CSSE2010 — Week 4 Notes

Table of contents

Lecture 7 — Counters

See [counters] for the reference material this lecture introduces, and shift-registers[1], sequential-circuits[2] and flip-flops-and-latches[3] for the recap slides it opens with.

Today's outline

- Poll on last week's bidirectional shift register
- Shift register types — recap from last week
- Counters, and binary counters
- A digital system application built around a counter
- State: current state, next state, and the D inputs
- Counter examples worked live in class
- Next time: state machines

From the teaching outline

Slide 2 is a screenshot of the course teaching outline table rather than a content slide. What it shows for now:

- **Week 4, Mon-Tue 17-18 Aug** — this lecture, *Lecture 7: Counters* (17 Aug).
- **Week 4, Thu-Fri 20-21 Aug** — *Lecture 8: Finite State Machines* (20 Aug).
- Learning labs this week: **Lab 6 Shift Registers** (P2, Mon-Tue) and **Lab 7 Counters** (P1, Thu-Fri).
- Week 5: *Lecture 9 ALU and Control Unit* (24 Aug), *Lecture 10 Introduction to AVR and Assembly Language* (27 Aug), with labs 8 (Finite State Machines) and 9 (AVR Assembly Programming 1).

Quiz: shift direction control

> Which of the following statements about the circuit below is true?
>
> A. When A is 1, flip-flop values are shifted to the right B. When A is 0, flip-flop values are shifted to the right C. When A is 0, flip-flop values stay the same D. When FN is 1, flip-flop values are shifted to the left E. When FN is 0, flip-flop values stay the same F. When FN is 0, flip-flop values are shifted to the left

[1] courses/csse2010/shift-registers.pdf
[2] courses/csse2010/sequential-circuits.pdf
[3] courses/csse2010/flip-flops-and-latches.pdf

The figure is the mux-per-stage construction from [[2026-08-13-shift-registers|last lecture]], two stages deep. Reading it off:

- **Stage 1**: a 2-to-1 mux with select $S = FN$. Its `1` input is the external signal A; its `0` input is fed back from the **stage 1 flip-flop's own Q output**. The mux output drives that flip-flop's D input.
- **Stage 2**: an identical mux, also selected by FN. Its `1` input is **stage 1's** Q; its `0` input is fed back from **stage 2's own** Q. Its output drives stage 2's D input.
- Both flip-flops share the same CLK.

So writing the two next-state equations, with Q_a = stage 1 and Q_b = stage 2:

$$D_a = \begin{cases} A & FN = 1 \\ Q_a & FN = 0 \end{cases} \qquad D_b = \begin{cases} Q_a & FN = 1 \\ Q_b & FN = 0 \end{cases}$$

- When $FN = 1$, A is loaded into stage 1 and stage 1's value moves into stage 2 — data shifts **left-to-right along the diagram**, one stage per clock.
- When $FN = 0$, each flip-flop's D input is its own Q, so every flip-flop reloads the value it already had — **nothing changes**.

FN here is therefore a shift *enable*, not a direction control (there is no left-shift path in this circuit at all), so every option mentioning "shifted to the left" is out, and A is plain data rather than a control signal, so A/B/C are out too.

Answer: E — when FN is 0, flip-flop values stay the same.

Shift register types — recap

Two recap slides repeat last week's figures: the seven shift-register configurations (serial in/shift right/serial out, serial in/shift left/serial out, parallel in/serial out, serial in/parallel out, parallel in/parallel out, rotate right, rotate left), and the clock-by-clock trace of `0101` being shifted serially into a 4-bit register versus being loaded into it in parallel in a single clock. All of that content lives in shift-registers[4].

Counters

The definition slide, the n-bit binary counter facts (n flip-flops, 2^n states, counts 0 to $2^n - 1$), and the 4-bit binary counting sequence table are all recorded in [counters].

[4] courses/csse2010/shift-registers.pdf

Digital system application

A block-diagram slide (from Floyd's *Digital Fundamentals*) showing where a counter sits in a real system — an automated tablet-bottling line:

- A **keypad** for the number of tablets per bottle feeds an **encoder**, which feeds **Register A**. Register A drives **Decoder A** and a 7-segment display showing the tablets-per-bottle setting, and also a **code converter** producing the binary code for the preset number.
- On the conveyor, a **sensor** emits one pulse per tablet passing the valve. Those pulses are the clock of a **counter**, so the counter holds the binary code for the actual number of tablets in the bottle. A separate pulse **resets the counter to zero** when the next bottle is in place.
- A **comparator** tests the counter value against the preset value. Its output goes HIGH when $A = B$: HIGH closes the valve and advances the conveyor, LOW keeps the valve open.
- The counter value also feeds an **adder**, whose other input is the running total held in **Register B**; the comparator's HIGH also causes the new sum to be stored back into Register B. Register B drives **Decoder B** (total display) and a **MUX/DEMUX** path that sends the accumulated total on to a computer for storage.

The point of the slide is that the counter, adder, register, decoder, comparator, encoder, mux and demux blocks covered so far in the course compose into a complete system — see combinational-logic-blocks[5] for the mux/decoder blocks and 2026-08-04-binary-arithmetic[6] for the adder.

One-bit counter

The slide is titled "One bit counter" and marked *"To be completed in class"*, but the figure itself is drawn: a single D flip-flop, clocked on CLK, with a wire from its $\bar{Q}$ output looped back around to its D input.

Derived from that figure: since D is the next state and $D = \bar{Q}$, the flip-flop's value inverts on every clock edge — the counter counts

$$0 \to 1 \to 0 \to 1 \to \ldots$$

which is exactly the $n = 1$ binary counter ($2^1 = 2$ states, counting 0 to $2^1 - 1 = 1$). This is the standard *toggle* configuration.

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

State

The slide that makes counter design mechanical, illustrated with two D flip-flops on a common CLK whose D_1 and D_0 inputs are both drawn as "?":

- the values stored in the flip-flops are the **current state** of the circuit;
- the D inputs are the **next state**;
- the D inputs are some (combinational) **function of the current state and the inputs**.

The design method that follows from this is written up as a recipe in [counters]; the underlying state terminology is in sequential-circuits[7].

Counter example (blank slide)

Slide 11 is a "Counter Example" marked *"To be completed in class"*. It gives an empty present-state / next-state table with columns $Q_1, Q_0 \mid D_1, D_0$ and rows for all four states $00, 01, 10, 11$, alongside a figure of two D flip-flops on a common CLK with **nothing connected** to their D inputs and no target sequence stated anywhere.

Because no count sequence is given, the intended answer is *not* recoverable from the deck — this one needs the lecture recording or a classmate's notes. Structurally it is the same exercise as the worked example below.

Quiz: what sequence does this counter count through?

> What sequence does the counter below count through? (Assume Q_1Q_0 starts at 00.)
>
> A. $00 \to 11 \to 10 \to 01 \to 00$ B. $00 \to 10 \to 11 \to 01 \to 00$ C. $00 \to 11 \to 01 \to 10 \to 00$ D. $00 \to 01 \to 11 \to 10 \to 00$ E. $00 \to 01 \to 10 \to 11 \to 00$ F. $00 \to 10 \to 01 \to 11 \to 00$

Reading the figure: two D flip-flops on a common CLK.

