Rechnerarchitektur

Einführung in die Informatik & Rechnerarchitektur (EIR1/EIF1)

Erik Pitzer

SE & MBI – FH Hagenberg – WS 2025/26

Moderne Architekturen

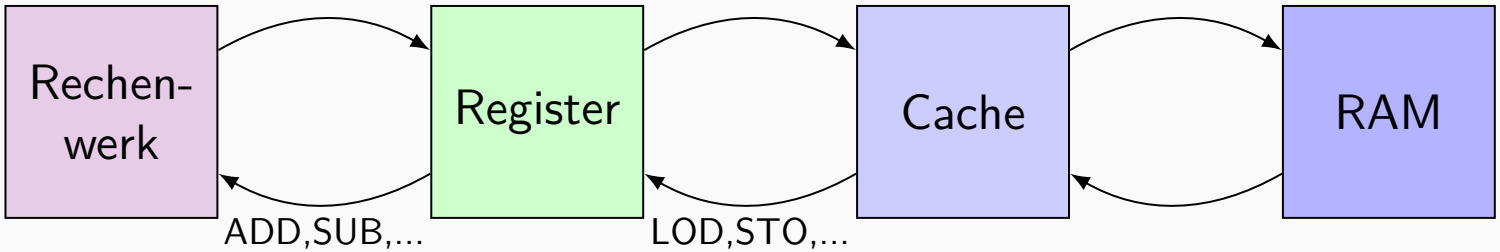
Complex Instruction Set Computing (CISC)

- Früher: Assemblerprogrammierung direkt
  - viele und umfangreiche Befehle
  - viele Adressierungsarten
  - wenige Register
  - weniger Speicherzugriffe (auf Code)
    - langsamer, kleiner, teurer RAM
- Später: höhere Sprachen mit Compiler (C, PASCAL, ...)
  - 20% der Instruktionen brauchen 60% des ROMs, aber nur 0.2% der Ausführungszeit[1]
  - komplexe Operationen kaum benutzt
  - häufig nur sehr kurze Instruktionen (1 Mikrobefehl)
  - größere, schnellere Speicher durch Caching
  - Pipelining

1

Reduced Instruction Set Computing (RISC)

- Anfang der 80er Jahre
- IBM RS/6000, PowerPC, DEC Alpha, SUN Sparc, HP PA
- Einfachere Befehle, teilweise weniger Befehle
- keine Mikroprogramme, direkte Hardwareimplementierung
- großteils ein Befehl pro Takt
- kein direkter Speicherzugriff von Rechenoperationen
- viele Register (32-256), mehrere “Registerfenster”
- Optimierung durch Compiler



RISC (1/2)

- Sprünge brauchen manchmal zwei Takte
  - nach Sprung entweder weiterer Befehl oder `NOP`
- z.T. auch komplexere Befehle
  - z.B. ARMv7

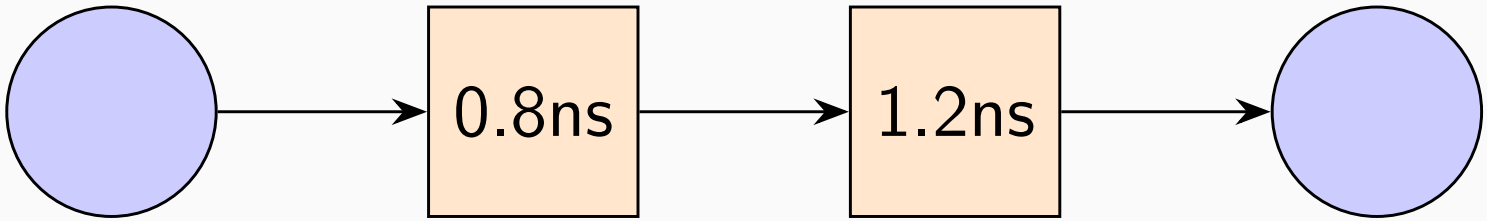
```
;; add and shift z.B. R0 := R0 * 5 ohne Multiplikation
ADD R0, R0, R0, LSL #2 ;; R0 := R0 + (R0 << 2)

;; multiply and accumulate
MLA R0, R1, R2, R3 ;; R0 := R3 + R1 * R2

;; load multiple (z.B. 54 byte), increment after
LDMIA R10!, { R0-R3, R12 }
  ;; (R0, ... R3, R12) ← [R10, ... R10+4]
  ;; R10 ← R10 + (5 * 4 Byte)
```

RISC (2/2)

- kleinere Chips
- weniger Stromverbrauch
- sehr weit verbreitet, z.B. in Smartphones
- “RISC im CISC”
  - moderne Intel Prozessoren seit P6 (Pentium Pro)
  - äußerlich CISC mit Mikrocode
  - enthalten RISC Kerne
  - kompatible zu alten CISC Modellen
- keine neue kommerzielle CISC ISA seit mehr als 30 Jahren
- RISC-V offener Standard von UC Berkeley
  - Krste Asanovic & David Patterson

