CSE 141 – Computer Architecture
Summer Session 1 2004

Lecture 2
Performance, ALU

Pramod V. Argade
CSE141: Introduction to Computer Architecture

Instructor:  Pramod V. Argade (p2argade@cs.ucsd.edu)
            Office Hours:
            Tue.  7:30 - 8:30 PM (AP&M 4141)
            Wed. 4:30 - 5:30 PM (AP&M 4141)

TA:
    Anjum Gupta (a3gupta@cs.ucsd.edu)
    Office Hour: Mon/Wed 12 - 1 PM
    Chengmo Yang (c5yang@cs.ucsd.edu)
    Office Hour: Mon 2 - 3 PM, Thu 3:30 – 4:30 PM

Lecture:  Mon/Wed. 6 - 8:50 PM, Center 109

Homework: Computer Organization & Design
            Authors: Patterson and Hennessy

Web-page:  http://www.cse.ucsd.edu/classes/su04/cse141
Announcements

- Two sections for 141:
  - Friday 11 - 11:50 PM Center 105
  - Friday 2 - 2:50 PM CSB 004

- Reading Assignment
  - Chapter 2: Performance
  - Chapter 4: Arithmetic for Computers, Sec. 4.6 - 4.11

- Homework 2: Due Wed., July 7 in class
  2.1 through 2.5, 2.18 through 2.21
  4.9, 4.12, 4.14, 4.22, 4.23, 4.44

- Quiz
  When: Wed., July 7, First 10 minutes of the class
  Topic: Performance, Chapter 2  Need: Paper, pen, calculator

- CSE 141 Lab Assignment has been posted on the web page
  [http://www.cse.ucsd.edu/classes/su04/cse141L](http://www.cse.ucsd.edu/classes/su04/cse141L)
<table>
<thead>
<tr>
<th>Lecture #</th>
<th>Date</th>
<th>Time</th>
<th>Room</th>
<th>Topic</th>
<th>Quiz topic</th>
<th>Homework Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mon. 6/28</td>
<td>6 - 8:50 PM</td>
<td>Center 109</td>
<td>Introduction, Ch. 1 ISA, Ch. 3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>Wed. 6/30</td>
<td>6 - 8:50 PM</td>
<td>Center 109</td>
<td>Performance, Ch. 2 Arithmetic, Ch. 4</td>
<td>ISA Ch. 3</td>
<td>#1</td>
</tr>
<tr>
<td></td>
<td>Mon. 7/5</td>
<td></td>
<td></td>
<td>No Class</td>
<td>July 4th Holiday</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>Wed. 7/7</td>
<td>6 - 8:50 PM</td>
<td>Center 109</td>
<td>Arithmetic, Ch. 4 Cont. Single-cycle CPU Ch. 5</td>
<td>Performance Ch. 2</td>
<td>#2</td>
</tr>
<tr>
<td>4</td>
<td>Mon. 7/12</td>
<td>6 - 8:50 PM</td>
<td>Center 109</td>
<td>Single-cycle CPU Ch. 5 Cont. Multi-cycle CPU Ch. 5</td>
<td>Arithmetic, Ch. 4</td>
<td>#3</td>
</tr>
<tr>
<td>5</td>
<td>Tue. 7/13</td>
<td>7:30 - 8:50 PM</td>
<td>Center 109</td>
<td>Multi-cycle CPU Ch. 5 Cont. (July 5th make up class)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>Wed. 7/14</td>
<td>6 - 8:50 PM</td>
<td>Center 109</td>
<td>Single and Multicycle CPU Examples and Review for Midterm</td>
<td>Single-cycle CPU Ch. 5</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>Mon. 7/19</td>
<td>6 - 8:50 PM</td>
<td>Center 109</td>
<td>Mid-term Exam Exceptions</td>
<td>-</td>
<td>#4</td>
</tr>
<tr>
<td>8</td>
<td>Tue. 7/20</td>
<td>7:30 - 8:50 PM</td>
<td>Center 109</td>
<td>Pipelining Ch. 6 (July 5th make up class)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>Wed. 7/21</td>
<td>6 - 8:50 PM</td>
<td>Center 109</td>
<td>Hazards, Ch. 6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>Mon. 7/26</td>
<td>6 - 8:50 PM</td>
<td>Center 109</td>
<td>Memory Hierarchy &amp; Caches Ch. 7</td>
<td>Hazards Ch. 6</td>
<td>#5</td>
</tr>
<tr>
<td>11</td>
<td>Wed. 7/28</td>
<td>6 - 8:50 PM</td>
<td>Center 109</td>
<td>Virtual Memory, Ch. 7 Course Review</td>
<td>Cache Ch. 7</td>
<td>#6</td>
</tr>
<tr>
<td>12</td>
<td>Sat. 7/31</td>
<td>7 - 10 PM</td>
<td>Center 109</td>
<td>Final Exam</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Performance Assessment

