

Digital Electronics

Definition

Abstraction is the hiding of data/complexity when they are not important, only expose the things needed to implement a particular task.

Layers of Abstraction

Layer	Examples
Application software	Programs
Operating systems	Device drivers
Architecture	Instructions, registers
Microarchitecture	The way the processor implements the architecture
Logic elements	Adders, memories
Digital circuits	Gates and stuff
Devices	Transistors
Physics	Electrons, quantum mechanics, Maxwell's equations

This course starts at the **devices** layer.

To design something on the stack, you need to know the layer below to implement it, and the layer above to see how the thing you design is used.

Combinatory Logic

Definition

Combinatory output depends only on current input.

- It has no memory.
- The same input always gives the same output, so is entirely predictable.

Boole algebra is used for simplifying logic so less gates are used, so is cheaper.

Logic Gates

Definition

Logical variables can only take on two values.

In electronic circuits, they are represented by

- High voltage for 1
- Low voltage for 0

Using only 2 voltages allows circuits to have greater immunity to signal corruption caused by interference.

Definitions

- **Logical circuits** have one or more inputs.
- Basic logic circuits are known as **gates**.

Logic gates are represented in **symbol**, **truth tables** and **algebra**.

- NOT \bar{a}

- The triangle means **buffer**.
- The circle means **complement**.
- AND $a \cdot b$.
 - A lot of these gates can be extended to have more than 2 inputs.
- OR $a + b$
- XOR $a \oplus b$
- NAND $\overline{a \cdot b}$
- NOR $\overline{a + b}$

Boolean Algebra Laws

AND takes precedence over OR.

OR	AND
$a + 0 = a$	$a \cdot 0 = 0$
$a + a = a$	$a \cdot a = a$
$a + 1 = 1$	$a \cdot 1 = a$
$a + \bar{a} = 1$	$a \cdot \bar{a} = 0$

Commutation

$$a + b = b + a$$

$$a \cdot b = b \cdot a$$

Association

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

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Distribution

$$a \cdot (b + c + \dots) = a \cdot b + a \cdot c + \dots$$

$$a + (b \cdot c \cdot \dots) = (a + b) \cdot (a + c) \cdot \dots$$

Absorption

$$a + (a \cdot b) = a$$

$$a \cdot (a + b) = a$$

Consensus Theorem

$$a \cdot b + \bar{a} \cdot c + b \cdot c = a \cdot b + \bar{a} \cdot c$$

$$(a + b) \cdot (\bar{a} + c) \cdot (b + c) = (a + b) \cdot (\bar{a} + c)$$

Technique

1. Expand each term until it includes one instance of each variable.
2. Then simplify the terms by cancelling terms.

$$x \cdot y \rightarrow x \cdot y \cdot \bar{z} + x \cdot y \cdot z \text{ to include } z.$$

DeMorgan's Theorem

$$\overline{a + b + c + \dots} = \bar{a} \cdot \bar{b} \cdot \bar{c} \cdot \dots$$

$$\overline{a \cdot b \cdot c \cdot \dots} = \bar{a} + \bar{b} + \bar{c} + \dots$$

Proof

First prove for 2 variables by truth table.

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

Extend to more variables by induction:

$$\begin{aligned}\overline{(a + b) + c} &= \overline{a + b} \cdot \overline{c} \\ &= \overline{a} \cdot \overline{b} \cdot \overline{c}\end{aligned}$$

Sometimes we wish to use NAND or NOR gates since they are usually simpler and faster. We can use DeMorgan's law to replace AND and OR to NAND and NOR.

Technique: Bubble Logic

- 2 consecutive bubble operations cancel.
- Bubbles change an AND into an OR when they pass through a gate (vice versa).

Logic Minification**Normal Forms**

A **minterm** contains all input variables of boolean function f so f and itself both evaluate to 1.

$x y z$	f	minterm
0 0 0	1	$\overline{x} \cdot \overline{y} \cdot \overline{z}$
0 0 1	0	
0 1 0	0	
0 1 1	1	$\overline{x} \cdot y \cdot z$
...

Definition

A boolean function f can be written in **disjunctive normal form** by expressing it as the *disjunction* (OR) of its minterms. (Sum of products)

$$f = \overline{x} \cdot \overline{y} \cdot \overline{z} + \overline{x} \cdot y \cdot \overline{z} + \dots$$

As an inverse to minterms, **maxterm** is all the input variables where f evaluates to 0.

