L8/9: Arithmetic Structures

Lecture Material Adapted From:
- Special thanks to Kevin Atkinson, Alice Wang, Rex Min
- Lecture Notes prepared by Professor Anantha Chandrakasan
How to represent negative numbers?

- Three common schemes: sign-magnitude, ones complement, twos complement

- **Sign-magnitude:** MSB = 0 for positive, 1 for negative
  - Range: \(-(2^{N-1} - 1)\) to \(+(2^{N-1} - 1)\)
  - Two representations for zero: 0000… & 1000…
  - Simple multiplication but complicated addition/subtraction

- **Ones complement:** if N is positive then its negative is \(\overline{N}\)
  - Example: 0111 = 7, 1000 = -7
  - Range: \(-(2^{N-1} - 1)\) to \(+(2^{N-1} - 1)\)
  - Two representations for zero: 0000… & 1111…
  - Subtraction implemented as addition and negation
Twos complement = bitwise complement + 1

0111 → 1000 + 1 = 1001 = -7
1001 → 0110 + 1 = 0111 = 7

- Asymmetric range: \(-2^{N-1}\) to \(+2^{N-1}-1\)
- Only one representation for zero
- Simple addition and subtraction
- Most common representation

\[\begin{array}{cccccc}
4 & 0100 & -4 & 1100 & 4 & 0100 \\
+3 & 0011 & +(-3) & 1101 & -3 & 1101 \\
7 & 0111 & -7 & 11001 & 1 & 10001 \\
\end{array}\]

[Katz05]
Overflow Conditions

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

5 + 3 = -8!
-7 - 2 = +7!

If carry in to sign equals carry out then can ignore carry out, otherwise have overflow.
Binary Full Adder

\[ S = A \oplus B \oplus C_i \]
\[ = ABC_i + ABC_i + ABC_i + ABC_i \]
\[ C_o = AB + C_i (A+B) \]
Ripple Carry Adder Structure

Worst case propagation delay linear with the number of bits

\[ t_{\text{adder}} = (N-1)t_{\text{carry}} + t_{\text{sum}} \]
Under twos complement, subtracting B is the same as adding the bitwise complement of B then adding 1

Combination addition/subtraction system:

Overflow occurs if carry in to sign bit differs from final carry out

Add 1 for subtraction using carry in

Add/Subtract
Comparator (one approach)

A < B = N
A = B = Z
A ≤ B = Z + N
### Alternate Adder Logic Formulation

**How to Speed up the Critical (Carry) Path?**  
(How to Build a Fast Adder?)

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$C_i$</th>
<th>$S$</th>
<th>$C_o$</th>
<th>Carry status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>delete</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>delete</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>propagate</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>propagate</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>propagate</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>propagate</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>generate</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>generate</td>
</tr>
</tbody>
</table>

**Generate** ($G$) = $AB$

**Propagate** ($P$) = $A \oplus B$

$$C_o(G, P) = G + PC_i$$

$$S(G, P) = P \oplus C_i$$

Note: can also use $P = A + B$ for $C_o$
Carry Bypass Adder

Can compute $P$, $G$ in parallel for all bits

Key Idea: if $(P_0 P_1 P_2 P_3)$ then $C_{0,3} = C_{i,0}$
Assume the following for delay each gate:
P, G from A, B: 1 delay unit
P, G, Cᵢ to Cₒ or Sum for a FA: 1 delay unit
2:1 mux delay: 1 delay unit

What is the worst case propagation delay for the 16-bit adder?
For the second stage, is the critical path:

BP2 = 0 or BP2 = 1?

