

Vorlesung: Rechnerstrukturen, Teil 2 (Modul IP7)

J. Zhang

zhang@informatik.uni-hamburg.de

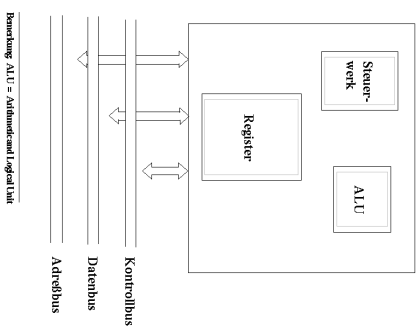
Universität Hamburg
Fachbereich Informatik
AB Technische Aspekte Multimodaler Systeme

Inhaltsverzeichnis

7. Computerarchitektur	182
Grundlagen	182
Befehlsätze	186
Sequenzielle Implementierung	197
Pipelining – Lässt mehr Vorgänge gleichzeitig ablaufen	214

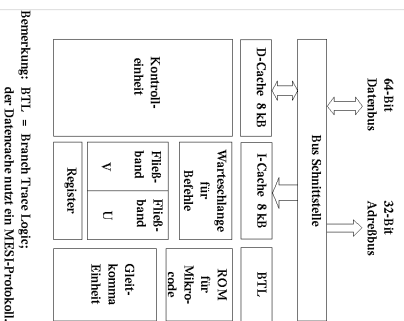
Grundlagen

Bild einer
Zentraleinheit:



Seite 182

Pentium Blockdiagramm



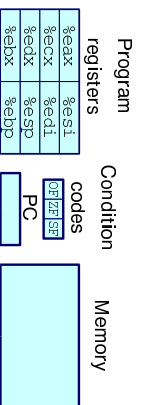
Seite 183

Vorgehensweise

- Mit Designs für einen speziellen Befehlsatz arbeiten
 - ◆ Y86 - eine vereinfachte Version des Intel IA32 (a.k.a. x86)
 - ◆ kennt man einen, kennt man mehr oder weniger alle
- Auf "Mikroarchitekturebene" arbeiten
 - ◆ grundlegende Hardwareblöcke per Assembler in die allgemeine Prozessorstruktur integrieren
 - >> Speicher, funktionale Einheiten, etc.
 - ◆ Mit Kontrolllogik umgehen um den richtigen Ablauf jeder Anweisung zu sichern
- Simple Hardware-Beschreibungssprache zur Beschreibung der Kontrolllogik benutzen
 - ◆ Lässt sich erweitern und modifizieren
 - ◆ Mit Simulationen testen

Seite 184

Y86 Prozessor Zustand



- Programmregister: 8 Register, wie IA32; jeweils 32 bits
- Zustandscodes
 - ◆ Einzel-Bit Flags von arithmetischen oder logischen Anweisungen gesetzt
 - >> OF: Overflow ZF: Zero SF: Negativ
- Programmzähler: gibt die Adresse der Anweisung an
- Speicher
 - ◆ Byte-adressierbarer Speicher
 - ◆ Wörter werden in "little-endian" Byte-Reihenfolge gespeichert

Seite 185

Befehlsätze

Format der Y86 Anweisungen:

- 1 - 6 Bytes an Information werden aus dem Speicher gelesen
 - ◆ Länge der Anweisung kann aus dem ersten Byte bestimmt werden
 - ◆ Nicht so viele Anweisungstypen und einfachere Kodierung als bei IA32
- Jede Anweisung greift auf einen Teil des Programmzustands zu und modifiziert ihn

Registerkodierung

Jedes Register hat 4-Bit ID

%eax	0
%ecx	1
%edx	2
%ebx	3

%esi	6
%edi	7
%esp	4
%ebp	5

- Kodierung wie bei IA32
- Register ID 8 gibt "kein Register" an
- Wir werden dies an zahlreichen Stellen im Hardwaredesign einsetzen

