Computer Architecture

Muchang Bahng

Spring 2024

Contents

1 Sequential Chips **2**
1.1 SR Latches . . . 2
1.2 Level and Edge Triggered D-Latches . . . 5
1.3 Flip Flops . . . 10
1.4 Registers . . . 16
1.5 Applications . . . 17

2 Binary Encodings **20**
2.1 Naturals/Unsigned and Integers/Signed . . . 20
2.1.1 Arithmetic Operations on Binary Numbers . . . 25
2.2 Rationals and Countable Sets . . . 25
2.3 Floats . . . 26
2.4 Characters . . . 27
2.4.1 ASCII . . . 28
2.4.2 ISO-10646, UCS . . . 30
2.4.3 Unicode, UTF-8 . . . 30
2.4.4 Text Files . . . 32
2.5 Representation of General Sets . . . 32

3 Combinational Logic **35**
3.1 Multi-Bit Gates . . . 35
3.2 Addition and Subtraction . . . 35
3.3 Comparator . . . 37
3.4 Multiplication . . . 37
3.5 Arithmetic Logical Unit (ALU) . . . 38
3.6 Conditionals . . . 38

4 Memory Banks **39**

1 Sequential Chips

Now that we know how the NAND gate—and therefore every other fundamental gate—works, and we have constructed the clock, the next natural step is to be able to store a string of bits.[1] Computers must be equipped with memory elements that can preserve data over time. These memory elements are built from *sequential chips*.

1.1 SR Latches

Ideally, we would like a way to store a bit in memory, and this can be done by cross-coupling gates with each other, forming a sort of positive feedback.Therefore, given a certain signal into our circuit which we call a *latch*, the outputs remain locked—or "latched"—into a state.

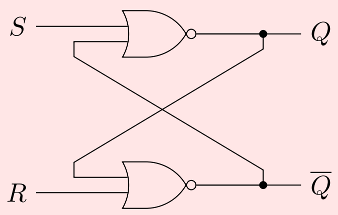
Definition 1.1 (SR Latch)

The **set-reset (SR) latch** is a circuit that stores 1-bit memory. This is based on *pulses* and we do not care about the duration of a signal. That is, if we activate a signal to inputs S, R at *any* point in time, then the output Q will remain locked in some state, even *after* the input signal disappears.

The SR latch—like all electronic circuits—require power to work, labeled with S and R. The output is really just Q, but we can add redundancy by making the inverse $\overline{Q}$ available as well. There are two implementations of an SRlatch, which have symmetric behaviors.

Theorem 1.1 (Active High SR Latch)

A NOR SR latch can be implemented in the following circuit below, with its corresponding truth table.



(a) Circuit diagram.

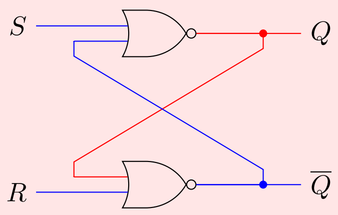
S	**R**	**Q**	$\overline{Q}$
0	0	1	0
		0	1
0	1	0	1
1	0	1	0
1	1	0	0

(b) Truth table.

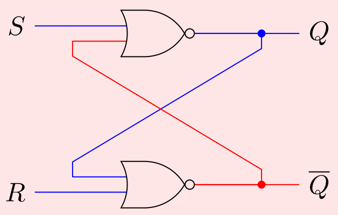
Figure 1: XOR SR Latch. This is

Setting both $R = S = 1$ would result in an invalid state since they would attempt to turn Q back and forth between 0 and 1, giving us a race condition.

[1]Most courses teach combinational logic first and then sequential, but this may not be the most optimal dependency sequence for two reasons. We can indeed do arithmetic without memory by directly applying an electric current to the input wires in a circuit, but this severely limits the computation that we can do. We would essentially have to do everything in "one shot" and immediately collect the results. While this is fine for addition, I cannot introduce an efficient schema of multiplication without knowing how to bit-shift, which is dependent on some form of memory. On a broader scale, almost all algorithms we worked with require some memory at some point, so memory may be more fundamental than computation. Thanks the Phillip Williams for talking with me on this!



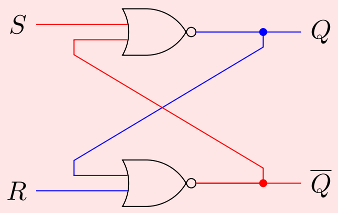
(a)



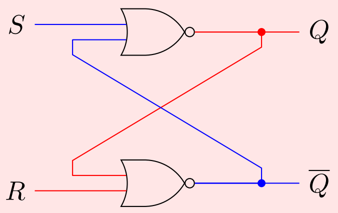
(b)

Figure 2: Two possible initial states. The default state is $R = 0, S = 0$, which are both *low states*, and Q may be either 0 or 1.

If one of R or S is set to a high state, the latch is activated, and hence this is called an **active high SR latch**. Note that regardless of what the previous state the latch was in, the output signals are completely determined.



(a) If we send a signal $R = 1$, then $Q = 0$, and even if we reset $R = 0$, Q is still locked at 0.



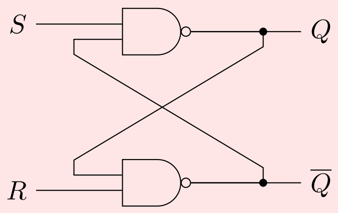
(b) If we send a signal $S = 1$, then $Q = 1$, and even if reset $S = 0$, Q is still locked at 1.

Figure 3

Now unlike the active high latches which are activated when the current is 1, active low latches are activated when the current is 0.

Theorem 1.2 (Active Low SR Latch)

A NAND SR latch can be implemented in the following circuit below, with its corresponding truth table.



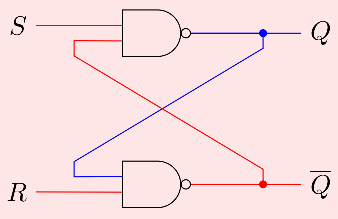
(a) Circuit diagram.

S	**R**	**Q**	$\overline{Q}$
0	0	1	1
0	1	1	0
1	0	0	1
1	1	0	1
		1	0

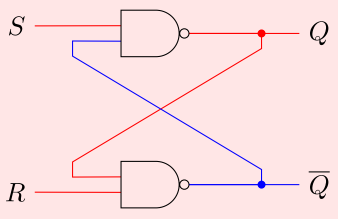
(b) Truth table.

Figure 4: NAND SR Latch

Setting both $R = S = 0$ would result in an invalid state since they would attempt to turn Q back and forth between 0 and 1, giving us a race condition.



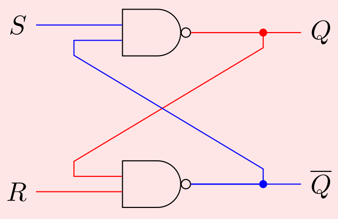
(a)



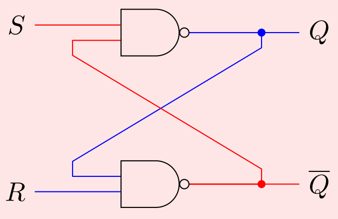
(b)

Figure 5: The default state is $R = 1, S = 1$, i.e. they are both *high states*, and Q may be either 0 or 1. This is known as an **active low SR latch**.

