Digital Electronics — ECE Master Cheat Sheet

Absolute Basics → Boolean Algebra → K-Maps → Combinational → Sequential → Timing → Memory → CMOS / VLSI → RTL bridge · OA & Interview traps

PRIORITY MUST must know · HIGH high probability · GOOD good to know · BONUS bonus // NOTATION A·B = AND · A+B = OR · A⊕B = XOR (^ in code) · $\overline{A}$ = A' = NOT A // SCOPE GEN general logic · CMOS · FPGA · ASIC technology-specific

// §1 NUMBER SYSTEMS — POSITIONAL NOTATION

1.1 What a base actually means MUST

Rule: in base *r*, the digit at position *i* (counting right from 0) is worth `digit × r^i`. Base *r* uses digits `0 … r-1`. That one sentence generates every conversion below.

```
  1  0  1  1  0  1   (base 2)
  |  |  |  |  |  +-- 2^0 = 1  x1 =  1
  |  |  |  |  +----- 2^1 = 2  x0 =  0
  |  |  |  +-------- 2^2 = 4  x1 =  4
  |  |  +----------- 2^3 = 8  x1 =  8
  |  +-------------- 2^4 =16  x0 =  0
  +----------------- 2^5 =32  x1 = 32
                                 ----
  101101(2) = 32+8+4+1             = 45(10)
```

Base	r	Digits	Written as	Why it exists
Decimal	10	0–9	45, 45d	human counting
Binary	2	0,1	0b101101	two stable transistor states
Octal	8	0–7	055, 0o55	1 digit = exactly 3 bits
Hex	16	0–9, A–F	0x2D	1 digit = exactly 4 bits

Powers of 2 — memorise cold:
`1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768 65536` = $2^0 \ldots 2^{16}$. Then 2^{20}=1 M (1 048 576), 2^{24}=16 M, 2^{30}=1 G, 2^{32}=4 294 967 296.

1.2 The four conversions you will be asked MUST

Bin → Dec	Add the place values of every 1. `1101 = 8+4+1 = 13`.
Dec → Bin	**Divide by 2, keep remainders, read them BOTTOM-UP.**
Bin ↔ Hex	Group bits in **4** from the **binary point**, pad the outside with 0.
Bin ↔ Oct	Same, but group in **3**.

```
DEC -> BIN : 45                 FRACTION: 0.6875 -> binary
  45 / 2 = 22 r 1  (LSB)          0.6875 x 2 = 1.375  -> 1 (MSB)
  22 / 2 = 11 r 0                 0.375  x 2 = 0.75   -> 0
  11 / 2 =  5 r 1                 0.75   x 2 = 1.5    -> 1
   5 / 2 =  2 r 1                 0.5    x 2 = 1.0    -> 1  stop
   2 / 2 =  1 r 0               0.6875(10) = 0.1011(2)
   1 / 2 =  0 r 1  (MSB)        integer : remainders BOTTOM-up
  read up -> 101101(2) = 45     fraction: carries    TOP-down
```

```
BIN <-> HEX  (group 4)        BIN <-> OCT  (group 3)
  1011 0101      = B5             10 110 101      = 265
  |    |                         |  |   |
  B    5   -> 0xB5               2  6   5  -> 0o265
Check: 0xB5 = 11x16+5 = 181 = 0o265 = 2x64+6x8+5 = 181  OK
Hex -> Oct is NOT direct: go via binary.
```

Trap — where the padding goes. Group **outward from the binary point**, never from the left of a fraction. `10.11` in hex is `0010.1100` = `0x2.C`, **not** `0x2.3`. Padding an integer on the left with 0 is free; padding a fraction on the *right* with 0 is free; the other way round changes the value.

1.3 Hex ↔ binary lookup HIGH

```
0=0000  1=0001  2=0010  3=0011  4=0100  5=0101  6=0110  7=0111
8=1000  9=1001  A=1010  B=1011  C=1100  D=1101  E=1110  F=1111
```

// §2 SIGNED NUMBERS & TWO'S COMPLEMENT

2.1 The four representations MUST

Scheme	MSB is	Range (N bits)	Zeros	Verdict
Unsigned	a value bit	0 … 2^N-1	1	counters, addresses
Sign-magnitude	sign only	$-(2^{N-1}-1)$ … $+(2^{N-1}-1)$	2	needs a real subtractor
1's complement	sign	$-(2^{N-1}-1)$ … $+(2^{N-1}-1)$	2	end-around carry; obsolete
2's complement	weight -2^{N-1}	**-2^{N-1} … $+2^{N-1}-1$**	**1**	**universal in hardware**

Why 2's complement won: exactly one representation of zero, and **one adder performs both add and subtract** — no sign comparison, no correction step. That is the entire reason every ALU uses it.

2.2 Negating in two's complement MUST

```
METHOD 1 (always works): invert every bit, then add 1
   +5 = 0000 0101
 invert 1111 1010
  add 1 1111 1011  = -5

METHOD 2 (faster by hand): copy from the right up to AND
 including the first 1, then invert everything to the left
   +12 = 0000 1100
              ^^^ copy "100", invert everything left of it
   -12 = 1111 0100

METHOD 3 (reading a negative): MSB carries weight -2^(N-1)
   1011 0110 = -128 +32+16+4+2 = -74
```

4-bit	Unsgn	2's c	4-bit	Unsgn	2's c
0000	0	0	1000	8	**-8**
0001	1	+1	1001	9	-7
0100	4	+4	1110	14	-2
0111	7	**+7**	1111	15	-1

Trap — the asymmetric end. `1000` (4-bit) is **-8** and has **no positive partner**: negating it returns `1000`. So `-(-2^(N-1))` overflows. And `1111…1` is **-1**, not the largest value.

2.3 Extending and truncating HIGH

Zero extension	Pad the left with **0**. Correct for **unsigned** only.
Sign extension	Pad the left by **replicating the MSB**. Correct for 2's complement: `1011` (-5, 4-bit) → `1111 1011` (-5, 8-bit).
Truncation	Dropping high bits is safe only if every dropped bit already equals the new MSB.

// §3 BINARY ARITHMETIC — CARRY VS OVERFLOW

3.1 Addition and subtraction MUST

```
ADD: 0+0=0  0+1=1  1+0=1  1+1=0 carry 1  1+1+1=1 carry 1
SUB: 0-0=0  1-0=1  1-1=0  0-1=1 borrow 1

  0110 1011  (107)     A - B  ==  A + (2's complement of B)
+ 0010 1101  ( 45)     so ONE adder does both: feed it
-----------            B^sub  with  Cin = sub
  1001 1000  (152)
```

3.2 CARRY-OUT is not OVERFLOW MUST

Carry-out (C) = the bit that falls off the MSB. It says an **UNSIGNED** result did not fit. **Overflow (V)** = the **SIGNED** result did not fit. They are computed differently and are independent — the same addition can raise either, both, or neither.

```
Two ways to detect SIGNED overflow (they agree exactly):
  (1) V=1 when both operands share a sign AND the result
      sign differs.    (+)+(+)=(-)   or   (-)+(-)=(+)
  (2) V = Cin(msb) ^ Cout(msb)      <- what silicon uses

4-bit examples                Cout  V | unsigned | signed
  0111 + 0001 = 1000           0    1 |  7+1=8 OK| 7+1=-8 BAD
  1111 + 0001 = 0000           1    0 | 15+1=0 X | -1+1=0 OK
  1000 + 1000 = 0000           1    1 |  8+8=0 X | -8-8=0 BAD
  0011 + 0010 = 0101           0    0 |  3+2=5 OK| 3+2=5  OK
```

Trap: "carry set means the answer is wrong" is false. After a **signed** add you read **V**; after an **unsigned** add you read **C**. Adding two numbers of *opposite* sign can **never** overflow, whatever the carry does.

// §4 BINARY CODES

4.1 The code table you should be able to rebuild HIGH

Dec	BCD 8421	Excess-3	Gray	Dec	BCD 8421	Excess-3	Gray
0	0000	0011	0000	5	0101	1000	0111
1	0001	0100	0001	6	0110	1001	0101
2	0010	0101	0011	7	0111	1010	0100
3	0011	0110	0010	8	1000	1011	1100
4	0100	0111	0110	9	1001	1100	1101

4.2 BCD (8421) HIGH

What: each *decimal digit* gets its own 4-bit group. `59` → `0101 1001`. **Not** the same as binary 59 (`0011 1011`).

Trap: `1010` … `1111` are **invalid** BCD. A BCD adder must correct a nibble by **adding 6 (0110)** whenever the nibble result exceeds 9 *or* produced a carry out — that correction step is the classic exam question.

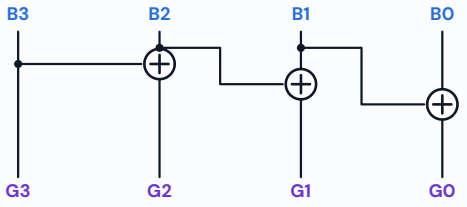
```
BCD ADD:  0111 (7)      0101 (5)          0111 (7)
        + 0110 (6)    + 0011 (3)        + 0110 (6)
        ---------     ---------         --------
          1101 >9       1000  fine        1101 -> +0110
        + 0110 fix                        ---------
        ---------                         1 0011 = 13
        1 0011 = 13
```

4.3 Excess-3 GOOD

Rule: Excess-3 = BCD + 3. It is **self-complementing**: bitwise-inverting the code gives the **9's complement** of the digit (3 → `0110` , invert → `1001` = 6 = 9−3). It is also unweighted, so it has no all-zero code — handy for detecting a dead line.

4.4 Gray code — only one bit changes MUST

```
BINARY -> GRAY :  G = B ^ (B >> 1)
  G[MSB] = B[MSB]
  G[i]   = B[i+1] ^ B[i]        (independent bits: fast)

GRAY -> BINARY :  running XOR from the MSB down
  B[MSB] = G[MSB]
  B[i]   = B[i+1] ^ G[i]        (serial: uses the new B[i+1])

  3-bit sequence: 000 001 011 010 110 111 101 100 (then wraps)
```



B = 1011 → G = 1110 only ONE bit changes between adjacent codes

Why it matters `GEN` **:** successive Gray values differ in **exactly one bit**, so a value sampled mid-transition is either the old or the new count — **never a wild intermediate**. That is why shaft encoders use it, why K-map rows are Gray-ordered, and why **async-FIFO pointers are Gray-coded** before crossing a clock domain (§24).

Trap: Gray code is **not weighted** — you cannot add Gray numbers or read place values off them. Convert to binary first. Also, the wrap from the last code back to `000` is single-bit only when the count length is a **power of two**.

4.5 Parity & error detection GOOD

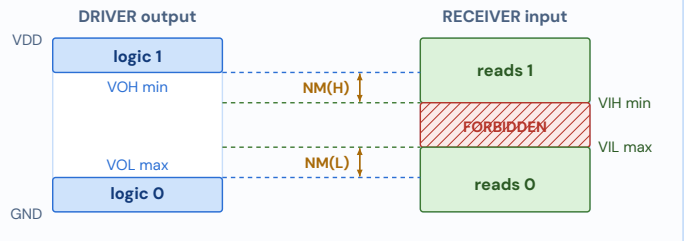
Even parity	Parity bit chosen so the **total** number of 1s (data + parity) is even. `P = XOR of all data bits.`
Odd parity	Total number of 1s is odd. `P = XNOR of all data bits.`
Power	Detects **any odd** number of bit errors (so all single errors). Cannot detect a double error and **cannot correct anything**.
Hamming	Adds *k* check bits over *m* data bits with `2^k ≥ m+k+1`; corrects one error (SEC), or SEC-DED with one extra bit.

// §5 DIGITAL SIGNALS & LOGIC LEVELS

5.1 Analog vs digital GOOD

Analog	Digital
continuous in value and time; infinite resolution in principle	two valid bands only; anything between is undefined
noise accumulates at every stage and is unrecoverable	noise is **rejected** at each gate while it stays under the noise margin
copies degrade	copies are exact; logic is regenerative

5.2 Logic levels and noise margin MUST



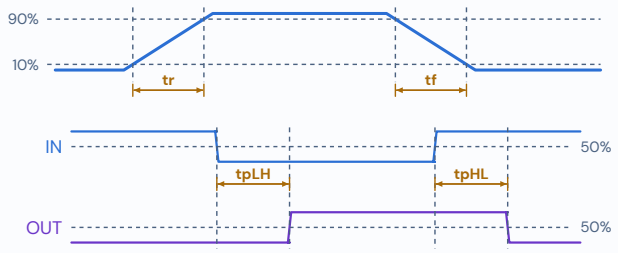
[FORMULA] Noise margins
`NM(H) = VOH(min) - VIH(min)` `NM(L) = VIL(max) - VOL(max)`
Both must be **positive** for the link to work. The usable noise margin is `min(NM_H, NM_L)` .

Family (5 V)	VOH min	VOL max	VIH min	VIL max	NM(H)	NM(L)
TTL 74LS `GEN`	2.7 V	0.5 V	2.0 V	0.8 V	0.7 V	0.3 V
CMOS 74HC `CMOS`	4.9 V	0.1 V	3.5 V	1.5 V	1.4 V	1.4 V

Typical datasheet values at rated load. CMOS input thresholds track the supply (≈0.7·VDD and ≈0.3·VDD), which is why CMOS noise immunity scales with VDD while TTL's does not.

Trap: a floating CMOS input is **not** a logic 0 — it drifts through the forbidden band, turns both transistors partly on and burns current. Always tie unused inputs to a rail through a pull-up/pull-down. TTL inputs float *high*, which is a different failure, not a safer one.

5.3 Timing of a real edge MUST



Propagation delay	`tpd` — input crossing 50% to output crossing 50%. Datasheets quote `tpLH`, `tpHL` separately; they are rarely equal, and that inequality is what creates **hazards** (§15).
Rise / fall time	10%→90% and 90%→10% of the swing. Set by the load capacitance and the drive strength, so they grow with fan-out.
Duty cycle	`D = t(high) / T × 100 %`. Matters for anything clocked on **both** edges, and for ripple counters used as dividers.

Divide-by-2 fact: a toggle flip-flop clocked by *any* duty cycle produces a **50 % duty-cycle** output at half the frequency, because it changes only on one clock edge. This is the standard way to clean up a lopsided clock.

// §6 LOGIC GATES — THE MASTER TABLE

6.1 All seven gates at once MUST

Gate	Expression	Output is 1…	Output is 0…	One-line identity
AND	`Y = A·B`	only when **all** inputs `1`	if **any** input `0`	`0` is the "killer" input
OR	`Y = A+B`	if **any** input `1`	only when **all** inputs `0`	`1` is the "killer" input
NOT	`Y = A'`	when A = `0`	when A = `1`	inverter / buffer with bubble
NAND	`Y = (A·B)'`	if **any** input `0`	**only when all inputs** `1`	**universal**; AND + bubble
NOR	`Y = (A+B)'`	**only when all inputs** `0`	if **any** input `1`	**universal**; OR + bubble
XOR	`Y = A⊕B`	when inputs **differ**	when inputs are equal	inequality / odd-1s detector
XNOR	`Y = (A⊕B)'`	when inputs are **equal**	when inputs differ	equality comparator

a bubble on the output = inversion · NAND and NOR are the two universal gates

```
A B | AND  OR  NAND  NOR  XOR  XNOR
----+-----------------------------
0 0 |  0   0    1    1    0    1
0 1 |  0   1    1    0    1    0
1 0 |  0   1    1    0    1    0
1 1 |  1   1    0    0    0    1
```

n-input rules `GEN` **:** an n-input **XOR** outputs 1 when an **odd** number of inputs are 1 — it is an **odd-parity generator**. Its complement is 1 for an **even** number of 1s. A 2-input XNOR is the 1-bit **equality** comparator.

Trap — XOR is not OR. They differ on exactly one row: `1,1` . OR gives 1, XOR gives 0. Half the "find the output" mistakes in an OA live on that single row.

6.2 Useful XOR identities HIGH

```
A ^ 0 = A          // XOR with 0 is a PASS-THROUGH
A ^ 1 = A'         // XOR with 1 is a CONTROLLED INVERTER
A ^ A = 0          A ^ A' = 1
A ^ B ^ A = B      // XOR is its own inverse -> used in swap, LFSR
A ^ B = A'B + AB'  (A ^ B)' = AB + A'B'   // XNOR
A ^ B ^ C = 1 when an ODD number of A,B,C are 1
```

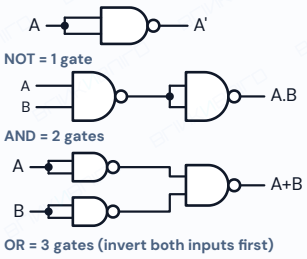
The controlled-inverter trick is why one adder can subtract: feed `B ^ sub` into the adder and set `Cin = sub`. `sub=0` → adds B; `sub=1` → adds B' + 1 = −B.

// §7 UNIVERSAL GATES

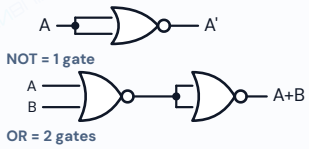
7.1 Why "universal" MUST

A gate is **universal** if any Boolean function can be built from it alone. **NAND and NOR are universal.** AND, OR and NOT individually are **not** — {AND, NOT} or {OR, NOT} together are, but no single one of them is.

FROM NAND ONLY



FROM NOR ONLY



The mirror rule
NAND: NOT 1 AND 2 OR 3 XOR 4
NOR : NOT 1 OR 2 AND 3 XNOR 4

Target	NAND gates	NOR gates	Construction
NOT	1	1	tie both inputs together
AND	2	3	NAND: invert the NAND output
OR	3	2	NAND: invert both inputs first
XOR	4	5	`Y = ((A·(AB)')·(B·(AB)'))'`
XNOR	5	4	dual of the above

The bubble rule (De Morgan drawn) MUST: an **AND with a bubbled output** ≡ an **OR with bubbled inputs**. So NAND = OR-of-inverted-inputs, and NOR = AND-of-inverted-inputs. Pushing bubbles through gates like this is how you convert an AND-OR network into an all-NAND network **with no change in function**: a two-level SOP maps to two levels of NAND, one for one.

Trap: "NAND is universal, so a NAND is always cheaper." In `CMOS` a NAND *is* the preferred primitive (§32), but building an OR out of NANDs costs 3 gates and **two** levels of delay. Universality is about possibility, not optimality.

