

CEDT – Digital Logic 2023

Day 1

Intro/Logic

Why study logic design?

- Obvious reasons

- this course is part of the CS/CompE requirements
- it is the implementation basis for all modern computing devices
 - building large things from small components
 - provide a model of how a computer works

- More important reasons

- the inherent parallelism in hardware is often our first exposure to parallel computation
- it offers an interesting counterpoint to software design and is therefore
useful in furthering our understanding of computation, in general

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
 - 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

Applications of logic design

- Conventional computer design
 - CPUs, busses, peripherals
- Networking and communications
 - phones, modems, routers
- Embedded products
 - in cars, toys, appliances, entertainment devices
- Scientific equipment
 - testing, sensing, reporting
- The world of computing is much much bigger than just PCs!

A quick history lesson

- 1850: George Boole invents Boolean algebra
 - maps logical propositions to symbols
 - permits manipulation of logic statements using mathematics
- 1938: Claude Shannon links Boolean algebra to switches
 - his Masters' thesis
- 1945: John von Neumann develops the first stored program computer
 - its switching elements are vacuum tubes (a big advance from relays)
- 1946: ENIAC . . . The world's first completely electronic computer
 - 18,000 vacuum tubes
 - several hundred multiplications per minute
- 1947: Shockley, Brittain, and Bardeen invent the transistor
 - replaces vacuum tubes
 - enable integration of multiple devices into one package
 - gateway to modern electronics

What is logic design?

■ What is design?

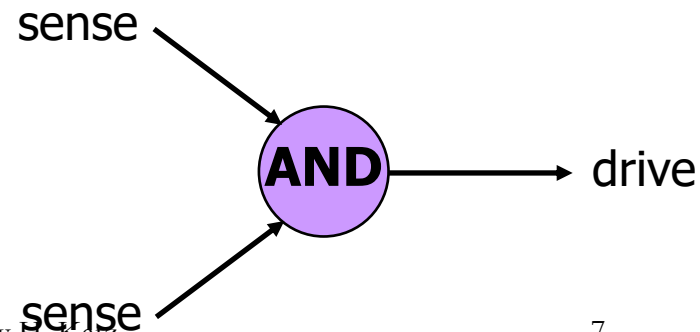
- given a specification of a problem, come up with a way of solving it choosing appropriately from a collection of available components
- while meeting some criteria for size, cost, power, beauty, elegance, etc.

■ What is logic design?

- determining the collection of digital logic components to perform a specified control and/or data manipulation and/or communication function and the interconnections between them
- which logic components to choose? – there are many implementation technologies (e.g., off-the-shelf fixed-function components, programmable devices, transistors on a chip, etc.)
- the design may need to be optimized and/or transformed to meet design constraints

What is digital hardware?

- Collection of devices that sense and/or control wires that carry a digital value (i.e., a physical quantity that can be interpreted as a “0” or “1”)
 - example: digital logic where voltage $< 0.8\text{v}$ is a “0” and $> 2.0\text{v}$ is a “1”
 - example: pair of transmission wires where a “0” or “1” is distinguished by which wire has a higher voltage (differential)
 - example: orientation of magnetization signifies a “0” or a “1”
- Primitive digital hardware devices
 - logic computation devices (sense and drive)
 - are two wires both “1” - make another be “1” (AND)
 - is at least one of two wires “1” - make another be “1” (OR)
 - is a wire “1” - then make another be “0” (NOT)
 - memory devices (store)
 - store a value
 - recall a previously stored value



What is happening now in digital design?

- Important trends in how industry does hardware design
 - larger and larger designs
 - shorter and shorter time to market
 - cheaper and cheaper products
- Scale
 - pervasive use of computer-aided design tools over hand methods
 - multiple levels of design representation
- Time
 - emphasis on abstract design representations
 - programmable rather than fixed function components
 - automatic synthesis techniques
 - importance of sound design methodologies
- Cost
 - higher levels of integration
 - use of simulation to debug designs
 - simulate and verify before you build

Concepts/skills/abilities

- Understanding the basics of logic design (concepts)
- Understanding sound design methodologies (concepts)
- Modern specification methods (concepts)
- Familiarity with a full set of CAD tools (skills)
- Realize digital designs in an implementation technology (skills)
- Appreciation for the differences and similarities (abilities) in hardware and software design

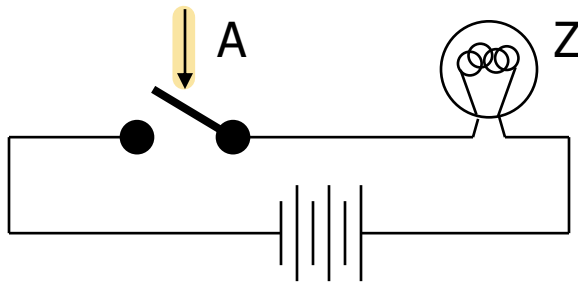
New ability: to accomplish the logic design task with the aid of computer-aided design tools and map a problem description into an implementation with programmable logic devices after validation via simulation and understanding of the advantages/disadvantages as compared to a software implementation

Computation: abstract vs. implementation

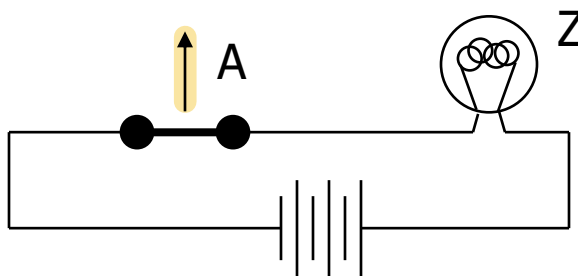
- Up to now, computation has been a mental exercise (paper, programs)
- This class is about physically implementing computation using physical devices that use voltages to represent logical values
- Basic units of computation are:
 - representation: "0", "1" on a wire
set of wires (e.g., for binary ints)
 - assignment: $x = y$
 - data operations: $x + y - 5$
 - control:
 - sequential statements: $A; B; C$
 - conditionals: $\text{if } x == 1 \text{ then } y$
 - loops: $\text{for } (i = 1 ; i == 10, i++)$
 - procedures: $A; \text{proc}(\dots); B;$
- We will study how each of these are implemented in hardware and composed into computational structures

Switches: basic element of physical implementations

- Implementing a simple circuit (arrow shows action if wire changes to “1”):



close switch (if A is “1” or asserted)
and turn on light bulb (Z)



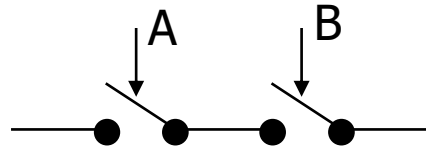
open switch (if A is “0” or unasserted)
and turn off light bulb (Z)

$$Z \equiv A$$

Switches (cont'd)

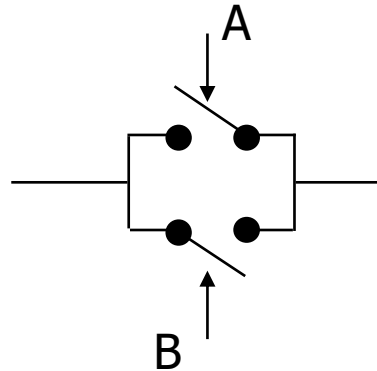
- Compose switches into more complex ones (Boolean functions):

AND



$$Z \equiv A \text{ and } B \equiv AB$$

OR



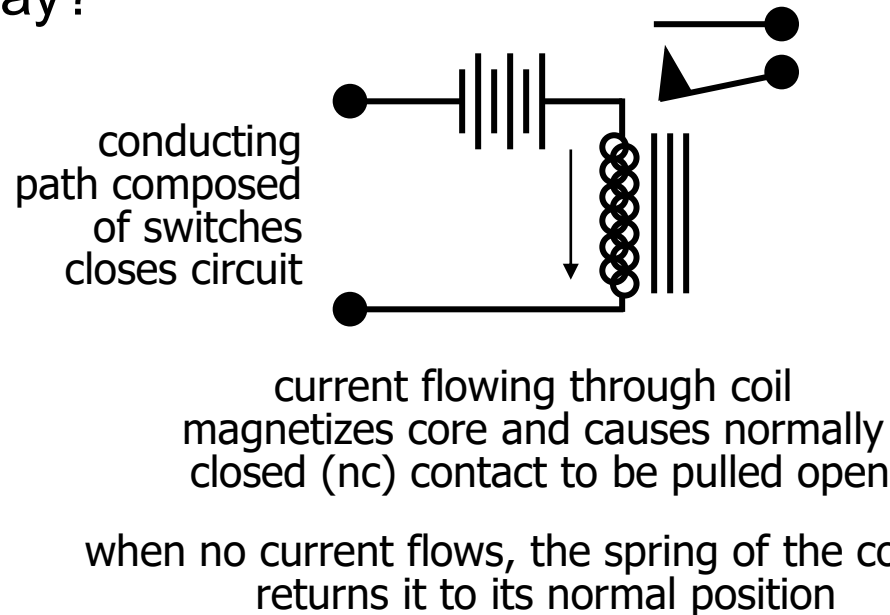
$$Z \equiv A \text{ or } B \equiv A+B$$

Switching networks

- Switch settings
 - determine whether or not a conducting path exists to light the light bulb
- To build larger computations
 - use a light bulb (output of the network) to set other switches (inputs to another network).
- Connect together switching networks
 - to construct larger switching networks, i.e., there is a way to connect outputs of one network to the inputs of the next.

Relay networks

- A simple way to convert between conducting paths and switch settings is to use (electro-mechanical) relays.
- What is a relay?



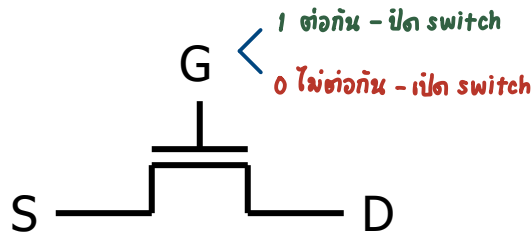
What determines the switching speed of a relay network?

Transistor networks

- Relays aren't used much anymore
 - some traffic light controllers are still electro-mechanical
- Modern digital systems are designed in CMOS technology
 - MOS stands for Metal-Oxide on Semiconductor
 - C is for complementary because there are both normally-open and normally-closed switches
- MOS transistors act as voltage-controlled switches
 - similar, though easier to work with than relays.

