Zusammenfassung Rechnerstrukturen 1

Anmerkungen

Bei Fehlern, Anmerkungen, Fragen oder Kritik bitte ich um Mail unter kontakt@carsten-buschmann.de. Weiterhin bin ich unter www.carsten-buschmann.de erreichbar.

Zur Notation: Indizes werden häufig durch Groß-/Kleinschreibung verdeutlicht, d.h. Emin = E_{min}. Die letzte Spalte Stellt die (ungefähre, weil jedes Jahr etwas andere) Seitenzahl der Vorlesungsunterlagen dar (würde man diese durchnummerieren).

1. Grundlagen des Rechnerentwurfes

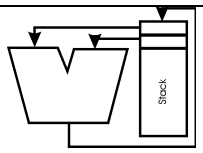
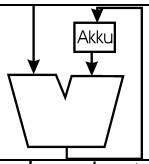
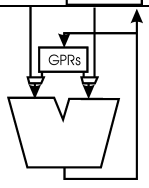
1.1	Def. Rechnerarchitektur	- die Disziplin, die sich mit der Entwicklung individueller Rechner aus einer Menge von allg. Bauteilen befaßt - **früher**: überwiegend technologische Probleme (z.B. Zuverlässigkeit, Kapazität, ...) - **heute**: ökonomische Probleme überwiegen: Kostenverfall macht Entwicklungskosten wichtiger, Austauschbarkeit durch Standardisierung → **heute überwiegend quantitativer** Entwurf von Digitalsystemen	2
	Gewinn durch Verkleinerung	um den Faktor 2: halbe Breite, halbe Höhe, doppelte Schaltgeschwindigkeit → Achtfache Rechenleistung	5
	Gründe für Entwicklung der Rechenleistung	- **1970 – 1980**: Technologiefortschritt und Innovationen in der Computerarchitektur → Steigerung um 25 – 30% pro Jahr - **ab Mitte 80er**: Hochsprachen statt Assembler, Standardisierte Betriebssysteme, breiteres Anwendungsspektrum durch mehr Leistung und Dominanz von Mikroprozessoren begünstigen Entwicklung neuer Architekturen → RISC → 40-50% Leistungssteigerung/Jahr	7
	Aufgaben des Rechnerarchitekten	- Festlegen der Anforderungen an den zu entwerfenden Rechner - Einhalten dieser unter Einhaltung des Kostenrahmens und Maximierung der Performance	9
	Funktionale Anforderungen	Preisvorgaben, Performancevorgaben, Software, Anwendungsspektrum, Abwärtskompatibilität, Special Purpose Requirements (wie MMX), Betriebssystemerfordernisse, Standards	10
	Aufgaben des Rechnerentwurfes	Entwurf des Befehlssatzes, funktionale Organisation (Speichersystem, Busstruktur, CPU), logischer Entwurf, Implementierung	10
	Optimierung	bei erfüllten funktionalen Vorgaben: Optimierung nach **Maßen** wie Performance, Kosten, Entwicklungszeit, Fehlertoleranz, Zuverlässigkeit	11
1.2	Trends in der Technologie	- **Integrierte Schaltungen**: Transistordichte: +50%/Jahr, Chipgröße: +10%/Jahr → 4x mehr Transistoren alle 3 Jahre - **DRAM**: Transistordichte: +60%/Jahr, Zykluszeit: -2%/Jahr → 4x mehr Seicherleistung alle 3 Jahre - **Platten**: Vervierfachung alle 3 Jahre	10
1.3	Kosten	- **Lernkurve** läßt Kosten sinken, bessere **Ausbeute** (Anteil der Produkte, die Test bestehen) - doppelte **Auflage** senkt Kosten um ca. 10% - **Wettbewerb** bringt Kostendruck	13
	Formeln Kosten	$IC-Kosten = \frac{Die-Kosten + Testkosten + Packagekosten}{Testausbeute}$ $Diekosten = \frac{Waferkosten}{Dies/Wafer * Dieausbeute}$ $Dies/Wafer = \frac{\pi * Waferradius^2}{Diefläche} - \frac{2\pi Waferradius}{\underbrace{\sqrt{2 * Diefläche}}_{Diediagonale}}$ $Dieausbeute = Waferausbeute * \left(1 + \frac{Defekte/Fläche * Diefläche}{\alpha}\right)^{\alpha}$ $\alpha \approx 3$ → insgesamt geht die Diefläche in 4ter Potenz in die Diekosten ein	14
	Komponenten-Kosten	machen ca. ein Drittel der Gesamtkosten aus	18
1.4	Performance	Rechner X ist n mal schneller als Rechner Y, wenn gilt $n = \frac{Executiontime_Y}{Executiontime_X} = \frac{\frac{1}{Performance_Y}}{\frac{1}{Performance_X}} = \frac{Performance_X}{Performance_Y}$ wichtig: Reproduzierbarkeit durch Angabe technischer Daten wie Compilerversion usw.	19
	Messung der Performance	- **CPU-Zeit**: Anteil der Zeit, den die CPU wirklich an dem Job arbeitet, unterteilt sich in User-CPU-Zeit und System-CPU-Zeit - **Programme zur Messung**: Reale Programme, Kernels (Schlüsselstücke realer Programme), Toybenchmarks (Programme, deren Ergebnis der Nutzer bereits kennt wie Quicksort), Synthetic Benchmarks (typische Verteilung der Befehle wie in realen Programmen), Benchmarksätze (Sammlung von Programmen zur Performancemessung)	20
	Benchmarksätze	gutes Werkzeug, aber Finden der Gesamtbewertung schwierig. Möglichkeiten: Summe, arithmetisches Mittel (mit/ohne Gewichtung), geometrisches Mittel	22
	arithmetisches Mittel	$\frac{1}{n}\sum_{i=1}^{n} Ausführungszeit_i$ Nachteil: Programme gehen unabhängig von der Häufigkeit der Ausführung ein	23
	Gewichtung	$\sum_{i=1}^{n} Gewicht_i * Ausführungszeit_i$ Vorteil: Ausführungshäufigkeit kann berücksichtigt werden	24

	geometrisches Mittel	mit normalisierten Ausführungszeiten relativ zu einer Referenzmaschine $\sqrt[n]{\prod_{i=1}^{n}\frac{Ausführungszeit_X}{Ausführungszeit_R}}$ Nachteile: setzt nicht die tatsächlichen Ausführungszeiten zueinander ins Verhältnis, alle Programme gehen gleich gewichtet ein	25

