CSE 141 – Computer Architecture
Summer Session I, 2005

Lecture 2
ALU
Pramod V. Argade
CSE141: Introduction to Computer Architecture

**Instructor:** Pramod V. Argade (p2argade@cs.ucsd.edu)  
Office Hour: Wed. 4:30 - 5:30 PM (EBU3 2204)

**TAs:**  
Chengmo Yang: c5yang@cs.ucsd.edu  
Wenjing Rao: wrao@cs.ucsd.edu

**Lecture:** Mon/Wed. 6:00 - 8:50 PM, HSS 1330

**Textbook:** Computer Organization & Design  
The Hardware Software Interface, 3rd Edition.  
Authors: Patterson and Hennessy

**Web-page:** [http://www.cse.ucsd.edu/classes/su05/cse141](http://www.cse.ucsd.edu/classes/su05/cse141)
# Summer Session I, 2005
## CSE141 Course Schedule

<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. 6/27</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Introduction, Ch. 1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ISA, Ch. 2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>Wed. 6/29</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Arithmetic, Ch. 3</td>
<td>ISA</td>
<td>#1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Ch. 2</td>
<td>Ch. 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mon. 7/4</td>
<td>No Class</td>
<td></td>
<td>July 4th Holiday</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>Wed. 7/6</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Performance, Ch. 4</td>
<td>Arithmetic</td>
<td>#2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Single-cycle CPU Ch. 5</td>
<td>Ch. 3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Mon. 7/11</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Single-cycle CPU Ch. 5 Cont.</td>
<td>Performance</td>
<td>#3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Multi-cycle CPU Ch. 5</td>
<td>Ch. 4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Tue. 7/12</td>
<td>7:30 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Multi-cycle CPU Ch. 5 Cont. (July 4th make up class)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>Wed. 7/13</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</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. 7/18</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Mid-term Exam Exceptions</td>
<td>-</td>
<td>#4</td>
</tr>
<tr>
<td>8</td>
<td>Tue. 7/19</td>
<td>7:30 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Pipelining Ch. 6 (July 4th make up class)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>Wed. 7/20</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Hazards, Ch. 6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>Mon. 7/25</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Memory Hierarchy &amp; Caches Ch. 7</td>
<td>Hazards Ch. 6</td>
<td>#5</td>
</tr>
<tr>
<td>11</td>
<td>Wed. 7/27</td>
<td>6 - 8:50 PM</td>
<td>HSS 1330</td>
<td>Virtual Memory, Ch. 7 Course Review</td>
<td>Cache Ch. 7</td>
<td>#6</td>
</tr>
<tr>
<td>12</td>
<td>Sat. 7/30</td>
<td>TBD</td>
<td>TBD</td>
<td>Final Exam</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Announcements

- **Discussions Sections for 141:**
  Thursday, 7:30 - 8:30 Room TBD (Check class web page)

- **Office Hour:**
  - Wed. 4:30 - 5:30 PM (EBU3 2204)

- **Reading Assignment**
  - Chapter 3. Arithmetic for Computers
    Sections 3.1 - 3.6, B5

- **Homework 2: Due Wed., July 6th in class**
  3.2, 3.4, 3.7, 3.9, 3.12, 3.13, 3.19, 3.28, 3.30, 3.36, 3.38
  4.23 (2nd Edition)
  Multiply -12x-13 using Booth’s algorithm and 5-bit 2’s complement representation of multiplicand and multiplier.

- **Quiz**
  **When:** Wed., July 6th, First 10 minutes of the class
  **Topic:** Arithmetic, Chapter 3
  **Need:** Paper, pen
Computer Arithmetic and ALU Design
Arithmetic -- The heart of instruction execution
Bits everywhere… What do they mean?

bits (011011011100010 ....01)

instruction
R-format I-format ...

data
number
text chars ..............

integer
signed unsigned

floating point
single precision double precision

... ...

... ...

... ...
Questions About Numbers and ALU

• How do you represent
  – signed and unsigned numbers
  – 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

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

\[
\begin{array}{c c c c c}
& & & & 1 \\
& 0 & 1 & 0 & 1
\end{array}
\]

3 + 3 = 6

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

\[
\begin{array}{c c c c c}
& & & & 1 \\
& 0 & 1 & 1 & 0
\end{array}
\]
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:**
  - $000 = +0$
  - $001 = +1$
  - $010 = +2$
  - $011 = +3$
  - $100 = -0$
  - $101 = -1$
  - $110 = -2$
  - $111 = -3$

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

- **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: $7 - 6 = 7 + (-6) = 1$  
  \[
  \begin{array}{cccc}
  1 & 1 & 1 & 1 \\
  0 & 1 & 1 & 1 \\
  + & 1 & 0 & 1 & 0 \\
  0 & 0 & 0 & 1 \\
  \end{array}
  \]