Message: Timing Analysis is Very Tricky – Must Carefully Consider Data Dependencies For False Paths
Re-express 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

Ripple effect has been eliminated!
Carry Lookahead Logic

Adder with propagate and generate outputs

Later stages have increasingly complex logic
Block Generate and Propagate

$G_{j:i}$ and $P_{j:i}$ denote the Generate and Propagate functions, respectively, for a group of bits from positions $i$ to $j$. We call them Block Generate and Block Propagate. $G_{j:i}$ equals 1 if the group generates a carry independent of the incoming carry. $P_{j:i}$ equals 1 if an incoming carry propagates through the entire group. For example, $G_{3:2}$ is equal to 1 if a carry is generated at bit position 3, or if a carry out is generated at bit position 2 and propagates through position 3. $G_{3:2} = G_3 + P_3 G_2$. $P_{3:2}$ is true if an incoming carry propagates through both bit positions 2 and 3. $P_{3:2} = P_3 P_2$

$$C_2 = (G_1 + P_1 G_0) + (P_1 P_0) C_0 = G_{1:0} + P_{1:0} C_0$$

$$C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$$

$$= (G_3 + P_3 G_2) + (P_3 P_2) C_{0,1} = G_{3:2} + P_{3:2} C_2$$

$$= G_{3:2} + P_{3:2}(G_{1:0} + P_{1:0} C_0) = G_{3:0} + P_{3:0} C_0$$

The carry out of a 4-bit block can thus be computed using only the block generate and propagate signals for each 2-bit section, plus the carry in to bit 0. The same formulation will be used to generate the carry out signals for a 16-bit adder using the block generate and propagate from 4-bit sections.
(g, p) • (g', p') = (g + pg', pp')

The above dot operator obeys the associative property, but it is not commutative

\[(G_{3:2}, P_{3:2}) = (G_{3}, P_{3}) \cdot (G_{2}, P_{2})\]

\[(C_{o, 3}, 0) = ((G_{3}, P_{3}) \cdot (G_{2}, P_{2}) \cdot (G_{1}, P_{1}) \cdot (G_{0}, P_{0})) \cdot (C_{i}, 0, 0)\]

\[(G_{3:0}, P_{3:0}) = [(G_{3}, P_{3}) \cdot (G_{2}, P_{2})] \cdot [(G_{1}, P_{1}) \cdot (G_{0}, P_{0})]\]
\[= (G_{3:2}, P_{3:2}) \cdot (G_{1:0}, P_{1:0})\]

\[(C_{o, k}, 0) = ((G_{k}, P_{k}) \cdot (G_{k-1}, P_{k-1}) \cdot \ldots \cdot (G_{0}, P_{0})) \cdot (C_{i}, 0, 0)\]
Logarithmic Look-Ahead Adder

Logarithmic Look-Ahead Adder

$t_p: O(N)$

$t_p: O(\log_2 N)$
16-bit Kogge-Stone Tree Adder

Sum Logic

Propagate, Generate Logic
Adder Performance

Delay vs. number of bits

- Ripple
- Bypass
- Select
- Lookahead
Addition of M, N-bit Numbers

\[ \begin{align*}
IN_{N-1} & \rightarrow + \rightarrow IN_{N-1} \\
IN_{N-2} & \rightarrow + \rightarrow IN_{N-2} \\
IN_{1} & \rightarrow + \rightarrow IN_{1} \\
IN_{0} & \rightarrow + \rightarrow IN_{0} \\
C_{in} = 0 & \rightarrow + \rightarrow C_{in} = 0
\end{align*} \]
### 74181 TTL 4-bit ALU (TI)

- **16 logic functions and 16 arithmetic operations**
- **Internal 4-bit carry lookahead adder**
- **Inputs can be active high or active low (active low is shown here)**
- **Carry in and out are opposite polarity from other inputs/outputs**

#### Table: Logic Functions and Arithmetic Operations

<table>
<thead>
<tr>
<th>S3</th>
<th>S2</th>
<th>S1</th>
<th>S0</th>
<th>M = H</th>
<th>M = L; ARITHMETIC OPERATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( F = A )</td>
<td>( F = A ) MINUS 1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( F = A )</td>
<td>( F = A ) MINUS 1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>( F = A + B )</td>
<td>( F = A ) MINUS 1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>( F = 1 )</td>
<td>( F = A ) MINUS 1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( F = A )</td>
<td>( F = A + B ) PLUS 1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( F = 1 )</td>
<td>( F = A + B ) PLUS 1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>( F = A + B )</td>
<td>( F = A + B ) PLUS 1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>( F = \overline{A} )</td>
<td>( F = A ) PLUS 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( F = A )</td>
<td>( F = A ) PLUS 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( F = A )</td>
<td>( F = A ) PLUS 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>( F = A + B )</td>
<td>( F = A ) PLUS 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>( F = \overline{A} )</td>
<td>( F = A ) PLUS 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( F = A )</td>
<td>( F = A ) PLUS 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( F = A )</td>
<td>( F = A ) PLUS 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>( F = A )</td>
<td>( F = A ) PLUS 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>( F = A )</td>
<td>( F = A ) PLUS 1</td>
</tr>
</tbody>
</table>

