Integrated circuits

Everything discussed so far is the theoretical grounds for constructing the computer processor. In this lecture, we will connect the theory to practice.

Basic building blocks

Transistors

As we saw in the description of history of computing machines, early computers did not use transistors but vacuum tubes. That, however, was not efficient in terms of size, reliability, and energy consumption. The technological revolution started at the end of 1940's when the first **transistors** were built. The detailed electronic description of a transistor is out of scope of this course, so we will not discuss its construction or possible types. The important thing for us that transistors have the capability of **controlling** the eletric signal (which means that, by choosing the voltage in one electric circuit, we can influence the voltage in another), allowing in addition to:

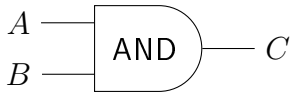
- choose the nature of propagation: depending on the transistor, *higher* voltage on the input can cause *higher* or *lower* voltage on the output;
- **amplify** the voltage changes (i.e. the range of output voltage values is broader than the range of input values) — which allows in practice combining transistors into *arbitrarily large* **integrated circuits**, without losing the original signal.

To understand the critical role of this fact for computing, we need to see the relationship between electric current and digital information storage. A low voltage (ca. 0 V) is interpreted as "0", while a high one (ca. 5 V) as "1". Of course, at some moments, there will also happen voltage values quite in between; however, these moments will be very short and will not influence the computations.

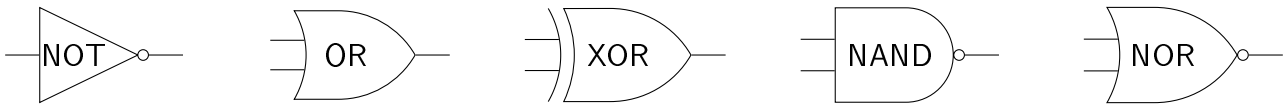
Logic gates

Thanks to the fact that transistors allow *de facto* opening or closing electric circuits depending on values of input signals, an appropriate combination of a few transistors can lead to what is called **logic gates**. A single logic gate manipulates electric signals according to a logic operation, that is: it reacts to the input signals (i.e. voltage levels, interpreted as "0" or "1" as explained above) and ensures that the output pin observes an appropriate output signal, corresponding to the result of the appropriate logic function. For example, the AND function is implemented by the AND gate, which is a combination of a few transistors which takes two input signals (denoted below by A and B) and generates the output signal ($C = A$ AND B).

Logic gates (particularly AND, OR, NOT, NAND, NOR) are typically treated as **basic building blocks** for constructing the more complex integrated circuits. Therefore, the diagrams presenting such circuits usually contains symbols denoting logic gates of various types, without digging into their internals. For example, the AND gate with inputs A, B and output C is depicted as follows:



And these are the other standard representations of the most common logic gates:



One can see some elements of a consistent **convention** here:

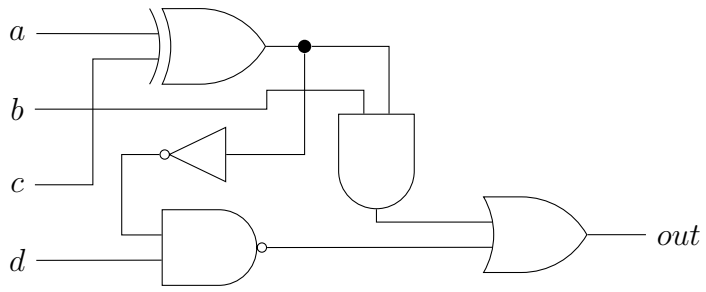
- All the logic gate symbols somewhat resemble an arrow (the left edge on our pictures is either fully flat or at least less "sharp" than the right edge); this allows easily identifying which signals are input and output;
- Each logic operation corresponds to a different gate shape; this allows skipping logic operation names, while keeping a clear diagram meaning;
- The tiny circle denotes negation; e.g. NAND differs from AND just be negating the result, similarly NOR from OR; finally, the NOT gate symbol also contains a circle to emphasize that a negation takes place there.
  (That circle *does not* constitute a stand-alone logic gate! For example, NAND gates are *not* typically implemented by joining an AND gate with negation, as it would be inefficient).
- Gates are *usually* oriented as shown above (inputs on the left, output on the right), though this isn't strictly required — they can also "lead" from the top to bottom, or in any other direction.

