Probeklausur

Rechnerorganisation

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

Aufgabe 1.

(a) Beschreiben Sie die grundlegende Struktur eines Von-Neumann-Rechners und nennen Sie die vier zentralen Bestandteile. Erläutern Sie den Ablauf eines typischen Befehlszyklus (Fetch-Decode-Execute).

(b) In einer vereinfachten Assemblersprache seien Befehle wie folgt möglich: LOAD R, D; STORE R, D; ADD R1, R2; JUMP A. Beschreiben Sie die Bedeutung der drei gängigen Adressierungsarten (unmittelbare, direkte, indirekte Adressierung) und erläutern Sie, wie diese im Prozessorspektrum realisiert werden können.

(c) Erklären Sie grob, wie Programme, die in höheren Programmiersprachen verfasst sind, in Maschinensprache übersetzt werden. Welche Rolle spielen Compiler und Assembler im Übersetzungsprozess, und wie hängen sie mit der Ausführung auf der Maschineneinheit zusammen?

Aufgabe 2.

(a) Beschreiben Sie das Konzept des Pipelinings in einer typischen CPU. Welche fünf Stufen werden in einem klassischen Fünf-Stufen-Pipeline-Modell unterschieden (IF, ID, EX, MEM, WB), und welche Aufgaben erfüllen sie?

(b) Gegeben seien drei aufeinander folgende Befehle in einer hypothetischen CPU: - Befehl 1: LOAD R1, 10 - Befehl 2: ADD R1, R2 - Befehl 3: STORE 0x2000, R1 Analysieren Sie, ob Datennachbarn (data hazards) auftreten können. Bestimmen Sie die Art der möglichen Hazards und skizzieren Sie, wie Forwarding oder Stalling zur Vermeidung verwendet werden könnten.

(c) Skizzieren Sie grob eine zeitliche Abfolge der drei Befehle durch eine Pipeline, wenn Forwarding vorhanden ist. Kommentieren Sie, an welchen Stellen Hazards auftreten könnten und welche Maßnahmen helfen würden, diese zu mildern.

Aufgabe 3.

(a) Beschreiben Sie die Speicherhierarchie eines typischen Rechners mit Cache-Ebenen. Erklären Sie die Rolle von L1-, L2-, ggf. L3-Caches, sowie den Unterschied zwischen Cache-Hit und Cache-Miss. Diskutieren Sie grob, wie Speicherzugriffe in der Praxis ablaufen.

(b) Erklären Sie den Unterschied zwischen Write-Through- und Write-Back-Strategien beim Cache-Schreiben. Welche Vor- und Nachteile ergeben sich jeweils für Konsistenz, Geschwindigkeit und Komplexität?

(c) Welche Probleme können in Mehrkernsystemen bei Cache-Kohärenz auftreten? Nennen Sie grob typische Kohärenzprotokolle oder -konzepte, ohne ins Detail zu gehen.

Aufgabe 4.

(a) Beschreiben Sie die grundlegende Organisation von Ein-/Ausgabe in Rechnern im Kontext einer adressierbaren Speicherarchitektur. Diskutieren Sie einfache I/O-Strategien, Interrupt-basiertes I/O und die Rolle von Adressierung bei Peripheriekommunikation.

(b) Erklären Sie das Konzept des Direct Memory Access (DMA). Welche Rolle übernimmt der DMA-Controller, und welche typischen Signale sind an einem DMA-Transfer beteiligt? Skizzieren Sie den Ablauf eines DMA-Transfers aus Peripherie in den Speicher.

(c) Diskutieren Sie grundsätzliche Synchronisationsmechanismen zwischen der CPU und Peripherie bei I/O, z. B. Polling, Interrupts und DMA-Kooperation. Welche Vor- und Nachteile ergeben sich in Bezug auf Reaktionszeit, Effizienz und Komplexität?

Lösungen

Lösung zu Aufgabe 1.