$x y z$	f	maxterm
0 0 0	1	
0 0 1	0	$\overline{x} \cdot \overline{y} \cdot z$
0 1 0	0	$\overline{x} \cdot y \cdot \overline{z}$
0 1 1	1	
...

Definition

Conjunctive normal form is where the function is written as the product of sums.

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

The CNF for f can be found by writing the DNF for \bar{f} (maxterms of f), then use DeMorgan's laws.

$$\bar{f} = \bar{x} \cdot \bar{y} \cdot z + \bar{x} \cdot y \cdot \bar{z} + \dots$$

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

K-Maps

Both DNF and CNF are not simplified, K-maps are useful for simplifying logical expressions of up to 5 variables.

SOP (Sum of Power) Simplification

1. Convert the truth table to K-map by plotting the minterms on the table in **grey code** order, so adjacent cells are logically next to each other.

		yz			
x		00	01	11	10
	0	1	1	1	1
	1			1	

2. Group the minterms, the larger the groups the better.

- Group sizes have to be a power of 2.
- Groups can wrap around edges.

		yz			
x		00	01	11	10
	0	1	1	1	1
	1			1	

So the simplified function is $f = \bar{x} + y \cdot z$

The simplified expression is in SOP form, suitable for implementations using AND and OR gates.

POS Simplification

1. Find a SOP Simplification for \bar{f} .
2. Use DeMorgan's to find a POS simplification for f .

This is suitable for implementing using NOR gates.

Don't Care Conditions

Some combinations will never happen, we can declare those *don't care* conditions.

- They are treated as 0 or 1 depending on which gives the simpler results.
- We write them as **X** in K-map.

Definitions

Keyword	Meaning
Cover	A term covers a minterm if the minterm is part of the term.
Prime implicant	A term that cannot be further combined.
Essential prime implicant	A term that covers a minterm that no other prime implicant covers.
Covering set	A minimum set of prime implicants which covers all minterms.

)

Q-M Method

1. List all minterms and *don't care* terms, group them by the number of 1s.

- Minterms: 4, 5, 6, 8, 9, 10, 13
 - *Don't care* terms: 0, 7, 15
- 0000
0100
1000

0101
0110
1001
1010

1101
0111
1111

2. If two terms differ by 1 digit (e.g. 0000 and 0100), merge them and write it on the next column (e.g. 0-00 where “-” is a wildcard), put a tick next to them. Merge all terms that can be merged, including previously ticked terms.

Repeat for the 2nd column, and write the merge terms to the 3rd column, until no more terms can be merged.

Column 1	Column 2	Column 3
0000 ✓	0-00 -000	01--
0100 ✓		-1-1
1000 ✓	010- ✓ 01-0 ✓	
0101 ✓	100-	
0110 ✓	10-0	
1001 ✓	-101 ✓	
1010 ✓	01-1 ✓	
1101 ✓	011- ✓	
0111 ✓	1-01	
1111 ✓	11-1 ✓ -111 ✓	

3. The unmergable terms (no ticks beside them) are the **prime implicants**. We need to get rid of some of those terms to find the **mincover**.

Create an **implication chart** with the **prime implicants** and the **minterms**. Put an X where the minterm can be expressed as the prime implicant. (e.g. $4_{10} = 0100_2$ can be expressed as $0-00$)

	4	5	6	8	9	10	13
0-00	X						
-000				X			
100-				X	X		
10-0				X		X	
1-01					X		X
01--	X	X	X				
-1-1		X					X

4. Look for **essential prime indicants**: 6 is only covered by 01--, and 10 only by 10-0, they are essential prime indicants. Cross the two row out, also cross out the numbers they cover.

	4	5	6	8	9	10	13
0-00	X						
-000				X			
100-				X	X		
10-0				X		X	
1-01					X		X
01--	X	X	X				
-1-1		X					X

5. Cross out as few rows as possible to cover the remaining (uncovered) numbers.

	4	5	6	8	9	10	13
0-00	X						
-000				X			
100-				X	X		
10-0				X		X	
1-01					X		X
01--	X	X	X				
-1-1		X					X

The prime implicants we picked are 10-0, 1-01 and 01--, giving the simplification.

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

Binary Adders

We are doing the **compositional approach**, where we put half adders and full adders together to build a **ripple carry adder**.

Half Adder

Adds two bits together.

