高等计算机系统结构
引 论
（第一讲）

程旭

2008年9月22日

“高等计算机系统结构”的教学目标

学习和把握将决定二十一世纪计算机具体形态的设计技术、机器结构、工艺要素、评价方法等

技术工艺
应用

并行性

编程语言

计算机系统结构
•指令系统设计
•组成
•硬件

软件/硬件界面设计(ISA)

测量和评测

历史

操作系统


cpu 基本逻辑单元
处理器基础知识

数据结构
应用系统
C语言编译
数字逻辑

cpu 存储器
调度并发
代码生成
优化

本课程在教学安排中的地位

• 计算机应用需要什么？
• 操作系统需要哪些功能支持？
• 优化编译可以利用和实现哪些功能？
• 我们能够建造什么样的机器？
• 今后的计算机将会怎样？

计算机系统结构研究人员必须具有宽厚的专业知识！
Charles Babbage 1791-1871

Lucasian Professor of Mathematics, Cambridge University, 1827-1839

---

Charles Babbage

° Difference Engine 1823

° Analytic Engine 1833

- The forerunner of modern digital computer!

Application
- Mathematical Tables – Astronomy
- Nautical Tables – Navy

Background
- Any continuous function can be approximated by a polynomial --- Weierstrass

Technology
- mechanical - gears, Jacquard's loom, simple calculators

---

Difference Engine

A machine to compute mathematical tables

Weierstrass:
- Any continuous function can be approximated by a polynomial
- Any polynomial can be computed from difference tables

An example

\[ f(n) = n^2 + n + 41 \]
\[ d1(n) = f(n) - f(n-1) = 2n \]
\[ d2(n) = d1(n) - d1(n-1) = 2 \]
\[ f(n) = f(n-1) + d1(n) = f(n-1) + (d1(n-1) + 2) \]

all you need is an adder!

<table>
<thead>
<tr>
<th>n</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>d2(n)</td>
<td></td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d1(n)</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>f(n)</td>
<td>41</td>
<td>43</td>
<td>47</td>
<td>53</td>
<td>61</td>
</tr>
</tbody>
</table>

---

Babbage’s Difference Engine 1 1832
Analytic Engine

1833: Babbage's paper was published
- conceived during a hiatus in the development of the difference engine

Inspiration: Jacquard Looms
- looms were controlled by punched cards
  - The set of cards with fixed punched holes dictated the pattern of weave ⇒ *program*
  - The same set of cards could be used with different colored threads ⇒ *numbers*

1871: Babbage dies
- The machine remains unrealized.
- It is not clear if the analytic engine could be built even today using only mechanical technology

1834 Babbage Analytical Engine

- The Mill: Memory unit consisting of counter wheels
- The Store: The arithmetic unit capable of 4 operations used a pair of register and produced results stored in another register in the store
- Operation Cards: Specified one of Four operations
- Variable Cards: Specified the memory location to be used
- Output: Printer or punch
Analytic Engine
The first conception of a general-purpose computer

1. The *store* in which all variables to be operated upon, as well as all those quantities which have arisen from the results of the operations are placed.
2. The *mill* into which the quantities about to be operated upon are always brought.

The program

<table>
<thead>
<tr>
<th>Operation</th>
<th>variable1</th>
<th>variable2</th>
<th>variable3</th>
</tr>
</thead>
</table>

An operation in the *mill* required feeding two punched cards and producing a new punched card for the *store*.

An operation to alter the sequence was also provided!

The first programmer
Ada Byron aka “Lady Lovelace” 1815-52
Ada’s tutor was Babbage himself!

1937, Alan Turing
While not using the practical technology of the era, Alan Turing developed the idea of a "Universal Machine" capable of executing any describable algorithm, and forming the basis for the concept of "computability". Perhaps more importantly Turing's ideas differed from those of others who were solving arithmetic problems by introducing the concept of "symbol processing".

1946年2月14日
J. Presper Eckert & John Mauchly
Moore School
University of Pennsylvania
Size: 80 feet long
8.5 feet high
18,000 vacuum tubes
5000 additions/sec.

The world's first general-purpose electronic computer conditional Jump and be programmable, distinguished it from earlier ones
Used for computing artillery firing tables

First General Electronic Computer -- ENIAC
Electronic Numerical Integrator and Calculator
Accumulator
° 28 vacuum tubes

ENIAC’S Application: Ballistic calculations
angle = f (location, tail wind, cross wind, air density, temperature, weight of shell, propellant charge, ...)

ENIAC was NOT a “stored program” device
° For each problem, someone analyzed the arithmetic processing needed and prepared wiring diagrams for the computers to use when wiring the machine
° Process was time consuming and error prone
° Cleaning personnel often knocked cables out of their place and just put them back somewhere

Wiring the machine
Electronic Discrete Variable Automatic Computer (EDVAC)

- ENIAC’s programming system was external
  - Sequences of instructions were executed independently of the results of the calculation
  - Human intervention required to take instructions “out of order”
- Eckert, Mauchly, John von Neumann and others designed EDVAC (1944) to solve this problem
  - Solution was the stored program computer
    ⇒ “program can be manipulated as data”
- First Draft of a report on EDVAC was published in 1945, but just had von Neumann’s signature!
  - In 1973 the court of Minneapolis attributed the honor of inventing the computer to John Atanasoff

The von Neumann Machine

- Stored Program Computer
- IAS (Institute for Advanced Study)
- Computer

- Main Memory
- Arithmetic Logic Unit
- Program Control Unit
- I/O Equipment

1946

存储程序的思想 即构成计算机程序的指令可同数据一样事先存放到存储器中，然后由计算机自己一条条取出执行。

这种思想很自然地引出了转移指令和可对指令的地址部分进行修改的概念，从而使一段程序的指令可以自动地被有意义地多次执行。
First, a set of comprehensive, operational, stored-program computers—EDSAC

The world’s first full-scale, operational, stored-program computer

Maurice Wilkes, Cambridge University
EDSAC: Electronic Delay Storage Automatic Calculator

1949年，EDSAC开始运行

其基于累加器的结构和其指令系统设计对以后一段时期的机器设计有着重要影响

Bell Labs

° 1940: Ohl develops the PN Junction
° 1945: Shockley’s laboratory established
° 1947: Bardeen and Brattain create point contact transistor (U.S. Patent 2,524,035)

Bell Labs

° 1951: Shockley develops a junction transistor manufacturable in quantity (U.S. Patent 2,623,108)

The Integrated Circuit

° 1959: Jack Kilby, working at TI, dreams up the idea of a monolithic “integrated circuit”
° Components connected by hand-soldered wires and isolated by “shaping”, PN-diodes used as resistors (U.S. Patent 3,138,743)
**Integrated Circuits**

- 1961: TI and Fairchild introduce the first logic ICs ($50 in quantity)
- 1962: RCA develops the first MOS transistor

**The Microprocessor**

- 1971: Intel introduces the 4004
  - General purpose programmable computer instead of custom chip for Japanese calculator company

**Microprocessor Performance**

- **Intel 4004 (1971)**: 4-bit processor, 2312 transistors, 0.4 MHz, 10 micron PMOS, 11 mm² chip
- **RISC II (1983)**: 32-bit, 5 stage pipeline, 40,760 transistors, 3 MHz, 3 micron NMOS, 60 mm² chip
  - 125 mm² chip, 0.065 micron CMOS
- **Intel 486™ DX CPU (1992)**: 33 MHz, 41 MIPS
- **Pentium™ Processor (1995)**: 333 MHz, 2.9 MIPS
- **Pentium® 4 Processor (2000)**: 3 GHz, 2312 RISC II+FPU+Icache+Dcache
- **Intel® Celeron® Processor (1998)**: 1.3 GHz, 2312 RISC II+FPU+Icache+Dcache

