SYLLABUS

ECE 685-001
DIGITAL COMPUTER STRUCTURE/ARCHITECTURE
COURSE SYLLABUS
FALL, 2002

• Instructor: Dr. J. Robert (Bob) Heath
• Office: 313 Electrical Engineering Annex
• Office Phone Number: (859) 257-3124
• Email: heath@ engr.uky.edu
• Web Page: http://www. engr.uky.edu/~heath
• Office Hours: M (3:30 pm-5:00 pm)
               W (2:30 pm-4:00 pm)


Heath


• **Meeting Schedule** MWF (10:00 am-10:50 am) 303 Slone Research Bldg.

• **Course Description** Study of fundamental concepts in digital computer system architecture/structure and design. Topics include: computer system modeling based on instruction set architecture models; architecture and design of datapaths, control units, processors, memory systems hierarchy, and input/output systems. Special topics include floating-point arithmetic, multiple level cache design, pipeline design techniques, multiple issue processors, and an introduction to parallel computer architectures. Use of Hardware Description Languages (HDLs) for architecture/design verification/validation via pre-synthesis simulation. Prereq: EE380 and EE581 or consent of instructor.
SYLLABUS (Continued)

• Topical Outline
  1. Introduction to Computer Architecture and Design Fundamentals.
  2. Instruction Set Architecture Models.
  3. Introduction to Computer Architecture/Design Verification via use of a Hardware Description Language (HDL) – VERILOG.
  4. Instruction Set Principles and Examples.
  5. Pipelining.
  6. Advanced Pipelining and Instruction-Level Parallelism.
  7. Memory Hierarchy Design.
 10. Computer Design & Design Documentation and Verification via VERILOG.
 11. Interconnection Networks.
 13. Introduction to Vector Processors.

• Grade:
  Test 1: (October 11) 25%
  Test 2: (November 25) 25%
  Homework: Design, Design Verification Projects: 25%
  Final Exam -Comprehensive (Fri. Dec. 20 (10:30 am)): 25%

Your final grade will generally be determined by the number of points you have accumulated from 100 possible points as follows:

A: 90-100 pts.
B: 80-89 pts.
C: 70-79 pts.
E: 69 or below

An equitable grade scale will be applied when warranted.
SYLLABUS (Continued)

- **Make-Up Examinations**
  Make-up examinations will only be given to students who miss examinations as a result of excused absences according to applicable university policy. Make-up exams may be of a different format from the regular exam format (Example: Oral format).

- **Cheating:**
  Cheating will not be allowed or tolerated. Anyone who cheats will be dealt with according to applicable university policy. (Assignment of a grade of E for the course).

- **Class Attendance**
  Attendance of all class lectures is required to assure maximum course performance. *You are responsible for all business conducted within a class.*

- **Homework Assignments**
  Homework assignments will be periodically made. All assignments may not be graded. Assignments are due at the beginning of the class period on due dates.
VERILOG

• A Hardware Description Language (HDL) Used To Describe Digital Systems Hardware Design and Structure.
  • A HDL Can Be Used For Digital System Design Verification/Validation and Implementation Via HDL Simulation, Synthesis, And Implementation.
  • Systems May Be Described At Three Levels: Behavioral, Register Transfer (Dataflow, Equation), And Gate (Structural) Levels.

• Example:
  • Binary Full Adder
Verilog Coding Example: Binary Full Adder (bfa)
Gate (Structural) Level Coding Style

module fulladd1 (I2, I1, I0, so, co); // Module Name and Input/Output Signal
    // Declaration
       input I2, I1, I0 ;                 // Input, Output, and Wire Declaration
       output so;
       output co;
       wire wx0, wa0, wa1, wa2;

       xor XO (wx0, I2, I1);           // Gate Identification, Instantiation Names, an Output/Input(s)
       xor X1 (so, wx0, I0);
       and A0 (wa0, I1, I0);
       and A1 (wa1, I2, I1);
       and A2 (wa2, I2, I0);
       or (co, wa0, wa1, wa2);

endmodule                           // End of Module
    // “and AI #2(wf,wc,wd)” Gate Delay Of 2 Simulation Time Units.
Verilog Coding Example: Binary Full Adder (bfa) 
Register-Transfer-Level (RTL), Dataflow, Or Equation Level Coding Style

// "bfa" RTL Coding Style.

module fulladd2 (I2, I1, I0, so, co); // Module Name and Input/Output Signal Declaration.

  input I2, I1, I0; // Input, Output, and Wire Declaration.
  output so;
  output co;

  assign so = I2 ^ I1 ^ I0; // Equations of Binary Full Adder (bfa). Bit Wise Exclusive-OR (^).
  assign co = (I2 && I0) || (I1 && I0) || (I2 && I1); //Logical AND (&&); Logical OR (||).
       //Bit Wise AND (&); Bit Wise OR (|).
endmodule // End of Module.
Verilog Coding Example: Binary Full Adder (bfa)  
Behavioral Level Coding Style

// "bfa" Behavioral Coding Style.

module fulladd3 (I2, I1, I0, so, co);  // Module Name and Input/Output Signal Declaration.
input I2, I1, I0;              // Input and Output Declaration.
output co, so;
reg co, so;                     // Port (Signal) Values Are Held Until They Change.

always @ (I2 or I1 or I0) //Following Code Executed Anytime I2, I1 or I0 Changes
begin                          //Value.
    case ({I2,I1,I0})       //Use Behavioral Level "Case" Structure.
        3'b000: begin co=1'b0; so=1'b0; end //Implements Truth Table..
        3'b001: begin co=1'b0; so=1'b1; end
        3'b010: begin co=1'b0; so=1'b1; end //3'b010 Implies 3-Bits Binary And They are //010.
        3'b011: begin co=1'b1; so=1'b0; end
        3'b100: begin co=1'b0; so=1'b1; end
        3'b101: begin co=1'b1; so=1'b0; end
        3'b110: begin co=1'b1; so=1'b0; end
        3'b111: begin co=1'b1; so=1'b1; end
    endcase
end
endmodule
module testfulladd1;
    reg I2, I1, I0, Cot, Sot, flag;  // Signals Declared to be Registers (Hold Values Until Changed)
    wire Sos,Cos;

    fulladd1 ADD0(I2, I1, I0, Sos, Cos);  // Instantiation of Module Under Test (MUT)

    initial                          // This Process Defines Signals We Want to View as
        begin                            // System is Simulated and the Code We Represent
            // Signals in. Choices are Binary, Octal, or Hex
            $monitor($time, "$time, "I2=%b I1=%b I0=%b Cos=%b Cot=%b Sos=%b Sot=%b flag=%b", I2,I1,I0,Cos,Cot,Sos,Sot,flag);
        end

    initial                          // This Process Generates Stimulus Inputs to MUT
        begin: I2_loop                   // and for Each a Theoretically Correct Output
            integer m;
            for (m = 0; m < 2; m=m+1)    // Stimulus Applied to MUT Inputs
                begin: I1_loop
                    integer n;
                    for (n = 0; n < 2; n=n+1)
                        begin: I0_loop
                            integer o;
                            for (o = 0; o < 2; o=o+1)
                                begin
                                    #1 I2 = m; I1 = n; I0 = o; // Stimulus Applied to MUT Inputs

                                end

                            end

                        end

                    end

                end

            end

        end

Heath
AUTOMATED TESTBENCH FOR “fulladd1” MODULE (Continued)

if(m == 0 && n == 0 && o == 0)  Cot = 0;    // Generation of
if(m == 0 && n == 0 && o == 0)  Sot = 0;     // Theoretically
if(m == 0 && n == 0 && o == 1)  Cot = 0;    // Correct Outputs
if(m == 0 && n == 0 && o == 1)  Sot = 1;
if(m == 0 && n == 1 && o == 0)  Cot = 0;
if(m == 0 && n == 1 && o == 0)  Sot = 1;
if(m == 0 && n == 1 && o == 1)  Cot = 1;
if(m == 0 && n == 1 && o == 1)  Sot = 0;
if(m == 1 && n == 0 && o == 0)  Cot = 0;
if(m == 1 && n == 0 && o == 0)  Sot = 1;
if(m == 1 && n == 0 && o == 1)  Cot = 1;
if(m == 1 && n == 0 && o == 1)  Sot = 0;
if(m == 1 && n == 1 && o == 0)  Cot = 1;
if(m == 1 && n == 1 && o == 0)  Sot = 0;
if(m == 1 && n == 1 && o == 1)  Cot = 1;
if(m == 1 && n == 1 && o == 1)  Sot = 1;

#1 if((Cos==Cot)&&(Sos==Sot)) flag = 0;    // Comparison of Theoretically
    else flag = 1;                      // Correct Output to MUT Output.
    end                // “==“ Logical Equality
end

#5 flag = 1;
end

initial                  // Process Which Will Shut Down a Run-Away Simulation
begin                    // ie, Getting in an Endless Loop.
#150 $finish;
end

endmodule

---

Heath 11
SIMUCAD SILOS III DEMONSTRATIONS

(Use Files “bfa_gate_level” And “testbfa_gate_level”)
(Use Files “mux4x1_gate_level” And “testmux4x1_gate_level”)

- Examples Of “Exhaustive Automated” Testbenches
CHAP. 1 –Fundamentals of Computer Design

- Performance Growth

<table>
<thead>
<tr>
<th>Year</th>
<th>Growth Rate</th>
<th>Technology and Architectural Innovation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1945</td>
<td>( \text{? % Per Year} )</td>
<td>Technology and Architectural Innovation.</td>
</tr>
<tr>
<td>1970</td>
<td>25% – 35% Per Year</td>
<td>Technology Only.</td>
</tr>
</tbody>
</table>
| 1980 | 50\% Per Year        | *RISC – Technology and Arch. Innovation (Inst. Level Parallelism (ILP) and Cache Memories.
|      |                      | * (Microprocessor Driven) Fewer: Mainframes Supercomputers |
| 1995 | 50\% Per Year        | *Technology and Architectural Innovation. (Super Scalar Pipelining, ILP, and Parallel Processing) |
|      |                      | *Application Classes: PCs, Servers, Embedded Computers. |
| 2002 |                      |                                         |
Growth In Microprocessor Performance Due to Technology and Architectural Innovation

© 2003 Elsevier Science (USA). All rights reserved.

