CSSE2010 — Week 3 Notes

Table of contents

Lecture 5 — Introduction to Sequential Circuits **2**
- Today's outline . . . 2
- Recap from last week . . . 2
- Quiz: multiplexer implementing a given function . . . 3
- Circuits that remember values . . . 3
- The D flip-flop . . . 4
- Quiz: D flip-flop output waveform . . . 4
- SR latch from NOR gates . . . 5
- Flip-flops vs latches . . . 5
- Combinational vs sequential circuits . . . 5
- Reminders . . . 6

Lecture 6 — Sequential Circuits 1: Shift Registers **6**
- Today's outline . . . 6
- Recap slides . . . 6
- Registers . . . 7
- Shift registers . . . 7
- Exercise: bidirectional shift register . . . 8
- Universal shift register . . . 9
- Wide shift registers . . . 9
- Lab 06 preparation task . . . 9
- Reminders . . . 10

Lab 4 Pre-Work **10**
- Question 1 . . . 10
- Question 2 . . . 12

Reference material **14**
- Binary Number Representations . . . 14
- Boolean Algebra . . . 16

Circuit Schematics 17
Combinational Logic Blocks 19
Counters 24
Device Pinouts 28
Flip-Flops and Latches 30
Logic Gates 35
Sequential Circuits 37
Shift Registers 38

Lecture 5 — Introduction to Sequential Circuits

See flip-flops-and-latches[1] and sequential-circuits[2] for the reference definitions, characteristic tables and symbols this lecture introduces.

Today's outline

- Recap of last week's combinational logic
- Circuits that remember values
- The D flip-flop: symbol, operation, characteristic table, waveforms
- The SR latch built from NOR gates
- Flip-flops vs latches; a real D flip-flop and its chips
- Combinational vs sequential circuits; synchronous sequential circuits

Recap from last week

The course so far stacks up in three layers, each built on the one below:

1. **Binary representations** — unsigned, sign-magnitude, 1's complement, 2's complement, excess-2^{N-1} (see binary-number-representations[3], 2026-08-04-binary-arithmetic[4]).
2. **Logic gates** — NOT, AND, OR, NAND, NOR, XOR, XNOR — and Boolean algebra (see logic-gates[5], boolean-algebra[6]).
3. **Combinational logic circuits** — adder, adder/subtractor, multiplexer, decoder (see combinational-logic-blocks[7]).

[1] courses/csse2010/flip-flops-and-latches.pdf
[2] courses/csse2010/sequential-circuits.pdf
[3] courses/csse2010/binary-number-representations.pdf
[4] courses/csse2010/2026-08-04-binary-arithmetic.pdf
[5] courses/csse2010/logic-gates.pdf
[6] courses/csse2010/boolean-algebra.pdf
[7] courses/csse2010/combinational-logic-blocks.pdf

The recap slide shows the 4-bit ripple carry adder (four full adders chained by their carries), the 4-bit adder/subtractor with its mode select M ($M = 0$ add, $M = 1$ subtract), a 4:1 MUX with selects S_1S_0, and a 3:8 decoder with inputs A, B, C and outputs $D_0 \ldots D_7$.

Quiz: multiplexer implementing a given function

Consider a 4:1 multiplexer with data inputs A, B, C, D on lines 0, 1, 2, 3 respectively and select inputs S_1S_0. What must A, B, C, D be so that the multiplexer output is

$$X = S_1.S_0 + \bar{S}_1.G$$

The options were: (1) $A = 0, B = 0, C = 0, D = 1$; (2) $A = G, B = G, C = 0, D = 1$; (3) $A = G, B = 0, C = 0, D = 1$; (4) $A = 0, B = G, C = 0, D = 1$; (5) none of the above; (6) I don't know.

Reasoning. A 4:1 MUX passes whichever data input its select code addresses, so in general

$$X = \bar{S}_1\bar{S}_0.A + \bar{S}_1S_0.B + S_1\bar{S}_0.C + S_1S_0.D$$

Now expand the target expression into the same four select terms. The $\bar{S}_1.G$ term covers *both* $\bar{S}_1\bar{S}_0$ and $\bar{S}_1S_0$, and the $S_1.S_0$ term is 1 for that one select code only:

$$X = \bar{S}_1\bar{S}_0.G + \bar{S}_1S_0.G + S_1\bar{S}_0.0 + S_1S_0.1$$

Matching term by term against the MUX expression: $A = G$, $B = G$, $C = 0$, $D = 1$.