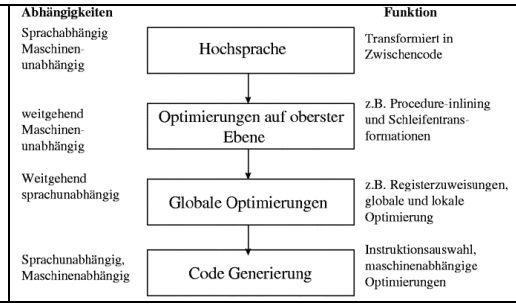
2. Quantitative Entwurfsprinzipien

2.1	Prinzip	**Make the common case fast**	27
	Speedup	$Speedup = \frac{Ausführungszeit_{unbeschleunigt}}{Ausführungszeit_{beschleunigt}}$	27
	Amdahls Law	$Ausführungszeit_{beschleunigt} = Ausführungszeit_{unbeschleunigt} * \left[1 - Anteil_{beschleunigt} + \frac{Anteil_{beschleunigt}}{Speedup_{beschleunigt}}\right]$ $Speedup = \frac{Ausführungszeit_{unbeschleunigt}}{Ausführungszeit_{beschleunigt}} = \frac{1}{\left[1 - Anteil_{beschleunigt} + \frac{Anteil_{beschleunigt}}{Speedup_{beschleunigt}}\right]}$	28
	max. Beschleunigung	Sei k der nicht beschleunigbare Anteil → Speedup<1/k (nach Amdahl mit Speedup = ∞)	29
2.2	Definitionen	Taktzykluszeit: reziproker Wert der Taktfrequenz, abhängig von Hardwaretechnologie und Organisation IC: Instruction count, abh. von Befehlssatzarchitektur und Compiler CPI: clocks per instruction, abh. von Organisation und Befehlssatzarchitektur	30
	CPU-Performance-Gleichung	CPU-Zeit = Anz. der Taktzyklen des Programmes * Taktzykluszeit CPU-Zeit = Anz. der Taktzyklen des Programmes / Taktfrequenz CPI = Anz. der Taktzyklen des Programmes / IC **CPU-Zeit = IC * CPI * Taktzykluszeit**	30
	Instruktionsklassen	$Anzahl\ Taktzyklen = \sum_{i=1}^{n} CPI_i * IC_i \rightarrow CPU - Zeit = \sum_{i=1}^{n} CPI_i * IC_i * Zykluszeit \rightarrow CPI = \frac{\overbrace{\sum_{i=1}^{n} CPI_i * IC_i}^{Anzahl\ Taktzyklen}}{IC}$	31
2.3	Lokalität	Daten/Befehle, die bereits benutzt wurden, werden mit großer Wahrscheinlichkeit wieder **benutzt (zeitliche Lokalität**), Daten, die nahe bei diesen liegen, werden ebenfalls wahrscheinlich benutzt (**räumliche Lokalität**) → solche Daten sollten schnell zugreifbar sein	34
	90-10-Regel	10% der Befehle eines Programmes sind für 90% der Ausführungszeit verantwortlich → solche Befehle sollten schnell zugreifbar sein	34
2.4	Prinzip	**Smaller ist faster**. Bsp: Speicher: - Signallaufzeiten, - kleinere Gatter, da kleinere Fanouts - weniger Leitungen	35
2.5	MIPS	Millionen Instruktionen pro Sekunde= IC/(CPU-Zeit * 10^6)	40
	MFlops	Millions of floating Point operations per second	41

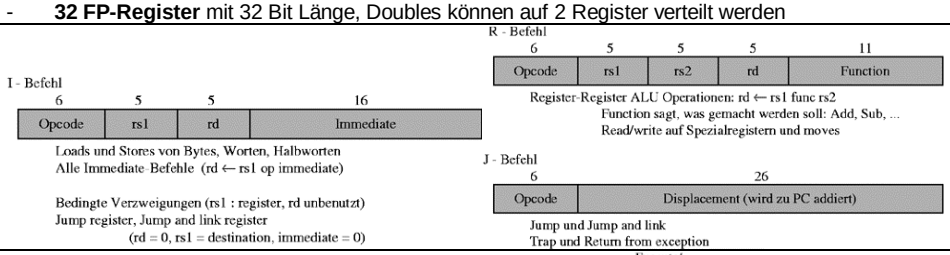
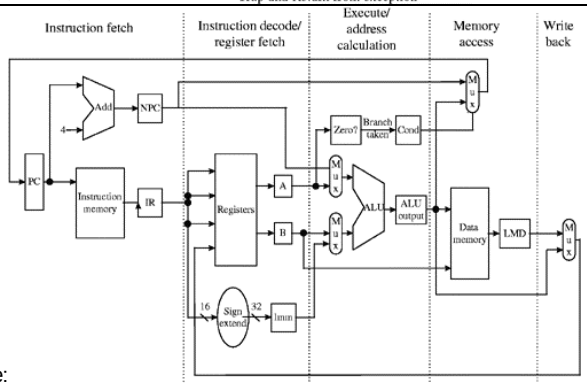
3. Befehlssatzarchitektur

3.1	Stack-Architektur	- CPU hat LIFO-Speicher, auf den man mit PUSH und POP zugreifen kann - es kann nur auf oberst Speicherstellen zugegriffen werden, alle Operationen beziehen sich auf die beider obersten Elemente → extrem einfach, aber keine effiziente Codeumsetzung möglich → heute praktisch bedeutungslos - **PUSH A; PUSH B; ADD; POP B** - **implizite Adressierung** von A und B bei ADD		43
	Akkumulator Architektur	- ausgezeichnetes Register: **Akku** - LOAD und STORE wirken nur auf Akku, er ist als expliziter Operand an jeder Operation beteiligt, jede Operation braucht nur eine Adresse → kompaktes Befehlsformat - **LOAD A; ADD B; STORE C**;		44
	GPR Architektur	- **General Purpose Registers** sind innerhalb eines Taktzykluses zugreifbare Speicher in der CPU, als Quelle und Ziel verwendbar, schneller als Speicher - einfach und effektiv für Compiler nutzbar, kürzere Adressierung als Speicher - können auch „Variablen" enthalten → verringert Speicherverkehr und beschleunigt Programm - Direkter Speicherzugriff weiterhin möglich - Klassifikation: **Typ (X, Y)** mit X: Anzahl Speicheradressen, Y: max. Anz. Operanden		44
	Register-Register-Maschinen	(**auch Load-Store-Architektur**) Typ (0, 3) - alle Operationen greifen auf Register zu, nur LOAD und STORE greifen auf Speicher zu - 32 – 512 GPRs - Vorteile: einfaches Befehlsformat fester Länge, einfaches Modell zur Codegenerierung, alle Instruktionen brauchen in etwa gleich lange - **Nachteil**: hoher IC - **LOAD R1, A; LAOD R2, B; ADD R3, R1, R2; STORE C, R3**;		46
	Register-Speicher-Maschinen	Type (1, 2) - Befehle können Operanden aus dem Speicher oder aus Registern holen		46