Anweisungsbeispiel – Addition

addl rA, rB 6 0 | rA | rB

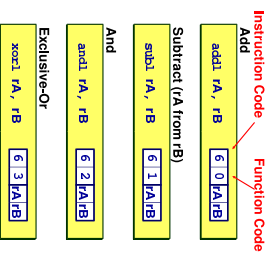
Encoded Representation

Zusätzliche Anweisung

- Addiert Wert im Register rA zu dem im Register rB
 - ◆ Speichert Resultate im Register rB
 - ◆ Merke: Y86 erlaubt Addition nur mit Registerdaten
- Setzt Zustandscodes je nach Ergebnis
- z.B., addl %eax, %esi Kodierung: 60 06
- Zwei-Byte Kodierung
 - ◆ Gibt als Erstes den Anweisungstyp an
 - ◆ Gibt zweitens die Quell- und Zielregister an

Seite 188

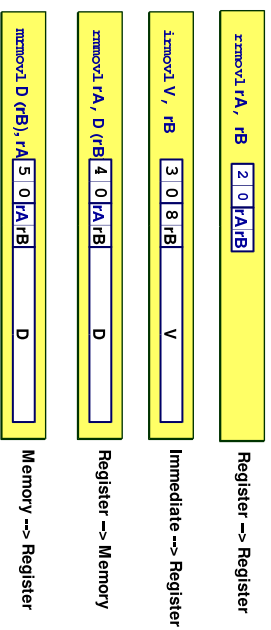
Arithmetische und logische Operationen



- Werden allgemein als "OPI" bezeichnet
- Kodierungen unterscheiden sich nur durch "Funktionscode" – Niederwertigste 4-Bites im ersten Anweisungswort
- Setzt Zustandscodes als Nebeneffekt

Seite 189

“Move” Operationen



- Wie die IA32 movl Anweisung
- Simpleres Format für Speicheradressen
- Verschiedene Namen, um sie auseinander zu halten

Seite 190

CISC Befehlsätze

- “Complex Instruction Set Computer”
- Dominanter Stil durch Mit-80er

- “Stack“-orientierter Befehlsatz
- Gebraucht “Stack” um Argumente zu übergeben und Programmzähler zu speichern
 - Explizite “Push” und “Pop” Anweisungen

Arithmetische Anweisungen können auf Speicher zugreifen

- `addl %eax, 12(%ebx, %ecx, 4)`
- ◆ Erfordert Schreib- und Lesezugriff auf den Speicher
- ◆ Komplexe Adressberechnung

Seite 191

CISC Befehlsätze II

Zustandscodes

- Nebeneffekt arithmetischer und logischer Anweisungen

Philosophie

- Fügt Anweisungen hinzu, um "typische" Programmieraufgaben durchzuführen

Seite 192

RISC Befehlsätze

- "Reduced Instruction Set Computer"
- Internes Projekt bei IBM, von Hennessy (Stanford) und Patterson (Berkeley) später bekannt gemacht

Weniger und simple Anweisungen

- Benötigen i. d. Regel mehr Anweisungen, um eine bestimmte Aufgabe zu erledigen
- Werden mit kleiner, schneller Hardware ausgeführt

Register-orientierter Befehlssatz

- Viel mehr (üblicherweise 32) Register
- Register für Argumente, "Return"-Adressenverweise, Zwischenergebnisse

Seite 193

RISC Befehlsätze II

Nur "load" und "store" Anweisungen können auf Speicher zugreifen

- Wie bei Y86 mrmovl und rmmovl

Keine Zustandscodes

- Testanweisungen liefern 0/1 im Register

CISC gegen RISC

Ursprüngliche Debatte

- Glühende Überzeugungen!
- CISC Anhänger - einfach für den Compiler; weniger Code Bytes
- RISC Anhänger - besser für optimierende Compiler; schnelle Abarbeitung auf einfacher Hardware

Aktueller Stand

- Für Desktop Prozessoren ist die Wahl von ISA kein Thema
 - ◆ Mit genügend Hardware wird alles schnell ausgeführt
 - ◆ Code-Kompatibilität ist wichtiger
- Für eingebettete Prozessoren ist RISC sinnvoll
 - ◆ Kleiner, billiger, weniger Leistung

