Combinational logic

- Logic functions, truth tables, and switches
  - NOT, AND, OR, NAND, NOR, XOR, ...
  - Minimal set
- Axioms and theorems of Boolean algebra
  - Proofs by re-writing
  - Proofs by perfect induction
- Gate logic
  - Networks of Boolean functions
  - Time behavior
- Canonical forms
  - Two-level
  - Incompletely specified functions
- Simplification
  - Boolean cubes and Karnaugh maps
  - Two-level simplification

Possible logic functions of two variables

There are 16 possible functions of 2 input variables:
in general, there are $2^{2^{n}}$ functions of $n$ inputs

- $X$ and $Y$
- $X$ or $Y$
- not $Y$
- not $X$
- $X$ nor $Y$
- $X$ nand $Y$
- not $(X$ or $Y)$
- not $(X$ and $Y)$

Cost of different logic functions

- Different functions are easier or harder to implement
  - Each has a cost associated with the number of switches needed
  - 0 (F0) and 1 (F15): require 0 switches, directly connect output to low/high
  - $X$ (F3) and $Y$ (F5): require 0 switches, output is one of inputs
  - $X'$ (F12) and $Y'$ (F10): require 2 switches for "inverter" or NOT-gate
  - $X$ nor $Y$ (F4) and $X$ nand $Y$ (F14): require 4 switches
  - $X$ or $Y$ (F7) and $X$ and $Y$ (F1): require 6 switches
  - $X = Y$ (F9) and $X \oplus Y$ (F6): require 16 switches

   - Because NOT, NOR, and NAND are the cheapest they are the functions we implement the most in practice

An algebraic structure

- An algebraic structure consists of
  - a set of elements $B$
  - binary operations $\{+,*\}$
  - and a unary operation $\{\prime\}$

  - such that the following axioms hold:

  1. set $B$ contains at least two elements, $a$, $b$, such that $a \neq b$
  2. closure: $a + b$ is in $B$, $a \cdot b$ is in $B$
  3. commutativity: $a + b = b + a$, $a \cdot b = b \cdot a$
  4. associativity: $(a + (b + c)) = (a + b) + c$, $(a \cdot (b \cdot c)) = (a \cdot b) \cdot c$
  5. identity: $a + 0 = a$, $a \cdot 1 = a$
  6. distributivity: $a + (b \cdot c) = (a + b) \cdot (a + c)$, $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
  7. complementarity: $a + a' = 1$, $a \cdot a' = 0$
Boolean algebra

• Boolean algebra
  – B = {0, 1}
  – + is logical OR, • is logical AND
  – ’ is logical NOT
• All algebraic axioms hold

Logic functions and Boolean algebra

• Any logic function that can be expressed as a truth table can be written as an expression in Boolean algebra using the operators: ’, +, and •

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X • Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X'</th>
<th>X • Y'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</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>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[(X • Y) + (X' • Y') = 1 \text{ if } X = Y\]

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X • Y</th>
<th>(X • Y) • (X' • Y')</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

X, Y are Boolean algebra variables
Boolean expression that is true when the variables X and Y have the same value and false, otherwise

Axioms and theorems of Boolean algebra

• Identity
  1. \(X + 0 = X\)
  2. \(X' + 1 = X\)

• Null
  1. \(X + 0 = X\)
  2. \(X • 0 = 0\)

• Idempotency
  3. \(X' + X = X\)
  3. \(X • X = X\)

• Involution
  4. \((X')' = X\)

• Complementarity
  5. \(X + X' = 1\)
  5. \(X • X' = 0\)

• Commutativity
  6. \(X + Y = Y + X\)
  6. \(X • Y = Y • X\)

• Associativity
  7. \((X + Y) + Z = X + (Y + Z)\)
  7. \((X • Y) • Z = X • (Y • Z)\)

• Distributivity
  8. \((X • Y) + (X • Z) = (X + Y) • (X + Z)\)

• Uniting
  9. \((X • Y) + (X • Y') = X\)
  9. \((X + Y) • (X + Y') = X\)