Heath
TASK OF A COMPUTER DESIGNER (ARCHITECT)

1. Determine Important Requirements/Attributes of a New Computer to be Developed.

2. Design to Meet Requirements/Attributes and to Maximize Performance (When Required) and Lower Cost.

• **Design Steps:**
  1. Assembly Language Instruction Set Design.
  3. Logic Design of Functional Units to RTL and Gate Level only when Required.
  4. Design Capture (Verilog or VHDL).
  6. Implementation.
    * IC Design, Layout, etc.
    * Power
    * Cooling
    * Low Power Design
    * Testing

• **Computer Architecture:**
  
  Includes: 1. Instruction Set Architecture Design.
  3. HDL Design Capture/Verification via HDL Simulation and Experimental Prototype Development and Testing.
### CURRENT COMPUTING MARKETS AND CHARACTERISTICS

#### 1.2 The Changing Face of Computing and the Task of the Computer Designer

<table>
<thead>
<tr>
<th>Feature</th>
<th>Desktop</th>
<th>Server</th>
<th>Embedded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price of system</td>
<td>$1000–$10,000</td>
<td>$10,000–$10,000,000</td>
<td>$10–$100,000 (including network routers at the high end)</td>
</tr>
<tr>
<td>Price of microprocessor module</td>
<td>$100–$1000</td>
<td>$200–$2000 (per processor)</td>
<td>$0.20–$200 (per processor)</td>
</tr>
<tr>
<td>Microprocessors sold per year</td>
<td>150,000,000</td>
<td>4,000,000</td>
<td>300,000,000 (32-bit and 64-bit processors only)</td>
</tr>
<tr>
<td>(estimates for 2000)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Critical system design issues</td>
<td>Price-performance, graphics performance</td>
<td>Throughput, availability, scalability</td>
<td>Price, power consumption, application-specific performance</td>
</tr>
</tbody>
</table>
## FUNCTIONAL REQUIREMENTS FACED BY ARCHITECTS
*(Architect Minimizes Cost and Optimizes Performance of Machine That Meets these Requirements)*

<table>
<thead>
<tr>
<th>Functional requirements</th>
<th>Typical features required or supported</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application area</strong></td>
<td>Target of computer</td>
</tr>
<tr>
<td>General-purpose desktop</td>
<td>Balanced performance for a range of tasks, including interactive performance for graphics, video, and audio (Ch. 2, 3, 4, 5)</td>
</tr>
<tr>
<td>Scientific desktops and servers</td>
<td>High-performance floating point and graphics (App. G, H)</td>
</tr>
<tr>
<td>Commercial servers</td>
<td>Support for databases and transaction processing; enhancements for reliability and availability; support for scalability (Ch. 2, 6, 8)</td>
</tr>
<tr>
<td>Embedded computing</td>
<td>Often requires special support for graphics or video (or other application-specific extension); power limitations and power control may be required (Ch. 2, 3, 4, 5)</td>
</tr>
<tr>
<td><strong>Level of software compatibility</strong></td>
<td>Determines amount of existing software for machine</td>
</tr>
<tr>
<td>At programming language</td>
<td>Most flexible for designer; need new compiler (Ch. 2, 6)</td>
</tr>
<tr>
<td>Object code or binary compatible</td>
<td>Instruction set architecture is completely defined—little flexibility—but no investment needed in software or porting programs</td>
</tr>
<tr>
<td><strong>Operating system requirements</strong></td>
<td>Necessary features to support chosen OS (Ch. 5, 8)</td>
</tr>
<tr>
<td>Size of address space</td>
<td>Very important feature (Ch. 5); may limit applications</td>
</tr>
<tr>
<td>Memory management</td>
<td>Required for modern OS; may be paged or segmented (Ch. 5)</td>
</tr>
<tr>
<td>Protection</td>
<td>Different OS and application needs: page vs. segment protection (Ch. 5)</td>
</tr>
<tr>
<td><strong>Standards</strong></td>
<td>Certain standards may be required by marketplace</td>
</tr>
<tr>
<td>Floating point</td>
<td>Format and arithmetic: IEEE 754 standard (App. H), special arithmetic for graphics or signal processing</td>
</tr>
<tr>
<td>I/O bus</td>
<td>For I/O devices: Ultra ATA, Ultra SCSI, PCI (Ch. 7, 8)</td>
</tr>
<tr>
<td>Operating systems</td>
<td>UNIX, PalmOS, Windows, Windows NT, Windows CE, CISCO IOS</td>
</tr>
<tr>
<td>Networks</td>
<td>Support required for different networks: Ethernet, Infiniband (Ch. 8)</td>
</tr>
<tr>
<td>Programming languages</td>
<td>Languages (ANSI C, C++, Java, FORTRAN) affect instruction set (Ch. 2)</td>
</tr>
</tbody>
</table>

**Figure 1.4** Summary of some of the most important functional requirements an architect faces. The left-hand
TECHNOLOGY AND COMPUTER USAGE TRENDS
(IMPACTS ARCHITECTS WORK)

- An Instruction Set Architecture Must Survive Changes In:
  1. Hardware/Software Technology.
  2. Changes In Applications.

Trends In Computer Usage

1. An Increasing Amount of Memory Used by Programs (> 1.5 To 2/Yr.).
   - Implications on Address Bits. (1/2 To 1 Bit/Yr.).

2. Less Use of Assembly Language. Compiler Writers Work Closely With Architects.

Trends In Implementation Technology

(Computer Designers Must Have An Awareness)

1. Integrated Circuit Logic Technology.
   - 55%/Yr. Growth Rate in Transistor Count/Chip.
     - Due To: Transistor Density Increase – 35%/Yr.
     - Quadruples in Approximately 4 Yrs.
     - Die Size Increase – 10% - 20% Yr.
2. ** Semiconductor DRAM.  
   • 40% - 60% Increase in Density/Yr.  
     * Quadruple In 3 - 4 Years.  
   • Slow Improvement in Cycle Time.  
     * Decrease of 1/3 In 10 Years.  

3. ** Magnetic Disk Technology.  
   • Disk Density Improvement of 100% / Yr. (Unbelievable!!!)  
     * Quadruple In 2 Years.  
   • Access Time Reduced by 1/3 In 10 Yrs.  
     ** RAID ?????** LATER !!!!

4. ** Network Technology and Performance  
   • Based on Performance of Switches and Transmission System (Latency and Bandwidth are Main Parameters). Bandwidth is Current focus.  
   • 10Mb to 100Mb Ethernet Technology took 10 Yrs.  
   • 100Mb to 1Gb in 5 Yrs.  
   • Internet Doubles In Bandwidth Every Year.
5. Integrated Circuits

- Transistor Performance Increases Linearly With Decreasing Feature Size.
- Power Is Challenge!! Energy Per Transistor is Proportional to Product of Load Capacitance (Worse for Smaller Transistors), Frequency of Switching, and Square of Voltage.
COSTS AND TRENDS

3 PHILOSOPHIES:

1. **High Performance at Any Cost!**
   Supercomputers. (Diminishing Market)

2. **Use Technology, New Architectures to Achieve Lower Cost, Higher Performance.**
   Increasing Market, Most Challenging! General Purpose and Special Purpose Processors. (Text Focus.)

3. **Low Price at Any Cost to Performance.**
   (PC Clones, Some Embedded Processor Applications!)

   **Learning Curve!!!** (What is It???)

   - **Principle That Drives Cost Down!**
     - Manufacturing Cost Decrease Over Time Due to Change In **“YIELD”.**
     - **EFFECT:** Cost Per Megabyte of DRAM Drops Approximately 40%/Yr.

   **Lets Take a Look!!**
COST FACTORS OTHER THAN LEARNING CURVE

2. **Commodities** - Products Sold By Multiple Vendors In Large Volumes (Printers, RAM Memory, Scanners, etc.)

COMPUTER COSTS ARE PROPORTIONAL TO IC COSTS!!


Cost of IC = Cost of Die + Cost of Testing Die + Cost of Packaging and Final Test

\[
\text{Final Test Yield}
\]

Where:

\[
\text{Cost of Die} = \frac{\text{Cost of Wafer}}{\text{Dies Per Wafer} \times \text{Die Yield}}
\]

And:

\[
\text{Dies per Wafer} = \pi \times (\text{Wafer Diameter}/2)^2 - \pi \times \text{Wafer Diameter} \\
\frac{\text{Die Area}}{\sqrt{2 \times \text{Die Area}}}
\]
Finally:  \[
\text{Die Yield} = \text{Wafer Yield} \times \left(1 + \text{Defects per Unit Area} \times \text{Die Area}\right)^{-\alpha}
\]

**DISTRIBUTION OF COSTS IN A SYSTEM**

<table>
<thead>
<tr>
<th>System</th>
<th>Subsystem</th>
<th>Fraction of total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cabinet</td>
<td>Sheet metal, plastic</td>
<td>2%</td>
</tr>
<tr>
<td></td>
<td>Power supply, fans</td>
<td>2%</td>
</tr>
<tr>
<td></td>
<td>Cables, nuts, bolts</td>
<td>1%</td>
</tr>
<tr>
<td></td>
<td>Shipping box, manuals</td>
<td>1%</td>
</tr>
<tr>
<td></td>
<td><strong>Subtotal</strong></td>
<td><strong>6%</strong></td>
</tr>
<tr>
<td>Processor board</td>
<td>Processor</td>
<td>22%</td>
</tr>
<tr>
<td></td>
<td>DRAM (128 MB)</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td>Video card</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td>Motherboard with basic I/O support, networking</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td><strong>Subtotal</strong></td>
<td><strong>37%</strong></td>
</tr>
<tr>
<td>I/O devices</td>
<td>Keyboard and mouse</td>
<td>3%</td>
</tr>
<tr>
<td></td>
<td>Monitor</td>
<td>19%</td>
</tr>
<tr>
<td></td>
<td>Hard disk (20 GB)</td>
<td>9%</td>
</tr>
<tr>
<td></td>
<td>DVD drive</td>
<td>6%</td>
</tr>
<tr>
<td></td>
<td><strong>Subtotal</strong></td>
<td><strong>37%</strong></td>
</tr>
<tr>
<td>Software</td>
<td>OS + Basic Office Suite</td>
<td>20%</td>
</tr>
</tbody>
</table>

*Figure 1.9* Estimated distribution of costs of the components in a $1000 PC in 2001.
• **How** Can a Computer Architect Help Reduce Computer System Costs??

*Reduce Die Area!!*

*Cost of Die = f (Die Area^4)*

**COST VS PRICE:**

---

**Figure 1.10** The components of price for a $1000 PC. Each increase is shown along the bottom as a tax on the prior price. The percentages of the new price for all elements are shown on the left of each column.
COMPUTER PERFORMANCE
(MEASUREMENT/REPORTING)

• Computer Performance is Based on the Time (Execution Time, Response Time) it Takes a Computer to Execute “Real Programs”.