(a) Die grundlegende Struktur eines Von-Neumann-Rechners besteht aus vier zentralen Bestandteilen: (i) der Zentraleinheit (CPU) mit Rechenwerk (ALU) und Steuerwerk, (ii) dem Speicher, (iii) der Ein-/Ausgabe (I/O) sowie (iv) dem gemeinsamen Bus- bzw. Speicher-System (Speicherhierarchie inkl. Adress- und Databus). Der Ablauf eines typischen Befehlszyklus – Fetch-Decode-Execute – läuft wie folgt ab: Der Befehl wird aus dem Speicher in die Befehlsregister geladen (Fetch), anschließend wird der geladene Befehl decodiert, d. h. seine Opcode- und Operanden-Anforderungen werden bestimmt (Decode), und schließlich wird der Befehl ausgeführt, wobei die Operation an der ALU vorgenommen, ggf. Speicherzugriffe ausgeführt oder Sprünge vorgenommen werden (Execute). Während dieses Zyklus wird der Programmzähler (PC) aktualisiert, um den nächsten Befehl zu adressieren.

(b) Adressierungsarten in der vereinfachten Assemblersprache: LOAD R, D; STORE R, D; ADD R1, R2; JUMP A. - Unmittelbare (Immediate) Adressierung: Die Operandenwerte sind direkt im Befehl kodiert (z. B. LOAD R, 5). Der Operand liegt als Konstante vor und wird typischerweise in einem Adressierungsfeld des Befehls aufgenommen. Umsetzung: Der unmittelbare Operand wird während Decode/Execute verwendet, kein weiterer Zugriff auf den Speicher ist nötig. - Direkte (Direct) Adressierung: Die Adressierung eines Operanden verweist direkt auf eine Speicheradresse, z. B. LOAD R, 0x1234 bedeutet: lade den Wert aus Speicheradresse 0x1234 in Register R. Umsetzung: MAR wird mit der im Befehl angegebenen Adresse beladen, MDR erhält den Dateninhalt aus dem Speicher. - Indirekte (Indirect) Adressierung: Die im Befehl angegebene Adresse verweist auf eine Speicherstelle, die wiederum die effektive Zieladresse enthält. Umsetzung: Zunächst wird die im Befehl kodierte Adresse gelesen, dann wird an dieser Adresse der richtige Zielwert bzw. -adresse gelesen, der anschließend verwendet wird. Diese Adressierungsart ermöglicht Zeiger- oder Referenzstrukturen.

Die Umsetzung im Prozessorspektrum erfolgt typischerweise durch Adressierungsbits im Opcode bzw. durch Adressierungsmodi in der ALU-/Register-Datei. Unabhängig vom Modus benötigt man in allen Fällen einen Adressbus (zur Adressübermittlung) sowie einen Datenbus (für den Datentransfer). Forwarding/Bypassing-Pfade innerhalb der CPU können dazu dienen, Operanden direkt aus nachfolgenden Pipeline-Stufen zu beziehen, um Verzögerungen zu vermeiden.

(c) Übersetzung von Programmen aus höheren Programmiersprachen in Maschinensprache erfolgt in mehreren Schichten. Höhere Sprachen (z. B. C/C++, Java) werden in Zwischensprachen (Executables, Bytecode) bzw. Assemblersprache übersetzt und dann in Maschinencode umgesetzt. Die Rolle von Compiler und Assembler im Übersetzungsprozess ist wie folgt: Der Compiler analysiert semantische Strukturen, optimiert den Code, generiert Zuweisungen, Kontrollflussstrukturen, Speicherzugriffe und ruft die entsprechenden Maschinensprachen-Operationen auf. Der Assembler wandelt anschließend die assemblersprachlichen Befehle in Maschinencodesetzen (Opcode, Operandenadressen) um. Die Ausführung erfolgt letztlich auf der Maschineneinheit, d. h. der CPU, dem Speicher und der Ein-/Ausgabe, wobei das erzeugte Maschinensymbolbild direkt vom Prozessor interpretiert und ausgeführt wird. Optimierungen im Compiler (z. B. Registerallokation, Pipelining, Levelling von Lastspitzen) beeinflussen damit direkt die Effizienz der Ausführung auf der Hardware.

Lösung zu Aufgabe 2.