If one of R or S is set to a low state, the latch is activated, and hence this is called an **active low SR latch**. Note that regardless of what the previous state the latch was in, the output signals are completely determined.



(a)



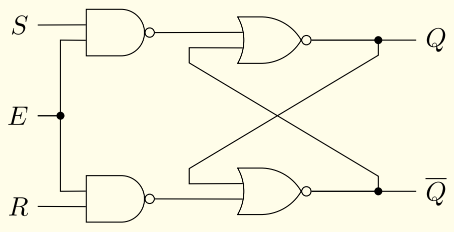
(b)

Figure 6: If we send a signal $S = 0$, then $Q = 0$, and even if reset $S = 1$, Q is still locked at 0.

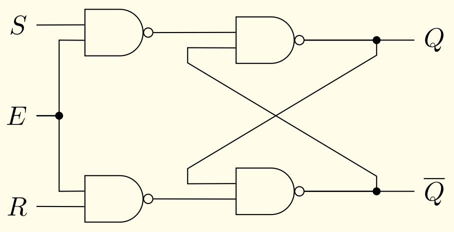
These signals may be noisy, and we might want more control over whether a latch can change states, i.e its *transparency*. This is done by adding an extra *gate* that explicitly tells us when the latch can change states.

Definition 1.2 (Gated SR Latch)

A **gated SR latch** is an SR latch that can only change state when it is enabled. This enabling is done with an additional 2 NAND gates, and so the SR latch is enabled only when $E = 1$.



(a) Gated XOR SR Latch.



(b) Gated NAND SR Latch.

Figure 7: Note that if $E = 0$, then the output of the leftmost two NAND gates will be 1 no matter what, and so the values of R, S does not have any effect.

Example 1.1 (Active High Gated SR Latch)

By keeping track of the voltages in the wires of interest and running them across a common time axis, we can visualize this circuit in action. Note that in here, we assume that electric current is instantaneous, resulting in the familiar *square waves*. Let's look at an active high SR latch.

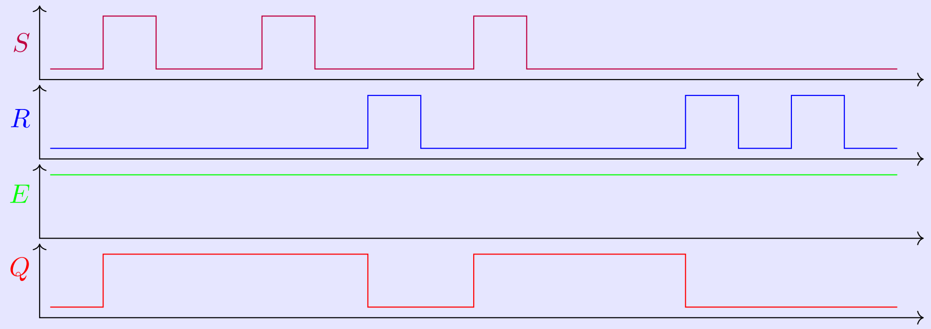


Figure 8: In here the gate is always enabled as $E = 1$ always. In the beginning $S = 1$ causing $Q = 1$, and this does not change until $R = 1$, at which point $Q = 0$. Note that the second pulse of S does not affect the state because it is already $Q = 1$. Soon after $S = 1$ again, causing $Q = 1$ and when $R = 1$ $Q = 0$.

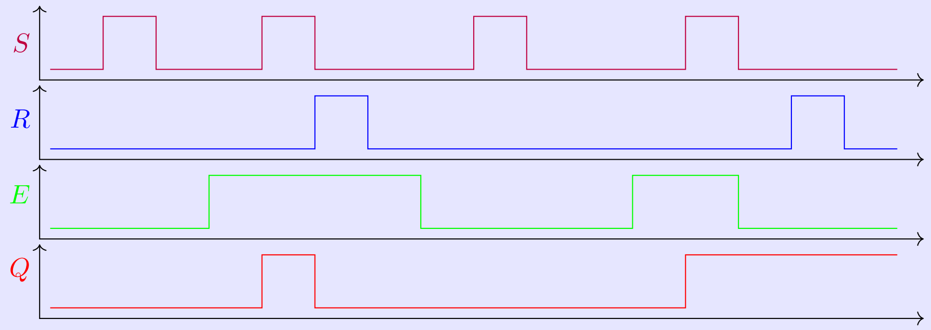


Figure 9: Now we toggle E on and off throughout. We can start off by filling in all the places where $E = 1$, where we want Q to basically copy S. At every other place, we just continue what the state Q was in.

1.2 Level and Edge Triggered D-Latches

Note that we still have the problem of invalid signals. For example, if there was an instance that at the same clock time a signal of $S = 1, R = 1$ (on either an ungated latch or a gated latch with $E = 1$), then both Q and $\overline{Q}$ will be 1, which will cause both to be 0, and then 1, and so on. This causes a race condition, which leads to unpredictable behavior.

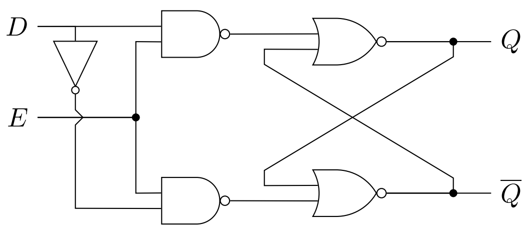
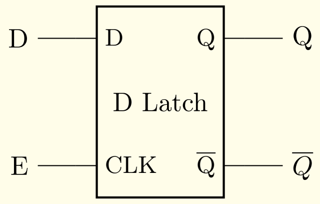


Figure 10

It turns out that we can simplify this circuit, making it cheaper to produce while still behaving identically. This gives us the D-latch.

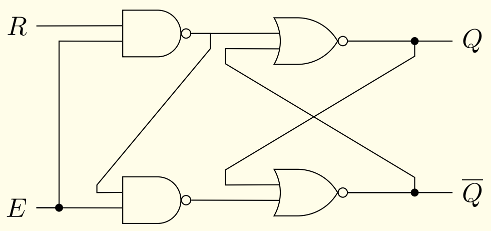
Definition 1.3 (Level Triggered D-Latch)

The **(gated) data latch (D-latch)**, also called a **clocked D-latch**, gives us more control over storing a 1-bit in memory.

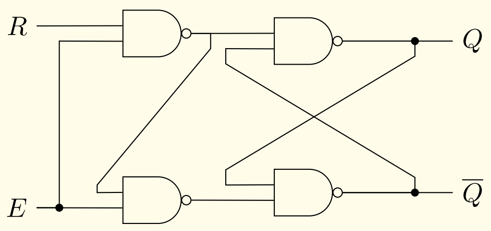


E	D	Q	$\overline{Q}$
0	0	Q_{prev}	$\overline{Q_{\text{prev}}}$
0	1	Q_{prev}	$\overline{Q_{\text{prev}}}$
1	0	0	1
1	1	1	0

Figure 11: Chip notation and truth table of a D latch. Note that when $E = 0$, the latch simply outputs the previously stored element $Q = Q_{\text{prev}}$.



(a) Gated XOR SR Latch.



(b) Gated NAND SR Latch.

Figure 12

Example 1.2 (Level Triggered D-Latch)

The essence of the behavior is the output follows the input while E is enabled.