• Computer X is “n” Times Faster Than Computer Y If:

\[ n = \frac{\text{Performance}_x}{\text{Performance}_y} = \frac{\text{ExecutionTime}_y}{\text{ExecutionTime}_x} \]

\[ = \frac{1}{\text{Performance}_y} \]

\[ = \frac{1}{\text{Performance}_x} \]
COMPUTER PERFORMANCE (continued)

• $\text{CPU}_{\text{time}} = \text{CPU}_{\text{time (user)}} + \text{CPU}_{\text{time (OS in Supporting User Program)}}$

• UNIX Time Command Can Acquire Above Information for Program Runs;

  $\text{CPU}_{\text{Time (User)}}$, $\text{System CPU}_{\text{Time}}$, Elapsed Time

• 90.7s 12.9s 2:39 65%

  Percentage of Elapsed Time That Is $\text{CPU}_{\text{Time}}$. 
PERFORMANCE EVALUATION PROGRAMS (Benchmarks)

Best => To => Inadequate

• **Real Applications** (Best!!)

• **Benchmark Suites** – Collections of Standardized Benchmark Programs Used to Measure Performance of Different Application Classes. Most popular – SPEC (Standard Performance Evaluation Corporation) Benchmark Set. [www.spec.org](http://www.spec.org)

• **Modified/Scripted Applications**

• **Kernels** (Livermore Loops, Linpack, etc.)

• **Toy Benchmarks** (10 to 100 Lines of Code – Puzzle, Quicksort, etc.)
  Worthless in Measuring Performance.

• **Synthetic Benchmarks** Similar to Kernels - Try to Match Average Frequency of Operations and Operands of a Large Set of Programs. Not Real Programs.

Some Computer Companies Spend Large Amounts of Time/Money Optimizing Their Computers (Hardware and Software) To Execute Benchmark Programs In Minimum Time.
BENCHMARKS FOR THREE PROCESSOR CLASSES

Desktop Benchmarks

• Two Classes:

1. CPU Intensive (SPEC CPU 2000 – 11 Integer Programs and 14 FP Programs) - See Next Slide.

2. Graphics Intensive (SPECviewperf - www.spec.org)

Server Benchmarks

• SPEC CPU 2000 – Measures Processing Rate of a Multiprocessor By Execution of a Copy on Each Processor. Leads to A Measurement Called ‘SPECrate’.
• File Server Benchmarks – SPECSFS
• Web Server Benchmarks – SPECWeb
• I/O System Benchmarks (Disk and Network) – SPECSFS
• Transaction Processing (TP) Benchmarks (Database Access and Updates) - TPCi
Embedded Benchmarks

• Least Developed Benchmarking State-of-the Art Because of Wide Range of Applications and Performance Requirements.
• For Applications That Can Be Characterized By Kernel Performance, the EDN Embedded Microprocessor Benchmark Consortium (EEMBC) Has Developed the EEMBC Benchmarks.

Five Classes: (34 Benchmarks Total)

1. Automotive/Industrial
2. Consumer
3. Networking
4. Office Automation
5. Telecommunications
### Programs In SPEC CPU2000 Benchmark Suites

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Type</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>Integer</td>
<td>C</td>
<td>Compression using the Lempel-Ziv algorithm</td>
</tr>
<tr>
<td>vpr</td>
<td>Integer</td>
<td>C</td>
<td>FPGA circuit placement and routing</td>
</tr>
<tr>
<td>gcc</td>
<td>Integer</td>
<td>C</td>
<td>Consists of the GNU C compiler generating optimized machine code</td>
</tr>
<tr>
<td>mcf</td>
<td>Integer</td>
<td>C</td>
<td>Combinatorial optimization of public transit scheduling</td>
</tr>
<tr>
<td>crafty</td>
<td>Integer</td>
<td>C</td>
<td>Chess-playing program</td>
</tr>
<tr>
<td>parser</td>
<td>Integer</td>
<td>C</td>
<td>Syntactic English language parser</td>
</tr>
<tr>
<td>eon</td>
<td>Integer</td>
<td>C++</td>
<td>Graphics visualization using probabilistic ray tracing</td>
</tr>
<tr>
<td>perlmbk</td>
<td>Integer</td>
<td>C</td>
<td>Perl (an interpreted string-processing language) with four input scripts</td>
</tr>
<tr>
<td>gap</td>
<td>Integer</td>
<td>C</td>
<td>A group theory application package</td>
</tr>
<tr>
<td>vortex</td>
<td>Integer</td>
<td>C</td>
<td>An object-oriented database system</td>
</tr>
<tr>
<td>bzip2</td>
<td>Integer</td>
<td>C</td>
<td>A block-sorting compression algorithm</td>
</tr>
<tr>
<td>twolf</td>
<td>Integer</td>
<td>C</td>
<td>Timberwolf: a simulated annealing algorithm for VLSI place and route</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>equake</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>fma3d</td>
<td>FP</td>
<td>F90</td>
<td>Finite element modeling of crash simulation</td>
</tr>
<tr>
<td>sixtrack</td>
<td>FP</td>
<td>F77</td>
<td>High-energy physics accelerator design simulation</td>
</tr>
<tr>
<td>apsi</td>
<td>FP</td>
<td>F77</td>
<td>A meteorological simulation of pollution distribution</td>
</tr>
</tbody>
</table>

**Figure 1.12** The programs in the SPEC CPU2000 benchmark suites. The 11 integer programs (all in C, except one in C++) are used for the CINT2000 measurement, while the 14 floating-point programs (6 in FORTRAN-77, 5 in C, and 3 in FORTRAN-90) are used for the CFP2000 measurement. See [www.spec.org](http://www.spec.org) for more on these benchmarks.
PERFORMANCE REPORTING AND DOCUMENTATION – IMPORTANT!!!!

**Strategy/Approach:** All Aspects of an Experiment Should Be Able to be Reproduced.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model number</td>
<td>Precision WorkStation 410</td>
</tr>
<tr>
<td>CPU</td>
<td>700 MHz, Pentium III</td>
</tr>
<tr>
<td>Number of CPUs</td>
<td>1</td>
</tr>
<tr>
<td>Primary cache</td>
<td>16KBI+16KBD on chip</td>
</tr>
<tr>
<td>Secondary cache</td>
<td>256KB(I+D) on chip</td>
</tr>
<tr>
<td>Other cache</td>
<td>None</td>
</tr>
<tr>
<td>Memory</td>
<td>256 MB ECC PC100 SDRAM</td>
</tr>
<tr>
<td>Disk subsystem</td>
<td>SCSI</td>
</tr>
<tr>
<td>Other hardware</td>
<td>None</td>
</tr>
</tbody>
</table>

**SPEC CINT2000 base tuning parameters/notes/summary of changes:**

+FDO: PASS1=-Qprof_gen PASS2=-Qprof_use

Base tuning: -QxK -Qipo_wp shlW32M.lib +FDO

shlW32M.lib is the SmartHeap library V5.0 from MicroQuill www.microquill.com

Portability flags:
176.gcc: -Dallocal alloca /F10000000 -Op
186.crafy: -DNT_i386
253.perlbmk: -DSPEC_CPU2000_NTOS -DPERLDLL /MT
254.gap: -DSYS_HAS_CALLOC_PROTO -DSYS_HAS_MALLOC_PROTO

Figure 1.14 The machine, software, and baseline tuning parameters for the CINT2000 base report on a Dell Precision WorkStation 410. These data are for the base CINT2000 report. The data are available online at www.spec.org/osg/cpu2000/results/cpu2000.html.
COMPARING/SUMMARIZING COMPUTER PERFORMANCE

• A Tricky Business!! – Be Careful!!
• Consider the following table and what one may infer from it in terms of performance:

\[
\begin{align*}
\text{A is 10 times faster than B for program P1.} \\
\text{B is 10 times faster than A for program P2.} \\
\text{A is 20 times faster than C for program P1.} \\
\text{C is 50 times faster than A for program P2.} \\
\text{B is 2 times faster than C for program P1.} \\
\text{C is 5 times faster than B for program P2.}
\end{align*}
\]

Taken individually, any one of these statements may be of use. Collectively, however, they present a confusing picture—the relative performance of computers A, B, and C is unclear.

<table>
<thead>
<tr>
<th>Program</th>
<th>Computer A</th>
<th>Computer B</th>
<th>Computer C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program P1 (secs)</td>
<td>1</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Program P2 (secs)</td>
<td>1000</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>Total time (secs)</td>
<td>1001</td>
<td>110</td>
<td>40</td>
</tr>
</tbody>
</table>

\text{Figure 1.15 Execution times of two programs on three machines. Data from Figure 1 of Smith [1988].}
How May We Do That?

- “Total Execution Time (CPU time)” Or Some Measure That Tracks Total Execution Time Is The Only Way To Measure Performance.

* We may use “Arithmetic Mean” to measure performance for workloads of n programs –

\[
\text{Arithmetic Mean} = \frac{1}{n} \sum_{i=1}^{n} \text{Time}_i
\]
2. Weighted Execution Time (Weighted Arithmetic Mean) When Unequal Workloads Exist (n = number of programs):

\[
\text{Weighted Arithmetic Mean} = \sum_{i=1}^{n} \text{Weight}_i \times \text{Time}_i
\]
COMPARING/SUMMARIZING COMPUTER PERFORMANCE (Continued)

- Examples of Using Weighted Arithmetic Mean:

<table>
<thead>
<tr>
<th>Programs</th>
<th>Weightings</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>Program P1 (secs)</td>
<td>1.00</td>
</tr>
<tr>
<td>Program P2 (secs)</td>
<td>1000.00</td>
</tr>
</tbody>
</table>

Arithmetic mean: W(1) = 500.50, 55.00, 20.00
Arithmetic mean: W(2) = 91.91, 18.19, 20.00
Arithmetic mean: W(3) = 2.00, 10.09, 20.00

Figure 1.16 Weighted arithmetic mean execution times for three machines (A, B, C) and two programs (P1 and P2) using three weightings (W1, W2, W3). The top table contains the original execution time measurements and the weighting factors, while the bottom table shows the resulting weighted arithmetic means for each weighting. W(1) equally weights the programs, resulting in a mean (row 3) that is the same as the unweighted arithmetic mean. W(2) makes the mix of programs inversely proportional to the execution times on machine B; row 4 shows the arithmetic mean for that weighting. W(3) weights the programs in inverse proportion to the execution times of the two programs on machine A; the arithmetic mean with this weighting is given in the last row. The net effect of the second and third weightings is to “normalize” the weightings to the execution times of programs running on that machine, so that the running time will be spent evenly between each program for that machine. For a set of \( n \) programs each taking \( \text{Time}_i \) on one machine, the equal-time weightings on that machine are

\[
w_j = \frac{1}{\text{Time}_i \times \sum_{i=1}^{n} \left( \frac{1}{\text{Time}_j} \right)}
\]
3. Normalized Execution Time and Geometric Means??

