Gate Level Minimization Chapter 3

EECE 256

Dr. Sidney Fels

Steven Oldridge

Topics

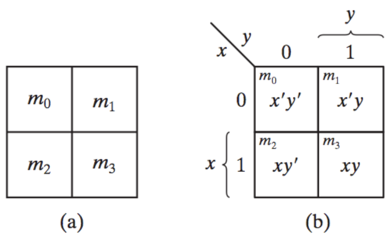
- Using maps to simplify Boolean functions
- Product of sums simplification
- Don't cares in the map
- NAND and NOR
- XOR and Parity
- a brief intro to HDL

Using geometry to simplify

- Remember when we did:
  - F = xy + xy' = x(y+y') = x? or
  - F = xy + x'y = y(x+x') = y?
- Finding these A+A' groups can be done easily with a map
- But, have to create the map carefully
- Easiest to see illustrated…

n-variable maps, Karnaugh maps, K-maps

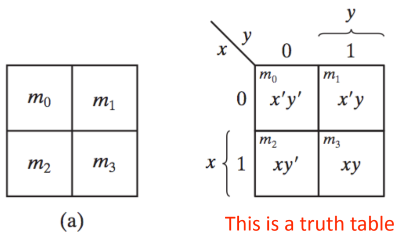
- let's look at two variable maps



9/23/10 (c) S. Fels, since 2010 4

n-variable maps, Karnaugh maps, K-maps

- let's look at two variable maps

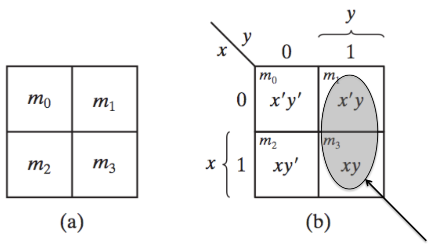


This is a truth table redrawn

9/23/10 (c) S. Fels, since 20 5

n-variable maps, Karnaugh maps, K-maps

- let's look at two variable maps



Notice: (x+x') here

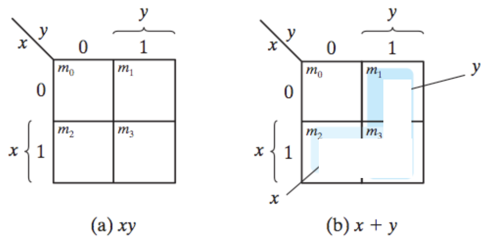
9/23/10 (c) S. Fels, since 2010

K-maps

- So, if we have a function, just use the map as a truth table and circle the ones
  - adjacent columns or rows indicate where simplification can happen

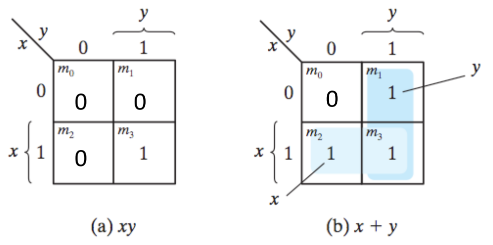
2 variable map example

F = m3 F = m1 + m2 + m3

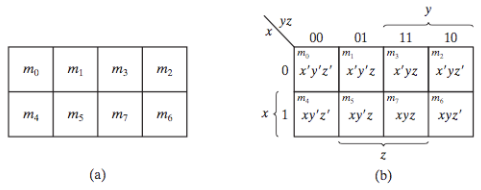


2 variable map example

F = m3 F = m1 + m2 + m3



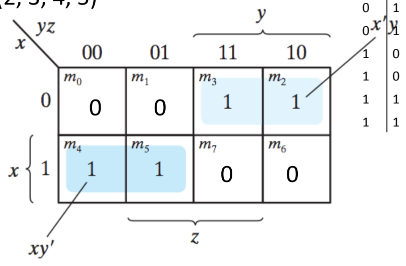
3 variable map example



3 variable map example

F = Σ(2, 3, 4, 5)

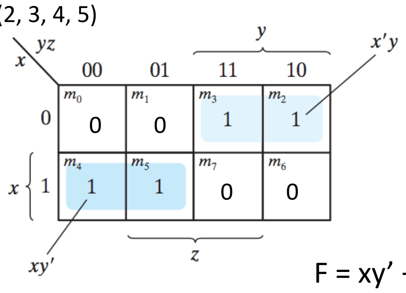
x	y	z	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0



yz
x
00 01 11 10
y
$x'y$
0 m_0 0 m_1 0 m_3 1 m_2 1
x 1 m_4 1 m_5 1 m_7 0 m_6 0
xy'
z

