Probeklausur

Rechnerorganisation

Universität:	Technische Universität Berlin
Kurs/Modul:	Rechnerorganisation
Bearbeitungszeit:	180 Minuten
Erstellungsdatum:	September 20, 2025

Bearbeitungszeit: 180 Minuten.

Aufgabe 1.

(a) Skizzieren Sie den Aufbau eines einfachen Von-Neumann-Rechners mit CPU, Speicher und I/O. Beschreiben Sie grob die Rolle der Steuereinheit, des Datenpfads und des Speichersystems.

(b) Nennen Sie drei gängige Adressierungsarten in einer typischen Assemblersprache und skizzieren Sie, wie sich diese in Maschinencode übersetzen würden.

(c) Erklären Sie grob, wie ein Programm, das in einer höheren Programmiersprache verfasst ist, in Maschinensprache überführt wird und welche Rolle der Assembler bzw. Compiler dabei spielt.

(d) Skizzieren Sie eine einfache Pipeline mit den Stufen Fetch, Decode und Execute. Erläutern Sie typische Pipeline-Hazards und nennen Sie eine einfache Gegenmaßnahme.

Aufgabe 2.

(a) Beschreiben Sie die Speicherhierarchie eines typischen Rechners mit Cache, Hauptspeicher und Sekundärem Speicher. Erläutern Sie, warum Caches zentral für die Leistungsfähigkeit sind.

(b) Definieren Sie die Begriffe *Hit-Rate* und *Miss-Rate* im Cache-Kontext und erläutern Sie deren Bedeutung für den Zugriff auf den Cache.

(c) Erklären Sie das Prinzip des virtuellen Speichers und welche Vorteile es gegenüber reinem physischen Speicher bietet.

(d) Zeichnen Sie eine typische Speicherhierarchie und erläutern Sie kurz, wie Adressübersetzung und Cache-Organisation zusammenwirken.

Aufgabe 3.

(a) Schreiben Sie in einer einfachen 8-Bit-Assembler-Sprache ein kurzes Programm, das zwei direkt gelieferte Konstanten addiert und das Ergebnis in eine Speicherstelle schreibt. Verwenden Sie eine sinnvolle Syntax mit Befehlen wie `LOAD`, `ADD`, `STORE` etc.

(b) Erläutern Sie kurz, wie bedingte Sprünge in Assembler realisiert werden und welche Rolle Vergleiche bzw. Flags dabei spielen.

(c) Nennen Sie drei typische Adressierungsarten in einer Assemblersprache und geben Sie je ein kurzes Beispiel in einer passenden Syntax an.

Aufgabe 4.

(a) Beschreiben Sie die Grundidee des Fließband- bzw. Pipeline-Ansatzes in der Rechnerarchitektur. Skizzieren Sie eine Pipeline mit vier Stufen und erläutern Sie, wie Daten durch die Stufen fließen.

(b) Nennen Sie drei Arten von Pipeline-Hazards und erläutern Sie, warum sie auftreten.

(c) Geben Sie eine kurze Übersicht typischer Gegenmaßnahmen gegen Hazards: Forwarding/Bypassing, Stalling, Branch Prediction usw. und kommentieren Sie deren Vor- und Nachteile.

(d) Diskutieren Sie, welche Auswirkungen Pipelining auf die Auslastung von Recheneinheiten, Taktfrequenz und Speicherzugriffe hat, sowie welche Herausforderungen sich beim Entwurf von Mehrzyklusimplementierungen ergeben.

Lösungen

Bearbeitungszeit: 180 Minuten.

Aufgabe 1.

(a) *Lösungsskizze und Beschreibung des Von-Neumann-Aufbaus.*

Der Von-Neumann-Rechner besteht aus drei Hauptkomponenten: einer Zentrale Verarbeitungseinheit (CPU), dem gemeinsamen Speicher und dem I/O-System. Die CPU umfasst Steuereinheit (Control Unit, CU) und Datenpfad (Register, ALU, ggf. spezielle Adressregister). Die CPU kommuniziert mit dem Speicher sowie dem I/O-System über gemeinsame Busse: Adressbus, Datenbus und Steuerbus. Die Steuereinheit koordiniert den Ablauf der Befehlsbearbeitung (Fetch, Decode, Execute) und steuert den Datenpfad, während der Datenpfad Operanden zwischen Registern, der ALU und dem Speicher verschiebt. Der Speicher dient als adressierbarer Speicherort für Programme und Daten; I/O ermöglicht den Austausch mit Peripherie. In der groben Ablaufbeschreibung geschieht typischerweise Folgendes:

