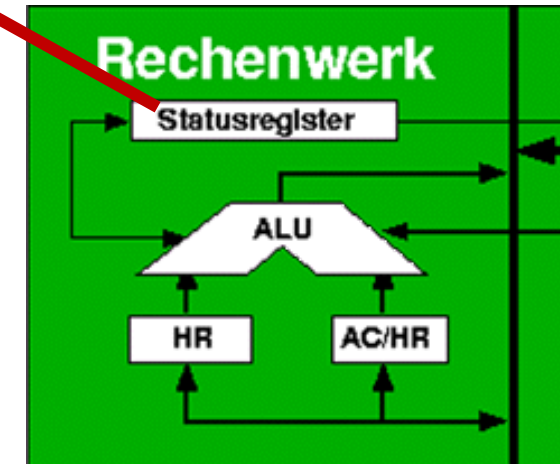
Statusregister (Zustandsregister, Condition Code Register CCR)

Einzelne Bits, die das Ergebnis einer arithmetischen Operation widerspiegeln, werden im Statusregister gespeichert



Das Statusregister enthält meist folgende Bits (Flags):

- Hilfsübertragsbit (Auxiliary Carry, AF)
- Übertragsbit (Carry Flag, CF)
- Nullbit (Zero Flag, ZF)
- Geradezahl (Even Flag, EF)
- Vorzeichenbit (Sign Flag, SF)
- Paritätsbit (Parity Flag, PF)
- Überlaufbit (Overflow Flag, OF)



Bedeutung der Statusflags

Carry Flag(CF)

- Übertrag aus dem höchstwertigsten Bit bei Addition oder Subtraktion (Borrow).
- ➔ sequentielle Addition und Subtraktion mit größerer Wortbreite ist möglich.

Aux Carry (AF)

- Übertrag von Bit 3 in Bit 4 des Ergebnisses.
- Wird für BCD-Arithmetik benötigt.

Zero Flag (ZF)

- Zeigt an, ob das Ergebnis der letzten Operation gleich 0 war.
- Wird für bedingte **Programmverzweigungen** und insbesondere **Zählschleifen** benötigt.

Bedeutung der Statusflags

Sign Flag (SF)

- Zeigt an, dass das Ergebnis negativ ist (Most Significant Bit = 1). Spiegelt das höchstwertige Bit eines Operanden wider.
- Wird für bedingte Programmverzweigungen benötigt

Overflow Flag

- Zeigt Bereichsüberschreitung im Zweierkomplement (z.B. bei Addition oder Subtraktion) an.

Even Flag (EV)

- Zeigt an, ob das Ergebnis eine gerade Zahl ist

Parity Flag (PF)

- Signalisiert ungerade Parität des Ergebnisses, d.h. eine ungerade Anzahl von Einsen.

Statusregister (Zustandsregister, Condition Code Register CCR)

Werte von Statusbits können direkt Einfluss auf die Ausführung des Mikroprogramms haben

➡ bedingte Programmverzweigung

Statusregister und Steuerregister werden häufig zur Erleichterung der Adressierung zusammengefasst betrachtet und manipuliert - und meist Prozessorstatuswort (PSW) genannt

Operationsvorrat einer ALU

Arithmetische Operationen

- Addieren ohne/mit Übertrag
- Subtrahieren ohne/mit Übertrag
- Inkrementieren/Dekrementieren
- Multiplizieren ohne/mit Vorzeichen
- Dividieren ohne/mit Vorzeichen
- Komplementieren (Zweierkomplement)

Logische bitweise Verknüpfungen

- Negation
- UND
- ODER
- Antivalenz / XOR

Operationsvorrat einer ALU

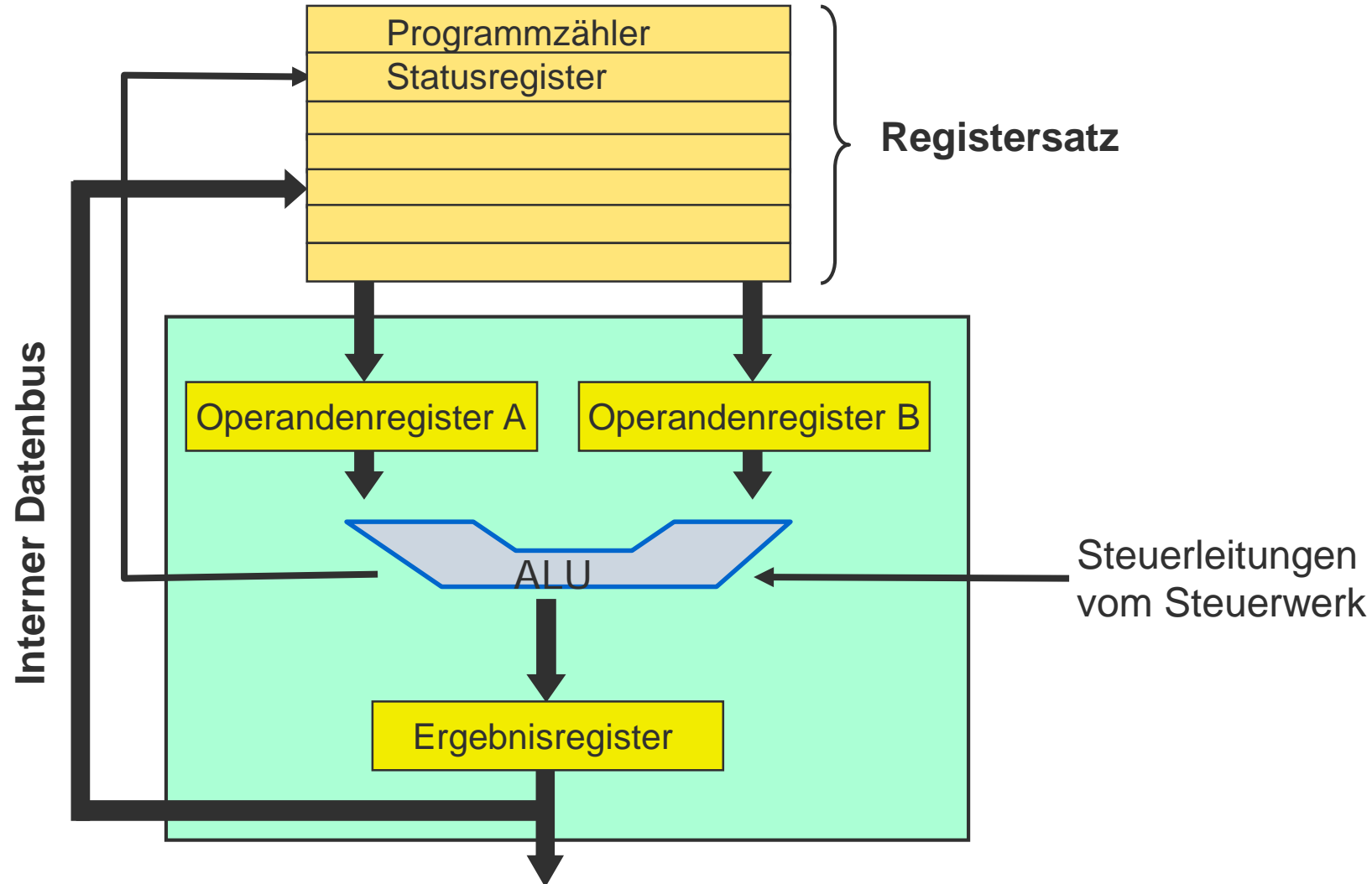
Schiebe- und Rotations-Operationen

- Links-Verschieben
- Rechts-Verschieben
- Links-Rotieren ohne Übertragsbit
- Links-Rotieren durchs Übertragsbit
- Rechts-Rotieren ohne Übertragsbit
- Rechts-Rotieren durchs Übertragsbit

Transport-Operationen

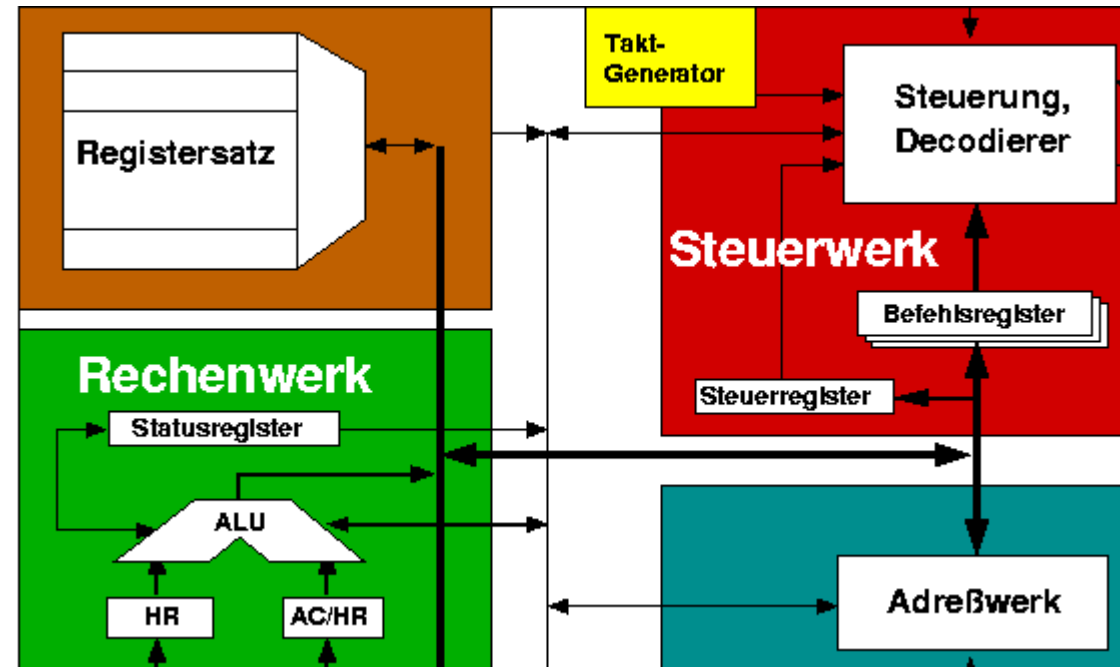
- Transferieren

Wiederholung: Rechenwerk



REGISTERSATZ

Registersatz



Erweiterung des Operationswerks:

Häufig benutzte Operanden können dort zwischengespeichert werden

➔ schnellerer Zugriff als auf den Hauptspeicher

Registersatz

Register

- Speicherzellen mit kleiner Zugriffszeit (wenige ns, ps)
- Auswahl einzelner Register durch individuelle Steuerleitungen
- kleine Register-Anzahl
 - ➔ keine Adressdecoder erforderlich
 - ➔ Zeit der Adressdecodierung entfällt
- Register sind auf dem Prozessorchip untergebracht
 - ➔ zeitraubendes Umschalten auf externe Daten- und Adresswege entfällt

Registersatz

Getrennte Ein-/Ausgänge

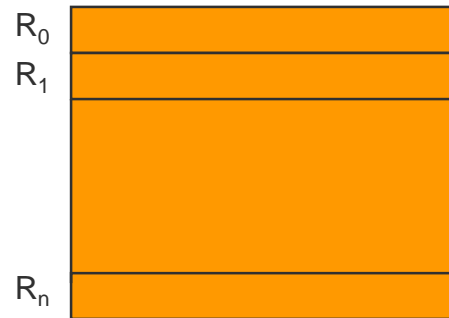
- ➔ Dual Port Speicher, zwischen Eingangs- und Ausgangsbuss gehängt
- ➔ Schreiben eines Registers und gleichzeitiges Lesen eines anderen Registers möglich
- Heutige superskalare Prozessoren: z.B. pro Takt 4 allgemeine Register schreiben und bis zu 8 allgemeine Register lesen

oft dynamische Speicherzellen ➔ Refresh erforderlich

Register mit Zusatzfunktionen

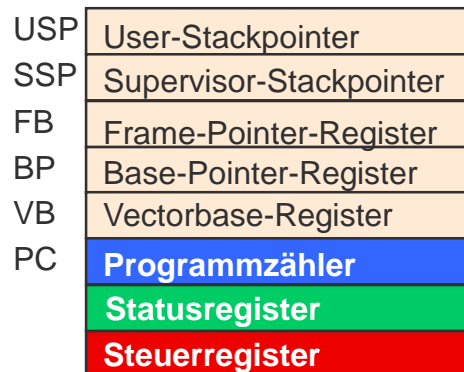
- Inkrementieren/Dekrementieren
- auf Null setzen
- Inhalt verschieben

Registersatz (Beispiel)



universelle, allgemeine Register
(*general purpose register*)

Daten- und Adressregister



Spezialregister
(*special purpose register*)

Registersatz

Daten- und Adressregister

Datenregister

- Zwischenspeichern von Operanden
- schneller Zugriff auf häufig benutzte Operanden
- bei modernen Prozessoren sind mehrere Datenregister als **Akkumulator** nutzbar

Adressregister

- Speichern von Adressen (oder Teile davon) eines Operanden im Hauptspeicher
- Basisregister
- Indexregister

Spezialregister (special purpose register)

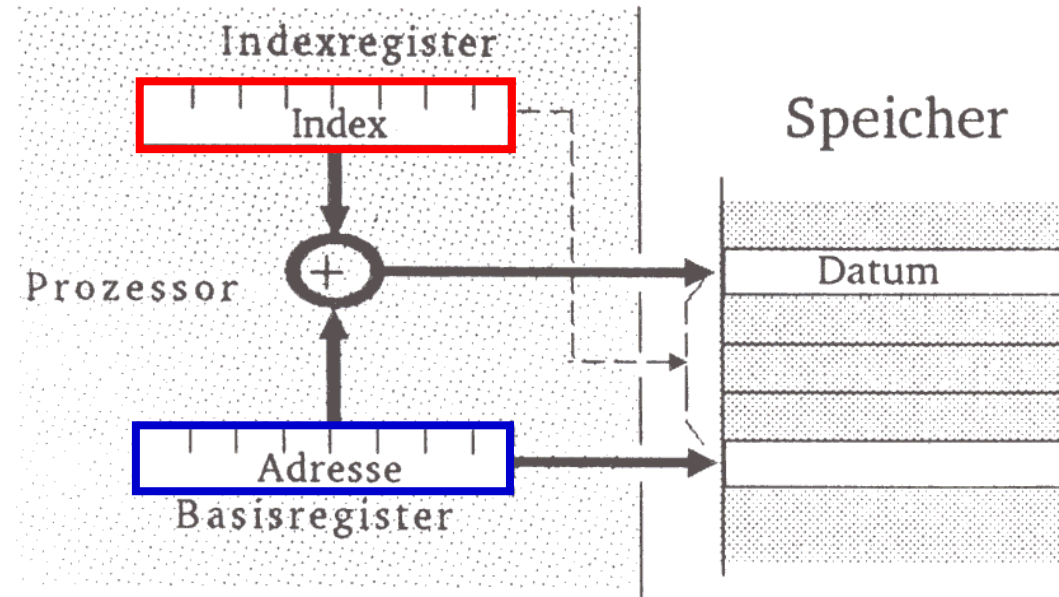
Register zur speziellen Verwendung:

- Programmzähler (instruction pointer)
- Steuerregister
- Statusregister
- Register für den Start von Interrupt-Behandlungen (interrupt vector base register)
- Stackregister (user und supervisor Stackpointer)

Funktion von Basis- und Indexregister

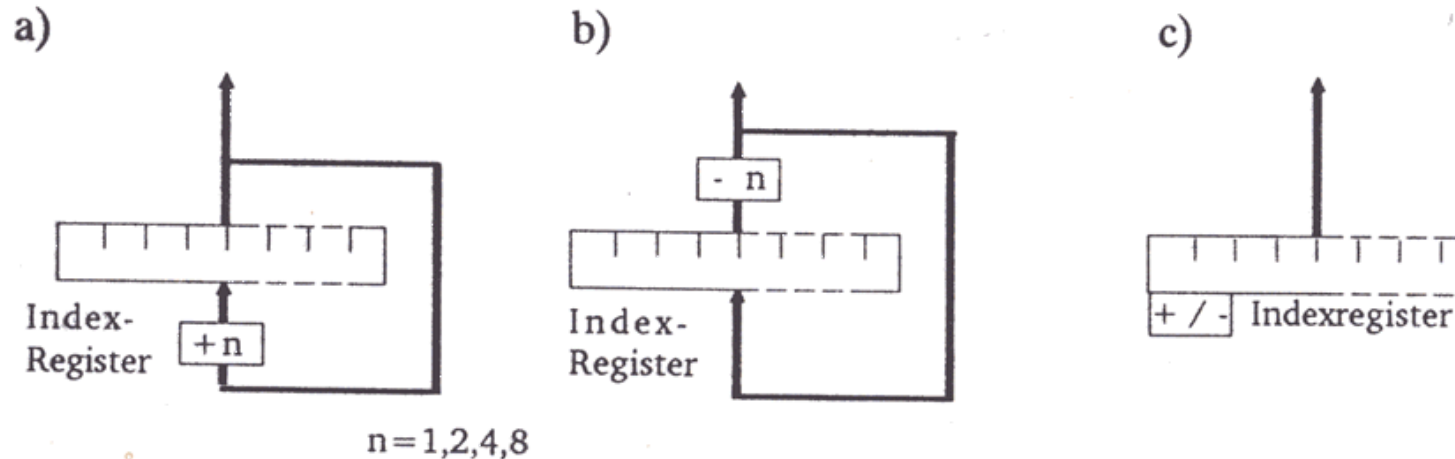
Basisregister enthält die Anfangsadresse eines Speicherbereichs.

Diese bleibt während der Bearbeitung des Speicherbereichs unverändert.



Indexregister enthält eine Distanz (Offset, Displacement) zu einer Basisadresse und dient zur Auswahl eines bestimmten Datums des Speicherbereichs.

Automatische Modifikation von Indexregistern



a) Post-Inkrement:

automatische Erhöhung des Registerwerts um $+n$ nach Adressierung einer Speicherzelle

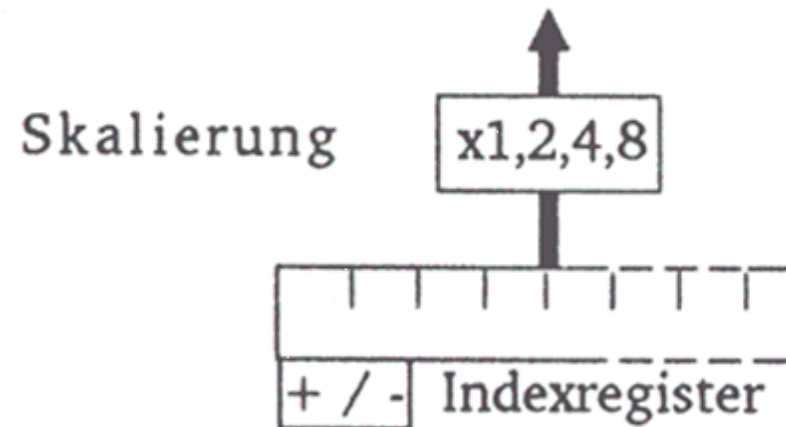
b) Pre-Dekrement:

automatische Erniedrigung des Registerwerts um $-n$ vor Adressierung einer Speicherzelle

c) Auto-Inkrement / Auto-Dekrement Register

Register mit Skalierung

Das Indexregister wird vor der Auswertung je nach aktueller Datenlänge (1 Byte, 2 Byte, 4 Byte, 8 Byte) mit dem Faktor 1, 2, 4 oder 8 multipliziert.



Vorteil:

bessere Ausnutzung der Registerbreite, da das Register selbst nur noch um 1 inkrementiert bzw. dekrementiert werden muss.