By inspection or simplifying minterms:

$$\text{sum} = a \oplus b$$

$$c_{\text{out}} = a \cdot b$$

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

Full Adder

$$\text{sum} = a \oplus b \oplus c_{\text{in}}$$

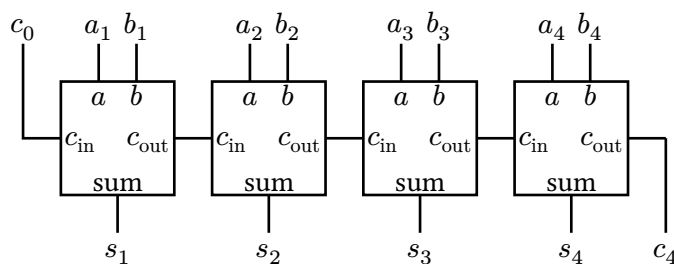
$$c_{\text{out}} = a \cdot b + c_{\text{in}} \cdot (a + b)$$

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

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

Ripple Carry Adder

Cascade multiple of these adders together to make a ripple carry adder.



A 4 bit adder has 4 full adders.

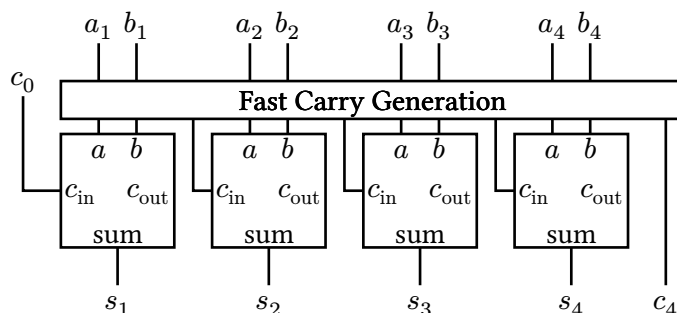
- c_0 propagates from left to right.
- Physically, they have finite propagation delay, so there is a delay between the inputs and outputs.

If we complement a then set c_0 to 1, we have $s = b - a$.

Speeding Things Up

- Design the adder as a single block of 2-level combinational logic with $2n$ inputs and n outputs.
 - Low delay.
 - Truth table is massive, and simplification is complicated. It is very complex to design.

Fast Carry Generation



c_{in}	a	b	c_{out}
?	0	0	0
?	0	1	?
?	1	0	?
?	1	1	1

- If $\overline{a_i} \cdot \overline{b_i} = 1$ then $c_{\text{out}} = 0$
- If $a_i \oplus b_i$ then $c_{\text{out}} = c_{\text{in}}$
- If $a_i \cdot b_i$ then $c_{\text{out}} = 1$

Sometimes, the carry bit may be known before c_{in} is given.

If that is not possible, the main speedup is that carry generation happens in parallel with the adders.

- Lot's of gates are used in carry generation.
- Usually two 4-bit carry generators are used in an 8-bit adder.

Multilevel Logic

The ripple adder is a multilevel logic - as it cascades, it forms many levels.

- It's difficult to buy large logic gates, we can use multilevel logic to expand the input.
- But it can introduce delays and **hazards**.

Reducing Number of Gates

Simplified SOP expressions requires a lot of gates to do.

The logic below uses 9 literals, 7 gates, 2 levels.

$$\begin{aligned} z &= a \cdot d \cdot f + a \cdot e \cdot f + b \cdot d \cdot f + b \cdot e \cdot f + c \cdot d \cdot f + c \cdot e \cdot f + g \\ &= (a \cdot d + a \cdot e + b \cdot d + b \cdot e + c \cdot d) \cdot f + g \\ &= (a + b + c) \cdot (d + e) \cdot f + g \end{aligned}$$

We have recursively factored out common literals and express z in **two-level form**.

$$\begin{aligned} x &= a + b + c \\ y &= d + e \\ z &= x \cdot y \cdot f + g \end{aligned}$$

It uses 9 literals, 4 gates, 3 levels. The more levels the bigger the propagation delay.

Hazards

Definition

Hazards are brief changes in output (when there is a change in input), also called a **glitch**.

Type	Description
Static hazard	Output undergoes momentary transition when one input changes when it is supposed to remain unchanged.
Static 1 hazard	Output should stay at 1, but as signal changes it briefly drops to zero.
Static 0 hazard	The opposite of a static 1 hazard.
Dynamic hazard	Output changes more than once as opposed to just once.

This is due to the difference in **propagation delay** between layers.

Removing Hazards

1. Plot the function on a K-map.
2. If the square it is moved from and to are not in the same group, there is a hazard.
3. Add an extra term containing the two squares to remove the static 1 hazard.

To remove a static 0 hazard, draw a K-map of the complement of the output.