Digital Electronics – Combinational Logic

Dr. I. J. Wassell

Introduction

Aims

- To familiarise students with
  - Combinational logic circuits
  - Sequential logic circuits
  - How digital logic gates are built using transistors
  - Design and build of digital logic systems

Course Structure

- 12 Lectures
- Hardware Labs
  - 6 Workshops
  - 7 sessions, each one 2.5h, alternate weeks
  - Thu. 10.30 to 1.00 or 1.30 to 4.00, beginning week 3
  - In Intel Lab. (SW11), William Gates Building (WGB)
  - In groups of 2

Objectives

- At the end of the course you should
  - Be able to design and construct simple digital electronic systems
  - Be able to understand and apply Boolean logic and algebra – a core competence in Computer Science
  - Be able to understand and build state machines

Books

- Lots of books on digital electronics, e.g.,
  - D. M. Harris and S. L. Harris, 'Digital Design and Computer Architecture,' Morgan Kaufmann, 2007 (1st Ed.), 2012 (2nd Ed.).
  - R. H. Katz, 'Contemporary Logic Design,' Benjamin/Cummings, 1994.
  - J. P. Hayes, 'Introduction to Digital Logic Design,' Addison-Wesley, 1993.
- Electronics in general (inc. digital)
  - P. Horowitz and W. Hill, 'The Art of Electronics,' CUP, 1989.

12/07/2016

Other Points

- This course is a prerequisite for
  - Computer Design, ECAD and Architecture Practical Classes (Part IB)
  - Comparative Architectures, System-on-Chip Design (Part II)
- Keep up with lab work and get it ticked.
- Have a go at supervision questions plus any others your supervisor sets.
- Remember to try questions from past papers

Semiconductors to Computers

- Increasing levels of complexity
  - Transistors built from semiconductors
  - Logic gates built from transistors
  - Logic functions built from gates
  - Flip-flops built from logic
  - Counters and sequencers from flip-flops
  - Microprocessors from sequencers
  - Computers from microprocessors

Semiconductors to Computers

- Increasing levels of abstraction:
  - Physics
  - ***Transistors***
  - ***Gates***
  - ***Logic***
  - Microprogramming (Computer Design Course)
  - Assembler (Computer Design Course)
  - Programming Languages (Compilers Course)
  - Applications

Combinational Logic

Introduction to Logic Gates

- We will introduce Boolean algebra and logic gates
- Logic gates are the building blocks of digital circuits

Logic Variables

- Different names for the same thing
  - Logic variables
  - Binary variables
  - Boolean variables
- Can only take on 2 values, e.g.,
  - TRUE or False
  - ON or OFF
  - 1 or 0

Logic Variables

- In electronic circuits the two values can be represented by e.g.,
  - High voltage for a 1
  - Low voltage for a 0
- Note that since only 2 voltage levels are used, the circuits have greater immunity to electrical noise

Uses of Simple Logic

- Example – Heating Boiler
  - If chimney is not blocked and the house is cold and the pilot light is lit, then open the main fuel valve to start boiler.
    - b = chimney blocked
    - c = house is cold
    - p = pilot light lit
    - v = open fuel valve
  - So in terms of a logical (Boolean) expression
    - v = (NOT b) AND c AND p

Logic Gates

- Basic logic circuits with one or more inputs and one output are known as *gates*
- *Gates* are used as the building blocks in the design of more complex digital logic circuits
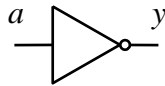
Representing Logic Functions

- There are several ways of representing logic functions:
  - Symbols to represent the gates
  - Truth tables
  - Boolean algebra
- We will now describe commonly used gates

NOT Gate

Symbol   Truth-table   Boolean
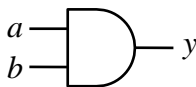
a ▷○ y

a	y
0	1
1	0

$y = \bar{a}$

- A NOT gate is also called an 'inverter'
- y is only TRUE if a is FALSE
- Circle (or 'bubble') on the output of a gate implies that it as an inverting (or complemented) output

AND Gate

Symbol   Truth-table   Boolean

a — b — y

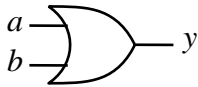| a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

$y = a.b$

- y is only TRUE only if a is TRUE and b is TRUE
- In Boolean algebra AND is represented by a dot .

OR Gate

Symbol   Truth-table   Boolean

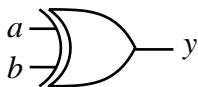| a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

$$y = a + b$$

- y is TRUE if a is TRUE or b is TRUE (or both)
- In Boolean algebra OR is represented by a plus sign $+$

EXCLUSIVE OR (XOR) Gate
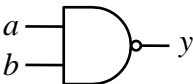
Symbol   Truth-table   Boolean

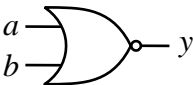| a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

$$y = a \oplus b$$

- y is TRUE if a is TRUE or b is TRUE (but not both)
- In Boolean algebra XOR is represented by an $\oplus$ sign

NOT AND (NAND) Gate

Symbol   Truth-table   Boolean

a b	y
0 0	1
0 1	1
1 0	1
1 1	0

$$y = \overline{a.b}$$

- y is TRUE if a is FALSE or b is FALSE (or both)
- y is FALSE only if a is TRUE and b is TRUE

NOT OR (NOR) Gate

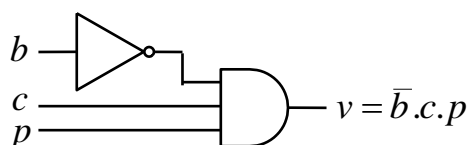Symbol   Truth-table   Boolean

a b	y
0 0	1
0 1	0
1 0	0
1 1	0

$$y = \overline{a + b}$$

- y is TRUE only if a is FALSE and b is FALSE
- y is FALSE if a is TRUE or b is TRUE (or both)

Boiler Example

- If chimney is not blocked and the house is cold and the pilot light is lit, then open the main fuel valve to start boiler.

b = chimney blocked c = house is cold

p = pilot light lit v = open fuel valve

$$v = \bar{b}.c.p$$

Boolean Algebra

- In this section we will introduce the laws of Boolean Algebra
- We will then see how it can be used to design *combinational logic* circuits
- Combinational logic circuits do not have an internal stored state, i.e., they have no memory. Consequently the output is solely a function of the current inputs.
- Later, we will study circuits having a stored internal state, i.e., sequential logic circuits.

Boolean Algebra

OR   AND

$$a + 0 = a \qquad a.0 = 0$$
$$a + a = a \qquad a.a = a$$
$$a + 1 = 1 \qquad a.1 = a$$
$$a + \bar{a} = 1 \qquad a.\bar{a} = 0$$

- AND takes precedence over OR, e.g.,

$$a.b + c.d = (a.b) + (c.d)$$

Boolean Algebra

- Commutation
  $$a + b = b + a$$
  $$a.b = b.a$$
- Association
  $$(a + b) + c = a + (b + c)$$
  $$(a.b).c = a.(b.c)$$
- Distribution
  $$a.(b + c + \ldots) = (a.b) + (a.c) + \ldots$$
  $$a + (b.c.\ldots) = (a + b).(a + c).\ldots \qquad \text{NEW}$$
- Absorption
  $$a + (a.c) = a \qquad \text{NEW}$$
  $$a.(a + c) = a \qquad \text{NEW}$$

Boolean Algebra - Examples

Show

$$a.(\overline{a}+b) = a.b$$
$$a.(\overline{a}+b) = a.\overline{a} + a.b = 0 + a.b = a.b$$

Show

$$a + (\overline{a}.b) = a + b$$
$$a + (\overline{a}.b) = (a + \overline{a}).(a + b) = 1.(a + b) = a + b$$

Boolean Algebra

- A useful technique is to expand each term until it includes one instance of each variable (or its compliment). It may be possible to simplify the expression by cancelling terms in this expanded form e.g., to prove the absorption rule:

$$a + a.b = a$$

$$a.b + a.\overline{b} + a.b = a.b + a.\overline{b} = a.(b + \overline{b}) = a.1 = a$$

Boolean Algebra - Example

Simplify

$x.y + \bar{y}.z + x.z + x.y.z$

$x.y.z + x.y.\bar{z} + x.\bar{y}.z + \bar{x}.\bar{y}.z + x.y.z + x.\bar{y}.z + x.y.z$