- Each bit is shifted to the next more significant position.

---

![Diagram of 74181 TTL 4-bit ALU](image-url)
74181 Addition (Active Low)

\[ \begin{align*}
\bar{G} &= A_3 B_3 + (A_3 + B_3) A_2 B_2 \\
&\quad + (A_3 + B_3) (A_2 + B_2) A_1 B_1 \\
&\quad + (A_3 + B_3) (A_2 + B_2) (A_1 + B_1) A_0 B_0 \\
\bar{P} &= (A_3 + B_3) (A_2 + B_2) (A_1 + B_1) (A_0 + B_0)
\end{align*} \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>AB</th>
<th>A+B</th>
<th>(AB)⊕(A+B)</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>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ \begin{align*}
F_0 &= A_0 \oplus B_0 \oplus \bar{C}_n \\
&= A_0 \oplus B_0 \oplus C_n \\
F_1 &= A_1 \oplus B_1 \oplus C_1 \\
F_2 &= A_2 \oplus B_2 \\
F_3 &= A_3 \oplus B_3
\end{align*} \]
- high speed carry lookahead generator
- used with 74181 to extend carry lookahead beyond 4 bits
- correctly handles the carry polarity of the 181

Active low example:

\[
C_{n+x} = \overline{G_0 \cdot P_0} + \overline{G_0 \cdot C_n}
\]
\[
= \overline{G_0 \cdot P_0} \cdot \overline{G_0 \cdot C_n}
\]
\[
= (G_0 + P_0)(G_0 + C_n) = G_0 + P_0 C_n
\]
\[
C_4 = G_{3:0} + P_{3:0} C_n
\]
\[
C_{n+y} = C_8 = G_{7:4} + P_{7:4} G_{3:0} + P_{7:4} P_{3:0} C_{i,0} = G_{7:0} + P_{7:0} C_n
\]
\[
C_{n+z} = C_{12} = G_{11:8} + P_{11:8} G_{7:4} + P_{11:8} P_{7:4} G_{3:0} + P_{11:8} P_{7:4} P_{3:0} C_n
\]
\[
= G_{11:0} + P_{11:0} C_n
\]
16-bit Carry Lookahead Schematic

181 configured for A+B: M = 0, S_{3-0} = 1001

182 computes $\overline{C_\text{in}}$ for later stages, using block G & P from earlier stages
**Binary Multiplication**

\[ \begin{array}{c}
\times \\
\hline
x_3 & x_2 & x_1 & x_0 \\
\hline
y_3 & y_2 & y_1 & y_0 \\
\hline
x_3y_0 & x_2y_0 & x_1y_0 & x_0y_0 \\
x_3y_1 & x_2y_1 & x_1y_1 & x_0y_1 \\
x_3y_2 & x_2y_2 & x_1y_2 & x_0y_2 \\
x_3y_3 & x_2y_3 & x_1y_3 & x_0y_3 \\
\hline
z_7 & z_6 & z_5 & z_4 & z_3 & z_2 & z_1 & z_0
\end{array} \]

- **Partial product computation**
- is simple (single and gate)

ӯ Partial product computation is simple (single and gate)
A Serial (Magnitude) Multiplier

