CSSE2010 — Week 2 Notes

Table of contents

Binary Arithmetic

See binary-number-representations[1] for signed-format background (signed magnitude, two's complement) and logic-gates[2]/boolean-algebra[3] for the gate-level building blocks used below.

Quiz: -32 in 8-bit signed magnitude

Signed magnitude splits the word into a sign bit (MSB, 1 = negative) and a magnitude in the remaining bits: 32 = 00100000, so with the sign bit set this is `10100000` (A).

Quiz: -32 in 8-bit two's complement

Two's complement of a positive value: invert all bits, then add 1.

$$00100000 \rightarrow \text{invert} \rightarrow 11011111 \rightarrow +1 \rightarrow 11100000$$

So -32 in 8-bit two's complement is `11100000` (D) — different from the signed-magnitude answer above, which is the point: the same bit pattern means different things depending on the declared format.

Universal gates recap

NOT/AND/OR can each be built from NAND-only or NOR-only gates (see logic-gates[4] for the equivalent-circuit diagrams) — this was revision from last lecture before moving into arithmetic.

[1] courses/csse2010/binary-number-representations.pdf
[2] courses/csse2010/logic-gates.pdf
[3] courses/csse2010/boolean-algebra.pdf
[4] courses/csse2010/logic-gates.pdf

Binary addition

Single-bit addition (no carry-in):

Addend	0	0	1	1
Augend	+0	+1	+0	+1
Sum	0	1	1	0
Carry	0	0	0	1

- The **bit-wise addition procedure is identical** regardless of whether the operands are interpreted as unsigned or two's complement — only the *interpretation* of the result differs.
- **Two's complement**: the carry-out from the MSB is simply discarded to get the correct result.
- **One's complement**: the carry-out from the MSB must be added back into the result (end-around carry) — a practical drawback of one's complement versus two's complement.

Worked example (8-bit):

```
  Decimal   Unsigned      Decimal   2's complement
  10        00001010      10        00001010
+ 243     + 11110011    +(-13)    + 11110011
--------- ------------  -------   -------------
  253       11111101      -3        11111101
```

Overflow

Overflow: the true result doesn't fit in the available bits, so the answer is wrong.

- **Unsigned**: overflow carry-out from the MSB.
- **Two's complement**: overflow carry-in to the MSB carry-out from the MSB, i.e. Overflow $= C_{in} \oplus C_{out}$.
- Equivalently for two's complement: overflow happens when two positives sum to a negative, or two negatives sum to a positive.

Worked example — both cases overflow:

```
  Decimal   Unsigned      Decimal   2's complement
  15        00001111      125       01111101
+ 243     + 11110011    +   4     + 00000100
--------- ------------  -------   -------------
  258       00000010     129        10000001   (wraps to -127, wrong)
```

Quiz: adding two 6-bit two's complement numbers

$110101 + 001111$ in 6 bits:

```
  110101
+ 001111
--------
 1000100   → truncate to 6 bits → 000100
```

Answer: `000100` (A) — the 7th bit is discarded since we're constrained to 6 bits (no overflow here since $C_{in} \oplus C_{out} = 0$ into/out of the MSB).

Half-adder

A device that adds 2 bits with **no carry-in**.

Truth table (inputs A, B; outputs Sum, Carry):

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$S = A \oplus B \qquad C = AB$$

Full adder

A half-adder alone can't handle a carry-in from a previous stage, so a **full adder** takes three inputs (A, B, C_{in}) and produces Sum and C_{out}:

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + C_{in}(A \oplus B)$$

A full adder is built from **two half-adders plus an OR gate**: the first half-adder combines A and B; its sum output feeds into a second half-adder along with C_{in}; the two half-adders' carry outputs are OR'd together to form C_{out}.

Ripple-carry adder

Cascading full adders — each stage's C_{out} feeding the next stage's C_{in} — builds a multi-bit binary adder (e.g. 4 full adders for 4-bit addition: $A_3B_3 \dots A_0B_0 \rightarrow S_3 \dots S_0, C_4$). For plain addition the initial carry-in C_0 is 0. This structure is called a **ripple-carry adder**, since the carry has to propagate ("ripple") through every stage before the final sum is valid.

Reminders

- Week 2 labs: P2 sessions (Mon-Tue) run Lab 2 (Logic Gates); P1 sessions (Thu-Fri) run Lab 3 (Binary Arithmetic).
- Complete the week 1 exercises folder and ask if unclear.
- Watch the summary video on Signed Binary Formats.

Lecture 4 — Combinational Logic

See combinational-logic-blocks[5] for the reference definitions/tables of the adder-subtractor, multiplexer and decoder introduced here, and timing-diagrams[6] for the timing-diagram representation and real-gate timing parameters.