		- **Vorteile**: einfaches Befehlsformat fester Länge, Daten können ohne Laden zugegriffen werden, geringerer IC - **Nachteile:** variierender und höherer CPI, arithmetische Operationen zerstören einen Operanden - **LOAD R1, A; ADD R1, B; STORE C, R1**	
	Speicher-Speicher-Maschinen	Typ (3,3) - alle Operanden können aus dem Speicher oder aus Registern sein → größte Allgemeinheit - **Vorteile:** sehr kompakt, braucht keine Register für Zwischenergebnisse - **Nachteile:** Hoher CPI, viele Speicherzugriffe (Flaschenhals) - **ADD A, B, C**	46
	Zusammenfassung	wir entscheiden uns für eine **GPR-Architektur** als **Register-Register-Maschine**, da der IC weniger stark steigt, als der CPI sinkt	47
3.2	Operanden	können **Bytes** (8 Bit), **Half Words** (16 Bit), **Words** (32 Bit) oder **Double Words** (64 Bit) sein	48
	Big/Little Endian	Big Endian: MSB steht an der niedrigsten Speicheradresse Little Endian: LSB steht an der niedrigsten Speicheradresse	48
	Speicher-Ausrichtung	Speicher sind auf Ausrichtung auf Wortgrenzen hin optimiert, sonst müssen zwei Reihen im Speicher ausgelesen werden → nicht ausgerichteter Zugriff schneller, von modernen Architekturen wird Ausrichtung meist erzwungen	49
	effektive Adresse	Speicheradresse, die durch die jeweilige Adressierungsart angesprochen wird	49
	Adressierungsarten	sortiert nach Wichtigkeit - **Displacement** oder Segment-Offset: ADD R4, 100(R2) - **Register:** ADD R4, R3 - **Immediate:** ADD R4, #3 - Register deferred oder **indirekt**: ADD, R4, (R1) - **Memory indirect**: ADD R1, @(R3) entspricht Mem(Mem(R3)) - **Direct** oder Absolut: ADD R1, (1001)	50
	Displacement	- häufig, da für Variablen - obwohl lange Displacements allgemeiner wären, ist Begrenzung der Länge sinnvoll, da sonst Befehlsformat zu lang wird - 16 Bit decken 99% der Fälle ab	51
	Immediate	- für arithmetische Operationen, Vergleiche, Laden von Konstanten - 8 Bit decken 50%, 16 Bit mehr als 80% der Fälle ab	53
	Zusammenfassung	**Adressierungsarten:** Displacement, Immediate, Register, Direct, Indirect **Länge Displacement**: 12 – 16 Bit decken 99% ab **Länge Immediate:** 8 – 16 Bit decken 50 – 80% ab	54
3.3	Operationen	- jedes System muß **Datentransferbefehle** (LOAD, STORE, MOVE, ...), **arithmetische Befehle** (ADD, OR, ...) und **Kontrollbefehle** (JUMP, BRANCE) haben - die Meisten Systeme haben auch **FP-Operationen** und **Systemoperationen** - einige Systeme haben zusätzlich **Dezimaloperationen** und **Grafikoperationen** (wie MMX), häufig als Kombination von Befehlen andere Kategorien	55
	Faustregel	Die am häufigsten genutzten Befehle sind die einfachsten Befehle des Befehlssatzes: load, store, add, sub, move register register, and, shift, compare (not) equal, branch, jump, call, return	55
	Sprungbefehle	- man unterscheidet (nach Häufigkeit) **Conditional Branches**, **Procedure Calls** und **Returns** sowie **Jumps** - meist wird ein **Displacement** zum PC+4 addiert (PC-relative Sprünge) → Sprungziele bleiben korrekt, egal, wohin das Programm geladen wird („**position independence**“) - Sprungziel fast immer in der Nähe des aktuellen PC → 12 Bit Displacement-Länge decken 99% aller Fälle ab	56
	indirekter Registersprung	- zur Implementierung von Sprüngen, deren Ziel zur Compilezeit noch nicht feststeht, z.B. bei case- oder switch-Anweisungen - dynamische Berechnung des Ziels: Sprungziel wird in Register geladen und dann in PC kopiert	58
	Sprung-Implementierungen	- **Condition Code:** in Abhängigkeit von von der Alu gesetzten Flags wird der Sprung ausgeführt. Kostet nichts, da die Alubefehle ohnehin ausgeführt werden, beschränkt aber die Reihenfolge der Befehle - **Bedingungsregister:** Ergebnis des Vergleiches wird in ein Register geschrieben. Einfach, belegt allerdings ein Register - **Vergleich und Sprung:** Vergleich ist Teil des Verzweigungsbefehls. geringere IC, aber CPI steigt	60
	Procedure Calls	- bei Prozeduraufruf muß Prozessorzustand gerettet werden - **Callee Saving:** aufgerufene Prozedur sichert nur die verwendeten Register - **Caller Saving:** Aufrufende Prozedur sichert Prozessorzustand	60
	Zusammenfassung	- die einfachsten Operationen sind die Wichtigsten und müssen somit schnell sein - PC-relative und Register-indirekte Sprünge - Sprungentfernung: 8-12 Bit decken 80% – 100% ab	61
3.4	Datenformate	Kodierung des Operandentyps: im OpCode (heute üblich) oder in Tag, mit dem Operand beginnt	61
	Zusammenfassung Architektur	- 32 Bit Architektur (d.h. Speicherbus ist 32 Bit breit) - wir unterstützen 8, 16, 32, 64-Bit-Speicherzugriffe - 32 Bit Adreßbus	62
3.5	Befehlsformate	- Befehl besteht aus OpCode, u.U. Adressierungsart und Adreßfeldern - zu treffende Entscheidungen: Anzahl der Operanden und der Adressierungsarten - **variabel:** Befehl besteht aus OpCode und einer Variablen Anz. von Paaren aus Adressierungsart und Adreßfeld - **fixed:** alle Befehle gleich lang, feste Anzahl von Adressfeldern, Adressierungsart im OpCode - **hybrid:** OpCode mit 1 Art und 1 Adresse, 2 Arten und 1 Adresse oder 1 Art und 2 Adressen	63
	Befehlslänge	- Abhängig von Anzahl der Operanden und Adressierungsarten - Ziele: So viele Adressierungsarten und Register wie möglich, kompaktes Befehlsformat, Befehlslänge einfach zugreifbar (Vielfaches von Byte) - festes Befehlsformat bedeutet wenige Adressierungsarten	65
	Zusammenfassung	- Optimierung hinsichtlich Performance → festes Befehlsformat mit 32 Bit - Adressierungsarten: nur Displacement und Immediate, indirekt durch LW R1, 0(R2), direkt durch LW	66

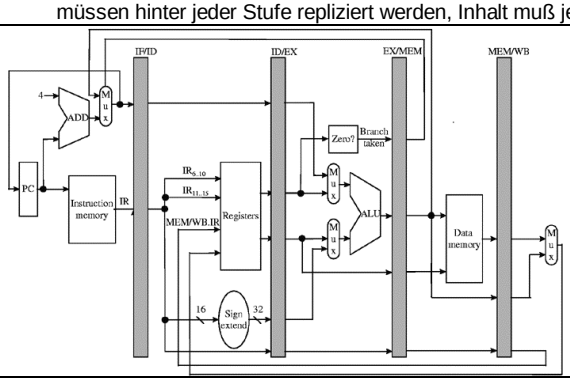
		R1, 100(R0)	
3.6	Compiler	- übersetzen das Programm in mehreren Durchläufen (Passes) jeweils von einer mehr problemorientierten in eine mehr maschinenabhängige Form - Ziele (nach Wichtigkeit): Korrektheit, Geschwindigkeit der Programme, schnelle Compilation, Debugging Möglichkeiten, Schnittstellen zw. versch. Hochsprachen - **Basic Blocks**: Codestücke ohne Verzweigungen, im Durchschnitt 6-7 Befehle lang → wenig Spielraum für Scheduling 	67
	Optimierung	procedure inlining (10%), lokale Optimierungen (5%), lokale und Registerzuweisung (26%), globale und lokale Opt. (14%), globale, lokale Opt. und Registerzuweisung (63%), mit Procedure inlining (81%)	71
	Unterstützung für Compilerbauer	- Orthogonalität: Operation, Datentyp und Adressierungsart sollten orthogonal zueinander (d.h. unabhängig) sein - Einfachheit: Compiler muß für den speziellen Anwendungsfall die optimale Befehlsfolge aus einfachen, atomaren, schnellen Einzelbefehlen zusammensetzen können - Vereinfachung von Tradeoffs: viele GPRs vereinfachen die Entscheidung, ob variable im Register bleibt - Instruktionen zum Binden konstanter Größen als Konstanten zur Compilezeit: Bsp.: wenn klar ist, welche Register gerettet werden müssen, soll der Compiler auch Code dafür generieren können	73

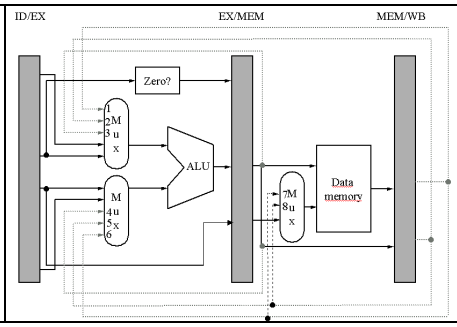
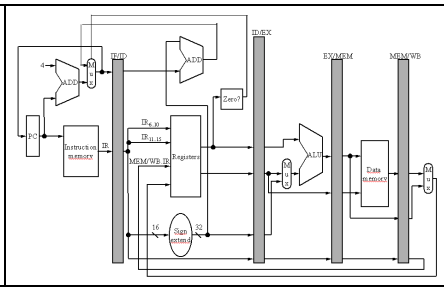
4. DLX