Answer: option 2.

Note the trap: it's tempting to put G on only one of the two $\bar{S}_1$ lines, but $\bar{S}_1.G$ says nothing about S_0, so *both* $\bar{S}_1$ inputs must carry G.

Circuits that remember values

- The output of any logic gate or **combinational circuit** depends on the **current value of the inputs only**.
- If an input changes, the output can change too — and the previous value is **lost forever**.
- In a **sequential circuit**, the current output depends not only on the current inputs but also on the **past** outputs.
- Circuits with **memory** can remember values even when the input changes.

That motivates the whole rest of the course: to build anything that accumulates, counts, or holds a result, you need a storage element.

The D flip-flop

The lecture's first memory element is the D flip-flop: **D** is the input, **Q** the output, and **CLK** the control input. Q copies the value of D — and remembers it — whenever CLK goes from 0 to 1 (the *rising edge*). Full symbol, characteristic table and worked waveform in flip-flops-and-latches[8].

The characteristic table is introduced here as the tabular definition of a flip-flop's operation, with the right-hand column $Q(t+1)$ meaning "what the output will be on the next clock edge".

Summary points from the lecture: a D flip-flop remembers a single bit, so n D flip-flops remember n bits and an n-bit **register** is by definition n D flip-flops. JK and T flip-flops also exist but are not covered in this course. Flip-flops can be made out of logic gates — which is the next slide.

Quiz: D flip-flop output waveform

Using the (rising-edge) D flip-flop presented previously, what is the output waveform for Q, given the D and CLK waveforms shown?

Reading the figure. The clock has four rising edges. D is low, rises just after the first clock pulse has ended, stays high across the second and third clock pulses, then falls before the fourth clock pulse.

Rising clock edge	D at that edge	Q after the edge
1st	0	0
2nd	1	1
3rd	1	1 (no change)
4th	0	0

So Q goes high at the **2nd** rising clock edge and stays high until the **4th** rising clock edge, where it returns to 0. Q is *not* a copy of D: it lags D's rise (waiting for the next clock edge) and lags D's fall (holding the 1 until the next clock edge).

Answer: option 3 — the trace that rises at the second rising clock edge and falls at the fourth rising clock edge.

Why the others are wrong:

- **Option 1** rises and falls exactly with D — that's D itself, i.e. no clocking at all.

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

- **Option 2** rises correctly at the second rising edge but falls at the *falling* edge of the third clock pulse — that would be a negative-edge-triggered device reacting to the wrong edge.
- **Option 4** follows D for the duration of each clock pulse (high while CLK is high and D is high, low again when CLK goes low) — that's **level**-triggered behaviour, i.e. a latch, not a flip-flop.

SR latch from NOR gates

The lecture builds a storage element from two cross-coupled NOR gates: S into the top gate (output $\bar{Q}$), R into the bottom gate (output Q), each gate's output fed back into the other gate's spare input.

The slide's truth table is **blank** — marked "to be completed in class", with a note to attempt it at home beforehand by assuming an initial value for Q and working out the rest for each combination of S and R. The full derivation (hold / reset / set, and why $S = R = 1$ is invalid) is in flip-flops-and-latches[9].

A follow-on slide sets a **homework**: re-analyse that same S-R latch with the NOR gates replaced by NAND gates, and complete the truth table. Worked through in flip-flops-and-latches[10] as well.

Flip-flops vs latches

Latches are level triggered; flip-flops are edge triggered. A clock has a positive edge (0→1) and a negative edge (1→0), so a D flip-flop can be positive- or negative-edge triggered. Details, plus the four schematic symbols (triangle = edge-triggered, bubble = falling edge) in flip-flops-and-latches[11].

The lecture also shows a real D flip-flop's internal NAND schematic, with its asynchronous active-low $\overline{\text{PRE}}$ and $\overline{\text{CLR}}$ inputs, and the chips that package them — the 74HCT74 (dual) and the 74HCT273 (eight flip-flops, i.e. one byte). See device-pinouts[12] for pinouts, and the device symbols PDF on Blackboard.