Today's outline

- Admin
- Binary subtractors
- More combinational logic circuits
  - Multiplexers
  - Decoders
  - Timing diagram representations

Recap — binary adder

Picking up from 2026-08-04-binary-arithmetic[7]: full adders can be **cascaded** to make a multi-bit binary adder. For 4 bits, four full adders take $A_3B_3 \dots A_0B_0$ and produce $S_3 \dots S_0$ plus C_4, with each stage's carry-out C_{i+1} wired to the next stage's carry-in. For addition the initial carry-in C_0 is **0**. This arrangement is a **ripple-carry adder**.

[5] courses/csse2010/combinational-logic-blocks.pdf
[6] courses/csse2010/timing-diagrams.pdf
[7] courses/csse2010/2026-08-04-binary-arithmetic.pdf

Binary subtractors

$A - B$ is implemented as $A + (-B)$, with $-B$ the two's complement of B (flip the bits, add 1). The two pieces needed on top of a plain adder are a **controlled inverter** (an XOR gate per bit, with the mode select M as the second input) and the extra $+1$, which is supplied via the adder's carry-in. Full derivation and the adder-subtractor circuit are in combinational-logic-blocks[8].

Two slides here are posed as questions and left blank on the handout:

- *"How can we use a gate to flip a bit — but only sometimes?"* ($Z = \bar{B}$ when $M = 1$, $Z = B$ when $M = 0$.) Derived: a 2-input XOR, $Z = B \oplus M$.
- *"What should carry-in be?"* on the adder-subtractor slide. Derived: $C_0 = M$ — the same mode-select line that drives the XOR gates, so subtract mode automatically contributes the $+1$ that two's complement needs.

Combinational circuits in general

A combinational circuit is a combination of logic gates with n inputs and m outputs; each output is a function of the n inputs, the **output depends on current inputs only**, and it can be written as a truth table with n input columns, m output columns and 2^n rows. See combinational-logic-blocks[9].

Multiplexers

2^n data inputs, 1 output, n select inputs that steer one data input to the output. The lecture covered the 4-to-1 mux (logic symbol, function table, and the AND-OR sum-of-products implementation with inverted select literals) and the 2-to-1 mux. Tables and circuit descriptions: combinational-logic-blocks[10]. Off-the-shelf part: the 74HCT157 quad 2-input mux, see device-pinouts[11].

Quiz: 2-to-1 multiplexer truth table

A 2-to-1 multiplexer has data inputs D_0 and D_1, control input S_0, and output F, with function table $F = D_0$ when $S_0 = 0$ and $F = D_1$ when $S_0 = 1$. Which of three candidate truth tables (columns S_0, D_0, D_1, F) goes with this circuit?

Reasoning from the function table, row by row in the order $S_0 D_0 D_1 = 000, 001, 010, 011, 100, 101, 110, 111$:

[8] courses/csse2010/combinational-logic-blocks.pdf
[9] courses/csse2010/combinational-logic-blocks.pdf
[10] courses/csse2010/combinational-logic-blocks.pdf
[11] courses/csse2010/device-pinouts.pdf

- The first four rows have $S_0 = 0$, so F must copy D_0 and ignore D_1: $F = 0, 0, 1, 1$.
- The last four rows have $S_0 = 1$, so F must copy D_1 and ignore D_0: $F = 0, 1, 0, 1$.

$$F = 0,\ 0,\ 1,\ 1,\ 0,\ 1,\ 0,\ 1$$

That is **option 2**. Option 1 has F following D_1 when $S_0 = 0$ and D_0 when $S_0 = 1$ — the two data inputs swapped. Option 3 is $F = 0, 0, 0, 0, 1, 1, 1, 1$, i.e. $F = S_0$: it ignores the data inputs completely.

Equivalently: $F = D_0\bar{S}_0 + D_1 S_0$.

Quiz: choosing mux inputs to realise $X = \bar{S}_0 + S_1$

A 4-to-1 multiplexer has A on data input 0, B on 1, C on 2, D on 3, selects S_1S_0, and output X. What must A, B, C, D be so that

$$X = \bar{S}_0 + S_1$$

Options: (1) A=0, B=1, C=0, D=0; (2) A=0, B=1, C=1, D=1; (3) A=1, B=0, C=1, D=1; (4) A=1, B=1, C=0, D=1.

Reasoning: the mux steers data input i to the output when S_1S_0 equals i in binary, so each data input must simply be **the value the desired function takes for that select combination**. Evaluate $\bar{S}_0 + S_1$ for all four:

S_1	S_0	$\bar{S}_0$	$X = \bar{S}_0 + S_1$	Selected input
0	0	1	1	A (input 0)
0	1	0	0	B (input 1)
1	0	1	1	C (input 2)
1	1	0	1	D (input 3)

So $A = 1, B = 0, C = 1, D = 1$ — **option 3**.