// §8 BOOLEAN ALGEBRA — FORMULA SHEET

8.1 The laws MUST

Law	OR form	AND form (dual)
Identity	`A + 0 = A`	`A · 1 = A`
Null / annulment	`A + 1 = 1`	`A · 0 = 0`
Idempotent	`A + A = A`	`A · A = A`
Complement	`A + A' = 1`	`A · A' = 0`
Involution	`(A')' = A`	
Commutative	`A + B = B + A`	`A·B = B·A`
Associative	`A+(B+C) = (A+B)+C`	`A(BC) = (AB)C`
Distributive	`A + BC = (A+B)(A+C)`	`A(B+C) = AB + AC`
Absorption	`A + AB = A`	`A(A+B) = A`
Redundancy	`A + A'B = A + B`	`A(A'+B) = AB`
De Morgan	`(A+B)' = A'·B'`	`(A·B)' = A' + B'`
Consensus	`AB + A'C + BC = AB + A'C`	`(A+B)(A'+C)(B+C) = (A+B)(A'+C)`

Trap — the distributive law you forget. `A + BC = (A+B)(A+C)` has **no counterpart in ordinary algebra** (there, `a + bc ≠ (a+b)(a+c)`). Boolean algebra is distributive **both ways**. Expect it in a simplification question.

Duality principle: swap every `·` ↔ `+` and every `0` ↔ `1`, leave variables alone, and a true identity stays true. That is why the table above has two columns — you only have to memorise one.
De Morgan is not duality: duality does not complement the variables, De Morgan does.

8.2 Simplification worked HIGH

```
F = A'BC + AB'C + ABC' + ABC
  = A'BC + ABC + AB'C + ABC' + ABC   // ABC re-used (A+A=A)
  = BC(A'+A) + AC(B'+B) + AB(C'+C)
  = BC + AC + AB                     // majority function

F = (A + B)(A + B')                  // distributive, AND form
  = A + BB' = A + 0 = A

F = A'B'C + A'BC + AB'C + ABC'
  = A'C(B'+B) + AB'C + ABC'
  = A'C + AB'C + ABC'
  = C(A' + AB') + ABC'   -> A'+AB' = A'+B'  (redundancy)
  = C(A'+B') + ABC' = A'C + B'C + ABC'
```

De Morgan on a long expression: complement the **whole** thing in one step — invert every variable and swap every operator.
`(AB + C'D)' = (AB)'·(C'D)' = (A'+B')·(C+D')`. Complementing term-by-term without swapping the operators is the single most common algebra error.

// §9 MINTERMS, MAXTERMS, SOP & POS

9.1 Definitions MUST

Term	Is	For row `A=1,B=0,C=1` (row 5)
Minterm `mi`	a **product** (AND) of all n variables, each once, true for **exactly one** row	`m5 = A·B'·C` (1 → plain, 0 → complemented)
Maxterm `Mi`	a **sum** (OR) of all n variables, false for **exactly one** row	`M5 = A' + B + C'` (the exact opposite rule)
SOP	OR of AND terms — `AB + C'D`	maps to a 2-level **AND-OR** = **NAND-NAND** network
POS	AND of OR terms — `(A+B)(C'+D)`	maps to a 2-level **OR-AND** = **NOR-NOR** network

The rule that generates both MUST:
Minterm — take the row where `F = 1`; a variable that is `1` appears **plain**, a `0` appears **complemented**; AND them.
Maxterm — take the row where `F = 0`; a variable that is `0` appears **plain**, a `1` appears **complemented**; OR them.
They are exact complements: `Mi = (mi)'`.

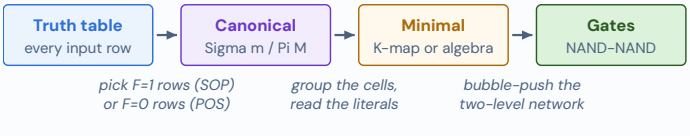
9.2 Σ and Π notation MUST

```
 Row  A B C | F        Canonical SOP (collect the F=1 rows):
  0   0 0 0 | 0          F = m1 + m2 + m4 + m7
  1   0 0 1 | 1            = A'B'C + A'BC' + AB'C' + ABC
  2   0 1 0 | 1          F = Sigma m(1,2,4,7)
  3   0 1 1 | 0
  4   1 0 0 | 1        Canonical POS (collect the F=0 rows):
  5   1 0 1 | 0          F = M0 . M3 . M5 . M6
  6   1 1 0 | 0            = (A+B+C)(A+B'+C')(A'+B+C')(A'+B'+C)
  7   1 1 1 | 1          F = Pi M(0,3,5,6)
```

[MEMORIZE] The index rule. For n variables the two lists **partition** `0…2^n-1`:
`F = Σm(S)` ⟺ `F = ΠM(all indices NOT in S)`
`F' = Σm(not S) = ΠM(S)` — so complementing a function just swaps the two lists.

Trap: the Σ list and the Π list of the **same** function are **different index sets**. `Σm(1,2,4,7) = ΠM(0,3,5,6)`, never `ΠM(1,2,4,7)`. Writing the same numbers under a Π is the classic slip.

9.3 Truth table → gates, the procedure HIGH



Which one do I pick? Count the rows. Few `1`s → SOP is shorter. Few `0`s → POS is shorter. They describe the **same function** and cost the same to *evaluate*; only the gate count differs.

9.4 Don't-care conditions MUST

A **don't-care** (`X` or `d`, written `Σd(…)`) is an input combination that **cannot occur**, or whose output **nobody uses**. You are free to call it `0` or `1` — whichever makes the logic smaller.

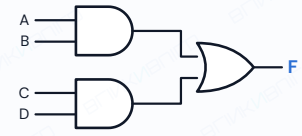
Where they come from	Unused BCD codes `1010`–`1111`; states a one-hot encoding can never reach; outputs ignored while an enable is low.
How to use them	In a K-map, treat an `X` as a **1 if it enlarges a group**, otherwise ignore it. Never form a group made **only** of don't-cares.
Written as	`F(A,B,C,D) = Σm(1,3,7,11,15) + Σd(0,2,5)`

Trap `GEN`: a don't-care is **not** "any value at runtime" — the synthesised gate produces a perfectly definite 0 or 1 for it. It means *you* promised that input never happens. If it does happen, the output is whatever minimisation happened to pick. In `FPGA` / `ASIC` flows this is exactly how an incompletely-specified `case` creates behaviour you did not intend.

9.5 Two-level implementation cost GOOD

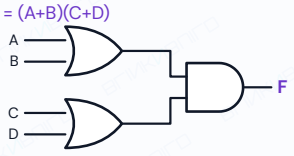
SOP AND-OR -> all NAND

F = AB + CD



POS OR-AND -> all NOR

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



Replace every gate with a NAND (SOP) or a NOR (POS) and the function is unchanged — same structure, one uniform gate library.

Why 2-level matters: every 2-level network has the **same propagation delay** regardless of the function's complexity — 2 gate delays. Multi-level factoring (`AB+AC = A(B+C)`) saves gates but **adds levels**, i.e. trades area for speed. That trade-off is the whole of logic synthesis.

// §10 KARNAUGH MAPS

10.1 Why the columns are in Gray order MUST

A K-map is a truth table redrawn so that **physically adjacent cells differ in exactly one variable**. That is the whole trick: adjacent cells can always be merged, because `XY + XY' = X`. Rows and columns therefore run `00, 01, 11, 10` — **Gray order** — and **not** `00, 01, 10, 11`.

2-variable — F(A,B)

A \ B	0	1
0	m0	m1
1	m2	m3

3-variable — F(A,B,C) note the 11 / 10 order

A \ BC	00	01	11	10
0	m0	m1	m3	m2
1	m4	m5	m7	m6

4-variable — F(A,B,C,D)

AB \ CD	00	01	11	10
00	m0	m1	m3	m2
01	m4	m5	m7	m6
11	m12	m13	m15	m14
10	m8	m9	m11	m10

Read a cell's minterm number by concatenating its row label and column label as binary: row `10`, column `11` → `1011` = m11. Never number the cells left-to-right 0,1,2,3.

10.2 The grouping rules MUST

Group size	A **power of two only**: 1, 2, 4, 8, 16. Never 3, 6 or 12.
Shape	Rectangular, contiguous in the wrapped map. **Diagonals are never adjacent.**
Maximise	Make every group **as large as possible** — each doubling deletes one variable from the term.
Wrap around	Left edge touches right edge, top touches bottom. The **four corners** of a 4-var map form one legal group of 4.
Overlap	**Allowed and often required.** A cell may belong to several groups.
Cover	Every 1 must be in at least one group (for SOP). Stop when they are; do not add redundant groups.
Don't-cares	Use an X as a 1 **only if it grows a group**. Never build a group out of don't-cares alone.

[FORMULA] Group size → term size. On an n-variable map, a group of 2^k cells eliminates **k** variables and leaves a product term of `n - k` literals. Group of 1 on a 4-var map → 4 literals; group of 8 → 1 literal.

10.3 Prime implicants MUST

Implicant	Any legal group of 1s (a product term that implies F).
Prime implicant (PI)	A group that **cannot be made larger** — it is not contained in any bigger legal group.
Essential PI (EPI)	A PI that covers at least one 1 **covered by no other PI**. Every EPI **must** appear in the minimal expression.
Method	Find all PIs → take every EPI → cover whatever 1s remain with the fewest remaining PIs.

10.4 Solved example — SOP MUST

`F(A,B,C,D) = Σm(0,1,2,5,8,9,10)`

AB \ CD	00	01	11	10
00	1^{ab}	1^{bc}	0	1^{a}
01	0	1^{c}	0	0
11	0	0	0	0
10	1^{ab}	1^{b}	0	1^{a}

a = m(0,2,8,10) — the **four corners**, wrapping both ways → B=0, D=0 → `B'D'`
b = m(0,1,8,9) — top and bottom rows, wrapping vertically → B=0, C=0 → `B'C'`
c = m(1,5) — a plain pair → A=0, C=0, D=1 → `A'C'D`

Answer: `F = B'C' + B'D' + A'C'D` — and all three are **essential**: m2 sits only in **a**, m9 only in **b**, m5 only in **c**. Cost: 3 AND gates (2,2,3 inputs) + one 3-input OR, versus 7 four-input ANDs for the canonical form.

10.5 Same map, POS HIGH

Procedure: group the **0s** exactly as before, then write each group as a **sum**, complementing each constant variable — because you are describing `F'` and then inverting it.

0-groups: m(3,7,11,15) = column CD=11 → C=1,D=1 → `(C'+D')` · m(12,13,14,15) = row AB=11 → `(A'+B')` · m(4,6,12,14) → B=1,D=0 → `(B'+D)`

Answer: `F = (A'+B')(B'+D)(C'+D')` — the same function as §10.4. Spot-check `A=0,B=1,C=0,D=1` (m5): `(1+0)(0+1)(1+0) = 1` . ✓

10.6 With don't-cares HIGH

`F = Σm(1,3,7,11,15) + Σd(0,2,5)` — a classic. Using the Xs at 0 and 2 lets the pair m(1,3) grow into the block m(0,1,2,3) = `A'B'`, and m(3,7,11,15) is the column CD=11 = `CD`.

AB \ CD	00	01	11	10
00	X^{b}	1^{b}	1^{ab}	X^{b}
01	0	X	1^{a}	0
11	0	0	1^{a}	0
10	0	0	1^{a}	0

a = `CD` · **b** = `A'B'` → `F = CD + A'B'`. The X at m5 is left as 0 — using it would not enlarge anything.

10.7 K-map traps MUST

[TRAP] The seven that cost marks

- **Diagonal cells are NOT adjacent** — they differ in two variables. m0 and m5 can never be grouped.
- **Edges wrap; corners are adjacent.** The four corners of a 4-var map are one group of 4.
- **Group sizes are powers of two only.** Three adjacent 1s must be covered as a pair plus an overlapping pair, not as a group of 3.
- **Bigger is always better** — a group of 4 beats two groups of 2, even though both are "legal".
- **Overlapping is fine**; a redundant *extra* group is not — unless you are removing a hazard on purpose (§15).
- **Label order is 00, 01, 11, 10.** Writing 00,01,10,11 silently destroys every adjacency.
- **Don't-cares are optional 1s, never mandatory ones.** A group of pure Xs implements logic for something that never happens.

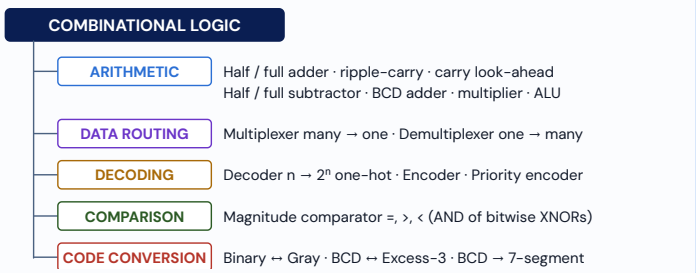
Minimal is not unique. A function can have two different minimal SOPs of equal cost — both are correct. What *is* unique is the set of essential prime implicants.

10.8 Beyond 4 variables BONUS

5 variables	Two 4-var maps side by side (E=0 and E=1). Cells in the **same position** on the two maps are adjacent — imagine them stacked.
6+ variables	K-maps stop being readable. Use **Quine–McCluskey**: tabular, systematic, gives all PIs, and is what a synthesis tool does (Espresso, heuristically).

// §11 COMBINATIONAL CIRCUITS — MASTER MAP

Definition: output is a function of the **present inputs only**. No memory, no clock, no feedback. Given the inputs, the output is fully determined once the gates settle.



// §12 ADDERS

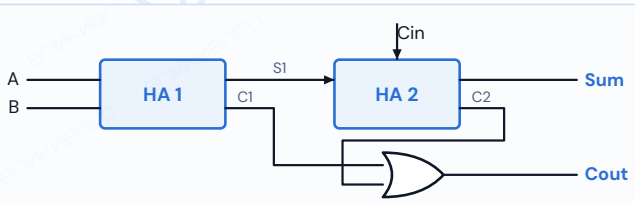
12.1 Half adder MUST

```
 A B | Sum Carry       Sum   = A ^ B      (XOR)
 ----+-----------      Carry = A . B      (AND)
 0 0 |  0    0
 0 1 |  1    0        Adds TWO bits. It has no carry input,
 1 0 |  1    0        so it cannot be cascaded on its own.
 1 1 |  0    1
```

12.2 Full adder MUST

```
 A B Cin | Sum Cout      Sum  = A ^ B ^ Cin
 --------+----------     Cout = A.B + A.Cin + B.Cin      (SOP)
 0 0  0  |  0   0             = A.B + Cin.(A ^ B)       (2-HA)
 0 0  1  |  1   0
 0 1  0  |  1   0       Sum  = 1 when an ODD number of
 0 1  1  |  0   1              the three inputs is 1.
 1 0  0  |  1   0       Cout = MAJORITY(A, B, Cin)
 1 0  1  |  0   1              = 1 when two or more are 1.
 1 1  0  |  0   1
 1 1  1  |  1   1       Both Cout forms are equal - prove it
                        by K-map, or just check all 8 rows.
```

Full adder = 2 half adders + 1 OR MUST



Why the OR never needs to be an XOR: C1 = A·B and C2 = Cin·(A⊕B) can **never both be 1** — if A·B = 1 then A⊕B = 0. Two carries out of one bit position is impossible, so OR and XOR give the same answer here. Interviewers love this one.

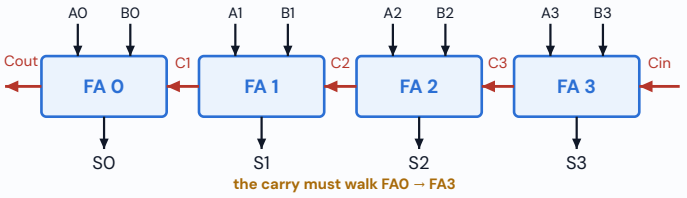
12.3 Subtractors HIGH

```
HALF SUBTRACTOR (A - B)
 A B | D  Bout
 ----+-------
 0 0 | 0   0
 0 1 | 1   1   <- borrow
 1 0 | 1   0
 1 1 | 0   0

 D    = A ^ B
 Bout = A' . B
```

```
FULL SUBTRACTOR (A - B - Bin)
 D    = A ^ B ^ Bin
 Bout = A'.B + A'.Bin + B.Bin
      = A'.B + Bin.(A ^ B)'

Note the symmetry with the full
adder: the ONLY change is that A
is complemented in the borrow term.
```

Nobody builds subtractors GEN. Real hardware subtracts with an **adder**: `A - B = A + B' + 1`. Feed `B ^ sub` to the adder inputs and tie `Cin = sub`. One block does both, and the same carry chain serves both. Subtractors survive in exams, not in silicon.

12.4 Ripple-carry adder (RCA) MUST



the carry must walk FA0 → FA3

Structure	*n* full adders chained, `Cout` of stage *i* → `Cin` of stage *i*+1.
Delay	**O(n)** — with a 2-level FA the carry costs ~2 gate delays per stage, so worst-case ≈ 2n gate delays, with the final sum ≈ 2n+1. **More stages → proportionally slower.**
Cost	Smallest area and the simplest layout of any adder. Still the right choice for narrow words.
Worst case	`1111…1 + 0000…1` — the carry propagates the entire width.

12.5 Carry look-ahead adder (CLA) MUST

The idea: stop waiting for the carry — **compute it directly from the inputs**. Each bit either **generates** a carry on its own, or **propagates** one that arrives.

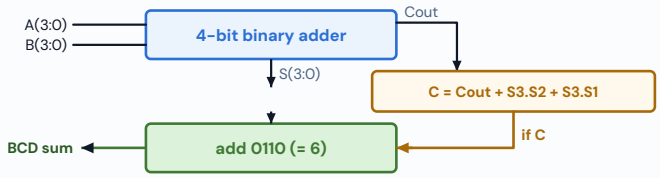
```
Gi = Ai . Bi        // GENERATE : this bit makes a carry
Pi = Ai ^ Bi        // PROPAGATE: this bit passes one along
C(i+1) = Gi + Pi.Ci              // the recurrence
Si     = Pi ^ Ci                 // the sum bit

Unrolled - every carry is now a 2-level function of A,B,C0:
C1 = G0 + P0.C0
C2 = G1 + P1.G0 + P1.P0.C0
C3 = G2 + P2.G1 + P2.P1.G0 + P2.P1.P0.C0
C4 = G3 + P3.G2 + P3.P2.G1 + P3.P2.P1.G0 + P3.P2.P1.P0.C0
```

Why faster	Every carry is a **2-level AND-OR** of the inputs, so all carries appear at the **same time**. Total ≈ **3 gate delays** (1 for P/G, 2 for the carry) regardless of width — `O(1)` instead of `O(n)`.
What it costs	Fan-in and gate count grow **quadratically**. A 16-bit flat CLA would need a 17-input gate, so real designs cascade **4-bit CLA blocks**.
Block level	`P* = P3P2P1P0`, `G* = G3 + P3G2 + P3P2G1 + P3P2P1G0` — then look ahead *between* blocks the same way.

