Combinational Circuits

- Circuits whose outputs depend only on current input values
  - no storage of past input values
  - no state
- Can be analyzed using laws of logic
  - Boolean algebra, similar to propositional calculus
Boolean Functions

- Functions operating on two-valued inputs giving two-valued outputs
  - 0, implemented as a low voltage level
  - 1, implemented as a high voltage level
- Function defines output value for all possible combinations of input value

Truth Tables

- Tabular definition of a Boolean function

<table>
<thead>
<tr>
<th>Logical OR</th>
<th>Logical AND</th>
<th>Logical NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>x</strong></td>
<td><strong>y</strong></td>
<td><strong>x + y</strong></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

OR gate

AND gate

inverter
Boolean Expressions

- Combination of variables, 0 and 1 literals, operators:
  \[(a \cdot b) + c\]
- Parentheses for order of evaluation
- Precedence: \(\cdot\) before +
  \[a \cdot b + c\]

Boolean Equations

- Equality relation between Boolean expressions
  - Often, LHS is a single variable name
  - The Boolean equation then defines a function of that name
  - Implemented as a combinational circuit

\[f = (x + y) \cdot \overline{z}\]
### Boolean Equations

Boolean equations and truth tables are both valid ways to define a function.

\[ f = (x + y) \cdot \overline{z} \]

Evaluate \( f \) for each combination of input values, and fill in table.

Q: How many rows in a truth table for an \( n \)-input Boolean function?

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( z )</th>
<th>( f )</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

### Minterms

- Given a truth table
- For each rows where function value is 1, form a minterm: AND of
  - variables where input is 1
  - NOT of variables where input is 0
- Form OR of minterms

\[ f = \overline{x} \cdot y \cdot \overline{z} + x \cdot \overline{y} \cdot \overline{z} + x \cdot y \cdot \overline{z} \]
P-terms

\[ x \cdot y \cdot z + x \cdot y \cdot \bar{z} + x \cdot y \cdot \bar{z} \]

- This is in sum-of-products form
- logical OR of p-terms (product terms)
- Not all p-terms are minterms
  - eg, the following also defines \( f \)
  \[ \bar{x} \cdot \bar{y} \cdot \bar{z} + \bar{x} \cdot \bar{z} \]

Equivalence

- These expressions all represent the same Boolean function
  \[ f' = (x + y) \cdot \bar{z} \]
  \[ = x \cdot \bar{y} \cdot \bar{z} + x \cdot \bar{y} \cdot \bar{z} + x \cdot y \cdot \bar{z} \]
  \[ = x \cdot \bar{y} \cdot \bar{z} + x \cdot \bar{z} \]

- The expressions are equivalent
  - Consistent substitution of variable values gives the same values for the expressions
**Optimization**

- Equivalence allows us to optimize
  - choose a different circuit that implements the same function more cheaply

- Caution: smaller gate count is not always better
  - choice depends on constraints that apply

**Complex Gates**

- All Boolean functions can be implemented using AND, OR and NOT
  - But other complex gates may meet constraints better in some fabrics

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>(x + y)</th>
<th>(x \cdot y)</th>
<th>(x \oplus y)</th>
<th>(\overline{x} \oplus y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</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>0</td>
</tr>
<tr>
<td>1</td>
<td>0</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>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Complex Gate Example

- These two expressions are equivalent:
  \[ f_1 = a \cdot b + c \quad f_2 = (a \cdot b) \cdot \overline{c} \]

- The NAND-NOR circuit is much smaller and faster in most fabrics!

Buffers

- Identity function: output = input
  - Needed for high fanout signals
Don’t Care Inputs

- Used where some inputs don’t affect the value of a function
- Example: multiplexer

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

Don’t Care Outputs

- For input combinations that can’t arise
  - don’t care if output is 0 or 1
  - let the synthesis tool choose

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>f</th>
<th>f₁</th>
<th>f₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Boolean Algebra – Axioms