	Vorgaben	- **GPR-Architektur**, Load-Store (Register-Register)-Maschine - **Adressierung**: Register, Displacement, Immediate, Direkt und Indirekt per Displacement - schnelle, einfache Befehle - 8-Bit, 16-Bit, 32-Bit Integer, mit 0-Replikation (unsigned) oder MSB-Replikation (signed) - 32-Bit und 64-Bit Floating Point nach IEEE 754 - **feste Befehlslänge** 32-Bit, wenige Formate			74
	Register	- **32 GPRs** mit 32 Bit Länge, R0 ist immer Null, R31 hält Rücksprungadresse bei Jump und Link - **32 FP-Register** mit 32 Bit Länge, Doubles können auf 2 Register verteilt werden			75
	Befehlsformate				76
	Aufbau	 jede Instruktion braucht 5 Takte:			87
	instruction fetch (IF)	IR ←$_{32}$ Mem[PC] NPC ← PC + 4			87
	instruction decode / register fetch (ID)	A ← Regs[IR$_{6..10}$] B ← Regs[IR$_{11..15}$] IMM ← (IR$_{16}$)^{16} ## IR$_{16..31}$ gilt wg. führender 0 auch für unsigned			87
	Execution / Effective Adress (EX)	**Alu-Befehl** ALUoutput ←A func B or ALUoutput ← A func Imm	**load/store** ALUoutput ←A + Imm	**Verzweigung** ALUoutput ← NPC + Imm Cond ← A op 0	88
	memory access / branch completion (mem)		LMD ← Mem[ALUoutput] oder MEM[ALUoutput] ← B	if cond then PC ← ALUoutput else PC ← NPC	89
	write back (wb)	Regs[IR$_{16..20}$] ← ALUoutput oder Regs[IR$_{11..15}$] ← ALUoutput	Regs[IR$_{11..15}$] ← LMD		89

5. Pipelining

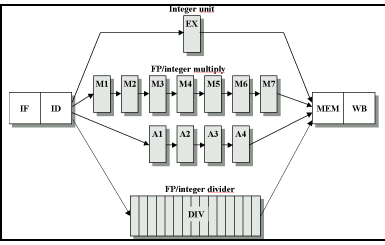
5.1	Pipelining	- Implementierungstechnik, verschiedene Stufen (pages, segments) arbeiten gleichzeitig - Weitergabe bei jedem Takt → Länge des Taktzyklus ist bestimmt durch die Verarbeitungszeit der langsamsten Einheit in der Pipeline → Verarbeitung in allen Stufen sollte etwa gleich lange dauern - kann CPI und/oder Taktzykluszeit verringern	85

		- ist für Benutzer transparent			
	Performance-Gewinn	- maximaler Speedup gleich der Anzahl der Pipelinestufen - Durchsatz wird verbessert, nicht die Ausführungszeit einer einzelnen Instruktion (wird leicht verschlechtert) - DLX ohne Staus: Speedup = 10ns * ((40% + 20%)*4 + 40%*5) / 11ns (100% * 1) = 44ns / 11ns = 4 - allg: $Speedup = \frac{CPI_{unpipelined}}{\underbrace{CPI_{pipelined}}_{1} + stall\ clock\ cycles\ per\ instruction}$			85 95
	Probleme	- **Speicherzugriff**: IF und MEM greifen auf den Speicher zu → bei nur einem Speicher Kollision → getrennter Daten- und Befehlscache - **Register**: ID und WB benutzen Register → 1. Zyklushälfte: schreiben, 2. Zyklushälfte: lesen - **PC**: PC wird in IF inkrementiert, Verzweigungen aber erst in MEM entschieden → evtl. werden „falsche“ Befehle ausgeführt - alle **Stufen der Pipeline gleichzeitig** aktiv → alle Kombinationen müssen möglich sein, Hilfsregister müssen hinter jeder Stufe repliziert werden, Inhalt muß jeweils weitergereicht werden			91
5.2	Aufbau				93
	IF	IF/ID.IR ←$_{32}$ Mem[PC] PC und IF/ID.NPC ← if EX/MEM.cond then (EX/MEM.ALUoutput) else (PC + 4)			94
	ID	ID/EX.A ← Regs[IF/ID.IR$_{6..10}$] ID/EX.B ← Regs[IF/ID.IR$_{11..15}$] ID/EX.IMM ← (IF/ID.IR$_{16}$)^{16} ## IF/ID.IR$_{16..31}$ ID/EX.IR ← IF/ID.IR ID/EX.NPC ← IF/ID.NPC			94
	EX	**Alu-Befehl** EX/MEM.ALUoutput ← ID/EX.A func ID/EX.B oder EX/MEM.ALUoutput ← ID/EX.A func ID/EX.Imm EX/MEM.IR ← ID/EX.IR EX/MEM.cond ← 0	**load/store** EX/MEM.ALUoutput ←ID/EX.A + ID/EX.Imm EX/MEM.cond ← 0 EX/MEM.IR ← ID/EX.IR EX/MEM.B ← ID/EX.B	**Verzweigung** EX/MEM.ALUoutput ← ID/EX.NPC + ID/EX.Imm EX/MEM.Cond ← ID/EX.A op 0	94
	MEM	MEM/WB.IR ← EX/MEM.IR MEM/WB.ALUoutput ← EX/MEM.ALUoutput	MEM/WB.LMD ← Mem[EX/MEM.ALUoutput] oder Mem[EX/MEM.ALUoutput] ← EX/MEM.B MEM/WB.IR ← EX/MEM.IR		94
	WB	Regs[MEM/WB.IR$_{16..20}$] ← MEM/WB.ALUoutput oder Regs[MEM/WB.IR$_{11..15}$] ← MEM/WB.ALUoutput	Regs[MEM/WB.IR$_{11..15}$] ← MEM/WB.LMD		94
5.3	Hazards	Ausführung einer Instruktion in der Pipeline nicht innerhalb eines Taktzyklus möglich → Stau (stall) der Pipeline: „ältere“ Instruktionen müssen weiter laufen, „neuere“ müssen gestaut werden			96
	Kontrollhazards	treten bei Verzweigungen/PC-verändernden Operationen auf			96
	Strukturhazards	Hardware kann die Kombination zweier Operationen in der Pipeline nicht gleichzeitig ausführen, weil - eine Einheit **nicht voll gepipelined** ist (d.h. länger als einen Takt braucht) - weil die **Register** nur einen Schreibvorgang pro Takt erlauben, aber 2 aufeinander folgende Phasen schreiben wollen - IF und MEM wollen gleichzeitig auf gemeinsamen **Daten- und Instruktionscache** zugreifen			97
	Datenhazards	Ergebnis einer Operation ist Operand einer darauffolgenden Instruktion, liegt aber nicht rechtzeitig vor			100
	Forwarding	- beschleunigte Weiterleitung von Ergebnissen aus temporären Registern an die Eingänge der richtigen Verarbeitungseinheit, wenn der natürliche Fluß von Ergebnissen dafür nicht ausreicht - können immer nur eine Stufe nach „rechts“ gehen			102
	RAW	- eine Operation schreibt, eine kurz darauf folgende Instruktion liest und würde noch das alte Ergebnis bekommen → kann häufig mit Forwarding behoben werden, folgen die beiden Instruktionen jedoch direkt aufeinander, wird Stauzyklus (Pipeline Interlock) nötig (Load Slot), der die Pipeline ab der Verbraucherinstruktion staut - **Bsp**: LW R1 R1, 0(R2); SUB R4, R1, R3: LW hat Wert für R1 erst nach deren MEM-Phase, während welcher bereits die EX-Phase von SUB läuft			103
	WAW	2 aufeinander folgende Operationen beschreiben das selbe Register. Braucht die 2. Operation in der Pipeline mehr Takte als die erste, überschreibt sie deren Ergebnis (kann passieren, wenn in unterschiedlichen Phasen der Pipeline in Register/Speicher geschrieben wird)			104

