UNIVERSITY OF BRISTOL
DEPARTMENT OF COMPUTER SCIENCE
http://www.cs.bris.ac.uk

A Practical Introduction to Computer Architecture

Dan Page
⟨page@cs.bris.ac.uk⟩

BASICS OF DIGITAL LOGIC

Scientists build to learn; Engineers learn to build.

– F. Brooks

In the previous Chapter, we made some statements regarding various features of digital logic without really backing them up with any real evidence or explanation. For example, we hinted that the binary digits 0 *and* 1 *could could be mapped onto electrical signals, and that this was useful in some way.*

This Chapter has two aims, taking a "from atoms upwards" approach. First it expands on previous statements, showing how *they can be satisfied using basic Physics. Since many people encounter Physics in school, and an in-depth discussion would require another book, the presentation is not rigorously accurate but simply a rough recap and overview of the pertinent details. Then, second, it illustrates* why *the result is so useful by building successively higher-level components capable of doing successively more involved computation. The Chapter culminates in techniques for concrete realisation of Finite State Machines (FSMs). This acts as a direct bridge between theory and practice, showing how simple, theoretical computers can be designed and manufactured using transistor-based logic gates.*

1 Switches and transistors

Even complex use of digital logic is, at the lowest level of detail, based on remarkably simple building blocks: fundamentally, all we really need is a way to manufacture a switch. We focus exclusively on **transistors**, whose design and behaviour depend on sub-atomic properties of the materials they are created from. There is a good reason for this focus: transistors are currently the dominant way to realise digital logic, and can be found in most if not all devices we routinely use. However, it is *crucial* to see that transistors are not the *only* option. Put another way, provided the right switch-like behaviour is provided, we can legitimately select *any* realisation we want. Looking forward to a time when new materials and manufacturing processes, applications and quality metrics become important, an understanding of the underlying principles is therefore as important as any one particular realisation.

1.1 A (very) quick tour of basic principles in Physics

1.1.1 Atoms and sub-atomic particles

Everything in the universe is composed from primitive building blocks called **atoms**. Each atom is composed from a group of sub-atomic particles: a group of nucleons, either **protons** or **neutrons**, in a central core or **nucleus**, and a cloud of **electrons** which orbit the nucleus. The number of protons dictates the **atomic number** (of family) of the atom; the number of neutrons dictates the **isotope**. Electrons can orbit the nucleus in one of several levels or **shells** in what is termed the **electron configuration**. For example,

- silicon has atomic number fourteen; it has three shells containing two, eight and four electrons respectively,

An aside: describing basic physics using the hydraulic analogy.

Since the electrical properties of atoms and sub-atomic particles may be unfamiliar, it is common (and useful) to explain them via the **hydraulic analogy**. Imagine a water tower (resp. battery) connected via a system of pipes (resp. wires) which eventually powers a water wheel (resp. lamp):

- The water pressure (resp. electrical potential) is dictated by how much water (resp. electrical charge) is held in the water tower,
- Water flows along the pipes; a wide pipe (resp. wire with low resistivity) allows water to flow more easily, and hence quicker, than a narrow pipe (resp. wire with low resistivity).
- When the water hits the water wheel, it causes it to turn as a result of two properties: the pressure of the water (resp. voltage), and the flow rate (resp. current).

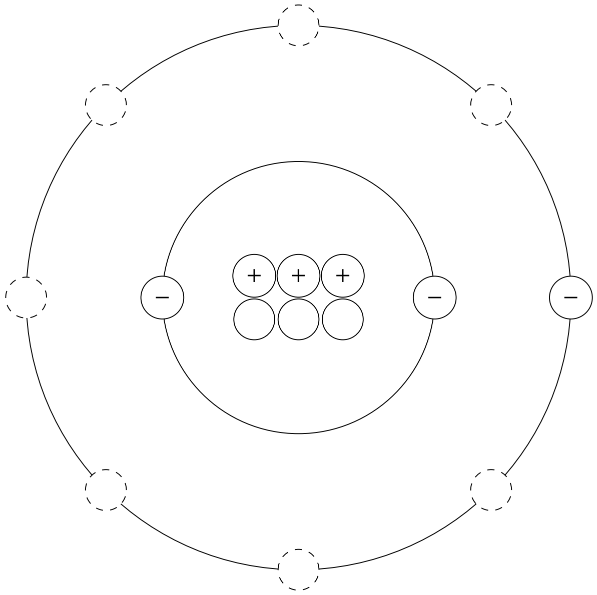


Figure 1: *The sub-atomic structure of a lithium atom.*

- lithium has atomic number three; it has two shells containing two and one electrons respectively.

Each sub-atomic particle carries an electrical **charge** characteristic of their type: electrons carry a negative charge, protons have a positive charge and neutrons have neutral charge. An **ion** is an atom with a non-neutral charge overall, meaning the number of protons and electrons differs: a negatively charged ion has more electrons in the electron cloud than protons in the nucleus, a positively charged ion has fewer electrons than protons.

1.1.2 Electrical charge and current

The sub-atomic particles are bound together in different ways. For example, nucleons are bound together by a "strong" nuclear force, whereas electrons are bound to the nucleus because of a "weak" electromagnetic attraction to the protons; electrons within more inner shells are bound more tightly. The "weak" binding of electrons means they can be displaced in a process called **ionisation**. The amount of energy required relates to how tightly coupled the electrons are to the nucleus, and is therefore dictated in part by the type of atom. This is described roughly by a conductivity "rating". For example a **conductor**, e.g., a metal, has high-conductivity (resp. low-resistivity) and allows electrons to move easily, but an **insulator**, e.g., a vacuum, has low-conductivity (resp. high-resistivity) and does not allow electrons to move easily.

Ionisation aside, electrons repel each other but are attracted by **holes** or space in an electron cloud. So given the right conditions electrons may move around between atoms, preferring to move from lower (negative charge) to higher (positive charge) electrical **potential** since this means they move toward holes

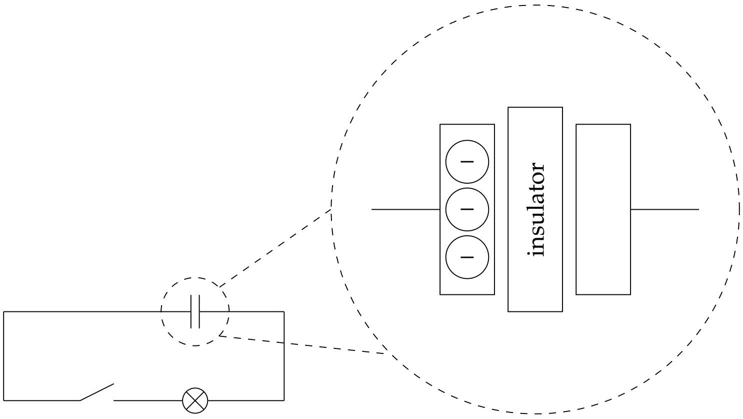


Figure 2: *A simple circuit conditionally connecting a capacitor (or battery) to a lamp depending on the state of a switch.*

(where there are fewer electrons and hence a more positive charge). This produces an electrical current, which is simply a flow of electrons, i.e., a flow of charge.

A good example of these principles in action is the use of a **capacitor** (or battery) to power a lamp. Figure 2 illustrates this sort of scenario. The capacitor is two conductors separated by an insulator; one of the conductors has many electrons and the other many holes. Since the electrons cannot jump across the insulator they instead flow along the wire (from low to high potential), causing a current which powers the lamp (when the switch is closed).

1.1.3 Manipulating atomic properties

All the properties described above depend on the materials we are using and their electron configurations: if we do not have a material with the right number of electrons or holes, it will not behave as required. However, we can *manipulate* a material to have the required properties via a process called **doping**. We take the starting material and dope it, or mix it, with a second material called the **donor**. Depending on the properties of the donor, the new material will have a different number of electrons or holes but otherwise retain very similar characteristics. As an example, consider pure silicon which has an electron cloud of four electrons (only about half full): it is more or less an insulator. Doping with a boron or aluminium donor creates extra holes while doping with phosphor or arsenic creates extra electrons. We call the new materials P-type and N-type **semiconductors** respectively; the material is neither conductor nor insulator, but somewhere in between.

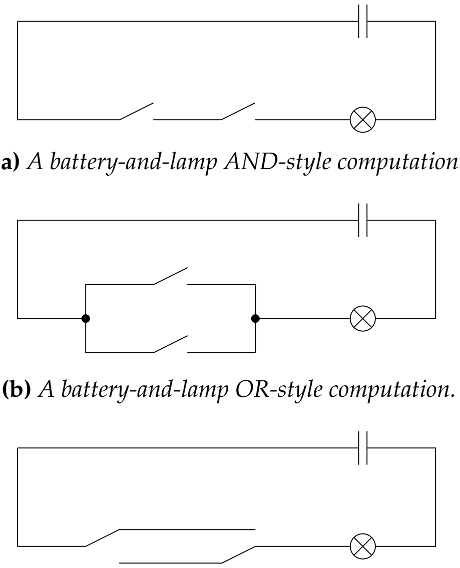
For certain tasks these materials are useful in isolation, but they become even more useful when used in combination. The idea is to sandwich them together in layers so that their behaviour overall matches our requirement. For example, given the description above it should be clear that electrons can flow from an N-type semiconductor layer to a P-type semiconductor layer but not in the other direction: the former has extra electrons, the latter has extra holes. Of course this is still a relatively simply behaviour. However, the general technique allows more complicated examples culminating in a switch, which basically adds control over when electrons can flow to which direction they flow in.

1.2 Building and packaging transistors

1.2.1 More advanced battery-and-lamp combinations: a step towards switches as computation

Before we dive into an explanation of transistors, a minor detour might be worthwhile: given *any* switch, whether a transistor, or not, why is it reasonable to assume we can do computation with it? For example, the switch within Figure 2 does not *seem* to do a lot; it just turns the lamp on or off. The idea is that if we use multiple switches, their *combined* state might turn the lamp on or off in interesting ways. More specifically, if we treat the switch states as inputs in some sense, the lamp state can be described by analogy to familiar Boolean operators. Figure 3 offers some simple examples:

1. Figure 3a controls the lamp via two switches, and is the analogy of an AND operator. Only when both of the switches are closed will the lamp be on: if either is open, there is no connection with the battery.



(a) *A battery-and-lamp AND-style computation.*

(b) *A battery-and-lamp OR-style computation.*

(c) *A battery-and-lamp XOR-style computation.*

Figure 3: *Some simple examples of Boolean-style control of a lamp by combinations of switches.*

2. Figure 3b controls the lamp via two switches, and is the analogy of an OR operator. When one or the other, or both the switches are closed will the lamp be on: there is connection with the battery unless both of the switches are open.

3. Figure 3c controls the lamp via two switches, and is the analogy of an XOR operator. This time, the switches sort of operate in the opposite way to each other; to make a connection between the lamp and battery along the top (resp. bottom) wire, the left-hand switch needs to be closed (resp. open) while the right-hand switch needs to be open (resp. closed): there is connection with the battery if one or the other, but not both switches are closed.

   You often find this sort of arrangement in homes where a single light on some stairs is controlled by switches located at the top *and* bottom.

1.2.2 History

As discussed, transistors are not the only way to realise this switch-like building block. Among various alternatives, the electrical **relay** and **vacuum tube** (or **thermionic valve**), which resembles a light bulb, represent two examples. The latter is perhaps more compelling, because it was used extensively by early generations of computing equipment; it still often plays a role in high-end audio equipment. The idea is to use a glass or ceramic envelope to maintain a vacuum surrounding an electron-producing filament (or cathode) and a metal plate (or anode). When the filament is heated, electrons are produced into the vacuum which are attracted by the plate resulting in a current between the two. In simple terms, this implements a switch: when the filament is heated the switch is on, when it is cooled the switch is off.

Several features of vacuum tubes turn out to be problematic however. As hinted at by Figure 4, they are physically quite large and operate relatively slowly. Crucially however, they are also unreliable. This unreliability relates in part to the filament degrading as a result of impurities in the tube, and the vacuum in the envelope which becomes less pure as time progresses. The heating and cooling process stresses the filament, and will eventually lead to it burning out; early users realised that this often happened during power-on or power-off of a vacuum tube-based device (much like a light bulb failing when turned on or off). Resolving these disadvantages, namely switching speed, size, reliability and so on, forms an ongoing challenge for modern alternatives such as transistors: particularly when the number of switches grows larger, they become *the* central limitation on complexity of digital logic components realised using them.

As an aside, the terms **bug** and **debug** in relation to programming both stem (at least in part) from failure of this sort. In 1945, programmers of the Harvard Mark II computer developed by Howard Aiken, discovered a moth inside one of the components; the unfortunate insect had shorted the component, which had resulted in the malfunction. Although it now seems the terms had been used previously in various other contexts, Grace Hopper and her team are often cited as introducing them to Computer Science. Certainly

Figure 4: *An operational vacuum tube (public domain image, source: http://en.wikipedia.org/wiki/File:5651RegulatorTubeInOperation.jpg).*

this real-life bug, shown in Figure 5, is so famous that it is still on display in the Smithsonian Museum of American History!

1.2.3 MOSFET transistors

A transistor can be used for a variety of tasks, for example they can amplify signals, but when used as switches they

1. allow charge to flow between two terminals (i.e., act as a conductor) when we turn the the switch "on", and
2. prevent change flowing between two terminals (i.e., act as a resistor) when we turn the the switch "off".

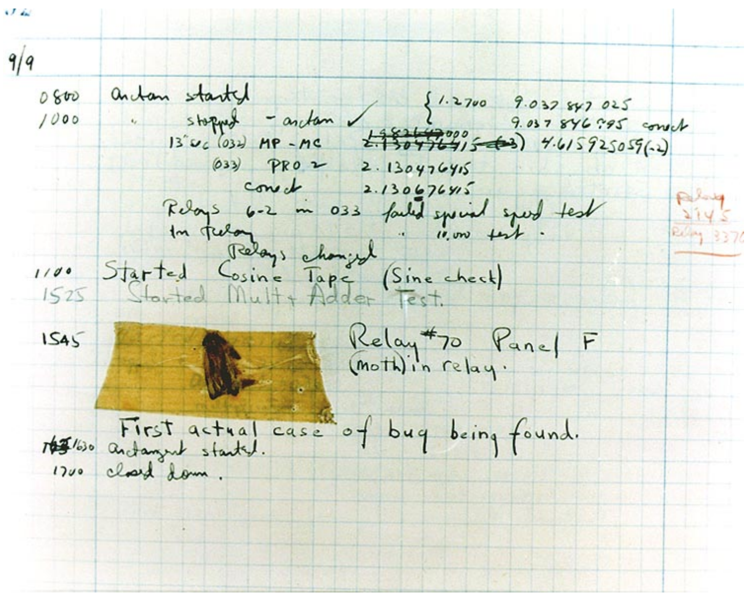
The word itself is a portmanteau of "transfer resistor" which hints at the underlying principle: a transistor is a resistor, but one we can control by altering *how* resistive it is. Improvement and different behavioural trade-offs have give us numerous transistor designs, but we focus on just one: the **Field Effect Transistor (FET)**, initially designed and patented by Julius Lilienfeld in 1925. At that point in time the general understanding of sub-atomic behaviour was less than now, meaning use of his design was limited: better understood alternatives (e.g., vacuum tubes) were preferred. In 1952 this changed when a team of Engineers, led by William Shockley at Bell Labs, invented what is now termed a junction gate FET (or JFET, after some legal wranglings wrt. to the original Lilienfeld patent). In turn, this gave rise to the **Metal Oxide Semiconductor Field-Effect Transistor (MOSFET)**, invented in 1960 by Dawon Kahng and Martin Atalla, also at Bell Labs. Crucially, such components solve problems relating to older technologies: they are easy to manufacture, physically small, operate very quickly, and are very reliable.

As motivated previously, the idea is to construct a MOSFET transistor from atomic-scale layers of semiconductor material plus metal or poly-silicon for the terminals; we term the two terminals the **source** and **drain**, and control the flow of current via the **gate**. A high-level description of the design is shown in Figure 7. Although a more accurate description requires a deeper investigation of the underlying Physics, the basic idea is to control the size of a so-called **depletion region** between the source and drain.

The resulting behaviour depends on our selection of materials. The two choices are represented symbolically per Figure 8, which abstracts away the implementation detail, retaining only the three terminals (where d, s and g denote the drain, source and gate); their behaviour is as follows.

Definition 0.1 *An* **N-MOSFET** *(or* **N-type MOSFET**, *or* **N-channel MOSFET**, *or* **NPN MOSFET***) is constructed from N-type semiconductor terminals and a P-type body:*

- *A positive potential difference between source and gate widens the depletion channel, meaning source and drain are connected (i.e., act like a conductor).*



9/9

0800 Anctan started

1000 " stopped - anctan ✓ {1.2700 9.037 847 025
9.037 846 995 conect

13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ (-3) 4.615925059(-2)

(033) PRO 2 2.130476415

conect 2.130676415

Relays 6-2 in 033 failed special speed test
In relay " 11,000 test.

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Mult + Adder Test.

1545 Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.

1630 Antangent started.

1700 closed down.

Relay 2145
Relay 3370

Figure 5: *A moth found by operations of the Harvard Mark 2; the "bug" was trapped within the computer and caused it to malfunction (public domain image, source: http://en.wikipedia.org/wiki/File:H96566k.jpg).*

- *A negative potential difference between source and gate narrows the depletion channel, meaning source and drain are disconnected (i.e., act like an insulator).*

Definition 0.2 *A* **P-MOSFET** *(or* **P-type MOSFET***, or* **P-channel MOSFET***, or* **PNP MOSFET***) is constructed from P-type semiconductor terminals and an N-type body:*

- *A positive potential difference between source and gate narrows the depletion channel, meaning source and drain are disconnected (i.e., act like an insulator).*

- *A negative potential difference between source and gate widens the depletion channel, meaning source and drain are connected (i.e., act like a conductor).*

Put another way, for an N-MOSFET a large potential difference between source and gate means a wider depletion region, allowing electrons, i.e., current, to flow in a channel that links the source and drain; a small potential difference (or at least smaller than some threshold) means a narrower depletion region which closes the channel. That is, changing the potential difference between gate and source changes the conductivity between source and drain, and hence regulates the current: if the potential difference is small, the flow of current is small and vice versa. Thus, the gate acts like a switch that can regulate the current between source and drain. This regulation has limits however. Unlike a light switch, which is either on or off, imperfections in the materials involved mean MOSFET transistors always allow a small level of conductivity between source and drain: they are never *fully* off. This effect is termed **leakage**, and relates roughly to the threshold voltage at which the depletion region is wide or narrow enough to be deemed effective.

Once we have a single transistor, the next step is to combine several transistors together in order to produce more complicated behaviour. Essentially this is a **circuit**, namely a network of transistors whose terminals are connected to each other or **power rails** that supply fixed V_{dd} and GND voltage levels.

1.2.4 CMOS-based logic gates

Rather than being used in isolation, MOSFET transistors are more commonly organised into larger components ultimately describing a **logic style**; this helps make the transistors easier to reason about both functionally *and* behaviourally, and can imply a particular approach to design.

A popular first step relates to organisation of the transistors themselves into **Complementary Metal-Oxide Semiconductor (CMOS)** cells; it was invented in 1963 by Frank Wanlass at Fairchild Semiconductor. The idea is to pair together N-MOSFET and P-MOSFET transistors as illustrated at a high-level by Figure 9; the two transistors are organised in a complementary manner so whenever one is on (i.e., connected), the other is off (i.e., disconnected). As such, the use of CMOS to design a circuit means specification of two parts: a so-called **pull-up network** of P-MOSFET transistors which sit between the positive power

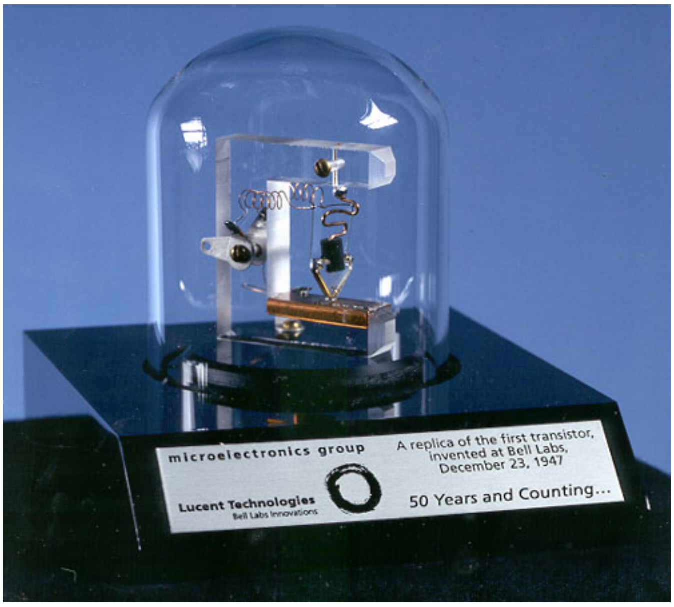


Figure 6: *A replica of the first point-contact transistor, a precursor of designs such as the MOSFET, constructed at Bell Labs (public domain image, source: http://en.wikipedia.org/wiki/File:Replica-of-first-transistor.jpg).*

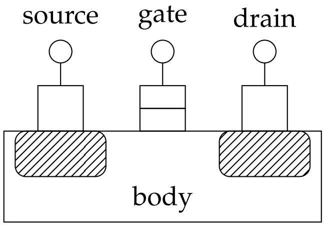


Figure 7: *A high-level diagram of a MOSFET transistor, showing the terminal and body materials.*

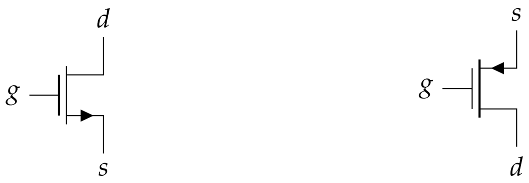
rail (supplying the V_{dd} voltage level) and the output, and a **pull-down network** of N-MOSFET transistors which sit between the negative power rail (supplying the GND voltage level) and the output. Only one of the networks will be active at a time, so aside from leakage a CMOS cell only consumes power when the inputs are **switched** (or **toggled**). Since we commonly aim to construct circuits with many transistors in close proximity to each other, this feature provides some significant advantages. In particular it means that the use of CMOS reduces power consumption and heat dissipation, which in turn aids reliability.

The next step is to package CMOS cells into small, commonly used circuits that act as the next-level component above transistors themselves. Such components are often termed **logic gates**: given some input, the idea is that they compute an output relating to the Boolean algebra covered in Chapter 1. That is,

- there we had a set of values, namely 1 and 0, and some binary and unary operators AND, OR and NOT, whereas
- here we have a set of values, namely V_{dd} and GND, and some binary and unary operators NAND, NOR and NOT.

For example, consider building a component which inverts the input, i.e., a logic gate for the NOT operator: when the input x is V_{dd}, the output is GND and vice versa. The behaviour required is realised by Figure 10a; to see that it works as required, we simply enumerate the different voltage levels x can take and work out the corresponding behaviour of each transistor. In short

1. connecting x to GND means the top P-MOSFET will be connected, the bottom N-MOSFET will be disconnected, so r will be connected to V_{dd}. while
2. connecting x to V_{dd} means the top P-MOSFET will be disconnected, the bottom N-MOSFET will be connected, so r will be connected to GND.



(a) *An N-MOSFET transistor.* **(b)** *A P-MOSFET transistor.*

Figure 8: *Symbolic descriptions of N-MOSFET and P-MOSFET transistors.*

An aside: naming conventions for voltage levels.

GND is fairly obvious in the sense it is the ground: roughly speaking it is a reference point other voltages are measured relative to. But why V_{dd}?! What does it actually *mean*? The answer is basically that the '*d*' stands for drain: V_{dd} can be read as "voltage level at the drain" and therefore it also makes sense to have V_{ss} read as "voltage level at the source" for MOSFET-based transistors. Apparently the naming convention stems originally from earlier bipolar-based transistors, where V_{cc} and V_{ee} are sort of the same thing but for collector and emitter terminals.

This all starts to become a little involved however, and beyond the scope of what we want to discuss. All we really care about is that *GND* and V_{dd} make our transistors work correctly, and we can tell them apart. Although it might be too informal for some tastes, it is therefore enough to keep the following in mind:

- V_{dd} is the high, drive voltage level, e.g., 3.3 V or 5 V say, and
- *GND* is the low, ground voltage level, e.g., 0 V.

Or, if $x = V_{dd}$ then $r = GND$ or $x = GND$ then $r = V_{dd}$. Note that even with this simply arrangement, we can already see the pull-up and pull-down networks (here there is just one transistors in each) in action. Further, more complicated components in a conceptually similar way; two examples are NAND (or NOT AND) and NOR (or NOT OR) in Figure 10b and Figure 10c respectively. We can reason about their behaviour in exactly the same way as above: for NAND,

1. connecting both x and y to *GND* means both top P-MOSFETs will be connected, both bottom N-MOSFETS will be disconnected, so r will be connected to V_{dd},
2. connecting x to V_{dd} and y to *GND* means the right-most P-MOSFET will be connected, the upper-most N-MOSFET will be disconnected, so r will be connected to V_{dd},
3. connecting x to *GND* and y to V_{dd} means the left-most P-MOSFET will be connected, the lower-most N-MOSFET will be disconnected, so r will be connected to V_{dd}, while
4. connecting both x and y to V_{dd} means both top P-MOSFETs will be disconnected, both bottom N-MOSFETS will be connected, so r will be connected to *GND*.

Likewise for NOR,

1. connecting both x and y to *GND* means both top P-MOSFETs will be connected, both bottom N-MOSFETS will be disconnected, so r will be connected to V_{dd},
2. connecting x to V_{dd} and y to *GND* means the upper-most P-MOSFET will be disconnected, the left-most N-MOSFET will be connected, so r will be connected to *GND*,
3. connecting x to *GND* and y to V_{dd} means the lower-most P-MOSFET will be disconnected, the right-most N-MOSFET will be connected, so r will be connected to *GND*, while
4. connecting both x and y to V_{dd} means both top P-MOSFETs will be disconnected, both bottom N-MOSFETS will be connected, so r will be connected to *GND*.

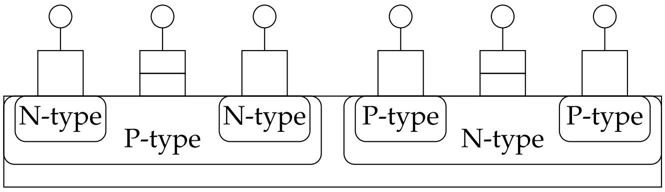


Figure 9: *A pair of N-MOSFET and P-MOSFET transistors, arranged to form a CMOS cell.*

So, from a starting point involving atomic-level concepts we now have components which we can reason about wrt. form *and* function. In short, we have a concrete implementation that relates directly to the underlying theory of Boolean algebra; instead of simply reasoning about computation in theory, we can now actually build (simple) components that actually *do* computation in practice.

2 Combinatorial logic

2.1 A suite of simplified logic gates

It is somewhat cumbersome to work at the transistor-level because of the low level of detail: worrying about the behaviour of transistors exacerbates the challenge of designing components with some required function. To make life (a lot) easier, we commonly adopt a more abstract view of logic gates by taking two steps:

1. we forget about the voltage levels GND and V_{dd}, abstractly calling them 0 and 1, then

2. we forget about the power rails and just *draw* each gate using a single symbol with inputs and outputs.

Now, by treating the gates as operators per Boolean algebra we can combine them together and design components that fall into a category often termed **combinatorial logic**; the gate behaviours combine to compute a result continuously, with their output updated whenever an input changes.

Figure 12 highlights several different notations for the resulting logic gates, including each of the NOT, NAND and NOR gates from above and also AND, OR and XOR from Chapter ; corresponding truth tables are shown in Figure 13. In addition to the notation introduced in Chapter , keep in mind the following when dealing with this suite of logic gates:

- An **inversion bubble** on the output of a gates is used to denote that fact that the output is inverted. As such, a buffer (or BUF) is simply a gate that passes the input straight through to the output; a NOT gate just a buffer that inverts the input to form the output. We encountered some uses for buffer gates later.

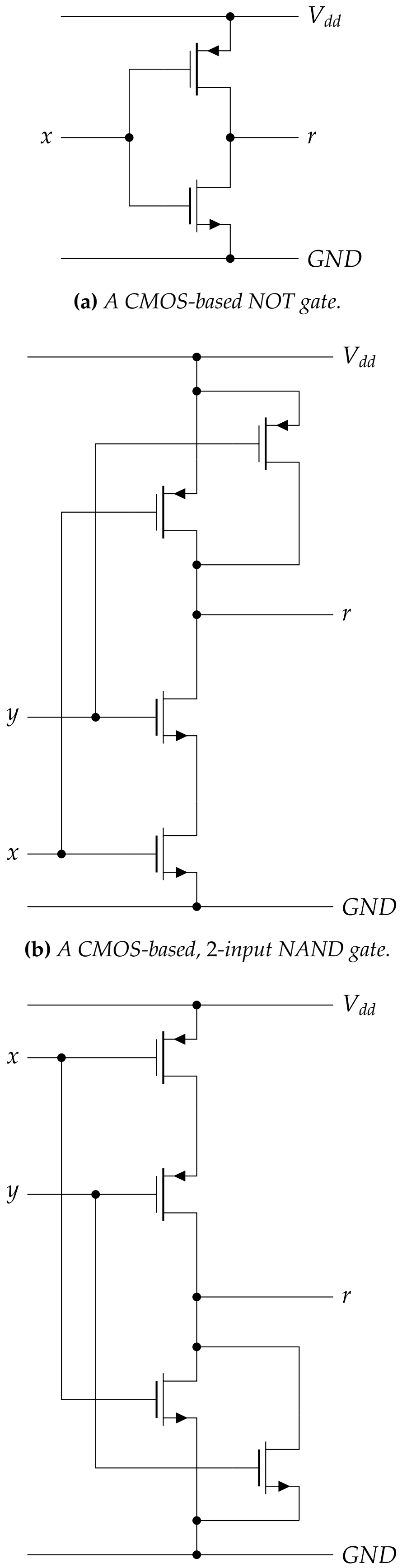
- For completeness we have included the NXOR (sometimes written XNOR) gate which has the obvious meaning but is seldom used in practise; per Chapter 1, we use $\overline{\wedge}$, $\overline{\vee}$ and $\overline{\oplus}$ as a short-hand to denote NAND, NOR and NXOR respectively. Clearly, for example, we have

$$x \overline{\wedge} y \equiv \neg(x \wedge y).$$

- For 2-input gates such as AND, OR and XOR we often use a short-hand and draw the gates with more inputs; this is equivalent to making a tree of 2-input gates since, for example, we have

$$(w \wedge x \wedge y \wedge z) \equiv (w \wedge x) \wedge (y \wedge z).$$

The natural question to now ask is why on earth we built transistor-based NAND and NOR gates when AND and OR seem to be more useful?! The answer is related to the cost of implementation; it is simply more costly to build an AND gate from transistors, the cheapest way being to compose a NAND gate with a NOT gate. Since NAND and NOR are the cheapest meaningful components we can build, it makes sense to construct a circuit from these alone. This is possible because NAND and NOR are both **universal** in the sense that one can implement *all* the other logic gates using them alone; one simply substitutes gates using the identities in Figure 14 to perform the translation.