<table>
<thead>
<tr>
<th>Combinative Laws</th>
<th>$x + y = y + x$</th>
<th>$x \cdot y = y \cdot x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Associative Laws</td>
<td>$(x + y) + z = x + (y + z)$</td>
<td>$(x \cdot y) \cdot z = x \cdot (y \cdot z)$</td>
</tr>
<tr>
<td>Distributive Laws</td>
<td>$x + (y \cdot z) = (x + y) \cdot (x + z)$</td>
<td>$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$</td>
</tr>
<tr>
<td>Identity Laws</td>
<td>$x + 0 = x$</td>
<td>$x \cdot 1 = x$</td>
</tr>
<tr>
<td>Complement Laws</td>
<td>$x + \overline{x} = 1$</td>
<td>$x \cdot \overline{x} = 0$</td>
</tr>
</tbody>
</table>

- Dual of a Boolean equation
  - substitute 0 for 1, 1 for 0, + for ·, · for +
  - if original is valid, dual is also valid

Hardware Interpretation

- Laws imply equivalent circuits
- Example: Associative Laws
More Useful Laws

<table>
<thead>
<tr>
<th>Law Type</th>
<th>Expression 1</th>
<th>Expression 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idempotence Laws</td>
<td>$x + x = x$</td>
<td>$x \cdot x = x$</td>
</tr>
<tr>
<td>Identity Laws</td>
<td>$x + 1 = 1$</td>
<td>$x \cdot 0 = 0$</td>
</tr>
<tr>
<td>Absorption Laws</td>
<td>$x + (x \cdot y) = x$</td>
<td>$x \cdot (x + y) = x$</td>
</tr>
<tr>
<td>DeMorgan Laws</td>
<td>$\overline{x + y} = \overline{x} \cdot \overline{y}$</td>
<td>$\overline{x \cdot y} = \overline{x} + \overline{y}$</td>
</tr>
</tbody>
</table>

Circuit Transformation

\[
f = (x + y \cdot \overline{z}) \cdot (y \cdot \overline{z})
\]

\[
= (x + y \cdot \overline{z}) \cdot (\overline{y + z})
\]

\[
= \overline{x \cdot (y + z) + \overline{y} \cdot \overline{z}}
\]

\[
= \overline{x \cdot y + x \cdot \overline{z} + \overline{0} + \overline{y} \cdot \overline{z}}
\]

\[
= \overline{x \cdot \overline{y} + x \cdot \overline{z} + 0 + \overline{y} \cdot \overline{z}}
\]

\[
= \overline{x \cdot \overline{y} + x \cdot \overline{z} + y \cdot \overline{z}}
\]

\[
= \overline{x \cdot \overline{y} + x \cdot \overline{z} + \overline{y} \cdot \overline{z}}
\]

\[
f(x, y, z) = \overline{x \cdot \overline{y} + x \cdot \overline{z} + \overline{y} \cdot \overline{z}}
\]
Optimization Methods

- How do we decide which Law to apply?
- What are we trying to optimize?
- Methods
  - Karnaugh maps, Quine-McClusky
    - minimize gate count
  - Espresso, Espresso-II, ...
    - multi-output minimization
- Manual methods are only tractable for small circuits
- Useful methods are embedded in EDA tools
  - We just specify constraints

Boolean Equations in VHDL

- Use logical operators in signal assignment statements

```
library ieee; use ieee.std_logic_1164.all;
entity circuit is
  port ( x, y, z : in std_logic;
    f : out std_logic );
end entity circuit;
architecture boolean_eqn of circuit is
begin
  f <= (x or (y and not z)) and not (y and z);
end architecture boolean_eqn;
```
**VHDL Logical Operators**

- **and** $a \land b$  
- **or** $a \lor b$  
- **nand** $a \cdot \overline{b}$  
- **nor** $\overline{a} + b$  
- **xor** $a \oplus b$  
- **xnor** $\overline{a} \oplus \overline{b}$  
- **not** $\overline{a}$

**Precedence**
- **not** has highest precedence
- remaining operators have equal precedence
- use parentheses to make order of evaluation clear

