CSE140L: Components and Design Techniques for Digital Systems
Circuit Delay

- Transistors have intrinsic resistance and capacitance
- Signals take time to propagate from the input to the output of a gate
- Sometimes delays are labeled as @<delay_value> in circuit drawings
1-Bit & Multi-bit Adders

**Half Adder**

\[ S = A \oplus B \]
\[ C_{out} = AB \]

**Full Adder**

\[ S = A \oplus B \oplus C_{in} \]
\[ C_{out} = AB + AC_{in} + BC_{in} \]

**Types of multi-bit adders**

- Ripple-carry (slow)
- Carry-lookahead (faster)
- Two-level logic adder (even faster)

**Symbol**

Sources: TSR, Katz, Boriello & Vahid
Ripple-Carry Adder

- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: slow

• Ripple-carry adder delay

\[ t_{\text{ripple}} = N t_{FA} \]

where \( t_{FA} \) is the delay of a full adder
Two-level Logic Adder

- No matter how many inputs you have,
  - look at the truth table,
  - convert to Kmap,
  - apply the algorithm for two-level logic minimization

- Very fast adder, but….

- Beyond 8 inputs, a shockingly large amount of gates!
  - Number of gates increases exponentially

Ripple carry adder  Carry-lookahead adder (next slide)  Two-level logic adder

FAST

COMPLEX

Sources: TSR, Katz, Boriello & Vahid
# Carry-lookahead adders

<table>
<thead>
<tr>
<th>c4</th>
<th>c3</th>
<th>c2</th>
<th>c1</th>
<th>c0</th>
<th>Carries</th>
</tr>
</thead>
<tbody>
<tr>
<td>a3</td>
<td>a2</td>
<td>a1</td>
<td>a0</td>
<td></td>
<td>First operand</td>
</tr>
<tr>
<td>b3</td>
<td>b2</td>
<td>b1</td>
<td>b0</td>
<td></td>
<td>Second operand</td>
</tr>
<tr>
<td>s3</td>
<td>s2</td>
<td>s1</td>
<td>s0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From the very beginning I can “look ahead” into the value of carries.
Carry-lookahead adders

- Adder with **propagate** (P) and **generate** (G) outputs:

\[ Ci+1 = Ai \cdot Bi + Ci \cdot (Ai \oplus Bi) \]

\[ Ci+1 = Gi + Ci \cdot Pi \]

The carry at some level is equal to 1 if either the generate signal is equal to one or if the propagate and the previous carry are both 1.
Carry-lookahead adders

\[ Ci+1 = Ai \, Bi + Ci \, (Ai \, \text{xor} \, Bi) \]
Carry-lookahead adders

Example: 4-bit CLA adder

Gi = ai bi \quad generate
Pi = ai \text{xor} bi \quad propagate

\begin{align*}
c1 &= G0 + P0 \ c0 \\
c2 &= G1 + P1 \ c1 \\
c3 &= G2 + P2 \ c2 \\
c4 &= G3 + P3 \ c3
\end{align*}

All “G” and “P” are immediately available, but “c” are not.

So you need to make substitutions:

\begin{align*}
c1 &= G0 + P0 \ c0 \\
c2 &= G1 + P1 \ (G0 + P0 \ c0) &= G1 + P1G0 + P1P0c0 \\
c3 &= G2 + P2 \ c2 &= (\text{derive at home}) \\
c4 &= G3 + P3 \ c3 &= (\text{derive at home})
\end{align*}
Carry-lookahead adders

Propagate/Generate circuit (one per each input bit)

![Propagate/Generate circuit diagram]

Carry circuits (implement the equations derived in the previous slide)

C0 P0 G0 C1 @ 3
C0 P0 G0 P1 C2 @ 3
C0 P0 G0 P1 G1

![Carry circuit diagrams]

Note: this approach of “looking ahead” for building multi-bit operations is not limited to adders!
What does the following piece of code represent?

```verilog
always_comb
begin
    case ( sel_i )
        2'd0 : z_o = a_i;
        2'd1 : z_o = b_i;
        2'd2 : z_o = c_i;
        2'd3 : z_o = d_i;
        default : z_o = 1’bx;
    endcase
end
endmodule
```

*always_comb* implicitly assume a complete sensitivity list. It’s the same as:

```verilog
always @(*)
```

Useful for homework 3
Synchronous design: specify a clock

- When writing a module with a clock signal, use behavioral description with the clock in the sensitivity list

```verilog
always @(posedge clk) begin
    // do stuff
end
```

Useful for homework 3
Generate a clock in a testbench

- When testing a module with multiple inputs, make sure you start from a known condition
- How to generate a clock signal in a testbench?

```verilog
initial
begin
    // initialize your module inputs here
end

always
    #5 clk = ! clk;
```

Useful for homework 3