Working with combinational logic

- Simplification
  - two-level simplification
  - exploiting don’t cares
  - algorithm for simplification
- Logic realization
  - two-level logic and canonical forms realized with NANDs and NORs
  - multi-level logic, converting between ANDs and ORs
- Time behavior
- Hardware description languages

Design example: two-bit comparator

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>LT</th>
<th>EQ</th>
<th>GT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</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>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>
<td>0</td>
</tr>
</tbody>
</table>

A B C D LT EQ GT
0 0 0 0 0 1 0
0 1 1 0 0 0 1
1 0 1 0 0 0 1
1 1 1 0 0 0 1

A B C D LT EQ GT
0 1 0 0 0 1
0 1 0 1 0 0 1
1 0 1 0 0 0 1
1 1 1 0 0 0 1

A B C D LT EQ GT
0 0 0 0 0 1
0 1 0 0 0 1
1 0 0 0 0 1
1 1 0 0 1 0

we'll need a 4-variable Karnaugh map for each of the 3 output functions
Design example: two-bit comparator (cont’d)

\[
\begin{align*}
    \text{LT} &= A' B' D + A' C + B' C D \\
    \text{EQ} &= A' B' C' D' + A' B C' D + A B C D + A B' C D' = (A \text{xnor } C) \cdot (B \text{xnor } D) \\
    \text{GT} &= B C' D' + A C' + A B D'
\end{align*}
\]

LT and GT are similar (flip A/C and B/D)

Design example: two-bit comparator (cont’d)

two alternative implementations of EQ with and without XOR

XNOR is implemented with at least 3 simple gates
Design example: 2x2-bit multiplier

<table>
<thead>
<tr>
<th>A2</th>
<th>A1</th>
<th>B2</th>
<th>B1</th>
<th>P8</th>
<th>P4</th>
<th>P2</th>
<th>P1</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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</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>1</td>
<td>1</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>1</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>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</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>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>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

4-variable K-map for each of the 4 output functions

Design example: 2x2-bit multiplier (cont’d)

P8 = A2A1B2B1


P1 = A1B1
Design example: BCD increment by 1

<table>
<thead>
<tr>
<th>I8</th>
<th>I4</th>
<th>I2</th>
<th>I1</th>
<th>O8</th>
<th>O4</th>
<th>O2</th>
<th>O1</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>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<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>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</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>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>0</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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

4-variable K-map for each of the 4 output functions

Design example: BCD increment by 1 (cont’d)

O8 = I4 I2 I1 + I8 I1’
O4 = I4 I2’ + I4 I1’ + I4’ I2 I1
O2 = I8’ I2’ I1 + I2 I1’
O1 = I1’
Definition of terms for two-level simplification

- **Implicant**
  - single element of ON-set or DC-set or any group of these elements that can be combined to form a subcube
- **Prime implicant**
  - implicant that can't be combined with another to form a larger subcube
- **Essential prime implicant**
  - prime implicant is essential if it alone covers an element of ON-set
  - will participate in ALL possible covers of the ON-set
  - DC-set used to form prime implicants but not to make implicant essential
- **Objective:**
  - grow implicant into prime implicants
    - (minimize literals per term)
  - cover the ON-set with as few prime implicants as possible
    - (minimize number of product terms)

Examples to illustrate terms

- 6 prime implicants: 
  - $A'B'D$, $BC'$, $AC$, $A'C'D$, $AB$, $B'CD$
  - minimum cover: $AC + BC' + A'B'D$

- 5 prime implicants: 
  - $BD$, $ABC'$, $ACD$, $A'BC$, $A'C'D$
  - minimum cover: 4 essential implicants
Algorithm for two-level simplification

- Algorithm: minimum sum-of-products expression from a Karnaugh map
  - Step 1: choose an element of the ON-set
  - Step 2: find "maximal" groupings of 1s and Xs adjacent to that element
    - consider top/bottom row, left/right column, and corner adjacencies
    - this forms prime implicants (number of elements always a power of 2)
  - Repeat Steps 1 and 2 to find all prime implicants
  - Step 3: revisit the 1s in the K-map
    - if covered by single prime implicant, it is essential, and participates in final cover
    - 1s covered by essential prime implicant do not need to be revisited
  - Step 4: if there remain 1s not covered by essential prime implicants
    - select the smallest number of prime implicants that cover the remaining 1s

Algorithm for two-level simplification (example)
Activity

- List all prime implicants for the following K-map:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Which are essential prime implicants?