**Sea Change in Chip Design**

- **Intel 4004 (1971)**: 4-bit processor, 2312 transistors, 0.4 MHz, 10 micron PMOS, 11 mm² chip
- **RISC II (1983)**: 32-bit, 5 stage pipeline, 40,760 transistors, 3 MHz, 3 micron NMOS, 60 mm² chip
  - 125 mm² chip, 0.065 micron CMOS
    - RISC II shrinks to ~0.02 mm² at 65 nm
Multicore

- Small number of cores, shared memory
- Some systems have multithreaded cores
- Trend to simplicity in cores (e.g. no branch prediction)
- Multiple threads share resources (L2 cache, maybe FP units)
- Deployment in embedded market as well as other sectors

Cell from IBM and Sony

- Heterogeneous multi-core system architecture
  - Power Processor Element for control tasks
  - Synergistic Processor Elements for data-intensive processing
- Synergistic Processor Element (SPE) consists of
  - Synergistic Processor Unit (SPU)
  - Synergistic Memory Flow Control (MFC)
    - Data movement and synchronization
    - Interaction to high-performance Element Interconnect Bus

Intel 80核芯片 (2007)

- 80个处理核心
- 1 Teraflop
- 100亿次运算/瓦特
- 主频3.1GHz
- 面积300mm²
- 各CPU内核与内存1对1地连接，分别拥有256MBps的内存带宽
- 32MB的片上静态RAM
- 单芯片整体的内存带宽达到了1TB/s

CPU技术发展简史

- Charles Babbage’s Engines (1823)
- Turing Machine (1937)
- ENIAC (1946) EDSAC (1949)
- CPU
- Microprocessor
- General Purpose Microprocessor VS. special CPU for HPC
- Multicore, Manycore or …
因特网访问方式的改进

<table>
<thead>
<tr>
<th>1985</th>
<th>1995</th>
<th>2005</th>
</tr>
</thead>
<tbody>
<tr>
<td>9百 万套</td>
<td>6千万套</td>
<td>2亿5千万套</td>
</tr>
</tbody>
</table>

信息家电
- 手持 Hand-helds
- 无线、手机 Cellphones & phone access
- 机顶盒 网络计算机 Set-tops & NCs
- 游戏机 Game Consoles

因特网访问方式的改进

因特网：网络的网络
- 公共和私用广域网
- 局域网和家庭网
- 人体网络
- 系统级网络
- 汽车、飞机等网络

电脑空间与人和其他物理世界的数字接口

驱动后PC时代的两大技术:
1) 移动消费类设备
   - 例如：新一代PDA、新一代移动通信设备、可穿戴计算机
2) 支持上述设备的基础设施:
   - 例如：新一代Big Fat Web Servers、Database Servers
嵌入式微处理器

What?

A programmable processor whose programming interface is not accessible to the end-user of the product. 

The only user-interaction is through the actual application.

Examples:
- Sharp PDA's are encapsulated products with fixed functionality 
- 3COM Palm pilots were originally intended as embedded systems. Opening up the programmers interface turned them into more generic computer systems.

Some interesting numbers

- The Intel 4004 was intended for an embedded application (a calculator)
- Of todays microprocessors
  - 95% go into embedded applications 
  - ARM: best selling embedded microprocessor (2001-)
  - 50% of microprocessor revenue stems from embedded systems
- Often focused on particular application area
  - Microcontrollers
  - DSPs
  - Media Processors
  - Graphics Processors
  - Network and Communication Processors
不同的评价标准

- Power
- Cost
- Flexibility

Flexible
Power
Cost
Performance as a Functionality Constraint
("Just-in-Time Computing")

- Components of Cost
  - Area of die / yield
  - Code density (memory is the major part of die size)
  - Packaging
  - Design effort
  - Programming cost
  - Time-to-market
  - Reusability

集成电路的成本

\[
\text{Die\_cost} = \frac{\text{Wafer\_cost}}{\text{Dies\_per\_wafer} \times \text{Die\_yield}}
\]

\[
\text{Dies\_per\_wafer} = \frac{\pi \times (\text{Wafer\_diameter}/2)^2}{\text{Die\_area}} - \frac{\pi \times \text{Wafer\_diameter}}{\sqrt{2} \times \text{Die\_area}}
\]

\[
\text{Die\_yield} = \text{Wafer\_yield} \times \left(1 + \frac{\text{Defects\_per\_unit\_area} \times \text{Die\_area}}{\alpha} \right)^{-\alpha}
\]

若\(\alpha = 3\)，晶模成本大致以晶模大小的四次方增长
封装成本：取决于管脚数量和散热要求

IC cost = Die cost + Testing cost + Packaging cost

Final test yield

Cost/Performance
What is Relationship of Cost to Price?

- Component Costs
- Direct Costs (add 25% to 40%) recurring costs: labor, purchasing, scrap, warranty
- Gross Margin (add 82% to 186%) nonrecurring costs:
  R&D, marketing, sales, equipment maintenance, rental, financing cost, pretax profits, taxes
- Average Discount to get List Price (add 33% to 66%): volume discounts and/or retailer markup

**Chip** | **Die cost** | **Package pins** | **Type cost** | **Test & Total Assembly** | **Costs**
--- | --- | --- | --- | --- | ---
386DX | $4 | 132 QFP | $1 | $4 | $9
486DX2 | $12 | 168 PGA | $11 | $12 | $35
PowerPC 601 | $53 | 304 QFP | $3 | $21 | $77
HP PA 7100 | $73 | 504 PGA | $35 | $16 | $124
DEC Alpha | $149 | 431 PGA | $30 | $23 | $202
SuperSPARC | $272 | 293 PGA | $20 | $34 | $326
Pentium | $417 | 273 PGA | $19 | $37 | $473

真实示例

Chip Metal Line Wafer Defect Area Dies/ Yield Die Cost
--- | --- | --- | --- | --- | ---
386DX | 2 layers 0.90 | $900 1.0 43 360 71% | $4
486DX2 | 3 layers 0.80 | $1200 1.0 61 181 54% | $12
PowerPC 601 | 4 layers 0.80 | $1700 1.3 121 115 28% | $53
HP PA 7100 | 3 layers 0.80 | $1300 1.0 196 66 27% | $73
DEC Alpha | 3 layers 0.70 | $1500 1.2 234 53 19% | $149
SuperSPARC | 3 layers 0.70 | $1700 1.6 256 48 13% | $272
Pentium | 3 layers 0.80 | $1500 1.5 296 40 9% | $417

From “Estimating IC Manufacturing Costs”, by Linley Gwennap,
*Microprocessor Report*, August 2, 1993, p. 15

功耗密度进一步恶化

Nuclear Reactor
Rocket Nozzle
Sun's Surface

Surpassed hot-plate power density in 0.5 μ
Not too long to reach nuclear reactor
优化能耗

- 高性能通用微处理器（例如，Pentiums）
  • 10~100 Watts, 100~1000MIPS = 0.01 Mips/mW

- 节能通用微处理器（例如，StrongARM）
  • 0.5 Watts, 160 MIPS = 0.3 Mips/mW

- 节能专用处理器（例如，MPEG2）
  • 100 Mops/mW

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

开关能耗

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

Switching Energy

\[
\text{Energy/transition} = C_L \times V_{dd}^2 \\
\text{Power} = \text{Energy/transition} \times f = C_L \times V_{dd}^2 \times f
\]

MIMD

Multiprocessors and Multicomputer Clusters

MIMD

Multicomputers

(Single address space with shared-memory)

NUMA

PVP (Cray T90)

SMP (Intel SHV, Sun, SGI, IBM)

COMA (KSR-1)

CC-NUMA (Standord Dash, SGI Origin,

NCC-NUMA (Cray T3E, etc.)