Pipelining Idee (1/2)

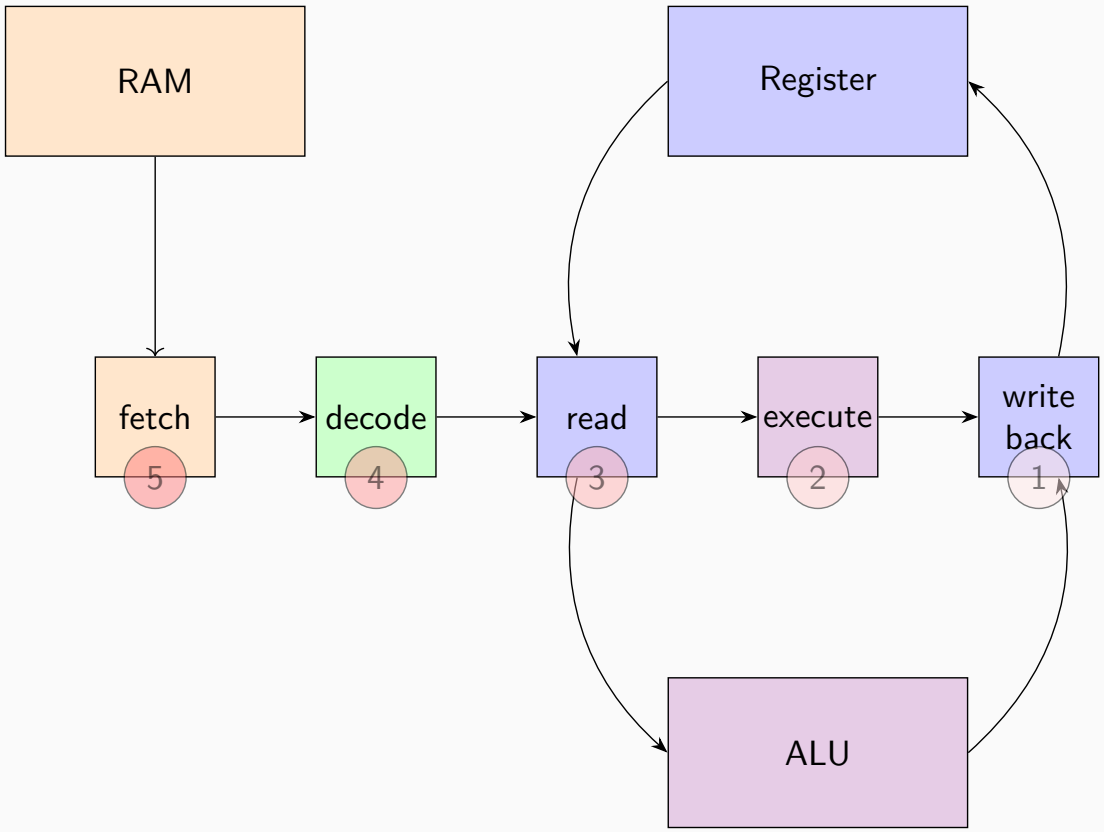
- Fließbandverarbeitung
- parallele Nutzung der Hardware
  - erhöht Durchsatz (gut), erhöht Latenz (nicht so gut)



- Latenz: 0.8ns + 1.2ns = 2ns
- Durchsatz: 1 Operation pro 1.2ns

Pipelining Idee (2/2)

- sobald ein Befehl in die nächste Phase wechselt, kann sofort der nächste Befehl nachkommen

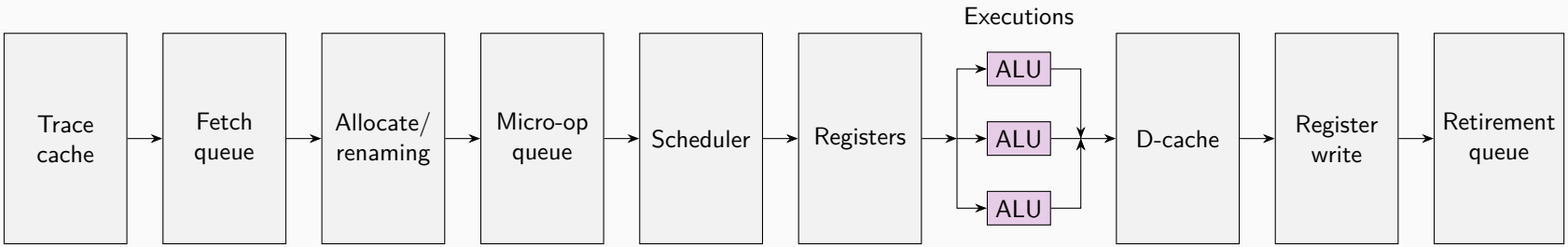


Pipelining: Probleme

- Pipeline kann verzögert werden (engl. *stall*)
  - z.B. wenn ALU Operation länger dauert
- Datenabhängigkeiten können auftreten
  - z.B. wenn Ergebnis von vorherigem Befehl benötigt wird
- Bei Verzweigungen und Sprüngen
  - Sprungziel ist nach Dekodierung noch nicht bekannt
- Bei komplexen Pipelines gibt es auch Ressourcenkonflikte (engl. *hazard*)