- The **left** flip-flop's $\bar{Q}$ output is wired back to its own D input, and its Q output is Q_1. So it is the toggle from the one-bit counter slide: $D_1 = \bar{Q}_1$.
- The **right** flip-flop's Q output is Q_0. Its D input is driven by a **two-input XNOR gate** (OR body with the extra input-side arc, and an inversion bubble on the output). The gate's inputs are Q_1 (tapped off the left flip-flop's Q) and Q_0 (fed back from the right flip-flop's own Q). So $D_0 = \overline{Q_1 \oplus Q_0}$, i.e. $D_0 = 1$ exactly when $Q_1 = Q_0$.

[7] courses/csse2010/sequential-circuits.pdf

Tabulating:

Q_1	Q_0	$D_1 = \bar{Q}_1$	$D_0 = Q_1 \odot Q_0$	next state
0	0	1	1	11
1	1	0	1	01
0	1	1	0	10
1	0	0	0	00

Starting from 00: $00 \to 11 \to 01 \to 10 \to 00 \to \ldots$

Answer: C.

Sanity check on the gate reading — if it were a plain NOR rather than an XNOR, $00 \to 11 \to 00$ would be a two-state cycle, which is not among the options; the XNOR reading is the one that produces a listed 4-state sequence.

Key points

The summary slide's four key points (next state is a function of the previous state and possibly inputs; the count sequence need not be binary; these circuits are synchronous with all flip-flops on the same clock; asynchronous counters exist but aren't covered) are recorded in [counters].

Example worked in class

The last content slide poses:

> 2-bit counter that counts $00 \to 10 \to 01 \to 00 \to$

and is otherwise blank, marked *"(to be worked through in class)"*. The slide after it is entirely blank — presumably the space the working was meant to fill.

Unlike the earlier blank, this one *is* recoverable, because the sequence is stated. The full present-state/next-state table and the derivation of $D_1 = \bar{Q}_1\bar{Q}_0$ and $D_0 = Q_1$ (with state 11 treated as a don't-care, and the resulting counter turning out to be self-correcting) are written up as the worked example in [counters].

Next time

More sequential circuits — **state machines**. The closing slide reuses the general synchronous sequential circuit block diagram (inputs and fed-back state into a combinational circuit, whose outputs are both the circuit outputs and the flip-flop inputs; flip-flops driven by clock pulses, their outputs fed back), see sequential-circuits[8], with the annotation:

> Note that a counter may or may not have external inputs.

The final slide of the deck is empty apart from the CSSE2010 banner.

Reminders

- **Lab 7 (Counters)** runs in the P1 sessions this week (Thu-Fri) — the pre-lab schematic for the 3-bit synchronous counter must be drawn beforehand. See week4-lab-lab-7-exercises[9], and device-pinouts[10] for the flip-flop packages.
- Lab 6 (Shift Registers) runs in the P2 sessions (Mon-Tue) this week.
- The teaching outline slide also flags the centrally scheduled 90-minute in-semester theory exam on **Saturday 5 September**, and the Digital Logic **lab exam during the scheduled lab sessions in week 7**.

Lab 7 Exercises

Provided task

Pre-Lab Preparation

You should complete a circuit schematic diagram as described below before your **Lab 7 session in week 4 (Thu-Fri)**. You should consult the device pinout information on Blackboard (captured in device-pinouts[11]; the drawing rules it has to satisfy are in circuit-schematics[12]). You will be testing this circuit (or a similar circuit) during the prac session.

Design Requirements

Design and draw a circuit schematic diagram for a 3-bit synchronous counter which counts through your allocated sequence (see table below).
Hardware Specifications:

[8] courses/csse2010/sequential-circuits.pdf
[9] courses/csse2010/week4-lab-lab-7-exercises.pdf
[10] courses/csse2010/device-pinouts.pdf

- **Clock:** You must use a push button for the clock signal.
- **Preset:** Use a single switch to allow your counter to be preset (i.e., to `111`).
- **Clear:** Use a single switch to allow your counter to be cleared (i.e., to `000`).
- **Outputs:** Your count output should be shown on three LEDs.
  - Q_2**:** Most Significant Bit (MSB)
  - Q_1**:** Middle bit
  - Q_0**:** Least Significant Bit (LSB)

  Note: Your allocated sequence has seven binary numbers; you may choose to do whatever you like with the missing number – i.e., if the counter ever has this value, then we treat the next count value as a "don't care" condition.

It is highly recommended that you simulate your circuit using Logisim to ensure it counts through the expected sequence.

Count Sequences

Find the sequence corresponding to the last digit of your 8-digit UQ student number:

Table 1: Count sequence by last digit of student ID

Last digit of UQ ID	Count Sequence
0	`111 -> 011 -> 101 -> 110 -> 001 -> 000 -> 010 -> 111 -> …`
1	`111 -> 010 -> 101 -> 110 -> 100 -> 000 -> 001 -> 111 -> …`
2	`000 -> 011 -> 010 -> 110 -> 100 -> 001 -> 111 -> 000 -> …`
3	`111 -> 100 -> 101 -> 000 -> 110 -> 001 -> 011 -> 111 -> …`
4	`000 -> 101 -> 011 -> 010 -> 001 -> 110 -> 100 -> 000 -> …`
5	`000 -> 110 -> 001 -> 100 -> 111 -> 011 -> 010 -> 000 -> …`
6	`111 -> 010 -> 000 -> 110 -> 001 -> 101 -> 100 -> 111 -> …`
7	`111 -> 001 -> 110 -> 101 -> 100 -> 000 -> 010 -> 111 -> …`
8	`111 -> 000 -> 001 -> 101 -> 011 -> 110 -> 100 -> 111 -> …`
9	`000 -> 100 -> 001 -> 110 -> 101 -> 011 -> 010 -> 000 -> …`

Working

My student number is ——**6**, so I will be sequencing: `111 -> 010 -> 000 -> 110 -> 001 -> 101 -> 100 -> 111 -> …`

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

[12] courses/csse2010/circuit-schematics.pdf

Let's map the current and next states:

Table 2: Current and next state

Current State ($Q_2Q_1Q_0$)	Next State ($D_2D_1D_0$)	Comment
0 0 0	1 1 0	$000 \to 110$
0 0 1	1 0 1	$001 \to 101$
0 1 0	0 0 0	$010 \to 000$
0 1 1	X X X	unused state
1 0 0	1 1 1	$100 \to 111$
1 0 1	1 0 0	$101 \to 100$
1 1 0	0 0 1	$110 \to 001$
1 1 1	0 1 0	$111 \to 010$

Deriving D_2

Rows where $D_2 = 1$:

$Q_2Q_1Q_0$	D_2
0 0 0	1
0 0 1	1
1 0 0	1
1 0 1	1

Each row gives one product term (1 for that row, 0 elsewhere):

$$
\begin{aligned}
D_2 &= \bar{Q}_2 \cdot \bar{Q}_1 \cdot \bar{Q}_0 + \bar{Q}_2 \cdot \bar{Q}_1 \cdot Q_0 + Q_2 \cdot \bar{Q}_1 \cdot \bar{Q}_0 + Q_2 \cdot \bar{Q}_1 \cdot Q_0 \\
&= \bar{Q}_2 \cdot \bar{Q}_1 \left(\bar{Q}_0 + Q_0\right) + Q_2 \cdot \bar{Q}_1 \left(\bar{Q}_0 + Q_0\right) && \text{(Distributive law)} \\
&= \bar{Q}_2 \cdot \bar{Q}_1 \cdot \mathbf{t} + Q_2 \cdot \bar{Q}_1 \cdot \mathbf{t} && \text{(Negation law)} \\
&= \bar{Q}_2 \cdot \bar{Q}_1 + Q_2 \cdot \bar{Q}_1 && \text{(Identity law)} \quad (1) \\
&= \bar{Q}_1 \left(\bar{Q}_2 + Q_2\right) && \text{(Distributive law)} \\
&= \bar{Q}_1 \cdot \mathbf{t} && \text{(Negation law)} \\
&= \bar{Q}_1 && \text{(Identity law)}
\end{aligned}
$$

D_2 is just $\bar{Q}_1$, so it needs no gate at all. FF1's $\overline{Q}$ output wires straight to FF2's D input.

Deriving D_1

Sum of products where $D_1 = 1$ (`000`, `100`, `111`):

$$\begin{aligned} D_1 &= \bar{Q}_2 \cdot \bar{Q}_1 \cdot \bar{Q}_0 + Q_2 \cdot \bar{Q}_1 \cdot \bar{Q}_0 + Q_2 \cdot Q_1 \cdot Q_0 \\ &= \bar{Q}_1 \cdot \bar{Q}_0 \left(\bar{Q}_2 + Q_2\right) + Q_2 \cdot Q_1 \cdot Q_0 && \text{(Distributive law)} \\ &= \bar{Q}_1 \cdot \bar{Q}_0 \cdot \mathbf{t} + Q_2 \cdot Q_1 \cdot Q_0 && \text{(Negation law)} \\ &= \bar{Q}_1 \cdot \bar{Q}_0 + Q_2 \cdot Q_1 \cdot Q_0 && \text{(Identity law)} \end{aligned} \tag{2}$$

Deriving D_0

Sum of products where $D_0 = 1$ (`001`, `100`, `110`), plus the unused state `011` — its next state is a don't care, so I am free to take $D_0 = 1$ there, which lets Q_1 cancel:

$$\begin{aligned} D_0 &= \bar{Q}_2 \cdot \bar{Q}_1 \cdot Q_0 + \bar{Q}_2 \cdot Q_1 \cdot Q_0 + Q_2 \cdot \bar{Q}_1 \cdot \bar{Q}_0 + Q_2 \cdot Q_1 \cdot \bar{Q}_0 \\ &= \bar{Q}_2 \cdot Q_0 \left(\bar{Q}_1 + Q_1\right) + Q_2 \cdot \bar{Q}_0 \left(\bar{Q}_1 + Q_1\right) && \text{(Distributive law)} \\ &= \bar{Q}_2 \cdot Q_0 \cdot \mathbf{t} + Q_2 \cdot \bar{Q}_0 \cdot \mathbf{t} && \text{(Negation law)} \\ &= \bar{Q}_2 \cdot Q_0 + Q_2 \cdot \bar{Q}_0 && \text{(Identity law)} \\ &= Q_2 \oplus Q_0 && \text{(definition of XOR)} \end{aligned} \tag{3}$$

$\bar{Q}_2 \cdot Q_0 + Q_2 \cdot \bar{Q}_0$ is exclusive-or, so D_0 is a single XOR gate.

Checking Robustness

`011` was left unused in Table 2, so its next state was treated as a don't care when deriving Equation 1, Equation 2 and Equation 3. The circuit will still compute *something* for this state, so I need to check it doesn't get stuck there. Substituting $Q_2 Q_1 Q_0 = 011$ into each equation:

$$\begin{aligned} D_2 &= \bar{Q}_1 \\ &= \bar{1} \\ &= 0 \end{aligned} \tag{4}$$

$$\begin{aligned} D_1 &= \bar{Q}_1 \cdot \bar{Q}_0 + Q_2 \cdot Q_1 \cdot Q_0 \\ &= \bar{1} \cdot \bar{1} + 0 \cdot 1 \cdot 1 \\ &= 0 \cdot 0 + 0 \\ &= 0 \end{aligned} \tag{5}$$

$$
\begin{aligned}
D_0 &= Q_2 \oplus Q_0 \\
&= 0 \oplus 1 \\
&= 1
\end{aligned}
\tag{6}
$$

So $D_2 D_1 D_0 = 001$, meaning `011 -> 001`. This is not `011` itself, and `001` is already part of my sequence, so the counter does not lock up in the unused state: it self-corrects back into the main count sequence within one clock cycle.

Taking the don't care as a 1 in D_1 as well would have simplified it to $\overline{Q_1 \oplus Q_0}$, but then `011 -> 011` and the counter would hang, so I only used it for D_0.

Conclusion

As per Equation 1, Equation 2 and Equation 3, the excitation equations for the counter are:

$$
\begin{aligned}
D_2 &= \bar{Q}_1 \\
D_1 &= \bar{Q}_1 \cdot \bar{Q}_0 + Q_2 \cdot Q_1 \cdot Q_0 \\
D_0 &= Q_2 \oplus Q_0
\end{aligned}
\tag{7}
$$

and the robustness check in Equation 4, Equation 5 and Equation 6 confirms that the one unused state, `011`, transitions to `001` rather than to itself. The counter is therefore self-correcting, and these three equations fully and safely implement my allocated count sequence.

That comes to five 2-input gates across five chips. The breadboard only fits six, so this leaves one spare. No inverters are needed anywhere: the `74HCT74` gives $\overline{Q}$ alongside Q, so $\overline{Q_1}$ and $\overline{Q_0}$ are free.

Circuit Schematic

Figure 1 is a logic diagram. Figure 2 is the same counter drawn as a circuit schematic (see circuit-schematics[13]), with device IDs, chip types and pin numbers from device-pinouts[14].

It takes five chips: `U1` and `U2` are `74HCT74` dual flip-flops, `U3` a `74HCT08` (the 3-input product is `U3:B` and `U3:C` cascaded, since the kit has no 3-input gate), `U4:A` a `74HCT32` and `U5:A` a `74HCT86`. On the IO board, CLOCK is button `B0`, PRESET and CLEAR are switches `S0` and `S1`, and $Q_2 Q_1 Q_0$ drive LEDs `L2`, `L1` and `L0`.

