CONTENTS

BASICS OF DIGITAL LOGIC

Scientists build to learn; Engineers learn to build.

– Brooks

In the previous Chapter, we made some statements regarding various features of digital logic without backing them up with any evidence or explanation. Adopting a "from atoms upwards" approach in order to support material in subsequent Chapters, this Chapter has two central aims that, in combination, describe **digital logic***. First it expands on previous statements, such as the above, demonstrating* how *they can be satisfied using introductory Physics. Note that a detailed, in-depth treatment of such material could fill another book, and, arguably, is not strictly required given the remit of* this *book; the focus is therefore at a high level, offering an overview of only pertinent details at the right level of abstraction. For example, to connect theory such as Boolean algebra to practice, it is important to understand how we can design and manufacture implementations of Boolean operators that can* physically *provide the same functionality. Then, second, it explains* why *doing so is useful and important: the bulk of the Chapter demonstrates, step-by-step, how successively higher-level components, capable of successively more complex and so useful computation, can be designed and implemented.*

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**.

In the subsequent Sections 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 remember that transistors are not the *only* option. Put another way, provided correct switch-like behaviour is possible, we might legitimately select *another* implementation technology. Since new materials and manufacturing processes, applications and quality metrics will all appear due to advances in technology, understanding the underlying principles is as important as any specific example (such as the transistor), because it, like anything, could be superseded over time.

1.1 A brief tour of fundamental principles

1.1.1 Atoms and sub-atomic particles

Everything around us is formed from building blocks called **atoms**; in turn, each atom is formed from sub-atomic particles including a) a group of nucleons, either **protons** or **neutrons**, in a central core or **nucleus**, and b) a cloud of **electrons** orbiting said nucleus. The *number* of such sub-atomic particles yields information about the associated atom. More specifically, the number of protons dictates the **atomic number** (or family: this is what we mean by the term **element**) whereas the number of neutrons dictates the **isotope** (or instance, within that family). Likewise, the electrons can orbit the nucleus in one of several levels (or **shells**) in what is termed the **electron configuration**.

An aside: describing basic physics using the hydraulic analogy.

For some, the electrical properties of atoms and sub-atomic particles may be an unfamiliar topic. As a result, it is common, and potentially quite useful, to align them with *more* familiar concepts via the so-called **hydraulic analogy**.

Imagine a water tower (resp. battery), connected via a pipes (resp. wire) 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 wider pipe (resp. a wire with lower resistivity) allows water to flow more easily, and hence quicker, than a narrower pipe (resp. wire with higher resistivity),
- when the water reaches the water wheel, it causes it to turn as a result of two properties: the pressure (resp. voltage), and the flow rate (resp. current) of the water.

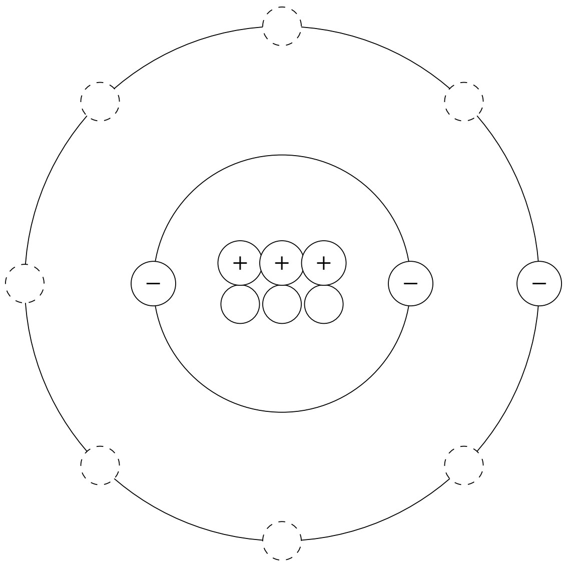


Figure 1: *The sub-atomic structure of a lithium atom.*

Example 0.1. By consulting a suitable periodic table, consider that

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

Definition 0.1. *Each type of sub-atomic particle carries a specific electrical* **charge***: electrons carry a negative charge, protons carry a positive charge, and neutrons carry no (or a neutral) charge; the unit of measurement is the* **coulomb** *(after Charles-Augustin de Coulomb). This suggests any atom with an imbalance of electrons and protons will have a non-neutral charge overall; we term such cases an* **ion***, st. negatively (resp. positively) charged ions will have more (resp. fewer) electrons than protons.*

1.1.2 Electrical charge, current, and voltage

The sub-atomic particles in an atom are bound together by forces that make sure they remain a cohesive whole. More specifically, nucleons are bound together by a "strong" nuclear force, whereas electrons are bound to the nucleus by a "weak" electromagnetic attraction to the protons; electrons in more inner shells are bound more tightly. That said, the force binding electrons can be overcome in a process called **ionisation**: an atom can be turned into an ion by displacing electrons, thereby producing imbalance between the number of electrons

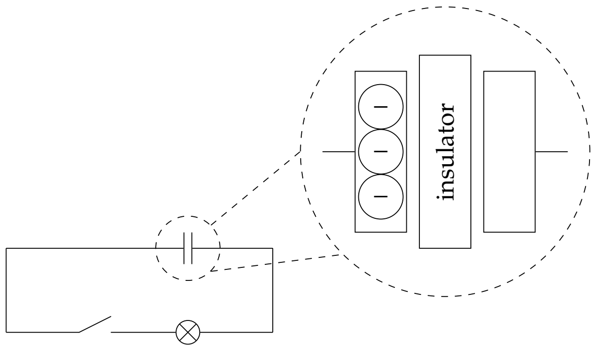
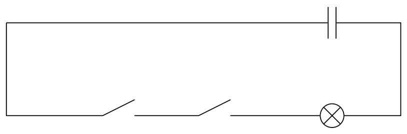
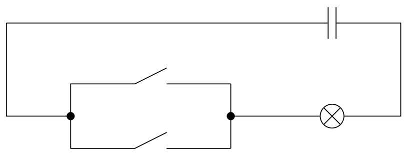


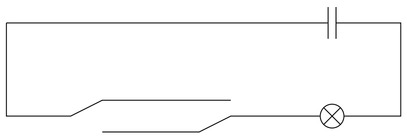
Figure 2: *A simple circuit conditionally connecting a capacitor (or battery) to a lamp depending on the state of a switch.*



(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.*

and protons, using some energy. The exact amount of energy required relates to how tightly the electrons are bound to the nucleus, and so by the type of atom. Electrons *also* exhibit a property whereby they repel each other, but are attracted by **holes** (or "gaps") in a given electron cloud; this implies they can *move*.

Definition 0.2. *Electrical* **current** *refers to a (net) flow of electric charge; the unit of measurement is the* **ampere** *(or* **amp***, after André-Marie Ampére).*

Definition 0.3. *Electrical* **potential difference** *(or, more often,* **voltage***) refers to the difference in electrical potential energy between two points per unit electric charge; the unit of measurement is the* **volt** *(after Alessandro Volta). Informally, you can think of voltage as the electrical work (or the* effort*) needed to move (or* **drive***) the electrons and hence cause a flow of current.*

Definition 0.4. *Electrical* **power** *refers to the rate of electrical work, i.e., the amount of charge driven, per unit of time, by a given voltage; the unit of measurement is the* **watt** *(after James Watt). We say electrical power is* **dissipated** *(or "consumed") when electrical potential energy associated with some charge is* converted *into another form (e.g., heat or light) by a component (or* **load***).*

An electron can move between atoms, doing so from a point of more negative charge toward a point of more positive charge, i.e., from lower to higher voltage, or driven by a potential difference. This movement or *flow* of **valence electrons** from one point to another suggests a (net) flow of charge and hence a current between the two points.

This is formally termed **electron current**, in part to distinguish it from **conventional current**: when we use the term current, we almost *universally* mean the latter. Although electron current describes the flow of negative charge, means we actually focus on what *would* be the flow of positive charge if that were possible (i.e., the opposite of electron current). Put another way, some electron moving from a more negative point X to a more negative point Y will make Y more negative and X more positive; the electron current is from X to Y, whereas the conventional current is from Y to X. This is why you might traditionally think of charge moving from a terminal labelled +ve to that labelled −ve on a battery. Set in the context of what we *now* now to be true, this *is* confusing[1]. However, it *also* has a clear historical linage we are now stuck with: Benjamin Franklin adopted this convention in the mid 1700*s*, also labelling charge using the positive or negative terminology, during his pioneering study of electricity. Either way, from here on, you should read current as a synonym for conventional current.

1.1.3 Conductors, insulators and semi-conductors

Definition 0.5. *Two materials with different sub-atomic composition may exhibit different properties wrt. their* **conductivity** *and* **resistivity***; these terms (which are antithetical) state whether a material allows or prevents the movement of electrons.*

Definition 0.6. *A* **conductor***, e.g., a metal, has high-conductivity (resp. low-resistivity) and allows electrons to move easily, whereas an* **insulator***, e.g., a vacuum, has low-conductivity (resp. high-resistivity) and does not allow electrons to move easily.*

When we describe a material as conductive or resistive, we typically mean it is on a spectrum *between* the two: although unlikely to represent a perfect conductor or insulator, we mean it is closer to one end of the spectrum or other (e.g., is more conductive or more resistive). Although such properties are inherent in the material, it is possible to explicitly manipulate the sub-atomic composition of semi-conductor materials using a process called **doping**. For example, imagine we need a material for some task; any non-ideal material will have non-ideal properties wrt. the task. The idea is instead to take some non-ideal material as a starting point, then **dope** (or combine) it with a **dopant** material: their combination should be similar to the starting point, but more ideal wrt. the properties required.

Example 0.2. 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.

An important use-case for doping is the production of **semi-conductor** materials. Although various materials might exhibit the properties of a semi-conductor, doping allows careful control over the ratio of electrons vs. hole and hence conductivity (resp.resistivity) of the result. Rather than rely on the perfect material being naturally available, we therefore produce a material with exactly the properties required for a given task.

Definition 0.7. *A doped semi-conductor material falls into one of two classes, namely*

1. *an* **N-type semi-conductor** *has an abundance of electrons produced by doping with a* **donor** *material, or*
2. *a* **P-type semi-conductor** *has an abundance of holes produced by doping with a* **acceptor** *material.*

1.1.4 Using switches for computation

Example 0.3. Consider Figure 2, which includes a capacitor (top), a lamp (bottom-right), and a push-button switch (bottom-left). The capacitor is constructed using two conductive plates separated by an insulator (called a dielectric); it stores electrical energy (meaning it is similar to a battery[2]), and, since this one has already been charged, one of the plates has many electrons and the other many holes. Electrons cannot move through the insulator, so the only way for them to move from the negative onto the positively charged plate (i.e., from low to high potential) is via the (conductive) wire. This is only possible when the switch is closed: when the switch is closed, electrons are allowed to flow through the lamp which causes it to light up.

Following from this example, it may be worth convincing ourselves that a switch *is* useful for something beyond controlling a lamp as above. To provide an answer, we just need to generalise the example: we a) use *multiple* switches, and b) treat each switch as an input and the lamp as an output. Put another way, imagine we have two switches labelled x and y; we are interested in how their combination controls the lamp labelled r, so, in effect, what the function f described by $r = f(x, y)$ is.

Example 0.4. Consider Figure 3:

1. Figure 3a controls the lamp via two switches, and models an AND operator: $r = f(x, y) \models x \wedge y$. Only when both of the switches are closed will the lamp be on: if either is open, there is no connection with the battery.

[1]See, e.g., http://xkcd.com/567/.

[2]Although the analogy is reasonable, keep in mind that a battery differs from a capacitor: behaviour of the former is due to chemical a process, which converts chemical energy to electrical energy and thus delivers a flow of electrons (i.e., a current).

2. Figure 3b controls the lamp via two switches, and models an OR operator: $r = f(x, y) \models x \vee y$. 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 models an XOR operator: $r = f(x, y) \models x \oplus y$. 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.

On one hand, the examples above should be encouraging: they show we can mirror the behaviour of Boolean operators, using a careful organisation of multiple switches. On the other hand, however, push-button switches are *mechanically* operated: we want an *electrically* operated switch, which is actuated (i.e., pressed or released) via an electrical property (e.g., a flow of electrons) rather than by hand. Crucially, this will allow the output of one such operator to be used as the input to another, and therefore the implementation of larger expressions.

1.2 Implementing transistors

1.2.1 Switches in a pre-transistor era: vacuum tubes

From a historical perspective, numerous different electrically operated switch designs have been conceived and used; it is both interesting and useful to examine them in some detail, because their properties act as motivation for modern alternatives. In particular, the **vacuum tube** (or **thermionic valve**), is a compelling example 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.

The design outlined above, plus an example in Figure 4, hint at some potential disadvantages: they offer the functionality we require, *but*, in relative terms, are physically large, operate slowly, and are unreliable. The latter properties both stem from the need to (repeatedly) heat and cool the filament. This takes some time, and stresses the filament up to a point where it fails (much like a light bulb failing when turned on or off). As an aside, the terms **bug** and **debug** (allegedly) stem from failure of this sort. In 1945, programmers using 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 the terms had been used previously in various other contexts, Grace Hopper and her team are now often cited as introducing them to Computer Science. Certainly this real-life bug, shown in Figure 5, is so famous it is still on display in the Smithsonian Museum of American History!

1.2.2 Design of MOSFET transistors

A transistor can in fact be used for a various tasks; for example, they can act as amplifiers. However, 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 transistor is a portmanteau of "transfer resistor", offering a hint as to the underlying principle: a transistor *is* a resistor, but one we can *control* by altering how resistive it is. Put more simply, we can control it st. it is conductive when we want to turn the switch on and resistive when we want to turn it off.

The question is then *how* such behaviour is realised. Improvement and different trade-offs have given us numerous transistor designs, but we focus on just one: the **Field Effect Transistor (FET)**, initially designed and patented by Julius Lilienfeld in 1925. However, at that point in time the general understanding of sub-atomic behaviour was less than now, meaning use of his design was limited. This changed in 1952, when a team of Engineers at Bell Labs, led by William Shockley, invented what is now termed a junction gate FET (or JFET, due to some legal wranglings wrt. to the Lilienfeld patent). In turn, this gave rise to the **Metal Oxide Semi-Conductor Field-Effect Transistor (MOSFET)**, invented in 1960 by Dawon Kahng and Martin Atalla, also at Bell Labs. These designs delivery the properties we require to avoid their limiting complexity of modern digital logic components; in particular, they a) have the switch-like functionality described as useful thus far, while also b) are simultenously physically small, operate quickly, are reliable, *and* easy to manufacture.

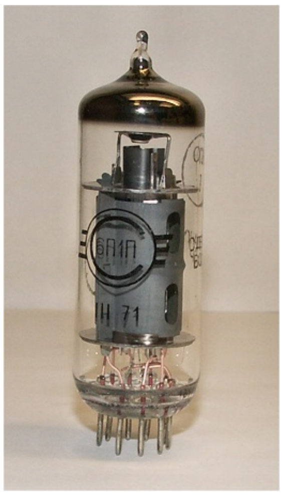
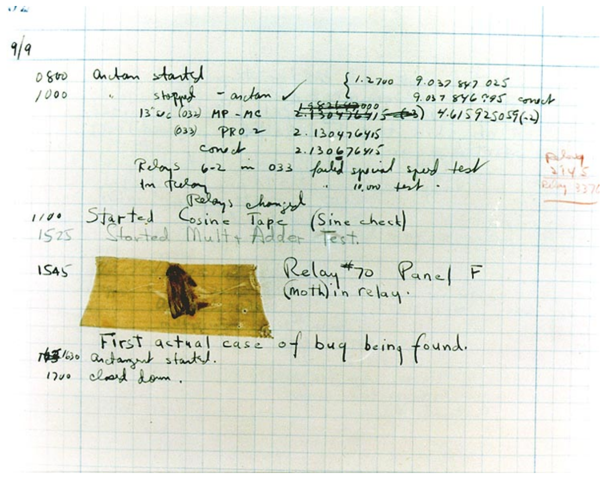


Figure 4: *A 6P1P (i.e., a 100W to 200W, photo-sensitive type) vacuum tube (public domain image, source:* http://en.wikipedia.org/wiki/File:6P1P.jpg*).*



9/9

0800 antan started
1000 " stopped - antan ✓ {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 " 10,000 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~~ 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*).*

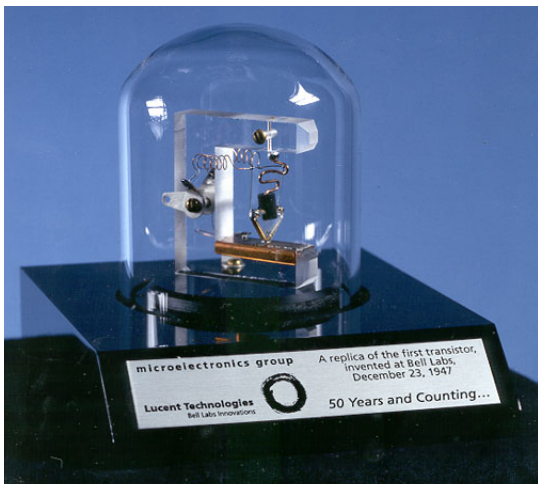


Figure 6: *A replica of the first point-contact transistor, a precursor of designs such as the MOSFET, constructed at Bell Labs (public domain image, source:* http://en.wikipedia.org/wiki/File:Replica-of-first-transistor.jpg*).*

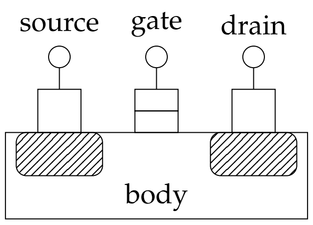


Figure 7: *A high-level diagram of a MOSFET transistor, showing the terminal and body materials.*

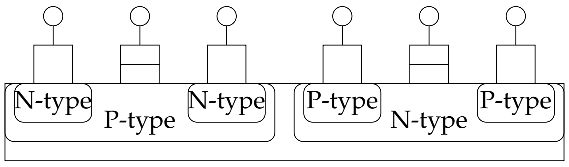


Figure 8: *A pair of N-MOSFET and P-MOSFET transistors, arranged to form a CMOS cell.*

Figure 7 offers a high-level description of a MOSFET, in which atomic-scale layers of semi-conductor material are combined with metal or poly-silicon layers for the terminals; although a lower-level, more detailed description would require deeper understanding of related Physics (see, e.g., [8]), we already have enough background to explain the basic concept at this high level. In short, the switch-like behaviour is realised by using the **gate** terminal to control a conductive channel between the **source** and **drain** terminals . Unlike a a JFET, where an explicit semi-conductor layer is constructed for use as the channel, in a MOSFET transistor the channel is *induced*. Specifically, applying a small potential difference to the gate terminal repels holes in the P-type body; doing so forms a **depletion layer** in which the number of holes is depleted. As the potential difference applied grows, an **inversion layer** is formed at the surface: the abundance of electrons relative to the number of (repelled) holes inverts the properties of the P-type body, turning it into N-type and so forming a conductive channel between N-type source and drain terminals.

Realising this behaviour in practice depends on the careful selection of semi-conductor materials; Figure 9 illustrates the symbols used for two MOSFET variants. These symbols abstract away the implementation detail (retaining only the terminals, with d, s and g denote the drain, source and gate), which is as follows:

Definition 0.8. *An* **N-MOSFET** *(or* **N-type MOSFET***, or* **N-channel MOSFET***, or* **NPN MOSFET***) is constructed from N-type semi-conductor terminals and a P-type body:*

- *applying a potential difference to the gate widens the conductive channel, meaning source and drain are connected (i.e., act like a conductor); the transistor is activated.*
- *removing the potential difference from the gate narrows the conductive channel, meaning source and drain are disconnected (i.e., act like an insulator); the transistor is deactivated.*

Definition 0.9. *A* **P-MOSFET** *(or* **P-type MOSFET***, or* **P-channel MOSFET***, or* **PNP MOSFET***) is constructed from P-type semi-conductor terminals and an N-type body:*

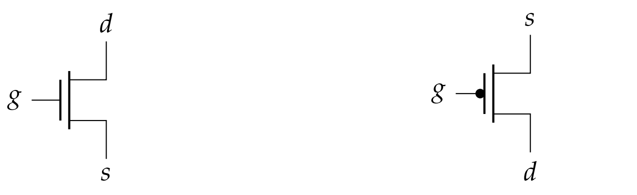
- *applying a potential difference to the gate narrows the conductive channel, meaning source and drain are disconnected (i.e., act like an insulator); the transistor is deactivated.*
- *removing the potential difference from the gate widens the conductive channel, meaning source and drain are connected (i.e., act like a conductor); the transistor is activated.*

Put another way, for an N-MOSFET, applying a large potential difference to the gate terminal produces a wider conductive channel, and so allows electrons (i.e., current) to flow between source and drain. Conversely, a small potential difference (or at least smaller than some threshold) means a narrower conductive channel, which prevents said flow. The gate terminal therefore offers functionality much like a switch: controlling the potential difference applied controls conductivity between source and drain, and hence regulates the current.

1.2.3 Physical properties of MOSFET transistors

Various physical properties stem from the design of MOSFET transistors; since they are related, we define these step-by-step in what follows.

Definition 0.10. *One or more* **power rails** *supply voltage levels to each transistor, connecting to the gate or source terminal.*



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

Figure 9: *Symbolic descriptions of N-MOSFET and P-MOSFET transistors.*

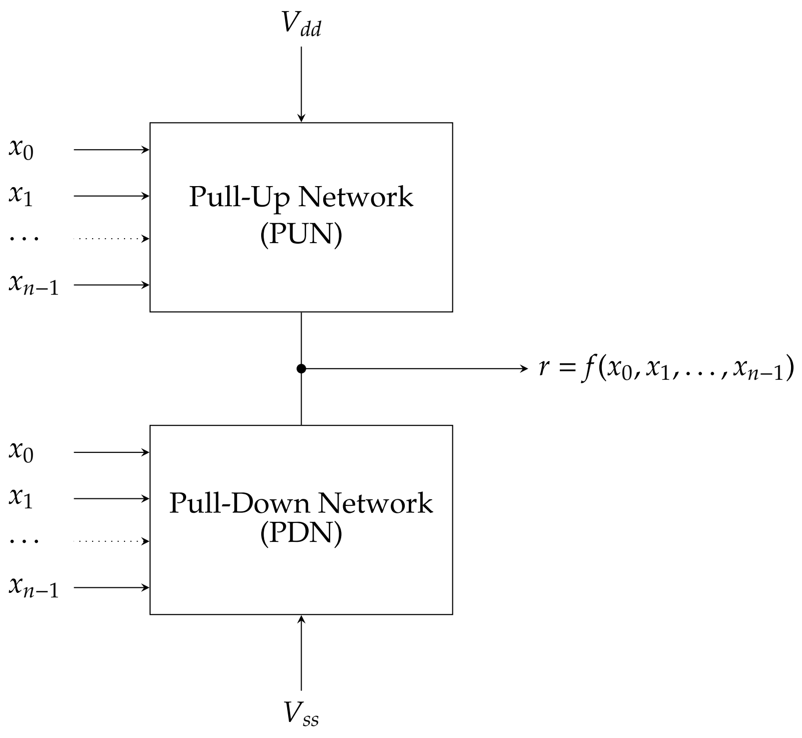


Figure 10: *A generic CMOS-based design strategy, utilising complementary pull-up and pull-down networks.*

Definition 0.11. *The* **threshold voltage** *of a given MOSFET (i.e., either N- or P-MOSFET) is the minimum voltage level (i.e., potential difference between gate and source) required to activate the transistor and thus connect the source and drain; below the threshold voltage, the source and drain remain disconnected.*

Definition 0.12. *The concept of* **sub-threshold leakage** *(or just* **leakage***) relates to a non-ideal properties of the conductive channel: below the threshold voltage the source and drain are not* perfectly *disconnected, st. a small flow of electrons (i.e., the leakage current) is possible.*

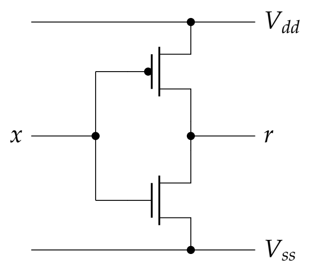
1.2.4 Organisation of MOSFET transistors into CMOS-based logic gates

Rather than use MOSFET transistors in isolation, it is common to organise them into larger combinations; by offering a higher level of abstraction, such combinations are usually easier to reason about from both functional *and* behavioural perspectives.