Der (Laufzeit-)Stack „Kellerspeicher“

Ein besonderer Speicherbereich, der normalerweise im Arbeitsspeicher angelegt ist (software stack) und der nach dem Kellerprinzip (LIFO, Last-In-First-Out) organisiert ist

Funktion:

- Abspeichern des Prozessorstatus und des Programmzählers beim Unterprogrammaufruf und Aufruf von Unterbrechungs-Routinen
- Parameterübergabe
- Kurzzeitige Lagerung von Daten bei der Ausführung

Bei modernen Prozessoren häufig mehrere getrennte Stackspeicher: System Stack, User Stack, Data Stack

Hardware-Unterstützung des Stacks

Stackregister (Stapelzeiger, Stack Pointer, SP):

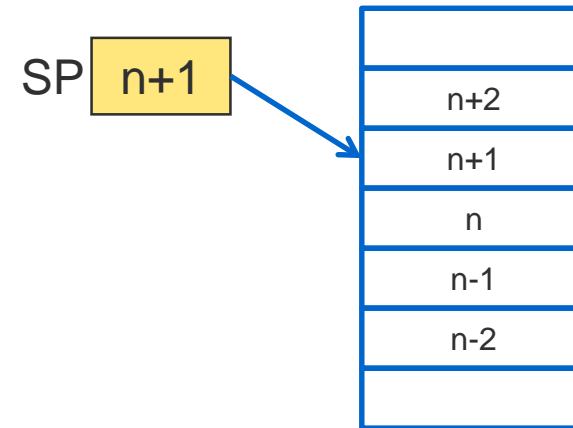
enthält die Adresse des zuletzt in den Stack eingetragenen Datums

Spezielle Befehle zur Datenübertragung in den bzw. aus dem Stack

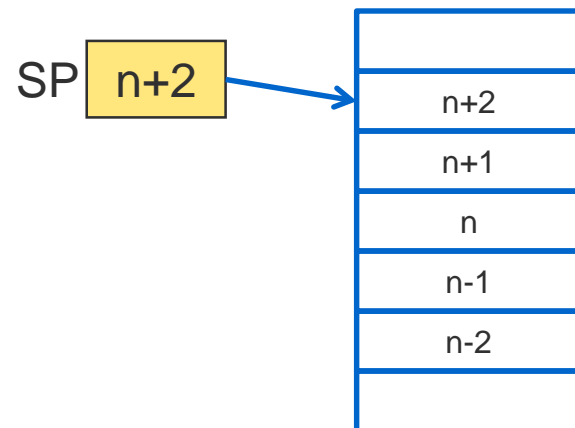
- PUSH
 - Inhalt eines Registers wird in den Stack übertragen
- POP (PULL)
 - Inhalt eines Registers wird vom Stack geladen

Verwaltung des Stackregisters

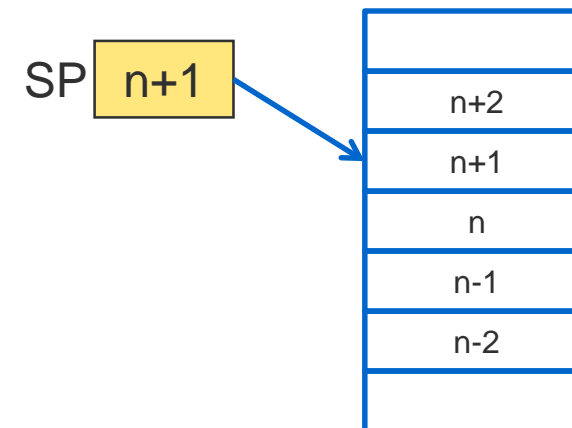
Initialzustand
Stackpointer SP zeigt auf
 $n+1$



Push-Operation

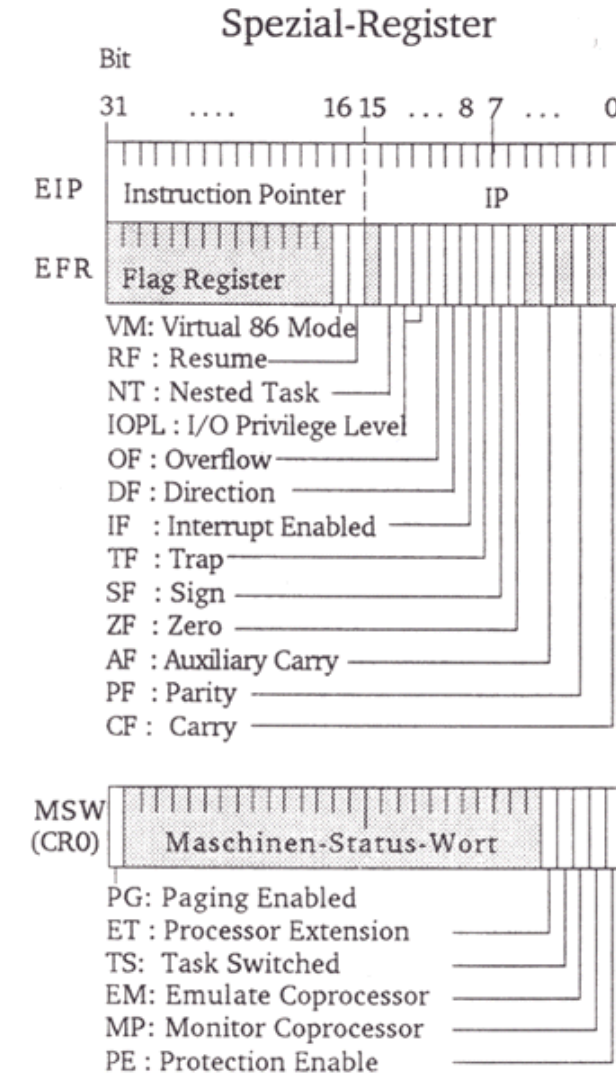
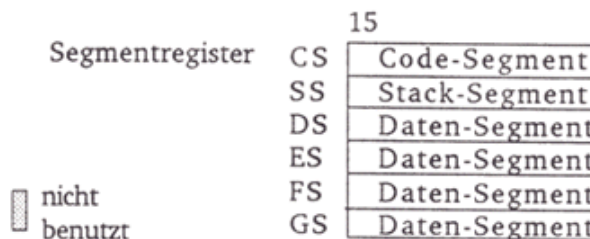
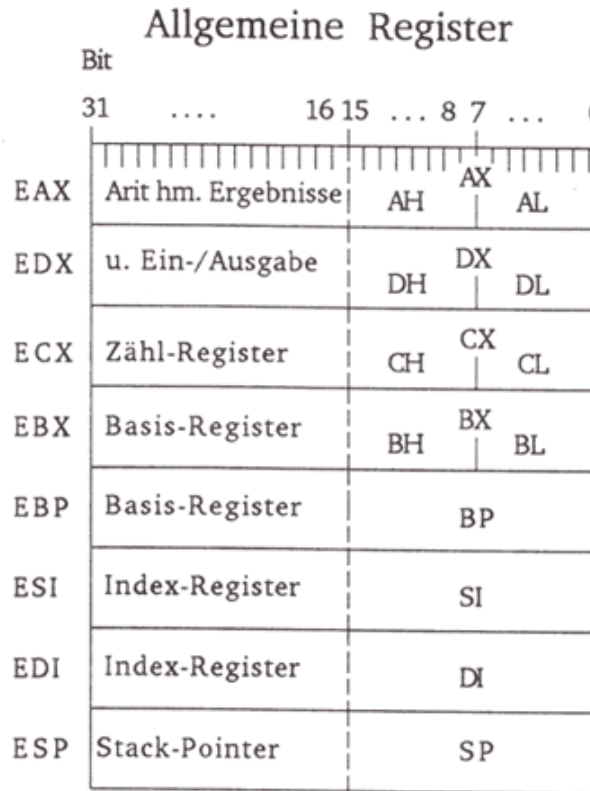


POP-Operation



Klassisches Programmiermodell des Intel 80386/80486

Programmiermodell:
die benutzer-
zugänglichen Register



Programmiermodell des Intel 80386/80486

Der Registersatz ist eine Aufwärts-Entwicklung der Registersätze von 8080 und 8086:

Register des 8080:

8 Bit Register	16 Bit Register
AL, DL, DH (DX) CL, CH (CX) BL, BH (BX)	SP, IP

Einige 8-bit-Register zusammengefasst zu 16-bit-Registern (DX, CX, BX)

AL mit dem Statusregister zu einem 16-bit-Register zusammengefasst (PSW, Processor Status Word)

Programmiermodell des Intel 80386/80486

Register des 8086

- Alle Register jetzt 16-bit-Register (AX, DX, CX, BX, SP, IP)
- zusätzliche 16-bit-Register BP, SI, DI
- zusätzliche 8-bit-Register CS, DS, SS

Aus Kompatibilitätsgründen und zur Unterstützung Byte-orientierter Probleme sind die Register AX, DX, CX und BX weiterhin byteweise ansprechbar

Die zusätzlichen 8-bit-Register CS, DS und SS dienen als Segmentregister zusammen mit SP und IP, sowie CX und DX zur Erweiterung des Adressraums auf 24 bit (16 Mbyte)

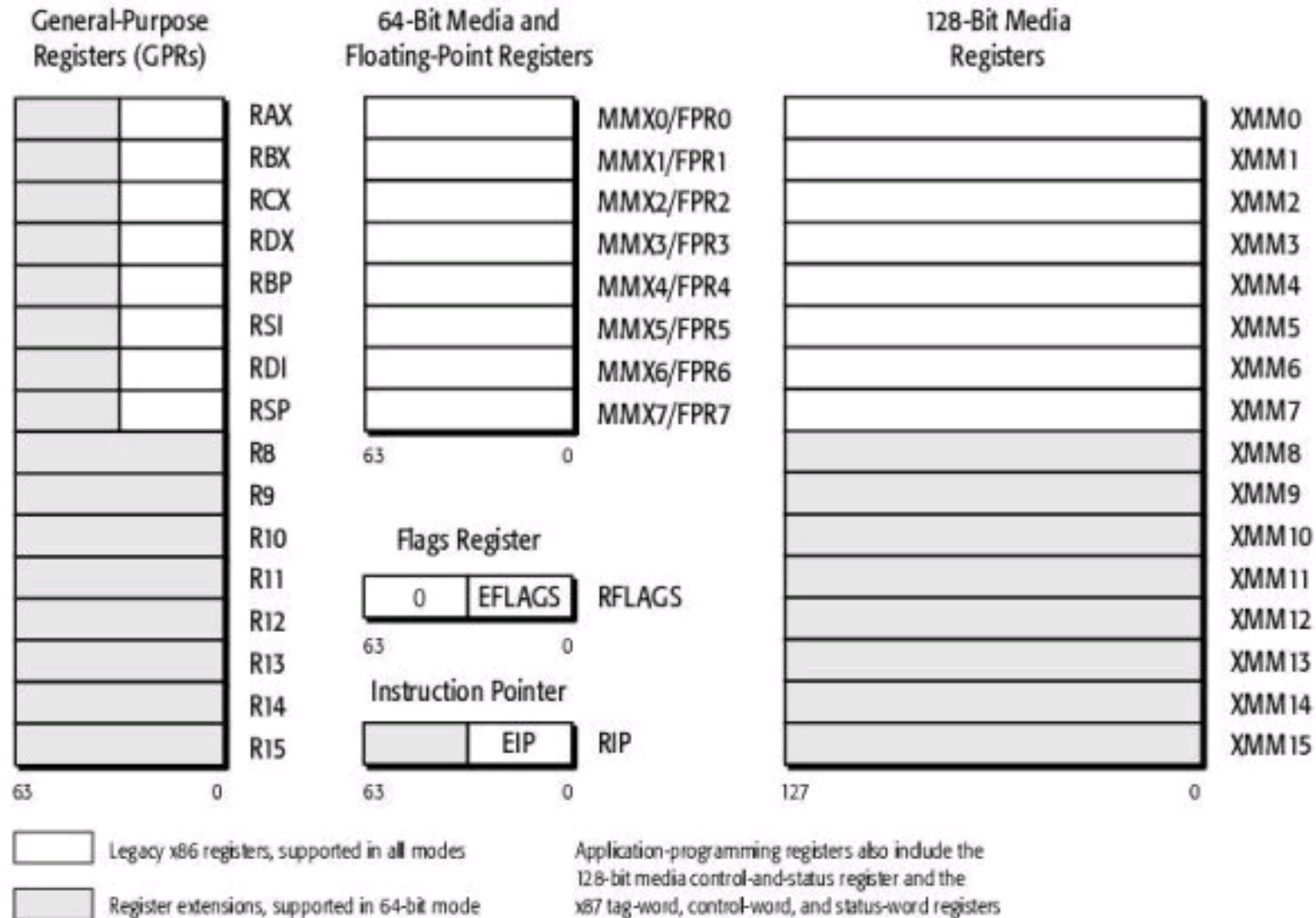
Programmiermodell des Intel 80386/80486

Erweiterung aller Register auf 32 Bit

Allgemeine Register

- Daten- und Adressregister, aber teilweise mit Spezialfunktionen
 - Akkumulator (EAX)
 - Datenregister für Ein-/Ausgabe (EDX)
 - Zählregister für Schleifen (ECX)
 - Basisregister (EBX, EBP)
 - Indexregister (ESI, EDI)
 - Stackregister (ESP)

Programmiermodell AMD-64/Intel 64



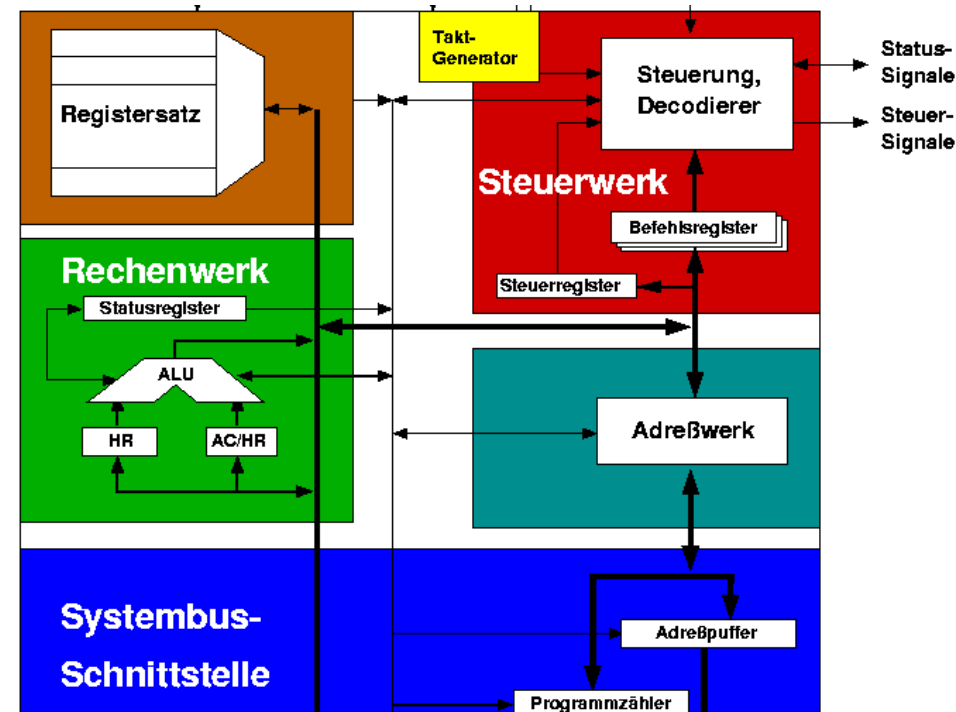
ADRESSWERK

Adresswerk

Berechnet nach den Vorschriften des Steuerwerks die Adresse eines Befehls oder eines Operanden

Früher häufig Bestandteil des Rechenwerks

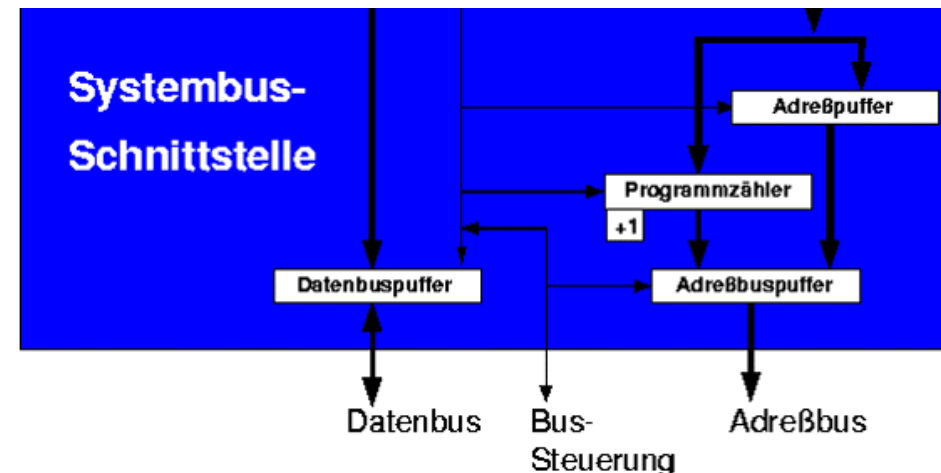
Heute sehr komplex (viele verschiedene komplexe Adressierungsarten, siehe MMU) und deshalb eigenständig



SYSTEMBUS-SCHNITTSTELLE

Die Systembus-Schnittstelle

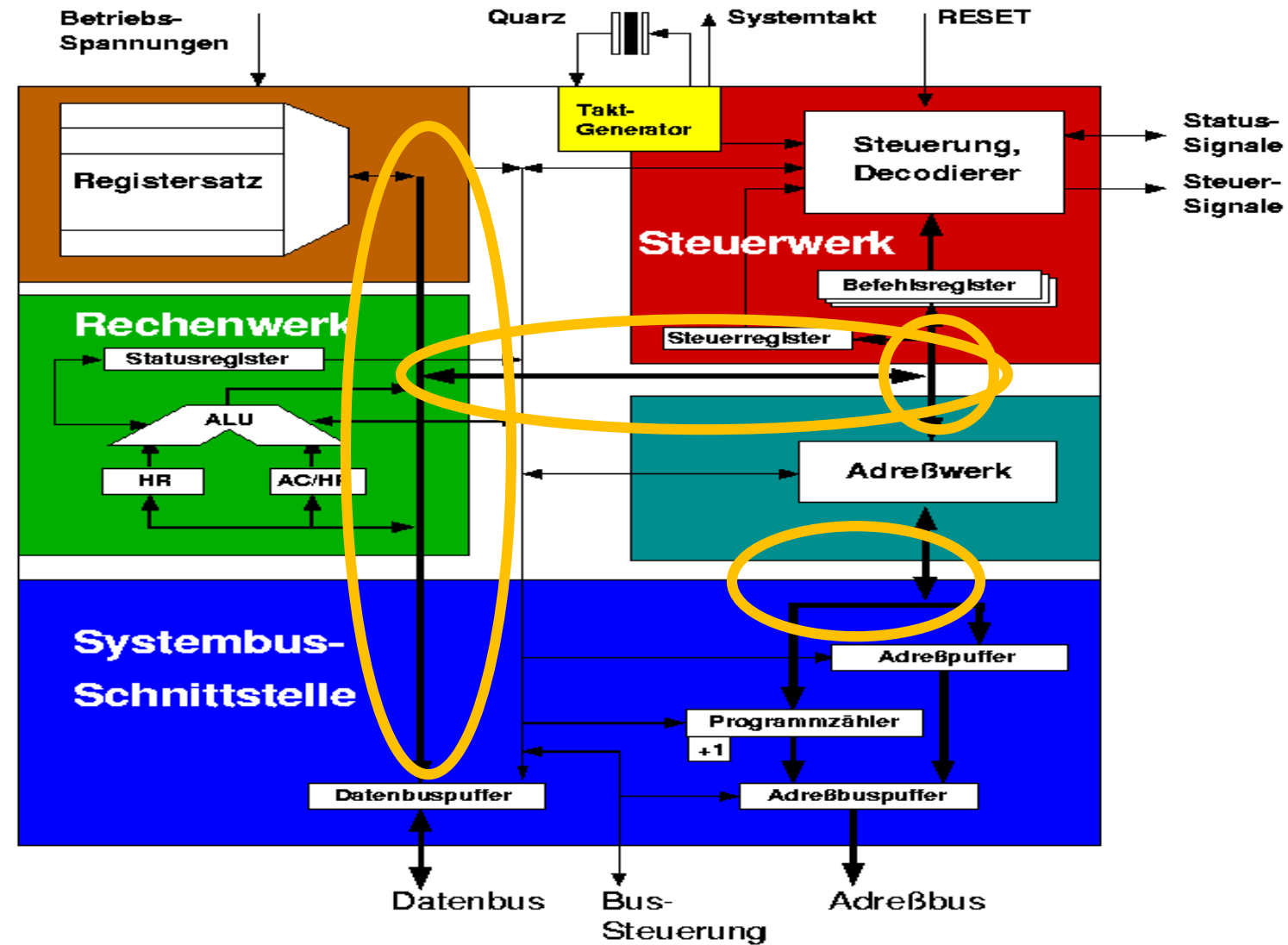
Die Systembus-Schnittstelle (Bus Interface Unit, BIU) stellt die Verbindung des Mikroprozessors zu seiner Umwelt (Komponenten des Mikrorechnersystems) dar.



Aufgaben

- kurzfristiges Zwischenspeichern (Puffern) von Adressen und Daten
- elektrische Anpassung der Signalpegel/Tristate

Bussystem

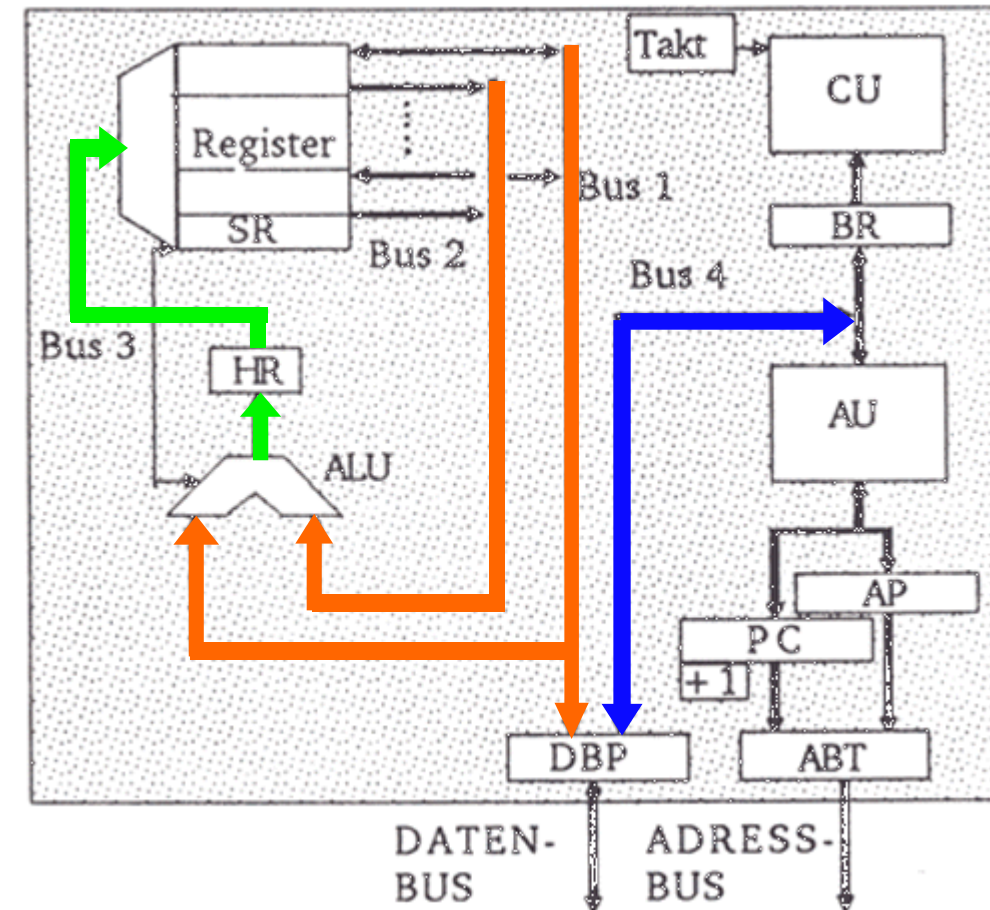


Varianten des internen Bussystems

Beispiel

Kombination aus:

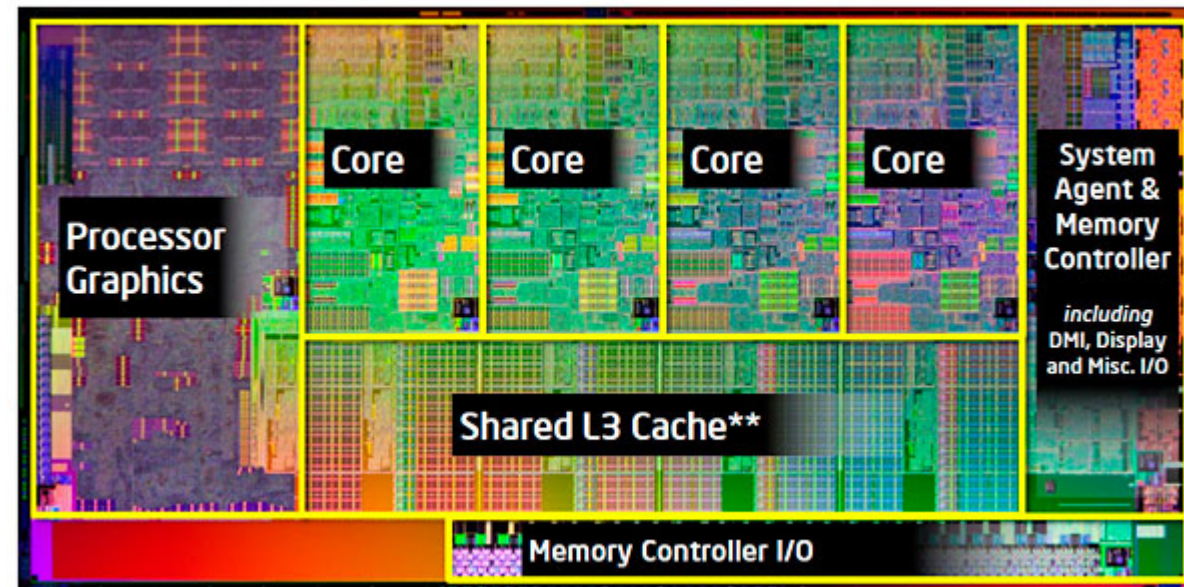
- Prefetch Bus (Bus 4)
- zwei Operandenbussen (Bus 1 und 2)
- ein Ergebnisbus (Bus 3)
- Hilfsregister wird nur bei gleichem Operanden- und Ergebnisregister benötigt



Weitere Funktionseinheiten

Bei modernen Mikroprozessoren:

- Speicherverwaltungseinheit (Memory Management Unit, MMU)
- Cache-Speicher (schnelle Zwischenspeicher) für Befehle und Daten
- Arithmetik-Koprozessor
- Graphikprozessor
- ...



