Spring 2010

Instructor Information
• Instructor: Jon Kuhl (That's me)
  – Office: 4322 SC
  – Office Hours: 9:00-10:30 a.m. TTh (Other times by appointment)
  – E-mail: kuhl@engineering.uiowa.edu
  – Phone: (319) 335-5958
• TA: Artistee Harris
  – Office: t.b.d
  – Office hours: t.b.d.

Class Info.
• Website: www.engineering.uiowa.edu/~hpca
• Texts:

Course Objectives
• Understand quantitative measures for assessing and comparing processor performance
• Understand modern processor design techniques, including:
  – Pipelining
  – High performance memory architecture
  – Instruction-level parallelism
  – Multi-threading
  – Multi-core architecture
• Master the use of modern design tools (HDLs) to design and analyze processors
• Do case studies of contemporary processors
• Discuss future trends in processor design
Expected Background

- A previous course in computer architecture/organization covering:
  - Instruction set architecture (ISA)
  - Addressing modes
  - Assembly language
  - Basic computer organization
  - Memory system organization
    - Cache
    - virtual
  - Etc.
- 22C:060 or 55:035 or equivalent

Course Organization

- Homework assignments--several
- Two projects (design/analysis exercises using the Verilog HDL and ModelSim simulation environment)
- Two exams:
  - Midterm—Th. March 11, in class
  - Final—Tues. May 11, 9:45-11:45 a.m.

Course Organization--continued

- Grading:
  - Exams:
    - Better of midterm/final exam score: 35%
    - Poorer of midterm/final exam scores: 25%
  - Homework: 10%
  - Projects 30%

Historical Perspectives

- The Decade of the 1970’s: "Birth of Microprocessors"
  - Programmable Controller
  - Single-Chip Microprocessors
  - Personal Computers (PC)
- The Decade of the 1980’s: "Quantitative Architecture"
  - Instruction Pipelining
  - Fast Cache Memories
  - Compiler Considerations
  - Workstations
- The Decade of the 1990’s: "Instruction-Level Parallelism"
  - Superscalar, Speculative Microarchitectures
  - Aggressive Compiler Optimizations
  - Low-Cost Desktop Supercomputing
Moore’s Law (1965)

- The number of devices that can be integrated on a single piece of silicon will double roughly every 18-24 months

- Moore’s law has held true for 40 years and will continue to hold for at least another decade, probably longer.

The Computer Architect’s Challenge

- Make Moore’s Law apply also to computer chip performance as well as density

- That is, make sure that the additional chip density (complexity) is utilized efficiently.
  - Note that Moore’s law has roughly held for both chip density and clock frequency—chips have been getting faster as well as denser.
  - So fully exploiting the increase in density and clock speed should lead to performance increases well exceeding the growth rate of Moore’s Law.
  - This hasn’t been the case.
  - Have Computer Architects failed????
Moore’s Law—Processor Performance

Performance Growth in Perspective

- Doubling every 24 months (1971-2009):
  - total of 524,000X
  - Cars travel at 40 million MPH; get 10^6 MPG.
  - Air travel: L.A. to N.Y. in 0.04 seconds
  - Corn yield: 100 million bushels per acre

A Quote from Robert Cringely

“If the automobile had followed the same development as the computer, a Rolls-Royce would today cost $100, get a million miles per gallon, and explode once a year killing everyone inside.”

Moore’s Law—Processor Power Consumption
Relationship between clock rate and power

- Intel Estimate\(^1\):
  - Increasing clock rate by 25% will yield approx. 15% performance increase
  - But power consumption will be \textit{doubled}
- Power Consumption/Heat Dissipation issues are ushering a new era in CPU design
  - Focus on performance per watt
  - Causing fundamental rethinking of architecture

\(^1\) Phillip E. Ross, “Why Clock Frequency Stalled”, \textit{IEEE Spectrum}, April, 2008

Processor Design

**ARCHITECTURE** (ISA) \textit{programmer/compiler view}:
- Functional appearance (interface) to user/system programmer
- Opcodes, addressing modes, architectured registers, IEEE floating point
- Serves as specification for processor design