**VHDL bit values**
- '0' and '1'

---

**Boolean Equation Example**

**Air conditioner control logic**
- $heater\_on = temp\_low \cdot auto\_temp + manual\_heat$
- $cooler\_on = temp\_high \cdot auto\_temp + manual\_cool$
- $fan\_on = heater\_on + cooler\_on + manual\_fan$

```vhdl
library ieee; use ieee.std_logic_1164.all;
entity aircon is
 port (  
temp\_low, temp\_high, auto\_temp : in std_logic;  
manual\_heat, manual\_cool, manual\_fan : in std_logic;  
heater\_on, cooler\_on, fan\_on : out std_logic );
end entity aircon;
```
Boolean Equation Example

architecture eqns of aircon is
signal heater_on_tmp, cooler_on_tmp : std_logic;
begin
heater_on_tmp <= (temp_low and auto_temp) or manual_heat;
cooler_on_tmp <= (temp_high and auto_temp) or manual_cool;
fan_on <= heater_on_tmp or cooler_on_tmp or manual_fan;
heater_on <= heater_on_tmp;
cooler_on <= cooler_on_tmp;
end architecture eqns;

- Can’t read an out-mode port
  - eg, heater_on, cooler_on
  - Use an internal signal instead

Binary Coding

- How do we represent information with more than two possible values?
  - eg, numbers
  - N voltage levels? — No.
- Multiple binary signals (multiple bits)
  - \((a_1, a_0)\): \((0, 0), (0, 1), (1, 0), (1, 1)\)
    - This is a binary code
    - Each pair of values is a code word
    - Uses two signal wires for \(a_1, a_0\)
**Code Word Size**

- An *n*-bit code has $2^n$ code words
- To represent *N* possible values
  - Need at least $\lceil \log_2 N \rceil$ code word bits
  - More bits can be useful in some cases
- Example: code for inkjet printer
  - black, cyan, magenta, yellow, red, blue
  - six values, $\lceil \log_2 6 \rceil = 3$
  - black: (0, 0, 1), cyan: (0, 1, 0), magenta: (0, 1, 1), yellow: (1, 0, 0), red: (1, 0, 1), blue: (1, 1, 0)

**One-Hot Codes**

- Each code word has exactly one 1 bit
- Traffic light:
  - red: (1,0,0), yellow: (0,1,0), green: (0,0,1)
  - Three signal wires: red, yellow, green
- Each bit of a one-hot code corresponds to an encoded value
  - No hardware needed to decode values
Binary Codes in VHDL

- Multiple bits represented by a vector
- ```
  signal s: std_logic_vector(4 downto 0);
  ```
  This is a five-element signal
  ```
  s(4), s(3), s(2), s(1), s(0)
  ```
- ```
  signal a: std_logic_vector(1 to 3);
  ```
  This is a three-element signal
  ```
  a(1), a(2), a(3)
  ```

Binary Coding Example

- Traffic-light controller with 1-hot code
  - ```
    enable = 1: lights_out = lights_in
    ```
  - ```
    enable = 0: lights_out = (0, 0, 0)
    ```

```
library ieee; use ieee.std_logic_1164.all;
entity light_controller is
  port ( lights_in : in std_logic_vector(1 to 3);
         enable   : in std_logic;
         lights_out : out std_logic_vector(1 to 3) );
end entity light_controller;
```
**Binary Coding Example**

```vhdl
architecture conditional_enable of light_controller is
begin
   lights_out <= lights_in when enable = '1' else "000";
end architecture conditional_enable;
```

```vhdl
architecture and_enable of light_controller is
begin
   lights_out(1) <= lights_in(1) and enable;
   lights_out(2) <= lights_in(2) and enable;
   lights_out(3) <= lights_in(3) and enable;
end architecture and_enable;
```

---

**Bit Errors**

- Electrical noise can change logic levels
  - Bit flip: 0 → 1, 1 → 0
- If flipped signal is in a code word
  - result may be a different code word
  - or an invalid code word
  - inkjet printer, blue: (1, 1, 0) → ?: (1, 1, 1)
- Could ignore the possibility of a bit flip
  - don’t specify behavior of circuit
  - ok if probability is low, effect isn’t disastrous, and application is cost sensitive
### Fail-Safe Design

- Detect illegal code words
  - produce a safe result
- Traffic-light controller with 1-hot code
  - illegal code $\Rightarrow$ red light

\[
green = s_{\text{red}} \cdot s_{\text{yellow}} \cdot s_{\text{green}} \\
yellow = s_{\text{red}} \cdot s_{\text{yellow}} \cdot s_{\text{green}} \\
red = s_{\text{red}} \cdot s_{\text{yellow}} \cdot s_{\text{green}} + (green + yellow)
\]