Representing Boolean functions

A single logic gate is not very useful. The interesting thing is joining them so that the output of one gate becomes an input of another. For instance, the Boolean function defined by the formula

$$out = ((a \ \mathsf{XOR}\ c) \ \mathsf{AND}\ b) \ \mathsf{OR}\ ((\ \mathsf{NOT}\ (a \ \mathsf{XOR}\ c)) \ \mathsf{NAND}\ d)$$

will be represented e.g. by the following diagram of gates:



The • symbol denotes here a branching of the network (which allows using the value a XOR c, once computed, in two different places of the diagram). Note that the picture also contains crossings of power lines not denoted by • — in those cases, the connections *do not* meet each other (in practice, one of them would be put over another).

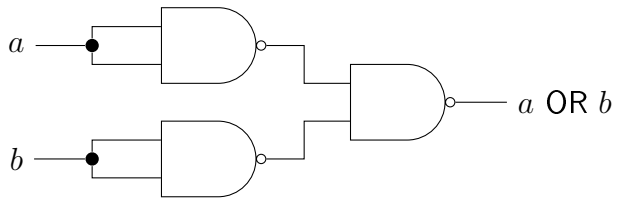
On lecture "Logic", we showed that every Boolean function can be expressed by just using the NAND operator. The same then holds for integrated circuits, that is: the **NAND gate suffices** to realize an arbitrary Boolean function. Similarly, just the NOR gate would suffice. (On the other hand, note

that e.g. the AND gate allow would *not* suffice; to see this, note that e.g. two truths on the input will never lead to false on the output, no matter how many AND gates we would use).

We will now show how to implement the OR function just with NAND gates. From lecture "Logic", we know that:

$$a \text{ OR } b = (a \text{ NAND } a) \text{ NAND } (b \text{ NAND } b),$$

so the corresponding integrated circuit will look as follows:



Circuits with memory

Combinational vs. sequential circuits

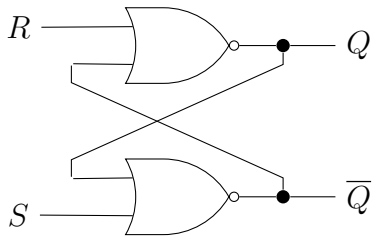
In all examples so far, the result depended **just on the inputs**. Such circuits are called **combinational**. We already know that they allow expressing any logic computation (even by using the NAND gate alone). Moreover, as the binary system reduces numbers to sequences of zeroes and ones, arithmetic computations become a special case of Boolean functions, so they turned out doable in such circuits, as well. Then, we're already close to building a computer! — however, one important problem still remains: how, after performing a computation, can we store its result, even temporarily?

The solution turns out to rely on introducing a *cyclic dependency* to the circuit structure: the input of some gates begins to (indirectly) depend on their output. Although such "loops" on the diagram can be shocking (how can a signal depend on itself?), in practice their behavior often turns out to be well-defined thanks to the **delay** in propagating changes of voltage through transistors. That delay, while very short, is measurable: for a single logic gate (consisting of a few transistors), a typical **propagation time** (the time between an input changes and the output is updated accordingly) is ca. 20 nanoseconds (where $1\,\text{ns} = 10^{-9}\,\text{s}$).

Then, in the circuits with "loops", called **sequential**, the output signal can **depend** not just on the input signals, but also **on the previous state** of the circuit. This lays the foundation for building a (short-lived) *digital memory*.