<table>
<thead>
<tr>
<th>Plane</th>
<th>DC to Paris time</th>
<th>Speed</th>
<th>Passengers</th>
<th>Throughput (p x mph)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Boeing 747</strong></td>
<td>6.5 hours</td>
<td>610 mph</td>
<td>470</td>
<td>286,700</td>
</tr>
<tr>
<td></td>
<td>3 hours</td>
<td>1350 mph</td>
<td>132</td>
<td>178,200</td>
</tr>
</tbody>
</table>

- Example definitions of performance
  - Passengers * MPH
  - Cruising speed
  - Passenger capacity
  - (Passenger * MPH)/$
  ⇐ need more data!

- Clearly formulate your definition of performance
Why worry about performance?

- Learn to measure, report, and summarize
- Make intelligent choices
- See through the marketing hype
- As a designer or purchaser, assess:
  - which system has best performance?
  - which system has lowest cost?
  - which system has highest performance/cost?
- Formulate metrics for performance measurement
  - How to report relative performance?
- Understand impact of architectural choices on performance
Computer Performance: TIME, TIME, TIME

- **Response Time (latency)**
  - How long does it take for my job to run?
  - How long does it take to execute a job?
  - How long must I wait for the database query?

- **Throughput**
  - How many jobs can the machine run at once?
  - What is the average execution rate?
  - How much work is getting done?

- *If we upgrade a machine with a new processor what do we decrease?*

- *If we add a new machine to the lab what do we increase?*
Execution Time

% time program
... program results ...
90.7u 12.9s 2:39 65%
%

- Elapsed Time (2:39)
  - Counts everything (*disk and memory accesses, I/O, etc.*)
  - A useful number, but often not good for comparison purposes

- Total CPU time (103.6 s)
  - Doesn't count I/O or time spent running other programs
  - Comprised of system time (12.9 s), and user time (90.7 s)

- Our focus: user CPU time
  - Time spent executing the lines of code that are "in" our program
A Definition of Performance

• For some program running on machine X,
  \[ \text{Performance}_X = \frac{1}{\text{Execution time}_X} \]

• “Machine X is n times faster than Y”
  \[ n = \frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} \]

• Problem:
  Execution time: Machine A: 12 seconds, B: 20 seconds
  – \( A/B = .6 \), so A is 40% faster, or 1.4X faster, or B is 40% slower
  – \( B/A = 1.67 \), so A is 67% faster, or 1.67X faster, or B is 67% slower

• Need a precise definition
  ⇒ “X is n times faster than Y”, \( n > 1 \)
  – A is 1.67 times faster than B
Clock Cycles

● Operation of conventional computer is controlled by Clock “ticks”

![Clock Cycles Diagram]

● Clock rate (frequency) = cycles per second (Hertz)
  - MHz = Million cycles per second
  - GHz = Giga (Billion) cycles per second

● Cycle time = time between ticks = seconds per cycle
  - A 2 GHz clock => Cycle time = 1/(2x10^9) s = 0.5 nanoseconds

● Instead of reporting execution time in seconds, we often use clock cycles

\[
\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}
\]
How many cycles are required for a program?

