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: 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 0 0 0</td>
<td>0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>0 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>0 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>0 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4-variable K-map for each of the 4 output functions
Design example: 2x2-bit multiplier (activity)

- K-map for P8
- K-map for P4
- K-map for P2
- K-map for P1

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

5 prime implicants:
BD, ABC', ACD, A'B'D

essential

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)

2 primes around $A'BC'D'$

2 primes around $ABC'D'$

2 primes around $ABC'D'$
Algorithm for two-level simplification (example)

3 primes around AB'C'D'
Algorithm for two-level simplification
(example)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

List all prime implicants for the following K-map:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

- Which are essential prime implicants?
- What is the minimum cover?
Activity

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

- Which are essential prime implicants? CD' BD AC'D

- What is the minimum cover? CD' BD AC'D

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
**Why NANDs and NORs**

- CMOS technology makes it easier to build NANDs and NORs than ANDs and ORs
- MOS transistors have three terminals: drain, gate, and source
  - N-type pass “0” well, P-type pass “1” well

  ![n-channel and p-channel transistors](image)

  - n-channel: open when voltage at G is low, closed when voltage \( G > S/D + \varepsilon \)
  - p-channel: closed when voltage at G is low, open when voltage \( G < S/D - \varepsilon \)

**A simple MOS transistor network (1-input)**

![MOS transistor network](image)

- what is the relationship between x and y?
- | x  | z   |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0v</td>
<td>0v</td>
</tr>
<tr>
<td>3v</td>
<td>3v</td>
</tr>
</tbody>
</table>

Autumn 2006  CSE370 - III - Working with Combinational Logic
What is the relationship between X, Y and Z1 and Z2?

Two-level logic using NAND and NOR gates

- NAND-NAND and NOR-NOR networks
  - de Morgan’s law: $(A + B)' = A' \cdot B'$  
    $(A \cdot B)' = A' + B'$
  - written differently: $A + B = (A' \cdot B')'$  
    $(A \cdot B) = (A' + B')'$
- 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
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 (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
Conversion between forms (cont’d)

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

\[ Z = \{ (A' + B')' + (C' + D')' \}' \]
\[ = \{ (A' + B') \cdot (C' + D') \}' \]
\[ = (A' + B')' + (C' + D')' \]
\[ = (A \cdot B) + (C \cdot D) \]

Conversion between forms (cont’d)

- Example: verify equivalence of two forms

\[ Z = \{ (A' + B')' + (C' + D')' \}' \]
\[ = \{ (A' + B') \cdot (C' + D') \}' \]
\[ = (A' + B')' + (C' + D')' \]
\[ = (A \cdot B) + (C \cdot D) \]
Activity: convert to NAND gates

- Example

(a) Original circuit

(b) Add double bubbles at inputs

(c) Distribute bubbles; some mismatches

(d) Insert inverters to fix mismatches
Multi-level logic

- \[ x = A D F + A E F + B D F + B E F + C D F + C E F + G \]
  - 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)
- \[ x = (A + B + C) (D + E) F + G \]
  - 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

- \[ F = A (B + C D) + B C' \]
  - original AND-OR network
  - introduction and conservation of bubbles
  - redrawn in terms of conventional NAND gates
Conversion of multi-level logic to NORs

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

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
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 remains high for three gate delays after A changes from low to high
F 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
module 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
module xor_gate (out, a, b);
  input a, b;
  output out;

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

- always block

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

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

Driving a simulation through a “testbench”

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);
      #10 $finish;
    end
  assign x = cnt[1];
  assign y = cnt[0];
endmodule
Complete simulation

- Instantiate stimulus component and device to test in a schematic

Comparator example

```verilog
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
```
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 (computation takes 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