[13] courses/csse2010/circuit-schematics.pdf
[14] courses/csse2010/device-pinouts.pdf

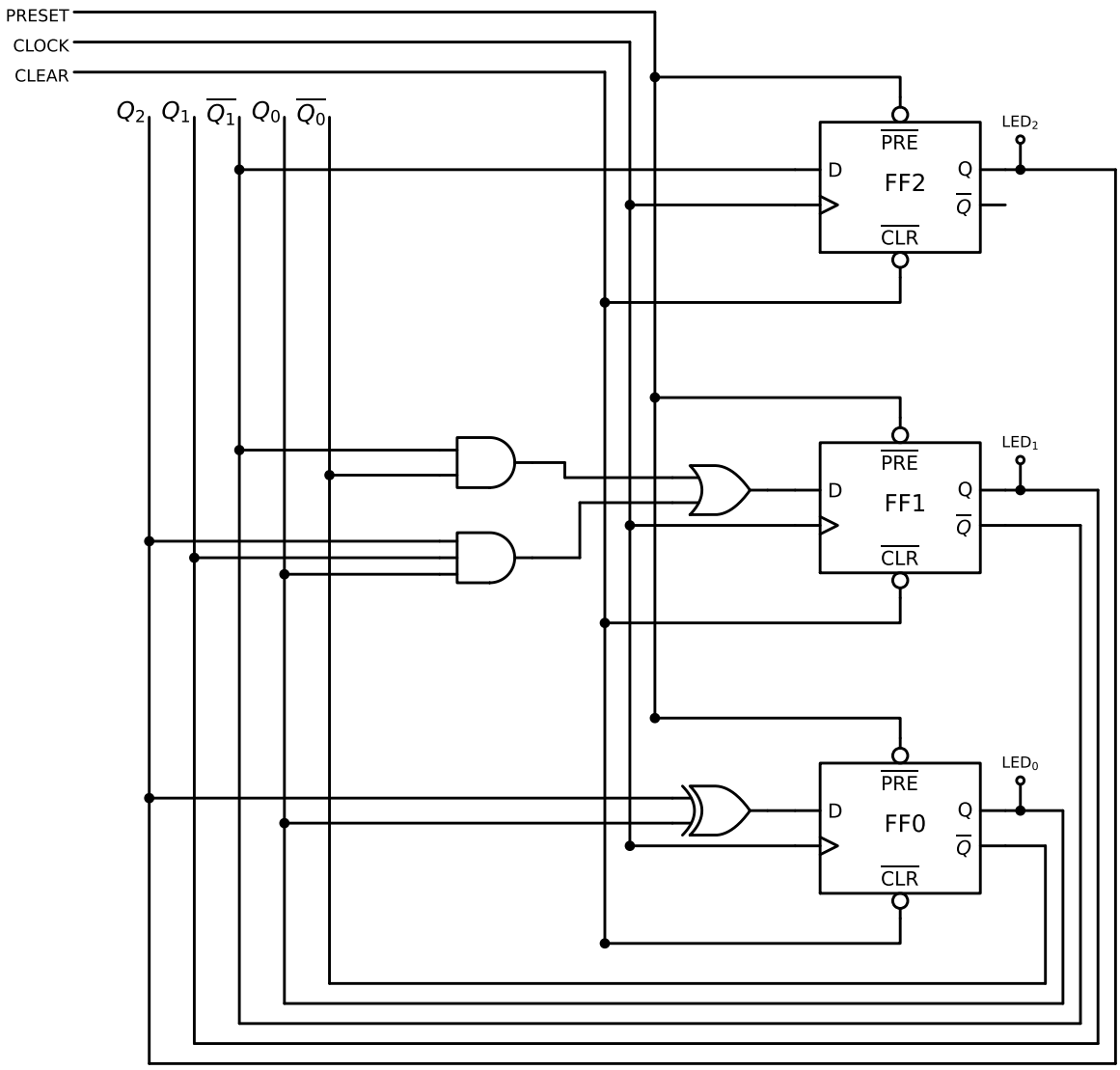

Figure 1: The complete counter: the gates of Equation 7 on the left, the three flip-flops on the right, and $Q/\overline{Q}$ fed back to the input rails. D_2 has no gate: it is FF1's $\overline{Q}$ wired straight across. CLOCK, PRESET and CLEAR come from the push button and the two switches; each Q drives one of the three LEDs.

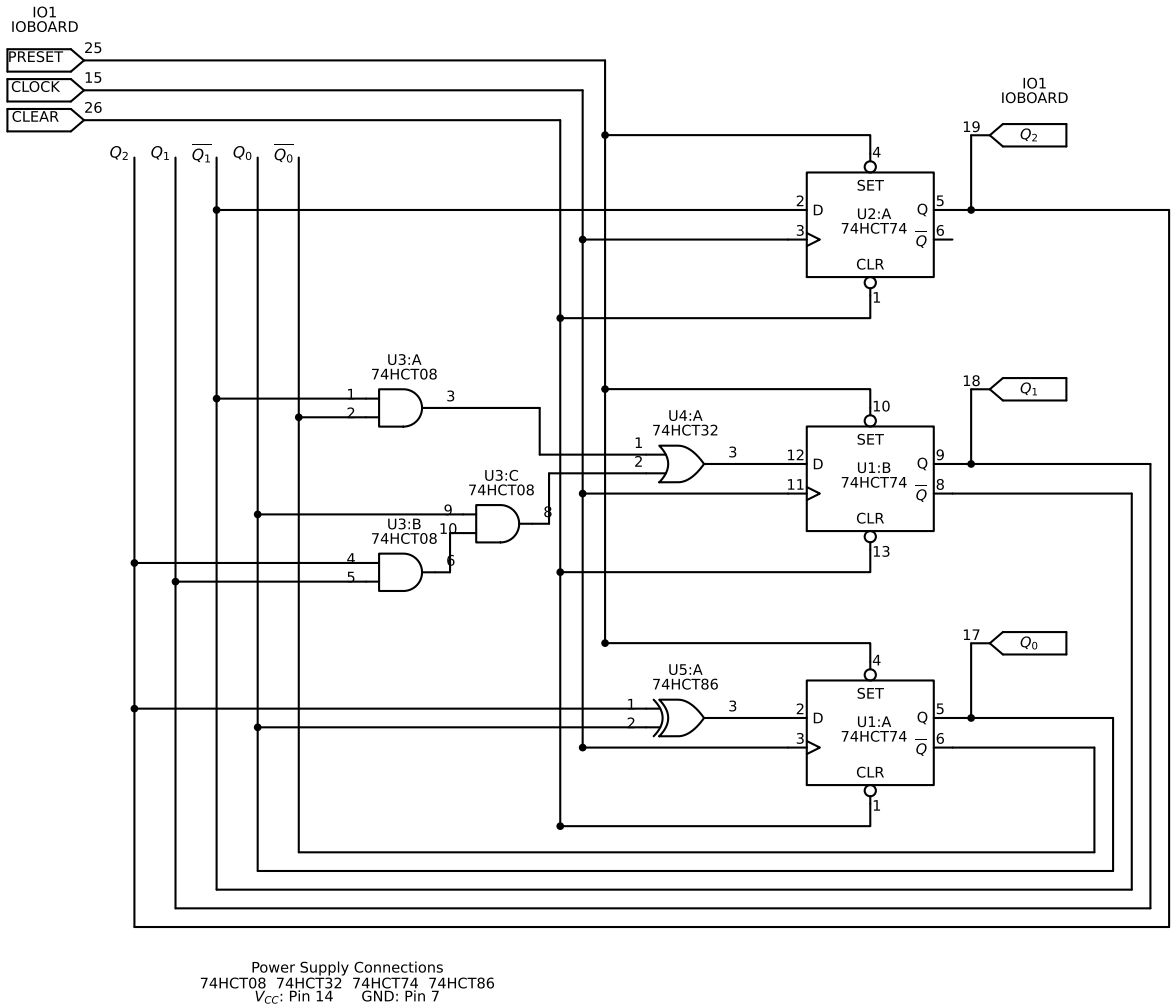

Figure 2: The counter as a circuit schematic, ready to build: device IDs and chip types on every gate, pin numbers on every connection, IO board symbols for the inputs and outputs, and the power connections given once for all four chip types. A dot marks a joined connection; crossing wires are not connected.