Superskalarität

- einzelner CPU-Kern erhält mehrere Rechenwerke, die parallel mit Operationen “gefüttert” werden können
- dadurch können pro Taktphase potentiell mehrere Befehle gleichzeitig ausgeführt werden
- z.B. Pentium 4 (“Netburst”) Architektur
- Hyper-Threading versucht dies auszunutzen, da zwei unabhängige Programme parallel abgearbeitet werden

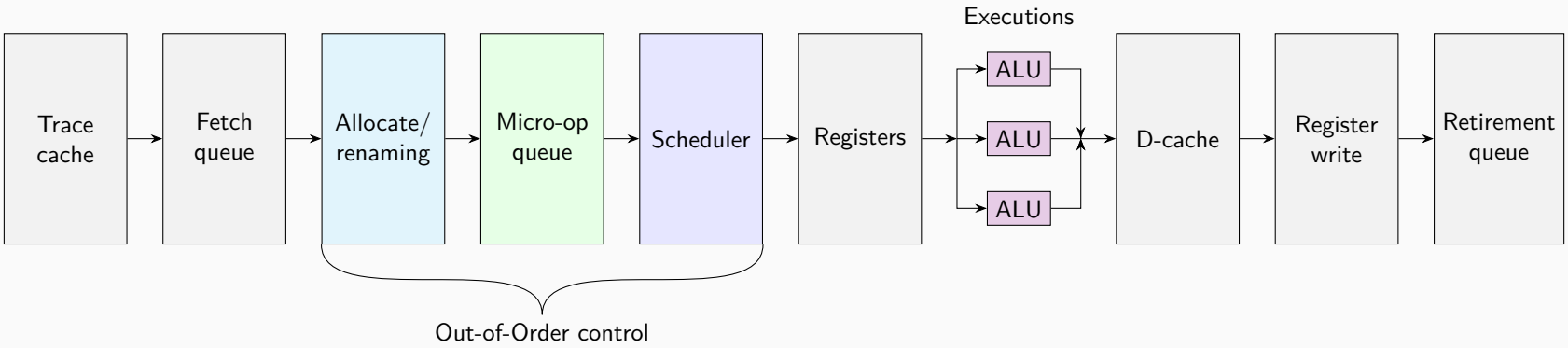


Superskalarität: Implementierung

- Möglichkeit 1: In-Order Execution
  - Vorbereitung durch Compiler
  - möglichst viele Kerne gleichzeitig nutzen, z.B.
    - very long instruction word (VLIW) oder
    - explicitly parallel instruction computer (EPIC)
    - Intel/HP Itanium Prozessor
- Möglichkeit 2: Out-of-Order Execution
  - Verteilen der Operationen durch die CPU selbst
  - sonst Verzögerung (*stall*)

Superskalarität: Branch Prediction

- Um Pipeline möglichst gut zu füllen, spekulative Ausführung obwohl nächster Befehl noch nicht bekannt, z.B.
  - Sprungzielspeicher (branch target buffer) oder
  - Sprungvorhersage (branch history table)



Aktuelle Probleme: “Meltdown” & “Spectre”

- basieren auf spekulativer Ausführung & Caching
- nutzen Timing-Angriffe (Side-Channel Attack)
- ähnliche, teilweise überlappende Lücken
  - Meltdown
    - erlaubt Zugriff auf geschützten Hauptspeicherbereich
    - vergleichsweise schnell (bis zu 503KB/s)
    - nutzt Cache-Timing
    - betrifft sehr viele Systeme
  - Spectre
    - erlaubt Zugriff auf beliebigen Speicher
    - muss vorher “trainiert” werden
    - nutzt allgemeines Timing
    - viel schwieriger zu beheben

Meltdown

- durch superskalare Ausführung und Caching wird auf geschützten Speicher zugegriffen

```
data:   resb 256*4096 ;; 256 Arrays, Größe 4096 (Cache-Line)
target: const 0x123   ;; geschützte Adresse (kein Zugriff)
_start:
  MOV eax, [target]   ;; dieser Zugriff ist nicht erlaubt
  MOV ebx, 4096       ;; ... restliche Befehle werden
  MUL ebx             ;; ... spekulativ ausgeführt
  MOV esi, eax        ;; ... ESI=4096*EAX
  MOV ebx, [data*esi] ;; Zugriff auf erlaubtes Feld
  ...                 ;; basierend auf geschütztem Wert
;; XXX unerlaubter Zugriff wird erkannt
;; XXX Operationen werden rückgängig gemacht
;; ... aber Zugriffszeit auf [data*target] ist kürzer als
;; ... auf andere Elemente, da der Wert im Cache liegt
```

- Zugriffszeiten in `data` erlauben Rekonstruktion von `target`