Convention note: some texts define propagate as `Pi = Ai + Bi`. Either definition produces the **same carries**, but only `Pi = Ai ⊕ Bi` can be re-used for the sum bit `Si = Pi ⊕ Ci`, which is why it is the usual choice.

Adder	Delay	Area	Use
Ripple carry	O(n)	lowest	narrow words, area-critical
Carry look-ahead	O(log n) blocked	high	the classic fast adder
Carry-select	O(√n)	~2×	compute both carries, MUX the answer
Carry-save BONUS	O(1) per add	low	multipliers: defer the carry to one final adder

12.6 BCD adder GOOD



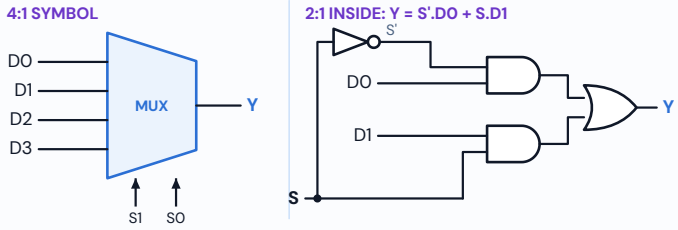
`S3S2` catches 12–15, `S3S1` catches 10, 11, 14, 15, and `Cout` catches 16–19 — together every result above 9.

12.7 ALU and multipliers GOOD

ALU	An adder plus a logic block, with a **function-select** input choosing add / sub / AND / OR / XOR / shift. Flags out: `Z` (zero), `N` (sign), `C` (carry), `V` (overflow) — see §3.2.
Array multiplier	*n*×*n* → **2n** bits. Partial products are `Ai·Bj` AND gates, summed by a matrix of adders. Delay `O(n)`; a Wallace/Dadda tree gets it to `O(log n)`.

// §13 MULTIPLEXER & DEMULTIPLEXER

13.1 MUX = data selector MUST




```
2:1   Y = S'.D0 + S.D1              // S=0 picks D0

4:1   Y = S1'S0'.D0 + S1'S0.D1 + S1S0'.D2 + S1S0.D3
        = SUM over i of ( minterm_i(S) . Di )

n selects, general form:  Y = SUM  m_i(S) . D_i     i = 0..2^n-1
```

A MUX is a minterm engine. The select lines generate every minterm; each data input decides whether that minterm reaches the output. That single observation is why a MUX can implement *any* Boolean function.

13.2 Implementing a function with a MUX MUST

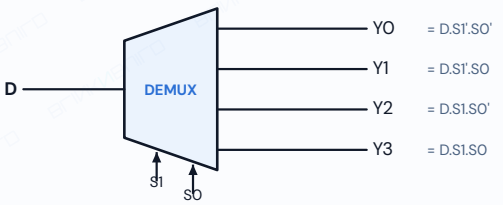
Direct method	A **2^n:1** MUX implements **any n-variable** function: wire the n variables to the selects and pour the truth-table output column straight into the data inputs. No gates at all.
Shannon method	A **2^n:1** MUX implements **any (n+1)-variable** function: n variables on the selects, and each data input takes one of **four** values — `0`, `1`, `X`, or `X'` — where X is the leftover variable.

```
EXAMPLE  F(A,B,C) = Sigma m(1,2,6,7)  on a 4:1 MUX
Put A,B on the selects; group the truth table in pairs of C:

  A B | C=0  C=1 | data input needed
  ----+---------+------------------
  0 0 |  0    1  | D0 = C
  0 1 |  1    0  | D1 = C'
  1 0 |  0    0  | D2 = 0
  1 1 |  1    1  | D3 = 1

  -> F = MUX(sel = A,B ; D0=C, D1=C', D2=0, D3=1)
     one 4:1 MUX + one inverter replaces the whole SOP.
```

How to read the table fast: for each select combination look at the two output rows. `0,0` → tie `0`. `1,1` → tie `1`. `0,1` → the variable itself. `1,0` → its complement. Four cases, nothing to derive.

Cascading: build a 16:1 from five 4:1 MUXes (four in the first rank fed by `S1S0`, one selecting between them with `S3S2`). A MUX with an **enable** can be cascaded by decoding the high select bits onto the enables instead.

13.3 DEMUX = data distributor HIGH



[MEMORIZE] A DEMUX with its data input tied HIGH *is* a decoder. The two blocks are the same silicon; the data pin becomes the enable. That is why parts are sold as "decoder/demultiplexer".

// §14 DECODERS, ENCODERS, COMPARATORS

14.1 Decoder — n to 2ⁿ MUST

```
2-to-4 DECODER (active high, with enable E)
  E A1 A0 | Y3 Y2 Y1 Y0      Y0 = E.A1'.A0'   = E . m0
  --------+-----------      Y1 = E.A1'.A0    = E . m1
  0  x  x |  0  0  0  0      Y2 = E.A1 .A0'   = E . m2
  1  0  0 |  0  0  0  1      Y3 = E.A1 .A0    = E . m3
  1  0  1 |  0  0  1  0
  1  1  0 |  0  1  0  0      Each output IS one minterm.
  1  1  1 |  1  0  0  0      Exactly one output is active.
```

Sizes	2→4, 3→8, 4→16. *n* inputs drive **up to** 2^n outputs (a BCD decoder has 4 inputs but only 10 used outputs).
Building SOP	**Decoder + OR gate.** `F = Σm(1,3,5)` → OR outputs Y1, Y3, Y5. With an **active-low** (NAND) decoder, use a **NAND** gate instead — same function.
Multiple outputs	One decoder can feed several OR gates, so *k* functions of the same variables cost one decoder and *k* gates. This is exactly how a **ROM** is organised (§29).
Cascading	Use the enable: a 4→16 is two 3→8 decoders with `A3` and `A3'` driving their enables.
Applications	**Memory address decoding / chip select**, instruction decode, one-hot FSM state decode, 7-segment drivers.

Trap — active-low outputs. Most catalogue decoders (74138, 74154) have **active-LOW** outputs: the selected line goes to **0** and all the others sit at 1. Reading such a part as active-high inverts your entire answer. Look for the bubble, or the bar in `Y0'` / the `_n` suffix (§16).

14.2 Encoder vs priority encoder MUST

Plain encoder ($2^n \to n$)	Priority encoder
Assumes **exactly one** input is active.	Accepts **any number** of active inputs.
Two inputs high → output is the OR of both codes, i.e. **garbage**.	Encodes the **highest-priority** active input and ignores the rest.
No input high → output `000`, indistinguishable from D0 being high.	A **valid** output `V` distinguishes "no request" from "D0 requesting".
Rarely used alone.	Interrupt controllers, arbiters, leading-one / priority-resolve logic.

```
4-to-2 PRIORITY ENCODER  (D3 = highest priority)
  D3 D2 D1 D0 | Y1 Y0  V      Y1 = D3 + D2
  ------------+----------     Y0 = D3 + D2'.D1
   0  0  0  0 |  x  x  0      V  = D3 + D2 + D1 + D0
   0  0  0  1 |  0  0  1
   0  0  1  x |  0  1  1      x = don't care. V=0 means the
   0  1  x  x |  1  0  1      Y outputs carry no meaning -
   1  x  x  x |  1  1  1      ALWAYS qualify them with V.
```

14.3 Magnitude comparator HIGH

```
1-bit:  A>B : A.B'        A<B : A'.B        A=B : (A ^ B)'

2-bit (A1A0 vs B1B0) - compare from the MSB DOWN:
  E1 = (A1 ^ B1)'                          // MSBs equal
  A>B = A1.B1' + E1.A0.B0'
  A<B = A1'.B1 + E1.A0'.B0
  A=B = E1 . (A0 ^ B0)'                    // AND of all XNORs
```

The pattern generalises: equality is the **AND of bitwise XNORs** (or the NOR of bitwise XORs). Greater-than is "the first differing bit, from the MSB, is 1 in A". Cascade inputs on parts like the 7485 let you chain comparators to any width.

14.4 Code converters GOOD

Binary → Gray	`G[MSB] = B[MSB], G[i] = B[i+1] ⊕ B[i]` — a row of XORs, all in parallel. See §4.4.
Gray → Binary	`B[MSB] = G[MSB], B[i] = B[i+1] ⊕ G[i]` — a **chain** of XORs, so it is inherently serial and slower.
BCD → Excess-3	Add `0011`. As logic: `E3 = B3+B2B1+B2B0, E2 = B2'(B1+B0) + B2B1'B0', E1 = (B1⊕B0)', E0 = B0'`.
Binary → BCD	**Double-dabble** (shift-and-add-3): shift left one bit at a time; before each shift, add 3 to any BCD digit currently ≥ 5.
BCD → 7-segment	Seven independent functions of 4 inputs, each minimised on its own K-map with `1010–1111` as don't-cares.

// §15 HAZARDS & GLITCHES

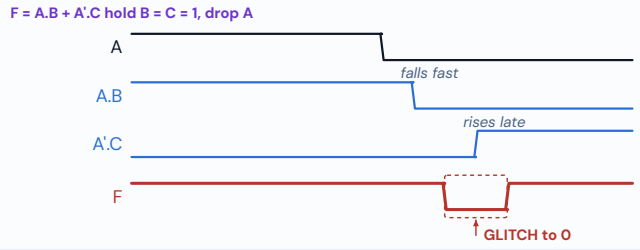
15.1 What a hazard is HIGH

A **hazard** is the *possibility* of an unwanted momentary transition on a combinational output caused by **unequal path delays**. The **glitch** is the pulse you actually observe. Steady-state logic is still perfectly correct — the fault exists only during the settling window.

Type	Output should…	But momentarily…	Lives in
Static-1	stay at **1**	dips to 0	SOP / AND-OR networks
Static-0	stay at **0**	jumps to 1	POS / OR-AND networks
Dynamic	change **once**	changes 3+ times (1→0→1→0)	multi-level (3+) networks

Logic hazard	Caused by a **single** input changing. **Removable** by adding a redundant gate.
Function hazard	Caused by **two or more** inputs changing at once. **Cannot be removed by any gate network** — you must avoid the simultaneous change.

15.2 The canonical static-1 hazard MUST



The fix is the consensus term. `F = AB + A'C + BC`. When `B=C=1` the extra term is 1 no matter what A does, so it **bridges** the handover. The term is logically redundant (§8.1 consensus) — that is exactly why the minimiser deleted it, and why you must put it back by hand.

Spot it on the K-map: a hazard exists wherever **two adjacent 1-cells are covered by different groups and no single group contains both**. Add an overlapping group that straddles the boundary. For a **static-0** hazard do the same with adjacent **0**-cells in the POS cover.

15.3 When do hazards actually matter? MUST

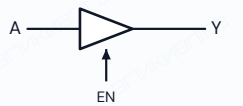
Usually not	In a **synchronous** design a glitch that settles before the setup window is **invisible** — the flip-flop only looks at the clock edge. This is the single biggest practical argument for synchronous design (§17).
Definitely yes	When the signal drives something **edge- or level-sensitive** outside the clocked path: a **clock**, an **asynchronous reset/set**, a **latch enable**, a write-strobe, an interrupt line, or any output leaving the chip.

Trap: "add a buffer to equalise the delays" does not remove a hazard — it moves it. Only a **redundant covering term** removes a logic hazard, and nothing removes a function hazard. Gating a clock with combinational logic is the classic way to turn a harmless glitch into a real failure.

// §16 TRI-STATE BUSES & ACTIVE-LOW NOTATION

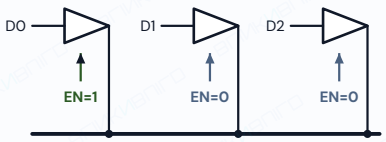
16.1 Tri-state logic HIGH

TRI-STATE BUFFER



EN=1 Y = A
EN=0 Y = Hi-Z

ONE BUS, MANY DRIVERS



shared bus — exactly one enable active at a time

```
EN A | Y
-----+----
 0 x | Z     <- HIGH IMPEDANCE: the driver is
 1 0 | 0        electrically DISCONNECTED, not
 1 1 | 1        driving a 0 and not driving a 1.
```

Third state	**Hi-Z** is *not* a logic level — the driver is electrically **disconnected**. The line then floats unless something else drives it.
Bus rule	Many drivers may share a wire, but **exactly one** enable may be active at a time. Two drivers fighting = **bus contention**: a low-resistance path from VDD to GND, large current, and possible damage.
Floating bus	Add a weak **pull-up or pull-down** (a "bus keeper") so an idle bus reads as a defined level instead of drifting.
Open-drain	Can only pull **low**; an external pull-up provides the high. Multiple devices wire-AND safely with no contention — this is how **I²C** and shared interrupt lines work.
Modern practice	FPGA ASIC Internal tri-state buses are obsolete — synthesis replaces them with **multiplexers**. Tri-state survives at **I/O pads**, where a bidirectional pin needs it.

16.2 Active-high vs active-low MUST

Active-HIGH	Active-LOW
Asserted (doing its job) when the signal is **1**.	Asserted when the signal is **0**.
Written plain: `reset`, `enable`, `we`.	Written `reset_n`, `cs_n`, `oe_b`, `nRESET`, RESET — or drawn with a **bubble** on the symbol.
`if (reset)` → we ARE in reset.	`if (reset_n)` → we are **NOT** in reset. Assertion is `if (!reset_n)`.

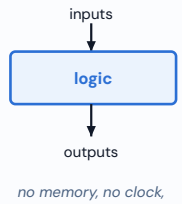
Why so many active-low signals GEN: bipolar outputs sank current far better than they sourced it, so asserting low was the stronger drive. It also **fails safe** — a broken or unconnected line is pulled high by its pull-up and reads as **de-asserted**. The convention outlived the technology, so most chip-select, output-enable, write-enable and reset pins are still active-low.

[TRAP] Reading a bubble. A bubble on an **input** means "this pin is active-low", not "invert the signal you send". A bubble on an **output** means the asserted state is 0. Mixing the two flips your entire truth table — see the decoder trap in §14.1.

// §17 COMBINATIONAL VS SEQUENTIAL — THE DIVIDE

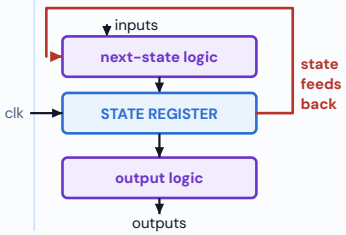
Feature	Combinational	Sequential
Memory	**No**	**Yes** — stores state
Output depends on	present inputs only	present inputs **+ stored state**
Feedback	none	**required** — that is what creates memory
Clock	not required	usually (synchronous); latches may be level-driven
Analysed with	truth table, K-map, Boolean algebra	state diagram, state table, excitation table
Same input twice	always the same output	output may differ — the state moved on
Examples	gates, adder, MUX, decoder, encoder, comparator, ALU	latch, flip-flop, register, counter, shift register, FSM, memory

COMBINATIONAL



no memory, no clock, no feedback path

SEQUENTIAL



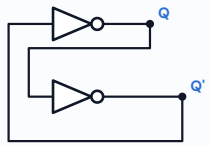
Why synchronous design wins MUST: with one clock, every path has the same deadline, so timing becomes a set of inequalities a tool can check exhaustively (§21–22). Glitches are free to settle between edges. Races and hazards inside the combinational cloud simply never get sampled. Asynchronous logic can be faster and lower power, but it has **no such analysis** — which is why almost all commercial digital design is synchronous.

Trap — "is this circuit combinational?" Look for a **feedback path**. Any loop of gates that feeds an output back to an input creates memory, whether you wanted it or not. Two cross-coupled NAND gates are an SR latch, not a combinational block (§18).

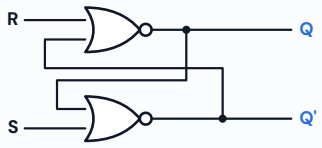
// §18 LATCHES

18.1 Where memory comes from MUST

2 CROSS-COUPLED INVERTERS = 1 BIT



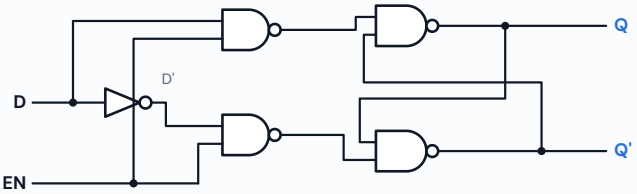
NOR SR LATCH (active HIGH)




```
NOR SR LATCH (active HIGH)        NAND SR LATCH (active LOW)
  S R | Q(next)                     S' R' | Q(next)
  ----+---------                    ------+---------
  0 0 | Q      hold                  1  1  | Q      hold
  0 1 | 0      reset                 1  0  | 0      reset
  1 0 | 1      set                   0  1  | 1      set
  1 1 | FORBIDDEN                    0  0  | FORBIDDEN
      | Q = Q' = 0                         | Q = Q' = 1
```

[TRAP] Why S=R=1 is "forbidden". It is not that the latch explodes — it is that **Q and Q' are both driven to the same level**, breaking the complement, and that **releasing both inputs together** leaves the next state decided by which gate happens to be faster. That is a **race condition**: the result is unpredictable and varies with temperature and voltage.

18.2 Gated SR and D latch MUST



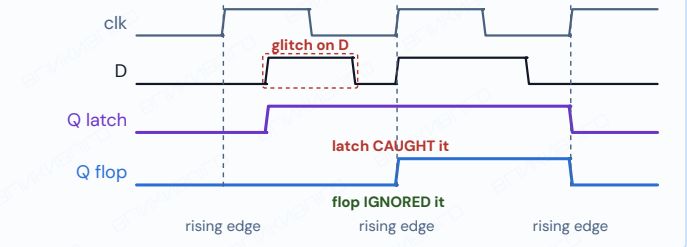

```
EN D | Q(next)       Q(next) = EN.D + EN'.Q
-----+---------
 0 x |   Q     HOLD  The D latch removes the forbidden
 1 0 |   0           state by making R the complement
 1 1 |   1           of S - so EN=1 just passes D on.
```

Level-sensitive / transparent means: while the enable is active, the output **tracks the input** — every change, and every **glitch**, passes straight through. The value is captured only at the moment the enable goes inactive. That is the whole difference from a flip-flop.