- Assumption: # of cycles = # of instructions

- Assumption is incorrect
  - Typically, different instructions take different number of clock cycles
    - Integer Multiply instructions are multi-cycle
    - Floating point instructions are multi-cycle
Now that we understand cycles

- A given program will require
  - some number of instructions (machine instructions)
  - some number of cycles
  - some number of seconds

- We have a vocabulary that relates these quantities:
  - cycle time (seconds per cycle)
  - clock rate (cycles per second)
  - CPI (cycles per instruction)

  *a floating point intensive application might have a higher CPI*
CPI Example 2.1

- Suppose we have two implementations of the same instruction set architecture (ISA).

For some program,
- Machine A has a clock cycle time of 10 ns. and a CPI of 2.0
- Machine B has a clock cycle time of 20 ns. and a CPI of 1.2

What machine is faster for this program, and by how much?
A compiler designer is trying to decide between two code sequences for a particular machine with 3 instruction classes.

- CPI: Class A = 1, Class B = 2, and Class C = 3
- Code sequence X has 5 instructions:
  - 2 of A, 1 of B, and 2 of C
- Code sequence Y has 6 instructions:
  - 4 of A, 1 of B, and 1 of C.

Which sequence will be faster? How much?
Execution time: Contributing Factors

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

<table>
<thead>
<tr>
<th></th>
<th>Instruction Count</th>
<th>CPI</th>
<th>Clock Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td>X</td>
<td>(X)</td>
<td></td>
</tr>
<tr>
<td>ISA</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Organization</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Technology</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

- Improve performance => reduce execution time
  - Reduce instruction count (Programmer, Compiler)
  - Reduce cycles per instruction (ISA, Machine designer)
  - Reduce clock cycle time (Hardware designer, Process engineer)
Performance

- Performance is determined by program execution time
- Do any of the other variables equal performance?
  - # of cycles to execute program?
  - # of instructions in program?
  - # of cycles per second?
  - average # of cycles per instruction?
  - average # of instructions per second?

- Common pitfall: thinking one of the variables is indicative of performance when it really isn’t.
## Performance Variation

\[
\text{CPU Execution Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}
\]

<table>
<thead>
<tr>
<th></th>
<th>Number of instructions</th>
<th>CPI</th>
<th>Clock Cycle Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same machine different programs</td>
<td>different</td>
<td>similar</td>
<td>same</td>
</tr>
<tr>
<td>same programs, different machines, same ISA</td>
<td>same</td>
<td>different</td>
<td>different</td>
</tr>
<tr>
<td>Same programs, different machines</td>
<td>somewhat different</td>
<td>different</td>
<td>different</td>
</tr>
</tbody>
</table>
Amdahl’s Law

- The impact of a performance improvement is limited by the percent of execution time affected by the improvement

\[
\text{Execution time after improvement} = \frac{\text{Execution Time Affected}}{\text{Amount of Improvement}} + \text{Execution Time Unaffected}
\]

- Make the common case fast!
- Amdahl’s law sets limit on how much improvement can be made
Amdahl’s Law: Example 2.3

- Example: A program runs in 100 seconds on a machine, with multiply responsible for 80 seconds of this time. How much do we have to improve the speed of multiplication if we want the program to run 4 times faster?

- Is it possible to get the program to run 5 times faster?
Metrics of Performance

Each metric has a place and a purpose, and each can be misused.
Benchmarks

- Performance best determined by running a real application
  - Use programs typical of expected workload
  - Or, typical of expected class of applications
e.g., compilers/editors, scientific applications, graphics, etc.