Zusammenfassung

Y86 Befehlsatz Architektur

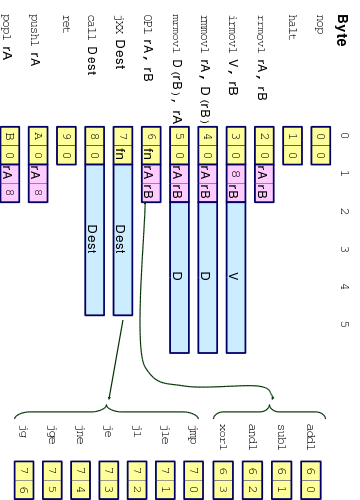
- Gleiche Zustände und Anweisungen wie IA32
- Simple Kodierungen
- Irgendwo zwischen CISC und RISC

Wie wichtig ist ISA Design?

- Heute weniger als früher
 - ◆ mit genügend Hardware wird alles schnell ausgeführt
- Intel bewegt sich weg von IA32
 - ◆ erlaubt nicht genug paralleles Ausführen
 - ◆ hat IA64 eingeführt
 - >> 64-Bit Wortgrößen (überwinden Adressraumlimits)
 - >> Radikal anderer Stil des Befehlsatzes mit expliziter Parallelität
 - >> Benötigt hoch entwickelte Compiler

Sequenzielle Implementierung

Y86 Befehlsätze:



SEQ Hardware Struktur

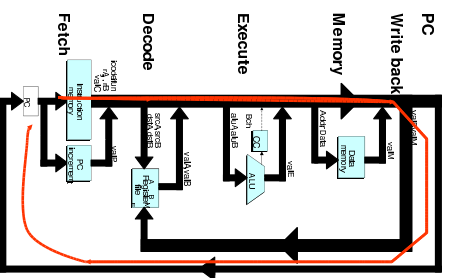
Zustand

- Programmzähler Register (PC)
- Zustandscode Register (CC)
- Registerbank
- Speicher
 - ◆ Gemeinsamer Speicherraum
 - ◆ Daten: zum Lesen/Schreiben von Programmdaten
 - ◆ Anweisung: zum Lesen von Anweisungen

Abarbeitung von Anweisungen

- Lesen der Anweisung bei der von PC angegebenen Adresse
- Verarbeitung durch die Stufen
- Aktualisierung des Programmzählers

Seite 198



Seite 199

SEQ Stufe

Holen (Fetch)

- Lies Anweisung aus Anweisungsspeicher
- Dekodieren (Decode)

● Lies Programmregister

Ausführen (Execute)

- Berechne Wert oder Adresse

Speichern (Memory)

- Lies oder schreib Daten

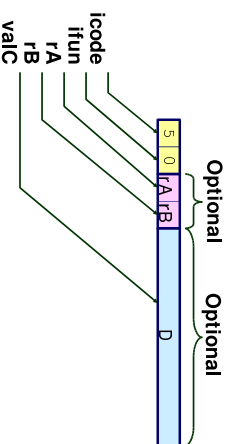
Zurückschreiben (Write Back)

- Schreibe in Programmregister

PC Schreiben

- Aktualisiere Programmzähler

Anweisungsdekodierung



Anweisungsformat

Anweisungs-Byte

Optionales Register Byte

Optionales konstantes Wort

icode:ifun

rA:rB

valC

Ausführen einer Arith./Logischen Operation

OP1 rA, rB



- Holen
- Lies 2 Bytes
- Dekodieren
- Lies Operanden Register
- Ausführen
- Alu Operation, Setze Zustandscodes
- Speichern
- Tue nichts
- Zurückschreiben
- Aktualisiere Register
- PC Schreiben
- Erhöhe PC um 2