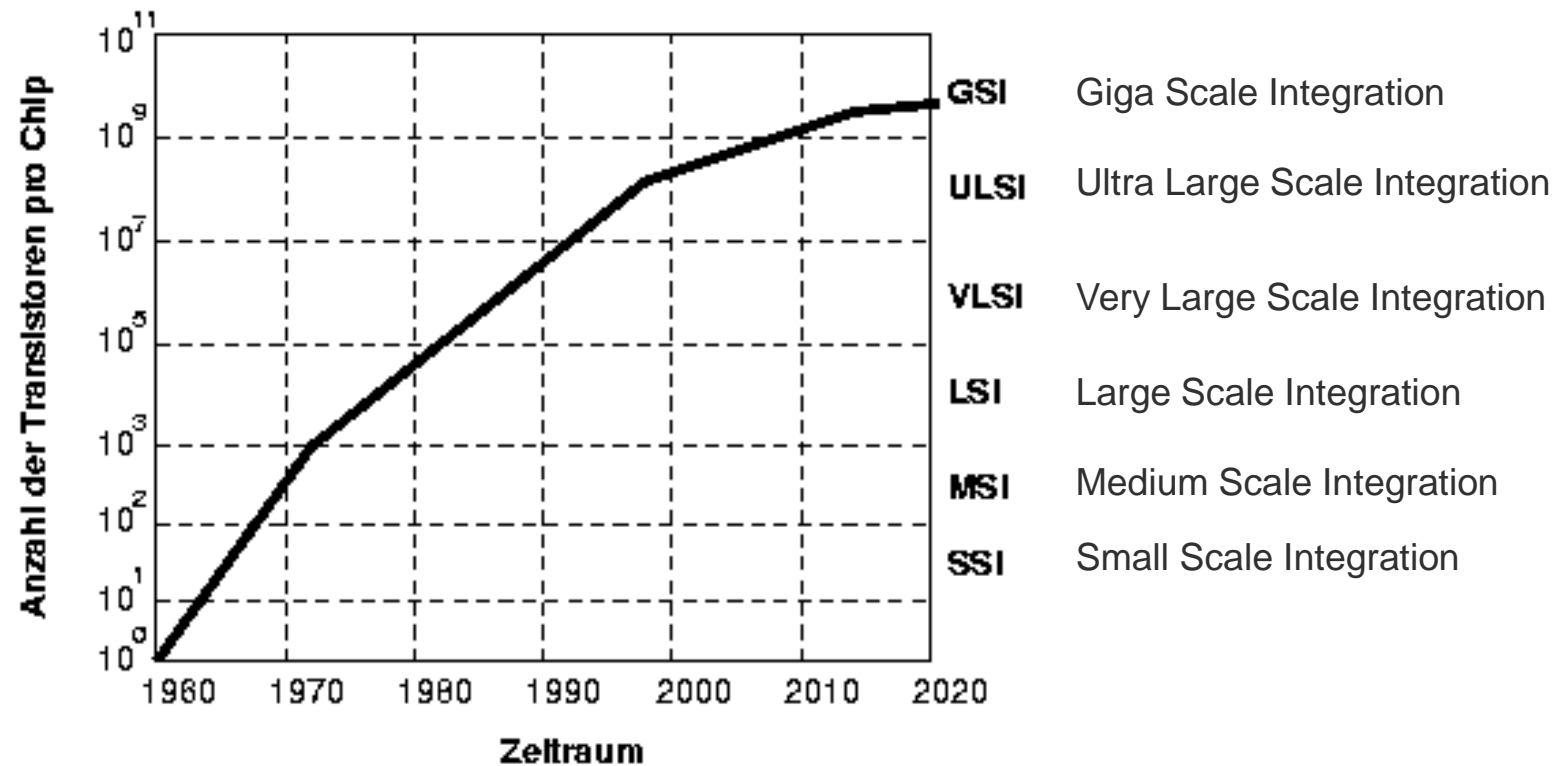
LEISTUNGSSTEIGERUNG IN RECHNERSYSTEMEN

Leistungssteigerung in Rechnersystemen

Welche Möglichkeiten hat man prinzipiell zur Leistungssteigerung in Rechnersystemen?

- Technologische Maßnahmen:
 - Anwendung schnellerer Technologien ➡ Redesign ist nötig.
 - Meist recht teuer, teilweise physikalische Schranken.
- Strukturelle Maßnahmen:
 - z.B. Anzahl der Transistoren erhöhen ➡ Parallelarbeit
 - Der aktuelle Trend geht mit Multi-Core-Prozessoren in diese Richtung

Technologie-Entwicklung



Leistungssteigerung in Rechnersystemen

STRUKTURELLE MASSNAHMEN

Strukturelle Maßnahmen

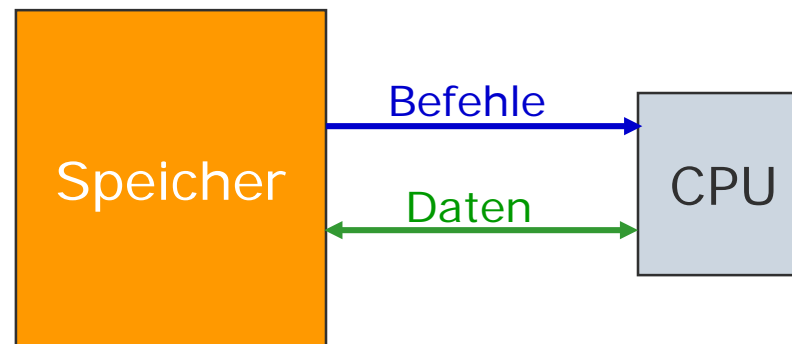
Klassifikation von Rechnerstrukturen nach Flynn

Achtung: [historische Klassifikation](#), passt nicht mehr so richtig

Unterscheidung bezüglich der gleichzeitig bearbeiteten
Befehls- und Datenströme

- SISD (Single Instruction Single Data):

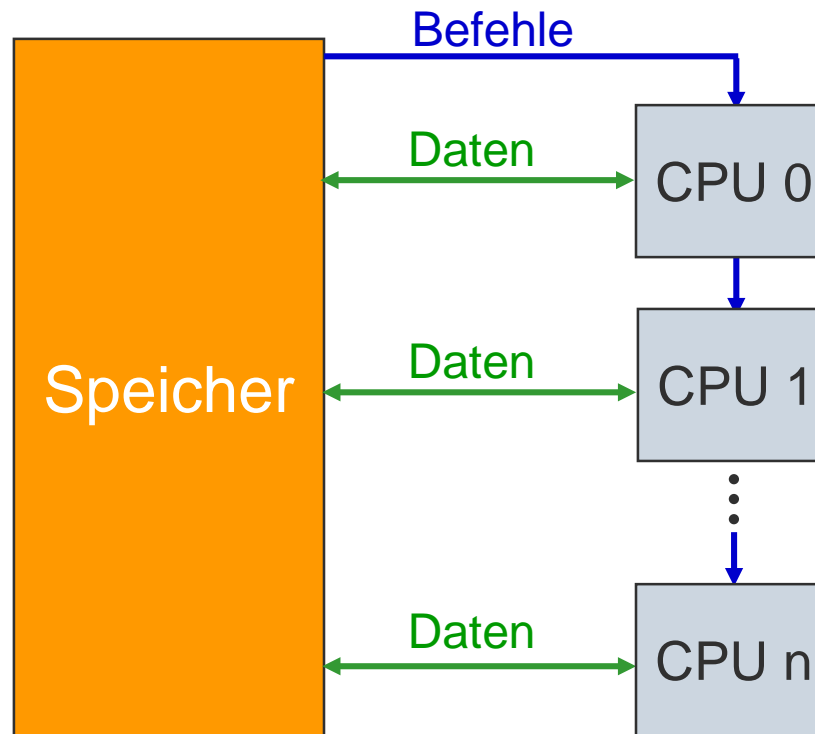
Ein Datenstrom wird entsprechend einer seriellen
Befehlsfolge verarbeitet (von-Neumann-Rechner)



Klassisch:
IBM-PC, IBM 370,
Micro-VAX von DEC

Strukturelle Maßnahmen

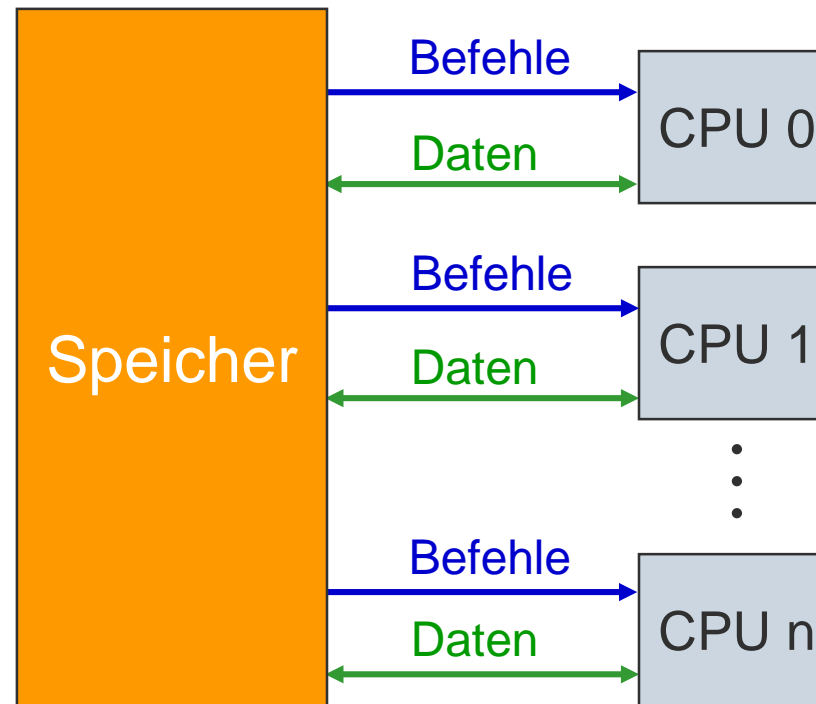
- SIMD (Single Instruction Multiple Data):
Alle Prozessoren führen gleichzeitig dieselben Befehle auf verschiedenen Daten aus (Array-Prozessoren).



Anwendung
Bildverarbeitung –
jedem Prozessor
wird ein Bildausschnitt
zugeordnet.

Strukturelle Maßnahmen

- MIMD (Multiple Instruction Multiple Data):
Alle Prozessoren führen gleichzeitig verschiedene Befehle auf verschiedenen Daten aus.



Klassisch:
IBM 3084, Cray-2,
Multiprozessor PCs –
im Prinzip alle heutigen
PCs dank Grafikkarten,
DMA-Controller,
Co-Prozessoren etc.

Strukturelle Maßnahmen

- MISD (Multiple Instruction Single Data):
Es wird nur ein Datenstrom bearbeitet. Bestimmte Ausführungseinheiten übernehmen die Ausführung bestimmter Teile einer Operation (Pipeline-Verarbeitung)

Parallelität auf Befehlsebene.

„Moderne“ Prozessoren: ab Intel 80286

- Bei vielen Autoren bleibt diese Klasse leer – darüber lässt sich beliebig diskutieren!

Strukturelle Maßnahmen

Mehrprozessorsysteme

- Mehrere Prozessoren mit unabhängigen Programmen arbeiten mit einem gemeinsamen Hauptspeicher

Feldrechner

- Mehrere Prozessoren arbeiten am gleichen Programm, aber mit verschiedenen Daten (Bsp: Bildverarbeitung)

System mit funktionsspezialisierten Prozessoren

- Mehrere Spezialprozessoren arbeiten unter einer CPU und mit einem Hauptspeicher

Fliessbandverarbeitung (Pipeline-Struktur)

- In einer Kette von Prozessoren übernimmt jeder die Ausführung bestimmter Teile einer Operation.

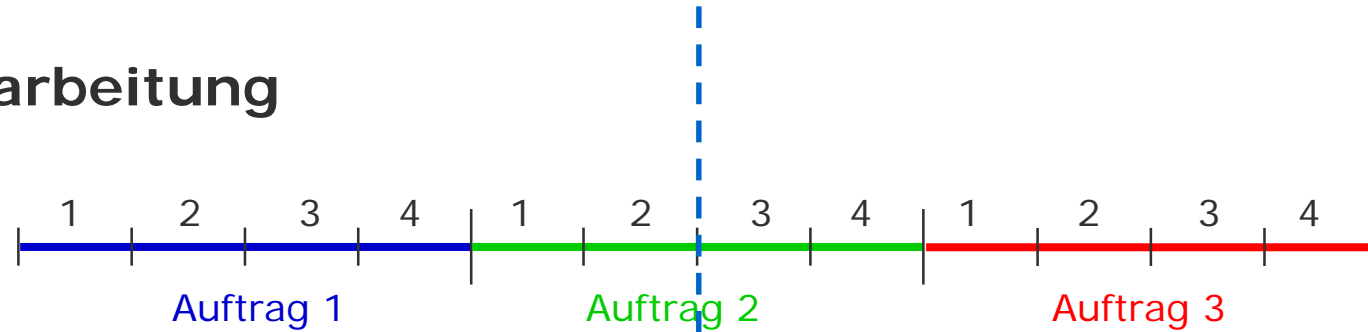
Leistungssteigerung in Rechnersystemen

PIPELINE-VERARBEITUNG

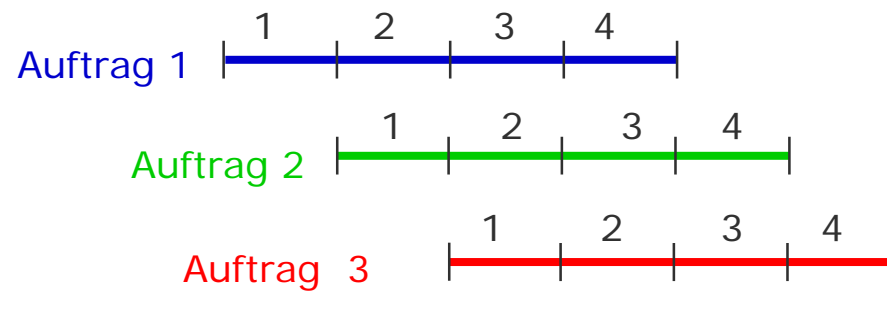
Pipeline-Verarbeitung

Ausführung von 3 gleichartigen Aufträgen in 4 Teilschritten.

Serielle Verarbeitung



Pipeline-Verarbeitung

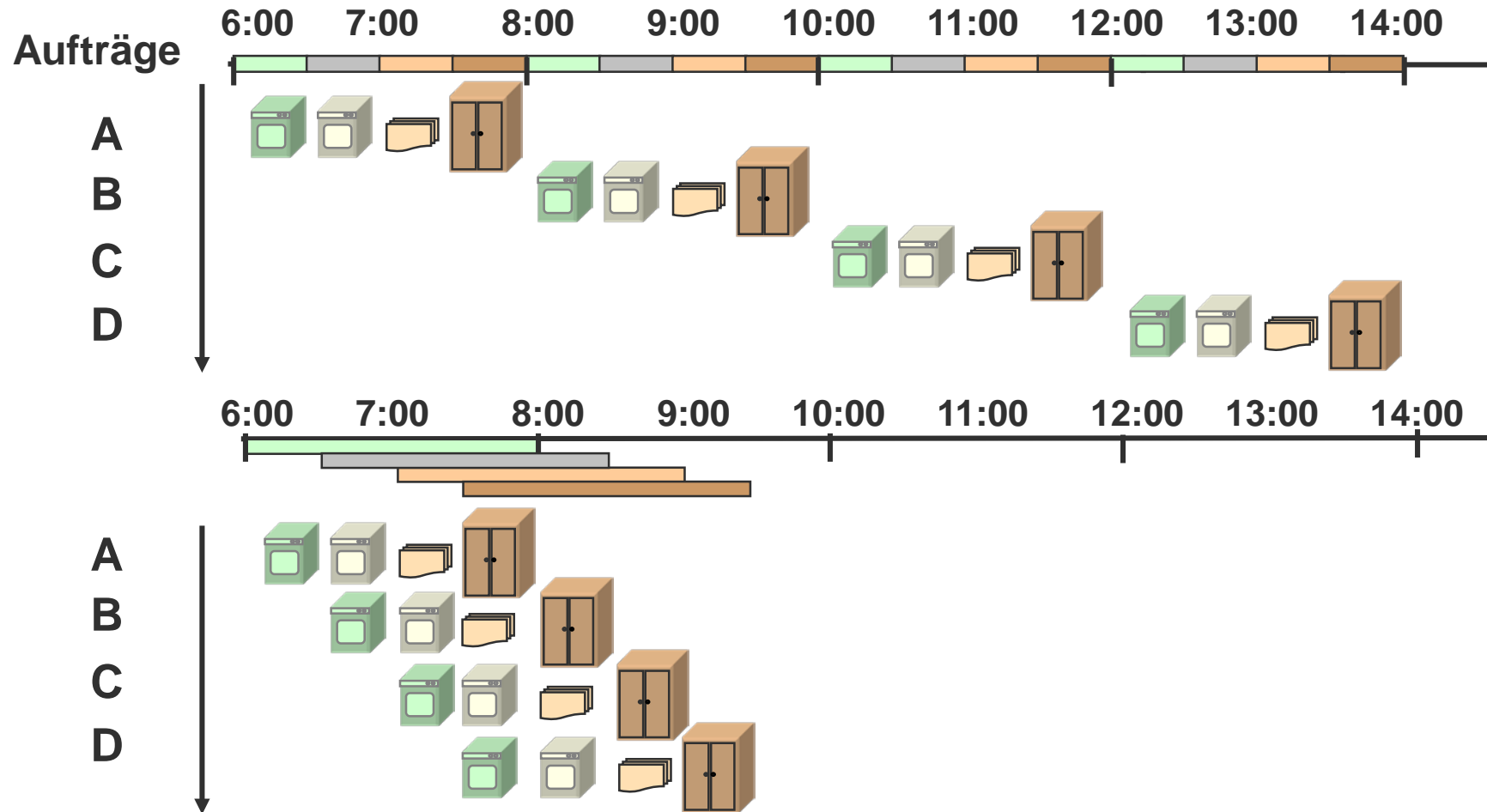


Beispiel: Wäsche-Pipelining

Ein Wäsche-Vorgang kann in 4 Teilvorgänge unterteilt werden

- Schmutzige Wäsche in die Waschmaschine
- Nasse Wäsche in den Trockner
- Falten, Bügeln, ...
- Kleider in den Schrank

Wäsche-Pipelining



Pipelining I

Pipelining:

- Subdivision of an operation into several phases or sub operations
- Synchronous execution of the sub operations in different functional units
- Each functional unit is responsible for a single function

All functional units together plus their interconnection is called **pipeline**.

Instruction pipelining:

- The pipeline principle is applied to processor instructions
- Successive instructions are executed one after another with a delay of a single cycle

Pipelining II

Each stage of a pipeline is called **pipeline stage** or **pipeline segment**.

Pipeline registers (latches) separate pipeline stages from each other.

The whole pipeline is clocked in a way that **each cycle** an instruction can be shifted one step further through the pipeline.

In an ideal scenario, an instruction is executed in a k **stage pipeline** within k cycles by k stages (...we will see problems due to hazards later).

If every clock cycle a new instruction is loaded into the pipeline, then k **instructions are executed simultaneously** and each instruction needs k cycles in the pipeline.

Pipelining III

Latency: Duration of the complete processing of an instruction. This is the time an instruction needs to go through all k stages of the pipeline.

Throughput: Number of instructions leaving the pipeline per clock cycle. This number should be close to 1 for a scalar processor.

Instruction execution

Hypothetical processor without pipeline: $n \times k$ cycles
(k stage pipeline, n instructions)

Pipelined processor with a k stage pipeline: $k + (n - 1)$ cycles
(under ideal conditions: k cycles latency, throughput of 1)

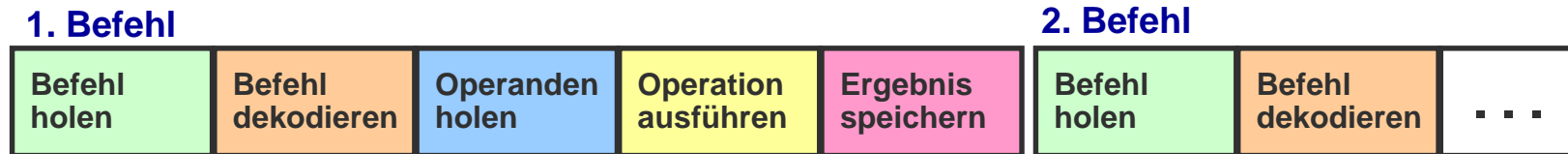
This results in a **speed-up** of:

$$S = \frac{nk}{k + n - 1} = \frac{k}{\frac{k}{n} + 1 - \frac{1}{n}}$$

Assuming an infinite number of instructions ($n \rightarrow \infty$) the speed-up of a processor with a k stage pipeline equals k .

Pipelining

Sequentielle Ausführung:



Pipelining:



Aufbau einer fünfstufigen Pipeline I

Befehl-Holphase (Instruction Fetch, IF)

- Der Befehl wird aus dem Arbeitsspeicher (bzw. dem Befehlscache) ins Befehlsregister geladen.
- Der Befehlszähler wird weitergeschaltet.

Decodierphase (Instruction Decode, ID)

- Aus dem Operationscode des Maschinenbefehls werden prozessorinterne Steuersignale erzeugt.