$x.y.z + x.y.\bar{z} + x.\bar{y}.z + \bar{x}.\bar{y}.z$

$x.y.(z + \bar{z}) + \bar{y}.z.(x + \bar{x})$

$x.y.1 + \bar{y}.z.1$

$x.y + \bar{y}.z$

DeMorgan's Theorem

$$\overline{a + b + c + \ldots} = \bar{a}.\bar{b}.\bar{c}. \ldots$$
$$\overline{a.b.c. \ldots} = \bar{a} + \bar{b} + \bar{c} + \ldots$$

- In a simple expression like $a + b + c$ (or $a.b.c$) simply change all operators from OR to AND (or vice versa), complement each term (put a bar over it) and then complement the whole expression, i.e.,

$$a + b + c + \ldots = \overline{\bar{a}.\bar{b}.\bar{c}. \ldots}$$
$$a.b.c. \ldots = \overline{\bar{a} + \bar{b} + \bar{c} + \ldots}$$

DeMorgan's Theorem

- For 2 variables we can show $\overline{a+b} = \overline{a}.\overline{b}$ and $\overline{a.b} = \overline{a}+\overline{b}$ using a truth table.

a b	$\overline{a+b}$	$\overline{a.b}$	$\overline{a}$ $\overline{b}$	$\overline{a}.\overline{b}$	$\overline{a}+\overline{b}$
0 0	1	1	1 1	1	1
0 1	0	1	1 0	0	1
1 0	0	1	0 1	0	1
1 1	0	0	0 0	0	0

- Extending to more variables by induction

$$\overline{a+b+c} = \overline{(a+b).c} = (\overline{a}.\overline{b}).\overline{c} = \overline{a}.\overline{b}.\overline{c}$$

DeMorgan's Examples

- Simplify $\quad a.\overline{b} + a.(\overline{b+c}) + b.(\overline{b+c})$

$$= a.\overline{b} + a.\overline{b}.\overline{c} + b.\overline{b}.\overline{c} \quad \text{(DeMorgan)}$$
$$= a.\overline{b} + a.\overline{b}.\overline{c} \quad (b.\overline{b} = 0)$$
$$= a.\overline{b} \quad \text{(absorbtion)}$$

DeMorgan's Examples

- Simplify $(a.b.(c+\overline{b.d})+\overline{a.b}).c.d$

$= (a.b.(c+\overline{b}+\overline{d})+\overline{a}+\overline{b}).c.d$ (De Morgan)

$= (a.b.c+a.b.\overline{b}+a.b.\overline{d}+\overline{a}+\overline{b}).c.d$ (distribute)

$= (a.b.c+a.b.\overline{d}+\overline{a}+\overline{b}).c.d$ $(a.b.\overline{b}=0)$

$= a.b.c.d+a.b.\overline{d}.c.d+\overline{a}.c.d+\overline{b}.c.d$ (distribute)

$= a.b.c.d+\overline{a}.c.d+\overline{b}.c.d$ $(a.b.\overline{d}.c.d=0)$

$= (a.b+\overline{a}+\overline{b}).c.d$ (distribute)

$= (a.b+\overline{a.b}).c.d$ (DeMorgan)

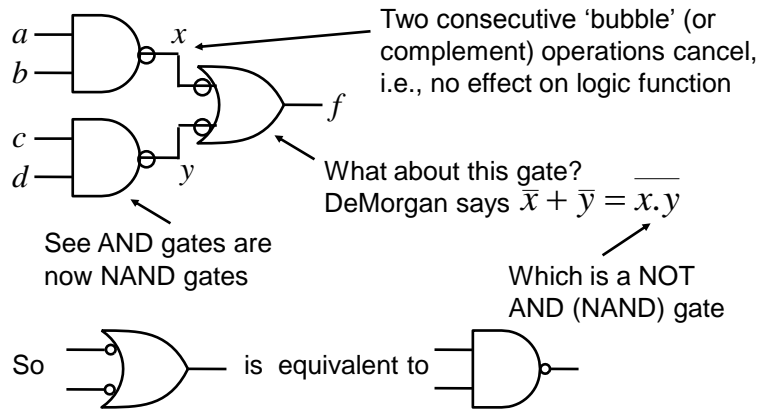$= c.d$ $(a.b+\overline{a.b}=1)$

DeMorgan's in Gates

- To implement the function $f = a.b+c.d$ we can use AND and OR gates
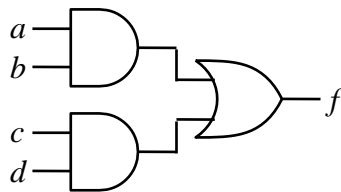
- However, sometimes we only wish to use NAND or NOR gates, since they are usually simpler and faster

DeMorgan's in Gates

- To do this we can use 'bubble' logic

Two consecutive 'bubble' (or complement) operations cancel, i.e., no effect on logic function

a ── x

b ──

f

c ──

d ── y

What about this gate?
DeMorgan says $\overline{x} + \overline{y} = \overline{x.y}$

See AND gates are now NAND gates

Which is a NOT AND (NAND) gate

So ─── is equivalent to ───

DeMorgan's in Gates

- So the previous function can be built using 3 NAND gates

a ──
b ──
f
c ──
d ──

a ──
b ──
f
c ──
d ──
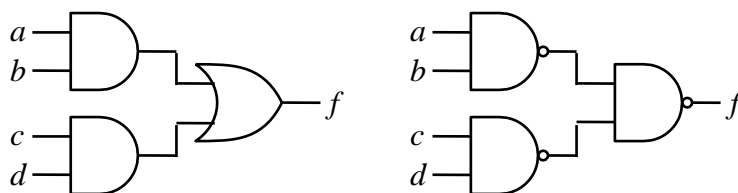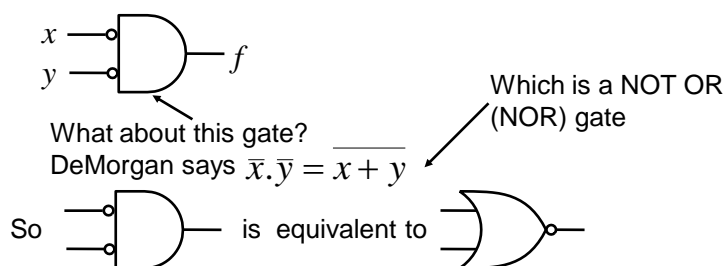
$$f = a.b + c.d$$

$$f = \overline{\overline{(a.b)}.\overline{(c.d)}}$$

DeMorgan's in Gates

- Similarly, applying 'bubbles' to the input of an AND gate yields

x y ---- f

What about this gate?
DeMorgan says $\overline{x}.\overline{y} = \overline{x + y}$

Which is a NOT OR (NOR) gate

So ---- is equivalent to ----

- Useful if trying to build using NOR gates

Logic Minimisation

- Any Boolean function can be implemented directly using combinational logic (gates)
- However, simplifying the Boolean function will enable the number of gates required to be reduced. Techniques available include:
    - Algebraic manipulation (as seen in examples)
    - Karnaugh (K) mapping (a visual approach)
    - Tabular approaches (usually implemented by computer, e.g., Quine-McCluskey)
- K mapping is the preferred technique for up to about 5 variables

Truth Tables

- f is defined by the following truth table

x	y	z	f	minterms
0	0	0	1	$\bar{x}.\bar{y}.\bar{z}$
0	0	1	1	$\bar{x}.\bar{y}.z$
0	1	0	1	$\bar{x}.y.\bar{z}$
0	1	1	1	$\bar{x}.y.z$
1	0	0	0	
1	0	1	0	
1	1	0	0	
1	1	1	1	$x.y.z$

- A *minterm* must contain all variables (in either complement or uncomplemented form)
  - Note variables in a minterm are ANDed together (conjunction)
  - One minterm for each term of f that is TRUE

- So $\bar{x}.y.z$ is a minterm but $y.z$ is not

Disjunctive Normal Form

- A Boolean function expressed as the disjunction (ORing) of its minterms is said to be in the Disjunctive Normal Form (DNF)

$$f = \bar{x}.\bar{y}.\bar{z} + \bar{x}.\bar{y}.z + \bar{x}.y.\bar{z} + \bar{x}.y.z + x.y.z$$

