Digital Logic Circuits

Computer Architecture — Module 2 Notes

Comprehensive study notes: logic gates, Boolean algebra, combinational & sequential circuits, and Karnaugh maps.

Table of Contents

1. Logic Gates and Truth Tables

1.1 Basic Logic Gates

Logic gates are the fundamental building blocks of digital circuits. Each gate performs a specific Boolean operation.

Gate	Symbol	Boolean Expr.	Truth Table
AND	· (or &)	Y = A · B	00|0 01|0 10|0 11|1
OR	+	Y = A + B	00|0 01|1 10|1 11|1
NOT	¬ or '	Y = ¬A	0|1 1|0
NAND	↑	Y = ¬(A·B)	00|1 01|1 10|1 11|0
NOR	↓	Y = ¬(A+B)	00|1 01|0 10|0 11|0
XOR	⊕	Y = A⊕B	00|0 01|1 10|1 11|0
XNOR	⊙	Y = ¬(A⊕B)	00|1 01|0 10|0 11|1

Truth-table columns read as A B | Y (inputs on the left of the bar, output on the right).

1.2 Universal Gates

NAND and NOR are Universal Gates — any Boolean function can be implemented using only NAND gates, or only NOR gates.

BUFFER using NAND: Y = NAND(A, A) = ¬(A·A) = ¬A·¬A = A

AND using NAND: Y = NAND(NAND(A,B), NAND(A,B))
= ¬(¬(A·B)·¬(A·B)) = A·B

OR using NAND: Y = NAND(NAND(A,A), NAND(B,B))
= ¬(¬A·¬B) = A+B

2. Boolean Algebra

2.1 Basic Laws and Theorems

Law	AND Form	OR Form
Identity	$A \cdot 1 = A$	$A + 0 = A$
Null	$A \cdot 0 = 0$	$A + 1 = 1$
Idempotent	$A \cdot A = A$	$A + A = A$
Complement	$A \cdot \neg A = 0$	$A + \neg A = 1$
Double Negation	$\neg(\neg A) = A$	—
Commutative	$A \cdot B = B \cdot A$	$A + B = B + A$
Associative	$(A \cdot B) \cdot C = A \cdot (B \cdot C)$	$(A+B)+C = A+(B+C)$
Distributive	$A \cdot (B+C) = A \cdot B + A \cdot C$	$A + B \cdot C = (A+B) \cdot (A+C)$
Absorption	$A \cdot (A+B) = A$	$A + A \cdot B = A$

2.2 De Morgan's Theorems (Important!)

Theorem 1: The complement of a product equals the sum of complements.

$$\neg(A \cdot B) = \neg A + \neg B$$

Theorem 2: The complement of a sum equals the product of complements.

$$\neg(A + B) = \neg A \cdot \neg B$$

Memory Aid: "Break the line, change the sign" — AND becomes OR, OR becomes AND.

Generalized form:

$$\neg(A_1 \cdot A_2 \cdot \ldots \cdot A_n) = \neg A_1 + \neg A_2 + \ldots + \neg A_n$$
$$\neg(A_1 + A_2 + \ldots + A_n) = \neg A_1 \cdot \neg A_2 \cdot \ldots \cdot \neg A_n$$

2.3 Boolean Algebra Example

Simplify: $F = \neg AB + AB + AB\neg C + \neg A\neg B\neg C$

$$\begin{aligned} F &= \neg AB + AB + AB\neg C + \neg A\neg B\neg C \\ &= B(\neg A + A + A\neg C) + \neg A\neg B\neg C \quad \text{[Factor B from first three terms]} \\ &= B(1 + A\neg C) + \neg A\neg B\neg C \quad \text{[}\neg A + A = 1\text{]} \\ &= B + \neg A\neg B\neg C \quad \text{[1 + anything = 1]} \\ &= B + \neg A\neg C \quad \text{[Absorption: } B + \neg A\neg B\neg C = B\text{]} \end{aligned}$$

3. Combinational Circuits

Combinational circuits produce outputs that depend only on current inputs (no memory).

3.1 Half Adder

Adds two single bits and produces a Sum and a Carry.

$$S = A \oplus B$$
$$C = A \cdot B$$

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Circuit: an XOR gate (A, B → S) and an AND gate (A, B → C) sharing the same two inputs.

3.2 Full Adder

Adds three bits: A, B, and the carry-in from the previous stage.

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

A	B	C_in	S	C_out
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

Implementation using two Half Adders:

$$Sum = (A \oplus B) \oplus C_{in}$$
$$C_{out} = (A \cdot B) + (A \oplus B) \cdot C_{in}$$

3.3 Ripple Carry Adder

An N-bit adder built from cascaded Full Adders, where each stage's carry-out feeds the next stage's carry-in.

$$A_3\ B_3 \quad A_2\ B_2 \quad A_1\ B_1 \quad A_0\ B_0$$

```
 ↓  ↓   ↓  ↓   ↓  ↓   ↓  ↓
[FA]---[FA]---[FA]---[FA]
 ↓    ↓    ↓    ↓
C₄   S₃   S₂   S₁   S₀
```

- Advantage: simple design.
- Disadvantage: slow, due to carry propagation delay through each stage.