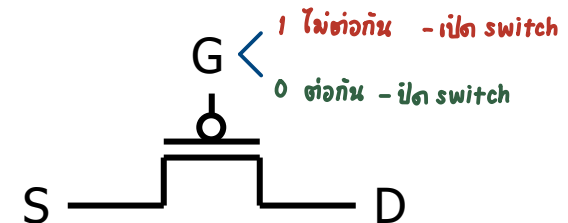
MOS transistors

- MOS transistors have three terminals: drain, gate, and source
 - they act as switches in the following way:
if the voltage on the gate terminal is (some amount) higher/lower than the source terminal then a conducting path will be established between the drain and source terminals



n-channel

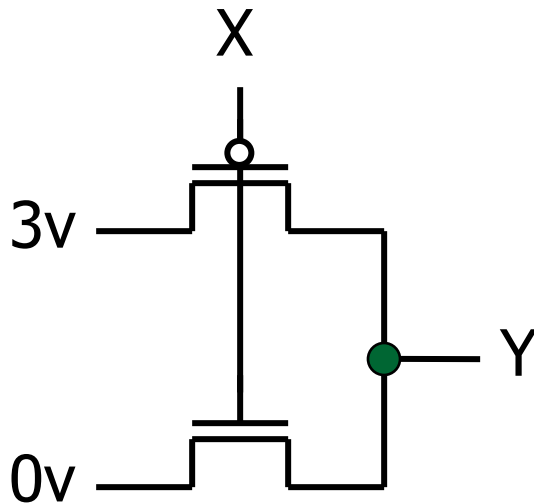
open when voltage at G is low
closes when:
 $\text{voltage}(G) > \text{voltage}(S) + \varepsilon$



p-channel

closed when voltage at G is low
opens when:
 $\text{voltage}(G) < \text{voltage}(S) - \varepsilon$

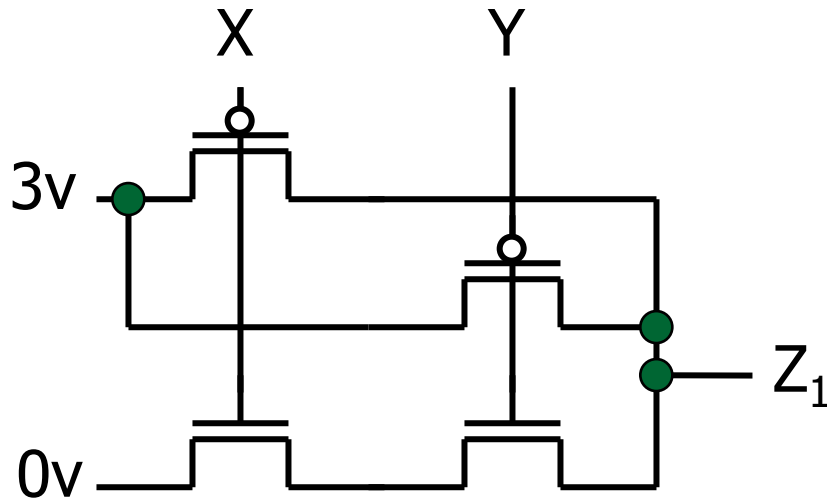
MOS networks



what is the
relationship
between x and y?

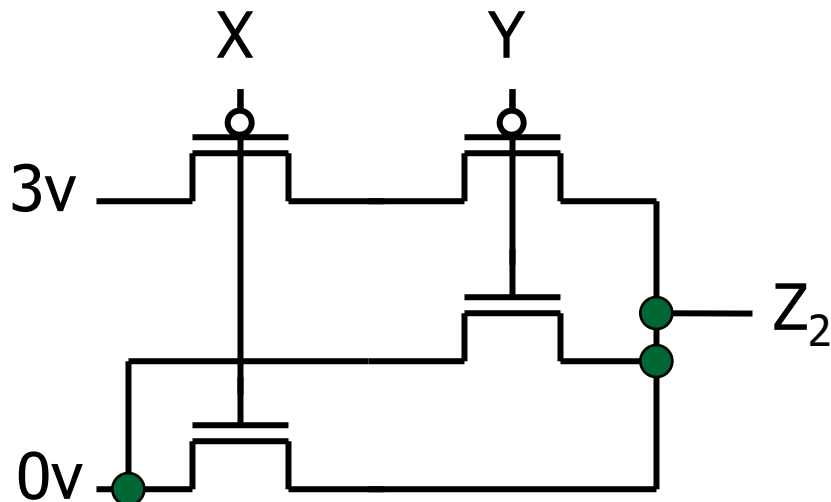
	x		y	
'0'	0 volts		3 volts	'1'
'1'	3 volts		0 volts	'0'
" NOT "				

Two input networks



what is the relationship between x, y and z?

x	y	z1	z2
0 volts	0 volts	3 volts	3 volts
0 volts	3 volts	3 volts	0 volts
3 volts	0 volts	3 volts	0 volts
3 volts	3 volts	0 volts	0 volts
		NAND	NOR



Speed of MOS networks

- What influences the speed of CMOS networks?
 - charging and discharging of voltages on wires and gates of transistors
- Capacitors hold charge
 - capacitance is at gates of transistors and wire material
- Resistors slow movement of electrons
 - resistance mostly due to transistors

Representation of digital designs

- Physical devices (transistors, relays)
- Switches
- Truth tables
- Boolean algebra
- Gates
- Waveforms
- Finite state behavior
- Register-transfer behavior
- Concurrent abstract specifications

Digital vs. analog

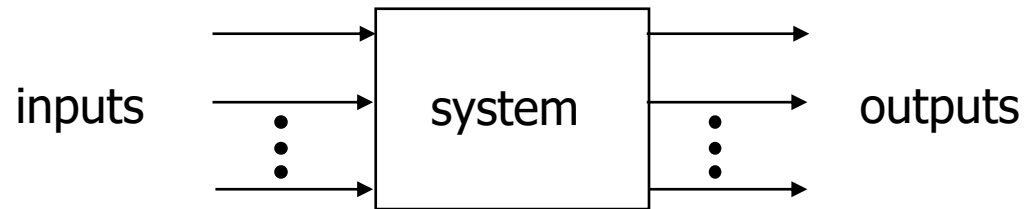
- Convenient to think of digital systems as having only discrete, digital, input/output values
- In reality, real electronic components exhibit continuous, analog, behavior
- Why do we make the digital abstraction anyway?
 - switches operate this way
 - easier to think about a small number of discrete values
- Why does it work?
 - does not propagate small errors in values
 - always resets to 0 or 1

Mapping from physical world to binary world

Technology	State 0	State 1
Relay logic	Circuit Open	Circuit Closed
CMOS logic	0.0-1.0 volts	2.0-3.0 volts
Transistor transistor logic (TTL)	0.0-0.8 volts	2.0-5.0 volts
Fiber Optics	Light off	Light on
Dynamic RAM	Discharged capacitor	Charged capacitor
Nonvolatile memory (erasable)	Trapped electrons	No trapped electrons
Programmable ROM	Fuse blown	Fuse intact
Bubble memory	No magnetic bubble	Bubble present
Magnetic disk	No flux reversal	Flux reversal
Compact disc	No pit	Pit

Combinational vs. sequential digital circuits

- A simple model of a digital system is a unit with inputs and outputs:



- **Combinational** means **"memory-less"**
 - a digital circuit is combinational if its output values only depend on its input values

Combinational logic symbols

- Common combinational logic systems have standard symbols called logic gates

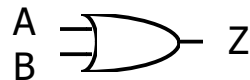
- Buffer, NOT



- AND, NAND



- OR, NOR



easy to implement
with CMOS transistors
(the switches we have
available and use most)

Sequential logic

- Sequential systems
 - exhibit behaviors (output values) that depend not only on the current input values, but also on previous input values
- In reality, all real circuits are sequential
 - because the outputs do not change instantaneously after an input change
 - why not, and why is it then sequential?
- A fundamental abstraction of digital design is to reason (mostly) about steady-state behaviors
 - look at the outputs only after sufficient time has elapsed for the system to make its required changes and settle down

Synchronous sequential digital systems

- Outputs of a combinational circuit depend only on current inputs
 - after sufficient time has elapsed
- Sequential circuits have memory
 - even after waiting for the transient activity to finish
- The steady-state abstraction is so useful that most designers use a form of it when constructing sequential circuits:
 - the memory of a system is represented as its state
 - changes in system state are only allowed to occur at specific times controlled by an external periodic clock
 - the ^{synchronous} clock period is the time that elapses between state changes it must be sufficiently long so that the system reaches a steady-state before the next state change at the end of the period

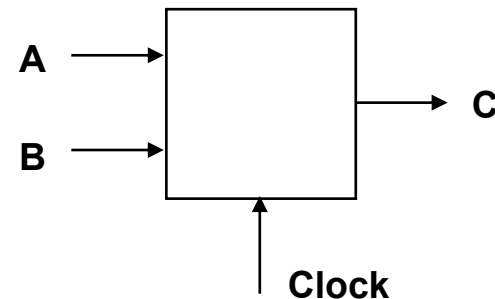
Example of combinational and sequential logic

■ Combinational: *ໄວ້ໄວ້ clock*

- ❑ input A, B
- ❑ wait for clock edge
- ❑ observe C
- ❑ wait for another clock edge
- ❑ observe C again: **will stay the same**

■ Sequential: *ມື້ clock*

- ❑ input A, B
- ❑ wait for clock edge
- ❑ observe C
- ❑ wait for another clock edge
- ❑ observe C again: **may be different**



Abstractions

- Some we've seen already
 - digital interpretation of analog values
 - transistors as switches
 - switches as logic gates
 - use of a clock to realize a synchronous sequential circuit
- Some others we will see
 - truth tables and Boolean algebra to represent combinational logic
 - encoding of signals with more than two logical values into binary form
 - state diagrams to represent sequential logic
 - hardware description languages to represent digital logic
 - waveforms to represent temporal behavior

An example

- Calendar subsystem: number of days in a month (to control watch display)
 - used in controlling the display of a wrist-watch LCD screen
 - inputs: month, leap year flag
 - outputs: number of days

Implementation in software

```
integer number_of_days ( month, leap_year_flag)
{
  switch (month) {
    case 1: return (31);
    case 2: if (leap_year_flag == 1) then return (29)
           else return (28);
    case 3: return (31);
    ...
    case 12: return (31);
    default: return (0);
  }
}
```

Implementation as a combinational digital system

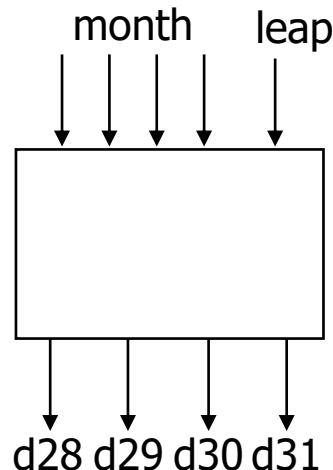
■ Encoding:

- ❑ how many bits for each input/output?
- ❑ binary number for month
- ❑ four wires for 28, 29, 30, and 31

■ Behavior:

- ❑ combinational
- ❑ truth table specification

เป็น transistor ไม่ได



“ Truth Table ”

month	leap	d28	d29	d30	d31
0000	—	—	—	—	—
0001	—	0	0	0	1
0010	0	1	0	0	0
0010	1	0	1	0	0
0011	—	0	0	0	1
0100	—	0	0	1	0
0101	—	0	0	0	1
0110	—	0	0	1	0
0111	—	0	0	0	1
1000	—	0	0	0	1
1001	—	0	0	1	0
1010	—	0	0	0	1
1011	—	0	0	1	0
1100	—	0	0	0	1
1101	—	—	—	—	—
111—	—	—	—	—	—

Combinational example (cont'd)

Boolean Algebra

■ Truth-table to logic to switches to gates

□ d28 = 1 when month=0010 and leap=0

□ $d28 = m8' \cdot m4' \cdot m2 \cdot m1' \cdot leap'$

□ d31 = 1 when month=0001 or month=0011 or ... month=1100

□ $d31 = (m8' \cdot m4' \cdot m2' \cdot m1) + (m8' \cdot m4' \cdot m2 \cdot m1) + \dots$
($m8 \cdot m4 \cdot m2' \cdot m1'$)

□ d31 = can we simplify more?

symbol
for and

symbol
for or

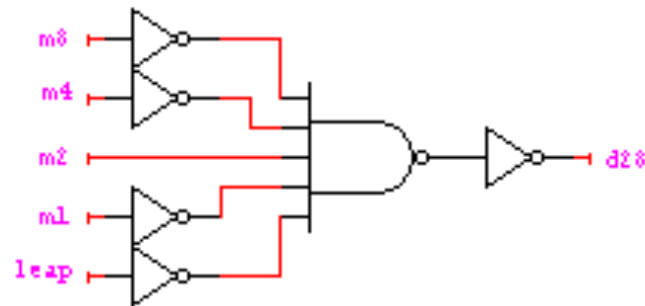
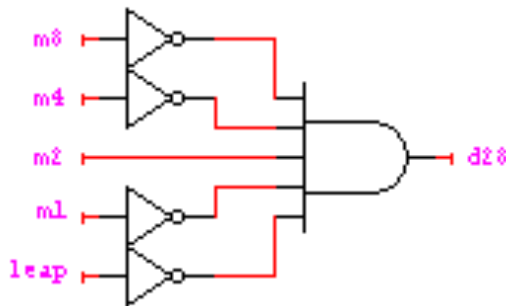
symbol
for not $A' \equiv \text{not } A$
 $\bar{A} \equiv \text{not } A$

month	leap	d28	d29	d30	d31
0001	—	0	0	0	1
0010	0	1	0	0	0
0010	1	0	1	0	0
0011	—	0	0	0	1
0100	—	0	0	1	0
...					
1100	—	0	0	0	1
1101	—	—	—	—	—
111—	—	—	—	—	—
0000	—	—	—	—	—

Combinational example (cont'd)

- $d28 = m8' \cdot m4' \cdot m2 \cdot m1' \cdot \text{leap}'$
- $d29 = m8' \cdot m4' \cdot m2 \cdot m1' \cdot \text{leap}$
- $d30 = (m8' \cdot m4 \cdot m2' \cdot m1') + (m8' \cdot m4 \cdot m2 \cdot m1') + (m8 \cdot m4' \cdot m2' \cdot m1) + (m8 \cdot m4' \cdot m2 \cdot m1)$
 $= (m8' \cdot m4 \cdot m1') + (m8 \cdot m4' \cdot m1)$
- $d31 = (m8' \cdot m4' \cdot m2' \cdot m1) + (m8' \cdot m4' \cdot m2 \cdot m1) + (m8' \cdot m4 \cdot m2' \cdot m1) + (m8' \cdot m4 \cdot m2 \cdot m1) + (m8 \cdot m4' \cdot m2' \cdot m1') + (m8 \cdot m4' \cdot m2 \cdot m1') + (m8 \cdot m4 \cdot m2' \cdot m1') + (m8 \cdot m4 \cdot m2 \cdot m1')$

$\left. \begin{array}{l} \\ \\ \\ \\ \\ \\ \\ \end{array} \right\} m'8 m1 + m8 m1'$



Activity I

- How much can we simplify d31?

d31 is true if : month is 7 or less and odd (1, 3, 5, 7) or
month is 8 or more and even (8, 10, 12, and includes 14)

d31 is true if : m8 is 0 and m1 is 1, or m8 is 1 and m1 is 0
$$d31 = m8'm1 + m8m1'$$

- What if we started the months with 0 instead of 1?
(i.e., January is 0000 and December is 1011)

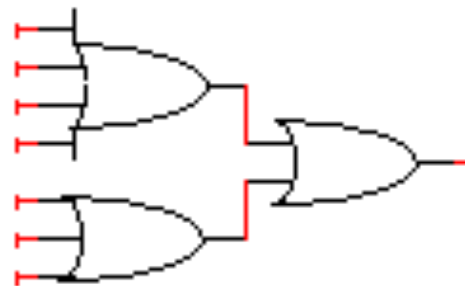
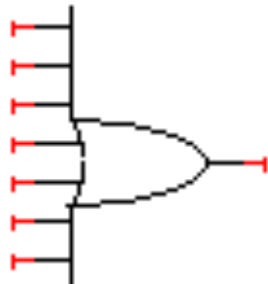
More complex expression (0, 2, 4, 6, 7, 9, 11)

$$d31 = m8'm4'm2'm1' + m8'm4'm2m1' + m8'm4m2'm1' + m8'm4m2m1' \\ + m8'm4m2m1 + m8m4'm2'm1 + m8m4'm2m1$$

$$d31 = m8'm1' + m8'm4m2 + m8m1 \text{ (includes 13 and 15)}$$

Combinational example (cont'd)

- $d28 = m8' \cdot m4' \cdot m2 \cdot m1' \cdot \text{leap}'$
- $d29 = m8' \cdot m4' \cdot m2 \cdot m1' \cdot \text{leap}$
- $d30 = (m8' \cdot m4 \cdot m2' \cdot m1') + (m8' \cdot m4 \cdot m2 \cdot m1') + (m8 \cdot m4' \cdot m2' \cdot m1) + (m8 \cdot m4' \cdot m2 \cdot m1)$
- $d31 = (m8' \cdot m4' \cdot m2' \cdot m1) + (m8' \cdot m4' \cdot m2 \cdot m1) + (m8' \cdot m4 \cdot m2' \cdot m1) + (m8' \cdot m4 \cdot m2 \cdot m1) + (m8 \cdot m4' \cdot m2' \cdot m4') + (m8 \cdot m4' \cdot m2 \cdot m1') + (m8 \cdot m4 \cdot m2' \cdot m1')$



Another example

- Door combination lock:
 - punch in 3 values in sequence and the door opens; if there is an error the lock must be reset; once the door opens the lock must be reset
 - inputs: sequence of input values, reset
 - outputs: door open/close
 - memory: must remember combination
or always have it available as an input

Implementation in software

```
integer combination_lock ( ) {  
    integer v1, v2, v3;  
    integer error = 0;  
    static integer c[3] = 3, 4, 2;  
  
    while (!new_value( ));  
    v1 = read_value( );  
    if (v1 != c[1]) then error = 1;  
  
    while (!new_value( ));  
    v2 = read_value( );  
    if (v2 != c[2]) then error = 1;  
  
    while (!new_value( ));  
    v3 = read_value( );  
    if (v2 != c[3]) then error = 1;  
  
    if (error == 1) then return(0); else return (1);  
}
```

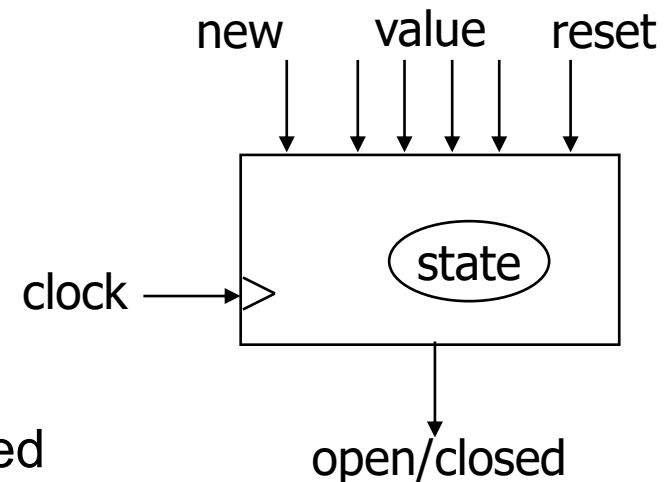
Implementation as a sequential digital system

■ Encoding:

- ❑ how many bits per input value?
- ❑ how many values in sequence?
- ❑ how do we know a new input value is entered?
- ❑ how do we represent the states of the system?

■ Behavior:

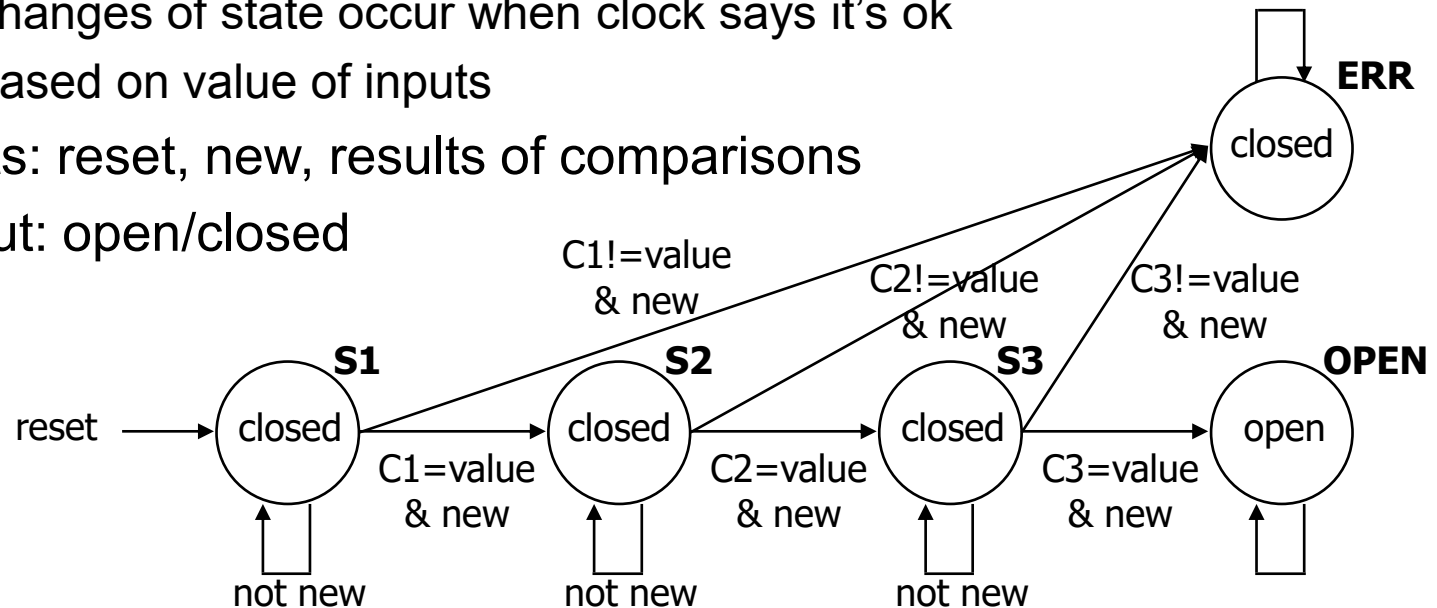
- ❑ clock wire tells us when it's ok to look at inputs (i.e., they have settled after change)
- ❑ sequential: sequence of values must be entered
- ❑ sequential: remember if an error occurred
- ❑ finite-state specification