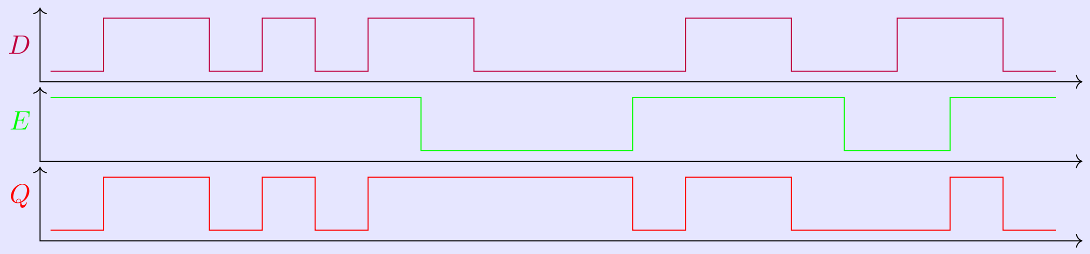


Figure 13: Again, we just let the result Q follow the input D whenever $E = 1$,and continue the rest for when $E = 0$.

Therefore, if we want to store a bit of information, we set $E = 1$, collect that bit from D, and then set $E = 0$ to latch it in place. This behavior is quite stable for storing 1-bit, but we need more control when storing a multi-bit buffer, where we need several D-latches working in tandem. The general idea is that if we have a multi-bit buffer, we want a set of D-latches to be enabled and disabled at once.

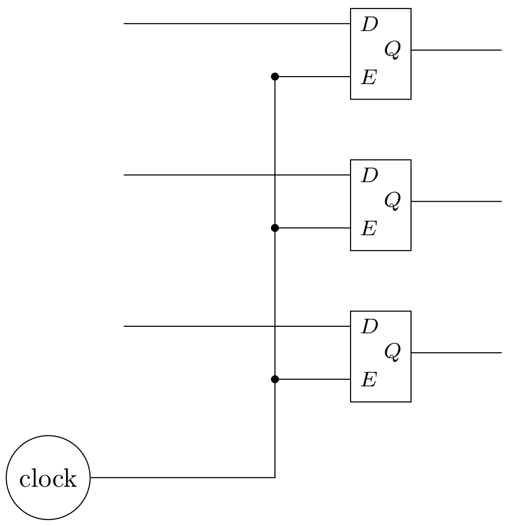


Figure 14: Multiple D-latches enabled and disabled by some external source. The system clock would be a good candidate.

Therefore, given the system clock, our waveforms would look like this.

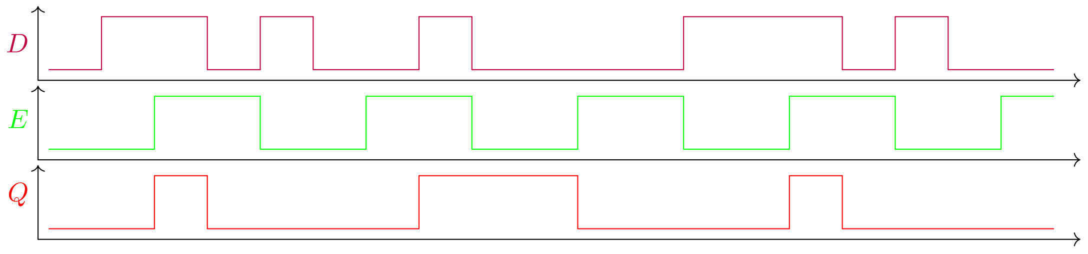


Figure 15: E is connected to a clock that ocsillates at regular intervals.

This is still not a perfect solution for synchronizing some components. Depending on the frequency of the clock, E may be high for as long at 50 microseconds. That's a long time for the data latch to be open to changes in D. For some applications, particularly those where the outputs are fed back to the inputs, we can avoid disorder and noise from D by drastically limiting the amount of time E is open during each clock cycle.

But simply increasing the frequency of the clock isn't a practical solution, given that a computer contains a mixture of fast and slow components. A more clever solution is to only allow changes to the latch when the clock input E is changing from low to high. Due to propagation delay, this is indeed a feasible solution since the waveforms are not truly square waves.

Definition 1.4 (Rising, Falling Edge)

The period when a signal
1. changes from 0 to 1 is called the **rising edge**.
2. changes from 1 to 0 is called the **falling edge**.

This usually takes a few nanoseconds.

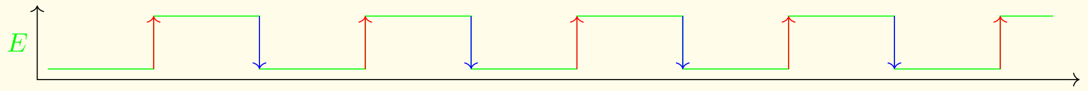
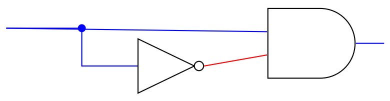
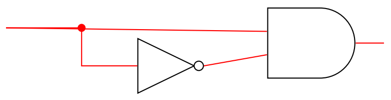


Figure 16: Rising edges are in red, falling edges in blue.

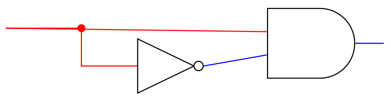
As a summary, for level-triggered (transparent) latches, we have active high and active low. Analogously, for edge-triggered latches, we have rising edge and falling edge. We want to build a D-latch that will respond to changes in D *at the rising edge*, with subsequent changes in D being ignored until the next rising edge.



(a) By default the input current is 0, and so the top input of the AND gate is 0 and the bottom is 1.



(b) If the electric current of 1 travels through the input wire, the top AND input becomes 1. There is a small delay where the current does not reach the output of the NOT gate, so the output is 1.



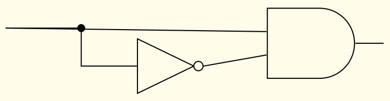
(c) The signal goes through the NOT gate, turning the AND output back to 0.

Figure 17: An edge detection device. Note that if we want to delay the signal even further, we can put an arbitrary amount of NAND gates.

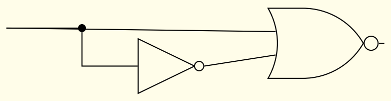
We take this idea to build an edge detection device.

Definition 1.5 (Edge Detection Device)

This is a rising edge detection device. Note that depending on many factors, like manufacturing, temperature, etc., there may not be a long enough delay to actually detect an edge, and in this case you can just add more (odd number of) NOT gates



(a) Positive edge triggered detection device.



(b) Negative edge triggered detection device.

Figure 18

It has the following waveform.

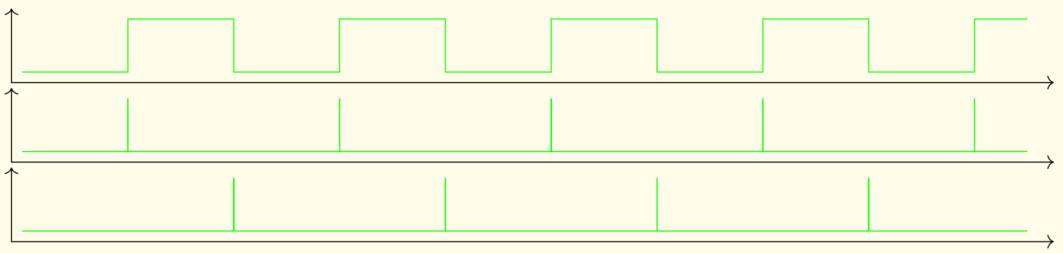


Figure 19: The clock cycle (top). Positive edge detection device (middle). Negative edge detection device (bottom).