Ausführen von rmmovl

rmmovl rA, D (rB



D

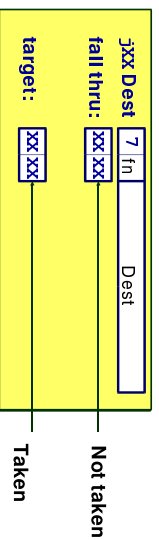
- Holen
- Lies 6 Bytes
- Dekodieren
- Lies Operanden Register
- Ausführen
- Berechne geltende Adresse
- Speichern
- Schreibe in Speicher
- Zurückschreiben
- Tue nichts
- PC Schreiben
- Erhöhe PC um 6

Ausführen von popl



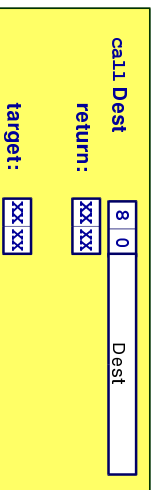
- Holen Les 2 Bytes Dekodieren
- Ausführen Les Stack Adressverweis (Stack Pointer)
- Speichern Erhöhe Stack Adressverweis um 4
- Zurückschreiben Lies aus altem Stack Adressverweis
- PC Schreiben Aktualisiere Stack Adressverweis, schreibe Resultat ins Register
- Erhöhe PC um 2

Ausführen von Jumps



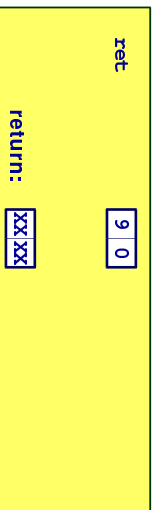
- Holen Lies 5 Bytes, Erhöhe PC um 5
- Dekodieren Tue nichts
- Ausführen Entscheide, ob abhängig vom Sprungbefehl (Jump) und Zustandscode gesprungen wurde
- Speichern Tue nichts
- Zurückschreiben Tue nichts
- PC Schreiben Setze PC auf Dest wenn der Sprung ausgeführt wird, auf erhöhtes PC wenn nicht

Ausführen von call



- Holen
 - Dekodieren
 - Ausführen
 - Speichern
 - Zurückschreiben
 - PC Schreiben
- Lies 5 Bytes, Erhöhe PC um 5
Lies Stack Adressverweis
Verringere Stack Adressverweis um 4
Schreibe erhöhten PC auf neuen Wert
des Stack Adressverweises
Aktualisiere Stack Adressverweis
Setze PC auf Dest

Ausführen von ret

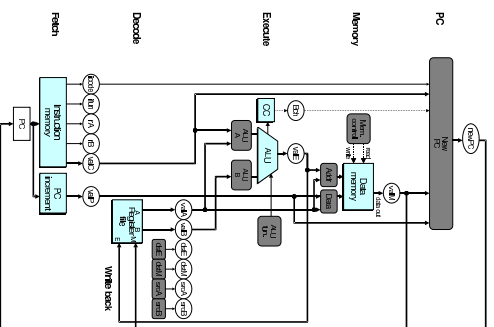


- Holen
 - Dekodieren
 - Ausführen
 - Speichern
 - Zurückschreiben
 - PC Schreiben
- Lies 1 Byte
Lies Stack Adressverweis
Erhöhe Stack Adressverweis um 4
Lies "Return"-Adresse des alten Stack Adressverweises
Aktualisiere Stack Adressverweis
Setze PC auf "Return"-Adresse

SEQ Hardware

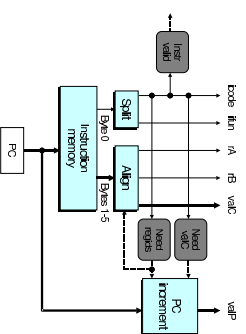
- Blaue Kästchen: vorher entworfene Hardwareblöcke
 - ◆ z.B. Speicher, ALU
- Graue Kästchen: Kontrolllogik
 - ◆ Wird in HCL beschrieben
- Weiße Ovale:
Label für Signale
- Fette Linien:
32-Bit Wortwerte
- Schmale Linien:
4-8 Bit Werte
- Gestrichelte Linien:
1-Bit Werte