Simple RS flip-flop

The simplest sequential circuits are **flip-flops** (also called **latches**), purposed to store 1 bit of information. Let's take a look at the first of them, called the **RS flip-flop**:

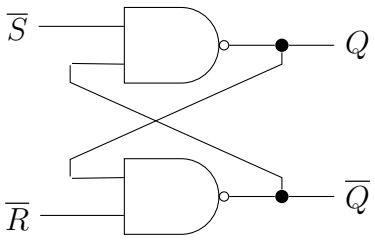


Input		Output		Description
R	S	Q	$\overline{Q}$	
0	0	same as previously		Q remembered
1	0	0	1	Q reset (to 0)
0	1	1	0	Q set (to 1)
1	1			(disallowed state)

This flip-flop consists of two NOR gates and has two input signals, named R (*Reset*) and S (*Set*). Its two output signals are always complementary to each other (at least, in the allowed states); that's why they're denoted by Q and $\overline{Q}$, where the over-bar denotes negation. Let's now trace how this flip-flop acts depending on the input signals:

- When $R = S = 0$, the circuit is in **remembering** state, that is, the values Q, $\overline{Q}$ do not change. (More precisely, they do not change also after the propagation time for the NOR gate passes).
- If $R = 1$ but $S = 0$, the upper gate has output 0, which then makes the lower gate generate output 1. Hence, such input configuration leads (in about double gate propagation time) to **resetting** the stored bit Q (i.e. assigning it the zero value).
- If $S = 1$ but $R = 0$, we analogously after some time obtain $Q = 1$, that is, Q has been **set**.
- Finally, if $R = S = 1$, the circuit enters a *disallowed state* in which both output signals have value 0 (so their values are no longer complementary).

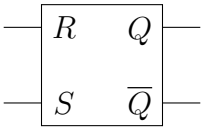
The RS flip-flop can be also efficiently built with NAND gates; in this case, the input shall contain the negated values $\overline{R}$ and $\overline{S}$.



Input		Output		Description
$\overline{R}$	$\overline{S}$	Q	$\overline{Q}$	
1	1	same as previously		Q remembered
0	1	0	1	bit Q reset (to 0)
1	0	1	0	bit Q set (to 1)
0	0			(disallowed state)

Therefore, the RS flip-flop constitutes the simplest kind of a **memory cell**, in which we can **store** a bit value (of course, provided a constant availability of electric energy); we can also, at any moment of our choice, **write** any value there (0 or 1, by setting the R or S signal accordingly).

As the RS flip-flop is a building block for the more complex circuits — similarly to logic gates — we will use a simplified visual representation also for it:

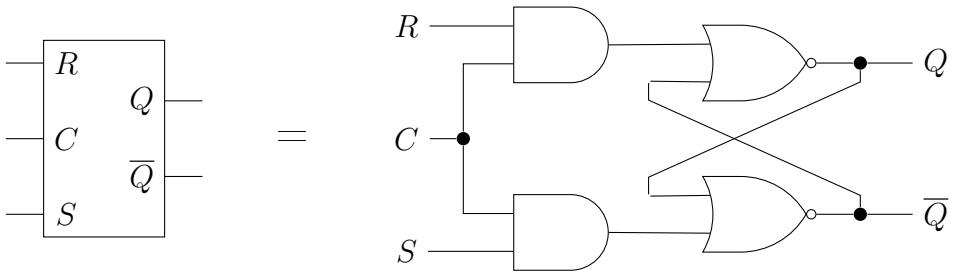


Clock synchronization

As we now can perform any logic (and thus also arithmetic operations), and can construct memory, it might seem that we gathered the whole toolbox to build a computer processor. In practice, however, when designing larger integrated circuits, we must care for **synchronizing** the computation to avoid e.g. a situation in which two partial results "missed each other" due to different propagation times involved in obtaining them.

The standard solution for this problem is to use a system **clock**, i.e. a special device generating a **clock signal** C which takes values 0 and 1 in an *alternating* fashion, following a *regular* pattern (will well known frequency). This allows to plan the whole computation along the **clock cycles**: as we'll see below, the processor registers allow storing new values only in a specific phase of the cycle, while using **counters** allows to manage complex computations across multiple cycles.