**NOT** the best method of measuring and comparing performance????

Is still sometimes used!!!

\[
\text{Geometric Mean} = \sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}
\]

Where \( \text{Execution time ratio}_i \) is the execution time, normalized to a reference machine, for the \( i \text{th} \) program of a total of \( n \) in the workload.
See Figure 1.17 – Geometric Mean Consistently Shows That Computer “C” Is Best Performer of The 3 Computers No Matter Which Of Three Computers (A,B,C) Execution Times Are Normalized To. Highest Performing Computer Is The One With The Smallest Geometric Mean Rating or Number. Geometric Mean Agrees With Arithmetic Mean On Predicting Which Computer Has Highest Performance.
**GENE AMDAHL’S LAW**

\[
\text{Speedup} = \frac{\text{Performance for Entire Task Using Enhancement}}{\text{Performance for Entire Task Without Using Enhancement}}
\]

\[
\text{Speedup} = \frac{\text{Exec. Time for Entire Task Without Using Enhancement}}{\text{Exec. Time for Entire Task Using Enhancement When Possible}}
\]
AMDAHL’S LAW (Continued)

- Execution Time_{new} = Execution Time_{old} \times ((1 - \text{Fraction}_{\text{enhanced}}) + ((\text{Fraction}_{\text{enhanced}}) / (\text{Speedup}_{\text{enhanced}})))

- Speedup_{overall} = (\text{Execution Time}_{old}) / (\text{Execution Time}_{new})

\[
= \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + ((\text{Fraction}_{\text{enhanced}}) / (\text{Speedup}_{\text{enhanced}}))}
\]
**CPU Time (and all it’s forms!)**

\[ CPU_{Time} = (CPU \text{ clock cycles for a program}) \times (\text{Clock cycle time}) \]

\[ CPU_{Time} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}} \]

\[ CPU_{Time} = IC \times CPI_{Avg} \times \text{Clock cycle time} \]

Where \( CPI = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}} \)

Dependencies (CPU Time):
- IC – ISA and Compiler Functionality (sophistication!!).
- \( CPI_{Avg} \) - Computer Organization/Architecture and ISA.
- Clock cycle time - IC technology and organization.
VERILOG CODING OF CLOCKED SYSTEMS (JK FF EXAMPLE)

module jkffpt(q[0],j[0],k[0],set,clr,clk); // "set" and "clr" are active low.
    // master/slave configuration related to clocking.
    output [0:0]q;
    input [0:0]j,k;
    input set,clr,clk;
    reg a,b;
    reg [0:0]q;

    initial
    q[0]=0;
    always @ (posedge clk or negedge set or negedge clr)
    begin
        a=j[0]; // Latch-on to j and k on posedge of clk.
        b=k[0]; // Use values to determine ff output on negedge of clk.
        if (clr==0) q[0]=0;
        else if (set==0) q[0]=1;
        else
            @(negedge clk)
            case ({a,b})
                2'b00: q[0]=q[0];
                2'b01: q[0]=0;
                2'b10: q[0]=1;
                2'b11: q[0]=~q[0];
            endcase
    end
endmodule
VERILOG CODING OF CLOCKED SYSTEMS (JK FF TESTBENCH)

```
module testjkffpt;
  reg [0:0] js, ks, qt;
  reg j_latch, k_latch, clrs, sets, clks, error;
  wire [0:0] q;
  jkffpt jkff0(q[0], js[0], ks[0], sets, clrs, clks);
initial
  begin
    $monitor($time, "clks=%b js[0]=%b ks[0]=%b sets=%b clrs=%b q[0]=%b qt[0]=%b error=%b\n", clks, js[0], ks[0], sets, clrs, q[0], qt[0], error);
  end
initial
  begin
    qt[0]=0;
    j_latch=0;
    k_latch=0;
  end
initial
  begin
    clks=1'b0;
  always
    #4 clks=~clks;
```
VERILOG CODING OF CLOCKED SYSTEMS (JK FF TESTBENCH)

• Initial  // Continued From Previous Slide

begin:testloop
integer m;
for (m=0;m<16;m=m+1)
begin:mloop
  #1
  js[0]=(m&8)>>3;ks[0]=(m&4)>>2;sets=~((m==3)||(m==9));clrs=~((m==6)||(m==9)||(m==13));
  @ (posedge clks or negedge sets or negedge clrs)
  begin
    j_latch=js[0];
    k_latch=ks[0];
    if(clrs==0) qt[0]=0;
    else if (sets==0) qt[0]=1;
    else if (clks==1)
      @ (negedge clks)
      begin
        if((j_latch==0)&&(k_latch==0)) qt[0]=qt[0];
        if((j_latch==0)&&(k_latch==1)) qt[0]=0;
        if((j_latch==1)&&(k_latch==0)) qt[0]=1;
        if((j_latch==1)&&(k_latch==1)) qt[0]=~qt[0];
      end
      end
  #1 if (q[0]==qt[0]) error=0;
  else error=1;
end
#10 error=1;
end
VERILOG CODING OF CLOCKED SYSTEMS (JK FF TESTBENCH)

• Initial //Continued From Previous Slide

• begin
• #110 $finish;
• end

• endmodule
SOME QUANTATIVE (and common sense!) PRINCIPLES OF COMPUTER DESIGN

1. Make the Common Case (Most Used) Fast!

2. Use Amdahl’s Law (Speedup = ?) and all It’s Forms to determine common case and it’s effect on performance.

3. Likewise, use the \( \text{CPU}_{\text{time}} = IC \times CPI \times \text{Time per Clock Cycle} \) equation to make intelligent informed decisions on hardware and software alternatives when designing a computer or comparing the performance of one computer to that of another.

4. Principle of Locality (90/10 Rule! – A program spends 90% of it’s time in 10% of the instructions or data!).
   - Temporal (Time) Locality. (Data)
   - Spatial Locality (Instructions)

5. Take Advantage of Parallelism- Processor, Instruction, and Detailed Design Levels.
<table>
<thead>
<tr>
<th>Vendor</th>
<th>Model</th>
<th>Processor</th>
<th>Clk. Rate (MHz)/P.</th>
<th>Perf/Pric Rating Int.</th>
<th>Perf/Pric FP.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compaq</td>
<td>Presario 7000</td>
<td>AMD Athlon</td>
<td>1400. ($2,091)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dell</td>
<td>Precision 420</td>
<td>Intel Pent. III</td>
<td>1000. ($3,834)</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Dell</td>
<td>Precision 530</td>
<td>Intel Pent. IV</td>
<td>1700 ($4,175)</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>HP</td>
<td>Workstat. c3600</td>
<td>PA 8600</td>
<td>552 ($12,631)</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>IBM</td>
<td>RS6000 44P/170</td>
<td>IBM III-2</td>
<td>450 ($13,889)</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Sun</td>
<td>Sunblade 100</td>
<td>UltraSPARC II-e</td>
<td>500 ($2,950)</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>Sun</td>
<td>Sunblade 1000</td>
<td>UltraSPARC III</td>
<td>750 ($9,950)</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
PERFORMANCE/PRICE RATINGS FOR SERVERS (TPC-C USED AS BENCHMARK)

1. IBM xSeries 370 c/s  280 Pent. III’s (900 MHz) $15,000,000.
2. IBM pSeries 680 7017-S85  24 IBM RS64-IV (600 MHz) $7,500,000.
3. HP 9000 Enterprise Server  48 HP PA-RISC 8600 (552 MHz) $8,500,000.
4. Compaq Alpha Server GS 320  32 Alpha 21264 (1 GHz) $10,200,000.
5. Fujitsu PRIMEPOWER 20000  48 SPARC64 GP (563 MHz) $9,600,000.
6. IBM iSeries 400 840-2420  24 iSeries 400 Model 840 (450 MHz) $8,400,000.
EMBEDDED PROCESSORS (HARD TO RATE!)
(Consider Several Examples)

- AMD Elan SC520 133MHz 16K/16K Pipelined Single Issue 1600mW $38 (x86 Inst.Set)
- AMD K6-2E+ 500MHz 32K/32K/128K Pipelined (3 Issues/Clk) 9600mW $78 (x86 Inst.Set)
- IBM PowerPC 750CX 500MHz 32K/32K/128K Pipelined (4 Issues/Clk) 6000mW $94
- NEC VR 5432 167MHz 32K/32K Pipelined (2 Issues/Clk) 2088mW $25 (MIPS64 Inst.Set)
- NEC VR 4122 180MHz 32K/16K Pipelined (Single Issue) 700mW $33 (MIPS64 Inst.Set)
FALLACIES!!!!

1. The relative performance of two processors with the same ISA can be judged by clock rate or by the performance of a single benchmark suite. (Pipeline Structure and Memory System are factors.

2. Benchmarks remain valid indefinitely.

3. Peak performance tracks observed performance.

4. The best design for a computer is the one that optimizes the primary objective without considering implementation.

5. Synthetic benchmarks predict performance for real programs.

6. MIPS is an accurate measure for comparing performance among computers. (Can vary inversely with performance!)
PITFALLS!!!


2. Neglecting the cost of software in either evaluating a system or examining cost-performance.

3. Falling prey to Amdahl’s Law. (Improving the performance of a functional unit before measuring it’s utilization.)
ANOTHER VERILOG EXAMPLE!!! (Clocked Systems)

- **TASKS:**
  - Design a clocked synchronous sequential circuit which uses clocked JK Flip-Flops (FFs) and other logic gates to implement the State Table shown on the following slide. Use the State Assignment shown below in your sequential circuit design. The JK FFs are to be positive edge triggered Master/Slave type with asynchronous active-low Clear (clr) and Set (set) inputs. Assume asynchronous operation takes precedence over clocked operation and that the clear operation takes precedence over the set operation. The sequential circuit can be asynchronously “cleared” to state A at anytime or it can be asynchronously set to state C at anytime except for the case when “clr” and “set” are both simultaneously low and in this case the circuit is cleared to state A. Develop a Behavioral and Register Transfer Level (RTL) only Verilog module description of your minimum logic sequential circuit design.
  - Develop an “exhaustive” Verilog testbench which you may use to exhaustively test and verify your Verilog description of the sequential circuit design operating in all modes and under all precedence conditions. Your Verilog test bench should be developed using only behavioral level code. (Hint: Most efficient coding may be achieved via maximum use of “case” type structures and statements.)
  - Use the Simucad Silos III Verilog pre-synthesis simulator to exhaustively test and verify a correct sequential circuit operation.
Verilog Example continued ----

\[ x[1]^k \times x[0]^k \]

<table>
<thead>
<tr>
<th>( y^k )</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B/10</td>
<td>C/11</td>
<td>D/01</td>
<td>A/10</td>
</tr>
<tr>
<td>B</td>
<td>D/00</td>
<td>A/00</td>
<td>C/11</td>
<td>C/10</td>
</tr>
<tr>
<td>C</td>
<td>A/01</td>
<td>B/10</td>
<td>D/01</td>
<td>B/00</td>
</tr>
<tr>
<td>D</td>
<td>C/10</td>
<td>A/11</td>
<td>C/01</td>
<td>B/00</td>
</tr>
</tbody>
</table>

\[ y^{k+1} / z[1]^k / z[0]^k \]
Verilog Example continued ----

- //DESIGN/VERIFICATION PROJECT 6 SOLUTION
- //ECE/CS 280
- //SUMMER 8-WK, 2002

- module synch_seq_ckt_6(x,clk,clr_to_A,set_to_C,z,j_out,k_out,y_out);
  // j_out Is An Output Port Used Only For Easy Observation of j.
  // k_out Is An Output Port Used Only For Easy Observation of k.
  // y_out Is An Output Port Used Only For Easy Observation of y.