This is the general trick: tying a mux's data inputs to constants turns it into a lookup table for any function of the select variables.

Decoders

A decoder converts an n-bit input to a logic 1 on exactly one of 2^n outputs. The lecture used the 3-to-8 decoder (inputs A, B, C; outputs $D_0 \ldots D_7$) with its truth table and the eight-AND-gate implementation. See combinational-logic-blocks[12].

Timing diagram representations

The lecture closed by adding the **timing diagram** as another logic representation — like a truth table but graphical — using an inverter as the worked example, then the convention for drawing 2-input waveforms so all input combinations are covered, then the reality that gates aren't perfect (propagation delay, fall time, rise time). All of this is in timing-diagrams[13].

Quiz: timing diagram for an XOR gate

Given input waveforms where A is low for the first half of the diagram then high for the second half, and B toggles at twice that rate (low, high, low, high) — so the four intervals are $AB = 00, 01, 10, 11$ — which of three candidate output waveforms represents an **XOR** gate?

Reasoning from the XOR truth table ($X = 1$ iff exactly one input is 1):

Interval	A	B	$A \oplus B$
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	0

So the output must be **low, high, high, low** — a single pulse spanning the middle two intervals. That is **option 2** — the only candidate with a pulse that goes high and comes back low again. Option 1 has a single rising edge and then stays high to the end, so it can never be XOR. Option 3 is the exact complement of option 2 (high, low, low, high), i.e. XNOR.

[12] courses/csse2010/combinational-logic-blocks.pdf
[13] courses/csse2010/timing-diagrams.pdf

Reminders

- **Lab 4 preparation task** — next week (week 3). It will be available on Blackboard under *Learning Resources*; it involves drawing some circuit schematic diagrams (see circuit-schematics[14]), and you must bring it to Lab 4 (P2 Mon-Tue sessions in week 3). Complete it before Lab 4.
- Supporting material for the labs is now on Blackboard.
- Attempt the posted exercises and quiz (not assessed).
- Any doubts: use course consultation, ask after the lecture, or use the Ed forum.

Majority Function Circuit Design

Learning Lab 2's task, following on from revision of gate truth tables and Boolean identities (see logic-gates[15] and boolean-algebra[16]) and the introduction of circuit-schematics[17]/device-pinouts[18] conventions. Main lab objective: verify the functionality of a logic circuit either as a hardware circuit built from logic ICs, or as a Logisim simulation.

Question 1 — 3-input majority function

The 3-input **majority function** Z is true if at least two of the inputs A, B, C are true. Already simplified in 2026-07-30-intro-to-logic-gates[19] (the lecture's M truth table/sum-of-products example is this same function):

$$Z = AB + AC + BC$$

(a) Convert Z to a NAND-only circuit (using Boolean algebra or a logic diagram).

(b) Draw a circuit schematic for the NAND-only circuit — inputs are switches, output is an LED.

(c) Wire up the circuit (or simulate in Logisim) and determine the truth table.

[14] courses/csse2010/circuit-schematics.pdf
[15] courses/csse2010/logic-gates.pdf
[16] courses/csse2010/boolean-algebra.pdf
[17] courses/csse2010/circuit-schematics.pdf
[18] courses/csse2010/device-pinouts.pdf
[19] courses/csse2010/2026-07-30-intro-to-logic-gates.pdf

Working

(a) NAND-only conversion. The standard **sum-of-products → NAND-only** technique (double negation + De Morgan's law, see boolean-algebra[20]): double-invert the whole expression, then push one inversion inside with De Morgan's law, turning each AND term into a NAND and the outer OR into a NAND of those NAND outputs:

$$Z = AB + AC + BC = \overline{\overline{AB + AC + BC}} = \overline{\overline{AB} \cdot \overline{AC} \cdot \overline{BC}}$$

i.e. $Z = \text{NAND}(\text{NAND}(A, B),\ \text{NAND}(A, C),\ \text{NAND}(B, C))$ — each first-level AND gate becomes a NAND gate, and the second-level OR gate becomes a NAND gate combining the (already-inverted) first-level outputs.
Note this final combining gate is a **3-input** NAND, which a 74HCT00 doesn't provide directly (it only has 2-input NAND gates) — building the 3-input NAND from 2-input NAND gates, and the resulting schematic/pin assignment, is the hands-on part of the lab task and isn't given directly in the source material.
(b)/(c): use circuit-schematics[21] labelling conventions and device-pinouts[22] for the 74HCT00 pinout and IO board switch/LED pin numbers (inputs A, B, C on three switches, output Z on an LED).

