CSE140: Components and Design Techniques for Digital Systems

Introduction

Tajana Simunic Rosing
Welcome to CSE 140!

- **Instructor:** Tajana Simunic Rosing  
  - Email: [tajana@ucsd.edu](mailto:tajana@ucsd.edu); please put CSE140 in the subject line  
  - Office Hours: T 2-3pm; Th 10-11am, CSE 2118

- **Instructor’s Assistant:** Sheila Manalo  
  - Email: [shmanalo@ucsd.edu](mailto:shmanalo@ucsd.edu)  
  - Phone: (858) 534-8873

- **TA:** Mingjing Chen  
  - Email: [mjchen@cs.ucsd.edu](mailto:mjchen@cs.ucsd.edu)

- **TA:** Swathi Karunamurthy  
  - Email: [skarunam@ucsd.edu](mailto:skarunam@ucsd.edu)

- **Class Website:**  
  - [http://www.cse.ucsd.edu/classes/wi10/cse140/](http://www.cse.ucsd.edu/classes/wi10/cse140/)

- **Grades:** [http://webct.ucsd.edu](http://webct.ucsd.edu)
Course Description

• Prerequisites:
  – CSE 20 or Math 15A, and CSE 30.
  – CSE 140L must be taken concurrently

• Objective:
  – Introduce digital components and system design concepts

• Grading
  – Homeworks (~9): 15%
    • Lowest grade on the HW will be dropped
    • A fraction of problems assigned are graded
    • HW picked up at beginning of the class, ZERO pts if late
  – Three exams: #1 – 20%; #2 – 30%; #3 – 35%
    • No makeup exams; exceptions only for:
      – documented illness (signed doctor’s statement), death in the family
  – Take home bonus final exam problem(s)
    • Due on the day of the final; worth max 1% bonus of the grade

• Regrade requests:
  – turn in a written request at the end of the class where your work (HW or exam) is returned
Textbook and Recommended Readings

• **Required textbook:**
  – Contemporary Logic Design
    by R. Katz & G. Borriello

• **Recommended textbook:**
  – Digital Design by F. Vahid

• Lecture slides are derived from the slides designed for both books
Why Study Digital Design?

- Look “under the hood” of computers
  - Become a better programmer when aware of hardware resource issues
- Everyday devices becoming digital
  - Enables:
    - Better devices: Better sound recorders, cameras, cars, cell phones, medical devices,...
    - New devices: Video games, PDAs, ...
  - Known as “embedded systems”
    - Thousands of new devices every year
    - Designers needed: Potential career
When Microprocessors Aren’t Good Enough

• With microprocessors so easy to work with, cheap, and available, why design a digital circuit?
  – Microprocessor may be too slow
  – Or too big, power hungry, or costly

Sample digital camera task execution times (in seconds) on a microprocessor versus a digital circuit:

<table>
<thead>
<tr>
<th>Task</th>
<th>Microprocessor</th>
<th>Custom Digital Circuit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>5</td>
<td>0.1</td>
</tr>
<tr>
<td>Compress</td>
<td>8</td>
<td>0.5</td>
</tr>
<tr>
<td>Store</td>
<td>1</td>
<td>0.8</td>
</tr>
</tbody>
</table>
What will we learn in this class?

• The language of logic design
  – Boolean algebra, logic minimization, state, timing, CAD tools

• The concept of state in digital systems
  – analogous to variables and program counters in software systems

• How to specify/simulate/compile/realize our designs (140L)
  – hardware description languages
  – tools to simulate the workings of our designs
  – logic compilers to synthesize the hardware blocks of our designs
  – mapping onto programmable hardware

• Contrast with software design
  – sequential and parallel implementations
  – specify algorithm as well as computing/storage resources it will use
The big picture

- We start with Boolean algebra $Y = A$ and $B$
- We end with a hardware design of a simple CPU

- What’s next? CSE141 – more complex CPU architecture
Outline

- **Number representations**
  - Analog vs. Digital
  - Digital representations:
    - Binary, Hexadecimal, Octal
  - Binary addition, subtraction, multiplication, division

- **Boolean algebra**
  - Properties
  - How Boolean algebra can be used to design logic circuits

- **Switches, MOS transistors, Logic gates**
  - What is a switch
  - How a transistor operates
  - Building logic gates out of transistors
  - Building larger functions from logic gates
CSE140: Components and Design Techniques for Digital Systems

Number representations & Binary arithmetic

Tajana Simunic Rosing
What Does “Digital” Mean?

