Arithmetic Circuits
(Part I)

Randy H. Katz
University of California, Berkeley
Fall 2005

Motivation
Arithmetic circuits are excellent examples of comb. logic design
- Time vs. Space Trade-offs
  Doing things fast requires more logic and thus more space
  Example: carry lookahead logic
- Arithmetic Logic Units
  Critical component of processor datapath
  Inner-most "loop" of most computer instructions

Overview
- Binary Number Representation
  - Sign & Magnitude, Ones Complement, Twos Complement
- Binary Addition
  - Full Adder Revisited
- ALU Design
- BCD Circuits
- Combinational Multiplier Circuit
- Design Case Study: 8 Bit Multiplier
- Sequential Multiplier Circuit

Number Systems
Representation of Negative Numbers
- Representation of positive numbers same in most systems
- Major differences are in how negative numbers are represented
- Three major schemes:
  - sign and magnitude
  - ones complement
  - twos complement
- Assumptions:
  - we'll assume a 4 bit machine word
  - 16 different values can be represented
  - roughly half are positive, half are negative

Number Systems
- Sign and Magnitude Representation
  - High order bit is sign: 0 = positive (or zero), 1 = negative
  - Three low order bits is the magnitude: 0 (000) thru 7 (111)
  - Number range for n bits = +/-2^n-1
  - Representations for 0
Number Systems

**Ones Complement**

Subtraction implemented by addition & 1's complement

Still two representations of 0! This causes some problems

Some complexities in addition

---

**Twos Complement Numbers**

\[ N^* = 2^n - N \]

Example: Twos complement of 7

\[ \begin{array}{c|c}
\text{Sub} & \text{Binary} \\
\hline
7 & 0111 \\
\hline
0101 & \text{repr. of -7}
\end{array} \]

Example: Twos complement of -7

\[ \begin{array}{c|c}
\text{Sub} & \text{Binary} \\
\hline
-7 & 1001 \\
\hline
0111 & \text{repr. of 7}
\end{array} \]

Shortcut method:
Twos complement = bitwise complement + 1

0111 -> 1000 + 1 -> 1001 (representation of -7)

1001 -> 0110 + 1 -> 0111 (representation of 7)

---

**Number Representations**

**Twos Complement**

\[ \begin{array}{c|c}
\text{Number} & \text{Binary} \\
\hline
-4 & 1001 \\
\hline
-3 & 0111 \\
\hline
-2 & 0100 \\
\hline
-1 & 0011 \\
\hline
0 & 0000 \\
\hline
1 & 0001 \\
\hline
2 & 0010 \\
\hline
3 & 0011 \\
\hline
4 & 0100 \\
\hline
\end{array} \]

Only one representation for 0

One more negative number than positive number

---

**Number Systems**

**Addition and Subtraction of Numbers**

**Ones Complement Calculations**

\[ \begin{array}{c|c|c|c}
\text{Binary} & \text{Result} & \text{Sign} \\
\hline
0100 & 0111 & +3 \\
\hline
0101 & 1000 & +4 \\
\hline
1111 & 1000 & -3 \\
\hline
1110 & 0111 & -4 \\
\hline
\end{array} \]

End around carry

1000

---

**Twos Complement Calculations**

**Why does end-around carry work?**

It's equivalent to subtracting \(2^n\) and adding 1

\[ M - N = M + (2^n - 1 - N) = (M - N) + 2^n - 1 \text{ (M + N)} \]

\[ M = (2^n - 1 - (M + N)) \text{ (M + N)} \]

\[ -M = (2^n - 1 - (M + N)) \text{ (M + N)} \]

\[ M + N = 2^n - 1 \text{ (M + N)} \]

After end around carry:

\[ 2^n - 1 \text{ (M + N)} \]

This is the correct form for representing \(-M + N\) in 1's compli

---

**Number Systems**

**Addition and Subtraction of Binary Numbers**

**Ones Complement Calculations**

**Twos Complement Calculations**

\[ \begin{array}{c|c|c|c}
\text{Binary} & \text{Result} & \text{Sign} \\
\hline
0100 & 0111 & +3 \\
\hline
0101 & 1000 & +4 \\
\hline
1111 & 1000 & -3 \\
\hline
1110 & 0111 & -4 \\
\hline
\end{array} \]

End around carry

1000
Number Systems

Addition and Subtraction of Binary Numbers

Two's Complement Calculations

<table>
<thead>
<tr>
<th>Add</th>
<th>0000</th>
<th>-4</th>
<th>1100</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0000</td>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>+3</td>
<td>0011</td>
<td>-3</td>
<td>1101</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>-7</td>
<td>1100</td>
</tr>
</tbody>
</table>

If carry-in to sign bit: carry-out then ignore carry

If carry-in differs from carry-out then overflow

Simpler addition scheme makes two's complement the most common choice for integer number systems within digital systems

Number Systems

Overflow Conditions

Add two positive numbers to get a negative number or two negative numbers to get a positive number

Overflow Conditions

Add two positive numbers to get a negative number or two negative numbers to get a positive number

<table>
<thead>
<tr>
<th>Add</th>
<th>Sum</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1 0 1</td>
<td>0 1 0</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 0 0</td>
<td>1 0 1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>0 1 0</td>
<td>1 0 1</td>
</tr>
</tbody>
</table>

Sum = A_iB_i + A_iB_i

Carry = A_iB_i

Number Systems

Addition and Subtraction of Binary Numbers

Two's Complement Calculations

Why can the carry-out be ignored?