Combinational vs sequential circuits

The formal contrast is drawn here for the first time — and re-presented in the next lecture. Combinational circuits are logic gates only, with the output uniquely determined by the inputs; sequential circuits include flip-flops, with the output determined by the current inputs *and* the

[9] courses/csse2010/flip-flops-and-latches.pdf
[10] courses/csse2010/flip-flops-and-latches.pdf
[11] courses/csse2010/flip-flops-and-latches.pdf
[12] courses/csse2010/device-pinouts.pdf

current state, and the output only able to change when the clock 'ticks'. The general block diagram (combinational logic + flip-flops + feedback path) and the definition of a **synchronous** sequential circuit are in sequential-circuits[13].

Reminders

Coming up:

- **Lab 4** (Mon–Tue this week) — combinational logic. Make sure you attempt the preparation task. Use logic ICs or Logisim software to test your circuits.
- **Lab 5** (Thu–Fri this week) — flip-flops. Use Logisim software to test the circuits. **Kit loaning happens in Lab 5.**

Lecture 6 — Sequential Circuits 1: Shift Registers

See shift-registers[14] for the reference material this lecture introduces, and sequential-circuits[15] and flip-flops-and-latches[16] for the recap slides it opens with.

Today's outline

- Admin
- Sequential circuits
- Shift registers

Recap slides

The first three content slides are repeats from [[2026-08-10-introduction-to-sequential-circuits|Lecture 5]] and their content lives in the concept notes:

- **"Reminder: Memory element — D Flip Flop"** — D input, Q output, CLK control input; Q copies (and remembers) D on the **rising edge** of CLK. Only D flip-flops are used in this course. Optional **asynchronous SET and CLR** inputs set/clear Q outside clock edges and are typically **active-low**. D flip-flops are what you build sequential circuits from — e.g. [counters]. Full detail in flip-flops-and-latches[17].

[13] courses/csse2010/sequential-circuits.pdf
[14] courses/csse2010/shift-registers.pdf
[15] courses/csse2010/sequential-circuits.pdf
[16] courses/csse2010/flip-flops-and-latches.pdf
[17] courses/csse2010/flip-flops-and-latches.pdf

- **"Combinational vs. Sequential Circuits"** — combinational = logic gates only, output uniquely determined by the inputs (the slide's example is $A(B + C)$, see logic-gates[18]); sequential = includes flip-flops, output determined by current inputs *and* current **state**, and can change when the clock ticks. See sequential-circuits[19].
- **"Sequential Circuits" / "Synchronous Sequential Circuit"** — state = values in the flip-flops, present state vs next state, and the rule that in a synchronous sequential circuit **all sequential elements share a common clock signal**. See sequential-circuits[20].

Registers

A **register** is a group of flip-flops: an n-bit register is n flip-flops storing n bits. Two points the slide makes explicitly:

- a register is a sequential circuit **without any combinational logic** — unlike the general sequential-circuit block diagram from the recap slides, which has a combinational block in the feedback path;
- registers store binary information (data/instructions) **inside a processor**.

The worked example is a 4-bit register with parallel inputs $I_0..I_3$, parallel outputs $A_0..A_3$, and common `Clock` and (active-low) `Clear` lines. Structure and behaviour in shift-registers[21].

Shift registers

A **shift register** is a register capable of shifting its binary information in one or both directions. The lecture's example is the 4-bit chain (serial input SI → four D flip-flops in series → serial output SO, all on a common CLK). Construction and a clock-by-clock trace are in shift-registers[22].

Two applications follow, both of which the slides leave as blank figures marked *"figure to be completed in class"*:

- **Serial parallel conversion** — the slide states that shift registers can do serial-to-parallel conversion and vice-versa, then shows four unconnected D flip-flops on a common `clk` with the figure left to be completed live.

[18] courses/csse2010/logic-gates.pdf
[19] courses/csse2010/sequential-circuits.pdf
[20] courses/csse2010/sequential-circuits.pdf
[21] courses/csse2010/shift-registers.pdf
[22] courses/csse2010/shift-registers.pdf

- **Parallel load and serial shift** — likewise four unconnected D flip-flops on a common `clk`, with the whole figure left blank. The next slide's phrase "using the same multiplexer concept" tells us the completed figure put a **multiplexer in front of each flip-flop's D input** to choose between the shift source and the parallel input — see combinational-logic-blocks[23] for the mux, and shift-registers[24] for the derived construction.