(a) Das Konzept des Pipelinings in einer typischen CPU teilt die Befehlsverarbeitung in mehrere aufeinanderfolgende Stufen, um die Durchsatzleistung zu erhöhen. In einem klassischen Fünf-Stufen-Pipeline-Modell werden folgende Stufen unterschieden und erfüllen diese Aufgaben: - IF (Instruction Fetch): Der nächste Befehl wird aus dem Speicher in das Instruction-Register geholt. - ID (Instruction Decode / Decode): Der Befehl wird decodiert, die Operanden identifiziert und/Register-Dateien gelesen. - EX (Execute): Die ALU führt die Rechenoperationen durch oder führt Adressberechnungen/Operandensynthese durch. - MEM (Memory Access): Falls erforderlich, erfolgt der Zugriff auf den Speicher (Laden/ Speichern von Daten). - WB (Write Back): Das Ergebnis wird in das Zielregister geschrieben bzw. das Ziel der Speicheroperation abgeschlossen.

(b) Gegeben seien drei aufeinander folgende Befehle: - Befehl 1: LOAD R1, 10 - Befehl 2: ADD R1, R2 - Befehl 3: STORE 0x2000, R1

Es können Data Hazards auftreten: - RAW-Hazard zwischen B1 und B2: B2 benötigt den Wert von R1, der aus B1 stammt (Wert 10). Da B1 ein Load ist, liegt der gelieferte Wert erst am Ende der MEM-Phase vor. Ohne Maßnahmen würde B2 frühzeitig mit dem alten Wert arbeiten. - RAW-Hazard zwischen B2 und B3: B3 speichert den Wert von R1, der durch B2 modifiziert wird. Ohne Verzögerung könnte der alte R1-Wert in das Store-Operandenfeld gelangen.

Hazards-Typen und Maßnahmen: - Load-Use-Hazard (zwischen B1 und B2): Typischerweise erfordert dies einen Kurzstopp (einzelner Bubble) bzw. ein Delay, da der Wert erst nach dem MEM-Zyklus verfügbar wird. Forwarding kann hier in der Regel den unmittelbaren Wert aus MEM/WB nicht liefern, da er erst am Ende MEM verfügbar wird. - RAW-Hazard zwischen B2 und B3: Falls der Wert von R1 durch B2 generiert wird, kann Forwarding vom EX/MEM oder MEM/WB-Pfad genutzt werden, um den Store in B3 mit dem korrekten Wert zu versorgen. Falls Forwarding nicht ausreicht, kann ein leerer Takt (Stall) nötig sein.

Zusammengefasst: Es existieren RAW-Hazards. Forwarding reduziert Hazards, reicht bei Load-Uses jedoch meist nicht aus und erfordert einen kurzen Stopp. Der Hazard zwischen B2 und B3 lässt sich durch Forwarding abfedern, kann aber abhängig von der genauen Pipeline-Implementierung zusätzliche Stalls benötigen.

(c) Zeitliche Abfolge der drei Befehle mit Forwarding (grobe Skizze; idealisierte Pipeline):

- Cycle 1: B1 IF - Cycle 2: B1 ID, B2 IF - Cycle 3: B1 EX, B2 ID, B3 IF - Cycle 4: B1 MEM, B2 ICE (Forwarding aus MEM zu EX ist hier noch nicht verfügbar, daher ggf. Stall); B3 ID - Cycle 5: B1 WB, B2 EX (mit dem aus B1 gelieferte Wert), B3 EX - Cycle 6: B2 MEM, B3 MEM (mit möglichen Forwarding für R1) - Cycle 7: B2 WB, B3 WB

Hinweise: - Der primäre Hazard entsteht durch den Load von B1, der die Zuweisung von 10 zu R1 erst am Ende der MEM-Phase liefert. Daher ist ein Stopp von B2 vor EX sinnvoll (Load-Use-Hazard). - Forwarding-Ketten (z. B. von MEM/WB zu EX) können die Zeitfenster so verkürzen, dass B3 den korrekten R1-Wert erhält, sobald er benötigt wird. - Ohne Forwarding oder Stalls würden Hazards auftreten; mit Forwarding werden die Risiken reduziert, aber ein kurzer Stopp bleibt bei der Load-Use-Situation meist dennoch sinnvoll.

Lösung zu Aufgabe 3.