3.4 Half Subtractor

Subtracts two bits: A − B.

Difference (D) = $A \oplus B$
Borrow (B_out) = $\neg A \cdot B$

A	B	D	B_out
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

3.5 Full Subtractor

Subtracts three bits: A − B − Borrow_in.

$D = A \oplus B \oplus B_in$
$B_out = \neg A \cdot B + \neg A \cdot B_in + B \cdot B_in$

3.6 Decoder

n-to-2ⁿ Decoder — converts n input lines to 2^n output lines.

2-to-4 Decoder — inputs A, B; outputs D_0–D_3:

$D_0 = \neg A \cdot \neg B$
$D_1 = \neg A \cdot B$
$D_2 = A \cdot \neg B$
$D_3 = A \cdot B$

A	B	D_3	D_2	D_1	D_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Applications: memory addressing, demultiplexing, 7-segment displays.

3.7 Encoder

2^n-to-n Encoder — performs the inverse operation of a decoder.

4-to-2 Encoder — inputs I_0–I_3; outputs A, B:

$$A = I_2 + I_3$$
$$B = I_1 + I_3$$

Priority Encoder — handles multiple simultaneously active inputs by prioritizing the highest one.

$$A = I_2 + I_3$$
$$B = I_1 \cdot \neg I_2 + I_3$$
$$V = I_0 + I_1 + I_2 + I_3 \quad \text{(Valid output)}$$

3.8 Multiplexer (MUX)

2^n-to-1 Multiplexer — selects one of 2^n inputs based on n select lines.

2-to-1 MUX:

$$Y = \neg S \cdot I_0 + S \cdot I_1$$

Select Line S	Output Y
0	I_0
1	I_1

4-to-1 MUX:

$$Y = \neg S_1 \cdot \neg S_0 \cdot I_0 + \neg S_1 \cdot S_0 \cdot I_1 + S_1 \cdot \neg S_0 \cdot I_2 + S_1 \cdot S_0 \cdot I_3$$

Applications: data routing, implementing logic functions, bus selection.

3.9 Demultiplexer (DEMUX)

1-to-2^n Demultiplexer — routes one input to one of 2^n outputs based on select lines.

1-to-4 DEMUX:

$$Y_0 = D \cdot \neg S_1 \cdot \neg S_0$$
$$Y_1 = D \cdot \neg S_1 \cdot S_0$$
$$Y_2 = D \cdot S_1 \cdot \neg S_0$$
$$Y_3 = D \cdot S_1 \cdot S_0$$

4. Circuit Simplification Using Karnaugh Maps

4.1 Introduction to K-Maps

Karnaugh Maps provide a graphical method for simplifying Boolean expressions.

- 2 variables → 4 cells (2×2 grid)
- 3 variables → 8 cells (2×4 grid)
- 4 variables → 16 cells (4×4 grid)

K-Map Rules:

- Gray-code ordering — adjacent cells differ by exactly one bit.
- Cells on the edges are adjacent.
- Corners are adjacent (wraparound).

4.2 2-Variable K-Map

```
     B
    0   1
  +----+----+
A0 | m0 | m1 |
  +----+----+
A1 | m2 | m3 |
  +----+----+

m0 = ¬A·¬B     m1 = ¬A·B
m2 = A·¬B      m3 = A·B
```

4.3 3-Variable K-Map

```
       BC
     00   01   11   10
    +----+----+----+----+
A  0  | m0 | m1 | m3 | m2 |
    +----+----+----+----+
   1  | m4 | m5 | m7 | m6 |
    +----+----+----+----+

m0=¬A¬B¬C   m1=¬A¬BC   m3=¬ABC   m2=¬AB¬C
m4=A¬B¬C    m5=A¬BC    m7=ABC    m6=AB¬C
```

4.4 4-Variable K-Map

```
       CD
     00   01   11   10
    +----+----+----+----+
AB 00 | m0 | m1 | m3 | m2 |
    +----+----+----+----+
```

```
01 | m4 | m5 | m7 | m6 |
  +----+----+----+----+
11 | m12| m13| m15| m14|
  +----+----+----+----+
10 | m8 | m9 | m11| m10|
  +----+----+----+----+
```

4.5 Grouping Rules

- Group sizes must be a power of 2: 1, 2, 4, 8, 16 cells.
- Larger groups produce simpler expressions.
- Groups must be rectangular.
- Groups can overlap.
- Wraparound is allowed (edges and corners).