**IMPLEMENTATION** (μarchitecture) \textit{processor designer view}:
- Logical structure or organization that performs the architecture
- Pipelining, functional units, caches, physical registers

**REALIZATION** (Chip) \textit{chip/system designer view}:
- Physical structure that embodies the implementation
- Gates, cells, transistors, wires
Iron Law

\[
\text{Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}
\]

Architecture --> Implementation --> Realization
Compiler Designer  Processor Designer  Chip Designer

Iron Law—Considering Power

\[
\text{Time} \times \text{Watts} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}} \times \text{Watts}
\]

Architecture --> Implementation --> Realization
Compiler Designer  Processor Designer  Chip Designer

Iron Law

- Instructions/Program
  - Instructions executed, not static code size
  - Determined by algorithm, compiler, ISA
- Cycles/Instruction
  - Determined by ISA and CPU organization
  - Overlap among instructions reduces this term
- Time/cycle
  - Determined by technology, organization, clever circuit design

Overall Goal

- Minimize time, which is the product, NOT isolated terms
- Common error to miss terms while devising optimizations
  - E.g. ISA change to decrease instruction count
  - BUT leads to CPU organization which makes clock slower
- Bottom line: terms are inter-related
- This is the crux of the RISC vs. CISC argument
**Iron Law Example**

- Machine A: clock 1ns, CPI 2.0, for program P
- Machine B: clock 2ns, CPI 1.2, for program P
- Which is faster and how much?

\[
\text{Time/Program} = \frac{\text{instr/program} \times \text{cycles/instr} \times \text{sec/cycle}}{\text{cycles/instr} \times \text{sec/cycle}}
\]

- Time(A) = \(N \times 2.0 \times 1 = 2N\)
- Time(B) = \(N \times 1.2 \times 2 = 2.4N\)

Compare: \(\frac{\text{Time}(B)}{\text{Time}(A)} = \frac{2.4N}{2N} = 1.2\)

- So, Machine A is 20% faster than Machine B for this program

---

**Iron Law Example**

Keep clock(A) @ 1ns and clock(B) @2ns

For equal performance, if CPI(B)=1.2, what is CPI(A)?

\[
\frac{\text{Time}(B)}{\text{Time}(A)} = 1 = \frac{(N \times 2 \times 1 \times \text{CPI}(A))}{(N \times 2 \times 1.2)}
\]

\[\text{CPI}(A) = 2.4\]

---

**Another Example**

<table>
<thead>
<tr>
<th>OP</th>
<th>Freq</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>43%</td>
<td>1</td>
</tr>
<tr>
<td>Load</td>
<td>21%</td>
<td>1</td>
</tr>
<tr>
<td>Store</td>
<td>12%</td>
<td>2</td>
</tr>
<tr>
<td>Branch</td>
<td>24%</td>
<td>2</td>
</tr>
</tbody>
</table>

- Assume stores can execute in 1 cycle by slowing clock 15%
- Should this be implemented?
Example-- Let's do the math:

<table>
<thead>
<tr>
<th>OP</th>
<th>Freq</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>43%</td>
<td>1</td>
</tr>
<tr>
<td>Load</td>
<td>21%</td>
<td>1</td>
</tr>
<tr>
<td>Store</td>
<td>12%</td>
<td>2</td>
</tr>
<tr>
<td>Branch</td>
<td>24%</td>
<td>2</td>
</tr>
</tbody>
</table>

- Old CPI = 0.43 + 0.21 + 0.12 \times 2 + 0.24 \times 2 = 1.36
- New CPI = 0.43 + 0.21 + 0.12 + 0.24 \times 2 = 1.24

- Speedup = old time/new time
  = (P \times \text{old CPI} \times T)/(P \times \text{new CPI} \times 1.15T)
  = (1.36)/(1.24 \times 1.15) = 0.95
- Answer: Don't make the change

Instruction Set Architecture

- ISA, the boundary between software and hardware
  - Specifies the logical machine that is visible to the programmer
  - Also, a functional spec for the processor designers
- What needs to be specified by an ISA
  - Operations
    - what to perform and what to perform next
  - TemporaryOperand Storage in the CPU
    - accumulator, stacks, registers
  - Number of operands per instruction
  - Operand location
    - where and how to specify the operands
  - Type and size of operands
  - Instruction-to-Binary Encoding

