Chapter # 5: Arithmetic Circuits

Contemporary Logic Design

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

Ones complement:

N is a positive number, then \( \overline{N} \) is its negative 1's complement
\[ \overline{N} = (2^n - 1) - N \]

Example: 1's complement of 7
\[ \begin{align*}
    2^4 &= 10000 \\
    -1 &= 00001 \\
    1111 &= -7 \\
    0111 &= 7 \text{ in 1's comp.}
\end{align*} \]

Shortcut method:
simply compute bit wise complement
0111 -> 1000
Number Systems

Ones Complement

Subtraction implemented by addition & 1’s complement

Still two representations of 0! This causes some problems

Some complexities in addition

Number Representations

Twos Complement

like 1’s comp except shifted one position clockwise

Only one representation for 0

One more negative number than positive number
Number Systems

Twos Complement Numbers

N* = 2^n - N

Example: Twos complement of 7

```
2^4 = 10000
```

Sub 7 = 0111

1001 = repr. of -7

Example: Twos complement of -7

```
2^4 = 10000
```

Sub -7 = 1001

0111 = repr. of 7

Shortcut method:

Twos complement = bitwise complement + 1

0111 -> 1000 + 1 -> 1001 (representation of -7)
1001 -> 0110 + 1 -> 0111 (representation of 7)

Number Representations

Addition and Subtraction of Numbers

Sign and Magnitude

Result sign bit is the same as the operands' sign

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0100</td>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>+3</td>
<td>0011</td>
<td>+(-3)</td>
<td>1011</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>-7</td>
<td>1111</td>
</tr>
</tbody>
</table>

When signs differ, operation is subtract, sign of result depends on sign of number with the larger magnitude

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0100</td>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>-3</td>
<td>1011</td>
<td>+3</td>
<td>0011</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>-1</td>
<td>1001</td>
</tr>
</tbody>
</table>
Number Systems

Addition and Subtraction of Numbers

Ones Complement Calculations

\[
\begin{array}{cccc}
4 & 0100 & -4 & 1011 \\
+3 & 0011 & +(-3) & 1100 \\
7 & 0111 & -7 & 10111 \\
\end{array}
\]

End around carry \[\rightarrow 1\]

\[
\begin{array}{cccc}
4 & 0100 & -4 & 1011 \\
-3 & 1100 & +3 & 0011 \\
1 & 10000 & -1 & 1110 \\
\end{array}
\]

End around carry \[\rightarrow 1\]

0001

Number Systems

Addition and Subtraction of Binary Numbers

Twos Complement Calculations

\[
\begin{array}{cccc}
4 & 0100 & -4 & 1100 \\
+3 & 0011 & +(-3) & 1101 \\
7 & 0111 & -7 & 11001 \\
\end{array}
\]

If carry-in to sign = carry-out then ignore carry

If carry-in differs from carry-out then overflow

\[
\begin{array}{cccc}
4 & 0100 & -4 & 1100 \\
-3 & 1101 & +3 & 0011 \\
1 & 10001 & -1 & 1111 \\
\end{array}
\]

Simpler addition scheme makes twos complement the most common choice for integer number systems within digital systems
**Number Systems**

**Overflow Conditions**

<table>
<thead>
<tr>
<th>Number</th>
<th>0111</th>
<th>0101</th>
<th>1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>-7</td>
</tr>
<tr>
<td>-3</td>
<td>0</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>-8</td>
<td>1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>1100</td>
</tr>
</tbody>
</table>

Overflow when carry in to sign does not equal carry out

<table>
<thead>
<tr>
<th>Number</th>
<th>0000</th>
<th>0101</th>
<th>1101</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>-3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>-5</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

No overflow

**Half Adder**

With twos complement numbers, addition is sufficient

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

Sum = $\overline{A_i}B_i + A_i\overline{B_i}$

Carry = $A_iB_i$

Half-adder Schematic
Cascaded Multi-bit Adder

- usually interested in adding more than two bits
- this motivates the need for the full adder

\[
\begin{array}{c|c|c|c|c}
A & B & C_i & S & C_o \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c}
A & B & C_i & S & C_o \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

\[
S = C_i \text{xor} A \text{xor} B
\]

\[
C_O = B C_i + A C_i + A B = C_i (A + B) + A B
\]
**Networks for Binary Addition**