Reference material

Binary Number Representations

Bits, bytes, and words

- **Bit** = binary digit (0 or 1).
- **Byte** = 8 bits, e.g. `01010111`.
- Modern computers deal with **words**, usually a power-of-2 number of bytes: 1, 2, 4, or 8 bytes = 8, 16, 32, 64 bits.

Representing whole (unsigned) numbers

Each bit position has a value — a power of 2, increasing from right (least significant) to left (most significant):

[22] courses/csse2010/boolean-algebra.pdf
[22] courses/csse2010/circuit-schematics.pdf
[22] courses/csse2010/device-pinouts.pdf

Bit position	9	8	7	6	5	4	3	2	1	0
Value	512	256	128	64	32	16	8	4	2	1

Binary → decimal: add the values of each position where the bit is 1. E.g. `10010001` = $128 + 16 + 1 = 145$.

- **Least significant bit (LSB)** — the bit position worth the least ($2^0 = 1$).
- **Most significant bit (MSB)** — the bit position worth the most. For an n-bit unsigned word, the MSB is worth 2^{n-1}.

Converting decimal to binary

Two equivalent methods (example: convert 53 to binary):

- **Method 1**: rewrite n as a sum of powers of 2, by repeatedly subtracting the largest power of 2 not greater than n. Assemble the binary number from 1's in the bit positions corresponding to those powers of 2, 0's elsewhere.
- **Method 2** (build up from the right/LSB): divide n by 2; the remainder (0 or 1) is the next bit; repeat with $n =$ the quotient, until $n = 0$.

Number range (unsigned)

- Smallest representable value: all 0's → 0.
- Largest representable value: all 1's → for an n-bit word, $2^n - 1$ (e.g. 255 for 8 bits).

Other radices

Radix = number system base. A radix-k number system has k distinct symbols for digits 0 to $k - 1$, and the value of each digit (from the right) is $k^0, k^1, k^2, ...$

- **Octal** (radix-8): symbols `0`–`7`. One octal digit corresponds to exactly 3 bits.
- **Hexadecimal** (radix-16): symbols `0`–`9`, `A`–`F`. One hex digit corresponds to exactly 4 bits — very convenient for grouping binary.

Dec	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Oct	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17	20	21
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11

Radix notation conventions

Since a bare number like `101` or `747` is ambiguous, a radix indicator is needed. A subscript works generically (e.g. 101_2, 101_{16}); in code/assembly, conventions vary:

Radix	Convention	Example	Where used
Hex	leading `0x`	`0x101`	C, Atmel AVR
Hex	trailing `h`	`101h`	some assembly languages
Hex	leading `$`	`$747`	Atmel AVR assembly
Octal	leading `0`	`0101`	C, Atmel AVR
Octal	trailing `q`	`101q`	some assembly languages
Octal	leading `@`	`@747`	some assembly languages
Binary	leading `0b`	`0b101`	Atmel AVR assembly, some C
Binary	trailing `b`	`101b`	some assembly languages
Binary	leading `%`	`%101`	some assembly languages

Boolean Algebra

Logic functions can be expressed as expressions of variables (literals, e.g. A, B, X) and functions (e.g. $+, \cdot, \oplus, \bar{}$). Variables and functions can only take values 0 or 1.

Notation conventions

- **Inversion**: overline, e.g. $\mathrm{NOT}(A) = \bar{A}$ ("A bar").
- **AND**: dot, or implied by adjacency, e.g. $\mathrm{AND}(A, B) = AB = A \cdot B$.
- **OR**: plus sign, e.g. $\mathrm{OR}(A, B, C) = A + B + C$.
- Other examples: $\mathrm{XOR}(A, B) = A \oplus B = \bar{A}B + A\bar{B}$; $\mathrm{NAND}(A, B, C) = \overline{ABC}$; $\mathrm{NOR}(A, B) = \overline{A + B}$.

Boolean identities

Name	AND form	OR form
Identity law	$1A = A$	$0 + A = A$
Null law	$0A = 0$	$1 + A = 1$
Idempotent law	$AA = A$	$A + A = A$
Inverse law	$A\bar{A} = 0$	$A + \bar{A} = 1$
Commutative law	$AB = BA$	$A + B = B + A$
Associative law	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Distributive law	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$

Name	AND form	OR form
Absorption law	$A(A+B) = A$	$A + AB = A$
De Morgan's law	$\overline{AB} = \bar{A} + \bar{B}$	$\overline{A+B} = \bar{A}\bar{B}$

De Morgan's law also means **AND and OR gates can be interchanged if you invert all the inputs and the output** — this is why NAND/NOR-only circuits (see logic-gates[23]) can implement any function.

Sum of products

Any logic function can be implemented as the **OR of AND combinations** of its inputs (a *sum of products*):

1. For each row where the truth table output is 1, write the AND term of the inputs (complementing wherever the input is 0) that produces that row.
2. OR all of those terms together.