- **Analog signal**
  - Infinite possible values
  - Ex: voltage on a wire created by microphone

- **Digital signal**
  - Finite possible values
  - Ex: button pressed on a keypad

Possible values: 1.00, 1.01, 2.0000009, … infinite possibilities

Possible values: 0, 1, 2, 3, or 4. That’s it.
How Do We Encode Data into Binary?

- Some inputs are inherently binary
  - Button: not pressed (0), pressed (1)

- Some inputs are inherently digital
  - Just need encoding in binary
  - e.g., multi-button input: encode red=001, blue=010, ...

- Other inputs are analog
  - Need analog-to-digital conversion

Binary digit = BIT
Has 2 values: 0 & 1
A/D conversion & digitization benefits

- Analog signal (e.g., audio) may lose quality
  - Voltage levels not saved/copied/transmitted perfectly
- Digitized version enables near-perfect save/cpy/trn.
  - "Sample" voltage at particular rate, save sample using bit encoding
  - Voltage levels still not kept perfectly
  - But we can distinguish 0s from 1s

Let bit encoding be:
1 V: “01”
2 V: “10”
3 V: “11”

Digitized signal not perfect re-creation, but higher sampling rate and more bits per encoding brings closer.

How fix -- higher, lower, ?
Can fix -- easily distinguish 0s and 1s, restore
Encoding Text: ASCII, Unicode

• ASCII: 7- (or 8-) bit encoding of each letter, number, or symbol
  
• Unicode: Increasingly popular 16-bit bit encoding
  – Encodes characters from various world languages

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>1010010</td>
</tr>
<tr>
<td>S</td>
<td>1010011</td>
</tr>
<tr>
<td>T</td>
<td>1010100</td>
</tr>
<tr>
<td>L</td>
<td>1001100</td>
</tr>
<tr>
<td>N</td>
<td>1001110</td>
</tr>
<tr>
<td>E</td>
<td>1000101</td>
</tr>
<tr>
<td>0</td>
<td>0110000</td>
</tr>
<tr>
<td>.</td>
<td>0101110</td>
</tr>
<tr>
<td>&lt;tab&gt;</td>
<td>0001001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>1110010</td>
</tr>
<tr>
<td>s</td>
<td>1110011</td>
</tr>
<tr>
<td>t</td>
<td>1110100</td>
</tr>
<tr>
<td>l</td>
<td>1101100</td>
</tr>
<tr>
<td>n</td>
<td>1101110</td>
</tr>
<tr>
<td>e</td>
<td>1100101</td>
</tr>
<tr>
<td>g</td>
<td>0111001</td>
</tr>
<tr>
<td>!</td>
<td>0100001</td>
</tr>
<tr>
<td>&lt;space&gt;</td>
<td>0100000</td>
</tr>
</tbody>
</table>

What does this ASCII bit sequence represent?
1010010 1000101 1010011 1010100
Encoding Numbers

• Each position represents a quantity; symbol in position means how many of that quantity
  – Base ten (*decimal*)
    • Ten symbols: 0, 1, 2, ..., 8, and 9
    • More than 9 -- next position
      – So each position power of 10
    • Nothing special about base 10 -- used because we have 10 fingers
  – Base two (*binary*)
    • Two symbols: 0 and 1
    • More than 1 -- next position
      – So each position power of 2
Bases Sixteen & Eight

• Base sixteen
  – nice because each position represents four base two positions
  – Used as compact means to write binary numbers
  – Basic digits: 0-9, A-F
  – Known as hexadecimal, or just hex

• Base eight
  – Used in some digital designs
  – Each position represents three base two positions
  – Basic digits: 0-7

Write 11110000 in hex

Write 11110000 in octal
Sign and magnitude

- One bit dedicate to sign (positive or negative)
  - sign: 0 = positive (or zero), 1 = negative

- Rest represent the absolute value or magnitude
  - three low order bits: 0 (000) thru 7 (111)

- Range for n bits
  - +/- 2^n - 1 - 1 (two representations for 0)

- Cumbersome addition/subtraction
  - must compare magnitudes to determine sign of result
2s complement

• If N is a positive number, then the negative of N (its 2s complement or $N^*$) is $N^* = 2n - N$
  – Shortcut: bit-wise complement plus 1
  – 7 -> -7: 0111 -> 1000 + 1 = 1001 (-7)
  – -7 -> 7: 1001 -> 0110 + 1 = 0111 (7)
Detecting Overflow: Method 1

- Assuming 4-bit two’s complement numbers, can detect overflow by detecting when the two numbers’ sign bits are the same but are different from the result’s sign bit
  - If the two numbers’ sign bits are different, overflow is impossible
    - Adding a positive and negative can’t exceed largest magnitude positive or negative