- input [1:0]x;
- input clk,clr_to_A,set_to_C; // "clr_to_A" and "set_to_C" Are Connected To The "clr"
  // And "set" Inputs of The FFs.
- output [1:0]z,j_out,k_out,y_out;
- wire [1:0]j,k,z,j_out,k_out,y_out;

- //Instantiation of JK FFs
- jkffpt FF1(y[1],j[1],k[1],set_to_C,clr_to_A,clk);
- jkffpt FF0(y[0],j[0],k[0],set_to_C,clr_to_A,clk);

- // Continious Assignment Statements (The Equivalent of Combinational Loic) Will Be Used
- // To Assign Values To The Output and Next-State Signals/Variables Of The Sequential Circuit.
- // This Is "Dataflow/RTL" Verilog Coding Style.
Verilog Example continued ----

//Circuit Output Equations
• assign z[1]= (~y[0]&~x[1]) | (~y[1]&~y[0]&~x[0]) | (~y[1]&y[0]&x[1]) | (y[1]&~x[1]&x[0]);
• assign z[0]=(x[1]&x[0]) | (~y[0]&x[0]) | (y[1]&y[0]&~x[1]&~x[0]);

//Circuit Next-State Equations
• assign j[1]=(y[0]&~x[0]) | (x[1]&x[0]) | (~y[0]&x[0]);
• assign k[1]=(x[1]&~x[0]) | (~x[1]&x[0]) | (y[0]&~x[0]);
• assign j[0]= (~x[1]&~x[0]) | (~y[1]&~x[1]) | (y[1]&x[1]);
• assign k[0]= (~x[1]&~x[0]) | (~y[1]&~x[1]) | (y[1]&x[1]&x[0]);

//Definition of "j_out", "k_out", and "y_out".
• assign j_out[1]=j[1];
• assign j_out[0]=j[0];
• assign k_out[1]=k[1];
• assign k_out[0]=k[0];
• assign y_out[1]=y[1];
• assign y_out[0]=y[0];

endmodule
Verilog Example continued ----

- module testsynch_seq_ckt_6;
  - reg [1:0]xs,xspt,yt,ynst,zt;           // "ynst" Implies 'y nextstate theoretical'.
  - reg [3:0]mut_ans,th_ans;              // "yt" Implies 'Present State'.
  - reg clks,clr_to_As,set_to_Cs,error;
  - wire [1:0]y_out,z,j_out,k_out;

  - synch_seq_ckt_6 MUT0(xs,clks,clr_to_As,set_to_Cs,z,j_out,k_out,y_out);

  - initial
  - begin
    - $monitor($time,"clks=%b clr_to_As=%b set_to_Cs=%b y_out=%b xs=%b z=%b
      yt=%b zt=%b error=%b\n",
        clks,clr_to_As,set_to_Cs,y_out,xs,z,yt,zt,error);
  - end

  - initial
  - begin
    - yt={1'b0,1'b0};
    - xs={1'b0,1'b0};
    - zt={1'b1,1'b0};
    - clr_to_As=1'b1;
    - set_to_Cs=1'b1;
    - clks=1'b0;
  - end
Verilog Example continued ----

- always
  
  `#4 clks=~clks;

- always @( yt or xs )
  
  begin

  case({yt,xs})

  4'b0000:zt={1'b1,1'b0};
  4'b0001:zt={1'b1,1'b1};
  4'b0010:zt={1'b1,1'b0};
  4'b0011:zt={1'b0,1'b1};
  4'b0100:zt={1'b0,1'b0};
  4'b0101:zt={1'b0,1'b0};
  4'b0110:zt={1'b1,1'b0};
  4'b0111:zt={1'b1,1'b1};
  4'b1000:zt={1'b1,1'b0};
  4'b1001:zt={1'b1,1'b1};
  4'b1010:zt={1'b0,1'b0};
  4'b1011:zt={1'b0,1'b1};
  4'b1100:zt={1'b0,1'b1};
  4'b1101:zt={1'b1,1'b0};
  4'b1110:zt={1'b0,1'b0};
  4'b1111:zt={1'b0,1'b1};

  endcase

end
Verilog Example continued ----

initial
begin: testloops
  integer m;
  for (m=0;m<32;m=m+1)
  begin: mloop
  #1 xs[1]=(m&4)>>2; xs[0]=(m&2)>>1;
  clr_to_As=!(m==9)||(m==18)||(m==24)||(m==29);
  set_to_Cs=!(m==12)||(m==24);
  @(posedge clks or negedge clr_to_As or negedge set_to_Cs)
  begin
    xspt=x;
    if((clr_to_As==0)&&(xs=={1'b0,1'b0}))
      begin
        yt={1'b0,1'b0};zt={1'b1,1'b0};
        end
    else if((clr_to_As==0)&&(xs=={1'b0,1'b1}))
      begin
        yt={1'b0,1'b0};zt={1'b1,1'b1};
        end
    else if((clr_to_As==0)&&(xs=={1'b1,1'b1}))
      begin
        yt={1'b0,1'b0};zt={1'b0,1'b1};
        end
    else if((clr_to_As==0)&&(xs=={1'b1,1'b0}))
      begin
        yt={1'b0,1'b0};zt={1'b1,1'b0};
        end
    else if((set_to_Cs==0)&&(xs=={1'b0,1'b0}))
      begin
        yt={1'b1,1'b1};zt={1'b0,1'b1};
        end
    else if((set_to_Cs==0)&&(xs=={1'b0,1'b1}))
      begin
        yt={1'b1,1'b1};zt={1'b1,1'b0};
        end
    else if((set_to_Cs==0)&&(xs=={1'b1,1'b1}))
      begin
        yt={1'b1,1'b1};zt={1'b0,1'b1};
        end
    else if((set_to_Cs==0)&&(xs=={1'b1,1'b0}))
      begin
        yt={1'b1,1'b1};zt={1'b0,1'b0};
        end
    else
      begin
        yt={1'b1,1'b0};zt={1'b0,1'b0};
      end
  end
Verilog Example continued ---

```verilog
case({yt,xs})
  4'b0000: ynst = {1'b0,1'b1};
  4'b0001: ynst = {1'b1,1'b1};
  4'b0010: ynst = {1'b0,1'b0};
  4'b0011: ynst = {1'b1,1'b0};
  4'b0100: ynst = {1'b0,1'b0};
  4'b0101: ynst = {1'b0,1'b0};
  4'b0110: ynst = {1'b1,1'b1};
  4'b0111: ynst = {1'b1,1'b1};
  4'b1000: ynst = {1'b1,1'b1};
  4'b1001: ynst = {1'b0,1'b0};
  4'b1010: ynst = {1'b0,1'b1};
  4'b1011: ynst = {1'b1,1'b1};
  4'b1100: ynst = {1'b0,1'b0};
  4'b1101: ynst = {1'b0,1'b1};
  4'b1110: ynst = {1'b0,1'b1};
  4'b1111: ynst = {1'b1,1'b0};
endcase

@(negedge clks)
begin
  if ((clr_to_As == 1) && (set_to_Cs == 1))
  begin
    yt = ynst;
  end
end
```

Heath
Verilog Example continued ---

- #1 mut_ans={y_out,z};
- th_ans={yt,zt};

- if(mut_ans==th_ans)error=0;
- else error=1;
- end
- #4 error=1;
- end

- initial
- begin
- #275 $finish;
- end

- endmodule
CHAPTER 2

Instruction Set Principles and Examples
CHAPTER TOPICS

• Instruction Set Architectures (ISAs)
• Classifying ISAs.
• Memory Addressing Modes.
• Memory Addressing Modes for Signal Processing.
• Type and Size of Operands.
• Operands for Media and Signal Processing.
• Operations in the Instruction Set.
• Operations for Media and Signal Processing.
• Instructions for Control Flow.
• Encoding of an Instruction Set.
• The Role of Compilers Related to an Instruction Set.
• An Example ISA – The MIPS ISA.
• Other Example ISAs.
INTRODUCTION

• Will Look at Instruction Set Architectures of 3 Classes of Computers.

1. Desktop Computers – Emphasis is on high-performance execution of arithmetic instructions with integer and floating point data with little concern for program size and power consumption.

2. Servers – Emphasis is on transaction processing in a database, file server, and Web applications environment. Integer arithmetic is mostly used even though most server ISAs include floating point instructions. Some concern for program size and little concern for power consumption.

3. Embedded Processors – Wide performance variation required using both integer and floating point data. Major concerns of power and cost (both need to be minimized), therefore minimal program size is of importance.