This always works, but doesn't necessarily give the minimum number of gates — the resulting expression can often be simplified further using the identities above. E.g. $Z = AB + AC = A(B + C)$ (distributive law) uses fewer gates than the raw sum-of-products form.

Circuit Schematics

A **logic diagram** shows the *idea* of a circuit and is hardware-independent (see logic-gates[24]). A **circuit schematic** goes further: it tells you how to actually **build** the circuit on real hardware — labelled ICs, pin numbers, power connections — so it's hardware-specific. Schematics are drawn for practicals and assessment using 74-series logic chips and the CSSE2010/CSSE7201 IO Board (see device-pinouts[25] for pin layouts).

> **Source:** *CSSE2010/CSSE7201 - Guide to Drawing Circuit Schematics* (Blackboard, v2.3). The rules below are that document's requirements; the errors are its annotated bad-schematic example.

[23] courses/csse2010/logic-gates.pdf
[24] courses/csse2010/logic-gates.pdf
[25] courses/csse2010/device-pinouts.pdf

Required elements

Every circuit schematic must show:

- **Labelled inputs and outputs** — every input/output needs a sensible, unique name. Inputs and outputs use different arrow symbols (which way they point), and by convention inputs are drawn on the left, outputs on the right.
- **Labelled devices** — each physical device (chip or IO board) gets a unique identifier: one or more letters for the device type, then a sequential number starting from 1.
  - Logic chips: `U1`, `U2`, `U3`, …
  - IO board: `IO1` (`IO2` for a second board, etc.) — when a group of inputs/outputs all belong to the same IO board, it only needs to be labelled once for the whole group, not per signal.
- **Gate labels within a chip** — where a chip contains multiple gates, each gate is labelled `A`, `B`, `C`, … after the chip ID, separated by a colon (e.g. `U1:A`, `U1:B`). Labels can be assigned to gates in any order. A chip with only one gate/device doesn't need a gate letter (just `U3`).
- **Device type** — written on the line below the device ID (e.g. `74HCT00`, `74HCT04`, `IOBOARD`).
- **Pin numbers** — every pin used must be numbered per the actual device pinout (see device-pinouts[26]).
- **Power supply connections** — shown once per device *type*, not per individual chip. Where two+ chip types share the same power pins, they can be grouped together. The IO board doesn't need power connections shown.

Common errors

From the guide's worked example of a bad schematic:

1. **Duplicate names** — two inputs given the same name (e.g. both called `A`). Every input needs a unique name.
2. **Missing junction dots** — where wires join or split, a dot must mark the connection. Wires that merely cross on the page, with no dot, are *not* electrically connected.
3. **Gate label exceeds chip capacity** — e.g. labelling a gate `U1:E` on a 74HCT00, which only has four 2-input NAND gates (`A`–`D`). A fifth gate needs a new chip ID (`U2:A`).
4. **Wrong device for the gate type** — a different gate type (e.g. NOT vs NAND) needs its own chip identifier; you can't add it under an existing chip's ID.
5. **Reusing pin numbers** — e.g. `U1:C` using the same pins as `U1:A`. You can't use the same physical gate twice.

[26] courses/csse2010/device-pinouts.pdf

6. **Inconsistent device type** — every gate labelled `U1:something` must show the *same* chip type; a chip ID can't switch part-way through (e.g. `U1:D` shown as a 74HCT47 when the rest of `U1` is a 74HCT00).
7. **Output errors** — two distinct mistakes to watch for:
    a. Using the input-arrow symbol where an output-arrow symbol is needed.
    b. Wiring a gate output directly into another output pin/signal — an input should never be tied straight to a gate's output; gate outputs should only feed output devices (e.g. LEDs) or the inputs of other gates. Doing this may destroy the device.

Combinational Logic Blocks

Reference note for the standard combinational building blocks: the adder-subtractor, the multiplexer, and the decoder. See logic-gates[27] for the gate symbols/truth tables and boolean-algebra[28] for the notation used below.

What makes a circuit combinational

A **combinational circuit** is a combination of logic gates with n inputs and m outputs:

- Each output can be expressed as a function of the n input variables.
- The **output depends on the current inputs only** — there is no memory of previous inputs. (Contrast with sequential-circuits[29], where the output also depends on stored state.)
- It can always be written as a truth table with n input columns, m output columns, and 2^n rows (one per possible input combination).

Binary subtraction

$A - B$ is usually implemented as $A + (-B)$, where:

- A and B are multi-bit quantities;
- "+" here means **addition**, not OR;
- $-B$ means the **two's complement** of B (see binary-number-representations[30]).

The two's complement of B is calculated by **flipping all the bits and adding 1**. So a subtractor is just an adder with two extra pieces: something that conditionally inverts B, and something that adds the extra 1.