4.6 Simplification Process

- Step 1 — Fill the K-Map with 1s from the truth table.
- Step 2 — Group the 1s in powers of 2.
- Step 3 — Write the product term for each group.
- Step 4 — OR all group expressions together.

Example — Simplify: F(A,B,C) = Σm(0,1,3,5,7)

```
       BC
    00  01  11  10
   +---+---+---+---+
A  0 | 1 | 1 | 1 | 0 |
   +---+---+---+---+
   1 | 0 | 1 | 1 | 0 |
   +---+---+---+---+

Group 1: m0, m1, m3 (wraps) → ¬A
Group 2: m1, m5          → ¬B·C
Group 3: m3, m7          → B·C

F = ¬A + ¬B·C + B·C = ¬A + C
```

4.7 Don't Care Conditions

Use "X" for don't-care inputs — they may be treated as either 0 or 1 to optimize the grouping.

Rules for Don't Cares:

- Treat X as 1 to form larger groups when it helps.
- Treat X as 0 when it is not useful.
- Use all Xs optimally to minimize the final expression.

Example: F(A,B,C) = Σm(0,1,5) + d(3,7)

F = ¬B·¬C + A·C (using the don't-care values to combine terms)

5. Sequential Circuits

Sequential circuits have memory — outputs depend on current inputs AND the previous state.

5.1 Basic Flip-Flops

Flip-Flop	Behavior	Characteristic Equation
SR	S=0,R=0: no change · S=0,R=1: Q=0 · S=1,R=0: Q=1 · S=1,R=1: invalid	Q(t+1) = S + ¬R·Q(t)
D	D=0 → Q(t+1)=0 · D=1 → Q(t+1)=1	Q(t+1) = D
JK	J=0,K=0: no change · J=0,K=1: Q=0 · J=1,K=0: Q=1 · J=1,K=1: toggle	Q(t+1) = J·¬Q + ¬K·Q
T	T=0: no change · T=1: toggle	Q(t+1) = T⊕Q

5.2 SR Flip-Flop (NAND Implementation)

Built from two cross-coupled NAND gates: inputs S and R feed one NAND gate each, and each gate's output feeds back as an input to the other, producing outputs Q and Q′.

S	R	Q(t+1)	Status
0	0	Q(t)	Hold / No change
0	1	0	Reset
1	0	1	Set
1	1	Invalid	Race condition

5.3 Clocked Flip-Flops

Edge-Triggered: respond only on a clock edge.

- Positive edge — rising edge (0→1).
- Negative edge — falling edge (1→0).

Level-Triggered: respond throughout the time the clock is high or low.

Master-Slave Flip-Flop:

- Master stage — enabled when the clock is high.
- Slave stage — enabled when the clock is low.

This master/slave arrangement eliminates race conditions.

5.4 Flip-Flop Conversions

SR to D Flip-Flop:

```
D = S
¬D = R
```

SR to JK Flip-Flop:

$$S = J \cdot \neg Q$$
$$R = K \cdot Q$$

D to T Flip-Flop:

$$D = T \oplus Q$$

6. Digital Components

6.1 Registers

A register is a group of flip-flops used for storing n-bit data.

Basic Register:

- D flip-flops arranged in parallel.
- Driven by a common clock signal.
- Stores n bits simultaneously.

Shift Register: data shifts on each clock pulse. Types: SISO, SIPO, PISO, PIPO.

```
4-bit Shift Register:
D₀ → FF₀ → FF₁ → FF₂ → FF₃
     ↓    ↓    ↓    ↓
    Q₀   Q₁   Q₂   Q₃
```

6.2 Memory Units

RAM (Random Access Memory):

- SRAM — Static, faster, more expensive.
- DRAM — Dynamic, needs periodic refreshing, cheaper.

ROM (Read Only Memory):

- MROM — Mask-programmed.
- PROM — Programmable once.
- EPROM — Erasable using UV light.
- EEPROM — Electrically erasable.

6.3 Counters

Ripple Counter (Asynchronous): flip-flops are clocked sequentially — simple but slow.

```
4-bit Ripple Counter:
CLK → FF₀ → FF₁ → FF₂ → FF₃
      ↓    ↓    ↓    ↓
     Q₀   Q₁   Q₂   Q₃
```

Synchronous Counter: all flip-flops share a common clock — faster and more reliable.

6.4 Adders

Ripple Carry Adder: simple but slow; delay is proportional to n bits, due to carry propagation.

Carry Lookahead Adder: calculates carries in parallel — faster, but more complex. Uses Generate (G) and Propagate (P) signals.

```
G = A·B
P = A⊕B
```

$C_{i+1} = G_i + P_i \cdot C_i$

6.5 Comparators

Magnitude Comparator: compares two n-bit numbers, A and B.