**Full Adder/Half Adder**

**Standard Approach: 6 Gates**

**Alternative Implementation: 5 Gates**

A + B + CI = A + B + CI

\[ A \oplus B = A \oplus B + B \oplus A \]

**Adder/Subtractor**

A - B = A + (-B) = A + B + 1
**Carry Lookahead Circuits**

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

---

**4 stage adder**

- Late arriving signal
- Two gate delays to compute CO
- Final sum and carry

---

**Carry Lookahead Circuits**

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

1111 + 0001 worst case addition

- T0: Inputs to the adder are valid
- T2: Stage 0 carry out (C1)
- T4: Stage 1 carry out (C2)
- T6: Stage 2 carry out (C3)
- T8: Stage 3 carry out (C4)

- 2 delays to compute sum
- But last carry not ready until 6 delays later

---
Carry Lookahead Logic

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

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

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

\[
S_i = A_i \text{xor} B_i \text{xor} C_i = P_i \text{xor} C_i
\]

\[
C_{i+1} = A_i B_i + A_i C_i + B_i C_i
= A_i B_i + C_i (A_i + B_i)
= A_i B_i + C_i (A_i \text{xor} B_i)
= G_i + C_i P_i
\]

Reexpress the carry logic as follows:

\[
C_1 = G_0 + P_0 C_0
\]

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

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

\[
C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0
+ P_3 P_2 P_1 P_0 C_0
\]

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!
Carry Lookahead Logic

\[ G_i = A_i B_i \quad Pi = A_i \text{xor} B_i \]

\[ C_1 = G_0 + P_0 C_0 \]
\[ = A_0B_0 \]

\[ C_2 = G_1 + P_1 C_1 \]
\[ = A_1B_1 + (A_1 \text{xor} B_1) A_0B_0 \]

\[ C_3 = G_2 + P_2 C_2 \]
\[ = A_2B_2 + (A_2 \text{xor} B_2)(A_1B_1 + (A_1 \text{xor} B_1) A_0B_0) \]

\[ C_4 = G_3 + P_3 C_3 \]
\[ = A_3B_3 + (A_3 \text{xor} B_3)(A_2B_2 + (A_2 \text{xor} B_2)(A_1B_1 + (A_1 \text{xor} B_1) A_0B_0)) \]

Carry Lookahead Implementation

Adder with Propagate and Generate Outputs

Increasingly complex logic
Cascaded Carry Lookahead

Carry lookahead logic generates individual carries

Sums computed much faster

Cascaded Carry Lookahead

4-bit adders with internal carry lookahead

Second level carry lookahead unit, extends lookahead to 16 bits
**Carry Select Adder**

Redundant hardware to make carry calculation go faster

compute the high order sums in parallel
one addition assumes carry in = 0
the other assumes carry in = 1

---

**Arithmetic Logic Unit Design**

<table>
<thead>
<tr>
<th>M = 0, Logical Bitwise Operations</th>
<th>( S_1 )</th>
<th>( S_0 )</th>
<th>Function</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>( F_i = A_i )</td>
<td>Input ( A_i ) transferred to output</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>( F_i = \neg A_i )</td>
<td>Complement of ( A_i ) transferred to output</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>( F_i = A_i \oplus B_i )</td>
<td>Compute XOR of ( A_i, B_i )</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>( F_i = A_i \oplus B_i )</td>
<td>Compute XNOR of ( A_i, B_i )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M = 1, C0 = 0, Arithmetic Operations</th>
<th>( S_1 )</th>
<th>( S_0 )</th>
<th>Function</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<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>1</td>
<td>( F = \neg A )</td>
<td>Complement of ( A ) passed to output</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>( F = A + B )</td>
<td>Sum of ( A ) and ( B )</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>( F = (\neg A) + B )</td>
<td>Sum of ( B ) and complement of ( A )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M = 1, C0 = 1, Arithmetic Operations</th>
<th>( S_1 )</th>
<th>( S_0 )</th>
<th>Function</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>( F = A + 1 )</td>
<td>Increment ( A )</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>( F = (\neg A) + 1 )</td>
<td>Two’s complement of ( A )</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>( F = A + B + 1 )</td>
<td>Increment sum of ( A ) and ( B )</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>( F = (\neg A) + B + 1 )</td>
<td>( B ) minus ( A )</td>
<td></td>
</tr>
</tbody>
</table>