  $3 - 5 = 3 + (-5) = -2$  
  \[
  \begin{array}{cccc}
  1 & 1 & 1 & 1 \\
  0 & 0 & 1 & 1 \\
  + & 1 & 0 & 1 & 1 \\
  1 & 1 & 1 & 0 \\
  \end{array}
  \]
Immediate Field in I-Format

- I-Format Instruction Example:
  ```
  addi $s3, $t0, 4
  ```
  
  - 6 bits  5 bits  5 bits
  
<table>
<thead>
<tr>
<th>I</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit address</th>
</tr>
</thead>
</table>

- For LW, SW, BEQ, BNE, ADDI
  - 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?
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 . b)

\[
\begin{array}{ccc}
0 & 0 & 0 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 1 \\
\end{array}
\]

2. OR gate (c = a + b)

\[
\begin{array}{ccc}
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}
\]

3. Inverter (c = a)

\[
\begin{array}{cc}
0 & 1 \\
1 & 0 \\
\end{array}
\]

4. Multiplexor
   (if d = 0, c = a; else c = b)

\[
\begin{array}{cc}
0 & a \\
1 & b \\
\end{array}
\]
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?
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</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 CarryOut

$$\text{CarryOut} = (!A \& B \& \text{ CarryIn}) | (A \& !B \& \text{ CarryIn}) | (A \& B \& !\text{ CarryIn})$$

$$= (B \& \text{ CarryIn}) | (A \& \text{ CarryIn}) | (A \& B)$$

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

<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>

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

- Implements functions:
  - AND
  - OR
  - ADD
- What about SUB and SLT?

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 \(!B\):
  - \(A - B = A + (-B) = A + (\!B + 1) = A + \!B + 1\)

- Binvert provides the negation
- How about “+1”? 
- For SUB
  - set Binvert = 1
  - CarryIn = 1 for LSB
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 ! = Carry out of MSB
  - For a N-bit ALU: Overflow = CarryIn[N - 1] XOR 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 (2nd Edition)
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
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!”
Our ALU has functionality but lacks speed...

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

- **Cycle Time** = \(\text{CLK-to-Q} + \text{Longest Gate Delay} + \text{Setup Time} + \text{Clock Skew}\)
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?

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

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

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

- Now define two new terms:
  - **Generate** Carry at Bit \( i \) \( g_i = A_i \& B_i \)
  - **Propagate** Carry via Bit \( i \) \( p_i = A_i \| B_i \)
  - \( \text{Cin1} = \text{Cin0}(A0 \| B0) | (A0 \& B0) = (\text{Cin0} \& p0) | g0 \)
  - \( \text{Cin2} = \text{Cin1}(A1 \| B1) | (A1 \& B1) = (\text{Cin1} \& p1) | g1 \)
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 $i$ $p_i = A_i \mid B_i$

- We can rewrite:
  - $C_i^{in1} = g_0 \mid (p_0 \& C_0^{in})$
  - $C_i^{in2} = g_1 \mid (p_1 \& g_0) \mid (p_1 \& p_0 \& C_0^{in})$
  - $C_i^{in3} = g_2 \mid (p_2 \& g_1) \mid (p_2 \& p_1 \& g_0) \mid (p_2 \& p_1 \& p_0 \& C_0^{in})$

- 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 ($C_0^{in}$) and bit 0, 1, and 2 all allow it to propagate ($p_2 \& p_1 \& p_0 \& C_0^{in}$)
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 c16
  - \( C_4 \) has 2 levels of logic with \( G_i, P_i \)
  - \( G_i \) has 2 levels of logic with gi, pi
  - gi & pi have 1 level of logic with inputs Ai, Bi
16-bit Adder from Four 4-bit ALUs

- 16-bit Ripple Carry (RC) adder has $16 \times 2 = 32$ gate delays
- 16-bit Carry-Look-Ahead adder (CLA) has 5 gate delays
- CLA adder is faster than RC by a factor of $32/5 \sim 6$
Grade school Multiplication algorithm

• In general (ignoring sign bits):
  • m bits x n bits = (m+n) bit product

• Binary makes it easy:
  • 0 => place 0 ( 0 x multiplicand)
  • 1 => place multiplicand ( 1 x multiplicand)

• Paper and pencil example of binary multiplication:
  (8*10 = 80, 0x8 * 0xa = 0x50 )

```
  1000 (multiplicand)
  x 1010 (multiplier)
    0000
    1000x
    0000xx
    1000xxx
  1010000 (Result)
```
Observations about Multiplication

- More complicated than addition
- Simple algorithm:
  - Accomplished via shift and add
- More time delay and more gates (=> silicon area)
- Let's look at 3 versions based on grade school algorithm
Multiplication: First Version

Initialization:
- Load 32-bit multiplicand and zero extend to 64 bits
- Load 64-bit product register with zero