- Simple circuit
  - overflow = a3’b3’s3 + a3b3s3’
  - Include “overflow” output bit on adder/subtractor

- If the numbers’ sign bits have the same value, which differs from the result’s sign bit, overflow has occurred.
Detecting Overflow: Method 2

- Even simpler method: Detect difference between carry-in to sign bit and carry-out from sign bit
- Yields simpler circuit: overflow = c3 xor c4

If the carry into the sign bit column differs from the carry out of that column, overflow has occurred.
Multiplication

- Generalized representation of multiplication by hand

\[
\begin{array}{cccc}
  a_3 & a_2 & a_1 & a_0 \\
  \times & b_3 & b_2 & b_1 & b_0 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
  b_0a_3 & b_0a_2 & b_0a_1 & b_0a_0 & \text{(pp1)} \\
  b_1a_3 & b_1a_2 & b_1a_1 & b_1a_0 & 0 & \text{(pp2)} \\
  b_2a_3 & b_2a_2 & b_2a_1 & b_2a_0 & 0 & 0 & \text{(pp3)} \\
  + & b_3a_3 & b_3a_2 & b_3a_1 & b_3a_0 & 0 & 0 & 0 & \text{(pp4)} \\
\end{array}
\]

\[
\begin{array}{cccccccc}
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 \\
\end{array}
\]
Division

• Repeated subtraction
  – Set quotient to 0
  – Repeat while dividend >= divisor
    • Subtract divisor from dividend
    • Add 1 to quotient
  – When dividend < divisor:
    • Reminder = dividend
    • Quotient is correct

Example:
• Dividend: 101; Divisor: 10

<table>
<thead>
<tr>
<th>Dividend</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>
Summary of number representation

- Conversion between basis
  - Decimal
  - Binary
  - Octal
  - Hex
- Addition & subtraction in binary
  - Overflow detection
- Multiplication
  - Partial products
    - For demo see: http://courses.cs.vt.edu/~cs1104/BuildingBlocks/multiply.010.html
- Division
  - Repeated subtraction
    - For demo see: http://courses.cs.vt.edu/~cs1104/BuildingBlocks/Binary.Divide.html
CSE140: Components and Design Techniques for Digital Systems

Boolean algebra

Tajana Simunic Rosing
Boolean algebra

- $B = \{0, 1\}$
- Variables represent 0 or 1 only
- Operators return 0 or 1 only
- Basic operators
  - $\cdot$ is logical AND: $a \text{ AND } b$ returns 1 only when both $a=1$ and $b=1$
  - $+$ is logical OR: $a \text{ OR } b$ returns 1 if either (or both) $a=1$ or $b=1$
  - $'$ is logical NOT: $\text{NOT } a$ returns the opposite of $a$ (1 if $a=0$, 0 if $a=1$)
- All algebraic axioms hold
Axioms and theorems of Boolean algebra

• identity
  1. \( X + 0 = X \)
  1D. \( X \cdot 1 = X \)

• null
  2. \( X + 1 = 1 \)
  2D. \( X \cdot 0 = 0 \)

• idempotency:
  3. \( X + X = X \)
  3D. \( X \cdot X = X \)

• involution:
  4. \( (X')' = X \)

• complementarity:
  5. \( X + X' = 1 \)
  5D. \( X \cdot X' = 0 \)

• commutativity:
  6. \( X + Y = Y + X \)
  6D. \( X \cdot Y = Y \cdot X \)

• associativity:
  7. \( (X + Y) + Z = X + (Y + Z) \)
  7D. \( (X \cdot Y) \cdot Z = X \cdot (Y \cdot Z) \)

• distributivity:
  8. \( X \cdot (Y + Z) = (X \cdot Y) + (X \cdot Z) \)
  8D. \( X + (Y \cdot Z) = (X + Y) \cdot (X + Z) \)
Axioms and theorems of Boolean algebra (cont’d)

• uniting:
  9.  $X \cdot Y + X \cdot Y' = X$  
  9D.  $(X + Y) \cdot (X + Y') = X$

• absorption:
  10.  $X + X \cdot Y = X$  
  10D.  $X \cdot (X + Y) = X$
  11.  $(X + Y') \cdot Y = X \cdot Y$  
  11D.  $(X \cdot Y') + Y = X + Y$

• factoring:
  12.  $(X + Y) \cdot (X' + Z) =$  
      $X \cdot Z + X' \cdot Y$  
  12D.  $X \cdot Y + X' \cdot Z =$  
      $(X + Z) \cdot (X' + Y)$