- Small benchmarks
  - nice for architects and designers
  - easy to standardize
  - can be abused

- SPEC (System Performance Evaluation Cooperative)
  - companies have agreed on a set of real program and inputs
  - can still be abused (Intel’s “other” bug)
  - valuable indicator of performance (and compiler technology)
SPEC ‘89

- Compiler “enhancements” and performance
### SPEC ’95

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>go</td>
<td>Artificial intelligence; plays the game of Go</td>
</tr>
<tr>
<td>m88ksim</td>
<td>Motorola 88k chip simulator; runs test program</td>
</tr>
<tr>
<td>gcc</td>
<td>The Gnu C compiler generating SPARC code</td>
</tr>
<tr>
<td>compress</td>
<td>Compresses and decompresses file in memory</td>
</tr>
<tr>
<td>li</td>
<td>Lisp interpreter</td>
</tr>
<tr>
<td>ijpeg</td>
<td>Graphic compression and decompression</td>
</tr>
<tr>
<td>perl</td>
<td>Manipulates strings and prime numbers in the special-purpose programming language Perl</td>
</tr>
<tr>
<td>vortex</td>
<td>A database program</td>
</tr>
<tr>
<td>tomcatv</td>
<td>A mesh generation program</td>
</tr>
<tr>
<td>swim</td>
<td>Shallow water model with 513 x 513 grid</td>
</tr>
<tr>
<td>su2cor</td>
<td>Quantum physics; Monte Carlo simulation</td>
</tr>
<tr>
<td>hydro2d</td>
<td>Astrophysics; Hydrodynamic Naiver Stokes equations</td>
</tr>
<tr>
<td>mgrid</td>
<td>Multigrid solver in 3-D potential field</td>
</tr>
<tr>
<td>applu</td>
<td>Parabolic/elliptic partial differential equations</td>
</tr>
<tr>
<td>trub3d</td>
<td>Simulates isotropic, homogeneous turbulence in a cube</td>
</tr>
<tr>
<td>apsi</td>
<td>Solves problems regarding temperature, wind velocity, and distribution of pollutant</td>
</tr>
<tr>
<td>fppppp</td>
<td>Quantum chemistry</td>
</tr>
<tr>
<td>wave5</td>
<td>Plasma physics; electromagnetic particle simulation</td>
</tr>
</tbody>
</table>
Does doubling the clock rate double the performance?
Can a machine with a slower clock rate have better performance?
MIPS, MFLOPS etc.

- MIPS - million instructions per second

\[
\text{MIPS} = \frac{\text{number of instructions executed in program}}{\text{execution time in seconds} \times 10^6} \times \frac{\text{Clock rate}}{\text{CPI} \times 10^6}
\]

- MFLOPS - million floating point operations per second

\[
\text{MFLOPS} = \frac{\text{number of floating point operations executed in program}}{\text{execution time in seconds} \times 10^6}
\]

- Hard to relate MIPS/MFLOPS with execution time

\[
\text{CPU Execution Time} = \frac{\text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}}{\text{Execution Time}}
\]

- Highly program-dependent metrics
- Deceptive (See pages 78 - 79 in the text)
  - MIPS would be higher for a program using simple instructions (e.g. a loop of NOPs!)
RISC Processor: Example 2.4

<table>
<thead>
<tr>
<th>Op</th>
<th>Freq</th>
<th>Cycles</th>
<th>CPI(i)</th>
<th>% Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• What is average CPI?
• What percentage of time is spent in each instruction class?
• How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?

• How does this compare with using branch prediction to shave a cycle off the branch time?

• What if two ALU instructions could be executed at once?
Performance assessment is tricky

- Performance is specific to particular program(s)
  - Total execution time is a consistent summary of performance

- For a given architecture performance increases come from:
  - Increases in clock rate (without adverse CPI affects)
  - Improvements in processor organization that lower CPI
  - Compiler enhancements that lower CPI and/or instruction count

- Pitfall: expecting improvement in one aspect of a machine’s performance to affect the total performance

- You should not always believe everything you read! Read carefully!
Announcements

- **Two sections for 141:**
  - Friday 11 - 11:50 PM Center 109
  - Friday 2 - 2:50 PM CSB 105

- **Reading Assignment**
  - Chapter 4: Arithmetic for Computers, Sec. 4.6 - 4.11
  - Chapter 5: The Processor: Datapath and Control, Sec.