- What is the minimum cover?

Implementations of two-level logic

- Sum-of-products
  - AND gates to form product terms (minterms)
  - OR gate to form sum

- Product-of-sums
  - OR gates to form sum terms (maxterms)
  - AND gates to form product
Two-level logic using NAND gates

- Replace minterm AND gates with NAND gates
- Place compensating inversion at inputs of OR gate

Two-level logic using NAND gates (cont’d)

- OR gate with inverted inputs is a NAND gate
  - de Morgan’s: \( A' + B' = (A \cdot B)' \)
- Two-level NAND-NAND network
  - inverted inputs are not counted
  - in a typical circuit, inversion is done once and signal distributed
Two-level logic using NOR gates

- Replace maxterm OR gates with NOR gates
- Place compensating inversion at inputs of AND gate

Two-level logic using NOR gates (cont’d)

- AND gate with inverted inputs is a NOR gate
  - de Morgan’s: \( A' \cdot B' = (A + B)' \)
- Two-level NOR-NOR network
  - inverted inputs are not counted
  - in a typical circuit, inversion is done once and signal distributed
Two-level logic using NAND and NOR gates

- NAND-NAND and NOR-NOR networks
  - de Morgan’s law: \((A + B)' = A' \cdot B'\) \(\quad\) \((A \cdot B)' = A' + B'\)
  - written differently: \(A + B = (A' \cdot B')'\) \(\quad\) \((A \cdot B) = (A' + B')'\)

- In other words —
  - OR is the same as NAND with complemented inputs
  - AND is the same as NOR with complemented inputs
  - NAND is the same as OR with complemented inputs
  - NOR is the same as AND with complemented inputs

Conversion between forms

- Convert from networks of ANDs and ORs to networks of NANDs and NORs
  - introduce appropriate inversions ("bubbles")
- Each introduced "bubble" must be matched by a corresponding "bubble"
  - conservation of inversions
  - do not alter logic function
- Example: AND/OR to NAND/NAND

\[ Z = (A \cdot B) + (C \cdot D) \]
\[ Z = (A' + B') (C' + D') \]
Conversion between forms (cont’d)

- Example: verify equivalence of two forms

\[ Z = [ (A \land B)' \cdot (C \land D)' ]' \]
\[ = [ (A' + B') \cdot (C' + D') ]' \]
\[ = [ (A' + B')' + (C' + D')' ] \]
\[ = (A \land B) + (C \land D) \checkmark \]

Conversion between forms (cont’d)

- Example: map AND/OR network to NOR/NOR network

Step 1: preserve "bubbles"

Step 2: conserve "bubbles"
Conversion between forms (cont’d)

Example: verify equivalence of two forms

\[ Z = \left( (A' + B')' + (C' + D')' \right)' \]
\[ = \left( A' + B' \right) \cdot \left( C' + D' \right)' \]
\[ = \left( A' + B' \right)' + \left( C' + D' \right)' \]
\[ = (A \cdot B) + (C \cdot D) \checkmark \]

Multi-level logic

- \[ x = A \cdot D \cdot F + A \cdot E \cdot F + B \cdot D \cdot F + B \cdot E \cdot F + C \cdot D \cdot F + C \cdot E \cdot F + G \]
  - reduced sum-of-products form – already simplified
  - 6 x 3-input AND gates + 1 x 7-input OR gate (that may not even exist!)
  - 25 wires (19 literals plus 6 internal wires)
- \[ x = (A + B + C) \cdot (D + E) \cdot F + G \]
  - factored form – not written as two-level S-o-P
  - 1 x 3-input OR gate, 2 x 2-input OR gates, 1 x 3-input AND gate
  - 10 wires (7 literals plus 3 internal wires)
Conversion of multi-level logic to NAND gates

\[ F = A (B + C D) + B C' \]