• Absorption
  10. \((X + Y) • (X + Y') = Y\)
  10. \((X + Y) • (X + Y') = X\)

• Factoring
  11. \((X + Y) • (Y + Z) = (X • Z) • (X' • Y)\)
  11. \((X • Y) • (X' • Y') = (X + Z) • (X' + Y)\)

• Consensus
  12. \((X + Y) • (Y + Z) = (X • Z) • (X' + Y)\)
  13. \((X + Y) • (Y + Z) = (X • Z) • (X' + Y)\)
Axioms and theorems of Boolean algebra (cont')

- de Morgan's:
  \[ (X + Y + ...)' = X' \cdot Y' \cdot ... \]
  \[ 14D. (X \cdot Y \cdot ...)' = X' + Y' + ... \]
- generalized de Morgan's:
  \[ 15. f(X_1,X_2,\ldots,X_n,0,1,+,\cdot) = f(X'_1,X'_2,\ldots,X'_n,1,0,\cdot,+) \]
- establishes relationship between \( \cdot \) and +

- Duality
  - Dual of a Boolean expression is derived by replacing \( \cdot \) by +, + by \( \cdot \), 0 by 1, and 1 by 0, and leaving variables unchanged
  - Any theorem that can be proven is thus also proven for its dual!
  - Meta-theorem (a theorem about theorems)
    - duality:
      \[ 16. X + Y + ... \iff X \cdot Y + ... \]
    - generalized duality:
      \[ 17. f(X_1,X_2,\ldots,X_n,0,1,+,\cdot) \iff f(X_1,X_2,\ldots,X_n,1,0,\cdot,+ \)]
- Different than deMorgan’s Law
  - this is a statement about theorems
  - this is not a way to manipulate (re-write) expressions

Proving theorems (rewriting)

- Using the axioms of Boolean algebra:
  - e.g., prove the theorem: \( X \cdot Y + X \cdot Y' = X \)
    - distributivity (8)
    - complementarity (5)
    - identity (1D)

- e.g., prove the theorem: \( X + X \cdot Y = X \)
  - identity (1D)
  - distributivity (8)
  - identity (2)
  - identity (ID)

Proving theorems (perfect induction)

- Using perfect induction (complete truth table):
  - e.g., de Morgan’s:
    \[
    \begin{array}{cccc|cc}
    X & Y & X' & Y' \\
    \hline
    0 & 0 & 1 & 1 \\
    0 & 1 & 0 & 0 \\
    1 & 0 & 0 & 0 \\
    1 & 1 & 1 & 1 \\
    \end{array}
    \]
    - NOR is equivalent to AND
    - with inputs complemented
    - NAND is equivalent to OR
    - with inputs complemented
A simple example

- 1-bit binary adder
  - inputs: A, B, Carry-in
  - outputs: Sum, Carry-out

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>S</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>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</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>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</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>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ S = A' B' Cin + A' B Cin' + A B' Cin + A B Cin \]
\[ Cout = A' B Cin + A B' Cin + A B Cin' + A B Cin \]

From Boolean expressions to logic gates

- More than one way to map expressions to gates
  - e.g., \( Z = A' \cdot B' \cdot (C \cdot D) = (A' \cdot (B' \cdot (C + D))) \)

use of 3-input gate

Waveform view of logic functions

- Looks like a sideways truth table
  - but note how edges don't line up exactly
  - it takes time for a gate to switch its output!

Choosing different realizations of a function

- two-level realization (we don’t count NOT gates)
- multi-level realization (gates with fewer inputs)
- XOR gate (easier to draw but costlier to build)
Which realization is best?

- Reduce number of inputs
  - literal: input variable (complemented or not)
    - can approximate cost of logic gate as 2 transistors per literal
    - why not count inverters?
  - Fewer literals means less transistors
    - smaller circuits
  - Fewer inputs implies faster gates
    - gates are smaller and thus also faster
  - Fan-ins (# of gate inputs) are limited in some technologies
- Reduce number of gates
  - Fewer gates (and the packages they come in) means smaller circuits
    - directly influences manufacturing costs

Which is the best realization? (cont’d)

- Reduce number of levels of gates
  - Fewer level of gates implies reduced signal propagation delays
  - Minimum delay configuration typically requires more gates
    - wider, less deep circuits
- How do we explore tradeoffs between increased circuit delay and size?
  - Automated tools to generate different solutions
  - Logic minimization: reduce number of gates and complexity
  - Logic optimization: reduction while trading off against delay

Are all realizations equivalent?

