Chapter 8

VLSI Subsystem Design

Jin-Fu Li
Department of Electrical Engineering
National Central University
Jungli, Taiwan
Outline

- Introduction
- Datapath Operators
- Control Structures
System-Level Hierarchy

System (Top)

Complex units (cores)

Simple Components

Logic

Circuits

Silicon
Categories of Components

- Types of digital component
  - Datapath operators
  - Memory elements
  - Control structures
  - I/O cells

- Tradeoff of selection
  - Speed
  - Density
  - Programmability
  - Easy of design
  - etc
### Adder Truth Table

<table>
<thead>
<tr>
<th>C</th>
<th>A</th>
<th>B</th>
<th>A.B(G)</th>
<th>A+B(P)</th>
<th>A⊕B</th>
<th>SUM</th>
<th>CARRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<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>1</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>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</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>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Generate Signal G(A.B):** occurs when a carry output (CARRY) is internally generated within the adder.

**Propagate Signal P(A+B):** when it is true, the carry in signal C is passed to the carry output (CARRY) when C is true.
SUM = A \oplus B \oplus C
CARRY = AB + AC + BC

Single-bit schematic of SUM
Single-bit schematic of CARRY
Optimized combinational adder schematic

\[ C_{i+1} = A_iB_i + A_iC_i + B_iC_i \]
\[ S_i = (A_i + B_i + C_i).\bar{C}_{i+1} + A_iB_iC_i \]
Symmetrical optimized combinational adder schematic
Parallel adder implementations

\[ \text{Datapath - Bit-Parallel Adder} \]

\[ S^{n} \]

\[ A^{n} \]

\[ C^{n} \]

\[ B^{n} \]

\[ C^{n+1} \]
Datapath - Bit-Parallel Adder

If (Subtract == 0)
   \{ S = A + B; \}
else
   \{ S = A - B; \}
Datapath - Bit-Serial Adder

\[
\begin{array}{cccccc}
A & B & \text{Augend} & \text{Addend} & \text{Cout} & \text{Result} \\
01101 & 01001 & 00 & 10 & & \\
\end{array}
\]
Datapath - Carry-Save Adder (CSA)

SIN<3>  A<3>  CIN<2>  B<3>  COUT
SIN<2>  A<2>  CIN<1>  B<2>  S<3>
SIN<1>  A<1>  CIN<0>  B<1>  S<2>
SIN<0>  A<0>  B<0>  COUT  S<1>  CPA Adder
nc
clk
Objective
- To avoid the linear growth of the carry delay, we use a Carry Look-Ahead Adder (CLA) in which the carries can be generated in parallel.

Feature
- The Carry of each bit is generated from the propagate and the generate signals as well as the input carry.
- The propagate and the generate signals are derived from the operand $A_i$ and $B_i$ by:
  - $G_i = A_i \cdot B_i$
  - $P_i = A_i + B_i$
Carry Look-Ahead Adder

\[ C_{i+1} = A_iB_i + (A_i+B_i)C_i = G_i + P_iC_i \]

\[ C_1 = G_0 + P_0C_0 \]
\[ C_2 = G_1 + P_1G_0 + P_1P_0C_0 \]
\[ C_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0 \]
\[ C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 \]

4-bit CLA

4-bit CLA Diagram:

- **C0**
  - \( P_0 \)
  - \( G_0 \)
  - **CLG1**
  - **SG1**
  - \( S_0 \)

- **P0-P1**
  - **G0-G1**
  - **CLG2**
  - **SG2**
  - \( S_1 \)

- **P0-P2\ G0-G2**
  - **CLG3**
  - **SG3**
  - \( S_2 \)

- **P0-P3\ G0-G3**
  - **CLG4**
  - **SG4**
  - \( S_3 \)
Datapath – Carry Look-Ahead Adder

CLG1

\[
\begin{align*}
C_0 & \rightarrow P_0 \\
G_0 & \rightarrow P_0 \\
C_0 & \rightarrow G_0 \\
P_0 & \rightarrow G_0
\end{align*}
\]
Datapath – Carry Look-Ahead Adder

CLG4

\[ C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0 \]
Manchester Carry Chain

\[ C_{i+1} = G_i + P_i C_i \]
\[ G_i = A_i B_i \]
\[ P_i = A_i + B_i \]

Introduce the carry-kill bit \( K_i \); this term gets its name from the fact that if \( K_i = 1 \), then \( P_i = 0 \) and \( G_i = 0 \), so that \( C_{i+1} = 0 \); \( K_i = 1 \) thus “kills” the carry-out bit.