(a) *A CMOS-based NOT gate.*

(b) *A CMOS-based, 2-input NAND gate.*

(c) *A CMOS-based, 2-input NOR gate.*

Figure 10: *MOSFET-based implementations of NOT, NAND and NOR logic gates.*

x	y	NOT	NAND	NOR
GND	GND	V_{dd}	V_{dd}	V_{dd}
GND	V_{dd}	V_{dd}	V_{dd}	GND
V_{dd}	GND	GND	V_{dd}	GND
V_{dd}	V_{dd}	GND	GND	GND

Figure 11: *A voltage-oriented truth table for NOT, NAND and NOR logic gates.*

r is x	$\equiv$	$r = x$	$\equiv$	`r = x`	$\equiv$
r is NOT x	$\equiv$	$r = \neg x$	$\equiv$	`r = ~x`	$\equiv$
r is x NAND y	$\equiv$	$r = x \overline{\wedge} y$	$\equiv$	`r = ~( x & y )`	$\equiv$
r is x NOR y	$\equiv$	$r = x \overline{\vee} y$	$\equiv$	`r = ~( x | y )`	$\equiv$
r is x AND y	$\equiv$	$r = x \wedge y$	$\equiv$	`r = x & y`	$\equiv$
r is x OR y	$\equiv$	$r = x \vee y$	$\equiv$	`r = x | y`	$\equiv$
r is x XOR y	$\equiv$	$r = x \oplus y$	$\equiv$	`r = x ^ y`	$\equiv$

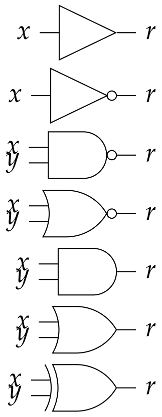


Figure 12: *Representation of standard logic gates in English, Boolean algebra,* C *and symbolic notations.*

2.2 Designing circuits for arbitrary combinatorial functions

Now that we have logic gates that act as physical implementations of the operators in Boolean algebra, the next challenge is to produce Boolean expressions (and hence circuits) for some (arbitrary) Boolean function, say f. Put another way, the challenge is to take a specification of f, e.g., a truth table, and produce a Boolean expression that computes it.

Fortunately, we have already covered a lot of material which enables us to do this in a fairly mechanical way. In fact, there are *several* methods that can answer the challenge: each produces a SoP form Boolean expression as a result.

2.3 Some design patterns

Before we look at how to produce Boolean expressions for arbitrary Boolean functions, it can be useful to start with some more specific examples that can be solved via simpler design patterns. As a motivating example, imagine have a 2-input, 1-bit AND gate:

1. If, within some larger circuit, we use an AND gate to compute

$$r = x \wedge y$$

and then, somewhere else,

$$r' = x \wedge y$$

then we can replace the two AND gates with one since clearly $r = r'$; the result is to **share** the gate between two usage points. The same simplification is harder to capture within a single Boolean expression. For example, if we had

$$r = (w \wedge x \wedge y) \vee (x \wedge y \wedge z)$$

it is usual to first define some intermediate, say

$$t = x \wedge y$$

then rewrite the expression as

$$r = (w \wedge t) \vee (t \wedge z).$$

This acts as an analogue to optimisations like common sub-expression elimination which are commonly applied by, for example, C compilers.

BUF	
x	r
0	0
1	1

(a) *A 1-input, 1-output buffer.*

AND		
x	y	r
0	0	0
0	1	0
1	0	0
1	1	1

(b) *A 2-input, 1-output AND gate.*

OR		
x	y	r
0	0	0
0	1	1
1	0	1
1	1	1

(c) *A 2-input, 1-output OR gate.*

XOR		
x	y	r
0	0	0
0	1	1
1	0	1
1	1	0

(d) *A 2-input, 1-output XOR gate.*

NOT	
x	r
0	1
1	0

(e) *A 1-input, 1-output NOT gate.*

NAND		
x	y	r
0	0	1
0	1	1
1	0	1
1	1	0

(f) *A 2-input, 1-output NAND gate.*

NOR		
x	y	r
0	0	1
0	1	0
1	0	0
1	1	0

(g) *A 2-input, 1-output NOR gate.*

NXOR		
x	y	r
0	0	1
0	1	0
1	0	0
1	1	1

(h) *A 2-input, 1-output NXOR gate.*

Figure 13: *Truth tables for standard logic gates.*

2. A 2-input, m-bit AND gate is simply **replication** of 2-input, 1-bit AND gates. That is, If x and y are m-bit values then

$$r = x \wedge y$$

is computed by

$$r_i = x_i \wedge y_i$$

for $0 \leq i < m$, i.e., m separate instantiations of the 2-input, 1-bit gate.

3. An n-input, 1-bit AND gate is simply a **cascade** of 2-input, 1-bit AND gates. That is,

$$r = \bigwedge_{i=0}^{n-1} x_i$$

for $n = 4$ is the same as

$$r = (x_0 \wedge x_1) \wedge (x_2 \wedge x_3).$$

Notice that this expression forms a tree of AND gates, and that the balanced tree above is more attractive than equivalents such as

$$r = x_0 \wedge (x_1 \wedge (x_2 \wedge x_3))$$

due, for example, to the shorter critical path of the former.

Of course these examples are specific to AND gates, but the point is that the *patterns* apply more generally.

2.4 Mechanical derivation method #1

Imagine we are tasked with deriving a Boolean expression that implements some Boolean function f. The function has n inputs $\mathcal{I}_0, \mathcal{I}_1, \ldots, \mathcal{I}_{n-1}$, and one output $\mathcal{O}$; we are given a truth table (with 2^n rows) which describes it. The idea is to follow a (fairly) simply algorithm:

1. Find a set T such that $i \in T$ iff. $\mathcal{O} = 1$ in the i-th row of the truth table.

2. For each $i \in T$, form a term t_i by AND'ing together all the variables while following two rules:

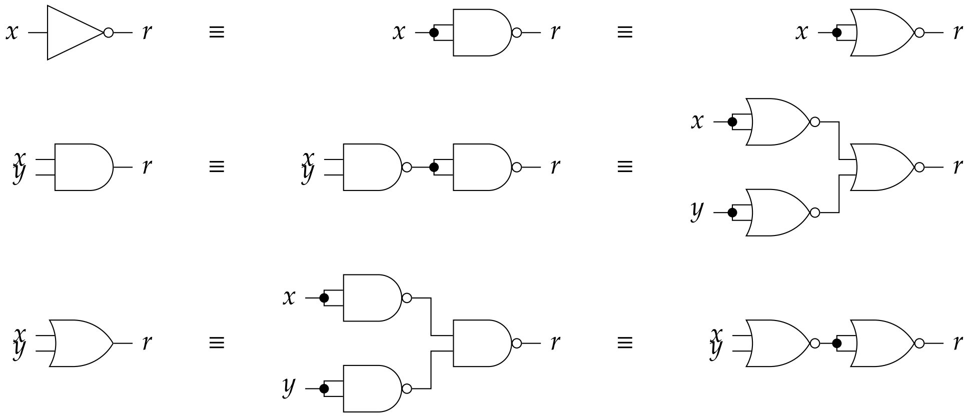


Figure 14: *Identities for standard logic gates in terms of NAND and NOR.*

(a) if $\mathcal{I}_j = 1$ in the i-th row, then we use

$$\mathcal{I}_j$$

as is, but

(b) if $\mathcal{I}_j = 0$ in the i-th row, then we use

$$\neg\mathcal{I}_j.$$

3. An expression implementing the function is then formed by OR'ing together all the terms, i.e.,

$$e = \bigvee_{i \in T} t_i,$$

which is in SoP form.

The large $\vee$ symbol used here is similar to the summation and product symbols (i.e., Σ and Π): it applies OR to many terms in a similar way to Σ and Π, which apply addition and multiplication.

An example makes the method easier to understand: consider the task of implementing an expression for XOR, whose truth table can be found in Figure 13 and reproduced here:

XOR		
x	y	r
0	0	0
0	1	1
1	0	1
1	1	0

1. Looking at the truth table, it is clear there are

   - $n = 2$ inputs that we denote $\mathcal{I}_0 = x$ and $\mathcal{I}_1 = y$, and
   - one output that we denote $\mathcal{O} = r$.

   Clearly $T = \{1, 2\}$ since $\mathcal{O} = 1$ in rows 1 and 2, while $\mathcal{O} = 0$ in rows 0 and 3.

2. Each term t_i for $i \in T = \{1, 2\}$ is formed as follows:

   - For $i = 1$, we find
     - $\mathcal{I}_0 = x = 0$ and so we use $\neg x$,
     - $\mathcal{I}_1 = y = 1$ and so we use y

     and hence form the term $t_1 = \neg x \wedge y$.
   - For $i = 2$, we find
     - $\mathcal{I}_0 = x = 1$ and so we use x,
     - $\mathcal{I}_1 = y = 0$ and so we use $\neg y$

     and hence form the term $t_2 = x \wedge \neg y$.

3. The expression implementing the function is therefore

$$\begin{array}{rcl} e & = & \bigvee_{i \in T} t_i \\ & = & \bigvee_{i \in \{1,2\}} t_i \\ & = & (\neg x \wedge y) \vee (x \wedge \neg y) \end{array}$$

   which is in SoP form.

The intuition behind the algorithm is that each $i \in T$ produces a minterm t_i in the SoP form; each minterm *fully* specifies values of the inputs for one case where the output is 1. In a sense, we are "covering" each row where the output is 1 by exhaustively listing the associated input values. For example, the case where $i = 1$ (and we examine the 1-st row) produces the minterm $\neg x \wedge y$ meaning "the row where $x = 0$ *and* $y = 1$". By combining all the minterms together in the final step, we get something which specifies the rows where the output should be 1 as "either $x = 0$ *and* $y = 1$, *or* $x = 1$ *and* $y = 0$".

2.5 Mechanical derivation method #2: Karnaugh maps

Consider the truth table in Figure 15a describing a 4-input Boolean function, and the derivation of an associated SoP expression using the method from above. After all the working out, we would end up with the following

$$\begin{array}{rcccccccccccl} r & = & (& \neg w & \wedge & \neg x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& \neg w & \wedge & \neg x & \wedge & \neg y & \wedge & z &) & \vee \\ & & (& \neg w & \wedge & \neg x & \wedge & y & \wedge & \neg z &) & \vee \\ & & (& \neg w & \wedge & x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& \neg w & \wedge & x & \wedge & \neg y & \wedge & z &) & \vee \\ & & (& w & \wedge & \neg x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& w & \wedge & \neg x & \wedge & y & \wedge & \neg z &) & \vee \\ & & (& w & \wedge & x & \wedge & y & \wedge & z &) & \end{array}$$

which is verbose to say the least!

The **Karnaugh map**, is an alternative method invented in 1953 by Maurice Karnaugh while working at Bell Labs [?]. It has two major advantages over the previous method: first it is more visual, and hence human-friendly, and second it accommodates various optimisations in a natural way, i.e., *without* resorting to manipulation via axioms. Although an example is more useful, the method can be summarised as another algorithm; again imagine we are tasked with deriving a Boolean expression that implements some Boolean function f with n inputs and one output.

1. Draw a $(p \times q)$-element grid, st. $p \cdot q = 2^n$ and each row and column represents one input combination; order rows and columns according to a **Gray code**.
2. Fill the grid elements with the output corresponding to inputs for that row and column.
3. Cover *rectangular* groups of adjacent 1 elements which are of total size 2^m for some m; groups can "wrap around" edges of the grid and overlap.
4. Translate each group into one term of an SoP form Boolean expression e where
   (a) *bigger* groups, and
   (b) *less* groups

   mean a simpler expression.

The intuition behind the method is fairly simply: if we can locate minterms, (i.e., cases where the output of the function is 1) which differ in only one input, we can simplify the resulting expression by eliminating that input. In our example, the minterms

$$(w, x, y, z) = (1, 0, 1, 0)$$

and

$$(w, x, y, z) = (1, 0, 1, 1)$$

An aside: binary versus Gray code.

Consider unsigned integers represented using n-bit sequences; selecting $n = 4$ for example and starting from zero, a sequence of such integers could be written as follows

$$\begin{array}{lcl} \langle 0,0,0,0 \rangle & \mapsto & 0_{(10)} \\ \langle 1,0,0,0 \rangle & \mapsto & 1_{(10)} \\ \langle 0,1,0,0 \rangle & \mapsto & 2_{(10)} \\ \langle 1,1,0,0 \rangle & \mapsto & 3_{(10)} \\ \langle 0,0,1,0 \rangle & \mapsto & 4_{(10)} \\ \langle 1,0,1,0 \rangle & \mapsto & 5_{(10)} \\ \langle 0,1,1,0 \rangle & \mapsto & 6_{(10)} \\ \langle 1,1,1,0 \rangle & \mapsto & 7_{(10)} \\ & \vdots & \end{array}$$

Notice that moving from $\langle 1,1,0,0 \rangle$ to the next entry $\langle 0,0,1,0 \rangle$ flips three bits: the 0-th and 1-st bits flip from 1 to 0, and the 2-nd bit from 0 to 1. Now consider an alternative ordering of the same integers:

$$\begin{array}{lcl} \langle 0,0,0,0 \rangle & \mapsto & 0_{(10)} \\ \langle 1,0,0,0 \rangle & \mapsto & 1_{(10)} \\ \langle 1,1,0,0 \rangle & \mapsto & 3_{(10)} \\ \langle 0,1,0,0 \rangle & \mapsto & 2_{(10)} \\ \langle 0,1,1,0 \rangle & \mapsto & 6_{(10)} \\ \langle 0,0,1,0 \rangle & \mapsto & 4_{(10)} \\ \langle 1,0,1,0 \rangle & \mapsto & 5_{(10)} \\ \langle 1,1,1,0 \rangle & \mapsto & 7_{(10)} \\ & \vdots & \end{array}$$

Here, moving from *any* entry to the next or previous flips one bit only: the ordering is called a **Gray code** after Frank Gray who cited it in a patent application in 1953; such codes had been known and used for a couple of hundred years before that.

As described above, the resulting sequence is simply a permutation of the original, more natural sequence; for what we use them for, it is enough to believe that such results are possible for any n, and also for other bases (i.e., other than binary).

w	x	y	z	r
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

(a) *A* 4*-input example.*

x	y	z	r
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	0	1
1	1	1	**?**

(b) *A* 3*-input example.*

Figure 15: 4*- and* 3*-input example Boolean functions respectively.*

(representing the 10-th and 11-th rows of the truth table) offer a good demonstration of this fact in the sense they differ only wrt. z (which is 0 in the first case and 1 in the second case). In our previous method, we would have implemented them using the two terms

$$w \wedge \neg x \wedge y \wedge \neg z$$

and

$$w \wedge \neg x \wedge y \wedge z.$$

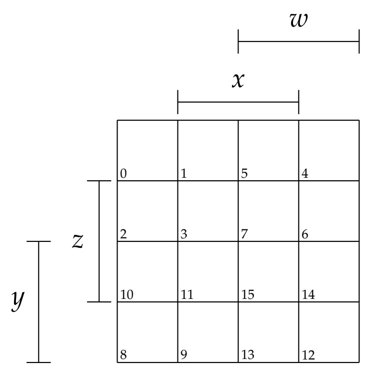
This is clearly wasteful however; it does not matter what value z takes since the overall result is still 1 as long as $w = 1$, $x = 0$ and $y = 1$. Thus we can eliminate the z term from our expressions above, and simply use the more simply alternative

$$w \wedge \neg x \wedge y$$

to cover both cases.

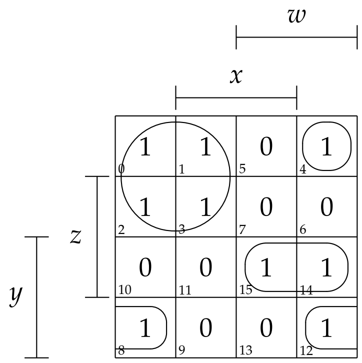
2.5.1 A simple example

The best way to illustrate this in practice is to fully examine the the truth table in Figure 15a. The first two steps act to encode this truth table on a grid; in this case the grid has $2^4 = 16$ elements, and so we can select $p = q = 4$ and draw the following square grid



Looking at the grid, the left-most column represents the values where $(w, x) = (0, 0)$ (i.e., $w = 0$ and $x = 0$) and reading from top to bottom where $(y, z) = (0, 0)$, $(0, 1)$, $(1, 1)$ and $(1, 0)$. The next columns are similar for y and z but for the cases where $(w, x) = (0, 1)$, $(1, 1)$ and $(1, 0)$. Put another way, using this diagrammatic representation the bars above and to the left of the grid denote where the associated variable is 1: the 1-st and 2-nd (or middle) columns are where $x = 1$ for example, and the 0-th and 3-rd (or outer) are where $x = 0$. Elsewhere you might also see numbers to the left of each row, or above each column to make the value of variables more explicit. Either way, the ordering might, reasonably, seem odd: note that in row- and column-wise directions, a Gray code is used. So from top to bottom, elements in a column are for $(y, z) = (0, 0)$, $(0, 1)$, $(1, 1)$ and $(1, 0)$ *not* $(y, z) = (0, 0)$, $(0, 1)$, $(1, 0)$ and $(1, 1)$ which might seem more natural. The value of this choice will become apparent as we progress through the example, but the rough idea is that *it* is what allows a more optimised result per the discussion above.

The next step is to cover 1 elements in the grid. In a sense this is analogous to what we did in the previous method by identifying rows in the truth table where the output was 1: there we would have a group for *each* 1, here we can form larger groups and cover more than a single 1 with each. As long as we follow the rules (i.e., form rectangular, potentially overlapping groups of size 2^m), each of the groups we finally end up with represents one term that we need to implement as part of the overall expression. The bigger the group, the less variables we need to include in each of the terms; the less groups, the less terms. An example grouping for example is as follows:



Here we have four groups:

1. a group of four elements in the top left-hand corner spanning the 0-th and 1-st rows and columns,

2. a group of one element in the top right-hand corner,

3. a group of two elements in the 2-nd row spanning the 2-nd and 3-rd columns, and

4. a group of two elements which wrap around the bottom-left and bottom-right corners.

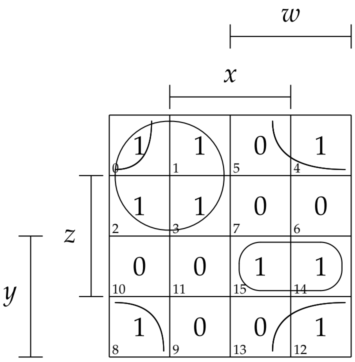
Consider the first group of four elements in the top left-hand corner. Looking at how the variables are assigned values in the grid, it does not matter what value x takes as long as $w = 0$ and it does not matter what value z takes as long as $y = 0$: we get a 1 as an output either way. As a result, we can implement this term as

$$\neg w \wedge \neg y$$

to cover all four cells in that group; this is much simpler than the four *separate* corresponding terms in our original implementation. Using the same approach to the other three groups, we find that

$$\begin{array}{rclccccccccccl} r & = & (& \neg w & & & \wedge & \neg y & & &) & \vee \\ & & (& w & \wedge & \neg x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& & & \neg x & \wedge & y & \wedge & \neg z &) & \vee \\ & & (& w & & & \wedge & y & \wedge & z &) & . \end{array}$$

Clearly this is significantly simpler than our initial effort *and* did not require tedious application of logical axioms. However, it is not the most simple description of the function: remember that less groups is better since it means less terms in our expression. As a result, we could consider the following alternative

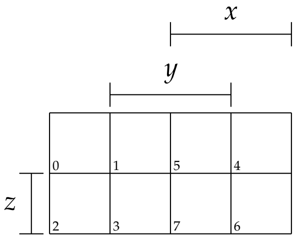


where we now wrap around all four corners, reducing the number of groups from four to three. The end result is a simpler expression, including one less term:

$$\begin{array}{rclcccccccl} r & = & (& \neg w & & \wedge & \neg y & & &) & \vee \\ & & (& & \neg x & & & \wedge & \neg z &) & \vee \\ & & (& w & & \wedge & y & \wedge & z &) & . \end{array}$$

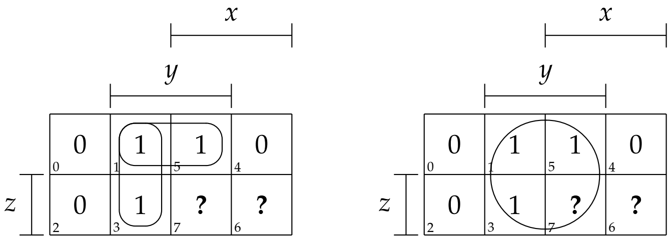
2.5.2 Constructive use of don't care states

Karnaugh maps can represent functions with *any* number of inputs; there is no reason that the grid should be square. Using the 3-input function in Figure 15b as an example, we now have $2^3 = 8$ elements and might select $p = 2$ and $q = 4$ as a result:



Another important feature of this truth table is that some of the outputs are neither 0 nor 1, but rather **?**. We call **?** a (or the) **don't care state** in the sense it can take any value we want: it is not that we do not know the value, we just do not *care* what the value is. If it helps, think of it like a "wildcard". One can rationalise this by thinking of a circuit whose output just does not matter given some combination of input, maybe this input is invalid and so the result is never used.

Either way, we can use these don't care states to our advantage within the context of Karnaugh maps. Specifically, we are free to treat them as 0 *or* 1: treating them as 0 means we do not need to cover them with a group, treating them as 1 means we can potentially form larger groups. Consider the following two groupings:



The left-hand option treats the element associated with $x = 1$ and $z = 1$ (which is a don't care) in the 1-st row, 2-nd column as 0: as such it is not covered by a group, and we are forced to form two rectangular groups as a result and the resulting expression is

$$r = (\neg x \wedge y) \vee (y \wedge \neg z).$$

In contrast, the right-hand option treats the don't care as a 1 meaning it can be included in a single, larger group. This produces the (much) simpler expression $r = y$.

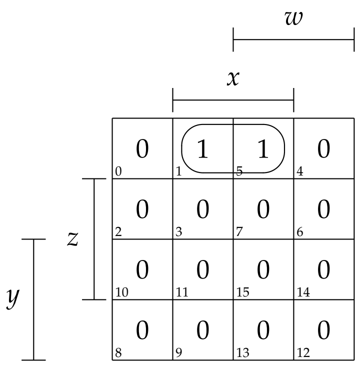
2.5.3 Why Gray code?!

Although use of Gray code is motivated informally above, a reasonable question is what happens if we did *not* use it. Put another way, what exact advantage does *it* produce specifically? The short answer is that it allows us to do steps #3 and #4 of the algorithm, and get the right result wrt. optimisation.

Imagine we have a very simple 4-input function

w	x	y	z	r
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

and write out a Karnaugh map using Gray code to get

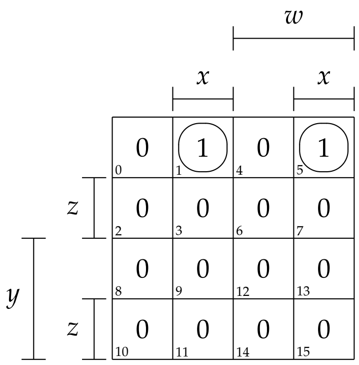


We can form a single group that covers the two elements in the 0-th column; these elements are where $x = 1$ and $w = 0$ and $x = 1$ and $w = 1$, but $y = 0$ and $z = 0$. Following the algorithm, this group produces the expression

$$r = x \wedge \neg y \wedge \neg z$$

since the value of w is irrelevant and so we omit it: as long as $x = 0$, $y = 0$ and $z = 0$, that is enough to specify the group.

Now imagine we try to write out the same Karnaugh map, but *without* using Gray code; we would get something like



indicating, for example, that the 2-nd column now represents cases where $w = 1$ and $x = 0$ and the 3-nd column now represents cases where $w = 1$ and $x = 1$, i.e., the 2-nd and 3-rd columns are swapped versus the original Karnaugh map (and likewise for the rows). The problem is that now we cannot make a single group that covers the same two elements: we instead need to make two groups, each covering one element. These groups produce a more complicated expression as a result, namely

$$\begin{array}{rclcccccccccl} r & = & (& w & \wedge & x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& \neg w & \wedge & x & \wedge & \neg y & \wedge & \neg z &) & \end{array}$$

where we now have a term for w even though we know it is not required; to get the same result as before, we now have to transform the expression by hand using suitable axiomatic steps. In short, using Gray code per the original example saves us work: we *could* get the same result by avoiding them, but we then throw away the advantage of automatically capturing various optimisations

2.6 Mechanical derivation method #3: Quine-McCluskey

Karnaugh maps can represent functions with any number of inputs, but actually drawing them for more than eight or so becomes tricky. That is, although the Karnaugh map *method* for simplification is attractive, it falls down in terms of scalability: when either the number of inputs grows too large, or it needs to be implemented in software it ceases to be as attractive.

In these cases we prefer **Quine-McCluskey minimisation**, a method developed independently by Willard Quine [**?**] and Edward McCluskey [**?**] in the mid 1950s. As an example of applying the method, we revisit the example detailed in Figure 15a.

2.6.1 Step #1: extraction of prime implicants

The first step produces a table, Table 16 for this example, which is extended with content by processing the i-th section to form each $(i + 1)$-th section. The first section of the table is initialised by extracting minterms from the truth table, i.e., all the combinations of input that result in a 1 in the output; this is again analogous to similar steps in methods #1 and #2 (i.e., use of Karnaugh maps) that we investigated previously. This time we use a number to identify each minterm and, since there are four inputs in this case, write them using a tuple

$$(w, x, y, z)$$

in a table which is extended as we progress: the initial content is shown in the top section of Table 16.

Each entry (i.e., row) is called an **implicant**; we separate implicants into groups according to the number of elements in associated tuple representations which equal 1. For example, implicant 0 is represented by $(0, 0, 0, 0)$ meaning that $w = 0$, $x = 0$, $y = 0$ and $z = 0$; the tuple has zero elements equal to 1 and is hence placed in group 0. Likewise, implicant 1 is represented by $(0, 0, 0, 1)$ meaning that $w = 0$, $x = 0$, $y = 0$ and $z = 1$; the tuple representation has one element equal to 1 and is hence placed in group 1 (along with implicants 2, 4 and 8).

Recall from our simplification using Karnaugh maps that we were able to apply a rule to implement both minterms

$$w \wedge \neg x \wedge y \wedge \neg z$$

and

$$w \wedge \neg x \wedge y \wedge z$$

with a single, simpler minterm

$$w \wedge \neg x \wedge y$$

because the value of z has no influence on the result. We use a similar underlying approach here. Starting from the initial content, the i-th section of the table more generally, we produce content for the $(i + 1)$-th section by compare each member of the j-th group with each member of the $(j + 1)$-th group: our goal is to find pairs of implicants whose tuple representations differ in only one element. Comparison between the j-th group and other groups is skipped since they cannot, by definition, satisfy this criterion. For example, to construct the 1-st section from the 0-th section we

- compare implicant 0 from group 0 with implicants 1, 2, 4 and 8 from group 1,
- compare implicants 1, 2, 4 and 8 from group 1 with implicants 5 and 10 from group 2,
- compare implicants 5 and 10 from group 2 with implicant 11 from group 3, and

Section	Group	Implicant		Used
0	0	0	$(0,0,0,0)$	✓
	1	1	$(0,0,0,1)$	✓
		2	$(0,0,1,0)$	✓
		4	$(0,1,0,0)$	✓
		8	$(1,0,0,0)$	✓
	2	5	$(0,1,0,1)$	✓
		10	$(1,0,1,0)$	✓
	3	11	$(1,0,1,1)$	✓
	4	15	$(1,1,1,1)$	✓
1	0	$0+1$	$(0,0,0,-)$	✓
		$0+2$	$(0,0,-,0)$	✓
		$0+4$	$(0,-,0,0)$	✓
		$0+8$	$(-,0,0,0)$	✓
	1	$1+5$	$(0,-,0,1)$	✓
		$4+5$	$(0,1,0,-)$	✓
		$2+10$	$(-,0,1,0)$	✓
		$8+10$	$(1,0,-,0)$	✓
	2	$10+11$	$(1,0,1,-)$	
	3	$11+15$	$(1,-,1,1)$	
2	0	$0+1+4+5$	$(0,-,0,-)$	
		$0+2+8+10$	$(-,0,-,0)$	
		$0+4+1+5$	$(0,-,0,-)$	duplicate
		$0+8+2+10$	$(-,0,-,0)$	duplicate

Figure 16: *Quine-McCluskey simplification, step #1: extraction of prime implicants.*

	0	1	2	4	8	5	10	11	15
$0+1+4+5$	✓	✓		✓		✓			
$0+2+8+10$	✓		✓		✓		✓		
$10+11$							✓	✓	
$11+15$								✓	✓

Figure 17: *Quine-McCluskey simplification, step #2: covering the prime implicants table.*

- compare implicant 11 from group 3 with implicant 15 from group 4.

The result of this process forms a new section of the table; in this case, the middle section of Table 16. In the new section, we replace the differing elements with a – character: combining implicants 0 and 1 represented by the tuples

$$(0, 0, 0, 0)$$

and

$$(0, 0, 0, 1),$$

for example, produces implicant 0 + 1 represented by

$$(0, 0, 0, -).$$

Furthermore, each implicant from the i-th section that is used to form an implicant in the $(i + 1)$-th section is marked with a ✓ next to it; in our example, implicants 0, 1, 2, 4 and 8 are all marked as a result of comparison between groups 0 and 1 and their use in forming implicants 0 + 1, 0 + 2, 0 + 4 and 0 + 8. The process is iterated to produce subsequent sections until we can no longer combine any implicants; in our example we terminate after three phases as shown in Table 16. However, notice that by the third phase we start to introduce *duplicate* implicants; these are deleted from further consideration upon detection.

2.6.2 Step #2: covering the prime implicants table

At the end of the first step, any implicants in any section which remain unmarked are called **prime implicants**: these need to be considered in a second step that produces the required Boolean expression. In our example we have four prime implicants, namely

$$\begin{array}{lcl} 0+1+4+5 & \mapsto & (0,-,0,-) \\ 0+2+8+10 & \mapsto & (-,0,-,0) \\ 10+11 & \mapsto & (1,0,1,-) \\ 11+15 & \mapsto & (1,-,1,1) \end{array}$$

A so-called prime implicant table is constructed which lists the prime implicants along the left-hand side and the original minterms along the top. In each cell where the prime implicant includes the corresponding minterm, we place a ✓ character; Table 17 shows this for our example. The goal now is to select a combination of the prime implicants which covers *all* of the original minterms. For example, the implicant 0 + 1 + 4 + 5 covers the prime implicants 0, 1, 4 and 5 so selecting this as well as implicant 10 + 11 will cover 0, 1, 4, 5, 10 and 11 in total. Before we start however, we can make our task easier by identifying **essential prime implicants**, i.e., those which are the *only* cover for a given minterm. Minterm 15 in our example demonstrates this since it is only covered by prime implicant 11 + 15; we *must* include this implicant in our expression as a result.

The process for coverage is fairly intuitive: starting with the essential prime implicants, we draw a line through the associated row in the prime implicants table; each time the line goes through a ✓, we then also draw a line through that column. The idea is that the resulting lines show which minterms are currently covered by prime implicants we have selected for inclusion in our final expression. For our example we

- draw a line through the row for implicant 11 + 15, and hence through the columns for implicants 11 and 15,
- draw a line through the row for implicant 0 + 1 + 4 + 5, and hence through the columns for implicants 0, 1, 4 and 5, and finally
- draw a line through the row for implicant 0 + 2 + 8 + 10, and hence through the columns for implicants 0, 2, 8 and 10.

The end result shows that by using only implicants 11 + 15, 0 + 1 + 4 + 5 and 0 + 2 + 8 + 10 can cover all the original minterms; we do not need to use implicant 0 + 1 + 4 + 5 for example, since implicants 0, 1, 4 and 5 are all covered elsewhere. Looking at the associated tuples, we have

$$\begin{array}{lcl} 0+1+4+5 & \mapsto & (0,-,0,-) \\ 0+2+8+10 & \mapsto & (-,0,-,0) \\ 11+15 & \mapsto & (1,-,1,1). \end{array}$$

Now, by treating the entries with − as don't care we thus implement the final expression for the function r as

$$\begin{array}{rcccccccccl} r & = & (& \neg w & & \wedge & \neg y & & &) & \vee \\ & & (& & \neg x & \wedge & & & \neg z &) & \vee \\ & & (& w & & \wedge & y & \wedge & z &) & \end{array}$$

as per our original attempt using Karnaugh maps.