Operanden-Holphase (Operand Fetch, OF)

- Das Steuerwerk schaltet die Operanden auf die Busse zum Rechenwerk. Diese Operanden stehen im Registersatz.
- Bei Lade-/Speicherbefehlen oder Verzweigungen wird die effektive Adresse durch das Adresswerk berechnet

Aufbau einer fünfstufigen Pipeline II

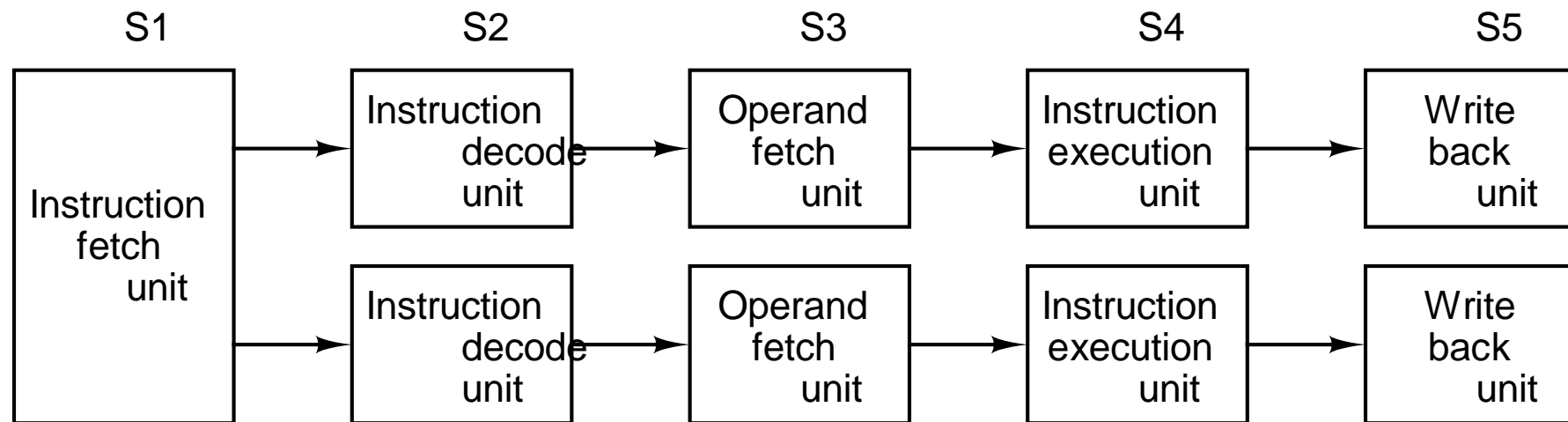
Ausführungsphase (Execution Phase, EXE, ALU Operation)

- Die verlangte Operation wird vom Rechenwerk ausgeführt

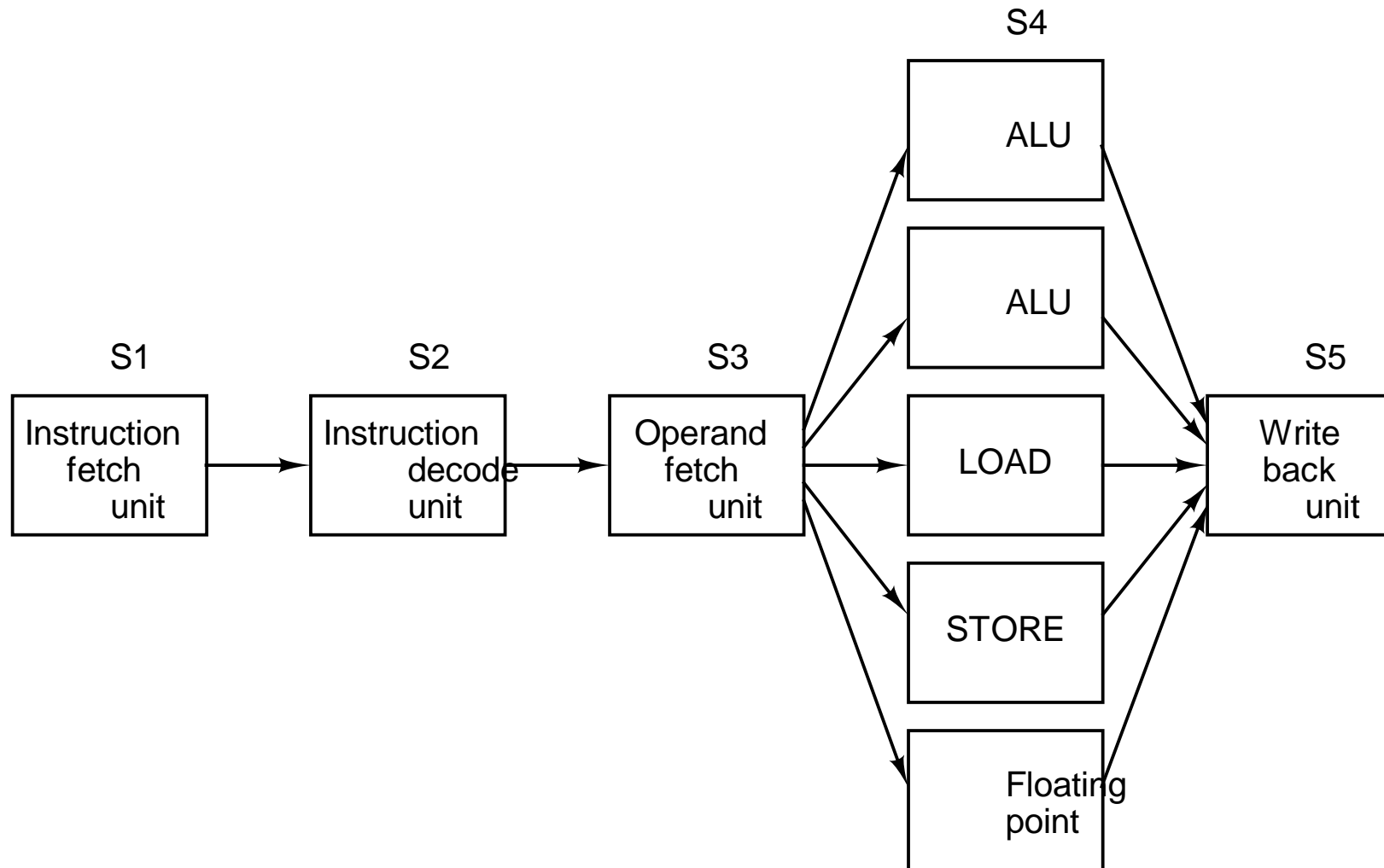
Abspeicherungsphase (result Write Back phase, WB)

- Das Ergebnis wird in einem Register oder im Speicher abgelegt.
- **Befehle ohne Ergebnis** durchlaufen diese Phase **passiv**.
- Bei Lade-/Speicherbefehlen wird die Adresse auf den Adressbus gelegt und das Datum zwischen Registersatz und Arbeitsspeicher übertragen

Performance enhancement: two pipelines



Performance enhancement: specialized EXE-units



Discussion

Pipeline hazards: phenomena that disrupt the smooth execution of a pipeline.

Example:

- If we assume a unified cache with a single read port (instead of separate I- and D-caches)
 - ➔ a **memory read conflict** appears among IF and MEM stages.
- The pipeline has to stall one of the accesses until the required memory port is available.

A stall is also called a **pipeline bubble**.

Leistungssteigerung in Rechnersystemen

PIPELINE-VERARBEITUNG - TYPES OF PIPELINE HAZARDS

Three types of pipeline hazards

1. **Data hazards** arise because of the unavailability of an operand
 - For example, an instruction may require an operand that will be the result of a preceding, still uncompleted instruction.
2. **Structural hazards** may arise from some combinations of instructions that cannot be accommodated because of resource conflicts
 - For example, if processor has only one register file write port and two instructions want to write in the register file at the same time.
3. **Control hazards** arise from branch, jump, and other control flow instructions
 - For example, a taken branch interrupts the flow of instructions into the pipeline
 - ➔ the branch target must be fetched before the pipeline can resume execution.

Common solution is to **stall the pipeline** until the hazard is resolved, inserting one or more “**bubbles**” in the pipeline.

Types of Pipeline Hazards

DATA HAZARDS

Pipeline hazards due to data dependence

After a load instruction the loaded value is not available to the following instruction in the next cycle.

If an instruction needs the result of a preceding instruction it has to wait.

Example:

ADD R1,R2,R1;	$R1 \leftarrow R1 + R2$
ADD R3,R1,R3;	$R3 \leftarrow R3 + R1$

Data hazards

Dependencies between instructions may cause data hazards when Instr1 and Instr2 are so close that their overlapping within the pipeline would change their access order to registers.

Three types of data hazards

- Read After Write (RAW)

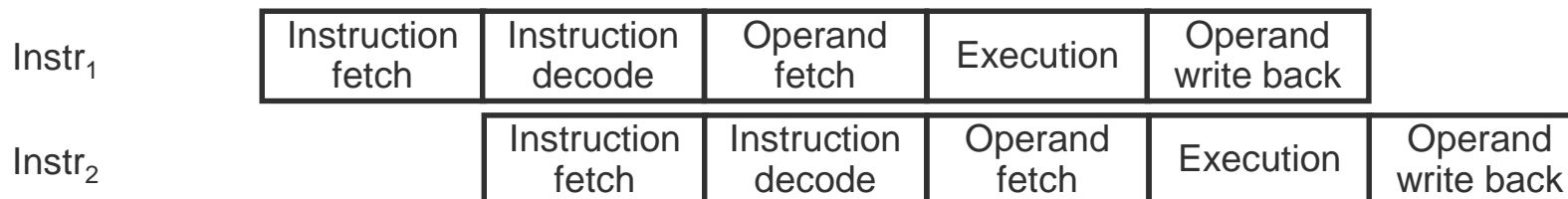
- Instr2 tries to read operand before Instr1 writes it

- Write After Read (WAR)

- Instr2 tries to write operand before Instr1 reads it

- Write After Write (WAW)

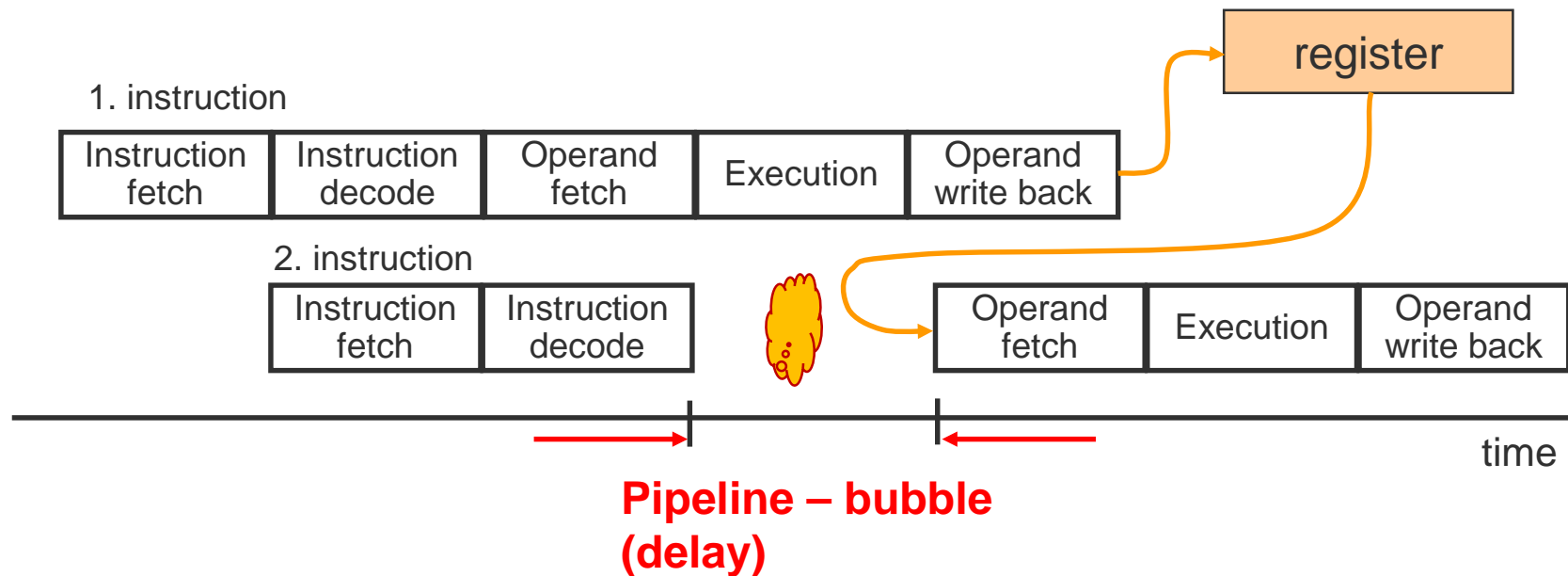
- Instr2 tries to write operand before Instr1 writes it



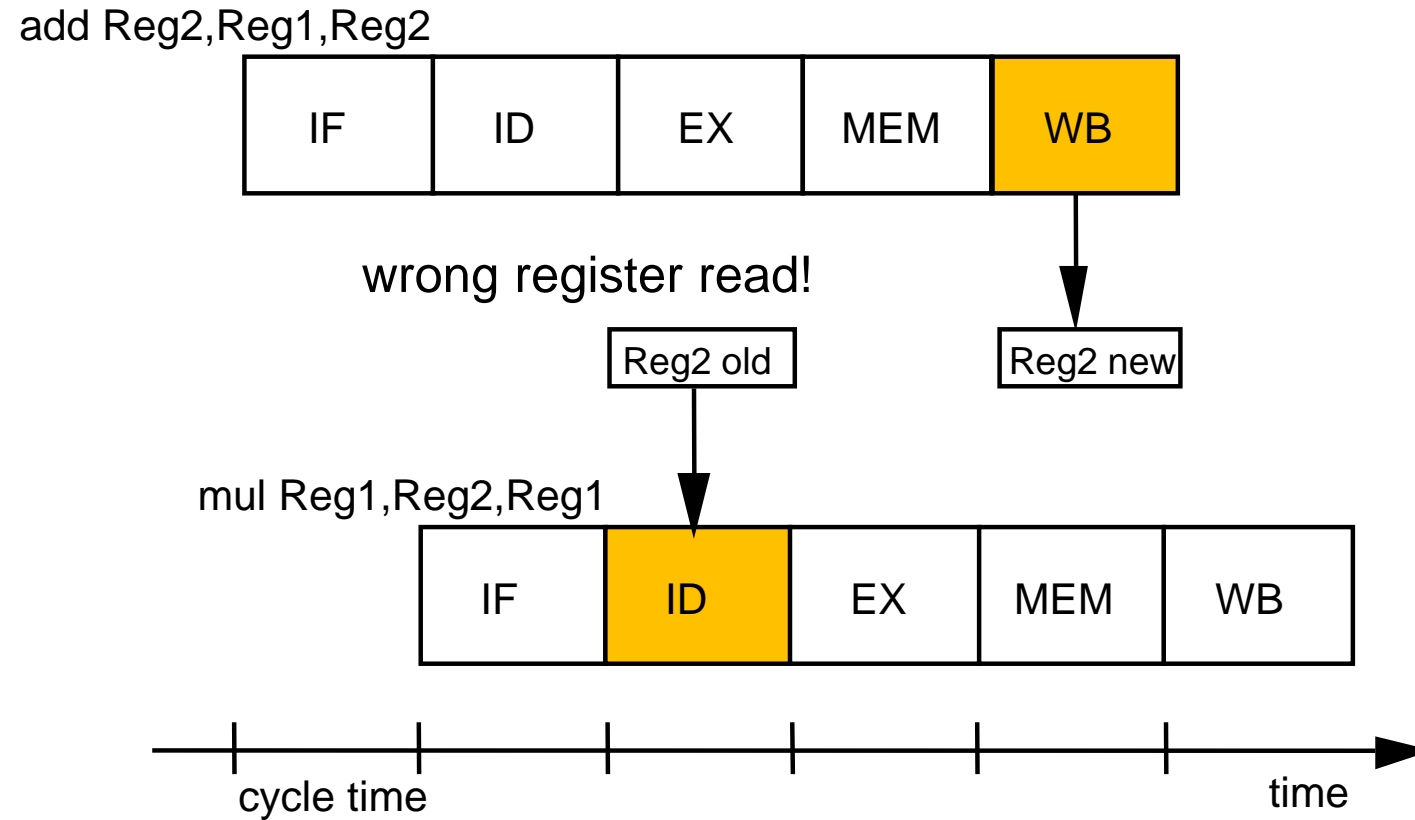
Read-after-Write-Conflict (True Dependence)

Using a simple 5 stage pipeline this example shows that the operand fetch phase of the 2nd instruction comes before the 1st instruction writes back its result

- Delaying the pipeline is necessary!

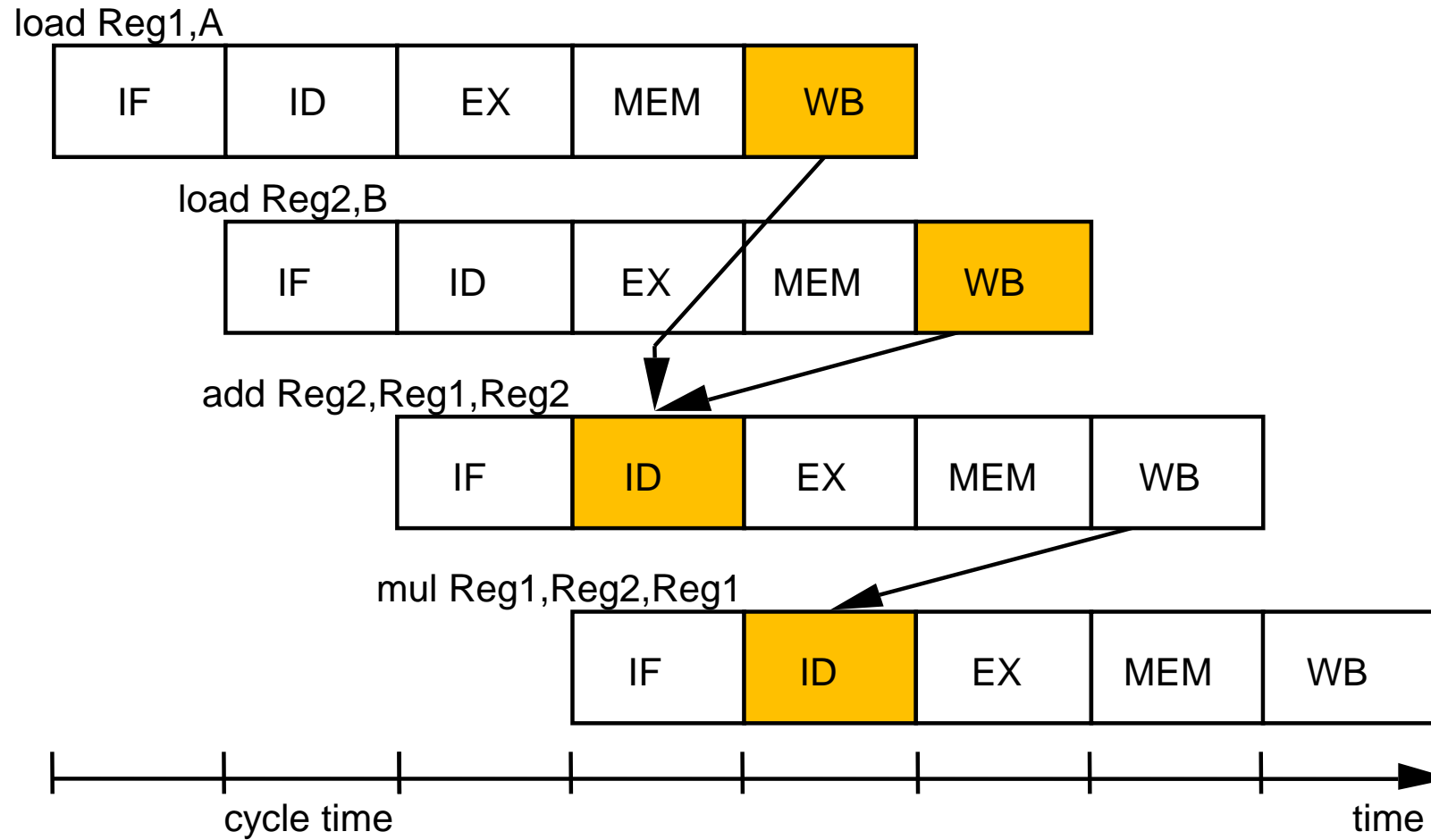


Pipeline conflict due to a data hazard



add **Reg2**,Reg1,Reg2; **Reg2** \leftarrow Reg1 + Reg2
mul Reg1,**Reg2**,Reg1; Reg1 \leftarrow Reg1 * **Reg2**

Data hazards in an instruction pipeline



WAR and WAW - Can they happen in our simple pipeline?

WAR and WAW can't happen in the simple DLX 5 stage pipeline, because:

- All instructions take 5 stages
- Register reads are always in stage 2
- Register writes are always in stage 5

WAR and WAW may happen e.g. in superscalar pipes.

Solutions for data hazards from true data dependences


Software solution (Compiler scheduling)

- putting **no-op** (NOP) instructions after each instruction that may cause a hazard
- **instruction scheduling**
 - ➔ rearrange code to reduce no-ops

Software solutions

Insertion of NOP instructions by the compiler


```
LOAD    <addr>, R1
NOP
ADD     R1, R2, R3
SUB     R4, R5, R6
```

Instruction fetch	Instruction decode	Operation execution	Write back result
LOAD
NOP	LOAD
ADD	NOP	LOAD	...
SUB	ADD 	NOP	----
instr 4	SUB	ADD	NOP
instr 5	instr 4	SUB	ADD
...	instr 5	instr 4	SUB

Software solutions

Reordering of instructions by the compiler

```
LOAD    <addr>, R1
SUB     R4, R5, R6
ADD     R1, R2, R3
```

Instruction fetch	Instruction decode	Operation execution	Write back result
LOAD
SUB	LOAD
ADD	SUB	LOAD	...
instr 4	ADD 	SUB	-- -- --
instr 5	instr 4	ADD	SUB
...	instr 5	instr 4	ADD

Hardware solutions

Hardware solutions: Hazard detection logic necessary!

Delay Insertion

- Stalling/Interlocking: stall pipeline for one or more cycles

Bypass techniques

- Forwarding:

- Result Forwarding

Example: The result in ALU output of Instr1 in EX stage can immediately be forwarded back to ALU input of EX stage as an operand for Instr2,

- Load Forwarding

Example: The load memory data register from MEM stage can be forwarded to ALU input of EX stage.

- Forwarding with interlocking: Assuming that Instr2 is data dependent on the load instruction Instr1 then Instr2 has to be stalled until the data loaded by Instr1 becomes available in the load memory data register in MEM stage.

Even when forwarding is implemented from MEM back to EX, one bubble occurs that cannot be removed.