(a) Die Speicherhierarchie eines typischen Rechners besteht aus mehreren Ebenen, deren Zweck es ist, die mittleren Zugriffszeiten zu reduzieren. Typische Ebenen sind: - L1-Cache (Instruction- und Data-Cache): Sehr schnelle, aber kleine Cache-Ebene nahe der CPU. - L2-Cache: Größer als L1, etwas langsamer, dient als Zwischenspeicher zwischen L1 und dem Hauptspeicher. - Ggf. L3-Cache: Größer als L2, oft gemeinsam über mehrere Kerne hinweg genutzt. - Hauptspeicher (RAM): Größere, langsamere Speicherschicht. - Sekundärer Speicher (Festplatte, SSD) als weitere Ebene außerhalb der Cache-Hierarchie.

Cache-Hits bedeuten, dass der benötigte Speicherinhalt in der jeweiligen Cache-Ebene vorhanden ist und direkt von dort geliefert wird. Cache-Misses bedeuten, dass der Inhalt nicht im Cache vorhanden ist und vom nächsten Level nachgeladen werden muss. Praktisch läuft ein Speicherzugriff oft so ab: Zuerst wird im L1-Cache geprüft (Hit/Miss). Bei einem Miss wird der Inhalt aus dem nächsthöheren Level geholt (L2, ggf. L3 oder RAM) und in den Cache zurückgeschrieben (Cache-Speicherhierarchie). Die Koordination und das Timing der Cache-Operationen bestimmen maßgeblich die Leistungsfähigkeit des Systems.

(b) Write-Through- und Write-Back-Strategien unterscheiden sich im Umgang mit Schreibzugriffen: - Write-Through: Jede Schreiboperation wird sowohl im Cache als auch im darunterliegenden Speicher (Main Memory) ausgeführt. Vorteil: einfache Konsistenz, da Speicher immer up-to-date ist. Nachteil: höhere Speicherbandbreitennutzung, potenziell geringere Schreibgeschwindigkeit, häufiger Speicherzugriff. - Write-Back: Schreiboperationen betreffen zunächst nur den Cache (Write-Back). Der unterliegende Speicher wird erst dann aktualisiert, wenn die betroffene Cache-Zeile aus dem Cache verdrängt wird (oder auf andere Weise invalidiert wird). Vorteil: geringere Speicherzugriffe, schnellere Schreiboperationen. Nachteil: komplexere Konsistenzlogik, potenzielle Inkonsistenzen zwischen Cache und Speicher, Notwendigkeit von Dirty-Bits bzw. Koordinationsmechanismen.

Vor- und Nachteile zusammengefasst: - Konsistenz: Write-Through ist in der Regel einfacher konsistent; Write-Back erfordert Kooperationsmechanismen, um Konsistenz zwischen Caches und Speicher sicherzustellen (z. B. Durchsetzung von Protokollen bei Cache-Überläufen). - Geschwindigkeit: Write-Back bietet meist höhere Schreibleistung, da Schreibzugriffe auf den Hauptspeicher reduziert werden. - Komplexität: Write-Through ist einfacher zu implementieren; Write-Back benötigt komplexe Protokolle (Dirty-Flags, Koordination bei Cache-Kohärenz).

(c) In Mehrkernsystemen können folgende Probleme bei Cache-Kohärenz auftreten: Wenn mehrere Kerne Caches besitzen, deren Kopien gemeinsamer Speicheradressen halten, müssen Schreiboperationen konsistent koordiniert werden. Typische Kohärenzprobleme sind falsches Teilen (false sharing) und Inkonsistenzen durch Cache-Hierarchien. Typische Kohärenzprotokolle bzw. Konzepte (ohne ins Detail zu gehen): - Snooping-Protokolle (z. B. MESI-Familie) in geteilten Cache-Architekturen, wo Caches über den Bus beobachten, welche Werte in anderen Caches geändert wurden, und entsprechend invalidieren bzw. invalidate- bzw. update-Signale senden. - Directory-basierte Kohärenz-Ansätze, bei denen ein Verzeichnis den Zustand jeder Speicheradresse über alle Caches verfolgt und die Coherence-Updates gezielt an betroffene Caches sendet.