\[ K_i = \overline{A_i} \overline{B_i} \]

<table>
<thead>
<tr>
<th>( A_i )</th>
<th>( B_i )</th>
<th>( P_i )</th>
<th>( G_i )</th>
<th>( K_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</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>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Manchester circuit styles

Static circuit

Dynamic circuit

Dynamic Manchester chain
Extension to wide adders

If we use a brute-force approach for an 8-bit design, then the carry-out bit $C_8$ would have a term of the form

$$P_7 P_6 P_5 P_4 P_3 P_2 P_1 P_0 C_0$$

Multilevel CLA networks can improve this problem
Datapath – Carry Look-Ahead Adder

4-bit Carry Lookahead Generator

\[ G_{[i,i+3]} = G_{i+3} + P_{i+3} G_{i+2} + P_{i+3} P_{i+2} G_{i+1} + P_{i+3} P_{i+2} P_{i+1} G_i \]

\[ P_{[i,i+3]} = P_{i+3} P_{i+2} P_{i+1} P_i \]
A carry-skip adder is designed to speed up a wide adder by aiding the propagation of a carry bit around a portion of the entire adder.

\[ P_{[i,i+3]} = P_{i+3}P_{i+2}P_{i+1}P_i \]

\[ \text{Carry} = C_{i+4} + P_{[i,i+3]}C_i \]
Datapath - Carry-Select Adder

4-bit adder U0

4-bit adder U1

4-bit adder L
Datapath - Conditional-Sum Adder

A0 B0

C0=Cin

A1 B1

S0 S1 S0 S1 C0 C1 C0 C1

A2 B2

S0 S1 S0 S1 C0 C1 C0 C1

A3 B3

S0 S1 S0 S1 C0 C1 C0 C1

C0=Cin

S0 S1 S2 S3 C4
Bit-level multiplier

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>axb</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Multiplication of two 4-bit words

\[
\begin{array}{cccccc}
& a_3 & a_2 & a_1 & a_0 \\
& b_3 & b_2 & b_1 & b_0 \\
\hline
a_3 b_0 & a_2 b_0 & a_1 b_0 & a_0 b_0 \\
a_3 b_1 & a_2 b_1 & a_1 b_1 & a_0 b_1 \\
a_3 b_2 & a_2 b_2 & a_1 b_2 & a_0 b_2 \\
a_3 b_3 & a_2 b_3 & a_1 b_3 & a_0 b_3 \\
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
\]
The product $a \times b$ is given by the 8-bit result
$$p = p_7p_6p_5p_4p_3p_2p_1p_0$$

The $i$th product term $p_i$ can be expressed as
$$p_i = \sum_{i=j+k} a_j b_k + c_{i-1}$$

Alternate view of multiplication process

\[
\begin{array}{cccc}
  a_3 & a_2 & a_1 & a_0 \\
  b_3 & b_2 & b_1 & b_0 \\
\end{array}
\begin{array}{c}
  (a_3) \times (b_3) \\
  (a_2) \times (b_2) \\
  (a_1) \times (b_1) \\
  (a_0) \times (b_0) \\
\end{array}
\begin{array}{c}
  (axb_0)2^0 \\
  (axb_1)2^1 \\
  (axb_2)2^2 \\
  (axb_3)2^3 \\
\end{array}
\begin{array}{cccc}
  p_7 & p_6 & p_5 & p_4 \\
  p_3 & p_2 & p_1 & p_0 \\
\end{array}
\]
Using a product register for multiplication

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

Product register

(axb₀)₂⁰

(axb₁)₂¹

(axb₂)₂²

(axb₃)₂³
### Shift-right multiplication sequence

#### add ($a_3b_0$)

<table>
<thead>
<tr>
<th>$a_3b_0$</th>
<th>$a_2b_0$</th>
<th>$a_1b_0$</th>
<th>$a_0b_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_3b_0$</td>
<td>$a_2b_0$</td>
<td>$a_1b_0$</td>
<td>$a_0b_0$</td>
</tr>
</tbody>
</table>

#### add ($a_3b_1$)

<table>
<thead>
<tr>
<th>$a_3b_1$</th>
<th>$a_2b_1$</th>
<th>$a_1b_1$</th>
<th>$a_0b_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_x$</td>
<td>$a_3b_1$</td>
<td>$a_2b_1$</td>
<td>$a_1b_1$</td>
</tr>
<tr>
<td>$c_x$</td>
<td>$a_3b_1$</td>
<td>$a_2b_1$</td>
<td>$a_1b_1$</td>
</tr>
</tbody>
</table>