- A Boolean function expressed as the ORing of ANDed variables (not necessarily minterms) is often said to be in Sum of Products (SOP) form, e.g.,

$$f = \bar{x} + y.z \quad \text{Note functions have the same truth table}$$

Maxterms

- A maxterm of n Boolean variables is the disjunction (ORing) of all the variables either in complemented or uncomplemented form.
  - Referring back to the truth table for f, we can write,

    $$\bar{f} = x.\bar{y}.\bar{z} + x.\bar{y}.z + x.y.\bar{z}$$

    Applying De Morgan (and complementing) gives

    $$f = (\bar{x} + y + z).(\bar{x} + y + \bar{z}).(\bar{x} + \bar{y} + z)$$

    So it can be seen that the maxterms of f are effectively the minterms of $\bar{f}$ with each variable complemented

Conjunctive Normal Form

- A Boolean function expressed as the conjunction (ANDing) of its maxterms is said to be in the Conjunctive Normal Form (CNF)

  $$f = (\bar{x} + y + z).(\bar{x} + y + \bar{z}).(\bar{x} + \bar{y} + z)$$

- A Boolean function expressed as the ANDing of ORed variables (not necessarily maxterms) is often said to be in Product of Sums (POS) form, e.g.,

  $$f = (\bar{x} + y).(\bar{x} + z)$$

Logic Simplification

- As we have seen previously, Boolean algebra can be used to simplify logical expressions. This results in easier implementation

  Note: The DNF and CNF forms are not simplified.

- However, it is often easier to use a technique known as Karnaugh mapping

Karnaugh Maps

- Karnaugh Maps (or K-maps) are a powerful visual tool for carrying out simplification and manipulation of logical expressions having up to 5 variables

- The K-map is a rectangular array of cells
  - Each possible state of the input variables corresponds uniquely to one of the cells
  - The corresponding output state is written in each cell

K-maps example

- From truth table to K-map

x	y	z	f
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Note that the logical state of the variables follows a Gray code, i.e., only one of them changes at a time

The exact assignment of variables in terms of their position on the map is not important

---

K-maps example

- Having plotted the minterms, how do we use the map to give a simplified expression?

  - Group terms
    - Having size equal to a power of 2, e.g., 2, 4, 8, etc.
    - Large groups best since they contain fewer variables
    - Groups can wrap around edges and corners

So, the simplified func. is,

$$f = \bar{x} + y.z \quad \text{as before}$$

K-maps – 4 variables

- K maps from Boolean expressions
  - Plot $f = \overline{a}.b + b.\overline{c}.\overline{d}$

	00	01	11	10
00				
01	1	1	1	1
11	1			
10				

- See in a 4 variable map:
  - 1 variable term occupies 8 cells
  - 2 variable terms occupy 4 cells
  - 3 variable terms occupy 2 cells, etc.

K-maps – 4 variables

- For example, plot

$$f = \overline{b} \qquad\qquad f = \overline{b}.\overline{d}$$

	00	01	11	10
00	1	1	1	1
01				
11				
10	1	1	1	1

	00	01	11	10
00	1			1
01				
11				
10	1			1

K-maps – 4 variables

- Simplify, $f = \bar{a}.b.\bar{d} + b.c.d + \bar{a}.b.\bar{c}.d + c.d$
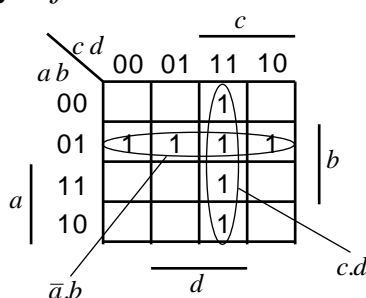
So, the simplified func. is,

$$f = \bar{a}.b + c.d$$

POS Simplification

- Note that the previous examples have yielded simplified expressions in the SOP form
  - Suitable for implementations using AND followed by OR gates, or only NAND gates (using DeMorgans to transform the result – see previous Bubble logic slides)
- However, sometimes we may wish to get a simplified expression in POS form
  - Suitable for implementations using OR followed by AND gates, or only NOR gates

POS Simplification

- To do this we group the zeros in the map
  - i.e., we simplify the complement of the function
- Then we apply DeMorgans and complement
- Use 'bubble' logic if NOR only implementation is required

POS Example

- Simplify $f = \bar{a}.b + b.\bar{c}.\bar{d}$ into POS form.
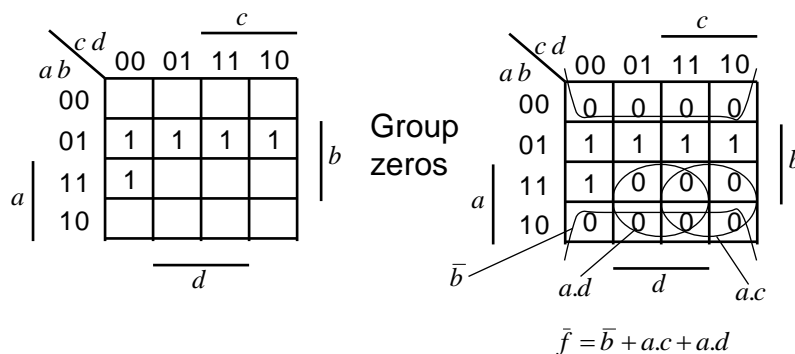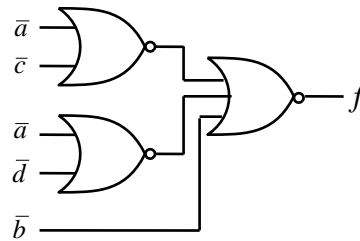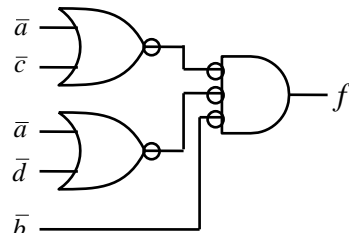

$$\bar{f} = \bar{b} + a.c + a.d$$

POS Example

- Applying DeMorgans to
$$\bar{f} = \bar{b} + a.c + a.d$$
gives,
$$\bar{f} = \overline{b.(\bar{a}+\bar{c}).(\bar{a}+\bar{d})}$$
$$f = b.(\bar{a}+\bar{c}).(\bar{a}+\bar{d})$$
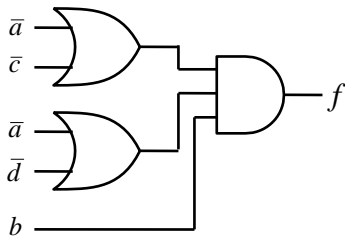

Expression in POS form

- Apply DeMorgans and take complement, i.e., $\bar{f}$ is now in SOP form
- Fill in zeros in table, i.e., plot $\bar{f}$
- Fill remaining cells with ones, i.e., plot f
- Simplify in usual way by grouping ones to simplify f

Don't Care Conditions

- Sometimes we do not care about the output value of a combinational logic circuit, i.e., if certain input combinations can never occur, then these are known as *don't care conditions*.
- In any simplification they may be treated as 0 or 1, depending upon which gives the simplest result.
  - For example, in a K-map they are entered as Xs

Don't Care Conditions - Example

- Simplify the function $f = \overline{a}.\overline{b}.d + \overline{a}.c.d + a.c.d$

With don't care conditions, $\overline{a}\,\overline{b}.\overline{c}.\overline{d}, \overline{a}\,\overline{b}.c.\overline{d}, \overline{a}.b.\overline{c}.d$
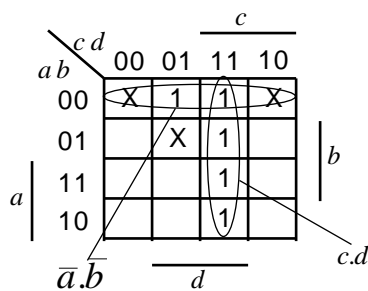
See only need to include Xs if they assist in making a bigger group, otherwise can ignore.

$$f = \overline{a}.\overline{b} + c.d \quad \text{or,} \quad f = \overline{a}.d + c.d$$

Some Definitions

- Cover – A term is said to cover a minterm if that minterm is part of that term
- Prime Implicant – a term that cannot be further combined
- Essential Prime Implicant – a prime implicant that covers a minterm that no other prime implicant covers
- Covering Set – a minimum set of prime implicants which includes all essential terms plus any other prime implicants required to cover all minterms