Lösung zu Aufgabe 4.

(a) Die grundlegende Organisation von Ein-/Ausgabe (I/O) in Rechnern im Kontext einer adressierbaren Speicherarchitektur lässt sich wie folgt skizzieren: I/O-Geräte sind in Adressräume eingebunden (Memory-Mapped I/O) oder besitzen separate Adressräume (Port-MMapped I/O). Die Peripherie kommuniziert über spezielle Schnittstellen, Interruptsignale oder Direct Memory Access (DMA). Es bestehen einfache I/O-Strategien wie Polling (aktives Abfragen eines Geräts), Interrupt-basiertes I/O (Gerät signalisiert Abschluss durch Interrupt an die CPU) sowie DMA-Kooperation (Peripherie greift direkt auf den Speicher zu, während die CPU entlastet wird). Die Adressierung spielt eine zentrale Rolle, da sie bestimmt, wie Peripherie register- oder steuerungsbasiert adressierbar ist und wie Speicherzugriffe bzw. Befehle an Geräte gerichtet werden.

(b) Direct Memory Access (DMA) ermöglicht Peripherie-Komponenten den direkten Zugriff auf den Hauptspeicher, ohne dass die CPU aktiviert bleibt. Der DMA-Controller übernimmt dabei die Koordination des Transfers und meldet der CPU nach Abschluss oder in Interrupts. Typische Signale und Schritte eines DMA-Transfers: - AD/ADIO-Signalpfade: Adress-/Datenpfade zwischen Peripherie, DMA-Controller und Speicher. - DMA-Anforderung (DMARQ) durch das Peripheriegerät: Signalisiert Bedarf an Speicherzugriff. - DMA-Acknowledge (DACK) oder Bus-Zugrifffreigabe: CPU erlaubt dem DMA-Controller den Zugriff auf den Speicherbus. - Adress- und Datenübertragung: Der DMA-Controller legt die Zieladresse fest und überträgt Daten direkt zwischen Peripherie und Speicher. - Terminal Count (TC) bzw. Completion-Indikator: Abschluss des Transfers; optional Interrupt an die CPU, um den Abschluss zu signalisieren.

Ablauf eines DMA-Transfers grob skizziert: 1) Peripherie fordert Zugriff an (DMARQ). 2) DMA-Controller erhält Freigabe (DACK) und setzt Adress- und Datenpfade effizient. 3) Daten werden direkt in/zur Speicheradresse transferiert. 4) Transferabschluss wird dem CPU-internen Steuerwerk gemeldet (TC) oder durch Interrupt signalisiert. 5) Peripherie setzt den Betrieb fort oder wird freigegeben.

(c) Grundsätzliche Synchronisationsmechanismen zwischen CPU und Peripherie bei I/O: - Polling: CPU fragt kontinuierlich den Status eines Gerätes ab. Vorteile: einfache Implementierung, deterministische Abläufe. Nachteile: ineffiziente Ressourcennutzung, geringe Reaktionsgeschwindigkeit bei anderen Aufgaben. - Interrupts: Peripherie signalisieren einem Interrupt-Controller bzw. der CPU, wenn ein Ereignis eingetreten ist. Vorteile: CPU kann andere Aufgaben erledigen, Reaktionszeit ist oft gering. Nachteile: Komplexere Programmierlogik, Interrupt-Latency, Interrupt-Storms möglich. - DMA-Kooperation: DMA überträgt Daten selbstständig zwischen Peripherie und Speicher, wobei die CPU entlastet wird. Vorteile: hohe Effizienz bei großen Datenmengen, geringer CPU-Beteiligung. Nachteile: Komplexere Koordination, mögliche Cache-Kohärenzprobleme und Synchronisationsbedarf.

Zusammenfassend ergeben sich je nach Applikation unterschiedliche Optimierungen: Polling ist einfach, Interrupts bieten gute Reaktionszeiten, DMA bietet hohen Durchsatz bei seriellen Transfers. In modernen Systemen kommt oft eine Kombination aus Interrupts und DMA zum Einsatz, um Reaktionszeiten zu wahren und zugleich Effizienz zu sichern.