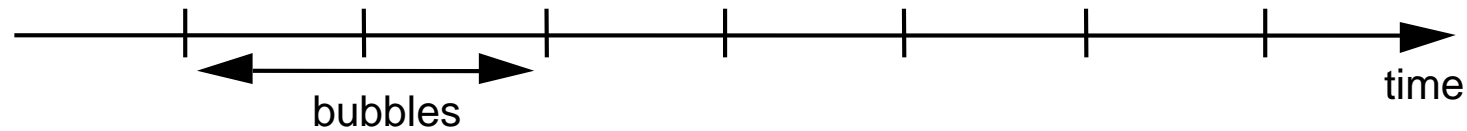
Data hazard: Hardware solution by interlocking

add Reg2,Reg1,Reg2

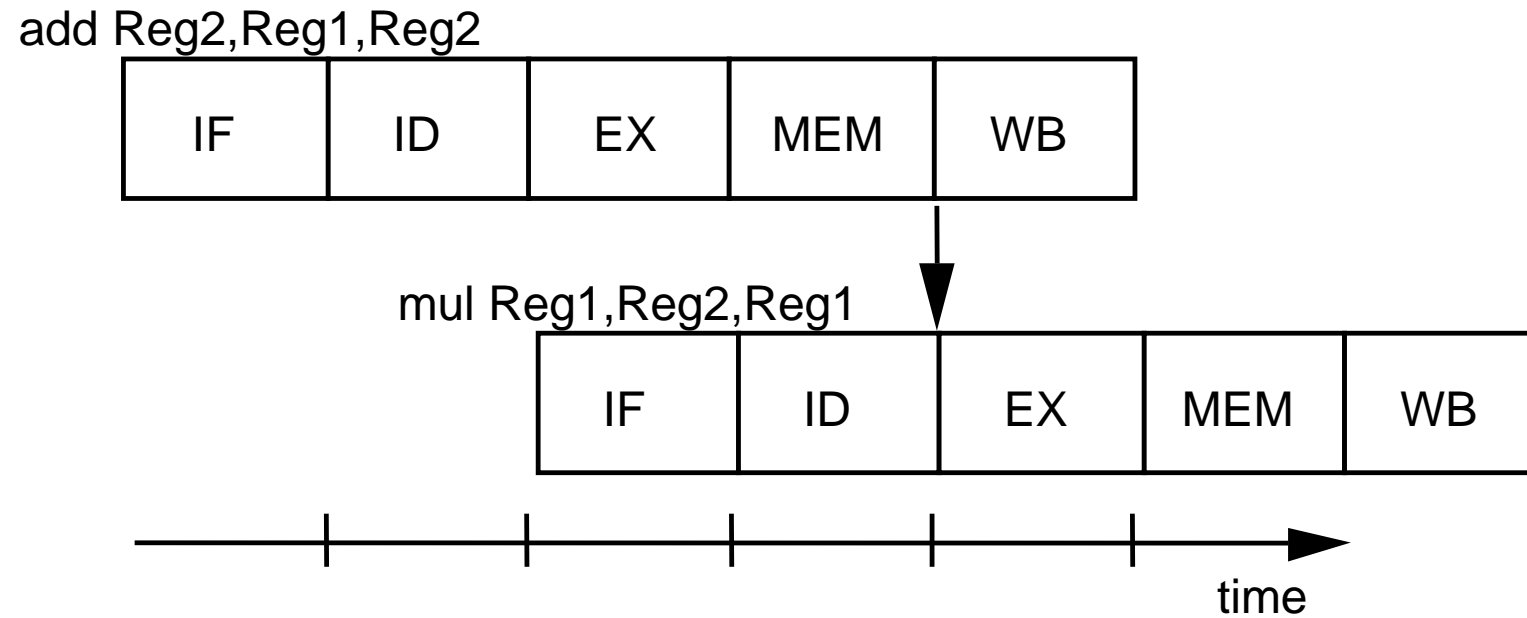


Register Reg2

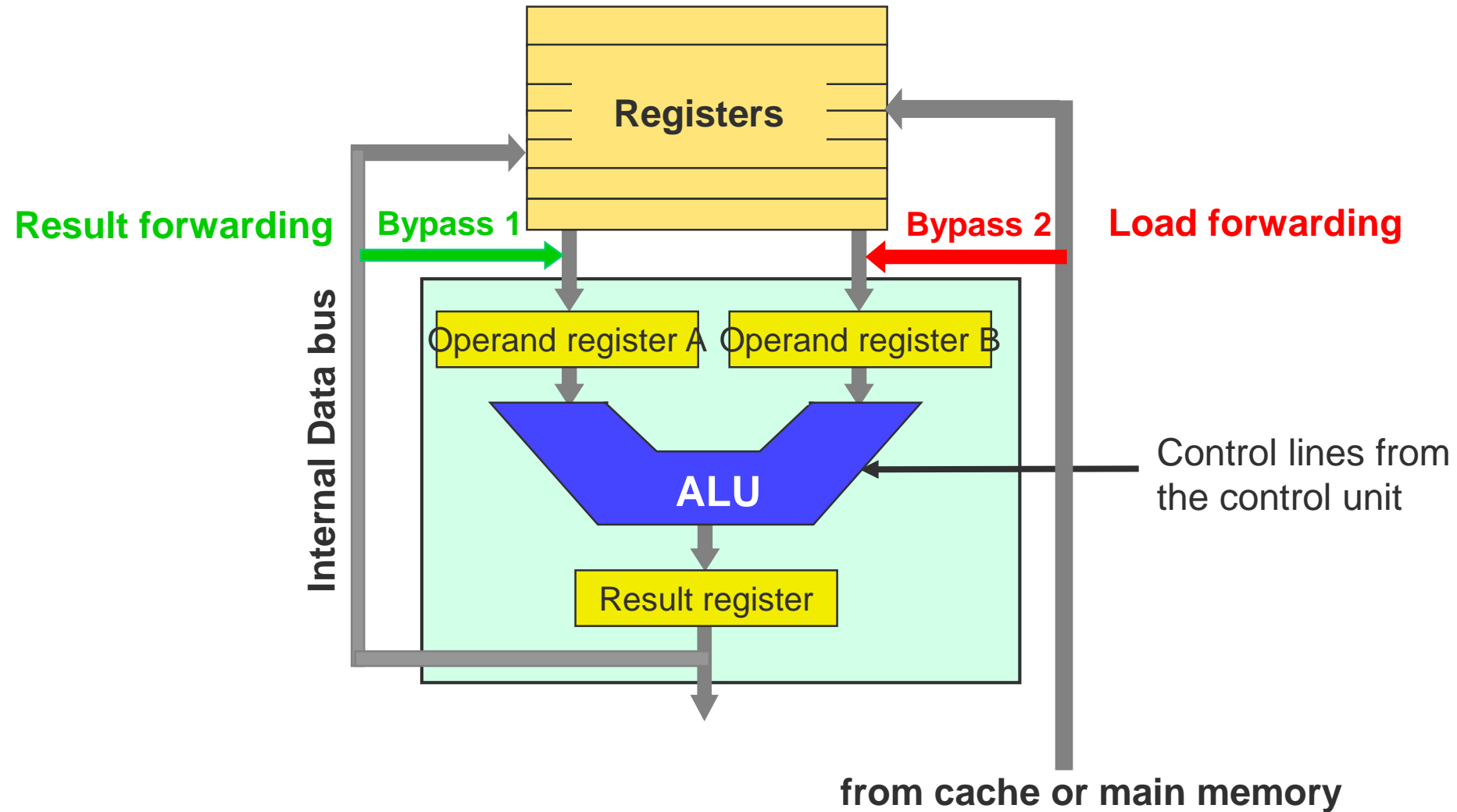
mul Reg1,Reg2,Reg1



Data hazard: Hardware solution by forwarding



Bypass techniques

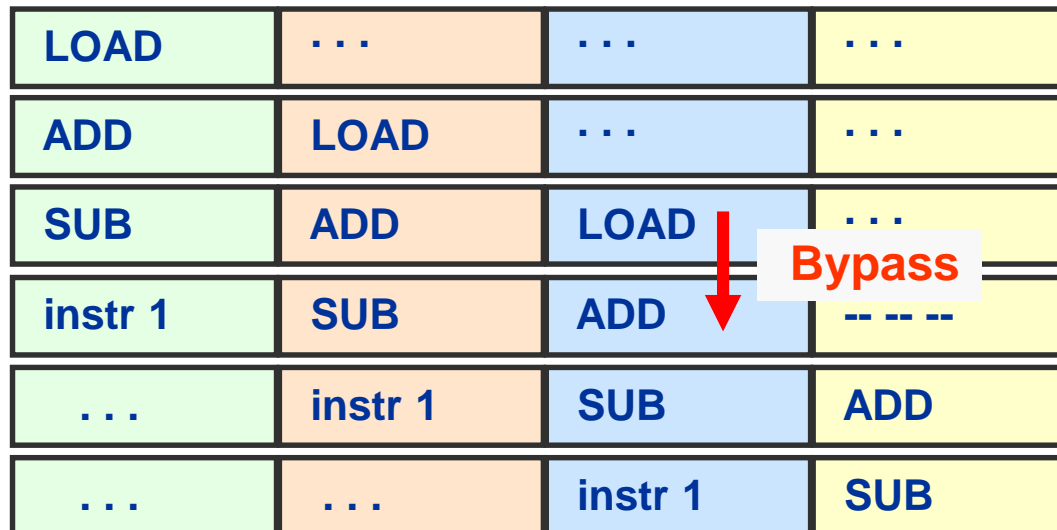


Example

LOAD	<Address>, R1	$R1 \leftarrow (<Address>)$
ADD	R1, R2, R3	$R3 \leftarrow R1 + R2$
SUB	R4, R5, R6	$R6 \leftarrow R4 - R5$



Yet another (simple) pipeline...



During the execution phase of the ADD instruction the result of the LOAD is written into the register and, thus, the bypass is needed to provide the result early enough.

Types of Pipeline Hazards

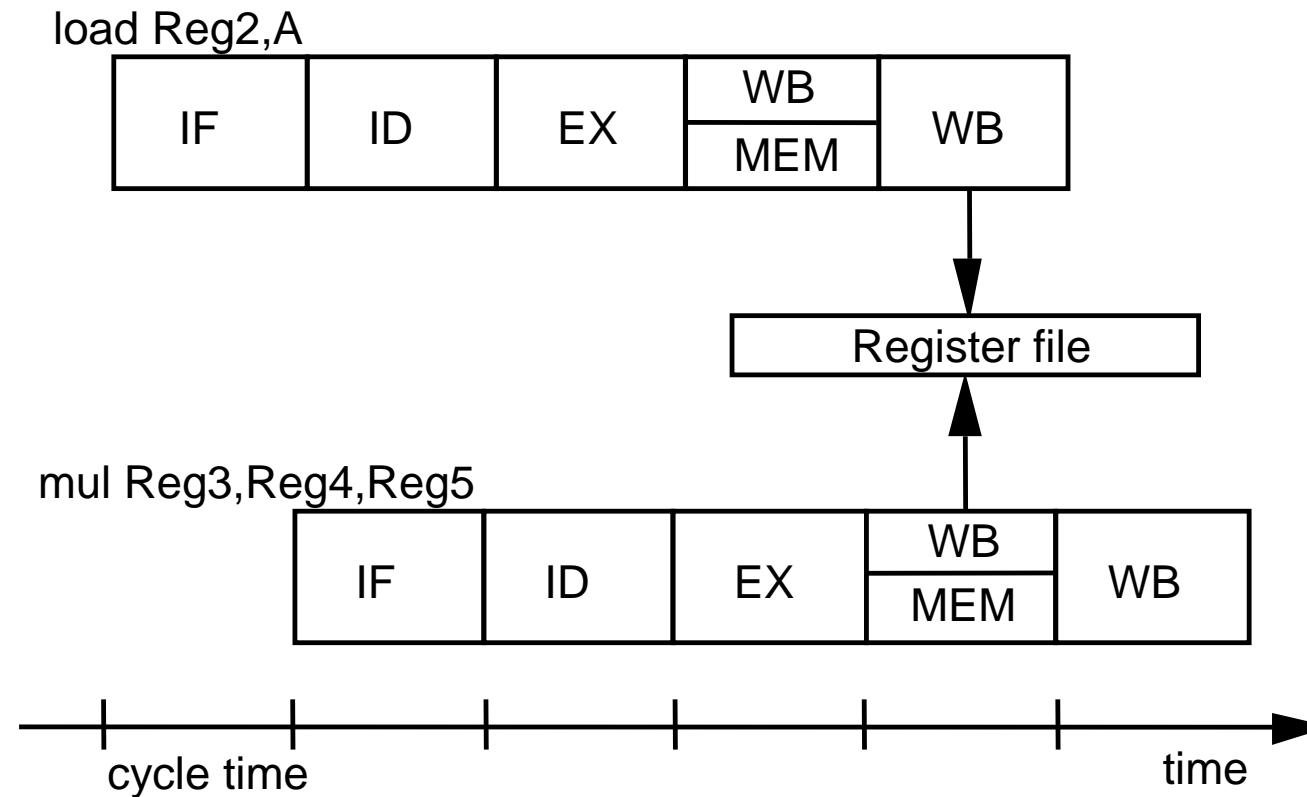
STRUCTURAL HAZARDS

Three types of pipeline hazards

1. **Data hazards** arise because of the unavailability of an operand
 - For example, an instruction may require an operand that will be the result of a preceding, still uncompleted instruction.
2. **Structural hazards** may arise from some combinations of instructions that cannot be accommodated because of resource conflicts
 - For example, if processor has only one register file write port and two instructions want to write in the register file at the same time.
3. **Control hazards** arise from branch, jump, and other control flow instructions
 - For example, a taken branch interrupts the flow of instructions into the pipeline
 - ➡ the branch target must be fetched before the pipeline can resume execution.

Common solution is to stall the pipeline until the hazard is resolved, inserting one or more “**bubbles**” in the pipeline.

Pipeline bubble due to a structural hazard



Solutions to the structural hazard

Arbitration with interlocking: hardware that performs resource conflict arbitration and interlocks one of the competing instructions



Resource replication: In the example a register file with multiple write ports would enable simultaneous writes.

- However, now output dependencies may arise!
- Therefore, additional arbitration and interlocking necessary
- or the first (in program flow) value is discarded and the second used.

Types of Pipeline Hazards

CONTROL HAZARDS

Three types of pipeline hazards

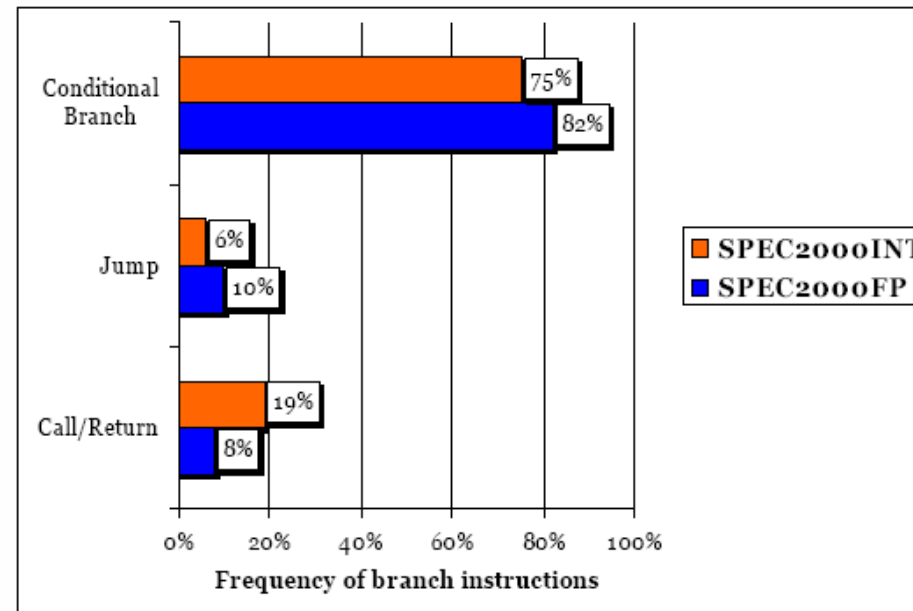
1. **Data hazards** arise because of the unavailability of an operand
 - For example,  an instruction may require an operand that will be the result of a preceding, still uncompleted instruction.
2. **Structural hazards** may arise from some combinations of instructions that cannot be accommodated because of resource conflicts
 - For example,  if processor has only one register file write port and two instructions want to write in the register file at the same time.
3. **Control hazards** arise from branch, jump, and other control flow instructions
 - For example, a taken branch interrupts the flow of instructions into the pipeline
 - ➡ the branch target must be fetched before the pipeline can resume execution.

Hazards due to control dependence

Conditional Jumps and branches stop linear program execution, program might continue elsewhere

Jump instruction normally detected in the Instruction Decode stage of the pipeline

- If a jump is detected, the pipeline already contains instructions, that are immediately behind this instruction



Source: H&P using Alpha

Hazards due to control dependence

Jumps are very common in programs

C Syntax

```
n=10
s=0
for(i=0; i<n; i++) {
    s = s + i;
}
...
```

MMIX Syntax

	LOC	#100
s	IS	\$1
i	IS	\$2
test	IS	\$3
n	IS	10
Main	SETL	s,0
	SETL	i,0
For	ADD	s,s,i
	ADD	i,i,1
	SUB	test,i,n
	BNZ	test,For
	...	
	TRAP	0,Halt,0

Example

```

ADC    R4,R5,R4    ; R4 ← R4 + R5 + C

CMP    R1,R2       ; R1 = R2 ?

BEQ    Label       ; PC ← <Label Adresse>

ADD    R3,R1,R2    ; R3 ← R1 + R2

Label: SUB    R6,R4,R5    ; R6 ← R4 - R5

SLL    R0           ; R0 ← shift_left(R0)

```

Example

```

CMP    R1,R2
ADC    R4,R5,R4
BEQ    Label
ADD    R3,R1,R2
Label: SUB    R6,R4,R5
      SLL    R0
  
```

Instruction fetch	Instruction decode	Operation execution	Write back result
ADC
BEQ	ADC
ADD	BEQ	ADC	...
SUB	ADD	-- -- --	ADC
SLL	SUB	ADD	-- -- --
...	SLL	SUB	ADD
...	...	SLL	SUB

The ADD instruction is still in the pipeline and therefore executed before the jump is realized!

Solutions

Hardware Solutions

Pipeline Flushing

- Flush (empty) pipeline before realising the jump

Speculative Branch

- In case of a conditional jump: Estimate result of condition and load pipeline (speculative)
- Wrong speculation: Pipeline Flushing
- Branch prediction used in most modern processors

Simple solutions

Hardware Interlocking

- This is the simplest way to deal with control hazards: the hardware must detect the branch and apply hardware interlocking to stall the next instruction(s).

Software Solutions

- Insertion of NOP instructions by the compiler after every branch
- Re-Ordering of instructions by the compiler
 - Instead of NOPs, instructions that will be executed anyway and that do not influence the branch condition will be put into the pipeline immediately after the branch instruction (early days of RISC processors)

Solution: Decide branch direction earlier

Flushing or Locking is often not acceptable

Reordering is often not possible

Calculation of the branch direction and of the branch target address should be done in the pipeline as early as possible.

Best solution: Already in ID stage after the instruction has become recognized as branch instruction.

Branch Prediction

Branch prediction foretells the outcome of conditional branch instructions, excellent branch handling techniques are essential for today's and for future microprocessors.

IF stage finds a branch instruction

➡ predict branch direction

The branch delay slots are speculatively filled with instruction

- of the consecutively following path
- of the path at the target address

After resolving of the branch direction

➡ decide upon correctness of prediction

In case of misprediction ➡ discard wrongly fetched instructions

- Rerolling when a branch is mispredicted is expensive:
 - 9 cycles on Itanium
 - 11 or more cycles in the Pentium II

Branch-Target Buffer or Branch-Target Address Cache

The **Branch Target Buffer (BTB)** or **Branch-Target Address Cache (BTAC)** stores branch and jump target addresses.

It should be known already in the IF stage whether the as-yet-undecoded instruction is a jump or branch.

The BTB is accessed during the IF stage.

The BTB consists of a table with branch addresses, the corresponding target addresses, and prediction information.

Variations:

- **Branch Target Cache (BTC)**: stores one or more target instructions additionally.
- **Return Address Stack (RAS)**: a small stack of return addresses for procedure calls and returns is used additional to and independent of a BTB.

Branch-Target Buffer or Branch-Target Address Cache

Branch address	Target address	Prediction <small>Bits</small>

Two Basic Techniques of Branch Prediction

Static Branch Prediction

- The prediction direction for an individual branch remains always the same!

Dynamic Prediction

- The prediction direction depends upon previous (the “history” of) branch executions.

Static Branch Prediction

The prediction direction for an individual branch remains always the same!

- the machine cannot dynamically alter the branch prediction (in contrast to dynamic branch prediction which is based on previous branch executions).

Static branch prediction comprises

- machine-fixed prediction (e.g. always predict taken)
- compiler-driven prediction.

If the prediction followed the wrong instruction path, then the wrongly fetched instructions must be squashed from the pipeline.

Static Branch Prediction - machine-fixed

Wired taken/not-taken prediction

- The static branch prediction can be wired into the processor by predicting that all branches will be taken (or all not taken).

Direction based prediction

- **Backward branches** are predicted **to be taken** and **forward branches** are predicted **to be not taken**
 - ➡ helps for loops

Static Branch Prediction - compiler-based

Opcode bit in branch instruction allows the compiler to reverse the hardware prediction.