2.7 Physical circuit properties

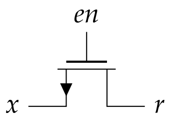
2.8 3-state logic

Roughly speaking, one can freely connect the output of one logic gate to the input of another; this mirrors the fact that we can form arbitrary Boolean expressions from operators. While it does not really make sense to connect two inputs together since *neither* will drive current along the wire, connecting two outputs together is more dangerous since *both* will drive current along the wire. The outcome depends on a number of factors but is seldom good: the transistors which are wired to fight against each other typically melt and stop working.

We can mitigate this issue by extending the idea of 2-state, Boolean logic values into **3-state logic**. There are basically two main ideas:

1. We introduce a new logic value, hence the name 3-state, called **Z** or **high impedance**; the easiest way to think about this value is as representing null, or disconnected.

2. We introduce a new logic gate, a so-called **enable gate**, which is essentially just a switch implemented using a single transistor, i.e.,



The truth table for said gate accommodates the high impedance value as follows:

Enable		
x	en	r
?	0	**Z**
0	1	0
1	1	1
Z	1	**Z**

Considering an enable gate, the idea is that when the enable signal *en* is set to 0 the gate does not pass anything through to the output r: it is not driven with anything, so in a sense some other device can be connected to and drive the same wire. However, when *en* is set to 1, the gate passes through the input x to the output r: nothing else should be driving a value along this wire or we are back to the situation which caused the original melted transistor. In summary, the high impedance value allows any other value to overpower it without causing damage to the surrounding transistors. Thus, careful use of enable gates allows two or more normal logic gate outputs to be connected to the same wire: as long as only one of them is enabled at a time, and hence driving a signal along the wire, there will be no disasters.

2.8.1 Fan-in and fan-out

Fan-in and **fan-out** are properties of logic gates relating to the number of inputs and outputs. The underlying problem is that transistors have physical constraints on the amount of current which should be driven through them: too much and a transistor can be damaged, too little and it will not function correctly (or at least not as expected, e.g., with a slower switching speed). This can be problematic if we treat logic gates as we would operators in Boolean algebra, creating arbitrary combinations.

For example, consider the (contrived) circuit in Figure 18: the source AND gate on the left-hand side is used to drive n other target AND gates to the right. Unless the source gate drives enough current onto its output, errors may occur because the target gates will not receive enough of a share to operate correctly. Each gate implementation will be rated wrt. fan-out, which essentially say how many is too many, i.e., the the number of target gates which can be safely connected to a source gate; CMOS-based gates have quite a high fan-out rating, perhaps 100 target gates or more can be connected to a single source.

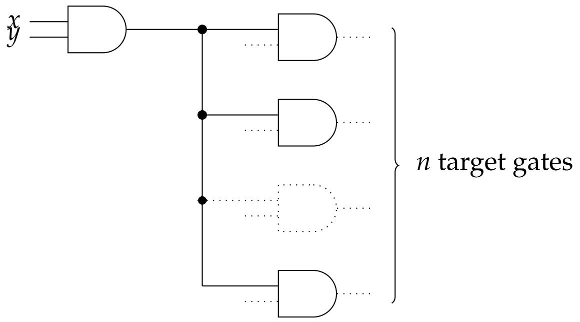
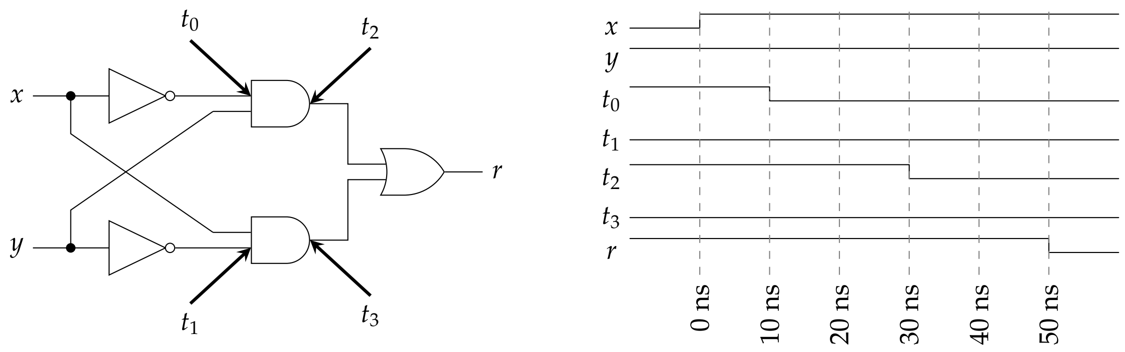


Figure 18: *A contrived circuit illustrating the idea of fan-out, whereby one source gate may need to drive n target gates.*



(a) *An annotated implementation of an XOR gate, using NOT, AND and OR gates.*

(b) *A timeline tracking intermediate results that occur when x is changed from* 0 *to* 1.

Figure 19: *A behavioural timeline demonstrating the effects of propagation delay on an XOR implementation.*

2.8.2 Propagation delay

CMOS-based logic gates do not operate instantaneously because of the way transistors work; there is a **gate delay** between when the inputs are supplied and when the output is produced called. This is compounded by the fact that there is *also* a delay associated with wires used to connect them together. A **wire delay** is typically smaller than gate delay but the same principle applies: a value driven onto one end takes an amount of time (proportional to the wire length) to get to the other. Although such delays are typically very small, when many gates are placed in series or when wires are very long, the delays add up; the problem of managing the resulting **propagation delay** is exacerbated as the complexity of said circuit increases.

Figure 19 tries to illustrate how gate delay impacts on the operation of an XOR gate constructed from NOT, AND and OR gates. If we consider the gate from a *static* point of view, focusing on function alone, then it is reasonable to imagine that by setting $x = 0$ and $y = 1$ the gate computes

$$\begin{array}{lclcl}
x & = & & = & 0 \\
y & = & & = & 1 \\
t_0 & = & \neg x & = & 1 \\
t_1 & = & \neg y & = & 0 \\
t_2 & = & t_0 \wedge y & = & 1 \\
t_3 & = & t_1 \wedge x & = & 0 \\
r & = & t_2 \vee t_3 & = & 1
\end{array}$$

However, this assumes that all the signals eventually propagate through the circuit without really thinking about when and what happens in the time between. Including gate delay gives a *dynamic* view; imagine the delay of

1. a NOT gate is 10 ns,
2. an AND gate is 20 ns, and
3. an OR gate is 20 ns

and we move from having $x = 0$, $y = 1$ to $x = 1$, $y = 1$. The behaviour is illustrated by Figure 19b. We start at time when the circuit is in the correct state given inputs of $x = 0$ and $y = 1$. The inputs are toggled to $x = 1$ and $y = 1$ at 0 ns, but the result does not appear instantly. In particular, we can examine points in the timeline and show that the final and intermediate results are wrong. For example, it takes until 90 ns before the NOT gates produce the correct outputs and the final result does not change to the correct value until 50 ns; before then, the output is wrong in the sense that it does not match the inputs.

The effects of general propagation delay imply a secondary, limiting feature of a given circuit: the **critical path**. This is essentially the the longest sequential sequence of gates (or more generally delays) in the circuit. In the XOR circuit above, the critical path goes through a NOT gate, an AND gate and then an OR gate: the path has a total delay of 50 ns. In essence this simply formalises what we found above, where the XOR gate took 50 ns to produce the correct output r from inputs x and y. The general rule is that the critical path limits the performance of circuits by representing their computational latency; as such, it is advantageous to design or optimise a circuit so as to minimise the critical path length.

As an aside, one can try to equalise the delay through different paths in the circuit using buffer gates: since the buffer takes some time to operate just like any other gate yet performs no change in input, one can place them throughout the circuit in order that values arrive at the same time. In the above, one can imagine adding buffers between y and the second input to the top AND gates for example: this would ensure that $\neg x$ and the buffered version of y arrive at the inputs at the same time. This does not solve the implications of our critical path, and does not cope with (minor) differences due to wire delay, but one could argue that some other issues are tamed.

2.9 Building block components

Before we dive straight into an investigation of large components that do complex (albeit combinatorial) computation, it is useful to first think about a number of standard building blocks: these act as smaller components that we can combine together in order to produce the larger ones. Put another way, they act as a step forward from logic gates but not quite a large enough step to get to where we want!

If the building blocks seem somewhat useless, there is no need to panic: think of them simply as a way to practice the techniques developed so far, and as something we will use in anger within a later Chapter.

2.9.1 Components for choosing between options

The idea of choice is crucial in constructing larger components: often we want to control the component, for example making it operate differently depending on some input. The idea is that

1. a **multiplexer** continuously drives one of many inputs onto a single output depending on a control signal, and
2. a **demultiplexer** continuously drives a single input onto one of many outputs depending on a control signal

where an m-input (resp. m-output), n-bit multiplexer (resp. demultiplexer) has

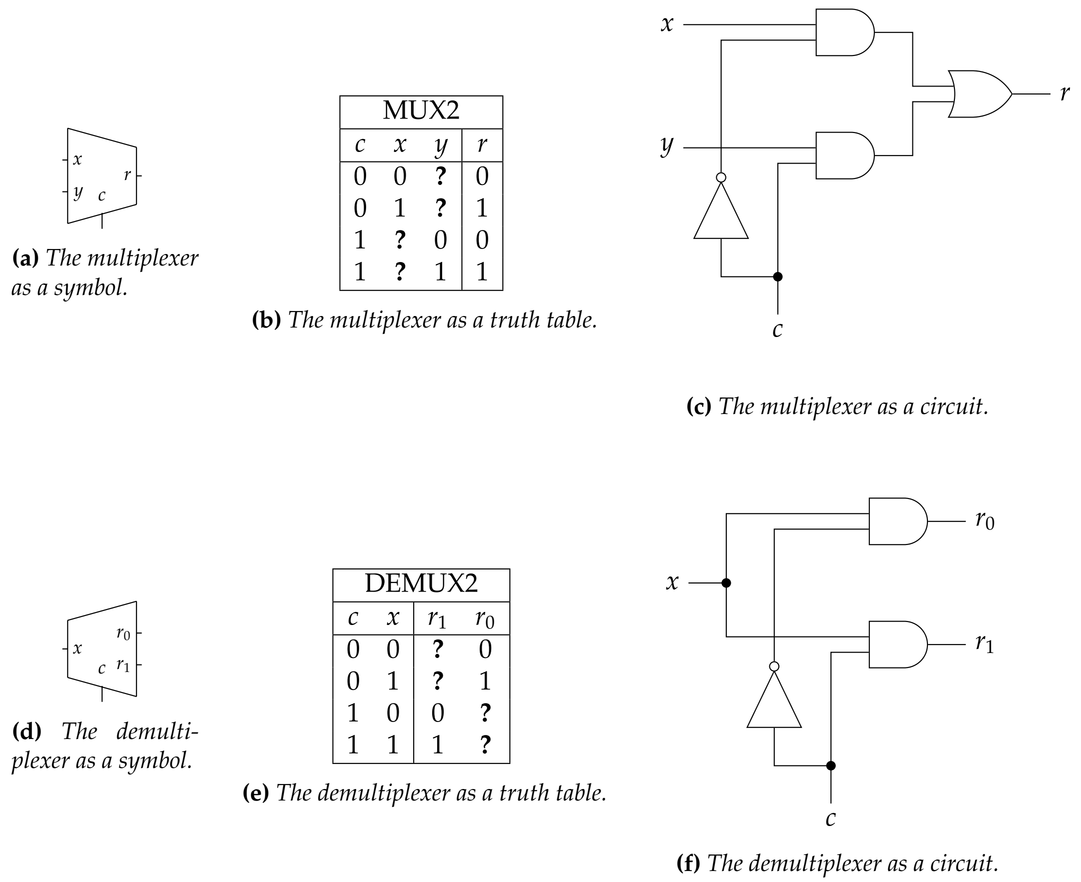
1. m inputs (resp. outputs), each having n bits, and
2. a ($\lceil \log_2(m) \rceil$)-bit control signal that selects between the inputs (resp. outputs).

It is useful to use C as an analogy here. For example, ignoring the number of bits in each input, output and control signal,

```
switch( c ) {
  case 0 : r = w; break;
  case 1 : r = x; break;
  case 2 : r = y; break;
  case 3 : r = z; break;
}
```

acts similarly to a 4-input multiplexer: depending on `c`, one of the inputs (e.g., `w`, `x`, `y`, or `z`) is assigned to the output (i.e., `r`). Likewise,

```
switch( c ) {
  case 0 : r0 = x; break;
  case 1 : r1 = x; break;
  case 2 : r2 = x; break;
  case 3 : r3 = x; break;
}
```



(a) *The multiplexer as a symbol.*

MUX2			
c	x	y	r
0	0	?	0
0	1	?	1
1	?	0	0
1	?	1	1

(b) *The multiplexer as a truth table.*

(c) *The multiplexer as a circuit.*

(d) *The demultiplexer as a symbol.*

DEMUX2			
c	x	r_1	r_0
0	0	?	0
0	1	?	1
1	0	0	?
1	1	1	?

(e) *The demultiplexer as a truth table.*

(f) *The demultiplexer as a circuit.*

Figure 20: *An overview of a 2-input (resp. 2-output), 1-bit multiplexer (resp. demultiplexer) cells.*

acts similarly to a 4-output demultiplexer: depending on `c`, one of the outputs (i.e., `r0`, `r1`, `r2`, or `r3`) is assigned from the input (i.e., `x`). However, we need to take care with this analogy: in C there is a sequence of steps evident, but in our circuit the operation is continuous. That is, the output is continuously being updated based on the inputs: if either the control signal or inputs change, the output will change to match.

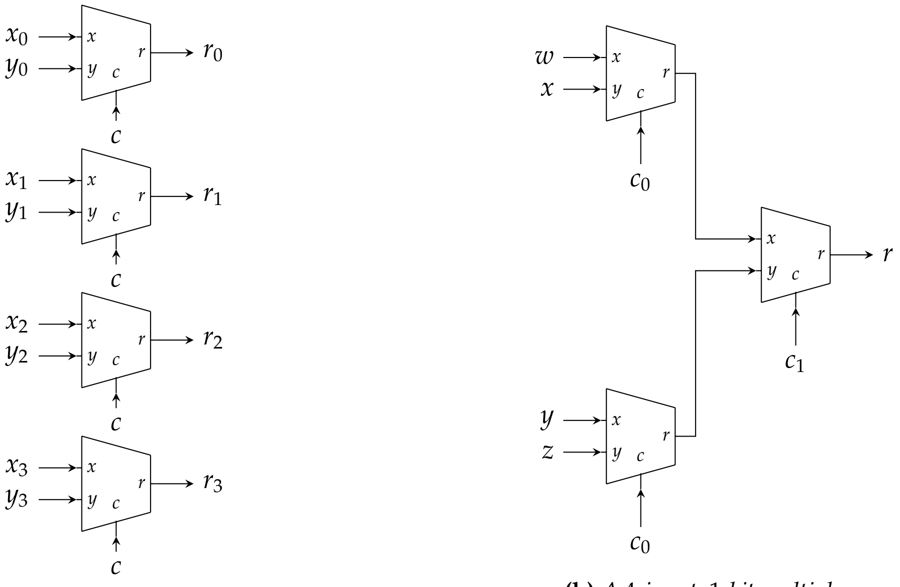
Thinking about the components as circuits is, however, not so hard: we simply need, as before, to write down a truth table to describe the behaviour we want and then to derive a Boolean expression to implement that behaviour. Consider the case of a 2-input (resp. 2-output), 1-bit multiplexer (resp. demultiplexer) outlined in Figure 20.

Taking the multiplexer first, the idea is that we have two 1-bit inputs x and y, and one 1-bit control signal c; we want to drive either x or y through to r depending on whether $c = 0$ or $c = 1$. Look at the truth table in Figure 20b: it looks reasonable in the sense that when $c = 0$ the output matches x and when $c = 1$ the output matches y. It takes the approach of using don't care states to show, for example, that when $c = 0$ it does not matter what the value of y is since the output should match x alone. From this truth table, we can arrive at the expression

$$\begin{array}{ccccccc} r & = & (& \neg c & \wedge & x &) & \vee \\ & & (& c & \wedge & y &) & \end{array}$$

which is shown diagrammatically, as a circuit, in Figure 20c. For the demultiplexer, the idea is that we have two 1-bit outputs r_0 and r_1, and one 1-bit control signal c; we want to drive x through to either r_0 or r_1 depending on whether $c = 0$ or $c = 1$. Clearly this is sort of the opposite of the multiplexer, but we can take a similar approach in the sense that the first step is to write down a truth table per Figure 20e: again the content should make some sense if read row-wise. For example, when $c = 0$ the output r_0 matches x and we don't care what r_1 is, and when $c = 1$ the output r_1 matches x and we don't care what r_0. Deriving two Boolean expressions (for r_0 and r_1) is simple:

$$\begin{array}{rcl} r_0 & = & \neg c \wedge x \\ r_1 & = & c \wedge x \end{array}$$



(a) *A 2-input, 4-bit multiplexer.*

(b) *A 4-input, 1-bit multiplexer.*

Figure 21: *Application of the replication and cascade design patterns.*

A natural question arises given this simple starting point, namely how do we extend these components to produce m-input, n-bit alternatives? Fortunately, we can apply the design patterns presented earlier: consider the multiplexer case as an example, but note that a similar approach applies to the demultiplexer as well.

1. Producing a 2-input, n-bit multiplexer is easy: we replicate n separate 2-input, 1-bit multiplexers where the i-th instance takes the i-th bit of each input and produces the i-th bit of the output. This technique is the same no matter what value of n we select; Figure 21a shows an example for $m = 4$.

2. Producing an m-input, 1-bit multiplexer is a little harder. If we use $m = 4$ as an example, one way to proceed would be to write a larger truth table, i.e.,

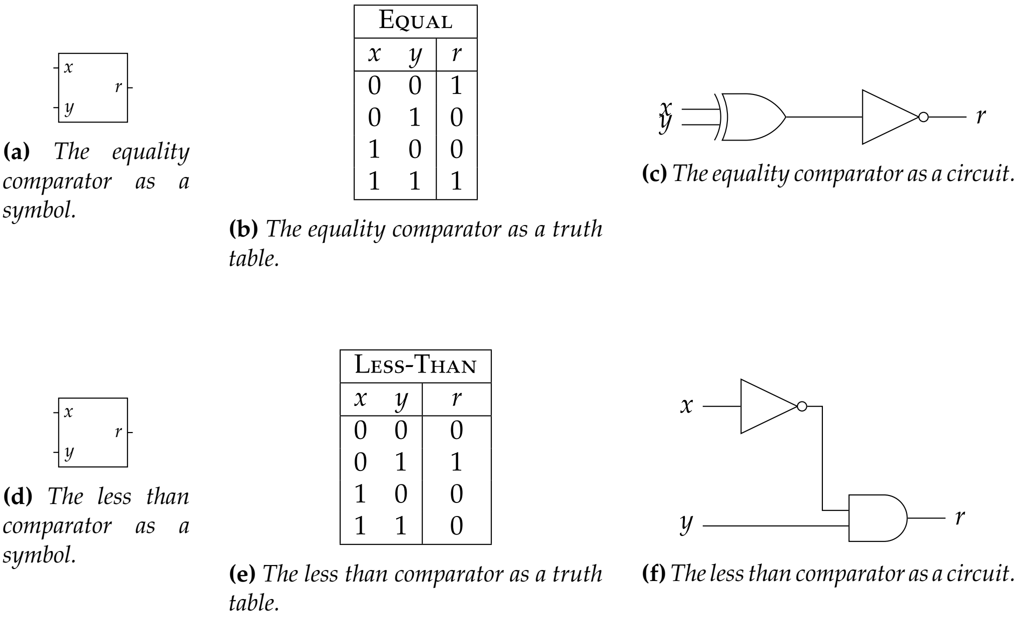
MUX4						
c_1	c_0	w	x	y	z	r
0	0	0	?	?	?	0
0	0	1	?	?	?	1
0	1	?	0	?	?	0
0	1	?	1	?	?	1
1	0	?	?	0	?	0
1	0	?	?	1	?	1
1	1	?	?	?	0	0
1	1	?	?	?	1	1

and then derive a larger Boolean expression

$$\begin{array}{rclcccccccl} r & = & (& \neg c_0 & \wedge & \neg c_1 & \wedge & w &) & \vee \\ & & (& c_0 & \wedge & \neg c_1 & \wedge & x &) & \vee \\ & & (& \neg c_0 & \wedge & c_1 & \wedge & y &) & \vee \\ & & (& c_0 & \wedge & c_1 & \wedge & z &) & . \end{array}$$

Of course this produces a reasonable result, but as the number of inputs grows our task becomes much more difficult. An alternative is to divide-and-conquer: using our existing 2-input, 1-bit multiplexers we can split the large decision task into smaller steps per Figure 21b.

The control signal c_0 is used to control two multiplexers in the first layer: the top one produces w if $c_0 = 0$ or x if $c_0 = 1$, the bottom one produces y if $c_0 = 0$ or z or $c_0 = 1$. These outputs are fed into a



(a) *The equality comparator as a symbol.*

Equal		
x	y	r
0	0	1
0	1	0
1	0	0
1	1	1

(b) *The equality comparator as a truth table.*

(c) *The equality comparator as a circuit.*

(d) *The less than comparator as a symbol.*

Less-Than		
x	y	r
0	0	0
0	1	1
1	0	0
1	1	0

(e) *The less than comparator as a truth table.*

(f) *The less than comparator as a circuit.*

Figure 22: *An overview of equality and less than comparators.*

second layer which uses c_1 to select appropriately: if $c_1 = 0$ the output of the top multiplexer in the first layer is selected, if $c_1 = 1$ the output of the bottom multiplexer in the first layer is selected. The overall result is the same as our dedicated design above, but hopefully it is clear that the cascaded design is simpler (and requires less work as a product of reuse).

2.9.2 Components for doing basic arithmetic

Cast your mind back to Chapter 1 and the topic of representing numbers as n-bit binary sequences. Given one or more such sequences, a natural thing to do is consider arithmetic, i.e., computation represented by some well-known operation such as integer addition. The first step toward meeting this challenge is to consider less general components that perform 1-bit arithmetic. Of course these are not so useful alone, but will act as building blocks when we study their more general alternatives.

Comparators In the same way as arithmetic, comparison of numbers is also a fundamental building block within many circuits. Given 1-input inputs x and y, an **equality comparator** computes

$$r = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

and a **less than comparator** computes

$$r = \begin{cases} 1 & \text{if } x < y \\ 0 & \text{otherwise} \end{cases}$$

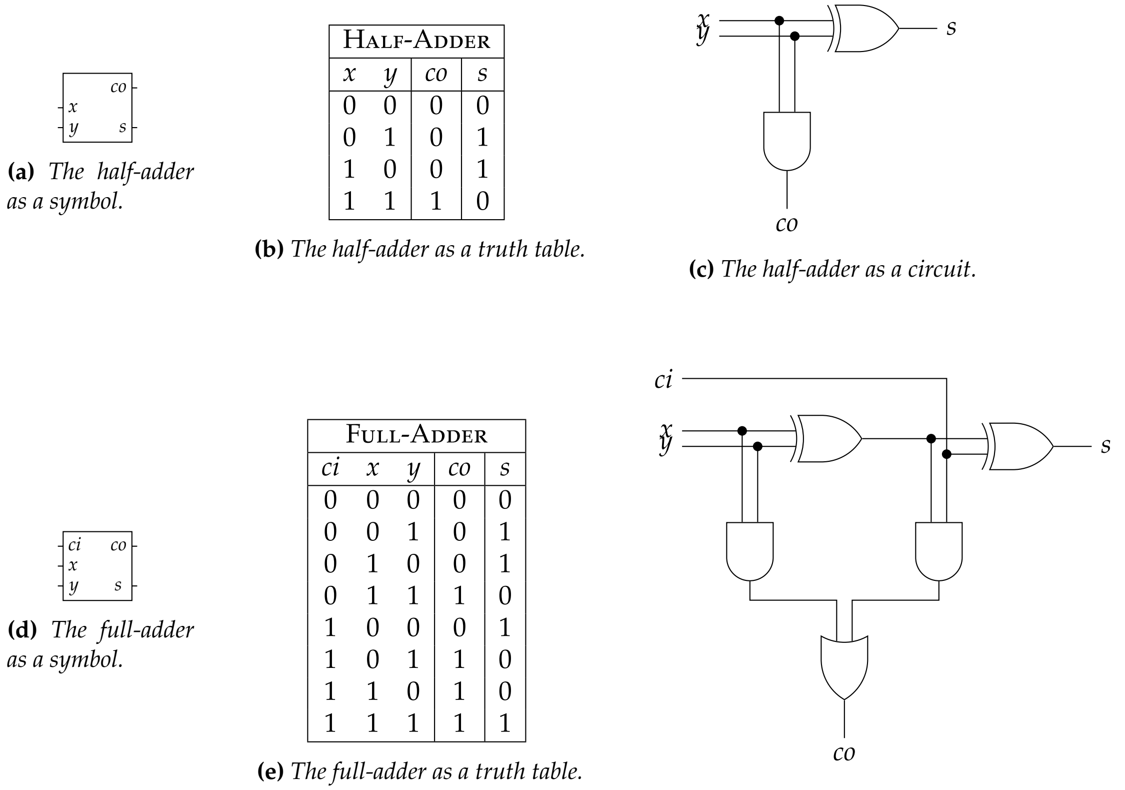
One can consider others, e.g., inequality or greater than, but these will typically be enough on their own.

The truth tables for these operations are shown in Figure 22b and Figure 22e. While fairly self explanatory, they may seem a little odd given we are dealing with 1-bit numbers. Reading them row-wise should give some confidence their content is sane: for example, the truth table for less than says we have that 0 is not less than 0, 0 is less than 1, 1 is not less than 0 and finally 1 is not less than 1. The simple form of each truth table translates into a simple Boolean expression for each operation. For example we have

$$r = \neg(x \oplus y),$$

i.e., NXOR, for the equality comparator and

$$r = \neg x \wedge y$$



(a) *The half-adder as a symbol.*

Half-Adder			
x	y	co	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(b) *The half-adder as a truth table.*

(c) *The half-adder as a circuit.*

(d) *The full-adder as a symbol.*

Full-Adder				
ci	x	y	co	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(e) *The full-adder as a truth table.*

(f) *The full-adder as a circuit.*

Figure 23: *An overview of half- and full-adder cells.*

for the less than comparator. Interestingly, the former implies that a comparator for inequality is simpler still: inverting the expression, we find $r = x \oplus y$ provides an inequality comparison

$$r = \begin{cases} 1 & \text{if } x \neq y \\ 0 & \text{otherwise} \end{cases}$$

This is because, by definition, when $x = y$ (i.e., $x = 0$ and $y = 0$ or $x = 1$ and $y = 1$) $x \oplus y = 0$ and when $x \neq y$ (i.e., $x = 0$ and $y = 0$ or $x = 1$ and $y = 1$) $x \oplus y = 1$.

Half- and full-adders The most simple arithmetic operation, conceptually at least, is addition: we can (hopefully) already do this by hand, so the reasoning is that it should be easier to reason about a circuit that computes results for us. The starting point for such a circuit is a 1-bit adder component which takes two 1-bit values, say x and y, and adds them together to produce a sum and a carry-out, say s and co. This is commonly termed a **half-adder**. The truth table in Figure 23b should make some sense in that it follows what you would do by hand:

- if we add $x = 0$ to $y = 0$, the sum is 0 and there is no carry-out,
- if we add $x = 0$ to $y = 1$, the sum is 1 and there is no carry-out,
- if we add $x = 1$ to $y = 0$, the sum is 1 and there is no carry-out, and finally
- if we add $x = 1$ to $y = 1$, the sum is 2: since we cannot represent 2 using a single bit, we have $s = 0$ and $co = 1$ indicating a carry-out.

Producing Boolean expressions for s and co follows the same technique as previously; we end up with

$$\begin{array}{rcl} co & = & x \wedge y \\ s & = & x \oplus y. \end{array}$$

However, the half-adder is only partly useful: in order to add together numbers larger than 1-bit we need a **full-adder** circuit that can accept a carry-in as well as produce a carry-out. The truth table for a full-adder is shown in Figure 23e. Although it has more rows as a result of the extra input ci, we can again produce Boolean expressions for s and co, e.g.,

$$\begin{array}{rcl} co & = & (x \wedge y) \vee ((x \oplus y) \wedge ci) \\ s & = & x \oplus y \oplus ci \end{array}$$

which are a little more complicated; to simplify the expressions note that there is a shared term $x \oplus y$. Looking at the corresponding circuits in Figure 23c and Figure 25d, it might help to point out that the full-adder is like two half-adders in series: the first takes x and y and produces a partial sum, the second takes the partial sum and ci and produces the final sum.

As an aside, the half-adder represents a simple enough example to explore the idea of gate universality in (a little) more detail. Figure 24 illustrates two alternative implementations: one based on NAND gates only, in Figure 24b, and one on NOR only, in Figure 24c. Both were produced simply by taking the original half-adder, expanding the XOR into NOT, AND and OR gates (per Figure 24a) then translating each gate type using the identities from Section 2.

Once this translation is complete, we can start to simplify the equations by eliminating and sharing logic. For example, it should be clear that in both versions the translation process has produced cases where we take some intermediate t and compute $\neg\neg t$: clearly we can eliminate the NOT gates since the result is still t. In fact, we can rewrite the expressions for the NOR version of the half-adder as follows:

$$\begin{array}{rcl} co & = & \neg x \mathbin{\overline{\vee}} \neg y \\ s & = & \neg((\neg\neg x \mathbin{\overline{\vee}} \neg y) \mathbin{\overline{\vee}} (\neg x \mathbin{\overline{\vee}} \neg\neg y)) \\ & = & \neg((x \mathbin{\overline{\vee}} \neg y) \mathbin{\overline{\vee}} (\neg x \mathbin{\overline{\vee}} y)) \\ & = & (x \mathbin{\overline{\vee}} y) \mathbin{\overline{\vee}} (\neg x \mathbin{\overline{\vee}} \neg y) \end{array}$$

from which we can then share the term $\neg x \mathbin{\overline{\vee}} \neg y$ between co and s. Clearly this reduces the number of NOR gates used from thirteen in the original, naive translation to five in the simplified version. Notice that the more natural description had a similar number. This acts to demonstrate, in a limited way at least and perhaps counter-intuitively, that using NOR (or NAND, having performed a similar simplification) is not too detrimental wrt. number of gates while offering tangible advantages wrt. regularity.