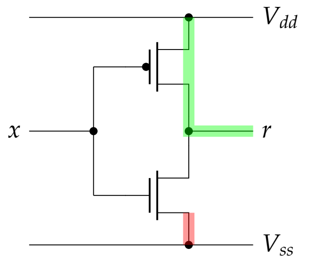
Ultimately the aim is to (re)produce Section 1.1 where we outlined Boolean-like functionality using mechanical switches, but now by using *transistors*. A popular[3] first step relates to organisation of *two* transistors (pairing an N-MOSFET with a P-MOSFET) to form *one* **Complementary Metal-Oxide Semi-Conductor (CMOS)** component we term a **cell**. This approach, as illustrated at a high-level by Figure 8, was first conceived in 1963 by Frank Wanlass at Fairchild Semi-conductor. The idea is to organise the transistors so they operate in a complementary manner:

Definition 0.13. *CMOS-based design strategies typically use two distinct parts to form a given component: there will be*

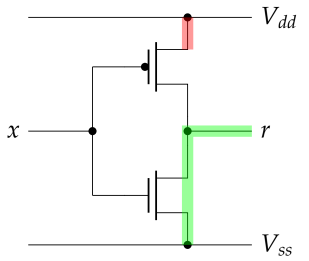
[3]It is important to stress that CMOS is not the *only* possible logic style: although it represents a first step here, it may not be necessary if an alternative is used instead.



(a) *The NOT gate implementation.*



(b) *Annotation for the case $x = V_{ss}$.*



(c) *Annotation for the case $x = V_{dd}$.*

Figure 11: *A CMOS-based NOT gate implementation.*

1. *a* **Pull-Up Network (PUN)** *of P-MOSFET transistors between the V_{dd} power rail and the output, and*
2. *a* **Pull-Down Network (PDN)** *of N-MOSFET transistors between the V_{ss} power rail and the output.*

A consequence of this logic style is that only one *of the pull-up or pull-down networks can be active (i.e., connected) at a time.*

Figure 10 captures this design strategy in a diagrammatic form. The idea is that the PUN will connect r to V_{dd} whereas the PDN will connect r to V_{ss}, each based on the inputs n inputs, i.e., x_i for $0 \leq i < n$, and in a complementary manner: either the PUN *or* the PDN will drive r, never both.

Definition 0.14. *The power dissipation of a CMOS cell, and hence a CMOS-based design more generally, can be described in terms of*

1. *a* **static** *component, where the transistors remain in a given state (to are "idle" in some sense), and*
2. *a* **dynamic** *component, where the transistors* **switch** *state, i.e., the gate is changes from being driven by V_{dd} to V_{ss}, or vice versa.*

CMOS exhibits a marginal amount of sub-threshold leakage, so the majority of power dissipation occurs due to switching activity.

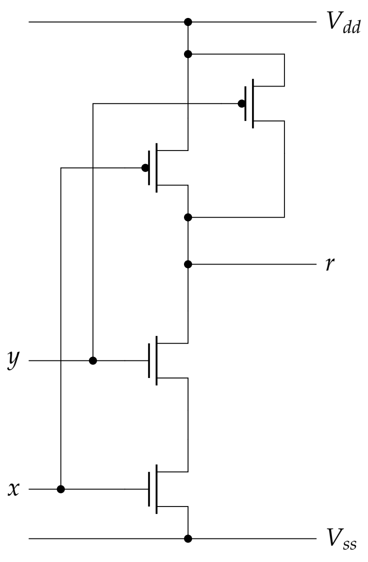
This has some obvious advantages, which make CMOS an attractive choice vs. alternatives. In particular, when organising lots of transistors in close proximity, CMOS will have lower overall power consumption *and* heat dissipation, and, in turn, better reliability.

The next step is to package CMOS cells into small, useful building-blocks that act as the next-level component above transistors themselves. As an example, consider building a component which inverts the input st. if the input x is V_{dd} the output is V_{ss} and vice versa.

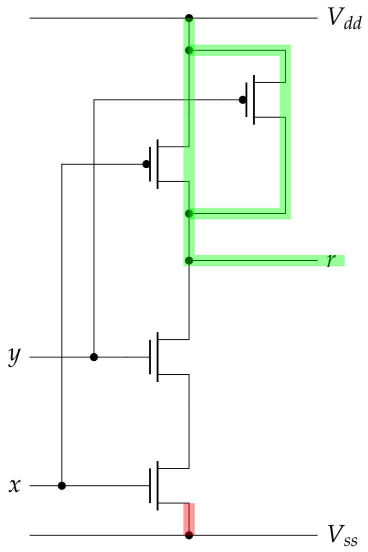
Example 0.5. Consider Figure 11, where

1. connecting x to V_{ss} 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 V_{ss}.

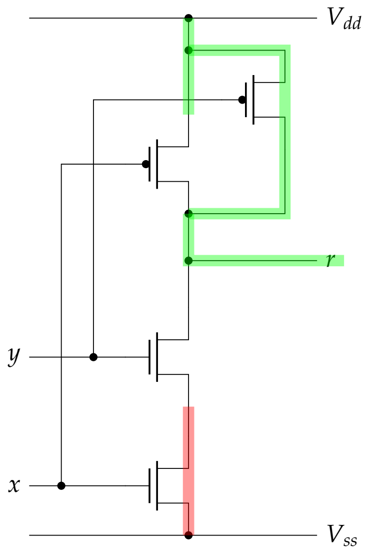
Note that even with this simple organisation, we can identify the pull-up and pull-down networks; although there is just one transistor in each, it is true that the P-MOSFET connects V_{dd} to the output iff. $x = V_{ss}$ and the N-MOSFET connects V_{ss} to the output iff. $x = V_{dd}$. We can of course consider more complex organisations under the *same* design strategy, by increasing the number of transistors.



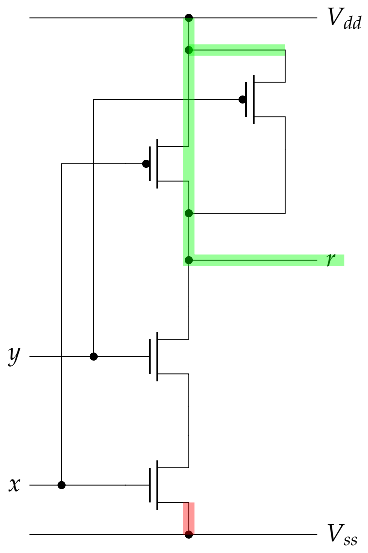
(a) *The NAND gate implementation.*



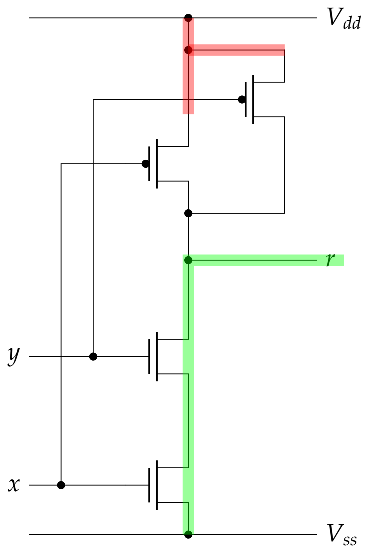
(b) *Annotation the case for* $x = V_{ss}$, $y = V_{ss}$.



(c) *Annotation the case for* $x = V_{dd}$, $y = V_{ss}$.

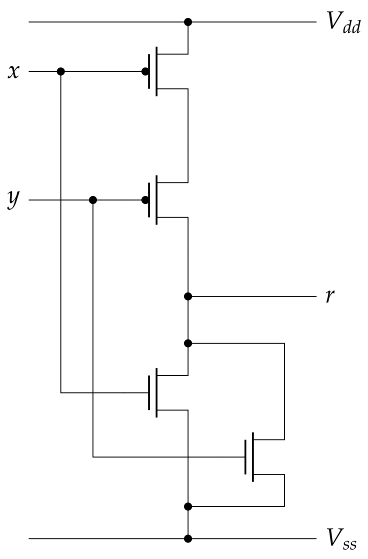


(d) *Annotation the case for* $x = V_{ss}$, $y = V_{dd}$.

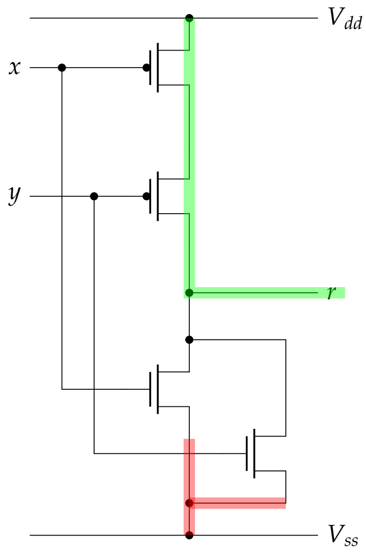


(e) *Annotation the case for* $x = V_{dd}$, $y = V_{dd}$.

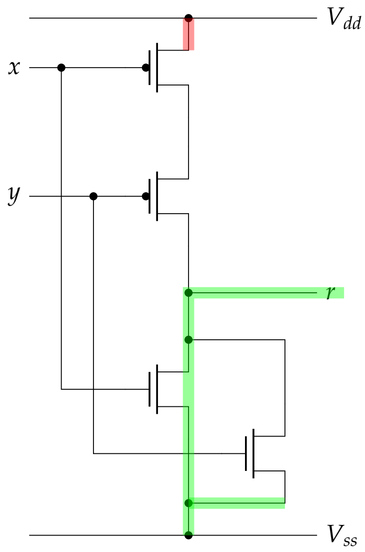
Figure 12: *A CMOS-based NAND gate implementation.*



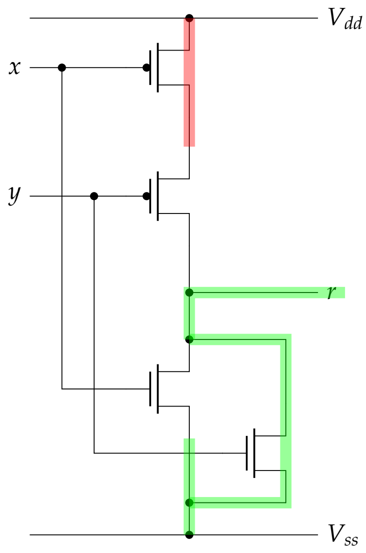
(a) *The NOR gate implementation.*



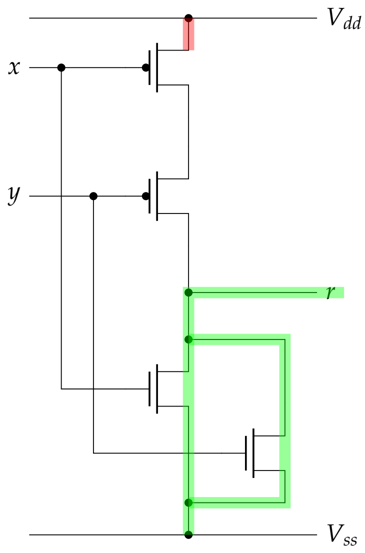
(b) *Annotation the case for* $x = V_{ss}, y = V_{ss}$.



(c) *Annotation the case for* $x = V_{dd}, y = V_{ss}$.



(d) *Annotation the case for* $x = V_{ss}, y = V_{dd}$.



(e) *Annotation the case for* $x = V_{dd}, y = V_{dd}$.

Figure 13: *A CMOS-based NOR gate implementation.*

x	y	NOT	$NAND$	NOR
V_{ss}	V_{ss}	V_{dd}	V_{dd}	V_{dd}
V_{ss}	V_{dd}	V_{dd}	V_{dd}	V_{ss}
V_{dd}	V_{ss}	V_{ss}	V_{dd}	V_{ss}
V_{dd}	V_{dd}	V_{ss}	V_{ss}	V_{ss}

Figure 14: *A voltage-oriented truth table for NOT, NAND, and NOR logic gates.*

An aside: naming conventions for voltage levels.

In a CMOS-based design strategy, we normally refer to the power rails as V_{dd} and V_{ss}. The 'd' stands for drain: V_{dd} could be read as "voltage level at the drain" st. it also makes sense to have V_{ss} read as "voltage level at the source". This naming convention seems to stems 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 V_{dd} and V_{ss} 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 or positive voltage level, e.g., 3.3V or 5V, and
- V_{ss} is the low or negative voltage level, e.g., 0V $\simeq GND$.

Note that GND refers to ground: this can be thought of as a) a reference point other voltages are measured relative to (note that voltage is a synonym for potential *difference*, meaning we need such a reference), or b) a (or the) return path, i.e., the point to which electrons will move due to their preference to move from high to low potential difference.

Example 0.6. Consider Figure 12, where

1. connecting both x and y to V_{ss} 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 V_{ss} 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 V_{ss} 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 V_{ss}.

Example 0.7. Consider Figure 13, where

1. connecting both x and y to V_{ss} 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 V_{ss} means the upper-most P-MOSFET will be disconnected, the left-most N-MOSFET will be connected, so r will be connected to V_{ss},
3. connecting x to V_{ss} 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 V_{ss}, 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 V_{ss}.

A second aspect of the design strategy is made evident by increasing the number of transistors. Specifically, the two examples include P-MOSFETs organised in *parallel* (st. *either* can be activated to connect V_{dd} to the output) and N-MOSFETs organised in *series* (st. *both* must be activated to connect V_{ss} to the output), or vice versa.

Hopefully it is obvious that the three examples model the NOT, NAND (or NOT AND) and NOR (or NOT OR) Boolean operators respectively; this fact is renforced by Table 14. Either way, the fact is that from a starting point involving atomic-level concepts we have developed components that we can reason about wrt. both theory *and* practice. That is, we have used electrical switches to implement Boolean algebra; instead of reasoning about computation involving the latter in theory, we can now actually build components that *do* that computation in practice.

1.2.5 Some common terminology in CMOS-based logic design

Definition 0.15. *The process used to manufacture organisations of transistors, plus their associated properties and constraints, is a* **logic style** *(or* **logic family***): examples include CMOS and TTL.*

Definition 0.16. *A given logic style will suggest an associated* **standard cell***, i.e., an organisation of transistors that realises a higher-level building block, namely either a a) computational component (e.g., a Boolean AND operator), or b) storage component (e.g., a latch); where the former is more naturally described as a* **logic gate***. Each such cell will have associated functional specification (i.e., a* **truth table** *or* **excitation table***), and behavioural specification (e.g., detailing propagation delay).*

Definition 0.17. *A* **standard cell library** *is a collection of standard cells, used as building-blocks in a design.*

Definition 0.18. *The* **standard cell methodology** *permits design abstraction, in the sense a design can be specified at a high- vs. low-level (i.e., in terms of standard cells, vs. transistors).*

Definition 0.19. *A* **Gate Equivalent (GE)** *is a unit of measurement used to assess the (area) complexity of a digital logic design* independently *from the manufacturing process technology. It is common (e.g., for CMOS) to consider a 2-input NAND gate as 1 GE: you can think about it as a normalisation factor for manufacturing processes, st. designs specified using* different *processes can be compared fairly.*

2 Combinatorial logic

2.1 A suite of simplified logic gates

It should already be clear that designing functionality, even as simple as single Boolean operators, is hard at the transistor-level: transistors are too low a level of abstraction, st. the amount of detail is prohibative at a larger scale or higher level. To address this problem, we usually adopt a more abstract view of logic gates by taking two steps: we 1) forget about the voltage levels V_{ss} and V_{dd}, abstractly labelling them 0 and 1, then 2) forget about the power rails, and just *draw* a symbol to represent each gate (with suitable inputs and outputs).

Figure 15 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 16. Keep the following in mind:

- We are assuming the voltage levels used to represent values on each wire are perfect in some sense. In short, we assume the associated signals have a "square" waveform and so are *digital* signals (i.e., only ever have a value of 0 or 1). In reality this can be dubious, because physical phenomena that underpin those voltage levels mean the edges of said signals might be "rounded" and so imperfect (e.g., have a value of 0.5 say); we basically ignore this issue, at least until later.
- 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 connects the input directly to the output; a NOT gate is then a buffer that inverts the input to form the output.
- For completeness we have included the NXOR (sometimes written XNOR) gate, which has the obvious meaning but is seldom used in practise; per Chapter **??**, 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

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

- Given 2-input gates such as AND, OR, and XOR, we 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

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

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.

2.2 Harnessing the universality of NAND and NOR

Following from the above, (at least) two questions should be immediately apparent:

1. Chapter **??** suggests NOT, AND, and OR are the operators to focus on, so *why* design NAND and NOR from transistors? and

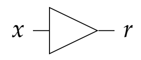
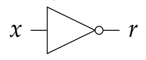
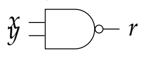
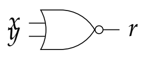
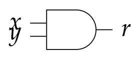
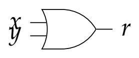
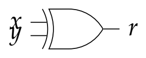
r is x	$\equiv$	$r = x$	$\equiv$	
r is NOT x	$\equiv$	$r = \neg x$	$\equiv$	
r is x NAND y	$\equiv$	$r = x \overline{\wedge} y$	$\equiv$	
r is x NOR y	$\equiv$	$r = x \overline{\vee} y$	$\equiv$	
r is x AND y	$\equiv$	$r = x \wedge y$	$\equiv$	
r is x OR y	$\equiv$	$r = x \vee y$	$\equiv$	
r is x XOR y	$\equiv$	$r = x \oplus y$	$\equiv$	

Figure 15: *Representation of standard logic gates in English, Boolean algebra, C and symbolic notations.*

x	r
0	0
1	1

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

x	r
0	1
1	0

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 16: *Truth tables for standard logic gates.*

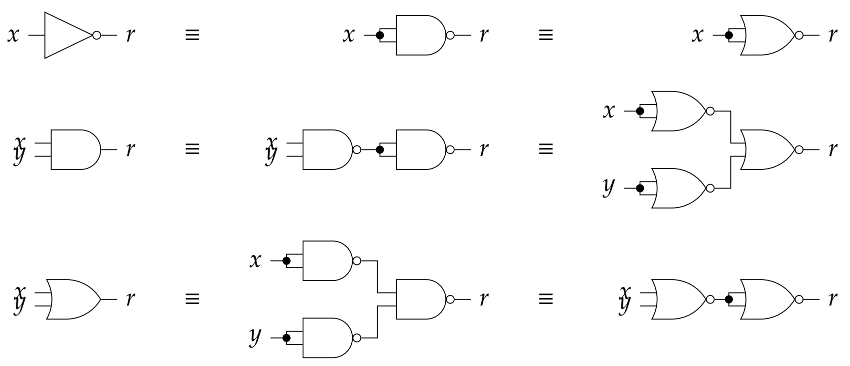


Figure 17: *Identities for standard logic gates in terms of NAND and NOR.*

2. given the design of NAND and NOR from transistors was an involved, detailed process, is there a way to avoid repeating this for AND and OR?

The answer to *both* questions stems from the functional completeness off NAND and NOR: they are *universal*, in the sense we can implement every *other* logic gate using one or other of them alone (as already discussed in Chapter **??**). The identities

$$\begin{array}{lclcl}
\neg x & & & \equiv & x \overline{\wedge} x \\
x \wedge y & & & \equiv & (x \overline{\wedge} y) \ \overline{\wedge} \ (x \overline{\wedge} y) \\
x \vee y & \equiv & \neg x \ \overline{\wedge} \ \neg y & \equiv & (x \overline{\wedge} x) \ \overline{\wedge} \ (y \overline{\wedge} y)
\end{array}$$

and

$$\begin{array}{lclcl}
\neg x & & & \equiv & x \overline{\vee} x \\
x \wedge y & \equiv & \neg x \ \overline{\vee} \ \neg y & \equiv & (x \overline{\vee} x) \ \overline{\vee} \ (y \overline{\vee} y) \\
x \vee y & & & \equiv & (x \overline{\vee} y) \ \overline{\vee} \ (x \overline{\vee} y)
\end{array}$$

replicated diagrammatically in Figure 17, demonstrate why; one can easily verify them via enumeration, e.g., in

x	y	$x \overline{\wedge} y$	$x \overline{\wedge} x$	$y \overline{\wedge} y$	$(x \overline{\wedge} y) \overline{\wedge} (x \overline{\wedge} y)$	$(x \overline{\wedge} x) \overline{\wedge} (y \overline{\wedge} y)$
0	0	1	1	1	0	0
0	1	1	1	0	0	1
1	0	1	0	1	0	1
1	1	0	0	0	1	1

and

x	y	$x \overline{\vee} y$	$x \overline{\vee} x$	$y \overline{\vee} y$	$(x \overline{\vee} x) \overline{\vee} (y \overline{\vee} y)$	$(x \overline{\vee} y) \overline{\vee} (x \overline{\vee} y)$
0	0	1	1	1	0	0
0	1	0	1	0	0	1
1	0	0	0	1	0	1
1	1	0	0	0	1	1

This is *enormously* important: it explains why designing NAND and NOR from transistors made sense in the first place, but, more over, it allows us to implement *any* Boolean expression, and so *any* Boolean function, from NAND and NOR gates alone. The manufacture of such implementations, which we cover in Section 5, will be vastly easier as a result. At the transistor-level, we only need deal with some (large) number of *one* building block (i.e., NAND or NOR) vs. the added complexity and effort associated with *many* such building blocks (i.e., AND, OR, XOR, and so on): *everything* at a low level is expressed in terms of NAND or NOR, so implemented by exactly the organisations of N- and P-MOSFETs we have already seen.