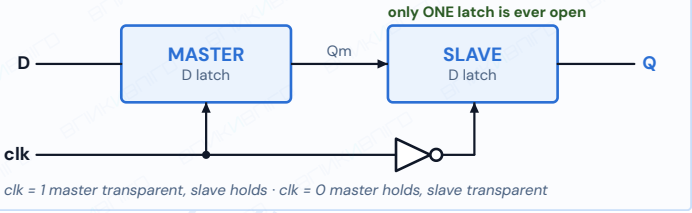
// §19 FLIP-FLOPS

19.1 Latch vs flip-flop MUST

Latch	Flip-flop
Level-sensitive	**Edge**-triggered (rising or falling)
Transparent for the whole active level	Samples in an **instant**; opaque otherwise
Input glitches propagate to the output	Glitches between edges are **ignored**
Smaller, faster, less power	≈2× the area (built from two latches)
Timing is a **window** — hard to analyse; needs time borrowing	Timing is a **point** — one clean setup/hold check (§21)
Deliberate in some ASIC flows; **accidental in RTL = a bug**	The default storage element of synchronous design



19.2 Master-slave: how an edge is made HIGH



clk = 1 master transparent, slave holds · clk = 0 master holds, slave transparent

Only **one** latch is ever transparent, so there is never a path straight from D to Q. That is exactly what makes the device edge-triggered — and it is why a flip-flop costs about twice a latch.

19.3 The four flip-flops MUST

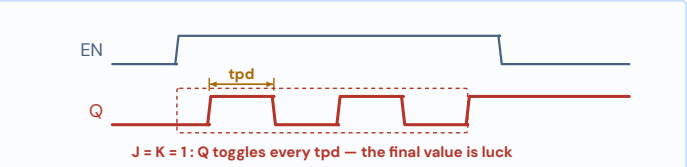
FF	Characteristic equation	Behaviour	Built from a D-FF by
D	`Qnext = D`	store / delay by one clock	— (it is the primitive)
T	`Qnext = T ⊕ Q`	T=1 toggle, T=0 hold	`D = T ⊕ Q`
JK	`Qnext = J·Q' + K'·Q`	set / reset / **toggle**, no illegal state	`D = J·Q' + K'·Q`
SR	`Qnext = S + R'·Q` (with `S·R = 0`)	set / reset, `S=R=1` illegal	`D = S + R'·Q`

```
CHARACTERISTIC (truth) TABLES  -  "given the inputs, what is Qnext?"

 D | Qn      T Q | Qn        J K | Qn          S R | Qn
 --+----     ----+----       ----+---------    ----+---------
 0 | 0       0 0 | 0         0 0 | Q   hold    0 0 | Q   hold
 1 | 1       0 1 | 1         0 1 | 0   reset   0 1 | 0   reset
             1 0 | 1         1 0 | 1   set     1 0 | 1   set
             1 1 | 0         1 1 | Q'  TOGGLE  1 1 | ILLEGAL
               toggle
```

JK is the "complete" flip-flop — it is an SR flip-flop in which the illegal `1,1` input has been redefined as **toggle**. Tie `J=K=T` and you have a T flip-flop; tie `J=D, K=D'` and you have a D flip-flop.

19.4 Race-around condition MUST

The problem `GEN`: in a **level-triggered** JK *latch* with `J = K = 1`, the output toggles. But the new Q feeds straight back to the inputs, so while the enable is still high it toggles **again**, and again. If the active pulse width `tp` is longer than the propagation delay `tpd`, the output oscillates and the final state is **unpredictable**.



Fix 1 — edge triggering	Sample at an instant, so there is no window in which to toggle twice. **The modern answer.**
Fix 2 — master-slave	Only one latch is open at a time, so the feedback cannot race round (§19.2).
Fix 3 — `tp < tpd`	Narrow the clock pulse below the propagation delay. Textbook answer, **impractical** in real silicon.

Say it precisely: race-around is a property of a **level-triggered** JK. A true **edge-triggered** JK flip-flop does not suffer from it, so "JK flip-flops have race-around" is only half right — and interviewers listen for that qualifier.

// §20 EXCITATION TABLES & FF CONVERSION

20.1 The excitation tables MUST

A characteristic table answers *"given the inputs, what is Q_{next}?"*. An **excitation table** answers the **design** question: *"I want this transition — what must I drive?"* This is the table you use to build counters and FSMs (§27, §28).

```
Q -> Qnext |  S R  |  J K  |  D  |  T
-----------+-------+-------+-----+-----
  0 -> 0   |  0 x  |  0 x  |  0  |  0
  0 -> 1   |  1 0  |  1 x  |  1  |  1
  1 -> 0   |  0 1  |  x 1  |  0  |  1
  1 -> 1   |  x 0  |  x 0  |  1  |  0
             ^       ^       ^     ^
         SR needs  JK is    D just  T = 1 when
         care with mostly   copies  the bit must
         S.R = 0   don't-   Qnext   CHANGE
                   cares
```

[MEMORIZE] Two one-line rules. `D = Qnext` — the D input is simply the value you want next.
`T = Q ⊕ Qnext` — the T input is 1 exactly when the bit must **change**.
JK has the most don't-cares (half the table), which is why JK-based counter designs minimise to the smallest logic — the reason it dominates older textbooks.

20.2 Converting one flip-flop into another HIGH

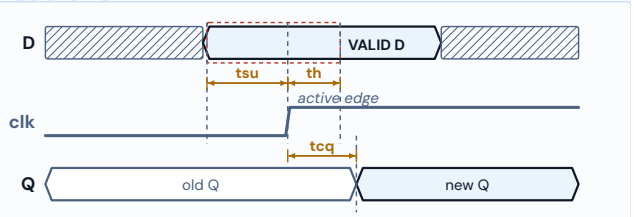
Have → Want	Excitation logic
D → T	`D = T ⊕ Q`
D → JK	`D = J·Q' + K'·Q`
D → SR	`D = S + R'·Q`
JK → D	`J = D, K = D'`
JK → T	`J = K = T`
T → D	`T = D ⊕ Q`

Method for any conversion: write the excitation table of the FF you **have**, drive it from the characteristic table of the FF you **want**, K-map each input against `(inputs, Q)`, and read off the logic. Two variables, four rows — it never takes more than a minute.

Reality check `FPGA` `ASIC`: modern libraries and FPGA fabrics provide only the **D flip-flop**. JK, T and SR are exam devices; synthesis builds them out of a D-FF plus the little bit of logic in the table above. Knowing the conversions is still worth marks — and it is exactly how you write a toggle in RTL: `q <= q ^ t;`

// §21 SETUP, HOLD & CLOCK-TO-Q

21.1 The three numbers MUST



Setup time `tsu`	How long D must already be stable **BEFORE** the active clock edge.
Hold time `th`	How long D must remain stable **AFTER** the active clock edge. Can be zero or even negative in some libraries.
Clock-to-Q `tcq`	From the active edge until Q is valid. It is the launching FF's contribution to the next path.
Aperture	`tsu + th` — the total window the flip-flop needs quiet. Violate it and you risk **metastability** (§23).

Mnemonic: **S**etup comes before, **H**old comes after — alphabetical order matches time order. Both are characterised properties of the **flip-flop cell**, not things you choose.

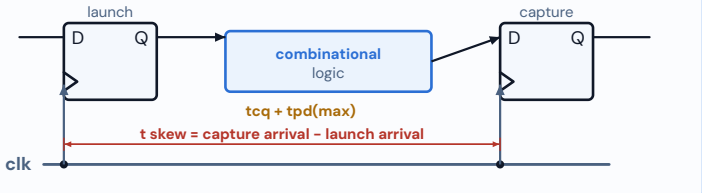
21.2 Propagation vs contamination delay MUST

Propagation delay `tpd`	Contamination delay `tcd`
MAXIMUM — the last moment the output is finally stable.	**MINIMUM** — the first moment the output may *start* to change.
The **longest** path. Drives the **SETUP** check.	The **shortest** path. Drives the **HOLD** check.
Limits how fast you can clock.	Independent of clock speed entirely.

Always `tcd ≤ tpd`. Both are needed: one alone cannot describe a path, because setup and hold are checked against **opposite extremes** of the same logic cloud.

// §22 SKEW, JITTER & MAXIMUM FREQUENCY

22.1 The two timing equations MUST



[FORMULA] Convention used here:
`tskew = (clock arrival at CAPTURE flop) - (arrival at LAUNCH flop)`.
Positive skew means the capture clock arrives **later**.

SETUP (the long path must arrive in time):
$T_{clk} \geq t_{cq} + t_{pd(max)} + t_{su} - t_{skew}$

HOLD (the short path must not arrive too early):
$t_{cq} + t_{cd(min)} \geq t_h + t_{skew}$

$f_{max} = 1 / T_{clk(min)}$. With zero skew this collapses to the exam form $T_{clk} \geq t_{cq} + t_{pd} + t_{su}$.

Assumptions: single clock, ideal (jitter-free) source, one combinational stage between two edge-triggered flops, no clock gating. Real STA adds jitter, on-chip variation and derating on top.

22.2 Setup vs hold failure — the key asymmetry MUST

	SETUP violation	HOLD violation
Cause	logic path too **slow**	logic path too **fast**
Depends on T_{clk}?	**Yes**	**No** — T_{clk} is not in the equation
Fix by slowing the clock	**Yes** — always works	**NO. Never.**
Real fixes	shorten the path, pipeline it, upsize cells, use **useful skew** (delay the capture clock)	**add delay** (buffers) on the data path, or reduce the clock skew
Severity	Recoverable — the chip just runs slower	**Fatal** — the silicon is dead at *every* frequency

[TRAP] The single most-asked timing question. "You have a hold violation — can you fix it by lowering the clock frequency?" **No.** Hold is a race between two paths launched by the **same edge**; the clock period never enters it. A hold violation must be fixed structurally, which is why hold failures found after tape-out kill a chip.

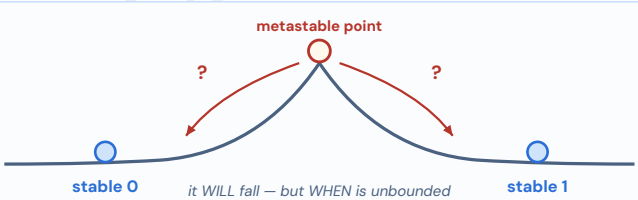
22.3 Skew, jitter and the critical path HIGH

Clock skew	A **static, spatial** difference in clock arrival between two flops, caused by unequal routing. Positive skew **helps setup, hurts hold**; negative skew does the reverse. Deliberately applied, it is called **useful skew**.
Clock jitter	A **dynamic, temporal** variation of the same edge cycle to cycle (PLL noise, supply noise). It eats **setup margin** and cannot be designed out — you budget for it.
Critical path	The register-to-register path with the **largest** $t_{cq}+t_{pd}+t_{su}$. It alone sets f_{max}. Speeding up anything else changes nothing.
Slack	`required - arrival`. Positive = passes, **negative = violation**. The critical path is the one with the worst slack (WNS).

Worked example. $t_{cq}=0.3$, $t_{pd}=4.2$, $t_{su}=0.2$ ns, skew 0.
$T_{min} = 0.3+4.2+0.2 = 4.7$ ns → $f_{max} \approx 212$ MHz. Pipeline the logic into two 2.1 ns halves → $T = 0.3+2.1+0.2 = 2.6$ ns → 385 MHz, at the cost of one extra cycle of **latency**. Throughput up, latency up — the classic trade.

// §23 METASTABILITY

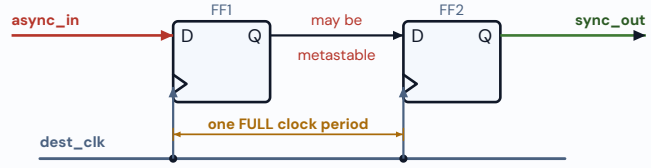
23.1 What it actually is MUST



Cause	A **setup or hold violation** — almost always an **asynchronous** input, or a signal crossing clock domains.
Symptom	Output sits between valid levels, resolves late, and different fanout loads may sample **different values** — the state machine splits.
Not fixable by	Faster flops, buffers, more careful routing. You cannot **prevent** it when sampling a truly asynchronous signal — you can only make it **rare**.

[FORMULA] MTBF — mean time between metastability-induced failures:
$MTBF = e^{(t_r/\tau)} / (T_0 \cdot f_{clk} \cdot f_{data})$
t_r = resolution time allowed · τ = the flop's settling time constant · T_0 = its metastability-window constant · f_{clk} = sampling rate · f_{data} = rate of async transitions.
The **exponential** in t_r is the whole point: each extra flop buys a full clock period of t_r, multiplying MTBF by $e^{(T/\tau)}$ — often many orders of magnitude.

23.2 The 2-flop synchroniser MUST



Be precise about what it does MUST: a 2-flop synchroniser does **not eliminate** metastability — FF1 still goes metastable exactly as often. It **reduces the probability that metastability propagates** into the design, by giving FF1 a whole period to resolve. Saying "the synchroniser removes metastability" is the wrong answer; saying "it raises MTBF to years/centuries" is the right one. Add a third flop for very high f_{clk}.

[TRAP] Rules for synchronisers: place the two flops **adjacent** (minimum routing between them); drive FF1's D from a **single source**, never from combinational logic that could glitch; never take a tap off FF1's output — **only FF2** may be used; and never feed one async source into two separate synchronisers, or they may disagree.

// §24 CLOCK DOMAIN CROSSING (CDC)

Signal type	Correct technique	Why
1-bit level	**2-flop synchroniser**	The standard case. Source must hold the level long enough to be sampled (see below).
1-bit pulse	**Toggle synchroniser** — convert the pulse to a level toggle, sync it, then edge-detect	A pulse shorter than the destination period can be missed entirely.
Multi-bit, related (a counter)	**Gray code** the value first (§4.4)	Only one bit changes per step, so a mid-flight sample gives the old or new value — never a wild one.
Multi-bit, arbitrary	**MUX recirculation**: sync a single *valid* bit, and load the data bus only when valid arrives	The bus is stable long before valid is used, so only one bit ever crosses asynchronously.
Streaming data	**Asynchronous FIFO** — dual-port RAM + Gray-coded read/write pointers, each synchronised into the other domain	Decouples the two rates entirely; the industry-standard block.
Request/ack	**Handshake** (4-phase or 2-phase)	Slow but bulletproof, and needs no assumption about relative frequency.

[TRAP] Never put a multi-bit bus through parallel 2-flop synchronisers. Each bit resolves independently, so on the cycle the bus changes you can latch a combination that **never existed** — `0111 → 1000` can be sampled as `1111` or `0000`. This is the single most common real CDC bug. Use Gray code, a handshake, or a FIFO.

The 3-edge rule: for a level to be reliably caught by a 2-flop synchroniser it must remain stable for at least **1.5–2 destination clock periods**. Crossing from a fast domain to a slow one, a short pulse can vanish — that is exactly when you need the toggle synchroniser.

// §25 RESET STRATEGY

	Synchronous reset	Asynchronous reset
Acts	only at the active clock edge	**immediately**, clock or no clock
Needs a running clock	**Yes** — dead if the clock is stopped	No
Glitches on reset	**Filtered** — sampled like data	**Dangerous** — a glitch resets the chip
Timing	Adds to the data path; must meet **setup**	Off the data path, but de-assertion must meet **recovery / removal**
Area `ASIC`	extra logic in front of D	uses the cell's dedicated reset pin — usually smaller

Recovery and removal are the async-reset twins of setup and hold, applied to the **de-asserting** edge of reset: **recovery** = reset must be released a minimum time *before* the clock edge; **removal** = it must stay released a minimum time *after*. Violate them and the flop goes metastable coming *out* of reset.

Best of both — "async assert, sync de-assert" MUST. Feed the raw reset into the async-reset pins of a 2-flop chain whose D input is tied to 1; use that chain's output as the design's reset. The assertion is immediate (works with no clock), while the release is aligned to the clock, so recovery/removal are met everywhere. This is the standard reset synchroniser in real `ASIC` and `FPGA` designs.

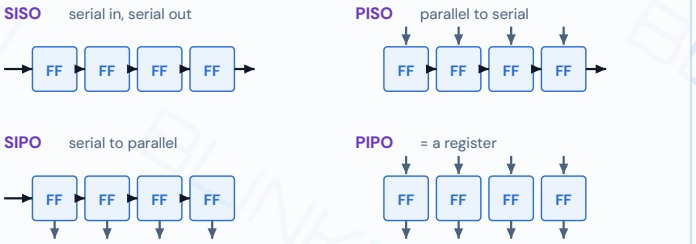
// §26 REGISTERS & SHIFT REGISTERS

26.1 Register HIGH

An *n*-bit **register** is simply *n* flip-flops sharing one clock — it stores a word. Add an **enable** and it becomes a load register: `Qnext = EN ? D : Q`, built as a 2:1 MUX in front of every D input.

Trap FPGA ASIC: implement an enable by **recirculating the data** through a MUX, **not** by gating the clock. A gated clock is a combinational signal driving a clock pin — it glitches, it breaks static timing analysis, and it defeats the whole point of synchronous design. (Real clock gating exists, but it uses a dedicated, glitch-free integrated clock-gating cell.)

26.2 The four shift registers MUST

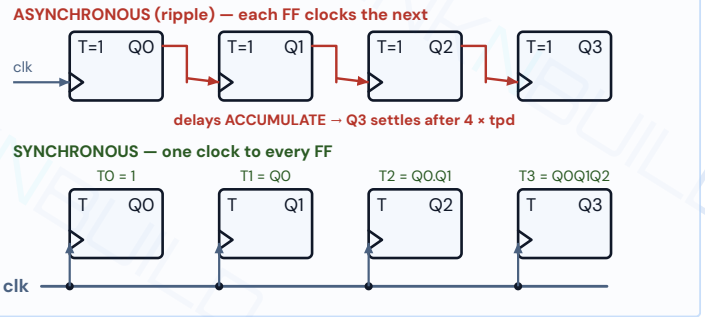


Type	Clocks to load	Clocks to read	Use
SISO	n	n	delay line, pipeline stage
SIPO	n	1	UART RX, SPI RX, de-serialise
PISO	1	n	UART TX, SPI TX, serialise
PIPO	1	1	storage register

Universal shift register (e.g. 74194): a 4:1 MUX on each D input gives four modes — **hold, shift left, shift right, parallel load** — selected by two mode bits. **LFSR** BONUS: a shift register whose input is an XOR of selected taps; with a maximal-length polynomial it cycles through all `2^n-1` non-zero states, giving a pseudo-random sequence or a very cheap counter. The all-zero state is a **lock-up** state.