DSM or SC-NUMA

TreadMarks, Wind Tunnel, Shrimp)

Cluster (IBM SP2, TrueCluster,

Solaris MC, Tandem Hymalaya,

Wolfpack, NOW, PearlCluster)

MPP (Intel Option Red, IBM Blue

Pacific, SGI/Cray Blue Mountain)

SMP

MIMD

Multiprocessors

(Untied address space

with shared-memory)

MPP

SMP

NUMA

PVP

Multiprocessors and Multicomputer Clusters

MIMD

Multicomputers

(Single address space

with shared-memory)

NUMA

PVP (Cray T90)

SMP (Intel SHV, Sun, SGI, IBM)

COMA (KSR-1)

CC-NUMA (Standord Dash, SGI Origin,

NCC-NUMA (Cray T3E, etc.)

DSM or SC-NUMA

TreadMarks, Wind Tunnel, Shrimp)

Cluster (IBM SP2, TrueCluster,

Solaris MC, Tandem Hymalaya,

Wolfpack, NOW, PearlCluster)

MPP (Intel Option Red, IBM Blue

Pacific, SGI/Cray Blue Mountain)

Nine Computer Price Tiers

1$: embeddables e.g. greeting card

10$: wrist watch & wallet computers

100$: pocket/ palm computers

1,000$: portable computers

10,000$: personal computers (desktop)

100,000$: departmental computers (closet)

1,000,000$: site computers (glass house)

10,000,000$: regional computers (glass castle)

100,000,000$: national centers

Super server: costs more than $100,000

“Mainframe”: costs more than $1 million

an array of processors, disks, tapes, comm ports

Nine Computer Price Tiers

1$: embeddables e.g. greeting card

10$: wrist watch & wallet computers

100$: pocket/ palm computers

1,000$: portable computers

10,000$: personal computers (desktop)

100,000$: departmental computers (closet)

1,000,000$: site computers (glass house)

10,000,000$: regional computers (glass castle)

100,000,000$: national centers

Super server: costs more than $100,000

“Mainframe”: costs more than $1 million

an array of processors, disks, tapes, comm ports
What is Computer Architecture?

In its broadest definition, computer architecture is the design of the abstraction layers that allow us to implement information processing applications efficiently using available manufacturing technologies.

Abstraction Layers in Modern Systems

The End of the Uniprocessor Era

Single biggest change in the history of computing systems

Conventional Wisdom in Computer Architecture

- Old Conventional Wisdom: Power is free, Transistors expensive
- New Conventional Wisdom: "Power wall" Power expensive, Transistors free
  (Can put more on chip than can afford to turn on)
- Old CW: Sufficient increasing Instruction-Level Parallelism via compilers, innovation (Out-of-order, speculation, VLIW, …)
- New CW: “ILP wall” law of diminishing returns on more HW for ILP
- Old CW: Multiplies are slow, Memory access is fast
- New CW: “Memory wall” Memory slow, multiplies fast
  (200 clock cycles to DRAM memory, 4 clocks for multiply)
- Old CW: Uniprocessor performance 2X / 1.5 yrs
- New CW: Power Wall + ILP Wall + Memory Wall = Brick Wall
  * Uniprocessor performance now 2X / 5(?) yrs
  ⇒ Sea change in chip design: multiple “cores”
    (2X processors per chip / ~ 2 years)
  * More, simpler processors are more power efficient
Uniprocessor Performance

- VAX: 25%/year 1978 to 1986
- RISC + x86: 52%/year 1986 to 2002
- RISC + x86: ??%/year 2002 to present


Problems with Sea Change

- Algorithms, Programming Languages, Compilers, Operating Systems, Architectures, Libraries, ... not ready to supply Thread-Level Parallelism or Data-Level Parallelism for 1000 CPUs / chip,
- Architectures not ready for 1000 CPUs / chip
  - Unlike Instruction-Level Parallelism, cannot be solved by computer architects and compiler writers alone, but also cannot be solved without participation of architects

Instruction Set Architecture: Critical Interface

- Properties of a good abstraction
  - Lasts through many generations (portability)
  - Used in many different ways (generality)
  - Provides convenient functionality to higher levels
  - Permits an efficient implementation at lower levels

Instruction Set Architecture

“... the attributes of a [computing] system as seen by the programmer, i.e. the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls the logic design, and the physical implementation.” — Amdahl, Blaauw, and Brooks, 1964

- Organization of Programmable Storage
- Data Types & Data Structures: Encodings & Representations
- Instruction Formats
- Instruction (or Operation Code) Set
- Modes of Addressing and Accessing Data Items and Instructions
- Exceptional Conditions
Example: MIPS

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>r1</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>r31</td>
<td>PC</td>
</tr>
</tbody>
</table>

Programmable storage
- $2^{32} \times$ bytes
- 31 x 32-bit GPRs (R0=0)
- 32 x 32-bit FP regs (paired DP)

HI, LO, PC

Arithmetic logical
- Add, AddU, Sub, SubU, And, Or, Xor, Nor, SLT, SLTU,
- Addi, AddIU, SLLI, SLLIU, Andi, Ori, Xori, LUI
- SLL, SRL, SRA, SLLV, SRLV, SRAV

Memory Access
- LB, LBU, LH, LHU, LW, LWL, LWR
- SB, SH, SW, SWL, SWR

Control
- J, JAL, JR, JALR
- BEq, BNE, BGTZ, BLTZ, BGZ, BLTZAL, BGEZAL

32-bit instructions on word boundary

Computer Architecture is an Integrated Approach

- What really matters is the functioning of the complete system
  - hardware, runtime system, compiler, operating system, and application
  - In networking, this is called the “End to End argument”

- Computer architecture is not just about transistors, individual instructions, or particular implementations
  - E.g., Original RISC projects replaced complex instructions with a compiler + simple instructions

Computer Architecture is Design and Analysis

- Computer system structure is an iterative and research process
- Study of system structure at all levels
- Explore design space

Creativity

Good ideas vs. bad ideas

Performance vs. cost

Computer architecture and VLSI

Computer system structure

Input/Output with storage

- 3x32 bits
- RAID
- DRAM
- Memory hierarchy
- Cache

- Data transfer and memory hierarchy
- Transient error correction, interleave, latency
- Cache technology, cache hierarchy, cache coherence
- VLSI, microarchitecture, processor, system, network

Computer Architecture is an Integrated Approach

- What really matters is the functioning of the complete system
  - hardware, runtime system, compiler, operating system, and application
  - In networking, this is called the “End to End argument”

- Computer architecture is not just about transistors, individual instructions, or particular implementations
  - E.g., Original RISC projects replaced complex instructions with a compiler + simple instructions
### What Computer Architecture brings to Table