Sequential example (cont'd): abstract control

■ Finite-state diagram

- states: 5 states
 - represent point in execution of machine
 - each state has outputs
- transitions: 6 from state to state, 5 self transitions, 1 global
 - changes of state occur when clock says it's ok
 - based on value of inputs
- inputs: reset, new, results of comparisons
- output: open/closed



Sequential example (cont'd): data-path vs. control

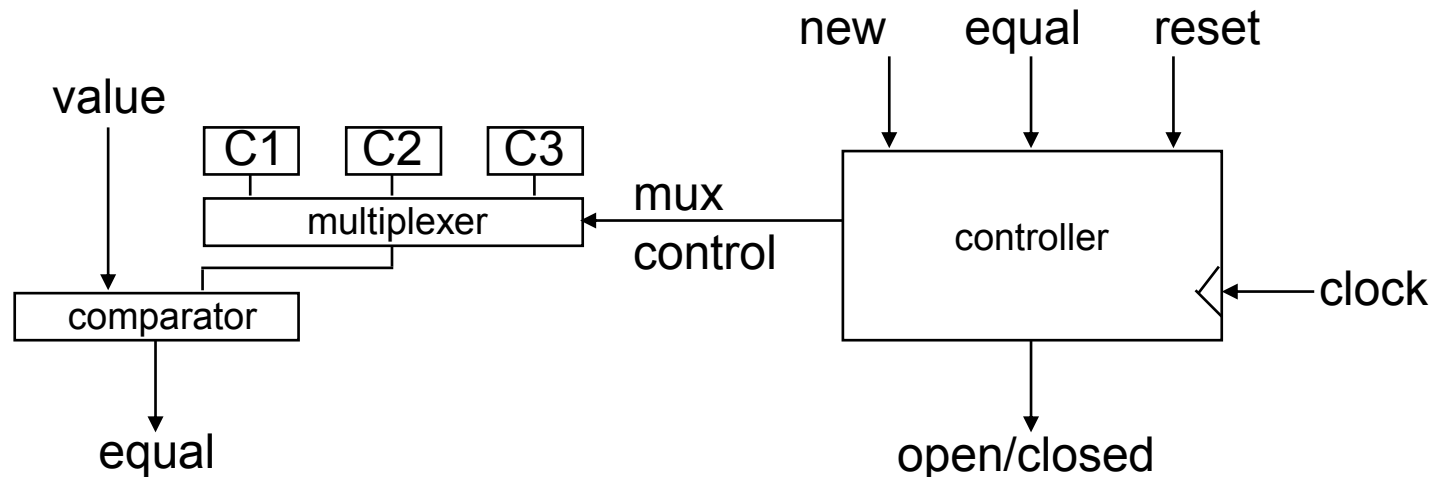
■ Internal structure

□ data-path

- storage for combination
- comparators

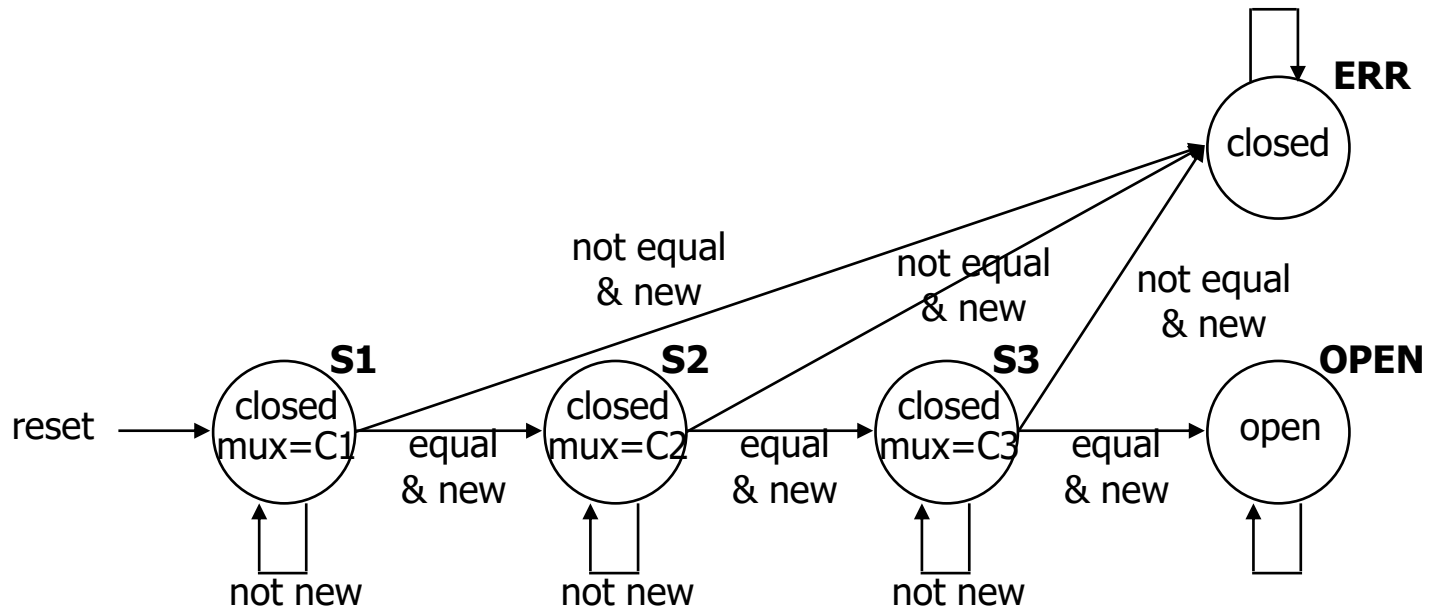
□ control

- finite-state machine controller
- control for data-path
- state changes controlled by clock



Sequential example (cont'd): finite-state machine

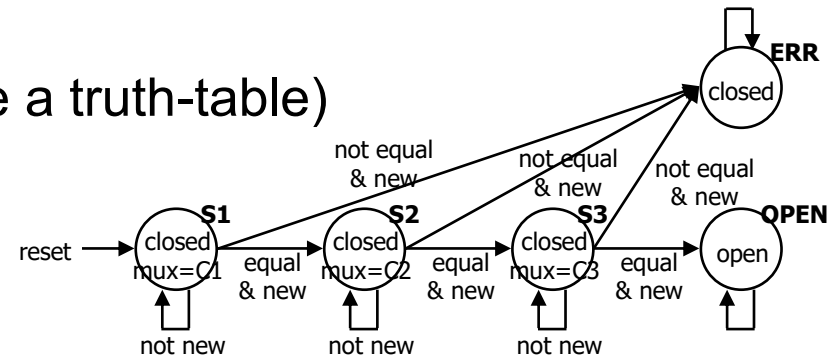
- Finite-state machine
 - refine state diagram to include internal structure



Sequential example (cont'd): finite-state machine

■ Finite-state machine

- generate state table (much like a truth-table)



reset	new	equal	state	next state	mux	open/closed
1	—	—	—	S1	C1	closed
0	0	—	S1	S1	C1	closed
0	1	0	S1	ERR	—	closed
0	1	1	S1	S2	C2	closed
0	0	—	S2	S2	C2	closed
0	1	0	S2	ERR	—	closed
0	1	1	S2	S3	C3	closed
0	0	—	S3	S3	C3	closed
0	1	0	S3	ERR	—	closed
0	1	1	S3	OPEN	—	open
0	—	—	OPEN	OPEN	—	open
0	—	—	ERR	ERR	—	closed

Sequential example (cont'd): encoding

■ Encode state table

- state can be: S1, S2, S3, OPEN, or ERR
 - needs at least 3 bits to encode: 000, 001, 010, 011, 100
 - and as many as 5: 00001, 00010, 00100, 01000, 10000
 - choose 4 bits: 0001, 0010, 0100, 1000, 0000
- output mux can be: C1, C2, or C3
 - needs 2 to 3 bits to encode
 - choose 3 bits: 001, 010, 100
- output open/closed can be: open or closed
 - needs 1 or 2 bits to encode
 - choose 1 bits: 1, 0

Sequential example (cont'd): encoding

■ Encode state table

- state can be: S1, S2, S3, OPEN, or ERR
 - choose 4 bits: 0001, 0010, 0100, 1000, 0000
- output mux can be: C1, C2, or C3
 - choose 3 bits: 001, 010, 100
- output open/closed can be: open or closed
 - choose 1 bits: 1, 0

reset	new	equal	state	next state	mux	open/closed
1	—	—	—	0001	001	0
0	0	—	0001	0001	001	0
0	1	0	0001	0000	—	0
0	1	1	0001	0010	010	0
0	0	—	0010	0010	010	0
0	1	0	0010	0000	—	0
0	1	1	0010	0100	100	0
0	0	—	0100	0100	100	0
0	1	0	0100	0000	—	0
0	1	1	0100	1000	—	1
0	—	—	1000	1000	—	1
0	—	—	0000	0000	—	0

good choice of encoding!

mux is identical to
last 3 bits of state

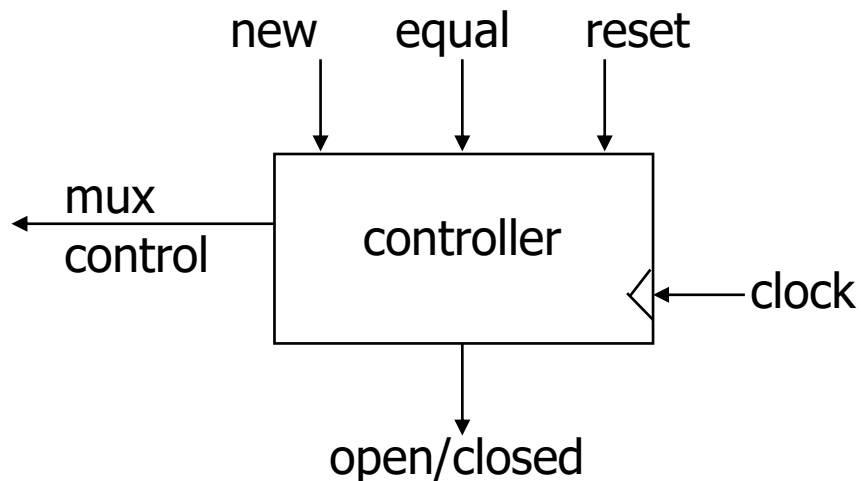
open/closed is
identical to first bit
of state