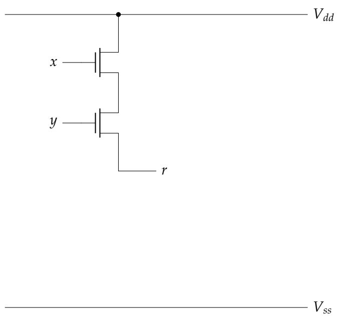
2.3 Designing circuits for arbitrary combinatorial functions

Now we have logic gates that act as physical implementations of each Boolean *operator*, the next challenge is how to produce Boolean *expressions* for some (arbitrary) Boolean *function*. Put another way, the challenge is to take a specification of a function f, e.g., a truth table, and derive a Boolean expression e which computes it.

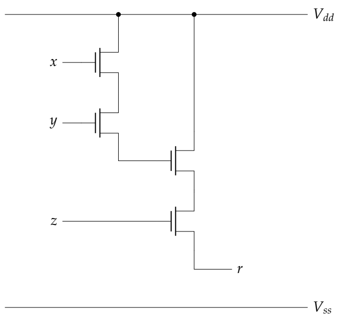
An aside: why NAND not AND?!

Arguments based on universality of NAND and NOR motivate a preference for these building blocks by a preference for *minimalism*: using a single building block to implement *every* other component will offer manufacturing advantages, for example, vs. a more diverse set.

That said, it is reasonable to question what *other* motivations exist. Put another way, what would happen if we *wanted* an AND design in similar, transistor-based terms? A common starting point for such questions is the following

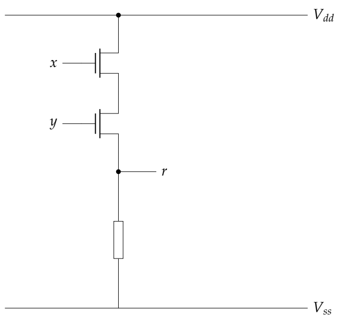


in which we only have a pull-up network. The reasoning is often that if $x = V_{dd}$ and $y = V_{dd}$ then $r = V_{dd}$ as required, whereas if $x = V_{ss}$ or $y = V_{ss}$ then r is disconnected; in defining what 0 and 1 mean, if we just define disconnected as 0 then maybe this design is valid? A counterargument (among many) is to think about what happens if we use r elsewhere as an input, e.g.,



Now, if r is disconnected, the top-most transistor in the second layer simply does not work: the gate terminal is disconnected from either V_{dd} or V_{ss} so the transistor cannot function.

It turns out there *is* a solution to this sort of issue, which is to opt for a pull-down *resistor* rather than *network* of transistors, i.e., something like



which you could think of as providing a “default” value to any disconnected wire. The problem is, now we have to reason about and manufacture another component (i.e., the resistor): both of these are out of scope, so, at least here, this approach is not viable.

Chapter **??** has provides a complete enough background that we can attempt to address this challenge in a mechanical, algorithmic manner; doing so contrasts with deriving or manipulating expressions by hand using the Boolean axioms. Several viable approaches and thus algorithms exist, which we investigate in the following Sections: each has advantages and disadvantages, and can be described as taking the description of f as input, and producing e in SoP form as output.

2.3.1 Introduction of don't care entries

An important feature or extension of truth tables, as defined so far, is the potential to include so-called **don't care** entries: rather than 0 nor 1, we use **?** to denote we do not *care* what the value is (vs. we do not *know* what the the value is, for example). When used in the context of an output, it can be rationalised by considering a component whose output simply does not matter given some combination of inputs: maybe this input is invalid, so the output is never used due to the resulting error.

Example 0.8. Consider the truth table

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

which describes some 2-input Boolean function, where don't care entries are used in two roles:

1. On the LHS, wrt. the input x. In this case, the **?** represents a wildcard (or short-hand) meaning 0 *and* 1. That is, by saying we don't care what the value of x is, we expand that one row into two: one for $x = 0$ and one for $x = 1$, which is like saying "irrespective of x (so if $x = 0$ *or* $x = 1$), provided $y = 0$ then $r = 1$".
2. On the RHS, wrt. the output r. In this case, the **?** represents a choice meaning 0 *or* 1. That is, by saying we don't care what the value of r is, we can choose whatever suits us.

This concept has various applications, but is immediately useful during the derivation of an expression from the specification (including don't care entries) of some function.

2.3.2 Some design patterns

Before dealing with *arbitrary* Boolean functions, it is useful to start with some *specific* examples that can be solved by using a **design pattern** (or template): although they may or may not apply to a particular problem, whenever they *do* apply they represent a pre-designed solution we can use as is without further effort.

We use a *specific* example to introduce each design pattern below: in each case, a 2-input, 1-bit AND gate is used to solve some sort of problem. It is crucial to remember that the example illustrates a more *general* pattern: we will see cases where this is true later.

1. If, within some larger design, we use an AND gate to compute
$$r = x \wedge y$$
and then, somewhere else, compute
$$r' = x \wedge y$$
we can replace the two AND gates with one: it is obvious that $r = r' = x \wedge y$, so the output of a single AND gate can be **shared** between the two usage points. This simplification is possible, but harder to capture within a single Boolean expression: using
$$r = (w \wedge x \wedge y) \vee (x \wedge y \wedge z)$$
as an example, it is usual to first define some intermediate, say
$$t = x \wedge y$$
then rewrite the expression as
$$r = (w \wedge t) \vee (t \wedge z).$$
Doing so acts as a direct analogue to sub-expression elimination, an optimisation commonly applied by C compilers to expressions in C programs.

2. A 2-input, m-bit AND gate can be realised using **independent replication** of "isolated" 2-input, 1-bit AND gates. That is, If x and y are m-bit values then

   $$r = x \wedge y$$

   is computed via

   $$r_i = x_i \wedge y_i$$

   for $0 \leq i < m$, i.e., m separate 2-input, 1-bit gates, each i-th instance of which uses x_i and y_i to produce the output r_i. For $n = 4$ this is the same as

   $$\begin{array}{lcl} r_0 & = & x_0 \wedge y_0 \\ r_1 & = & x_1 \wedge y_1 \\ r_2 & = & x_2 \wedge y_2 \\ r_3 & = & x_3 \wedge y_3 \end{array}$$

3. An n-input, 1-bit AND gate can be realised using **dependent replication** or "cascade" of 2-input, 1-bit AND gates. That is,

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

   is computed via

   $$r = x_0 \wedge (x_1 \wedge \cdots (x_{n-1})).$$

   For $n = 4$ this is the same as

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

   This expression forms a tree of AND gates, which, in this case is balanced; it is more attractive than equivalents such as

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

   because although they use the same number of gates, the critical path of the former is shorter (i.e., representing 2 rather than 3 such gates).

2.3.3 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 that describes it. The idea is to follow a (fairly) simple 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:
   (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.

Intuitively, each $i \in T$ will produce a minterm t_i in the SoP form: each term t_i ANDs inputs together (to form their product), whereas e ANDs together the terms (to form their sum). Each minterm fully specifies an input assignment (i.e., a value for each input) for a row of the truth table where the output is 1; in a sense, we are "covering" (or dealing with) each such row by doing so.

Example 0.9. Consider the task of implementing an expression for XOR, i.e., an e in SoP form which implements $f(x, y) = x \oplus y$, a truth table for which is reproduced (cf. Figure 16) here for clarity:

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 $\mathcal{O} = r$.

   Likewise, it is clear that $T = \{1, 2\}$ because $\mathcal{O} = 1$ in rows 1 and 2, whereas $\mathcal{O} = 0$ 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.

For example, notice that the row for $i = 1$ produces the minterm $t_1 = \neg x \wedge y$ meaning "the row where $x = 0$ *and* $y = 1$", whereas the row for $i = 2$ produces the minterm $t_2 = x \wedge \neg y$ meaning "the row where $x = 1$ *and* $y = 0$"; combining the minterms together, we get an SoP expression that specifies rows where the output should be 1 as "either $x = 0$ *and* $y = 1$, *or* $x = 1$ *and* $y = 0$".

2.3.4 Mechanical derivation method #2: Karnaugh maps

Example 0.10. Consider the truth table in Figure 18a which describes a 4-input Boolean function, and the SoP expression

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

resulting from application of the method above.

Although it only becomes apparent when you *do* so, deriving such an expression is tedious and error prone; although the algorithm is simple, it *could* be described as machine-friendly (in the sense it is best executed by a computer). The complexity of the expression, in the sense it contains many operators, is more obvious. Although we could simplify it by applying Boolean axioms, for example, this is again quite tedious. It is obvious to ask, therefore, whether (and if so, how) we can improve the original method wrt. these problems?

An aside: binary versus Gray code.

Consider a sequence of unsigned, n-bit integers; selecting $n = 4$, for example, and starting from zero, such a sequence would be

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

where the RHS describes a (decimal) value, and the LHS describes the (binary) representation of that value. Notice that moving from $\langle 1,1,0,0 \rangle$ to the next entry $\langle 0,0,1,0 \rangle$ means changing 3 bits: the 0-th and 1-st bits toggle from 1 to 0, and the 2-nd bit from 0 to 1. Now consider an alternative ordering of the same integers:

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

Now, moving from *any* entry to the next *or* the previous one will *always* toggle one bit: such an ordering is termed a **Gray code** after Frank Gray who made reference to it in a 1953 patent application (such orderings had been known and used for quite some time before that). Crucially,

1. we can produce an ordering that satisfies the same property for *any* n, and
2. the alternative ordering is just a permutation of the original: we keep the same values (and the same representations), but just rearrange them within the sequence.

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 18: *4- and 3-input example Boolean functions respectively.*

The **Karnaugh map** invented in 1953 by Maurice Karnaugh while working at Bell Labs [2], is an alternative method which offers (at least) two advantages over the original: it a) offers a more visual and so arguably human-friendly way to derive the resulting expression, and b) automatically applies various optimisations *while* doing so, st. we no longer need to apply (as much) post-derivation optimisation by hand. Although an example more usefully illustrates how to use a Karnaugh map, and so the advantages above, the method itself is best summarised using 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 rectangular $(p \times q)$-element grid, st.

   (a) $p \equiv q \equiv 0 \pmod 2$, and

   (b) $p \cdot q = 2^n$

   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.

Based on this description, the underlying reason it delivers the claimed (or in fact *any*) advantages is far from intuitive. However, there is a way to explain it: the central observation is that *if* we find two minterms st. their input assignment differs in exactly one input, we can simplify the resulting expression by eliminating that input. If you (re-)consider Figure 18a, the minterms associated with

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

and

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

i.e., rows of the truth table for $i = 10$ and $i = 11$, satisfy exactly this condition: they differ wrt. z, which is 0 in the first case and 1 in the second case. In the original method, we would implement them using *the* two minterms

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

and

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

However, this is overly pessimistic and so sub-optimal: the value of z is irrelevant provided $w = 1$, $x = 0$, and $y = 1$, because the output is 1 either way. As such, we eliminate z and use the LHS of

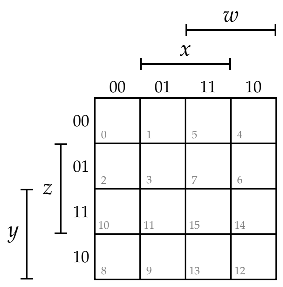
$$w \wedge \neg x \wedge y \quad \equiv \quad (w \wedge \neg x \wedge y \wedge \neg z) \vee (w \wedge \neg x \wedge y \wedge z)$$

which is equivalent to the RHS and therefore cover *both* cases via a single, simpler expression.

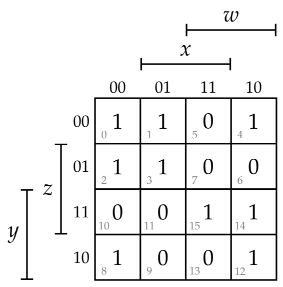
Example 0.11. The best way to illustrate this in practice is to fully examine the the truth table in Figure 18a:

1. Essentially, the first two steps just translate information from the truth table into the map (or grid); keep in mind that we are representing the *same* information, i.e., the specification of f, in both cases.

   Since f has $n = 4$ inputs, the associated truth table has $2^4 = 16$ rows; by selecting $p = q = 4$, we can draw the following square grid with enough elements to capture those rows

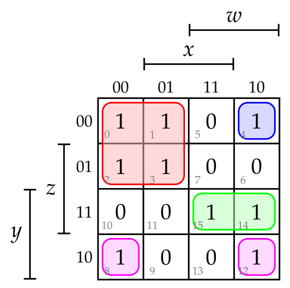
   Correctly interpreting the grid layout is crucial, since we need to translate rows of the truth table into the correct elements. Note that w and x relate to the columns (or horizontal axis), whereas y and z relate to the rows (or vertical axis). The left-most column, for example, relates to cases where w and x both have the values 0, i.e., where $(w, x) = (0, 0)$; reading that column top-to-bottom, the rows within it relates to cases where $(y, z) = (0, 0)$, $(0, 1)$, $(1, 1)$ and $(1, 0)$. The other columns, read from left-to-right, are similar for y and z, but for the remaining cases where $(w, x) = (0, 1)$, $(1, 1)$ and $(1, 0)$. As such, we can now fill each element in the grid with an output listed in the corresponding truth table row to get

   Bars above and to the left of *this* grid denote cases where the associated input is 1: the 1-st and 2-nd (or middle) columns are where $x = 1$, for example, whereas the 0-th and 3-rd (or outer) columns are where $x = 0$. Elsewhere you might also see numbers to the left of each row, or above each column to make the values more explicit: they might show $(0, 0)$ and $(1, 0)$ (or just $00_{(2)}$ and $01_{(2)}$) for the 1-st and 2-nd (or middle) columns, and $(0, 1)$ and $(1, 1)$ (or just $10_{(2)}$ and $11_{(2)}$) for the 0-th and 3-rd (or outer) columns. Either way, the ordering might, reasonably, seem odd: note that in row- and column-wise directions, a Gray code is used. 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 reason for this choice will be made apparent later, but, for now, keep in mind that it is what allows the Karnaugh map to deliver the advantages outlined above.

2. The next step is to cover 1 elements in the grid. In a sense, this is analogous to what we did in the original method when we identified each row in the truth table where the output was 1: there we would have a group for *each* 1 element, but here we can form larger groups and cover *multiple* 1 elements.

   The rules state we can form rectangular, potentially overlapping groups whose size is a power-of-two (i.e., 2^m for some m): provided we follow them, each group formed will represent a term we then need to implement as part of the SoP expression. The larger the group, the fewer inputs we be included in each of the terms; the fewer groups, the fewer terms there are. An example grouping in this case is as follows:



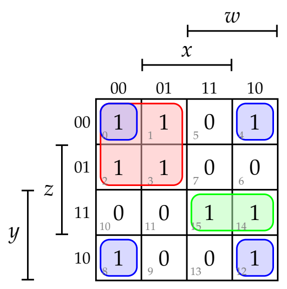
Here we have four groups:

- a group of four elements in the top left-hand corner spanning the 0-th and 1-st rows and columns,
- a group of one element in the top right-hand corner,
- a group of two elements in the 2-nd row spanning the 2-nd and 3-rd columns, and
- a group of two elements which wrap around the bottom-left and bottom-right corners.

3. Finally, we need to translate each group into a term in the SoP expression. As an example, consider the first group (i.e., of four elements in the top left-hand corner) and the values each input is assigned within it. It should become clear that the value of x is irrelevant provided that $w = 0$. Put another way, fixing $w = 0$ means we include the two left-most columns only (excluding the two right-most columns because they relate to cases where $w = 1$). In the same way, the value of z is irrelevant provided that $y = 0$.

   By specifying values for each relevant input and ignoring the *ir*relevant inputs, we can implement this term as
$$\neg w \wedge \neg y$$
   to cover all four cells in that group; we are specifying "the columns where $w = 0$ *and* rows where $y = 0$", which restricts us precisely to elements within the group. By applying similar reasoning to the other three groups, we find that
$$\begin{array}{rclcccccccccl} r & = & (& \neg w & & & \wedge & \neg y & & &) & \vee \\ & & (& w & \wedge & \neg x & \wedge & \neg y & \wedge & \neg z &) & \vee \\ & & (& & & \neg x & \wedge & y & \wedge & \neg z &) & \vee \\ & & (& w & & & \wedge & y & \wedge & z &) & \end{array}$$
   which is equivalent to but clearly simpler than the result we derived originally: there are a) fewer terms, *and* b) each term is the combination of fewer inputs.

Example 0.12. The result above *is* simpler than the original, but it turns out we can do better still by more careful formation of the groups. More specifically, we could consider the following alternative



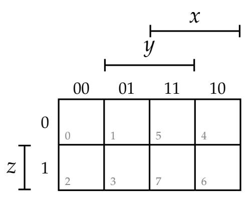
where there are now three groups, namely

- a group of four elements in the top left-hand corner spanning the 0-th and 1-st rows and columns,
- a group of two elements in the 2-nd row spanning the 2-nd and 3-rd columns, and
- a group of four elements which wrap around the top-left, bottom-left, top-right, *and* bottom-right corners.

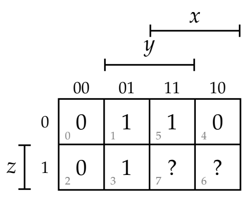
The end result is a simpler expression, including one less term:
$$\begin{array}{rclcccccccccl} r & = & (& \neg w & & & \wedge & \neg y & & &) & \vee \\ & & (& & & \neg x & & & \wedge & \neg z &) & \vee \\ & & (& w & & & \wedge & y & \wedge & z &) & . \end{array}$$

Constructive use of don't care entries Both the original method, and Karnaugh map alternative, can be described as aiming to cover 1 entries in the truth table (either individually, or in a group); in both cases, fewer 1 entries leads to a simpler SoP expression. As such, it makes sense to deal with don't care entries (in the output) in a way that helps: we are free to treat them as 0 *or* 1, so a) treating them as 0 means we do not need to cover them with a group, whereas b) treating them as 1 means we can potentially form larger groups.

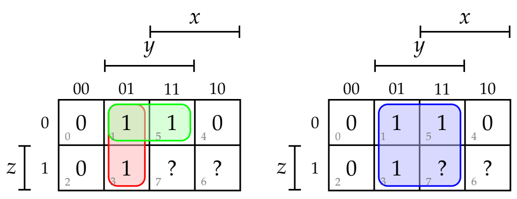
Example 0.13. Consider the truth table in Figure 18b which describes a 3-input Boolean function and thus has $2^3 = 8$ rows; selecting $p = 2$ and $q = 4$ yields the (empty) map



then filled as follows:



Consider the following two groupings:



The left-hand option treats the element associated with $x = 1$ and $z = 1$ 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 st. the resulting expression is

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

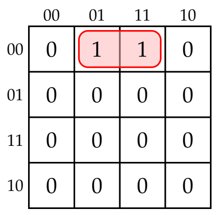
In contrast, the right-hand option treats the element as a 1, meaning it can be included in a single, larger group. This produces the (much) simpler expression $r = y$.

Why Gray code?! In the example above, we informally cited the use of Gray code ordering for rows and columns in a Karnaugh map as important wrt. the advantages it then offers. The easiest way to see *why* this is true, is via another example where we do *not* use this approach.

Example 0.14. Consider the truth table

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

which describing some 4-input function f. By *using* a Gray code ordering, we translate it into the following Karnaugh map

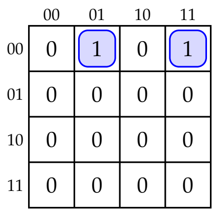


	00	01	11	10
00	0	1	1	0
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

that allows formation of a single group that covers the two elements in the 0-th column; this group produces the SoP expression

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

noting that the value of w is irrelevant in this case (i.e., provided $x = 0$, $y = 0$ and $z = 0$, that alone is enough to cover the group). Now consider a similar Karnaugh map *without* a Gray code ordering



	00	01	10	11
00	0	1	0	1
01	0	0	0	0
10	0	0	0	0
11	0	0	0	0

which is more like a **Veitch diagram** [10], a precursor to the Karnaugh map. Note, 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 $x = 1$: the 2-nd and 3-rd columns are swapped versus the original Karnaugh map (and likewise for the rows). The problem is, now we cannot make a single group that covers the same two elements: we now need two groups, each covering one element. These groups obviously produce a more complicated SoP expression, namely

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

where we now *include* w even though we know it is not required; to get the same result as before, we would now have to manipulate the expression by hand using suitable axiomatic steps.

This basically demonstrates that by using a Gray code ordering, where one bit will always toggle in the input assignment when moving between rows and/or columns, we support precisely the observation outlined at the start of the Section. Put another way, we wanted to identify input assignments that differed wrt. one input only so as to eliminate that input; by ensuring that two adjacent (including wrap-around) rows or columns satisfy this property, a group that spans them will naturally translate into a term that eliminates the single, different input that identifies them.

2.3.5 Mechanical derivation method #3: Quine-McCluskey

Although Karnaugh maps can represent functions with *any* number of inputs, they become unwieldy to draw and use for larger n; the reason for this scalability problem stems fundamentally from the emphasis on a human-friendly vs. machine-friendly algorithm. However, we can address the problem by investigating **Quine-McCluskey minimisation**: this is a method developed independently by Willard Quine [7] and Edward McCluskey [4] in the mid 1950s. It is reasonable to think of Quine-McCluskey as offering the advantages of *both* the previous methods: it a) can be automated easily, while *also* b) automatically applying various optimisations, and so avoids the need for (as much) post-derivation optimisation by hand. Unlike the previous methods where we could write a concise description of the algorithm, it is easiest to explain this one *inline* with an example. The following Sections do so by (re)considering the truth table in Figure 18a.

Step #1: extraction of prime implicants The first step is to produce a table, Table 19 for this example, that we extend step-by-step: we

1. initialise the 0-th section by extracting each minterm from the truth table (i.e., each input assignment st. the output is 1), then
2. process the i-th section to construct the $(i + 1)$-th section, iterating until no progress can be made.

Based on an input assignment represented as a tuple, in this case (z, y, x, w), we identify each minterm using an integer: you can see the seven mintems extracted from Figure 18a at the top of Table 19. In the table, each entry (i.e., each row) is called an **implicant**; they are assigned a group based on the number of elements in associated

Section	Group	Implicant		Used
0	0	0	$(0,0,0,0)$	✓
	1	1	$(1,0,0,0)$	✓
		2	$(0,1,0,0)$	✓
		4	$(0,0,1,0)$	✓
		8	$(0,0,0,1)$	✓
	2	5	$(1,0,1,0)$	✓
		10	$(0,1,0,1)$	✓
	3	11	$(1,1,0,1)$	✓
	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	$1+5$	$(1,0,?,0)$	✓
		$4+5$	$(?,0,1,0)$	✓
		$2+10$	$(0,1,0,?)$	✓
		$8+10$	$(0,?,0,1)$	✓
	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 19: *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 20: *Quine-McCluskey simplification, step #2: covering the prime implicants table.*

tuple that equal 1. Consider section 0 for example. Implicant 0 represented by $(0,0,0,0)$ (st. $w = 0$, $x = 0$, $y = 0$ and $z = 0$) is assigned group 0 because zero elements of the representation equal 1, In contrast, implicant 5 represented by $(1,0,1,0)$ (st. $w = 0$, $x = 1$, $y = 0$ and $z = 1$) and implicant 10 represented by $(0,1,0,1)$ (st. $w = 0$, $x = 1$, $y = 0$ and $z = 1$) are both assigned group 2 because two elements of their representations equal 1.