- Fetch: Die CPU liest den nächsten Befehl aus dem Speicher an der Adresse des Program Counters (PC) über den Adressbus; der Befehl wird auf dem Datenbus an die CPU übertragen. - Decode: Die Steuereinheit interpretiert den Befehl, aktiviert die entsprechenden Signale und wählt operative Daten aus den Registern bzw. aus dem Speicher. - Execute: Die ALU oder andere Funktionsbausteine führen die Operation aus (Arithmetic/Logik, Adressbildung, Datentransfer); das Ergebnis wird in einem Zielregister oder im Speicher abgelegt. - Update des PC: Der PC wird auf die Adresse des nächsten Befehls inkrementiert oder durch Sprungadresse ersetzt. - Speicherzugriffe: Falls der Befehl einen Speicherzugriff erfordert, werden Daten über den Datenbus mit dem Speicher ausgetauscht.

Eine einfache, schematische Darstellung (textuell statt Grafik) der Interaktion:

$$\text{CPU} \leftrightarrow \text{Speicher}$$

mit Daten-, Adress- und Steuerbus-Verbindungen. Die I/O-Anbindung erfolgt über spezialisierte Anschlussflächen bzw. Busse, die Lese-/Schreibzugriffe an externe Peripherie ermöglichen.

(b) *Adressierungsarten und Übersetzung in Maschinencode.*

In einer typischen Assemblersprache treten mehrere Adressierungsarten auf. Drei gängige sind:

- Immediate-Adressierung (direkt gelieferter Operand): - Assembler-Instruktion: `ADD R0, #5` - Maschinencode (Pseudocode): Opcode für ADD, Zielregister R0, Operand 5 direkt im Befehl codiert. - Zweck: Konstante wird direkt als Operand genutzt, kein Zugriff auf Speicher nötig.

- Direkte Adressierung (Operand ist eine Speicheradresse): - Assembler-Instruktion: `LOAD R1, 0x40` - Maschinencode: Opcode ADD/LOAD, Zielregister R1, Adresse 0x40 im Befehl. - Zweck: Operand wird aus der angegebenen Speicheradresse gelesen.

- Indirekte Adressierung (Operand befindet sich an einer im Befehl angegebenen Adresse): - Assembler-Instruktion: `ADD R2, [R3]` - Maschinencode: Opcode für ADD, Zielregister R2, Indirekt-Adresskennzeichen (z. B. Adress- oder Registertupel, das auf eine Speicheradresse zeigt) codiert. - Zweck: Der eigentliche Operand wird an der zur Laufzeit ermittelten Adresse gefunden (Zugriff über einen Zeiger/Indexregister).

Beispielhafte Übersetzung (Pseudo-Maschinencode zur Veranschaulichung): - Immediate: - Assembly: `ADD R0, #7` - Maschinencode: `opcode(ADD) | R0 | imm=07` - Direct: - Assembly:

`LOAD R1, 0x20` - Maschinencode: `opcode(LOAD)` | `R1` | addr=0x20 - Indirect: - Assembly: `LOAD R2, [R3]` - Maschinencode: `opcode(LOAD)` | `R2` | indirect via R3

(c) *Übersetzung von höherwertigen Sprachen in Maschinensprache.*

- Ein Programm, das in einer höheren Sprache verfasst ist, wird zunächst durch den Compiler bzw. Interpreter in eine Zwischen- bzw. Assemblersprache übersetzt. Typischer Ablauf: - Frontend-Phasen: Lexikalische Analyse, Parsing, Semantikprüfung (Typprüfung, Gültigkeitsprüfungen). - Zwischencode/IR (Intermediate Representation): Plattformunabhängige Repräsentation der Logik, Optimierungen auf IR-Ebene. - Backend/Code-Generation: Übersetzung der IR in Assemblersprache bzw. direkt in Maschinencode, unter Berücksichtigung der Zielarchitektur (Instruktionssatz, Registerallokation). - Optimierung: Registernutzung, Befehlssortierung, Eliminierung unnötiger Berechnungen. - Assemblierung/Linking: Der Assembler wandelt Assemblersprache in Objektcode um, der Linker verbindet mehrere Objekte zu einem ausführbaren Programm; hierbei werden Symbole aufgelöst, Relokationen angepasst. - Rolle von Compiler und Assembler: - Compiler: Übersetzung der Hochsprache in eine Architektur-nahe Repräsentation (oft Assembly); Optimierung von Algorithmen, Datenstrukturen, Datentypen und Speicherzugriffen. - Assembler: Übersetzung der Assemblersprache in Maschinencode, Auflösung von Symbolen (Labels, Funktionsadressen) und Generierung von Objektdateien. - Schlüsselunterschiede: Compiler arbeitet üblicherweise an einer höheren Abstraktionsebene und führt umfangreiche Optimierungen aus, während der Assembler die konkrete Maschinencodierung und Adressauflösung übernimmt. Subsequentes Linking verknüpft verschiedene Objekte zu einem lauffähigen Programm.