Logical and Arithmetic Operations
Not all operations appear useful, but “fall out” of internal logic
Arithmetic Logic Unit Design

Sample ALU

Clever Multi-level Logic Implementation

S1 = 0 blocks Bi
Happens when operations involve Ai only

Same is true for Ci when M = 0

Addition happens when M = 1

Bi, Ci to Xor gates X2, X3

S0 = 0, X1 passes A
S0 = 1, X1 passes A

Arithmetic Mode:
Or gate inputs are Ai Ci and Bi (Ai xor Ci)

Logic Mode:
Cascaded XORs form output from Ai and Bi

74181 TTL ALU

<table>
<thead>
<tr>
<th>Selection</th>
<th>M = 1 Logic Function</th>
<th>Cn = 0</th>
<th>M = 0, Arithmetic Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>S3 S2 S1 S0</td>
<td>Fi = F(A)</td>
<td>Fi = F(A)</td>
<td>Fi = F(A)</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>Fi = A nor B</td>
<td>Fi = A nor B</td>
<td>Fi = A nor B</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>Fi = A + B</td>
<td>Fi = A + B</td>
<td>Fi = A + B</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>Fi = A + B</td>
<td>Fi = A + B</td>
<td>Fi = A + B</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>Fi = A + B</td>
<td>Fi = A + B</td>
<td>Fi = A + B</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>Fi = A + B</td>
<td>Fi = A + B</td>
<td>Fi = A + B</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>Fi = A + B</td>
<td>Fi = A + B</td>
<td>Fi = A + B</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>Fi = A + B</td>
<td>Fi = A + B</td>
<td>Fi = A + B</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>Fi = A + B</td>
<td>Fi = A + B</td>
<td>Fi = A + B</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>Fi = A + B</td>
<td>Fi = A + B</td>
<td>Fi = A + B</td>
</tr>
</tbody>
</table>
Note that the sense of the carry in and out are OPPOSITE from the input bits.

Fortunately, carry lookahead generator maintains the correct sense of the signals.

**BCD Addition**

| 5  | 0101 |
| 3  | 0011 |
| 1000 | 8  |

**Problem:** when digit sum exceeds 9

**Solution:** add 6 (0110) if sum exceeds 9!

| 5  | 0101 |
| 8  | 1000 |
| 1101 | 13 |

| 9  | 1001 |
| 7  | 0111 |
| 10000 | 16 in binary |

| 6  | 0110 |
| 10011 | 13 in BCD |
| 10110 | 16 in BCD |
**BCD Addition**

Add 0110 to sum whenever it exceeds 1001 (11XX or 1X1X)

**Combinational Multiplier**

*Basic Concept*

- **multiplicand**: 1101 (13)
- **multiplier**: * 1011 (11)

**Partial products**:
- 1101
- 1101
- 0000
- 1101

**Product of 2 4-bit numbers is an 8-bit number**

10001111 (143)
Partial Product Accumulation

<table>
<thead>
<tr>
<th></th>
<th>A3</th>
<th>A2</th>
<th>A1</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>B3</td>
<td>A2</td>
<td>A2</td>
<td>A1</td>
<td>A0</td>
</tr>
<tr>
<td>B2</td>
<td>A2</td>
<td>A1</td>
<td>A0</td>
<td>B0</td>
</tr>
<tr>
<td>B1</td>
<td>A3</td>
<td>B1</td>
<td>A1</td>
<td>B0</td>
</tr>
<tr>
<td>B0</td>
<td>A3</td>
<td>B2</td>
<td>A0</td>
<td>B2</td>
</tr>
</tbody>
</table>

S7 S6 S5 S4 S3 S2 S1 S0

Note use of parallel carry-outs to form higher order sums

12 Adders, if full adders, this is 6 gates each = 72 gates
16 gates form the partial products
total = 88 gates!
Case Study: 8 x 8 Multiplier

TTL Multipliers

Two chip implementation of 4 x 4 multiplier