- **Homework 2:** Due Wed., July 7 in class
  - 2.1 through 2.5, 2.18 through 2.21
  - 4.9, 4.12, 4.14, 4.22, 4.23, 4.44

- **Quiz**

  **When:** Wed., July 7, First 10 minutes of the class
  **Topic:** Performance, Chapter 2  **Need:** Paper, pen, calculator

- **CSE 141 Lab Assignment has been posted on the web page**
  
  [http://www.cse.ucsd.edu/classes/su04/cse141L](http://www.cse.ucsd.edu/classes/su04/cse141L)
Computer Arithmetic and ALU Design
Bits everywhere… What do they mean?

bits (011011011100010 ....01)

instruction

R-format I-format ...

data

number

text chars ..............

integer

floating point

signed unsigned

single precision double precision

... ... ... ...
Questions About Numbers

• How do you represent
  – negative numbers?
  – fractions?
  – really large numbers?
  – really small numbers?

• How do you
  – do arithmetic?
  – identify errors (e.g. overflow)?

• What is an ALU and what does it look like?
  – ALU=arithmetic logic unit
Introduction to Binary Numbers

Consider a 4-bit binary number

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>7</td>
<td>0111</td>
</tr>
</tbody>
</table>

Examples of binary arithmetic:

\[ 3 + 2 = 5 \]

\[
\begin{array}{cccc}
0 & 0 & 1 & 1 \\
+ & 0 & 0 & 1 & 0 \\
\hline
0 & 1 & 0 & 1 \\
\end{array}
\]

\[ 3 + 3 = 6 \]

\[
\begin{array}{cccc}
0 & 0 & 1 & 1 \\
+ & 0 & 0 & 1 & 1 \\
\hline
0 & 1 & 1 & 0 \\
\end{array}
\]
Negative Numbers?

- We would like a number system that provides
  - obvious representation of positive and negative integers
  - uses the same adder for addition and subtraction
  - single value of 0
  - equal coverage of positive and negative numbers
  - easy detection of sign
  - easy negation
Possible Representations

- **Sign Magnitude:**
  - 000 = +0
  - 001 = +1
  - 010 = +2
  - 011 = +3
  - 100 = -0
  - 101 = -1
  - 110 = -2
  - 111 = -3

- **One's Complement:**
  - 000 = +0
  - 001 = +1
  - 010 = +2
  - 011 = +3
  - 100 = -3
  - 101 = -2
  - 110 = -1
  - 111 = -0

- **Two's Complement:**
  - 000 = +0
  - 001 = +1
  - 010 = +2
  - 011 = +3
  - 100 = -4
  - 101 = -3
  - 110 = -2
  - 111 = -1

- 2’s complement representation of negative numbers
  - Take the bitwise inverse and add 1

- **Issues:** balance, number of zeros, ease for HW/SW

- **Which one is best? Why?**
Two’s Complement Arithmetic

<table>
<thead>
<tr>
<th>Decimal</th>
<th>2’s Complement Binary</th>
<th>Decimal</th>
<th>2’s Complement Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>-1</td>
<td>1111</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>-2</td>
<td>1110</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>-3</td>
<td>1101</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>-5</td>
<td>1011</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>-6</td>
<td>1010</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>-7</td>
<td>1001</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>-8</td>
<td>1000</td>
</tr>
</tbody>
</table>

- Examples: \( 7 - 6 = 7 + (-6) = 1 \)

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 \\
\hline
+ & 1 & 0 & 1 & 0 \\
\hline
0 & 0 & 0 & 1
\end{array}
\]

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 \\
\hline
+ & 1 & 0 & 1 & 1 \\
\hline
1 & 1 & 1 & 0
\end{array}
\]

\( 3 - 5 = 3 + (-5) = -2 \)
Immediate Field in I-Format

- For LW, SW, BEQ, BNE, ADDI, ADDIU, SLT, SLTIU
  - 16-bit Immediate field is signed
    - Copy sign bit to all upper 16 bits
  - It is sign extended to 32 bits before use
    - Sign extension:
      - 0010 in 4 bits is the same as 0000 0010 in 8 bits (+2)
      - 1110 in 4 bits is the same as 1111 1110 in 8 bits (-2)
  - Why is there no SUBI instruction for MIPS?