• ISA compatibility from one generation computer to another is important. An example is the X86 ISA of Intel. Solution is now sometimes RISC within X86!
Figure 2.1 Operand locations for four instruction set architecture classes. The arrows indicate whether the operand is an input or the result of the ALU operation, or both an input and result. Lighter shades indicate inputs, and the
INSTRUCTION SET ARCHITECTURE CLASSIFICATIONS (Instruction Set View)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Register (register-memory)</th>
<th>Register (load-store)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>Load R1,A</td>
<td>Load R1,A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td>Add R3,R1,B</td>
<td>Load R2,B</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td>Store R3,C</td>
<td>Add R3,R1,R2</td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td>Store R3,C</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.2  The code sequence for $C = A + B$ for four classes of instruction sets. Note that the Add instruction has implicit operands for stack and accumulator architectures, and explicit operands for register architectures. It is assumed that $A$, $B$, and $C$ all belong in memory and that the values of $A$ and $B$ cannot be destroyed. Figure 2.1 shows the Add operation for each class of architecture.
MEMORY ADDRESSES PER ARCHITECTURE TYPE (Affects Instruction Length and Performance)

<table>
<thead>
<tr>
<th>Number of memory addresses</th>
<th>Maximum number of operands allowed</th>
<th>Type of architecture</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>Register-register</td>
<td>Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, Trimedia TM5200</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Register-memory</td>
<td>IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Memory-memory</td>
<td>VAX (also has three-operand formats)</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Memory-memory</td>
<td>VAX (also has two-operand formats)</td>
</tr>
</tbody>
</table>

Figure 2.3 Typical combinations of memory operands and total operands per typical ALU instruction with examples of computers. Computers with no memory reference per ALU instruction are called load-store or register-register computers. Instructions with multiple memory operands per typical ALU instruction are called register-memory or memory-memory, according to whether they have one or more than one memory operand.
### Advantages/Disadvantages of 3 Types of GP Reg. Computers

<table>
<thead>
<tr>
<th>Type</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register-register</td>
<td>Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute (see App. A.).</td>
<td>Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density leads to larger programs.</td>
</tr>
<tr>
<td>(0, 3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register-memory</td>
<td>Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density.</td>
<td>Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location.</td>
</tr>
<tr>
<td>(1, 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory-memory</td>
<td>Most compact. Doesn’t waste registers for temporaries.</td>
<td>Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)</td>
</tr>
<tr>
<td>(2, 2) or (3, 3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.4 Advantages and disadvantages of the three most common types of general-purpose register computers. The notation \((m, n)\) means \(m\) memory operands and \(n\) total operands. In general, computers with fewer alter-
### Value of 3 low-order bits of byte address

<table>
<thead>
<tr>
<th>Width of object</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 byte (byte)</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>2 bytes (half word)</td>
<td>A</td>
<td></td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>2 bytes (half word)</td>
<td></td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>4 bytes (word)</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>4 bytes (word)</td>
<td></td>
<td></td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>4 bytes (word)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>8 bytes (double word)</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>8 bytes (double word)</td>
<td></td>
<td></td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
</tbody>
</table>

**Figure 2.5** Aligned and misaligned addresses of byte, half-word, word, and double-word objects for byte-addressed computers. For each misaligned example some objects require two memory accesses to complete. Every aligned object can always complete in one memory access, as long as the memory is as wide as the object. The figure
### MEMORY ADDRESSING (Addressing Modes—All on DEC VAX 11/780)

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example instruction</th>
<th>Meaning</th>
<th>When used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>Add R4,#3</td>
<td>Regs[R4] ← Regs[R4] + 3</td>
<td>For constants</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R4,100(R1)</td>
<td>Regs[R4] ← Regs[R4] + Mem[100+Reg(R1)]</td>
<td>Accessing local variables (+ simulates register indirect, direct addressing modes).</td>
</tr>
<tr>
<td>Register indirect</td>
<td>Add R4,(R1)</td>
<td>Regs[R4] ← Regs[R4] + Mem[Reg(R1)]</td>
<td>Accessing using a pointer or a computed address.</td>
</tr>
<tr>
<td>Indexed</td>
<td>Add R3,(R1+R2)</td>
<td>Regs[R3] ← Regs[R3] + Mem[Reg(R1)+Reg(R2)]</td>
<td>Sometimes useful in array addressing: R1 = base of array; R2 = index amount.</td>
</tr>
<tr>
<td>Direct or absolute</td>
<td>Add R1,(1001)</td>
<td>Regs[R1] ← Regs[R1] + Mem[1001]</td>
<td>Sometimes useful for accessing static data; address constant may need to be large.</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Add R1,@(R3)</td>
<td>Regs[R1] ← Regs[R1] + Mem[Mem[Reg(R3)]]</td>
<td>If R3 is the address of a pointer p, then mode yields *p.</td>
</tr>
<tr>
<td>Autoincrement</td>
<td>Add R1,(R2)+</td>
<td>Regs[R1] ← Regs[R1] + Mem[Reg(R2)]</td>
<td>Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, d.</td>
</tr>
<tr>
<td>Autodecrement</td>
<td>Add R1,—(R2)</td>
<td>Regs[R2] ← Regs[R2] — d Regs[R1] ← Regs[R1] + Mem[Reg(R2)]</td>
<td>Same use as autoincrement. Autodecrement/increment can also act as push/pop to implement a stack.</td>
</tr>
<tr>
<td>Scaled</td>
<td>Add R1,100(R2)[R3]</td>
<td>Regs[R1] ← Regs[R1] + Mem[100+Reg(R2)] + Regs[R3] * d</td>
<td>Used to index arrays. May be applied to any indexed addressing mode in some computers.</td>
</tr>
</tbody>
</table>

Figure 2.6 Selection of addressing modes with examples, meaning, and usage. In autoincrement/decrement and scaled addressing modes, the variable d designates the size of the data item being accessed (i.e., whether the memory is an array). It is only useful when the elements being
USE OF ADDRESSING MODES (Register Addressing Used Approximately 50% Of Time)

Figure 2.7 Summary of use of memory addressing modes (including immediates). These major addressing modes account for all but a few percent (0% to 3%) of the memory accesses. Register modes, which are not counted, account for one-half of the
DISPLACEMENT ADDRESSING MODE – How Wide Should Displacement Address Field Be?

Figure 2.8 Displacement values are widely distributed. There are both a large number of small values and a fair number of large ones. The wide distribution of displacement values is due to multiple storage areas for variables.
Figure 2.10 The distribution of immediate values. The x-axis shows the number of bits needed to represent the magnitude of an immediate value—0 means the immediate field value was 0. The majority of the immediate values

Heath 72
# FREQUENCY OF ADDRESSING MODES FOR SIGNAL PROCESSING (On TI TMS320C54x DSP)

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Assembly symbol</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>#num</td>
<td>30.02%</td>
</tr>
<tr>
<td>Displacement</td>
<td>ARx(num)</td>
<td>10.82%</td>
</tr>
<tr>
<td>Register indirect</td>
<td>*ARx</td>
<td>17.42%</td>
</tr>
<tr>
<td>Direct</td>
<td>num</td>
<td>11.99%</td>
</tr>
<tr>
<td>Autoincrement, preincrement (increment register <em>before</em> using contents as address)</td>
<td>*+ARx</td>
<td>0%</td>
</tr>
<tr>
<td>Autoincrement, postincrement (increment register <em>after</em> using contents as address)</td>
<td>*ARx+</td>
<td>18.84%</td>
</tr>
<tr>
<td>Autoincrement, preincrement with 16b immediate</td>
<td>*+ARx(num)</td>
<td>0.77%</td>
</tr>
<tr>
<td>Autoincrement, preincrement, with circular addressing</td>
<td>*ARx+%</td>
<td>0.08%</td>
</tr>
<tr>
<td>Autoincrement, postincrement with 16b immediate, with circular addressing</td>
<td>*ARx+(num)%</td>
<td>0%</td>
</tr>
<tr>
<td>Autoincrement, postincrement by contents of AR0</td>
<td>*ARx+0</td>
<td>1.54%</td>
</tr>
<tr>
<td>Autoincrement, postincrement by contents of AR0, with circular addressing</td>
<td>*ARx+0%</td>
<td>2.15%</td>
</tr>
<tr>
<td>Autoincrement, postincrement by contents of AR0, with bit reverse addressing</td>
<td>*ARx+0B</td>
<td>0%</td>
</tr>
<tr>
<td>Autodecrement, postdecrement (decrement register <em>after</em> using contents as address)</td>
<td>*ARx-</td>
<td>6.08%</td>
</tr>
<tr>
<td>Autodecrement, postdecrement, with circular addressing</td>
<td>*ARx-%</td>
<td>0.04%</td>
</tr>
<tr>
<td>Autodecrement, postdecrement by contents of AR0</td>
<td>*ARx-0</td>
<td>0.16%</td>
</tr>
<tr>
<td>Autodecrement, postdecrement by contents of AR0, with circular addressing</td>
<td>*ARx-0%</td>
<td>0.08%</td>
</tr>
<tr>
<td>Autodecrement, postdecrement by contents of AR0, with bit reverse addressing</td>
<td>*ARx-0B</td>
<td>0%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>100.00%</strong></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 2.11 Frequency of addressing modes for TI TMS320C54x DSP. The C54x has 17 data addressing modes, not*
TYPE AND SIZE OF OPERANDS (Fct. Of Applications and Desired Performance)

![Bar chart showing distribution of data accesses by size for benchmark programs.

Floating-point average: 70%, Integer average: 58%, Word (32 bits): 29%, Double word (64 bits): 26%, Half word (16 bits): 5%, Byte (8 bits): 1%.

Figure 2.12 Distribution of data accesses by size for the benchmark programs. The double-word data type is used for double-precision floating point in floating-point programs and for addresses, since the computer uses 64-bit addresses. On a 32-bit address computer the 64-bit addresses would be replaced by 32-bit addresses, and so almost all double-word accesses in integer programs would become single-word accesses.
MEDIA AND SIGNAL PROCESSING OPERANDS

• Graphics Applications – 2D and 3D Images
  • Vertex – 3D Data Type – x,y,z Co-ordinates + A Fourth Co-ordinate, w
    • Encoded Into 32 Bits.

• Signal Processing Applications
  • Majority of Applications Require 16-Bit Operands

  • Fixed-Point DSP Format – 32 Bits, Binary Point to Right of SB.
    • Cheap Floating-Point.

• Image Processing Applications

  • Pixel – 32Bits – R, G, B (8 Bits/Each) + A Fourth 8 Bits, Surface Transparency
### MEDIA AND SIGNAL PROCESSING OPERANDS

<table>
<thead>
<tr>
<th>Generation</th>
<th>Year</th>
<th>Example DSP</th>
<th>Data width</th>
<th>Accumulator width</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1982</td>
<td>TI TMS32010</td>
<td>16 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>2</td>
<td>1987</td>
<td>Motorola DSP56001</td>
<td>24 bits</td>
<td>56 bits</td>
</tr>
<tr>
<td>3</td>
<td>1995</td>
<td>Motorola DSP56301</td>
<td>24 bits</td>
<td>56 bits</td>
</tr>
<tr>
<td>4</td>
<td>1998</td>
<td>TI TMS320C6201</td>
<td>16 bits</td>
<td>40 bits</td>
</tr>
</tbody>
</table>

**Figure 2.13** Four generations of DSPs, their data width, and the width of the registers that reduces round-off error. Section 2.8 explains that multiply-accumulate operations use wide registers to avoid losing precision when accumulating double-length products [Bier 1997].