Now if we combine our D latch with the edge detection device, we change it from a level-triggered device to an edge-triggered device. Since we are using a clock as our trigger, we also call this a *pulse D-latch*.

Definition 1.6 (Edge-Triggered D-Latch, Pulse Latch)

An **edge-triggered D-latch**, also known as a **pulse D-latch**,[a] is a D-latch that is enabled on the rising edge of a clock cycle.

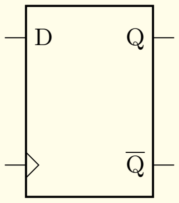


Figure 20: A clocked D latch. Note that the triangle is used to indicate that the clock is inputted.

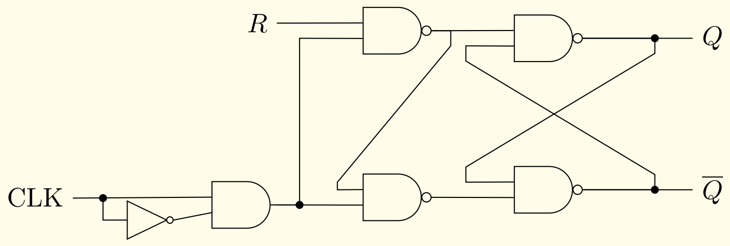


Figure 21

[a]Often, edge-triggered latches are in general referred to as a flip flop, but we will distinguish that a bit later.

Example 1.3 (Edge-Triggered D-Latch Waveforms)

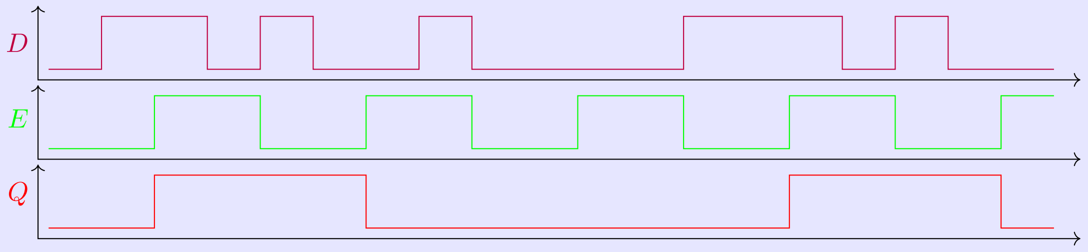


Figure 22: Again, we just let the result Q follow the input D whenever $E = 1$,and continue the rest for when $E = 0$.

Definition 1.7 (Set-Reset Inputs)

Another enhancement we can make is to have an option to manually set the latch to be either $Q = 1$ or 0, independent of the clock. This gives us the **pulse D-latch**, which allows us to initialize it unconditionally.

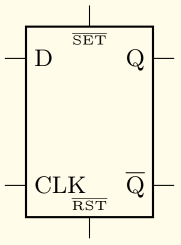


Figure 23: D latch with asynchronous set/reset.

The implementation is to simply add extra inputs after the NAND gates.

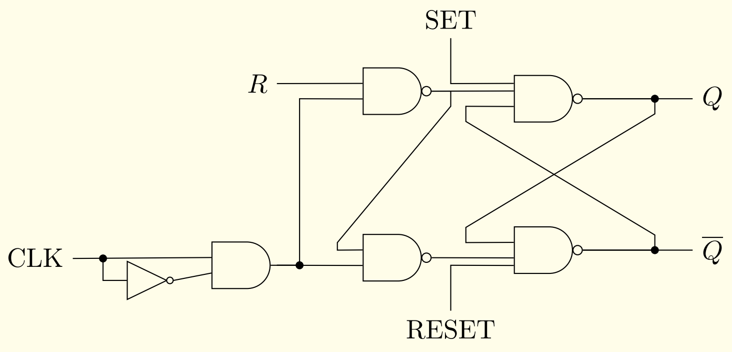


Figure 24

This gives us a reliable device for storing 1 bit of memory. It is enabled and disabled by a clock signal, and used in registers, memory circuits, and counters as we will see later.

1.3 Flip Flops

So far, we have considered various mechanisms that allowed for greater control of a latch, along with robustness to noise. Now we revisit the final problem of attempting to *coordinate* a group of latches, where timing is a fundamental consideration. Just like the conductor of an orchestra, the clock sets the timing and the pace of everything in the computer, which consists of both fast and slow moving parts.

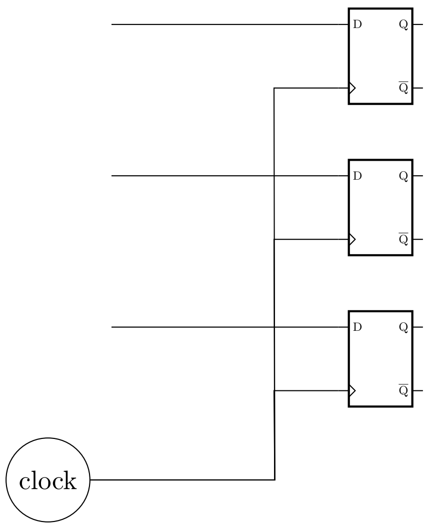


Figure 25

Ideally, to synchronize the setting of these latches we'd make all of the inputs the way we want them to be while the clock signal is low. Then, when the clock signal becomes high, these input values would be transmitted to the latches and their values stored.

But unwanted fluctuations—known as *glitches*—can occur on the data lines because of propagation delays and even noise. Conceivably, we can have a situation in which our latches haven't had enough time to achieve their correct values before the clock pulse ends. It is crucial that these inputs are allowed to settle into their correct values while the clock signal is high. This is because there is a different circuit ready to make iemmediate use of the data in the register, even perhaps during the very next clock cycle. The outputs of these latches have to be stable before they are sampled. The data in this register must be accurate before something else reads it. Otherwise, we would have complete garbage outputs.

We could try to avoid the problem caused by glitches by speeding up the clock, allowing less time for them to matter, but we also have to allow time for the components to do their jobs. We have to cater for their propagation delays. If a clock is running too quickly, some components won't be able to keep up.

We can also make circuits less susceptible to glitches by building edge-triggered devices like pulse latches, but the rising edge of a clock cycle is only in the order of a few nanoseconds, and even with very careful design, there might not be enough time for everything to keep pace. Therefore, the clock period must be so that all of the other circuits have time to stabilize during the same high phase of the same clock cycle.

Ultimately, if all circuits in a computer work on the basis that only one signal change per clock cycle matters, then their behavior can be coordinate reliably. One way that we can ensure that this is the case is to build a memory device that is immune to glitches, called the *master-slave D-type flip-flop*. With this, we can precisely control the moment at which a group of them will change state.