Activity II (no submission in MCV)

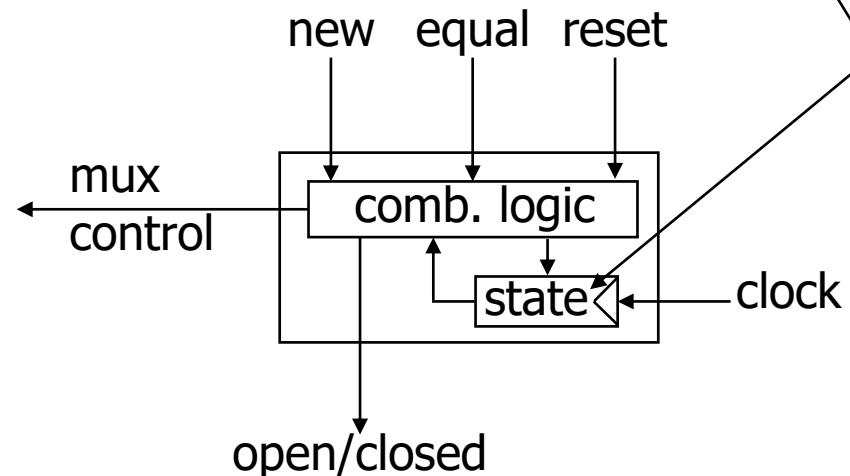
- Have lock always wait for 3 key presses exactly before making a decision

Sequential example (cont'd): controller implementation

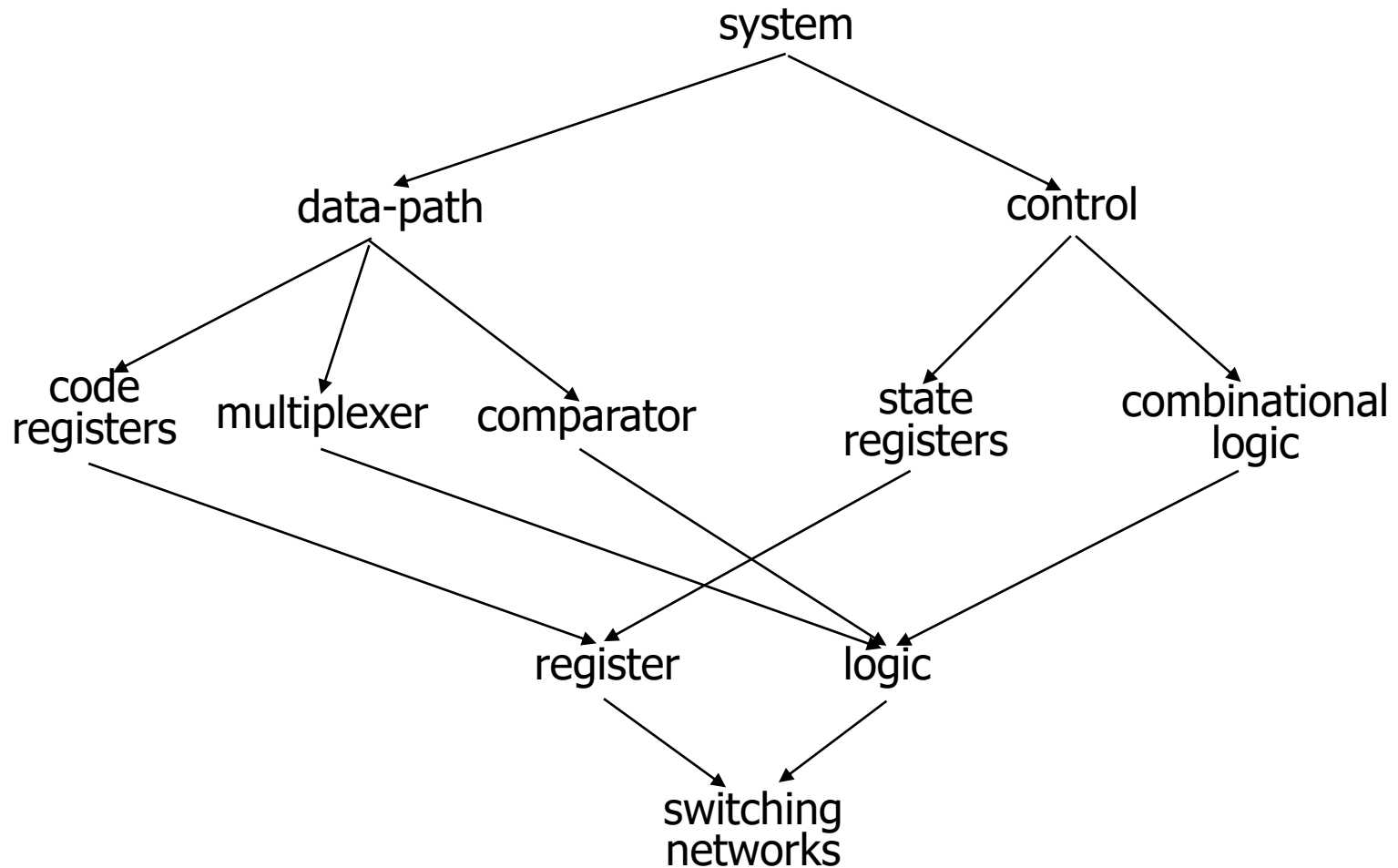
■ Implementation of the controller



special circuit element,
called a register, for
remembering inputs
when told to by clock



Design hierarchy



Summary

- That was what the entire course is about
 - converting solutions to problems into combinational and sequential networks effectively organizing the design hierarchically
 - doing so with a modern set of design tools that lets us handle large designs effectively
 - taking advantage of optimization opportunities

Combinational logic

- Basic logic
 - Boolean algebra, proofs by re-writing, proofs by perfect induction
 - logic functions, truth tables, and switches
 - NOT, AND, OR, NAND, NOR, XOR, . . ., minimal set
- Logic realization
 - two-level logic and canonical forms
 - incompletely specified functions
- Simplification
 - uniting theorem
 - grouping of terms in Boolean functions
- Alternate representations of Boolean functions
 - cubes
 - Karnaugh maps

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	Y	16 possible functions (F0–F15)															
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
		X and Y		X		Y		X \oplus Y		X or Y		X nor Y not (X or Y)		X = Y		not Y	
																not X	
																X nand 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
- thus, because NOT, NOR, and NAND are the cheapest they are the functions we implement the most in practice

Minimal set of functions

- Can we implement all logic functions from NOT, NOR, and NAND?
 - For example, implementing X and Y is the same as implementing $\text{not } (X \text{ nand } Y)$
- In fact, we can do it with only NOR or only NAND
 - NOT is just a NAND or a NOR with both inputs tied together

X	Y	X nor Y
0	0	1
1	1	0

X	Y	X nand Y
0	0	1
1	1	0

- and NAND and NOR are "duals",
that is, its easy to implement one using the other

$$\begin{aligned} X \text{ nand } Y &\equiv \text{not } ((\text{not } X) \text{ nor } (\text{not } Y)) \\ X \text{ nor } Y &\equiv \text{not } ((\text{not } X) \text{ nand } (\text{not } Y)) \end{aligned}$$

- But lets not move too fast . . .
 - lets look at the mathematical foundation of logic

An algebraic structure

- An algebraic structure consists of
 - a set of elements B
 - binary operations $\{ + , \cdot \}$
 - and a unary operation $\{ ' \}$
 - such that the following axioms hold:

1. the set B contains at least two elements: a, 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
 - $B = \{0, 1\}$
 - variables
 - $+$ is logical OR, \cdot 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 •

X	Y	$X \bullet Y$
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	X'	$X' \bullet Y$
0	0	1	0
0	1	1	1
1	0	0	0
1	1	0	0

X	Y	X'	Y'	$X \bullet Y$	$X' \bullet Y'$	$(X \bullet Y) + (X' \bullet Y')$
0	0	1	1	0	1	1
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	1	0	0	1	0	1

= NOR

$$(X \bullet Y) + (X' \bullet Y') \equiv X = Y$$

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 **เอกลักษณ์**

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. $(X')' = X = \overline{\overline{X}}$

- complementarity:

5. $X + X' = 1$

5D. $X \cdot X' = 0$

- commutativity: **สลับที่**

6. $X + Y = Y + X$

6D. $X \cdot Y = Y \cdot X$

- associativity: **เปลี่ยนกลุ่ม**

7. $(X + Y) + Z = X + (Y + Z)$

7D. $(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$

Axioms and theorems of Boolean algebra (cont'd)

- distributivity: *מחזור*

$$8. X \cdot (Y + Z) = (X \cdot Y) + (X \cdot Z) \quad 8D. X + (Y \cdot Z) = (X + Y) \cdot (X + Z)$$

- * ■ uniting: $X \cdot (Y + Y') = X \cdot 1 = X$

$$9. X \cdot Y + X \cdot Y' = X$$

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

- 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: $(X \cdot X) + (X \cdot Z) + (X' \cdot Y) + (Y \cdot Z) = (X \cdot Z) + (X' \cdot Y) + (Y \cdot Z \cdot (X + X'))$

$$12. \underbrace{(X + Y) \cdot (X' + Z)}_{X \cdot Z + X' \cdot Y} = \underbrace{(X \cdot Z) + (X' \cdot Y) + (X \cdot Y \cdot Z)}_{(X \cdot Z)(1 + Y') + (X' \cdot Y)(1 + Z)}_{= X \cdot Z + X' \cdot Y} \quad 12D. X \cdot Y + X' \cdot Z = (X + Z) \cdot (X' + Y)$$

- consensus:

$$13. (X \cdot Y) + (Y \cdot Z) + (X' \cdot Z) = X \cdot Y + X' \cdot Z \quad 13D. (X + Y) \cdot (Y + Z) \cdot (X' + Z) = (X + Y) \cdot (X' + Z)$$

Axioms and theorems of Boolean algebra (cont'd)

* ■ de Morgan's: לוגיקה NOT

$$14. (X + Y + \dots)' = X' \cdot Y' \cdot \dots \quad 14D. (X \cdot Y \cdot \dots)' = X' + Y' + \dots$$

■ generalized de Morgan's:

$$15. f'(X_1, X_2, \dots, X_n, 0, 1, +, \cdot) = f(X_1', X_2', \dots, X_n', 1, 0, \cdot, +)$$

■ establishes relationship between \cdot and $+$

Axioms and theorems of Boolean algebra (cont'd)

■ Duality

- a dual of a Boolean expression is derived by replacing
 - by +, + by •, 0 by 1, and 1 by 0, and leaving variables unchanged
- any theorem that can be proven is thus also proven for its dual!
- a meta-theorem (a theorem about theorems)