-M + N where N > M:

\[ M^* + N = (2^n - M) + N = 2^n + (N - M) \]

Ignoring carry-out is just like subtracting \( 2^n \)

-M + -N where N > M or \( 2^n \):

\[ M^* + (-N) = M^* + N^* = (2^n - M) + (2^n - N) \]

For M, N > 0

\[ = 2^n - (M + N) = 2^n \]

After ignoring the carry, this is just the right two's complement representation for \( -(M + N) \)

Number Systems

Overflow Conditions

Add two positive numbers to get a negative number or two negative numbers to get a positive number

Overflow Conditions

Add two positive numbers to get a negative number or two negative numbers to get a positive number

<table>
<thead>
<tr>
<th>Add</th>
<th>Sum</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1 0 1</td>
<td>0 1 0</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 0 0</td>
<td>1 0 1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>0 1 0</td>
<td>1 0 1</td>
</tr>
</tbody>
</table>

Sum = A_iB_i + A_iB_i

Carry = A_iB_i

Number Systems

Addition and Subtraction of Binary Numbers

Two's Complement Calculations

Why can the carry-out be ignored?

-M + N where N > M:

\[ M^* + N = (2^n - M) + N = 2^n + (N - M) \]

Ignoring carry-out is just like subtracting \( 2^n \)

-M + -N where N > M or \( 2^n \):

\[ M^* + (-N) = M^* + N^* = (2^n - M) + (2^n - N) \]

For M, N > 0

\[ = 2^n - (M + N) = 2^n \]

After ignoring the carry, this is just the right two's complement representation for \( -(M + N) \)

Number Systems

Overflow Conditions

Add two positive numbers to get a negative number or two negative numbers to get a positive number

Overflow Conditions

Add two positive numbers to get a negative number or two negative numbers to get a positive number

<table>
<thead>
<tr>
<th>Add</th>
<th>Sum</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1 0 1</td>
<td>0 1 0</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 0 0</td>
<td>1 0 1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>0 1 0</td>
<td>1 0 1</td>
</tr>
</tbody>
</table>

Sum = A_iB_i + A_iB_i

Carry = A_iB_i

Number Systems

Addition and Subtraction of Binary Numbers

Two's Complement Calculations

Why can the carry-out be ignored?

-M + N where N > M:

\[ M^* + N = (2^n - M) + N = 2^n + (N - M) \]

Ignoring carry-out is just like subtracting \( 2^n \)

-M + -N where N > M or \( 2^n \):

\[ M^* + (-N) = M^* + N^* = (2^n - M) + (2^n - N) \]

For M, N > 0

\[ = 2^n - (M + N) = 2^n \]

After ignoring the carry, this is just the right two's complement representation for \( -(M + N) \)

Number Systems

Overflow Conditions

Add two positive numbers to get a negative number or two negative numbers to get a positive number

Overflow Conditions

Add two positive numbers to get a negative number or two negative numbers to get a positive number

<table>
<thead>
<tr>
<th>Add</th>
<th>Sum</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1 0 1</td>
<td>0 1 0</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 0 0</td>
<td>1 0 1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>0 1 0</td>
<td>1 0 1</td>
</tr>
</tbody>
</table>

Sum = A_iB_i + A_iB_i

Carry = A_iB_i

Number Systems

Addition and Subtraction of Binary Numbers

Two's Complement Calculations

Why can the carry-out be ignored?

-M + N where N > M:

\[ M^* + N = (2^n - M) + N = 2^n + (N - M) \]

Ignoring carry-out is just like subtracting \( 2^n \)

-M + -N where N > M or \( 2^n \):

\[ M^* + (-N) = M^* + N^* = (2^n - M) + (2^n - N) \]

For M, N > 0

\[ = 2^n - (M + N) = 2^n \]

After ignoring the carry, this is just the right two's complement representation for \( -(M + N) \)

Networks for Binary Addition

Half Adder

With two complement numbers, addition is sufficient

\[
\text{Sum} = A_iB_i + A_iB_i = A_iB_i
\]

\[
\text{Carry} = A_iB_i
\]

Networks for Binary Addition

Full Adder

Cascaded Multi-bit Adder

usually interested in adding more than two bits

this motivates the need for the full adder

Lecture #23: Arithmetic Circuits
Lecture #23: Arithmetic Circuits

**Networks for Binary Addition**

*Full Adder*

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>S</th>
<th>CO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