Definition 1.8 (Master-Slave D-Type Flip Flop)

The **master-slave D-Type flip-flop (DFF)** consists of two active-high gated latches. The left portion, called the *master*, is a gated D-latch. The right portion, called the *slave*, is a gated SR latch that takes the output of the master as its input and is enabled by the inverse of the clock signal. Taken together, the data and the clock inputs enable to the DFF to implement the time-based behavior

$$\text{out}(t) = \text{in}(t-1) \tag{1}$$

That is, the DFF outputs the input value from the previous time unit.

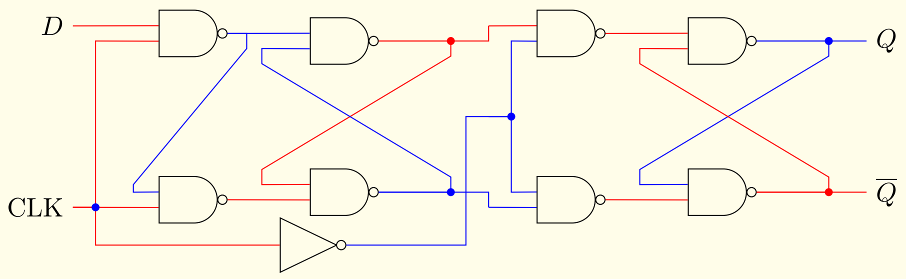


Figure 26: The master reads the input value D when the clock signal is high (or more specifically, the rising edge of the clock cycle) and latches onto it. Meanwhile, the slave is disabled, so the new output from the flip flop is not available just yet.

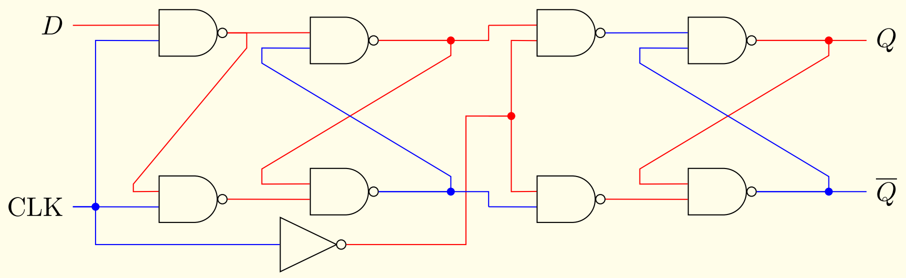


Figure 27: When the clock cycle falls to low, the slave is enabled, making this an edge triggered device. Data is passed from the master to the slave and is therefore available at the output.

Essentially a DFF is analogous to an airlock consisting of two doors that can never be both open at the same time. The flip flop is never open so an input signal cannot pass straight through like a regular D latch. The output of the flip flop occurs during the next phase of the same clock cycle.

Example 1.4 (Waveforms of DFF)

Let's look at an example where a regular D latch would be insufficient, but a DFF fixes the problem. Note that Q_m behaves just like a D-latch since it is.

1. $t = 2$. C has a rising edge and $D = 0$, so $Q_m = 0$.
2. $t = 5$. D has become high, presumably because we want the output at Q_m to go high. But because $C = 0$, this doesn't happen just yet, and the master is still latched in a low state.
3. $t = 6$. When $C = 1$, Q_m reacts immediately to follow D, and so it becomes high.
4. $t = 8$. When C goes low again, D is high and so is Q_m, and so the master is latched in a high state.
5. $t = 9$. D goes low again, presumably because we want to change the state of the master latch back to low again. But because C is low, Q_m does not follow.
6. $t = 10$. When C goes high, Q_m immediately goes low.
7. $t = 11$. We can see D changing again while the clock is high. Suppose that a completely different circuit depended on the output of Q_m being low. Since Q_m was both low and then high in the same high clock cycle, the circuit might have missed its chance to read from Q_m. We want to avoid this, and ideally, we want to (1) set the value of D *before* the clock goes high, and (2) *not* have D change in the middle of a clock high cycle.
8. $t = 12$. Now C is low, and the master is latched in a high state.
9. $t = 22$. We can see that the value of D is changing again during a high phase of the clock cycle—another glitch.

If we take a look at the output of the slave Q_s, which follows Q_m since the master's output is the slave's input. But more importantly, Q_s only follows Q_m while C is low (because the slave is being fed the inverse of the clock signal).

1. $t = 6$. Q_m is changing from low to high, but Q_s remains low since C is high. While the flip flop is responding to a change in input, the output remains the same.
2. $t = 8$. At the falling edge of the clock cycle, Q_s follows Q_m to become high. Notice that the master's output Q_m cannot be changed now because C is low. This means that changes to the input of the flip-flop cannot impact the output at this point.
3. $t = 9$. The input at D has changed from high to low, as if in readiness for another change to the state of the flip flop.
4. $t = 10$. When C goes high, the output of the master Q_m changes but this has no impact on Q_s. The slave isn't listening.
5. $t = 11$. D goes high again during the high phase of the clock cycle. But this glitch has no effect on the output of the flip flop.
6. $t = 20$. We see Q_s changing again to follow Q_m while the clock signal is low. The master will also ignore any changes in the input while the flip flop's output is made available.
7. $t = 21$. We see D goes high, as if to set the state of the flip-flop to high on the next high cycle.
8. $t = 22$. When the clock goes high, Q_m follows D to become high as well.
9. $t = 23$. But the input D falls to low while the clock is high, and so does Q_m.
10. $t = 24$. By the time the clock falls to low again, and the slave is once again responding to changes to its input, the flip flop has ignore yet another glitch.

In summary, the DFF effectively ignores any input fluctuations because the master and slave are enabled on opposite phases of the clock cycle. It is safe because it allows sufficient time for propagation delays and therefore time for the inputs to change and settle down without affecting the output. It is however more complicated and resource-intensive than regular latches.

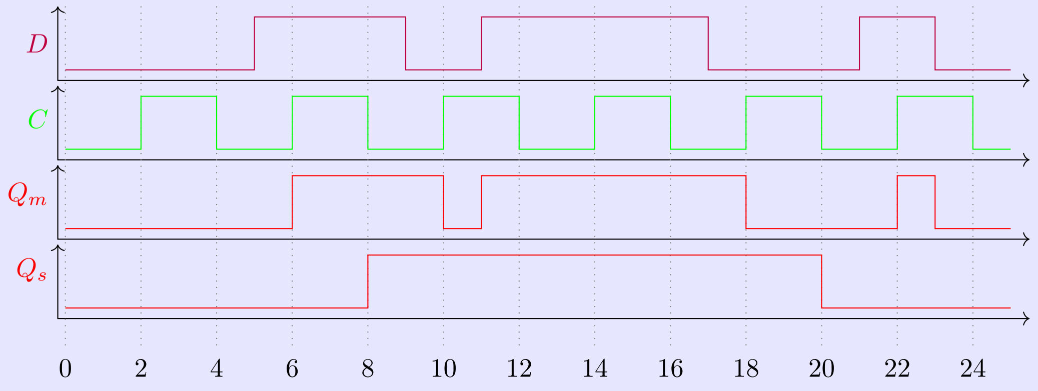
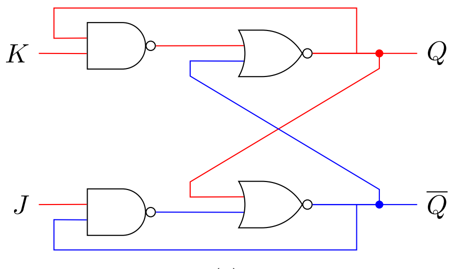
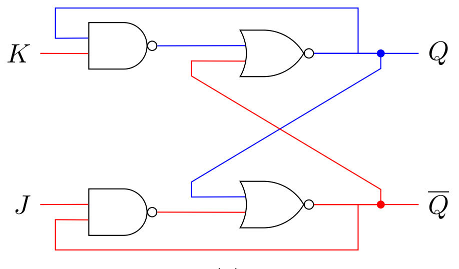


Figure 28

Now let's look at two more types of flip-flops. We revisit the problem of invalid states for SR latches, which lead to race conditions (both 1s for active high and both 0s for active low). We introduce the JK latch, which is not a flip flop yet. Note that you put a pulse through K to reset $Q = 0$, and a pulse through J to set $Q = 1$.