■ duality:

$$16. X + Y + \dots \Leftrightarrow X \cdot Y \cdot \dots$$

■ generalized duality:

$$17. f(X_1, X_2, \dots, X_n, 0, 1, +, \cdot) \Leftrightarrow f(X_1, X_2, \dots, X_n, 1, 0, \cdot, +)$$

■ Different than deMorgan's Law

- this is a statement about theorems
- this is not a way to manipulate (re-write) expressions

Proving theorems (rewriting)

สำนวน

- Using the **axioms** of Boolean algebra:

- 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)	$X \cdot (1)$	$=$	$X \checkmark$

Activity III

- 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 $(X \cdot Y) + (X' \cdot Z) \checkmark$

Proving theorems (perfect induction)

- Using perfect induction (complete truth table):
 - e.g., de Morgan's:

$(X + Y)' = X' \cdot Y'$
NOR is equivalent to AND
with inputs complemented

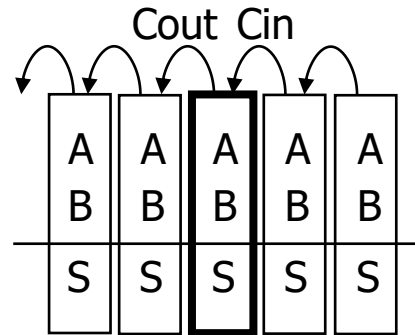
X	Y	X'	Y'	$(X + Y)'$	$X' \cdot Y'$
0	0	1	1	1	1
0	1	1	0	0	0
1	0	0	1	0	0
1	1	0	0	0	0

$(X \cdot Y)' = X' + Y'$
NAND is equivalent to OR
with inputs complemented

X	Y	X'	Y'	$(X \cdot Y)'$	$X' + Y'$
0	0	1	1	1	1
0	1	1	0	1	1
1	0	0	1	1	1
1	1	0	0	0	0

A simple example: 1-bit binary adder

- Inputs: A, B, Carry-in ກົດເຂົ້າ
- Outputs: Sum, Carry-out ກົດອອກ



A	B	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



$$S = A' B' Cin + A' B Cin' + A B' Cin' + A B Cin$$

$$Cout = A' B Cin + A B' Cin + A B Cin' + A B Cin$$

Apply the theorems to simplify expressions

- The theorems of Boolean algebra can simplify Boolean expressions
 - e.g., full adder's carry-out function (same rules apply to any function)

$$\begin{aligned}\text{Cout} &= A' B \text{Cin} + A B' \text{Cin} + A B \text{Cin}' + A B \text{Cin} \\ &= A' B \text{Cin} + A B' \text{Cin} + A B \text{Cin}' + \boxed{A B \text{Cin}} + A B \text{Cin} \\ &= \underline{A' B \text{Cin}} + \underline{A B \text{Cin}} + A B' \text{Cin} + A B \text{Cin}' + A B \text{Cin} \\ &= (A' + A) B \text{Cin} + A B' \text{Cin} + A B \text{Cin}' + A B \text{Cin} \\ &= (1) B \text{Cin} + A B' \text{Cin} + A B \text{Cin}' + A B \text{Cin} \\ &= B \text{Cin} + A B' \text{Cin} + A B \text{Cin}' + \boxed{A B \text{Cin}} + A B \text{Cin} \\ &= B \text{Cin} + \underline{A B' \text{Cin}} + \underline{A B \text{Cin}} + A B \text{Cin}' + A B \text{Cin} \\ &= B \text{Cin} + A (B' + B) \text{Cin} + A B \text{Cin}' + A B \text{Cin} \\ &= B \text{Cin} + A (1) \text{Cin} + A B \text{Cin}' + A B \text{Cin} \\ &= B \text{Cin} + A \text{Cin} + A B (\text{Cin}' + \text{Cin}) \\ &= B \text{Cin} + A \text{Cin} + A B (1) \\ &= B \text{Cin} + A \text{Cin} + A B\end{aligned}$$

adding extra terms
creates new factoring
opportunities

Activity IV

- Fill in the truth-table for a circuit that checks that a 4-bit number is divisible by 2, 3, or 5

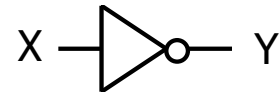
X8	X4	X2	X1	By2	By3	By5
0	0	0	0	1	1	1
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	0	1	1	0	1	0

- Write down Boolean expressions for By2, By3, and By5

Activity IV

From Boolean expressions to logic gates

■ NOT X' \bar{X} $\sim X$



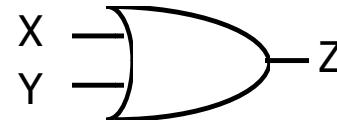
X	Y
0	1
1	0

■ AND $X \cdot Y$ XY $X \wedge Y$



X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

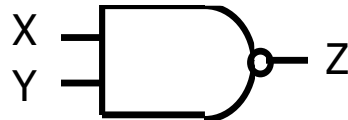
■ OR $X + Y$ $X \vee Y$



X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

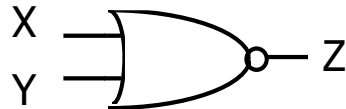
From Boolean expressions to logic gates (cont'd)

■ NAND



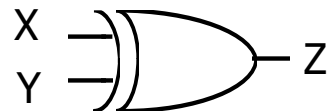
X	Y	Z
0	0	1
0	1	1
1	0	1
1	1	0

■ NOR



X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	0

■ XOR $X \oplus Y$



X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

$X \text{ xor } Y = X Y' + X' Y$
X or Y but not both
("inequality", "difference")

■ XNOR $X = Y$



X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	1

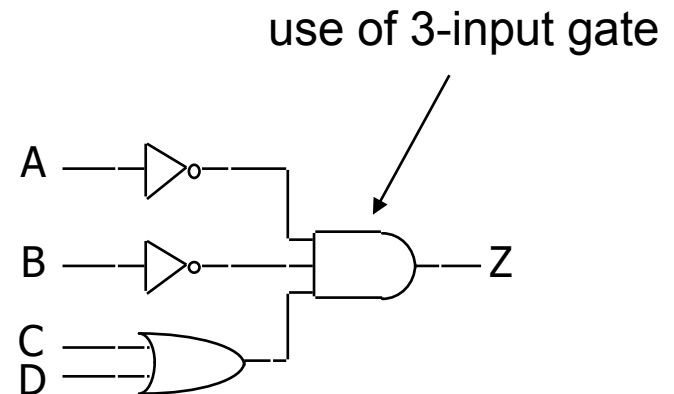
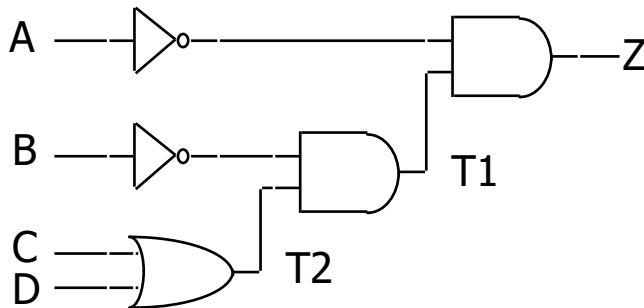
$X \text{ xnor } Y = X Y + X' Y'$
X and Y are the same
("equality", "coincidence")

From Boolean expressions to logic gates (cont'd)

- More than one way to map expressions to gates

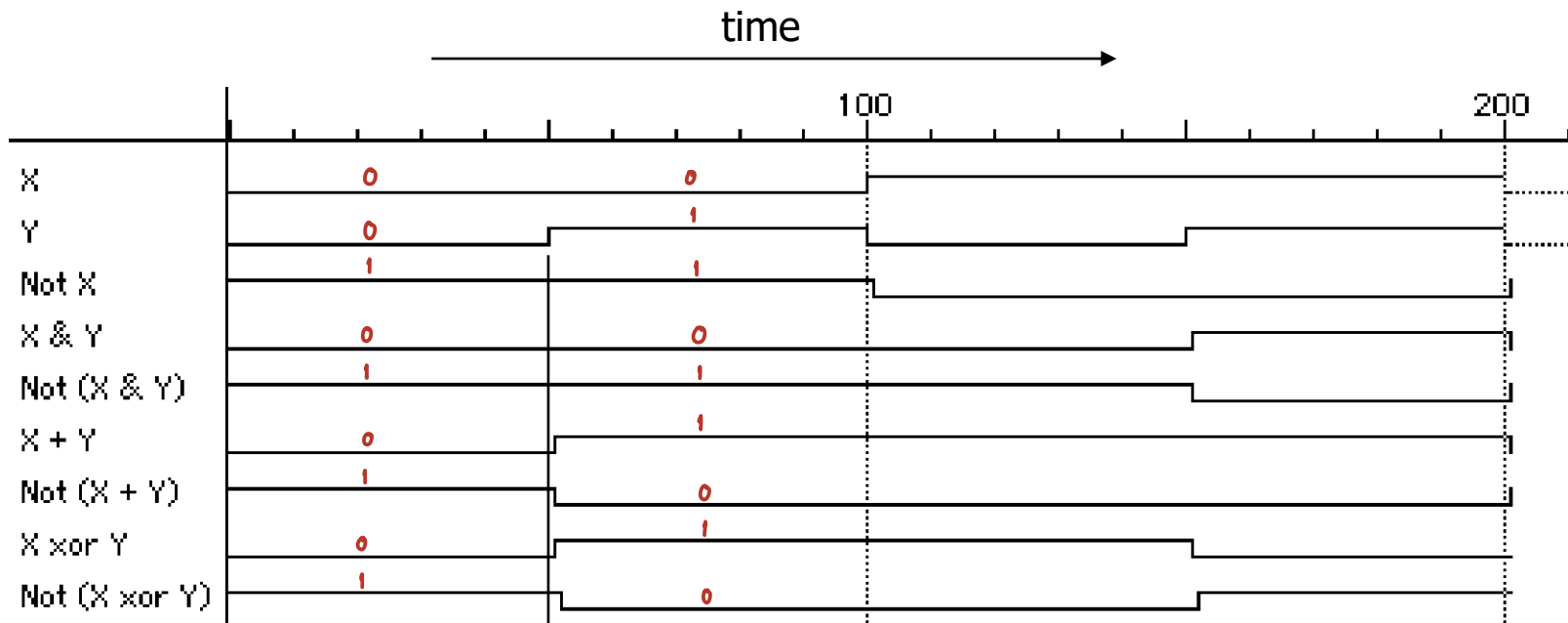
□ e.g., $Z = A' \cdot B' \cdot (C + D) = (A' \cdot (B' \cdot (C + D)))$