Outputs: $A>B$, $A=B$, $A<B$

7. Important Formulas and Equations

7.1 Boolean Algebra Summary

Law	AND Form	OR Form
Identity	A·1 = A	A+0 = A
Null	A·0 = 0	A+1 = 1
Idempotent	A·A = A	A+A = A
Complement	A·¬A = 0	A+¬A = 1
Double Negation	¬(¬A) = A	—
Commutative	A·B = B·A	A+B = B+A
Associative	(A·B)·C = A·(B·C)	(A+B)+C = A+(B+C)
Distributive	A·(B+C) = A·B + A·C	A+B·C = (A+B)·(A+C)
Absorption	A·(A+B) = A	A+A·B = A

7.2 De Morgan's Laws

¬(A·B) = ¬A + ¬B
¬(A+B) = ¬A · ¬B

7.3 Flip-Flop Characteristic Equations

SR: Q(t+1) = S + ¬R·Q(t)
D: Q(t+1) = D
JK: Q(t+1) = J·¬Q + ¬K·Q
T: Q(t+1) = T⊕Q

7.4 Adder Formulas

Half Adder: S = A⊕B, C = A·B

Full Adder: S = A⊕B⊕Cin
Cout = A·B + A·Cin + B·Cin

8. Practice Problems

Problem 1 — Simplify using Boolean Algebra

Simplify: F = ¬A·B·C + ¬A·B·¬C + ¬A·¬B·C

```
F = ¬A·B·(C + ¬C) + ¬A·¬B·C
  = ¬A·B + ¬A·¬B·C
  = ¬A·(B + ¬B·C)
  = ¬A·(B + C)
```

Problem 2 — K-Map Simplification

Simplify: F(A,B,C) = Σm(0,2,3,6,7)

```
        BC
     00   01   11   10
    +----+----+----+----+
A   0  | 1  | 0  | 1  | 1  |
    +----+----+----+----+
    1  | 0  | 0  | 1  | 1  |
    +----+----+----+----+

Group 1: m0, m2 → ¬A·¬C
Group 2: m3, m7 → B·C
Group 3: m6, m7 → A·B

F = ¬A·¬C + B·C + A·B
```

Problem 3 — Design a 3-to-8 Decoder

Inputs: A, B, C Outputs: D_0 to D_7

```
D₀ = ¬A·¬B·¬C     D₁ = ¬A·¬B·C
D₂ = ¬A·B·¬C      D₃ = ¬A·B·C
D₄ = A·¬B·¬C      D₅ = A·¬B·C
D₆ = A·B·¬C       D₇ = A·B·C
```

Problem 4 — Full Adder Implementation

Design a Full Adder using two Half Adders and an OR gate:

```
HA1: S1 = A⊕B,  C1 = A·B
HA2: S  = S1⊕Cin,  C2 = S1·Cin
Cout = C1 + C2 = A·B + (A⊕B)·Cin
```

Problem 5 — Boolean Expression Implementation

Implement F = ¬A·B + A·¬B using universal (NAND) gates:

Step 1: F = A⊕B
Step 2: XOR using NAND gates
F = NAND(NAND(A, NAND(A,B)), NAND(B, NAND(A,B)))

9. Quick Reference Card

Gate Functions

Gate	Boolean	Output is 1 When...
AND	A·B	Only when A = B = 1
OR	A+B	Anytime except A = B = 0
NOT	¬A	Inverts the input
NAND	¬(A·B)	Anytime except A = B = 1
NOR	¬(A+B)	Only when A = B = 0
XOR	A⊕B	When A ≠ B

K-Map Group Sizes

Group Size	Variables Eliminated
2 cells	1 variable
4 cells	2 variables
8 cells	3 variables
16 cells	4 variables

Sequential vs. Combinational

Feature	Combinational	Sequential
Output depends on	Current inputs only	Current inputs + previous state
Has memory	No	Yes
Contains	Gates only	Gates + Flip-flops
Example	Adder, MUX, Decoder	Counter, Register

10. Key Takeaways

✔️ Logic gates are the building blocks of all digital circuits.
✔️ Boolean algebra provides the mathematical foundation for circuit design.
✔️ Karnaugh maps are the most common manual simplification method.
✔️ Combinational circuits have no memory (adders, multiplexers, decoders).
✔️ Sequential circuits have memory (flip-flops, registers, counters).
✔️ Universal gates (NAND, NOR) can implement any logic function.
✔️ De Morgan's laws are essential for circuit simplification.
✔️ Flip-flops store single bits; registers store multiple bits.
✔️ Don't-care conditions help optimize circuit design.

These comprehensive notes cover Module 2 of Computer Architecture. Focus on understanding the fundamental concepts and practicing K-map simplification, Boolean algebra, and circuit design problems.