(a)



(b)

Figure 29: JK active high latch. When you set $J = K = 1$, the latch oscillates between $Q = 0$ and $Q = 1$ very fast, but this eliminates the possibility where Q is both 1 or both 0.

Note that we can do the same with an active low JK latch, which will be functionally identical to the active high one. Now we are a step closer to the JK flip flop.

Definition 1.9 (Level Triggered JK Flip-Flop)

By adding a gate/enabler and synching it with the clock, we can get the **level triggered JK flip flop**.

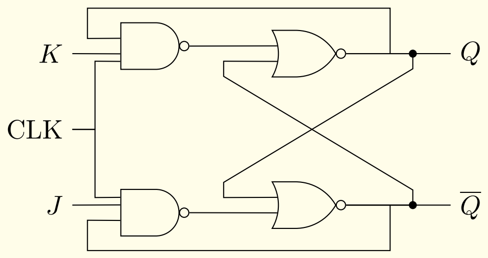


Figure 30: Level Triggered JK Flip Flop.

Example 1.5 (Waveforms of Level Triggered JK Flip Flop)

Let's go through the timing diagram.

1. $t = 1$. When $K = 1$, there is no change in Q since the clock is low. The flip flop is disabled.
2. $t = 2$. The clock is high and the reset signal goes through the AND gate, and $Q = 0$.
3. $t = 10$. J becomes high and the clock is high, enabling the latch again, and consequently Q is high again.
4. $t = 18, 22, 26$. When C, J, K are all high, then the circuit begins to oscillate uncontrollably.

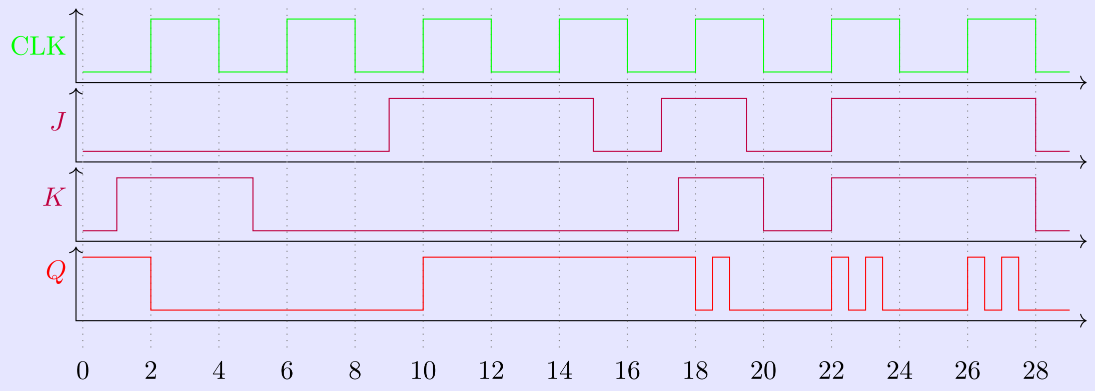


Figure 31

To take advantage of this oscillation, we need a flip flop that will only react the inputs while the clock singal is changing from low to high, i.e. on the rising edge.

Definition 1.10 (Edge Triggered JK Flip Flop)

To fix the weird oscillation issues, we can use the edge detection device to get the **edge triggered JK flip-flop**.

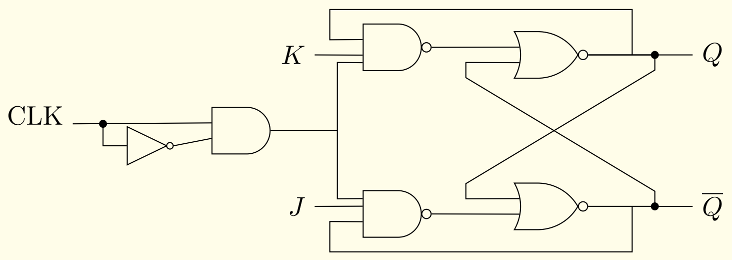


Figure 32: Edge Triggered JK Flip Flop.

It is also called the *universal programmable flip flop* since you can make other types of flip flops from JK flip flops.

Example 1.6 (Waveforms of Edge Triggered JK Flip Flop)

With the edge detector, only the rising edge of each clock pulse has any effect. Notice that when J and K are both high, a clock pulse will cause the flip flop to toggle from one state to the other (at times $t = 18, 22, 26$).

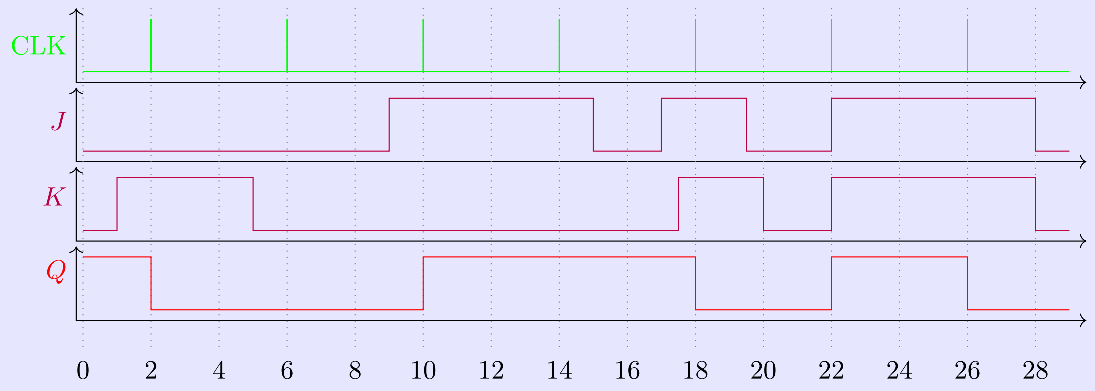


Figure 33

Another simple modification of the JK flip flop gives us another type of flip flop. B simply connecting together J and K to make one input, we now have a device that will toggle from one state to the other when the input is high at the rising edge of the clock.

Definition 1.11 (Toggle Flip Flip)

The **toggle flip-flop**, also known as the **T-Type flip-flop**, is used as an oscillator (?).