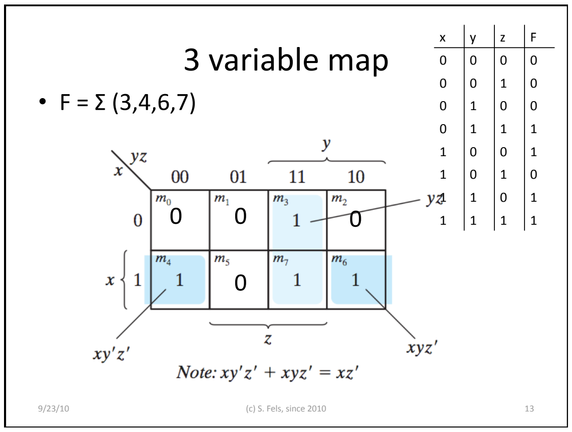
3 variable map example

F = Σ(2, 3, 4, 5)



yz
x
00 01 11 10
y
$x'y$
0 m_0 0 m_1 0 m_3 1 m_2 1
x 1 m_4 1 m_5 1 m_7 0 m_6 0
xy'
z

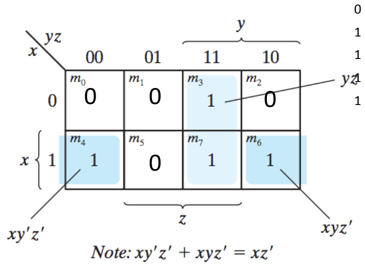
F = xy' + x'y



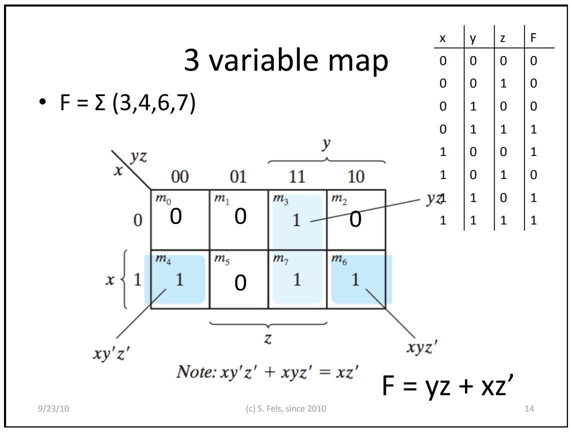
3 variable map

- F = Σ (3,4,6,7)

x	y	z	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1



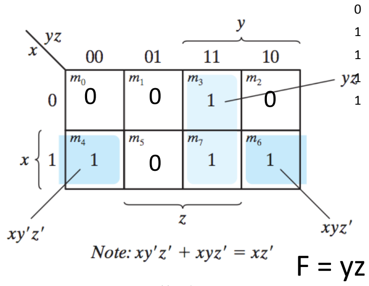
Note: $xy'z' + xyz' = xz'$



3 variable map

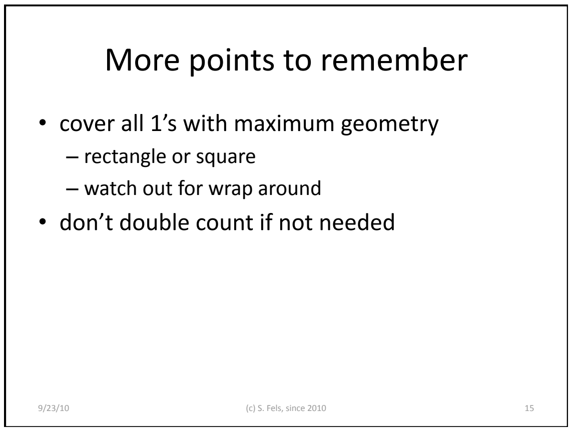
- F = Σ (3,4,6,7)

x	y	z	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1



Note: $xy'z' + xyz' = xz'$

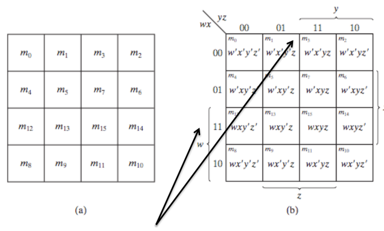
F = yz + xz'