- For ANDI, ORI
  - 16-bit Immediate field is zero-extended to 32 bits before use
    - Copy zero to all upper 16 bits
  - Why?
Arithmetic -- The heart of instruction execution

Instruction Fetch

Instruction Decode

Operand Fetch

Execute

Result Store

Next Instruction

ALU

operation

result

a

b

32

32

32
Foundation of ICs: Field Effect Transistor (FET)

- N-FET: Conducts when Gate is high
- P-FET: Conducts when Gate is low

- CMOS Inverter
  - Input high, output low
  - Input low, output high
  - Power consumed only during transition

- Current processing technology 0.09 μ, next 0.065 μ
Some basics of digital logic

1. AND gate \((c = a \cdot b)\)

2. OR gate \((c = a + b)\)

3. Inverter \((c = \overline{a})\)

4. Multiplexor
   (if \(d = 0\), \(c = a\); else \(c = b\))
A One-bit Full Adder

- This is also called a (3, 2) adder
- Truth Table:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>A  B  CarryIn</td>
<td>CarryOut</td>
<td>Sum</td>
</tr>
<tr>
<td>0  0  0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0  0  1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0  1  0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0  1  1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1  0  0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1  0  1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1  1  0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1  1  1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Logic Equation for CarryOut

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>CarryIn</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

# Logic Equation for Sum

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A &amp; B &amp; CarryIn</td>
<td>CarryOut &amp; Sum</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>CarryIn</th>
<th>CarryOut</th>
<th>Sum</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 + 0 + 0 = 00</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0 + 0 + 1 = 01</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0 + 1 + 0 = 01</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0 + 1 + 1 = 10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1 + 0 + 0 = 01</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1 + 0 + 1 = 10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1 + 1 + 0 = 10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1 + 1 + 1 = 11</td>
</tr>
</tbody>
</table>

Sum = (!A & !B & CarryIn) | (!A & B & !CarryIn) | (A & !B & !CarryIn) | (A & B & CarryIn)
1-bit ALU

- **ALU Control Lines (ALUop)** Function
  - 000 And
  - 001 Or

- **ALU Control Lines (ALUop)** Function
  - 000 And
  - 001 Or
  - 010 Add

But how do we make the adder?
1-bit ALU

- Implements functions:
  - AND
  - OR
  - ADD
- What about SUB and SLT?
- Ripple carry delay is large...

32-bit ALU

The 32-bit ALU
32-bit ALU: Subtraction

- Keep in mind the following:
  - \((A - B)\) is the same as: \(A + (-B)\)
- Bit-wise inverse of \(B\) is \(\overline{B}\):
  - \(A - B = A + (-B) = A + (\overline{B} + 1) = A + \overline{B} + 1\)

- Binvert provides the negation
- How about “+1”?
- For SUB, for LSB:
  - set Binvert = 1
  - CarryIn = 1
Detecting Overflow

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- Overflow occurs when the value affects the sign:
  - overflow when adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive and get a negative
  - or, subtract a positive from a negative and get a positive
- Consider the operations $A + B$, and $A - B$
  - Can overflow occur if $B$ is 0?
  - Can overflow occur if $A$ is 0?
- Response of MIPS to overflow will be covered later in the course
Overflow Detection