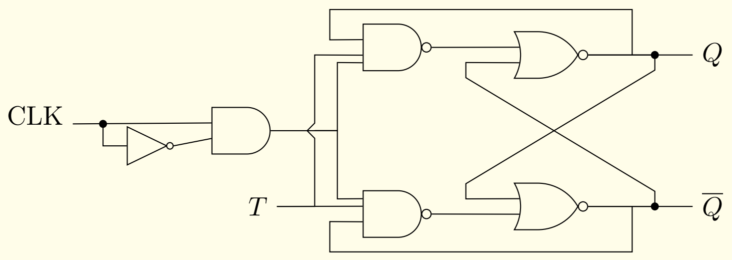


Figure 34: Toggle flip flop.

1.4 Registers

The DFF then unlocks the our first type of memory.

Definition 1.12 (Register)

A w**-bit register** is a memory device, composed up of w DFFs, that can hold w bits of memory. It supports

1. *Read.*
2. *Write.*

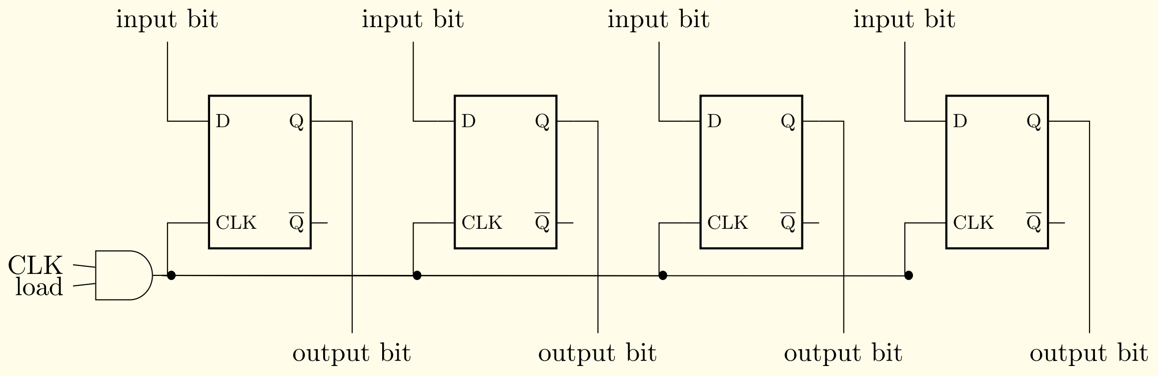


Figure 35: 4-bit register. Since $\overline{Q}$ is redundant we do not consider it as an output in our register.

So we now have the flexibility to construct memory of arbitrary size. The general convention is to make the registers multiples of 2.

Definition 1.13 (Word)

The **word size** of a register w is the number of bits it can hold.

1. 8-bit and 16-bit registers were used in the early days of computing.
2. 32-bit personal computers were introduced in the 1980s, but they are mostly considered obsolete as of 2025. These machines are known as *32-bit machines*.
3. 64-bit personal computers are the most dominant, known as *64-bit machines*.

It is sort of understandable that making word sizes as power of 2 makes things a bit more convenient. They can be stacked and grouped together conveniently, giving us the following familiar terms.

Definition 1.14 (Byte)

A **Byte** is a group of 8-bits.

1.5 Applications

Definition 1.15 (Divide by 2 Chip)

The **divide-by-2 chip** simply splits the frequency of the clock input by 2. It is implemented with a pulse latch—i.e. an edge-triggered D-latch.

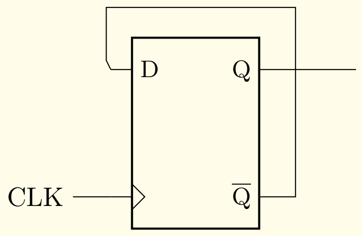


Figure 36: The output $\overline{Q}$ gets rewired into the input D, causing the frequency to slow down.

Example 1.7 (Waveforms of Divide by 2 Latch)

The way the divide-by-2 chip acts on a clock waveform is pretty straightforward.

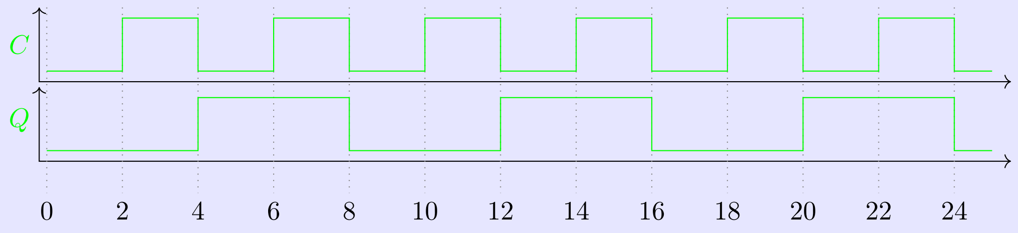


Figure 37

This actually gives us a pretty surprising circuit.

Definition 1.16 (Counter Chip)

A ***n*-bit counter chip** outputs values that increment at every transition of a clock cycle in the following manner.

```
0...000
0...001
0...010
0...011
0...100
...
1...111
0...000
```

It is implemented by stacking n divide-by-2 chips together, composing them to get lower and lower frequencies of the same clock.

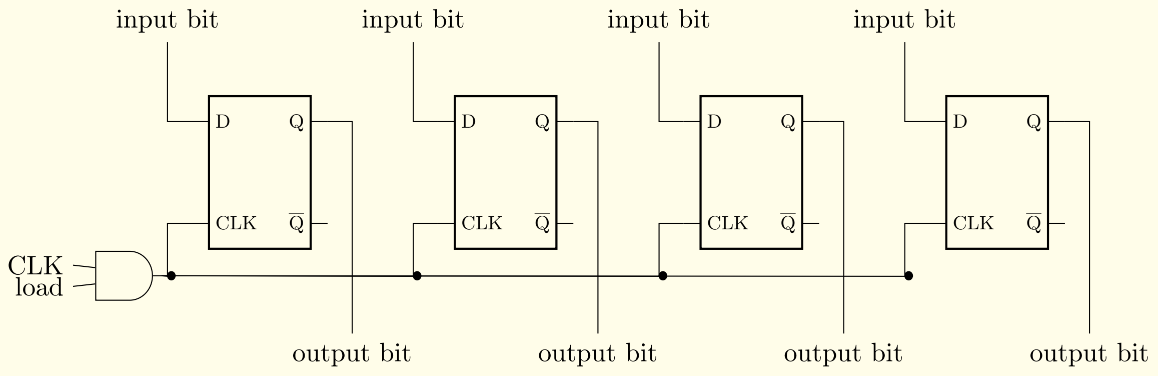


Figure 38: 4-bit register. Since $\overline{Q}$ is redundant we do not consider it as an output in our register. To reset it, we can use the set-reset latches connected to one power source to initialize it to 0.

Example 1.8 (4-Bit Counter Chip)

If we analyze the waveforms of a clock and the effects of a 4-bit counter, note that at every step, we simply divide by 2. However, if we look at the values of Q at every timestep, the oscillations of each divide-by-2 chip result in a counter!