S = CI xor A xor B
CO = B CI + A CI + A B + CI (A + B) + A B

---

**Networks for Binary Addition**

*Adder/Subtractor*

\[ A - B = A + (-B) = A + B' + 1 \]

---

**Networks for Binary Addition**

*Carry Lookahead Circuits*

Critical delay: the propagation of carry from low to high order stages

Two gate delays to compute CO

4 stage adder

Final sum and carry

---

**Networks for Binary Addition**

*Carry Lookahead Logic*

Carry Generate \( G_i = A_i \overline{B_i} \) must generate carry when \( A = B = 1 \)

Carry Propagate \( P_i = A_i \overline{B_i} \) carry in will equal carry out here

Sum and carry can be reexpressed in terms of generate/propagate:

\[
\begin{align*}
S_i &= A_i \overline{B_i} \overline{C_i} + P_i \overline{C_i} + C_i \\
C_{i+1} &= A_i \overline{B_i} + A_i C_i + B_i C_i \\
&= A_i \overline{B_i} + A_i C_i + B_i (A_i + B_i) \\
&= A_i \overline{B_i} + C_i (A_i \overline{B_i}) \\
&= g_i + P_i 
\end{align*}
\]
Each Networks for Binary Addition

Carry Lookahead Logic

Reexpress the carry logic as follows:

\[
C_1 = G_0 \cdot P_0 \cdot C_0
\]

\[
C_2 = G_1 \cdot G_0 \cdot C_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0
\]

\[
C_3 = G_2 \cdot P_2 \cdot C_2 + G_2 \cdot P_2 \cdot G_1 + P_2 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0
\]

\[
C_4 = G_3 \cdot P_3 \cdot C_3 + G_4 \cdot P_2 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0
\]

Each of the carry equations can be implemented in a two-level logic network.

Variables are the adder inputs and carry in to stage 0!

Networks for Binary Addition

Carry Lookahead Logic

Cascade Carry Lookahead

Carry lookahead logic generates individual carries

Sums computed much faster

Networks for Binary Addition

Carry Lookahead Logic

Cascaded Carry Lookahead

4-bit adders with internal carry lookahead

second level carry lookahead unit, extends lookahead to 16 bits

Group P = P_3 \cdot P_2 \cdot P_1 \cdot C_0
Group G = G_3 \cdot G_2 \cdot G_1 \cdot P_3 \cdot P_2 \cdot G_1 \cdot P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0

Networks for Binary Addition

Carry Select Adder

Redundant hardware to make carry calculation go faster

Adder Low

Adder High

compute the high order sums in parallel

one addition assumes carry in = 0

the other assumes carry in = 1

Arithmetic Logic Unit Design

Sample ALU

<table>
<thead>
<tr>
<th>M</th>
<th>Logical Shift Operations</th>
<th>G</th>
<th>S</th>
<th>Function</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F = A \oplus B</td>
<td>0</td>
<td>0</td>
<td>Complement of A transferred to output</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F = A \oplus B \oplus C</td>
<td>1</td>
<td>0</td>
<td>Compute XOR of A, B</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F = \text{not } A \oplus B \oplus C</td>
<td>1</td>
<td>1</td>
<td>Compute XOR of A, B</td>
<td></td>
</tr>
<tr>
<td>M = 1, C_0 = 0, Arithmetic Operations</td>
<td>0</td>
<td>0</td>
<td>F = A</td>
<td>Input A passed to output</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>F = A \oplus B</td>
<td>1</td>
<td>0</td>
<td>Complement of A passed to output</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F = A \oplus B</td>
<td>1</td>
<td>1</td>
<td>Sum of A and B</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F = \text{not } A \oplus B = 1 \oplus A \oplus B</td>
<td>1</td>
<td>1</td>
<td>Sum of B and complement of A</td>
<td></td>
</tr>
<tr>
<td>M = 1, C_0 = 1, Arithmetic Operations</td>
<td>0</td>
<td>0</td>
<td>F = A</td>
<td>Increment A</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>F = A \oplus 1 \oplus C_0</td>
<td>1</td>
<td>0</td>
<td>Two's complement of A</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>F = A \oplus B + 1 \oplus C_0</td>
<td>1</td>
<td>1</td>
<td>Increment sum of A and B</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F = \text{not } A \oplus B + 1 \oplus C_0</td>
<td>1</td>
<td>1</td>
<td>Increment B</td>
<td></td>
</tr>
</tbody>
</table>

Logical and Arithmetic Operations

Not all operations appear useful, but "fall out" of internal logic.
Lecture Review

We have covered:

- **Binary Number Representation**
  positive numbers the same
  difference is in how negative numbers are represented
  twos complement easiest to handle:
    one representation for zero, slightly
    complicated complementation, simple addition

- **Binary Networks for Additions**
  basic HA, FA
  carry lookahead logic

- **ALU Design**
  specification and implementation