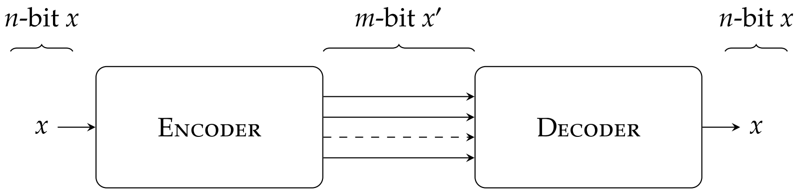
2.9.3 Encoders and decoders

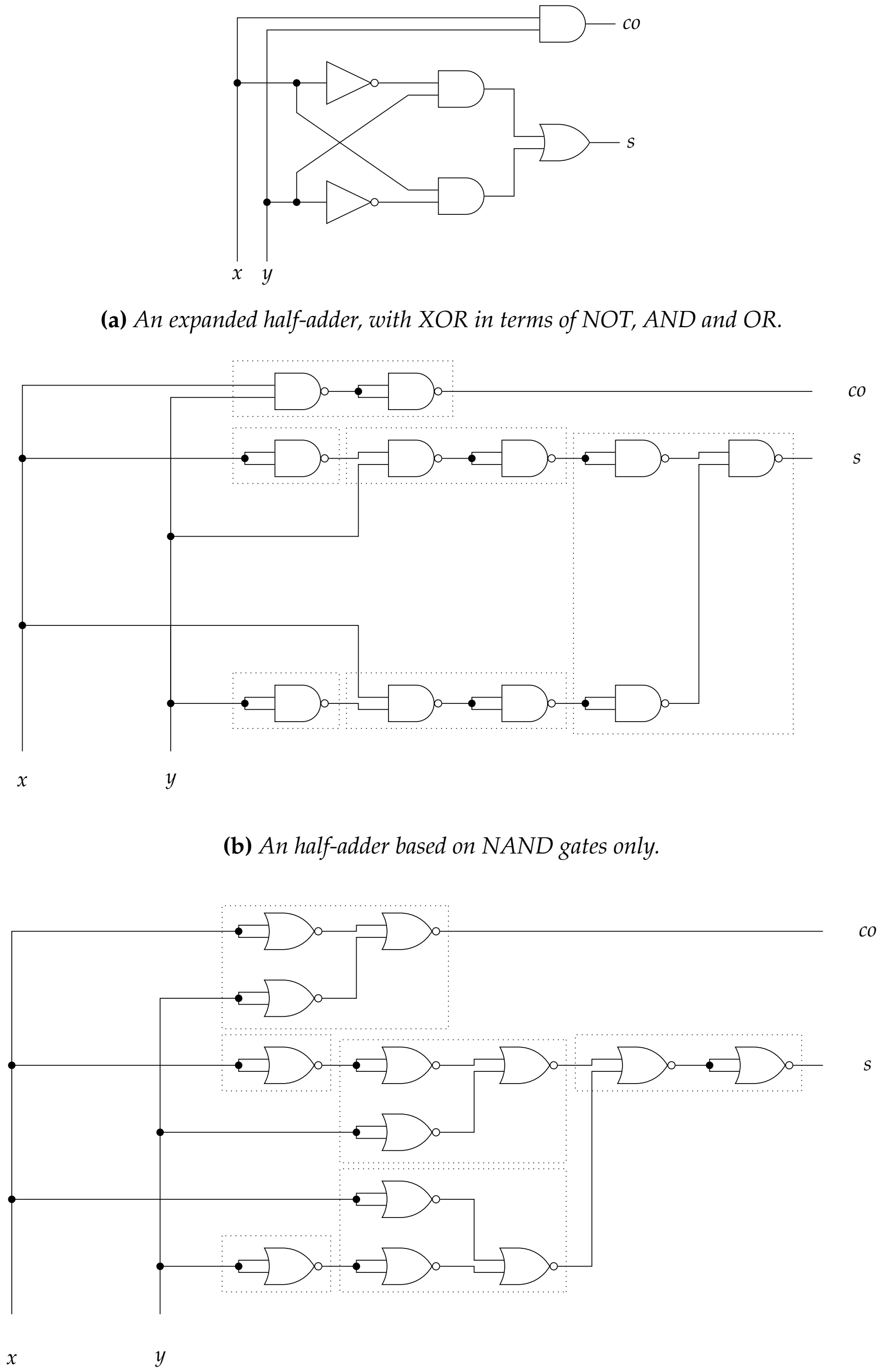
Informally at least, **encoder** and **decoder** components can be viewed as "translators" at either end of some communication medium: the former takes some input and encodes it into a form which is transmitted, the latter does the opposite by decoding the transmission back into the input. More formally, we usually say decoders translate, or map, some n-bit input into one of 2^m possible output values; encoders perform the converse by translating one of 2^m possible input values into an n-bit output. Of course, some inputs or outputs might be ignored in order to relax this restriction so more generally we say an encoder or decoder is an ***n*-to-*m*** device if it has n inputs and m outputs, a 4-to-2 encoder for example. So, we have that

1. an n-to-m encoder translates an n-bit input into some m-bit code word, and
2. an m-to-n decoder translates an m-bit code word back into the same n-bit output.

This terminology can be a bit confusing however, in that a decoder only really make sense for a given encoder; otherwise you might just as well call *both* encoders given the free choice of n and m. For example, there is no strict requirement that $m > n$ or vice versa: it depends largely on the application. Some encoders (resp. decoders) demand that exactly one bit of their output (resp. input) is 1 at a time, these are normally called **one-of-many** devices, but the term is often dropped when the assumption is that all devices are one-of-many.

One can visualise the basic idea as follows:





(a) *An expanded half-adder, with XOR in terms of NOT, AND and OR.*

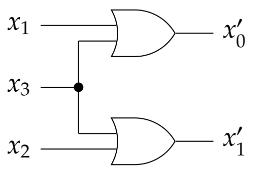
(b) *An half-adder based on NAND gates only.*

(c) *An half-adder based on NOR gates only.*

Figure 24: *Gate universality used to implement a NAND- and NOR-based half-adder. Note that the dashed boxes in the NAND and NOR implementations (middle and bottom) are translations of the primitive gates within the more natural description (top).*

Enc-4-to-2					
x_3	x_2	x_1	x_0	x'_1	x'_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

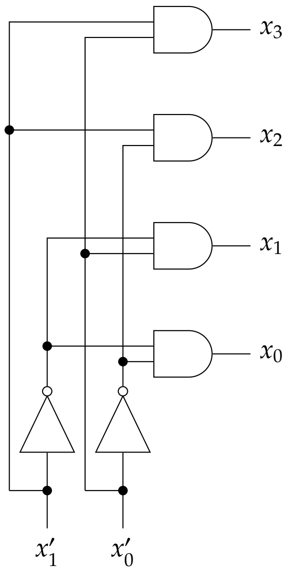
(a) *The encoder as a truth table.*



(b) *The encoder as a circuit.*

Dec-2-to-4					
x'_1	x'_0	x_3	x_2	x_1	x_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

(c) *The decoder as a truth table.*



(d) *The decoder as a circuit.*

Figure 25: *An example encoder/decode pair.*

Here x, an n-bit input, is fed into the encoder which translates it into some m-bit code word x'; the corresponding decoder takes this m-bit code word and translates it back into x. This masks a subtle point however: not *all* encoders have a corresponding decoder (at least not a deterministic one) that will reproduce the same x. So although translation is a good metaphor for the encoder and decoder roles, it need not be 2-way translation.

A simple example Since their design depends on the encoding task at hand, general encoder and decoder building blocks are hard to describe. Instead, consider an example: imagine we are tasked with taking n inputs, say x_i for $0 \leq i < n$, and producing a unsigned integer x' that determines which $x_i = 1$ (given that for all $j \neq i$, $x_j = 0$). In other words, we want a one-of-many encoder that takes x and produces some x' as a result; the opposite task of taking x' and producing each x_k for $0 \leq k < n$ requires a corresponding one-of-many decoder. This problem might be motivated by a need to control other components: if we have n such components in a system, the decoder can be used to enable one of them at a time given some control signal.

If we set $n = 4$ our encoder (resp. decoder) has four inputs (resp. outputs), meaning $x' \in \{0, 1, 2, 3\}$ and hence a 2-bit output (resp. input). Figure 25a and Figure 25c show truth tables for the two components. For the encoder, we produce the Boolean expressions

$$\begin{array}{rcl} x'_0 & = & x_1 \vee x_3 \\ x'_1 & = & x_2 \vee x_3 \end{array}$$

and for the decoder

$$\begin{array}{rcrcr} x_0 & = & \neg x'_0 & \wedge & \neg x'_1 \\ x_1 & = & x'_0 & \wedge & \neg x'_1 \\ x_2 & = & \neg x'_0 & \wedge & x'_1 \\ x_3 & = & x'_0 & \wedge & x'_1 \end{array}$$

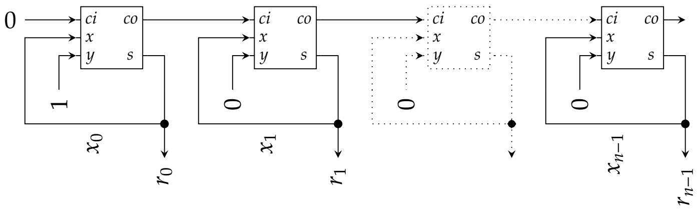


Figure 26: *A correct counter design using sequential logic components.*

Priority encoders It is often useful to extend the concept of encoders and decoders with the idea of **priority**. In a one-of-many encoder, exactly one input bit is allowed to equal 1 but in practise this is hard to ensure. Using the previous example for motivation, imagine we break the rules and set both $x_1 = 1$ *and* $x_2 = 1$: the encoder fails, producing

$$\begin{array}{lclcl} x'_0 & = & x_1 \vee x_3 & = & 1 \\ x'_1 & = & x_2 \vee x_3 & = & 1 \end{array}$$

as the result.

To cope with this possibility in a more graceful manner, we might like to consider a **priority encoder** whereby priority (or preference) to one of the (now several) inputs that equal 1. In the above we might like to give x_2 priority over x_1 for example (even though both equal 1), and guarantee that in this case $x'_0 = 0$ and $x'_1 = 1$ to match. More generally, we might say input x_j has priority over each x_k for $j > k$ (although other schemes are obviously possible).

To capture this scheme, we can rewrite the truth table as follows:

PriorityEnc-4-to-2					
x_3	x_2	x_1	x_0	x'_1	x'_0
0	0	0	1	0	0
0	0	1	**?**	0	1
0	1	**?**	**?**	1	0
1	**?**	**?**	**?**	1	1

Take the 2-nd row for example: although potentially $x_0 = 1$ or $x_1 = 1$ the output gives priority to x_2. That is, as long as $x_2 = 1$ and $x_3 = 0$ (since if $x_3 = 1$, it would have priority) the output will be $x'_0 = 0$ and $x'_1 = 1$ irrespective of x_0 and x_1. The resulting Boolean expressions are updated accordingly:

$$\begin{array}{lclcl} x'_0 & = & (x_1 \wedge \neg x_2 \wedge \neg x_3) & \vee & x_3 \\ x'_1 & = & (x_2 \wedge \neg x_3) & \vee & x_3 \end{array}$$

3 Sequential logic

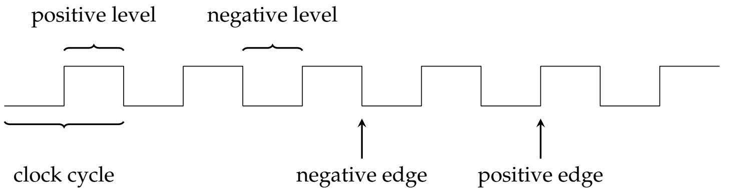
Imagine we need to design an n-bit **counter**, i.e., a component whose n-bit output r steps through values $0, 1, \ldots, 2^n - 1$ and then cycles (i.e., starts from 0 again). Given we already have 1-bit full-adder component, and in Chapter 3 will later extend this into an n-bit adder that can compute $x + y$, an first attempt might be to compute $r \leftarrow r + 1$ over and over again somehow.

Figure 26 captures the result: we essentially try to set one input of the adder $y = 1$ and the other to $x = r$, suggesting the adder will compute $r + 1$. This might *sound* like a reasonable approach in theory, but has (at least) two practical flaws:

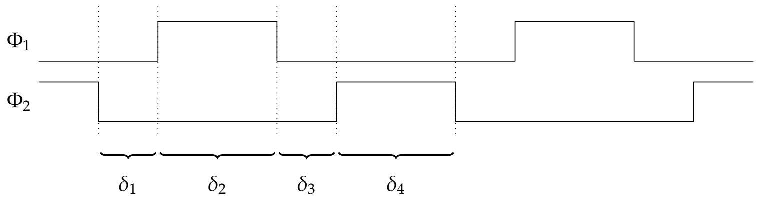
1. we cannot initialise the value, and
2. we do not let the output of each full-adder settle before it is used again as an input: they are computed continuously as a result of the "loop" in our circuit.

So this design will not work correctly: it is intended to illustrate some more general, fundamental limitations of combinatorial logic. Specifically, we cannot control *when* a circuit computes some output (since it does so continuously), nor have it *remember* the output once produced. In short, we need a different approach. This approach, and the components used to support it, are broadly termed **sequential logic**. We need

1. some way to synchronise components in the circuit,



(a) *A* 1*-phase clock.*



(b) *A* 2*-phase clock.*

Figure 27: *An illustration of standard features in* 1*- and* 2*-phase clocks.*

2. one or more components that remember what state they are in, and
3. a mechanism to perform computation as a sequence of steps rather than continuously.

3.1 Clocks

In order to execute a sequence of operations correctly, we need some means of controlling and synchronising them. The concept of a **clock** is key to achieving this. When we say clock within the context of digital logic, we mean a signal that oscillates between 0 and 1 in a regular fashion rather than something that tells the time. This sort of clock is more like a metronome; features in the clock signal keep other components in step with each other.

Clock features A **clock signal** is just the same as any other signal but we assume it approximates a square wave so the value is either 0 or 1. As with other signals, physical characteristics mean this is dubious in practice (the corners of the signal may be "rounded"), but suffices for the discussion here. Beyond this, several features of the signal are worth highlighting; we refer to Figure 27a throughout.

Definition 0.3 *A transitions of the clock signal from* 0 *to* 1 *(resp.* 1 *to* 0*) is called a positive (resp. negative) clock* **edge***; at any point in time where the clock signal is* 1 *(resp.* 0*) we say it is at a positive (resp. negative)* **level***. The region between a positive and negative clock edge is termed a* **clock pulse***.*

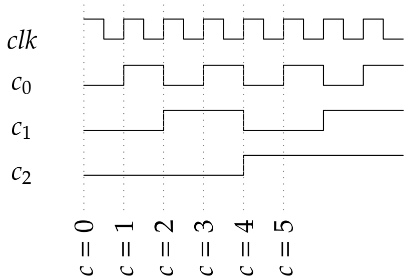
Definition 0.4 *The interval between one positive or negative edge and the next positive or negative edge respectively is termed a* **clock cycle***; the time taken for one clock cycle to happen is the* **clock period** *and the number of clock cycles that occur each second is the* **clock frequency** *(or* **clock rate***).*

Definition 0.5 *The time the clock signal spends at positive and negative levels need not be equal; the term* **duty cycle** *is used to describe the ratio between these times. A clock will typically have a duty cycle of a half, meaning the signal is at a positive level (literally "doing its duty") for the same time it is at a negative level, but clearly other ratios are valid.*

One or more of these and other features can be utilised within a **clocking strategy**, which you can read as "a way to use the clock". For example, we might try to use a clock edge to trigger the start of some computation, or a clock level to enable or disable (e.g., suspend or reset) some computation.

Clock generation and distribution The clock is represented by an ordinary another signal (or signals), *but* it needs to be generated somehow. There are two main choices: either we assume it is an input which needs to be supplied externally, or produce it internally using a **clock generator**. Either way, a common method for generating the required behaviour is to use a piezoelectric crystal which, when a voltage is applied across it, oscillates according to a natural frequency (related to the physical characteristics of the crystal). Roughly speaking, one can use the resulting electrical field generated by this oscillation as the clock signal.

Multiplying a reference clock, or increasing the frequency, is best done using a dedicated component and is somewhat beyond the scope of discussion here. Dividing a reference clock, or decreasing the frequency, is achieved much more easily. Imagine that on each positive edge of our clock *clk*, a counter called *c* is incremented (starting at zero). The individual bits of *c* can be visualised as follows:



Notice that each successive bit of *c* looks like *clk*, but with a period which is twice as long. More formally, the $(i-1)$-th bit of the counter *c* acts like *clk* divided by 2^i. So if we let $i = 1$, we get a clock which is $\frac{1}{2^i} = \frac{1}{2^1} = \frac{1}{2}$ times the speed (i.e., half the speed, or twice as slow) by looking at the 0-th bit of *c*.

Definition 0.6 *Even if we divide or alter the phase of one clock signal to produce others, we say they are from the same family or* **clock domain***: the idea is that if we control components using clocks signals from the same clock domain, we can reason robustly about their behaviour.*

If the behaviour of a component depends on more than one clock domain (i.e., more than one clock generator), we say it has crossed a clock domain. Since the communication of signals from one clock domain to another needs careful control (e.g., to maintain the synchronisation wrt. both clock signals), one typically attempts to minimise where such cases occur.

Once the clock signal has been generated, it needs to be supplied to each component that needs it, much like power rails; we say it is distributed by a **clock network**. Example network topologies include the **H-tree**, which is a form of space filling curve. The advantage of a H-tree is that from the clock generator to *each* target component, the wire delay is the same. This helps to reduce the effect of **clock skew**, a phenomena that occurs if a clock signal arrives at one component along a slightly longer wire than another; it means the arrival time will differ slightly, and the clocks will be unsynchronised. Depending on how the signal is used, this may present a problem because the components are no longer perfectly in step.

From 1-phase to *n*-phase clocks Although it is easiest to think of a single clock signal, as illustrated in Figure 27a, more complicated arrangements are both possible and useful. A central example is the concept of an ***n*-phase clock**, which sees the clock distributed as *n* separate signals along *n* separate wires.

A common instance of this general concept is 2-phase clock: the idea is that the clock is represented by two signals, often labelled Φ_1 and Φ_2. Figure 27b shows how the signals behave relative to each other. Note that features within a 1-phase clock, e.g., the clock period, levels and edges, translate naturally to both Φ_1 and Φ_2. However, notice the additionally guarantee which means their positive levels are non-overlapping: while Φ_1 is at a positive level, Φ_2 is always at a negative level and vice versa. This behaviour is controlled by four parameters

- δ_1 is the period between a negative edge on Φ_2 and a positive edge on Φ_1,
- δ_2 is the period between a positive edge on Φ_1 and a negative edge on Φ_1,
- δ_3 is the period between a negative edge on Φ_1 and a positive edge on Φ_2, and
- δ_4 is the period between a positive edge on Φ_2 and a negative edge on Φ_2.

Adjusting these parameters will shorten or elongate the period of Φ_1 and/or Φ_2, or the "gaps" between them, but the central principle of their being non-overlapping is maintained.

3.2 Latches, flip-flops and registers

Our second requirement is a component which remembers what state it is in, which is to say it stores a value (state and value are used synonymously). Put more formally, it should retain some current state Q (which can also be read as an output), and allow update to some next state Q' (which is provided as an input, meaning we basically store the input value). Such components can be classified as being

1. an **astable**, where the component is not stable in either state and flips uncontrolled between states,
2. a **monostable**, where the component is stable in one states and flips uncontrolled but periodically between states, or
3. a **bistable**, where the component is stable in two states and flips between states under control of an input.

The third class or bistables is often the most useful, and our focus here, since it has the most useful behaviour.

Definition 0.7 *Given a suitable bistable component controlled using an* **enable signal** *en that determines when updates happen, we say it can be*

1. **level-triggered**, *i.e., updated by a given level on en, or*
2. **edge-triggered**, *i.e., updated by a given edge on en.*

The former type is typically termed a **latch**, *with the latter termed a* **flip-flop**. *Latches are sometimes described as* **transparent**: *this term refers to the fact that while enabled, their input and output will match because the state (which matches the output) is being updated with the input. Flip-flops are not the same, because their state is only updated at the exact instant of an edge.*

Definition 0.8 *Whether a positive or negative level (resp. edge) of some signal controls the component depends on whether it is* **active high** *or* **active low**; *a signal en is often written ¬en to denote the latter case.*

Definition 0.9 *It is common for a given latch or flip-flop design to include additional control signals; a central example is a* **reset signal** *rst. We largely ignore such features, reasoning they represent a fairly natural extension to the basic design(s) presented.*

Our description of such components has so far been very abstract; the goal in what follows is to remedy this situation. First, we give a high-level outline of some concrete latch and flip-flip components by specifying the associated behaviour in terms of inputs and outputs. Then, we show how this behaviour can be realised at a lower-level by using logic gates. We focus on the implementation of a D-type latch and flip-flop, working step-by-step toward the goal of a general-purpose n-bit register.

3.2.1 Common latch and flip-flop types

High-level descriptions of behaviour There are four common, concrete instantiations of the somewhat abstract components described above. That is, we usually rely on four common latch and flip-flop types:

1. An **SR-type latch** (resp. **SR-type flip-flop**) component has two inputs S (or set) and R (or reset):
   - when enabled and
     - $S = 0, R = 0$ the component retains Q,
     - $S = 1, R = 0$ the component updates to $Q = 1$,
     - $S = 0, R = 1$ the component updates to $Q = 0$,
     - $S = 1, R = 1$ the component is meta-stable,

     but
   - when not enabled, the component is in storage mode and retains Q.

   The corresponding behaviour is described as follows:

SR-Latch/SR-FlipFlop					
		Current		Next	
S	R	Q	$\neg Q$	Q'	$\neg Q'$
0	0	0	1	0	1
0	0	1	0	1	0
0	1	?	?	0	1
1	0	?	?	1	0
1	1	?	?	?	?

2. A **D-type latch** or "data latch" (resp. **D-type flip-flop**) component has one input D:

   - when enabled and
     - $D = 1$ the component updates to $Q = 1$,
     - $D = 0$ the component updates to $Q = 0$,

     but
   - when not enabled, the component is in storage mode and retains Q.

   The corresponding behaviour is described as follows:

D-Latch/D-FlipFlop				
	Current		*Next*	
D	Q	$\neg Q$	Q'	$\neg Q'$
0	**?**	**?**	0	1
1	**?**	**?**	1	0

3. A **JK-type latch** (resp. **JK-type flip-flop**) component has two inputs J (or set) and K (or reset):

   - when enabled and
     - $J = 0, K = 0$ the component retains Q,
     - $J = 1, K = 0$ the component updates to $Q = 1$,
     - $J = 0, K = 1$ the component updates to $Q = 0$,
     - $J = 1, K = 1$ the component toggles Q,

     but
   - when not enabled, the component is in storage mode and retains Q.

   The corresponding behaviour is described as follows:

JK-Latch/JK-FlipFlop					
		Current		*Next*	
J	K	Q	$\neg Q$	Q'	$\neg Q'$
0	0	0	1	0	1
0	0	1	0	1	0
0	1	**?**	**?**	0	1
1	0	**?**	**?**	1	0
1	1	0	1	1	0
1	1	1	0	0	1

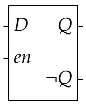
4. A **T-type latch** or "toggle latch" (resp. **T-type flip-flop**) component has one input T:

   - when enabled and
     - $T = 0$ the component retains Q,
     - $T = 1$ the component toggles Q,

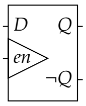
     but
   - when not enabled, the component is in storage mode and retains Q.

   The corresponding behaviour is described as follows:

T-Latch/T-FlipFlop				
	Current		*Next*	
T	Q	$\neg Q$	Q'	$\neg Q'$
0	0	1	0	1
0	1	0	1	0
1	0	1	1	0
1	1	0	0	1



(a) *A level-triggered, D-type latch.*



(b) *A edge-triggered, D-type flip-flop.*

Figure 28: *Symbolic descriptions of D-type latch and flip-flop components (note the triangle annotation around en).*

It is useful to look in more detail at the D-type component, since this will help explain the basic concepts. The component has

- in addition to the enable signal *en*, one input called *D*, and
- two outputs, *Q* and ¬*Q*; we can ignore ¬*Q* usually, but note that it should *always* be the inverse of *Q*.

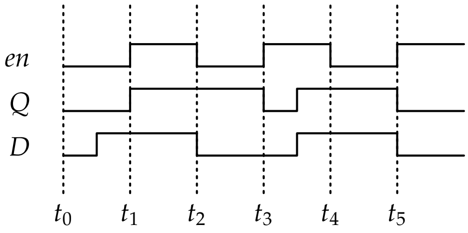
The truth table that describes the behaviour is split into two halves, which is unlike what we have seen previously: the left-hand half is a description of the current state, the right-hand a description of the next state, i.e., *after* we perform an update. Sometimes this is termed an **excitation table** to distinguish it from standard truth tables. So, for example,

1. the first row can be read as "if $D = 0$, then no matter what the current state is then the next state should be $Q = 0$", and
2. the second row can be read as "if $D = 1$, then no matter what the current state is then the next state should be $Q = 1$".

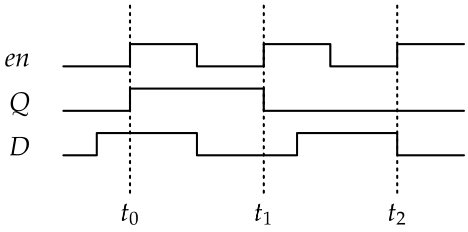
Put another way, this component works as required: we can either update it to store *D* when enabled, or operate it in storage mode to retain *Q* otherwise.

Armed with this knowledge, we can already think about using such components in our designs: we expand on their internal design in the following Sections, but can already use more abstract symbols shown in Figure 28 to differentiate between the latch and flip-flop versions. Similar symbols describe components other than the D-type one we have focused on; they typically retain the the triangle annotation (or absence thereof) on *en* to highlight the difference.

Low(er)-level descriptions of behaviour Still focusing on the D-type component, lower-level use can be illustrated using a timing diagram, which shows the behaviour of the enable signal *en* (which we assume is active high), the input *D* and the output *Q*. For a D-type latch we have something like the following:



The vertical dashed lines highlight important points in time; between t_1 and t_2, and t_3 and t_4 for instance, *en* is at a positive level so the latch state is updated to match *D*. Otherwise, for example between t_0 and t_1, *en* is at a negative level so changes to *D* do not effect the latch state: the latch is in storage mode, meaning it retains the current state. Swapping to a D-type flip-flop, the behaviour changes:



Now the flip-flop state will be updated to match D only at the points in time where en transitions from 0 to 1; this happens at t_0, t_1 and t_2, meaning interim changes to D have no effect on the flip-flop state.

Definition 0.10 *Using a component of this type is more difficult in practice than alluded to by these examples. Although we largely ignore them from here on, the following are important:*

1. *The* **setup time** *(resp.* **hold time***) is the minimum period of time that D must be stable before (resp. after) use to update the component.*

   Think of the clock feature (either level or edge) as triggering the act of sampling from D in order to update the state. As such, the two timing restrictions mentioned make sure the sample is reliable: if D has to be stable for some period of time, it cannot change mid-update for instance.

2. *The* **clock-to-output time** *is an artefact of propagation delay: a delay will exist between the update event being triggered by the associated clock feature, and the output Q changing to match.*

These time periods or delays will be determined by the implementation of the component; ideally they will be minimised, which makes the component easier to use (i.e., more tolerant).

3.2.2 Implementation step #1: a basic SR latch

The first step is somewhat counter-intuitive. We start by looking at the Set-Reset or SR latch: the circuit shown in Figure 29a has two inputs called S and R which are the set and reset signals, and two outputs Q and $\neg Q$. Internally, the arrangement will probably seem odd in comparison to other designs we have seen so far: the outputs of each NOR gate are wired to the input of the other, an arrangement we say is **cross-coupled**.

The result of this odd design is that the outputs are not uniquely defined by the inputs. For example, when we set the inputs to $S = 0$ and $R = 0$, *either* of Figure 30a and Figure 30b describe valid states (in the sense that the outputs make sense). When we set the inputs to $S = 1$ and $R = 0$, we *force* the latch into a state shown in Figure 30c; Q is set to 1 and $\neg Q$ to 0, whatever they were before, because the top NOR gate must output 0. In contrast, when we set the inputs to $S = 0$ and $R = 1$, we force the latch into a state shown in Figure 30d; Q is set to 0 and $\neg Q$ to 1, again because the bottom NOR gate must output 0. The final issue is what happens if we set the inputs to $S = 1$ and $R = 1$. In this case we end up in a state shown in Figure 30e, which is contradictory (hence the outputs are shown as unknown): both outputs should become 0, yet we know they must also be the inverse of each other. The term **meta-stable** is often used to describe the result, in that the latch behaviour is neither predictable nor stable in the same sense it is for other input combinations.

In addition, the Q and $\neg Q$ outputs from the component swap over as well. In short, the NAND-based version still achieves the same goal, but we need to carefully translate the behaviour when using it within a larger design. It is often termed an $\overline{\text{SR}}$ latch rather than SR latch to highlight this fact, which we adopt to avoid confusion about which type of component is meant.

This high-level description avoids two perfectly reasonable questions, namely

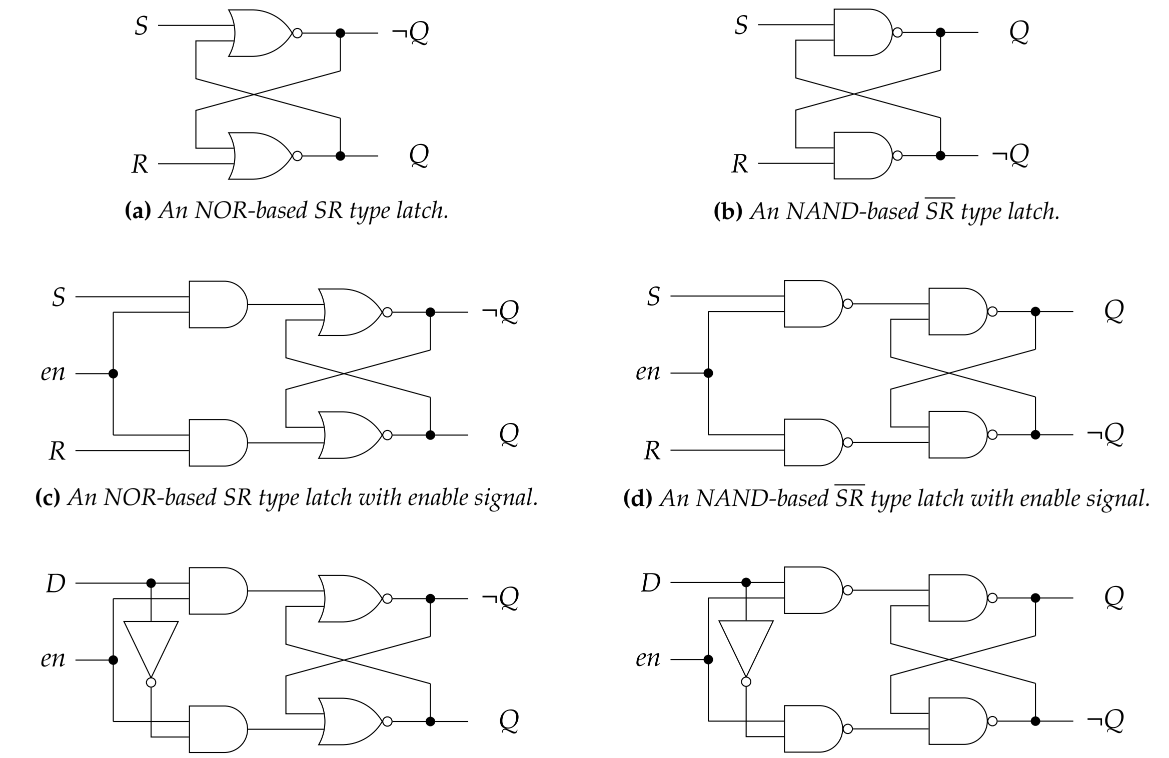
1. how does the latch settle into *any* state, particularly given the case where $S = R = 0$ seems to imply there are two options, and
2. how does it stay in one of those states when $S = R = 0$.

To answer both questions, we rely on Figure 31. The idea is it decomposes the SR latch design into eight individual transistors (labelled t_0 through to t_7) which implement the two NOR gates; this annotation is important because it allows a clearer explanation of their behaviour.

Question #1: how does the latch settle into a state? You can use a similar reasoning for all four cases, but focus on $S = 0$ and $R = 1$ which mean

- t_0 is a P-MOSFET, so is connected since $S = 0$,
- t_2 is an N-MOSFET, so is disconnected since $S = 0$,
- t_4 is a P-MOSFET, so is disconnected since $R = 1$, and
- t_6 is an N-MOSFET, so is connected since $R = 1$.