$$\frac{\quad T2}{T1}$$



Waveform view of logic functions

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

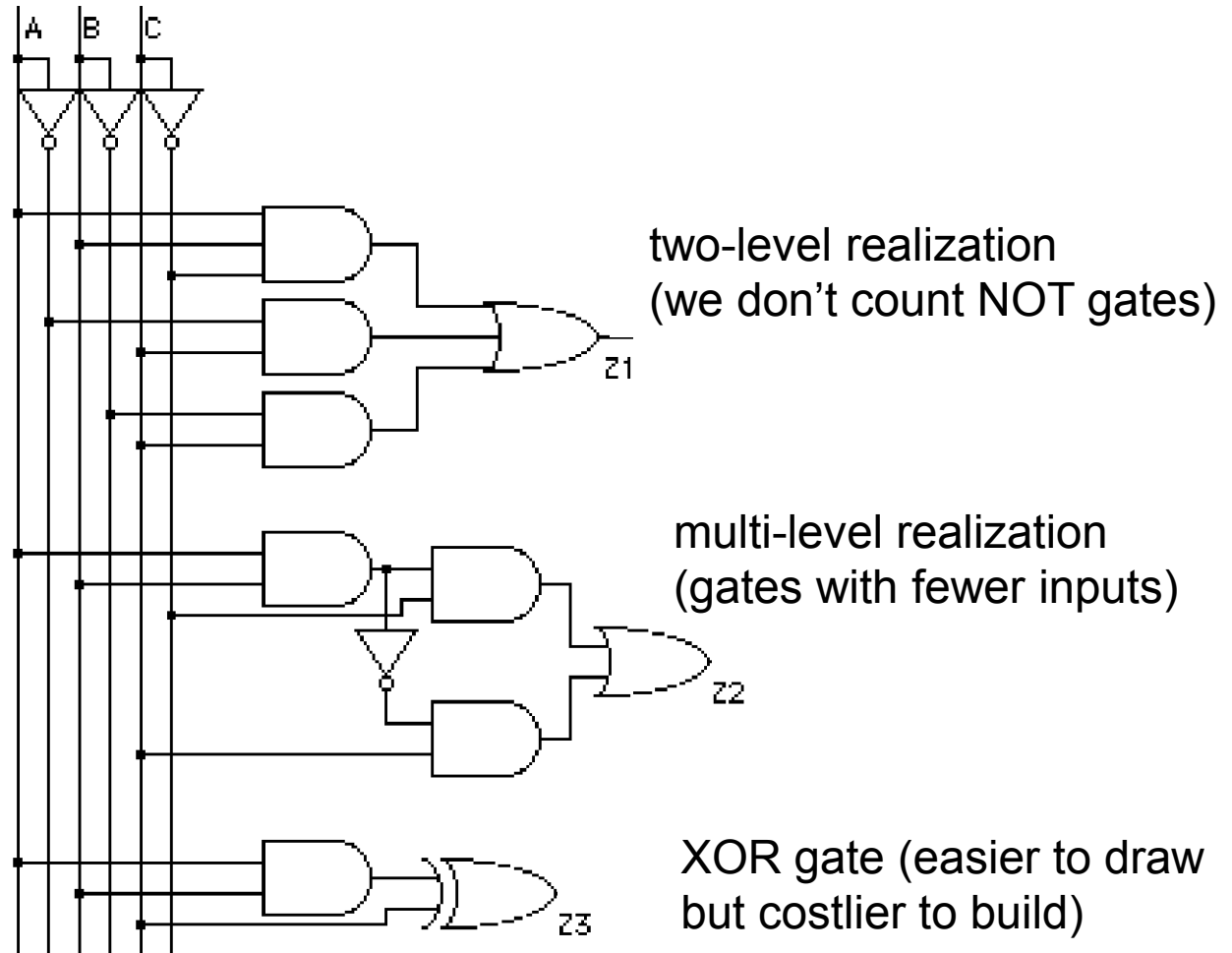


change in Y takes time to "propagate" through gates

Choosing different realizations of a function

A	B	C	Z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

$\bar{A} \cdot \bar{B} \cdot C$



Which realization is best?

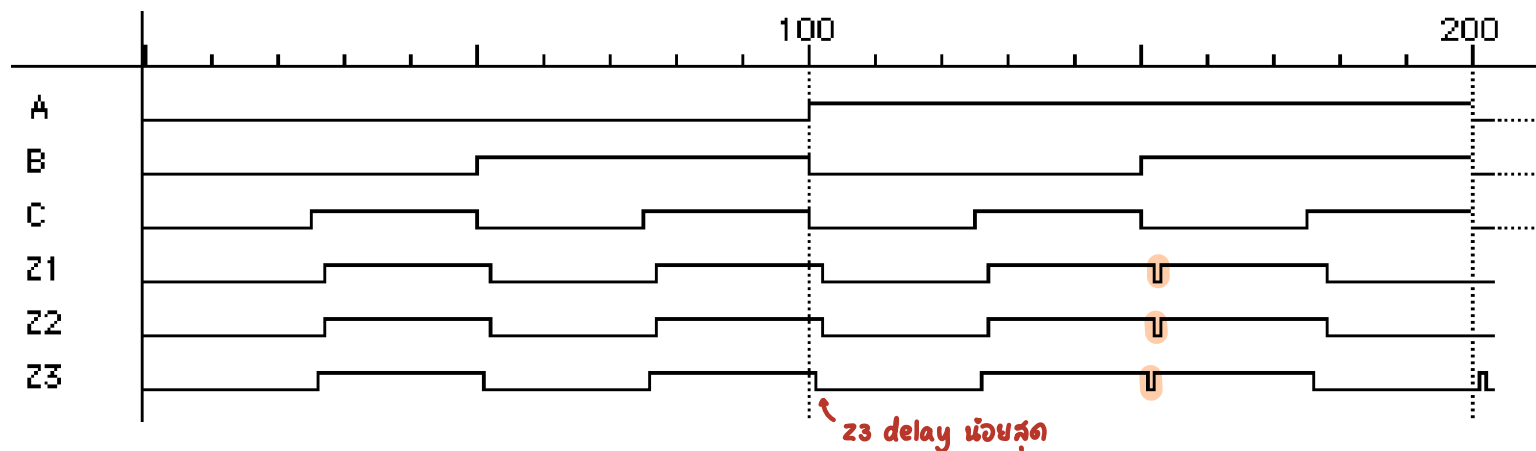
- Reduce number of inputs
 - literal: input variable (complemented or not)
 - can approximate cost of logic gate as 2 transistors per literal
 - why not count inverters?
 - fewer literals means less transistors
 - smaller circuits
 - fewer inputs implies faster gates
 - gates are smaller and thus also 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 – these could be bad, it depends
 - variations due to differences in number of gate levels and structure
- The 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

Canonical forms

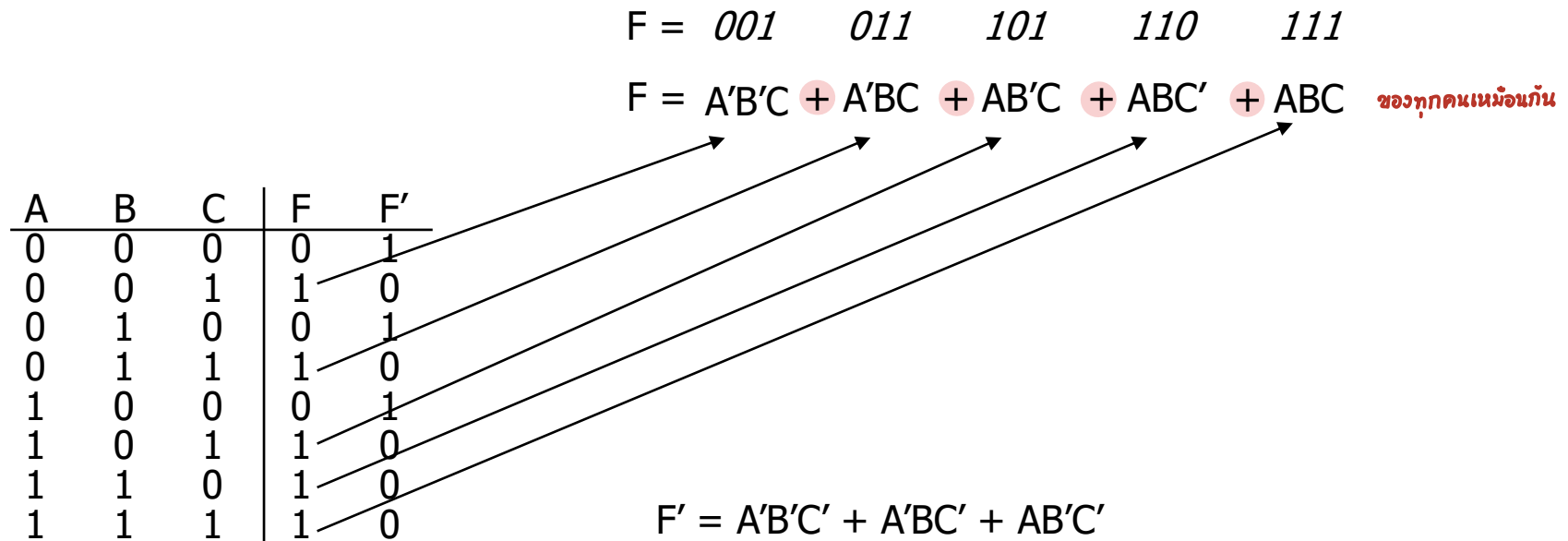
- Truth table is the unique signature of a Boolean function
- The same truth table can have many gate realizations
- Canonical forms
 - standard forms for a Boolean expression
 - provides a unique algebraic signature

Sum-of-products canonical forms

(SoP)

(DNF)

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




Sum-of-products canonical form (cont'd)

■ Product term (or minterm)

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

A	B	C	minterms	
0	0	0	$A'B'C'$	m0
0	0	1	$A'B'C$	m1
0	1	0	$A'BC'$	m2
0	1	1	$A'BC$	m3
1	0	0	$AB'C'$	m4
1	0	1	$AB'C$	m5
1	1	0	ABC'	m6
1	1	1	ABC	m7

short-hand notation for
minterms of 3 variables



F in canonical form:

$$\begin{aligned}F(A, B, C) &= \Sigma m(1,3,5,6,7) \\&= m1 + m3 + m5 + m6 + m7 \\&= A'B'C + A'BC + AB'C + ABC' + ABC\end{aligned}$$

canonical form \neq minimal form เขียนแล้วเหมือนกันหมด?

$$\begin{aligned}F(A, B, C) &= A'B'C + A'BC + AB'C + ABC + ABC' \\&= (A'B' + A'B + AB' + AB)C + ABC' \\&= ((A' + A)(B' + B))C + ABC' \\&= C + ABC' \\&= ABC' + C \\&= AB + C\end{aligned}$$

Product-of-sums canonical form

(PoS)

(CNF)

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

A	B	C	F	F'
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

$F = 000$
 $F = (A + B + C) (A + B' + C) (A' + B + C)$

(SoP) $\overline{F'} = \overline{A'B'C' + A'BC' + AB'C'}$
 de Morgan's $F = (A + B + C) \cdot (A + \overline{B} + c) \cdot (\overline{A} + B + c)$