There are two approaches the compiler can use to statically predict which way a branch will go:

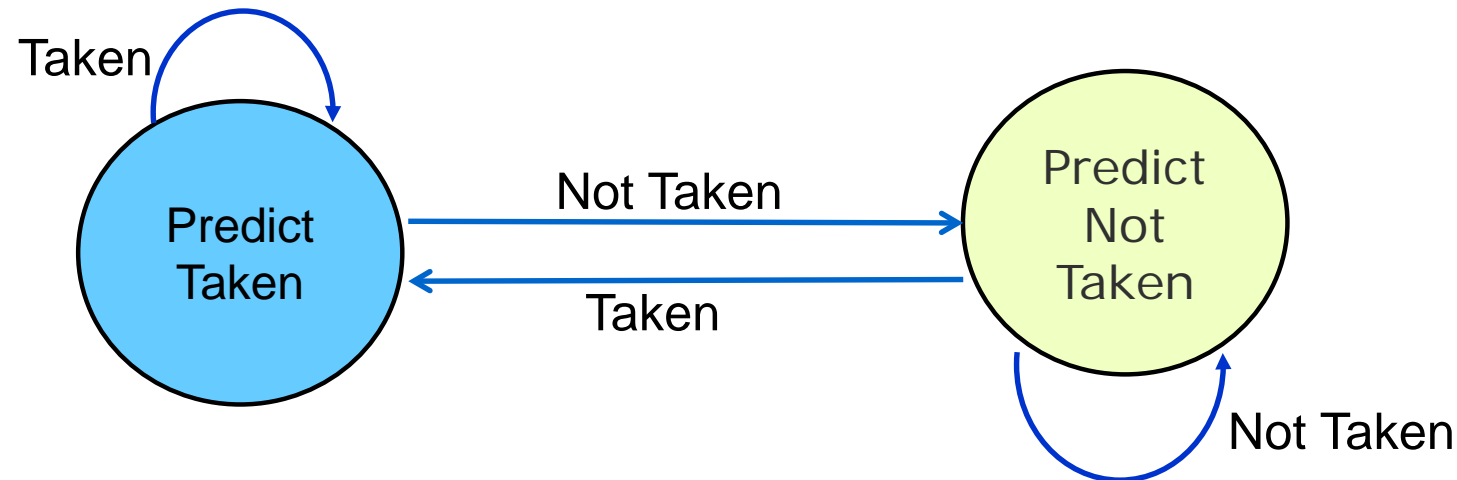
- it can examine the program code, or
- it can use profile information (collected from earlier runs)

Dynamic Branch Prediction

In dynamic branch prediction the prediction is decided on the computation history of the program execution.

In general, dynamic branch prediction gives better results than static branch prediction, but at the cost of increased hardware complexity.

Example: One-bit predictor



Formerly used: Alpha 21064 (1bit in instruction cache), Motorola PowerPC 604

One-bit vs. Two-bit Predictors

A one-bit predictor correctly predicts a branch at the end of a loop iteration, as long as the loop does not exit.

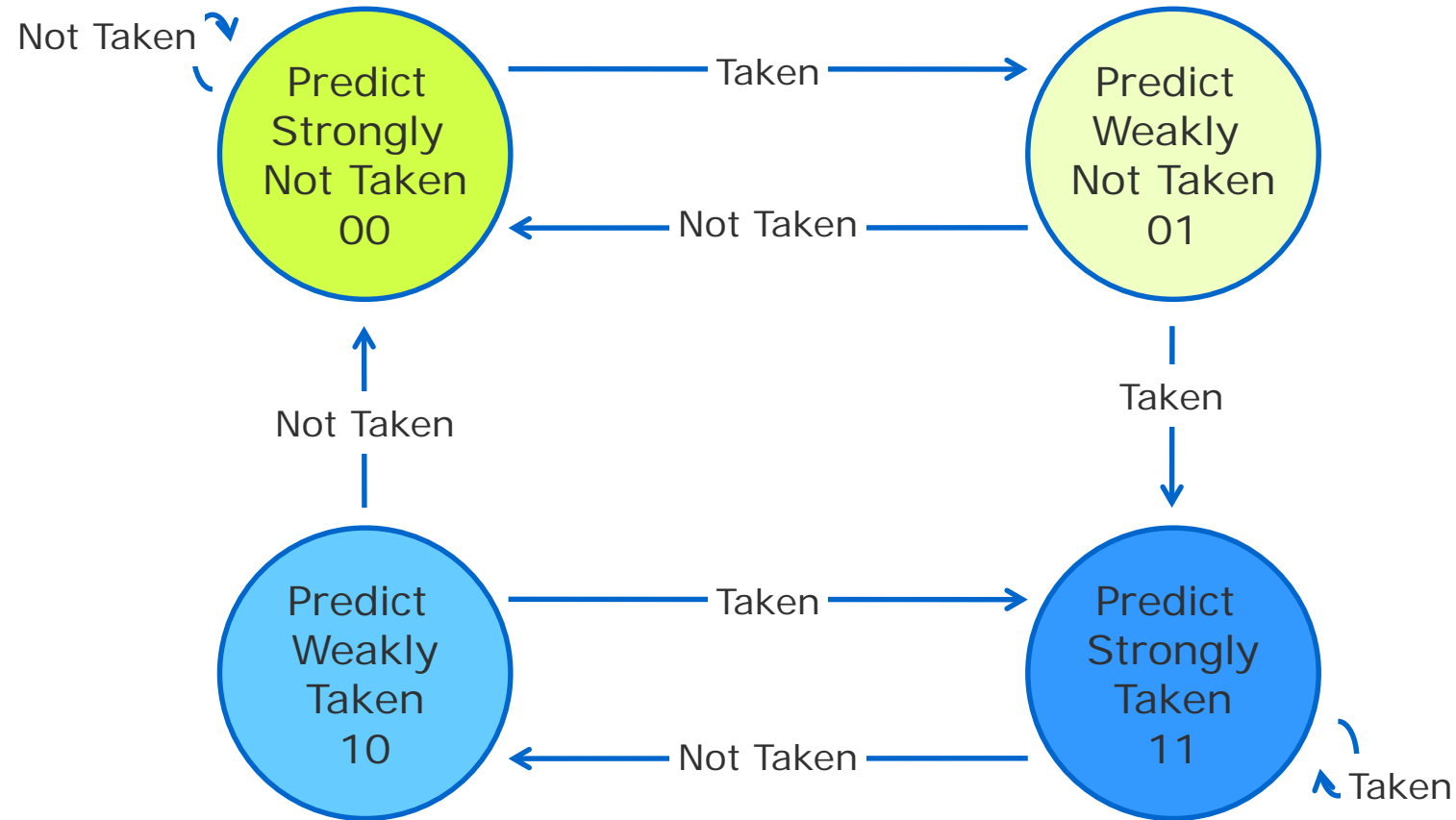
In nested loops, a one-bit prediction scheme will cause two mispredictions for the inner loop:

- One at the end of the loop, when the iteration exits the loop instead of looping again, and
- One when executing the first loop iteration, when it predicts exit instead of looping.

Such a double misprediction in nested loops is avoided by a two-bit predictor scheme.

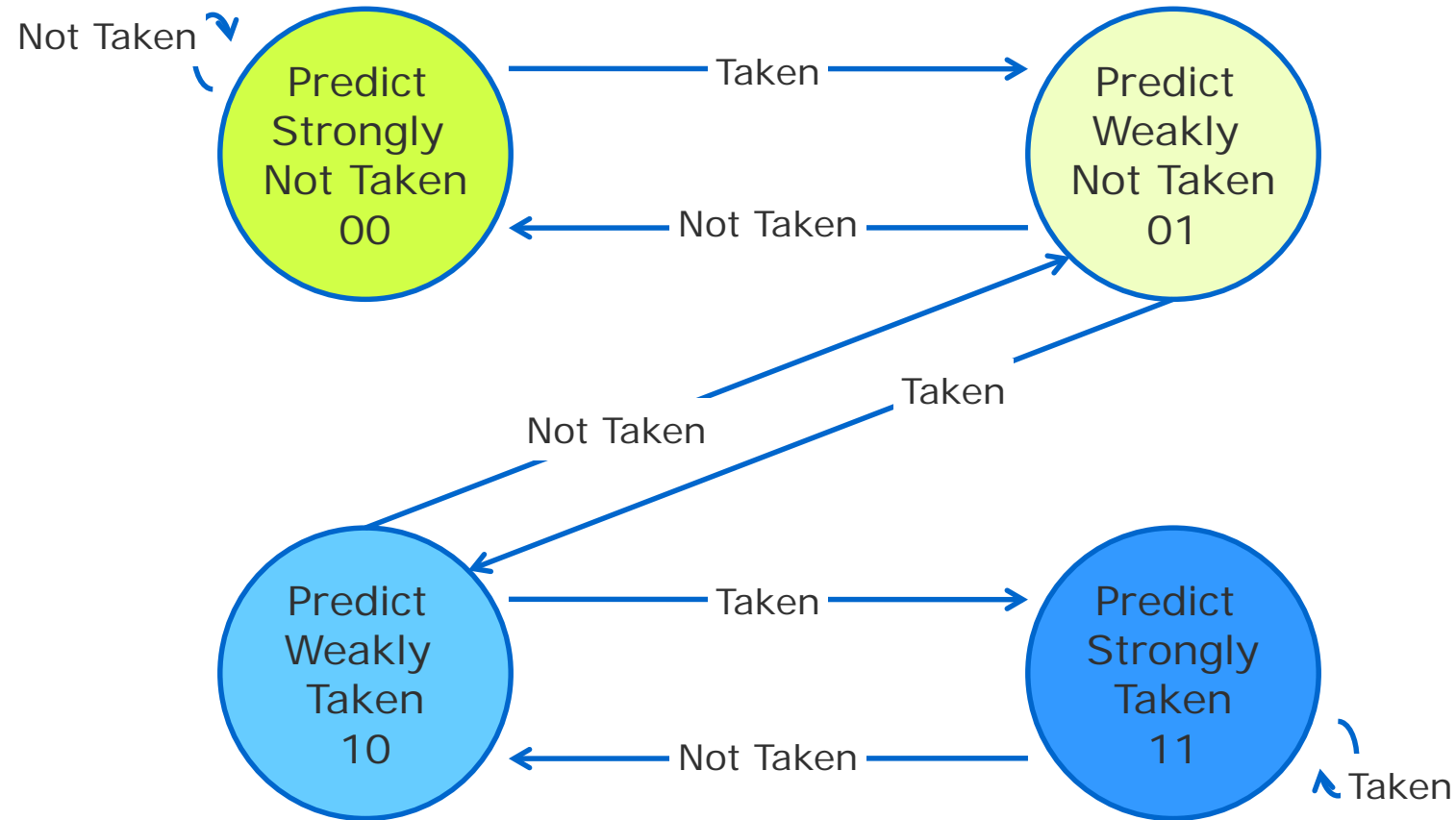
Two-bit Prediction: A prediction must miss twice before it is changed when a two-bit prediction scheme is applied.

Two-bit Predictors (Hysteresis Scheme)



Realization: Intel XScale, Sun UltraSPARC III

Two-bit Predictors (Saturation Counter Scheme)



Predicated Instructions

Provide **predicated** or **conditional instructions** and one or more predicate registers.

Predicated instructions use a predicate register as additional input operand.

The Boolean result of a condition testing is recorded in a (one-bit) predicate register.

Predicated instructions are fetched, decoded, and placed in the instruction window like non predicated instructions.

Predication Example

if ($x == 0$) {	<i>/* branch b1 */</i>
$a = b + c;$	
$d = e - f;$	
}	
$g = h * i;$	<i>/* instruction independent of branch b1 */</i>
$(Pred = (x == 0))$	<i>/* branch b1: Pred is set to true if x equals 0 */</i>
if $Pred$ then $a = b + c;$	<i>/* The operations are only performed */</i>
if $Pred$ then $d = e - f;$	<i>/* if Pred is set to true */</i>
$g = h * i;$	

Predication

Pro

- Able to eliminate a branch and therefore the associated branch prediction ➡ increasing the distance between mispredictions.
- The run length of a code block is increased ➡ better compiler scheduling.

Contra

- Predication affects the instruction set, adds a port to the register file, and complicates instruction execution.
- Predicated instructions that are discarded still consume processor resources; especially the fetch bandwidth.

Predication is most effective when control dependencies can be completely eliminated, such as in an `if-then` with a small `then` body.

The use of predicated instructions is limited when the control flow involves more than a simple alternative sequence.

Branch handling techniques and implementations

Technique	Implementation examples
No branch prediction	Intel 8086
Static prediction	
always not taken	Intel i486
always taken	Sun SuperSPARC
backward taken, forward not taken	HP PA-7x00
semistatic with profiling	early PowerPCs
Dynamic prediction	
1-bit	DEC Alpha 21064, AMD K5
2-bit	PowerPC 604, MIPS R10000, Cyrix 6x86 and M2, NexGen 586
two-level adaptive	Intel PentiumPro, Pentium II, AMD K6
Hybrid prediction	DEC Alpha 21264
Predication	Intel/HP Merced, most DSPs, ARM processors, TI TMS320C6201, ...
Eager execution (limited)	IBM mainframes: IBM 360/91, IBM 3090
Disjoint eager execution	none yet

Performance of branch handling techniques

Class	Technique	Rough Accuracy (Spec 89)
Static	always not taken	40%
	always taken	60%
	backward taken, forward not taken	65%
Software	Static analysis	70%
	Profiling	75%
Dynamic	1-bit	80%
	2-bit	93%
	two-level adaptive	95 – 97.5%

Adapted from: Dave Archer, Branch Prediction: Introduction and Survey, 2007

Pipelining basics: Summary

Hazards limit performance

- Structural hazards: need more HW resources
- Data hazards: need detection and forwarding
- Control hazards: early evaluation, delayed branch, prediction

Compilers may reduce cost of data and control hazards

- Compiler Scheduling
- Branch delay slots
- Static branch prediction

Increasing length of pipe increases impact of hazards

Pipelining helps instruction bandwidth, not latency

Multi-cycle operations (floating-point) and interrupts make pipelining harder

Pipelining

VECTOR-PIPELINING

Vektor-Pipelining

Vektorrechner einen Rechner mit pipelineartig aufgebautem/n Rechenwerk/en zur Verarbeitung von Arrays von Gleitpunktzahlen.

Vektor = Array (Feld) von Gleitpunktzahlen (\neq math. Vektor!)

Im Gegensatz zur Vektorverarbeitung wird die Verknüpfung einzelner Operanden als **Skalarverarbeitung** bezeichnet.

Ein Vektorrechner enthält neben der Vektoreinheit auch noch eine oder mehrere **Skalareinheiten**. Dort werden die skalaren Befehle ausgeführt, d.h. Befehle, die nicht auf ganze Vektoren angewendet werden sollen.

Die Vektoreinheit und die Skalareinheit(en) können parallel zueinander arbeiten, d.h. Vektorbefehle und Skalarbefehle können parallel ausgeführt werden.

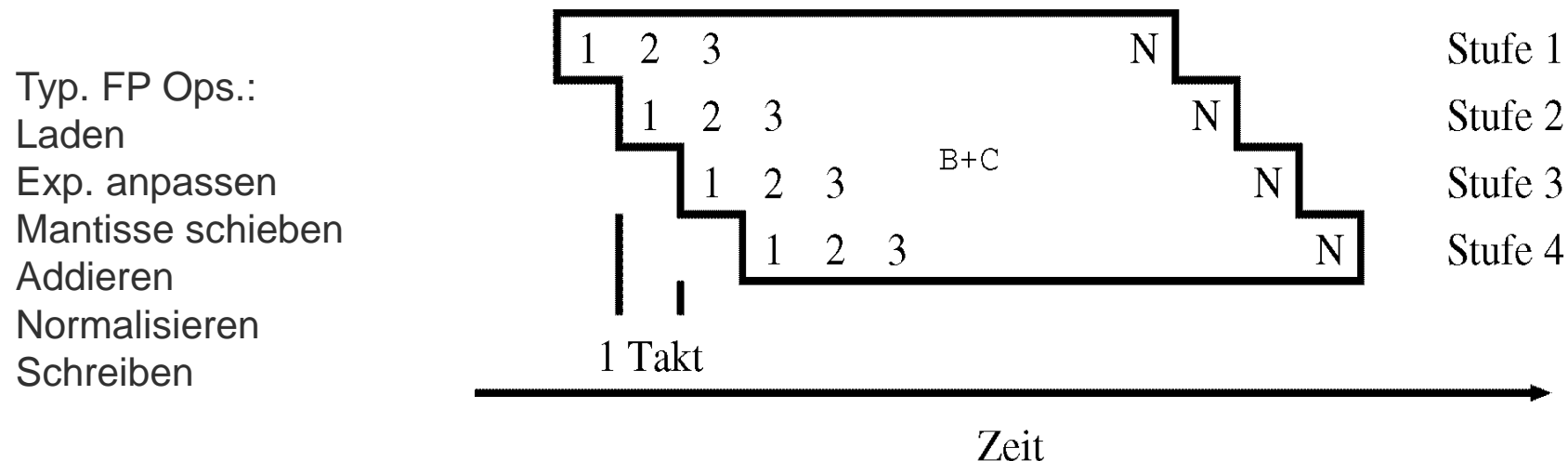
Vielfach genutzt z.B. in Grafikkarten und als Erweiterungen zum Befehlssatz (SSE, AltiVec, ...)

Beispiel Addition

$$A(J) = B(J) + C(J), \quad J = 1, 2, \dots, N$$

Hier werden die Vektoren B und C , d.h. die Felder $B(1), \dots, B(N)$ und $C(1), \dots, C(N)$, mit *einem* Befehl komponentenweise addiert und im Ergebnisvektor A abgespeichert.

Die Vektoren werden dabei *sequentiell* und *überlappt* abgearbeitet, d.h. zuerst wird die Berechnung $B(1)+C(1)$ gestartet, dann $B(2)+C(2)$, usw.



Besonderheit einer Vektorpipeline

Die Pipeline-Verarbeitung wird mit einem Vektorbefehl für zwei Felder von Gleitpunktzahlen durchgeführt.

Die bei den Gleitpunkteinheiten skalarer Prozessoren nötigen Adressberechnungen entfallen.

Bei ununterbrochener Arbeit in der Pipeline kann man nach einer gewissen Einschwingzeit bzw. Füllzeit, die man braucht, um die Pipeline zu füllen, mit jedem Pipeline-Takt ein Ergebnis erwarten.

Dabei ist die Taktdauer durch die Dauer der längsten Teilverarbeitungszeit zuzüglich der Stufentransferzeit gegeben.

Man kann sich z.B. die EXE-Stufe in Prozessoren für FP-Befehle als eine solche Vektorpipeline vorstellen.

Verkettung (Chaining)

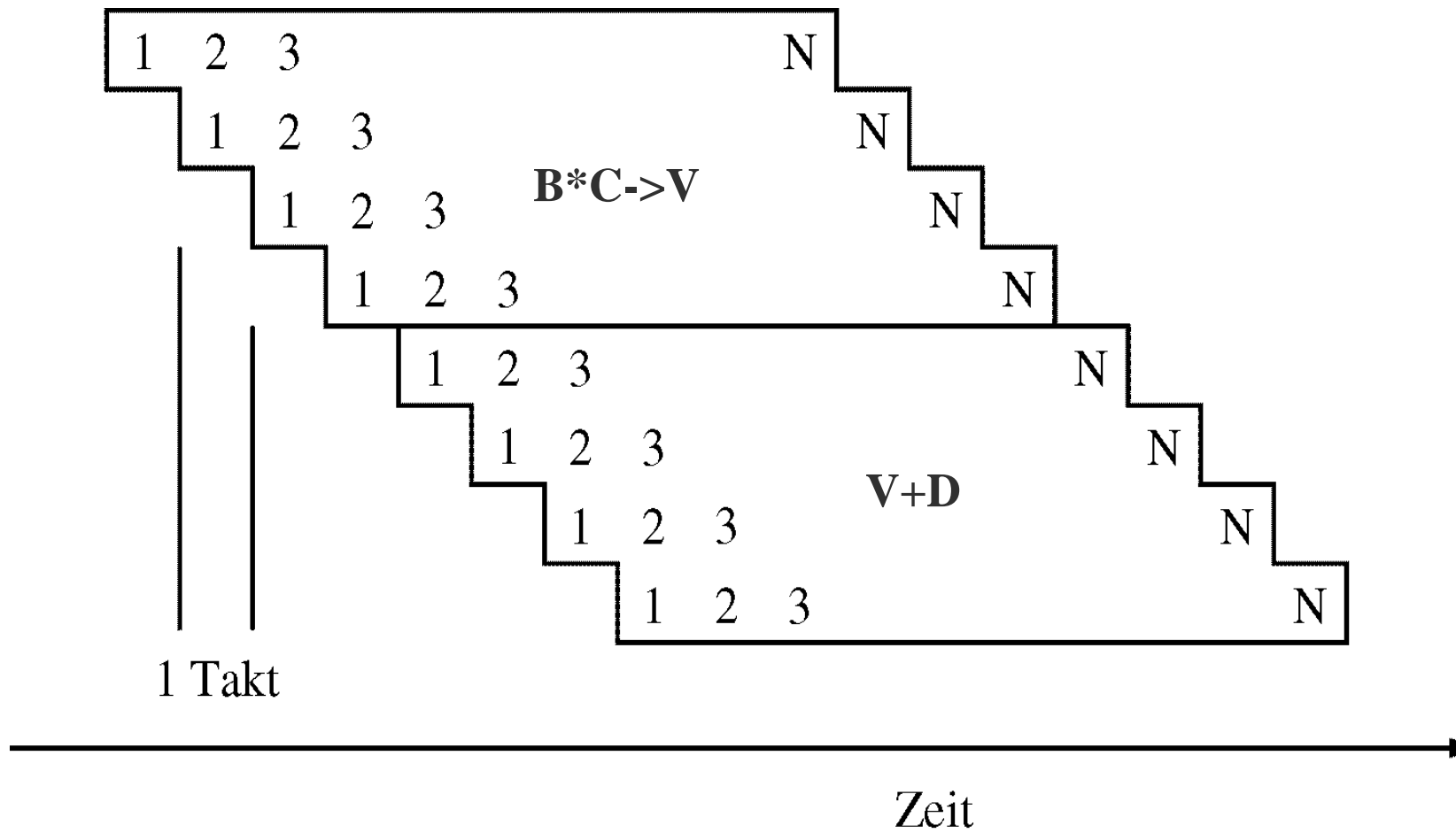
Verkettung: Das Pipeline-Prinzip kann auch auf eine Folge von Vektoroperationen erweitert werden.