This means $r_1 = 0$ because t_6 is connected and t_4 is disconnected. Now we can see that



(a) *An NOR-based SR type latch.*

(b) *An NAND-based $\overline{SR}$ type latch.*

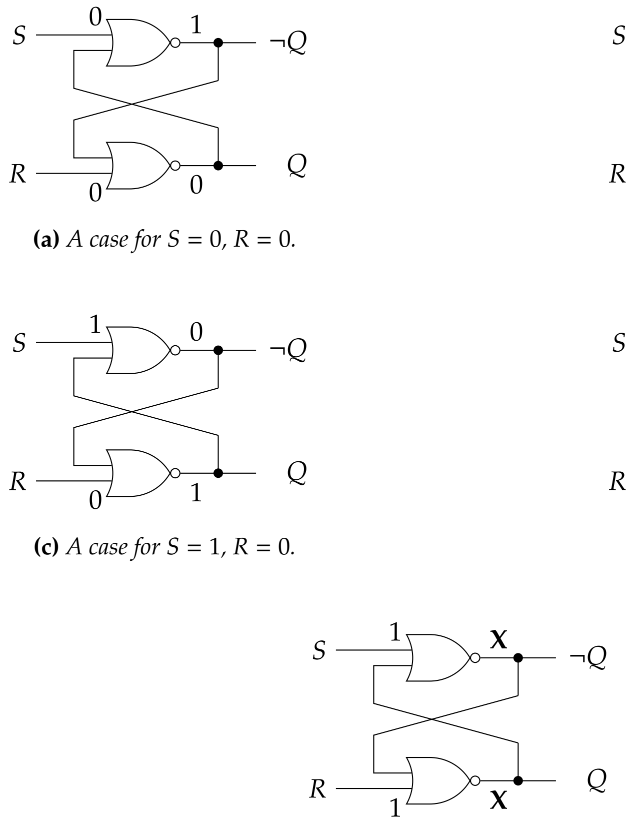
(c) *An NOR-based SR type latch with enable signal.*

(d) *An NAND-based $\overline{SR}$ type latch with enable signal.*

(e) *An NOR-based SR type latch with enable signal and $R = \neg S$.*

(f) *An NAND-based $\overline{SR}$ type latch with enable signal and $R = \neg S$.*

Figure 29: *A collection of NOR- and NAND-based SR type latches, with simpler (top) to more complicated (middle and bottom) control features.*



(a) *A case for $S = 0$, $R = 0$.*

(b) *A case for $S = 0$, $R = 0$.*

(c) *A case for $S = 1$, $R = 0$.*

(d) *A case for $S = 0$, $R = 1$.*

(e) *A case for $S = 1$, $R = 1$.*

Figure 30: *A case-by-case overview of NOR-based SR latch behaviour; notice that there are* two *sane cases for $S = 0$ and $R = 0$, and no sane cases for when $S = 1$ and $R = 1$.*

An aside: NAND- rather than NOR-based latches.

As an aside, we can construct (more or less) the same component using NAND rather than NOR gates; the NAND-based versions are shown alongside each of the associate NOR-based Figures. This change implies a subtle difference in behaviour however. Essentially, the storage and meta-stable states are swapped over: when enabled and

- $S = 1$, $R = 1$ (rather than $S = 0$ and $R = 0$) the component retains Q, and
- $S = 0$, $R = 0$ (rather than $S = 1$ and $R = 1$) the component is meta-stable.

- t_1 is a P-MOSFET, so is connected since $r_1 = 0$,
- t_3 is an N-MOSFET, so is disconnected and since $r_1 = 0$.

This means $r_0 = 1$ because t_0 and t_1 are connected, while t_2 and t_3 are disconnected. Finally, we can check for consistency, noting

- t_5 is a P-MOSFET, so is disconnected since $r_0 = 1$, and
- t_7 is an N-MOSFET, so is connected since $r_0 = 1$.

This means $r_1 = 0$ because t_6 and t_7 are connected, while t_4 and t_5 are disconnected: we knew that anyway. So, in short, the circuit settles into a stable state even though it might seem the "loop" would prevent it doing so, *and* is valid in the sense that r_0 and r_1 (i.e., Q and $\neg Q$) are each others inverse as expected.

Question #2: how does the latch remain in a state? Now imagine we flip to $S = R = 0$, meaning we would like to retain the state fixed above, i.e., keep $Q = 0$ until we want to update it again. Two transistors change as a result of R changing

- t_4 is a P-MOSFET, so is now connected since $R = 0$,
- t_6 is an N-MOSFET, so is now disconnected since $R = 0$.

However, everything else stays the same, i.e.,

- t_5 is a P-MOSFET, so is still disconnected since $r_0 = 1$, meaning that t_4 being connected does not connect r_1 to V_{dd}, and
- t_7 is an N-MOSFET, so is still connected since $r_0 = 1$, meaning that t_6 being disconnected does not disconnect r_1 from GND.

That is, there is no motivation (or physical stimulus) for the transistors to flip into into the other stable state (i.e., where $S = R = 0$ and $Q = 1$) and so the current state is therefore retained.

3.2.3 Implementation step #2: controlling latch updates

The initial SR latch design is arguable *too* simple, in the sense it is hard to use. We have little or no control over when an update happens for instance, because any change to S or R might provoke this; it is also unattractive that we can produce unpredictable behaviour by (perhaps unintentionally) driving it with inputs that would cause meta-stability. Fortunately, both of these problems can be solved with only simple alterations to the original design:

1. To control when an update happens, we **gate** S and R. The general term gate means to "conditionally turn off". The idea is to alter the design so we can decide when S and R do or do not act as input to the latch itself. This is achieved by adding an extra input en and two AND gates, so the original latch inputs become

$$\begin{array}{rcl} S' & = & S \wedge en \\ R' & = & R \wedge en \end{array}$$

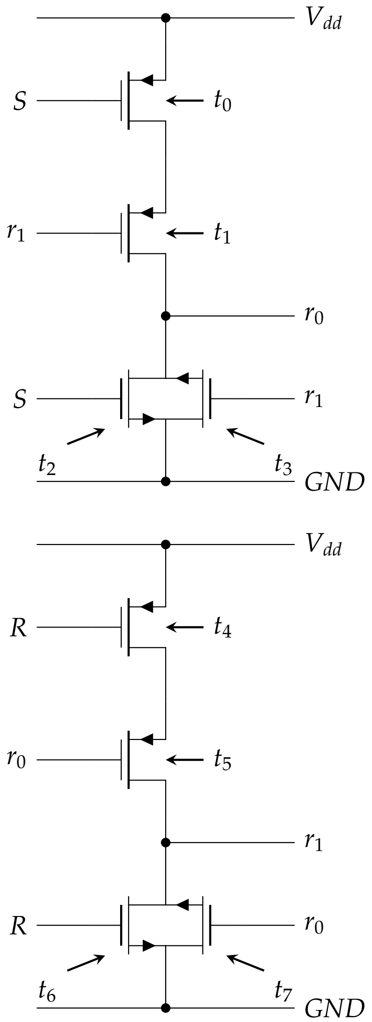
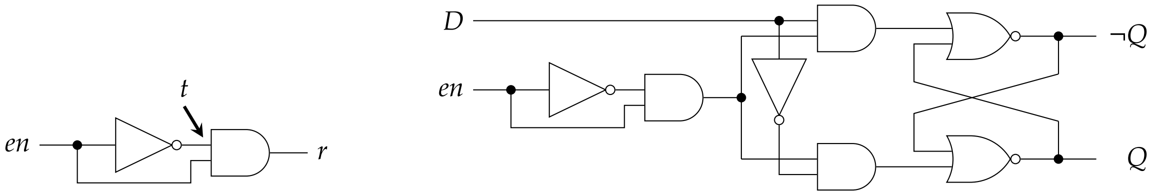


Figure 31: *An annotated SR latch, decomposed into two NOR gates and then into transistors; r_0, the output of the top NOR gate, is used as input by the bottom NOR gate and r_1, the output from the bottom NOR gate, is used as input by the top NOR gate (although the physical connections are not drawn).*



(a) *A pulse generator.*

(b) *The existing NOR-based D-type latch, but now driven using the pulse generator to form a flip-flop.*

Figure 32: *A NOR-based D-type flip-flop created by using a pulse generator.*

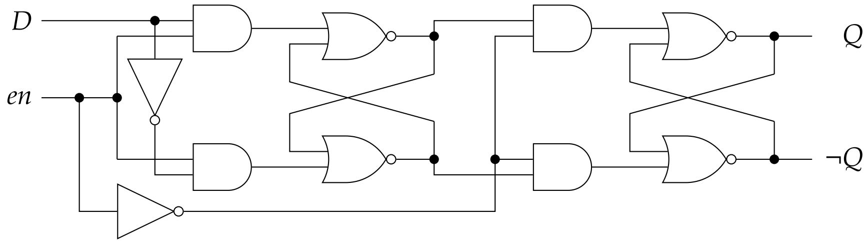


Figure 33: *A NOR-based D-type flip-flop created using a master-slave arrangement of latches.*

When $en = 0$, S and R are irrelevant: S' and R' will always be 0 because, for example, $S' = S \wedge 0 = 0$. This means when $en = 0$ the latch can never be updated. When $en = 1$ however, S and R are passed through into the latch as input because $S' = S \wedge 1 = S$.

Put another way, the result shown in Figure 29c is now clearly level-triggered because S and R only matter during a positive level of en. Note that although en can be considered a generic enable signal, we can use a clock signal to provoke regular, synchronised updates.

2. To avoid the situation where $S = R = 1$, we simply force $R = \neg S$ by inserting a NOT gate between them to disallow the case where $S = R$; Figure 29e shows the result, where the single input is now labelled D. By following the above, the latch inputs become

$$\begin{array}{rcl} S' & = & D \wedge en \\ R' & = & \neg D \wedge en \end{array}$$

This might *seem* to imply that we cannot put the latch into storage mode any longer. However, remember that when $en = 0$ we *always* have $S' = R' = 0$ irrespective of D, so en basically decides if we retain Q (if $en = 0$) or update it with D (if $en = 1$).

The result now represents the D-type latch component discussed originally. Reiterating, when enabled and

- $D = 1$ the component updates to $Q = 1$,
- $D = 0$ the component updates to $Q = 0$,

but when not enabled, the component is in storage mode and retains Q.

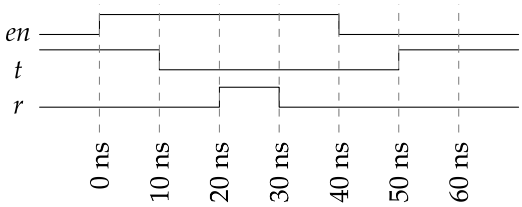
3.2.4 Implementation step #3: from latch to flip-flop

Our next problem is that although the level triggered D-type latch gives *some* control, it is not very fine-grained. Put simply, although we restrict updates to a positive (resp. negative, for active low components) level where $en = 1$ (resp. $en = 0$), this is potentially a lengthy period of time; the input D may potentially change several times during this period for instance. To give more precise control over updates, we might try to convert the latch into a flip-flop: this means restricting updates to the precise, and hence much smaller, instant in time that an edge on en occurs.

There are various ways to realise this alteration; flip-flop design as a topic is broad enough that it starts to go outside our scope wrt. level of detail. In the following Sections we therefore cover two approaches, both at a somewhat high level.

Using a pulse generator A simple approach is to construct a circuit that generates a pulse (or glitch), i.e., an output that is 1 for a short period of time, when *en* transitions from 0 to 1. The pulse then *approximates* an edge, even though we are still actually using a level; the approach can be rationalised by saying that as long as the period is small, it will give us fin*er* grained control than the original latch.

Figure 32a shows the (annotated) pulse generator circuit. The intuition is that the inputs to the AND gate are forced to arrive at different times because of an imbalance in delay: it takes longer for *en* to propagate through the NOT gate than along the wire. Imagine we set the delay associated with NOT and AND gates to 10 ns and 20 ns, then flip $en = 0$ to $en = 1$ and back again:



The result is a pulse, matching the delay of a NOT gate. Put another way, for a short period of time both $en = 1$ *and* $t = 1$ due to the delays involved: this means for a short period of time, the AND gate actually outputs 1. As long as the delay of a NOT gate is short the pulse will be short, and therefore approximate an edge.

Figure 32b illustrates the original D-type latch, now driven by the pulse generator in order to approximate an edge-triggered flip-flop.

Using a master-slave arrangement An alternative approach is to adopt a **master-slave** arrangement of *two* latches in order to get the edge-triggered behaviour we want. Figure 33 shows the result, which is basically one latch (the master, on the left) in series with a second latch (the slave, on the right). The idea is to split a clock cycle into to half-cycles such that

1. while $en = 1$, i.e., during the first half-cycle, the master latch is enabled,
2. while $en = 0$, i.e., during the second half-cycle, the slave latch is enabled.

In practical terms, this means while $en = 1$, i.e., during a positive level on *en*, the master latches the input. Then, the instant $en = 0$, i.e., a negative edge on *en*, the slave latches the output from the master: you can think of it as triggering a transfers from master to slave, or as the slave only being sensitive to updates in the master rather than the inputs. The fact that the transfer is instantaneous, occurring as the result of an edge (in this case a negative edge, when *en* flips from 1 to 0) means we get what we want, i.e., an edge-triggered flip-flop.

3.2.5 Implementation step #4: an *n*-bit register

The D-type component we have, either the latch or flip-flop version, holds a 1-bit state; to store a larger, *n*-bit state we simply group together *n* such components into a **register**. This just means replicating the relevant component type, and synchronising updates to them all using the same enable signal.

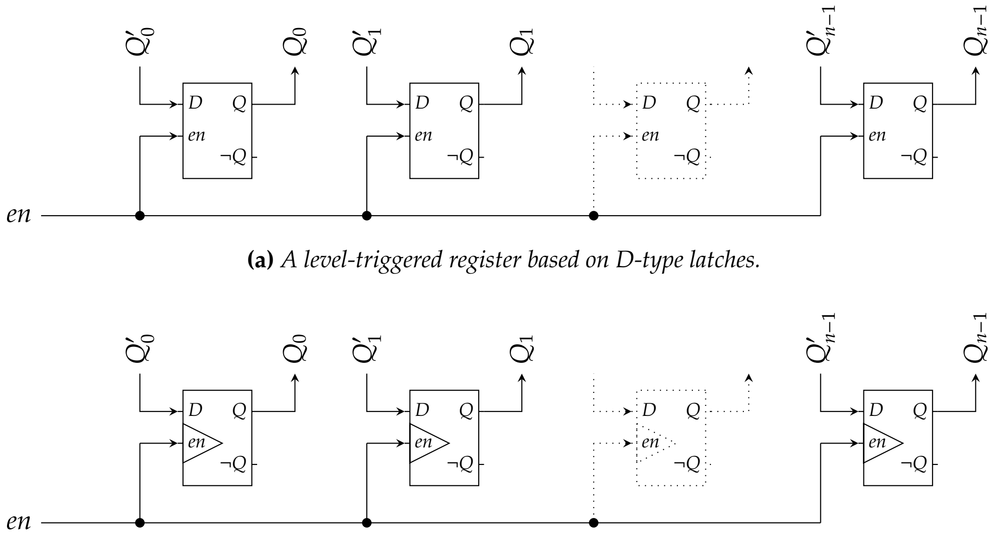
Figure 34 shows the general structure. We can read the current value of the register from the Q outputs: Q_i is the current state of the i-th bit held by the register. We can latch (or store) a new value Q' into the register by driving each D_i with Q'_i then waiting for an update to be triggered (which depending on the component type, means waiting for an appropriate level or edge on *en*).

3.3 Putting everything together: general clocking strategies

3.3.1 A robust *n*-bit counter design

So now think back to our original problem outlined at the beginning of the Section: given everything accumulated so far, how do we solve it? We can use one or other of two designs; both attempt to "break" the loop evident in the original design by inserting storage components, and therefore differ as a result of opting for either flip-flops or latches.

Figure 35a represents a solution based on use of flip-flops, which implies a 1-phase clocking strategy. The top-half of the Figure shows an *n*-bit ripple-carry adder; the idea is that it computes $r' = r + 1$. This part is roughly the same as the initial, faulty solution. The bottom-half of the Figure shows an *n*-bit, edge-triggered



(a) *A level-triggered register based on D-type latches.*

(b) *An edge-triggered register based on D-type flip-flops.*

Figure 34: *An n-bit register, with n replicated* 1*-bit components synchronised using the same enable signal.*

register; the idea is that it stores the current value of r. Beyond this, two features of the design are vitally important:

1. Notice that each full-adder input x_i is produced by AND'ing $\neg rst$ (which is active high) and r_i. This acts as a (limited form of) reset mechanism: if $rst = 1$ then $\neg rst = 0$ and each $x_i = 0$ because $r_i \wedge \neg rst = r_i \wedge 0 = 0$, so the current value is *forced* to be zero (which is important, because when powered-on it will be undefined and hence unusable).

2. Notice that each D-type flip-flop in the register is synchronised by clk (which we assume is a clock): positive edges on clk provoke them to update the stored value r with $r' = r + 1$.

   The original loop is broken, because the update is instantaneous not continuous: there is a "gap" between computing and storing values, in the sense that the adder has an entire clock cycle to compute the result $r + 1$ given r is stored in the flip-flops. Provided that that the propagation delay associated with the adder is less than the clock period (i.e., we do not update r faster than r' is computed) the problem is solved and r cycles through the required values in discrete steps controlled by the clock.

Figure 35a represents a solution based on use of latches, which implies a 2-phase clocking strategy. A reasonable question to ask is why we cannot just replace the flip-flops with latches? Imagine we did this: since the latches are level-triggered, they will be updated when $clk = 1$. So one one hand we have broken the original loop, but on the other hand the loop is still there when $clk = 1$ because the latches are essentially transparent.

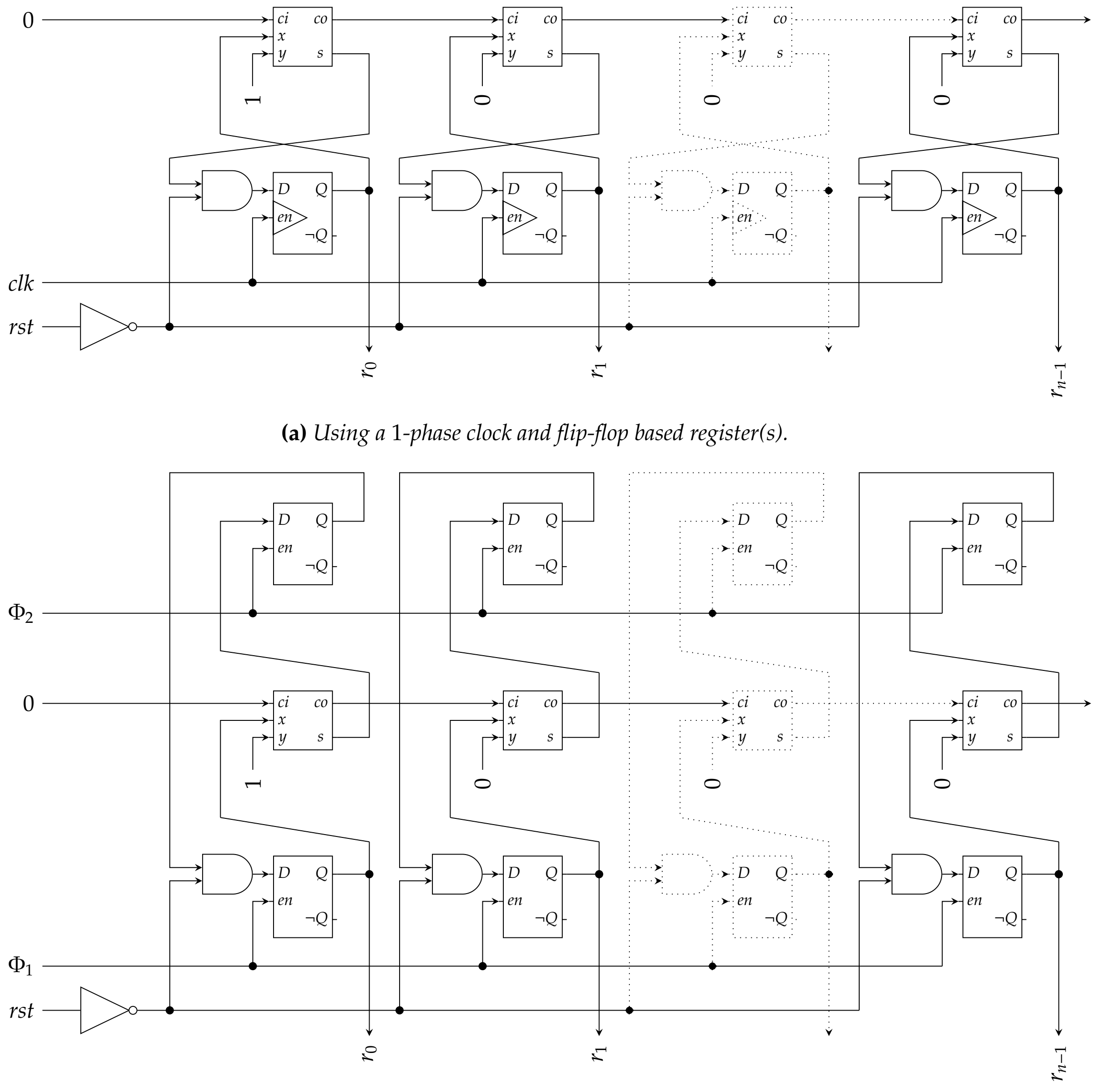
To resolve this the design uses two sets of latches, one to store the adder input and one to store the adder output. Only one set is enabled at a time, because we use a 2-phase clock to control them; when $\Phi_2 = 1$ the output latches store the adder output, then when $\Phi_1 = 1$ the input latches store whatever the output latches stored and subsequently provide a new input to the adder. Clearly we need more storage components to support this approach, but you can think of this as a trade-off wrt. reduced complexity of latches versus flip-flops. Put another way, the design might be less efficient in terms of area but is much easier to reason about.

3.3.2 Generalising the two design strategies

Figure 37 generalises the two counter solutions in the previous Section; you can think of both as general frameworks, or architectures, that can be filled-in with concrete details to realise the solution to a specific problem. These can be generalised a little further by noting the following:

Definition 0.11 *A typical circuit based on sequential logic will be comprised of*

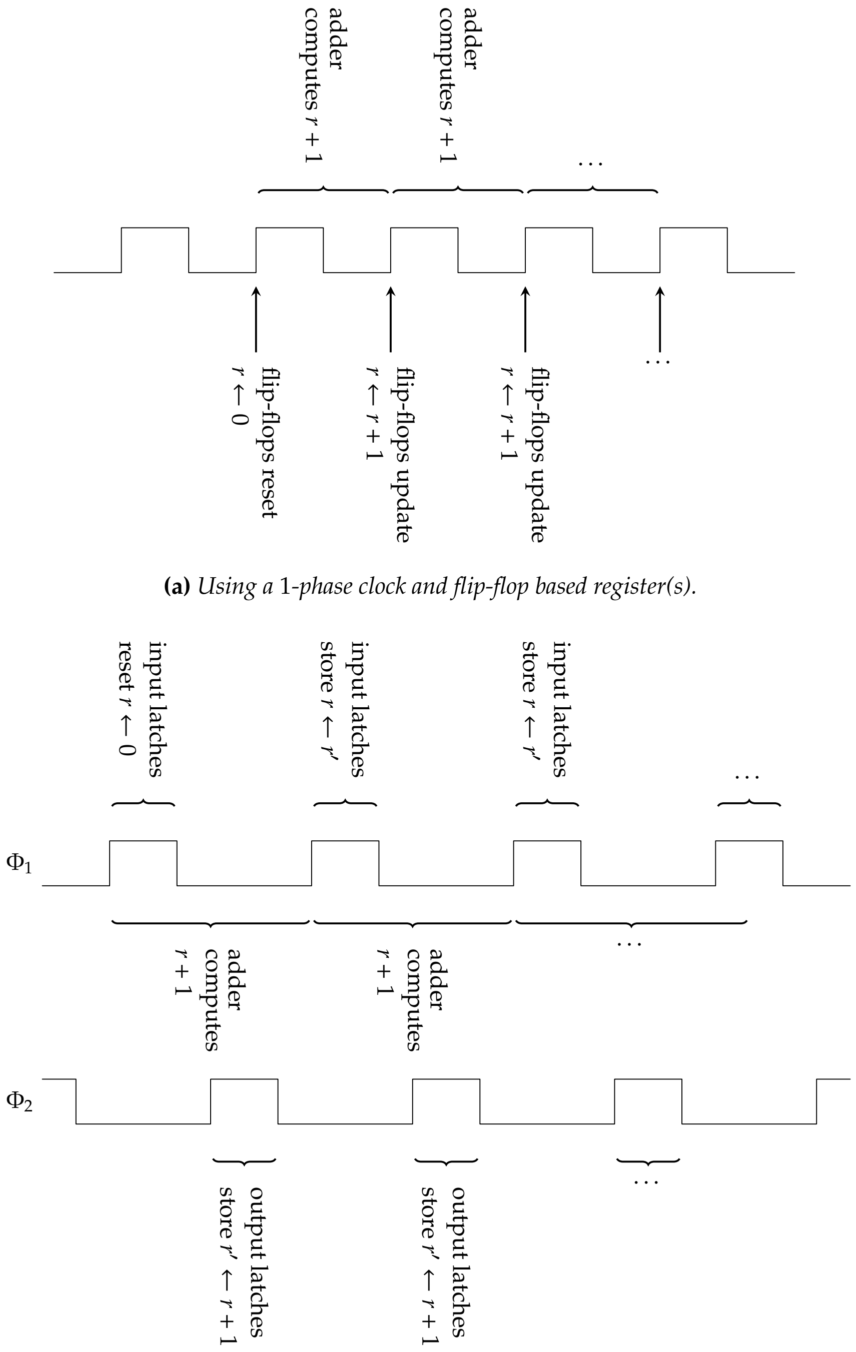
1. *a* **data-path**, *of computational and/or storage components, and*



(a) *Using a* 1*-phase clock and flip-flop based register(s).*

(b) *Using a* 2*-phase clock and latch based register(s).*

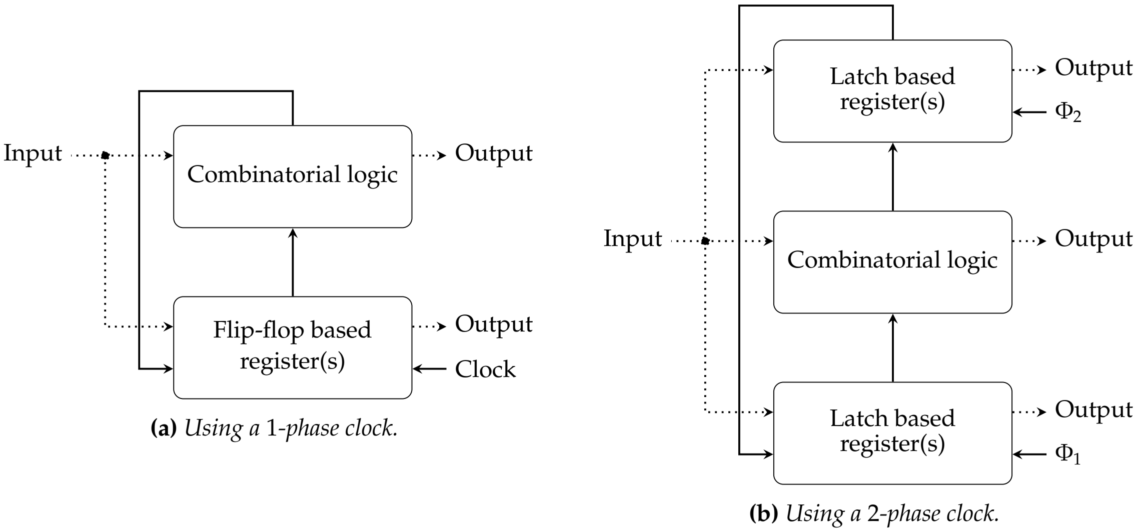
Figure 35: *A correct counter design using sequential logic components.*


adder computes $r+1$
adder computes $r+1$
...
flip-flops reset $r \leftarrow 0$
flip-flops update $r \leftarrow r+1$
flip-flops update $r \leftarrow r+1$
...
input latches reset $r \leftarrow 0$
input latches store $r \leftarrow r'$
input latches store $r \leftarrow r'$
...
Φ_1
adder computes $r+1$
adder computes $r+1$
...
Φ_2
output latches store $r' \leftarrow r+1$
output latches store $r' \leftarrow r+1$
...

(a) *Using a* 1*-phase clock and flip-flop based register(s).*

(b) *Using a* 2*-phase clock and latch based register(s).*

Figure 36: *Two illustrative timelines, outlining stages of computation within the associated counter design.*



(a) *Using a 1-phase clock.*

(b) *Using a 2-phase clock.*

Figure 37: *Two different high-level clocking strategies.*

2. *a* **control-path***, that tells components in the data-path what to do and when to do it.*

For example, within the two counter solutions we clearly have computational (i.e., the adder) and storage components (i.e., the register), and also mechanisms to control them (i.e., the reset AND gates).

3.4 State machines: from simple to more complex control-paths

The topic of automaton, specifically **Finite State Machines (FSMs)**, has a very formal basis; basically they are models of computation, not too far from topics such as Turing Machines (TMs). Put another way, you can think of an FSM as a computer, albeit a simple one.

The control-path in Figure 35 is *very* simple: this is partly an artefact of the problem at hand of course, but masks the difficulty of dealing with more complicated problems. FSMs represent an attractive solution however, allowing us to reason about and implement more complicated, general-purpose control-paths.

3.4.1 A rough overview of FSM-related theory

Definition 0.12 *An FSM is a (theoretical) machine that can be in a finite set of* **states***. The machine consumes input* **symbols** *from an* **alphabet** *(which defines which symbols are valid and so on) one at a time; symbols make the machine* **transition** *from one state to another according to a* **transition function***. When the input is exhausted, the machine halts; depending on the state it halts in, the machine is said to* **accept** *or* **reject** *the input. The set of inputs accepted by the machine is termed the language accepted; this can be used to classify the machine itself.*

Definition 0.13 *Based on the fact that*

1. **entry actions** *happen when entering a given state,*
2. **exit actions** *happen when exiting a given state,*
3. **input actions** *happen based on the state and any input received, and*
4. **transition actions** *happen when a given transition between states is performed*

we can categorise an FSM based on output behaviour:

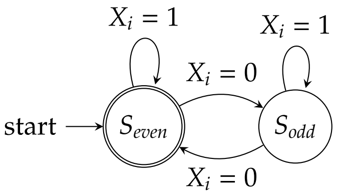
1. *a* **Moore-style** *FSM only uses entry actions, i.e., the output depends on the state only, while*
2. *a* **Mealy-style** *FSM only uses input actions, i.e., the output depends on the state and the input.*

An alternative classification relates to transition behaviour, where an FSM is deemed

1. **deterministic** *if for each state there is always one transition for each possible input (i.e., we always know what the next state should be), or*

	δ	
Q	Q'	
	$X_i = 0$	$X_i = 1$
S_{even}	S_{odd}	S_{even}
S_{odd}	S_{even}	S_{odd}

(a) *A tabular description.*



(b) *A diagrammatic description.*

Figure 38: *An example FSM to decide whether there is an odd or even number of* 0 *elements in some sequence X.*

2. **non-deterministic** *if for each state there might be zero, one or more transitions for each possible input (i.e., we only know what the next state could be).*

Definition 0.14 *A given FSM can be defined via the following:*

1. *S, a finite set of states and a distinguished start state $s \in S$.*
2. *$A \subseteq S$, a finite set of accepting states.*
3. *An input alphabet Σ and output alphabet Γ.*
4. *A transition function*
$$\delta : S \times \Sigma \rightarrow S.$$
5. *An output function*
$$\omega : S \rightarrow \Gamma$$
in the case of a Moore FSM, or
$$\omega : S \times \Sigma \rightarrow \Gamma$$
in the case of a Mealy FSM.