Basic ISA Classification

- Stack Architecture (zero operand):
  - Operands popped from stack
  - Result pushed on stack
- Accumulator (one operand):
  - Special accumulator register is implicit operand
  - Other operand from memory
- Register-Memory (two operand):
  - One operand from register, other from memory or register
  - Generally, one of the source operands is also the destination
  - A few architectures—e.g. VAX, M68000—have allowed mem. to mem. operations
- Register-Register or Load/Store (three operand):
  - All operands for ALU instructions must be registers
  - General format R_d <= R_s op R_t
  - Separate Load and Store instructions for memory access

Other Important ISA Considerations

- Number of registers
- Addressing modes
- Data types/sizes
- Instruction functionality—simple vs. complex
- Branch/jump/subroutine call functionality
- Exception handling
- Instruction format/size/regularity
- Etc.
Addressing Modes

- **register**: \( R_i \)
- **displacement**: \( M[R_i + \#n] \)
- **immediate**: \( \#n \)
- **register indirect**: \( M[R_i] \)
- **indexed**: \( M[R_i + R_j] \)
- **absolute**: \( M[\#n] \)
- **memory indirect**: \( M[M[R_i]] \)
- **auto-increment**: \( M[R_i]; R_i += d \)
- **auto-decrement**: \( M[R_i]; R_i -= d \)
- **scaled**: \( M[R_i + \#n + R_j \times d] \)
- **update**: \( M[R_i = R_i + \#n] \)

- Modes 1-4 account for 93% of all VAX operands
  [Clark and Emer]

Operations

- arithmetic and logical - and, add ...
- data transfer - move, load, store
- control - branch, jump, call
- system - system call, traps
- floating point - add, mul, div, sqrt
- decimal - addd, convert
- string - move, compare
- multimedia? 2D, 3D? e.g., Intel MMX/SSE and Sun VIS

Control Instructions (Branches)

1. Types of Branches
   - A. Conditional or Unconditional
   - B. Save PC?
   - C. How is target computed?
     - Single target (immediate, PC+immediate)
     - Multiple targets (register)
2. Branch Architectures
   - A. Condition code or condition registers
   - B. Register

Save or Restore State

- What state?
  - function calls: registers (CISC)
  - system calls: registers, flags, PC, PSW, etc
- Hardware need not save registers
  - caller can save registers in use
  - callee save registers it will use
- Hardware register save
  - IBM STM, VAX CALLS
  - faster?
- Most recent architectures do no register saving
  - Or do implicit register saving with register windows (SPARC)
### Anatomy of a “Modern” ISA

- **Operations**
  - simple ALU op’s, data movement, control transfer
- **Temporary Operand Storage in the CPU**
  - Large General Purpose Register (GPR) File
- **Load/Store Architecture**
  - Three operands per ALU instruction (all registers)
    - \( A \leftarrow B \ op \ C \)
- **Addressing Modes**
  - Limited addressing modes—e.g. register indirect addressing
- **Type and size of operands**
  - 32/64-bit integers, IEEE floats
- **Instruction-to-Binary Encoding**
  - Fixed width, regular fields

*Exceptions: Intel x86, IBM 390 (aka z900)*

### MIPS ISA

- **The MIPS ISA was one of the first RISC instruction sets (1985)**
- **Similar to ISAs of other RISC processors: Sun SPARC, HP PA-RISC, DEC Alpha**
- **Main characteristics**
  - Load-store architecture
  - Three operand format (\( R_d \leftarrow R_s \ op \ R_t \) )
  - 32 General Purpose Registers (actually 31)
  - Only one addressing mode for memory operands: reg.indirect w. displ.
  - Limited, highly orthogonal instruction set: 52 instructions
  - Simple branch/jump/subroutine call architecture