### Redundant Codes

- Include extra error code words
  - each differs from a valid code word by a bit-flip
  - ensure no two valid code words are a bit-flip apart
- Detect error code words
  - take exceptional action
  - eg, stop, error light, etc
Parity

- Extend a code word with a *parity bit*
  - Even parity: even number of 1 bits
    - 001010110, 100100011
  - Odd parity: odd number of 1 bits
    - 001010111, 100100010
- To check for bit flip, count the 1s
  - even parity: 001010110 → 000010110
- What if there are two bit flips?
  - even parity: 001010110 → 000110110

Parity Using XOR Gates

- XOR gives even parity for two bits
  - extends to multiple bits, associatively
Combinational Components

- We can build complex combination components from gates
  - Decoders, encoders
  - Multiplexers
  - ...  
- Use them as subcomponents of larger systems
  - Abstraction and reuse

Decoders

- A decoder derives control signals from a binary coded signal
  - One per code word
  - Control signal is 1 when input has the corresponding code word; 0 otherwise
- For an $n$-bit code input
  - Decoder has $2^n$ outputs
- Example: $(a_3, a_2, a_1, a_0)$
  - Output for $(1, 0, 1, 1)$: $y_{11} = a_3 \cdot a_2 \cdot a_1 \cdot a_0$
Decoder Example

<table>
<thead>
<tr>
<th>Color</th>
<th>Codeword (c₂, c₁, c₀)</th>
</tr>
</thead>
<tbody>
<tr>
<td>black</td>
<td>0, 0, 1</td>
</tr>
<tr>
<td>cyan</td>
<td>0, 1, 0</td>
</tr>
<tr>
<td>magenta</td>
<td>0, 1, 1</td>
</tr>
<tr>
<td>yellow</td>
<td>1, 0, 0</td>
</tr>
<tr>
<td>red</td>
<td>1, 0, 1</td>
</tr>
<tr>
<td>blue</td>
<td>1, 1, 0</td>
</tr>
</tbody>
</table>

```vhdl
library ieee; use ieee.std_logic_1164.all;

entity ink_jet_decoder is
  port ( color2, color1, color0 : in std_logic;
         black, cyan, magenta, yellow, red, blue : out std_logic );
end entity ink_jet_decoder;

architecture eqn of ink_jet_decoder is
begin
  black   <= not color2 and not color1 and color0;
  cyan    <= not color2 and color1 and not color0;
  magenta <= not color2 and color1 and color0;
  yellow  <=     color2 and not color1 and not color0;
  red     <=     color2 and not color1 and    color0;
  blue    <=     color2 and color1 and not color0;
end architecture eqn;
```
Encoders

- An encoder encodes which of several inputs is 1
  - Assuming (for now) at most one input is 1 at a time
- What if no input is 1?
  - Separate output to indicate this condition

![Encoder Truth Table]

Encoder Example

- Burglar alarm: encode which zone is active

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity alarm is
  port ( zone : in std_logic_vector (1 to 8);
         intruder_zone : out std_logic_vector(2 downto 0);
         valid : out std_logic);
end entity alarm;
```

<table>
<thead>
<tr>
<th>Zone</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zone 1</td>
<td>0, 0, 0</td>
</tr>
<tr>
<td>Zone 2</td>
<td>0, 0, 1</td>
</tr>
<tr>
<td>Zone 3</td>
<td>0, 1, 0</td>
</tr>
<tr>
<td>Zone 4</td>
<td>0, 1, 1</td>
</tr>
<tr>
<td>Zone 5</td>
<td>1, 0, 0</td>
</tr>
<tr>
<td>Zone 6</td>
<td>1, 0, 1</td>
</tr>
<tr>
<td>Zone 7</td>
<td>1, 1, 0</td>
</tr>
<tr>
<td>Zone 8</td>
<td>1, 1, 1</td>
</tr>
</tbody>
</table>
Encoder Example