More points to remember

- cover all 1's with maximum geometry
  - rectangle or square
  - watch out for wrap around
- don't double count if not needed

4 variable map

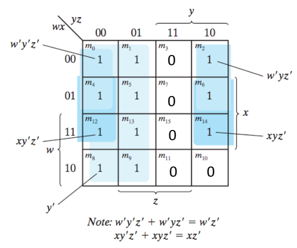


yz
wx
00
01
11
10
y
x
w
z
m_0 m_1 m_3 m_2
m_4 m_5 m_7 m_6
m_{12} m_{13} m_{15} m_{14}
m_8 m_9 m_{11} m_{10}
$w'x'y'z'$ $w'x'y'z$ $w'x'yz$ $w'x'yz'$
$w'xy'z'$ $w'xy'z$ $w'xyz$ $w'xyz'$
$wxy'z'$ $wxy'z$ $wxyz$ $wxyz'$
$wx'y'z'$ $wx'y'z$ $wx'yz$ $wx'yz'$
(a)
(b)

Don’t forget Gray coding here

4 variable map

- F=Σ (0,1,2,4,5,6,8,9,12,13,14)



yz
wx
00
01
11
10
y
x
w
z
$w'y'z'$
$w'yz'$
$xy'z'$
xyz'
y'
m_0 1 m_1 1 m_3 0 m_2 1
m_4 1 m_5 1 m_7 0 m_6 1
m_{12} 1 m_{13} 1 m_{15} 0 m_{14} 1
m_8 1 m_9 1 m_{11} 0 m_{10} 0
Note: $w'y'z' + w'yz' = w'z'$
$xy'z' + xyz' = xz'$

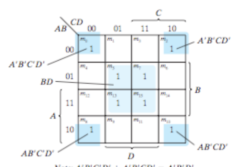
w	x	y	z	F
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Prime implicants

- largest box called Prime Implicant
- Prime implicants that have to be there are called:
  - essential prime implicants

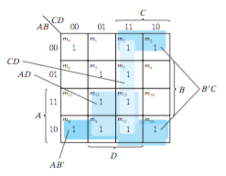
For example

- F = Σ(0,2,3,5,7,8,9,10,11,13,15)



Note: $A'B'C'D' + A'B'CD' = A'B'D'$
$AB'C'D' + AB'CD' = AB'D'$
$A'B'D' + AB'D' = B'D'$

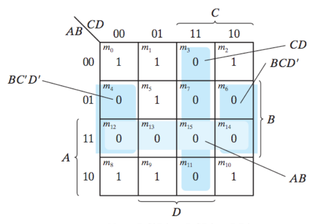
(a) Essential prime implicants BD and $B'D'$



(b) Prime implicants CD, $B'C$, AD, and AB'

You can easily do product-of-sum too

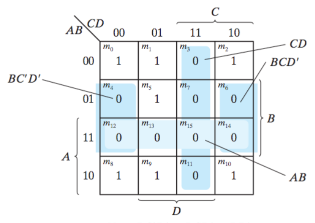
- F(A,B,C,D)=Σ(0,1,2,5,8,9,10)
  - circle the 0's to get F' and use de'Morgans



Note: $BC'D' + BCD' = BD'$

You can easily do product-of-sum too

- F'(A,B,C,D)=AB + CD + BD'
- F(A,B,C,D) = F'' = (A' + B')(C' + D')(B' + D)



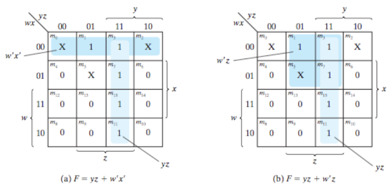
Note: $BC'D' + BCD' = BD'$

Don't care conditions

- Often, there are parts of the function that are unused values
  - i.e., BCD only uses 0-9 encodings, so the others don't matter
  - in this case, you can choose either a 0 or 1 for it to make your simplification better