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 expression $w \wedge \neg x \wedge y$ because the value of z is irrelevant. We use a similar approach here, using the i-th section to construct the $(i+1)$-th section by comparing members of the j-th and $(j+1)$-th groups in the former; our goal is to find pairs of implicants whose representations differ in one element, and combine them together. We skip comparison of the j-th group and groups other than the $(j+1)$-th, because by definition they cannot satisfy the criterion. As an example, consider construction of 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
- compare implicant 11 from group 3 with implicant 15 from group 4.

In the new section, we replace the differing element of paired implicants with **?** to highlight the fact we don't care about that input: combining implicants 0 and 1 represented by the tuples $(0,0,0,0)$ and $(1,0,0,0)$, for example, produces an implicant represented by $(\mathbf{?},0,0,0)$. Furthermore, each implicant from the i-th section which is used to form an implicant in the $(i+1)$-th section is marked with a ✓ next to it; implicants 0, 1, 2, 4 and 8 are thus marked due to the 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, constructing subsequent sections until we can no longer make progress, i.e., there are no implicants that can be combined. Table 19 includes three sections, noting that section 2 has no implicants that be combined and so is the last constructed. In addition, it illustrates the fact combination of implicants in the i-th section *may* produce duplicates in the $(i+1)$-th section: here, we can see $(0,\mathbf{?},0,\mathbf{?})$ and $(\mathbf{?},0,\mathbf{?},0)$ are duplicated. Whenever this occurs, we ignore the duplicates and omit them from further comparisons.

Step #2: covering the prime implicants table Any unmarked implicants are termed **prime implicants**: these form the focus of a second step whose task is to produce the SoP expression. The content of Table 19 includes four prime implicants, namely

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

These are used to form a prime implicant table, as in Table 20: it lists the prime implicants along the left-hand side and the original minterms along the top, and includes a ✓ character in every elements where a given prime implicant includes a given minterm.

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; selecting this as well as implicant $10+11$ will cover 0, 1, 4, 5, 10 and 11. Before doing so, we can make our task easier by identifying the set of **essential prime implicants**, i.e., those which are the *only* cover for a given minterm. We can see the prime implicant $11+15$ is such a case in Table 20, because it is the only way to cover minterm 15; as a result, we *must* include it in our expression.

The process for coverage is fairly intuitive: we start with essential prime implicants, and then draw a line through the associated row in the prime implicants table; when a line goes through a ✓, we *also* draw a line through that column. The resulting lines show which minterms are currently covered by prime implicants we have selected for inclusion in our SoP 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 prime implicants $0+1+4+5$, $0+2+8+10$, and $11+15$, we can can cover all the original minterms; we need not include prime implicant $0+1+4+5$ for example, since minterms 0, 1, 4 and 5 are all covered elsewhere. Looking at the associated tuples, we have

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

Following the rule that for some input t

$$\begin{array}{lcc} \text{if } t=0 & \rightsquigarrow & \text{use } \neg t \\ \text{if } t=1 & \rightsquigarrow & \text{use } t \\ \text{if } t=\textbf{?} & \rightsquigarrow & \text{ignore} \end{array}$$

we form a term for each prime implicant listed and thus implement the SoP expression as

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

as per our original attempt using Karnaugh maps.

2.4 Physical properties of combinatorial logic

2.4.1 Delay: from static to dynamic (i.e., including time) evaluation

Definition 0.20. *Within some combinatorial logic, two classes of* **delay** *(which is often described as* **propagation delay***, with a hint toward delay of signals more generally) dictate the time between change to some input and corresponding change (if any) in an output: these are*

- **wire delay**, *which relates to the time taken for current to move through the conductive wire from one point to another, and*
- **gate delay**, *which relates to the time taken for transistors in each gate to switch between connected and unconnected states.*

The latter is typically larger than the former, and both relate to the associated implementations: the latter relates to properties of the transistors used, the former to properties of the wire (e.g., conductivity, length, and so on). x

Definition 0.21. *The* **critical path** *through some combinatorial logic is the longest sequential path between an input and output, i.e., the path which has the largest total delay (stemming from the individual wire and/or gate delays).*

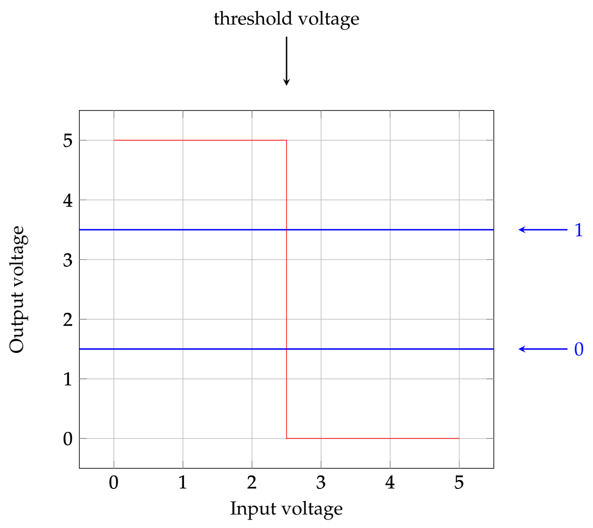
Although such wire and gate delays are typically very small, when many gates are placed in series or when wires are very long, the delays add up; the problem of managing the result is multiplied as the complexity of combinatorial logic increases. The concept of wire delay is perhaps more intuitive than gate delay, so it make sense to expand a little on the latter; the example below attempts to explain the cause.

Example 0.15. Consider Figure 21, which includes an idealised (left-hand side, in Figure 21a) and (more) realistic (right-hand side, in Figure 21b) illustration of what happens when the input to a MOSFET-based NOT gate, i.e., Figure 11, switches.

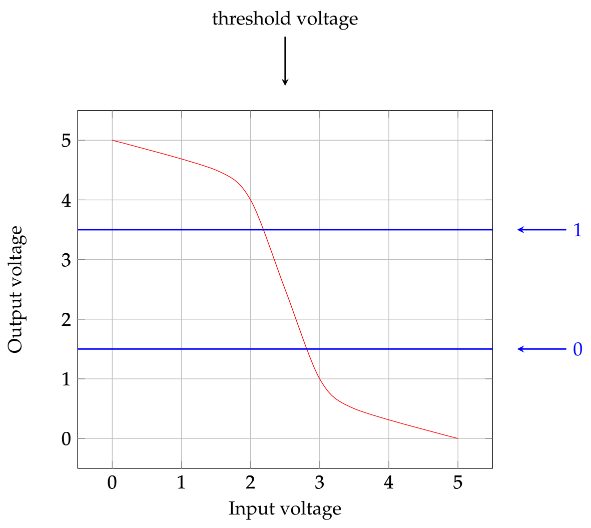
The idea is to stress the fact that in the idealised case, there is an instantaneous change in the output voltage: the plot representing the output is square-edged, changing (or swinging) from 5V (i.e., 1) to 0V (i.e., 0) the instant that the input voltage changes from 0V (i.e., 0) to 5V (i.e., 1), or, more precisely, when it reaches the threshold voltage. Note that the illustration includes output voltage levels *above* 0V and *below* 5V that represent the threshold at which said output is interpreted as a 0 or 1, but since the change is instantaneous these are irrelevant.

In contrast, the realistic case suggests a non-instantaneous change in the output voltage, i.e., it takes some time. The characteristics of the now curved plot relate to properties of the transistors. However, the important thing to realise is that the input voltage will take some time to change between 0V (i.e., 0) and 5V (i.e., 1), so there is some delay in the output voltage changing from 5V (i.e., 1) and 0V (i.e., 0); this *also* suggests there is a (short) period of time where the output voltage cannot be interpreted is either 0 or 1.

Although this property is often abstracted when illustrating the value of a wire in a waveform, meaning transitions from 0 to 1, or vice versa, are square-edged, it *can* be captured with sloped-edges as shown in

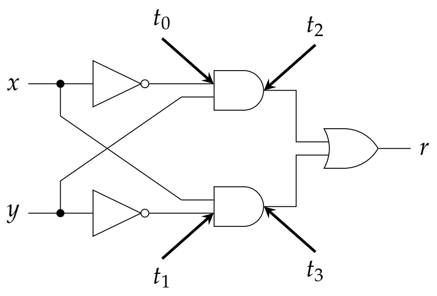


(a) *Idealised, square switching activity.*

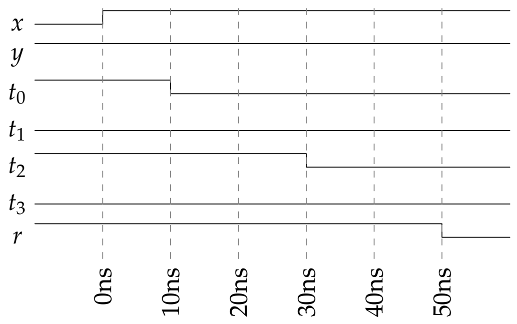


(b) *Realistic, curved switching activity.*

Figure 21: *An illustration of idealised and realistic switching activity wrt. a MOSFET-based NOT gate.*



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



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

Figure 22: *A behavioural waveform demonstrating the effects of propagation delay on an XOR implementation.*

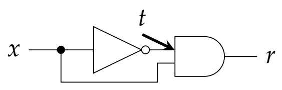


Figure 23: *A simple design, involving just a NOT and an AND gate, that exhibits glitch-like behaviour.*

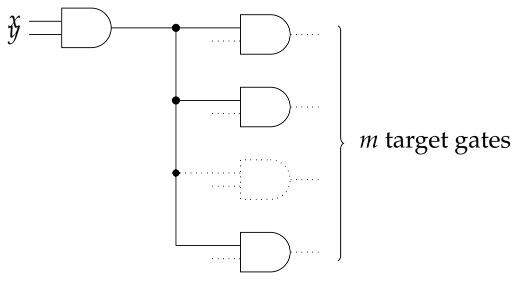
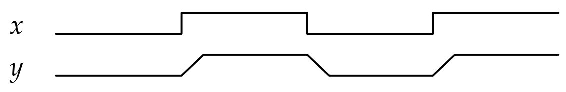


Figure 24: *A contrived circuit illustrating the idea of fan-out, whereby one source gate may need to drive n target gates.*



Notice that x and y toggle between 0 and 1 in the same way, but transitions in the former (resp. latter) are instantaneous (resp. take some time). Whether implicit or explicit, the gate delay property still exists, and has an impact on evaluation of larger combinatorial designs:

Example 0.16. Consider Figure 22a, which shows the implementation of an XOR gate (using, so derived from NOT, AND and OR gates). If we take a *static* approach to evaluating the output using the inputs, it is reasonable that by setting $x = 0$ and $y = 1$ we get

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

However, this ignores the impact of delay on the evaluation process; if we take a *dynamic* approach and imagine the delay of

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

this changes matters. Imagine we toggle the inputs from $x = 0$, $y = 1$ to $x = 1$, $y = 1$; immediately we introduce time, in the sense we have introduced previous values of x and y rather than just current values. An illustration of the gate behaviour is given in Figure 22b, however simplistic. The waveform starts when the gate is in the correct state given the inputs $x = 0$, $y = 1$, after which the inputs are toggled to $x = 1$, $y = 1$ (at 0ns). Notice that the the result is *not* valid immediately. In particular, we can examine points in the waveform and show that the final and intermediate results are actually incorrect. For example, it takes 10ns before either NOT gate produces the correct output on t_0 and t_1; the result r remains incorrect until 50ns; gate delay has caused a gap between the inputs being toggled, and output being valid.

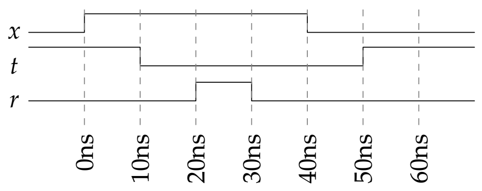
To conclude, it is important to stress the central role a critical path has: it is a limiting factor or bound on how quickly some combinatorial logic computes outputs, i.e., *it* dictates the associated latency. That may not *seem* important, but obviously we prefer an optimised design that has lower latency; this implies a design challenge, in that we almost *always* want to minimise the critical path.

Example 0.17. Following the example above, consider Figure 22a: this XOR design has a critical path that goes through a NOT gate, then an AND gate, and then an OR gate: the path has a total delay of 50ns. In a way, this formalises what we found above: it took 50ns to get the correct output r from inputs x and y. However, examining the critical path delivers this information with no evaluation; it basically tells us the design can *never* compute outputs in less time, which of course might imply the system said design is placed in is further limited as a result.

2.4.2 Glitches as a by-product of delay

Definition 0.22. *A* **glitch** *is normally defined to describe a (momentary) change wrt. some wire, which may cause a (momentarily) invalid or incorrect output if used as an input to some gate; the cause is typically delay of some sort, e.g., a mismatch in when two gate inputs become valid.*

Example 0.18. Consider Figure 23, wherein the two AND gate inputs are forced to be valid at *different* times due to imbalanced delay: it clearly takes longer for the value of x to propagate through the NOT gate than directly along the wire. The net result is that if we toggle $x = 0$ to $x = 1$ then back again, we produce a short glitch, i.e.,



matching the NOT gate delay.

2.4.3 On the sanity of buffer gates

Figure 16 included a so-called buffer gate, whose function can be described as $r = x$: no computation is performed per se, because the output matches the input. As such, it is reasonable to question the purpose of such a gate; we could eliminate it (or just replace it with a wire) and produce an equivalent result. It turns out the buffer can be used in two somewhat subtle roles:

1. Although the functionality of a buffer is $r = x$, there is still some associated gate delay (roughly equivalent to a NOT gate); it can thus be used to equalise the delay through different paths in some combinatorial logic, and thus help solve the glitch problem outlined above. Within Figure 22a, for example, one can imagine adding a buffer between y and the second input to the top AND gate; this would ensure that $\neg x$ and the buffered version of y arrive at the inputs to said gate at the same time.

2. Recall that the output of *each* MOSFET-based gate was formed by conditionally connecting V_{dd} or V_{ss} to r; the inputs, e.g., x and y, simply control which connection was made. This is important, because it implies that *even if* the inputs are in some way "weak" then the output will be amplified, so equal to the "strong" levels V_{dd} or V_{ss}. A buffer can therefore be viewed a way to get r, an identical but amplified version of x.

Neither of these fact is particularly important within the remit of what we cover, but is is nonetheless important to keep them in mind iff. you see buffer gates in designs elsewhere.

2.4.4 Fan-in and fan-out

The terms **fan-in** and **fan-out** refer to properties of logic gates associated with their inputs and outputs:

Definition 0.23. *Consider a given logic gate:*

- *The term* **fan-in** *is used to describe the number of inputs to a given gate.*
- *The term* **fan-out** *is used to describe the number of inputs (so in a rough sense the number of* other *gates) the output of a given gate is connected to.*

The former is easier to explain: it is just a way to formalise the fact that, wlog. a 2-input AND gate that computes $r = x \wedge y$ has fan-in of 2, whereas a 3-input AND gate that computes $r = x \wedge y \wedge z$ has fan-in of 3. A gate with higher fan-in will typically switch more slowly than a gate with lower fan-in; this stems from the fact the larger number of inputs are processed using a more complex internal organisation of transistors.

The latter is still easy to explain, but harder to justify as important. The idea is that, ideally, we are free to connect the output of a given source gate to the inputs of say m other target gates; in practice, however, there is a limit on m. It stems from increased load on the source gate, and so longer propagation delay: it basically takes longer for the driving voltage to meet the required threshold. In addition, a transistor is limited wrt. the current driven through it before it will malfunction in some way; if the fan-out requires this to be exceeded, then the under-supplied source gate will fail somehow. So, in a sense, fan-out is an intrinsic versus extrinsic implication of propagation delay (where the latter simply delays computation in some sense, the former disrupts it). For example, consider the contrived design in Figure 24: the source AND gate on the left-hand side is used to drive m other target AND gates to the right. Unless the source gate drives enough current onto its output, it may malfunction because the target gates will not receive enough of a share to operate correctly. The implementation of each gate 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.

2.4.5 3-state logic

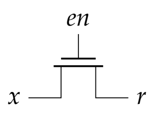
In a sense, fan-out constrains m, the number of target gates we might connect to $n = 1$ source gate. But what about n, and, in particular, what happens if we drive any number of target gates with the output from $n = 0$ source gates (i.e., none), or $n > 1$ source gates (e.g., by two rather than one)?

Suspend disbelieve for a moment and assume these cases *could* be of use in some way; hopefully it is obvious that neither is likely to yield the outcome we want, or indeed can reason precisely about. In the first case, the input is neither 0 or 1 so it is unclear what the output will be. Perhaps the only caveat to this is where one input along can dictate the output; reconsider Figure 12 for example, which implements a NAND gate and so computes $r = x \overline{\wedge} y$. The truth table for NAND suggests if $y = 0$ the $r = 1$ irrespective of x: this reasoning is validated by the implementation, since if $y = 0$ one P-MOSFET will always connect V_{dd} to r irrespective of the other. This aside, however, so in general, if an input is not a Boolean value then it remains unlikely we get the Boolean-like behaviour intended. In the same way, in the second case we basically join n outputs together: this is more dangerous, because *both* drive current along the wire. The outcome depends on a number of factors, but is, again, normally not a positive one wrt. the behaviour we want.

We can mitigate this issue by extending the idea of 2-state, Boolean logic values into 3**-state logic**. There are 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 a null, or disconnected value that can be safely "overpowered" by any other value (i.e., 0 or 1).

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 associated truth table accommodates the high impedance value as follows:

x	en	r
0	0	**Z**
1	0	**Z**
Z	0	**Z**
0	1	0
1	1	1
Z	1	**Z**
0	**Z**	**Z**
1	**Z**	**Z**
Z	**Z**	**Z**

In combination, these steps allow us to cope with both cases above. The first case is now less of an issue: we still might not get the behaviour we wanted, but at least we can reason about it. In the second case, we can use the enable gate to allow conditional access to a shared wire: if $en = 0$ the output is **Z** so not driven, meaning another driver could be safely connected to and use the same wire. However, when $en = 1$ the output is x; nothing else should be driving a value along this wire or we are back to the situation which caused the original problem.

2.4.6 Stable, unstable, and meta-stable states

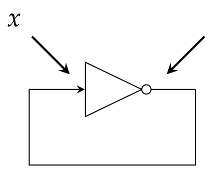
Definition 0.24. *Consider a component with a given output: the output (or component) can be said to be in*

- *a* **stable** *state if the output is* predictable, *i.e., either be* 0 *or* 1, *whereas*
- *an* **unstable** *state if the output is* unpredictable, *e.g., either be* 0, 1, *a voltage level between the threshold for either, or oscilate between the two somehow.*

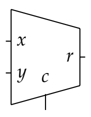
Definition 0.25. *A* **meta-stable** *state is an unstable state, which, after some period of time, will resolve to* some *stable state: the output eventually settles to either a* 0 *or* 1 *(i.e., become stable), but we cannot predict which* or *when.*

Instances of instability *typically* stem from some form of logical inconsistency in a design, and, in the case of meta-stability, are only ever resolved due to physical characteristics of the implementation (e.g., strength of transistors).

Example 0.19. Consider the following example



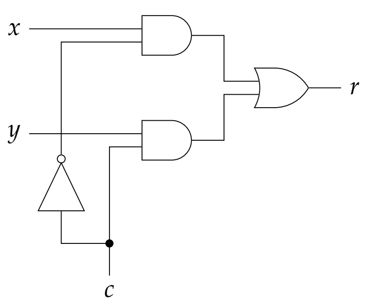
which could be captured using the (logically inconsistent) expression $x = r = \neg x$. Clearly there is a problem, because of $x = 0$ it should be 1 due to the NOT gate, and if $x = 1$ it should be 0; as a result, the output r will be unstable and oscillate somehow (potentially at a rate that is related to the gate delay involved).



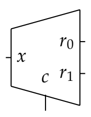
(a) *The multiplexer as a symbol.*

c	x	y	r
0	0	**?**	0
0	1	**?**	1
1	**?**	0	0
1	**?**	1	1

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



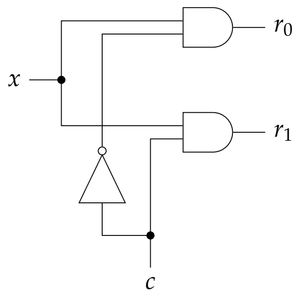
(c) *The multiplexer as a circuit.*



(d) *The demultiplexer as a symbol.*

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

2.5 Building block components

We have already seen it is convenient to design combinatorial logic using logic gates rather than transistors; in short, this allows a higher level of abstraction. In the same way, it *may* be convenient to design larger, more complex combinatorial logic components using smaller, less complex combinatorial logic components. The latter are, in a sense, just standard building blocks that are useful when designing the former. Where appropriate, they allow us to decompose a larger component into smaller components; this is often attractive, in that designing the larger component within one, monolithic task is often a lot more difficult.

Without a context, it is easy to look at the building blocks we cover in the following and deem them odd or even useless. Keep in mind that each one is covered specifically because it *is* useful; think of them as a way to practice the techniques developed so far, and believe we will make use of them later (e.g., in Chapter **??**).

2.5.1 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**

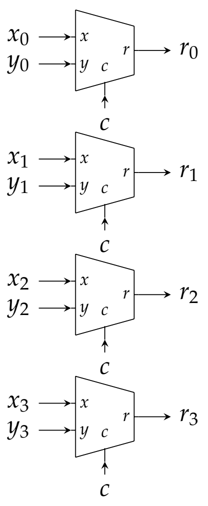
   - has m inputs,
   - has 1 output,
   - uses a $(\lceil \log_2(m) \rceil)$-bit control signal input to choose which input is connected to the output,

   while

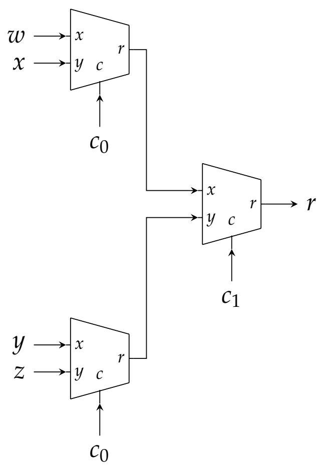
2. a **demultiplexer**

   - has 1 input,
   - has m outputs,
   - uses a $(\lceil \log_2(m) \rceil)$-bit control signal input to choose which output is connected to the input,

noting that each the input and output is n-bit. We can describe how the components behave using C as an analogy. For example, ignoring the number of bits in each input, output and control signal, the statement



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



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

Figure 26: *Application of the independent and dependent replication design patterns.*

```
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 the control signal `c`, one of the inputs (i.e., `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;
}
```

acts similarly to a 4-output demultiplexer: depending on the control signal `c`, one of the outputs (i.e., `r0`, `r1`, `r2`, or `r3`) is assigned from the input (i.e., `x`). Although attractive, using such an analogy needs care. In particular, keep in mind the C fragments include an implicit, discrete order wrt. the assignments. In contrast, the component design means an analogous connection is evaluated in a continuous manner: *whenever* either the control signal *or* any input changes, the output may change to match.