- Other fields often borrow ideas from architecture
- Quantitative Principles of Design
  1. Take Advantage of Parallelism
  2. Principle of Locality
  3. Focus on the Common Case
  4. Amdahl’s Law
  5. The Processor Performance Equation
- Careful, quantitative comparisons
  - Define, quantify, and summarize relative performance
  - Define and quantify relative cost
  - Define and quantify dependability
  - Define and quantify power
- Culture of anticipating and exploiting advances in technology
  - Culture of well-defined interfaces that are carefully implemented and thoroughly checked

### 1) Taking Advantage of Parallelism

- Increasing throughput of server computer via multiple processors or multiple disks
- Detailed HW design
  - Carry lookahead adders uses parallelism to speed up computing sums from linear to logarithmic in number of bits per operand
  - Multiple memory banks searched in parallel in set-associative caches
- Pipelining: overlap instruction execution to reduce the total time to complete an instruction sequence.
  - Not every instruction depends on immediate predecessor \( \Rightarrow \) executing instructions completely/partially in parallel possible
  - Classic 5-stage pipeline:
    1) Instruction Fetch (Ifetch),
    2) Register Read (Reg),
    3) Execute (ALU),
    4) Data Memory Access (Dmem),
    5) Register Write (Reg)
### Pipelined Instruction Execution

**Time (clock cycles)**

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
<td>DMem</td>
<td>Ifetch</td>
<td>Reg</td>
<td>ALU</td>
</tr>
</tbody>
</table>

**Limits to pipelining**

- **Hazards** prevent next instruction from executing during its designated clock cycle
  - **Structural hazards**: attempt to use the same hardware to do two different things at once
  - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline
  - **Control hazards**: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

### 2) The Principle of Locality

- **The Principle of Locality**:
  - Program access a relatively small portion of the address space at any instant of time.
- **Two Different Types of Locality**:
  - **Temporal Locality** (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
  - **Spatial Locality** (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straight-line code, array access)
- Last 30 years, HW relied on locality for memory perf.

### Levels of the Memory Hierarchy

<table>
<thead>
<tr>
<th>Capacity</th>
<th>Access Time</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Registers</td>
<td>100 bytes</td>
<td>300 - 500 ps (0.3 - 0.5 ms)</td>
</tr>
<tr>
<td>L1 and L2 Cache</td>
<td>10x - 100 x K bytes</td>
<td>~1 ns - ~10 ns</td>
</tr>
<tr>
<td>~ $1000/ Gbyte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Main Memory</td>
<td>64 - 256 bytes</td>
<td>~80ns - 200 ns</td>
</tr>
<tr>
<td>~ $1000 / Gbyte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disk</td>
<td>10x T bytes, 10 ms</td>
<td>~ $1 / Gbyte</td>
</tr>
<tr>
<td>~ $10,000,000 ns</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tape</td>
<td>infinite, sec-min</td>
<td>~ $1 / Gbyte</td>
</tr>
<tr>
<td>~ 4K - 8K bytes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Upper Level**

- **faster**
- **Larger**

**Lower Level**

- **Standing Xfer Unit**
3) Focus on the Common Case

° Common sense guides computer design
  - Since we're engineering, common sense is valuable
  - In making a design trade-off, favor the frequent case over the
  infrequent case
  - E.g., Instruction fetch and decode unit used more frequently than
    multiplier, so optimize it 1st
  - E.g., If database server has 50 disks / processor, storage
    dependability dominates system dependability, so optimize it 1st
  - Frequent case is often simpler and can be done faster than the
    infrequent case
  - E.g., overflow is rare when adding 2 numbers, so improve performance
    by optimizing more common case of no overflow
  - May slow down overflow, but overall performance improved by
    optimizing for the normal case
  - What is frequent case and how much performance improved by
    making case faster => Amdahl's Law

4) Amdahl's Law

\[
\text{Speedup}_{\text{enhanced}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}})} \times \text{Fraction}_{\text{enhanced}} \text{Speedup}_{\text{enhanced}}
\]

\[
\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \text{Fraction}_{\text{enhanced}} \text{Speedup}_{\text{enhanced}}}
\]

Best you could ever hope to do:

\[
\text{Speedup}_{\text{maximum}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}})}
\]

Amdahl's Law example

° New CPU 10X faster
° I/O bound server, so 60% time waiting for I/O

\[
\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}})} \times \frac{\text{Fraction}_{\text{enhanced}} \text{Speedup}_{\text{enhanced}}}{10} = \frac{1}{0.4} \times \frac{0.4}{0.64} = 1.56
\]

° Apparently, it's human nature to be attracted
  by 10X faster, vs. keeping in perspective it's
  just 1.6X faster

5) Processor performance equation

"Iron Law of Performance"

\[
\text{CPU time} = \frac{\text{Seconds}_{\text{Program}}}{\text{Instructions}_{\text{Program}}} \times \frac{\text{Cycles}_{\text{Instruction}}}{\text{Seconds}_{\text{Cycle}}}
\]

- 指令总数  | CPI | 时钟周期
- 程序  | X |
- 编译器  | X | (X)
- 指令密集体组织结构  | X | X
- 框架  | X | X
- 工艺技术  | X |
Metrics used to Compare Designs

- Cost
  - Die cost and system cost
- Execution Time
  - Average and worst-case
  - Latency vs. Throughput
- Energy and Power
  - Also peak power and peak switching current
- Reliability
  - Resiliency to electrical noise, part failure
  - Robustness to bad software, operator error
- Maintainability
  - System administration costs
- Compatibility
  - Software costs dominate

Cost of Processor

- Design cost (Non-recurring Engineering Costs, NRE)
  - Dominated by engineer-years
  - Also mask costs (exceeding $1M per spin)
- Cost of die
  - Die area
  - Die yield (maturity of manufacturing process, redundancy features)
  - Cost/size of wafers
  - Die cost ~ f(die area^4) with no redundancy
- Cost of packaging
  - Number of pins (signal + power/ground pins)
  - Power dissipation
- Cost of testing
  - Built-in test features?
  - Logical complexity of design
  - Choice of circuits (minimum clock rates, leakage currents, I/O drivers)

Architect affects all of these

System-Level Cost Impacts

- Power supply and cooling
- Support chipset
- Off-chip SRAM/DRAM/ROM
- Off-chip peripherals

What is Performance?

- Latency (or response time or execution time)
  - Time to complete one task
- Bandwidth (or throughput)
  - Tasks completed per unit time
Definition: Performance

° Performance is in units of things per sec
  • bigger is better
° If we are primarily concerned with response time

\[
\text{Performance}(X) = \frac{1}{\text{ExTime}(X)}
\]

"X is n times faster than Y" means

\[
n = \frac{\text{ExTime}(Y)}{\text{ExTime}(X)} = \frac{\text{Performance}(X)}{\text{Performance}(Y)}
\]

Types of Benchmark

° Synthetic Benchmarks
  • Designed to have same mix of operations as real workloads, e.g., Dhrystone, Whetstone
° Toy Programs
  • Small, easy to port. Output often known before program is run, e.g., Nqueens, Bubblesort, Towers of Hanoi
° Kernels
  • Common subroutines in real programs, e.g., matrix multiply, FFT, sorting, Livermore Loops, Linpack
° Simplified Applications
  • Extract main computational skeleton of real application to simplify porting, e.g., NAS parallel benchmarks, TPC
° Real Applications
  • Things people actually use their computers for, e.g., car crash simulations, relational databases, Photoshop, Quake