The simplest example of clock synchronization is the **synchronous RS flip-flop** in which modifying the stored bit is allowed only when $C = 1$:



The difference (compared to previous version) is that both input signals R, S are subject to conjunction with the clock signal C. Therefore:

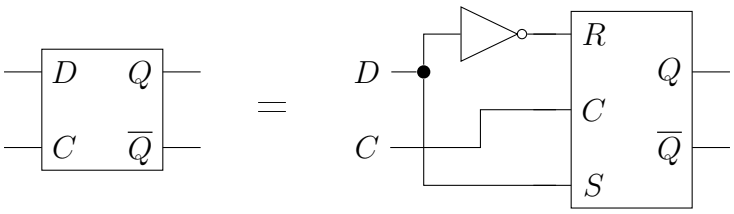
- As long as $C = 1$, the circuit is **active** and acts exactly as described for the previous version.
- Assigning $C = 0$ causes zeroing the input signals and hence **locking** the inner RS flip-flop in its remembering state.

Depending on its construction, a particular synchronous circuit can have various conditions for activation:

- Activation with a value of the clock signal (e.g. $C = 1$) — just as in the above example;
- Activation with a **slope** (i.e. change of signal value): $C = 0 \nearrow 1$ (*rising slope*) or $C = 1 \searrow 0$ (*falling slope*) — a bit more tricky to implement (as it requires capturing the value of C with a time shift) but allowing stricter control over data flow thanks to a short time of activity.

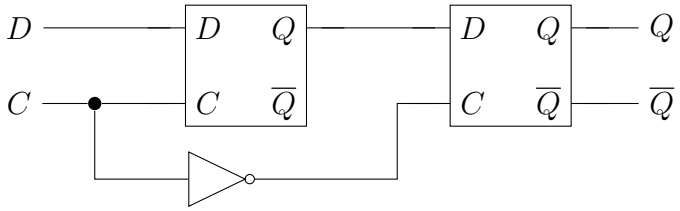
D flip-flop — a synchronous memory cell

Having a synchronous RS flip-flop, we can simplify it to what is known as a **D flip-flop**:



Its operation is very simple: it stores the value of D in the memory, as long as it's active (which happens when $C = 1$). That's because $D = 1$ means that $R = 0$ and $S = 1$ (so the *Set* mode), while $D = 0$ conversely implies $R = 1$ and $S = 0$ (i.e. the *Reset* mode).

Then, taking two D flip-flops as above, we can easily build a variation of the D flip-flop **activated by falling slope** of the clock signal:



To have the input value D appear on the output Q, it must be "passed through" by the flip-flop on the left side (which requires $C = 1$), and *then* by the flip-flop on the right (which in turn requires $C = 0$). This means that the whole circuit allows storing values *only* when $C = 0$ happens *just after* $C = 1$, that is, on the falling slope of the clock signal. In all other cases, changing the output is blocked by either one of the two sub-flip-flops.

Registers

As was already explained, flip-flops allow remembering 1 bit of information. However, as mentioned on lecture "Encoding data", programs usually process data in somewhat larger chunks, typically 1, 2, 4 or 8 bytes. This leads to the notion of **register** — an integrated circuit which allows storing that amount of data. Therefore, registers will be typically **8-, 16-, 32- or 64-bit**.

The simplest implementation of a register is an **array of independent D flip-flops**, purposed for storing consecutive bits. (However, that's not the only possible implementation: sometimes, flip-flops are joined in a *series*, which leads to *shift registers* in which the stored bits are moved by 1 position on every clock cycle).

Other useful circuits

From a purely theoretical viewpoint, what we described so far (logic gates, registers, and the clock) make up all the elements necessary for building a computer processor: we can compute anything, store results, and manage complex computations over time. However, such approach ignores efficiency completely. For example, although we know a general method to express any Boolean function with the NAND operation, its outcome if often much more complex than the optimal one.

Therefore, it's important to construct circuits which **efficiently** realize practical tasks, starting with those most frequently met during computing. Examples of such specialized circuits will be described below.