This behaviour stems from a design based on combinatorial logic, which is easy to develop for both components; in a similar way to before, we write down a truth table that describes the behaviour we require, then derive a Boolean expression to implement that behaviour:

Example 0.20. Consider the case of a 2-input (resp. 2-output), 1-bit multiplexer, a truth table for which is outlined in Figure 25b. The idea is we have two 1-bit inputs x and y, and one 1-bit control signal c; we want to drive r with either x or y depending on whether $c = 0$ or $c = 1$. The truth table should make sense in that when $c = 0$ the output r matches x, and when $c = 1$ the output r matches y; the don't care entries, and so truth table as a whole, can be read as "if $c = 0$ then $r = x$ irrespective of y, whereas if $c = 1$ then $r = y$ irrespective of x". From the truth table, we can arrive at the expression

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

which is shown diagrammatically in Figure 25c.

Example 0.21. Consider the case of a 2-input (resp. 2-output), 1-bit demultiplexer, a truth table for which is outlined in Figure 25e. The idea is we have two 1-bit outputs r_0 and r_1, and one 1-bit control signal c; we want to drive either r_0 or r_1 with x depending on whether $c = 0$ or $c = 1$. The truth table should make sense in that when $c = 0$ the output r_0 matches x, and when $c = 1$ the output r_1 matches x; the don't care entries, and so truth table as a whole, can be read as "if $c = 0$ then $r_0 = x$ and r_1 is irrelevant, whereas if $c = 1$ then $r_1 = y$ and r_0 is irrelevant". From the truth table, we can derive the expression

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

shown diagrammatically in Figure 25f.

For more general m-input (resp. m-output), n-bit alternatives, we employ the design patterns outlined earlier using the 2-input (resp. 2-output), 1-bit components as a starting point.

Example 0.22. Consider the task of designing a 2-input, n-bit multipliexer, wlog. taking $n = 4$ as an example. Note that with $m = 2$ inputs, we need $\lceil \log_2(m) \rceil = 1$ control signals: one of $2^1 = 2$ possible input assignments is used to select each input.

Figure 26a illustrates the design, which uses replication. The idea is simple: we use n separate 2-input, 1-bit multiplexers where the i-th instance accepts the i-th bit of each input x and y and produces the i-th bit of the output r. Or, put another way, since each instance is controlled by the same c, they are *all* either selecting some bit of x or of y to produce r.

Example 0.23. Consider the task of designing a m-input, 1-bit multipliexer, wlog. taking $m = 4$ as an example. Note that with $m = 4$ inputs, we need $\lceil \log_2(m) \rceil = 2$ control signals: one of $2^2 = 4$ possible input assignments is used to select each input.

One strategy would be to simply write down a larger truth table, i.e.,

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

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

This yields a reasonable result, but as the number of inputs grows the task becomes more difficult. An alternative is to divide-and-conquer, using 2-input, 1-bit multiplexers to decompose the larger decision task into smaller steps. Figure 26b illustrates the design, which uses a cascade. The first, left-most layer of multipliexers is controlled by c_0: the top-most instance produces w if $c_0 = 0$, or x if $c_0 = 1$, whereas the bottom-most instance produces y if $c_0 = 0$, or z if $c_0 = 1$. These outputs are fed into a second, right-most layer that uses c_1 to select appropriately: if $c_1 = 0$ the output of the top-most multiplexer in the first layer is selected, whereas if $c_1 = 1$ the output of the bottom-most multiplexer in the first layer is selected. The overall result r is the same as our dedicated design above, but hopefully it is clear the cascaded design is conceptually a *lot* simpler.

2.5.2 Components for doing basic arithmetic

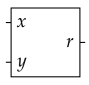
Chapter **??** addressed the challenge of representing numbers, integers for example, as n-bit binary sequences; a question left open was how we might do arithmetic with those numbers, or, more precisely, how we might do computation with the associated representations. Since we are now able to design arbitrary Boolean functionality, we can start to investigate this question.

The general, high-level task is to design a large, more complex combinatorial logic component that implements some arithmetic operation (e.g., integer addition): it might accept n-bit inputs $\hat{x}$ and $\hat{y}$ that represent x and y, and produce an n-bit result $\hat{r}$ st.

$$\hat{r} = f(\hat{x}, \hat{y}) \mapsto x + y,$$

i.e., an $\hat{r}$ that represents the sum of $\hat{x}$ and $\hat{y}$. The content of Chapter **??** does exactly this. As a means of support, however, a more specific, lower-level first step considers a set of less complex 1-bit building block components: although not so useful alone, they will act as building blocks within the more general alternatives.

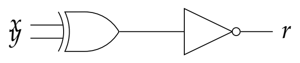
Comparators In contrast to arithmetic proper, where we expect both inputs *and* output to be numbers, a comparison *compares* numerical inputs thus produces a Boolean output. Various types of comparison are useful, but it is enough to consider two in particular: the others are derived from these **comparators**, that deal with 1-bit inputs.



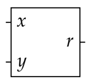
(a) *The equality comparator as a symbol.*

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

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



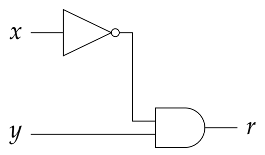
(c) *The equality comparator as a circuit.*



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

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 27: *An overview of equality and less than comparators.*

Example 0.24. Given 1-input inputs x and y, an **equality comparator** computes

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

From the associated truth tables is shown in Figure 27b, we can derive the expression

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

Example 0.25. Given 1-input inputs x and y, a **less than comparator** computes

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

From the associated truth tables is shown in Figure 27e, we can derive the expression

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

While fairly self explanatory, the truth tables may seem a little odd as a result of their dealing with 1-bit inputs. However, reading through them row-wise should demonstrate their content is sane: using less than as an example, consider than the truth table mirrors your intuition wrt. this comparison by stating 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. Note that the equality comparator design hints that an *in*equality comparator can be simpler still: inverting the expression, we find $r = x \oplus y$ provides an inequality comparison

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

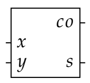
because, by definition, when $x = y$ (i.e., $x = 0$ and $y = 0$ or $x = 1$ and $y = 1$) $x \oplus y = 0$ and when $x \neq y$ (i.e., $x = 0$ and $y = 0$ or $x = 1$ and $y = 1$) $x \oplus y = 1$.

Adders The simplest arithmetic operation, conceptually at least, is addition. There are two variants of a 1-bit adder, instances of which will be sufficient to construct larger, n-bit adders later:

Example 0.26. Given 1-bit inputs x and y, a **half-adder** component computes a 1-bit sum s and carry-out co (i.e., the LSB and MSB of the 2-bit sum $x + y + ci$), as output. The corresponding truth table shown in Figure 28b can be used to derive associated Boolean expressions

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

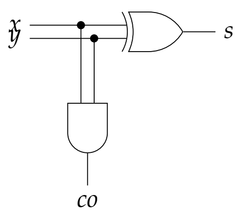
illustrated in Figure 28c.



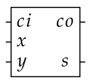
(a) *The half-adder as a symbol.*

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.*



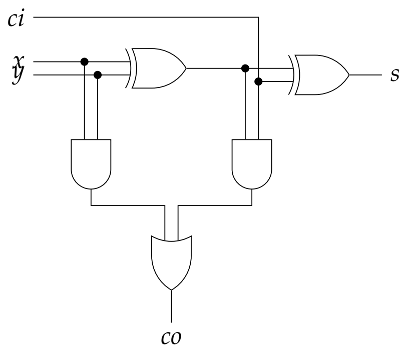
(c) *The half-adder as a circuit.*



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

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

Example 0.27. Given 1-bit inputs x and y and a carry-in ci, a **full-adder** component computes a 1-bit sum s and carry-out co (i.e., the LSB and MSB of the 2-bit sum $x + y + ci$), as output. The corresponding truth table shown in Figure 28e can be used to derive associated Boolean expressions

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

illustrated in Figure 30d.

As was the case with comparators, the truth tables may seem a little odd as a result of their dealing with 1-bit inputs; again, reading through them row-wise should demonstrate their content is sane. The half-adder, for example, is st.

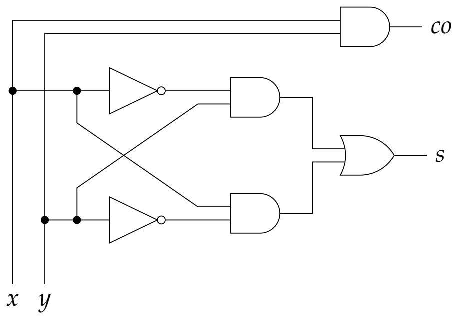
- if we add $x = 0$ to $y = 0$, the sum is 0 and there is no carry-out,
- if we add $x = 0$ to $y = 1$, the sum is 1 and there is no carry-out,
- if we add $x = 1$ to $y = 0$, the sum is 1 and there is no carry-out, and finally
- if we add $x = 1$ to $y = 1$, the sum is 2: since we cannot represent 2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

Note that the full-adder design is essentially two half-adders joined in a cascade: to accommodate the extra carry-in the first instance computes $t = x + y$ with the second one then computing $s = t + ci$. Also, note that the Boolean expressions listed for a full-adder effectively include two (equivalent) options for co. One reason to prefer the second is that given we need to compute both co *and* s, it contains the shared term $x \oplus y$ which can be capitalised on during optimisation.

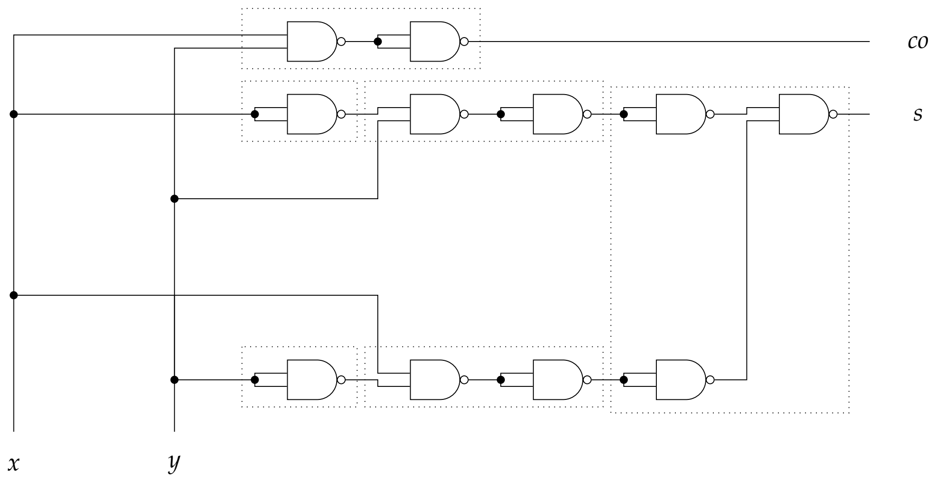
As an aside, the half-adder represents a simple enough design to explore the idea of gate universality in (a little) more detail:

Example 0.28. Given the natural half-adder implementation in Figure 29a as a starting point, Figure 29b describes an equivalent using NAND gates only.

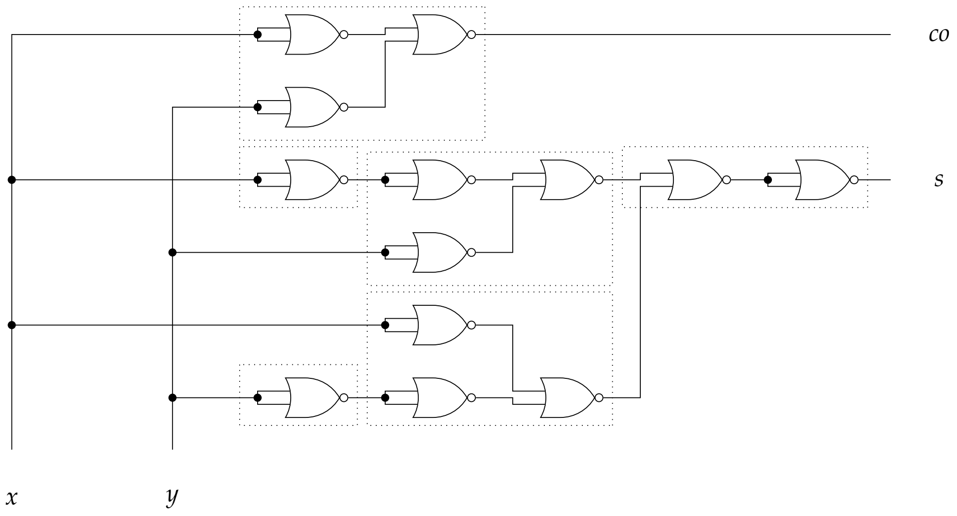
Example 0.29. Given the natural half-adder implementation in Figure 29a as a starting point, Figure 29c describes an equivalent using NOR gates only.



(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 29: *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).*

Focusing on the NAND-based variant, for example, naive translation using identities (annotated using dashed boxes, wrt. the original gates in the natural implementation) yields an implementation with 11 NAND gates. As you might expect, we can improve this with some careful optimisation: capitalising on the equivalence

$$x \overline{\wedge} (y \overline{\wedge} y) \;\equiv\; x \overline{\wedge} (x \overline{\wedge} y),$$

we can write

$$\begin{array}{rcll}
s & = & x \oplus y & \\
 & = & (\neg x \wedge y) \vee (x \wedge \neg y) & \text{(XOR identity)} \\
 & = & \neg(\neg x \wedge y) \overline{\wedge} \neg(x \wedge \neg y) & \text{(OR into NAND identity)} \\
 & = & (\neg\neg x \vee \neg y) \overline{\wedge} (\neg x \vee \neg\neg y) & \text{(de Morgans)} \\
 & = & (x \vee \neg y) \overline{\wedge} (\neg x \vee y) & \text{(involution)} \\
 & = & (\neg x \overline{\wedge} \neg\neg y) \overline{\wedge} (\neg\neg x \overline{\wedge} \neg y) & \text{(OR into NAND identity)} \\
 & = & (\neg x \overline{\wedge} y) \overline{\wedge} (x \overline{\wedge} \neg y) & \text{(involution)} \\
 & = & ((x \overline{\wedge} x) \overline{\wedge} y) \overline{\wedge} (x \overline{\wedge} (y \overline{\wedge} y)) & \text{(NOT into NAND identity)} \\
 & = & ((x \overline{\wedge} y) \overline{\wedge} y) \overline{\wedge} (x \overline{\wedge} (x \overline{\wedge} y)) &
\end{array}$$

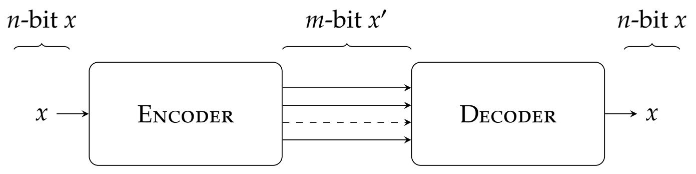
which uses 4 NAND gate due to the shared term $x \overline{\wedge} y$, which is also shared with

$$\begin{array}{rcl}
co & = & x \wedge y \\
 & = & (x \overline{\wedge} y) \overline{\wedge} (x \overline{\wedge} y)
\end{array}$$

meaning 5 NAND gates for the whole half-adder (which is roughly the same number of *non*-NAND gates within the natural implementation). There is a more direct ways to manipulate the expression for s, but notice that in the above a) steps 1 to 5 yield a result equivalent to Figure 29b, b) steps 6 to 7 eliminate any (obviously) redundant NOT gates, and c) step 8 reorganises the gates to maximise shared logic (rather than eliminating any gates outright). Although this is a specific example, these steps demonstrate a general strategy that often has a counter-intuitive impact on any given design: correctly optimised, using NAND (or NOR) often yields a lower (if any) increase in gate count vs. your expectation or an initial translation. Put another way, although they can be harder to work with, they do not *imply* a less efficient result wrt. area (while also retaining advantages such as regularity).

2.5.3 Components that translate between representations

Informally at least, **encoder** and **decoder** components can be viewed as *translators*. Consider the communication between two parties (or components) as an example



where the encoder accepts the input x, and encodes it into an x' then transmitted; the decoder receives x', and decodes it so as to recover the original x. Phrased as such, both encoder and decoder are basically translating between representations because x' could be thought of as a different representation of x we normally term a **code word**.

Definition 0.26. *Modelling an encoder and decoder as two functions*

$$\begin{array}{rcl}
\textsc{Encoder} & : & \{0,1\}^n \rightarrow \{0,1\}^m \\
\textsc{Decoder} & : & \{0,1\}^m \rightarrow \{0,1\}^n
\end{array}$$

we use the term n**-to-**m *to describe either component where it has n inputs and m outputs:*

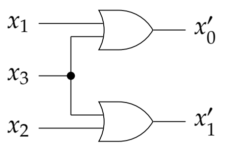
1. *an n-to-m encoder translates an n-bit input value into an m-bit code word, and*
2. *an m-to-n decoder translates an m-bit code word into an n-bit output value.*

Definition 0.27. *If for every code word x we have $HW(x) = 1$, i.e., every possible code word has exactly one bit set to 1, we call the associated encoder (resp. decoder)* **one-hot** *(or* **one-of-many***).*

Definition 0.28. *A* **priority encoder** *is st. priority (or preference) is given to one input over another. This concept is most obviously useful in a one-hot encoder, allowing it to cope gracefully with erroneous situations where $HW(x) > 1$: the idea is that if $x_i = 1$ and $x_j = 1$, then priority is given to x_i say (meaning the fact $x_j = 1$ is ignored).*

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

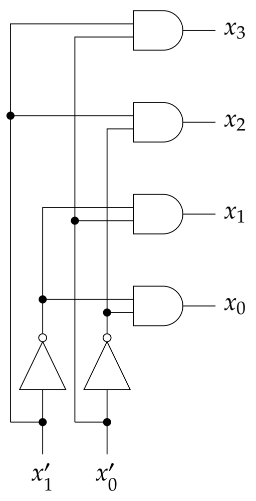
(a) *The encoder as a truth table.*



(b) *The encoder as a circuit.*

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 30: *An example encoder/decode pair.*

These formalisms hide various subtleties, most notably the fact that it only makes sense to discuss encoder and decoders in context: both a) the encoding (resp. decoding) scheme and so structure of code words, and b) parameterisation of said scheme (e.g., n and m), are totally domain-specific, meaning we cannot describe a *general* encoder (resp. decoder) in a sensible manner.

- The n-to-m terminology suggests inputs (resp. outputs) drawn from sets of 2^n (resp. 2^m) values. However, it is clearly possible, and often useful for some code words to remain unused; as such, it can be useful to relax the terminology this think of n-value and m-value sets instead. Note there is no strict requirement that $m > n$, or vice versa.

- Normally we need to consider the encoder and decoder together, as their behaviour is related: we normally expect
$$(\textsc{Decode} \circ \textsc{Encode})(x) = x,$$
i.e., $\textsc{Decode} = \textsc{Encode}^{-1}$. This fact implies that it is not *always* possible to describe a valid decoder (resp. encoder) for a given encoder (resp. decoder): some functions have no inverse. That said, however, some contexts do not *need* a decoder: the problem at hand may be st. the code word is useful as is, and the corresponding x need not be recovered.

Example 0.30. Consider the task of 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 an encoder that takes x and produces some x' as a result; the task of taking x' and recovering each x_k for $0 \leq k < n$ demands a corresponding decoder. This problem might be motivated by a need to control components: if we have n such components in a system, the decoder could, for instance, be used to enable one of them at a time.

By setting $n = 4$ for example, the encoder (resp. decoder) will have four inputs x_0, x_1, x_2, and x_3; this implies $x' \in \{0, 1, 2, 3\}$ and hence $m = 2$, meaning two outputs x'_0 and x'_1. Figure 30a and Figure 30c show truth tables for the two components. For the encoder, we derive the Boolean expressions

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

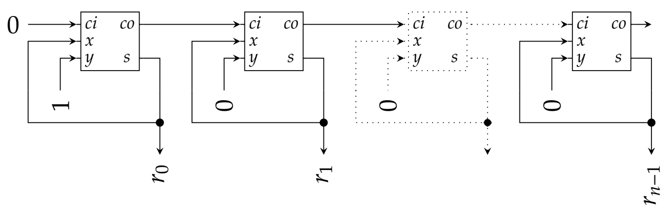


Figure 31: *An incorrect counter design, using naive "looped" feedback.*

and for the decoder

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

Example 0.31. 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

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

as the code word (incorrectly suggesting that $x_3 = 1$). To address problems of this sort, we can employ a priority encoder, giving x_2 priority over x_1 for example (or, more generally, every x_i has priority over x_j for $i > j$). To capture this requirement, we rewrite the truth table as follows:

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, provided that $x_2 = 1$ and $x_3 = 0$ (i.e., irrespective of x_0 and x_1) the output will be st. $x'_0 = 0$ and $x'_1 = 1$. The associated Boolean expressions are updated accordingly to

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

3 Sequential logic

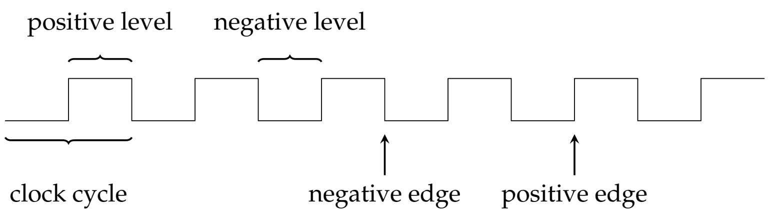
Imagine we are tasked with designing an n-bit **counter**, i.e., a component whose n-bit output r steps through values $0, 1, \ldots, 2^n - 1$ and then cycles (i.e., starts from 0 again). Recall that we have a 1-bit full-adder component; Chapter **??** later explains how to extend this into an adder that can compute $x + y$ for n-bit x and y, the basic idea being to organise n full-adder instances in a cascade where the carry-out of each i-th instance connects to the the carry-in of the next, $(i + 1)$-th instance. A natural attempt at the counter design might therefore use such an adder, computing $r \leftarrow r + 1$ as Figure 31 illustrated in: we essentially 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 (because there is a loop, from x through the full-adder to r and so back to x).

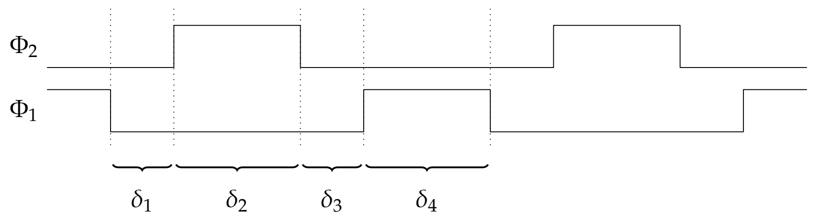
So despite the fact it intuitively functions as required, this design is far from ideal and, in fact, invalid. Perhaps the only use it has is to illustrate some fundamental limitations of combinatorial logic. More specifically, we cannot control *when* a component computes an output (since it does so continuously), nor have it *remember* said output once produced. We need a different approach, which along with components used to support it, is termed **sequential logic**: we need

- some way to control (e.g., synchronise) components,
- one or more components that remember what state they are in, and
- a mechanism to perform computation as a sequence of steps, rather than continuously,

which are addressed step-by-step in the following Sections.