<table>
<thead>
<tr>
<th>Data size</th>
<th>Memory operand in operation</th>
<th>Memory operand in data transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bits</td>
<td>89.3%</td>
<td>89.0%</td>
</tr>
<tr>
<td>32 bits</td>
<td>10.7%</td>
<td>11.0%</td>
</tr>
</tbody>
</table>

**Figure 2.14** Size of data operands for TMS320C540x DSP. About 90% of operands are 16 bits. This DSP has two 40-bit accumulators. There are no floating-point operations, as is typical of many DSPs, so these data are all fixed-point integers. For details on these measurements, see the caption of Figure 2.11 on page 104.
OPERATIONS TO INCLUDE IN THE INSTRUCTION SET

• Arithmetic and Logical
• Data Transfer
• Control
• System
• Floating Point
• Decimal
• String
• Graphics
# TYPICAL INSTRUCTION USAGE BY RANK (Integer Average)

<table>
<thead>
<tr>
<th>Rank</th>
<th>Instruction</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LOAD</td>
<td>22%</td>
</tr>
<tr>
<td></td>
<td>(Make Fast!!!!) What are involved Fct. Units?</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>CONDITIONAL BRANCH</td>
<td>20%</td>
</tr>
<tr>
<td>3</td>
<td>COMPARE</td>
<td>16%</td>
</tr>
<tr>
<td>4</td>
<td>STORE</td>
<td>12%</td>
</tr>
<tr>
<td>5</td>
<td>ADD</td>
<td>8%</td>
</tr>
<tr>
<td>6</td>
<td>AND</td>
<td>6%</td>
</tr>
<tr>
<td>7</td>
<td>SUB</td>
<td>5%</td>
</tr>
<tr>
<td>8</td>
<td>MOVE REGISTER-REGISTER</td>
<td>4%</td>
</tr>
<tr>
<td>9</td>
<td>CALL</td>
<td>1%</td>
</tr>
<tr>
<td>10</td>
<td>RETURN</td>
<td>1%</td>
</tr>
</tbody>
</table>

Total 96%
### Media and Signal Processing Operations (Many are SIMD Mode)

<table>
<thead>
<tr>
<th>Instruction Category</th>
<th>Alpha MAX</th>
<th>HP PA-RISC MAX2</th>
<th>Intel Pentium MMX</th>
<th>PowerPC Altivec</th>
<th>SPARC VIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add/subtract</td>
<td>4H</td>
<td>8B, 4H, 2W</td>
<td>16B, 8H, 4W</td>
<td>4H, 2W</td>
<td></td>
</tr>
<tr>
<td>Saturating add/sub</td>
<td>4H</td>
<td>8B, 4H</td>
<td>16B, 8H, 4W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply</td>
<td>4H</td>
<td>16B, 8H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compare</td>
<td>8B (&gt;=)</td>
<td>8B, 4H, 2W</td>
<td>16B, 8H, 4W</td>
<td>4H, 2W</td>
<td></td>
</tr>
<tr>
<td>Shift right/left</td>
<td>4H</td>
<td>4H, 2W</td>
<td>16B, 8H, 4W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift right arithmetic</td>
<td>4H</td>
<td>16B, 8H, 4W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply and add</td>
<td></td>
<td>8H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift and add (saturating)</td>
<td>4H</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>And/or/xor</td>
<td>8B, 4H, 2W</td>
<td>8B, 4H, 2W</td>
<td>16B, 8H, 4W</td>
<td>8B, 4H, 2W</td>
<td></td>
</tr>
<tr>
<td>Absolute difference</td>
<td>8B</td>
<td></td>
<td>16B, 8H, 4W</td>
<td>8B</td>
<td></td>
</tr>
<tr>
<td>Maximum/minimum</td>
<td>8B, 4W</td>
<td></td>
<td>16B, 8H, 4W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pack (2n bits → n bits)</td>
<td>2W → 2B, 4H → 4B</td>
<td>2*4H → 8B</td>
<td>4H → 4B, 2W → 2H</td>
<td>4W → 4B, 8H → 8B</td>
<td>2W → 2H, 4H → 4B</td>
</tr>
<tr>
<td>Permute/shuffle</td>
<td>4H</td>
<td></td>
<td></td>
<td>16B, 8H, 4W</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 2.17* Summary of multimedia support for desktop RISCs. Note the diversity of support, with little in common.
MEDIA AND SIGNAL PROCESSING OPERATIONS (Many are SIMD Mode)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>store mem16</td>
<td>32.2%</td>
</tr>
<tr>
<td>load mem16</td>
<td>9.4%</td>
</tr>
<tr>
<td>add mem16</td>
<td>4.8%</td>
</tr>
<tr>
<td>call</td>
<td>3.0%</td>
</tr>
<tr>
<td>push mem16</td>
<td>5.0%</td>
</tr>
<tr>
<td>subtract mem16</td>
<td>4.9%</td>
</tr>
<tr>
<td>multiple-accumulate (MAC) mem16</td>
<td>4.6%</td>
</tr>
<tr>
<td>move mem-mem 16</td>
<td>4.0%</td>
</tr>
<tr>
<td>change status</td>
<td>3.7%</td>
</tr>
<tr>
<td>pop mem16</td>
<td>2.8%</td>
</tr>
<tr>
<td>conditional branch</td>
<td>2.6%</td>
</tr>
<tr>
<td>load mem32</td>
<td>2.5%</td>
</tr>
<tr>
<td>return</td>
<td>2.3%</td>
</tr>
<tr>
<td>store mem32</td>
<td>2.0%</td>
</tr>
<tr>
<td>branch</td>
<td>2.0%</td>
</tr>
<tr>
<td>repeat</td>
<td>2.0%</td>
</tr>
<tr>
<td>multiply</td>
<td>1.9%</td>
</tr>
<tr>
<td>NOP</td>
<td>1.5%</td>
</tr>
<tr>
<td>add mem32</td>
<td>1.3%</td>
</tr>
<tr>
<td>subtract mem32</td>
<td>0.9%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>97.2%</strong></td>
</tr>
</tbody>
</table>

Figure 2.18: Mix of instructions for TMS320C540x DSP. As in Figure 2.16, simple
CONTROL FLOW INSTRUCTIONS

• **JUMP**- Change in control is unconditional.

• **BRANCH**- Change in control is conditional.

• TYPES OF CONTROL FLOW CHANGE:
  1. Conditional Branches
  2. Jumps
  3. Procedure (Subroutine) Calls
  4. Procedure Returns
CONTROL FLOW INSTRUCTION ADDRESSING MODES (How is destination address specified?)

- Destination address is explicitly specified in most cases:
  - PC Relative Addressing Is Most Common Method of Explicitly Specifying a Destination Address. “Offset” to PC is carried in instruction. Provides “Position Independence”. Offset Width??

![Graph showing percentage of distance vs. bits of branch displacement]
## HOW DO WE SPECIFY BRANCH CONDITION? (There are three techniques)

<table>
<thead>
<tr>
<th>Name</th>
<th>Examples</th>
<th>How condition is tested</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition code (CC)</td>
<td>80x86, ARM, PowerPC, SPARC, SuperH</td>
<td>Tests special bits set by ALU operations, possibly under program control.</td>
<td>Sometimes condition is set for free.</td>
<td>CC is extra state. Condition codes constrain the ordering of instructions since they pass information from one instruction to a branch.</td>
</tr>
<tr>
<td>Condition register</td>
<td>Alpha, MIPS</td>
<td>Tests arbitrary register with the result of a comparison.</td>
<td>Simple.</td>
<td>Uses up a register.</td>
</tr>
<tr>
<td>Compare and branch</td>
<td>PA-RISC, VAX</td>
<td>Compare is part of the branch. Often compare is limited to subset.</td>
<td>One instruction rather than two for a branch.</td>
<td>May be too much work per instruction for pipelined execution.</td>
</tr>
</tbody>
</table>

**Figure 2.21** The major methods for evaluating branch conditions, their advantages, and their disadvantages. Although condition codes can be set by ALU operations that are needed for other purposes, measurements on programs show that this rarely happens. The major implementation problems with condition codes arise when the con-
FREQUENCY OF DIFFERENT TYPES OF COMPARES IN CONDITIONAL BRANCH INSTRUCTIONS

- Not equal
  - Floating-point average: 5%
  - Integer average: 2%

- Equal
  - Floating-point average: 16%
  - Integer average: 18%

- Greater than or equal
  - Floating-point average: 0%
  - Integer average: 11%

- Greater than
  - Floating-point average: 0%
  - Integer average: 0%

- Less than or equal
  - Floating-point average: 44%
  - Integer average: 33%

- Less than
  - Floating-point average: 34%
  - Integer average: 35%
PROCEDURE CALLS/RETURNS

1. CONTROL TRANSFER:

   Jmp XY or Call XY – (PC +4) Pushed to Top of Procedure Register or Memory Stack. Address of Instruction with Label XY is “jammed” to PC.

   RTP or RTS (Return from Procedure or Subroutine – Top of Register or Memory Stack is Popped to PC. (Register Indirect Jumps)

2. STATE SAVING (Important Registers, etc.)

   “Caller” Saving

   “Callee” Saving
INSTRUCTION SET ENCODING

ISA PREFERENCES
• Register – Register Architecture.
• Register, Displacement, Immediate, and Register-Indirect Addressing Modes.
• 8, 16, 32, and 64-bit integer data.
• 32 and 64-bit floating-point data.
• PC-relative conditional branches.
• Jump and link instructions for procedure (subroutine) calls.
• Register-indirect jumps for procedure returns.

TO BE BALANCED IN INSTRUCTION SET ENCODING
• Desire for a large number of registers and addressing modes.
• Impact of the size of the register and addressing mode fields on the average instruction size and on program size.
• Instruction lengths that can easily be handled by a pipelined architecture (to gain performance).
## INSTRUCTION ENCODING CHOICES
(Code Density/Performance/Arch.
Simplicity Issues)

<table>
<thead>
<tr>
<th>Operation and no. of operands</th>
<th>Address specifier 1</th>
<th>Address field 1</th>
<th>...</th>
<th>Address specifier</th>
<th>Address field</th>
</tr>
</thead>
</table>

(a) Variable (e.g., VAX, Intel 80x86)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address field 1</th>
<th>Address field 2</th>
<th>Address field 3</th>
</tr>
</thead>
</table>

(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier</th>
<th>Address field</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier 1</th>
<th>Address specifier 2</th>
<th>Address field</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier</th>
<th>Address field 1</th>
<th>Address field 2</th>
</tr>
</thead>
</table>

(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)
Figure 2.24 Compilers typically consist of two to four passes, with more highly optimizing compilers having more passes. This structure maximizes the probability that a program compiled under it will be efficient on the target machine.