Neither completed figure is recoverable from the deck; only the derived standard constructions are recorded.

Exercise: bidirectional shift register

> Using the same multiplexer concept, draw a **3-bit shift register which allows data to be shifted in either direction**.
>
> Hint: consider this element, where **DIRN will be 0 for left shift, 1 for right shift**.

The hint element given on the slide is a **2-to-1 multiplexer feeding one D flip-flop**: the mux has data inputs labelled `0` and `1`, its select input is `DIRN`, its output goes to the flip-flop's D input, and the flip-flop's Q is the stage output.

The answer slide is blank — the worked construction was drawn live and is not in the deck. Derived from the hint element and the standard construction, the answer is: instantiate three copies of the element, share one `DIRN` line and one clock across all three, and for each stage wire

- **mux input 0** (selected when DIRN = 0, left shift) to the neighbouring stage on the left-shift *source* side, and
- **mux input 1** (selected when DIRN = 1, right shift) to the neighbouring stage on the right-shift *source* side,

with an external serial input supplying whichever end of the register has no neighbour in that direction. Writing the bits as $Q_2Q_1Q_0$ with Q_2 most significant, and taking "left shift" to move bits towards the more-significant end:

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

[23] courses/csse2010/combinational-logic-blocks.pdf
[24] courses/csse2010/shift-registers.pdf

Sanity check on $Q_2Q_1Q_0 = 110$ with both serial inputs at 0: one edge with DIRN = 0 gives 100; one edge with DIRN = 1 gives 011.

The essential insight the exercise is testing: **a bidirectional shift register is just a unidirectional one with a mux per flip-flop**, and the direction control is a single line shared by every mux, so the whole register commits to one direction per clock edge.

Universal shift register

The slide titled "Universal Shift Register" is **completely blank** — title only. Everything on it was done live. The concept (hold / shift left / shift right / parallel load in one device, selected by control inputs) is recorded in shift-registers[25].

Wide shift registers

Shift registers can shift **multiple bits at a time**. The lecture's example is a **4-stage, 8-bit queue**: four 8-bit registers in a row (inputs $A..H$, outputs $Q_1..Q_8$, each with an `ENB` enable), each register's outputs feeding the next register's inputs as an 8-bit bus, all on a common clock. A byte at `Input` emerges at `Output` four clock edges later. See shift-registers[26].

Lab 06 preparation task

Stated on the slides as (verbatim):

> **2-digit lock/unlock circuit**: User inputs two decimal digits (4 bits each) **AB** in serial and the circuit should match the two input digits with a code (say **CD**) and unlock if the input matches with the code (i.e. **AB = CD**).

Both Lab 06 preparation-task slides are marked *"to be discussed in class"* and the second is blank, so **no worked solution is in the deck**. The shape of the solution is signposted by the lecture, though: serial digit entry into a shift register is exactly the serial-to-parallel conversion above — clock the 8 serially-entered bits into an 8-bit shift register, then compare the parallel output against the stored code CD with combinational logic (see combinational-logic-blocks[27]) and drive the unlock output.

[25] courses/csse2010/shift-registers.pdf
[26] courses/csse2010/shift-registers.pdf
[27] courses/csse2010/combinational-logic-blocks.pdf

Reminders

- **Labs 6 and 7** (next week, week 4) have **preparation tasks** which should be attempted **before** coming to the labs.
- Attempt the weekly exercise and quizzes.

Lab 4 Pre-Work

Question 1

Draw a circuit schematic diagram for a 2-to-1 multiplexer that uses only 2-input NAND gates. The data inputs should be connected to push buttons. The select input should be connected to a switch. The output should be connected to an LED.

Hint: Do the truth table (or a function table) for a 2-to-1 multiplexer, obtain the logic expression, then draw a logic circuit and convert it to the NAND equivalent circuit by replacing each gate with its NAND version. If you use Logisim, do not use the MUX module.