- Under the same input stimuli, the three alternative implementations have almost the same waveform behavior
  - delays are different
  - glitches (hazards) may arise
  - variations due to differences in number of gate levels and structure
- Three implementations are functionally equivalent

Implementing Boolean functions

- Technology independent
  - Canonical forms
  - Two-level forms
  - Multi-level forms
- Technology choices
  - Packages of a few gates
  - Regular logic
  - Two-level programmable logic
  - Multi-level programmable logic
Sum-of-products canonical forms

- Also known as disjunctive normal form
- Also known as minterm expansion

F = \sum m(1, 3, 5, 6, 7)
= \sum m(1) + \sum m(3) + \sum m(5) + \sum m(6) + \sum m(7)
= m1 + m3 + m5 + m6 + m7
= A'B'C + A'BC + AB'C + ABC' + ABC

Product-of-sums canonical form

- Also known as conjunctive normal form
- Also known as maxterm expansion

F = \Pi M(0, 2, 4)
= M0 • M2 • M4
= (A + B + C) (A + B’ + C) (A’ + B + C)

Product-of-sums canonical form (cont’d)

- Sum term (or maxterm)
  - ORed sum of literals – input combination for which output is false
  - each variable appears exactly once, in true or inverted form (but not both)

F in canonical form:
F(A, B, C) = (A + B + C) (A + B’ + C) (A’ + B + C)
= (A + C) (B + C)

F in minimal form:
F(A, B, C) = (A + B + C) (A + B’ + C)
= (A + C) (B + C)
S-o-P, P-o-S, and de Morgan's theorem

- **Sum-of-products**
  - $F' = A'B'C' + A'BC' + AB'C'$
  - Apply de Morgan's
    - $(F')' = (A'B'C' + A'BC' + AB'C')'$
    - $F = (A + B + C)(A + B' + C)(A' + B + C)$

- **Product-of-sums**
  - $F' = (A + B + C')(A + B' + C')(A' + B + C')(A' + B' + C')(A' + B' + C')$
  - Apply de Morgan's
    - $(F')' = ((A + B + C')(A + B' + C')(A' + B + C')(A' + B' + C)(A' + B' + C'))'$
    - $F = A'B'C + A'BC + AB'C + ABC' + ABC$

Waveforms for the four alternatives

- Waveforms are essentially identical
  - Except for timing hazards (glitches)
  - Delays almost identical (modeled as a delay per level, not type of gate or number of inputs to gate)

Mapping between canonical forms

- **Minterm to maxterm conversion**
  - Use maxterms whose indices do not appear in minterm expansion
    - e.g., $F(A,B,C) = \Sigma m(1,3,5,6,7) = \Pi M(0,2,4)$
  - **Maxterm to minterm conversion**
    - Use minterms whose indices do not appear in maxterm expansion
      - e.g., $F(A,B,C) = \Pi M(0,2,4) = \Sigma m(1,3,5,6,7)$
  - **Minterm expansion of F to minterm expansion of F'**
    - Use minterms whose indices do not appear
      - e.g., $F(A,B,C) = \Pi M(0,2,4)$ $F'(A,B,C) = \Sigma m(0,2,4)$
  - **Maxterm expansion of F to maxterm expansion of F'**
    - Use maxterms whose indices do not appear
      - e.g., $F(A,B,C) = \Pi M(0,2,4)$ $F'(A,B,C) = \Sigma M(1,3,5,6,7)$
Incompleteley specified functions

- Example: binary coded decimal increment by 1
  - BCD digits encode decimal digits 0 – 9 in bit patterns 0000 – 1001
  
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</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>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>0</td>
<td>1</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>1</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>1</td>
<td>1</td>
<td>1</td>
</tr>
<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>1</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>1</td>
<td>1</td>
<td>1</td>
<td>1</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>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