Need a state machine to control operation

32 Iterations are required
- Each iteration takes 3 clocks
- Total $96 + 3 = 99$ clocks

Observations?
Multiplication: Second Version

Initialization:
- Load 32-bit multiplicand to 32-bit register
- Load 32-bit multiplier to 32-bit register
- Load 64-bit product register with zero

Need a state machine to control operation

Observations?
Multiplication: Third Version

**Initialization:**
- Load 32-bit multiplicand to 32-bit register
- Load upper 32 bits of product register with zero
- Load lower 32 bits of product register with multiplier

**Need a state machine to control operation**

**Observations?**
Multiplying Signed Numbers

- Convert all operands to positive
- Determine sign of the product
  - Sign of the product = \text{sign}(\text{op1}) \ ^\wedge \text{sign}(\text{op2})
- Multiply positive operands (only 31 bits)
- If the sign of the result is negative, negate the result
- Previous approach will work:
  - Must extend sign of the product when shifting

Is there a better way?
Booth’s Algorithm

- An elegant approach to multiplying signed numbers
- With ability to add, subtract and shift
  - There are multiple ways to do multiply
- Consider signed operands A and B

\[
A = (A_{31} \times 2^{31}) + (A_{30} \times 2^{30}) + (A_{29} \times 2^{29}) + \ldots + (A_1 \times 2^1) + (A_0 \times 2^0) \\
= (-A_{31} \times 2^{31}) + (2A_{30} - A_{30}) \times 2^{30} + (2A_{29} - A_{29}) \times 2^{29} + \ldots + (2A_0 - A_0) \times 2^0 \\
= (A_{30} - A_{31}) \times 2^{31} + (A_{29} - A_{30}) \times 2^{30} + \ldots + (A_1 - A_2) \times 2^1 + (A_0 - A_0) \times 2^0
\]

\[
A \times B = [(A_{30} - A_{31}) \times 2^{31} + (A_{29} - A_{30}) \times 2^{30} + \ldots + (A_1 - A_2) \times 2^1 + (A_0 - A_0) \times 2^0] \times B \\
= (A_{30} - A_{31}) \times 2^{31} \times B + (A_{29} - A_{30}) \times 2^{30} \times B + \ldots + (A_1 - A_2) \times 2^1 \times B + (A_0 - A_0) \times 2^0 \times B
\]

- Recipe:
  Evaluate \((A_{i-1} - A_i)\)
  
  0: Do nothing
  1: Add B
  -1: Subtract B
Booths algorithm: Signed multiplication

\[ A \times B = (A_{30} - A_{31})2^{31}B + (A_{29} - A_{30})2^{30}B + \ldots + (A_{1} - A_{2})2^{1}B + (A_{-1} - A_{0})2^{0}B \]

<table>
<thead>
<tr>
<th>Current Bit</th>
<th>Bit to the Right</th>
<th>Explanation</th>
<th>Example</th>
<th>Op</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Begins run of 1s</td>
<td>0001111000</td>
<td>sub</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Middle of run of 1s</td>
<td>0001111000</td>
<td>none</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>End of run of 1s</td>
<td>00111100</td>
<td>add</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Middle of run of 0s</td>
<td>00111100</td>
<td>none</td>
</tr>
</tbody>
</table>

Originally for Speed (when shift was faster than add)

- Replace a string of 1s in multiplier with an initial subtract when we first see a one and then later add for the bit after the last one
- Potential speed up recognizing that string of 0’s and 1’s requires no operation!
Booth’s Algorithm

• Recipe: for A*B
  Add $A_{i-1} = 0$
  Evaluate $(A_{i-1} - A_i)$
    0: Do nothing
    1: Add B
    -1: Subtract B

• Example: Use Booth’s Algorithm for following multiplication
  $2 \times (-6) = 0010 \times 1010 = -12 = 1111\ 0100$
**Division**

\[
\begin{array}{c|c|c}
\text{Divisor} & 1000 & \text{Dividend} \\
\hline
0 & 1001010 & \\
\hline
-1000 & 10101 & \\
-1000 & 1010 & \\
-1000 & 10 & \\
\end{array}
\]

See how big a number can be subtracted, creating quotient bit on each step

Binary => 1 * divisor or 0 * divisor

Dividend = Quotient x Divisor + Remainder

=> sizeof(Dividend) = sizeof(Quotient) + sizeof(Divisor)

3 versions of divide, successive refinement
Division 1.0

- Initialization:
- 32-bit quotient register = 0
- 64-bit remainder = dividend
- 64-bit Divisor = (32-bit divisor << 32)
Division 1.0

1. Subtract the Divisor register from the Remainder register, and place the result in the Remainder register.

   Test Remainder

   Remainder >= 0
   2a. Shift the Quotient register to the left setting the new rightmost bit to 1.