// §27 COUNTERS

27.1 Asynchronous (ripple) vs synchronous MUST



	Synchronous	Asynchronous (ripple)
Clock	one common clock to all FFs	each FF clocked by the previous output
Delay	**constant** — one `tcq` + a little logic	**cumulative** — `n × tpd`
Max frequency	high, and independent of width	falls as `1/n`
Logic	more (AND chain / carry logic)	almost none — the simplest counter there is
Decoding	clean	**glitches** — outputs change at different times, so decoded terms spike
STA	fully analysable	not properly analysable; avoid in real designs

[TRAP] The ripple-counter glitch. Going `0111 → 1000`, a 4-bit ripple counter passes through `0110, 0100, 0000` as the carry walks along. Any decoder watching it emits **spurious pulses**. Harmless if you resample synchronously; fatal if it drives a clock, a reset or a chip select.

27.2 Counter formulas MUST

[FORMULA] With *N* flip-flops the maximum modulus is `MOD = 2^N`.
For a required modulus *M*: `N = ceil(log2 M)` flip-flops.
A MOD-*M* counter counts `0 … M-1` and has `2^N - M` **unused states**.
Ripple counter maximum frequency ≈ `1 / (N · tpd)`; synchronous ≈ `1 / (tcq + tlogic + tsu)`.

Counter	FFs	States	Sequence
Binary MOD-2^N	N	2^N	full binary count
Decade (BCD)	4	10	0000…1001, 6 unused
Ring	N	**N**	one-hot: `1000→0100→0010→0001`
Johnson (twisted ring)	N	**2N**	`0000→1000→1100→1110→1111→0111→0011→0001`

Ring vs Johnson. A ring counter feeds `Qlast` back to the first input; a Johnson counter feeds back `Q'last` — one inverter doubles the state count. Both need **initialisation** (they are not self-starting) and both decode with a **2-input gate per state** and produce **no glitches**, which is why they are used for clean multi-phase clocks and stepper drives. Cost: very poor state-count-per-flop.

27.3 Designing a synchronous counter MUST

an unused state that loops onto itself = a counter that never starts

Worked: MOD-5 up-counter with D flip-flops

```
present   next     required D inputs
Q2Q1Q0 -> Q2Q1Q0   D2 D1 D0
0 0 0     0 0 1     0  0  1
0 0 1     0 1 0     0  1  0
0 1 0     0 1 1     0  1  1
0 1 1     1 0 0     1  0  0
1 0 0     0 0 0     0  0  0
1 0 1  \
1 1 0   >  unused -> treat as don't-care X
1 1 1  /
                          (D = Qnext, so the two right
K-map each column:         blocks are the SAME numbers)

  D2 = Q1.Q0
  D1 = Q1 ^ Q0
  D0 = Q2'.Q0'
```

Check the unused states — always. Feed each one through the equations: `101 → 010`, `110 → 010`, `111 → 100`
Every unused state re-enters the legal sequence, so this counter is **self-starting**. Had one of them looped back onto itself or onto another unused state, the counter could power up in a **dead cycle** and never count — a real and frequently-examined failure.

27.4 Non-power-of-two moduli HIGH

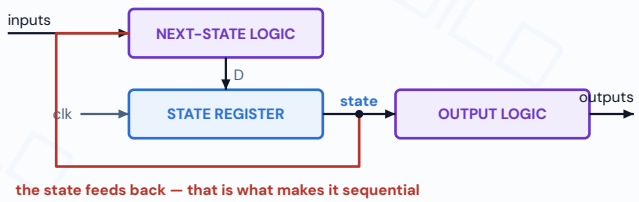
Full design	Do §27.3 with the exact sequence. Best result; more work.
Truncate with a decoder	Detect the terminal count and **synchronously clear or load**. For MOD-10, detect `1001` and load 0 on the next edge.
Async-clear truncation	Detect **M** (e.g. `1010`) and drive the **asynchronous** clear. **Works, but the counter momentarily displays the illegal count** and produces a runt pulse. **Prefer synchronous clear.**
Up/down	Add a direction input: each stage's toggle term becomes `(UP·Qlower…) + (DOWN·Q'lower…)`. A down-counter simply ANDs the **complements** of the lower bits.

Frequency division: a MOD-*M* counter divides the clock by **M**. The MSB of a MOD-2^N binary counter is a clean ÷2^N square wave, but the terminal-count output of an arbitrary MOD-*M* counter is a **narrow pulse**, not a 50 % square wave. For a 50 % duty odd divide you need a dedicated (half-cycle) divider.

// §28 FINITE STATE MACHINES

28.1 The model MUST

An FSM is a sequential circuit described by a finite set of **states**, a set of **inputs**, a **next-state function**, and an **output function**. Every controller — a protocol engine, a bus arbiter, a traffic light, a vending machine — is one.



the state feeds back — that is what makes it sequential

A **Mealy** machine additionally routes the inputs into the output logic — which is why its output responds in the same cycle, and why it can glitch.

28.2 Moore vs Mealy MUST

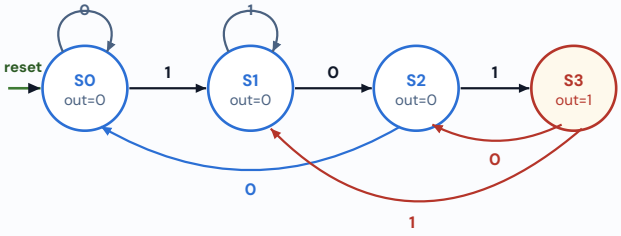
	Moore	Mealy
Output	`output = f(state)`	`output = f(state, input)`
Drawn	output written **inside** the state bubble	output written **on the arrow** as `in / out`
States needed	usually **more**	usually **fewer**

	Moore	Mealy
Reacts	one clock **later**	**same cycle** as the input
Output timing	**glitch-free**, changes only after a clock edge — safe to use directly	follows the input **asynchronously**; can glitch, and can create a combinational path between modules
Best for	outputs that drive other clocked logic, external strobes, anything used as an enable	low-latency response, fewer flops

The one-line answer: Moore output depends only on the state; Mealy depends on the state and the current input. A Mealy machine is faster to respond and smaller; a Moore machine has cleaner, registered outputs. You can always convert Mealy → Moore by adding states (splitting each state by its output), which is exactly why Moore needs more of them.

28.3 Worked: "101" sequence detector, overlapping MUST

```
MOORE - 4 states, output labels the STATE
  S0 idle/0    S1 saw "1"   S2 saw "10"   S3 saw "101" out=1

   state | in=0 | in=1 | out      1     0     1
   ------+------+------+----   S0 -> S1 -> S2 -> S3  out=1
    S0   |  S0  |  S1  |  0
    S1   |  S2  |  S1  |  0     from S3 the trailing 1 may
    S2   |  S0  |  S3  |  0     start a NEW match, so
    S3   |  S2  |  S1  |  1     S3 -1-> S1 and S3 -0-> S2

MEALY - 3 states, output labels the TRANSITION
   state | in=0      | in=1
   ------+-----------+-----------
    S0   | S0 / 0    | S1 / 0
    S1   | S2 / 0    | S1 / 0
    S2   | S0 / 0    | S1 / 1     <=== output asserted HERE

 input   1  0  1  1  0  1
 Moore  ..0  0  1  0  0  1   (one cycle after the last 1)
 Mealy  ..0  0  1  0  0  1   (in the SAME cycle as it)
```



The Moore machine for `101`, overlapping. After S3 the trailing `1` may begin a new match, so S3 goes to **S1** on a 1 and to **S2** on a 0 — never back to S0.

Overlapping vs non-overlapping. "Overlapping" means the tail of one match may begin the next — that is why S3/S2 goes back to **S1** on a 1, not to S0. A **non-overlapping** detector returns to **S0** after every detection. Read the question carefully: the two answers differ by exactly those arrows.

28.4 Design procedure MUST

one state per distinct piece of history the output depends on

How many states do I need? One per *distinct piece of history* the output depends on — not one per input value. A detector for an n-bit pattern needs at most n+1 states (Moore) or n (Mealy). Two states are **equivalent** (and can be merged) if they give the same output and go to equivalent states for **every** input.

28.5 State encoding MUST

Encoding	FFs for n states	Next-state logic	Use when
Binary	$\lceil \log_2 n \rceil$	most complex, deepest	flip-flops are scarce `ASIC`
Gray	$\lceil \log_2 n \rceil$	similar to binary	only one bit changes per step → less switching noise; useful if the state bits leave the domain
One-hot	**n**	**simplest and fastest** — each next-state term is a small OR; decode is a single bit	the usual `FPGA` default: flip-flops are plentiful, LUT depth is what costs f_{max}

[TRAP] Illegal states. One-hot with n flops has `2^n - n` illegal codes; a binary MOD-5 FSM in 3 bits has 3. If noise, a soft error or a reset glitch lands the machine in one, it may **never return**. A **safe FSM** adds a `default` branch that forces a return to the reset state. In `FPGA` flows the tool has an explicit "safe FSM" option that inserts this recovery logic.

28.6 FSM → RTL, the structural bridge HIGH

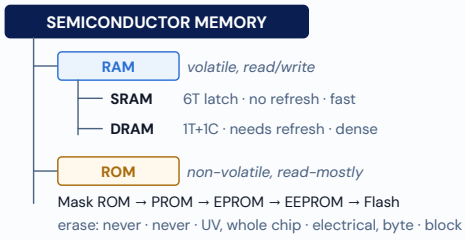
```
// The three blocks map 1:1 onto the three parts of the model.
// 1. STATE REGISTER - sequential, the only clocked block
always_ff @(posedge clk or negedge rst_n)
  if (!rst_n) state <= S0;
  else        state <= next_state;

// 2. NEXT-STATE LOGIC - purely combinational
always_comb begin
  next_state = state;              // default: hold
  case (state)
    S0: if (din) next_state = S1;
    S1: next_state = din ? S1 : S2;
    S2: next_state = din ? S3 : S0;
    S3: next_state = din ? S1 : S2;
    default: next_state = S0;      // safe FSM recovery
  endcase
end

// 3. OUTPUT LOGIC
assign y_moore = (state == S3);            // state only
assign y_mealy = (state == S2) && din;     // state AND input
```

Read the two output lines again — they *are* the Moore/Mealy definition in code. The Moore output is a function of a register, so it is glitch-free and one cycle late. The Mealy output contains `din`, so it responds immediately and inherits every glitch on `din`.

Why the `default:` in the combinational block matters twice. It gives the FSM its illegal-state recovery, *and* it makes the block fully specified — an incompletely assigned combinational block infers a **latch** (§19.1), which is the most common RTL bug there is. Assign a default to every output at the top of the block.

// §29 MEMORY — ROM & RAM

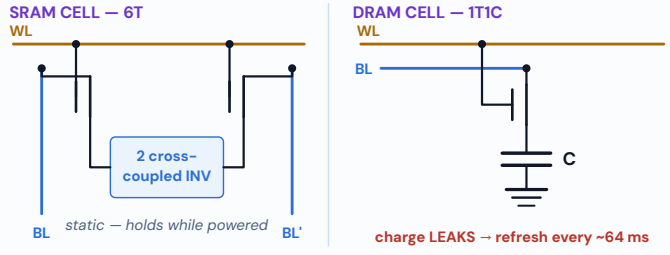
29.1 The family tree HIGH



Device	Written by	Erased by	Granularity	Typical use
Mask ROM	the foundry	never	—	huge-volume fixed code
PROM	user, **once**	never	—	one-time config
EPROM	programmer	UV light	whole chip	legacy; needs a quartz window
EEPROM	in-circuit	electrical	**byte**	calibration, small config
Flash	in-circuit	electrical	**block/sector**	firmware, SSD, SD card

Trap: Flash cannot rewrite a byte in place — it can only clear bits **1 → 0**; restoring a 0 to 1 needs a whole-block **erase**. That block granularity, plus finite erase endurance, is why flash needs wear levelling and a controller. EEPROM erases per byte, which is exactly why it survives for small parameter storage.

29.2 SRAM vs DRAM MUST



	SRAM	DRAM
Storage	cross-coupled inverters (a latch)	charge on a capacitor
Transistors/bit	**6** (sometimes 4T+2R)	**1** + 1 capacitor
Refresh	**No** — static while powered	**Yes** — the charge leaks away
Speed	**Fast** (~ns); no sense delay	Slower; needs sense amps + row activation
Density / cost per bit	Low density, expensive	**High density, cheap**
Read	non-destructive	**destructive** — the cell must be written back after every read

	SRAM	DRAM
Interface	simple SRAM bus; address in one go	multiplexed row/column address (RAS/CAS)
Used for	**caches, registers, FPGA block RAM**	**main memory** (DDR)

Both are volatile. The names mislead: "static" and "dynamic" refer to **whether the cell needs refreshing**, not to whether it survives power-off. Neither does. A typical DRAM row must be refreshed within ~64 ms; DDR issues a refresh command roughly every 7.8 µs to walk through all rows.

// §30 MEMORY ORGANISATION & DECODING

30.1 The size formulas MUST

[FORMULA]

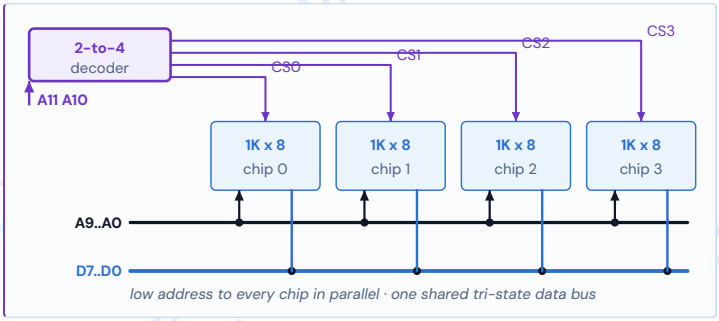
```
Memory size = (number of locations) × (bits per location)
Address lines = log2(number of locations)
Data lines = bits per location
Number of chips = (required capacity) ÷ (capacity of one chip)
```

```
"1K x 8"  means      1024 locations, 8 bits each
                     = 8192 bits total  (1 KB)
                     address lines = log2(1024) = 10  (A9..A0)
                     data lines    = 8               (D7..D0)

  4K x 8   -> 12 address lines, 8 data lines,  32768 bits
 16K x 16  -> 14 address lines, 16 data lines, 262144 bits
  1M x 1   -> 20 address lines, 1 data line
```

Trap — capacity vs address lines. The *width* (8, 16, 32) never affects the address-line count; only the **number of locations** does. And `1K` here is **1024**, not 1000. A part quoted as "64 Kbit" organised as `8K × 8` needs **13** address lines, not 16.

30.2 Expanding memory HIGH

WORD expansion (more locations)	BIT expansion (wider words)
e.g. 4K×8 from four 1K×8 chips	e.g. 1K×8 from two 1K×4 chips
Low address bits → **all** chips in parallel	The **same** address to all chips
High address bits → a **decoder** → one **chip select** each	All chip selects tied **together** (always both active)
Data buses tied together (chips are tri-state)	Data buses **concatenated**: D7–D4 and D3–D0



low address to every chip in parallel · one shared tri-state data bus

Chip select (CS)	Enables one device onto a shared bus. Without decoding, two chips would drive together — **bus contention** (§16.1).
OE / WE	**Output enable** gates the data drivers for a read; **write enable** latches the bus into the array. Both usually active-low.
Address decoding	**Full** decoding uses every high address bit — each location has one unique address. **Partial** decoding ignores some bits, which is cheaper but makes the device appear at several **aliased** addresses.

30.3 ROM as a logic device HIGH

A `2^n × m` ROM implements **any m Boolean functions of n variables** — just store the truth table. Internally it is a **fixed AND array** (a full n→2^n decoder generating every minterm) feeding a **programmable OR array**. No minimisation is needed or possible: cost depends only on *n* and *m*, never on how complex the functions are.

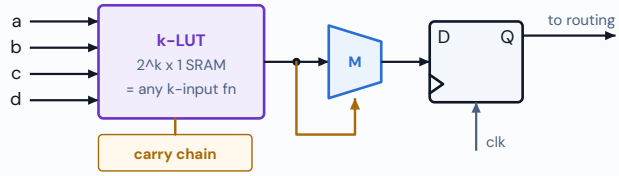
// §31 PLD, FPGA & ASIC

31.1 The programmable-array family HIGH

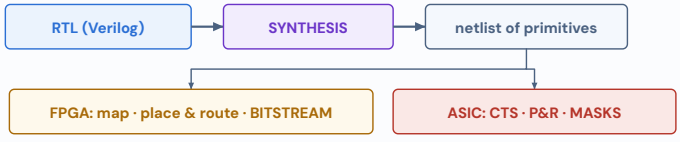
Device	AND array	OR array	Consequence
PROM	fixed (full decoder)	**programmable**	all minterms available; wasteful for sparse functions
PAL	**programmable**	fixed	cheap and fast; each output has a fixed number of product terms
PLA	**programmable**	**programmable**	most flexible, product terms shared between outputs; slowest & costliest

Mnemonic: in **PAL** the **A**ND is programmable; in **PROM** the **OR** (the memory array) is; a **PLA** has both. **CPLD** = many PAL-like blocks plus a global interconnect — predictable timing, non-volatile, instant-on. **FPGA** = a fine-grained LUT fabric, far larger, usually SRAM-based so it loads a bitstream at power-up.

31.2 What is inside an FPGA HIGH



LUT	The truth table **is** the implementation. Complexity is irrelevant — only the **number of inputs** matters, which is why one-hot FSM encoding suits FPGAs (§28.5).
CLB / slice	A cluster of LUTs + flip-flops + carry chain + wide-function MUXes.
Routing	A programmable interconnect. On a full device, routing — not logic — dominates both area and delay.
Block RAM	Dedicated dual-port **SRAM** blocks. Far more efficient than building memory from LUTs (distributed RAM).
DSP slice	Hardened multiply-accumulate. Uses far less area and runs much faster than a LUT-built multiplier.
Also on die	Clock managers (PLL/MMCM), high-speed I/O and SerDes, sometimes hard CPU cores.



31.3 FPGA vs ASIC MUST

	FPGA	ASIC
Configurability	**Reprogrammable** in seconds	Fixed at fabrication
NRE / upfront cost	**Low** — buy the part	**Very high** — mask set + engineering
Unit cost at volume	Higher per unit	**Much lower** at high volume
Time to first hardware	Days to weeks	Many months
Speed / power / area	Generally worse — the programmable routing costs delay and leakage	**Optimised** for the one function
A bug after shipping	Field-upgradable bitstream	Re-spin: new masks, big money
Best for	prototyping, low/medium volume, evolving standards, emulation	high volume, hard power/performance targets

Comparisons are qualitative and shift with process node, device family and design size — treat the direction as reliable and any specific multiplier as design-dependent. **Structured ASIC** and **eFPGA** sit between the two.