1. Function Table

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

2. Sum of Products & Simplification

Deriving the unsimplified Sum of Products from the true outputs in the table:

- $S_0 = 0,\ D_0 = 1,\ D_1 = 0 \rightarrow \bar{S_0} \cdot D_0 \cdot \bar{D_1}$
- $S_0 = 0,\ D_0 = 1,\ D_1 = 1 \rightarrow \bar{S_0} \cdot D_0 \cdot D_1$
- $S_0 = 1,\ D_0 = 0,\ D_1 = 1 \rightarrow S_0 \cdot \bar{D_0} \cdot D_1$
- $S_0 = 1,\ D_0 = 1,\ D_1 = 1 \rightarrow S_0 \cdot D_0 \cdot D_1$

Using Boolean algebra, we can simplify this expression:

$$\begin{aligned} F &= \bar{S_0} D_0 \bar{D_1} + \bar{S_0} D_0 D_1 + S_0 \bar{D_0} D_1 + S_0 D_0 D_1 \\ &= \bar{S_0} D_0 (\bar{D_1} + D_1) + S_0 D_1 (\bar{D_0} + D_0) \\ &= \bar{S_0} D_0 (1) + S_0 D_1 (1) \\ &= \bar{S_0} D_0 + S_0 D_1 \end{aligned}$$

3. NAND Equivalent Conversion

To implement this circuit using *only* 2-input NAND gates, we must convert the simplified AND-OR expression into a NAND-NAND expression. We can achieve this by applying double negation and De Morgan's Laws:

$$\begin{aligned} F &= \overline{\overline{\bar{S_0} D_0 + S_0 D_1}} \\ &= \overline{(\overline{\bar{S_0} D_0}) \cdot (\overline{S_0 D_1})} \end{aligned}$$

4. Circuit Schematic

Based on the final NAND logic expression, the circuit requires exactly four 2-input NAND gates: 1. One NAND gate to act as an inverter for S_0 (by wiring S_0 to both inputs) to create $\bar{S_0}$. 2. One NAND gate to compute $\overline{\bar{S_0} D_0}$. 3. One NAND gate to compute $\overline{S_0 D_1}$. 4. One final NAND gate to combine the outputs of gates 2 and 3.

All four gates fit on a single 74HCT00 (quad 2-input NAND — see device-pinouts[28]). Using circuit-schematics[29] labelling conventions, gates 1–4 above become `U1:A`–`U1:D`, wired as:

- `U1:A`: inputs 1, 2 tied together to S_0 (switch) $\rightarrow$ pin 3 $= \bar{S_0}$
- `U1:B`: inputs 4 $= \bar{S_0}$ (`U1:A` pin 3), 5 $= D_0$ (push button) $\rightarrow$ pin 6 $= \overline{\bar{S_0} D_0}$
- `U1:C`: inputs 9 $= S_0$ (switch), 10 $= D_1$ (push button) $\rightarrow$ pin 8 $= \overline{S_0 D_1}$
- `U1:D`: inputs 12 = `U1:B` pin 6, 13 = `U1:C` pin 8 $\rightarrow$ pin 11 $= F$ (LED)

VCC (pin 14) and GND (pin 7) are shown once for `U1`.

[28] courses/csse2010/device-pinouts.pdf
[29] courses/csse2010/circuit-schematics.pdf

Question 2

Circuit 2

Draw a circuit schematic diagram for a 2-to-4 decoder - using whichever logic gates you prefer (from those available in the CSSE2010/CSSE7201 lab). The two inputs should be connected to switches. The outputs should be connected to LEDs.

A 2-to-4 decoder has the following truth table (inputs A1, A0 and outputs X0, X1, X2, X3):

A1	A0	X0	X1	X2	X3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

1. Boolean Expressions

Since A_1, A_0 can only take four combinations, each output's truth-table row is already a single minterm — no Boolean simplification is needed, just one 2-input AND term per output:

- $X_0 = \bar{A_1} \cdot \bar{A_0}$
- $X_1 = \bar{A_1} \cdot A_0$
- $X_2 = A_1 \cdot \bar{A_0}$
- $X_3 = A_1 \cdot A_0$

2. Circuit Schematic

Two chip types cover this: a 74HCT04 (hex inverter) to generate $\bar{A_1}$ and $\bar{A_0}$ once each, and a 74HCT08 (quad 2-input AND — see device-pinouts[30]) for the four output terms. Using circuit-schematics[31] conventions:

- `U1` (74HCT04): `U1:A` (pin 1 → 2): A_1 (switch) → $\bar{A_1}$. `U1:B` (pin 3 → 4): A_0 (switch) → $\bar{A_0}$.
- `U2` (74HCT08):
  - `U2:A` (in 1, 2 → out 3): $\bar{A_1}$ (`U1:A` pin 2), $\bar{A_0}$ (`U1:B` pin 4) → X_0 (LED)
  - `U2:B` (in 4, 5 → out 6): $\bar{A_1}$ (`U1:A` pin 2), A_0 (switch) → X_1 (LED)
  - `U2:C` (in 9, 10 → out 8): A_1 (switch), $\bar{A_0}$ (`U1:B` pin 4) → X_2 (LED)
  - `U2:D` (in 12, 13 → out 11): A_1 (switch), A_0 (switch) → X_3 (LED)

[30] courses/csse2010/device-pinouts.pdf
[31] courses/csse2010/circuit-schematics.pdf

VCC/GND (pin 14/7 on both chips) are shown once per chip type.

Circuit 3
Using a 74HCT04, 74HCT157 and a 74HCT283, draw a circuit schematic diagram for a circuit which can act as a 4-bit two's complement adder or subtractor. The output (4-bits from the 74HCT283, which should be shown on LEDs) will have the value A+B or A-B (where A and B are the 4-bit inputs which are taken from switches).

A push button input (M, mode) will determine whether the circuit performs addition or subtraction - if the value is 0, addition will be performed; if the value is 1, subtraction will be performed by calculating A+(-B).

The circuit is similar to that shown in week3-lecture 4-slide 7 except the selection of the "B" input bits as B or not(B) is to be performed using multiplexers instead of XOR gates. (The 74HCT157 is a quad 2-to-1 multiplexer with a shared select input.) The carry output should be shown on a LED also.

1. Circuit Design Reasoning

In two's complement, $-B = \bar{B} + 1$, so subtraction can be built from the same adder used for addition: $A - B = A + \bar{B} + 1$. This means mode input M needs to do two things at once:

- Select whether the adder's B input is B (add) or $\bar{B}$ (subtract) — done with the 74HCT157 MUX.
- Inject the "+1" that turns one's complement into two's complement — done by wiring M straight into the adder's carry-in, since $C_0 = 0$ for addition (no extra bit needed) and $C_0 = 1$ for subtraction (adds the required 1).

So M is wired to both the 74HCT157's select line and the 74HCT283's C_0 pin — no separate control logic needed.

2. Circuit Schematic

Three chips (see device-pinouts[32] for exact pins), following circuit-schematics[33] conventions:

- `U1` (74HCT04, hex inverter) — generates $\bar{B_0}$–$\bar{B_3}$ for the MUX's B-channel:
  - `U1:A` (1→2): B_0 (switch) → $\bar{B_0}$
  - `U1:B` (3→4): B_1 (switch) → $\bar{B_1}$
  - `U1:C` (5→6): B_2 (switch) → $\bar{B_2}$
  - `U1:D` (9→8): B_3 (switch) → $\bar{B_3}$

[32] courses/csse2010/device-pinouts.pdf
[33] courses/csse2010/circuit-schematics.pdf

- `U2` (74HCT157, quad 2-to-1 MUX) — selects B_i (channel A, $M = 0$) or $\bar{B}_i$ (channel B, $M = 1$) per bit. `SEL` (pin 1) = M (push button); `ENABLE` (pin 15) tied to GND (always active):
  - Channel 0: `A0` (pin 2) = B_0 switch, `B0` (pin 3) = $\bar{B}_0$ (`U1:A` pin 2), `Y0` (pin 4) → adder `B0`
  - Channel 1: `A1` (pin 5) = B_1 switch, `B1` (pin 6) = $\bar{B}_1$ (`U1:B` pin 4), `Y1` (pin 7) → adder `B1`
  - Channel 2: `A2` (pin 11) = B_2 switch, `B2` (pin 10) = $\bar{B}_2$ (`U1:C` pin 6), `Y2` (pin 9) → adder `B2`
  - Channel 3: `A3` (pin 14) = B_3 switch, `B3` (pin 13) = $\bar{B}_3$ (`U1:D` pin 8), `Y3` (pin 12) → adder `B3`
- `U3` (74HCT283, 4-bit adder) — sums A with the MUX's selected $B/\bar{B}$, carry-in doubling as the mode's "+1":
  - `A0`–`A3` (pins 5, 3, 14, 12) = A_0–A_3 switches (unchanged by mode)
  - `B0`–`B3` (pins 6, 2, 15, 11) = `U2` `Y0`–`Y3`
  - `C0` (pin 7, carry-in) = M (same signal as `U2`'s `SEL`)
  - `S0`–`S3` (pins 4, 1, 13, 10) → LEDs (result)
  - `C4` (pin 9, carry-out) → LED

VCC/GND are shown once per chip type.