Reference material

Circuit Schematics

A **logic diagram** shows the *idea* of a circuit and is hardware-independent (see logic-gates[15]). 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[16] 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.

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[17]).
- **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.

[15] courses/csse2010/logic-gates.pdf
[16] courses/csse2010/device-pinouts.pdf
[17] courses/csse2010/device-pinouts.pdf

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.
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[18] for the gate symbols/truth tables and boolean-algebra[19] 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[20], 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).

[18] courses/csse2010/logic-gates.pdf
[19] courses/csse2010/boolean-algebra.pdf
[20] courses/csse2010/sequential-circuits.pdf

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[21]).

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.

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**.

[21] courses/csse2010/binary-number-representations.pdf

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[22]) plus four XOR gates:

- Data input A ($A_0 \dots A_3$) goes **straight** into the adder's A inputs.
- Data input B ($B_0 \dots B_3$) 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 $S_0 \dots S_3$, 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

$$C_0 = M$$

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[23].

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 $D_0 \dots D_3$, 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:

[22] courses/csse2010/2026-08-04-binary-arithmetic.pdf
[23] courses/csse2010/device-pinouts.pdf

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.

That is exactly the sum-of-products form:

$$F = D_0\bar{S}_1\bar{S}_0 + D_1\bar{S}_1 S_0 + D_2 S_1\bar{S}_0 + D_3 S_1 S_0$$

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

S_0	D_0	D_1	F
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. $F = D_0\bar{S}_0 + D_1 S_0$. 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[24].

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[25].

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

[24] courses/csse2010/device-pinouts.pdf
[25] courses/csse2010/2026-08-06-combinational-logic.pdf

A	B	C	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
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:

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

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

Counters

A **counter** is a multi-bit register that goes through a *determined sequence of states* (values) upon the application of input (clock) pulses. It is a [[sequential-circuits|synchronous sequential circuit]] built from [[flip-flops-and-latches|D flip-flops]] plus some combinational logic in the feedback path.

Binary counters

A counter which follows the binary number sequence is a **binary counter**.

An n**-bit binary counter**:

- has n flip-flops;
- has 2^n different states;
- counts from 0 to $2^n - 1$, then wraps back to 0.

For example, a 2-bit binary counter counts

$$00 \to 01 \to 10 \to 11 \to 00 \to ...$$

i.e. $0 \to 1 \to 2 \to 3 \to 0 \to ...$

4-bit binary counter — counting sequence

Bit 3	Bit 2	Bit 1	Bit 0	Value
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15
0	0	0	0	0

Note how each bit toggles at half the rate of the bit to its right — bit 0 changes every count, bit 1 every second count, bit 2 every fourth, bit 3 every eighth.

Key points

- The **next state is a function of the present state** (and possibly of external inputs).
- The count sequence **can** be the binary numbers but **does not have to be** — if it is, the counter is a binary counter; otherwise it is just some arbitrary cycle through states.
- These circuits are **synchronous**: all flip-flops share the same clock.
- **Asynchronous** counters exist too, but are not discussed in this course.

State: the design idea

This is the whole trick, and it is what makes counter design mechanical:

- The values stored in the flip-flops are the **current (present) state** of the circuit.
- The D inputs to the flip-flops are the **next state** — whatever is sitting on D at the clock edge becomes the new state.
- The D inputs are some **combinational function of the current state** (and of any external inputs).

So for a counter with state bits $Q_{n-1} \ldots Q_1 Q_0$, designing the counter means finding n Boolean expressions

$$D_i = f_i(Q_{n-1}, \ldots, Q_1, Q_0)$$

and building each one out of gates[26].

Design recipe

1. **Write down the count sequence** you want, as a cycle of state values.
2. **Build a present-state / next-state table.** One row per possible state — all 2^n of them, not just the ones in your sequence. Columns: present state $Q_{n-1} \ldots Q_0$, then next state $D_{n-1} \ldots D_0$. Fill each next-state entry by reading off what follows that state in your sequence.
3. **Handle unused states.** If your sequence doesn't use all 2^n states, the next state for the unused ones is a **don't-care** (X). You may set each X to whatever makes the algebra simplest — but check afterwards where the unused state actually goes, because a self-correcting counter (one whose unused states lead back into the cycle) is nicer than one that can get stuck.
4. **Read each D column as a Boolean function of the Qs.** The table *is* a truth table with the Qs as inputs and D_i as the output, so write it in sum-of-products form: OR together one AND term per row where $D_i = 1$. See boolean-algebra[27].
5. **Simplify** each expression using the Boolean algebra laws (and the don't-cares), then draw the circuit: the combinational logic for D_i feeding flip-flop i, with every flip-flop on a **common clock**. See circuit-schematics[28] for schematic conventions and device-pinouts[29] for the 74HCT74 / 74HCT175 D flip-flop packages.

[26] courses/csse2010/logic-gates.pdf
[27] courses/csse2010/boolean-algebra.pdf
[28] courses/csse2010/circuit-schematics.pdf
[29] courses/csse2010/device-pinouts.pdf

The smallest case falls straight out of the recipe: a **1-bit counter** counts $0 \to 1 \to 0 \to \ldots$, so its next state is always the complement of its present state, giving $D = \bar{Q}$ — a single flip-flop with its $\bar{Q}$ output wired back to its own D input (a *toggle*).

Worked example — a 2-bit counter for $00 \to 10 \to 01 \to 00$

> **Note**
>
> This example is posed on the lecture slides but left blank ("to be worked through in class"). The derivation below is reconstructed, not copied from the slides.

Step 1–2 — present-state / next-state table. The sequence uses three of the four states; 11 is unused, so its next state is a don't-care.

Q_1	Q_0	D_1	D_0	comment
0	0	1	0	$00 \to 10$
0	1	0	0	$01 \to 00$
1	0	0	1	$10 \to 01$
1	1	X	X	unused state

Step 4 — read off the columns.