Performance: What to measure

° Usually rely on benchmarks vs. real workloads
° To increase predictability, collections of benchmark applications-- benchmark suites -- are popular
  ° SPECCPU: popular desktop benchmark suite
    • CPU only, split between integer and floating point programs
    • SPECint2000 has 12 integer, SPECfp2000 has 14 integer pgms
    • SPECCPU2006 to be announced Spring 2006
    • SPECSFS (NFS file server) and SPECweb (WebServer) added as server benchmarks
  ° Transaction Processing Council measures server performance and cost-performance for databases
    • TPC-C Complex query for Online Transaction Processing
    • TPC-H models ad hoc decision support
    • TPC-W a transactional web benchmark
    • TPC-App application server and web services benchmark

Simplifying Performance

<table>
<thead>
<tr>
<th>System</th>
<th>Rate (Task 1)</th>
<th>Rate (Task 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>B</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>

Which system is faster?
... depends who’s selling

<table>
<thead>
<tr>
<th>System</th>
<th>Rate (Task 1)</th>
<th>Rate (Task 2)</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>B</td>
<td>20</td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>

Average throughput

<table>
<thead>
<tr>
<th>System</th>
<th>Rate (Task 1)</th>
<th>Rate (Task 2)</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.50</td>
<td>2.00</td>
<td>1.25</td>
</tr>
<tr>
<td>B</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Throughput relative to B

<table>
<thead>
<tr>
<th>System</th>
<th>Rate (Task 1)</th>
<th>Rate (Task 2)</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>B</td>
<td>2.00</td>
<td>0.50</td>
<td>1.25</td>
</tr>
</tbody>
</table>

Throughput relative to A

Summarizing Performance over Set of Benchmark Programs

Arithmetic mean of execution times $t_i$ (in seconds)

$$\frac{1}{n} \sum_i t_i$$

Harmonic mean of execution rates $r_i$ (MIPS/MFLOPS)

$$\frac{n}{\sum_i \left( \frac{1}{r_i} \right)}$$

- Both equivalent to workload where each program is run the same number of times
- Can add weighting factors to model other workload distributions

Normalized Execution Time and Geometric Mean

- Measure speedup relative to reference machine
  
  \[ \text{ratio} = \frac{t_{\text{Ref}}}{t_A} \]

- Average time ratios using geometric mean
  
  \[ n\sqrt[n]{\prod_i \text{ratio}_i} \]

- Insensitive to machine chosen as reference
- Insensitive to run time of individual benchmarks
- Used by SPEC89, SPEC92, SPEC95, ..., SPEC2006

SPEC: System Performance Evaluation Cooperative

- 第一版 1989
  - 10个程序(6Fp+4Int)产生单一数值(SPECmarks)
- 第二版 1992
  - SPECInt92 (6Int) 和 SPECfp92 (14Fp)
    - 不限制编译器的开关。DEC 4000 Model 610在1993年3月:
      - spice: unix.c/def=(sysv,has_bcopy,memcpy(a,b,c)=memcpy(b,a,c)
      - wave5: /all=(all,dcom=nat)/ag=a/ur=4/ur=200
      - nasa7: /moreno/ag=a/ur=4/ur=200/loc=blas
- 第三版 1995
  - 一组新的程序: SPECInt95 (8Int) 和 SPECfp95 (10Fp)
  - 使用期 三年
  - 对所有程序使用同一开关设置: SPECInt_base95, SPECfp_base95
第一版SPEC

- 1989年，第一版：10个程序，用单一数值来总结性能（6Fp+4Int），相对于VAX 11/780
- 其中有一个程序：99%的时间耗费在该程序的单一行代码上
- 新型前端编译器可以非常显著地改进它的性能

**Benchmark Descriptions**

- go 人工智选：玩围棋游戏
- m88ksim 摩托罗拉88k芯片仿真器：运行测试程序
- gcc GNU C编译器生成SPARC代码
- compress 压缩和解压缩文件
- li Lisp解释器
- eqntott 将线性方程组转化为矩阵形式
- matrix300 使用300x300矩阵进行计算
- fpppp 量子化学
- wave5 等离子体物理：电磁粒子模拟

**SPEC95**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Type</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>Integer C</td>
<td>Compression using the Lempel-Ziv algorithm</td>
<td></td>
</tr>
<tr>
<td>vpr</td>
<td>Integer C</td>
<td>FPGA circuit placement and routing</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>Integer C</td>
<td>Consists of the GNU C compiler generating optimized machine code</td>
<td></td>
</tr>
<tr>
<td>code</td>
<td>Integer C</td>
<td>Combinatorial optimization of public transit scheduling</td>
<td></td>
</tr>
<tr>
<td>mcf</td>
<td>Integer C</td>
<td>Chess-playing program</td>
<td></td>
</tr>
<tr>
<td>parser</td>
<td>Integer C</td>
<td>Syntactic English language parser</td>
<td></td>
</tr>
<tr>
<td>perl</td>
<td>Integer C</td>
<td>Perl (an interpreted string-processing language) with four input scripts</td>
<td></td>
</tr>
<tr>
<td>gap</td>
<td>Integer C</td>
<td>A group theory application package</td>
<td></td>
</tr>
<tr>
<td>vortex</td>
<td>Integer C</td>
<td>An object-oriented database system</td>
<td></td>
</tr>
<tr>
<td>bzip2</td>
<td>Integer C</td>
<td>A block-sorting compression algorithm</td>
<td></td>
</tr>
<tr>
<td>bzip2</td>
<td>Integer C</td>
<td>A simulated annealing algorithm for VLSI place and route</td>
<td></td>
</tr>
<tr>
<td>wupwise</td>
<td>FP</td>
<td>F77</td>
<td>Lattice gauge theory model of quantum chromodynamics</td>
</tr>
<tr>
<td>swim</td>
<td>FP</td>
<td>F77</td>
<td>Solves shallow water equations using finite difference equations</td>
</tr>
<tr>
<td>mgrid</td>
<td>FP</td>
<td>F77</td>
<td>Multigrid solver over three-dimensional field</td>
</tr>
<tr>
<td>apply</td>
<td>FP</td>
<td>F77</td>
<td>Parabolic and elliptic partial differential equation solver</td>
</tr>
<tr>
<td>mesa</td>
<td>FP</td>
<td>C</td>
<td>Three-dimensional graphics library</td>
</tr>
<tr>
<td>galgel</td>
<td>FP</td>
<td>F90</td>
<td>Computational fluid dynamics</td>
</tr>
<tr>
<td>art</td>
<td>FP</td>
<td>C</td>
<td>Image recognition of a thermal image using neural networks</td>
</tr>
<tr>
<td>image</td>
<td>FP</td>
<td>C</td>
<td>Simulation of seismic wave propagation</td>
</tr>
<tr>
<td>facerec</td>
<td>FP</td>
<td>C</td>
<td>Face recognition using wavelets and graph matching</td>
</tr>
<tr>
<td>ammp</td>
<td>FP</td>
<td>C</td>
<td>Molecular dynamics simulation of a protein in water</td>
</tr>
<tr>
<td>lucas</td>
<td>FP</td>
<td>F90</td>
<td>Performs primality testing for Mersenne primes</td>
</tr>
<tr>
<td>mgrid</td>
<td>FP</td>
<td>F90</td>
<td>Finite element modeling of crash simulation</td>
</tr>
</tbody>
</table>