```
arithmetic eqn of alarm is
begin
  intruder_zone(2) <= zone(5) or zone(6)
    or zone(7) or zone(8);
  intruder_zone(1) <= zone(3) or zone(4)
    or zone(7) or zone(8);
  intruder_zone(0) <= zone(2) or zone(4)
    or zone(6) or zone(8);
  valid <= zone(1) or zone(2) or zone(3)
    or zone(4) or zone(5) or zone(6)
    or zone(7) or zone(8);
end arithmetic eqn;
```

Priority Encoders

- If more than one input can be 1
- Encode input that is 1 with highest priority

<table>
<thead>
<tr>
<th>zone</th>
<th>intruder_zone</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>(2)</td>
<td>(3)</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
### Priority Encoder Example

```vhdl
architecture priority_1 of alarm is
begin
intruder_zone <= "000" when zone(1) = '1' else
                 "001" when zone(2) = '1' else
                 "010" when zone(3) = '1' else
                 "011" when zone(4) = '1' else
                 "100" when zone(5) = '1' else
                 "101" when zone(6) = '1' else
                 "110" when zone(7) = '1' else
                 "111" when zone(8) = '1' else
                 "000";
valid <= zone(1) or zone(2) or zone(3) or zone(4)
       or zone(5) or zone(6) or zone(7) or zone(8);
end architecture priority_1;
```

### BCD Code

- Binary coded decimal
- 4-bit code for decimal digits

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>1</td>
<td>0001</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>6</td>
<td>0110</td>
<td>7</td>
</tr>
</tbody>
</table>
Seven-Segment Decoder

- Decodes BCD to drive a 7-segment LED or LCD display digit
- Segments: (g, f, e, d, c, b, a)

```
library ieee; use ieee.std_logic_1164.all;
entity seven_seg_decoder is
port ( bcd : in std_logic_vector (3 downto 0);
       blank : in std_logic;
       seg : out std_logic_vector (7 downto 1) );
end entity seven_seg_decoder;
```
Seven-Segment Decoder

```vhdl
architecture behavior of seven_segment_decoder is
    signal seg_tmp : std_logic_vector (7 downto 1);
begin
    with bcd select
    seg_tmp <= "0111111" when "0000", -- 0
                "0000110" when "0001", -- 1
                "1011011" when "0010", -- 2
                "1001111" when "0011", -- 3
                "1101111" when "0100", -- 4
                "1001100" when "0101", -- 5
                "1100111" when "0110", -- 6
                "1101011" when "0111", -- 7
                "1110011" when "1000", -- 8
                "1111111" when "1001", -- 9
                "1000000" when others; -- "-";
    seg <= "0000000" when blank = '1' else
           seg_tmp;
end architecture behavior;
```

Multiplexers

-Chooses between data inputs based on the select input

2-to-1 mux

<table>
<thead>
<tr>
<th>sel</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a_0</td>
</tr>
<tr>
<td>1</td>
<td>a_1</td>
</tr>
</tbody>
</table>

4-to-1 mux

<table>
<thead>
<tr>
<th>sel</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>a_0</td>
</tr>
<tr>
<td>01</td>
<td>a_1</td>
</tr>
<tr>
<td>10</td>
<td>a_2</td>
</tr>
<tr>
<td>11</td>
<td>a_3</td>
</tr>
</tbody>
</table>

two select bits

N-to-1 multiplexer needs $\lceil \log_2 N \rceil$ select bits
Multiplexer Example

```
library ieee; use ieee.std_logic_1164.all;

entity multiplexer_4_to_1 is
  port ( a : in std_logic_vector (3 downto 0);
        sel : in std_logic_vector (1 downto 0);
        z : out std_logic);
end entity multiplexer_4_to_1;

architecture eqn of multiplexer_4_to_1 is
begin
  with sel select
    z <= a(0) when "00", a(1) when "01",
        a(2) when "10", a(3) when others;
end architecture eqn;
```

Multi-bit Multiplexers

- To select between $N$ $m$-bit codeword inputs
  - Connect $m N$-input multiplexers in parallel
- Abstraction
  - Treat this as a component
Multi-bit Mux Example