(d) *Pipeline-Ansatz (Fetch-Decode-Execute) und Hazards.*

- Grundidee: Die Befehlsbearbeitung erfolgt parallel über mehrere Stufen, sodass verschiedene Befehle gleichzeitig unterschiedliche Phasen durchlaufen. Dadurch erhöht sich der Durchsatz, während die Taktfrequenz oft durch die kritische Pfadlänge der Stufen beeinflusst wird. - Drei-Stufen-Pipeline (Fetch – Decode – Execute): - Fetch (F): Nächsten Befehl aus dem Speicher laden. - Decode (D): Befehl semantisch entschlüsseln, Operanden auswählen bzw. laden. - Execute (E): ALU-Operation durchführen, Ergebnis speichern oder another Speicherzugriff vorbereiten. - Typische Pipeline-Hazards: - Struktur-Hazard: Mehrere Befehle benötigen gleichzeitig denselben Ressourcenteil (z. B. derselbe Speicherzugriff). - Daten-Hazard: Eine Instruktion hängt von einem Ergebnis der vorherigen Instruktion ab (Write-After-Read, Read-After-Write etc.). - Control-Hazard: Sprungbefehle führen zu Unsicherheit über die nächste Befehlsadresse; Branching beeinflusst den korrekten Befehl in der nächsten Stufe. - Einfache Gegenmaßnahme: - Forwarding/Bypassing: Ergebnisse werden direkt von einer Stufe zur nächsten weitergegeben, ohne auf das Schreiben in Register zu warten, reduziert Data-Hazards signifikant. - Stalling (Bubble-NOPs): Bei relevanten Hazard-Situationen werden Leerlaufzyklen eingefügt, um korrekte Daten zu gewährleisten. - Branch Prediction: Vorhersage der Sprungbedingungshandlung, um Control-Hazards zu minimieren; bei Fehlvorhersage kosten Mehrzyklen, aber moderne Prozessoren nutzen oft komplexe Prädiktoren. - Nutzung von Delay-Slots oder spekulativer Ausführung in bestimmten Architekturen. - Kurzbewertung: Forwarding erhöht die Auslastung der Recheneinheiten, kann aber zusätzliche Hardware erfordern; Stalling reduziert die Auslastung und erhöht das Energie-Nutzungsverhältnis; Branch Prediction verbessert den Durchsatz, benötigt jedoch komplexe Logik und kann zu Spekulationsfehlern führen.

Aufgabe 2.

(a) *Speicherhierarchie eines typischen Rechners.*

- Typische Ebenen der Hierarchie: Registersatz, L1-Cache (klein, sehr schnell, oft pro Kern), L2-Cache (größer, etwas langsamer), ggf. L3-Cache (gemeinsam mehrere Kerne), Hauptspeicher (RAM), Sekundärspeicher (Festplatte/SSD). - Prinzip der räumlichen und zeitlichen Lokalität: Programme arbeiten häufig mit benachbarten Speicheradressen (Temporal Locality) und verwenden aufeinanderfolgende Adressen (Spatial Locality). Caches nutzen dies aus, um Zugriffe zu beschleunigen. - Warum Caches zentral für Leistungsfähigkeit sind: Die Zugriffszeit auf den L2/L3-Hauptspeicher ist oft um Größenordnungen größer als CPU-Taktzyklen; Caches eliminieren repetitive Zugriffe auf langsame Speicherebenen, reduzieren Latenz und erhöhen Durchsatz wesentlich.

(b) *Hit-Rate und Miss-Rate im Cache-Kontext.*

- Hit-Rate: $H = \frac{\text{Anzahl der Cache-Hits}}{\text{Anzahl der Cache-Zugriffe}}$. - Miss-Rate: $M = 1 - H = \frac{\text{Anzahl der Cache-Misses}}{\text{Anzahl der Cache-Zugriffe}}$. - Bedeutung: Eine hohe Hit-Rate bedeutet, dass die meisten Speicherzugriffe direkt aus dem Cache bedient werden, während Misses häufig zu Zugriffszeiten auf die nächste Ebene (L2/L3 oder RAM) führen. Miss-Rate beeinflusst die effektive Speicherlatenz und damit die Gesamtsleistung eines Programms.