**EEMBC**

<table>
<thead>
<tr>
<th>Benchmark type</th>
<th>Number of kernels</th>
<th>Example benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automotive/Industrial</td>
<td>16</td>
<td>6 microbenchmarks (arithmetic operations, pointer chasing, memory performance, matrix arithmetic, table lookup, bit manipulation), 5 automobile control benchmarks, and 5 filter or signal processing benchmarks</td>
</tr>
<tr>
<td>Consumer</td>
<td>5</td>
<td>JPEG compress/decompress, filtering, and RGB Conversions</td>
</tr>
<tr>
<td>Networking</td>
<td>3</td>
<td>Shortest-path calculation, IP routing, and packet flow operations</td>
</tr>
<tr>
<td>Office automation</td>
<td>4</td>
<td>Graphics and text benchmarks (Bézier curve calculation, dithering, image rotation, text processing)</td>
</tr>
<tr>
<td>Telecommunications</td>
<td>6</td>
<td>Filtering and DSP benchmarks (autocorrelation, FFT, decoder, encoder)</td>
</tr>
</tbody>
</table>
如何总结性能

- 算术平均值（或者加权算术平均值）追踪 执行时间：
  \[ \text{SUM}(T_i)/n \text{ 或者 SUM}(W_i \times T_i) \]
- 比率（例如MFLOPS） 的 调和平均值（或者加权调和平均值）
  执行时间：
  \[ n/\text{SUM}(1/R_i) \text{ 或者 } n/\text{SUM}(W_i/R_i) \]
- 为了按比例缩放性能，规格化执行时间 是 非常便捷的！
  例如， 参照机器的时间 ÷ 被评测机器的时间
- 注意，不可使用规格化的执行时间的算术平均值，而应该使用几何平均值！
- 几何平均值平等对待所有的改进情况：

<table>
<thead>
<tr>
<th></th>
<th>A 程序的执行</th>
<th>B 程序的执行</th>
<th>Normalized to A</th>
<th>Normalized to B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time 1</td>
<td>2 秒</td>
<td>2000 秒</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Time 2</td>
<td>1 秒</td>
<td>1000 秒</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Arithmetic mean</td>
<td>500.5</td>
<td>500.5</td>
<td>5.05</td>
<td>5.05</td>
</tr>
<tr>
<td>Geometric mean</td>
<td>31.6</td>
<td>31.6</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

为什么对规格化数值要进行几何平均？

- Normalized to A
- Normalized to B

性能评测

- or better or worse, benchmarks shape a field
- Good products created when have:
  - Good benchmarks
  - Good ways to summarize performance
- Given sales is a function in part of performance relative to competition, investment in improving product as reported by performance summary
- If benchmarks/summary inadequate, then choose between improving product for real programs vs. improving product to get more sales;
  Sales almost always wins!
- Execution time is the measure of computer performance!

How to Mislead with Performance Reports

- Select pieces of workload that work well on your design, ignore others
- Use unrealistic data set sizes for application (too big or too small)
- Report throughput numbers for a latency benchmark
- Report latency numbers for a throughput benchmark
- Report performance on a kernel and claim it represents an entire application
- Use 10-bit fixed-point arithmetic (because it’s fastest on your system)
- Even though application requires 64-bit floating-point arithmetic
- Use a less efficient algorithm on the competing machine
- Report speedup for an inefficient algorithm (bubblesort)
- Compare hand-optimized assembly code with unoptimized C code
- Compare your design using next year’s technology against competitor’s year old design (1% performance improvement per week)
- Ignore the relative cost of the systems being compared
- Report averages and not individual results
- Report speedup over unspecified base system, not absolute times
- Report efficiency not absolute times
- Report MFLOPS not absolute times (use inefficient algorithm)

[David Bailey “Twelve ways to fool the masses when giving performance results for parallel supercomputers”]
Power and Energy

- Energy to complete operation (Joules)
  - Corresponds approximately to battery life
  - (Battery energy capacity actually depends on rate of discharge)
- Peak power dissipation (Watts = Joules/second)
  - Affects packaging (power and ground pins, thermal design)
- $\frac{di}{dt}$, peak change in supply current (Amps/second)
  - Affects power supply noise (power and ground pins, decoupling capacitors)

A "Typical" RISC ISA

- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (RO contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store: base + displacement
  - no indirection
- Simple branch conditions
- Delayed branch

see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

Peak Power versus Lower Energy

- System A has higher peak power, but lower total energy
- System B has lower peak power, but higher total energy

Example: MIPS (-MIPS)

<table>
<thead>
<tr>
<th>Register-Register</th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register-Immediate</th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rd</td>
<td></td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Branch</th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2/Opx</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Jump / Call</th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td></td>
<td></td>
<td></td>
<td>target</td>
<td></td>
</tr>
</tbody>
</table>
Datapath vs Control

Datapath: Storage, FU, interconnect sufficient to perform the desired functions
- Inputs are Control Points
- Outputs are signals
Controller: State machine to orchestrate operation on the data path
- Based on desired function and signals

Approaching an ISA

- Instruction Set Architecture
  - Defines set of operations, instruction format, hardware supported data types, named storage, addressing modes, sequencing
  - Meaning of each instruction is described by RTL on architected registers and memory
  - Given technology constraints assemble adequate datapath
    - Architected storage mapped to actual storage
    - Function units to do all the required operations
    - Possible additional storage (eg. MAR, MBR, …)
    - Interconnect to move information among reg and FUs
- Map each instruction to sequence of RTLs
- Collate sequences into symbolic controller state transition diagram (STD)
- Lower symbolic STD to control points
- Implement controller

5 Steps of MIPS Datapath
Figure A.2, Page A-8

Instruction Fetch
- Next PC = Next SEQ PC
- IR <= mem[PC];
- PC <= PC + 4
- Reg[IRrd] <= Reg[IRrs] or Reg[IRrt]

Instr. Decode
- Reg. Fetch
- Addr. Calc
- Memory Access
- Write Back

Execute
- Address Calc
- ALU
- MUX
- Memory
- Reg File
- MUX
- Data
- Memory
- MUX
- Sign
- Extend

IR <= mem[PC];
PC <= PC + 4
A <= Reg[IRrs];
B <= Reg[IRrt];
rslt <= A opIRop B
Reg[IRrd] <= WB
WB <= rslt

Memory Access
- Write Back
- WB Data
- Next PC
- Next SEQ PC
Inst. Set Processor Controller

IR <= mem[PC];
PC <= PC + 4

A <= Reg[IRrs];
B <= Reg[IRrt]

caseIRop:
        opIFetch-DCD
        if bop(A,b)
        PC <= PC+IRim
        br jmp

r <= A opIRop IRim
Reg[IRrd] <= WB
WB <= r

5 Steps of MIPS Datapath

Visualization Pipelining

Pipelining is not quite that easy!