Note that:

- *The FSM itself might be enough to solve a given problem, but it is common to control an associated data-path using the outputs.*
- *A special "empty" (or null) input denoted ϵ allows a transition which can* always *occur.*
- *It is common to allow δ to be a partial function, i.e., a function which is not defined for all inputs.*
- *If the FSM is non-deterministic, then δ might instead give a set of possibilities that is sampled from.*

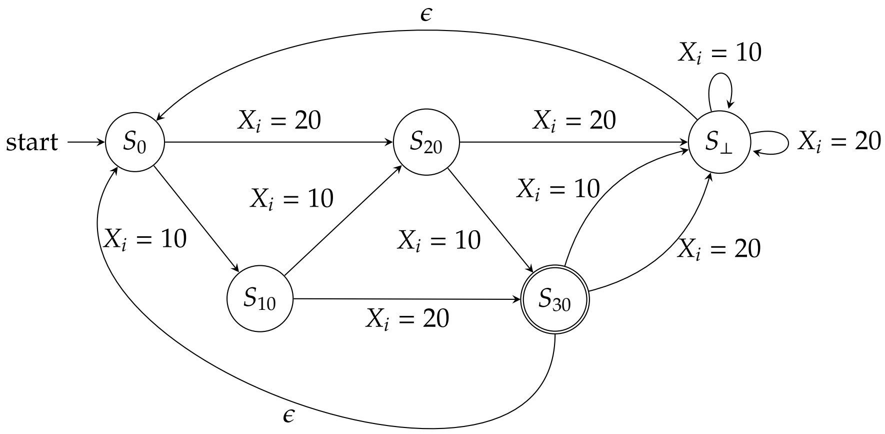
More simply, you can think of an FSM as a directed graph where moving between nodes (which represent each state) means consuming the input on the corresponding edge. Some examples should show that the fairly formal description above translates into a much more manageable reality.

Example #1: even or odd number of 0 **elements** Imagine we are tasked with designing an FSM that decides if a binary sequence X has an even or odd number 0 elements in it. The input alphabet in this case is
$$\Sigma = \{0, 1\}$$
since each X_i can either be 0 or 1. The FSM can clearly be in two states: having consumed the input so far, it can either have seen an even or odd number of 0 elements. Therefore we can say
$$S = \{S_{even}, S_{odd}\},$$
have $s = S_{even}$ as the starting state, and $A = \{S_{even}\}$ as the (singleton) set of accepting states. There is no output as such, so the and output alphabet Γ and output function ω are irrelevant.

Our final task is to define the transition function. Figure 38 includes a tabular and a diagrammatic description of the same thing. The tabular, truth table style description is easier to discuss. The idea is that it lists the current state (left-hand side), alongside the next state for each possible input (right-hand side). In words, the rows read as follows:

Q	δ	
	Q'	
	$X_i = 10$	$X_i = 20$
S_0	S_{10}	S_{20}
S_{10}	S_{20}	S_{30}
S_{20}	S_{30}	$S_{\perp}$
S_{30}	$S_{\perp}$	$S_{\perp}$

(a) *A tabular description.*



(b) *A diagrammatic description.*

Figure 39: *An example FSM modelling a simple vending machine.*

- if we are in state S_{even} and the input $X_i = 0$ then we move to state S_{odd},
- if we are in state S_{even} and the input $X_i = 1$ then we stay in state S_{even},
- if we are in state S_{odd} and the input $X_i = 0$ then we move to state S_{even}, and
- if we are in state S_{odd} and the input $X_i = 1$ then we stay in state S_{odd}.

The intuition is, for example and with a similar argument possible for the state S_{odd}, that if we have seen an even number of 0 elements, i.e., are in state S_{even}, and the next input is 0 then we have now seen an odd number hence move to state S_{odd}. Conversely, if we have seen an even number of 0 elements, i.e., are in state S_{even}, and the next input is 1 then we have *still* seen an even number hence stay in state S_{even}.

Consider some examples of the FSM in operation

1. For the input $X = \langle 1, 0, 1, 1 \rangle$ the transitions are

$$\rightsquigarrow S_{even} \overset{X_0=1}{\rightsquigarrow} S_{even} \overset{X_1=0}{\rightsquigarrow} S_{odd} \overset{X_2=1}{\rightsquigarrow} S_{odd} \overset{X_3=1}{\rightsquigarrow} S_{odd}$$

   meaning we start in state S_{even} then

   (a) stay in S_{even} since $X_0 = 1$,
   (b) move to S_{odd} since $X_1 = 0$,
   (c) stay in S_{odd} since $X_2 = 1$, and finally
   (d) stay in S_{odd} since $X_3 = 1$.

   Since we finish in state S_{odd}, the input is rejected and hence we conclude it has an odd number of 0 elements.

2. For the input $X = \langle 1, 0, 1, 0 \rangle$ the transitions are

$$\rightsquigarrow S_{even} \overset{X_0=1}{\rightsquigarrow} S_{even} \overset{X_1=0}{\rightsquigarrow} S_{odd} \overset{X_2=1}{\rightsquigarrow} S_{odd} \overset{X_3=0}{\rightsquigarrow} S_{even}$$

   meaning we start in state S_{even} then

   (a) stay in S_{even} since $X_0 = 1$,
   (b) move to S_{odd} since $X_1 = 0$,

(c) stay in S_{odd} since $X_2 = 1$, and finally

(d) move to S_{even} since $X_3 = 0$.

Since we finish in state S_{even}, the input is accepted and hence we conclude it has an even number of 0 elements.

3.4.2 Example #2: a vending machine

Imagine we are tasked with designing an FSM that controls a vending machine. The machine accepts tokens worth 10 or 20 units: when the total value of tokens entered reaches 30 units it delivers a chocolate bar but it does not give change. That is, the exact amount must be entered otherwise an error occurs, all tokens are ejected and we start afresh.

The design is clearly a little more complex this time. The input alphabet is basically just the tokens that the machine can accept, so we have

$$\Sigma = \{10, 20\}.$$

The set of states the machine can be in is easy to enumerate: it can either have accepted tokens totalling 0, 10, 20 or 30 units in it or be in the error state which we denote by $\perp$. Thus, we can say

$$S = \{S_\perp, S_0, S_{10}, S_{20}, S_{30}\}$$

and clearly set $s = S_0$ since initially the machine has accepted no tokens. There is one accepting state, which is when a total of 30 tokens has been accepted, so $A = \{S_{30}\}$. Since there is again no output, our final task is again to define the transition function. As before, Figure 39 outlines a tabular and diagrammatic description.

1. For the input $X = \langle 10, 20 \rangle$ the transitions are

$$\rightsquigarrow S_0 \overset{X_0=10}{\rightsquigarrow} S_{10} \overset{X_1=20}{\rightsquigarrow} S_{30}$$

meaning we start in state S_0 then

(a) move to S_{10} since $X_0 = 10$, and finally

(b) move to S_{30} since $X_3 = 30$.

Since we finish in state S_{30}, the input is accepted and we get a chocolate bar as output!

2. For the input $X = \langle 20, 20 \rangle$ the transitions are

$$\rightsquigarrow S_0 \overset{X_0=20}{\rightsquigarrow} S_{20} \overset{X_1=20}{\rightsquigarrow} S_\perp$$

meaning we start in state S_0 then

(a) move to S_{20} since $X_0 = 20$, and finally

(b) move to $S_\perp$ since $X_3 = 20$.

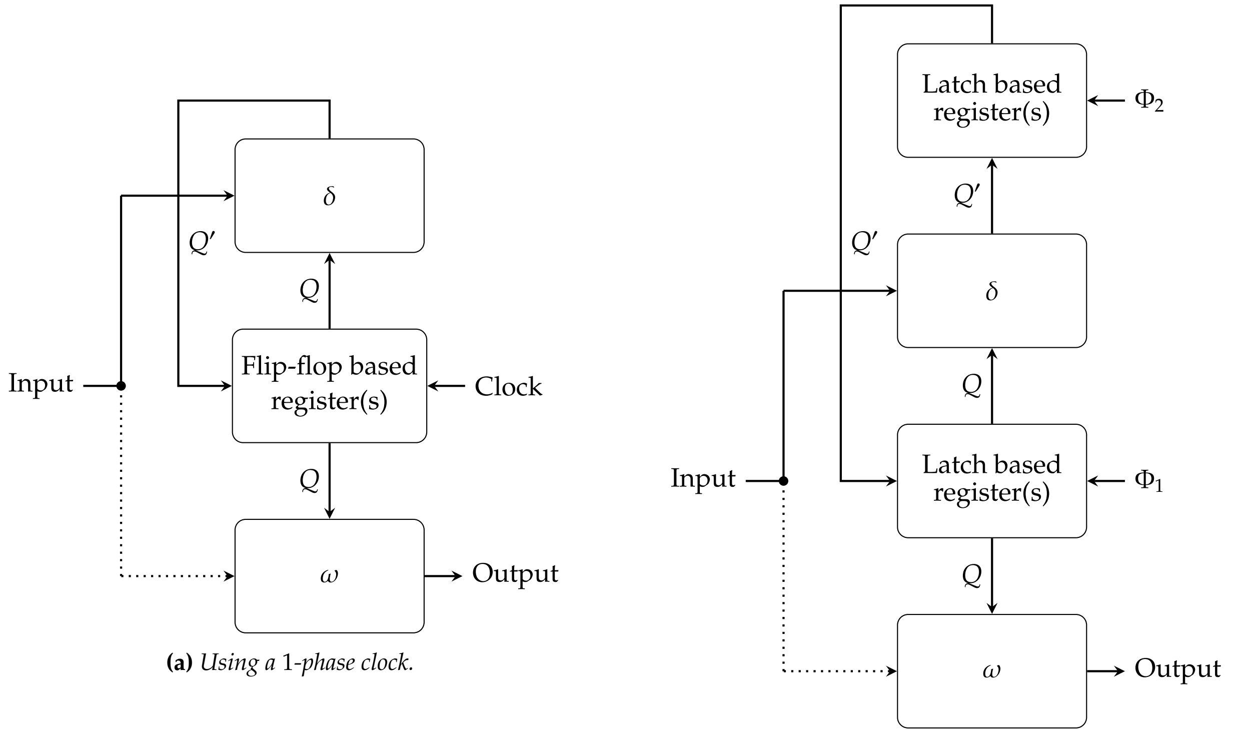
Since we finish in state $S_\perp$, the error state, the input is rejected and the tokens are returned.

Note that the input marked ϵ is the empty input; that is, with no input we can move between the accepting or error states back into the start state thus resetting the machine. So for example, once we accept or reject the input we might assume the machine returns to state S_0.

3.4.3 Practical implementation of FSMs in hardware

Based on the formal definition above, Figure 40 illustrates a general framework into which we can place concrete implementations of the component parts in a specific FSM. It is crucial to notice that when drawn as a diagram like this, we can have

1. the state implemented by register (i.e., a group of latches or flip-flops), and
2. the δ and ω functions implemented using combinatorial logic only: they are functions of the current state and any input.



(a) *Using a* 1*-phase clock.*

(b) *Using a* 2*-phase clock.*

Figure 40: *Two generic FSM frameworks (for different clocking strategies) into which one can place implementations of the state, δ (the transition function) and ω (the output function).*

The behaviour of the framework is illustrated by Figure 41. The idea is that within a given current clock cycle

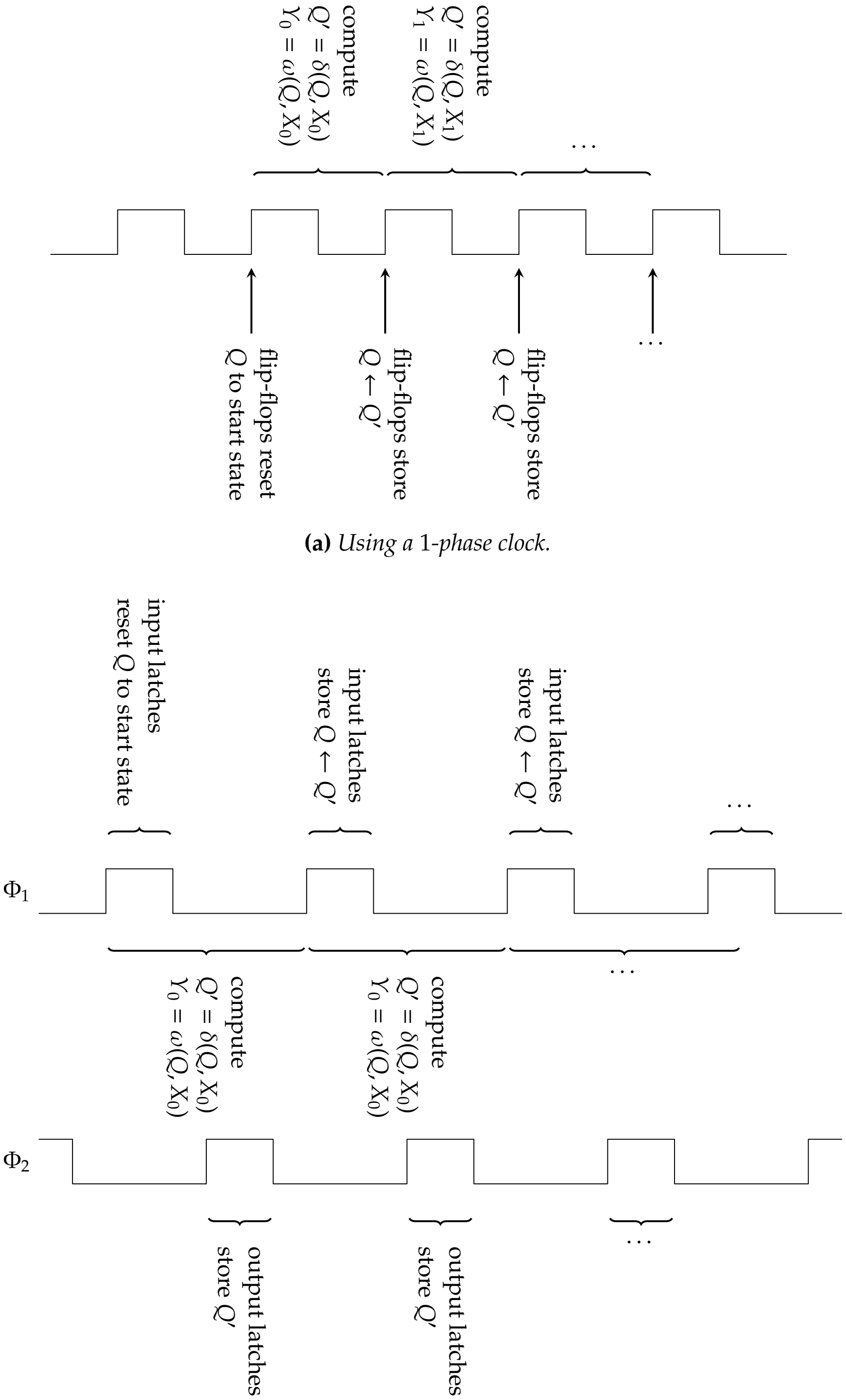
1. ω computes the output from the current state and input, and
2. δ computes the next state from the current state and input

such that the next state is latched by the positive clock edge marking the next clock cycle. So we have a period of computation in which ω and δ operate, then an update triggered by a positive clock edge which steps the FSM from the current state into the next state. What results is a series of steps, under control of the clock, each performing some computation. As such, it should be clear that the clock frequency determines how quickly computation occurs; it has to be fast enough to to satisfy the design goals, yet slow enough to cope with the critical path of a given step of computation. That is, the faster the clock oscillates the faster we step though the computation, but if it is too fast we cannot finish one step before the next one starts.

To summarise, this is a framework for a *computer* we can *build*: we know how each of the components function, and can reason about their behaviour from the transistor-level upward. To solve a concrete problem using the framework, we follow a (fairly) standard sequence of steps:

1. Count the number of states required, and give each state an abstract label.
2. Describe the state transition and output functions using a tabular or diagrammatic approach.
3. Decide how the states will be represented, i.e., assign concrete values to the abstract labels, and allocate a large enough register to hold the state.
4. Express the functions δ and ω as (optimised) Boolean expressions, i.e., combinatorial logic.
5. Place the registers and combinatorial logic into the framework.

Versus a theoretical alternative, it is less common for a hardware-based FSM to have have an accepting states since we cannot usually halt the circuit (without turning it off); we might include **idle** or **error** states to cope. In addition, and although the framework does not show it, it is common to have a **reset** input that (re)initialises the FSM into the start state. For one thing, this avoids the need to turn the FSM off then on again to reset it!



(a) *Using a* 1*-phase clock.*

(b) *Using a* 2*-phase clock.*

Figure 41: *Two illustrative timelines (for different clocking strategies), outlining stages of computation within the associated FSM framework.*

An aside: binary versus one-hot encodings.

The fact that state assignment occurs quite late in the design of a given FSM is intentional: it allows us to optimise the representation based on what we do with it. So far, we have used a fairly natural binary encoding to represent the i-th of n states as a ($\lceil \log_2(n) \rceil$)-bit unsigned integer i. For example, we have S_0 is represented by 0, S_1 by 1 and so on. But this is not the only option.

A **one-hot encoding** is where for state i, a valid code word X has $X_i = 1$ and $X_j = 0$ for $j \neq i$. For example, if $n = 6$ then

$$\begin{array}{lcl} S_0 & \mapsto & \langle 1,0,0,0,0,0 \rangle \\ S_1 & \mapsto & \langle 0,1,0,0,0,0 \rangle \\ S_2 & \mapsto & \langle 0,0,1,0,0,0 \rangle \\ S_3 & \mapsto & \langle 0,0,0,1,0,0 \rangle \\ S_4 & \mapsto & \langle 0,0,0,0,1,0 \rangle \\ S_5 & \mapsto & \langle 0,0,0,0,0,1 \rangle \end{array}$$

meaning that for S_0, the 0-th bit is 1 and all others are 0. On one hand, and depending on n, this might mean we need more flip-flops to store the state (i.e., n instead of $\lceil \log_2(n) \rceil$). On the other hand, we potentially get two advantages, namely

1. transition between states is easier (we simply rotate any given encoding by the right distance to get another), and
2. switching behaviour (and hence power consumption) is reduced since only two bits toggle for any change (one from 1 to 0, and one from 0 to 1).

Example #1: an ascending modulo 6 counter Imagine we are tasked with designing an FSM that acts as a cyclic counter modulo n (rather than 2^n as before). If $n = 6$ for example, we want a component whose output r steps through values

$$0, 1, 2, 3, 4, 5, 0, 1, \ldots,$$

with the modular reduction representing control behaviour (versus the uncontrolled counter that was cyclic by default). In this case it is clear the FSM can be in one of 6 states (since the counter value is is one of $0, 1, \ldots, 5$), which we label $S_0, S_1, \ldots, S_5$. Figure 42 includes tabular and diagrammatic descriptions of the transition function, both of which are a little dull: they simply move from one state to the next (with the ϵ meaning no input is required), cycling from S_5 back to S_0.

Clearly $2^3 = 8 > 6$, so we can represent the current state using a 3-bit integer $Q = \langle Q_0, Q_1, Q_2 \rangle$. That is,

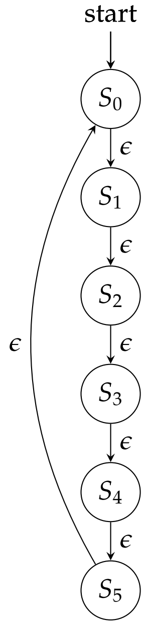
$$\begin{array}{lcl} S_0 & \mapsto & \langle 0,0,0 \rangle \\ S_1 & \mapsto & \langle 1,0,0 \rangle \\ S_2 & \mapsto & \langle 0,1,0 \rangle \\ S_3 & \mapsto & \langle 1,1,0 \rangle \\ S_4 & \mapsto & \langle 0,0,1 \rangle \\ S_5 & \mapsto & \langle 1,0,1 \rangle \end{array}$$

To implement the FSM, all we need to do is derive Boolean equations for the transition function δ so it can compute the next state Q' from Q; with this FSM there is no input, so δ is a function of the current state. To do so, we first rewrite the tabular description of δ by replacing the abstract labels with concrete values. The result is a truth table, i.e.,

				δ			ω	
Q_2	Q_1	Q_0	Q'_2	Q'_1	Q'_0	r_2	r_1	r_0
0	0	0	0	0	1	0	0	0
0	0	1	0	1	0	0	0	1
0	1	0	0	1	1	0	1	0
0	1	1	1	0	0	0	1	1
1	0	0	1	0	1	1	0	0
1	0	1	0	0	0	1	0	1
1	1	0	?	?	?	?	?	?
1	1	1	?	?	?	?	?	?

	δ	ω
Q	Q'	r
S_0	S_1	0
S_1	S_2	1
S_2	S_3	2
S_3	S_4	3
S_4	S_5	4
S_5	S_0	5

(a) *A tabular description.*

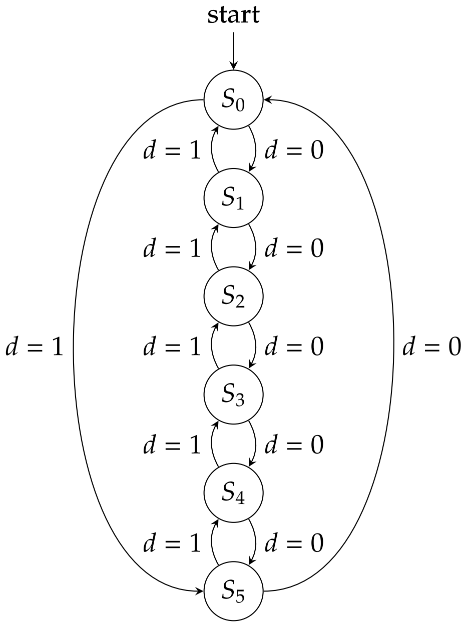


(b) *A diagrammatic description.*

Figure 42: *An example FSM modelling an ascending modulo 6 counter.*

	δ		ω		
Q	Q'		r	f	
	$d = 0$	$d = 1$		$d = 0$	$d = 1$
S_0	S_1	S_5	0	0	1
S_1	S_2	S_0	1	0	0
S_2	S_3	S_1	2	0	0
S_3	S_4	S_2	3	0	0
S_4	S_5	S_3	4	0	0
S_5	S_0	S_4	5	1	0

(a) *A tabular description.*

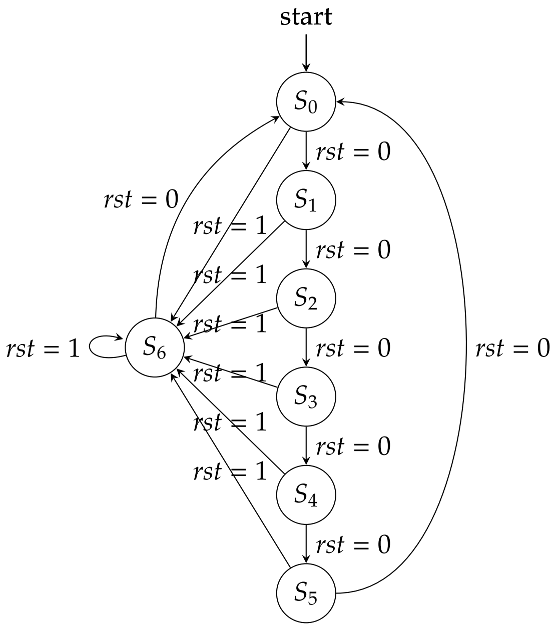


(b) *A diagrammatic description.*

Figure 43: *An example FSM modelling an ascending or descending modulo 6 counter.*

	δ		ω					
Q	Q'		M_g	M_a	M_r	A_g	A_a	A_r
	$rst = 0$	$rst = 1$						
S_0	S_1	S_6	1	0	0	0	0	1
S_1	S_2	S_6	0	1	0	0	0	1
S_2	S_3	S_6	0	0	1	0	1	0
S_3	S_4	S_6	0	0	1	1	0	0
S_4	S_5	S_6	0	0	1	0	1	0
S_5	S_0	S_6	0	1	0	0	0	1
S_6	S_0	S_6	0	0	1	0	0	1

(a) *A tabular description.*

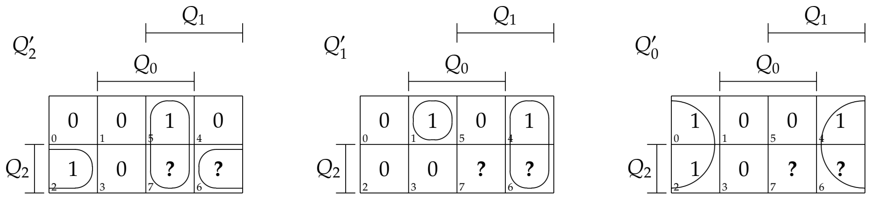


(b) *A diagrammatic description.*

Figure 44: *An example FSM modelling a traffic light controller.*

which encodes the same information. For example, if the current state is $Q = \langle 0, 0, 0 \rangle$ (i.e., we are in state S_0) then the next state should be $Q' = \langle 1, 0, 0 \rangle$ (i.e., state S_1). Note that there are 2 unused states, namely $\langle 0, 1, 1 \rangle$ and $\langle 1, 1, 1 \rangle$, which we include in the table: the next state in either of these cases does not matter since they are invalid, so the entries are don't care.

To summarise, we need to derive Boolean expressions for each of Q'_2, Q'_1 and Q'_0 in terms of Q_2, Q_1 and Q_0. This can be achieved by applying the Karnaugh map technique to get



Q'_2	0	1	5	4
	0	0	1	0
Q_2	1	0	?	?

Q'_1	0	1	5	4
	0	1	0	1
Q_2	0	0	?	?

Q'_0	0	1	5	4
	1	0	0	1
Q_2	1	0	?	?

which produce

$$\begin{array}{lllllll} Q'_2 = (& & & Q_1 & \wedge & Q_0 &) \vee \\ (& Q_2 & \wedge & & & \neg Q_0 &) \\ \\ Q'_1 = (& \neg Q_2 & \wedge & \neg Q_1 & \wedge & Q_0 &) \vee \\ (& & & Q_1 & \wedge & \neg Q_0 &) \\ \\ Q'_0 = (& & & & & \neg Q_0 &) \end{array}$$

Now we have enough to fill in the FSM framework: the state is simply a 3-bit register, δ is represented by circuit analogues of the expressions above. Note that tn this case, the output function ω is trivial: the counter output $r = Q$ due to our state assignment, so in a sense ω is just the identity function.

Example #2: an ascending or descending modulo 6 counter No imagine we need to upgrade the previous example: we are tasked with designing an FSM that again acts as a cyclic counter modulo n, but whose direction can also be controlled. If $n = 6$ for example, we want a component whose output r steps through values

$$0, 1, 2, 3, 4, 5, 0, 1, \ldots$$

or

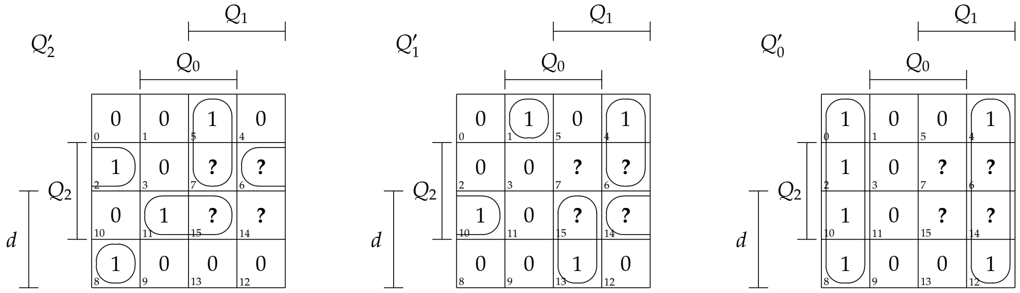
$$0, 5, 4, 3, 2, 1, 0, 5, \ldots$$

depending on some input d, plus has an output f to signal when the cycle occurs (i.e., when the current value is last or first in the sequence, depending on d).

The possible states are the same as before: we still have 6 states, labelled $S_0, S_1, \ldots S_6$. The difference is how transitions between states occur; this is illustrated by Figure 43, in which the new tabular and diagrammatic descriptions of the transition function are shown. Although it *looks* more complicated, we take exactly the same approach as before: we start by rewriting the tabular description of δ by replacing the abstract labels with concrete values to yield:

					δ			ω		
d	Q_2	Q_1	Q_0	Q'_2	Q'_1	Q'_0	r_2	r_1	r_0	f
0	0	0	0	0	0	1	0	0	0	0
0	0	0	1	0	1	0	0	0	1	0
0	0	1	0	0	1	1	0	1	0	0
0	0	1	1	1	0	0	0	1	1	0
0	1	0	0	1	0	1	1	0	0	0
0	1	0	1	0	0	0	1	0	1	1
0	1	1	0	?	?	?	?	?	?	?
0	1	1	1	?	?	?	?	?	?	?
1	0	0	0	1	0	1	0	0	0	1
1	0	0	1	0	0	0	0	0	1	0
1	0	1	0	0	0	1	0	1	0	0
1	0	1	1	0	1	0	0	1	1	0
1	1	0	0	0	1	1	1	0	0	0
1	1	0	1	1	0	0	1	0	1	0
1	1	1	0	?	?	?	?	?	?	?
1	1	1	1	?	?	?	?	?	?	?

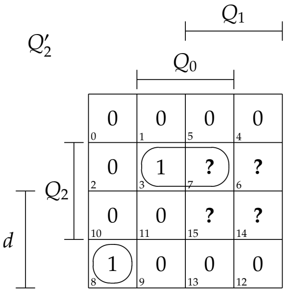
The table is larger since we need to consider d as input as well as Q, but the process is the same: to compute δ, we just need a set of appropriate Boolean expressions. So next we translate the truth table into a set of Karnaugh maps



and finally produce

$$\begin{array}{rcccccccccl}
Q'_2 = (& \neg d & & & \wedge & Q_1 & \wedge & Q_0 &) \vee \\
(& \neg d & \wedge & Q_2 & & & \wedge & \neg Q_0 &) \vee \\
(& d & \wedge & Q_2 & & & \wedge & Q_0 &) \vee \\
(& d & \wedge & \neg Q_2 & \wedge & \neg Q_1 & \wedge & \neg Q_0 &) \\
\\
Q'_1 = (& \neg d & \wedge & \neg Q_2 & \wedge & \neg Q_1 & \wedge & Q_0 &) \vee \\
(& \neg d & & & \wedge & Q_1 & \wedge & \neg Q_0 &) \vee \\
(& d & \wedge & Q_2 & & & \wedge & \neg Q_0 &) \vee \\
(& d & & & \wedge & Q_1 & \wedge & Q_0 &) \\
\\
Q'_0 = (& & & & & & & \neg Q_0 &)
\end{array}$$

This time however, we need to deal with ω more carefully: we can still generate the counter output trivially as $r = Q$, but also need to compute f somehow. This is straight-forward of course, because using the truth table we can write



0 [0]	0 [1]	0 [5]	0 [4]
0 [2]	1 [3]	? [7]	? [6]
0 [10]	0 [11]	? [15]	? [14]
1 [8]	0 [9]	0 [13]	0 [12]

and finally produce

$$\begin{array}{rcccccccl} f = (& \neg d & \wedge & Q_2 & & & \wedge & Q_0 &) \vee \\ (& d & \wedge & \neg Q_2 & \wedge & \neg Q_1 & \wedge & \neg Q_0 &) \end{array}$$

which completes the design in the sense we have now specified all components of the framework.

Example #3: a traffic light controller Imagine we are tasked with designing a traffic light controller for two roads (a main road and an access road) that intersect. The requirements are to

1. stop cars crashing into each other, so the behaviour should see
   (a) green on main road and red on access road, then
   (b) amber on main road and red on access road, then
   (c) red on main road and amber on access road, then
   (d) red on main road and green on access road, then
   (e) red on main road and amber on access road, then
   (f) amber on main road and red on access road,

   and then cycle, and
2. allow an emergency stop button to force red on both main and access roads while pushed, then reset the system into an initial start state when released.