D_1 is 1 only in the row $Q_1Q_0 = 00$. Making the don't-care 1 would give $Q_1 \odot Q_0$ (XNOR), which is no simpler, so take the don't-care as 0:

$$D_1 = \bar{Q}_1\bar{Q}_0$$

D_0 is 1 only in the row $Q_1Q_0 = 10$, which gives $D_0 = Q_1\bar{Q}_0$. Here the don't-care *does* help: setting it to 1 makes $D_0 = 1$ for both rows with $Q_1 = 1$, so

$$D_0 = Q_1$$

Step 5 — the circuit. Two D flip-flops on a common clock. Flip-flop 0's D input is wired directly to flip-flop 1's Q_1 output — no gate at all. Flip-flop 1's D input is driven by a 2-input NOR of Q_1 and Q_0 (since $\bar{Q}_1\bar{Q}_0 = \overline{Q_1 + Q_0}$), or equivalently by an AND of the two $\bar{Q}$ outputs.

Check, including the unused state:

present	$D_1 D_0$	next
00	1 0	10
10	0 1	01
01	0 0	00
11	0 1	01

The cycle is right, and the unused state 11 falls into the cycle at 01 on the next clock edge, so the counter is **self-correcting**. (Choosing $D_0 = Q_1\bar{Q}_0$ instead — don't-care set to 0 — would send $11 \to 00$, also fine, at the cost of one more gate input.)

Applied exercise

week4-lab-lab-7-exercises[30] is exactly this recipe at $n = 3$: design a 3-bit synchronous counter that steps through an allocated 7-state sequence, with the eighth (missing) state treated as a don't-care.

Related

- flip-flops-and-latches[31] — the D flip-flop and its characteristic table.
- sequential-circuits[32] — state, present state / next state, synchronous sequential circuits.
- shift-registers[33] — the other family of registers built from a flip-flop chain.
- boolean-algebra[34] — sum of products and the simplification laws used in step 4–5.
- binary-number-representations[35] — the binary counting sequence itself.

Device Pinouts

Pin numbers for the 74-series logic chips and IO board used in circuit-schematics[36]. 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.

[30] courses/csse2010/week4-lab-lab-7-exercises.pdf
[31] courses/csse2010/flip-flops-and-latches.pdf
[32] courses/csse2010/sequential-circuits.pdf
[33] courses/csse2010/shift-registers.pdf
[34] courses/csse2010/boolean-algebra.pdf
[35] courses/csse2010/binary-number-representations.pdf
[36] courses/csse2010/circuit-schematics.pdf

Power and ground

Power/ground connections are shown separately from gate pins on a schematic (see circuit-schematics[37]): 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

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.

[37] courses/csse2010/circuit-schematics.pdf

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[38]). Use sensible

[38] courses/csse2010/circuit-schematics.pdf

signal names rather than the raw pin labels where possible.

Flip-Flops and Latches

Flip-flops and latches are the **storage elements** — the circuits that let a design remember a value even after the input that produced it has changed. They're what turns a combinational circuit into a [[sequential-circuits|sequential circuit]].

The D flip-flop

The D flip-flop is the **only** flip-flop type covered in this course. Its symbol is a rectangle with:

- **D** — the data input, on the left.
- **Q** — the output, on the right.
- **CLK** — the clock, a *control* input on the left, drawn with a small triangle where it meets the box.

How it works: **Q copies the value of D (and remembers it) whenever CLK goes from 0 to 1** — the *rising edge*. In between clock edges the flip-flop is not responsive, so it holds the stored value. Q changes **only** on that 0→1 transition.

Summary: a D flip-flop remembers a single bit — either a "1" or a "0". It keeps that value until the next clock edge, at which point the D input is transferred to output Q.

- To remember n bits you need n D flip-flops.
- An **n-bit register** is (by definition) n D flip-flops. See shift-registers[39].
- Flip-flops can themselves be made out of logic gates.

Characteristic table

A **characteristic table** defines the operation of a flip-flop in tabular form. The left column is the input; the right column, $Q(t+1)$, is what the output will be **on the next clock edge** (i.e. when CLK goes 0→1).

D	Q(t+1)	
0	0	Reset
1	1	Set

That is the whole behaviour: the flip-flop is *reset* (to 0) or *set* (to 1) according to D, at the clock edge.

[39] courses/csse2010/shift-registers.pdf

Worked waveform

The lecture's example: D is 0, 0, then 1, 1 across a clock with two rising edges; CLK starts low, pulses high, goes low, then pulses high again after D has risen.

Clock edge	D at that edge	Q after
(before the first edge)	—	unknown/undefined
1st rising edge	0	0
2nd rising edge	1	1

Before the first rising edge Q is marked with an "X" on the slide — its value is undefined, because nothing has been clocked in yet. Between the two rising edges Q stays 0 even though D has already risen to 1: the flip-flop is only sensitive at the edge.

Other flip-flop types

Other types exist — **JK flip-flops** and **T flip-flops** — but they are **not covered in this course**. Only the D flip-flop is used.

Latches vs flip-flops

- **Latches** are **level triggered** devices — they latch the output and respond to changes of logic *levels* on the inputs.
- A latch circuit can be modified so that it becomes sensitive to an **edge** (a momentary transition) of a control input, i.e. a clock signal. Such circuits are called **flip-flops**.
- A flip-flop can store 1 bit of information while being sensitive to a clock edge — it changes its output only *at* the clock edges, based on the inputs.
- So: **latches are level triggered; flip-flops are edge triggered.**

A clock signal has two edges:

Edge	Transition
Positive (rising) edge	$0 \rightarrow 1$
Negative (falling) edge	$1 \rightarrow 0$

A D flip-flop can therefore be **positive edge triggered** or **negative edge triggered**. In a positive edge triggered D flip-flop, D is copied to Q at the positive edge of the clock; in between clock edges the flip-flop is not responsive, thus stores the value.

The **state** of a flip-flop is the value it is storing. Flip-flops are more useful than latches in practice.

The SR latch (cross-coupled NOR gates)

Structure: two 2-input NOR gates cross-coupled.

- The **top** NOR gate takes S as one input and the *bottom* gate's output as its other input; its output is $\bar{Q}$.
- The **bottom** NOR gate takes R as one input and the *top* gate's output ($\bar{Q}$) as its other input; its output is Q.

So each gate's output feeds back into the other gate's input — that feedback loop is what stores the bit. Recall (from logic-gates[40]) that a NOR gate outputs 1 only when **both** inputs are 0; any 1 on an input forces the output to 0.

The truth table on the slide is **left blank** ("to be completed in class"). Derived from the NOR behaviour and the cross-coupling, assuming a stable starting state Q:

S	R	Q	$\bar{Q}$	Meaning
0	0	Q (unchanged)	$\bar{Q}$ (unchanged)	**Hold** — remembers the previous value
0	1	0	1	**Reset** — Q forced to 0
1	0	1	0	**Set** — Q forced to 1
1	1	0	0	**Invalid** — see below

Reasoning for each row:

- $S = 0, R = 0$: neither gate is forced. Say $Q = 1$; then the top gate sees $S = 0$ and $Q = 1$, so $\bar{Q} = 0$; the bottom gate sees $R = 0$ and $\bar{Q} = 0$, so $Q = 1$ — consistent. The same argument works with $Q = 0$. Both states are self-consistent, so the latch simply holds whatever it already had.
- $S = 1, R = 0$: the top gate has a 1 on an input, so $\bar{Q} = 0$. The bottom gate then sees $R = 0$ and $\bar{Q} = 0$, so $Q = 1$. S **sets** the latch.
- $S = 0, R = 1$: mirror image — the bottom gate has a 1 on an input, so $Q = 0$; the top gate then sees $S = 0$ and $Q = 0$, so $\bar{Q} = 1$. R **resets** the latch.
- $S = 1, R = 1$: both gates have a 1 on an input, so both outputs are 0 — $Q = \bar{Q} = 0$. This breaks the whole point of the two outputs being complements, which is why the combination is called forbidden/invalid. What happens *after* both inputs return to 0 simultaneously is not uniquely determined (the two gates race), so the resulting state is

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

indeterminate. The slide is blank here and the worked answer was only done live — check the lecture recording for the exact form the lecturer wrote in this row.