[27] courses/csse2010/logic-gates.pdf
[28] courses/csse2010/boolean-algebra.pdf
[29] courses/csse2010/sequential-circuits.pdf
[30] courses/csse2010/binary-number-representations.pdf

The controlled inverter (XOR trick)

We need a gate that flips a bit *only sometimes*: given a mode bit M,

$$Z = \bar{B} \text{ when } M = 1, \qquad Z = B \text{ when } M = 0$$

The slide poses this as a question and is otherwise blank. Derived from the XOR truth table: XOR with one input held at 0 passes the other input through unchanged, and XOR with one input held at 1 inverts it. So the gate is a **2-input XOR**:

$$Z = B \oplus M$$

M	B	$Z = B \oplus M$	Effect
0	0	0	pass through
0	1	1	pass through
1	0	1	invert
1	1	0	invert

This is called a **controlled inverter**.

Adder-subtractor circuit

A 4-bit adder-subtractor is built from a 4-bit binary adder (a ripple-carry adder — see 2026-08-04-binary-arithmetic[31]) plus four XOR gates:

- Data input A ($A_0 \ldots A_3$) goes **straight** into the adder's A inputs.
- Data input B ($B_0 \ldots B_3$) goes into the adder's B inputs **via one XOR gate per bit**; the second input of every XOR gate is the shared **mode select** line M.
- $M = 0$ means **add**, $M = 1$ means **subtract**.
- The adder produces the data output $S_0 \ldots S_3$, plus a carry-out C_4 and taking a carry-in C_0.

What should the carry-in be? The slide asks this and leaves it blank. Derived: with $M = 1$ the XOR gates supply $\bar{B}$, and two's complement needs $\bar{B} + 1$ — the extra +1 is supplied by feeding it in as the carry-in. With $M = 0$ we want plain addition, which needs a carry-in of 0. Both cases are satisfied by

$$C_0 = M$$

[31] courses/csse2010/2026-08-04-binary-arithmetic.pdf

so the mode select line is wired to both the XOR gates and the adder's carry-in.

A 4-bit adder is available as an off-the-shelf IC — the 74HCT283, see device-pinouts[32].

Multiplexer (mux)

A **multiplexer** has:

- 2^n **data** inputs,
- **1** output,
- n **control** (or **select**) inputs, which *select* one of the data inputs to be "sent" or "steered" to the output.

4-to-1 multiplexer

Data inputs $D_0 \ldots D_3$, select inputs $S_1 S_0$, output F. The logic symbol is the characteristic trapezoid with the data inputs on the wide (left) edge, F on the narrow (right) edge, and the select inputs entering the bottom.

Function table:

S_1	S_0	F
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

Note this is a **function table**, not a full truth table — the full truth table would need $2^6 = 64$ rows for the six inputs $S_1, S_0, D_0, D_1, D_2, D_3$.

4-to-1 mux logic circuit implementation

- S_1 and S_0 each feed an inverter, giving $\bar{S}_1$ and $\bar{S}_0$ as well as the true forms.
- Four **3-input AND gates**, one per data input. Each AND gate takes its data input plus the select-literal pair that identifies it: D_0 with $\bar{S}_1 \bar{S}_0$, D_1 with $\bar{S}_1 S_0$, D_2 with $S_1 \bar{S}_0$, D_3 with $S_1 S_0$.
- The four AND outputs feed a single **4-input OR gate** whose output is F.

[32] courses/csse2010/device-pinouts.pdf

That is exactly the sum-of-products form:

$$F = D_0\bar{S}_1\bar{S}_0 + D_1\bar{S}_1S_0 + D_2S_1\bar{S}_0 + D_3S_1S_0$$

Only the AND gate whose select literals match the current S_1S_0 can be 1, so exactly one data input reaches the OR gate at a time.

2-to-1 multiplexer

Data inputs D_0 and D_1, one control input S_0, output F.

Function table:

S_0	F
0	D_0
1	D_1

Expanded to a full truth table over all three inputs:

S_0	D_0	D_1	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

i.e. $F = D_0\bar{S}_0 + D_1S_0$. When $S_0 = 0$ the output tracks D_0 and ignores D_1; when $S_0 = 1$ it tracks D_1 and ignores D_0.

A quad 2-input multiplexer is available as an off-the-shelf IC — the 74HCT157, see device-pinouts[33].

[33] courses/csse2010/device-pinouts.pdf

Using a mux to implement a logic function

Because the data inputs can be tied to constant 0 or 1, an n-select mux with its selects wired to the function's variables can implement *any* function of those n variables: set data input D_i to the value the function should take when the selects equal i. See the worked polling question in 2026-08-06-combinational-logic[34].

Decoder

A **decoder** converts an n-bit input to a logic 1 on **exactly one** of its 2^n outputs (the one whose index equals the input value); all other outputs are 0.

