L8/9: Arithmetic Structures

Acknowledgements:

Materials in this lecture are courtesy of the following sources and are used with permission.

- Rex Min
- Kevin Atkinson
- Prof. Randy Katz (Unified Microelectronics Corporation Distinguished Professor in Electrical Engineering and Computer Science at the University of California, Berkeley) and Prof. Gaetano Borriello (University of Washington Department of Computer Science & Engineering) from Chapter 2 of R. Katz, G. Borriello. Contemporary Logic Design. 2nd ed. Prentice-Hall/Pearson Education, 2005.
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 \(\bar{N}\)
  - 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 Representation

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

[4] 0100  
\[+3\] 0011  
7 0111  

[4] 1100  
[(-3)] 1101  
-7 11001  

[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

\[ S = A \oplus B \oplus C_i \]
\[ = A\overline{B}C_i + A\overline{B}C_i + A\overline{B}C_i + ABC_i \]

\[ C_0 = AB + C_i (A+B) \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C_i</th>
<th>S</th>
<th>C_0</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>1</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>0</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>1</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>1</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>Cl</th>
<th>A B</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0 0</td>
</tr>
<tr>
<td>01</td>
<td>1 0</td>
</tr>
<tr>
<td>11</td>
<td>1 0</td>
</tr>
<tr>
<td>10</td>
<td>0 1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Cl</th>
<th>A B</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0 0</td>
</tr>
<tr>
<td>01</td>
<td>0 1</td>
</tr>
<tr>
<td>11</td>
<td>1 0</td>
</tr>
<tr>
<td>10</td>
<td>1 0</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Cl</th>
<th>A B</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0 0</td>
</tr>
<tr>
<td>01</td>
<td>0 1</td>
</tr>
<tr>
<td>11</td>
<td>1 0</td>
</tr>
<tr>
<td>10</td>
<td>1 1</td>
</tr>
</tbody>
</table>
```
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:

- mux selects $B$ for addition, $B^\overline{}$ for subtraction
- Overflow occurs if carry in to sign bit differs from final carry out
Comparator (one approach)

A < B = N
A = B = Z
A ≤ B = Z + N
How to Speed up the Critical (Carry) Path?  
(How to Build a Fast Adder?)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></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 \((P) = A \oplus B\)

\[C_o(G,P) = G + P C_i\]

\[S(G,P) = P \oplus C_i\]

Note: can also use \(P = A + B\) for \(C_o\)
Can compute $P$, $G$ in parallel for all bits

$\text{Key Idea: if (} P_0 \ P_1 \ P_2 \ P_3 \text{)} \text{ then } C_{0,3} = C_{i,0}$
Assume the following for delay each gate:

P, G from A, B: 1 delay unit
P, G, C_i to C_o 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:

\[ BP2 = 0 \text{ or } BP2 = 1? \]

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

$G_{ji}$ and $P_{ji}$ 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. $G_{ji}$ equals 1 if the group generates a carry independent of the incoming carry. $P_{ji}$ 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. $G_{3:2} = G_3 + P_3G_2$. $P_{3:2}$ is true if an incoming carry propagates through both bit positions 2 and 3. $P_{3:2} = P_3P_2$

\[
C_2 = (G_1 + P_1G_0) + (P_1P_0)C_0 = G_{1:0} + P_{1:0}C_0
\]

\[
C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0
\]

\[= (G_3 + P_3G_2) + (P_3P_2)C_{0,1} = G_{3:2} + P_{3:2}C_2\]

\[= G_{3:2} + P_{3:2}(G_{1:0} + P_{1:0}C_0) = G_{3:0} + P_{3:0}C_0\]

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) \bullet (g', p') = (g + pg', pp')

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

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

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

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

\[(C_{0}, k, 0) = ((G_{k}, P_{k}) \bullet (G_{k-1}, P_{k-1}) \bullet \cdots \bullet (G_{0}, P_{0})) \bullet (C_{i}, 0, 0)\]
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
Addition of $M, N$-bit Numbers

$IN_{N-1}$, $IN_{N-2}$, $IN_{N-3}$, $IN_{N-4}$

$IN_{N-1}$, $IN_{N-2}$, $IN_{N-3}$, $IN_{N-4}$

$C_{in} = 0$

$C_{in} = 0$

$C_{in} = 0$

$C_{in} = 0$
16-bit Carry Lookahead Schematic

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

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

\[
\begin{array}{cccc}
\times & x_3 & x_2 & x_1 & x_0 \\
x_3y_0 & x_2y_0 & x_1y_0 & x_0y_0 \\
x_3y_1 & x_2y_1 & x_1y_1 & x_0y_1 \\
x_3y_2 & x_2y_2 & x_1y_2 & x_0y_2 \\
x_3y_3 & x_2y_3 & x_1y_3 & x_0y_3 \\
\end{array}
\]

Partial product computation is simple (single and gate)

Result

\[z_7 \quad z_6 \quad z_5 \quad z_4 \quad z_3 \quad z_2 \quad z_1 \quad z_0\]
A Serial (Magnitude) Multiplier

Diagram of a serial magnitude multiplier circuit with inputs and outputs labeled.

Diagram includes:
- Inputs: xBus[0-7], Shift/LD, CLK
- Outputs: add_out[0-7], acc_out, yReg[0-3], CLK
- Logic gates for multiplication and addition operations

Diagram notes:
- Shift/LD control signals
- CLK timing signals
- yReg is the result register
- xBus as the input data bus
- Adder for summing inputs

Diagram symbols:
- XOR gates
- AND gates
- D flip-flops for temporary storage

Diagram features:
- Serial input processing
- Parallel output generation
- Clock synchronization for operation

Diagram flow:
- Data input on xBus flows through logic gates,
- Shift/LD controls data flow,
- CLK triggers operations,
- Adder combines signals,
- Outputs are accumulated or shifted as required.
Timing Diagram

CLK

Shift

xreg

yreg

Acc_out

X*Y
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

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
Twos Complement Multiplication

\[
\begin{array}{cccccc}
\times & x_3 & x_2 & x_1 & x_0 & \text{Multiplier} \\
& y_3 & y_2 & y_1 & y_0 & \text{Multiplier} \\
\hline
x_3 y_0 & x_2 y_0 & x_1 y_0 & x_0 y_0 & \\
x_3 y_1 & x_2 y_1 & x_1 y_1 & x_0 y_1 & \\
x_3 y_2 & x_2 y_2 & x_1 y_2 & x_0 y_2 & \\
x_3 y_3 & x_2 y_3 & x_1 y_3 & x_0 y_3 & \\
\hline
\end{array}
\]

\[+ 1\]

\[z_7 \quad z_6 \quad z_5 \quad z_4 \quad z_3 \quad z_2 \quad z_1 \quad z_0\]
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)