Homework: latches from NAND gates

Slide 11 is otherwise blank and sets a homework: *analyse the S-R latch circuit from the previous slide when the NOR gates are replaced with NAND gates, and complete the truth table.*

The slide gives no answer. Derived, keeping the same wiring and the same input labels (top gate: S and the bottom output; bottom gate: R and the top output), and recalling that a NAND outputs 0 only when **both** inputs are 1 — so any **0** on an input forces the output to **1**:

S	R	top output	bottom output	Meaning
1	1	unchanged	unchanged	**Hold**
0	1	1	0	bottom output forced to 0
1	0	0	1	bottom output forced to 1
0	0	1	1	**Invalid** — outputs no longer complementary

The headline result: with NAND gates the inputs become **active low** (the latch holds at $S = R = 1$ rather than $S = R = 0$, and the invalid combination moves to $S = R = 0$), and with the labels left unchanged the set/reset roles swap over relative to the NOR version. This is why NAND latches are conventionally drawn with the inputs labelled $\bar{S}$ and $\bar{R}$. Worth confirming against the lecturer's own labelling.

A real D flip-flop

A real D flip-flop is built from cross-coupled NAND gates — the lecture's schematic uses six NAND gates. As well as D, CLK and the outputs Q and $\bar{Q}$, it has two extra inputs:

- $\overline{\textbf{PRE}}$ **(Preset)** — forces Q to 1.
- $\overline{\textbf{CLR}}$ **(Clear)** — forces Q to 0.

Both are drawn with overbars, i.e. they are **active low**, and both are **asynchronous**: they act on the latch directly, independent of the clock, rather than waiting for a clock edge. In the schematic they feed into the NAND gates of the input latches directly, bypassing the D/CLK path.

Symbols used

Four related symbols, all drawn as a rectangle with **D** in at the top left, **Q** out at the top right, and a clock input **CK** at the bottom left. What distinguishes them is the marking on that CK input:

Symbol	CK input marking	Device	Triggered on
(a)	plain input	Latch	High level of CK
(b)	**bubble** (inversion circle)	Latch	Low level of CK
(c)	**triangle**	Flip-flop	**Rising** (positive) edge of the clock
(d)	**bubble + triangle**	Flip-flop	**Falling** (negative) edge of the clock

The two markings read independently and are the whole key to the notation:

- The **triangle** indicates **edge-triggered**, and therefore that the device is a flip-flop rather than a latch. No triangle means level-triggered, i.e. a latch.
- The **bubble** indicates inversion, i.e. the *falling* edge (or the low level) rather than the rising edge (or the high level).

The slide only annotates (c) and (d) explicitly; the readings given above for (a) and (b) are derived from those two rules.

D flip-flop chips

- **74HCT74** — dual D flip-flop (two independent D flip-flops in a 14-pin package), each with CLR and PR (preset) inputs and $Q/\bar{Q}$ outputs. Pinout in device-pinouts[41].
- **74HCT273** — eight D flip-flops in a 20-pin package, so it can hold **one byte** (8 bits) of information.

The lecture points to the "device symbols PDF on Blackboard" for the full symbol set.

Sequential Circuits

A **sequential circuit** is a circuit whose output depends not only on the current inputs but also on the values it is currently storing. The storage is done with flip-flops — see flip-flops-and-latches[42] for the storage elements themselves.

[41] courses/csse2010/device-pinouts.pdf
[42] courses/csse2010/flip-flops-and-latches.pdf

Combinational vs sequential

	Combinational	Sequential
Contents	Logic gates only (**no** flip-flops)	Includes **flip-flops** as well as logic gates
Output determined by	The inputs alone — uniquely	Current inputs **and** current **state**
Repeatability	Same inputs always give the same output	Same inputs can give different outputs, depending on state
When output changes	Whenever an input changes	Only when the clock 'ticks'
Examples	Adders, multiplexers, decoders, demultiplexers, encoders (see combinational-logic-blocks[43])	Counters, registers (see [counters], shift-registers[44])

A combinational example from earlier in the course: A into one input of an AND gate, B and C into an OR gate whose output feeds the AND gate's other input, giving $X = A(B + C)$. Nothing in that circuit remembers anything — change A, B or C and the output follows immediately (after propagation delay; see timing-diagrams[45]).

The key contrast: if an input to a combinational circuit changes, the output can change too and the **previous value is lost forever**. A sequential circuit can hold onto a value even after the input that produced it has gone away.

State

- **State** = the value stored in the flip-flops.
- **Output** depends on the inputs *and* the state.
- **Next state** depends on the inputs *and* the (present) state.

"Present state" is what the flip-flops hold right now; "next state" is what they will hold after the next clock edge. The combinational logic computes the next state from the present state and the inputs; the flip-flops only adopt it when the clock edge arrives.

[43] courses/csse2010/combinational-logic-blocks.pdf
[44] courses/csse2010/shift-registers.pdf
[45] courses/csse2010/timing-diagrams.pdf

General structure of a sequential circuit

The standard block diagram has two blocks and a feedback loop:

- A **combinational circuit** block (just logic gates). It takes the external **inputs** *plus* the feedback from the flip-flops, and produces the external **outputs** *plus* the values to be loaded into the flip-flops.
- A **flip-flops** block (the storage elements). Its data inputs come from the combinational circuit; it also takes **clock pulses** as a separate control input.
- A **feedback path** runs from the flip-flops' outputs back around into the combinational circuit's inputs. That loop is what makes the circuit sequential — the stored state is fed back in and influences the next output and the next state.

Note that the clock pulses go **only** to the flip-flops, not to the combinational logic.

Synchronous sequential circuits

In a **synchronous** sequential circuit the storage elements can only change at **discrete instants of time**, set by a common clock signal:

- Assume a clock signal — a regularly repeating square wave alternating between 0 and 1.
- The outputs of the storage elements change **only on the edges** of that control signal.
- Contrast with logic gates, whose outputs change whenever their inputs change.

Because every flip-flop shares the same clock, the whole circuit's state changes in one step at each clock edge, which is what makes such circuits tractable to design and analyse.

Shift Registers

Registers and shift registers are the simplest useful [[sequential-circuits|sequential circuits]] — groups of [[flip-flops-and-latches|D flip-flops]] sharing a common clock.

Registers

- A **register** is a group of flip-flops. An n**-bit register** consists of n flip-flops and is capable of storing n bits.
- A register is a sequential circuit **without any combinational logic** — it is just the storage elements and their shared control lines.
- Registers are used to store binary information (data/instructions) **inside a processor**.