- off-set of W
- on-set of W
- don’t care (DC) set of W

- these inputs patterns should never be encountered in practice
- “don’t care” about associated output values, can be exploited in minimization

Notation for incompletely specified functions

- Don’t cares and canonical forms
  - So far, only represented on-set
  - Also represent don’t-care-set
  - Need two of the three sets (on-set, off-set, dc-set)

- Canonical representations of the BCD increment by 1 function:
  - \( Z = m_0 + m_2 + m_4 + m_6 + m_8 + d_{10} + d_{11} + d_{12} + d_{13} + d_{14} + d_{15} \)
  - \( Z = \sum (m(0,2,4,6,8) + d(10,11,12,13,14,15)) \)
  - \( Z = M_1 \cdot M_3 \cdot M_5 \cdot M_7 \cdot M_9 \cdot D_{10} \cdot D_{11} \cdot D_{12} \cdot D_{13} \cdot D_{14} \cdot D_{15} \)
  - \( Z = \Pi (M(1,3,5,7,9) \cdot D(10,11,12,13,14,15)) \)

Simplification of two-level combinational logic

- Finding a minimal sum of products or product of sums realization
  - Exploit don’t care information in the process
- Algebraic simplification
  - Not an algorithmic/systematic procedure
  - How do you know when the minimum realization has been found?
- Computer-aided design tools
  - Precise solutions require very long computation times, especially for functions with many inputs (> 10)
  - Heuristic methods employed – “educated guesses” to reduce amount of computation and yield good if not best solutions
- Hand methods still relevant
  - To understand automatic tools and their strengths and weaknesses
  - Ability to check results (on small examples)

The uniting theorem

- Key tool to simplification: \( A \cdot (B' + B) = A \)
- Essence of simplification of two-level logic
  - Find two element subsets of the ON-set where only one variable changes its value – this single varying variable can be eliminated and a single product term used to represent both elements

\[ F = A \cdot B' + A \cdot B = (A' + A)B' + B' \]
Karnaugh maps

- Alternative to truth-tables to help visualize adjacencies
  - Guide to applying the uniting theorem
  - On-set elements with only one variable changing value are adjacent unlike the situation in a linear truth-table

<table>
<thead>
<tr>
<th>B</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Karnaugh maps (cont’d)

- Numbering scheme based on Gray–code
  - e.g., 00, 01, 11, 10
  - Only a single bit changes in code for adjacent map cells

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

More Karnaugh map examples

\[
G(A,B,C) = A
\]

\[
F(A,B,C) = \Sigma m(0,4,5,7) = AC + B'C'
\]

\[
F'(A,B,C) = \Sigma m(1,2,3,6) = BC' + A'C
\]

Karnaugh map: 4-variable example

\[
F(A,B,C,D) = \Sigma m(0,2,3,5,6,7,8,10,11,14,15)
\]

\[
F = C + A'BD + B'D'
\]
Design example: BCD increment by 1

Block diagram and truth table

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

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: $A'B'B'D$, $BC'$, $AC$, $A'C'D$, $AB$, $B'CD$

Minimum cover: $AC + BC' + A'B'B'D$

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

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

Hazards/Glitches

- Hazards/glitches: unwanted switching at the outputs
  - Occur when different paths through circuit have different propagation delays
  - Dangerous if logic causes an action while output is unstable
    - May need to guarantee absence of glitches
- Usual solutions
  - 1) Wait until signals are stable (by using a clock): preferable (easiest to design when there is a clock – synchronous design)
  - 2) Design hazard-free circuits: sometimes necessary (clock not used – asynchronous design)

Types of Hazards

- Static 1-hazard
  - Input change causes output to go from 1 to 0 to 1
- Static 0-hazard
  - Input change causes output to go from 0 to 1 to 0
- Dynamic hazards
  - Input change causes a double change from 0 to 1 to 0 or from 1 to 0 to 1
Static Hazards

- Due to a literal and its complement momentarily taking on the same value
  - Thru different paths with different delays and reconverging
- May cause an output that should have stayed at the same value to momentarily take on the wrong value
- Example: multiplexer

![Static Hazards Diagram]

Dynamic Hazards

- Due to the same versions of a literal taking on opposite values
  - Thru different paths with different delays and reconverging
- May cause an output that was to change value to change 3 times instead of once.