### MIPS Instruction Format

<table>
<thead>
<tr>
<th>Type</th>
<th>format (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>( \text{opcode (6)} \ \ \ \ r_s (5) \ \ \ rt (5) \ \ \ rd (5) \ \ \ shamt (5) \ \ \ \text{funct (6)} )</td>
</tr>
<tr>
<td>I</td>
<td>( \text{opcode (6)} \ \ \ \ r_s (5) \ \ \ rt (5) \ \ \ \text{immediate (16)} )</td>
</tr>
<tr>
<td>J</td>
<td>( \text{opcode (6)} \ \ \ \ \text{address (26)} )</td>
</tr>
</tbody>
</table>

*Four instruction formats (branch format, not shown above, is similar to I format)*

*All instructions are 32 bits with fixed op-code size*

### A CISC ISA—x86 (IA-32)

- **This ISA was first introduced with the Intel 8086 processor in 1978**
- **Has evolved, with many additions over the years**
- **Main characteristics:**
  - Reg-mem architecture—ALU instructions can have memory operands
  - Two operand format—one source operand is also destination
  - Eight general purpose registers
  - Seven memory addressing modes
  - More than 500 instructions
  - Instruction set is non-orthogonal
  - Highly variable instruction size and format—instruction size varies from 1 to 17 bytes.
X86 Instruction Format

<table>
<thead>
<tr>
<th>Instruction Format</th>
<th>Opcodes</th>
<th>ModRM</th>
<th>SI</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up to four</td>
<td>1 or 2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>pre-indexed</td>
<td>byte</td>
<td>byte</td>
<td>byte</td>
<td>displacement of 1, 2, or 4 bytes or none</td>
<td>immediate</td>
</tr>
<tr>
<td>bytes</td>
<td></td>
<td>(if required)</td>
<td>(if required)</td>
<td></td>
<td>none of 1, 2, or 4 bytes or none</td>
</tr>
<tr>
<td>(optional)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

X86 Addressing Modes

- Absolute
- Register indirect
- Based
- Based indexed
- Based indexed with displacement
- Based with scaled index
- Based with scaled index and displacement

Dynamic-Static Interface

- Semantic gap between s/w and h/w
- Placement of DSI determines how gap is bridged

Dynamic-Static Interface

- Low-level DSI exposes more knowledge of hardware through the ISA
  - Places greater burden on compiler/programmer
- Optimized code becomes specific to implementation
  - In fact: happens for higher-level DSI also
The Role of the Compiler

- Phases to manage complexity
  - Parsing $\rightarrow$ intermediate representation
  - Procedure inlining
  - Loop Optimizations
  - Common Sub-Expression
  - Jump Optimization
  - Constant Propagation
  - Register Allocation
  - Strength Reduction
  - Pipeline Scheduling
  - Code Generation $\rightarrow$ assembly code

Performance and Cost

- Which computer is fastest?
- Not so simple
  - Scientific simulation $-$ FP performance
  - Program development $-$ Integer performance
  - Commercial workload $-$ Memory, I/O

Performance of Computers

- Want to buy the fastest computer for what you want to do?
  - Workload is all-important
  - Correct measurement and analysis
- Want to design the fastest computer for what the customer wants to pay?
  - Cost is always an important criterion
- Speed is not always the only performance criteria:
  - Power
  - Area

Defining Performance

- What is important to whom?
- Computer system user
  - Minimize elapsed time for program $=$ $time_{end} - time_{start}$
  - Called response time
- Computer center manager
  - Maximize completion rate $=$ $\#jobs/second$
  - Called throughput
Improve Performance

• Improve (a) response time or (b) throughput?
  – Faster CPU
    • Helps both (a) and (b)
  – Add more CPUs
    • Helps (b) and perhaps (a) due to less queuing

Performance Comparison

• Machine A is n times faster than machine B iff
  \( \frac{\text{perf}(A)}{\text{perf}(B)} = \frac{\text{time}(B)}{\text{time}(A)} = n \)
• Machine A is x% faster than machine B iff
  \( \frac{\text{perf}(A)}{\text{perf}(B)} = \frac{\text{time}(B)}{\text{time}(A)} = 1 + \frac{x}{100} \)
  
  E.g. time(A) = 10s, time(B) = 15s
  – 15/10 = 1.5 => A is 1.5 times faster than B
  – 15/10 = 1.5 => A is 50% faster than B