Structure of the 4-bit register example (Mano, *Digital Design*, 3rd ed.):

Signal	Connection
$I_0 \ldots I_3$	the four parallel data inputs, one to each flip-flop's D input
$A_0 \ldots A_3$	the four parallel outputs, taken from each flip-flop's Q
`Clock`	a single common line driven to the C (clock) input of **all four** flip-flops
`Clear`	a single common line driven to the R (reset) input of all four flip-flops, drawn **active-low** (bubble on the input)

So on each rising clock edge all four flip-flops simultaneously capture their I inputs; asserting `Clear` forces all four outputs to 0 asynchronously (see flip-flops-and-latches[46] for asynchronous SET/CLR).

Shift registers

> A **shift register** is a register which is capable of shifting its binary information in one or both directions.

The 4-bit shift register example is built as a **chain**: the serial input SI feeds the D input of the first flip-flop; each flip-flop's Q output feeds the next flip-flop's D input; the last flip-flop's Q is the serial output SO. All four flip-flops share a common CLK line.

On each rising clock edge every stored bit moves one place along the chain, the bit currently on SI enters the first stage, and the bit that was in the last stage leaves at SO.

Worked through by hand, with stages labelled Q_0 (nearest SI) through Q_3 (= SO), starting from all zeros and shifting in the bit sequence 1, 0, 1, 1:

Clock edge	SI	Q_0	Q_1	Q_2	Q_3 (= SO)
(initial)	—	0	0	0	0
1	1	1	0	0	0
2	0	0	1	0	0
3	1	1	0	1	0
4	1	1	1	0	1

After 4 clock edges the 4 serially-supplied bits sit in the 4 flip-flops, and one further edge would push the first of them out of SO.

[46] courses/csse2010/flip-flops-and-latches.pdf

Serial parallel conversion

Shift registers can be used to do **serial-to-parallel conversion, and vice versa**. This is the reason they show up everywhere data has to travel over a single wire but be *used* a word at a time.

- **Serial in, parallel out**: drive the bits one per clock edge into SI; after n edges, the n flip-flop Q outputs — read as a group — are the parallel word. This is exactly the table above: clock 4 bits in, then read $Q_0Q_1Q_2Q_3$ in parallel.
- **Parallel in, serial out**: load the n bits into the flip-flops in one clock edge (a parallel load — see below), then clock n more times, reading one bit per edge off SO.

> **Note**
>
> The lecture slide for this figure is blank and marked *"figure to be completed in class"*. The description above is derived from the standard construction: the parallel outputs are simply the Q pins of the flip-flops already present in the 4-bit shift register, brought out as a group.

Parallel load vs serial shift

A plain shift register can only be filled one bit per clock. To also allow **parallel load** — all n bits written at once from n parallel inputs — each flip-flop's D input is fed from a **2-to-1 multiplexer** (see combinational-logic-blocks[47]) instead of directly from the previous stage:

Mux select	Mux data input chosen	Effect on that stage
"shift"	the previous stage's Q (or SI for the first stage)	serial shift by one place
"load"	that stage's parallel data input I_i	parallel load of the whole word in one clock edge

The select line is common to all the muxes, so on each clock edge the register either shifts by one place or loads the whole parallel word.

> **Note**
>
> The lecture slide for this figure is also blank and marked *"figure to be completed in class"*, so the exact drawing done in the lecture is not recoverable from the deck. The mux-per-flip-flop construction above is derived, and is confirmed by the following slide,

[47] courses/csse2010/combinational-logic-blocks.pdf

which refers back to “the same multiplexer concept”.

The mux-based bidirectional shift element

The lecture gives a single building block for a shift register that shifts in **either** direction:

- a **2-to-1 multiplexer** whose output drives the D input of one D flip-flop;
- the mux's select line is a control signal called **DIRN**;
- **DIRN = 0** selects mux input **0** → **left shift**;
- **DIRN = 1** selects mux input **1** → **right shift**;
- the flip-flop's Q is that stage's output, and the flip-flop is clocked from the common clock.

Chaining n of these elements and wiring, for each stage, mux input 0 to the neighbour on the left-shift side and mux input 1 to the neighbour on the right-shift side gives an n-bit **bidirectional shift register**: one shared DIRN line decides which way the whole register shifts on the next clock edge.

For a 3-bit register with bits $Q_2Q_1Q_0$ (Q_2 most significant), taking *left shift* to mean bits move towards the more-significant end:

Stage	Mux input 0 (DIRN=0, left shift)	Mux input 1 (DIRN=1, right shift)
Q_2	Q_1	SI_R (serial input for right shifts)
Q_1	Q_0	Q_2
Q_0	SI_L (serial input for left shifts)	Q_1

One clock edge applied to a 3-bit register holding $Q_2Q_1Q_0 = 110$, with both serial inputs held at 0:

DIRN	Operation	Before ($Q_2Q_1Q_0$)	After ($Q_2Q_1Q_0$)
0	left shift	110	100
1	right shift	110	011

The physical layout drawn in class is not in the slides; the derivation above is the standard construction. See 2026-08-13-shift-registers[48] for the exercise as it was set.

[48] courses/csse2010/2026-08-13-shift-registers.pdf

Universal shift register

A **universal shift register** is the combination of all of the above capabilities in one device: it can hold its value, shift left, shift right, and be parallel-loaded, with the operation selected by control inputs.

> ⚠ Warning
>
> The "Universal Shift Register" slide in the lecture deck is **entirely blank** — the title only. The one-line description above is the standard definition; everything specific (the control encoding, the mux width, the figure) was covered in class and is not in the slides.

Wide (multi-bit) shift registers

A shift register does not have to shift one *bit* at a time — it can shift a whole multi-bit word per clock edge. The lecture's example is an **8-bit wide, 4-stage queue**:

- four **registers** in a row, each 8 bits wide (inputs labelled A through H, outputs labelled Q_1 through Q_8);
- the 8-bit `Input` bus feeds the first register's 8 data inputs;
- each register's 8 Q outputs feed the next register's 8 data inputs, as a bus;
- the last register's outputs are the 8-bit `Output`;
- all four registers share a **common clock** line, and each has an `ENB` (enable) input.

Functionally this is the same chain as the 1-bit shift register, but each "stage" holds a byte rather than a bit — so it behaves as a 4-deep queue of bytes: a byte presented at `Input` appears at `Output` four clock edges later.

Related

- flip-flops-and-latches[49] — the D flip-flop these are built from
- sequential-circuits[50] — state, present/next state, synchronous clocking
- combinational-logic-blocks[51] — the multiplexer used for load/direction selection
- [counters] — the other classic thing built from a row of D flip-flops
- device-pinouts[52] — 74HCT175 quad D flip-flop and 74HCT157 quad 2-input multiplexer pinouts

[49] courses/csse2010/flip-flops-and-latches.pdf
[50] courses/csse2010/sequential-circuits.pdf
[51] courses/csse2010/combinational-logic-blocks.pdf
[52] courses/csse2010/device-pinouts.pdf