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support this combination of instructions (single person to fold and put clothes away)
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock)
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).
One Memory Port/Structural Hazards

Figure A.4, Page A-14

<table>
<thead>
<tr>
<th>Time (clock cycles)</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Speed Up Equation for Pipelining

\[
CPI_{\text{pipelined}} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}
\]

\[
\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth} \times \text{Cycle Time}_{\text{unpipeline}}}{\text{Ideal CPI} \times \text{Pipeline stall CPI} \times \text{Cycle Time}_{\text{pipelined}}}
\]

For simple RISC pipeline, CPI = 1:

\[
\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipeline}}}{\text{Cycle Time}_{\text{pipelined}}}
\]

Example: Dual-port vs. Single-port

° Machine A: Dual ported memory ("Harvard Architecture")
° Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
° Ideal CPI = 1 for both
° Loads are 40% of instructions executed

\[
\text{SpeedUp}_A = \frac{\text{Pipeline Depth}}{1 + 0} \times \frac{\text{clock}_{\text{unpipe}}}{\text{clock}_{\text{pipe}}}
\]

\[
\text{SpeedUp}_B = \frac{\text{Pipeline Depth}}{1 + 0.4 \times 1} \times \frac{\text{clock}_{\text{unpipe}}}{\text{clock}_{\text{unpipe}} / 1.05}
\]

\[
= \frac{\text{Pipeline Depth}}{1.4} \times 1.05
\]

\[
= 0.75 \times \text{Pipeline Depth}
\]

\[
\text{SpeedUp}_A / \text{SpeedUp}_B = \frac{\text{Pipeline Depth}}{0.75 \times \text{Pipeline Depth}} = 1.33
\]

° Machine A is 1.33 times faster
Three Generic Data Hazards

- **Read After Write (RAW)**
  Instr$_j$ tries to read operand before Instr$_i$ writes it
  
  \[
  \text{I: add r1, r2, r3} \\
  \text{J: sub r4, r1, r3}
  \]

- Caused by a “Dependence” (in compiler nomenclature). This hazard results from an actual need for communication.

Three Generic Data Hazards

- **Write After Read (WAR)**
  Instr$_j$ writes operand before Instr$_i$ reads it
  
  \[
  \text{I: sub r4, r1, r3} \\
  \text{J: add r1, r2, r3} \\
  \text{K: mul r6, r1, r7}
  \]

- Called an “anti-dependence” by compiler writers. This results from reuse of the name “r1”.

- Can’t happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5

Three Generic Data Hazards

- **Write After Write (WAW)**
  Instr$_j$ writes operand before Instr$_i$ writes it.
  
  \[
  \text{I: sub r4, r1, r3} \\
  \text{J: add r1, r2, r3} \\
  \text{K: mul r6, r1, r7}
  \]

- Called an “output dependence” by compiler writers.
  This also results from the reuse of name “r1”.

- Can’t happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5

- Will see WAR and WAW in more complicated pipes.
Forwarding to Avoid Data Hazard

**Figure A.7, Page A-19**

Instr. Order

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Time (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r1, r2, r3</td>
<td></td>
</tr>
<tr>
<td>sub r4, r1, r3</td>
<td></td>
</tr>
<tr>
<td>and r6, r1, r7</td>
<td></td>
</tr>
<tr>
<td>or r8, r1, r9</td>
<td></td>
</tr>
<tr>
<td>xor r10, r1, r11</td>
<td></td>
</tr>
</tbody>
</table>

HW Change for Forwarding

**Figure A.23, Page A-37**

What circuit detects and resolves this hazard?

Data Hazard Even with Forwarding

**Figure A.9, Page A-21**

Instr. Order

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Time (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw r1, 0(r2)</td>
<td></td>
</tr>
<tr>
<td>sub r4, r1, r6</td>
<td></td>
</tr>
<tr>
<td>and r6, r1, r7</td>
<td></td>
</tr>
<tr>
<td>or r8, r1, r9</td>
<td></td>
</tr>
</tbody>
</table>

Forwarding to Avoid LW-SW Data Hazard

**Figure A.8, Page A-20**

Instr. Order

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Time (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r1, r2, r3</td>
<td></td>
</tr>
<tr>
<td>lw r4, 0(r1)</td>
<td></td>
</tr>
<tr>
<td>sw r4, 12(r1)</td>
<td></td>
</tr>
<tr>
<td>or r8, r6, r9</td>
<td></td>
</tr>
<tr>
<td>xor r10, r9, r11</td>
<td></td>
</tr>
</tbody>
</table>
Data Hazard Even with Forwarding
(Similar to Figure A.10, Page A-21)

Time (clock cycles)

 lw r1, 0(r2)

 sub r4,r1,r6

 and r6,r1,r7

 or r8,r1,r9

How is this detected?

Software Scheduling to Avoid Load Hazards

Try producing fast code for
\[
a = b + c; \\
d = e - f;
\]
assuming a, b, c, d, e, and f in memory.

Slow code:
\[
\text{LW } Rb,b \text{ LW } Rc,c \text{ ADD } Ra,Rb,Rc \text{ LW } Re,e \text{ LW } Rf,f \text{ SUB } a,Ra \text{ ADD } Ra,Rb, Rc \text{ LW } Re,e \text{ SW } a,Ra \text{ SW } d,Rd \text{ SW } a,Ra \text{ SW } d,Rd
\]

Fast code:
\[
\text{LW } Rb,b \text{ LW } Rc,c \text{ ADD } Ra,Rb,Rc \text{ LW } Re,e \text{ LW } Rf,f \text{ SUB } a,Ra \text{ ADD } Ra,Rb, Rc \text{ LW } Re,e \text{ SW } a,Ra \text{ SW } d,Rd \text{ SW } a,Ra \text{ SW } d,Rd
\]

Compiler optimizes for performance. Hardware checks for safety.

Control Hazard on Branches
Three Stage Stall

10: beq r1,r3,36

14: and r2,r3,r5

18: or r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11

What do you do with the 3 instructions in between?
How do you do it?
Where is the "commit"?

Branch Stall Impact

- If CPI = 1, 30% branch, Stall 3 cycles => new CPI = 1.9!
- Two part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- MIPS branch tests if register = 0 or ≠ 0
- MIPS Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3
### Four Branch Hazard Alternatives

**#1: Stall until branch direction is clear**
- Execute successor instructions in sequence
- "Squash" instructions in pipeline if branch actually taken
- Advantage of late pipeline state update
- 47% MIPS branches not taken on average
- PC+4 already calculated, so use it to get next instruction

**#2: Predict Branch Not Taken**
- Execute successor instructions in sequence
- "Squash" instructions in pipeline if branch actually taken
- Advantage of late pipeline state update
- 53% MIPS branches taken on average
- But haven't calculated branch target address in MIPS
- MIPS still incurs 1 cycle branch penalty
- Other machines: branch target known before outcome

**#3: Predict Branch Taken**
- 53% MIPS branches taken on average
- PC+4 already calculated, so use it to get next instruction
- Other machines: branch target known before outcome

**#4: Delayed Branch**
- Define branch to take place AFTER a following instruction
- Branch delay of length $n$
- 1 slot delay allows proper decision and branch target address
- MIPS uses this

---

**Scheduling Branch Delay Slots (Fig A.14)**

**A. From before branch**

```
add $1,$2,$3
if $2=0$ then
delay slot
```

**B. From branch target**

```
sub $4,$5,$6
if $1=0$ then
delay slot
```

**C. From fall through**

```
add $1,$2,$3
if $1=0$ then
delay slot
sub $4,$5,$6
```

- A is the best choice, fills delay slot & reduces instruction count (IC)
- In B, the sub instruction may need to be copied, increasing IC
- In B and C, must be okay to execute sub when branch fails
Delayed Branch

° Compiler effectiveness for single branch delay slot:
   • Fills about 60% of branch delay slots
   • About 80% of instructions executed in branch delay slots useful in computation
   • About 50% (60% x 80%) of slots usefully filled

° Delayed Branch downside: As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot
   • Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
   • Growth in available transistors has made dynamic approaches relatively cheaper

Evaluating Branch Alternatives

\[
\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}
\]

Assume 4% unconditional branch, 6% conditional branch-untaken, 10% conditional branch-taken