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



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

Figure 32: *An illustration of standard features in* 1*- and* 2*-phase clocks.*

3.1 Clocks

If we want to perform computation as a sequence of steps, we need to exert control over the components involved: for example, it could be important to synchronise each component st. they all start (or stop) computation at the same time. We use a special control signal to do this:

Definition 0.29. *A* **clock signal** *is simply a digital signal that oscillates between* 0 *and* 1 *in a regular fashion.*

Note that despite the terminology, in the context of digital logic a clock is somewhat analogous to a metronome: rather than tracking the (wall-clock) time, for example, it simply produces a regular series of "ticks" (or features) that are used for synchronise associated actions.

Clock features Since a clock signal is a digital signal, it shares features such as positive and negative edges and levels as previously outlined within Chapter **??** and now by Figure 32a. That said, however, several specific features are also important:

Definition 0.30. *The interval between a given positive (resp. negative) edge and the next positive (resp. negative) edge is termed a* **clock cycle***. Additional terms you commonly encounter stem from this definition: for example, the* **clock period** *is the time taken for a clock cycle to complete, while the* **clock frequency** *(or* **clock rate***) is the number of clock cycles completed in a unit of time (typically each second, and hence the inverse of the clock period).*

Definition 0.31. *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.*

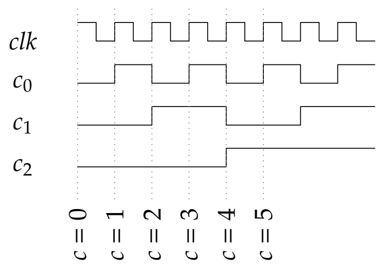
These features are harnessed by a **clocking strategy**, which is a formal way of saying "a way of *using* the clock signal". For example, we might use a clock edge to trigger the start some computation, or a clock level to enable or disable (e.g., reset) some computation.

Clock generation In a sense, *any* signal could be deemed a clock signal provided it satisfies the definition(s) above. However, in practice there is set of distinguished clock signals generated by a) an *external* or b) an *internal* **clock source** (or **clock generator**) component.

Example 0.32. An external clock source is commonly provided using a piezoelectric crystal. When a voltage is applied across such a component, it will oscillate 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 a clock signal.

Definition 0.32. *It can be useful to manipulate a given clock signal, in order to alter it wrt. features such as frequency; this is common whenever the clock signal is provided as an input to a design, but the design has specific internal requirements. In this context, the original and manipulated cases are sometimes termed the* **reference clock signal** *and* **derived clock signal***.*

Increasing the frequency of, i.e., multiplying, a reference clock is possible but somewhat beyond our scope; dedicated designs exist to solve this problem, but we omit a detailed overview. *Decreasing* the frequency of, i.e., dividing, a reference clock is much easier. Imagine that each positive edge of the reference clock clk causes a counter c to be incremented: assuming $c = 0$ initially, the individual bits of c can be visualised as



Notice that each successive bit of c models clk, but with a period that is twice as long: formally, the $(i-1)$-th bit of the counter c acts like clk *divided* by 2^i. This means, for example, that if $i = 1$ we can extract a clock signal with $\frac{1}{2^i} = \frac{1}{2^1} = \frac{1}{2}$ times (i.e., half) the frequency via the 0-th bit of c.

Clock distribution

Definition 0.33. *As with the power rails, a given clock signal must be* **distributed** *(or supplied) to each component that makes use of it; a* **clock distribution network** *is tasked with doing so.*

Definition 0.34. *The term* **clock skew** *describes a phenomena whereby a clock signal arrives at one component along a different path to another, and so at a different time; this suggests the two components are unsynchronised.*

Example 0.33. Example clock distribution network topologies include the **H-tree**, which is a form of space filling curve. The advantage of a H-tree is that wire delay from the clock generator to each target component, is uniform: this helps minimise the potential for clock skew.

Definition 0.35. *The term* **clock domain** *defines the influence of control exerted by a specific clock signal; every component in a given clock domain is controlled by the same clock signal.*

It is hard(er) to reason about the relationship between the features in *different* clock signals that imply *different* clock domains. This means, for example, that a) synchronising, and/or b) communicating values between components in two, *different* clock domains is harder than if the same components are in the one, *single* clock domain: intuitively, for example, it is hard to tell when positive edges on said clocks may occur at the same time (and so synchronise the components, say). As a result, points of interaction between (i.e., at the boundary of) clock domains (e.g., so-called **clock domain crossings**) demand careful attention.

From 1-phase to n-phase clocks Although it is easiest to think of a single clock signal, as illustrated in Figure 32a, 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[4] 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 32b 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.

[4]Based admittedly on limited experience, it *seems* that relatively few textbooks cover *both* 1- and 2-phase clocking strategies: in some ways this is a pity, since the use of 2-phase clocks is certainly simpler given the requirement for latches rather than flip-flops. If you want an alternative overview, then [11, Section 5] offers an example.

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).

Definition 0.36. *A stateful component 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 0.37. *A given bistable component controlled by an enable signal en 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 termed a **latch***, whereas the latter type is 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 since the state (which matches the output) is being updated with the input. This is not the case with flip-flops, because their state is only updated at the exact instant of an edge.

Definition 0.38. *The behaviour of a bistable is described by an* **excitation table***, and sometimes expressed using a* **characteristic equation***: versus, e.g., a truth table, the idea is to capture the notion of time (cf. current and next).*

Definition 0.39. *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 $\neg en$ to denote the latter case.*

Definition 0.40. *It is common for a given latch or flip-flop design to include additional control signals; an important example is a* **reset signal** *rst, that is often included to allow (re)initialisation of a design.*

Definition 0.41. *When used as a verb rather than a noun (cf. logic gate),* **gate** *means to conditionally turn off some component or feature.*

Example 0.34. Consider a component whose 1-bit input x is gated by AND'ing it with a control signal g: the input provided to the component is

$$x' = g \wedge x.$$

We say g gates x because if $g = 0$ then $x' = g \wedge x = 0 \wedge x = 0$: whatever the value of x, the component gets $x' = 0$ as input if $g = 0$, hence x has been "turned off" by g. In contrast, if $g = 1$ then $x' = g \wedge x = 1 \wedge x = x$: the component gets $x' = x$ as normal if $g = 1$.

Our description of such components has so far been very abstract; the goal in what follows is to remedy this situation. First, we describe the high-level design *and* behaviour of some latch and flip-flip components. Then we show how this behaviour can be realised, using a lower-level design expressed in terms of logic gates. In combination, we focus specifically on the goal of developing an n-bit register based on D-type latches and/or flip-flops.

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:

Definition 0.42. *An "SR" latch/flip-flop component has two inputs S (or* **set***) and R (or* **reset***):*

- *when enabled, if*
  - *$S = 0$ and $R = 0$, the component retains Q,*
  - *$S = 1$ and $R = 0$, the component updates to $Q = 1$,*
  - *$S = 0$ and $R = 1$, the component updates to $Q = 0$,*

  - *$S = 1$ and $R = 1$, the component is meta-stable*

  but

- *when not enabled, the component is in storage mode so retains Q.*

The behaviour of such a component is specified by

$$Q' = S \vee (\neg R \wedge Q),$$

and/or

		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	?	?	?	?

Definition 0.43. *A "D" latch/flip-flop component has one input D:*

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

  but

- *when not enabled, the component is in storage mode so retains Q.*

The behaviour of such a component is specified by

$$Q' = D,$$

and/or

	Current		Next	
D	Q	$\neg Q$	Q'	$\neg Q'$
0	?	?	0	1
1	?	?	1	0

Definition 0.44. *A "JK" latch/flip-flop component has two inputs J (or* **set***) and K (or* **reset***):*

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

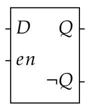
  but

- *when not enabled, the component is in storage mode so retains Q.*

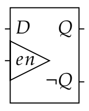
The behaviour of such a component is specified by

$$Q' = (J \wedge \neg Q) \vee (\neg K \wedge Q),$$

and/or



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



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

Figure 33: *Symbolic descriptions of D-type latch and flip-flop components (note the triangle annotation around en).*

		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

Definition 0.45. *A "T" latch/flip-flop component has one input T:*

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

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

The behaviour of such a component is specified by

$$Q' = (T \wedge \neg Q) \vee (\neg T \wedge Q) = T \oplus Q,$$

and/or

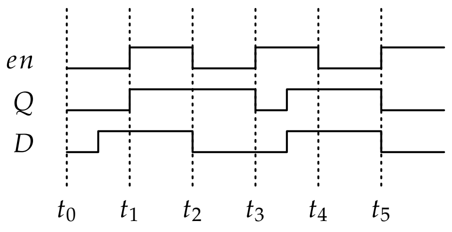
	Current		Next	
T	Q	$\neg Q$	Q'	$\neg Q'$
0	0	1	0	1
0	1	0	1	0
1	0	1	1	0
1	1	0	0	1

It is useful to look in more detail at the D-type component, since this will help explain the basic concepts. The component has one input D (in addition to the enable signal en), and two outputs, Q and $\neg Q$; we can ignore $\neg Q$ usually, but note that it should *always* be the inverse of Q.

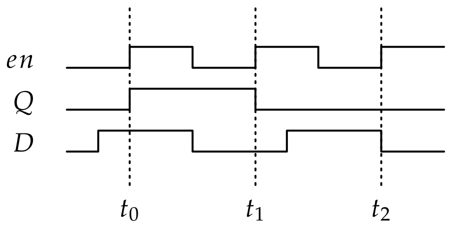
The table-based that describes the behaviour is split into two halves, which is unlike what we have seen previously. Put simply the left-hand half is a description of the current state, whereas the right-hand a description of the next state, i.e., *after* we perform an update; this distinguishes what is a "time-aware" *excitation* table from previous a a "time-agnostic" *truth* table used previously. So, for example, 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 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 33 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, and commonly omit any unused outputs (e.g., $\neg Q$).

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 0.46. *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:*

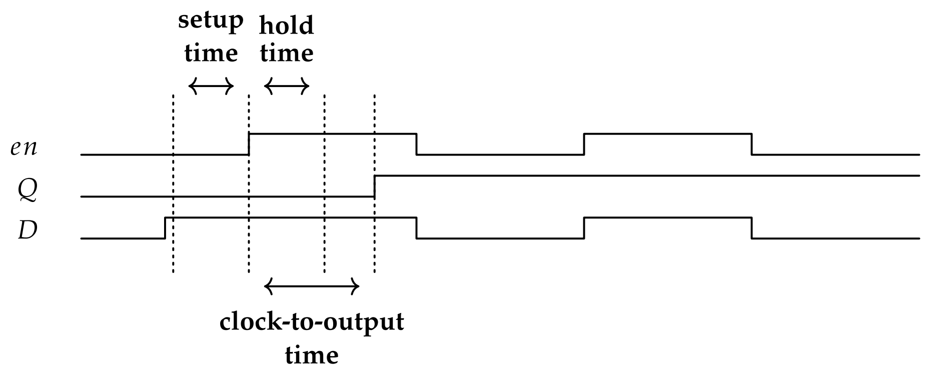
1. *The* **setup time** *(resp.* **hold time***) is the minimum period of time that D must be stable before (resp. after) use to update the component.*

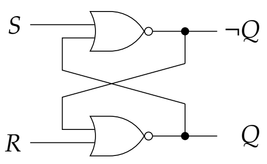
   Think of the clock feature (either level or edge) as triggering the act of sampling from D in order to update the state. As such, the two timing restrictions mentioned make sure the sample is reliable: they specify a window, around the change to en, where D has to be stable for some period of time.

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.*

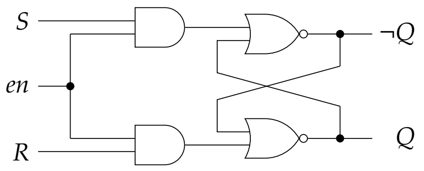
These time periods or delays will be determined by the implementation of the component; ideally they will be minimised, which makes the component easier to use (i.e., more tolerant).

Example 0.35. The concepts of setup, hold, and clock-to-output time are illustrated in the following (intentionally exaggerated) waveform relating to a D-type (edge triggered) flip-flop:

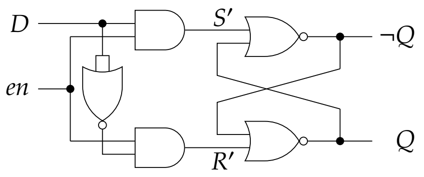




(a) *A NOR-based SR latch.*

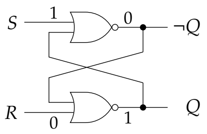


(b) *A NOR-based SR latch with enable signal.*

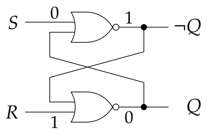


(c) *A NOR-based SR latch with enable signal and* $R = \neg S$*.*

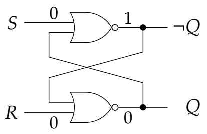
Figure 34: *NOR-based SR type latches, with simpler (top) to more complicated (middle and bottom) control features.*



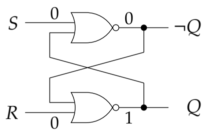
(a) *A case for* $S = 1$*,* $R = 0$*.*



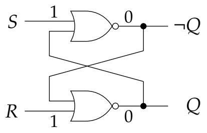
(b) *A case for* $S = 0$*,* $R = 1$*.*



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



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



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

Figure 35: *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.

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.

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 34a 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 is wired to the input of the other, an arrangement we say is **cross-coupled**.

Understanding the behaviour of the design as a whole depends on a property of the NOR gates. Recall (e.g., from Figure 16) that we can describe NOR using a truth table as follows:

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

In particular, this illustrate the fact that if *either* $x = 1$ *or* $y = 1$ then the result *must* be $r = x \overline{\vee} y = 0$. Put another way, we can write two axioms

$$\begin{array}{rcl} x \overline{\vee} 1 & = & 0 \\ 1 \overline{\vee} y & = & 0 \end{array}$$

These are important, because they allow us to resolve the loop introduced by the cross-coupled nature of NOR gates in this design. We can see how, on a case-by-case basis, by observing output for each possible assignment: this is shown in Figure 35.

- if $S = 1, R = 0$ (Figure 35a) then we *force* $Q = 1, \neg Q = 0$ (irrespective of what they were previously) because the top NOR gate *must* output 0 because we know $1 \overline{\vee} y = 0$,
- if $S = 0, R = 1$ (Figure 35b) then we *force* $Q = 0, \neg Q = 1$ (irrespective of what they were previously) because the bottom NOR gate *must* output 0 because we know $x \overline{\vee} 1 = 0$,
- if $S = 0, R = 0$ then the outputs are not uniquely defined by the inputs: there are in fact *two* logically consistent possibilities (Figure 35c and Figure 35d), namely $Q = 1, \neg Q = 0$ *or* $Q = 0, \neg Q = 1$,
- if $S = 1, R = 1$ (Figure 35e) then we *force* $Q = 0, \neg Q = 0$: in a sense this is contradictory, because we expect each to be the inverse of the other, but hints at another problem.

In the final case, the latch could be (and we have) described as being in a meta-stable state because the *eventual* output is not predictable. An intuitive reading is that it makes no sense to both set *and* reset the value, so *some* form of unexpected behaviour for $S = R = 1$ is therefore not unreasonable. More specifically though, once we return to $S = 0, R = 0$ the latch *must* settle in one or other of the two possibilities outlined above: we cannot predict which one, however, so the subsequent state of the latch is essentially random.

Note that in terms of the specified behaviour, the design does what we want. For example, we can set or reset the current state (per Figure 35a and Figure 35b) or retain the current state (per Figure 35c and Figure 35d) as need be. However, this high-level description avoids two perfectly reasonable questions, namely

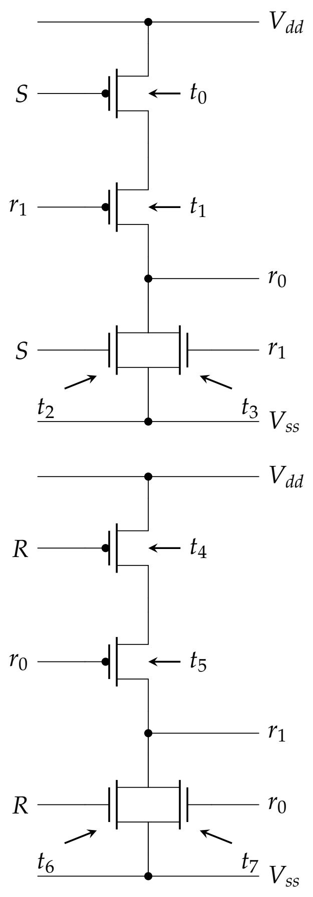


Figure 36: *An annotated SR latch, decomposed into two NOR gates and then into transistors; r_0, the output of the top NOR gate, is used as input by the bottom NOR gate and r_1, the output from the bottom NOR gate, is used as input by the top NOR gate (although the physical connections are not drawn).*

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 $S = R = 0$.

Up to a point, it is reasonable to consider that *if* it the latch settles into one of the two logically consistent states, there is just no motivation for it to subsequently change into the other; therefore, it will retains the same state. To provide greater detail, however, we rely on Figure 36. 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 clear 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

- 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 V_{ss}.

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 by adding an extra input en and two AND gates: the internal latch inputs become
$$\begin{array}{rcl} S' & = & S \wedge en \\ R' & = & R \wedge en \end{array}$$
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 34b 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 **??** 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.

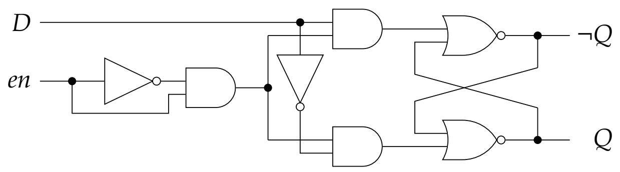


Figure 37: *A NOR-based D-type flip-flop created using a glitch generator.*

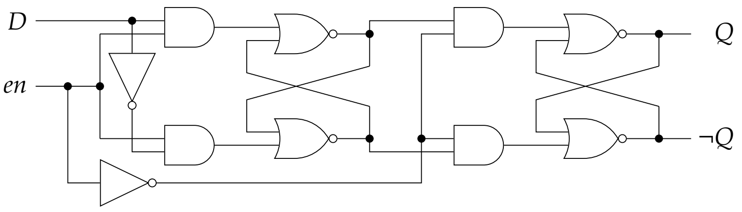


Figure 38: *A NOR-based D-type flip-flop created using a primary-secondary organisation of latches.*

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

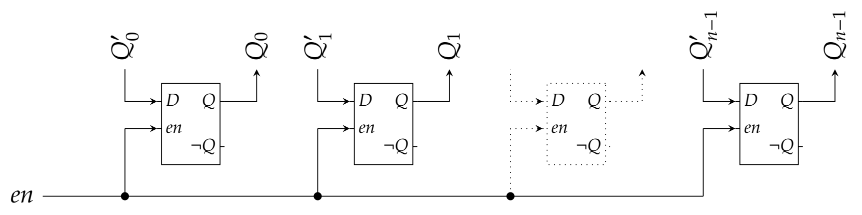
Example 0.36. Reconsider Figure 23, whereby a glitch is is generated (in that case unintentionally) for a (short) period of time when $en = 1$ *and* $t = 1$. We can drive the original D-type latch using such a design: Figure 37 illustrates the result, which now approximates a flip-flop due to the approximation of edge-based triggering.

Using a primary-secondary organisation An alternative approach is a **primary-secondary**[5] organisation of *two* latches, which yields the edge-triggered behaviour we want. Figure 38 shows the result, which is basically one latch (the primary, on the left) in series with a second latch (the secondary, on the right). The idea is to split a clock cycle into to half-cycles such that

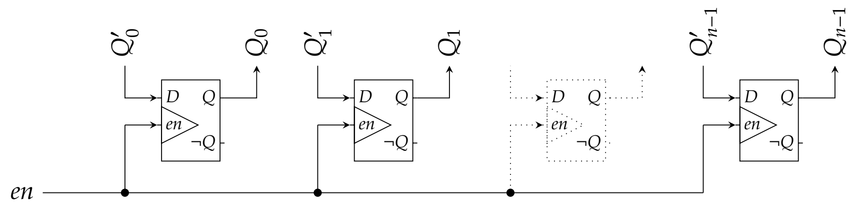
1. while $en = 1$, i.e., during the first half-cycle, the primary latch is enabled,
2. while $en = 0$, i.e., during the second half-cycle, the secondary latch is enabled.

In practical terms, this means while $en = 1$, i.e., during a positive level on en, the primary latch stores the input. Then, the instant $en = 0$, i.e., at a negative edge on en, the secondary latch stores the output of the primary latch: you can think of it as triggering a transfer from the primary to the secondary latch, or as the secondary latch only being sensitive to the output of the primary latch rather than the input. The fact that the transfer is instantaneous, in the sense it occurs as the result of an (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.

[5]Historically, the terms master and slave have often been used in place of primary and secondary. Per [3, Section 1.1], however, and despite some debate, the former are typically viewed as inappropriate now. We deliberately use the latter, therefore, noting that doing so may imply a need to translate the former when aligning with other literature.



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



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

Figure 39: *An n-bit register, with n replicated 1-bit components synchronised using the same enable signal.*

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 39 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.

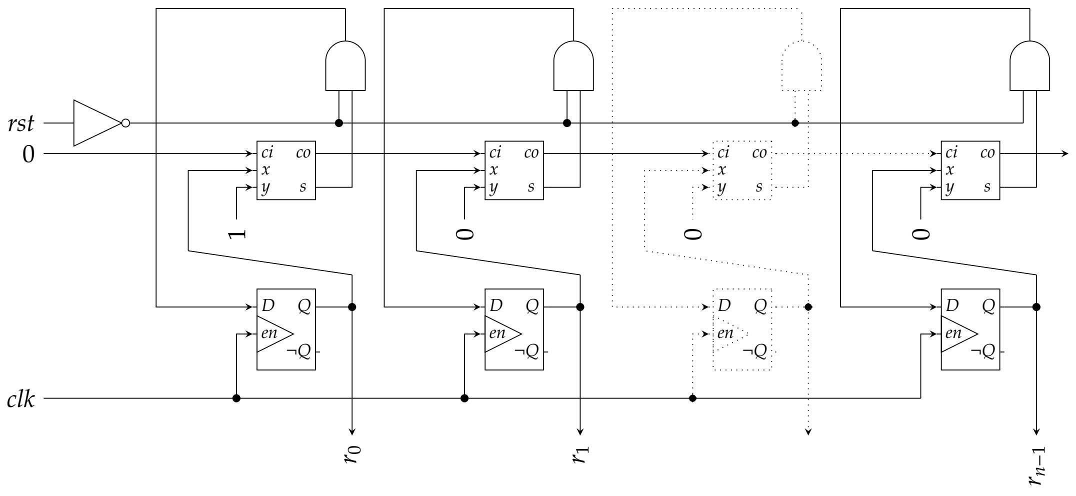
Example 0.37. Figure 40a represents a solution based on use of flip-flops, which implies a 1-phase clocking strategy. The top-half of the Figure shows an n-bit ripple-carry adder; the idea is that it computes $r' \leftarrow r + 1$. This part is roughly the same as the initial, faulty solution. The bottom-half of the Figure shows an n-bit, edge-triggered 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 the 1-bit sum produced as output by each full-adder is AND'ed with $\neg rst$. This acts as a limited reset mechanism, in the sense rst gates the output register input (resp. adder output): if $rst = 1$ (so $\neg rst = 0$) then the register input will always be zero, whereas if $rst = 0$ (so $\neg rst = 1$) then the register input will match the adder output. Put another way, if $rst = 1$ then the value subsequently latched by the input flip-flop is *forced* to be zero: this is important, because when powered-on the current value will be undefined and hence unusable.