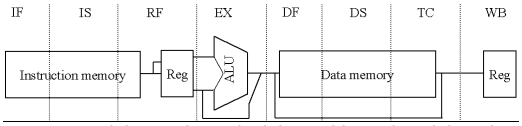
	WAR	ein Befehl muß ein Register lesen, bevor er dazu kommt, hat sein Nachfolger jedoch bereits in dieses Register geschrieben. Kann nur passieren, wenn 1 Stufe vor WB noch gelesen wird	104
	Implementierung DLX	- Erzeugung von Steuersignalen in der ID Phase, um auf Hazards zu reagieren (Forwarding oder Interlock) - Wichtig: Vermeiden von Situationen, bei denen bereits vorgenommene Änderungen des CPU-Status zurückgenommen werden müssen (aufwendig)	114
	mögliche Fälle	- **no dependence** (1): unkritisch - **Dependence requiring stall** (2): LW R1, 0(R2); ADD R5, R1, R7 oder ADD R5, R7, R1 oder Load/Store oder ALU Immediate oder branch/jump register - **dependence overcome by forwarding** (3) - **dependence with acces in order** (4): unkritisch	114
	Forwarding Fälle (3) /Pfade DLX	s. Folie 117 	118
5.4	Gründe Kontrollhazards	- erst in der EX-Phase wird die Zieladresse berechnet und die Sprungbedingung ausgewertet → neuer PC steht erst in der MEM-Phase zur Verfügung - vor ID Phase weiß CPU noch nicht, daß es ein Verzweigungsbefehl ist → Nachfolger kommt bis in die IF-Phase und muß (im Falle des Sprunges) in No-Op umgewandelt werden (IF/ID-Regs auf 0 setzen)	119
	Gegenmaßnahmen Hardware	- früher feststellen, ob verzweigt wird oder nicht - Ziel früher berechnen - zusätzlicher Addierer und Berechnen von COND in ID 	120
	freeze	- Pipeline stauen, bis Ergebnis bekannt ist. - **Vorteil:** einfach in Hard- und Software - **Nachteil**: hoher CPI-Wert	123
	treat as not taken	- es werden weiter folgende Instruktionen geholt und abgearbeitet, im Verzweigungsfall werden diese von Hardware in No-Ops umgewandelt - **Vorteil**: keine Penalty im Nicht-Verzweigungsfall - **Nachteil**: die meisten Verzweigungen werden genommen	123
	treat as taken	- es werden Instruktionen ab dem Sprungziel geholt und abgearbeitet, im Nicht-Verzweigungsfall werden diese in No-Ops umgewandelt - **Vorteile**: keine/geringe Strafe bei Verzweigung - **Nachteil**: hoher Hardwareaufwand für frühe Zielberechnung (idealer Weise in der ersten Phase)	124

	delayed branch	- Delay slots (Befehle, die vor Branchentscheidung geholt werden) werden mittels Scheduling mit von der Verzweigung unabhängigen Instruktionen aufgefüllt
		- **Schedule from before**: immer von Vorteil, benötigt aber einen Befehl, der Bedingung nicht beeinflußt
		- **Schedule from target**: wählt Instruktion vom Sprungziel (wenn Sprung wahrscheinlich)
		  - Instruktion muß kopiert werden, da Anspringen auch von anderer Stelle möglich → IC steigt
		  - Instruktion darf im Nicht-Verzweigungsfall nicht falsch sein (Reg muß redundant sein)
		  - Sprungadresse muß „geändert“ werden
		- **schedule from fall through**: wählt Befehl von „nach dem branch“ (wenn Sprung unwahrscheinlich), Befehl muß aber auch im Sprungfalle korrekt sein
		- **Vorteil** gegenüber „not taken“: from before bringt immer Vorteile, ansonsten kann man Information über Wahrscheinlichkeiten nutzen, Vorteile gering bei tiefer Pipeline (man findet nicht genug geeignete Befehle)
		- zusätzliches Register für Interruptbehandlung notwendig (da PCs nicht aufeinanderfolgend)
			125

	Cancelling Branch	- Compiler gibt Maschinencode Informationen über das prognostizierte Sprungverhalten mit - liegt die Prognose richtig, wird entsprechender Befehl ausgeführt, wenn nicht, wird er gecancelt	129
	Zusammenfassung	die meisten Maschinen bieten **cancelling** und **nicht-cancelling** delayed branches	129
5.5	Latency	Anzahl der Takte, die ein Befehl zusätzlich zur alten EX Phase braucht → soviele+1 Stufen sieht man vor	145
	Initializing Interval	Anzahl der Takte nach Beginn einer Berechnung, nach denen eine neue Berechnung beginnen kann	145

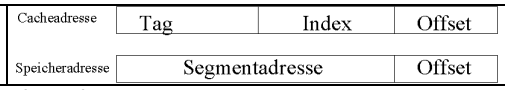
	Erweiterung FP	einen Takt lange FP Operationen in EX Stufe zu integrieren würde hohen Hardwareaufwand nach sich ziehen → **mehrere, unterschiedlich lange EX-Einheiten** (EX, Int/FP mult, FP Add, FP/Int Div) → zusätzliche Register M1/M2, M2/M3, ..., DIV/MEM notwendig → unterschiedlich lange Pipelines verursachen Probleme		144
	RAW Staus	wesentlich häufiger, langwieriger und Kontrolle aufwendiger, obwohl nach gleichen Prinzipien wie bei Integer Pipeline möglich		148
	Strukturhazards	- weil Division nicht gepipelined: nicht schlimm, weil selten - mehrere Stufen wollen gleichzeitig in Register schreiben → Lösungen: mehrere Writeports (zu teuer) oder Befehle geeignet stauen - **erst in MEM-Phase stauen**: Vorteil: man kann evtl vorher auftretende Staus zur Entzerrung nutzen. Nachteil: man muß ganze Pipeline rückwärts stauen (Aufwendig) → - **in ID-Phase** Stauerkennung mittels **Schieberegister** der Länge n des längsten Befehls. Jeder Befehl schreibt eine 1 an die Stelle k, wenn er in n-k Takten schreiben will. Steht dort schon eine 1, wird er einen Takt lang gestaut		149
	WAW Hazards	werden möglich. - geschieht aber nur, wenn das erste Ergebnis nie gebraucht wird, da der zweite Befehl sonst durch den RAW-Hazard verzögert würde → entweder ersten Befehl am Schreiben hindern oder zweiten Befehl geeignet stauen		150

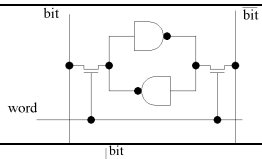
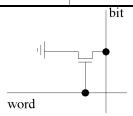
	out of order completion	- wenn frühere, aber längere Befehle nach späteren, aber kürzeren Befehlen fertig werden
		- Problem, wenn letztere einen Interrupt verursachen, zu deren Behandlung erstere vervollständigt werden müssen, wobei ein weiterer Interrupt auftreten kann→
		  - Ignorieren
		  - Ergebnisse zwischenspeichern, bis alle vorherigen Befehle geschrieben haben, erst dann schreiben
		  - Interrupt darf zeitweilig ungenau sein, man hält aber genug Information, um dies durch Interruptroutine zu reparieren
		  - bevor geschrieben wird sicherstellen, daß alle vorherigen Befehle keinen Interrupt mehr erzeugen
				151