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):

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[34]) 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[35]). 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[36] 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.

[34] courses/csse2010/logic-gates.pdf
[35] courses/csse2010/logic-gates.pdf
[36] 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[37]).
- **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.

[37] 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[38] for the gate symbols/truth tables and boolean-algebra[39] 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[40], 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[41]).

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.

[38] courses/csse2010/logic-gates.pdf
[39] courses/csse2010/boolean-algebra.pdf
[40] courses/csse2010/sequential-circuits.pdf
[41] 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[42]) 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

$$C_0 = M$$

[42] 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[43].

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.

[43] 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[44].

[44] 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[45].

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:

[45] 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.

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

Bit 3	Bit 2	Bit 1	Bit 0	Value
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[46].

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

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[47].
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[48] for schematic conventions and device-pinouts[49] for the 74HCT74 / 74HCT175 D flip-flop packages.

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	$10 \to 01$

[47] courses/csse2010/boolean-algebra.pdf
[48] courses/csse2010/circuit-schematics.pdf
[49] courses/csse2010/device-pinouts.pdf

Q_1	Q_0	D_1	D_0	comment
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_1D_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[50] 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.

[50] courses/csse2010/week4-lab-lab-7-exercises.pdf

Related

- flip-flops-and-latches[51] — the D flip-flop and its characteristic table.
- sequential-circuits[52] — state, present state / next state, synchronous sequential circuits.
- shift-registers[53] — the other family of registers built from a flip-flop chain.
- boolean-algebra[54] — sum of products and the simplification laws used in step 4–5.
- binary-number-representations[55] — the binary counting sequence itself.

Device Pinouts

Pin numbers for the 74-series logic chips and IO board used in circuit-schematics[56]. 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[57]): 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

[51] courses/csse2010/flip-flops-and-latches.pdf
[52] courses/csse2010/sequential-circuits.pdf
[53] courses/csse2010/shift-registers.pdf
[54] courses/csse2010/boolean-algebra.pdf
[55] courses/csse2010/binary-number-representations.pdf
[56] courses/csse2010/circuit-schematics.pdf
[57] courses/csse2010/circuit-schematics.pdf

Chip	Function	Gate A	Gate B	Gate C	Gate D	VCC	GND
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.

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[58]). Use sensible 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]].

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

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[59].
- 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.

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

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

Clock edge	D at that edge	Q after
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[60]) 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

[60] 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[61].
- **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.

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[62] for the last of these, and for real-gate propagation delay / rise / fall times).

[61] courses/csse2010/device-pinouts.pdf
[62] courses/csse2010/timing-diagrams.pdf

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
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).

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[63] for the storage elements themselves.

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[64])	Counters, registers (see [counters], shift-registers[65])

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[66]).

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.

[63] courses/csse2010/flip-flops-and-latches.pdf
[64] courses/csse2010/combinational-logic-blocks.pdf
[65] courses/csse2010/shift-registers.pdf
[66] courses/csse2010/timing-diagrams.pdf

"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.

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
$I_0 \dots I_3$	the four parallel data inputs, one to each flip-flop's D input
$A_0 \dots A_3$	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[67] 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:

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

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.

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[68]) instead of directly from the previous stage:

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

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, 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[69] for the exercise as it was set.

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.

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

Related

- flip-flops-and-latches[70] — the D flip-flop these are built from
- sequential-circuits[71] — state, present/next state, synchronous clocking
- combinational-logic-blocks[72] — the multiplexer used for load/direction selection
- [counters] — the other classic thing built from a row of D flip-flops
- device-pinouts[73] — 74HCT175 quad D flip-flop and 74HCT157 quad 2-input multiplexer pinouts

[70] courses/csse2010/flip-flops-and-latches.pdf
[71] courses/csse2010/sequential-circuits.pdf
[72] courses/csse2010/combinational-logic-blocks.pdf
[73] courses/csse2010/device-pinouts.pdf