Heath 89
COMPILER OBJECTIVES AND COMPUTER ARCHITECTURE

- Compiler Writers Objectives (Goals) - Correctness and Performance

- Compiler Optimizations (Implemented Via Multiple Passes of Program Through Compiler) -
  - High-level.
  - Local – Optimizes code within a Basic Block (straight-line code fragment).
  - Global – Extend local optimizations across Branches and Loops are Optimized.
  - Register allocation – Associates registers with operands. Based on Graph Coloring Algorithms. Computationally complex (therefore, heuristic algorithms are used) and at least 16 registers are needed.
  - Processor dependent – Attempt to compile to a specific computer architecture.

- How Can The Computer Architect Help the Compiler Writer? (Allow compiler writer to develop a compiler that can generate “high-performance” compiled code!)
  - Provide regularity – Orthogonality between op codes, addressing modes, and data types.
  - Provide primitives and not solutions.
  - Simplify trade-offs among alternatives.
  - Provide instructions that bind the quantaties known at compile time as constants.
MIPS ISA (64-Bit Load/Store RISC Architecture)

- **Goals:**
  - Use of general-purpose registers within a load/store architecture.
  - Support register, displacement (address offset size of 12-16 bits), immediate, and register indirect addressing.
  - Support data sizes and types of: 8-, 16-, 32-, and 64-bit integers and 64-bit IEEE 754 format FP numbers.
  - Support simple load, store, add, subtract, move register, and shift instructions.
  - Support compare equal, compare not equal, compare less, branch (with a PC-relative address at least 8-bits long), jump, call, and return instructions.
  - Use fixed instruction encoding to enhance performance.
  - Provide at least 16 general-purpose registers.

- **Resulting MIPS Register Sets:**
  - 32 64-bit general-purpose registers (GPRs) named R0, R1, ---, R31 (Integer Registers).
  - 32 64-bit floating-point registers (FPRs) named F0, F1, ---, F31. They can hold 32 single precision values or 32 double precision values. Some operations can be performed on two single-precision operands in the same register.
  - The value of R0 is always ‘0’.
MIPS ISA (Continued)

- **Data Types:**
  - 8-Bit Bytes.
  - 16-Bit Half Words.
  - 32-Bit Words.
  - 64-Bit Double Words for Integer Data.
  - 32-Bit Single Precision Floating Point.
  - 64-Bit Double Precision Floating Point.

**MIPS Operations** work on 64-bit integers and 32- and 64-bit floating point.

- **Data Transfer Addressing Modes:**
  - Immediate (16-bit field). Encoded into op-code.
  - Displacement (16-bit field). Encoded into op-code.
  
  Register Indirect addressing achieved when displacement field is zero (0).
  
  Absolute addressing with a 16-bit field is achieved by using R0 as the base register.
MIPS INSTRUCTION FORMATS

I-type Instruction

- Opcode
- rs
- rt
- Immediate

Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt = rs op immediate)

Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd = 0, rt = destination, immediate = 0)

R-type Instruction

- Opcode
- rs
- rt
- rd
- shamt
- funct

Register-register ALU operations: rd = rs funct rt
Function encodes the data path operation: Add, Sub, ...
Read/write special registers and moves

J-type Instruction

- Opcode
- Immediate

Jump and jump and link
Trap and return from exception
**EXAMPLE MIPS INSTRUCTIONS**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>LD R1,30(R2)</code></td>
<td>Load double word</td>
<td>$\text{Regs}[R1] \leftarrow 64 \text{ Mem}[30+\text{Regs}[R2]]$</td>
</tr>
<tr>
<td><code>LD R1,1000(R0)</code></td>
<td>Load double word</td>
<td>$\text{Regs}[R1] \leftarrow 64 \text{ Mem}[1000+0]$</td>
</tr>
<tr>
<td><code>LW R1,60(R2)</code></td>
<td>Load word</td>
<td>$\text{Regs}[R1] \leftarrow 64 \text{ Mem}[60+\text{Regs}[R2]]^{32} # \text{ Mem}[60+\text{Regs}[R2]]$</td>
</tr>
<tr>
<td><code>LB R1,40(R3)</code></td>
<td>Load byte</td>
<td>$\text{Regs}[R1] \leftarrow 64 \text{ Mem}[40+\text{Regs}[R3]]^{56} # \text{ Mem}[40+\text{Regs}[R3]]$</td>
</tr>
<tr>
<td><code>LBU R1,40(R3)</code></td>
<td>Load byte unsigned</td>
<td>$\text{Regs}[R1] \leftarrow 64 \text{ Mem}[40+\text{Regs}[R3]]^{56} # \text{ Mem}[40+\text{Regs}[R3]]$</td>
</tr>
<tr>
<td><code>LH R1,40(R3)</code></td>
<td>Load half word</td>
<td>$\text{Regs}[R1] \leftarrow 64 \text{ Mem}[40+\text{Regs}[R3]]^{48} # \text{ Mem}[40+\text{Regs}[R3]] # \text{ Mem}[41+\text{Regs}[R3]]$</td>
</tr>
<tr>
<td><code>L.S F0,50(R3)</code></td>
<td>Load FP single</td>
<td>$\text{Regs}[F0] \leftarrow 64 \text{ Mem}[50+\text{Regs}[R3]]^{32} # 0^{32}$</td>
</tr>
<tr>
<td><code>L.D F0,50(R2)</code></td>
<td>Load FP double</td>
<td>$\text{Regs}[F0] \leftarrow 64 \text{ Mem}[50+\text{Regs}[R2]]$</td>
</tr>
<tr>
<td><code>SD R3,500(R4)</code></td>
<td>Store double word</td>
<td>$\text{Mem}[500+\text{Regs}[R4]] \leftarrow 64 \text{ Regs}[R3]$</td>
</tr>
<tr>
<td><code>SW R3,500(R4)</code></td>
<td>Store word</td>
<td>$\text{Mem}[500+\text{Regs}[R4]] \leftarrow 32 \text{ Regs}[R3]$</td>
</tr>
<tr>
<td><code>S.S F0,40(R3)</code></td>
<td>Store FP single</td>
<td>$\text{Mem}[40+\text{Regs}[R3]] \leftarrow 32 \text{ Regs}[F0]_{0..31}$</td>
</tr>
<tr>
<td><code>S.D F0,40(R3)</code></td>
<td>Store FP double</td>
<td>$\text{Mem}[40+\text{Regs}[R3]] \leftarrow 64 \text{ Regs}[F0]$</td>
</tr>
<tr>
<td><code>SH R3,502(R2)</code></td>
<td>Store half</td>
<td>$\text{Mem}[502+\text{Regs}[R2]] \leftarrow 16 \text{ Regs}[R3]_{48..63}$</td>
</tr>
<tr>
<td><code>SB R2,41(R3)</code></td>
<td>Store byte</td>
<td>$\text{Mem}[41+\text{Regs}[R3]] \leftarrow 8 \text{ Regs}[R2]_{56..63}$</td>
</tr>
</tbody>
</table>

Figure 2.28 The load and store instructions in MIPS. All use a single addressing mode and require that the memory be aligned. Of course, both loads and stores are available for all the data types shown.
### EXAMPLE MIPS INSTRUCTIONS

<table>
<thead>
<tr>
<th>Example Instruction</th>
<th>Instruction name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDU R1, R2, R3</td>
<td>Add unsigned</td>
<td>$\text{Reg}[R1] \leftarrow \text{Reg}[R2] + \text{Reg}[R3]$</td>
</tr>
<tr>
<td>ADDIU R1, R2, #3</td>
<td>Add immediate unsigned</td>
<td>$\text{Reg}[R1] \leftarrow \text{Reg}[R2] + 3$</td>
</tr>
<tr>
<td>LUI R1, #42</td>
<td>Load upper immediate</td>
<td>$\text{Reg}[R1] \leftarrow 0^{32} #42 #0^{16}$</td>
</tr>
<tr>
<td>DSLL R1, R2, #5</td>
<td>Shift left logical</td>
<td>$\text{Reg}[R1] \leftarrow \text{Reg}[R2] \ll 5$</td>
</tr>
<tr>
<td>DSLT R1, R2, R3</td>
<td>Set less than</td>
<td>( 1^f ) if (\text{Reg}[R2] &lt; \text{Reg}[R3]) \else \text{Reg}[R1] \leftarrow 0 )</td>
</tr>
</tbody>
</table>

Figure 2.29 Examples of arithmetic/logical instructions on MIPS, both with and without immediates.
### EXAMPLE MIPS INSTRUCTIONS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instruction name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>J name</td>
<td>Jump</td>
<td>$PC_{31\ldots 0} \leftarrow \text{name}$</td>
</tr>
<tr>
<td>JAL name</td>
<td>Jump and link</td>
<td>$\text{Regs[R31]} \leftarrow PC+4; \ PC_{31\ldots 0} \leftarrow \text{name}; ((PC+4)-2^{17}) \leq \text{name} \leq ((PC+4)+2^{17})$</td>
</tr>
<tr>
<td>JALR R2</td>
<td>Jump and link register</td>
<td>$\text{Regs[R31]} \leftarrow PC+4; \ PC \leftarrow \text{Regs[R2]}$</td>
</tr>
<tr>
<td>JR R3</td>
<td>Jump register</td>
<td>$PC \leftarrow \text{Regs[R3]}$</td>
</tr>
<tr>
<td>BEQZ R4, name</td>
<td>Branch equal zero</td>
<td>$1^f \ (\text{Regs[R4]} == 0) \ PC \leftarrow \text{name}; ((PC+4)-2^{17}) \leq \text{name} \leq ((PC+4)+2^{17})$</td>
</tr>
<tr>
<td>BNE R3, R4, name</td>
<td>Branch not equal zero</td>
<td>$1^f \ (\text{Regs[R3]} \neq \text{Regs[R4]}) \ PC \leftarrow \text{name}; ((PC+4)-2^{17}) \leq \text{name} \leq ((PC+4)+2^{17})$</td>
</tr>
<tr>
<td>MOVZ R1, R2, R3</td>
<td>Conditional move if zero</td>
<td>$1^f \ (\text{Regs[R3]} == 0) \ \text{Regs[R1]} \leftarrow \text{Regs[R2]}$</td>
</tr>
</tbody>
</table>

Figure 2.30: Typical control flow instructions in MIPS. All control instructions, except jumps to an address in a register, are PC-relative. Note that the branch distances are...