So how do we detect overflow?
Overflow Detection Logic

- Carry into MSB $\neq$ Carry out of MSB
  - For a N-bit ALU: Overflow = $\text{CarryIn}[N - 1]$ XOR $\text{CarryOut}[N - 1]$

```
<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X XOR Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```
**BEQ/BNE: Zero Detection Logic**

- Zero Detection Logic is just one BIG NOR gate
  - Any non-zero input to the NOR gate will cause its output to be zero
SLT: Set-on-less-than Logic

- SLT $1, $2, $3
  - if ($2 < $3)
    $1 = 1;
  else $1 = 0;

- To test $A < B$, do a subtraction ($A - B$)
  - ($A < B$) if ($A - B$) < 0, i.e. negative

- Use sign bit
  - Route the sign bit to bit 0 of result
  - Set bits 1 - 31 to zero

- There is a complication due to overflow
  - Work out solution in Homework problem 4.23
A Complete 32-bit ALU

Functionality
- Arithmetic Operations:
  - ADD, SUB
- Logical Operations:
  - AND, OR
- Compare
  - SLT
- Support for branch
  - BEQ, BNE
- Exception detection
  - Overflow

Note: “Less” is connected to “Set” input for bit 0. For all other bits, less is connected to zero.
Designing an Arithmetic Logic Unit

- **ALU Control Lines (ALUop)**
  - 000: And
  - 001: Or
  - 010: Add
  - 110: Subtract
  - 111: Set-on-less-than
Our ALU has functionality but lacks speed...

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

- Cycle Time = CLK-to-Q + Longest Gate Delay + Setup Time + Clock Skew
Adder in our ALU is in timing critical path

- The adder we just built is called a “Ripple Carry Adder”
  - The carry bit may have to propagate from LSB to MSB
  - Worst case delay for an N-bit RC adder: 2N-gate delay

- Single gate delay = 0.02 ns (inverter “speed” of 50 GHz)
- 32 bit adder => 64 gate delay => 1.28 ns delay
- Accounting for CLK2Q, set up time and clock skew, the ALU will run at << 789 MHz
Problem: ripple carry adder is slow

- Is there more than one way to do addition?
  - two extremes: ripple carry and sum-of-products

Can you see the ripple? How could you get rid of it?

\[
\begin{align*}
c_1 &= b_0c_0 + a_0c_0 + a_0b_0 \\
c_2 &= b_1c_1 + a_1c_1 + a_1b_1 \\
c_3 &= b_2c_2 + a_2c_2 + a_2b_2 \\
c_4 &= b_3c_3 + a_3c_3 + a_3b_3
\end{align*}
\]
The Theory Behind Carry Look-ahead

- Recall: CarryOut = (B & CarryIn) | (A & CarryIn) | (A & B)
  - Cin2 = Cout1 = (B1 & Cin1) | (A1 & Cin1) | (A1 & B1)
  - Cin1 = Cout0 = (B0 & Cin0) | (A0 & Cin0) | (A0 & B0)

- Substituting Cin1 into Cin2:
  - Cin2 = (A1 & A0 & B0) | (A1 & A0 & Cin0) | (A1 & B0 & Cin0) | (B1 & A0 & B0) | (B1 & A0 & Cin0) | (B1 & B0 & Cin0) | (A1 & B1)

- Now define two new terms:
  - **Generate** Carry at Bit i, \( g_i = A_i \& B_i \)
  - **Propagate** Carry via Bit i, \( p_i = A_i \lor B_i \)
Carry Lookahead: 1st Level Abstraction

- Using the two new terms we just defined:
  - Generate Carry at Bit $i$ \( g_i = A_i \& B_i \)
  - Propagate Carry via Bit $i$ \( p_i = A_i \| B_i \)

- We can rewrite:
  - \( C_{in1} = g_0 \| (p_0 \& C_{in0}) \)
  - \( C_{in2} = g_1 \| (p_1 \& g_0) \| (p_1 \& p_0 \& C_{in0}) \)