Conversion of multi-level logic to NORs

\[ F = A (B + C D) + B C' \]
Conversion between forms

Example

(a) original circuit

(b) add double bubbles at inputs

(c) distribute bubbles
some mismatches

(d) insert inverters to fix mismatches

AND-OR-invert gates

- AOI function: three stages of logic — AND, OR, Invert
  - multiple gates "packaged" as a single circuit block

logical concept

possible implementation

2x2 AOI gate symbol

3x2 AOI gate symbol
Conversion to AOI forms

- General procedure to place in AOI form
  - compute the complement of the function in sum-of-products form
  - by grouping the 0s in the Karnaugh map
- Example: XOR implementation
  - \( A \text{ xor } B = A' B + A B' \)
  - AOI form:
    - \( F = (A' B' + A B)' \)

\[ \begin{array}{c|c|c|c}
   & A' & B' & \text{\&} \\
\hline
A & & & \\
B & & & \\
\hline
\end{array} \]
\[ \begin{array}{c|c|c|c|c|c|c}
   & & & & & & F \\
\hline
& & & & & & \text{+} \\
\hline
\end{array} \]

Examples of using AOI gates

- Example:
  - \( F = A B + A C' + B C' \)
  - \( F = (A' B' + A' C + B' C)' \)
  - Implemented by 2-input 3-stack AOI gate

  - \( F = (A + B) (A + C') (B + C') \)
  - \( F = [(A' + B') (A' + C) (B' + C)]' \)
  - Implemented by 2-input 3-stack AOI gate

- Example: 4-bit equality function
  - \( Z = (A0 B0 + A0' B0')(A1 B1 + A1' B1')(A2 B2 + A2' B2')(A3 B3 + A3' B3') \)
  - each implemented in a single 2x2 AOI gate
Examples of using AOI gates (cont’d)

- Example: AOI implementation of 4-bit equality function

```
+  &  &
&  &
A0 B0
A1 B1
A2 B2
A3 B3

- high if A0 ≠ B0
- low if A0 = B0
- conservation of bubbles

Z = NOR
- if all inputs are low
- then Ai = Bi, i=0,...,3
- output Z is high
```

Summary for multi-level logic

- Advantages
  - circuits may be smaller
  - gates have smaller fan-in
  - circuits may be faster
- Disadvantages
  - more difficult to design
  - tools for optimization are not as good as for two-level
  - analysis is more complex
Time behavior of combinational networks

- Waveforms
  - visualization of values carried on signal wires over time
  - useful in explaining sequences of events (changes in value)
- Simulation tools are used to create these waveforms
  - input to the simulator includes gates and their connections
  - input stimulus, that is, input signal waveforms
- Some terms
  - gate delay — time for change at input to cause change at output
    - min delay – typical/nominal delay – max delay
    - careful designers design for the worst case
  - rise time — time for output to transition from low to high voltage
  - fall time — time for output to transition from high to low voltage
  - pulse width — time that an output stays high or stays low between changes

Momentary changes in outputs

- Can be useful — pulse shaping circuits
- Can be a problem — incorrect circuit operation (glitches/hazards)
- Example: pulse shaping circuit
  - \( A' \cdot A = 0 \)
  - delays matter

\[ D \text{ remains high for three gate delays after } A \text{ changes from low to high} \]
\[ F \text{ is not always 0 pulse 3 gate-delays wide} \]
Oscillatory behavior

- Another pulse shaping circuit

Hardware description languages

- Describe hardware at varying levels of abstraction
- Structural description
  - textual replacement for schematic
  - hierarchical composition of modules from primitives
- Behavioral/functional description
  - describe what module does, not how
  - synthesis generates circuit for module
- Simulation semantics
**HDLs**

- **Abel** (circa 1983) - developed by Data-I/O
  - targeted to programmable logic devices
  - not good for much more than state machines
- **ISP** (circa 1977) - research project at CMU
  - simulation, but no synthesis
- **Verilog** (circa 1985) - developed by Gateway (absorbed by Cadence)
  - similar to Pascal and C
  - delays is only interaction with simulator
  - fairly efficient and easy to write
  - IEEE standard
- **VHDL** (circa 1987) - DoD sponsored standard
  - similar to Ada (emphasis on re-use and maintainability)
  - simulation semantics visible
  - very general but verbose
  - IEEE standard

**Verilog**

- Supports structural and behavioral descriptions
- **Structural**
  - explicit structure of the circuit
  - e.g., each logic gate instantiated and connected to others
- **Behavioral**
  - program describes input/output behavior of circuit
  - many structural implementations could have same behavior
  - e.g., different implementation of one Boolean function
- We’ll mostly be using behavioral Verilog in Aldec ActiveHDL
  - rely on schematic when we want structural descriptions
### Structural model

```verilog
class xor_gate (out, a, b);
  input a, b;
  output out;
  wire abar, bbar, t1, t2;

  inverter invA (abar, a);
  inverter invB (bbar, b);
  and_gate and1 (t1, a, bbar);
  and_gate and2 (t2, b, abar);
  or_gate or1 (out, t1, t2);
endmodule
```