   Remainder < 0
   2b. Restore the original value by adding the Divisor register to the Remainder register, and place the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0.

3. Shift the Divisor register right 1 bit.

33rd repetition?
No: < 33 repetitions

   Yes: 33 repetitions

   Done
Divide Algorithm

- Optimizations similar to that for multiply algorithm can be done
  - 32-bit Divisor register
  - 32-bit ALU
  - Quotient bits are left shifted into the remainder register
- In case the result of subtraction is negative, remainder register has to be restored
  - Takes one extra clock cycle
- Non-restoring divide algorithm removes this step
- Divide overflow case
  - 0x80000000/-1
Floating Point: Introduction

- We need a way to represent real numbers
  - Numbers with fractions, e.g., 3.14159265… (recognize me?)
  - Very small numbers, e.g., 0.0000000000000000000000000013621
  - Very large numbers, e.g., 9,349,398,989,787,762,244,859,087,678

- Binary Fractions:
  \[1011_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0\]
  so...
  \[101.011_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}\]
  e.g.,
  \[.75 = 0.5 + 0.25 = 1/2 + 1/4 = .11_2\]
Recall Scientific Notation

\[ 6.02 \times 10^{23} \]

- **Mantissa**
- **Radix (base)**
- **Exponent**

**IEEE Single Precision F.P.**

\[ \pm 1.M \times 2^{e-127} \]
IEEE 754 Single-precision Floating-Point

\[ N = (-1)^S (1.M) 2^{E-127} \]

• Example:
  Convert -325.75 to IEEE Single Precision Floating Point Representation
IEEE 754 Double-precision Floating-Point

\[ N = (-1)^S \cdot (1.M) \cdot 2^{E-1023} \]

- Example:
  Convert -325.75 to IEEE Double Precision Floating Point Representation
IEEE 754 Single Precision FP

- If $E=255$ and $F$ is nonzero, then $V = \text{NaN} \ ("Not a number")$
- If $E=255$ and $F$ is zero and $S$ is 1, then $V = -\text{Infinity}$
- If $E=255$ and $F$ is zero and $S$ is 0, then $V = \text{Infinity}$
- If $0 < E < 255$ then $V = (-1)^{S} \times 2^{(E-127)} \times (1.F)$
- If $E=0$ and $F$ is zero and $S$ is 1, then $V = -0$
- If $E=0$ and $F$ is zero and $S$ is 0, then $V = 0$

In particular,

- $0 \ 00000000 \ 00000000000000000000000000 = 0$
- $1 \ 00000000 \ 00000000000000000000000000 = -0$
- $0 \ 11111111 \ 00000000000000000000000000 = \text{Infinity}$
- $1 \ 11111111 \ 00000000000000000000000000 = -\text{Infinity}$
- $0 \ 11111111 \ 00000100000000000000000000 = \text{NaN}$
- $1 \ 11111111 \ 0010001000100101010101010 = \text{NaN}$
- $0 \ 10000000 \ 00000000000000000000000000 = +1 \times 2^{**(128-127)} \times 1.0 = 2$
Floating Point Addition

1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent

2. Add the significands

3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent

4. Round the significand to the appropriate number of bits

Overflow or underflow?

Still normalized?

Done

Exception
Floating Point Addition

Example: $0.5 + (-0.4375)$

$L = 1.000 \times 2^{-1}$

$R = -1.110 \times 2^{-2}$

$L_e - R_e = -1 - (-2) = 1$

Result $= 1.000 \times 2^{-4}$
IEEE 754 Floating Point

- Increasing the size of significand enhances accuracy
- Increasing the size of exponent increases the range of the numbers that can be represented
- Special representation of 0 (E = 00000000)
- Overflow or underflow can happen
- Can do integer compare for greater-than, sign: Fast comparison!
- Single Precision
  - Range of about $2 \times 10^{-38}$ to $2 \times 10^{38}$
- Double Precision
  - Range of about $2 \times 10^{-308}$ to $2 \times 10^{308}$
- Infinite variety of real numbers exist between, say, 0 and 1
  - Not more than $2^{53}$ can be represented exactly in double precision
Floating Point Complexities

- Operations are somewhat more complicated
- In addition to overflow we can have “underflow”
- Accuracy can be a big problem
  - IEEE 754 keeps two extra bits, guard and round
  - four rounding modes
  - positive divided by zero yields “infinity”
  - zero divide by zero yields “not a number”
- Implementing the standard can be tricky
- Not using the standard can be even worse
  - See text for description of 80x86 and Pentium bug!
Summary

- Multiplication and division take much longer than addition, requiring multiple addition steps.
- Floating Point extends the range of numbers that can be represented, at the expense of precision (accuracy).
- FP operations are very similar to integer, but with pre- and post-processing.