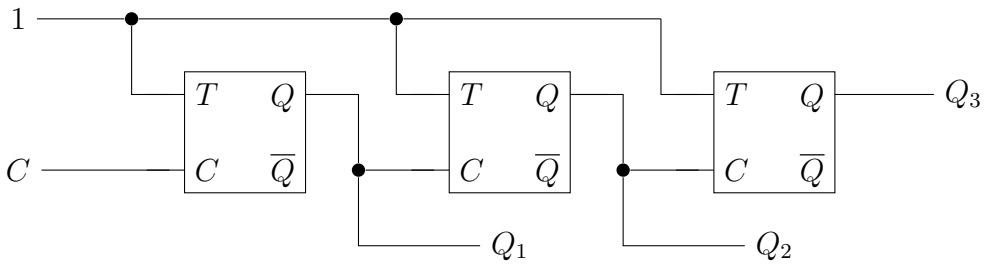
Counter

Counters are circuits which (as the name suggests) count the number of clock cycles. They're essential for synchronizing complex computations.

Counters are build with **T flip-flops** (whose structure will not be described here, as it's somewhat more sophisticated). The important thing about them is that they behave similarly to a D flip-flops activated by a clock signal slope, with one notable difference: activation of a T flip-flop results in **storing** in the memory not the input value T, but its **combination** with the old value: T XOR Q. This means that, at the moment when the flip-flop gets activated:

- If $T = 0$, the stored value Q remains unchanged;
- If $T = 1$, the stored value Q is replaced by its negation.

T flip-flops can be used to build a k**-bit counter** which, after the n-th clock cyle, will store k last bits of the binary representation of the number n. The picture below shows such a counter for $k = 3$:



The table below presents the output values Q_1, Q_2, Q_3 after the consecutive clock signals:

Cycle	Q_3	Q_2	Q_1
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
8	0	0	0
9	0	0	1

To explain why this happens, note that:

- The Q_1 value keeps alternating (being replaced with its negation) in every clock cycle.
- As the *clock input* for the middle sub-flip-flop is Q_1, the output of that one (Q_2) will change its value as a response to Q_1 changing from 1 to 0. This happens in every second clock cycle.

- Analogously, as the clock input for the right-most sub-flip-flop is Q_2, its output value (Q_3) will change in every fourth cycle.

As can be seen, combining k T flip-flops in this way, one can build a k-bit counter for an arbitrary k.

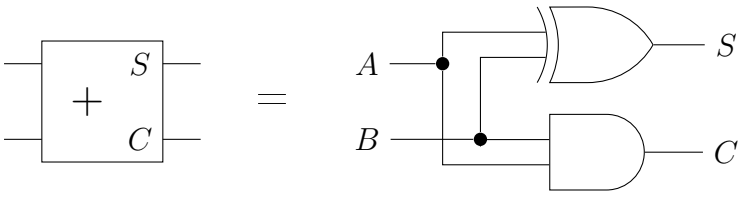
Adder

Now that we count with counters, it's time to tackle addition. Here, we will use **adders**. Before presenting those, it's good to recall the school method of adding. As usually, we operate in the binary positional system. This means that the "table addition" looks as follows:

	^{1}	^{1}				^{1}		
	1	1	1	1	0	0	1	0
+	0	1	1	0	0	0	1	1
1	0	1	0	1	0	1	0	1

We see that the i-th digit of the result (counting from the right) depends on the i-th digits of the summands and the "carry bit" (denoted above with the tiny "1" on the top), i.e. an information whether an overflow has occurred just before. This means that we need a circuit with **three inputs** (the i-th digit of the 1st summand, the i-digit of the 2nd summand, and the carry bit) and **two outputs** (the i-th digit of the result, and the carry bit for the $(i+1)$-th digit).)