Some Definitions - Example
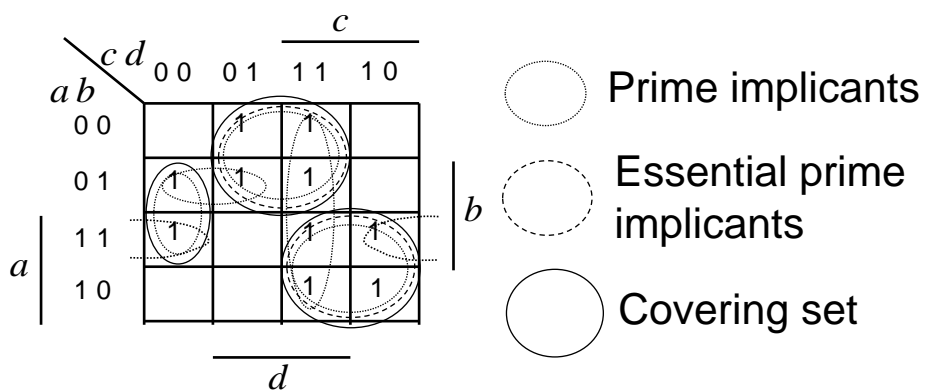
Tabular Simplification

- Except in special cases or for sparse truth tables, the K-map method is not practical beyond 6 variables
- A systematic approach known as the *Quine-McCluskey (Q-M) Method* finds the minimised representation of any Boolean expression
- It is a tabular method that ensures all the prime implicants are found and can be automated for use on a computer

Q-M Method

- The Q-M Method has 2 steps:
    - First a table, known as the *QM implication table*, is used to find all the prime implicants;
    - Next the minimum cover set is found using the *prime implicant* chart.
- We will use a 4 variable example to show the method in operation:
    - Minterms are: 4,5,6,8,9,10,13
    - Don't cares are: 0,7,15.

Q-M Method

- The first step is to list all the minterms and don't cares in terms of their minterm indices represented as a binary number
  - Note the entries are grouped according to the number of 1s in the binary representation
  - The 1st column contains the minterms
  - After applying the method, the 2nd column will contain 3 variable terms. Similarly for subsequent columns.

Q-M Method

- The method begins by listing groups of minterms and don't cares in groups containing ascending numbers of 1s with a blank line between the groups
  - Thus the first group has zero ones, the second group has a single 1 and the third has two 1s and so on
- We next apply the so called *uniting theorem* iteratively as follows

Q-M Method – Uniting Theorem

- Compare elements in the 1st group (no 1s) with all elements in the 2nd group. If they differ by a single bit, it means the terms are adjacent (think K-map)
- Adjacent terms are placed in the 2nd column with the single bit that differs replaced by a dash (-). Terms in the 1st column that contribute to a term in the second are *ticked*, i.e., they are *not* prime implicants.
- Now repeat for the groups in the 2nd column
- As before groups must differ only by a single bit but they must also have a – in the same position
- Groups in 2nd column that do not contribute to the 3rd column are marked with an asterix (*), i.e., they are prime implicants

Q-M – Implication Table

- Minterms are: 4,5,6,8,9,10,13
- Don't cares are: 0,7,15.

Column 1	Column 2	Column 3
0 0 0 0 ✓	0 - 0 0 *	0 1 - - *
0 1 0 0 ✓	- 0 0 0 *	- 1 - 1 *
1 0 0 0 ✓	0 1 0 - ✓	
0 1 0 1 ✓	0 1 - 0 ✓	
0 1 1 0 ✓	1 0 0 - *	
1 0 0 1 ✓	1 0 - 0 *	
1 0 1 0 ✓	0 1 - 1 ✓	
0 1 1 1 ✓	- 1 0 1 ✓	
1 1 0 1 ✓	0 1 1 - ✓	
1 1 1 1 ✓	1 - 0 1 *	
	- 1 1 1 ✓	
	1 1 - 1 ✓	

Q-M – Finding Min Cover

- The second step is to find the lowest number of prime implicants that cover the function – this is achieved using the *prime implicant chart*
- This chart is organised as follows:
  - Label columns with the minterm indices (don't include don't cares)
  - Label rows with minterms covered by a given prime implicant. To do this dashes (-) in a prime implicant are replaced by all combinations of 0s and 1s
  - Place an X in the (row, column) location if the minterm represented by the column index is covered by the prime implicant associated with the row
  - The next slide shows the initial prime implicant chart

Q-M – Prime Implicant Chart

	4	5	6	8	9	10	13
0,4 (0-0 0)	X						
0,8 (- 0 0 0)				X			
8,9 (1 0 0 -)				X	X		
8,10 (1 0 - 0)				X		X	
9,13 (1- 0 1)					X		X
4,5,6,7 (0 1 - -)	X	X	X				
5,7,13,15 (- 1 - 1)		X					X

* Terms in Implication Table

Minterms (exc. don't cares)

- Now we look for the essential prime implicants – These are indicated when there is only a single X in any column, i.e., This means there is a minterm covered by one and only prime implicant

33

Q-M – Prime Implicant Chart

- The essential terms must be included in the final cover
  - Draw lines in the column and row that have a X associated with an essential prime implicant and draw a box around the prime
  - These minterms are already covered by the essential primes

```
                          4 5 6 8 9 10 13
        0,4 (0-0 0)  X
        0,8 (-0 0 0)           X
        8,9 (1 0 0 -)          X X
       8,10 (1 0 -0) ..........X...(X).....
       9,13 (1-0 1)              X   X
    4,5,6,7 (0 1- -) x.x.(X)...........
  5,7,13,15 (-1-1)   X              X
```

Q-M – Prime Implicant Chart

- The essential prime implicants usually cover additional minterms.
  - We must also cross out any columns that have an X in a row associated with an essential prime since these minterms are already covered by the essential primes

```
                          4 5 6 8 9 10 13
        0,4 (0-0 0)  X
        0,8 (-0 0 0)           X
        8,9 (1 0 0 -)          X X
       8,10 (1 0 -0) +.........X...(X).....
       9,13 (1-0 1)              X   X
    4,5,6,7 (0 1- -) x.x.(X)...........
  5,7,13,15 (-1-1)   X              X
```

Q-M – Prime Implicant Chart

- We see 2 minterms are still uncovered (cols. 9 and 13)
  - The final step is to find as few primes as possible to cover the remaining minterms
  - We see the single prime implicant 1-01 covers both of them
  - The boxed terms show the final covering set

```
              4 5 6 8 9 10 13
   0,4 (0-0 0)  X
   0,8 (-0 0 0)        X
   8,9 (1 0 0 -)       X X
   8,10(1 0-0)         X    (X)
   9,13(1-0 1)             X     X
 4,5,6,7(01- -)  X X(X)
5,7,13,15 (-1-1)   X            X
```
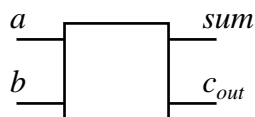

Binary Adders

Binary Adding Circuits

- We will now look at how binary addition may be implemented using combinational logic circuits. We will consider:
  - Half adder
  - Full adder
  - Ripple carry adder

Half Adder

- Adds together two, single bit binary numbers a and b (note: no carry input)
- Has the following truth table:

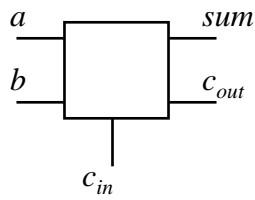| a b	c_{out} sum
0 0	0 0
0 1	0 1
1 0	0 1
1 1	1 0

a ⟶ [   ] ⟶ sum

b ⟶ [   ] ⟶ c_{out}

- By inspection:

$$sum = \overline{a}.b + a.\overline{b} = a \oplus b$$
$$c_{out} = a.b$$

Full Adder

- Adds together two, single bit binary numbers a and b (note: with a carry input)

a sum

b c_{out}

c_{in}

- Has the following truth table:

Full Adder

c_{in} a b	c_{out}	sum
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

$$sum = \bar{c}_{in}.\bar{a}.b + \bar{c}_{in}.a.\bar{b} + c_{in}.\bar{a}.\bar{b} + c_{in}.a.b$$

$$sum = \bar{c}_{in}.(\bar{a}.b + a.\bar{b}) + c_{in}.(\bar{a}.\bar{b} + a.b)$$

From DeMorgan

$$\bar{a}.\bar{b} + a.b = \overline{(a+b).(\bar{a} + \bar{b})}$$

$$= \overline{(a.\bar{a} + a.\bar{b} + b.\bar{a} + b.\bar{b})}$$

$$= \overline{(a.\bar{b} + b.\bar{a})}$$

So,

$$sum = \bar{c}_{in}.(\bar{a}.b + a.\bar{b}) + c_{in}.\overline{(\bar{a}.b + a.\bar{b})}$$

$$sum = \bar{c}_{in}.x + c_{in}.\bar{x} = c_{in} \oplus x = c_{in} \oplus a \oplus b$$

Full Adder

c_{in} a b	c_{out} sum
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

$$c_{out} = \bar{c}_{in}.a.b + c_{in}.\bar{a}.b + c_{in}.a.\bar{b} + c_{in}.a.b$$

$$c_{out} = a.b.(\bar{c}_{in} + c_{in}) + c_{in}.\bar{a}.b + c_{in}.a.\bar{b}$$

$$c_{out} = a.b + c_{in}.\bar{a}.b + c_{in}.a.\bar{b}$$

$$c_{out} = a.(b + c_{in}.\bar{b}) + c_{in}.\bar{a}.b$$

$$c_{out} = a.(b + c_{in}).(b + \bar{b}) + c_{in}.\bar{a}.b$$

$$c_{out} = b.(a + c_{in}.\bar{a}) + a.c_{in} = b.(a + c_{in}).(a + \bar{a}) + a.c_{in}$$

$$c_{out} = b.a + b.c_{in} + a.c_{in}$$

$$c_{out} = b.a + c_{in}.(b + a)$$

Full Adder

- Alternatively,

c_{in} a b	c_{out} sum
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

$$c_{out} = \bar{c}_{in}.a.b + c_{in}.\bar{a}.b + c_{in}.a.\bar{b} + c_{in}.a.b$$

$$c_{out} = c_{in}.(\bar{a}.b + a.\bar{b}) + a.b.(c_{in} + \bar{c}_{in})$$

$$c_{out} = c_{in}.(a \oplus b) + a.b$$

- Which is similar to previous expression except with the OR replaced by XOR

Ripple Carry Adder

- We have seen how we can implement a logic to add two, one bit binary numbers (inc. carry-in).
- However, in general we need to add together two, n bit binary numbers.
- One possible solution is known as the Ripple Carry Adder
  - This is simply n, full adders cascaded together

Ripple Carry Adder

- Example, 4 bit adder
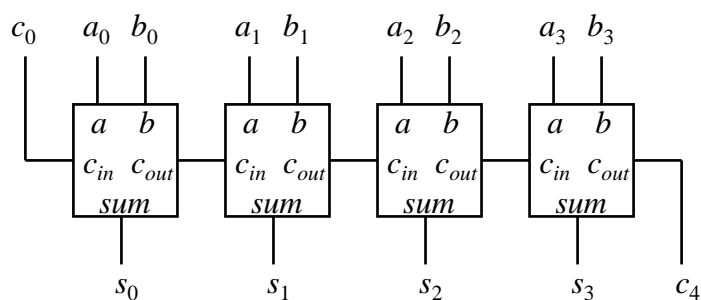
- Note: If we complement a and set c_o to one we have implemented $s = b - a$

To Speed up Ripple Carry Adder

- Abandon compositional approach to the adder design, i.e., do not build the design up from full-adders, but instead design the adder as a block of 2-level combinational logic with $2n$ inputs (+1 for carry in) and n outputs (+1 for carry out).
- Features
  - Low delay (2 gate delays)
  - Need some gates with large numbers of inputs (which are not available)
  - Very complex to design and implement (imagine the truth table!

To Speed up Ripple Carry Adder

- Clearly the 2-level approach is not feasible
- One possible approach is to make use of the full-adder blocks, but to generate the carry signals independently, using fast carry generation logic
- Now we do not have to wait for the carry signals to ripple from full-adder to full-adder before output becomes valid

Fast Carry Generation
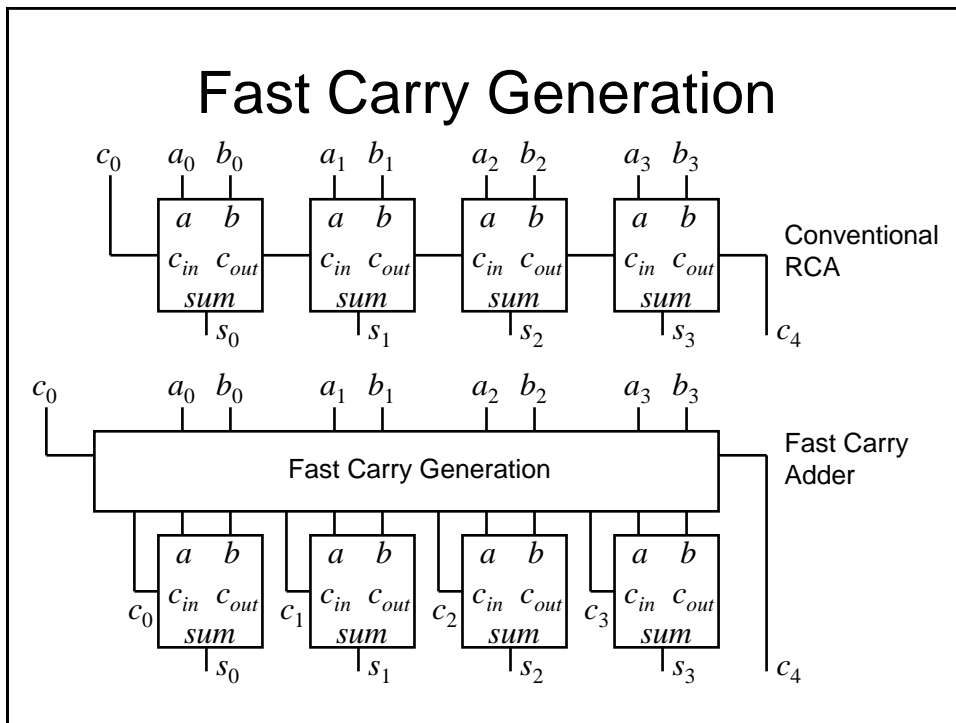
Fast Carry Generation

- We will now determine the Boolean equations required to generate the fast carry signals
- To do this we will consider the carry out signal, c_{out}, generated by a full-adder stage (say i), which conventionally gives rise to the carry in (c_{in}) to the next stage, i.e., c_{i+1}.

Fast Carry Generation

c_i	a	b	s_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Carry out always zero.
Call this *carry kill*

$$k_i = \bar{a}_i.\bar{b}_i$$

Carry out same as carry in.
Call this *carry propagate*

$$p_i = a_i \oplus b_i$$

Carry out generated independently of carry in.
Call this *carry generate*

$$g_i = a_i.b_i$$

Also (from before), $s_i = a_i \oplus b_i \oplus c_i$

Fast Carry Generation

- Also from before we have,

$$c_{i+1} = a_i.b_i + c_i.(a_i + b_i) \quad \text{or alternatively,}$$
$$c_{i+1} = a_i.b_i + c_i.(a_i \oplus b_i)$$

Using previous expressions gives,

$$c_{i+1} = g_i + c_i.p_i$$

So,

$$c_{i+2} = g_{i+1} + c_{i+1}.p_{i+1}$$
$$c_{i+2} = g_{i+1} + p_{i+1}.(g_i + c_i.p_i)$$
$$c_{i+2} = g_{i+1} + p_{i+1}.g_i + p_{i+1}.p_i.c_i$$

Fast Carry Generation

Similarly,

$$c_{i+3} = g_{i+2} + c_{i+2} \cdot p_{i+2}$$
$$c_{i+3} = g_{i+2} + p_{i+2} \cdot (g_{i+1} + p_{i+1} \cdot (g_i + c_i \cdot p_i))$$
$$c_{i+3} = g_{i+2} + p_{i+2} \cdot (g_{i+1} + p_{i+1} \cdot g_i) + p_{i+2} \cdot p_{i+1} \cdot p_i \cdot c_i$$

and

$$c_{i+4} = g_{i+3} + c_{i+3} \cdot p_{i+3}$$
$$c_{i+4} = g_{i+3} + p_{i+3} \cdot (g_{i+2} + p_{i+2} \cdot (g_{i+1} + p_{i+1} \cdot g_i) + p_{i+2} \cdot p_{i+1} \cdot p_i \cdot c_i)$$
$$c_{i+4} = g_{i+3} + p_{i+3} \cdot (g_{i+2} + p_{i+2} \cdot (g_{i+1} + p_{i+1} \cdot g_i)) + p_{i+3} \cdot p_{i+2} \cdot p_{i+1} \cdot p_i \cdot c_i$$