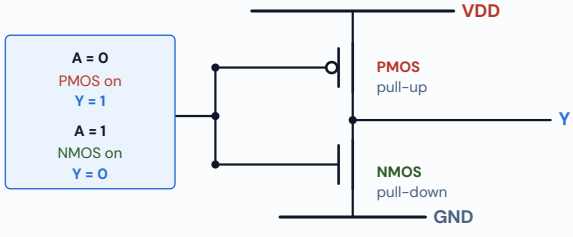
// §32 CMOS DIGITAL BASICS

32.1 The two switches MUST

Device	Conducts when gate is…	Passes well	Passes badly
NMOS	**HIGH** (1)	a strong **0**	a weak 1 — only up to `VDD-Vth`
PMOS	**LOW** (0) — hence the bubble	a strong **1**	a weak 0 — only down to `|Vth|`

That table dictates the whole topology: PMOS devices go in the **pull-up network** (to V_{DD}, where a strong 1 is needed) and NMOS devices in the **pull-down network** (to GND). Getting this backwards is why "why not build the inverter the other way round?" has a real answer: you would get degraded levels at both rails.

32.2 The CMOS inverter MUST



Output levels are rail-to-rail — a full `0 V` or `VDD`, with no resistive divider — which is why CMOS noise margins are so large (§5.2) and why they scale with the supply.

32.3 Building any gate: PUN / PDN duality MUST

```
STATIC CMOS GATE = PULL-UP NETWORK  (PMOS, to VDD)
                 + PULL-DOWN NETWORK (NMOS, to GND)
The two networks are DUALS: series in one <-> parallel in
the other, so exactly one of them conducts for any input.

 gate     PMOS pull-up     NMOS pull-down    transistors
 -------  ---------------  ---------------   -----------
 INV      1                1                   2
 NAND-2   2 in PARALLEL    2 in SERIES         4
 NOR-2    2 in SERIES      2 in PARALLEL       4
 NAND-n   n in PARALLEL    n in SERIES        2n
 AND-2    = NAND-2 then an INVERTER            6
 AOI21    (AB+C)' in ONE stage                 6
```

[FORMULA] A static CMOS gate with *n* inputs uses `2n` transistors. Note the consequence: **AND and OR cost MORE than NAND and NOR**, because they are the inverting gate plus an inverter. Static CMOS is **naturally inverting** — there is no 2-transistor non-inverting gate.

Why designers prefer NAND `CMOS` `MUST`: electron mobility is roughly 2–3× hole mobility, so an NMOS is intrinsically stronger than a PMOS of the same size. **NAND** puts the weak PMOS devices in **parallel** (each can stay small) and the strong NMOS devices in series. **NOR** does the opposite — stacking already-weak PMOS devices in series, so they must be made very wide to keep the rise time, costing area, input capacitance and speed. Hence NAND-dominated libraries.

Pass transistor & transmission gate `GOOD`: a lone NMOS pass transistor delivers a degraded 1 and a lone PMOS a degraded 0. A **transmission gate** — one NMOS and one PMOS in **parallel** with complementary gate signals — passes a full rail in both directions. It is the standard building block for compact MUXes and latches, but it is **not restoring**, so it cannot be cascaded indefinitely without a buffer.

// §33 DIGITAL POWER

33.1 Where the power goes MUST

[FORMULA] Total power

`Ptotal = Pdynamic + Pshort-circuit + Pstatic`

`Pdynamic = α · CL · VDD^2 · f`

Convention used here: `α` is the average number of **0→1** transitions per clock cycle (a free-running clock has α = 1; random data ≈ 0.5). Texts that count *every* transition instead write `P = ½ α C V^2 f` — the same physics, a different α.

Term	What it is	Lever to reduce it
`α` activity factor	how often the node actually switches	clock gating, operand isolation, avoid glitchy logic
`CL` load capacitance	gate + wire capacitance driven	smaller cells, shorter routes, fewer loads
`VDD^2` supply	**quadratic** — the strongest lever by far	voltage scaling, DVFS, multiple voltage domains
`f` frequency	linear	lower clock, or clock only what is needed
Short-circuit	both networks conduct briefly *during* an input transition	keep input **edges fast**; it grows with slow slew
Static / leakage	subthreshold + gate tunnelling current, flowing even when idle	power gating, multi-V_{th} cells, body bias

[TRAP] Halving the clock does not halve the energy per operation. It halves **power**, but the same job then takes twice as long, so the **energy** is unchanged — and leakage, which is time-based, actually goes *up*. Lowering V_{DD} is what reduces energy, because energy per transition is `C·V^2`. This is the reasoning behind DVFS and behind "race to idle".

Why leakage grew up: scaling V_{DD} forces V_{th} down to keep the drive current, and subthreshold leakage rises **exponentially** as V_{th} falls. In deep-submicron nodes static power can rival dynamic power, which is why power gating and dark silicon exist. Figure of merit: **power-delay product** (energy per operation).

// §34 FAN-IN, FAN-OUT & LOGIC FAMILIES

34.1 Fan-in and fan-out MUST

Fan-in	The number of **inputs to one gate**. High fan-in means more transistors in series, so the gate is **slower and larger** — which is why synthesis breaks a 16-input AND into a tree of small gates.
Fan-out	The number of gate **inputs one output drives** while still meeting its specified levels and timing.
DC limit `CMOS`	Essentially unlimited — a CMOS gate input draws almost no DC current. **Not** the real constraint.
AC limit `CMOS`	**The real one.** Every load adds capacitance, so `tpd` and the rise/fall times grow roughly **linearly** with fan-out. Overloading a net is a timing failure, not a logic failure.
DC limit `GEN` **TTL**	Current-driven: `N = min( IOL/IIL , IOH/IIH )`, typically 10–20.
Fix	**Buffer / repeater insertion** — drive a tree of buffers rather than one huge fan-out net. Tools do this automatically during synthesis and CTS.

Trap — do not confuse them. Fan-**IN** counts what arrives *at* a gate; fan-**OUT** counts what a gate *drives*. Fan-in is fixed by the logic function; fan-out is a load you can reduce by buffering.

34.2 Logic families GOOD

	TTL (74LS)	CMOS (74HC)	ECL `BONUS`
Static power	high (always drawing)	**≈ 0**	**very high** (always on)
Dynamic power	weakly frequency-dependent	rises with **f** (§33)	flat
Speed	moderate	comparable to fast TTL and now much better	**fastest** — transistors never saturate
Noise margin (5 V)	≈ 0.3–0.7 V	**≈ 1.4 V** (scales with V_{DD})	small — swing is only ~0.8 V
Fan-out	~10–20 (current-limited)	very high DC, **capacitance-limited**	high
Unused inputs	float **high** — still tie them off	**must** be tied; floating = shoot-through	tie off

74LS/74HC are chosen here because they are the families exam questions quote. Modern board-level logic is LVCMOS/LVTTL at 3.3 V, 2.5 V or 1.8 V, with LVDS for high-speed links; on-chip, everything is static CMOS. The **direction** of every comparison above still holds.

Interfacing rule: a driver's `VOH` must exceed the receiver's `VIH`, and its `VOL` must sit below `VIL` (§5.2). TTL drives 5 V CMOS badly because TTL's `VOH` (≈2.7 V) is below 5 V CMOS's `VIH` (≈3.5 V) — the classic fix is a pull-up resistor, or an HCT-series part whose inputs use TTL thresholds.

// §35 ADC / DAC & SAMPLING

35.1 Resolution and step size MUST

[FORMULA] For an N-bit converter

Number of distinct codes / levels = `2^N`
Step size (1 LSB) = `Δ = FSR / 2^N` where FSR is the full-scale *range*
Fractional resolution = `1 / 2^N` · Quantisation error = `±Δ/2` (rounding quantiser)
RMS quantisation noise = `Δ / √12` · `SQNR ≈ 6.02 N + 1.76 dB` (full-scale sine)

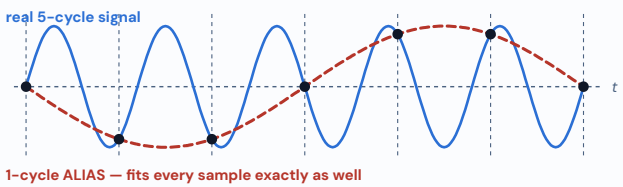
```
8-bit ADC, FSR = 0 .. 5.00 V (so Vref = 5 V)
  levels = 2^8 = 256          codes 0 .. 255
  1 LSB  = 5.00 / 256 = 19.53 mV
  max quantisation error = +/- 9.77 mV
  SQNR   = 6.02(8) + 1.76 = 49.9 dB

12-bit, FSR 3.3 V : 1 LSB = 3.3/4096 = 0.806 mV
16-bit, FSR 3.3 V : 1 LSB = 3.3/65536 = 50.4 uV
```

Convention — read this before answering an exam question. This sheet uses `Δ = FSR / 2^N`, the usual ADC definition: the input span is divided into 2^N equal bins, and the largest DAC output sits **one LSB below** V_{ref}. Some texts instead define a DAC's step as `Vref / (2^N - 1)`, which is right when the **maximum code is defined to output exactly V_{ref}**. The two differ by one part in 2^N — always state which you are using.

Trap: resolution is **not** accuracy. An N-bit converter resolves `2^N` steps, but offset, gain, INL and DNL errors decide how close a reading is to the truth. A 16-bit ADC with 4 LSB of INL is not a 16-bit-accurate instrument. **ENOB** (effective number of bits) is the honest figure: `ENOB = (SINAD - 1.76) / 6.02`.

35.2 Sampling, Nyquist and aliasing MUST

[FORMULA] Nyquist condition: `fs > 2 fmax` — the sample rate must exceed **twice the highest frequency present** in the signal, for that bandlimited signal to be reconstructable. `fs/2` is the **Nyquist frequency**.



Once sampled the two are **indistinguishable** — no later processing can separate them, because the information is already gone.

```
A component at f > fs/2 does not disappear - it FOLDS BACK
and appears at  |f - k.fs|  for the nearest multiple k.

fs = 1000 Hz   ->  Nyquist limit = 500 Hz
  input  300 Hz -> reads 300 Hz                 correct
  input  700 Hz -> reads |700-1000|  = 300 Hz  ALIAS!
  input 1300 Hz -> reads |1300-1000| = 300 Hz  ALIAS!
```

Anti-aliasing filter: an analogue **low-pass filter placed BEFORE the sampler**, which removes energy above `fs/2` while it can still be removed. It must be analogue and it must be first — a digital filter after the ADC is far too late. **Oversampling** (sampling well above Nyquist) lets you use a gentler, cheaper analogue filter and pushes the real work into the digital domain; it is the core idea behind sigma-delta converters.

Sample & hold	Freezes the input while the ADC converts, so the value cannot move mid-conversion. **Aperture jitter** in this stage limits high-frequency performance.
Conversion time	How long one sample takes. **Throughput** = 1/(conversion + acquisition time) — not always simply 1/conversion time.

35.3 Converter architectures — one line each GOOD

ADC	How	Speed	Bits	Cost of it
Flash	`2^N-1` comparators, all at once	fastest	~6–8	area and power double per bit
SAR	binary search, 1 bit per clock	medium	8–18	N clocks per sample
Pipeline	flash stages + residue amplification	fast	10–16	latency of several samples
Sigma-delta	heavy oversampling + noise shaping	slow	16–24	low bandwidth, digital filter needed
Dual-slope	integrate up, then down, count	very slow	high	excellent noise rejection — DMMs

DAC	How it works	Note
Binary-weighted	one resistor per bit, values `R, 2R, 4R…`	needs a huge, precisely matched resistor spread — impractical past a few bits
R-2R ladder	only **two** resistor values, repeated	easy to match on-chip; the standard structure
Current steering	switched current sources summed	fast; used for video and RF

35.4 Digital interface basics GOOD

	UART	SPI	I²C
Clock	**None** — asynchronous; both ends must agree a baud rate	Shared `SCLK` from the master	Shared `SCL` from the master
Wires	2 (TX, RX)	**4** + one `CS_n` per slave	**2** (SDA, SCL)
Framing	start bit, data, optional parity, stop bit	none — pure shift register	start, 7/10-bit address, R/W, ACK per byte
Duplex	full	**full** (MOSI + MISO)	half
Speed	low	**highest** (tens of MHz)	100 k / 400 k / 1 M+
Electrical	push-pull	push-pull	**open-drain + pull-ups** (§16.1)
Addressing	point-to-point	by chip select	by **address on the bus**, multi-master capable

The digital-logic connection: a UART receiver is a **SIPO shift register** plus an FSM that finds the start bit and samples each bit at its centre; a transmitter is a **PISO** shift register plus a baud-rate counter. Every one of these interfaces is just §26 and §28 wired together — which is exactly why interviewers ask you to sketch one.

// §36 HIGH-PROBABILITY OA TRAPS — THE DISCRIMINATORS

Every row is a pair an online assessment will try to make you confuse. Learn the **right-hand column**: one sentence that separates them.

Confusable pair	The one-line discriminator
XOR vs **OR**	They differ on **one row only**: inputs `1,1`. OR → 1, XOR → 0. XOR = "inputs **differ**".
NAND/NOR universality	**Both** are universal **alone**. NOT, AND, OR are **not** universal individually; {AND,NOT} and {OR,NOT} are.
Carry vs **signed overflow**	**C** = the unsigned result did not fit. **V** = the signed result did not fit. Independent: `V = Cin(msb) ⊕ Cout(msb)`.
Latch vs **flip-flop**	Latch = **level**-sensitive and transparent. Flip-flop = **edge**-triggered and opaque between edges.
Sync vs **async counter**	Synchronous = one clock to **every** flop, constant delay. Async/ripple = each flop clocks the next, delay **accumulates** and decodes glitch.
Moore vs **Mealy**	Moore output = `f(state)` only. Mealy = `f(state, input)`. Mealy reacts **one cycle earlier** and needs **fewer states**.
Setup vs **hold**	Setup = stable **before** the edge, fixed by a **slower clock**. Hold = stable **after** the edge, **NOT** fixable by a slower clock.
Propagation vs **contamination**	`tpd` = **max**, longest path, checks **setup**. `tcd` = **min**, shortest path, checks **hold**. Always `tcd ≤ tpd`.
Encoder vs **decoder**	Decoder: **n** → 2^n, one-hot out (expands). Encoder: 2^n → **n**, compresses. A **priority** encoder additionally resolves multiple active inputs.
MUX vs **DEMUX**	MUX: **many** → **one**, selects (data *in*puts + select). DEMUX: **one** → **many**, distributes. A DEMUX with data tied high **is** a decoder.
SRAM vs **DRAM**	SRAM: 6T latch, **no refresh**, fast, big, caches. DRAM: 1T+1C, **needs refresh**, dense, cheap, main memory. **Both volatile.**
Binary vs **Gray**	Gray changes **exactly one bit** per step and is **unweighted** — you cannot do arithmetic on it or read place values.
Combinational vs **sequential**	Look for a **feedback path**. Feedback = memory = sequential, whether or not you intended it.
Active-high vs **active-low**	`reset_n` / a bubble means asserted at **0**. `if (reset_n)` means **NOT** in reset.
Sync vs **async reset**	Sync needs a running clock and filters glitches. Async acts immediately but its **de-assertion** must meet **recovery/removal**.
Race-around	Only in a **level-triggered** JK with `J=K=1`. An edge-triggered JK does **not** have it.
Skew vs **jitter**	Skew = **static, spatial** (flop to flop). Jitter = **dynamic, temporal** (edge to edge, same flop).
Fan-in vs **fan-out**	Fan-**in** = inputs *to* a gate. Fan-**out** = loads a gate *drives*. In CMOS, fan-out is limited by **capacitance**, not current.
Minterm vs **maxterm**	Minterm = **product**, true for one row (1 → plain). Maxterm = **sum**, false for one row (0 → plain). `Mi = (mi)'`.
PI vs **essential PI**	PI = a group that **cannot grow**. EPI = a PI that **uniquely** covers at least one 1 — it must be in the answer.
Static-1 vs **static-0** hazard	Static-1: output should stay **1**, dips to 0 — lives in **SOP**. Static-0: should stay **0**, spikes to 1 — lives in **POS**.
Ring vs **Johnson**	Ring feeds back `Q` → **N** states. Johnson feeds back `Q'` → **2N** states. Neither is self-starting.
PROM / PAL / PLA	PROM: AND fixed, OR programmable. PAL: **AND** programmable, OR fixed. PLA: **both** programmable.
Resolution vs **accuracy**	N bits gives `2^N` **steps**; offset, gain, INL/DNL decide correctness. **ENOB** is the honest number.
FPGA vs **ASIC**	FPGA: reprogrammable, low NRE, higher unit cost. ASIC: fixed, huge NRE, better power/speed/area at volume.

// §37 "WHAT DOES THIS DO?" — CIRCUITS & WAVEFORMS

Q1 · What function is this MUX?

```
2:1 MUX, select = A :   D0 = B ,  D1 = B'
```

A **XOR.** `Y = A'·D0 + A·D1 = A'B + AB' = A ⊕ B`. Change the data inputs to `D0=0, D1=B` and you get AND; `D0=B, D1=1` gives OR. A 2:1 MUX plus one inverter covers every 2-variable function.

Q2 · A 4:1 MUX has sel = {A,B} and D0=0, D1=1, D2=1, D3=1. What is Y?

A `Y = A + B` — an OR gate. Only the select combination `00` reaches a 0. Reading a MUX this way is faster than expanding the algebra: the data inputs **are** the truth-table column.

Q3 · What counter is this, and how many states?

```
3 D flip-flops:  D0 = Q2' ,  D1 = Q0 ,  D2 = Q1
```

A **A 3-bit Johnson (twisted-ring) counter — 6 states** (`2N`). The `Q2'` feedback is the giveaway; a plain **ring** counter would use `D0 = Q2` and have only 3 states.
Sequence from `000`:
`000 → 100 → 110 → 111 → 011 → 001 → 000`.
The two unused states `010` and `101` form their own closed loop, so it is **not self-starting** — it must be reset.

Q4 · What sequence does this generate, and which state must you avoid?

```
3-bit shift register:  D0 = Q2 ^ Q1 ,  D1 = Q0 ,  D2 = Q1
```

A **A maximal-length LFSR.** From `001`:
`001→100→010→101→110→111→011→001` — **7 states = 2^3-1**, every non-zero value exactly once.
Avoid `000`: the XOR feedback produces 0, so the register locks up there for ever. Real designs either reset to a non-zero seed or add logic to force an escape.

Q5 · A 4-bit ripple counter is clocked at 16 MHz. What is on Q3, and when is it valid?