• consensus:
  13.  $(X \cdot Y) + (Y \cdot Z) + (X' \cdot Z) =$  
      $X \cdot Y + X' \cdot Z$  
  13D.  $(X + Y) \cdot (Y + Z) \cdot (X' + Z) =$  
      $(X + Y) \cdot (X' + Z)$

• de Morgan’s:
  14.  $(X + Y + ...)’ = X’ \cdot Y’ \cdot ...$  
  14D.  $(X \cdot Y \cdot ...)’ = X’ + Y’ + ...$

– generalized de Morgan’s:  
  $f(X_1,X_2,...,X_n,0,1,+,\cdot) =$  
  $f(X_1’,X_2’,...,X_n’,1,0,\cdot,+)$
Boolean Duality

• Derived by replacing • by +, + by •, 0 by 1, and 1 by 0 & leaving variables unchanged

\[ X + Y + \ldots \Leftrightarrow X \cdot Y \cdot \ldots \]

• Generalized duality:

\[ f(X_1, X_2, \ldots, X_n, 0, 1, +, \cdot) \Leftrightarrow f(X_1, X_2, \ldots, X_n, 1, 0, \cdot, +) \]

• Any theorem that can be proven is also proven for its dual! Note: this is NOT deMorgan’s Law
Proving theorems

• Using the axioms of Boolean algebra (or a truth table):
  – e.g., prove the theorem: \( X \cdot Y + X \cdot Y' = X \)
    
    - **distributivity (8)** \( X \cdot Y + X \cdot Y' = X \cdot (Y + Y') \)
    - **complementarity (5)** \( X \cdot (Y + Y') = X \cdot (1) \)
    - **identity (1D)** \( X \cdot (1) = X \checkmark \)

  – e.g., prove the theorem: \( X + X \cdot Y = X \)
    
    - **identity (1D)** \( X + X \cdot Y = X \cdot 1 + X \cdot Y \)
    - **distributivity (8)** \( X \cdot 1 + X \cdot Y = X \cdot (1 + Y) \)
    - **identity (2)** \( X \cdot (1 + Y) = X \cdot (1) \)
    - **identity (1D)** \( X \cdot (1) = X \checkmark \)
Proving theorems example

- Prove the following using the laws of Boolean algebra:
  \[(X \cdot Y) + (Y \cdot Z) + (X' \cdot Z) = X \cdot Y + X' \cdot Z\]

\[
(X \cdot Y) + (Y \cdot Z) + (X' \cdot Z)
\]

identity \[
(X \cdot Y) + (1) \cdot (Y \cdot Z) + (X' \cdot Z)
\]

complementarity \[
(X \cdot Y) + (X' + X) \cdot (Y \cdot Z) + (X' \cdot Z)
\]

distributivity \[
(X \cdot Y) + (X' \cdot Y \cdot Z) + (X \cdot Y \cdot Z) + (X' \cdot Z)
\]

commutativity \[
(X \cdot Y) + (X \cdot Y \cdot Z) + (X' \cdot Y \cdot Z) + (X' \cdot Z)
\]

factoring \[
(X \cdot Y) \cdot (1 + Z) + (X' \cdot Z) \cdot (1 + Y)
\]

null \[
(X \cdot Y) \cdot (1) + (X' \cdot Z) \cdot (1)
\]

identity \[
(X \cdot Y) + (X' \cdot Z) \checkmark
\]
Completeness of NAND

• Any logic function can be implemented using just NAND gates. Likewise for NOR. Why?
  – Boolean algebra: need AND, OR and NOT
Combinational Circuit Introduction

- We’ll start with a simple form of circuit:
  - **Combinational circuit**
    - A digital circuit whose outputs depend solely on the *present combination of the circuit inputs’ values*
    - *Built out of simple components: switches and gates*
Design 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>Cout</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Convert equation to logic gates

• More than one way to map expressions to gates
e.g., \( Z = A' \cdot B' \cdot (C + D) = (A' \cdot (B' \cdot (C + D))) \)
  – Use only two input gates first; then with three input gates
CSE140: Components and Design Techniques for Digital Systems

Representation of logic functions

Tajana Simunic Rosing
Logic Design Process: Three 1s Detector

• Problem:
  – Detect 3 consecutive 1s in 8-bit input: abcdefgh
    • 00011101 →
    • 01111000 →
    • 10101011 →

Step 1: Capture the function
  Truth table or equation?
  Truth table too big: $2^8=256$ rows
  Equation:
  $$y = abc + bcd + cde + def + efg + fgh$$