```vhdl
library ieee; use ieee.std_logic_1164.all;
entity multiplexer_3bit_2_to_1 is
  port ( a0, a1 : in std_logic_vector (2 downto 0);
         sel : in std_logic;
         z : out std_logic_vector (2 downto 0) );
end entity multiplexer_3bit_2_to_1;

architecture eqn of multiplexer_3bit_2_to_1 is
begin
  z <= a0 when sel = '0' else a1;
end architecture eqn;
```

Active-Low Logic

- We’ve been using active-high logic
  - 0 (low voltage): falsehood of a condition
  - 1 (high voltage): truth of a condition
- Active-low logic
  - 0 (low voltage): truth of a condition
  - 1 (high voltage): falsehood of a condition
  - reverses the representation, not negative voltage!
- In circuit schematics, label active-low signals with overbar notation
  - eg, lamp_lit: low when lit, high when not lit
Active-Low Example

- Night-light circuit, lamp connected to power supply

Overbar indicates active-low

Match bubbles with active-low signals to preserve logic sense

Implied Negation

- Negation implied by connecting
  - An active-low signal to an active-high input/output
  - An active-high signal to an active-low input/output
Active-Low Signals and Gates

- DeMorgan’s laws suggest alternate views for gates
  - They’re the same electrical circuit!
  - Use the view that best represents the logical function intended
  - Match the bubbles, unless implied negation is intended

Active-Low Logic in VHDL

- Can’t draw an overbar in VHDL
  - Use _N suffix on signal or port name
  - '0' and '1' in VHDL mean low and high
  - For active-low logic
    - '0' means the condition is true
    - '1' means the condition is false
- Example
  - lamp_lit_N <= '0';
  - turns the lamp on
Combinational Verification

- Combination circuits: outputs are a function of inputs
  - Functional verification: making sure it's the right function!

Verification Testbench

Verification Example

- Verify operation of traffic-light controller
- Property to check
  - enable = '1' $\Rightarrow$ lights_out = lights_in
  - enable = '0' $\Rightarrow$ all lights are inactive
- Represent this as an assertion in the checker
Testbench Entity/Architecture

```
entity light_testbench is
end entity light_testbench;
```

```
library ieee; use ieee.std_logic_1164.all;
architecture verify of light_testbench is
  signal lights_in : std_logic_vector (1 to 3);
  signal enable : std_logic;
  signal lights_out : std_logic_vector (1 to 3);
begin
  duv : entity work.light_controller(and_enable)
    port map (lights_in, enable, lights_out);
```

Applying Test Cases

```
apply_test_cases : process is
begin
  enable <= '0'; lights_in <= "000"; wait for 1 sec;
  enable <= '0'; lights_in <= "001"; wait for 1 sec;
  enable <= '0'; lights_in <= "010"; wait for 1 sec;
  enable <= '0'; lights_in <= "100"; wait for 1 sec;
  enable <= '1'; lights_in <= "001"; wait for 1 sec;
  enable <= '1'; lights_in <= "010"; wait for 1 sec;
  enable <= '1'; lights_in <= "100"; wait for 1 sec;
  enable <= '1'; lights_in <= "000"; wait for 1 sec;
  enable <= '1'; lights_in <= "111"; wait for 1 sec;
  wait;
end process apply_test_cases;
```
### Checking Assertions

```vhDL
check_outputs : process is
  begin
    wait on enable, lights_in;
    wait for 10 ms;
    assert (enable = '1' and lights_out = lights_in)
    or (enable = '0' and lights_out = "000");
  end process check_outputs;
end architecture verify;
```

### Functional Coverage

- Did we test all possible input cases?
- For large designs, exhaustive testing is not tractable
  - \( N \) inputs: number of cases = \( 2^N \)
- Functional coverage
  - Proportion of test cases covered by a testbench
  - It can be hard to decide how much testing is enough
Summary

- Combinational logic: output values depend only on current input values
- Boolean functions: defined by truth tables and Boolean equations
- Equivalence of functions $\Rightarrow$ optimization
- Binary codes used to represent information with more than two values

Summary

- Combinational components
  - gates: AND, OR, inverter, 2-to-1 mux
  - complex gates: NAND, NOR, XOR, XNOR, AOI
  - decoder, encoder, priority encoder
- Active-low logic
- Verification testbench
  - apply test cases to DUV
  - checker contains assertions