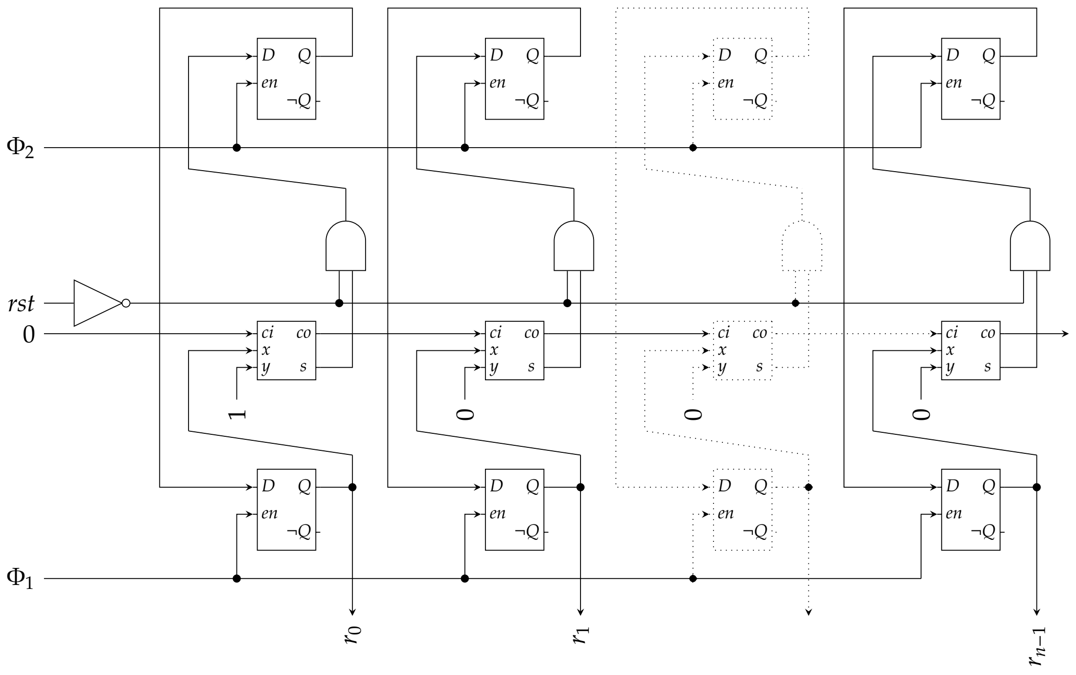
2. Notice that each D-type flip-flop in the register is synchronised by clk (which we assume is a clock): positive edges on clk provoke them to update the stored value r with $r' \leftarrow r + 1$.

   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.

Example 0.38. Figure 40a 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

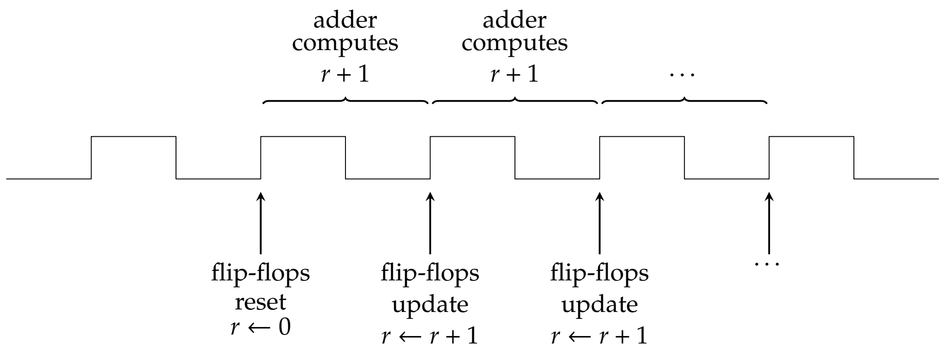


(a) *Using a* 1*-phase clock and flip-flop based register(s).*



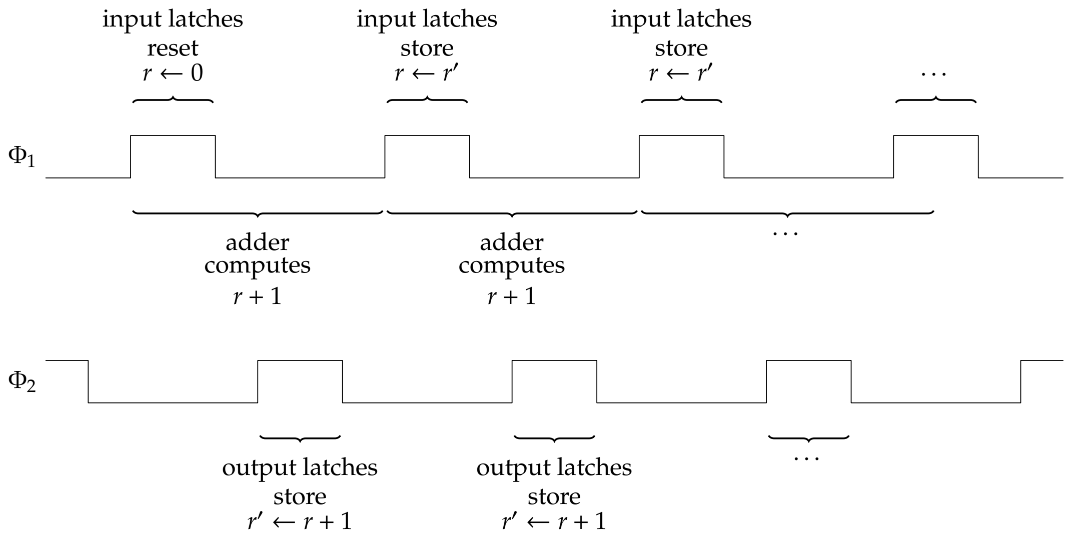
(b) *Using a* 2*-phase clock and latch based register(s).*

Figure 40: *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 $r \leftarrow r + 1$
...

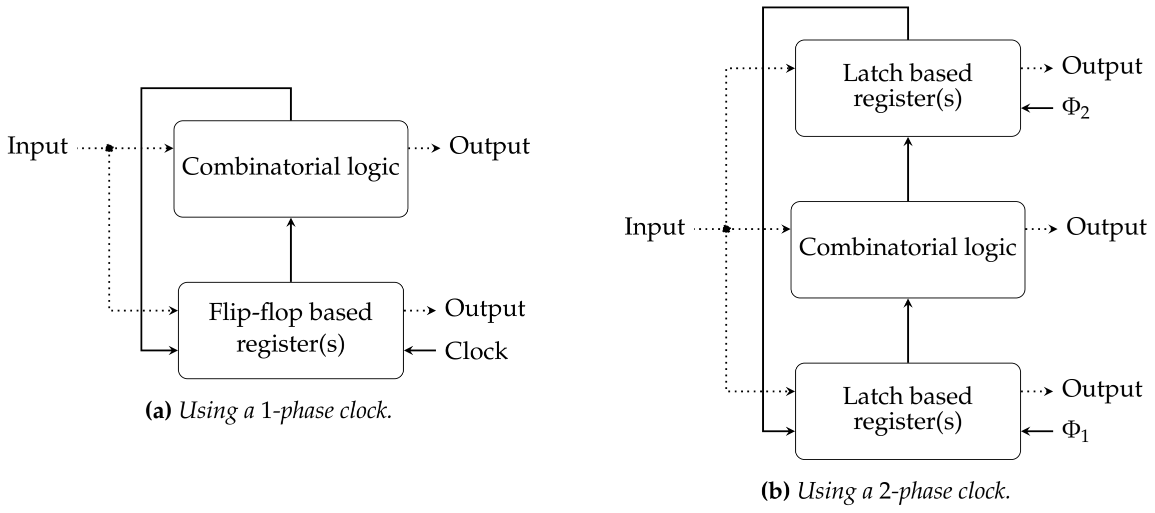
(a) *Using a 1-phase clock and flip-flop based register(s).*



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 $r' \leftarrow r + 1$
...

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

Figure 41: *Two illustrative waveforms, outlining stages of computation within the associated counter design.*



(a) *Using a 1-phase clock.*
(b) *Using a 2-phase clock.*

Figure 42: *Two different high-level clocking strategies.*

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 42 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 0.47. *A typical circuit based on sequential logic will be comprised of*

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

4 Pipelined logic

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 logic, 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

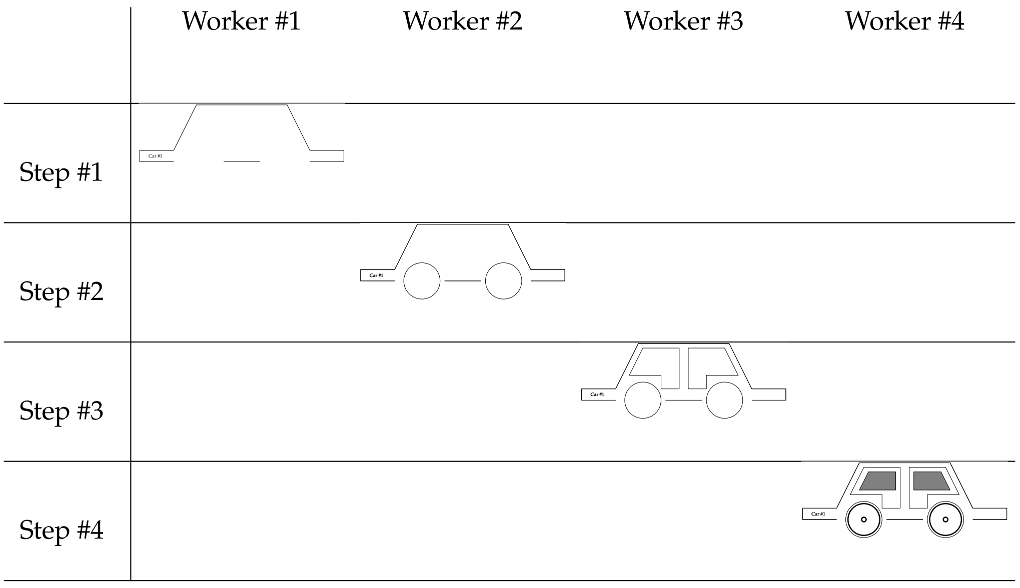


Figure 43: *Production line #1, staffed with pre-Ford workers.*

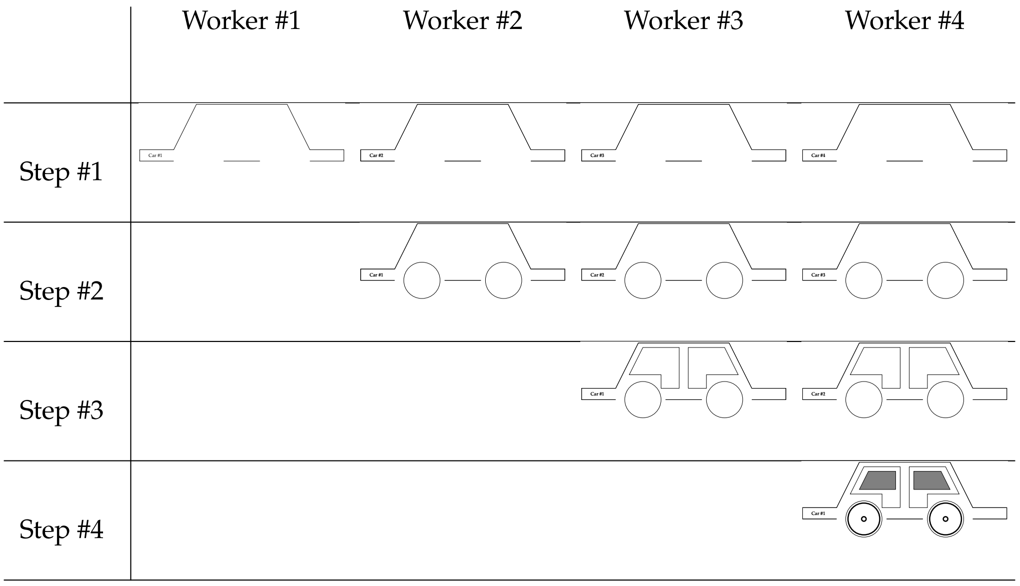


Figure 44: *Production line #2, staffed with post-Ford workers.*

production lines to build cars. While one person was assembling the engine of 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 43 and Figure 44 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 0.48. *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.*

Definition 0.49. *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 logic as a production line

Fortunately, pipelined logic does 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}.

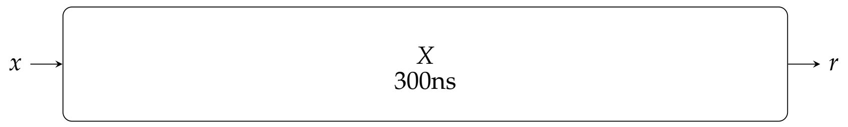
4.2.1 Problem #1: how to structure the pipeline

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?

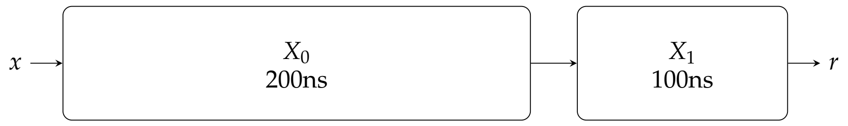
A generic answer to the first question is hard, since it depends on the component itself. About the most general approach we can start with is to identify natural splitting points, i.e., look at X and see where there are steps in the overall computation that can be grouped together. The second question 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 45 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.

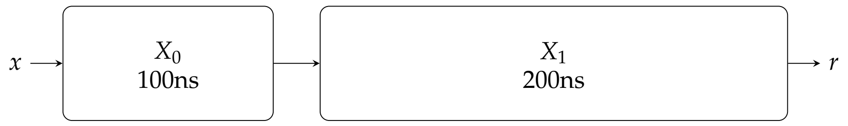
Focusing *only* on the idea of balancing the stages, the last option is most attractive: since all stages take the same time to compute their part of the overall result, selecting this option will minimise potential idleness.



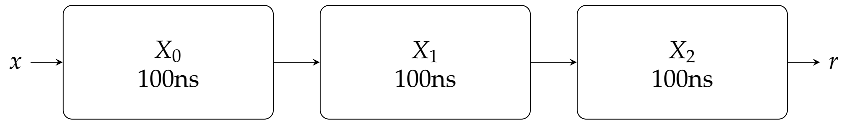
(a) *Option #1: 1-stage, unpipelined.*



(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 45: *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 46a, 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 46c 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 47 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 Some concrete examples

So far, our discussion has been necessarily abstract: many details of a concrete pipeline depend on the component under consideration.

Example 0.39. Consider Figure 48, wherein an abstract component X is shown in both unpipelined and pipelined forms. In the unpipelined case we find that the latency is

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

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.

while the throughput is

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

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

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

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 120ns rather than 320ns. The drawback is that the overall latency of a given operation is slightly more, i.e., 360ns rather than 320ns.

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.

Example 0.40. Consider 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 **??**, but the idea is illustrated by Figure 49a. 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

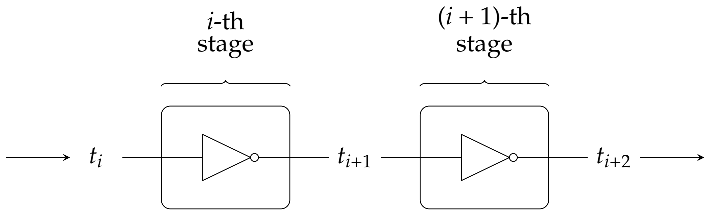
1. since $y_0 = 0$, the 0-th stage passes the input x through unaltered to form the intermediate result x', then
2. since $y_1 = 1$, the 1-st stage shifts the intermediate input x' by a distance of $2^1 = 2$ bits to form the intermediate result x'', then
3. since $y_2 = 1$, the 2-nd stage shifts the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

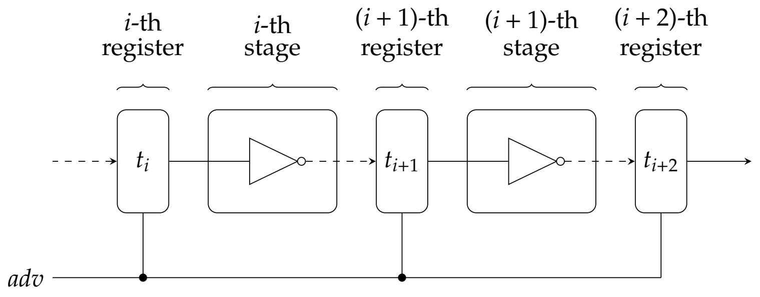
Applying the same reasoning as above Figure 49b splits the design into a 3-stage pipeline; this decision is natural given that the computation is trivially split into three stages of equal latency. Now, the critical path is now determined by just *one* stage rather than all *three* since each stages 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.

5 Implementation and fabrication technologies

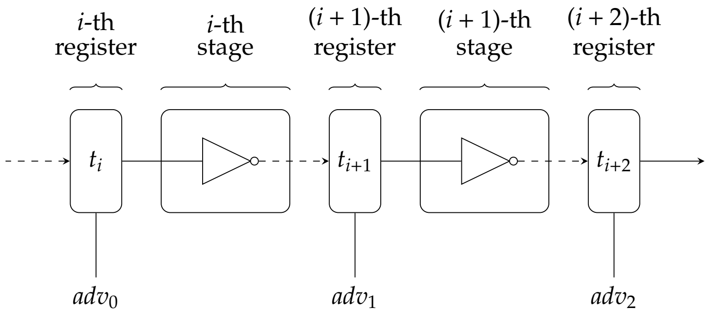
When we write software (i.e., a program), we usually intend to *use* it somehow (i.e., execute it on a computer). The program (or description of behaviour) is often compiled into an form we can use; depending on the



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



(b) *Option #2: with pipeline registers and a global control signal.*



(c) *Option #3: with pipeline registers and multiple, per-stage control signals.*

Figure 46: *A problematic pipeline, and a solution involving the use of pipeline registers and a control signal to indicate when each stage should advance.*

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 semi-conductors to construct 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 semi-conductor-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.

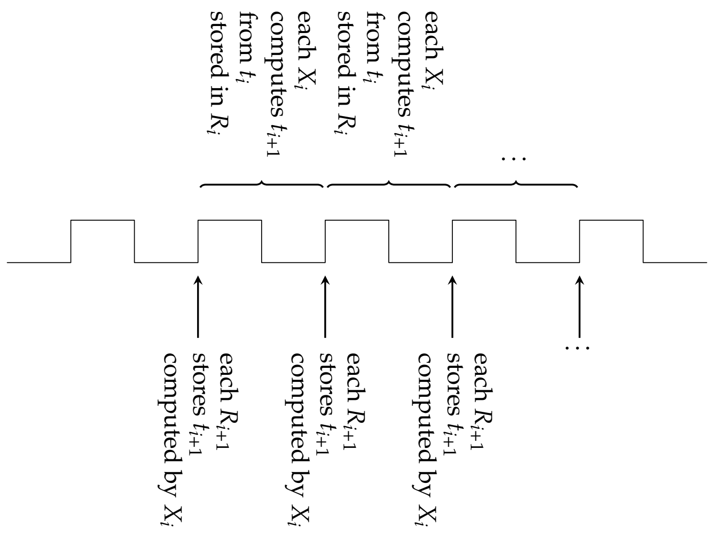
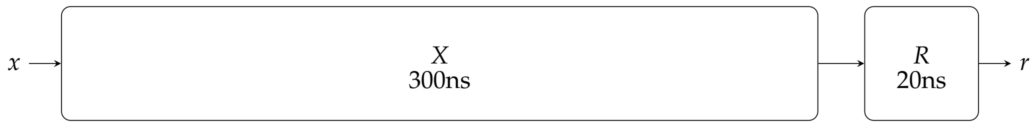
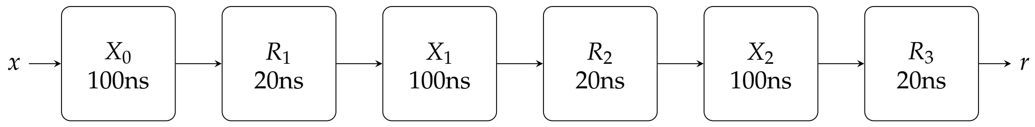

each X_i computes t_{i+1} from t_i stored in R_i
each X_i computes t_{i+1} from t_i stored in R_i
...
each R_{i+1} stores t_{i+1} computed by X_i
each R_{i+1} stores t_{i+1} computed by X_i
each R_{i+1} stores t_{i+1} computed by X_i
...

Figure 47: *An illustrative waveform, 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.*

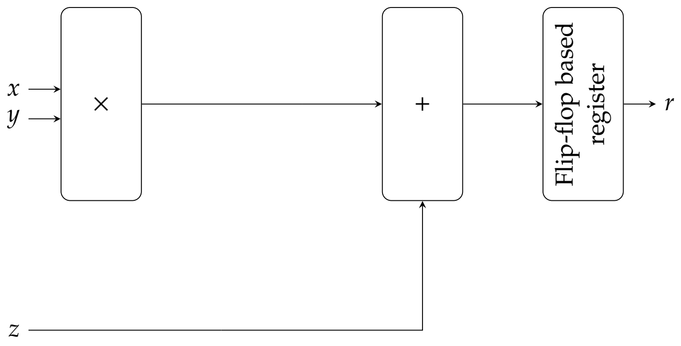
Figure 48: *An unpipelined, abstract combinatorial circuit and a 3-stage pipelined alternative.*

For semi-conductors the analogous process is **photolithography**, and involves very similar steps which are illustrated by Figure 51. We again start (Figure 51a) 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 51b), for example a doped silicon or metal. Then we coat the whole thing with a photosensitive chemical, usually called a **photo-resist** (Figure 51c). 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 51d), 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 51e to Figure 51g).