A **1 MHz** (`16 MHz / 2^4`), a clean 50 % square wave. But Q3 changes up to **$4 \cdot t_{pd}$** after the clock edge, and during a carry the outputs are momentarily **inconsistent** — `0111→1000` passes through `0110, 0100, 0000`. Never decode a ripple counter combinationally; resample it in the destination clock domain.

Q6 · Same D and clk into a D-latch and a D-flip-flop. Draw both outputs.

```
clk  __|----|____|----|____
D    ____|--------|________      (D rises mid-high-phase)
```

A **Latch:** transparent whenever `clk=1`, so Q follows D **immediately** at the moment D rises inside the high phase, then holds when clk falls.
Flip-flop: samples only at the rising edge. D was still 0 at that edge, so Q stays 0 until the **next** rising edge. The flip-flop's output is one full clock behind the latch — this is the whole latch/flop distinction on one picture.

Q7 · Is there an inferred latch here, and where is the critical path?

```
always_comb begin
  if (sel) y = a + b;      // no else branch
end
```

A **Yes — a latch on `y`.** When `sel = 0` the block assigns nothing, so `y` must **hold its previous value**, and synthesis can only do that with a level-sensitive latch. Fix: assign a default (`y = '0;`) at the top of the block, or add the `else`. **Critical path** here runs through the adder into whatever samples `y` — but with a latch in the way, static timing analysis can no longer check it as a simple edge-to-edge path.

Q8 · A 4-bit register has enable EN. What happens when EN = 0, and how should it be built?

A **The register holds** its value — clocks still arrive, but the D inputs are **recirculated from Q** through a 2:1 MUX. It must **not** be built by ANDing EN into the clock: a gated clock is a combinational signal on a clock pin, so it glitches, it breaks STA, and it can clock the register at the wrong instant.

Q9 · An 8-bit shift register holds `0000 0000`. Serial input is held at 1. What is the output after 5 clocks, and after 8?

A After 5 clocks: `1111 1000` (5 ones shifted in). After 8: `1111 1111`. A **SISO** serial output only starts producing the original data after **n** clocks — that latency of n is exactly why a SISO is used as a delay line.

Q10 · Where is the critical path, and what is f_{max}?

```
FF -> 4-input AND (0.4ns) -> XOR (0.3) -> OR (0.3) -> FF
tcq = 0.25 ns   tsu = 0.15 ns   skew = 0
```

A Critical path = the whole combinational chain: `0.4+0.3+0.3 = 1.0 ns`.
`Tmin = tcq + tpd + tsu = 0.25 + 1.0 + 0.15 = 1.40 ns` →
`fmax ≈ 714 MHz`.
Register between the XOR and OR and the longest stage becomes 0.7 ns →
`T = 1.10 ns`, **909 MHz** — at the price of one more cycle of latency.

// §38 TOP 40 INTERVIEW QUESTIONS

Answer in **2–4 sentences**, lead with the definition, then give the consequence. Where a question has a "but…", say it — that qualifier is usually what is being tested.

1. Combinational vs sequential?	Combinational output depends on the **present inputs only** — no memory, no feedback. Sequential output depends on inputs **and stored state**, so it needs feedback and (usually) a clock. Test: is there a feedback path?
2. Latch vs flip-flop?	A latch is **level-sensitive** and **transparent** while its enable is active, so glitches pass through. A flip-flop is **edge-triggered** and samples at an instant. The flop costs ~2× area (it is two latches) and buys a single clean timing check.
3. What are setup and hold time?	**Setup** = how long data must be stable **before** the clock edge; **hold** = how long after. Together they form the aperture window the flop needs quiet. Violating either risks **metastability**.
4. What is metastability?	When setup/hold is violated the flop's bistable loop is left near its unstable point, so Q sits at an invalid level for an **unbounded, probabilistic** time. It cannot be prevented when sampling a truly asynchronous signal — only made improbable, via **MTBF**.
5. Does a 2-FF synchroniser eliminate metastability?	**No.** FF1 still goes metastable just as often. It gives FF1 a **full clock period to resolve** before FF2 samples, raising MTBF exponentially — `MTBF = e^(tr/τ) / (T0·fclk·fdata)`.
6. Synchronous vs asynchronous reset?	Sync is sampled by the clock — glitch-immune but useless with the clock stopped, and it adds to the data path. Async acts immediately but its **de-assertion** must meet **recovery/removal**. Best practice: **assert async, de-assert synchronously**.
7. Can you fix a hold violation by slowing the clock?	**No.** Hold is a race between two paths launched by the **same** edge, so `Tclk` never appears in the equation. Fix it by **adding delay** to the data path or reducing clock skew. This is why hold failures found in silicon are fatal.

8. Write the setup and hold equations.	With `tskew` = capture-clock arrival minus launch-clock arrival:
	`Tclk ≥ tcq + tpd(max) + tsu - tskew`
	`tcq + tcd(min) ≥ th + tskew`

9. Propagation vs contamination delay?	`tpd` is the **maximum** — the longest path, used for the **setup** check. `tcd` is the **minimum** — the shortest path, used for **hold**. Always `tcd ≤ tpd`.
10. What is the critical path?	The register-to-register path with the **worst slack** — largest `tcq+tpd+tsu`. It alone sets f_{max}; optimising anything else changes nothing until the critical path moves.
11. Clock skew vs jitter?	Skew is a **static, spatial** difference in arrival between two flops (routing). Jitter is a **dynamic, temporal** variation of the same edge (PLL/supply noise). Positive skew helps setup and hurts hold; jitter only ever eats margin.
12. Why is synchronous design preferred?	One clock gives every path the same deadline, so timing reduces to inequalities a tool can verify **exhaustively**. Glitches settle between edges and are never sampled, and behaviour is repeatable across process, voltage and temperature.
13. What is CDC and why is it hard?	A signal sampled by a clock unrelated to the one that launched it will eventually violate setup/hold. Single-bit levels need a **2-FF synchroniser**; multi-bit buses need **Gray code**, a handshake, or an **async FIFO**.
14. Why can't a bus go through parallel 2-FF synchronisers?	Each bit resolves **independently**, so on the cycle the bus changes you can latch a combination that never existed — `0111→1000` can read as `1111`. This is the most common real CDC bug.
15. Why Gray-code an async FIFO pointer?	Only **one bit changes** per increment, so a pointer sampled mid-transition is either the old or the new value — never a wild one. That bounds the error to one count, which full/empty comparison can tolerate.
16. Moore vs Mealy?	Moore output = `f(state)`; Mealy = `f(state, input)`. Mealy responds in the **same cycle** and usually needs **fewer states**; Moore outputs are registered, so they are **glitch-free** and safer to drive other logic with.
17. How do you choose an FSM state encoding?	**Binary** minimises flops, **Gray** minimises simultaneous bit changes, **one-hot** minimises next-state logic depth. FPGAs usually default to one-hot because flops are plentiful and LUT depth costs f_{max}.
18. What is a safe FSM?	One that recovers from illegal states. One-hot with n flops has `2^n-n` illegal codes; a `default` branch back to the reset state guarantees the machine cannot get stuck.
19. What causes an inferred latch in RTL?	An **incompletely assigned** combinational block — a missing `else` or an unlisted `case` branch. The signal must hold its old value, and only a latch does that. Fix: assign defaults at the top of the block.
20. Race-around condition?	In a **level-triggered** JK with `J=K=1`, the output toggles repeatedly while the enable is high, so the final state is unpredictable. Fixed by **edge triggering** or a **master-slave** structure. An edge-triggered JK does not have it.
21. What is a hazard, and how do you remove one?	The possibility of an unwanted transient from **unequal path delays**. A static-1 hazard in SOP is removed by adding the **redundant consensus term** that bridges two adjacent groups. Function hazards (2+ inputs changing) cannot be removed by any gate.
22. When do glitches actually matter?	Never in a purely synchronous path — they settle before the setup window. They matter when the signal drives something **edge- or level-sensitive**: a clock, an async reset, a latch enable, a write strobe, or a chip output.
23. Why are NAND and NOR universal?	Each can build NOT, AND and OR, which is a functionally complete set. NOT alone is not universal; {AND, NOT} together are. In CMOS, NAND/NOR are also the **natural** gates — AND and OR cost an extra inverter.
24. Why do CMOS libraries prefer NAND over NOR?	Electron mobility is ~2–3× hole mobility. NAND stacks the **strong NMOS** in series and puts the weak PMOS in **parallel**; NOR stacks already-weak PMOS in series, so they must be made very wide — more area, more capacitance, slower.
25. Explain the CMOS inverter.	PMOS pull-up to V_{DD}, NMOS pull-down to GND, gates tied together. Input 0 → PMOS on, output 1; input 1 → NMOS on, output 0. Exactly one conducts in steady state, so **static current is ~zero** and output swings rail to rail.
26. Give the dynamic power formula.	`Pdyn = α·CL·VDD^2·f`, with α the 0→1 activity per cycle. Total power adds short-circuit and leakage. **V_{DD} is quadratic**, which is why voltage scaling dominates every low-power strategy.
27. Does halving the clock halve energy?	**No** — it halves **power**, but the work takes twice as long so energy per operation is unchanged, and leakage (time-based) increases. Lowering **V_{DD}** is what reduces energy, since energy per transition is `C·V^2`.
28. Fan-in vs fan-out?	Fan-in = inputs **to** a gate (more inputs → more series devices → slower). Fan-out = loads a gate **drives**. In CMOS the DC limit is negligible; the real constraint is the added **capacitance**, which stretches the edges.
29. SRAM vs DRAM?	SRAM: 6T cross-coupled latch, **no refresh**, fast, low density — caches and FPGA block RAM. DRAM: 1 transistor + 1 capacitor, **refresh every ~64 ms**, destructive read, high density — main memory. **Both are volatile.**
30. A memory is "1K × 8". How many address and data lines?	1024 locations × 8 bits = 8192 bits. Address lines = `log2(1024) = 10`; data lines = **8**. Width never affects the address count — only the number of locations does.
31. How do you build 4K×8 from 1K×8 chips?	Four chips. `A9–A0` go to all of them in parallel; `A11, A10` feed a **2-to-4 decoder** whose outputs are the four **chip selects**. Data buses are tied together and the chips' outputs are tri-state.
32. PROM vs PAL vs PLA?	PROM: fixed AND (full decoder) + programmable OR. PAL: **programmable AND** + fixed OR. PLA: **both** programmable — most flexible, slowest and costliest.
33. What is a LUT?	A `2^k×1` SRAM addressed by the logic inputs, so a k-LUT implements **any** k-input function. Cost depends only on the input count, never on how complex the function is — which is why one-hot FSMs suit FPGAs.
34. FPGA vs ASIC?	FPGA: reprogrammable, low NRE, days to hardware, higher unit cost, worse power/area/speed. ASIC: fixed at fabrication, very high NRE, months of lead time, far better per-unit cost and efficiency at volume.

35. Why two's complement?	One representation of zero, and **a single adder performs both add and subtract** (`A-B = A + B' + 1`) with no sign comparison or correction. Range for N bits: `-2^(N-1) … 2^(N-1)-1`.
36. Carry-out vs overflow?	Carry says the **unsigned** result did not fit; overflow says the **signed** result did not. `V = Cin(msb) ⊕ Cout(msb)`, or "both operands same sign, result different". Adding opposite signs can never overflow.
37. Why is a carry look-ahead adder faster?	It computes each carry **directly from the inputs** — `Ci+1 = Gi + PiCi` unrolled into a 2-level function — so all carries appear together in ~3 gate delays instead of rippling. The price is fan-in and gate count growing quadratically, hence 4-bit blocks.
38. How do you implement a function with a MUX?	A `2^n:1` MUX implements **any n-variable** function — pour the truth-table column into the data inputs. It also implements any **(n+1)**-variable function by tying each data input to `0`, `1`, the leftover variable, or its complement.
39. What is a don't-care and what is the risk?	An input combination that cannot occur or whose output is unused; it may be treated as 0 or 1 to enlarge a K-map group. The risk: the synthesised gate produces a **definite** value for it, so if the "impossible" input ever happens you get whatever minimisation picked.
40. What does an ADC's resolution mean?	N bits gives `2^N` levels and a step of `FSR/2^N`, with quantisation error `±½ LSB` and `SQNR ≈ 6.02N + 1.76 dB`. Resolution is **not accuracy** — offset, gain and INL/DNL decide that, and **ENOB** reports the truth.

// §39 DIGITAL ELECTRONICS → RTL / VERILOG BRIDGE

The point of this table is the **mapping**, not the syntax: every block you drew in §11–§28 has one canonical RTL shape.

Digital concept	RTL form	Watch out for
AND / OR / NOT	`& | ~`	bitwise & vs logical &&
XOR / XNOR	`^ ~^`	^ is XOR, not "power"
Combinational logic	`assign`, or `always_comb`	assign a **default** to every output or you infer a latch
MUX	`y = sel ? a : b;` or a `case`	a full `case` avoids priority logic you did not want
Decoder	`case` / one-hot shift `1'b1 << sel`	cover every input value
Priority encoder	`if / else if` chain, or `casez`	the `if` chain **is** the priority — order matters
Adder	`{cout, sum} = a + b + cin;`	make the result one bit wider or the carry is lost
Flip-flop	`always_ff @(posedge clk)`	use non-blocking `<=` in every clocked block
Register with enable	`if (en) q <= d;`	**never** gate the clock to make an enable
Async reset	`@(posedge clk or negedge rst_n)`	de-assert it synchronously (§25)
Counter	`q <= (q == MAX) ? '0 : q + 1'b1;`	define the wrap explicitly, not by overflow
Shift register	`q <= {q[N-2:0], sin};`	check the direction of the concatenation
Toggle / T-FF	`q <= q ^ t;`	this *is* the T-FF characteristic equation
FSM	state register + `always_comb` next-state + output logic	always include `default:` — safe FSM and no latch
Moore output	`assign y = (state == S3);`	registered, glitch-free
Mealy output	`assign y = (state == S2) && din;`	combinational — inherits input glitches
2-FF synchroniser	two flops in series, no logic between	only the **second** flop's output may be used
Memory	an array indexed in a clocked block	the read/write style decides block RAM vs registers
Tri-state pad	`assign io = oe ? out : 1'bz;`	legal at I/O only — inside the chip, use a MUX

// §40 FORMULA BOX

```
RANGES      unsigned N-bit    0 .. 2^N - 1
            2's complement   -2^(N-1) .. 2^(N-1) - 1
            negate           invert all bits, add 1
            overflow  V = Cin(msb) ^ Cout(msb)
```

```
ADDERS      Sum  = A ^ B ^ Cin
            Cout = AB + Cin(A ^ B) = majority(A,B,Cin)
            CLA  P = A ^ B   G = A.B   C(i+1) = G + P.C
```

```
SELECT      2:1 MUX   Y = S'.D0 + S.D1
            n selects -> 2^n data inputs
            decoder   n inputs -> up to 2^n outputs
```

```
KMAP        group of 2^k cells -> (n - k) literals
            Sigma m(S) = Pi M(all indices not in S)
```

```
FLIP-FLOPS  D  : Qn = D          T  : Qn = T ^ Q
            JK : Qn = J.Q' + K'.Q
            SR : Qn = S + R'.Q     (with S.R = 0)
            excitation:  D = Qnext ,  T = Q ^ Qnext
```

```
TIMING      Tclk >= tcq + tpd(max) + tsu - tskew
            tcq + tcd(min) >= th + tskew
            fmax = 1 / Tclk(min)
            MTBF = e^(tr/tau) / (T0 . fclk . fdata)
```

```
COUNTERS    MOD = 2^N          N = ceil(log2 MOD)
            ring: N states     Johnson: 2N states
            ripple fmax ~ 1 / (N . tpd)
```

```
MEMORY      size = locations x width
            address lines = log2(locations)
            1K x 8 = 1024 x 8 = 8192 bits, 10 addr, 8 data
```

```
CMOS        P(dyn) = alpha . C . V^2 . f
            static gate, n inputs -> 2n transistors
            NM(H) = VOH - VIH    NM(L) = VIL - VOL
```

```
CONVERTERS  levels = 2^N       1 LSB = FSR / 2^N
            quantisation error = +/- LSB/2
            SQNR ~ 6.02 N + 1.76 dB
            Nyquist  fs > 2 . fmax
```

// §41 50 THINGS TO MEMORISE

1. NAND and NOR are universal on their own; NOT, AND and OR are not.
2. XOR is 1 when its inputs **differ**; an n-input XOR is 1 for an **odd** number of 1s.
3. An N-bit unsigned number ranges `0` to `2^N-1`.
4. An N-bit two's-complement number ranges `-2^(N-1)` to `2^(N-1)-1`.
5. Negate in two's complement by **inverting every bit and adding 1**.
6. `1111…1` is **−1**; `1000…0` is the most negative value and has **no positive partner**.
7. Carry-out means the **unsigned** result did not fit; `V = Cin(msb) ⊕ Cout(msb)` means the **signed** one did not.
8. Adding two numbers of **opposite sign can never overflow**.
9. Sign-extend two's complement by replicating the MSB; zero-extend unsigned.
10. `Sum = A⊕B⊕Cin`; `Cout = AB + Cin(A⊕B)` = majority of the three.
11. A full adder is **two half adders plus one OR gate**.
12. Ripple-carry delay grows as **O(n)**; carry look-ahead produces all carries in about **3 gate delays**.
13. `Pi = Ai⊕Bi`, `Gi = AiBi`, `Ci+1 = Gi + PiCi`.
14. A 2:1 MUX is `Y = S'D0 + SD1`; n select lines drive `2^n` data inputs.
15. A `2^n:1` MUX implements **any n-variable** function directly, and any **(n+1)**-variable function using `0/1/X/X'` on the data inputs.
16. A **DEMUX with its data input tied high is a decoder**.
17. A decoder with n inputs has up to `2^n` outputs, and each output is one **minterm**.
18. Most catalogue decoders have **active-LOW** outputs.
19. A **priority** encoder resolves several active inputs; a plain encoder produces garbage.
20. An encoder's output is meaningless unless its **valid** bit is set.
21. Equality is the **AND of bitwise XNORs**.
22. `G = B ⊕ (B>>1)`; back the other way, `B[i] = B[i+1] ⊕ G[i]`.
23. Gray code changes **exactly one bit** per step and is **unweighted** — never do arithmetic on it.
24. BCD codes `1010` – `1111` are invalid; a BCD adder **adds 6** to correct.
25. Excess-3 is BCD+3 and is **self-complementing**.
26. K-map rows and columns run **00, 01, 11, 10** — Gray order.
27. K-map groups are **powers of two**; diagonals are never adjacent; edges and corners **wrap**.
28. A group of `2^k` cells on an n-variable map leaves an `(n-k)`-literal term.
29. An **essential** prime implicant uniquely covers at least one 1, so it must appear in the answer.
30. `Σm(S) = ΠM(every index NOT in S)`.
31. A **latch is level-sensitive** and transparent; a **flip-flop is edge-triggered**.
32. **Setup** is measured before the sampling edge; **hold** after it.
33. A **hold violation cannot be fixed by slowing the clock** — `Tclk` is not in the equation.
34. `Tclk ≥ tcq + tpd(max) + tsu - tskew`.
35. `tcd ≤ tpd` always; `tpd` checks setup, `tcd` checks hold.
36. `D = Qnext`, and `T = Q ⊕ Qnext` — the two excitation rules worth memorising.
37. JK characteristic equation: `Qnext = J·Q' + K'·Q`.
38. Race-around affects **level-triggered** JK devices with `J=K=1` — not edge-triggered ones.
39. A 2-flop synchroniser **reduces the probability** that metastability propagates; it does not eliminate metastability.
40. **Never** pass a multi-bit bus through parallel 2-flop synchronisers — use Gray code, a handshake, or a FIFO.
41. A binary counter with n flip-flops has up to `2^n` states; `N = ceil(log2 M)`.
42. A **ring** counter with N flops has **N** states; a **Johnson** counter has **2N**.
43. Neither ring nor Johnson counters are **self-starting** — they must be initialised.
44. Ripple counters **glitch when decoded**, because their bits change at different times.
45. **Moore output = f(state)**; **Mealy = f(state, input)**.
46. A Mealy machine needs **fewer states** and reacts **one cycle earlier** than the equivalent Moore machine.
47. An **incompletely assigned combinational block infers a latch** — the commonest RTL bug.
48. SRAM needs **no refresh**; DRAM does, and its read is **destructive**. **Both are volatile.**
49. Memory size = locations × width; **address lines = $\log_2$(locations)**, and width never affects that count.