  - \( C_{in3} = g_2 \| (p_2 \& g_1) \| (p_2 \& p_1 \& g_0) \| (p_2 \& p_1 \& p_0 \& C_{in0}) \)

- Carry going into bit 3 is 1 if
  - We generate a carry at bit 2 (\( g_2 \))
  - Or we generate a carry at bit 1 (\( g_1 \)) and bit 2 allows it to propagate (\( p_2 \& g_1 \))
  - Or we generate a carry at bit 0 (\( g_0 \)) and bit 1 as well as bit 2 allows it to propagate (\( p_2 \& p_1 \& g_0 \))
  - Or we have a carry input at bit 0 (\( C_{in0} \)) and bit 0, 1, and 2 all allow it to propagate (\( p_2 \& p_1 \& p_0 \& C_{in0} \))
Carry Lookahead: 2nd Level of Abstraction

- **Propagate signals for 4-bit adders (1 gate delay to combine p’s)**
  - \( P_0 = p_3.p_2.p_1.p_0 \)
  - \( P_1 = p_7.p_6.p_5.p_4 \)
  - \( P_2 = p_{11}.p_{10}.p_9.p_8 \)
  - \( P_3 = p_{15}.p_{14}.p_{13}.p_{12} \)

- **Generate signals for 4-bit adders (2 gate delays to combine p’s and g’s)**
  - \( G_0 = g_3 + (p_3.g_2) + (p_3.p_2.g_1) + (p_3.p_2.p_1.g_0) \)
  - \( G_1 = g_7 + (p_7.g_6) + (p_7.p_6.g_5) + (p_7.p_6.p_5.g_4) \)
  - \( G_2 = g_{11} + (p_{11}.g_{10}) + (p_{11}.p_{10}.g_9) + (p_{11}.p_{10}.p_9.g_8) \)
  - \( G_3 = g_{15} + (p_{15}.g_{14}) + (p_{15}.p_{14}.g_{13}) + (p_{15}.p_{14}.p_{13}.g_{12}) \)

- **4-bit adder carry computation (2 gate delays to combine G’s P’s and c0)**
  - \( C_1 = G_0 + (P_0.c_0) \)
  - \( C_2 = G_1 + (P_1.G_0) + (P_1.P_0.c_0) \)
  - \( C_3 = G_2 + (P_2.G_1) + (P_2.P_1.G_0) + (P_2.P_1.P_0.c_0) \)

- **Total 2 + 2 + 1 = 5 levels of logic to compute c32**
  - \( C_4 \) has 2 levels of logic with \( G_i, P_i \)
  - \( G_i \) has 2 levels of logic with \( g_i, p_i \)
  - \( g_i \) & \( p_i \) have 1 level of logic with inputs \( A_i, B_i \)
Carry Look-ahead Adder (CLA)

- 16-bit Ripple carry (RC) adder has
  \[16 \times 2 = 32 \text{ gate delays}\]
- 4-bit carry-look-ahead adder has
  5 gate delays
- CLA adder is faster than RC by a factor of
  \[\frac{32}{5} \approx 6\]
Conclusion

- We can build an ALU to support the MIPS instruction set
  - Key idea: use multiplexor to select the output we want
  - Efficiently perform subtraction using two’s complement
  - Replicate a 1-bit ALU to produce a 32-bit ALU

- Important points about hardware
  - All of the gates are always working
  - The speed of a gate is affected by the number of inputs to the gate
  - The speed of a circuit is affected by the number of gates in series (on the “critical path” or the “deepest level of logic”)
    - For computer hardware, “Speed is it!”
Announcements

- Two sections for 141:
  - Friday 11 - 11:50 PM Center 109
  - Friday 2 - 2:50 PM CSB 105

- Reading Assignment
  - Chapter 4: Arithmetic for Computers, Sec. 4.6 - 4.11
  - Chapter 5: The Processor: Datapath and Control, Sec.

- Homework 2: Due Wed., July 7 in class
  - 2.1 through 2.5, 2.18 through 2.21
  - 4.9, 4.12, 4.14, 4.22, 4.23, 4.44

- Quiz
  - When: Wed., July 7, First 10 minutes of the class
  - Topic: Performance, Chapter 2
  - Need: Paper, pen, calculator

- CSE 141 Lab Assignment has been posted on the web page
  [http://www.cse.ucsd.edu/classes/su04/cse141L](http://www.cse.ucsd.edu/classes/su04/cse141L)