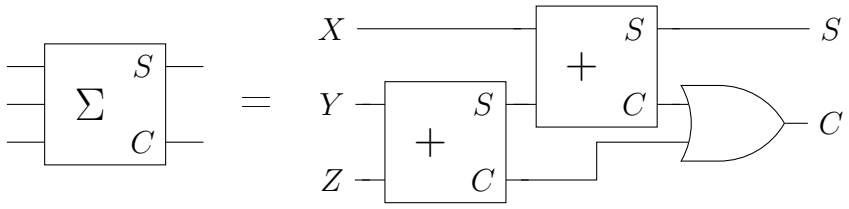
Before we present that circuit, we'll show a simpler one, which will be helpful as a building block. That is the **half adder**, serving a similar task of adding bits but just for **two inputs**. That one is defined as follows (also see the transition table on the right):



Input		Output	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

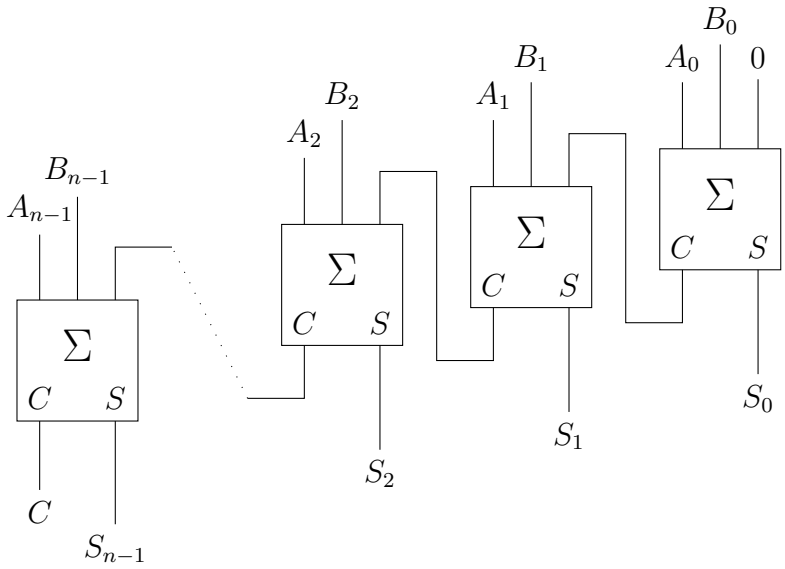
The output signal S is the bit sum, while C is the carry bit.

A **bit adder**, responsible for adding *three* bits, can be built as follows:



Input			Output	
X	Y	Z	S	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Now, we ready to construct a **multi-bit adder**. If A_i, B_i, S_i denote respectively the i-th digits of the 1st summand, 2nd summand, and the result (starting from 0, counting from the *right*!), then an n-bit adder can be built by joining n single-bit adders in the following way:



Comparator

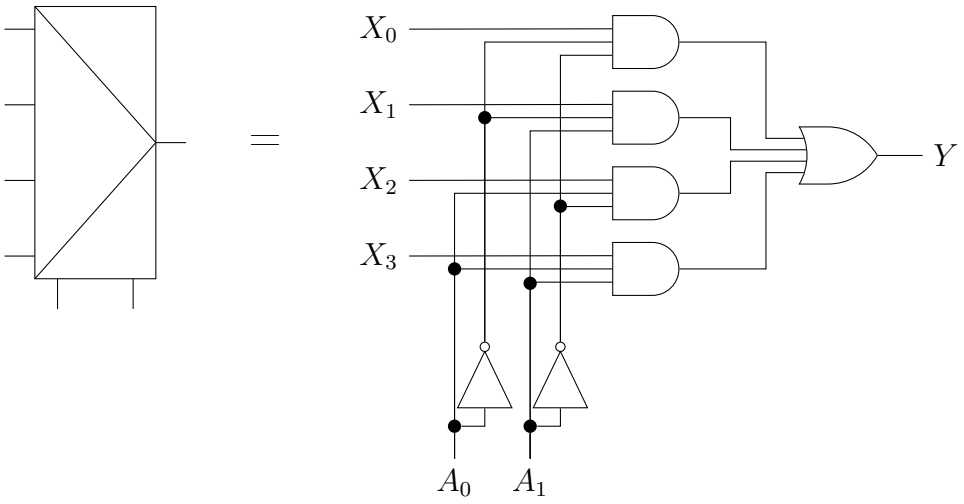
Another important arithmetic operation is comparing two values. This task is done by **comparators**.