### Simple behavioral model

- **Continuous assignment**

```verilog
class xor_gate (out, a, b);
  input a, b;
  output out;
  reg out;

  assign #6 out = a ^ b;
endmodule
```
Simple behavioral model

- always block

```verbatim
module xor_gate (out, a, b);
  input         a, b;
  output        out;
  reg           out;

  always @(a or b) begin
      #6 out = a ^ b;
  end
endmodule
```

This example will not work. Why not?

```verbatim
//swap bytes in word
always @(posedge clk)
begin
  word[15:8] = word[ 7:0];
  word[ 7:0] = word[15:8];
end
```
Non-blocking assignment

- The <= token represents a **non-blocking** assignment
- > Evaluated and assigned in two steps: The right-hand side is evaluated immediately
- > The assignment to the left-hand side is postponed until other evaluations in the current time step are completed
- > Execution flow within the procedure continues until a timing control is encountered (flow is not blocked)
- This example will work. Why?
  - //swap bytes in word
  - always @(posedge clk)
  - begin
  - word[15:8] <= word[ 7:0];
  - word[ 7:0] <= word[15:8];
  - end

Driving a simulation through a “testbench”

```verilog
module testbench (x, y);

output        x, y;
reg [1:0]     cnt;

initial begin
  cnt = 0;
  repeat (4) begin
    #10 cnt = cnt + 1;
    $display (@ time=%d, x=%b, y=%b, cnt=%b;
    $time, x, y, cnt); end
    #10 $finish;
  end

assign x = cnt[1];
assign y = cnt[0];
endmodule
```

2-bit vector
initial block executed only once at start of simulation
directive to stop simulation
print to a console
Complete simulation

- Instantiate stimulus component and device to test in a schematic

```vhdl
module Compare1 (Equal, Alarger, Blarger, A, B);
  input     A, B;
  output    Equal, Alarger, Blarger;
  assign #5 Equal = (A & B) | (~A & ~B);
  assign #3 Alarger = (A & ~B);
  assign #3 Blarger = (~A & B);
endmodule
```

Comparator example

```vhdl
module Compare1 (Equal, Alarger, Blarger, A, B);
  input     A, B;
  output    Equal, Alarger, Blarger;
  assign #5 Equal = (A & B) | (~A & ~B);
  assign #3 Alarger = (A & ~B);
  assign #3 Blarger = (~A & B);
endmodule
```
More complex behavioral model

module life (n0, n1, n2, n3, n4, n5, n6, n7, self, out);
input n0, n1, n2, n3, n4, n5, n6, n7, self;
output out;
reg out;
reg [7:0] neighbors;
reg [3:0] count;
reg [3:0] i;

assign neighbors = {n7, n6, n5, n4, n3, n2, n1, n0};

always @(neighbors or self) begin
    count = 0;
    for (i = 0; i < 8; i = i+1) count = count + neighbors[i];
    out = (count == 3);
    out = out | ((self == 1) & (count == 2));
end
endmodule

Hardware description languages vs. programming languages

- Program structure
  - instantiation of multiple components of the same type
  - specify interconnections between modules via schematic
  - hierarchy of modules (only leaves can be HDL in Aldec ActiveHDL)
- Assignment
  - continuous assignment (logic always computes)
  - propagation delay (computations take time)
  - timing of signals is important (when does computation have its effect)
- Data structures
  - size explicitly spelled out - no dynamic structures
  - no pointers
- Parallelism
  - hardware is naturally parallel (must support multiple threads)
  - assignments can occur in parallel (not just sequentially)
Hardware description languages and combinational logic

- Modules - specification of inputs, outputs, bidirectional, and internal signals
- Continuous assignment - a gate’s output is a function of its inputs at all times (doesn’t need to wait to be "called")
- Propagation delay - concept of time and delay in input affecting gate output
- Composition - connecting modules together with wires
- Hierarchy - modules encapsulate functional blocks

Working with combinational logic summary

- Design problems
  - filling in truth tables
  - incompletely specified functions
  - simplifying two-level logic
- Realizing two-level logic
  - NAND and NOR networks
  - networks of Boolean functions and their time behavior
- Time behavior
- Hardware description languages
- Later
  - combinational logic technologies
  - more design case studies