First we need to take stock of the problem itself: there is basically one input (the emergency stop button, denoted *rst*) and six outputs (namely the traffic light values, denoted M_g, M_a and M_r for the main road and A_g, A_a and A_r for the access road). Next we try to develop a precise description of the FSM behaviour. We need 7 states in total: $S_0, S_1, \ldots, S_5$ represent steps in the normal traffic light sequence, and S_6 is an extra emergency stop state. Figure 44 shows both tabular and diagrammatic descriptions of the transition function; in essence, it is similar to the counter example (in the sense that it cycles from S_0 through to S_5 and back again) provided $rst = 0$, but if $rst = 1$ in *any* state then we move to the S_6. As an aside however, it is important to see this description represents *one* solution among several derived from what is (by design) an imprecise question. Put another way, we have already made several choices. On example is the decision to use a separate emergency stop state, and have the FSM enter this as the next state of *any* current state provided $rst = 1$; the red lights are both forced on by virtue of being in the emergency stop state, rather than by *rst* per se. Another valid approach might be to have ω depend on *rst* as well (rather than just Q, so it turns from a Moore-based into a Mealy-based FSM) and forcing the red lights on as soon as $rst = 1$ and irrespective of what state the FSM is in. In some ways this is arguably more attractive, in the sense that the emergency stop is instant: we no longer need to wait for the next clock cycle when the next state is latched. Likewise, we have opted to make the first state listed in the question (i.e., green on the main road and red on the access road) the initial state; since the sequence is cyclic this choice seems a little arbitrary, so other choices (plus what state the FSM restarts in after an emergency stop) might also seem reasonable.

Given our various choices however, we next follow standard practice by translating the description into an implementation. Since $2^3 = 8 > 7$ we can represent the current and next states via 3-bit integers $Q = \langle Q_0, Q_1, Q_2 \rangle$ and $Q' = \langle Q_0', Q_1', Q_2' \rangle$. where

$$\begin{array}{lcl} S_0 & \mapsto & \langle 0,0,0 \rangle \\ S_1 & \mapsto & \langle 1,0,0 \rangle \\ S_2 & \mapsto & \langle 0,1,0 \rangle \\ S_3 & \mapsto & \langle 1,1,0 \rangle \\ S_4 & \mapsto & \langle 0,0,1 \rangle \\ S_5 & \mapsto & \langle 1,0,1 \rangle \\ S_6 & \mapsto & \langle 0,1,1 \rangle \end{array}$$

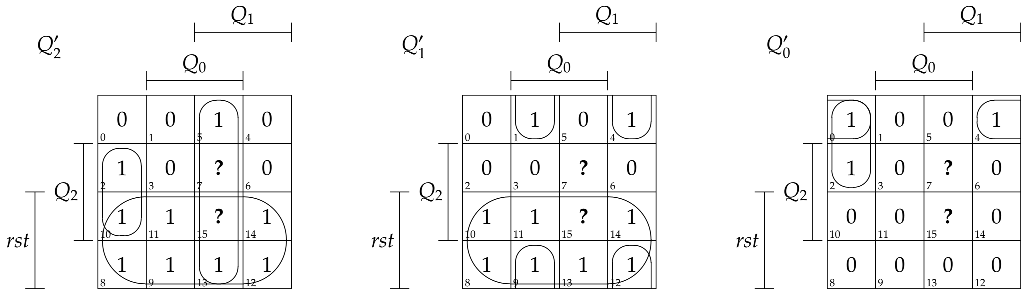
and we have one unused state (namely $\langle 1,1,1 \rangle$). As such, both input and output registers will be comprised of three 1-bit storage components, in this case D-type latches. Now we have a concrete value for each abstract state label, we can expand the tabular description of the FSM into a (lengthy) truth table:

				δ			ω					
rst	Q_2	Q_1	Q_0	Q_2'	Q_1'	Q_0'	M_g	M_a	M_r	A_g	A_a	A_r
0	0	0	0	0	0	1	1	0	0	0	0	1
0	0	0	1	0	1	0	0	1	0	0	0	1
0	0	1	0	0	1	1	0	0	1	0	1	0
0	0	1	1	1	0	0	0	0	1	1	0	0
0	1	0	0	1	0	1	0	0	1	0	1	0
0	1	0	1	0	0	0	0	1	0	0	0	1
0	1	1	0	0	0	0	0	0	1	0	0	1
0	1	1	1	?	?	?	?	?	?	?	?	?
1	0	0	0	1	1	0	1	0	0	0	0	1
1	0	0	1	1	1	0	0	1	0	0	0	1
1	0	1	0	1	1	0	0	0	1	0	1	0
1	0	1	1	1	1	0	0	0	1	1	0	0
1	1	0	0	1	1	0	0	0	1	0	1	0
1	1	0	1	1	1	0	0	1	0	0	0	1
1	1	1	0	1	1	0	0	0	1	0	0	1
1	1	1	1	?	?	?	?	?	?	?	?	?

Although this *looks* intimidating, the point is that

- the transition function δ is just three Boolean expressions, one for each Q_i', using rst, Q_2, Q_1 and Q_0 as input,
- the output function ω is just six Boolean expressions, one for each M_i and A_j, using rst, Q_2, Q_1 and Q_0 as input.

So we just need to derive each expression. For δ, the Karnaugh maps



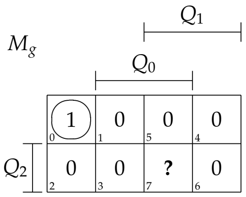
can be used to produce

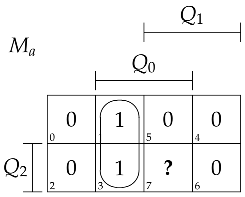
$$
\begin{array}{rcccccccl}
Q_2' = (& rst & & & & & & &) \vee \\
(& & & Q_2 & \wedge & \neg Q_1 & \wedge & \neg Q_0 &) \vee \\
(& & & & & Q_1 & \wedge & Q_0 &)
\end{array}
$$

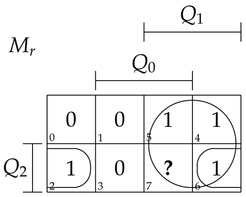
$$
\begin{array}{rcccccccl}
Q_1' = (& rst & & & & & & &) \vee \\
(& & & \neg Q_2 & \wedge & \neg Q_1 & \wedge & Q_0 &) \vee \\
(& & & \neg Q_2 & \wedge & Q_1 & \wedge & \neg Q_0 &)
\end{array}
$$

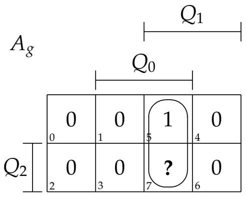
$$
\begin{array}{rcccccccl}
Q_0' = (& \neg rst & & & \wedge & \neg Q_1 & \wedge & \neg Q_0 &) \vee \\
(& \neg rst & \wedge & \neg Q_2 & & & \wedge & \neg Q_0 &)
\end{array}
$$

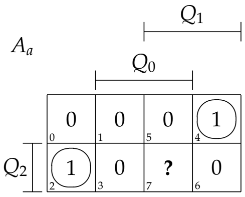
Likewise for ω, we find

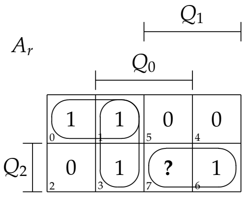












can be used to produce

$$
\begin{array}{lcl}
M_g & = & (\neg Q_2 \wedge \neg Q_1 \wedge \neg Q_0) \\
M_a & = & (\neg Q_1 \wedge Q_0) \\
M_r & = & (Q_1) \vee \\
 & & (Q_2 \wedge \neg Q_0)
\end{array}
\qquad
\begin{array}{lcl}
A_g & = & (Q_1 \wedge Q_0) \\
A_a & = & (\neg Q_2 \wedge Q_1 \wedge \neg Q_0) \vee \\
 & & (Q_2 \wedge \neg Q_1 \wedge \neg Q_0) \\
A_r & = & (\neg Q_2 \wedge \neg Q_1) \vee \\
 & & (\neg Q_1 \wedge Q_0) \vee \\
 & & (Q_2 \wedge Q_1)
\end{array}
$$

As before, these expressions can be used to fill in the FSM framework to yield a resulting design for the controller.

4 Pipelined circuits

Consider some combinatorial logic component called X. In terms of efficiency, the critical path of the component presents a major hurdle: *it* is what limits how quickly a result can be produced. To cope we might attempt one of at least two approaches, namely

1. try to apply various low-level optimisations with the goal of reducing the critical path of X, *or*
2. apply the higher-level technique of **pipelining**, restructuring X as investigated by the rest of this Section.

4.1 An analogy: car production lines

Production (or assembly) lines in the context of manufacturing offer a great analogy for the concept of pipelined circuits, which is simpler than you might expect. The basic idea of a production line is for the result to be produced as the combination of a number of stages.

Though probably not the first to employ such a process, the manufacture of cars within the Ford Motor Company is a good example. Ford, under direction of the owner Henry Ford, used a system of continuous production lines to build cars. While one person was assembling the engine to car number one, another could be attending to the body work on car number two while yet another could be fitting the wheels to car number three. By around 1913, Ford had his production line down to such a fine art that they were able to double the output of all their competitors, selling half of all cars purchased in the USA. Although assigning each worker a dedicated task reduced accident and wasted time through their wandering around the factory, the fact that they stood in the same place for long periods performing repetitive tasks meant that RSI-type injuries were common. Ford combated the resulting high turnover of staff by increasing wages to $5 a day, cutting shift lengths to eight hours a day and installing a dedicated medical department. Productivity soared and the cost of producing each vehicle decreased as a result.

Figure 45 and Figure 46 show two production lines: imagine #1 is pre-Ford and #2 is post-Ford if you want. Notice that the production of a given car is still sequential: it moves through the stages of production in order, one at a time in both cases. However, production line #2 benefits by overlapping production of *different* cars with each other, i.e., producing more than one at a time, in parallel. We can measure the efficiency of the production lines #1 and #2 using two metrics, the first of which probably seems more natural:

Definition 0.15 *The* **latency** *is the total time elapsed before a given input is operated on to produce an output. This is simply the sum of the latencies of each stage.*

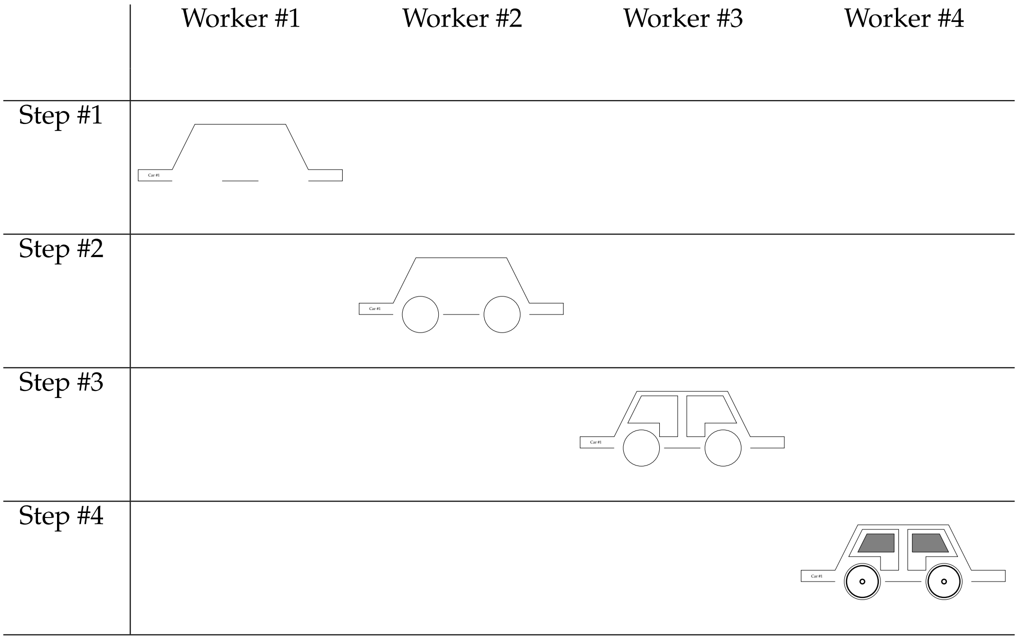


Figure 45: *Production line #1, staffed with "dumb" workers.*

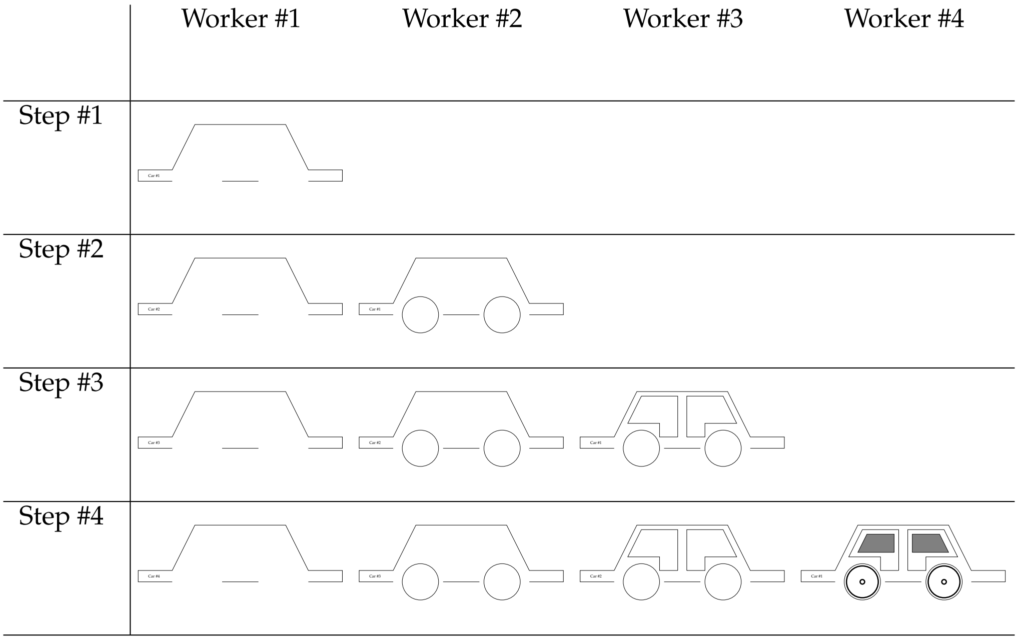


Figure 46: *Production line #2, staffed with "smart" workers.*

Definition 0.16 *The* **throughput** *(or* **bandwidth***) is the rate at which new inputs can be supplied (resp. outputs collected).*

The point is that although the latency associated with one car is not changed (it takes 4 time units to produce a car in both production lines), the throughput *is*: in production line #2 we produce a new car *every* time unit (once the production line is full), whereas we only produce one every 4 time units in #1. In a sense this is an obvious byproduct of the fact that in production line number #1 some of the stages are idle at any given time, but in number #2 they are *all* active eventually.

If we generalise, an n-stage production line will ideally give us an n-fold improvement in throughput. However, there are some caveats:

- The maximum improvement comes only when we can keep the production line full of work: if the first stage does not start because there is a lack of demand, the production line as a whole is utilised less efficiently.
- If we cannot advance the production line for some reason (perhaps one stage is delayed by a lack of parts), we say it has stalled; this also reduces utilisation.
- The speed at which the production line can be advanced is limited by the slowest stage; to minimise idle time, balance is needed between the workload of stages. That is, if there is one stage that takes significantly longer than the rest (e.g., it involves some relatively time consuming task), *it* will hold up the rest.
- Usually a production line will not be perfect: moving the result of one stage to the next will take some time, so there is some (perhaps small) overhead associated with *all* stages. This overhead typically reduces efficiency; minimising it means we can get closer to the ideal n-fold improvement.

4.2 Treating a circuit as a production line

Fortunately, pipelined circuits do not suffer from the human-related problems that the Ford production line did: our logic gates never tire, get RSI or complain about wages for example! Other than this, the principles are almost exactly the same. That is, we aim to

1. split some combinatorial logic X into a **pipeline** of n **pipeline stages**, say X_i for $0 \leq i < n$, arranged in sequence,
2. have each stage perform one step of the overall computation, with **in-flight** (or active) partial computation advancing through the pipeline stage-by-stage, and
3. supply inputs into the first stage X_0, and collect outputs from the last stage X_{n-1}.

4.2.1 Problem #1: how to structure the pipeline

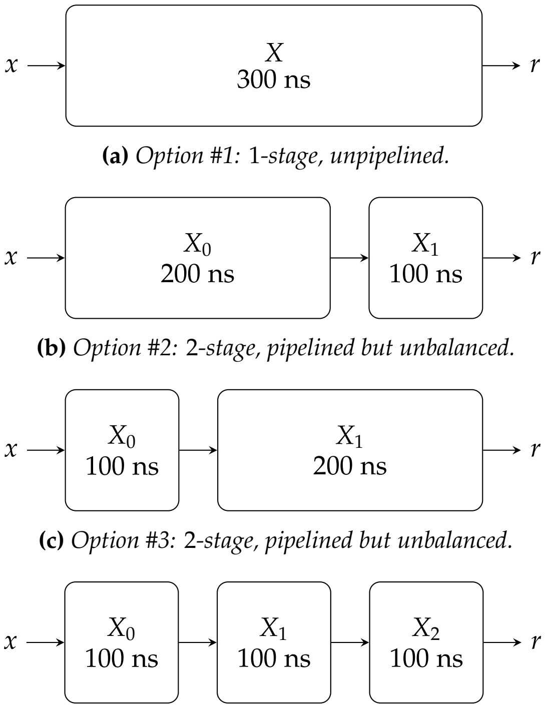
Given X, our first problem is in two parts: first where *can* we split it to produce the X_i (which depends heavily on what X *is*), and second where *should* we split it?

A generic answer to the first question is hard, since it depends on the component itself. About the most general approach we can start with is to identify natural splitting points, i.e., look at X and see where there are steps in the overall computation that can be grouped together. The second question is, however, easier: once we have an idea where we can split X, we can look at all the options and select the one that produces the best result.

More specifically, we know the slowest stage dictates how fast we can advance the pipeline; our goal is therefore to balance the stages (as far as possible) so idleness is minimised (i.e., we avoid one stage waiting for another). This is illustrated by Figure 47 wherein four options for splitting some component X into stages X_i are given:

1. a 1-stage unpipelined design, basically representing the original component X,
2. a 2-stage pipelined design where X_0 has a larger latency than X_1,
3. a 2-stage pipelined design where X_1 has a larger latency than X_0, and
4. a 3-stage pipelined design where all stages have equal latency.

Focusing *only* on the idea of balancing the stages, the last option is most attractive: since all stages take the same time to compute their part of the overall result, selecting this option will minimise potential idleness.



(a) *Option #1: 1-stage, unpipelined.*

(b) *Option #2: 2-stage, pipelined but unbalanced.*

(c) *Option #3: 2-stage, pipelined but unbalanced.*

(d) *Option #4: 3-stage, pipelined and balanced.*

Figure 47: *Four different ways to split a (hypothetical) component X into stages.*

4.2.2 Problem #2: how to control the pipeline

The next problem is how we control the pipeline so it does what we want, at the right time, to produce the right outputs from the right inputs. Consider Figure 48a, which outlines some generic pipeline stages (whose behaviour is irrelevant). There are two key problems:

1. Fundamentally, the stages cannot operate on different inputs if there is nowhere to store those inputs: if we supply a new input to X_0 at each step, where does the first input go once the first step is finished? It should be processed by X_1, but instead it will vanish when replaced with the input for the second step.

2. Imagine in the j-th clock cycle the i-th stage X_i computes a partial result t_i required by the $(i+1)$-th stage. If the stages are connected by a wire, *as soon as* the i-th stage changes t_i this potentially disrupts what the $(i+1)$-th stage is doing.

So instead, we connect the stages with by pipeline registers, say R_i. This means the $(i+1)$-th stage can have a separate, stable input which only changes when the register latches a new value, i.e., when the pipeline advances. However, each pipeline register takes time to operate, and so adds to the total latency.

Figure 48b outlines the new structure, which resolves both problems above. The structure is controlled by *adv*, shown here as a single global signal that advances all stages at the same time by having the output of each X_i stored in R_i and hence used subsequently as input to X_{i+1}. Figure 49 gives a high-level overview of progression through the pipeline, controlled by positive edges on *adv*.

The implication of this structure is that we need to take more care wrt. how we split X into stages. Specifically, more pipeline registers means larger overall latency; as a result, we cannot simply split X into as many stages as we need to have them balanced. Rather, we must make a trade-off between increased latency (as the result of some pipeline registers) and increased throughput (as the result of the pipelined design overall).

4.3 A concrete example

So far, our discussion has been necessarily abstract: many details of a concrete pipeline depend on the component under consideration. An example is Figure 50, wherein an abstract component X is shown in

An aside: synchronous versus asynchronous pipelines.

A **synchronous pipeline** is a term used to describe a pipeline structure where *all* stages are globally synchronised, controlled using a single global signal *adv* which you can think of as a clock; to re-enforce this fact, the period between advances is often termed a **pipeline cycle**.

In an **asynchronous pipeline** the aim is to remove the need for global control over when the pipeline advances, and hence remove the need for a global clock. Roughly speaking, control is devolved into the pipeline stages themselves: for one stage to advance, it must engage in a simple handshake with the preceding and subsequent stages to agree when to advance. More formally each X_i controls adv_i, the local signal that determines when it advances, by communicating with X_{i-1} and X_{i+1}.

This is advantageous in that stages can operate as fast or slow as their workload, rather than a global clock, dictates: the asynchronous pipeline can advance whenever the result is ready rather than being pessimistic and *forcing* advancement at the rate of the slowest stage. However, although the global clock is removed one potential disadvantage of this approach is overhead in provision of the handshake mechanism that has to exist between stages; clearly this can become quite complex depending on the pipeline structure.

both unpipelined and pipelined forms. Notice that for the standard, unpipelined design we find that the latency is

$$300 + 20 \text{ ns} = 320 \text{ ns},$$

while the throughput is

$$1/320 \text{ ns} = 3.12 \times 10^6 \ \mathit{operations}/\text{s}$$

if we measure the latency of computing and storing the result. However, for a 3-stage pipeline (using the same measure) the latency is

$$100 + 20 + 100 + 20 + 100 + 20 \text{ ns} = 360 \text{ ns},$$

while the throughput is

$$1/120 \text{ ns} = 8.33 \times 10^6 \ \mathit{operations}/\text{s}.$$

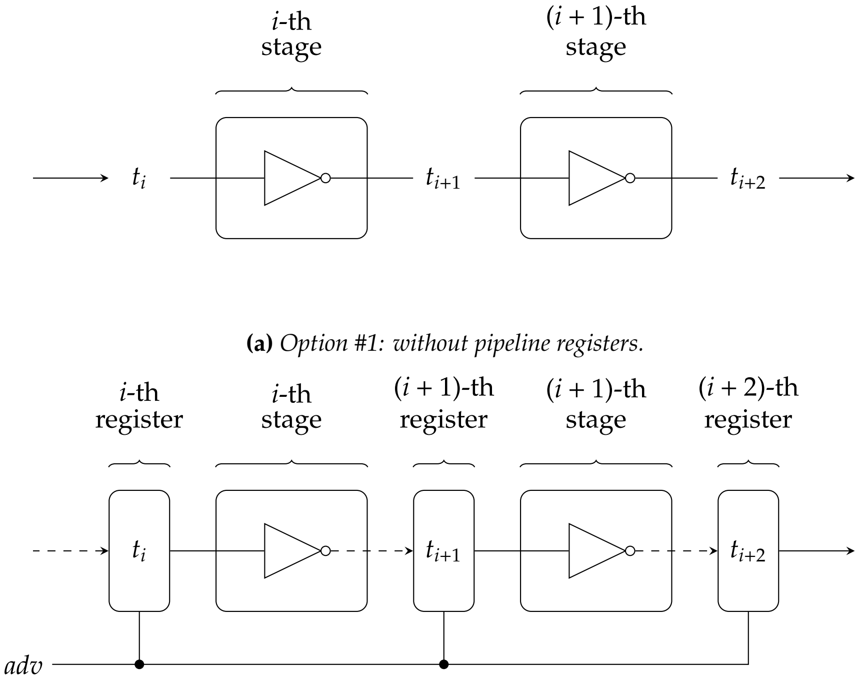
That is, we have improved the throughput by (roughly) a factor of three: we now get an output from the pipeline (resp. can provide new input) every 120 ns rather than 320 ns. The drawback is that the overall latency of a given operation is slightly more, i.e., 360 ns rather than 320 ns.

Great. But what use is this? The point is, we can relate this abstract example to a concrete component which acts as motivation for why such an improvement is worthwhile. Our example will be a component that performs the logical left-shift of some 8-bit vector x by a distance of $y \in \{0, 1, \ldots, 7\}$ bits. There are a variety of approaches to designing a circuit with the required behaviour, but one of the simplest is a combinatorial, logarithmic shifter. We will look at the design in detail in Chapter 3, but the idea is illustrated by Figure 51a. In short, the result is computed using three steps: each step produces an intermediate result by either shifting an intermediate input by some fixed distance (the i-th stage shifts by 2^i bits), or simply passing it through unaltered. For example, if we select $y = 6_{(10)} = 110_{(2)}$ then

1. since $y_0 = 0$, the 0-th stage passes the input x through unaltered to form the intermediate result x', then
2. since $y_1 = 1$, the 1-st stage shifts the intermediate input x' by a distance of $2^1 = 2$ bits to form the intermediate result x'', then
3. since $y_2 = 1$, the 2-nd stage shifts the intermediate input x'' by a distance of $2^2 = 4$ bits to form the result r

meaning overall, x is shifted by $2 + 4 = 6$ bits as required.

Applying the same reasoning as above Figure 51b splits the design into a 3-stage pipeline; this decision is natural given that the computation is trivially split into three stages of equal latency. Now, the critical path is now determined by just *one* stage rather than all *three* since each stages works independently; the 1-st and 2-nd stages, for example, compute results using an input in the 1-st and 2-nd pipeline registers while the 0-th stage computes a result using the input x. As such, we get a similar benefit as the abstract example: basically we improve the throughput by nearly a factor of three, with a slight increase in overall latency as a result of the extra registers.



(a) *Option #1: without pipeline registers.*

(b) *Option #2: with pipeline registers.*

Figure 48: *A problematic pipeline, and a solution involving the use of pipeline registers and a control signal to indicate when each stage should advance.*

5 Implementation and fabrication technologies

When we write software (i.e., a program), we usually intend to *use* it somehow (i.e., execute it on a computer). The program (or description of behaviour) is often compiled into an form we can use; depending on the processor we want to use, our program might be compiled in different ways and produce different executable forms.

In a rough sense, the same process applies to circuits: once we have a description of behaviour, we need to actually realise the corresponding components (i.e., logic gates or transistors) so that we can use them. There are various ways to achieve this, which depend on the underlying technology used: using semiconductor-based transistors is *not* the only option. Although the topic is somewhat beyond the scope of this book, it is useful to understand some approaches and technologies involved: at very least, it acts to connect theoretical concepts with their practical realisation.

5.1 Silicon fabrication

5.1.1 Lithography

The construction of semiconductor-based circuits is very similar to how pictures are printed, or at least *were* printed before the era of digital photography and laser printers! The act of printing pictures onto a surface is termed **lithography** and has been used for a couple of centuries to produce posters, maps and so on; the process involves controlled use of chemical processes within a controlled environment, often termed a dark room. The basic idea is to coat a surface, which we usually call the **substrate**, with a photosensitive chemical. We then expose the substrate to light projected through a negative, or **mask**, of the required image; the end result is a representation of said image left on the substrate where light reacts with the chemical. After washing the substrate, one can treat it with further chemicals so that the treated areas representing the original image are able to accept inks while the rest of the substrate cannot.

For semiconductors the analogous process is **photolithography**, and involves very similar steps which are illustrated by Figure 53. We again start (Figure 53a) with a substrate, which is usually a wafer of silicon; this is often circular by virtue of machining it from a synthetic ingot, or boule, of very pure silicon. After being cut into shape, the wafer is polished to produce a surface suitable for the next stage. We can now coat it with a layer of base material we wish to work with (Figure 53b), for example a doped silicon or metal. Then we coat the whole thing with a photosensitive chemical, usually called a **photo-resist** (Figure 53c).

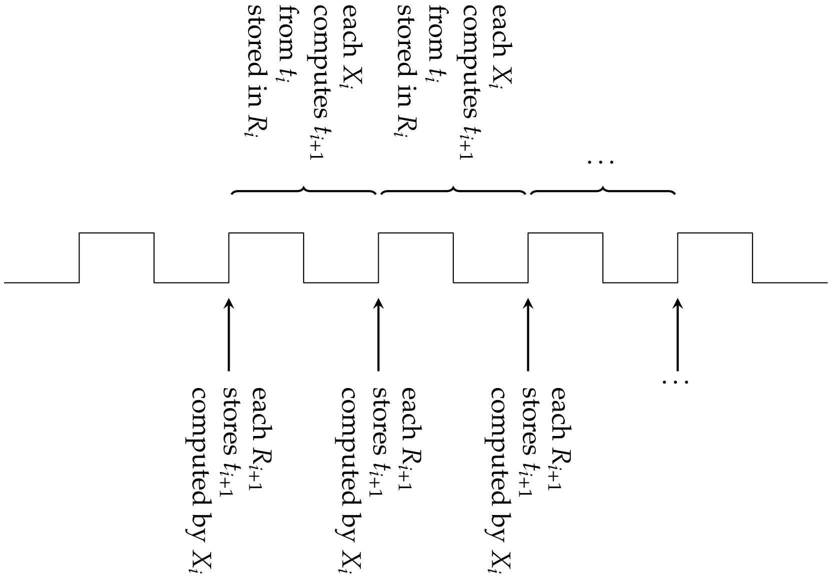
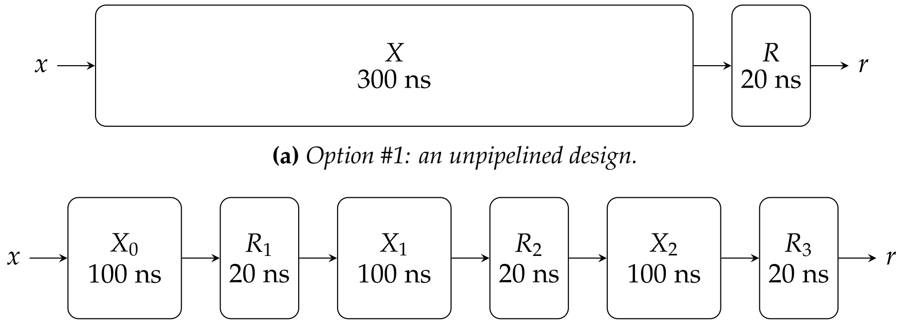

each X_i computes t_{i+1} from t_i stored in R_i
each X_i computes t_{i+1} from t_i stored in R_i
. . .
each R_{i+1} stores t_{i+1} computed by X_i
each R_{i+1} stores t_{i+1} computed by X_i
each R_{i+1} stores t_{i+1} computed by X_i
. . .