3-to-8 decoder

Inputs A, B, C (with A the most significant), outputs $D_0 \ldots D_7$.

A	B	C	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	**1**	0	0	0	0	0	0	0
0	0	1	0	**1**	0	0	0	0	0	0
0	1	0	0	0	**1**	0	0	0	0	0
0	1	1	0	0	0	**1**	0	0	0	0
1	0	0	0	0	0	0	**1**	0	0	0
1	0	1	0	0	0	0	0	**1**	0	0
1	1	0	0	0	0	0	0	0	**1**	0
1	1	1	0	0	0	0	0	0	0	**1**

(The slide tabulates only the first four rows and elides the rest with "…"; the remaining four rows are filled in above from the definition — the 1 always sits in the column whose index is the binary value of ABC.)

3-to-8 decoder logic circuit implementation

- Each of A, B, C is fanned out to an inverter, so all six literals $A, \bar{A}, B, \bar{B}, C, \bar{C}$ are available on a vertical bus.
- **Eight 3-input AND gates**, one per output. Each taps the three literals matching its index:

[34] courses/csse2010/2026-08-06-combinational-logic.pdf

$$D_0 = \bar{A}\bar{B}\bar{C}, \quad D_1 = \bar{A}\bar{B}C, \quad D_2 = \bar{A}B\bar{C}, \quad D_3 = \bar{A}BC$$
$$D_4 = A\bar{B}\bar{C}, \quad D_5 = A\bar{B}C, \quad D_6 = AB\bar{C}, \quad D_7 = ABC$$

Each AND gate is the minterm for one input combination, so exactly one is high at any time.

Device Pinouts

Pin numbers for the 74-series logic chips and IO board used in circuit-schematics[35]. All quad 2-input gate chips (`74HCT00`, `08`, `32`, `86`) share the **same** pin layout — only the gate function differs.

> **Source:** *CSSE2010/CSSE7201 Device Pinouts and Symbols* (Blackboard, v4). This is the whole parts list: **every gate chip in it is 2-input** (plus the hex inverter), so a design needing a 3-or-more-input gate has to be built up from 2-input ones. There is no 3-input AND/OR/NAND in the kit.

Power and ground

Power/ground connections are shown separately from gate pins on a schematic (see circuit-schematics[36]): logic high/power uses a `VCC` symbol, logic low/ground uses a ground symbol.

Quad 2-input gates (74HCT00 / 08 / 32 / 86)

Chip	Function	Gate A	Gate B	Gate C	Gate D	VCC	GND
74HCT00	Quad 2-input NAND	in 1,2 → out 3	in 4,5 → out 6	in 9,10 → out 8	in 12,13 → out 11	14	7
74HCT08	Quad 2-input AND	in 1,2 → out 3	in 4,5 → out 6	in 9,10 → out 8	in 12,13 → out 11	14	7
74HCT32	Quad 2-input OR	in 1,2 → out 3	in 4,5 → out 6	in 9,10 → out 8	in 12,13 → out 11	14	7
74HCT86	Quad 2-input XOR	in 1,2 → out 3	in 4,5 → out 6	in 9,10 → out 8	in 12,13 → out 11	14	7

[35] courses/csse2010/circuit-schematics.pdf
[36] courses/csse2010/circuit-schematics.pdf

74HCT04 — Hex inverter

Six independent NOT gates. VCC = pin 14, GND = pin 7.

Gate	A	B	C	D	E	F
in → out	1→2	3→4	5→6	9→8	11→10	13→12

74HCT74 — Dual D-type flip-flop (14-pin)

Pin	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Signal	CLR1	D1	CLK1	SET1	Q1	$\overline{Q1}$	GND	$\overline{Q2}$	Q2	SET2	CLK2	D2	CLR2	VCC

Set and clear inputs can be omitted from the schematic if not needed.

74HCT157 — Quad 2-input multiplexer (16-pin)

Pin	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Signal	SEL	A0	B0	Y0	A1	B1	Y1	GND	Y2	B2	A2	Y3	B3	A3	ENABLE	VCC

74HCT175 — Quad D-type flip-flop (16-pin)

Pin	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Signal	CLR	Q0	$\overline{Q0}$	D0	D1	Q1	$\overline{Q1}$	GND	CLK	Q2	$\overline{Q2}$	D2	D3	Q3	$\overline{Q3}$	VCC

74HCT283 — 4-bit full adder (16-pin)

Pin	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Signal	S1	B1	A1	S0	A0	B0	C0	GND	C4	S3	B3	A3	S2	A2	B2	VCC

$C0$ is the carry-in, $C4$ the carry-out; A_i, B_i are the two 4-bit operands and S_i the 4-bit sum.