Fast Carry Generation

- So for example to generate c_4, i.e., $i = 0$,

$$c_4 = g_3 + p_3 \cdot (g_2 + p_2 \cdot (g_1 + p_1 \cdot g_0)) + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0$$
$$c_4 = G + Pc_0$$

where,

$$G = g_3 + p_3 \cdot (g_2 + p_2 \cdot (g_1 + p_1 \cdot g_0))$$
$$P = p_3 \cdot p_2 \cdot p_1 \cdot p_0$$

- See it is quick to evaluate this function

Fast Carry Generation

- We could generate all the carrys within an adder block using the previous equations
- However, in order to reduce complexity, a suitable approach is to implement say 4-bit adder blocks with only c_4 generated using fast generation.
  - This is used as the carry-in to the next 4-bit adder block
  - Within each 4-bit adder block, conventional RCA is used

Fast Carry Generation
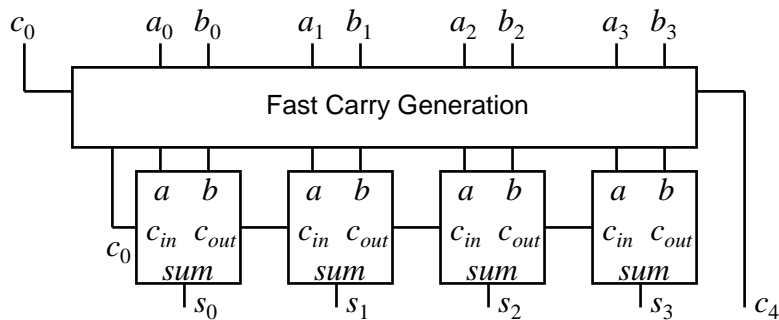
Combinational Logic Design

## Further Considerations

Multilevel Logic

- We have seen previously how we can minimise Boolean expressions to yield so called '2-level' logic implementations, i.e., SOP (ANDed terms ORed together) or POS (ORed terms ANDed together)
- Note also we have also seen an example of 'multilevel' logic, i.e., full adders cascaded to form a ripple carry adder – see we have more than 2 gates in cascade in the carry chain

Multilevel Logic

- Why use multilevel logic?
  - Commercially available logic gates usually only available with a restricted number of inputs, typically, 2 or 3.
  - System composition from sub-systems reduces design complexity, e.g., a ripple adder made from full adders
  - Allows Boolean optimisation across multiple outputs, e.g., common sub-expression elimination

Building Larger Gates

- Building a 6-input OR gate
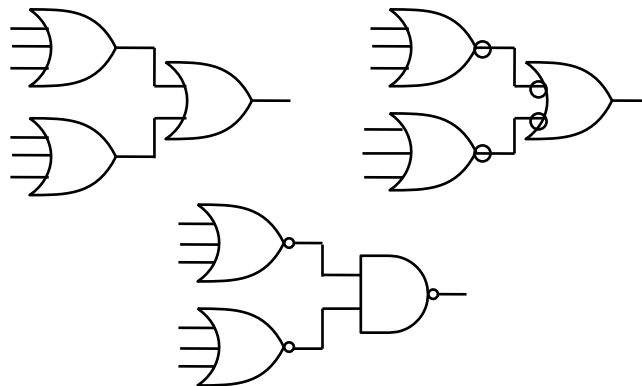
Common Expression Elimination

- Consider the following minimised SOP expression:

$$z = a.d.f + a.e.f + b.d.f + b.e.f + c.d.f + c.e.f + g$$

- Requires:
  - Six, 3 input AND gates, one 7-input OR gate – total 7 gates, 2-levels
  - 19 literals (the total number of times all variables appear)

Common Expression Elimination

- We can recursively factor out common literals

$$z = a.d.f + a.e.f + b.d.f + b.e.f + c.d.f + c.e.f + g$$

$$z = (a.d + a.e + b.d + b.e + c.d + c.e).f + g$$

$$z = ((a+b+c).d + (a+b+c).e).f + g$$

$$z = (a+b+c).(d+e).f + g$$

- Now express z as a number of equations in 2-level form:

$$x = a+b+c \qquad y = d+e \qquad z = x.y.f + g$$

- 4 gates, 9 literals, 3-levels

Gate Propagation Delay

- So, multilevel logic can produce reductions in implementation complexity. What is the downside?
- We need to remember that the logic gates are implemented using electronic components (essentially transistors) which have a finite switching speed.
- Consequently, there will be a finite delay before the output of a gate responds to a change in its inputs – *propagation delay*

Gate Propagation Delay

- The cumulative delay owing to a number of gates in cascade can increase the time before the output of a combinational logic circuit becomes valid
- For example, in the Ripple Carry Adder, the sum at its output will not be valid until any carry has 'rippled' through possibly every full adder in the chain – clearly the MSB will experience the greatest potential delay

Gate Propagation Delay

- As well as slowing down the operation of combinational logic circuits, gate delay can also give rise to so called '*Hazards*' at the output
- These *Hazards* manifest themselves as unwanted brief logic level changes (or *glitches*) at the output in response to changing inputs
- We will now describe how we can address these problems

Hazards

- Hazards are classified into two types, namely, static and dynamic
- Static Hazard – The output undergoes a momentary transition when *one input changes* when it is supposed to remain unchanged
- Dynamic Hazard – The output changes more than once when it is supposed to change just once
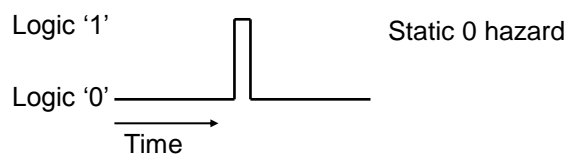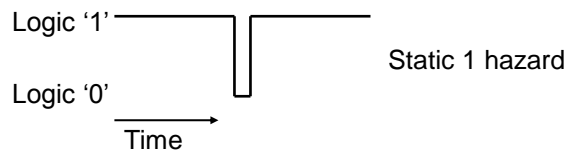
Timing Diagrams

- To visually represent Hazards we will use the so called *'timing diagram'*
- This shows the logical value of a signal as a function of time, for example the following timing diagram shows a transition from 0 to 1 and then back again
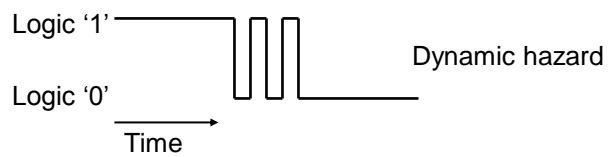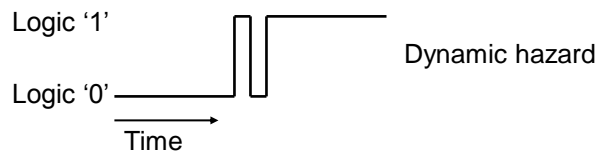
Logic '1'

Logic '0'
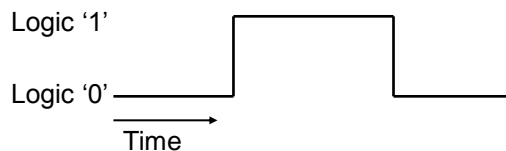
Time

Timing Diagrams

- Note that the timing diagram makes a number simplifying assumptions (to aid clarity) compared with a diagram which accurately shows the actual voltage against time
  - The signal only has 2 levels. In reality the signal may well look more 'wobbly' owing to electrical noise pick-up etc.
  - The transitions between logic levels takes place instantaneously, in reality this will take a finite time.