The simplest comparator — for 1-bit values — is simply a negated XOR gate. It will return 1 if and only if the two input values coincide. To obtain a comparator checking equality of multi-bit values, one can combine many one-bit comparators with an AND gate. The resulting circuit will return 1 when the values on all corresponding pairs of input bits match.

Multiplexers

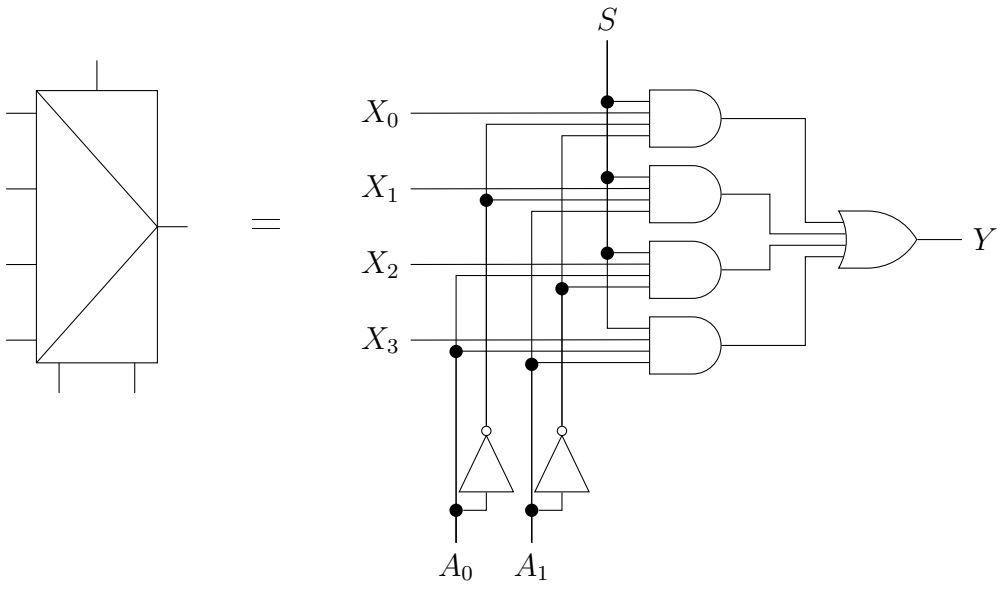
A **multiplexer** is used when, out of many **data inputs** $X_0, \ldots, X_{n-1}$, we wish to forward one value X_k — where the index k is determined by additional **selector inputs** (on which we'll typically

expect the representation of k in the binary positional system). On the left, we show the typical graphic symbol representing a multiplexer; on the right, we show an example multiplexer for $n = 4$:

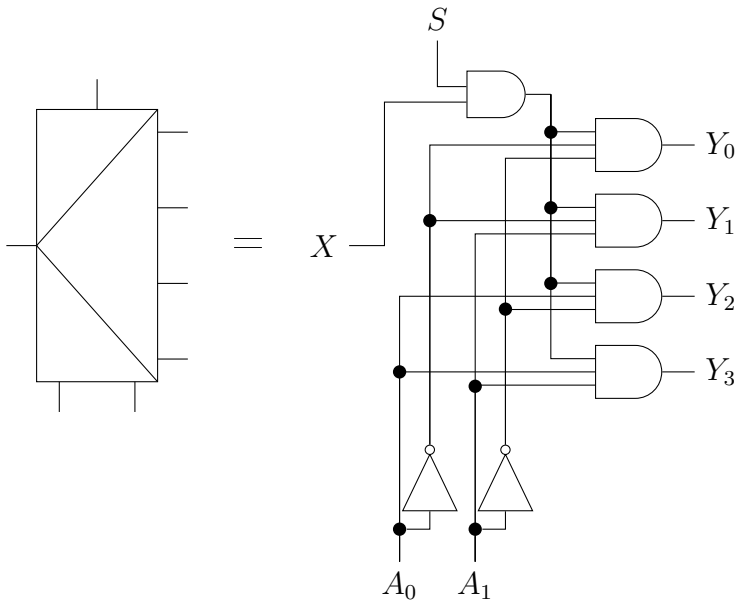


A_0	A_1	Y
0	0	$= X_0$
0	1	$= X_1$
1	0	$= X_2$
1	1	$= X_3$

In practice, one often adds another **strobe input** which makes the circuit active (i.e. passing data from the input) only when the strobe has value 1 (analogously to activating flip-flops by the clock signal value). Such an enhanced circuit may look as follows:



There's also some need for circuits acting in the opposite way: these are **demultiplexers** — having **many outputs** but only **one data input** and again some selector inputs, which are interpreted as an address (this time — defining to which output Y_k we should forward the input value X; the remaining otuputs Y_i will be just zeroed). For example, given the data input 1 and address 10, a demultiplexer should pass the value 1 to the output with index $10_{(2)} = 2$, that is, to Y_2. Again, we may enhance this with an additional strobe input.

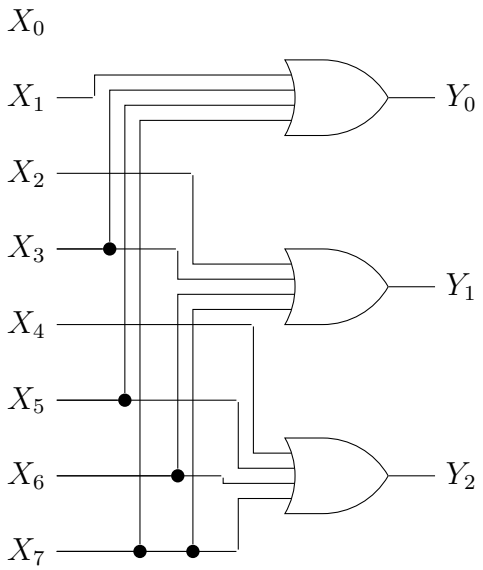


S	A_0	A_1	Y_0	Y_1	Y_2	Y_3
1	0	0	$=X$	0	0	0
	0	1	0	$=X$	0	0
	1	0	0	0	$=X$	0
	1	1	0	0	0	$=X$
0	whatever		0			

Coders and decoders

The last kind of integrated circuits to be discussed today are coders and decoders.

A **coder** is a circuit with many inputs — but assuming that exactly one of them will carry a "1" value. If that happens, the coder will process the index of input carrying the "1", and encode it in some shorter bit sequence (e.g. using the binary positional system representation). Below, we show a coder doing exactly that, for eight inputs and three outputs:



(The input X_0 is deliberately unused: even if $X_0 = 1$, this does not force any of the Y_i bits to have value 1. Moreover, as we assume that *exactly one* of the X_i bits is set, this means that, in fact, the value of X_0 can be deduced from the remaining values $X_1, \ldots, X_7$).

For example, for $X_5 = 1$ (and the remaining X_i zeroed), we get:

$$Y_0 = 1, \qquad Y_1 = 0, \qquad Y_2 = 1,$$

so the output signals Y_i indeed form a binary encoding of the number 5.

A **decoder** is a circuit acting conversely, that is, taking an encoded representation of some number on the input (e.g. its representation in the binary positional system), and setting the output with the appropriate index to 1, while all other outputs to 0. In case of the encoding based on binary system, a decoder can be obtained just by taking a demultiplexer of the appropriate size and setting its X and S inputs to 1 — this will produce a circuit decoding the signals A_i to the signals Y_i.

For completeness, let us mention that there also exist specialized circuits transforming one number encoding to another (so: equivalent to a combination of a decoder for one code and an encoder for another, though implemented more efficiently) — these are named **transcoders**.