(c) *Prinzip des virtuellen Speichers und Vorteile gegenüber reinem physischem Speicher.*

- Virtueller Speicher (VS) verwendet Seiten und Seitentabellen, um einen Prozessahnen-Adressraum zu schaffen, der größer ist als der physisch verfügbare Speicher. - Vorteile: - Größerer Adressraum pro Prozess, Unabhängigkeit der Adressräume (Speicherschutz und Isolation). - Flexible Nutzung des RAM durch Paging (dem Muster der Seitenein-/auslagerung). - Ermöglicht Speicher-Überlappung, Shared-Memory zwischen Prozessen, einfache Abstraktion der physischen Speicherhierarchie. - Typische Mechanismen: Seitentabellen, Translation Lookaside Buffer (TLB) für schnelle Adressübersetzung; Demand Paging (Speicher wird nur bei Bedarf geladen); Seitenauslagerung (swap) für freien Speicher. - Nebenwirkungen: Seitentabellenzugriffe erhöhen den Speicheraufwand; TLB-Hits vs. -Misses beeinflussen die Leistung stark.

(d) *Typische Speicherhierarchie – Adressübersetzung und Cache-Organisation.*

- Typische Abbildung: CPU-Register → L1-Cache → L2-Cache → L3-Cache (falls vorhanden) → Hauptspeicher → Sekundärspeicher. - Adressübersetzung: Virtuelle Adressen werden durch den MMU in physische Adressen übersetzt; TLB beschleunigt diese Übersetzung, Caches arbeiten mit virtuellen bzw. physischen Adressen je nach Architektur (je nach inklusiv/exklusiv Cache-Policy). - Interaktion von Adressübersetzung und Cache: Beim Cache-Zugriff wird zuerst die adressierte Zeile gesucht; Übersetzungskette über den TLB/Seiten-Tabellen erfolgt, anschließend wird die passende Cache-Line geprüft. Einige Architekturen verwenden virtuelle Caches (TLB-basiert) oder physische Caches; Direct-M mapped oder set-associative Organisation beeinflusst die Wahrscheinlichkeit eines Misses. - Wichtige Konzepte: Cache-Topologien (Direct-Mapped, 2-, 4-way Set-Associative), inklusiv vs. exklusiv, Cache-Kohärenz in Mehrkernsystemen, Pre-Fetching-Strategien.

Aufgabe 3.

(a) *8-Bit-Assembler-Programm: zwei direkt gelieferte Konstanten addieren und das Ergebnis in eine Speicherstelle schreiben.*

Beispielprogramm (Syntax nach einer typischen einfachen 8-Bit-Assembler-Sprache):

```
; Zwei Konstanten addieren und Speicherstelle schreiben
LOAD  R0, #11      ; R0 = 11
LOAD  R1, #22      ; R1 = 22
ADD   R0, R1        ; R0 = R0 + R1 = 33
STORE R0, 0x10      ; Speicherstelle 0x10 := R0 (33)
```

Erklärung: - LOAD mit immediate-Operand lädt eine direkte Konstante in das Zielregister. - ADD führt eine zweistellige Addition der Operanden durch (hier R0 und R1) durch. - STORE schreibt das Ergebnis aus dem Zielregister in die angegebene Speicheradresse.

(b) *Bedingte Sprünge, Vergleiche und Flags.*

- C- oder Flag-basierte Sprünge ermöglichen bedingte Feldzugriffe basierend auf Vergleichen. Typische Vorgehensweise: - Vergleich (CMP) zweier Registerwerte setzt Statusflags (Zero Z, Negative N, Carry C, Overflow V) entsprechend dem Ergebnis. - Bedingte Sprünge (z. B. BEQ, BNE, BZ, BZ) prüfen diese Flags; z. B. BEQ springt, wenn Z-Flag gesetzt ist. - Typisches Beispiel:

```
CMP R0, R1        ; vergleiche R0 und R1, setzt Flags
BEQ label         ; springen, wenn Z (gleich) gesetzt
```

- Rolle der Flags: Ermöglichen numerische Vergleiche, Testen von Gleichheit, Größer/Kleiner, und konditionale Programmsteuerung.

(c) *Drei typische Adressierungsarten mit kurzen Beispielen.*

- Immediate (In-zwischenliegende Konstante): - Beispiel: `LOAD R0, 5` - Direct (direkte Speicheradresse): - Beispiel: `LOAD R0, 0x40` - Indirect (über Adresse, Zeiger): - Beispiel: `LOAD R0, [R3]` oder `LOAD R0, (R3)` - Hinweis: Die konkrete Notation hängt von der Architektur ab; die hier gezeigten Formen sind gängig in vielen 8-Bit-Architekturen.