Figure 49: *An illustrative timeline, outlining the stages of computation as a pipeline is driven by a clock.*



(a) *Option #1: an unpipelined design.*

(b) *Option #2: a 3-stage pipelined design.*

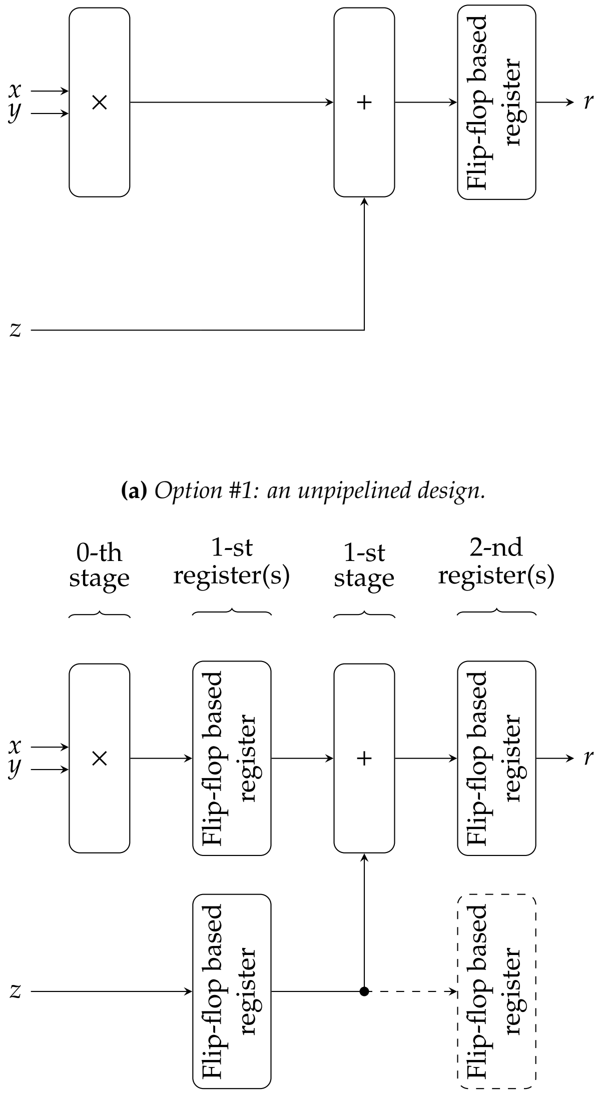
Figure 50: *An unpipelined, abstract combinatorial circuit and a 3-stage pipelined alternative.*

Two types exist, a positive one which hardens when hidden from light and a negative one which hardens when exposed to light. By projecting a mask of the circuit onto the result (Figure 53d), one can harden the photo-resist so that only the required areas are covered with a hardened covering. After baking the result to fix the hardened photo-resist, and **etching** to remove the surplus base material, one is left with a layer of the base material only where dictated by the mask (Figure 53e to Figure 53g).

The process iterates to produce *many* layers of potentially different materials, i.e., the result is 3D not 2D. We might need layers of N-type and P-type semiconductor and a metal layer to produce transistors, for example. The **feature size** (e.g., 90 nm CMOS) relates to the resolution of this process; for example, accuracy of the photolithographic process dictates the width of wires or density of transistors. Regularity of such features is a major advantage: we can manufacture many similar components in a layer using one photolithographic process. For example, if we aim to manufacture many transistors they will all be composed of the same layers albeit in different locations on the substrate.

5.1.2 Packaging

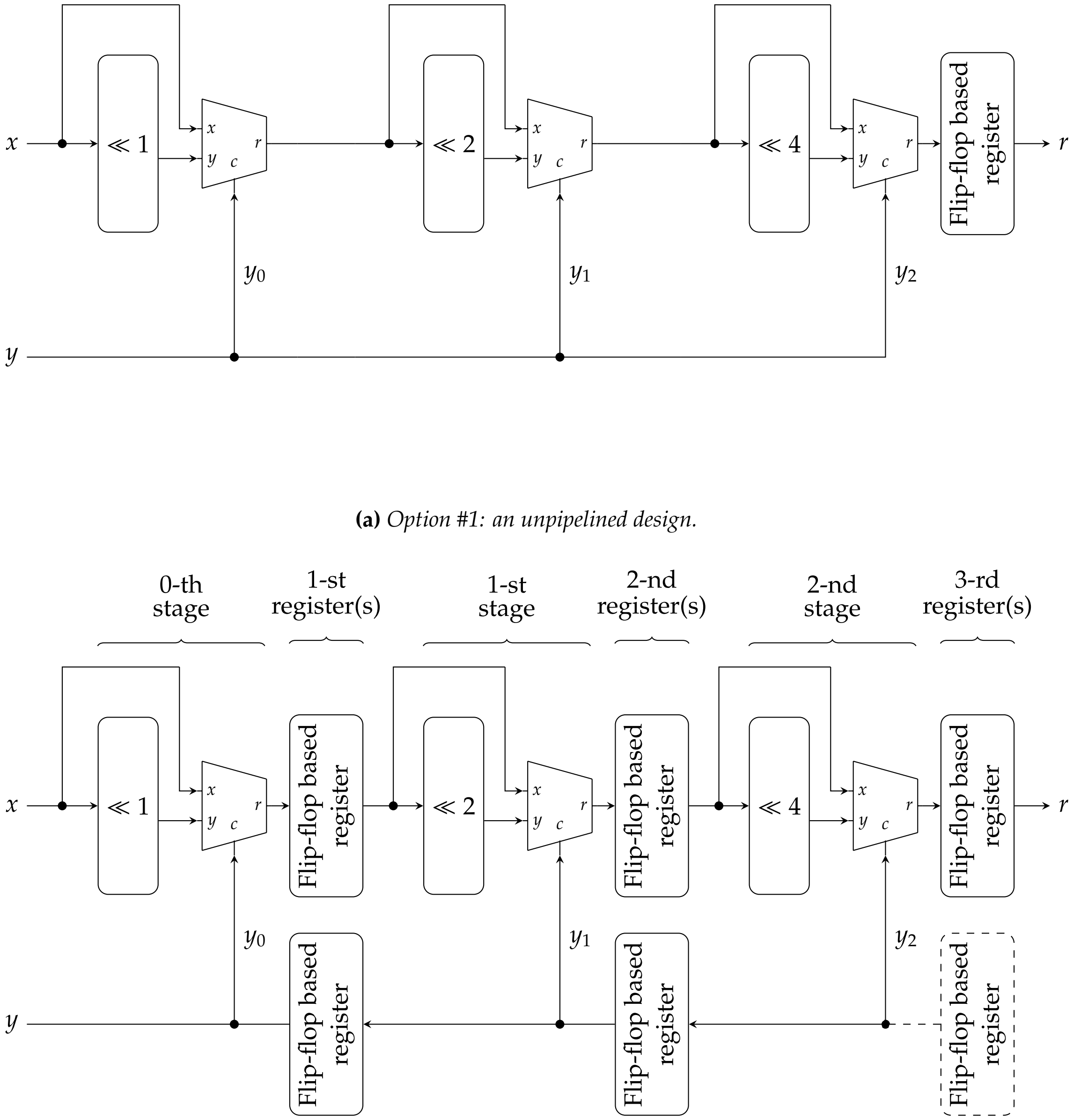
Before we can use the "raw" output from the photolithography, a process of packaging is typically applied. At very least, the first step is to cut out individual components from the resulting wafer: remember that we can produce *many* identical components using the same process, so this step gives us a single component we can use. Before we do so however, each component is typically mounted on a plastic base and connected to externally accessible **pins** (or **pads**) with **bonding wires**. This makes the inputs to and outputs from the component (which may be physically tiny *and* delicate) easier to access. A protective, often plastic, package is also applied to prevent physical damage; large or power-hungry components might also mandate use of



(a) *Option #1: an unpipelined design.*

(b) *Option #2: a* 2*-stage pipelined design.*

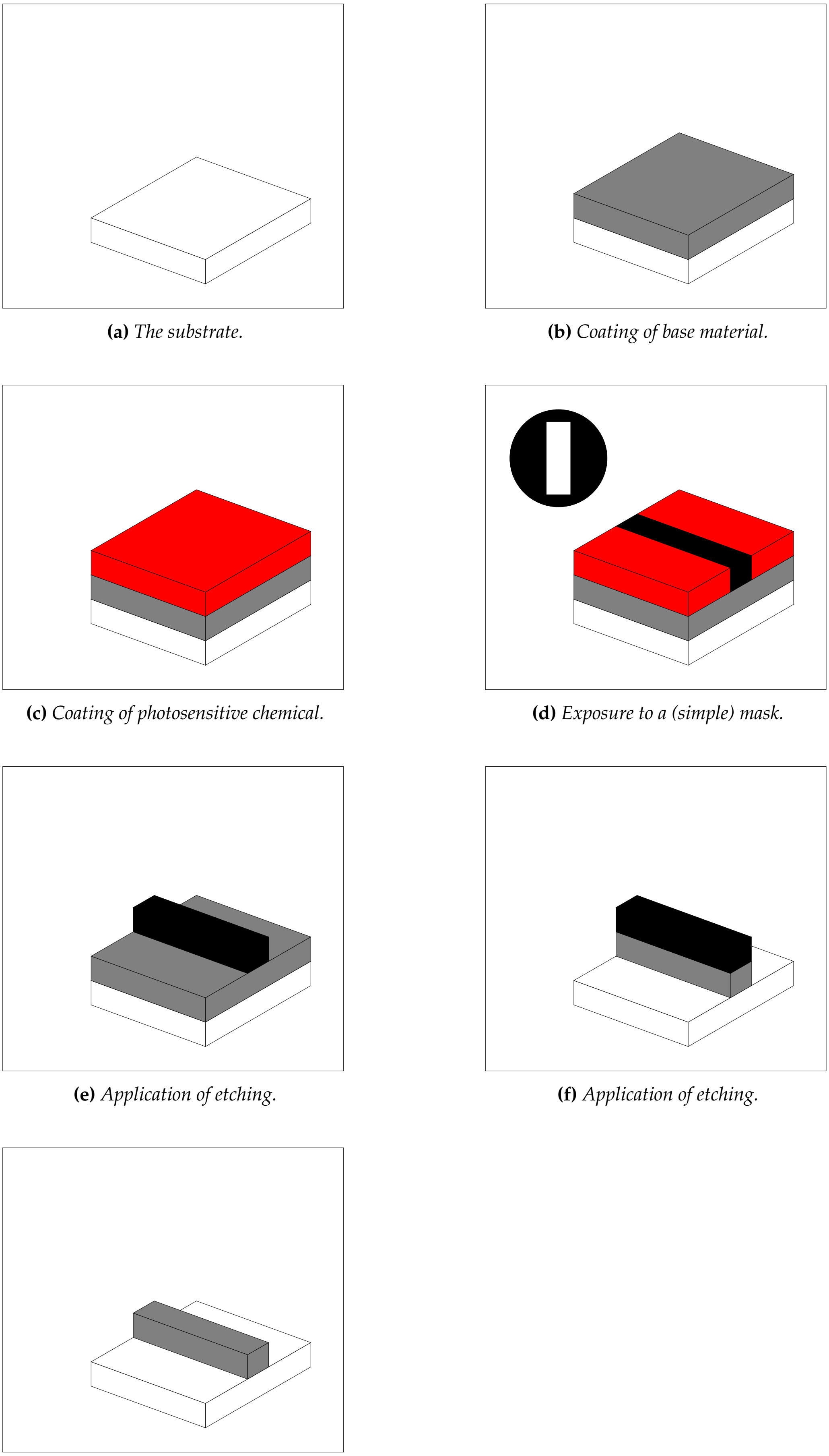
Figure 51: *An unpipelined,* 8*-bit Multiply-ACumulate (MAC) circuit and a* 3*-stage pipelined alternative.*



(a) *Option #1: an unpipelined design.*

(b) *Option #2: a 3-stage pipelined design.*

Figure 52: *An unpipelined, 8-bit logarithmic shift circuit and a 3-stage pipelined alternative.*



(a) *The substrate.*

(b) *Coating of base material.*

(c) *Coating of photosensitive chemical.*

(d) *Exposure to a (simple) mask.*

(e) *Application of etching.*

(f) *Application of etching.*

(g) *Final result.*

Figure 53: *A high-level illustration of a lithography-based fabrication process.*

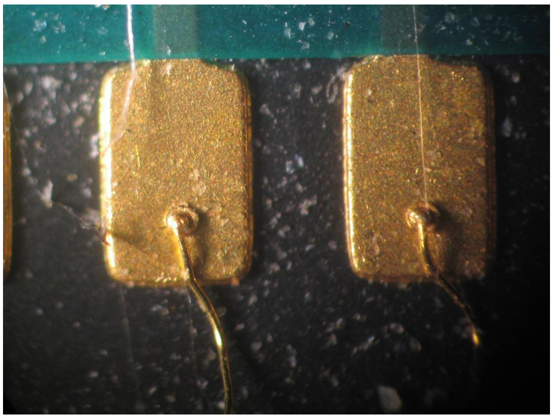


Figure 54: *Bonding wires connected to a high quality gold pad (public domain image, source: http://en.wikipedia.org/wiki/Image:Wirebond-ballbond.jpg).*

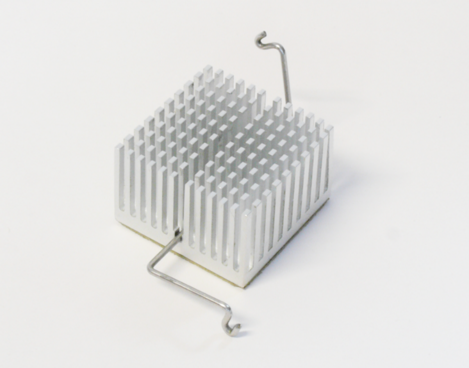


Figure 55: *A heatsink ready to be attached, via the z-clip, to a circuit in order to dissipate heat (public domain image, source: http://en.wikipedia.org/wiki/File:Pin_fin_heat_sink_with_a_z-clip.png).*

a heat sink (and fan) to dissipate heat.

The final result is a self-contained component, which we can describe as a **microchip** (or simply a **chip**) and start to integrate with other components to construct a larger system.

5.1.3 Moore's Law

Gordon Moore, co-founder of Intel, is credited with identification of an important and influential trend associated with development of transistor-based technology. The so-called **Moore's Law** was originally an observation [**?**] in 1965

> *The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least* 10 *years. That means by* 1975*, the number of components per integrated circuit for minimum cost will be* 65, 000.
>
> – G.E. Moore

and later updated: in short "the number of transistors that can be fabricated in unit area doubles roughly every two years". In a sense, this has become a form of a self-fulfilling prophecy in that the "law" is now an accepted truth: industry is *forced* to deliver improvements, and is in part driven by the law rather than the other way around!

Figure 56 demonstrates the manifestation of Moore's Law on the development of Intel processors. The implications for design of such processors, and circuits more generally, can be viewed in (at least) two ways:

1. If one can fit more transistors in unit area, the transistors are getting smaller and hence working faster due to their physical characteristics. As a result one can take a fixed design and, over time, it will get faster or use less power as a result of Moore's Law.

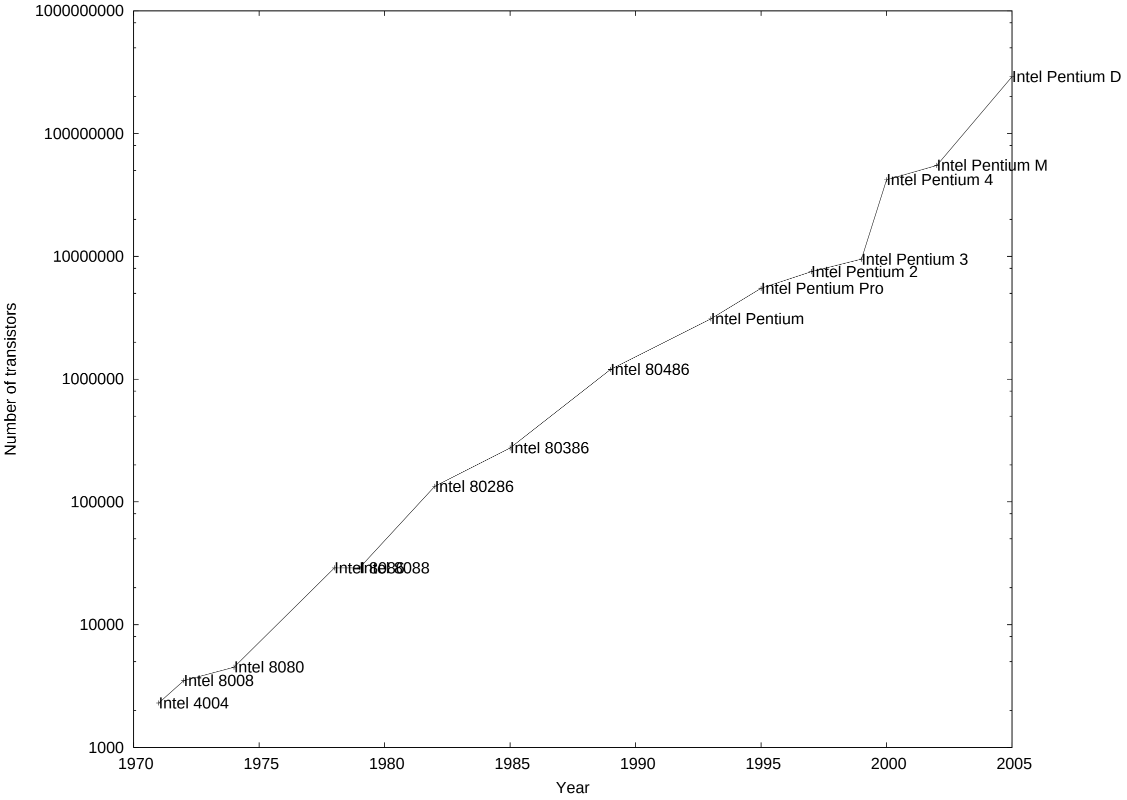


Figure 56: *A timeline of Intel processor innovation demonstrating Moore's Law (data from* http://www.intel.com/technology/mooreslaw/*).*

2. If one can fit more transistors in unit area, then one can design and implement more complex structures in the same fixed area. As a result, over time, one can use the extra transistors to improve the design yet keep it roughly the same size.

There is no "free lunch" however; Moore notes that as feature size decreases (i.e., transistors get smaller) two problems become more and more important. First, power consumption and heat dissipation become an issue: it is harder to distribute power to the more densely packed transistors *and* keep with within operational temperature limits. Second, process variation, which may imply defects and reduce yield, starts to increase meaning a higher chance that a manufactured chip malfunctions.

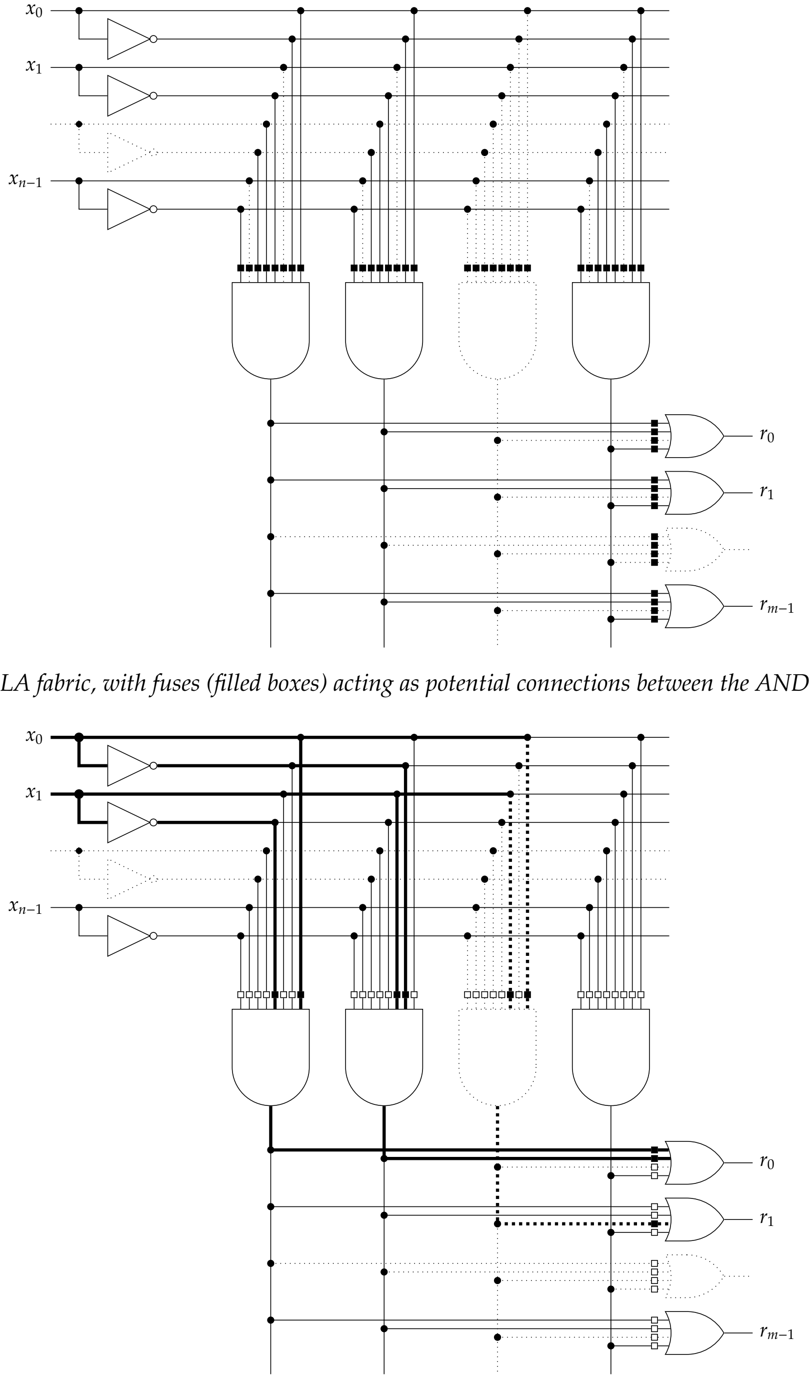
5.2 (Re)programmable fabrics

Among many alternatives to manufacture of circuits using silicon-based transistors, two in particular are interesting. You can think of them as making two steps from silicon (implying a fixed circuit once manufactured), toward a **fabric** that can be reprogrammed again and again (more like software) to form any circuit required. The resulting performance and flexibility characteristics blur traditional boundaries between hardware and software, and such fabrics are therefore increasingly important components with a broad range of applications.

5.2.1 Programmable Logic Arrays (PLAs)

A **Programmable Logic Array (PLA)** is a general-purpose fabric that can be configured to implement specific SoP or PoS expressions as combinatorial circuits. The fabric itself accepts n inputs, say x_i for $0 \leq i < n$, and produces m outputs, say r_j for $0 \leq j < m$ via logic gates arranged in two **planes**. Using an AND-OR type PLA as an example, the first plane computes a set of minterms using AND gates; those minterms are fed as input to a second plane of OR gates whose output is the required SoP expression. An OR-AND type PLA simply reverses the ordering of the planes, thus allows implementation of PoS expressions.

This does not hint at a PLA being particularly remarkable: why is it any different to the combinatorial circuits we have seen already? The crucial difference is *how* we end up with the required circuit. The starting point is a generic, clean fabric as shown in Figure 57a. At this point you can think of *all* of the gates being



(a) *A "clean" PLA fabric, with fuses (filled boxes) acting as potential connections between the AND and OR planes.*

(b) *The PLA fabric with blown fuses (empty boxes) to implement a half-adder.*

Figure 57: *Conceptual diagrams of a PLA fabric.*

connected to *all* corresponding gate inputs via existing connection points at wire junctions (filled circles), and **fuses** at the gate inputs (filled boxes). This is transformed into a specific circuit using a process roughly analogous to programming: we selectively blow fuses, guided by a **configuration** that is derived from the circuit design. Normally a fuse acts as a conductive material, somewhat like a wire; when the fuse is blown using some directed energy however, it becomes a resistive material. Therefore, to form the required connections we simply blow all the fuses[1] where no connection is required. Figure 57b shows an example, where fuses have been blown (now shown as unfilled boxes) to form various connections (shown as thick lines). As a result, this PLA computes

$$r_0 = (x_0 \wedge \neg x_1) \vee (\neg x_0 \wedge x_1) = x_0 \oplus x_1$$

and

$$r_1 = x_0 \wedge x_1,$$

i.e., it is a half-adder.

We say that a PLA fabric is **one-time programmable**. Put simply, once the fuses (or antifuses) are blown, they cannot be *un*blown: the fabric can only be configured once. But even *one*-time programmable gives a different set of features to normal silicon fabrication, and these can potentially be useful in specific contexts.

5.2.2 Field Programmable Gate Arrays (FPGAs)

Although a PLA might be useful for some tasks, two clear limitations are evident: such a fabric

1. is special-purpose in so much as it implements only SoP- or PoS-type designs, as a result of the wiring and gate structure, and is constrained by parameters such as n and m, and

2. is only one-time programmable, since once the fuses are irreversibly blown it then implements a fixed circuit.

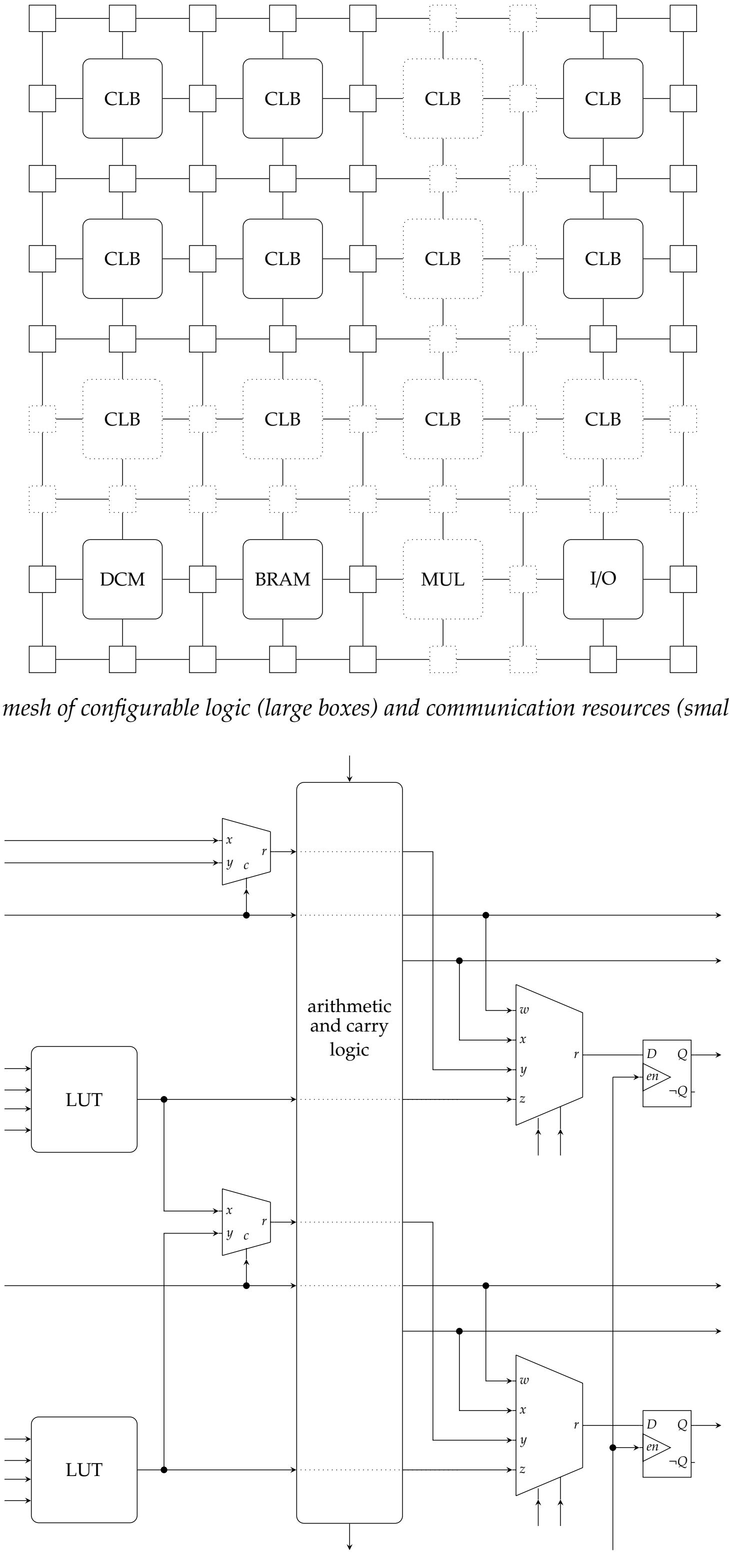
As such, one could consider generalising the underlying idea by a) allowing the wiring *and* gate structure to be configured freely, and then b) allowing this configuration to be performed *multiple* times, using some type of memory instead of fuses for each element of configuration data. A **Field Programmable Gate Array (FPGA)** fabric is the result, whose goal is basically to offer a general-purpose, **many-time programmable** fabric: the FPGA can be configured with one circuit design and then re-configured with another design at a later point in time.

Figure 58a is a conceptual representation of an FPGA fabric, which is basically a collection of logic resources (or blocks) organised in a two-dimensional mesh; the logic blocks are connected using routing resources placed between them. Both the logic and routing resources are controlled by a configuration termed a **bit-stream**. For instance, the routing resources are conceptually similar to fuses in the sense they determine connectivity; unlike fuses, they are re-configurable switches that can be turned on and off as required rather than blown in a one-off act. In a similar way the logic resources are analogous to logic gates, but now their functional can be changed to suit as part of the configuration process: a specific logic resource might be configured to act as an AND gate in one circuit, then as an XOR gate in another at some later point in time. This produces a much more flexible structure than a PLA, plus limit of one-time programmability.

This alone is a fairly big step forward, but the logic resources offer even more features internally: they are not *just* reconfigurable logic gates. Although the architecture of different brands and families within a brand differ, we focus on Xilinx Virtex-4 devices as an example. The central Vertex-4 logic resource is called a **Configurable Logic Block (CLB)**: each CLB is connected to (and hence can communicate with) immediate neighbours, and contains four **slices**. Figure 58b is a block diagram of a Vertex-4 slice, which contains

- two 4-input, 1-output Look-Up Tables (LUTs),
- two D-type flip-flops,
- a suite of arithmetic cells, including two 1-bit full-adders, and
- several interconnected multiplexers.

[1] Alternatively, one can consider an **antifuse** which acts in the opposite way to a fuse (normally it is a resistor but when blown it is a conductor). Using antifuses at each junction means the configuration process blows each antifuse at a junctions where a connection is required.



(a) *The mesh of configurable logic (large boxes) and communication resources (small boxes).*

(b) *A example Vertex-5 slice, including two LUTs, two D-type flip-flops and a suite of arithmetic cells.*

Figure 58: *Conceptual diagrams of an FPGA fabric.*

The important thing to grasp is that although this *looks* like fixed circuit design, various aspects of it are reconfigurable. A good example is the LUT content. Each LUT is basically a 16-cell SRAM memory: given a 4-bit input *i*, it reads the *i*-th SRAM cell and uses this as the 1-bit output. So by storing appropriate values in the SRAM during the device configuration phase, the LUT can be used to compute any 4-input, 1-output Boolean function. Likewise the 4-input multiplexers acting as input to the two flip-flops are controlled by the device configuration, not control-signals generated by another part of the circuit. In addition to standard CLBs, Vertex-4 FPGAs also offer various other special-purpose logic resources. Figure 58a attempts to show this fact by including

- a **Digital Clock Manager (DCM) block**, which allows a fixed input clock to be manipulated in a way that suits the device configuration,
- a **Block RAM (BRAM) block**, instances of which act like memory devices, and are often realised using SRAM or similar,
- an **Input/Output (I/O) block**, which allow off-fabric communication.

Other possibilities include common arithmetic building blocks, multipliers for instance, which would be relatively costly to construct using the CLB resources yet are often required.

The added complexity of supporting such flexibility typically means FPGAs have a lower maximum clock frequency, and will consume more power than a comparable implementation directly in silicon. As such, they are often used as a prototyping device for designs which will eventually be fabricated using a more high-performance technology. Other applications include those where debugging and updating hardware is important, meaning an FPGA-based solution is as flexible as software while also improving performance. Consider space exploration for example: it turns out to be exceptionally useful to be able to remotely fix bugs in hardware rather than write off a multi-million pound satellite which is orbiting Mars (and hence out of the reach of any local repair men).

BIBLIOGRAPHY (AND FURTHER READING)

[1] D. Harris and S. Harris. *Digital Design and Computer Architecture: From Gates to Processors*. Morgan-Kaufmann, 2007.

[2] C. Petzold. *Code: Hidden Language of Computer Hardware and Software*. Microsoft Press, 2000.