50

Static Hazard

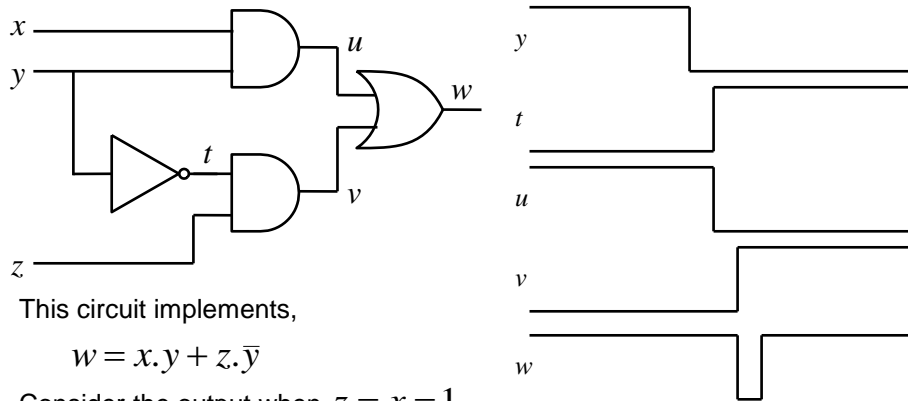Logic '1' ⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯ Static 1 hazard

Logic '0' ⎯⎯⎯⎯⎯⎯⎯

Time →

Logic '1' ⎯⎯⎯⎯⎯⎯⎯ Static 0 hazard

Logic '0' ⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯

Time →

Dynamic Hazard

Logic '1' Dynamic hazard

Logic '0' ⎯⎯⎯⎯

Time →

Logic '1' ⎯⎯⎯⎯⎯ Dynamic hazard

Logic '0' ⎯⎯⎯⎯

Time →

Static 1 Hazard

This circuit implements,

$$w = x.y + z.\overline{y}$$

Consider the output when $z = x = 1$ and y changes from 1 to 0

Hazard Removal

- To remove a 1 hazard, draw the K-map of the output concerned. Add another term which overlaps the essential terms
- To remove a 0 hazard, draw the K-map of the complement of the output concerned. Add another term which overlaps the essential terms (representing the complement)
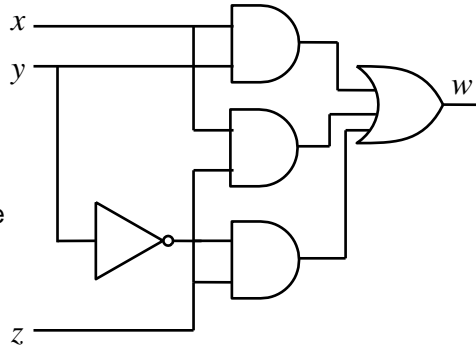- To remove dynamic hazards – not covered in this course!

Removing the static 1 hazard

$w = x.y + z.\bar{y}$

Extra term added to remove hazard, consequently,

$w = x.y + z.\bar{y} + x.z$

Beyond Simple Logic Gates

- **Multiplexor (Mux)/selector** – chooses 1 of many inputs to steer to its single output under the direction of control inputs, e.g., if the input to a circuit can come from several places a Mux is one way to funnel the multiple sources selectively to the single input.

Multiplexor

- The hazard example is actually a 2-to-1 Mux, i.e., it can select either input x or z to appear at output w under control of y

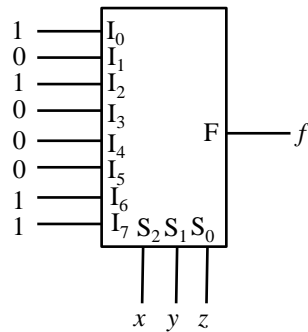| x	z	y	w
0	0	0	0
0	1	0	1
1	0	0	0
1	1	0	1
0	0	1	0
0	1	1	0
1	0	1	1
1	1	1	1
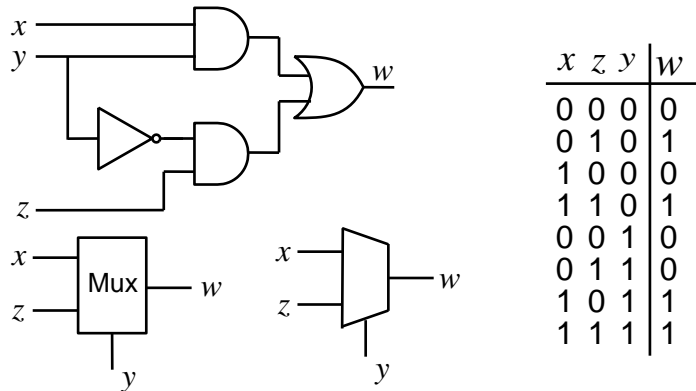
Multiplexor

- Clearly an n-to-1 Mux is also possible. For example, an 8-to-1 Mux will need 3 control inputs.
- A Mux can also be used to implement combinational logic functions. For example, an 8 input Mux can be used to implement functions expressed as a sum of minterms, i.e., DNF.

$$f = \bar{x}.\bar{y}.\bar{z} + \bar{x}.y.\bar{z} + x.y.\bar{z} + \bar{x}.y.z + x.y.z$$

Multiplexor

$$f = \bar{x}.\bar{y}.\bar{z} + \bar{x}.y.\bar{z} + x.y.\bar{z} + x.y.z$$

```
1 ──── I₀
0 ──── I₁
1 ──── I₂
0 ──── I₃
0 ──── I₄        F ──── f
0 ──── I₅
1 ──── I₆
1 ──── I₇ S₂ S₁ S₀

         x  y  z
```

- The control inputs are used to select the minterms required at the output. The Mux is sometimes called a hardware *look-up* table.

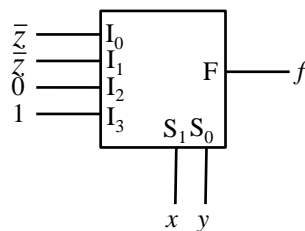# Multiplexor

- In this example if we use one of the inputs as a variable, then we can get away with a 4-to-1 Mux

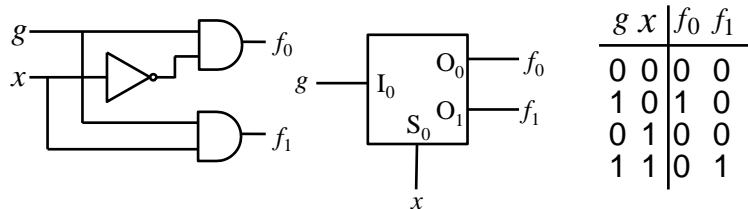$$f = \bar{x}.\bar{y}.\bar{z} + \bar{x}.y.\bar{z} + x.y.\bar{z} + x.y.z$$

$$f = (\bar{x}.\bar{y} + \bar{x}.y).\bar{z} + x.y.(z + .\bar{z})$$

$$f = (\bar{x}.\bar{y} + \bar{x}.y).\bar{z} + x.y$$

```
z̄ ──── I₀
z̄ ──── I₁        F ──── f
0 ──── I₂
1 ──── I₃  S₁ S₀

         x  y
```

Demultiplexor

- A demultiplexor is the opposite of a Mux: a single input is directed to exactly one of its outputs
- The truth table for a 1-to-2 Demux (i.e., 1 select (control) input and 2 outputs is:

g	x	f_0	f_1
0	0	0	0
1	0	1	0
0	1	0	0
1	1	0	1

Demultiplexor

- Clearly an 1-to-n Demux is also possible. For example, an 1-to-8 Demux will need 3 control inputs.
- A related function is a *Decoder*. In this case the input g is effectively connected to a logical 1. So the truth table becomes:

g	x	f_0	f_1
0	0	0	0
1	0	1	0
0	1	0	0
1	1	0	1

g =1

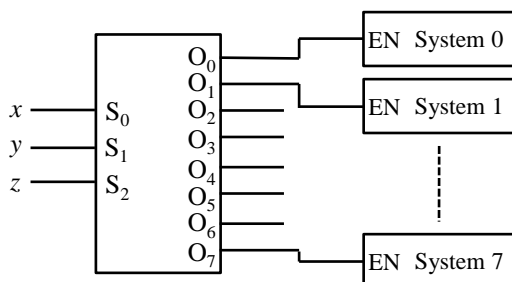| x	f_0	f_1
0	1	0
1	0	1

- Thus only one output is logical 1 at a time

Decoder

- Clearly an 1-to-n Decoder is also possible. For example, an 1-to-8 Decoder will need 3 control inputs.
- A typical application would be to 'Enable (EN)' 1 out-of-n logic sub-systems.