Zu diesem Zweck werden die (spezialisierten) Pipelines miteinander verkettet,

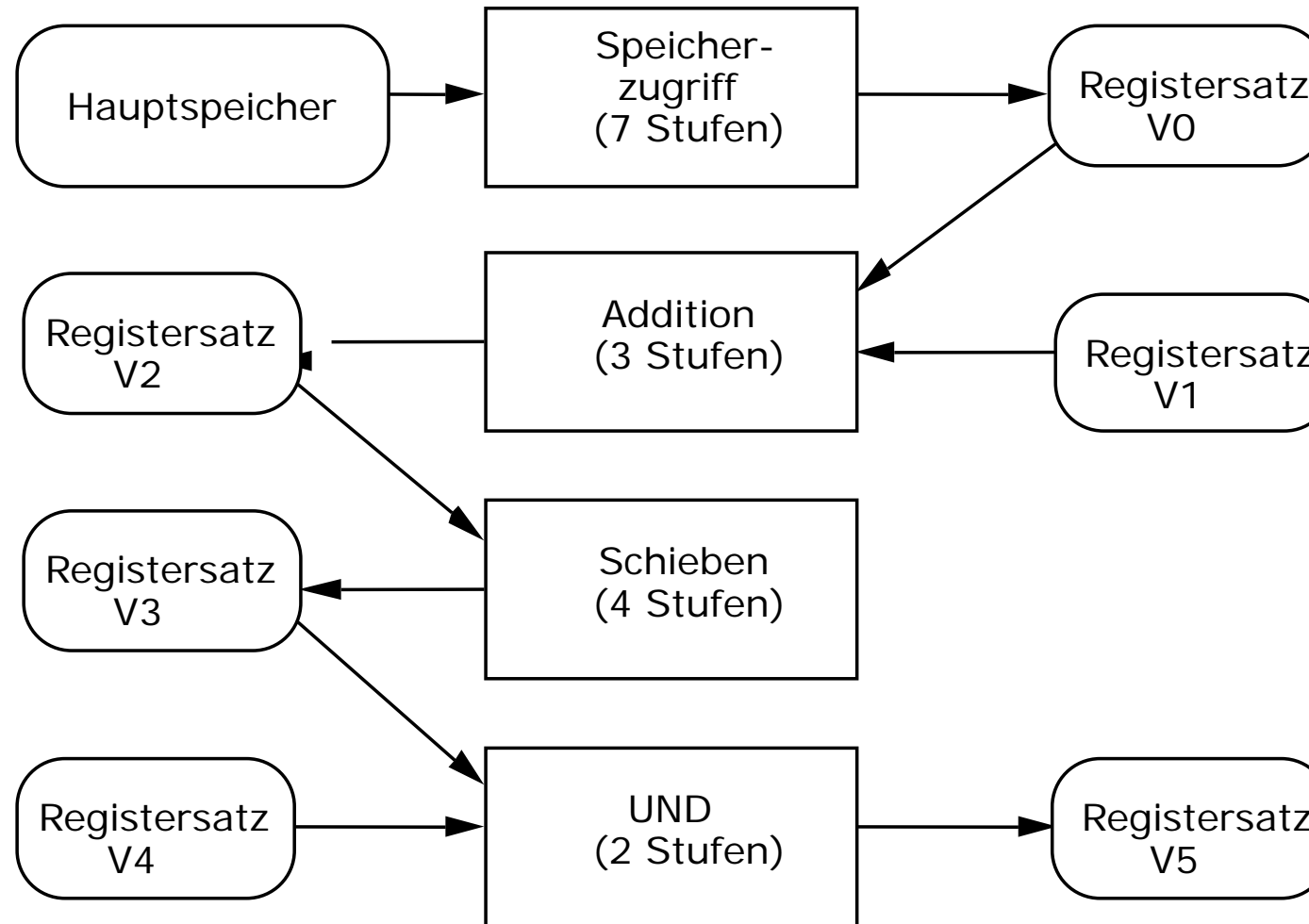
d.h. die Ergebnisse einer Pipeline werden sofort der nächsten Pipeline zur Verfügung gestellt.

Beispiel: Pipelineverkettung

$$B(J)*C(J)+D(J), J=1,2,...,N$$



Verkettung von vier Pipelines (aus Cray 1)



Simple Example

Component-wise multiplication of vectors

- $A(i) = B(i) * C(i)$, $i = 1, \dots, 100$

Standard processor

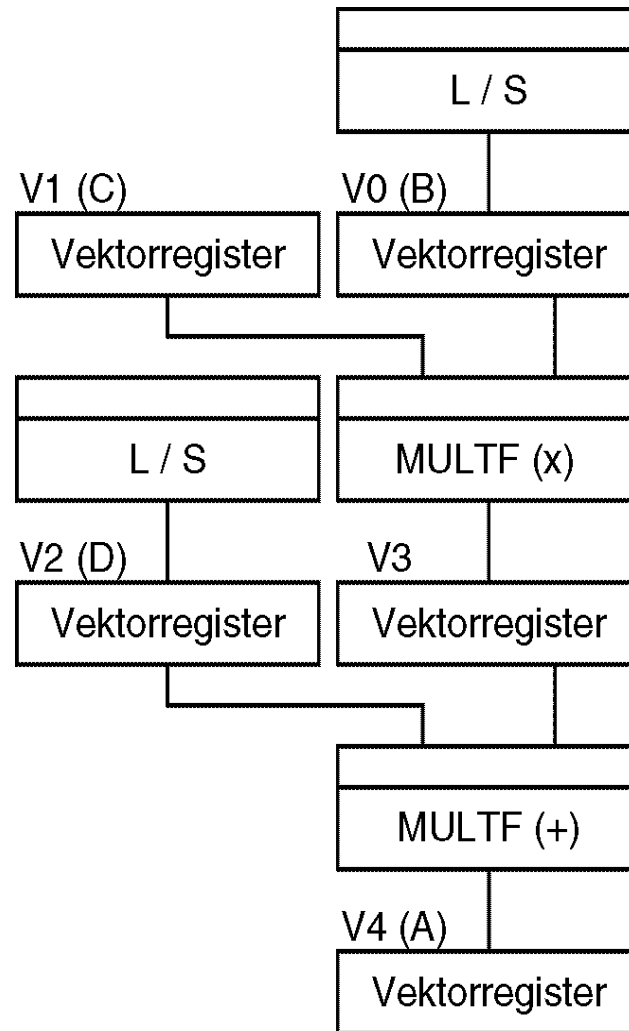
```
i = 1
b = b(i)
c = c(i)
a = b * c
a(i) = a
i++
if i < 101
```



Vector processor

```
vload b, 1, 100
vload c, 1, 100
vmult a, b, c
vstore a, 1, 100
```

Beispiel: Pipelineverarbeitung von $A(I)=B(I)*C(I)+D(I)$



Übersetzung in Vektorbefehle
(Vektor c sei in v1 geladen):

`VL B, V0 ; lade B in das Vektorregister V0`

`VL D, V2 ; lade D in das Vektorregister V2`

`VM V0, V1, V3 ; V0 * V1 -> V3`

`VA V2, V3, V4 ; V2 + V3 -> V4`

`VST V4, A ; speichere Vektorregister V4 in A`

`VMA V0,V1,V2,V4`

(Triadischer Vektorbefehl *vector multiply and add*)

SUPERSCALAR PROCESSORS

Superscalar processors

Definition

- **Superscalar** machines are distinguished by their ability to (dynamically) issue multiple instructions each clock cycle from a conventional linear instruction stream.

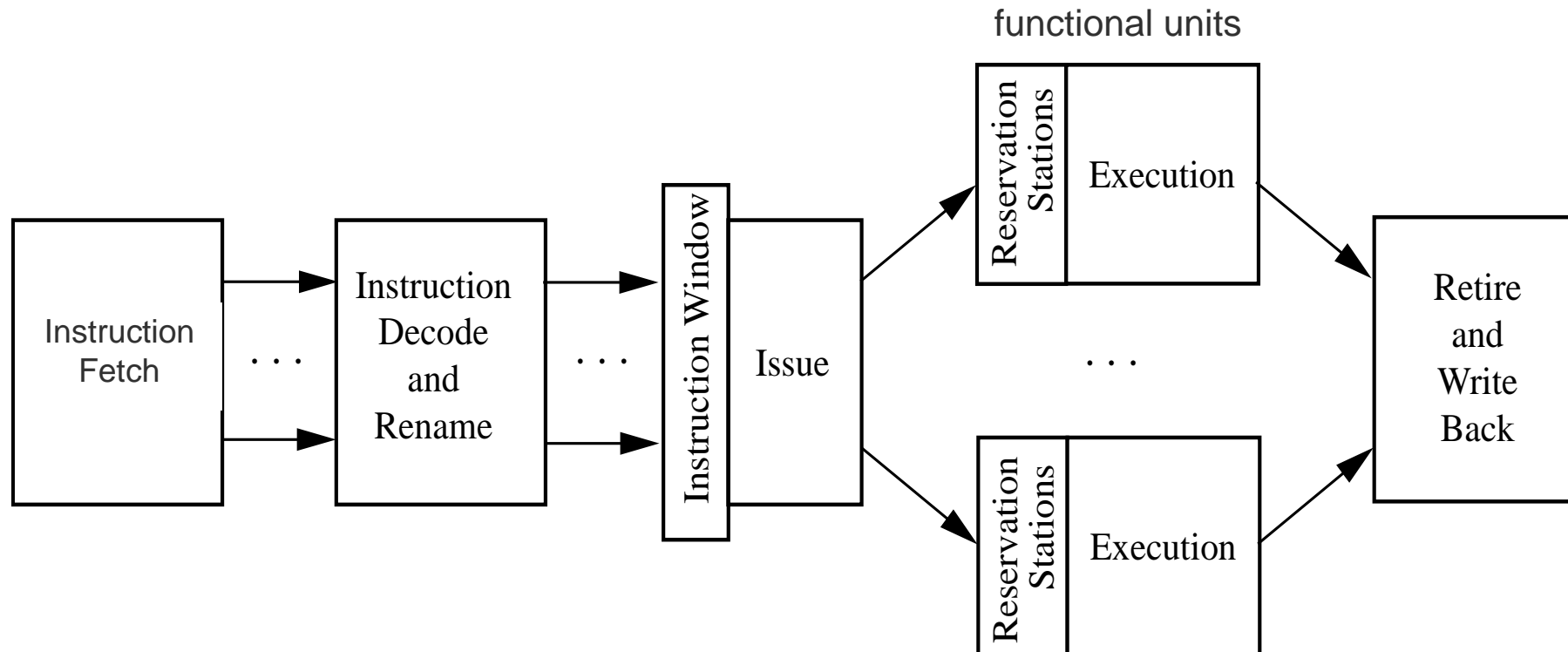
Consequence

- value of CPI (cycles per instructions) $\ll 1.0$ possible!

Superscalar Pipeline

Instructions in the instruction window are **free from control dependences** due to branch prediction, and **free from name dependences** due to register renaming.

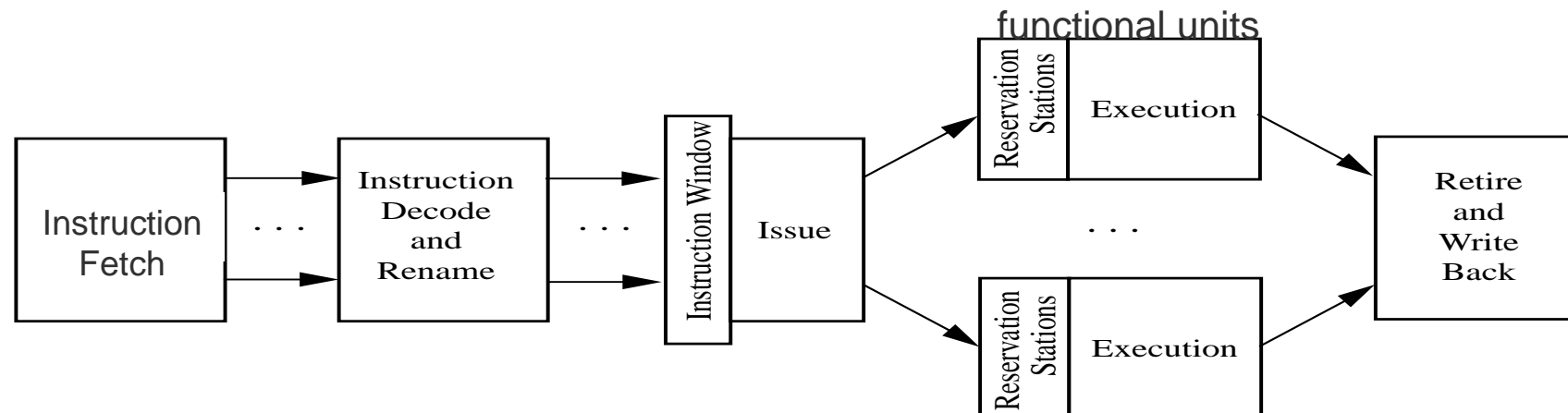
So, only **(true) data dependencies** and **structural conflicts** remain to be solved.



Sections of a Superscalar Pipeline

The ability to issue and execute instructions out-of-order partitions a superscalar pipeline in three distinct sections

- **in-order section** with the instruction fetch, decode and rename stages - the issue is also part of the in-order section in case of an in-order issue,
- **out-of-order section** starting with the issue in case of an out-of-order issue processor, the execution stage, and usually the completion stage, and again an
- **in-order section** that comprises the retirement and write-back stages.



Fetch, decode, rename

Fetch

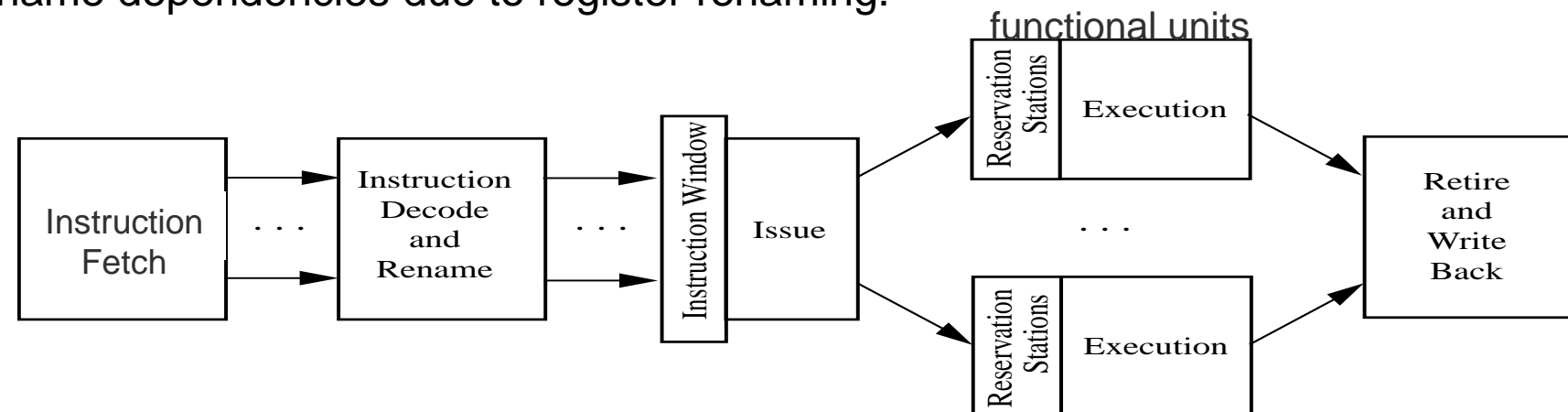
- Get a bunch of commands

Decode

- Decode the new instruction

Rename

- Externally visible register are mapped to internal shadow registers
- ➔ Avoid WAW/WAR-conflicts
- ➔ Mapping stored in a rename map (Intel: alias table)
- ➔ core execution units free from name dependencies due to register renaming.



Issue

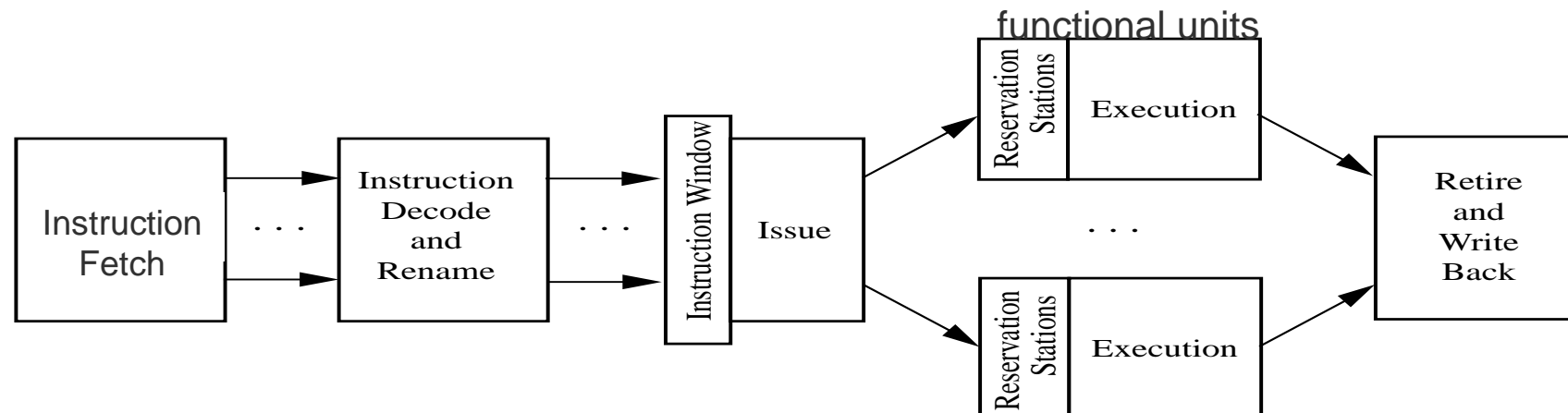
The **issue logic** examines the waiting instructions in the instruction window and simultaneously assigns (**issues, dispatches**) a number of instructions to the functional units (FUs) up to a maximum issue bandwidth.

Several instructions can be issued simultaneously (the **issue bandwidth**).

The program order of the issued instructions is stored in the reorder buffer.

Instruction issue from the instruction buffer can be:

- **in-order** (only in program order) or **out-of-order**

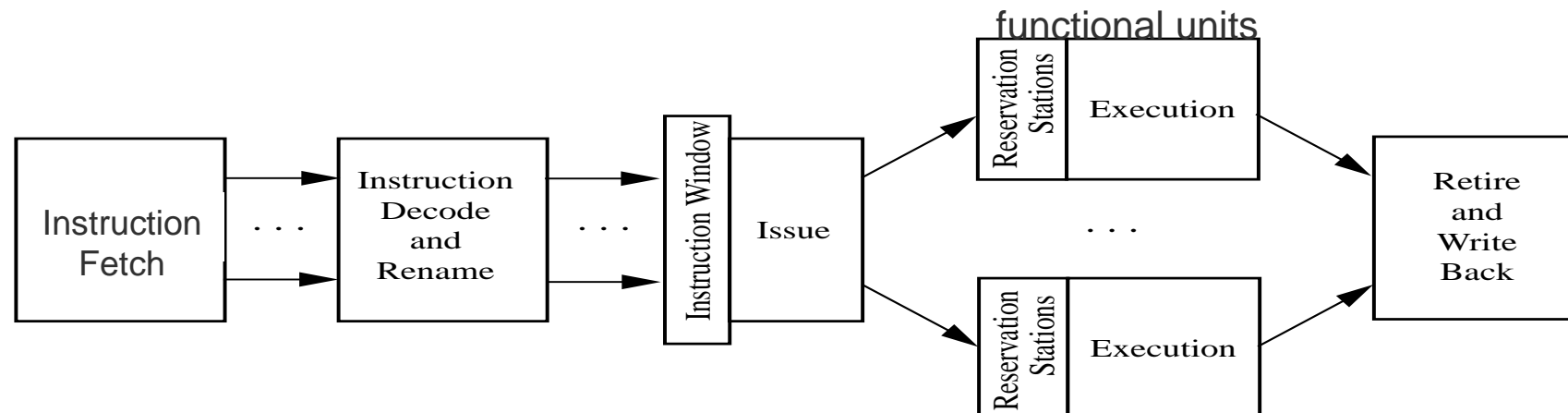


Reservation Station(s)

A **reservation station** is a buffer for a single instruction with its operands.

Reservation stations can be central to a number of FUs or each FU has one or more own reservation stations.

Instructions await their operands in reservation stations.



Dispatch

An instruction is said to be **dispatched** from a reservation station to the FU when all operands are available, and execution starts.

If all its operands are available during issue and the FU is not busy, an instruction is immediately dispatched, starting execution in the next cycle after the issue.

So, the dispatch is usually not a pipeline stage.

An issued instruction may stay in the reservation station for zero to several cycles.

Dispatch and execution is performed *out of program order*.

Other authors interchange the meaning of *issue* and *dispatch* or use different semantic.

Completion

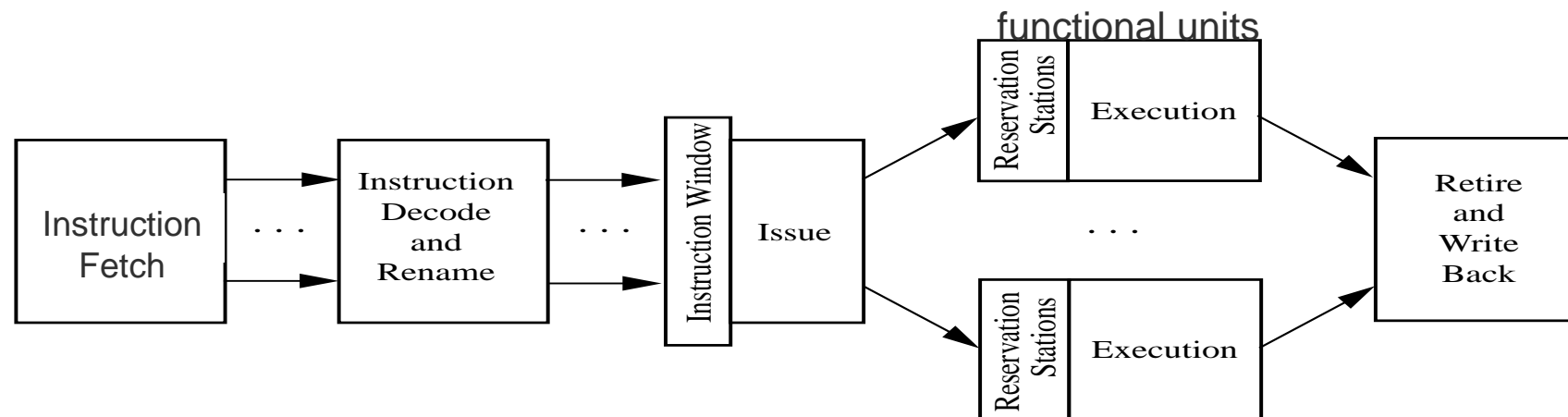
When the FU finishes the execution of an instruction and the result is ready for forwarding and buffering, the instruction is said **to complete**.

Instruction completion is out of program order.

During completion the reservation station is freed and the state of the execution is noted in the **reorder buffer**.

The state of the reorder buffer entry can denote an interrupt occurrence.

The instruction can be completed and still be speculatively assigned, which is also monitored in the reorder buffer.



