CSE 370 Spring 2006
Introduction to Digital Design
Lecture 12: Adders

Last Lecture
- PLAs and PALs

Today
- Adders

Full adder: Alternative Implementation

- Multilevel logic
  - Slower
  - Less gates
    - 2 XORs, 2 ANDs, 1 OR
  
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>S</th>
<th>Cout</th>
<th>C_out</th>
</tr>
</thead>
<tbody>
<tr>
<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>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</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>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<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>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Sum = (A ⊕ B) ⊕ Cin
Cout = ACin + BCin + AB

Half Adder

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>Sum</th>
<th>Cout</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>1</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>

Binary full adder

- 1-bit full adder
  - Computes sum, carry-out
  - Carry-in allows cascaded adders
  
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>S</th>
<th>Cout</th>
<th>C_out</th>
</tr>
</thead>
<tbody>
<tr>
<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>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</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>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<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>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Sum = Cin xor A xor B
Cout = ACin + BCin + AB

Full Adder

2-bit ripple-carry adder

<table>
<thead>
<tr>
<th>A1</th>
<th>B1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>C_in</td>
</tr>
<tr>
<td>A2</td>
<td>B2</td>
</tr>
</tbody>
</table>

1-Bit Adder

Overflow
4-bit ripple-carry adder/subtractor

- Circuit adds or subtracts
- 2s complement: \( A - B = A + (-B) = A + B' + 1 \)

Circuit adds or subtracts

Note: Can replace 2:1 muxes with XOR gates

Problem: Ripple-carry delay

- Carry propagation limits adder speed

Circuit adds or subtracts

Ripple-carry adder timing diagram

- Critical delay
  - Carry propagation
    - 1111 + 0001 = 10000 is worst case

One solution: Carry lookahead logic

- Compute all the carries in parallel
  - Derive carries from the data inputs
    - Not from intermediate carries
  - Use two-level logic
- Compute all sums in parallel
- Cascade simple adders to make large adders
- Speed improvement
  - 16-bit ripple-carry: ~32 gate delays
  - 16-bit carry-lookahead: ~8 gate delays
- Issues
  - Complex combinational logic
Full adder again

Carry-lookahead logic

- **Carry generate**: \( G_i = A_i B_i \)
- Generate carry when \( A = B = 1 \)
- **Carry propagate**: \( P_i = A_i \ xor \ B_i \)
- Propagate carry-in to carry-out when \( (A \ xor \ B) = 1 \)
- Sum and Cout in terms of generate/propagate:
  \[
  S_i = A_i \ xor \ B_i \ xor \ C_i
  = P_i \ xor \ C_i
  \]
  \[
  C_{i+1} = A_i B_i + C_i (A_i \ xor \ B_i)
  = G_i + C_i P_i
  \]

Carry-lookahead logic (cont’d)

- Re-express the carry logic in terms of \( G \) and \( P \)
  - \( C_1 = G_0 + P_0 C_0 \)
  - \( C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0 \)
  - \( C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0 \)
  - \( C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0 \)
- Implement each carry equation with two-level logic
  - Derive intermediate results directly from inputs
    - Rather than from carries
  - Allows "sum" computations to proceed in parallel

Implementing the carry-lookahead logic

Logic complexity increases with adder size
Cascaded carry-lookahead adder

- 4 four-bit adders with internal carry lookahead
- Second level lookahead extends adder to 16 bits

Another solution: Carry-select adder

- Redundant hardware speeds carry calculation
- Compute two high-order sums while waiting for carry-in (C4)
- Select correct high-order sum after receiving C4

We've finished combinational logic...

- What you should know
  - Twos complement arithmetic
  - Truth tables
  - Basic logic gates
  - Schematic diagrams
  - Timing diagrams
  - Minterm and maxterm expansions (canonical, minimized)
  - de Morgan's theorem
  - AND/OR to NAND/NOR logic conversion
  - K-maps, logic minimization, don't cares
  - Multiplexers/demultiplexers
  - PLAs/PALs
  - ROMs
  - Adders

Sequential versus combinational

Apply fixed inputs A, B
Wait for clock edge
Observe C
Wait for another clock edge
Observe C again

Combinational: C will stay the same
Sequential: C may be different
Sequential logic

- Two types
  - **Synchronous** = clocked
  - **Asynchronous** = self-timed
- Has state
  - **State** = memory
- Employs feedback
- Assumes steady-state signals
  - Signals are valid after they have settled
  - State elements hold their settled output values

Sequential versus combinational (again)

- Combinational systems are memoryless
  - Outputs depend only on the present inputs

  ![Combinational system diagram](image)

- Sequential systems have memory
  - Outputs depend on the present and the previous inputs

  ![Sequential system diagram](image)

Synchronous sequential systems

- **Memory** holds a system’s state
  - Changes in state occur at specific times
  - A periodic signal times or clocks the state changes
  - The clock period is the time between state changes

  ![Synchronous system diagram](image)

Steady-state abstraction

- Outputs retain their **settled values**
  - The clock period must be long enough for all voltages to settle to a steady state before the next state change

  ![Steady-state abstraction diagram](image)
Example: A sequential system

- Door combination lock
  - Enter 3 numbers in sequence and the door opens
  - If there is an error the lock must be reset
  - After the door opens the lock must be reset
- Inputs: Sequence of numbers, reset
- Outputs: Door open/close
- Memory: Must remember the combination