Seite 208



Seite 209

“Holen” (Fetch) Logik

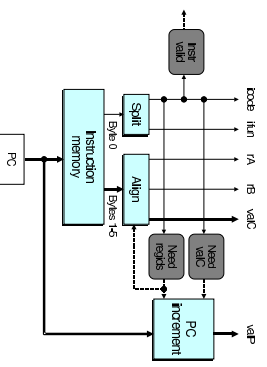


Vordefinierte Blöcke

- “PC” : Register, das den Programmzähler enthält
- “Instruction memory” : Liest 6 Bytes (PC nach PC+5)
- “Split” : Teilt Anweisungsbyte in icode und ifun auf
- “Align” : Holt Felder für rA, rB, und valC

Seite 210

“Holen” (Fetch) Logik



Kontrolllogik

- “Instr. Valid” : Ist diese Anweisung gültig?
- “Need regids” : Enthält diese Anweisung Registeradressen?
- “Need valC” : Enthält diese Anweisung eine Konstante?

Seite 211

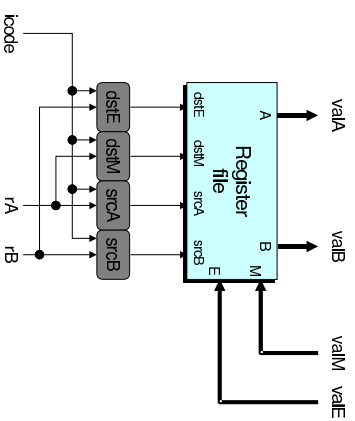
Dekodierungslogik

Registerbank

- Liest Ports A, B
- Schreibt Ports E, M
- Adressen sind Register IDs oder 8 (kein Zugriff)

Kontrolllogik

- srcA, srcB: Lese-Adressen
- dstA, dstB: Schreib-Adressen



Seite 212

SEQ Zusammenfassung

Implementierung

- Jede Anweisung wird als Serie simpler Schritte formuliert
- Gleichartiger Ablauf für jeden Anweisungstyp
- Zusammengesetzt aus Registern, Speicher, vorher entworfenen kombinatorischen Blöcken
- Verbunden mit Kontrolllogik

Limitierungen

- Zu langsam für den praktischen Einsatz
- Muss in einem Zyklus auf Anweisungsspeicher, Registerdatei, ALU und Datenspeicher zugreifen
- Takt müsste sehr langsam laufen
- Hardwareinheiten sind nur für Bruchteile eines Taktzyklus aktiv

Seite 213

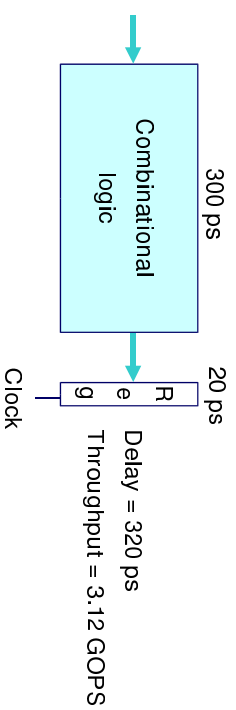
Pipelining

“Real-World Pipelines”: Autowaschanlagen.

Konzept

- Prozess in unabhängige Abschnitte aufteilen
- Objekt sequentiell durch diese Abschnitte laufen lassen
- Zu jedem gegebenen Zeitpunkt werden zahlreiche Objekte bearbeitet

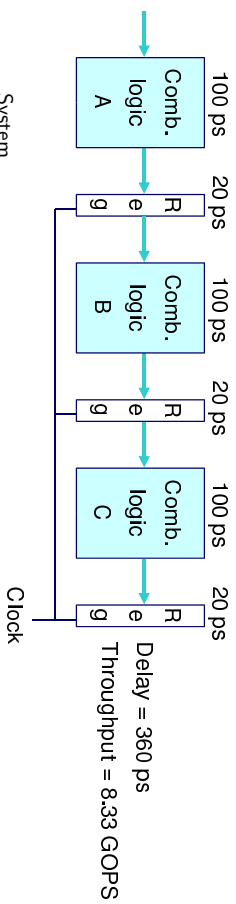
Berechnungsbeispiel



System

- Berechnung erfordert insgesamt 300 Pikosekunden
- Zusätzliche 20 Pikosekunden um das Resultat im Register zu speichern
- kann/ muss Zykluszeit von mindestens 320 ps haben