Commitment

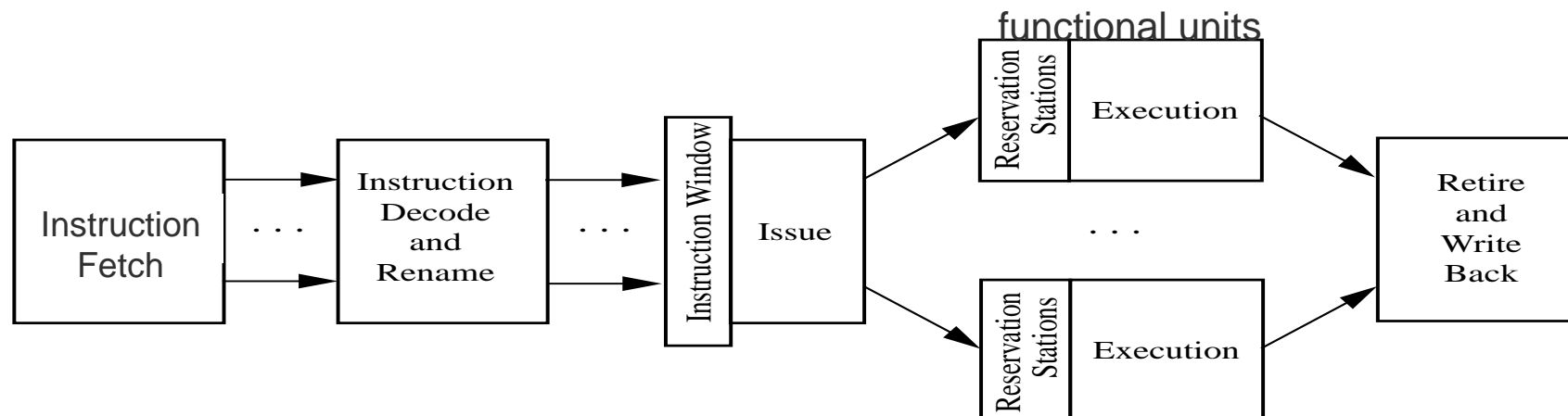
After completion, operations are committed in-order.

An instruction can be **committed**:

- if all previous instructions due to the program order are already committed or can be committed in the same cycle,
- if no interrupt occurred before and during instruction execution, and
- if the instruction is no more on a speculative path.

By or after commitment, the result of an instruction is made permanent in the architectural register set,

- usually by writing the result back from the rename register to the architectural register.



Precise Interrupt / Precise Exception

If an interrupt occurred, all instructions that are in program order before the interrupt signaling instruction are committed, and all later instructions are removed.

Precise exception means that all instructions before the faulting instruction are committed and those after it can be restarted from scratch.

Depending on the architecture and the type of exception, the faulting instruction should be committed or removed without any lasting effect.

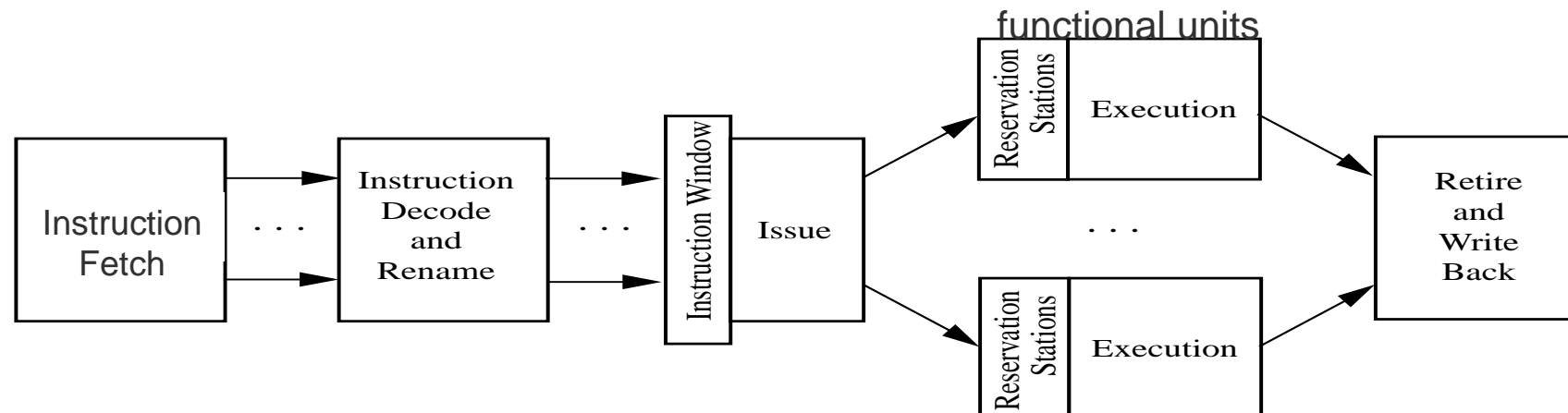
Retirement

An instruction **retires** when the reorder buffer slot of an instruction is freed either

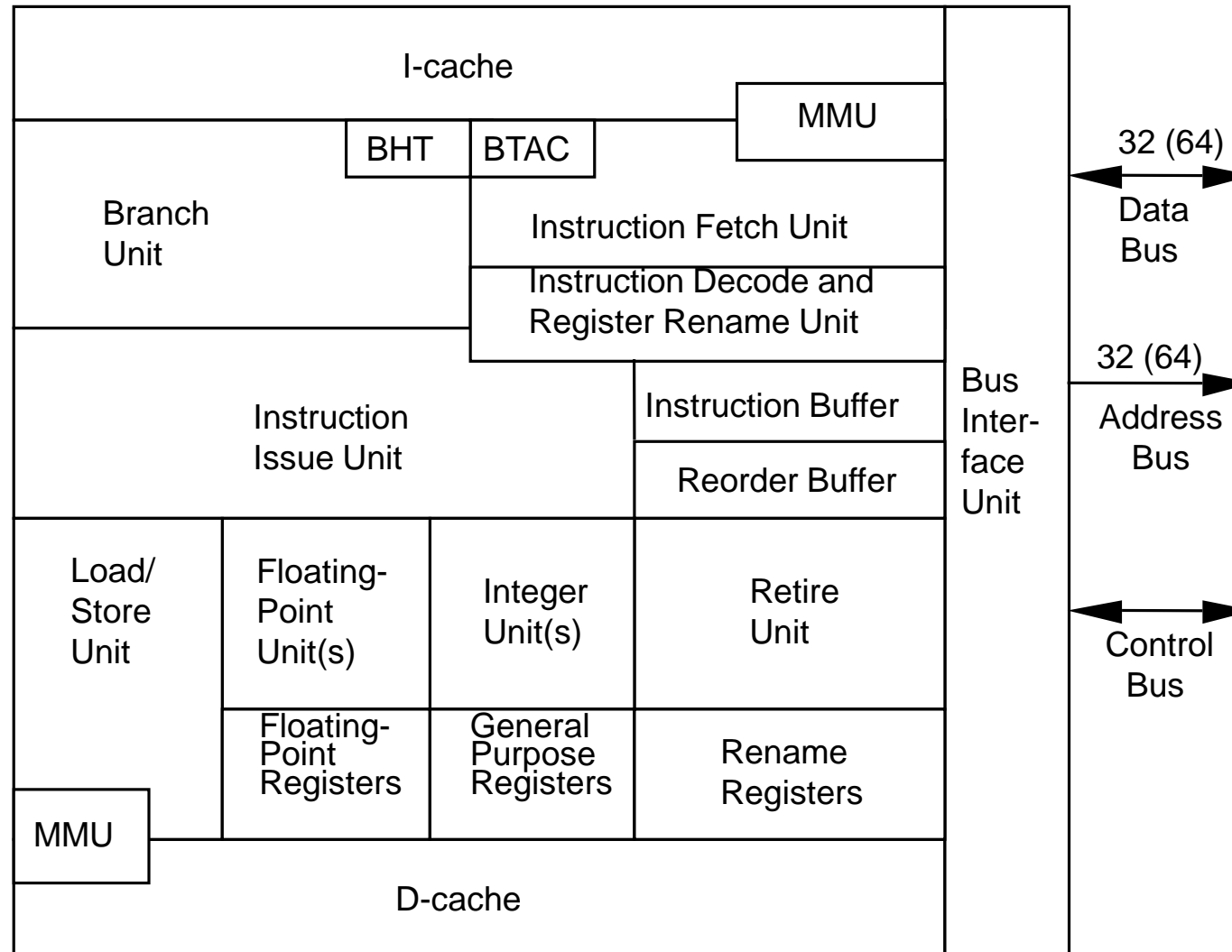
- because the instruction commits (the result is made permanent) or
- because the instruction is removed (without making permanent changes).

A result is made permanent by copying the result value from the rename register to the architectural register.

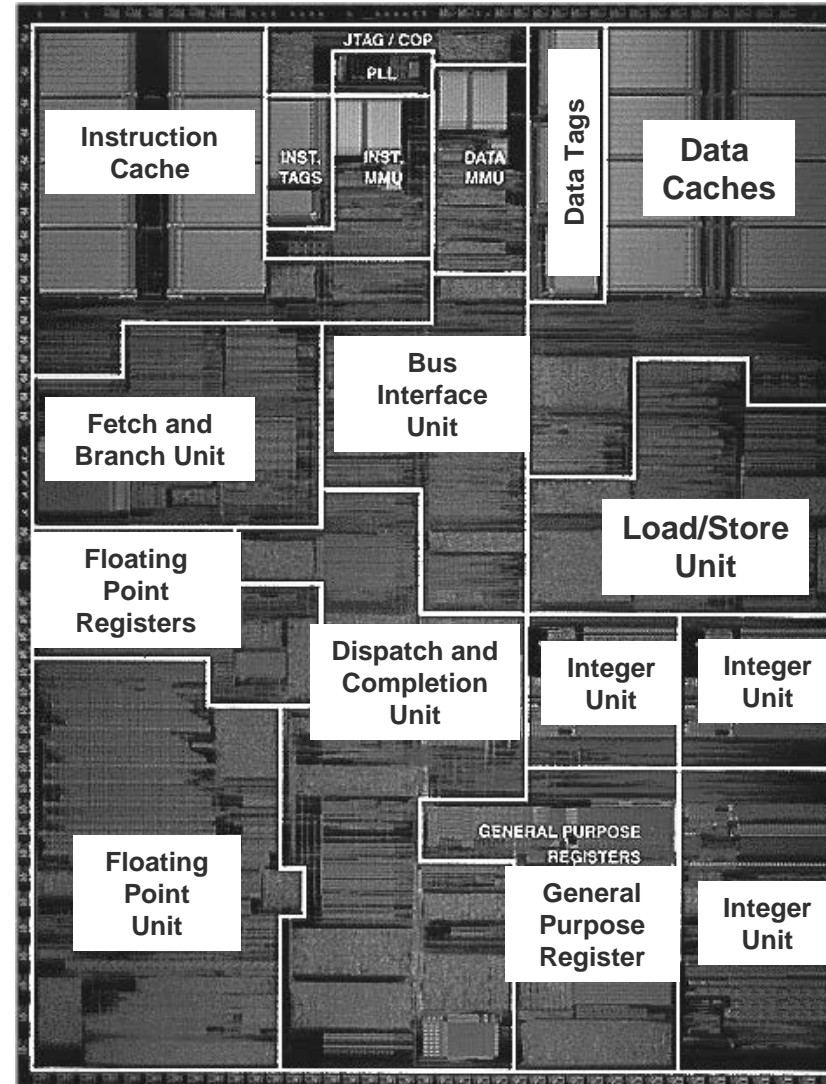
- This is often done in an own stage after the commitment of the instruction with the effect that the rename register is freed one cycle after commitment.



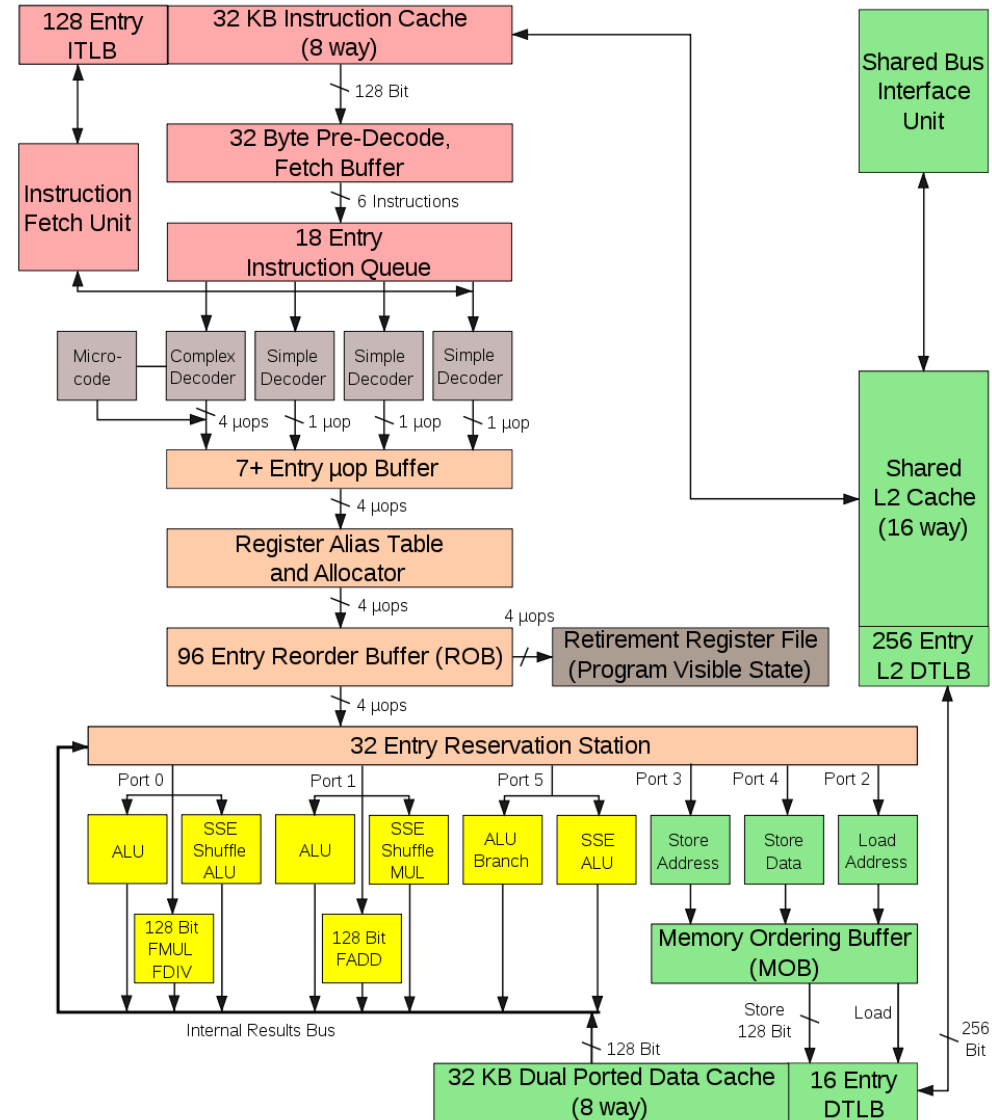
Example Components of a Superscalar Processor



Floor plan of the PowerPC 604



Example: Intel Core 2 Architecture



(Future) Processor Architecture Principles

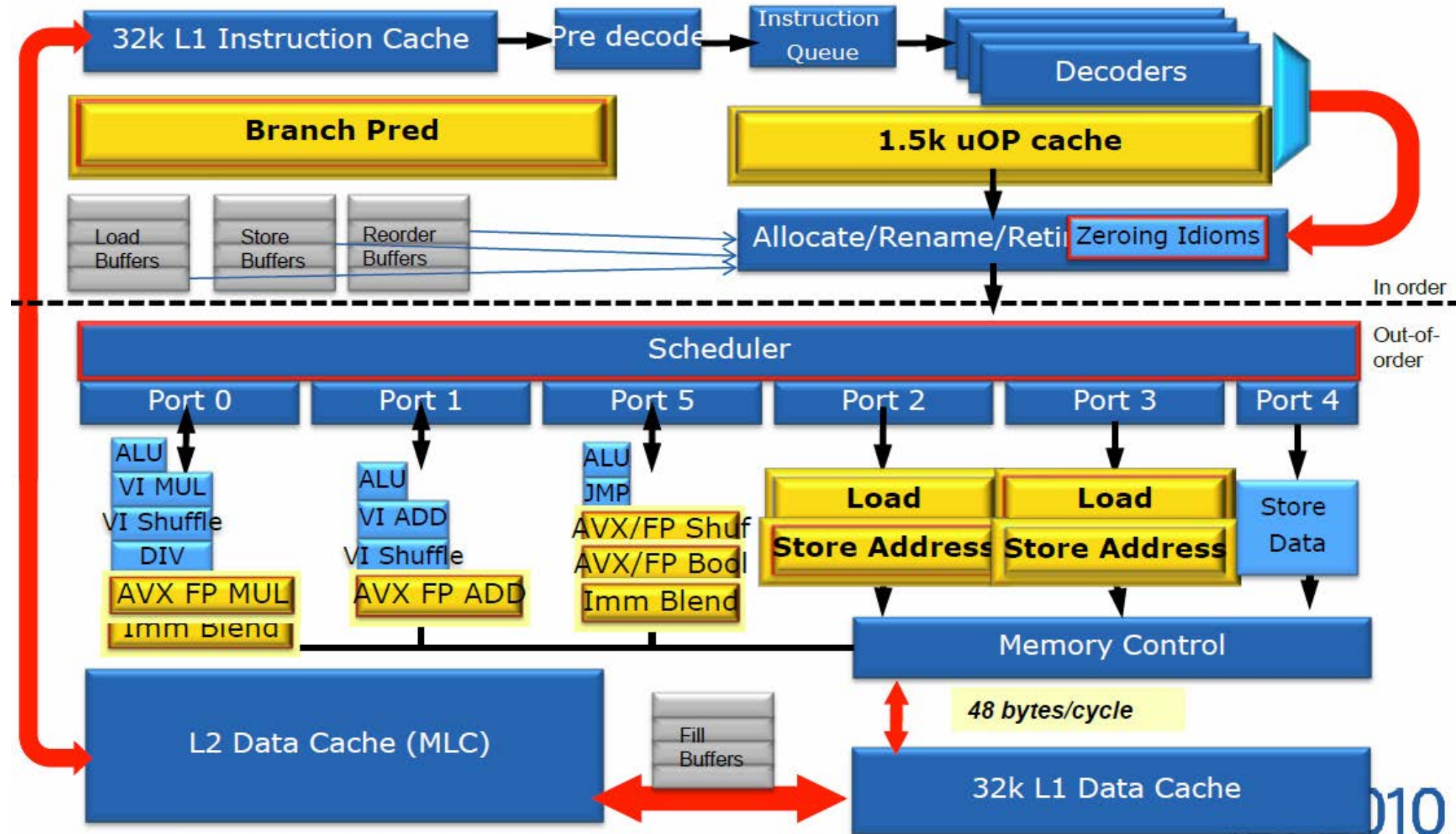
Speed-up of a single-threaded application

- Simultaneous Multithreading
- Advanced superscalar
- Superspeculative
- Multiscalar processors

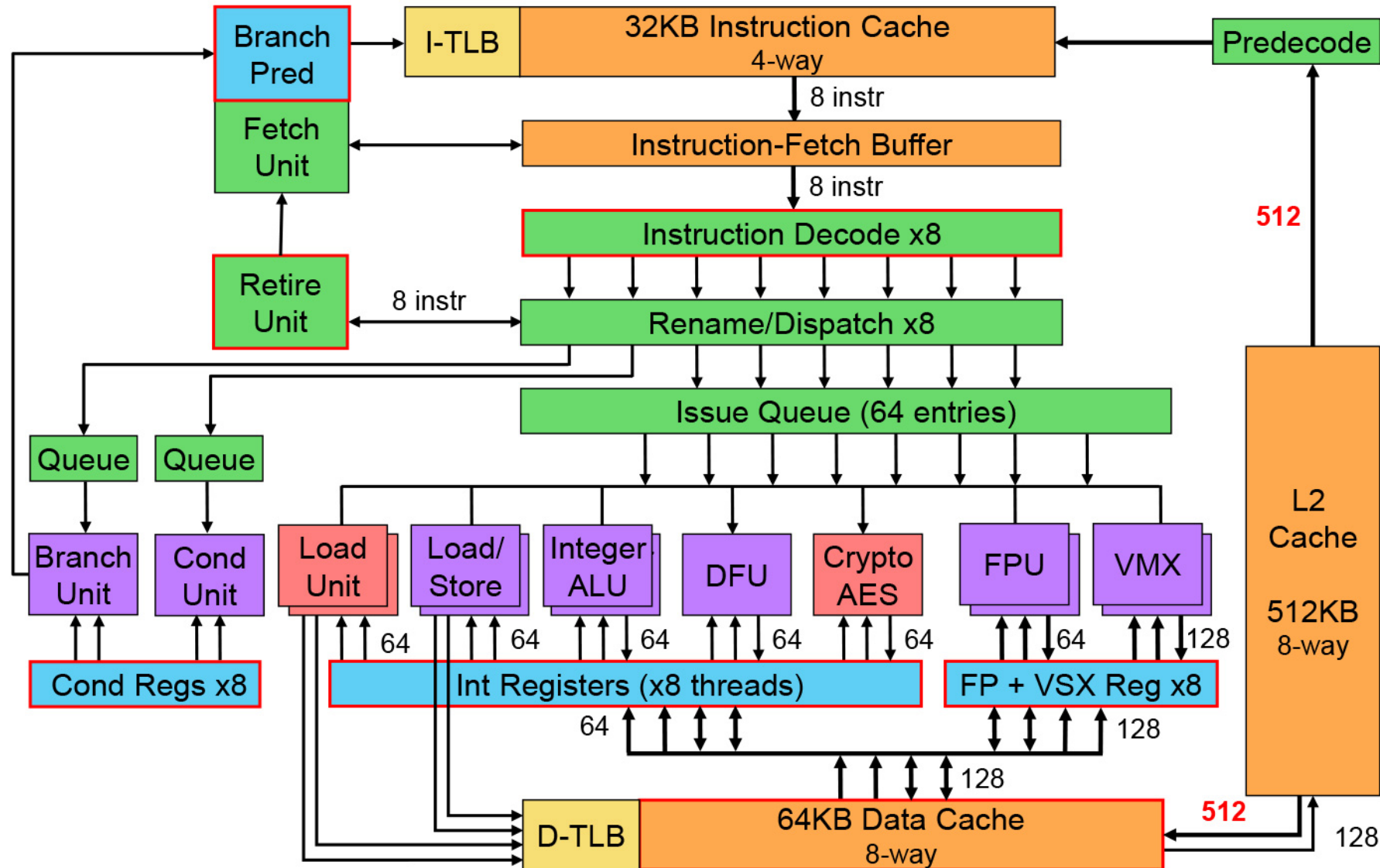
Speed-up of multi-threaded applications

- Chip multiprocessors (CMPs)
- Simultaneous multithreading

Example: Intel Sandy Bridge



Example: Power8



Summary

Aufbau eines Mikroprozessors

- Steuer-, Rechen-, Adresswerk, Registersatz, Systembusschnittstelle

Leistungssteigerung von Rechensystemen

- Technologische Maßnahmen
- Strukturelle Maßnahmen

Pipelineverarbeitung

- Methodik
- Hazards
- Vektor-Pipelining

Superskalare Prozessoren