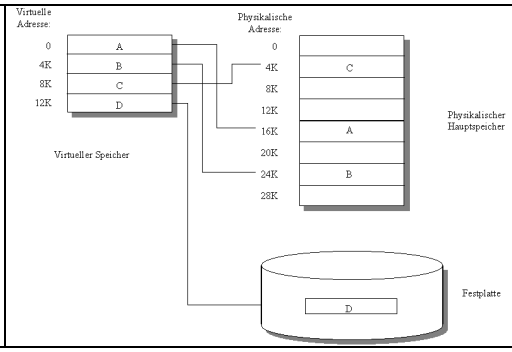
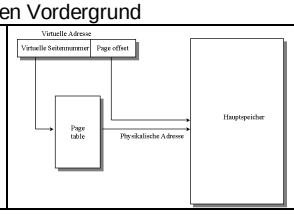
	Zusammenfassung	- Befehle unterschiedlicher Länge führen zu schwieriger Hazarderkennung - RAW-Staus sind die schwerwiegendsten für Performance - Komplizierte Adressierungsarten machen Interruptbehandlung schwierig - Implizit gesetzte Kondition machen Verzweigungsbehandlung schwierig: wenn jeder Befehl Flags setzt, ist es schwer, unabhängige Befehle für Delay-Slot zu finden		153
5.6	MIPS R4000 Pipeline	- 200 MHz, 8 Stufen - Speicherzugriff bei höheren Taktraten Bremse → in den ersten beiden Takten Datum aus Cache, im 3. TagCheck - **IF** IF erste Hälfte; PC Aktualisierung, **IS** IF zweite Hälfte **RF** ID, register fetch, Hazard Erkennung, cache miss Erkennung **EX** ALU, Berechnen der effektiven Adresse, Sprungzielberechnung, Bedingung **DF** erster Speicherzyklus **DS** zweiter Speicherzyklus **TC** cache miss Erkennung **WB** WB  - 2 Loadslots, 3 branch-delay Zyklen mit 1 delay-slot und 2x predict not taken		154
	FP-Pipeline	FP-Einheit hat 8 Stufen, die von Multiplikation, Division und Addition gemeinsam genutzt werden		158
	Performance	- **Integer CPI**: Load- und Branch-Staus spielen Signifikante Rolle - **FP CPI**: RAW-Staus spielen signifikante Rolle, Struktur Hazard eher unwichtig - **Latency** sollte verringert werden, um Datenhazard seltener und weniger aufwendig zu machen		162
	optimale Pipelinetiefe	- hoher CPI-Wert langer Pipelines spricht für kürzere Pipeline - hohe Taktraten langer Pipelines sprechen für längere Pipeline - bei Vektorrechnung wegen starker Unabhängigkeit Tiefen bis 20-30 Stufen sinnvoll - normalerweise optimal: **4 – 8 Stufen**		164
5.7	Interrupts	- **Ursachen**: I/O-Geräteforderung, Arithmetiküberlauf, Breakpoint, Seitenfehler, Hardwarefehler, Betriebssystemaufruf, ... - **Eigenschaften**: Synchron (bei selben Daten immer an der selben Stelle)/asynchron, maskierbar (vom Benutzer unterdrückbar)/nicht maskierbar, während/zwischen Instruktionen, fortführend/terminierend - **Häufigkeit**: im Millisekundenbereich		134
	Behandlung	1. TRAP-Befehl wird in die Pipeline gezwungen und startet Interruptroutine des OS 2. Verursacher und folgende Befehle werden in No-Ops umgewandelt 3. Die Interruptroutine im Betriebssystem rettet als erstes den PC		135
	Reihenfolge	- Interrupts müssen in der Reihenfolge der Befehle abgearbeitet werden, da sonst evtl. Befehle nicht ordnungsgemäß beendet werden - Verwendung eines Schieberegisters - Bei Interrupt wird eine 1 gesetzt, Befehl läuft weiter (um zu gucken, ob ein voriger Befehl noch einen Interrupt verursacht), aber alle Schreibzugriffe werden unterdrückt - Interrupt wird dann in WB-Phase behandelt		136
	genauer Interrupt	- gewährleistet, daß alle vorigen Instruktionen korrekt beendet werden und ab der unterbrochenen		137

		Instruktion wieder gestartet werden kann - Da dies u.U. sehr schwierig, meist 2 Modi: sehr langsamer genauer Modus und z.T. ungenauer Performance Mode	
5.8	Scheduling	- Umstellung der Reihenfolge von Befehlen zur Vermeidung von Pipeline-Staus durch Nutzung von delay slots, stalls und Loadslots - Das Verhalten des Programms muß dabei erhalten bleiben. - Kostet im allgemeines zusätzliche Register. - Abhängigkeiten von Befehlen werden in gerichtetem Graphen dargestellt, dieser dann so topologisch geordnet, daß wenig Staus entstehen	109
	Loop unrolling	- Schleifenrümpfe werden mehrfach hintereinander kopiert, um den Aufwand an branches zu reduzieren - es muß erkannt werden, daß Schleifendurchläufe unabhängig von einander sind	170
	Abhängigkeiten	- sind im Gegensatz zu Staus (Eigenschaften der Pipeline) Eigenschaften von Programmen - können Hazards verursachen, die Reihenfolge von Befehlen erzwingen, die Parallelität begrenzen - **Gegenmaßnahmen**: Abhängigkeiten abschaffen durch Codetransformation, Hazards verhindern unter Beibehaltung der Abhängigkeiten	
	Daten-Abhängigkeiten	- eine Instruktion (oder ihre transitive Hülle) benötigt das Ergebnis einer vorhergehenden Instruktion - solche Instruktionen können nicht gleichzeitig, überlappend oder in umgekehrter Reihenfolge bearbeitet werden - Erkennung bei Speicheradressierung schwieriger als in Registern - Entfernen von Datenabhängigkeiten erfordert Kenntnis über die globale Struktur des Programmes und ist Aufgabe des Compilers - Hazardverhinderung durch Scheduling kann durch Hardware oder Compiler gemacht werden	173
	Namens-abhängigkeiten	- 2 Instruktionen wollen das gleiche Register/die gleiche Speicherstelle (Namen) verwenden, obwohl zwischen ihnen kein Datenfluß stattfindet (zufällig gleicher Name für 2 Operanden) - **Antidependence**: Befehl i liest, danach schreibt j - **output dependence**: erst schreibt Befehl i, dann j - → Verwenden anderer Register (**Register Renaming**), um überschreiben zu verhindern	175
	Kontroll-Abhängigkeiten	- Ausführung eines Befehls ist von einem Sprung abhängig - zum Entfernen braucht man Informationen über die globale Struktur des Programms - **Loop unrolling** ist ein statisches Verfahren (zur Compilezeit)	177