3-Wege "Pipelined" Version



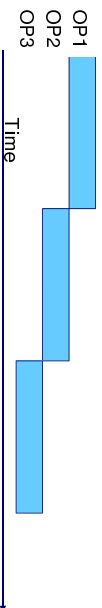
System

- Teilt Kombinationslogik in 3 Blöcke zu je 100 ps auf
- Kann neue Operation anfangen, sobald die vorherige durch Abschnitt A durchgelaufen ist. – Beginnt alle 120 ps neue Operation
- Allgemeine Latenzzunahme
 - ◆ 360 ps vom Start zum Ende

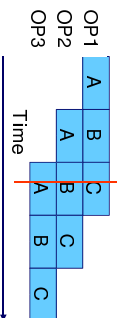
Seite 216

Pipeline-Diagramme

Unpipelined

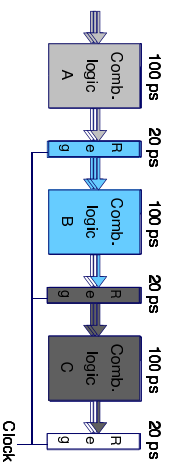
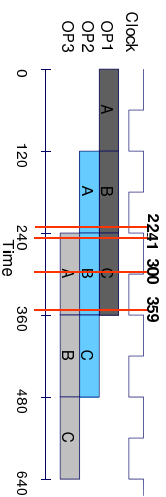


3-Way Pipelined



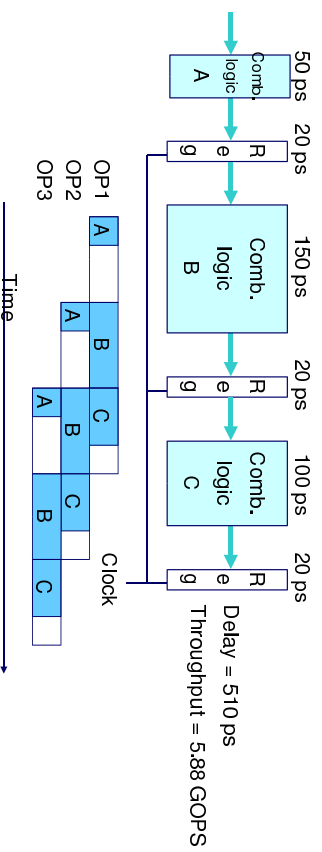
Seite 217

Funktionsweise einer Pipeline



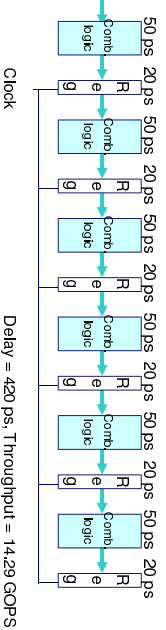
Seite 218

Limitierungen: Nichtuniforme Verzögerungen



Seite 219

Limitierungen: Register "Overhead"



- Mit der Länge der Pipeline gewinnt der Anteil des (registerbedingten) Overhead an Bedeutung
- Prozent der Taktzeit, der für das Laden des Registers verbraucht wird:
 - ◆ 1-Register "Pipeline": 6,25%
 - ◆ 3-Register "Pipeline": 16,67%
 - ◆ 6-Register "Pipeline": 28,57%
- Hohe Geschwindigkeiten moderner Prozessordesigns werden durch sehr grobe "Pipelinelänge" erreicht

Seite 220

Ergänzende Literatur

Zur Rechnerarchitektur (Teil 2, 5. Termin):

Literatur

- [1] Randal E. Bryant and David O'Hallaron. Computer systems. pages 258–317. Pearson Education, Inc., New Jersey, 2003.

Seite 221