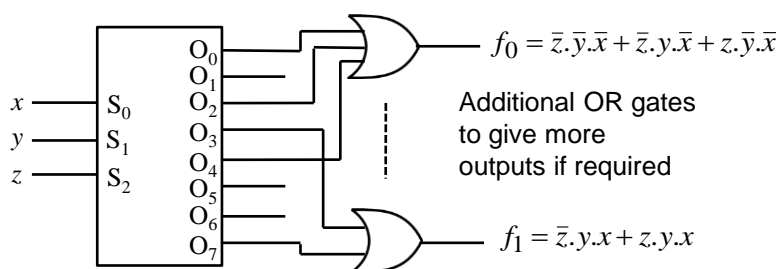- So, letting x=1, y=z=0 will enable System 1

Decoder

- We can see that an 1-to-n Decoder will generate all the possible minterms having n variables.
- Consequently, a logical expression having DNF form can be implemented by ORing together the required minterms at the decoder output.
- Multiple output logic blocks can be created by using multiple OR gates at the decoder output, i.e., one for each output.

Decoder

- Decoder implementation of a 3 variable, 2 output combinational logic block.

$$f_0 = \bar{z}.\bar{y}.\bar{x} + \bar{z}.y.\bar{x} + z.\bar{y}.\bar{x}$$

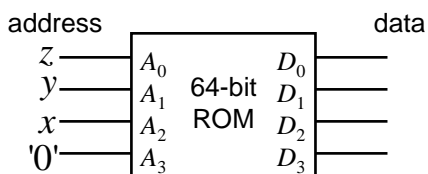Additional OR gates to give more outputs if required

$$f_1 = \bar{z}.y.x + z.y.x$$

Even More Ways to Implement Combinational Logic

- We have seen how combinational logic can be implemented using logic gates (e.g., AND, OR), Mux and Demux.
- However, it is also possible to generate combinational logic functions using memory devices, e.g., Read Only Memories (ROMs)

ROM Overview

- A ROM is a data storage device:
  - Usually written into once (either at manufacture or using a programmer)
  - Read at will
  - Essentially is a look-up table, where a group of input lines (say n) is used to specify the *address* of locations holding m-bit *data* words
  - For example, if $n = 4$, then the ROM has $2^4 = 16$ possible locations. If $m = 4$, then each location can store a 4-bit word
  - So, the total number of bits stored is $m \times 2^n$, i.e., 64 in the example (very small!) ROM

ROM Example

address   data

z —— A_0 64-bit D_0 ——
y —— A_1 ROM D_1 ——
x —— A_2 D_2 ——
'0' —— A_3 D_3 ——

address (decimal)	x	y	z	f	D_3	D_2	D_1	D_0
0	0	0	0	1	X	X	X	1
1	0	0	1	1	X	X	X	1
2	0	1	0	1	X	X	X	1
3	0	1	1	1	X	X	X	1
4	1	0	0	0	X	X	X	0
5	1	0	1	0	X	X	X	0
6	1	1	0	0	X	X	X	0
7	1	1	1	1	X	X	X	1

Design amounts to putting minterms in the appropriate address location

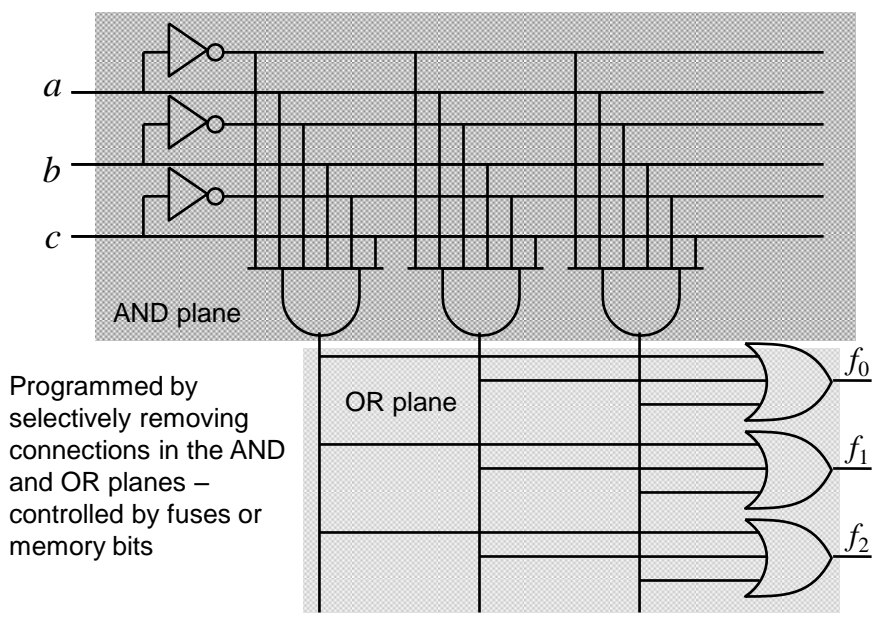No logic simplification required

Useful if multiple Boolean functions are to be implemented, e.g., in this case we can easily do up to 4, i.e., 1 for each output line

Reasonably efficient if lots of minterms need to be generated

ROM Implementation

- Can be quite inefficient, i.e., become large in size with only a few non-zero entries, if the number of minterms in the function to be implemented is quite small

- Devices which can overcome these problems are known as programmable array logic (PAL)

- In PALs, only the required minterms are generated using a separate AND plane. The outputs from this plane are ORed together in a separate OR plane to produce the final output

Basic PAL Structure

a

b

c

AND plane

Programmed by selectively removing connections in the AND and OR planes – controlled by fuses or memory bits

OR plane

f_0

f_1

f_2

Other Memory Devices

- Non-volatile storage is offered by ROMs (and some other memory technologies, e.g., FLASH), i.e., the data remains intact, even when the power supply is removed
- Volatile storage is offered by Static Random Access Memory (SRAM) technology
  - Data can be written into and read out of the SRAM, but is lost once power is removed

Memory Application

- Memory devices are often used in computer systems
- The central processing unit (CPU) often makes use of busses (a bunch of wires in parallel) to access external memory devices
- The *address bus* is used to specify the memory location that is being read or written and the data bus conveys the data too and from that location
- So, more than one memory device will often be connected to the same data bus

Bus Contention

- In this case, if the output from the data pin of one memory was a 0 and the output from the corresponding data pin of another memory was a 1, the data on that line of the data bus would be invalid
- So, how do we arrange for the data from multiple memories to be connected to the some bus wires?

Bus Contention

- The answer is:
  - *Tristate* buffers (or drivers)
  - Control signals
- A tristate buffer is used on the data output of the memory devices
  - In contrast to a normal buffer which is either 1 or 0 at its output, a tristate buffer can be electrically disconnected from the bus wire, i.e., it will have no effect on any other data currently on the bus – known as the *'high impedance'* condition

Tristate Buffer

Symbol

Bus line

Output Enable
(OE) = 1

OE = 0

Functional analogy

Bus line

OE = 1

OE = 0
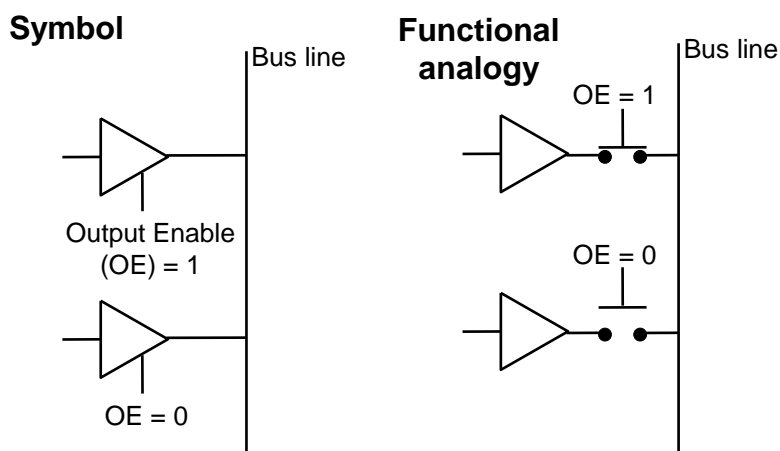
Control Signals

- We have already seen that the memory devices have an additional control input (OE) that determines whether the output buffers are enabled.
- Other control inputs are also provided:
  - Write enable (WE). Determines whether data is written or read (clearly not needed on a ROM)
  - Chip select (CS) – determines if the chip is activated
- Note that these signals can be active low, depending upon the particular device