Possible Performance Metrics

• MIPS and MFLOPS
  • MIPS = \( \frac{\text{instruction count}}{(\text{execution time} \times 10^6)} = \frac{\text{clock rate}}{(\text{CPI} \times 10^6)} \)
  • But MIPS has serious shortcomings

Problems with MIPS

• E.g. without FP hardware, an FP op may take 50 single-cycle instructions
• With FP hardware, only one 2-cycle instruction
  
  • Thus, adding FP hardware:
    – CPI increases (why?) 50/50 => 2/1
    – Instructions/program decreases (why?) 50 => 1
    – Total execution time decreases 50 => 2
  • BUT, MIPS gets worse! 50 MIPS => 2 MIPS
Problems with MIPS

- Ignore program
- Usually used to quote peak performance
  - Ideal conditions => guarantee not to exceed!
- When is MIPS ok?
  - Same compiler, same ISA
  - E.g. same binary running on Pentium-III, IV
  - Why? Instr/program is constant and can be ignored

Other Metrics

- MFLOPS = FP ops in program/(execution time x 10^6)
- Assuming FP ops independent of compiler and ISA
  - Often safe for numeric codes: matrix size determines # of FP ops/program
  - However, not always safe:
    - Missing instructions (e.g. FP divide, sqrt/sin/cos)
    - Optimizing compilers
- Relative MIPS and normalized MFLOPS
  - Normalized to some common baseline machine
    - E.g. VAX MIPS in the 1980s

Which Programs

- Execution time of what program?
- Best case – you always run the same set of programs
  - Port them and time the whole workload
- In reality, use benchmarks
  - Programs chosen to measure performance
  - Predict performance of actual workload
  - Saves effort and money
  - Representative? Honest? Benchmarking…

Types of Benchmarks

- Real programs
  - representative of real workload
  - only accurate way to characterize performance
  - requires considerable work
- Kernels or microbenchmarks
  - "representative" program fragments
  - good for focusing on individual features not big picture
- Instruction mixes
  - instruction frequency of occurrence; calculate CPI
Benchmarks: SPEC2000

- System Performance Evaluation Cooperative
  - Formed in 80s to combat benchmarking
  - SPEC89, SPEC92, SPEC95, Spec2000, now SPEC2006
- 12 integer and 14 floating-point programs
  - Sun Ultra-5 300MHz reference machine has score of 100
  - Report geometric mean of ratios to reference machine

Benchmarks: SPEC CINT2000

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>Compression</td>
</tr>
<tr>
<td>175.vpr</td>
<td>FPGA place and route</td>
</tr>
<tr>
<td>176.gcc</td>
<td>C compiler</td>
</tr>
<tr>
<td>181.mcf</td>
<td>Combinatorial optimization</td>
</tr>
<tr>
<td>186.crafty</td>
<td>Chess</td>
</tr>
<tr>
<td>197.parser</td>
<td>Word processing, grammatical analysis</td>
</tr>
<tr>
<td>252.eon</td>
<td>Visualization (ray tracing)</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>PERL script execution</td>
</tr>
<tr>
<td>254.gap</td>
<td>Group theory interpreter</td>
</tr>
<tr>
<td>255.vortex</td>
<td>Object-oriented database</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>Compression</td>
</tr>
<tr>
<td>300.twolf</td>
<td>Place and route simulator</td>
</tr>
</tbody>
</table>