![Dynamic Hazards Diagram]

Mux and Demux

- Uses of multiplexers/demultiplexers in multi-point connections

![Mux and Demux Diagram]

Multiplexers/Selectors

- Multiplexers/Selectors: general concept
  - 2^n data inputs, n control inputs (called "selects"), 1 output
  - Used to connect 2^n points to a single point
  - Control signal pattern forms binary index of input connected to output

![Multiplexers/Selectors Diagram]
Multiplexers/Selectors (cont’d)

- 2:1 mux: \[ Z = A' I_0 + A I_1 \]
- 4:1 mux: \[ Z = A' B' I_0 + A' B I_1 + A B' I_2 + A B I_3 \]
- 8:1 mux: \[ Z = A'B'C'I_0 + A'B'C I_1 + A'BC'I_2 + A'BC I_3 + AB'C'I_4 + AB'C I_5 + ABC'I_6 + ABCI_7 \]

Gate Level Implementation of Muxes

- 2:1 mux
- 4:1 mux

Cascading Multiplexers

- Large multiplexers implemented by cascading smaller ones

Multiplexers as General-purpose Logic

- \( 2^n : 1 \) multiplexer implements any function of \( n \) variables
  - With the variables used as control inputs and
  - Data inputs tied to 0 or 1
  - In essence, a lookup table
- Example:
  \[ F(A,B,C) = m0 + m2 + m6 + m7 = A'B'C' + A'B'C + ABC' + ABC \]
  \[ = A'B'(C') + A'B(C') + AB'(0) + AB(1) \]
Multiplexers as General-purpose Logic (cont’d)

- $2^{n-1}:1$ mux can implement any function of $n$ variables
  - With $n-1$ variables used as control inputs and
  - Data inputs tied to the last variable or its complement
- Example:
  - $F(A, B, C) = m_0 + m_2 + m_6 + m_7$
    $\quad = A'B'C' + A'BC' + ABC' + ABC$
    $\quad = A'B'(C') + A'B(C') + AB'(0) + AB(1)$

Demultiplexers/Decoders

- Decoders/demultiplexers: general concept
  - Single data input, $n$ control inputs, $2^n$ outputs
  - Control inputs (called “selects” ($S$)) represent binary index of output to which the input is connected
  - Data input usually called “enable” ($G$)

Demultiplexers as General-purpose Logic

- $n:2^n$ decoder implements any function of $n$ variables
  - With the variables used as control inputs
  - Enable inputs tied to 1 and
  - Appropriate minterms summed to form the function

Gate Level Implementation of Demultiplexers

- 1:2 Decoders
  - active-high enable
  - active-low enable

- 2:4 Decoders
Demultiplexers as General-purpose Logic (cont'd)

- F1 = \( A'B'C'D + A'B'C'D + ABD \)
- F2 = \( AB'C'D + ABD \)
- F3 = \( (A' + B' + C' + D') \)

Cascading Decoders

- 5:32 decoder
  - 1x 2:4 decoder
  - 4x 3:8 decoders

Logical Function Unit

- Multi-purpose Function Block
  - 3 control inputs to specify operation to perform on operands
  - 2 data inputs for operands
  - 1 output of the same bit-width as operands

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>Function</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>always 1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A + B</td>
<td>logical OR</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>(A + B)'</td>
<td>logical NAND</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A xor B</td>
<td>logical xor</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A xor B</td>
<td>logical xor</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A + B</td>
<td>logical AND</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>(A + B)'</td>
<td>logical NOR</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>always 0</td>
</tr>
</tbody>
</table>
Adder/Subtractor

- Use an adder to do subtraction thanks to 2s complement representation
  - $A - B = A + (-B) = A + B' + 1$
  - Control signal selects $B$ or 2s complement of $B$