6. Speicherhierarchie

6.1	Cache	- kleiner, schneller Speicher, der räumlich nahe an der CPU angeordnet ist und von der Hardware verwaltet wird, Daten werden immer in Blöcken (4-8 Worte) in den Cache geschrieben - **Cache hit**: Datum wird im Cache vorgefunden - **Cache miss**: Datum wird nicht im Cache vorgefunden - **Miss penaltiy**: hängt ab Zugriffszeit und Bandbreite des Hauptspeichers - **Lokale Miss Rate**: #Misses in diesem Cache/ #Zugriffe auf diesen Cache - **Globale Miss Rate**: #Misses in diesem Cache/ #Zugriffe auf den Speicher insgesamt - **Unified Cache**: gemeinsamer Cache für Daten und Instruktionen - **Split Cache**: getrennte Caches für Instruktionen (deutlich geringere Missrate) und Daten → Insgesamt bessere Hit rate	36 204
	Zugriffszeiten	- **Durchschnittliche Zugriffszeit** = Hit Zeit + Missrate * Miss Penalty - **CPU-Zeit** = (CPU-Taktzyklen + Memory-stall-Zyklen) * Zykluslänge = (CPI * IC + IC * Zugriffe/Instruktion * Miss Rate * Miss Penalty) * Zykluslänge	36 205
	Block Placement	- **Voll assoziativ**: Block kann überall im Speicher stehen - **mengenassoziativ** (b-Weg-assoziativ): b=1, 2, 4, 8; wenn Block die Adresse m hat und der Cache aus k Mengen besteht, kann der Block an einer der b Stellen der Menge m mod k stehen - **direct mapped** (entspr. 1-Weg-Ass.): wenn der Block die Adresse m hat und der Cache k Blöcke faßt, steht er im Block m mod k	197
	Blockidentifikation	**Offset**: Position des Datums im Block/Segment **Index**: bestimmt den Cachebock **Tag**: dient Erkennung des Blockes (Hit/Miss) 	198
	Block replacement	- **FIFO**: einfach zu realisieren, schlechtere Missrate als random - **random**: am einfachsten zu realisieren, bessere Missrate als FIFO - **least recently used**: beste Missrate, aber aufwendig zu realisieren	199
	valid Bit	besagt, ob Daten im Cache valide sind oder Datenmüll enthalten (z.B. nach Einschalten/Taskwechsel)	200
	Zugriffshäufigkeiten	- Datenlesen: 26%, da jede Berechnung 2 Operanden braucht - Datenschreiben: 9% - 75% aller Zugriffe sind Instruktionslesezugriffe, da jeder Befehl aus dem Cache geholt werden muß - → Lesezugriffe sollten optimiert werden	200
	Strategien Write-Hit	- generell: erst Tag Check, dann erst schreiben - **write through**: jeder Schreibvorgang wird auch gleich im Hauptspeicher vorgenommen → HS ist immer aktuell, read misses bewirken kein zurückschreiben, einfach zu realisieren - um **write stalls** (Warten der CPU auf Write through) zu minimieren, wird **write puffer** eingesetzt (trotzdem write stall wenn voll), evtl. aber **write merging möglich** - **write back**: erst wird nur in den Cache geschrieben und **dirty bit** gesetzt. Wird der Block dann ersetzt und ist dirty bit gesetzt, wird der Block in den Speicher zurückgeschrieben → Schreiben mit Geschwindigkeit des Caches möglich, mehrere Writes erzeugen nur einen Speicherzugriff, weniger Speicherverkehr	201
	Strategien Write-Miss	- **fetch on write**: wg. Lokalität wird Block erst geholt und dann geschrieben - **write around**: es wird direkt in den HS geschrieben	202
6.2	Verringerung Miss rate	- **Miss Kategorien**: cold start misses, Capaticity misses, conflict misses - **Größere Blöcke**: verbessern die Hit Rate wg. Lokalität, verschlechtern allerdings miss penalty und	209 -

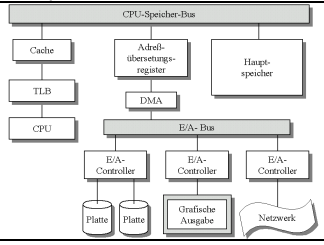
		können bei fester Cachegröße conflict misses hervorrufen; optimale Größe von Speicherbrandbreite und Zugriffszeit abhängig → je nach Cachegröße sind **32-64 Byte** optimal - **Höhere Assoziativität**: verbessert Miss Rate (wg. conflict misses, dabei 8-fach fast so gut wie voll assoziativ) steigert aber Zugriffszeit (wg notwendigem Mux) → bis 16 kB Cachegröße ist **8-fach** optimal, darüber **4-fach** - **Victim Cache**: Kleiner vollassoziativer Cache zwischen Cache und HS, der ersetzte Blöcke aufnimmt. Bei Zugriff wird er parallel geprüft, wird ein Datum hier gefunden, wird es mit dem im Cache getauscht. Geeignet zur Reduzierung von conflict misses bei direct mapped Caches - **Pseudo-assoziative** Caches: bei direct mapped Cache wird erst unter der normalen, bei miss unter einer zweiten Stelle (z.B. mit invertiertem Index-MSB) geschaut → für Hit „erster Klasse" steigt die Zugriffszeit nicht - **Hardware prefetching**: bei miss (insbesondere im Instruction Cache) werden der gesuchte (in den Cache) und der nächste Block (in den **Instruction stream buffer**) geholt. Bei erneutem Miss wird zunächst dort geschaut, bei Treffer wird dieser Block in den Cache übertragen und ein neuer gefetcht. 1 Block kann missrate um **15-25%** reduzieren	219
	Verringerung Miss Penalty	- **Early Restart**: sobald das gewünschte Datum des Blockes im Cache ist, kann die CPU weitermachen - **Critical word first**: Block wird nicht in der ursprünglichen Reihenfolge aus dem HS gelesen, sondern zuerst der Teil des Blockes, in dem das gesuchte Datum steht - **L2 Cache**: langsamerer Cache, der dem L1 nachgeschaltet ist und groß genug, um Miss Rate zu senken → 256-1024 mal größer als L1, höhere Assoziativität zur Senkung der Missrate, Write Through bei L1 und write back bei L2	222 – 224
	Verringerung Hit Zeit	- **small ist fast**: kleiner, direct mapped Cache - **Gepipelinete Schreibzugriffe**: beim schreiben: Delayed Write Buffer zwischen Cache und CPU, in den zuerst geschrieben wird. Bei Hit kopieren in den Cache, sonst write around oder fetch on write	225
6.3	Hauptspeicher	Leistung von **Zugriffszeit, Zykluszeit und Bandbreite** abhängig	226
	SRAM	- **8-16** mal so schnell und teuer wie DRAM - ein Bit wird durch 2 rückgekoppelte **Inverter** stabilisiert - zum Lesen wird die Wordleitung aktiviert, die dann Transistor steuert - verwendet für **Caches** 	226
	DRAM	- optimiert auf **Speicherdichte** und geringe Pinanzahl (durch zeitmultiplexen von Zeilen- (**RAS**) und Spaltenadresse (**CAS**) - Speicherung eines Bits in einem **einzigen Transistor** → Wert liegt nur auf Gate-Source Kapazität → verliert langsam sein Potential → muß nach gewisser Zeit **gerefreshed** werden (lesen und wieder schreiben) - beim Lesen wird Wert zerstört → Wert muß danach zurückgeschrieben werden - wird für **HS** verwendet 	227
	Breite	- Anzahl der Bits, die bei einem Lesezugriff der Außenwelt bereitgestellt werden können - Breite von Caches ist üblicherweise Wortbreite der CPU	228
	höhere Bandbreite	- höhere Breite verbreitert Bus und Kapazität, Multiplexer zur Anpassung der Busbreite zwischen Cache und CPU unerwünscht (kritischer Pfad bzgl. Laufzeit) - verdoppeln der Bandbreite verdoppelt kleinste Erweiterungseinheit	229
	Interleaving	- nutzt potentielle Parallelität durch Organisation des Speichers in Banken, die 1 Wort breit sind, so daß die Speicherbusbreite nicht verändert werden muß - gleiche Adresse wird an alle Banken angelegt → Zeit für das Senden der Adresse nur einmal erforderlich, Übertragungszeit hängt nur noch von der Anzahl übertragener Worte ab → Vorteile, wenn sequentiell auf mehrere Worte zugegriffen wird (Caches laden immer ganze Blöcke) - **Anzahl Banken** sollte größer gleich der Zugriffsdauer in Zyklen einer Bank sein - Problem: sequentielle Zugriffe auf dieselbe Bank z.B. bei Arrays → primzahlige Anzahl Banken, da zu üblichen Zahlen teilerfremd - **Lokation**: Banknummer = Adresse mod Anzahl Banken Adresse in der Bank = Adresse div Anzahl Banken - wenn **Anzahl Banken = 2^k-1**: Adresse in der Bank = Adresse mod (Anzahl Worte pro Bank) (Bitselektion)	229
	DRAM-spezifische Optimierungen	Nibble-Mode Page Mode Static Column	233
6.4	Prozeß	- Programm mit eigenem Adreßraum - bei Prozeßwechsel muß die Hardware den Adreßraum mit wechseln → Aufteilung des Speichers in Blöcke, die Prozesse anfordern können	234
	Protection	- **Schutzmechanismus**, der verhindert, daß ein Prozess auf fremde Speicherblöcke zugreifen kann - bereitstellen eines Registerpaares mit erster (**Base**) und letzter (**Bound**) Speicherstelle des Prozesses, die bei jedem Prozesswechsel vom OS passend gesetzt werden → - Verschiedene OS-Modi nötig: **User Mode** (versch. Zugriffe nicht erlaubt), **System Mode** für OS-Prozesse - spezielle Register in CPU notwendig: Base, Bound, user/Supervisory Bit, Exception enable/disable Bit - Befehle zum Umschalten der Modi nötig (TRAP, RFE bei DLX) - in Page Table für jeden Eintrag ein read und ein write permission flag	234 244