Benchmarks: SPEC CFP2000

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>168.wupwise</td>
<td>Physics/Quantum Chromodynamics</td>
</tr>
<tr>
<td>171.swim</td>
<td>Shallow water modeling</td>
</tr>
<tr>
<td>172.mgrid</td>
<td>Multi-grid solver: 3D potential field</td>
</tr>
<tr>
<td>173.applu</td>
<td>Parabolic/elliptic PDE</td>
</tr>
<tr>
<td>177.mesa</td>
<td>3-D graphics library</td>
</tr>
<tr>
<td>178.galgel</td>
<td>Computational Fluid Dynamics</td>
</tr>
<tr>
<td>179.art</td>
<td>Image Recognition/Neural Networks</td>
</tr>
<tr>
<td>183.earthquake</td>
<td>Seismic Wave Propagation Simulation</td>
</tr>
<tr>
<td>187.facerec</td>
<td>Image processing: face recognition</td>
</tr>
<tr>
<td>188.ammp</td>
<td>Computational chemistry</td>
</tr>
<tr>
<td>189.lucas</td>
<td>Number theory/primality testing</td>
</tr>
<tr>
<td>191.m3d</td>
<td>Finite-element Crash Simulation</td>
</tr>
<tr>
<td>200.sixtrack</td>
<td>High energy nuclear physics accelerator design</td>
</tr>
<tr>
<td>301.apsi</td>
<td>Meteorology: Pollutant distribution</td>
</tr>
</tbody>
</table>

Benchmark Pitfalls

- Benchmark not representative
  - Your workload is I/O bound, SPECint is useless
- Benchmark is too old
  - Benchmarks age poorly; benchmarketing pressure causes vendors to optimize compiler/hardware/software to benchmarks
  - Need to be periodically refreshed
Benchmark Pitfalls

• Choosing benchmark from the wrong application space
  – e.g., in a realtime environment, choosing gcc
• Choosing benchmarks from no application space
  – e.g., synthetic workloads, esp. unvalidated ones
• Using toy benchmarks (dhrystone, whetstone)
  – e.g., used to prove the value of RISC in early 80’s
• Mismatch of benchmark properties with scale of features studied
  – e.g., using SPECINT for large cache studies

Benchmark Pitfalls

• Carelessly scaling benchmarks
  – Truncating benchmarks
  – Using only first few million instructions
  – Reducing program data size
• Too many easy cases
  – May not show value of a feature
• Too few easy cases
  – May exaggerate importance of a feature

Scalar to Superscalar

• Scalar processor—Fetches and issues at most one instruction per machine cycle
• Superscalar processor—Fetches and issues multiple instructions per machine cycle
• Can also define superscalar in terms of how many instructions can complete execution in a given machine cycle.
• Note that only a superscalar architecture can achieve a CPI of less than 1

Processor Performance

\[
\text{Processor Performance} = \frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}
\]

• In the 1980’s (decade of pipelining):
  – CPI: 5.0 => 1.15
• In the 1990’s (decade of superscalar):
  – CPI: 1.15 => 0.5 (best case)
Amdahl’s Law
(Originally formulated for vector processing)

- \( f \) = fraction of program that is vectorizable
- \( 1-f \) = fraction that is serial
- \( N \) = speedup for vectorizable portion
- Overall speedup:

\[
\text{Speedup} = \frac{1}{(1-f) + \frac{f}{N}}
\]

Amdahl’s Law--Continued

- Sequential bottleneck
- Even if \( N \) is infinite
  - Performance limited by nonvectorizable portion \((1-f)\)

\[
\lim_{N\to\infty} \frac{1}{1-f + \frac{f}{N}} = \frac{1}{1-f}
\]

Ramifications of Amdahl’s Law

- Consider: \( f = 0.9, (1-f) = 0.1 \)
  For \( N \to \), Speedup \( \to 10 \)
- Consider: \( f = 0.5, (1-f) = 0.5 \)
  For \( N \to \) infinity, Speedup \( \to 2 \)
- Consider: \( f = 0.1, (1-f) = 0.9 \)
  For \( N \to \) infinity, Speedup \( \to 1.1 \)

Maximum Achievable Speedup

[Graph showing the maximum achievable speedup vs. parallelizable fraction \( f \)]
Pipelining

Unpipelined operation

Inputs $I_1, I_2, I_3, \ldots$

Outputs $O_1, O_2, \ldots$

Time required to process $K$ inputs = $KT$

Perfect Pipeline ($N$ stages):

- Time required to process $K$ inputs = $(K + N - 1)T/N$

Note: For $K \gg N$, the processing time approaches $KT/N$

Amdahl's Law Applied to Pipelining

- $g = \text{fraction of time the pipeline is full}$
- $(1-g) = \text{fraction that it is not full}$
- $N = \text{pipeline depth}$
- Overall speedup:

$$\text{Speedup} = \frac{1}{(1-g) + \frac{g}{N}}$$
**Pipelined Performance Model**