Step 2: Simplify if desired

Step 3: Implement with gates
Canonical Form -- Sum of Minterms

• Truth tables are too big for numerous inputs
• Use standard form of equation instead
  – Known as *canonical form*
  – Regular algebra: group terms of polynomial by power
    • $ax^2 + bx + c$ ($3x^2 + 4x + 2x^2 + 3 + 1 \rightarrow 5x^2 + 4x + 4$)
  – Boolean algebra: create a sum of minterms
    • *Minterm*: product term with every literal (e.g. $a$ or $a'$) appearing
      exactly once

Determine if $F(a,b)=ab+a'$ is same function as $F(a,b)=a'b'+a'b+ab$, by converting the first equation to canonical form
Sum-of-products canonical forms

- Also known as disjunctive normal form
- Minterm expansion:

\[
F = A'B'C' + A'BC' + AB'C' + ABC' + ABC
\]

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

\[
F' = A'B'C' + A'BC' + AB'C'
\]
Sum-of-products canonical form (cont’d)

• Product minterm
  – ANDed product of literals – input combination for which output is 1
  – each variable appears exactly once, true or inverted (but not both)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A'B'C' m0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A'B'C m1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A'BC' m2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A'BC m3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>AB'C' m4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>AB'C m5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ABC' m6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>ABC m7</td>
</tr>
</tbody>
</table>

short-hand notation for minterms of 3 variables

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

canonical form ≠ minimal form
F(A, B, C) = A'B'C + A'BC + AB'C + ABC + ABC'
= (A'B' + A'B + AB' + AB)C + ABC'
= ((A' + A)(B' + B))C + ABC'
= C + ABC'
= ABC' + C
= AB + C
**Product-of-sums canonical form**

- Also known as conjunctive normal form
- Also known as maxterm expansion
- Implements “zeros” of a function

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

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

\[
F' = (A + B + C') (A + B' + C') (A' + B + C') (A' + B' + C) (A' + B' + C')
\]
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, true or inverted (but not both)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>maxterms</th>
<th>F in canonical form:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$A+B+C$</td>
<td>$F(A, B, C) = \Pi M(0,2,4)$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$A+B+C'$</td>
<td>$= M0 \cdot M2 \cdot M4$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$A+B'+C$</td>
<td>$= (A + B + C) (A + B' + C) (A' + B + C)$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$A+B'+C'$</td>
<td>$= (A + B + C) (A + B' + C)$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$A'+B+C$</td>
<td>$= (A + B + C)$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$A'+B+C'$</td>
<td>$= (A + B + C) (A + B' + C)$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$A'+B'+C$</td>
<td>$= (A + B + C) (A' + B + C)$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$A'+B'+C'$</td>
<td>$= (A + C) (B + C)$</td>
</tr>
</tbody>
</table>

short-hand notation for maxterms of 3 variables
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
  – $(F')' = ( (A + B + C')(A + B' + C')(A' + B + C')(A' + B' + C)(A' + B' + C') )'$
  – $F = A'B'C + A'BC + AB'C + ABC' + ABC$
Mapping between canonical forms

- **Minterm to maxterm conversion**
  - use maxterms whose indices do not appear in minterm expansion
  - e.g., \( F(A,B,C) = \sum m(1,3,5,6,7) = \Pi M(0,2,4) \)

- **Maxterm to minterm conversion**
  - use minterms whose indices do not appear in maxterm expansion
  - e.g., \( F(A,B,C) = \Pi M(0,2,4) = \sum m(1,3,5,6,7) \)

- **Minterm expansion of \( F \) to minterm expansion of \( F' \)**
  - use minterms whose indices do not appear
  - e.g., \( F(A,B,C) = \sum m(1,3,5,6,7) \quad F'(A,B,C) = \sum m(0,2,4) \)

- **Maxterm expansion of \( F \) to maxterm expansion of \( F' \)**
  - use maxterms whose indices do not appear
  - e.g., \( F(A,B,C) = \Pi M(0,2,4) \quad F'(A,B,C) = \Pi M(1,3,5,6,7) \)
Incompletely specified functions

- **Example:** binary coded decimal increment by 1
  - BCD digits encode the decimal digits 0 – 9
  - 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)

- **Table:**

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

- **Diagrams:**
  - 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
Alternative two-level implementations of $F = AB + C$
Summary

• What we covered thus far:
  – Number representations
  – Switches, Logic gates
  – Boolean algebra
  – Combinatorial logic representations

• What is next:
  – Combinatorial logic:
    • Minimization
    • Implementations