	virtueller Speicher	- Prozesse können Speicher auf eine Weise adressieren, die nicht mit der physikalischen Adressierung übereinstimmen - realer wie virtueller Speicher werden vom **Betriebssystem verwaltet**, Adreßumsetzung gemeinsam mit Hardware - **Page fault**: Zugegriffene Page ist nicht im Speicher, sondern auf der Platte → CPU wartet nicht, sondern schaltet auf anderen Task um - wegen hoher Miss Penalty: **andere Strategien als bei Caches** - **Größe** durch Adreßbreite der CPU bestimmt 	
	Page	- Adreßbereich ist in Blöcke **fester** Länge (4-64KB) eingeteilt, jede Page ist entweder im Speicher oder auf der Festplatte - ersetzen einfach wegen gleicher Länge - interne Fragmentierung - effizienter Plattenverkehr einfach (Größe kann an Zugriffszeit und Transferzeit angepaßt werden)	236
	Segmente	- Blocke variabler Länge (1-2^32Byte) im Speicher - Ersetzen u.U. schwierig (passendes „Loch“ muß gefunden werden) → heute unüblich - externe Fragmentierung - effizienter Plattenverkehr schwierig	238
	Paged Segments	**Kompromiss aus Page und Segment**: Segmente mit Größen eines Vielfachen einer Grundeinheit	239
	Page Placement	**Voll assoziativ**, enorm hohe Miss Penalty stellt geringere Miss Rate in den Vordergrund	239
	Pageidentifikation	- **Page Table**: Tabelle, die die Umsetzung von virtuelle auf physikalische Adresse enthält - wahlweise alle Pages in dieser Tabelle (Tabelle dann sehr groß) oder nur die im HS befindlichen Pages, u.U. mehrstufige Page Tables 	239
	Page replacement	- **LRU** wegen Reduzierung der Missrate - **used bit** wird bei Benutzung auf 1 gesetzt, used Bits von Zeit zu Zeit vom OS auf 0 gestzt, OS führt Buch über Pages, deren used-Bit über längere Zeit 0 war - **alternativ: time stamp**	240
	Schreibstrategien	**write back**, write through wegen langsamer Platte zu teuer	240
	TLB	- jeder Speicherzugriff besteht aus Zweien: einen auf die Page Table, einen auf das Datum selbst - Page Table paßt nicht in den Cache, der verwendet allerdings im Gegensatz zur CPU physikalische Adressen → **Translation-look-aside Buffer** hält einen kleinen Teil der Page Table - häufig kleiner und schneller als Cache selbst - **Einträge:** Tag, Phys. Adresse der Seite, valid bit, used bit, dirty bit, protection information - wenn OS die Page Table ändern will, muß es ggf. zunächst write back aus TLB machen - **Trick**: bei kleinen Caches kann man Index voll in den Bereich des Offsets legen → während der Adreßumsetzung (ergibt Tag für Tag Check) kann man unter dem Index bereits lesen, muß bei Miss dann allerdings wieder verwerfen	241
	Seitengrößen	- für **größere Seiten** spricht: kleinere Page Table, effizienterer Plattenverkehr, „mehr Speicher“ paßt in TLB - für **kleinere Seiten** spricht: geringere interne Fragmentierung, bei Miss schnellere Übertragung	243

7. Platten

	Platten	- nicht flüchriger Langzeitspeicher, unterste Stufe der Speicherhierarchie - Aufteilung der Platten in 500-2500 konzentrische Spuren mit üblicherweise 64 Sektoren - **Zugriffszeit** = Positionierung + Rotationsverzögerung + Transferzeit + Controllerverzögerung	248

8. Busse

8.1	Busse	- Leitung(-sbündel), auf das mehrere Teilnehmer zugreifen können - **geringe Hardwarekosten**, kann aber zum **Flaschenhals** werden - **getrennte Leitungen** für Adresse/Daten oder **Zeitmultiplex** - **synchron** (gemeinsamer Takt, schneller) oder **asynchron** (Protokoll notwendig, längere Distanzen möglich, da Kommunikationspartner warten, bis Transaktion des anderen abgeschlossen ist) - Transaktionen werden vom aktiven Teilnehmer (**Master**) aus betrachtet	249
	CPU-Speicher-Busse I/O-Bus	- gehören unveränderlich zur Architektur → Entwerfer kennt die teilnehmenden Komponenten mit Ihren Lasten - **Teilnehmer**: Cache, Hauptspeicher, I/O-Bus-Controller - 100MB-1GB/s kann lang sein und sehr unterschiedliche Teilnehmer haben 	249
	Memory Mapping	- kleiner Teil des HS wird für I/O reserviert. - Wird diese Adresse angesprochen, weiß der Bus-Adapter, welche I/O-Einheit gemeint ist - Daten werden im Busadapter gepuffert und über I/O-Bus an diese Einheit gesandt/von dieser Einheit geholt	253

	DMA	-	Polling oder Interrupts sehr hoher Overhead → Direct memory access - DMA-Controller (Spezialhardware) wird in Busadapter aufgenommen, dieser kann autonom einen Datenblock vom oder zum Speicher übertragen, während die CPU weiterrechnet - kann als Busmaster agieren - CPU schreibt in DMA-Register, wieviel Daten von wo nach wo übertragen werden sollen, DMA-Controller teilt ihr fertige Übertragung per Interrupt mit	254
8.2	PCI	-	**peripheral component bus** - alle Komponenten sollen den Bus aktiv (als i/O-Controller) nutzen können - 32 Leitungen, Adress/Daten-Multiplexing - 0-33MHz, bis 133 MB/s - 1995: Erweiterung auf 66 MHz und 64Bit Breite → bis 532 MB/s	255
	Arbitration	-	zentraler **Arbiter** vergibt Masterrechte und implementiert fairen und blockierungsfreien Zugriff, jede Einheit ist per REQ# und GNT# direkt mit ihm verbunden - REQ# bittet um Bus, GNT# gewährt ihn - Entziehen des Busses: Arbiter deaktiviert GNT#, Master darf dann Transaktion noch beenden - für mehrere Transaktionen läßt man REQ# aktiviert	259