IO board

Pins	Function
1	TX
2	RX
3	SSD cathode
4	SSD decimal point
5–11	SSD segments G to A
12–15	Buttons B3 to B0
16	Logic probe
17–24	LEDs L0 to L7
25–32	Switches S0 to S7

On a schematic the IO board is labelled `IO1` (`IO2` for a second board), with inputs using an input-style arrow and outputs an output-style arrow (see circuit-schematics[37]). Use sensible signal names rather than the raw pin labels where possible.

Logic Gates

Every logic gate has one or more inputs and exactly one output, and can be described four equivalent ways: **logic symbol, truth table, Boolean expression, timing diagram** (see timing-diagrams[38] for the last of these, and for real-gate propagation delay / rise / fall times).

The 7 basic gate types

Gate	Behaviour	Truth table ($A, B \to X$)
NOT (inverter)	Inverts the input	`0→1`, `1→0`
AND	$X = 1$ iff all inputs are 1	00→0, 01→0, 10→0, 11→1
OR	$X = 1$ iff at least one input is 1	00→0, 01→1, 10→1, 11→1
NAND	$X = 1$ iff at least one input is 0 (NOT AND)	00→1, 01→1, 10→1, 11→0
NOR	$X = 1$ iff all inputs are 0 (NOT OR)	00→1, 01→0, 10→0, 11→0
XOR	$X = 1$ iff exactly one input is 1 ("odd function")	00→0, 01→1, 10→1, 11→0

[37] courses/csse2010/circuit-schematics.pdf
[38] courses/csse2010/timing-diagrams.pdf

Gate	Behaviour	Truth table ($A, B \to X$)
XNOR	$X = 1$ iff inputs are the same ("even function")	00→1, 01→0, 10→0, 11→1

Useful to remember: **XOR is the odd function**, **XNOR is the even function** — this generalises directly to more than 2 inputs (XOR = 1 iff an odd number of inputs are 1).

Universal (complete) gates

All circuits can be constructed from **NAND-only** or **NOR-only** gates — these are called **complete** (or universal) gates, because NOT, AND, and OR can each be built purely from one gate type:

- **NOT**: a NAND (or NOR) gate with both inputs tied together.
- **AND**: two NANDs in series (NAND followed by another NAND used as an inverter), or the NOR equivalent.
- **OR**: built similarly from NAND-only or NOR-only combinations.

NAND and NOR are preferred in practice because they're easier to build from transistors than AND/OR directly.

Gates on ICs

Example: the **74HCT00** integrated circuit packages four independent 2-input NAND gates into a single 14-pin DIP chip (Vcc = power e.g. 5V, GND = ground/0V; pin spacing 0.1"×0.3", chip ~15mm long).

Timing Diagrams

A **timing diagram** is the fourth of the four equivalent ways to describe a logic function, alongside the **logic symbol**, the **truth table**, and the **Boolean expression** (see logic-gates[39]).

[39] courses/csse2010/logic-gates.pdf

Timing diagram as a logic representation

It is like a truth table, but in **graphical format**: time runs left to right along the horizontal axis, and each signal gets its own horizontal track that steps between the logic 0 and logic 1 levels (drawn as two dashed reference lines).

Worked example — an inverter. The input waveform starts at logic 0, rises to logic 1 for a period, then falls back to logic 0. The output waveform is the mirror image: it starts at logic 1, falls to logic 0 for exactly that period, then returns to logic 1. Each **change in the input causes the output to change**.

Convention for drawing the input waveforms

The input waveforms must be drawn so that **all possible input combinations are covered** — that is what makes the diagram equivalent to the full truth table.

The standard construction for a 2-input gate divides the time axis into four equal intervals and drives the inputs like a binary count:

- A is low for the first half of the diagram, then high for the second half.
- B toggles at twice that rate: low, high, low, high.

Reading the four intervals left to right therefore gives $AB = 00, 01, 10, 11$ — every combination, once each, in truth-table order. For n inputs the same scheme generalises: input i toggles at half the rate of input $i + 1$, and the diagram is 2^n intervals wide.

To read a gate's output off the diagram, work interval by interval: identify the AB combination in that interval, look up the gate's truth table row, and draw the output at that level for the width of the interval.

Real gates aren't perfect: the reality of timing

The idealised diagrams above show instantaneous, perfectly square transitions. Real gates do not behave that way — real waveforms have sloped edges, and the output changes some time *after* the input does. Three quantities describe this:

Term	Definition
Propagation delay	Time for a change in the input to affect the output.
Fall time	Time taken for the output to fall from 1 to 0.
Rise time	Time for the output to rise from 0 to 1.

Measured on an inverter: the input ramps up, and only after the propagation delay does the output begin to move; the output's downward edge then takes the fall time to complete, and its later upward edge takes the rise time.