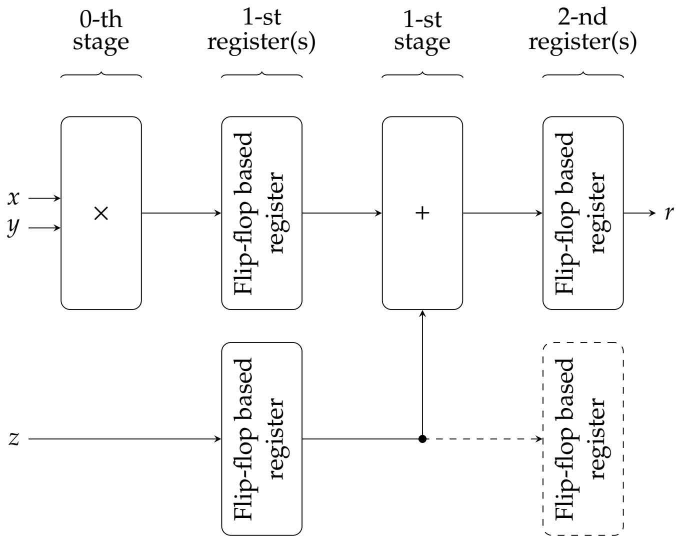
The process iterates to produce *many* layers of potentially different materials, i.e., the result is 3D not 2D. We might need layers of N-type and P-type semi-conductor and a metal layer to produce transistors, for example. The **feature size** (e.g., 90nm CMOS) relates to the resolution of this process; 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

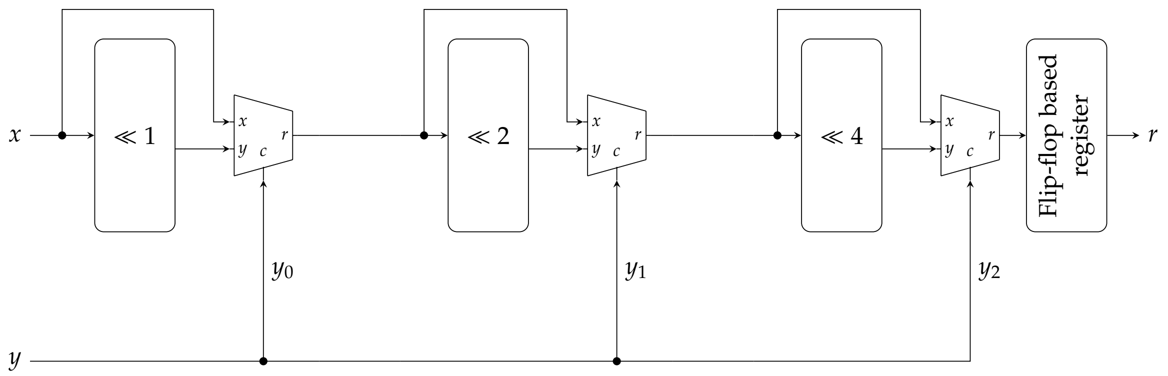


(a) *Option #1: an unpipelined design.*

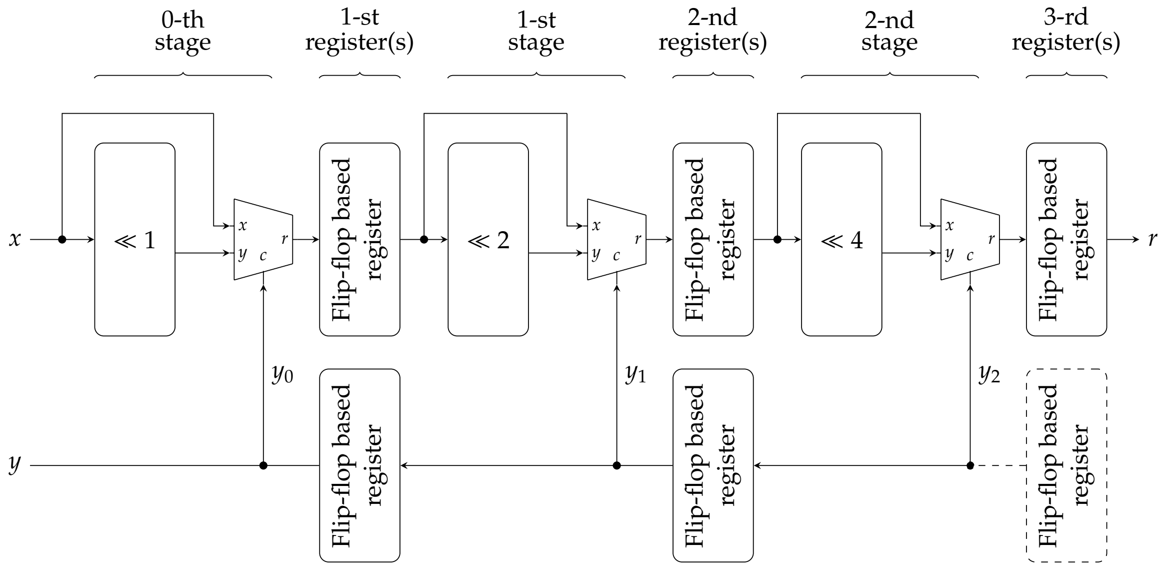


(b) *Option #2: a 2-stage pipelined design.*

Figure 49: *An unpipelined, 8-bit Multiply-ACumulate (MAC) circuit and a 3-stage pipelined alternative.*

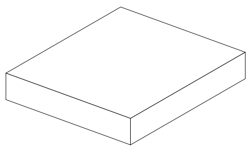


(a) *Option #1: an unpipelined design.*

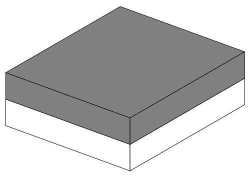


(b) *Option #2: a 3-stage pipelined design.*

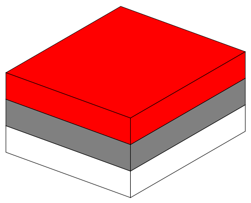
Figure 50: *An unpipelined, 8-bit logarithmic shift circuit and a 3-stage pipelined alternative.*



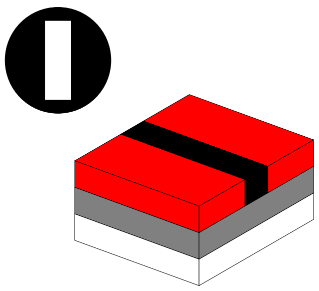
(a) *The substrate.*



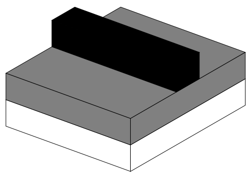
(b) *Coating of base material.*



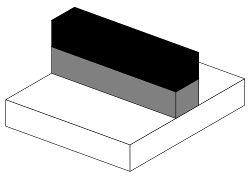
(c) *Coating of photosensitive chemical.*



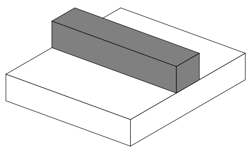
(d) *Exposure to a (simple) mask.*



(e) *Application of etching.*



(f) *Application of etching.*



(g) *Final result.*

Figure 51: *A high-level illustration of a lithography-based fabrication process.*

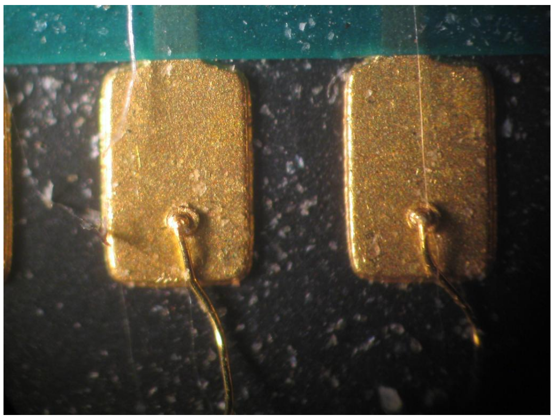


Figure 52: *Bonding wires connected to a high quality gold pad (public domain image, source: `http://en.wikipedia.org/wiki/Image:Wirebond-ballbond.jpg`).*

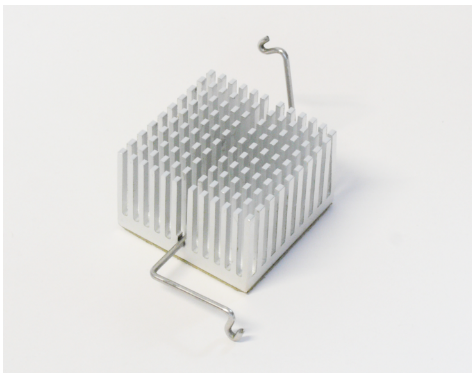


Figure 53: *A heatsink ready to be attached, via the z-clip, to a circuit in order to dissipate heat (public domain image, source: `http://en.wikipedia.org/wiki/File:Pin_fin_heat_sink_with_a_z-clip.png`).*

also applied to prevent physical damage; large or power-hungry components might also mandate use of 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 [5] 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.
>
> – 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 54 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.

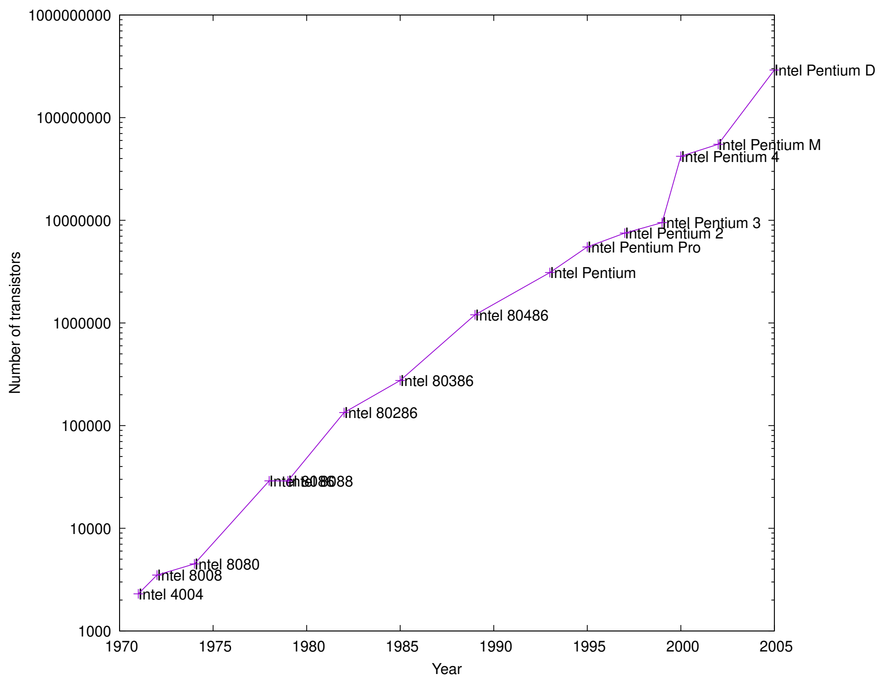


Figure 54: *A timeline of Intel processor innovation demonstrating Moore's Law (data from* http://www.intel.com/technology/mooreslaw/*).*

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.

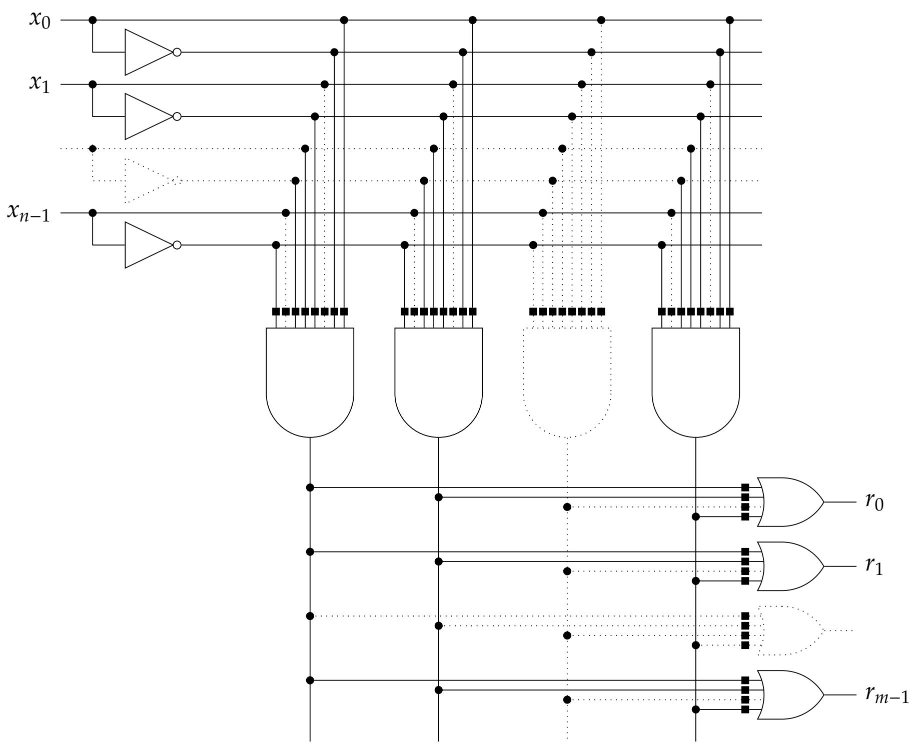
5.2 (Re)programmable fabrics

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.

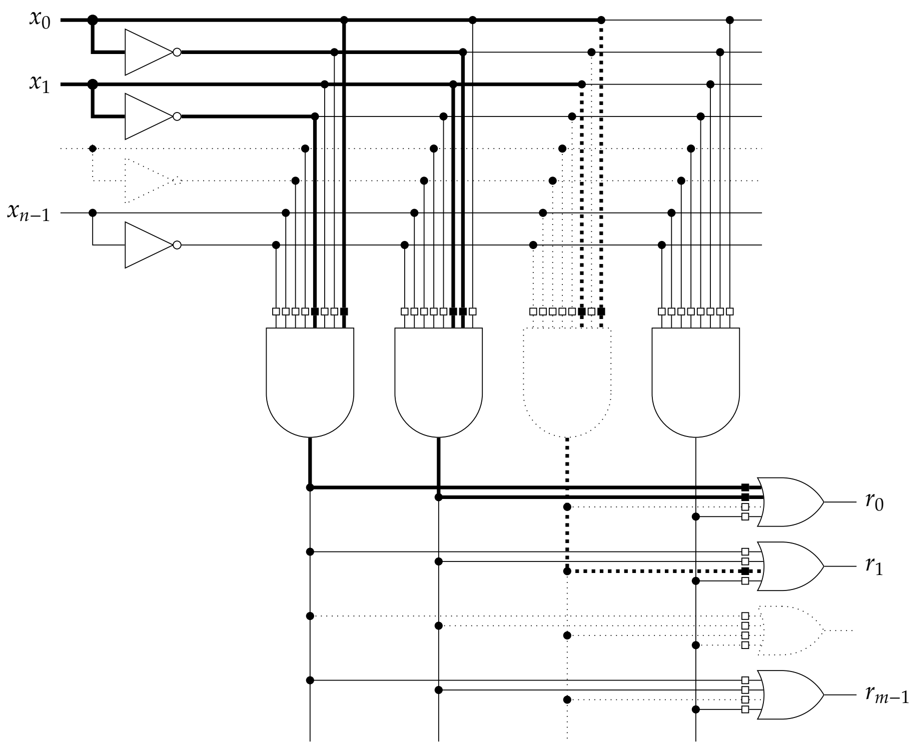
5.2.1 Programmable Logic Arrays (PLAs)

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 55a. At this point you can think of *all* of the gates being 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



(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 55: *Conceptual diagrams of a PLA fabric.*

simply blow all the fuses[6] where no connection is required. Figure 55b 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 a fuse (or antifuse) is blown, it cannot be *un*blown. Since a PLA can only be configured once, it is not unreasonable to think of a PLA as like a ROM (in the sense that once programmed, the content is fixed) but has the advantage of being able to optimise for don't care entries. However, the fixed structure means that versus conventional combinatorial logic, it has the disadvantage of being less (easily) able to capitalise on optimisations such as sharing logic for common sub-experssions.

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.

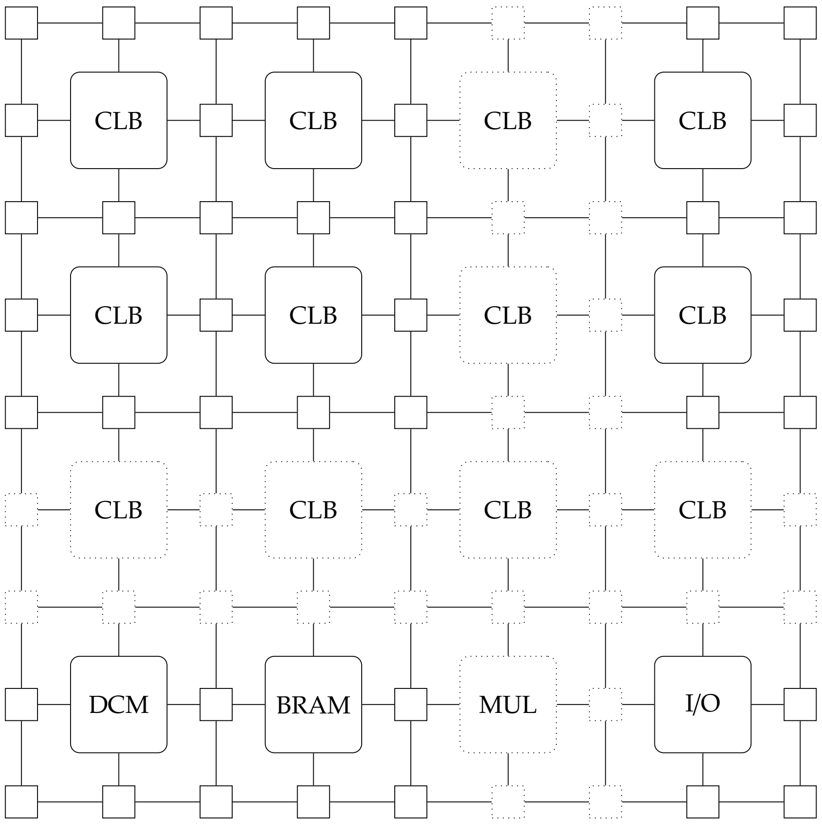
Figure 56a is a conceptual representation of an FPGA fabric, which is basically a collection of logic resources (or blocks) organised in a two-dimensional mesh; the logic blocks are connected using routing resources placed between them. Both the logic and routing resources are controlled by a configuration 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 56b is a block diagram of a Vertex-4 slice, which contains

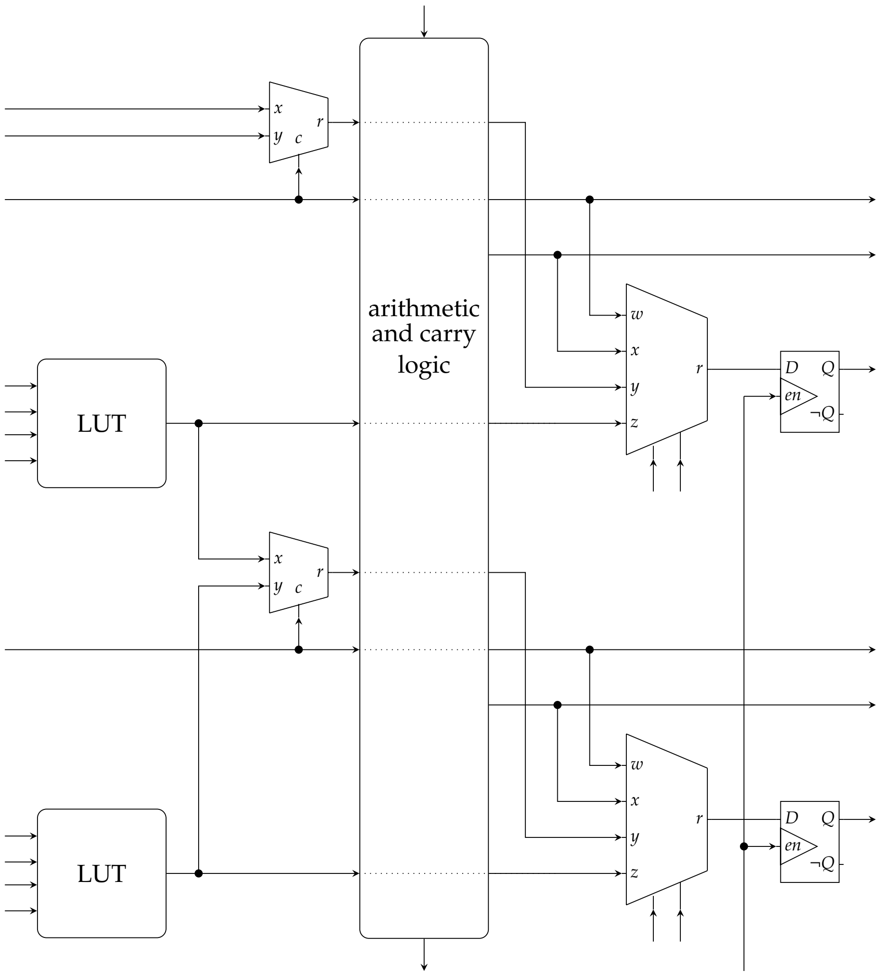
- two 4-input, 1-output Look-Up Tables (LUTs),
- two D-type flip-flops,
- a suite of arithmetic cells, including two 1-bit full-adders, and
- several interconnected multiplexers.

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 56a attempts to show this fact by including

[6]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 56: *Conceptual diagrams of an FPGA fabric.*

- 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).

References

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

[2] M. Karnaugh. "The map method for synthesis of combinatorial logic circuits". In: *Transactions of American Institute of Electrical Engineers* 72.9 (1953), pp. 593–599 (see p. 23).

[3] M. Knodel and N. ten Oever. *Terminology, Power and Oppressive Language*. Internet Engineering Task Force (IETF) Internet Draft. 2018. URL: https://tools.ietf.org/id/draft-knodel-terminology-00.html (see p. 54).

[4] E.J. McCluskey. "Minimization of Boolean function". In: *Bell System Technical Journal* 35.5 (1956), pp. 1417–1444 (see p. 27).

[5] G.E. Moore. "Cramming more components onto integrated circuits". In: *Electronics Magazine* 38.8 (1965), pp. 114–117 (see p. 68).

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

[7] W.V. Quine. "The problem of simplifying truth functions". In: *The American Mathematical Monthly* 59.8 (1952), pp. 521–531 (see p. 27).

[8] R.J. Smith and R.C. Dorf. "Chapter 12: Transistors and Integrated Circuits". In: *Circuits, Devices and Systems*. 5th ed. Wiley, 1992 (see p. 9).

[9] A.S. Tanenbaum and T. Austin. *Structured Computer Organisation*. 6th ed. Prentice Hall, 2012.

[10] E.W. Veitch. "A Chart Method for Simplifying Truth Functions". In: *ACM National Meeting*. 1952, pp. 127–133 (see p. 27).

[11] N.H.E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*. 2nd ed. Addison Wesley, 1993 (see p. 45).