ลูกตัวที่ได้อ 0
 ตารางข้าง

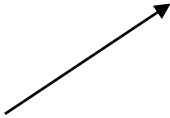
$$F' = (A + B + C') (A + B' + C') (A' + B + C') (A' + B' + C) (A' + B' + C')$$

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)

A	B	C	maxterms	
0	0	0	$A+B+C$	M0
0	0	1	$A+B+C'$	M1
0	1	0	$A+B'+C$	M2
0	1	1	$A+B'+C'$	M3
1	0	0	$A'+B+C$	M4
1	0	1	$A'+B+C'$	M5
1	1	0	$A'+B'+C$	M6
1	1	1	$A'+B'+C'$	M7

short-hand notation for
maxterms of 3 variables



F in canonical form:

$$\begin{aligned} F(A, B, C) &= \prod M(0,2,4) \\ &= M0 \bullet M2 \bullet M4 \\ &= (A + B + C) (A + B' + C) (A' + B + C) \end{aligned}$$

canonical form \neq minimal form

$$\begin{aligned} F(A, B, C) &= (A + B + C) (A + B' + C) (A' + B + C) \\ &= (A + B + C) (A + B' + C) \\ &\quad (A + B + C) (A' + B + C) \\ &= (A + C) (B + C) \end{aligned}$$

S-o-P, P-o-S, and de Morgan's theorem

SoP $\xleftrightarrow{\text{de Morgan's}}$ PoS

- 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

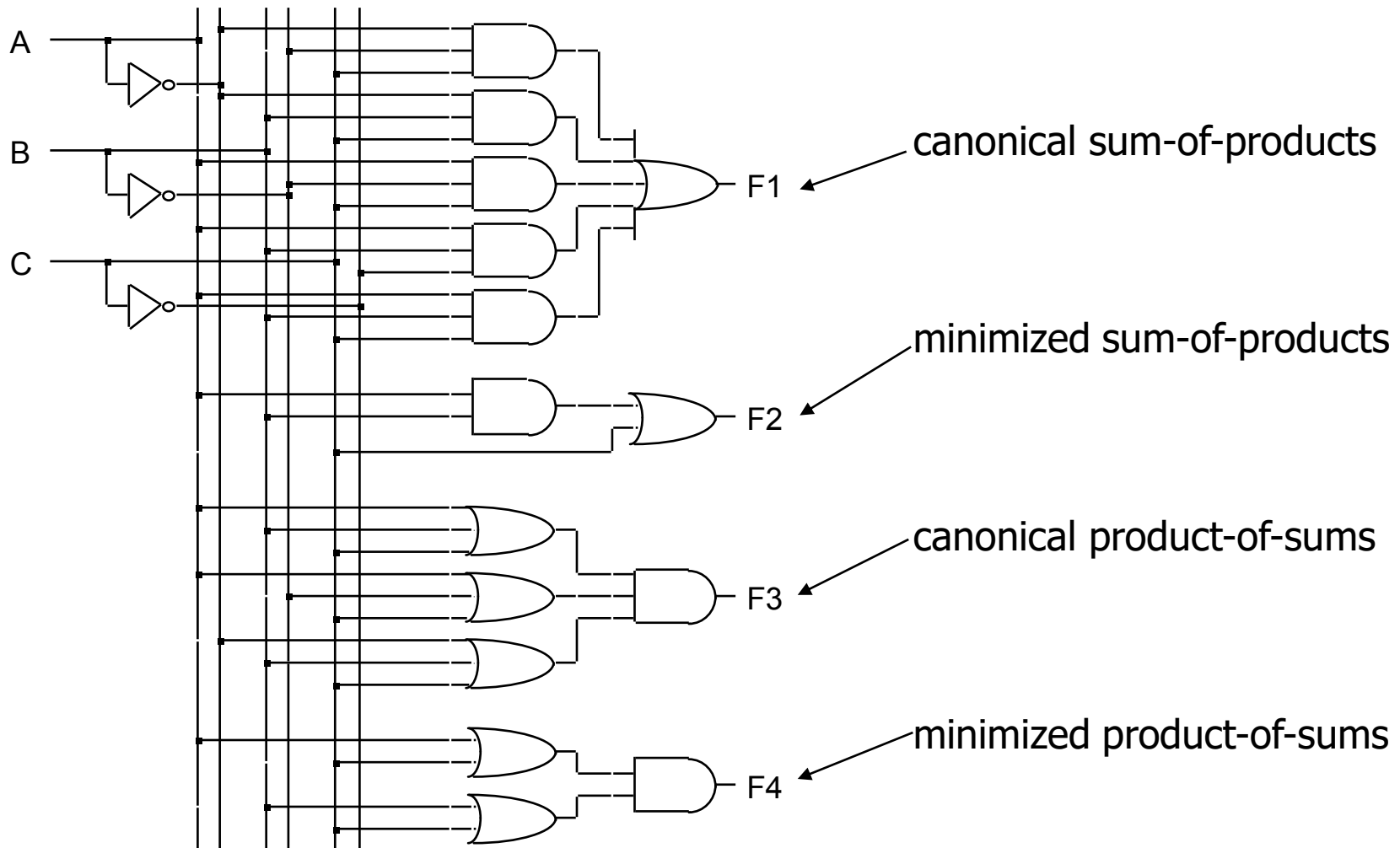
- $F' = (A + B + C')(A + B' + C')(A' + B + C')(A' + B' + C)(A' + B' + C')$

- Apply de Morgan's

- $(F')' = ((A + B + C')(A + B' + C')(A' + B + C')(A' + B' + C)(A' + B' + C'))'$

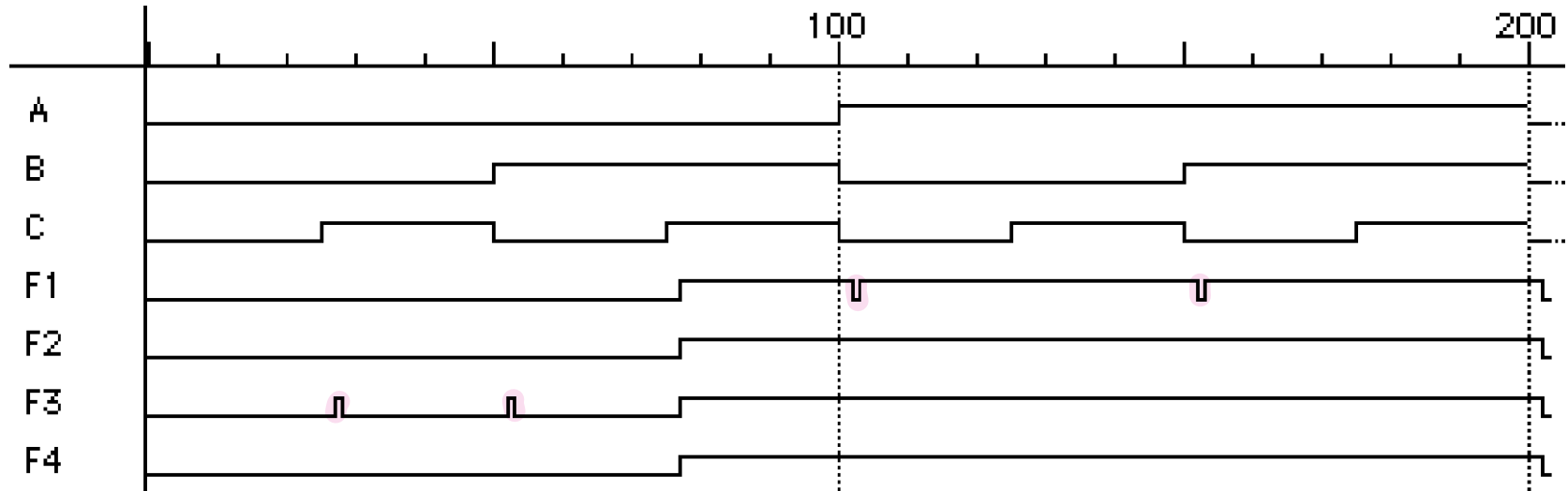
- $F = A'B'C + A'BC + AB'C + ABC' + ABC$

Four alternative two-level implementations of $F = AB + C$



Waveforms for the four alternatives

- Waveforms are essentially identical
 - except for timing hazards (glitches) ← *clock* *clock*
 - 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., $F(A,B,C) = \Sigma m(1,3,5,6,7) \quad F'(A,B,C) = \Sigma m(0,2,4)$
- Maxterm expansion of F to maxterm expansion of F'
 - use maxterms whose indices do not appear
 - e.g., $F(A,B,C) = \Pi M(0,2,4) \quad F'(A,B,C) = \Pi M(1,3,5,6,7)$

Incompletely specified functions

- Example: binary coded decimal increment by 1
 - BCD digits encode the decimal digits 0 – 9
in the bit patterns 0000 – 1001

A	B	C	D	W	X	Y	Z	
0	0	0	0	0	0	0	1	
0	0	0	1	0	0	1	0	off-set of W
0	0	1	0	0	0	1	1	
0	0	1	1	0	1	0	0	on-set of W
0	1	0	0	0	1	0	1	
0	1	0	1	0	1	1	0	don't care (DC) set of W
0	1	1	0	0	1	1	1	
0	1	1	1	1	0	0	0	
1	0	0	0	1	0	0	1	
1	0	0	1	0	0	0	0	
1	0	1	0	X	X	X	X	these inputs patterns should never be encountered in practice – "don't care" about associated output values, can be exploited in minimization <small>ทำให้วงจรเล็กลง</small>
1	0	1	1	X	X	X	X	
1	1	0	0	X	X	X	X	
1	1	0	1	X	X	X	X	
1	1	1	0	X	X	X	X	
1	1	1	0	X	X	X	X	
1	1	1	1	X	X	X	X	

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:
 - $Z = m_0 + m_2 + m_4 + m_6 + m_8 + d_{10} + d_{11} + d_{12} + d_{13} + d_{14} + d_{15}$
 - $Z = \Sigma [m(0,2,4,6,8) + d(10,11,12,13,14,15)]$
 - $Z = M_1 \cdot M_3 \cdot M_5 \cdot M_7 \cdot M_9 \cdot D_{10} \cdot D_{11} \cdot D_{12} \cdot D_{13} \cdot D_{14} \cdot D_{15}$
 - $Z = \Pi [M(1,3,5,7,9) \cdot D(10,11,12,13,14,15)]$

Combinational logic summary

- Logic functions, truth tables, and switches
 - NOT, AND, OR, NAND, NOR, XOR, . . . , minimal set
- Axioms and theorems of Boolean algebra
 - proofs by re-writing and perfect induction
- Gate logic
 - networks of Boolean functions and their time behavior
- Canonical forms
 - two-level and incompletely specified functions
- Simplification
 - a start at understanding two-level simplification
- Later
 - automation of simplification
 - multi-level logic
 - time behavior
 - hardware description languages
 - design case studies