Aufgabe 4.

(a) *Fließband-/Pipeline-Grundidee; vier Stufen.*

- Grundidee: Durch Aufteilen der Befehlsbearbeitung in mehrere Stufen können mehrere Befehle gleichzeitig in unterschiedlichen Phasen bearbeitet werden. - Vierstufige Pipeline (IF – ID – EX – WB): - Instruction Fetch (IF): Nächsten Befehl aus dem Speicher holen. - Instruction Decode (ID): Befehl entschlüsseln, Operanden auswählen, ggf. Adressbildung. - Execute (EX): Durchführung der arithmetischen/logischen Operation, Adressberechnungen, Vorbereitung von Speicherzugriffen. - Write Back (WB): Ergebnis in Register schreiben bzw. Speicher aktualisieren. - Datenfluss-Beispiel (stufenweise): - Zyklus 1: Instr A in IF - Zyklus 2: Instr A in ID, Instr B in IF - Zyklus 3: Instr A in EX, Instr B in ID, Instr C in IF - Zyklus 4: Instr A im WB, Instr B im EX, Instr C in ID, Instr D in IF - Vorteil: Erhöhter Durchsatz; Nachteil: erfordert Synchronisation, Hazard-Behandlung.

(b) *Drei Arten von Pipeline-Hazards und Ursachen.*

- Structural Hazard: Mehrere Befehle benötigen dieselbe Ressource (z. B. gleiche Speicherzugriffslogik oder ALU) gleichzeitig. - Data Hazard: Eine Anweisung hängt an den Wert einer vorhergehenden Anweisung (z. B. Lesen eines Registers, das noch geschrieben wird). - Control Hazard: Branch-Befehle führen zu Unsicherheit über die nächste Befehlsadresse und damit über das, was in der nächsten Pipeline-Stufe verarbeitet wird.

(c) *Gegenmaßnahmen gegen Hazards; Vor- und Nachteile.*

- Forwarding/Bypassing: Ergebnisse direkt von einer Stufe zur nächsten weiterleiten, um Data-Hazards zu umgehen. - Vorteil: Höherer Durchsatz; Nachteil: Komplexere Datenpfade und Logik. - Stalling (Bubble-Einfügungen): Verzögerungen in der Pipeline, wenn Hazard erkannt wird. - Vorteil: Einfach; Nachteil: Reduziert Durchsatz. - Branch Prediction: Vorhersage der Sprungbedingung; spekulatives Ausführen, Abbruch, falls Vorhersage falsch. - Vorteil: Reduziert Control-Hazards; Nachteil: Komplexität; potenzielle Kosten bei Fehlvorhersagen. - Delay-Slots/NOPs: Spezifische Architekturen nutzen Leerlaufzeiten nach Sprüngen; Nachteil: Code-Verluste. - Weitere Ansätze: Dynamische Scheduling, Out-of-Order-Execution, Cache-Hilfen, spekulative Ausführung.

(d) *Auswirkungen von Pipelining auf Auslastung, Taktfrequenz und Speicherzugriffe; Herausforderungen bei Mehrzyklus-Implementierungen.*

- Auswirkungen: - Erhöhter Durchsatz: Mehr Anweisungen pro Taktzyklus bei ausreichender Pipeline-Ausnutzung. - Höhere Anforderungen an den Takt: Die Länge der Pipeline-Stufen muss innerhalb eines einzelnen Taktzyklus abgearbeitet werden; dies beeinflusst die maximale sichere Taktrate. - Speicherzugriffe: Gefahr von Latenzdiskrepanzen; Speichersysteme müssen die Kehrseiten des Durchsatzzuwachses unterstützen (Cache-Hierarchie, Speicher-Bandbreite). - Herausforderungen bei Mehrzyklus-Implementierungen: - Koordination der Stufen über Zyklen hinweg, Synchronisation der Registerdateien. - Hazard-Erkennung und -Behandlung (Hazard-Detektionseinheit). - Konsistenz von Branch- und Sprungvorhersagen über mehrere Zyklen hinweg. - Kompakte Breitband- und Ressourcen-Nutzung bei heterogenen Stufen (z. B. Lade-/Schreibzugriffe, Speicherzugriffszeit). - Fazit: Pipelining verbessert

typischerweise den Durchsatz signifikant, erfordert aber umfangreiche Synchronisation, Hazard-Management und robuste Adress-/Datenpfad-Architekturen.