- Tyranny of Amdahl’s Law (Bob Colwell)
  - When g is even slightly below 100%, a big performance hit will result
  - Stalled cycles are the key adversary and must be minimized as much as possible

**Superscalar Proposal**

- Moderate tyranny of Amdahl’s Law
  - Ease sequential bottleneck
  - More generally applicable
  - Robust (less sensitive to f)
  - Revised Amdahl’s Law:

\[
\text{Speedup} = \frac{1}{(1 - f) + \frac{f}{s}}
\]

**Motivation for Superscalar**

[Agerwala and Cocke]

Speedup jumps from 3 to 4.3 for N=6, f=0.8, but s = 2 instead of s=1 (scalar)

**Limits on Instruction Level Parallelism (ILP)**

<table>
<thead>
<tr>
<th>Source</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weiss and Smith [1984]</td>
<td>1.58</td>
</tr>
<tr>
<td>Sohi and Vajapeyam [1987]</td>
<td>1.81</td>
</tr>
<tr>
<td>Tjaden and Flynn [1970]</td>
<td>1.86 (Flynn's bottleneck)</td>
</tr>
<tr>
<td>Tjaden and Flynn [1973]</td>
<td>1.86</td>
</tr>
<tr>
<td>Ull [1986]</td>
<td>2.00</td>
</tr>
<tr>
<td>Smith et al. [1988]</td>
<td>2.00</td>
</tr>
<tr>
<td>Johnson [1991]</td>
<td>2.50</td>
</tr>
<tr>
<td>Acosta et al. [1999]</td>
<td>2.79</td>
</tr>
<tr>
<td>Wedig [1982]</td>
<td>3.00</td>
</tr>
<tr>
<td>Butler et al. [1991]</td>
<td>5.6</td>
</tr>
<tr>
<td>Nien and Pall [1991]</td>
<td>6.0</td>
</tr>
<tr>
<td>Wall [1991]</td>
<td>7.0 (Jouppi disagreed)</td>
</tr>
<tr>
<td>Kuck et al. [1972]</td>
<td>8.0</td>
</tr>
<tr>
<td>Raseeman and Foster [1972]</td>
<td>31 (no control dependences)</td>
</tr>
<tr>
<td>Nicolas and Fisher [1994]</td>
<td>30 (Fisher's opinion)</td>
</tr>
</tbody>
</table>

**Vectorizability f**

**Typical Range**
Superscalar Proposal

- Go beyond single instruction pipeline, achieve IPC > 1
- Dispatch multiple instructions per cycle
- Provide more generally applicable form of concurrency (not just vectors)
- Geared for sequential code that is hard to parallelize otherwise
- Exploit fine-grained or instruction-level parallelism (ILP)

Classifying ILP Machines

[Jouppi, DECWRL 1991]

- Baseline scalar RISC
  - Issue parallelism = IP = 1
  - Operation latency = OP = 1
  - Peak IPC = 1

Superpipelined: cycle time = 1/m of baseline
- Issue parallelism = IP = 1 inst / minor cycle
- Operation latency = OP = m minor cycles
- Peak IPC = m instr / major cycle (m x speedup?)

Superscalar:
- Issue parallelism = IP = n inst / cycle
- Operation latency = OP = 1 cycle
- Peak IPC = n instr / cycle (n x speedup?)
Classifying ILP Machines

[Jouppi, DECWRL 1991]
- VLIW: Very Long Instruction Word
  - Issue parallelism = IP = n inst / cycle
  - Operation latency = OP = 1 cycle
  - Peak IPC = n instr / cycle = 1 VLIW / cycle

Superscalar vs. Superpipelined

- Roughly equivalent performance
  - If n = m then both have about the same IPC
  - Parallelism exposed in space vs. time

Superpipelined-Superscalar
- Issue parallelism = IP = n inst / minor cycle
- Operation latency = OP = m minor cycles
- Peak IPC = n x m instr / major cycle