#### add ($a_3b_2$)

<table>
<thead>
<tr>
<th>$a_3b_2$</th>
<th>$a_2b_2$</th>
<th>$a_1b_2$</th>
<th>$a_0b_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_y$</td>
<td>$a_3b_2$</td>
<td>$a_2b_2$</td>
<td>$a_1b_2$</td>
</tr>
<tr>
<td>$c_y$</td>
<td>$a_3b_2$</td>
<td>$a_2b_2$</td>
<td>$a_1b_2$</td>
</tr>
</tbody>
</table>

#### add ($a_3b_3$)

<table>
<thead>
<tr>
<th>$a_3b_3$</th>
<th>$a_2b_3$</th>
<th>$a_1b_3$</th>
<th>$a_0b_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_7$</td>
<td>$a_3b_3$</td>
<td>$a_2b_3$</td>
<td>$a_1b_3$</td>
</tr>
<tr>
<td>$p_7$</td>
<td>$a_3b_3$</td>
<td>$a_2b_3$</td>
<td>$a_1b_3$</td>
</tr>
</tbody>
</table>

\[ p_{i+1} = (p_i + a2^n b_i)2^{-1} \]
Datapath - Register-Based Multiplier

- Multiplicand
- Multiplier
- MUX
- n-bit adder
- Product register (2n)
- clk
- shr

Advanced Reliable Systems (ARES) Lab.
Jin-Fu Li, EE, NCU
Consider two unsigned binary integers X and Y

\[ X = \sum_{i=0}^{n-1} X_i 2^i \quad Y = \sum_{j=0}^{n-1} Y_j 2^j \]

\[ P = X \times Y = \sum_{i=0}^{n-1} X_i 2^i \cdot \sum_{j=0}^{n-1} Y_j 2^j \]

\[ = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (X_i Y_j) 2^{i+j} \]

\[ = \sum_{k=0}^{n+n-1} P_k 2^k \]
Datapath - Array Multipliers
Datapath - Array Multipliers

X3Y0  X2Y0  X1Y0  X0Y0
P0

X3Y1  X2Y1  X1Y1  X0Y1
P1

X3Y2  X2Y2  X1Y2  X0Y2
P2

X3Y3  X2Y3  X1Y3  X0Y3
P3

P4

P5

P6

P7
Serial multiplier

1. Require MN clock cycles to produce a product for an N-bit multiplier and a M-bit multiplicand
Datapath – Serial Multiplication

Serial/parallel multiplier

1. Require M+N clock cycles to produce a product for an N-bit multiplier and a M-bit multiplicand

2. The critical path consists of the adders
Control – FSM

Moore

Mealy
FSM design procedure
- Draw the state-transition diagram
- Check the state diagram
- Write state equations (Write HDL)

An example of state-transition diagram

IDLE: (S1,S0)=(00)
WAIT: (S1,S0)=(01)
EXIT: (S1,S0)=(10)

A: car-in
C: change-ok
R: rst
Control – FSM

☐ Check the state-transition diagram
  ■ Ensure all states are represented, including the IDLE state
  ■ Check that the OR of all transitions leaving a state is TRUE. This is a simple method of determining that there is a way out of a state once entered.
  ■ Verify that the pairwise XOR of all exit transitions is TRUE. This ensures that there are not conflicting conditions that would lead to more than one exit-transition becoming active at any time.
  ■ Insert loops into any state if it is not guaranteed to otherwise change on each cycle.

☐ Formal FSM verification method
  ■ Perform conformance checking
module toll_booth(clk,rst,car_in,change_ok,green);
input          clk,rst,car_in,change_ok;
output        green;
reg[1:0]      state_reg, next_state;
parameter IDLE  = 2'b00;
parameter WAIT = 2'b01;
parameter EXIT  = 2'b11;
always @(posedge clk or posedge rst) begin
    If (rst==1'b1) state_reg<=IDLE;
    else state_reg<=next_state;
end
always @(state_reg or car_in or change_ok) begin
    case(state_reg):
        IDLE: if (car_in==1’1) begin
            next_state=WAIT;
            green=1'b0;
        end else begin
            next_state=IDLE;
        end
        WAIT: if (change==1'b1) begin
            next_state=EXIT;
            green=1'b1;
        end else begin
            next_state=WAIT;
        end
        EXIT: if (car_in==1’1) begin
            next_state=EXIT;
            green=1'b1;
        end else begin
            next_state=IDLE;
            green=1'b0;
        end
        default: begin
            next_state=IDLE;
        end
        endcase
end
endmodule