<table>
<thead>
<tr>
<th>Scheduling scheme</th>
<th>Branch CPI</th>
<th>Speedup v. unpipelined</th>
<th>Speedup v. stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall pipeline</td>
<td>3</td>
<td>1.60</td>
<td>3.1</td>
</tr>
<tr>
<td>Predict taken</td>
<td>1</td>
<td>1.20</td>
<td>4.2</td>
</tr>
<tr>
<td>Predict not taken</td>
<td>1</td>
<td>1.14</td>
<td>4.4</td>
</tr>
<tr>
<td>Delayed branch</td>
<td>0.5</td>
<td>1.10</td>
<td>4.5</td>
</tr>
</tbody>
</table>

Problems with Pipelining

° Exception: An unusual event happens to an instruction during its execution
   • Examples: divide by zero, undefined opcode

° Interrupt: Hardware signal to switch the processor to a new instruction stream
   • Example: a sound card interrupts when it needs more audio output samples (an audio “click” happens if it is left waiting)

° Problem: It must appear that the exception or interrupt must appear between 2 instructions (I_i and I_{i+1})
   • The effect of all instructions up to and including I_i is totalling complete
   • No effect of any instruction after I_i can take place
   • The interrupt (exception) handler either aborts program or restarts at instruction I_{i+1}

Precise Exceptions In-Order Pipelines

Key observation: architected state only change in memory and register write stages.
Summary: Metrics and Pipelining

- Machines compared over many metrics
  - Cost, performance, power, reliability, compatibility, ...
  - Difficult to compare widely differing machines on benchmark suite
  - Control VIA State Machines and Microprogramming
  - Just overlap tasks; easy if tasks are independent
  - Speed Up ≤ Pipeline Depth; if ideal CPI is 1, then:
    \[
    \text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}}{\text{Cycle Time}_\text{unpipelined}}
    \]
  - Hazards limit performance on computers:
    - Structural: need more HW resources
    - Data (RAW, WAR, WAW): need forwarding, compiler scheduling
    - Control: delayed branch, prediction
    - Exceptions, Interrupts add complexity

Since 1980, CPU has outpaced DRAM ...

Four-issue 2GHz superscalar accessing 100ns DRAM could execute 800 instructions during time for one memory access!

Addressing the Processor-Memory Performance GAP

- **Goal**: Illusion of large, fast, cheap memory.
  Let programs address a memory space that scales to the disk size, at a speed that is usually as fast as register access

- **Solution**: Put smaller, faster “cache” memories between CPU and DRAM. Create a “memory hierarchy”.

Levels of the Memory Hierarchy

```
<table>
<thead>
<tr>
<th>Upper Level</th>
<th>Lower Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>faster</td>
<td>Larger</td>
</tr>
</tbody>
</table>

Today's Focus

- Registers
- Instr. Operands
- Cache
- Blocks
- Memory
- Pages
- Disk
- Files
- Tape
- User/operator

CPU Registers
- Registers
- Cache
- Main Memory
- File
- Tape

Capacity: Access Time: Cost

- CPU Registers: 100 bytes, 10-100ns, 1-0.1 cents/bit
- Cache: 10-1000 bytes, 10-100ns, 0.001-10 cents/bit
- Main Memory: M bytes, 20-500ns, 0.0001-0.00001 cents/bit
- Disk: G bytes, 20-5000ms, .001-10 cents/bit
- Tape: infinite sec/min, 0.001-10 cents/bit
```
Common Predictable Patterns

Two predictable properties of memory references:

° **Temporal Locality:** If a location is referenced, it is likely to be referenced again in the near future (e.g., loops, reuse).

° **Spatial Locality:** If a location is referenced it is likely that locations near it will be referenced in the near future (e.g., straightline code, array access).

Caches

Caches exploit both types of predictability:

- Exploit **temporal locality** by remembering the contents of recently accessed locations.
- Exploit **spatial locality** by fetching blocks of data around recently accessed locations.

Cache Algorithm (Read)

Look at Processor Address, search cache tags to find match. Then either:

- **HIT - Found in Cache**
  - Return copy of data from cache
- **MISS - Not in cache**
  - Read block of data from Main Memory
  - Wait ...
  - Return data to processor and update cache

**Hit Rate** = fraction of accesses found in cache
**Miss Rate** = 1 – Hit rate
**Hit Time** = RAM access time + time to determine HIT/MISS
**Miss Time** = time to replace block in cache + time to deliver block to processor

Inside a Cache

- Copy of main memory location 100
- Copy of main memory location 101
- Line
- Data Block
- Address Tag
4 Questions for Memory Hierarchy

- Q1: Where can a block be placed in the cache? (Block placement)
- Q2: How is a block found if it is in the cache? (Block identification)
- Q3: Which block should be replaced on a miss? (Block replacement)
- Q4: What happens on a write? (Write strategy)

Q2: How is a block found?

- Index selects which set to look in
- Tag on each block
  - No need to check index or block offset
  - Increasing associativity shrinks index, expands tag. Fully Associative caches have no index field.

Direct-Mapped Cache

<table>
<thead>
<tr>
<th>Block Address</th>
<th>Block Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Index</td>
</tr>
</tbody>
</table>
What causes a MISS?

- Three Major Categories of Cache Misses:
  - **Compulsory Misses**: first access to a block
  - **Capacity Misses**: cache cannot contain all blocks needed to execute the program
  - **Conflict Misses**: block replaced by another block and then later retrieved - (affects set assoc. or direct mapped caches)

  Nightmare Scenario: ping pong effect!

Block Size and Spatial Locality

Block is unit of transfer between the cache and memory

<table>
<thead>
<tr>
<th>Tag</th>
<th>Word0</th>
<th>Word1</th>
<th>Word2</th>
<th>Word3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4 word block, b=2</td>
</tr>
</tbody>
</table>

Split CPU address

- **block address**
- **offset**

<table>
<thead>
<tr>
<th>32-b bits</th>
<th>b bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^b = block size</td>
<td>in bytes</td>
</tr>
</tbody>
</table>

Larger block size has distinct hardware advantages
- less tag overhead
- exploit fast burst transfers from DRAM
- exploit fast burst transfers over wide busses

What are the disadvantages of increasing block size?
Fewer blocks => more conflicts. Can waste bandwidth.
Q3: Which block should be replaced on a miss?

- Easy for Direct Mapped
- Set Associative or Fully Associative:
  - Random
  - Least Recently Used (LRU)
    - LRU cache state must be updated on every access
    - true implementation only feasible for small sets (2-way)
    - pseudo-LRU binary tree often used for 4-8 way
  - First In, First Out (FIFO) a.k.a. Round-Robin
    - used in highly associative caches
- Replacement policy has a second order effect since replacement only happens on misses

Q4: What happens on a write?

- Cache hit:
  - write through: write both cache & memory
    - generally higher traffic but simplifies cache coherence
  - write back: write cache only
    - (memory is written only when the entry is evicted)
    - a dirty bit per block can further reduce the traffic
- Cache miss:
  - no write allocate: only write to main memory
  - write allocate (aka fetch on write): fetch into cache

- Common combinations:
  - write through and no write allocate
  - write back with write allocate

5 Basic Cache Optimizations

- Reducing Miss Rate
  1. Larger Block size (compulsory misses)
  2. Larger Cache size (capacity misses)
  3. Higher Associativity (conflict misses)

- Reducing Miss Penalty
  4. Multilevel Caches

- Reducing hit time
- Giving Reads Priority over Writes
  - E.g., Read complete before earlier writes in write buffer