50. Dynamic CMOS power is `α·C·V^2·f`; a static CMOS gate with n inputs uses `2n` transistors; an N-bit converter has `2^N` levels and a `FSR/2^N` step.

// §42 30 COMMON TRAPS — TRAP → CORRECT RULE

The trap	The correct rule
"Carry set means the answer is wrong"	After a **signed** add read **V**; after an **unsigned** add read **C**.
"XOR is basically OR"	They differ at `1,1`: OR gives 1, XOR gives 0.
"A latch is just a slow flip-flop"	A latch is **transparent** for a whole level; a flop samples at an instant.
"Slow the clock to fix the hold violation"	Hold has **no** `Tclk` **term** — add data-path delay instead.
"The synchroniser removes metastability"	It only reduces the **probability of propagation**, raising MTBF.
"Moore and Mealy behave identically"	Mealy asserts **one cycle earlier** and is not glitch-free.
"Encoder and decoder are the same size"	Decoder is **n→2ⁿ**; encoder is **2ⁿ→n** and needs a valid bit.
"MUX and DEMUX are interchangeable"	Opposite directions: many→one vs one→many.
"SRAM is non-volatile"	**Both** SRAM and DRAM lose data at power-off.
"You can add two Gray-code numbers"	Gray is **unweighted** — convert to binary first.
"These diagonal K-map cells are adjacent"	They differ in **two** variables; only orthogonal neighbours merge.
"I grouped three adjacent 1s"	Groups are **powers of two**: use a pair plus an overlapping pair.
"Σm and ΠM use the same index list"	They are **complementary** sets over `0…2^n-1`.
"A don't-care stays undefined in hardware"	Synthesis picks a **definite** 0 or 1 for it.
"`reset_n` is asserted when it is 1"	Active-low asserts at **0**; `if (reset_n)` means **not** in reset.
"A floating CMOS input reads as 0"	It drifts through the forbidden band and causes shoot-through. Tie it off.
"A ripple counter's outputs are clean"	Bits change at different times, so decoded terms **glitch**.
"Ring and Johnson counters self-start"	Both have illegal loops and **must be initialised**.
"You need gates to implement logic with a MUX"	The **data inputs are the truth-table column**.
"NAND is universal, so NAND is always cheapest"	An OR from NANDs costs **3 gates and 2 levels**.
"AND is cheaper than NAND"	In CMOS, AND = NAND + inverter = **6 transistors** vs 4.
"Halving the clock halves the energy"	It halves **power**; energy per operation is unchanged. Lower V_{DD}.
"Add a buffer to remove the hazard"	Only a **redundant covering (consensus) term** removes a logic hazard.
"Gate the clock to build an enable"	**Recirculate through a MUX**; a gated clock glitches and breaks STA.
"A 2-FF sync will catch that pulse"	A pulse shorter than ~2 destination periods can vanish — use a **toggle synchroniser**.
"1K memory = 1000 locations"	`1K = 1024`, so **10** address lines.
"Wider data means more address lines"	Only the **number of locations** sets the address count.
"16-bit ADC means 16-bit accuracy"	Resolution ≠ accuracy — offset, gain, INL/DNL matter; quote **ENOB**.
"A digital filter can remove the alias"	The anti-aliasing filter must be **analogue and before** the sampler.
"All JK flip-flops suffer race-around"	Only **level-triggered** ones do.

// §43 NUMERICAL DRILL — 12 WORKED PROBLEMS

These are the arithmetic shapes an OA actually asks. Work each one before reading the answer; every formula used is in §40.

P1 · Convert 173 (decimal) to binary, hex and octal.

A `173 = 128+32+8+4+1` → `1010 1101`.
Hex: group in 4 → `1010|1101` = `0xAD`. Octal: group in 3 from the right → `010|101|101` = `0o255`. Check: `2·64+5·8+5 = 173`. ✓

P2 · Write −45 in 8-bit two's complement.

A `+45 = 0010 1101` → invert → `1101 0010` → add 1 → `1101 0011` = `0xD3`.
Verify by weights: `-128+64+16+2+1 = -45`. ✓

P3 · 8-bit: `0x7F + 0x01`. What are C and V?

A `0111 1111 + 0000 0001 = 1000 0000`.
C = 0 (nothing left the MSB) but the carry *into* the MSB was 1, so **V = 1 ⊕ 0 = 1**.
Unsigned `127+1 = 128` is **correct**; signed `127+1 = -128` is **wrong**. One addition, two different verdicts — exactly the §3.2 point.

P4 · Minimise `F(A,B,C,D) = Σm(0,2,5,7,8,10,13,15)`.

A Two groups of four: the **corners** m(0,2,8,10) → `B'D'`, and the centre column pair m(5,7,13,15) → `BD`.
`F = B'D' + BD = (B ⊕ D)'` — an **XNOR of B and D**. A and C drop out entirely. Eight minterms collapse to one gate.

P5 · A 4:1 MUX with A,B on the selects must realise `F = Σm(0,3,5,6)`. Give D0–D3.

A Pair the rows by C: `D0 = C'`, `D1 = C`, `D2 = C`, `D3 = C'`.
That function is `(A ⊕ B ⊕ C)'` — 1 for an **even** number of 1s, i.e. an even-parity checker.

P6 · How many flip-flops for a MOD-12 counter, and how many unused states?

A `N = ceil(log2 12) = 4` flip-flops (3 would only give 8 states).
Unused states = `2^4 - 12 = 4`. Every one of them must be checked to confirm the counter is self-starting (§27.3).

P7 · A 4-bit ripple counter uses flip-flops with tpd = 8 ns. Maximum clock?

A The carry must walk all four stages: `4 × 8 = 32 ns`.
`fmax ≈ 1/32 ns = 31.25 MHz`. A **synchronous** counter of the same width would run at `1/(tcq+tlogic+tsu)` — several times faster, and independent of width.

P8 · tcq = 0.40, tpd(max) = 3.20, tsu = 0.25 ns, skew = +0.15 ns. Find f_{max}.

A `Tclk ≥ 0.40 + 3.20 + 0.25 - 0.15 = 3.70 ns` → `fmax ≈ 270 MHz`.
The skew is **positive** (capture clock arrives later), so it **helps** setup — it subtracts. It will hurt the hold check instead.

P9 · Same path: tcd(min) = 0.20 ns, th = 0.30 ns, skew = +0.15 ns. Does hold pass?

A Need `tcq + tcd ≥ th + tskew` → `0.40 + 0.20 = 0.60` vs `0.30 + 0.15 = 0.45`.
Passes, with 0.15 ns of slack. Note `Tclk` never appeared — which is why no clock change could have rescued it had it failed.

P10 · Build 32K × 16 from 8K × 8 chips. How many chips, and how is the address split?

A **Bit** expansion: 2 chips side by side give 16 bits. **Word** expansion: `32K/8K = 4` banks. Total = `4 × 2 = 8 chips`.
Total address lines `log2(32768) = 15`: `A12–A0` (13 lines) go to every chip; `A14,A13` drive a **2-to-4 decoder** producing the four bank chip-selects.

P11 · C = 20 pF, VDD = 1.2 V, f = 500 MHz, α = 0.15. Find P_{dyn}. Then drop VDD to 0.9 V.

A `P = αCV^2f = 0.15 × 20p × 1.44 × 500M = 2.16 mW`.
At 0.9 V: `(0.9/1.2)^2 = 0.5625` → `1.22 mW`. A **25 % voltage cut buys a 44 % power cut** — the quadratic term is why DVFS exists.

P12 · A 10-bit ADC spans 0–3.3 V. Give the LSB, the worst quantisation error and the SQNR. Then: the signal reaches 22 kHz but you sample at 30 kHz — what do you see?

A `LSB = 3.3/2^(10) = 3.3/1024 = 3.22 mV`; error `= ±LSB/2 = ±1.61 mV`; `SQNR ≈ 6.02(10)+1.76 = 62 dB`.
Nyquist needs `fs > 44 kHz`. At 30 kHz the limit is 15 kHz, so the 22 kHz tone **aliases** to `|22 - 30| = 8 kHz` and is now indistinguishable from a real 8 kHz signal. Only an **analogue** filter before the sampler could have prevented it.

// §44 FINAL RAPID REVISION — LAST 30 MINUTES

1 · Foundations

```
NUMBERS → base r place value → dec/bin/oct/hex → 2's COMPLEMENT (invert+1) →
range -2^(N-1) .. 2^(N-1)-1 → sign-extend → CARRY ≠ OVERFLOW (V = Cin ^ Cout) →
CODES BCD (+6 fix) · Excess-3 · GRAY (1 bit changes) · parity →
LEVELS VOH/VIH/VIL/VOL → NM(H)=VOH-VIH · NM(L)=VIL-VOL → tpd · tr/tf →
GATES AND OR NOT NAND NOR XOR XNOR → NAND/NOR UNIVERSAL → bubble push →
BOOLEAN identity · null · absorption · redundancy · DE MORGAN · CONSENSUS
· duality →
SOP/POS minterm(1→plain) · maxterm(0→plain) · Σm = ΠM(rest) · don't-care
```

→
K-MAP Gray order 00 01 11 10 · powers of 2 · wrap · corners · PI · EPI

2 · Circuits and timing

ADDERS HA · FA = 2HA+OR · RCA O(n) · CLA (P=A^B, G=AB) · BCD adder →
ROUTING MUX (Y=S'D0+SD1, implements any function) · DEMUX=decoder · decoder=minterms ·
encoder vs PRIORITY ENCODER · comparator = AND of XNORs →
HAZARDS static-1 (SOP) · static-0 (POS) · dynamic · fix = redundant term →
SEQUENTIAL feedback = memory → LATCH (level) vs FLIP-FLOP (edge) → master-slave →
D · T (Q^T) · JK (JQ'+K'Q) · SR → EXCITATION D=Qn, T=Q^Qn → race-around →
TIMING tsu before · th after · tcq → Tclk ≥ tcq+tpd+tsu-tskew → fmax → slack →
tpd(max)=setup · tcd(min)=hold · HOLD IS NOT FIXED BY A SLOWER CLOCK · skew vs jitter →
METASTABILITY → MTBF = e^(tr/tau)/(T0·fclk·fdata) → 2-FF SYNC → CDC (Gray / handshake / FIFO) →
reset: async assert + sync de-assert · recovery/removal

3 · Structures and silicon

REGISTERS SISO SIPO PISO PIPO · LFSR → COUNTERS MOD=2^N · N=⌈log2 M⌉ · sync vs ripple · ring N · Johnson 2N · unused states → self-starting? →
FSM MOORE=f(state) · MEALY=f(state,input) · state diagram → table → encoding (one-hot on FPGA) →
MEMORY size = locations × width · addr = log2(locations) · SRAM(6T, no refresh) vs DRAM(1T1C, refresh) ·
ROM/PROM/EPROM/EEPROM/Flash · PROM/PAL/PLA · LUT · FPGA vs ASIC →
CMOS PMOS up / NMOS down · 2n transistors · NAND preferred · P = αCV²f · leakage · fan-in vs fan-out →
ADC/DAC 2^N levels · LSB = FSR/2^N · ±½LSB · SQNR ≈ 6.02N+1.76 · fs > 2fmax · anti-alias filter

If you have five minutes left, recite these six: two's complement range and `V = Cin ⊕ Cout` · K-map order `00 01 11 10` · latch is level, flop is edge · `Tclk ≥ tcq+tpd+tsu-tskew` and hold ignores `Tclk` · Moore = state only, Mealy = state + input · `P = αCV^2f`.

// §45 GLOSSARY — ABBREVIATIONS YOU WILL MEET

Abbr	Stands for	Abbr	Stands for
ALU	Arithmetic Logic Unit	**LUT**	Look-Up Table (FPGA logic cell)
ASIC	Application-Specific Integrated Circuit	**LVDS**	Low-Voltage Differential Signalling
BCD	Binary-Coded Decimal	**MSB / LSB**	Most / Least Significant Bit
BRAM	Block RAM — dedicated on-chip SRAM	**MTBF**	Mean Time Between Failures
CDC	Clock Domain Crossing	**MUX / DEMUX**	Multiplexer / Demultiplexer
CLA	Carry Look-Ahead adder	**NRE**	Non-Recurring Engineering cost
CLB	Configurable Logic Block	**OE / WE / CS**	Output Enable / Write Enable / Chip Select
CMOS	Complementary Metal-Oxide-Semiconductor	**PAL / PLA**	Programmable Array Logic / Programmable Logic Array
CPLD	Complex Programmable Logic Device	**PI / EPI**	Prime Implicant / Essential Prime Implicant
CTS	Clock Tree Synthesis	**PLL**	Phase-Locked Loop
DAC / ADC	Digital-to-Analogue / Analogue-to-Digital Converter	**PPA**	Power, Performance, Area
DFT	Design For Test	**PROM**	Programmable Read-Only Memory
DNL / INL	Differential / Integral Non-Linearity	**PUN / PDN**	Pull-Up Network / Pull-Down Network
DRAM	Dynamic Random-Access Memory	**PVT**	Process, Voltage, Temperature corners
DVFS	Dynamic Voltage and Frequency Scaling	**RCA**	Ripple-Carry Adder
ECL	Emitter-Coupled Logic	**RTL**	Register-Transfer Level
EDA	Electronic Design Automation	**SAR**	Successive-Approximation Register (ADC)
EEPROM	Electrically Erasable Programmable ROM	**SDC**	the standard timing-constraint file format
ENOB	Effective Number Of Bits	**SerDes**	Serialiser / Deserialiser
EPROM	UV-Erasable Programmable ROM	**SISO … PIPO**	Serial/Parallel In, Serial/Parallel Out
FIFO	First In, First Out queue	**SoC**	System on Chip
FPGA	Field-Programmable Gate Array	**SOP / POS**	Sum Of Products / Product Of Sums
FSM	Finite State Machine	**SQNR**	Signal-to-Quantisation-Noise Ratio
HDL	Hardware Description Language	**SRAM**	Static Random-Access Memory
Hi-Z	High impedance — the third, disconnected state	**STA**	Static Timing Analysis
JTAG	the boundary-scan / debug port standard	**TTL**	Transistor-Transistor Logic
LFSR	Linear-Feedback Shift Register	**UART**	Universal Asynchronous Receiver/Transmitter
MOD-N	a counter with N distinct states	**WNS / TNS**	Worst / Total Negative Slack

Symbol	Meaning	Symbol	Meaning
`tcq`	clock-to-Q delay of a flip-flop	`tskew`	clock arrival difference, capture - launch
`tsu` / `th`	setup / hold time	`VOH` `VOL`	output HIGH / LOW voltage of a driver
`tpd` / `tcd`	propagation (max) / contamination (min) delay	`VIH` `VIL`	input HIGH / LOW threshold of a receiver
`Σm` / `ΠM`	sum of minterms / product of maxterms	`α`	switching activity factor (§33)
`FSR`	full-scale range of a converter	`Δ`	one LSB step, `FSR/2^N`

// §46 STANDARD 74-SERIES PARTS BY FUNCTION

Exam questions name these parts without explaining them. Each one is a section of this sheet in a package.

Part	Function	See
`7400` / `7402` / `7404`	quad 2-input **NAND** / quad **NOR** / hex **inverter**	§6
`7408` / `7432` / `7486`	quad **AND** / quad **OR** / quad **XOR**	§6
`7483`	4-bit binary **full adder** with fast carry	§12
`7485`	4-bit **magnitude comparator**, cascadable	§14.3
`74138` / `74139`	3-to-8 **decoder** / dual 2-to-4 — **active-LOW** outputs	§14.1
`74147` / `74148`	10-to-4 and 8-to-3 **priority encoder**	§14.2
`74151` / `74153` / `74157`	8:1 **MUX** / dual 4:1 / quad 2:1	§13
`7447`	BCD to **7-segment** decoder / driver	§14.4
`7474` / `7476`	dual edge-triggered **D** flip-flop / dual **JK**	§19
`74161` / `74163`	synchronous 4-bit counter — **async** clear / **sync** clear	§27.4
`74190` / `74191`	synchronous **up/down** counter, BCD / binary	§27.4
`74164` / `74165`	8-bit **SIPO** / **PISO** shift register	§26.2
`74194`	4-bit **universal** bidirectional shift register	§26.2
`74373` / `74374`	octal **latch** / octal **flip-flop**, 3-state out	§19.1
`74245`	octal bus **transceiver**, 3-state — bidirectional bus	§16.1

Reading a part number: the digits after `74` give the function; the letters in between give the **family** — `74LS138` is low-power Schottky TTL, `74HC138` is CMOS, `74HCT138` is CMOS with TTL-compatible input thresholds (§34.2). Same pinout and logic, very different electrical behaviour.