Combinational Building Blocks

- Large circuits usually have repeating structures
  - E.g., 64-bit arithmetic circuits in a CPU
- Design approach: reuse and replicate blocks
  - More practical than Karnaugh-map-style optimization
  - Optimize for circuit size over minimal depth
  - Learn from prior practice instead of first principles
  - CAD systems have “libraries” like software
- Our examples:
  - Multiplexers and friends
  - Addition and other basic arithmetic

Binary vs. One-hot Encoding

- Say we want to represent 4 possibilities
- Binary encoding: 00, 01, 10, 11
  - Fewest bits, wires
  - Combination can require complex logic
  - <50% unused patterns
- One-hot encoding: 0001, 0010, 0100, 1000
  - More like “unary” than binary
  - More wires needed
  - Combination logic is simpler
  - Many bit patterns are illegal

Encoders and Decoders

- (“Line”) Encoder: convert one-hot to binary
- (“Line”) Decoder: convert binary to one-hot

Notation conventions:
- Example: 4 combinations, 2 bits in binary
  - Binary value is \( b_1 b_0 \)
  - One-hot lines are \( w_0, w_1, w_2, \) and \( w_3 \)

2-line to 4-line Decoder

- Basic idea:
  - Each one-line corresponds to one product term

4-line to 2-line Encoder

- Basic idea:
  - Each binary bit is the OR of the one-hot lines in whose number it is set
Multiplexers and Demultiplexers

- Similar families to (de/en)coders
  - Relate \( n \) wires to \( 2^n \) wires
- But:
  - Different purpose: switch one of several values on a wire
  - Binary selector is an input to both mux and demux

2:4 Demultiplexer

- Basic idea:
  - Like decoder, but with an extra multiplexed/enable signal \( z \)

Multiplexers: Other Perspectives

- 2:1 mux is the circuit analog of if-then-else (?:)
- Another construction strategy: smaller muxes
  - 4:1 mux made out of 2:1 muxes

Quiz 2 Statistics

- \( N = 190 \) (both sections)
- Mean = 50.2
- Median = 51
- Standard dev’n = 18.1

Binary Addition

- Addition table is simple
  - But the result is not always a single bit
- Same situation as carrying in grade-school
  - Second bit of result: “carry out”
- Formulas are just XOR and AND
Half Adder

![Half Adder Diagram]

Full Adder

- What do we do with the carry?
  - Probably include it in another addition
  - Need a new input: "carry in"
  - Sum of three bits still fits in two output bits

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

Why “half” and “full”?

- A direct implementation is more common, because it has fewer delays

Ripple Carry Adder

- Basic design for multi-bit adder:
  - Chain carries from position to position

- Major disadvantage:
  - Long delay for carry propagation
  - For 64-bit add, if each adder takes time $t$, carries take $64t$

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

Carry Lookahead Ideas

- Basic tradeoff:
  - Add more gates to decrease delay

- Design principles:
  - Compute as much as possible before the carry-in is available
  - Group several bit positions together (commonly 4)
  - Fast path for transmission from group to group
  - Groups can themselves be grouped (like a tree)

Carry Lookahead Formulas

- “Generate”
  - Produces carry-out even without carry-in
  - $g = a \& b$

- “Propagate”
  - Carry-out if there’s a carry in
  - $p = a \| b$

- Basic relation:
  - $c_i = g_i \| (p_i \& c_{i-1})$

- Unrolled:
  - $c_i = g_i \| (g_i \& p_i) \| (g_i \& p_i \& p_{i-1}) \| (c_{i-1} \& p_i \& p_{i-1} \& p_{i-2})$
  - Complex, but only two-level
Basic ALU Design

- Repeated design ("slice") for each bit position
  - Slices operate in parallel except for carries
  - Control inputs select operation, same for all
  - Initial carry-in can also be controlled
- Typical supported operations:
  - Bitwise NOT, AND, OR, XOR, (NAND, NOR, XNOR, ...)
  - Add, subtract, negate, add 1
  - Shift left one (as \( a + a \))
- Not possible with this design:
  - Multiple shift, variable shift, right shift
  - Multiply, divide, modulo
  - Floating point

Barrel Shifter

- Goal: fast implementation of variable shift or rotate
- Idea 1: direct gate implementation
  - Complicated: every output depends on every input
- Idea 2: \( N, N:1 \) multiplexers
  - \( N \)-line decoder for control inputs
  - Also requires a lot of gates
- Idea 3: \((\log N)\) levels of \( 2:1 \) multiplexers
  - Shift by 0 or 4 based on \( 2^2 \) bit of shift amount, etc.
  - Fewer gates, more delay
- Idea 4: crossbar switch
  - A switch is a non-gate abstraction, but cheap in this application

Crossbar Barrel Shifter

- Introduce elements that keep state
- Cyclic connections between gates
- Makes more interesting computations possible
  - Processing changing inputs over time
  - E.g., CPU
- Raises more issues related to timing
  - Coordinating timing of operations
  - Time margins for reliable operation
  - Avoiding transient incorrect results

Paired Inverters

- Good: maintains a particular state
- Bad: no way to set

S-R Latch

- Stable state: storing 1
- Stable state: storing 0
- \( R \)
  - Reset to 0
- \( S \)
  - Set to 1
- \( Q \)
  - Normal output
- \( \overline{Q} \)
  - Inverted output
Coordination and Clock Signals
- In sequential design, must control when events occur
- Standard approach: clock signal
  - Alternates between 0 and 1
  - Same signal used throughout circuit
  - Challenge in high-speed designs: propagation speed
  - Rate controls speed of entire circuit
  - Design circuit to allow highest possible clock speed
  - Example: 3.0 GHz CPU
- Use clock to control when sequential devices “read”

Level-sensitivity
- First approach:
  - Value updated only when an enable signal (E) is high
  - Called a “level-sensitive” or “gated” device
- Example: gated S-R latch

Transparency
- Definition:
  - A device is transparent if input changes immediately propagate to the output
  - S-R latch is an example
- Transparent devices in series
  - Connect output of one latch to input of another
  - Input causes both devices to change at the same time
  - Undesirable in many situations
    - E.g., remember Y86 pipeline stages

Edge-triggered Devices
- Idea: update only on a clock “edge”:
  - Positive/rising edge: 0 to 1
  - Negative/falling edge: 1 to 0
  - One update per clock cycle
- An edge-triggered bit-storage device is a “flip-flop”
- Flip-flops in series:
  - Previous output changes only after next input is “read”
  - Leads to lock-step propagation, one flip-flop per cycle

D Flip-Flop
- Triangle indicates clock input
  - No bubble → rising edge triggered
- On edge, store the value of D (“data”)
- This was our main building block for Y86 registers
- !Q is often unused, but available for free

S-R to D
- D = 1: S = 1, R = 0
- D = 0: S = 0, R = 1
- Avoids ever having S and R together
- Trickier: how to build edge-triggering?
**Transient timing**

- What does this circuit do?
- Functional perspective:
  - \((x \& !!(x)) = (x \& !x) = 0\), useless?
- Actually, rising edge of \(x\) causes a brief output pulse
  - Fast path goes to 1 before delayed path goes to 0

---

**Master-slave D flip-flop**

- Make flip-flop out of two gated latches
- Updates only on rising clock edge
  - Master freezes first
  - Then slave is enabled