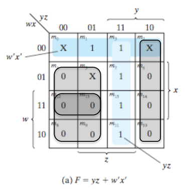
Example

- F=Σ(1,3,7,11,15)
- d = Σ(0,2,5)



don't care Example

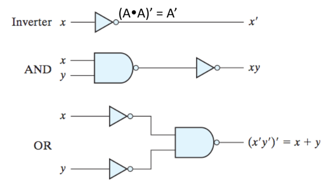
- Of course, if you want product-of-sum form, circle 0's and x to get minimal forms



F' = y'x + wy + yz'
F = (y + x')(w' + y')(y' + z)

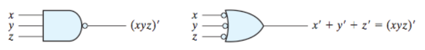
NAND and NOR

- NAND: Universal gate (any digital circuit can be implemented using only NAND gates)
- We just have to show that AND, OR and NOT can be implemented with NANDs



Another way to look at it:

- two ways to draw a NAND gate

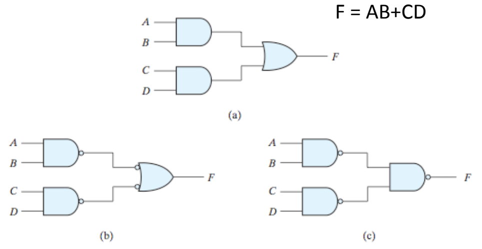


(a) AND-invert (b) Invert-OR

- Suggests a way to create all NAND gate implementation
  - move the bubbles around in a sum-of-products implementation…
  - you can also use algebra

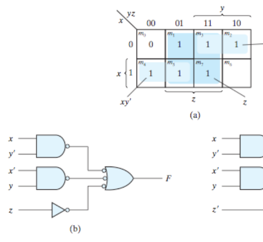
Moving the inverters for all-NAND

F = AB+CD



NAND only implementations: ex 1

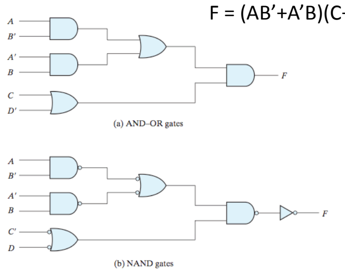
x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1



F = xy' +x'y + z

multi-level all-NAND

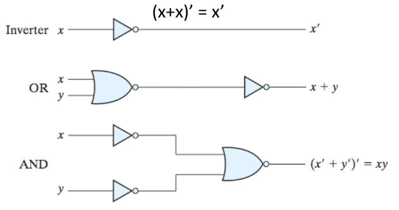
F = (AB'+A'B)(C+D')



all-NOR implementation

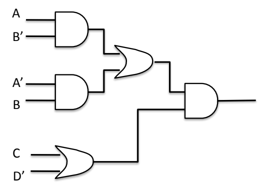
- Same idea, just start with product-of-sum notation

(x+x)' = x'



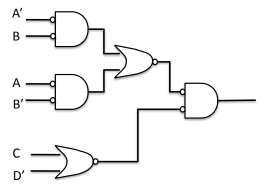
all NOR implementation

F=(AB'+A'B)(C+D')



all NOR implementation

F=(AB'+A'B)(C+D')



XOR and Parity

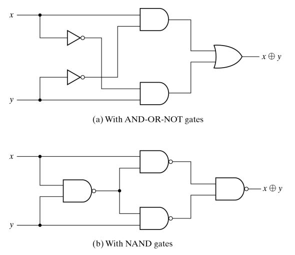
- Notice, in K-map we sometime get a checker board pattern…

XOR

- Exclusive-OR
  - x ⊕ y = x'y + xy'

x	y	F
0	0	0
0	1	1
1	0	1
1	1	0

		x: 0	x: 1
y	0	0	1
	1	1	0

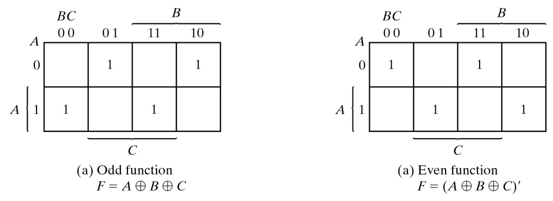


(a) With AND-OR-NOT gates

(b) With NAND gates

Fig. 3-32 Exclusive-OR Implementations

3 input XOR – Even/Odd checker



A \ BC	00	01	11	10
0		1		1
1	1		1	

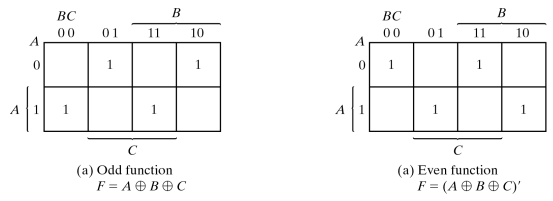
(a) Odd function
$F = A \oplus B \oplus C$