Shift/LD

xBus

rst

add_out

D Q

acc_out

Shift

CLK

LD

yReg

CLK

CLK

CLK

CLK

CLK
Timing Diagram

CLK

Shift

xreg

yreg

Acc_out

X*Y

PRODUCT

PRODUCT
module serialmult(shift, clk, x, y, xy);
input shift, clk;
input [3:0] x, y;
output [7:0] xy;
reg [7:0] xReg;
reg [3:0] yReg;
reg [7:0] xBus, acc_out,
xy_int;
wire [7:0] add_out;
assign add_out = xBus +
acc_out;
assign xy = xy_int;

always @ (yReg[0] or xReg)
begin
if (yReg[0] == 1'b0) xBus = 8'b0;
else xBus = xReg;
end // if yReg[0]

always @ (posedge clk)
begin
if (shift == 1'b0)
begin
xReg <= {4'b0, x};
yReg <= y;
acc_out <= 8'b0;
xy_int <= add_out;
end
else
begin
xReg <= {xReg[6:0], 1'b0};
yReg <= {y[3], yReg[3:1]};
acc_out <= add_out;
xy_int <= xy;
end // if shift
end // always
endmodule
Assuming X and Y are 4-bit two's complement numbers:

$$X = -2^3x_3 + \sum_{i=0}^{2} x_i2^i \quad Y = -2^3y_3 + \sum_{i=0}^{2} y_i2^i$$

The product of X and Y is:

$$XY = x_3y_32^6 - \sum_{i=0}^{2} x_iy_32^{i+3} - \sum_{j=0}^{2} x_3y_j2^{j+3} + \sum_{i=0}^{2} \sum_{j=0}^{2} x_iy_j2^{i+j}$$

For two's complement, the following is true:

$$-\sum_{i=0}^{3} x_i2^i = -2^4 + \sum_{i=0}^{3} \overline{x_i}2^i + 1$$

The product then becomes:

$$XY = x_3y_32^6 + \sum_{i=0}^{2} \overline{x_i}y_32^{i+3} + 2^3 - 2^6 + \sum_{j=0}^{2} \overline{x_3}y_j2^{j+3} + 2^3 - 2^6 + \sum_{i=0}^{2} \sum_{j=0}^{2} x_i\overline{y_j}2^{i+j}$$

$$= x_3y_32^6 + \sum_{i=0}^{2} \overline{x_i}y_32^{i+3} + \sum_{j=0}^{2} \overline{x_3}y_j2^{j+3} + \sum_{i=0}^{2} \sum_{j=0}^{2} x_i\overline{y_j}2^{i+j} + 2^4 - 2^7$$

$$= -2^7 + x_3y_32^6 + (\overline{x_2y_3} + \overline{x_3y_2})2^5 + (x_1y_3 + x_3y_1 + x_2y_2 + 1)2^4$$

$$+ (\overline{x_0y_3} + \overline{x_3y_0} + x_1y_2 + x_2y_1)2^3 + (x_0y_2 + x_1y_1 + x_2y_0)2^2 + (x_0y_1 + x_1y_0)2^1$$

$$+ (x_0y_0)2^0$$
Twos Complement Multiplication

\[
x_3 \ x_2 \ x_1 \ x_0 \ \text{Multiplicand}
\]
\[
y_3 \ y_2 \ y_1 \ y_0 \ \text{Multiplier}
\]
\[
\begin{array}{cccc}
1 & y_0 & x_1 y_0 & x_0 y_0 \\
x_3 y_0 & x_2 y_0 & x_1 y_0 & x_0 y_0 \\
x_3 y_1 & x_2 y_1 & x_1 y_1 & x_0 y_1 \\
x_3 y_2 & x_2 y_2 & x_1 y_2 & x_0 y_2 \\
x_3 y_3 & x_2 y_3 & x_1 y_3 & x_0 y_3 \\
\end{array}
\]

\[
+ 1
\]
\[
\begin{array}{cccccc}
z_7 & z_6 & z_5 & z_4 & z_3 & z_2 & z_1 & z_0
\end{array}
\]

[Diagram of twos complement multiplication]
Performance of arithmetic blocks dictate the performance of a digital system

Architectural and logic transformations can enable significant speed up (e.g., adder delay from $O(N)$ to $O(\log_2(N))$

Similar concepts and formulation can be applied at the system level

**Timing analysis is tricky**: watch out for false paths!

Area-Delay trade-offs (serial vs. parallel implementations)