A \ BC	00	01	11	10
0	1		1	
1		1		1

(a) Even function
$F = (A \oplus B \oplus C)'$

Fig. 3-33 Map for a Three-variable Exclusive-OR Function

3 input XOR – Even/Odd checker



A \ BC	00	01	11	10
0		1		1
1	1		1	

(a) Odd function
$F = A \oplus B \oplus C$

A \ BC	00	01	11	10
0	1		1	
1		1		1

(a) Even function
$F = (A \oplus B \oplus C)'$

Fig. 3-33 Map for a Three-variable Exclusive-OR Function

F=Σ(1,3,5,7)
odd # of 1's only

F=Σ(0,2,4,6)
even number of 1's only

Even/Odd checker extends to multiple inputs

- called Parity
- Useful for error checking and correcting

Example: Design a 3-bit even parity generator

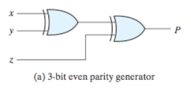
x	y	z	P
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Example 1: Design a 3-bit even parity generator (total bits even)

x	y	z	P
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

$$\begin{array}{c|c|cccc} & & & yz & & \\ & & 00 & 01 & 11 & 10 \\ \hline x & 0 & 0 & 1 & 0 & 1 \\ & 1 & 1 & 0 & 1 & 0 \end{array}$$

$$P = x \oplus y \oplus z$$



(a) 3-bit even parity generator

Transmit x,y,z,P – always should have even # of bits

Example 2: Design a 3-bit w/even parity checker (4 bits should be even)

P	x	y	z	E
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

Example 2: Design a 3-bit w/even parity checker (4 bits should be even)

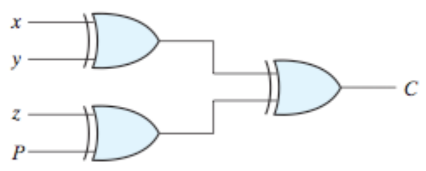
P	x	y	z	E
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Px \ yz	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

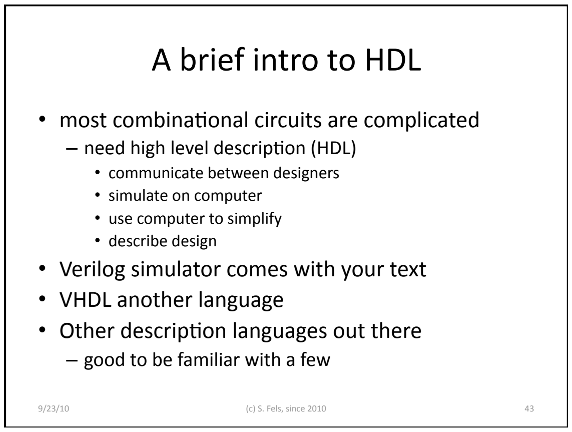
$$E = P \oplus x \oplus y \oplus z$$

- Outputs a 1 if the data has an odd number of 1's.

4bit Even Parity Checker

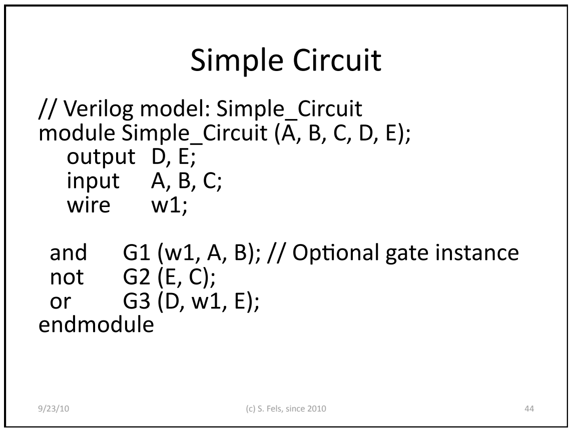


(b) 4-bit even parity checker



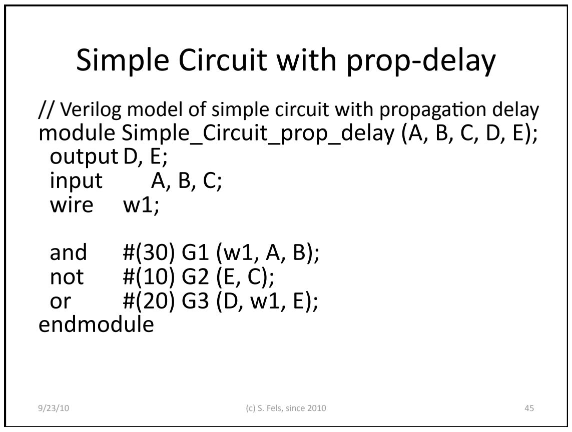
A brief intro to HDL

- most combinational circuits are complicated
  - need high level description (HDL)
    - communicate between designers
    - simulate on computer
    - use computer to simplify
    - describe design
- Verilog simulator comes with your text
- VHDL another language
- Other description languages out there
  - good to be familiar with a few



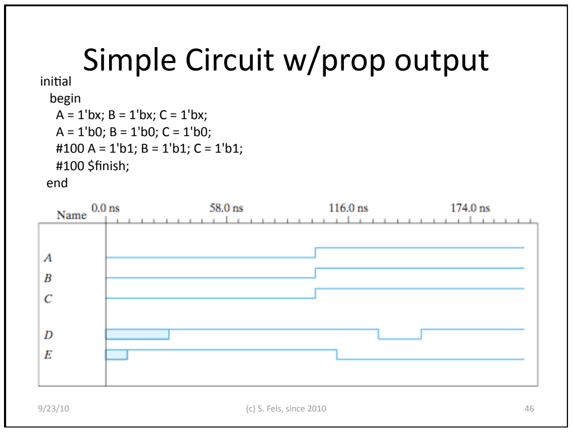
Simple Circuit

```
// Verilog model: Simple_Circuit
module Simple_Circuit (A, B, C, D, E);
    output  D, E;
    input   A, B, C;
    wire    w1;

  and     G1 (w1, A, B); // Optional gate instance
  not     G2 (E, C);
  or      G3 (D, w1, E);
endmodule
```



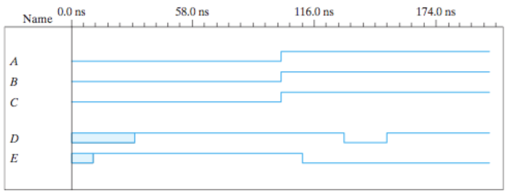
Simple Circuit with prop-delay

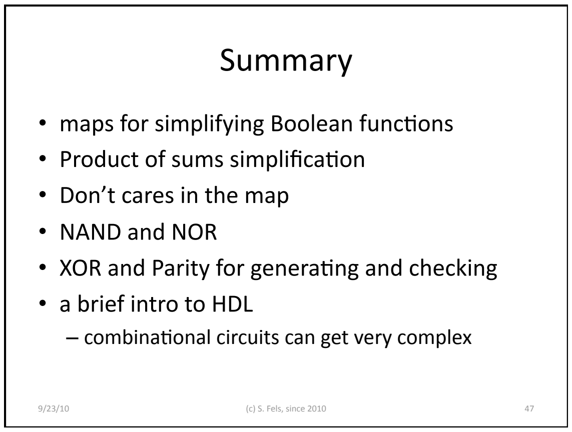
```
// Verilog model of simple circuit with propagation delay
module Simple_Circuit_prop_delay (A, B, C, D, E);
  output D, E;
  input      A, B, C;
  wire    w1;

  and     #(30) G1 (w1, A, B);
  not     #(10) G2 (E, C);
  or      #(20) G3 (D, w1, E);
endmodule
```



Simple Circuit w/prop output

```
initial
  begin
    A = 1'bx; B = 1'bx; C = 1'bx;
    A = 1'b0; B = 1'b0; C = 1'b0;
    #100 A = 1'b1; B = 1'b1; C = 1'b1;
    #100 $finish;
  end
```





Summary

- maps for simplifying Boolean functions
- Product of sums simplification
- Don’t cares in the map
- NAND and NOR
- XOR and Parity for generating and checking
- a brief intro to HDL
  - – combinational circuits can get very complex