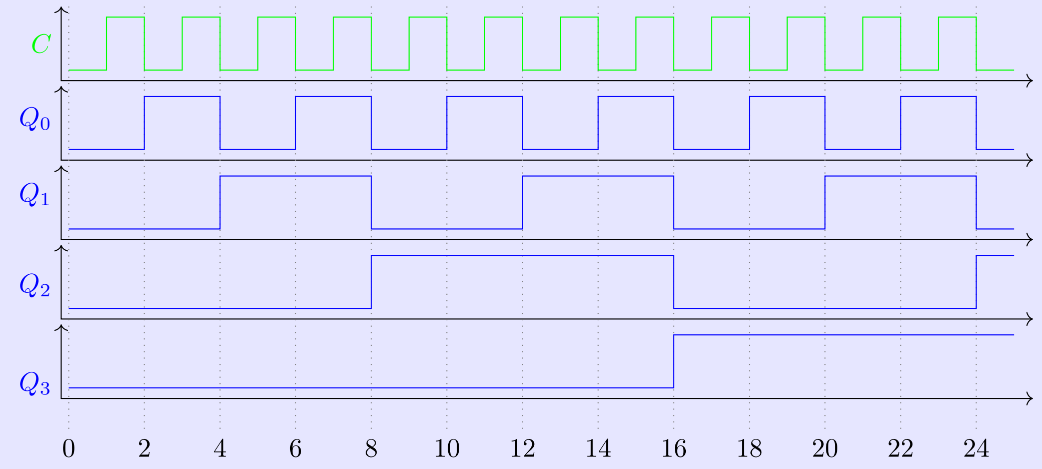


Figure 39: From $t = 0$ to $t = 1$, we have $Q = Q_3Q_2Q_1Q_0 = 0000$. The next time period, we have $Q = 0001$, and so on. This is precisely a counter.

2 Binary Encodings

We have motivated the need for *binary* encodings through the construction of the transistor. In retrospect, we can therefore see why we want to develop a theory around binary alphabets in $\{0,1\}^*$. Now that we know how to work with them, the remaining task of encoding elements of an arbitrary set $S \to \{0,1\}^* = \sqcup_n \{0,1\}^n$ is mathematically trivial.

Definition 2.1 (Representation Scheme)

A **representation scheme** is an encoding of an object s to a unique binary string $E(s) \in \{0,1\}^*$. It is an injective function

$$E : X \longrightarrow \{0,1\}^* \tag{2}$$

Therefore, when we say that a program P takes x as an input, we really mean that P takes as input the *representation of* x as a binary string.

Since $\{0,1\}^*$ is countable, there always exists an injective map $f : S \to \{0,1\}^*$ as long as S is at most countable. But in practicality, we would like to find a good encoding that is easy to work with. Throughout this chapter, we will consider different sets S and introduce the standard encodings for each set.

In order to get into memory, it is helpful to know the theory behind how primitive types are stored in memory.

Definition 2.2 (Collections of Bits)

There are many words that are used to talk about values of different data types:

1. A **bit** (b) is either 0 or 1.
2. A **Hex** (x) is a collection of 4 bits, with a total of $2^4 = 16$ possible values, and this is used since it is easy to read for humans.
3. A **Byte** (B) is a collection of 8 bits or 2 hex, with a total of $2^8 = 256$ possible values, and most computers will work with Bytes as the smallest unit of memory.

Definition 2.3 (Collections of Bytes)

Sometimes, we want to talk about slightly larger collections, so we group them by how many bytes they have. However, note that these may not always be the stated size, depending on what architecture or language you are using. This is more of a general term, and they may have different names in different languages. If there is a difference, we will state it explicitly.

1. A **word** (w) is 2 Bytes.
2. A **long** (l) is 4 Bytes.
3. A **quad** (q) is 8 Bytes.

Try to know which letter corresponds to which structure, since that will be useful in both C and Assembly.

2.1 Naturals/Unsigned and Integers/Signed

Definition 2.4 (Representation of the Naturals)

A representation for natural numbers (note that in this context, $0 \in \mathbb{N}$) is the (non-surjective) regular binary representation denoted

$$NtS : \mathbb{N} \longrightarrow \{0,1\}^* \quad (NtS = \text{"Naturals to Strings"}) \tag{3}$$

recursively defined as

$$NtS(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ NtS(\lceil n/2 \rceil \, parity(n) & n > 1 \end{cases}$$

where given strings $x, y \in \{0,1\}^*$, xy denotes the concatenation of x and y, and $parity : \mathbb{N} \longrightarrow \{0,1\}^*$ is defined

$$parity(n) = \begin{cases} 0 & n \text{ is even} \\ 1 & n \text{ is odd} \end{cases}$$

Since NtS in injective, its inverse $StN : \text{Im} NtS \subset \{0,1\}^* \longrightarrow \mathbb{N}$ is well-defined.

Definition 2.5 (Representation of the Integers)

To construct a representation scheme for $\mathbb{Z}$, we can just add one more binary digit to represent the sign of the number. The binary representation $ZtS : \mathbb{Z} \longrightarrow \{0,1\}^*$ is defined

$$ZtS(m) = \begin{cases} 0 \, NtS(m) & m \geq 0 \\ 1 \, NtS(-m) & m < 0 \end{cases}$$

where NtS is defined as before. Again this function must be injective but need not be surjective.

The most primitive things that we can store are integers. Let us talk about how we represent some of the simplest primitive types in C: unsigned short, unsigned int, unsigned long, unsigned long long.

Definition 2.6 (Unsigned Integer Types in C)

In C, there are several integer types. We use this hierarchical method to give flexibility to the programmer on the size of the integer and whether it is signed or not.

1. An **unsigned short** is 2 bytes long and can be represented as a 4-digit hex or 16 bits, with values in $[0 : 65,535]$. Therefore, say that we have
2. An **unsigned int** is 4 bytes long and can be represented as an 8-digit hex or 32 bits, with values in $[0 : 4,294,967,295]$.
3. An **unsigned long** is 8 bytes and can be represented as an 16-digit hex or 64 bits, but they are only guaranteed to be stored in 32 bits in other systems.
4. An **unsigned long long** is 8 bytes and can be represented as an 16-digit hex or 64 bits, and they are guaranteed to be stored in 64 bits in other systems.

Theorem 2.1 (Bit Representation of Unsigned Integers in C)

To encode a signed integer in bits, we simply take the binary expansion of it.

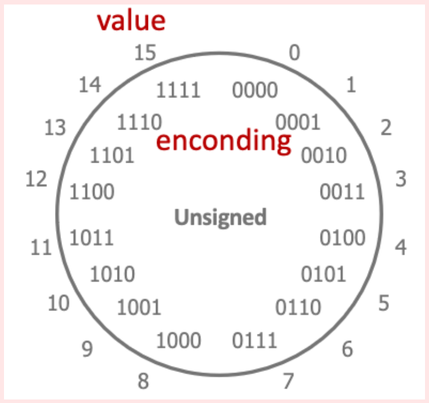


Figure 40: Unsigned encoding of 4-bit integers in C.

Example 2.1 (Bit Representation of Unsigned Integers in C)

We can see for ourselves how these numbers are represented in bits. Printing the values out in binary requires to make new functions, but we can easily convert from hex to binary.

```
int main() {

  unsigned short x = 13;
  unsigned int y = 256;

  printf("%x\n", x);
  printf("%x\n", y);

  return 0;
}
```

```
d
100
.
.
.
.
.
.
.
.
```

So far, the process of converting unsigned numbers to bits seemed simple. Now let's introduce signed integers.

Definition 2.7 (Signed Integer Types in C)

In C, there are several signed integer types. We use this hierarchical method to give flexibility to the programmer on the size of the integer and whether it is signed or not.

1. A **signed short** is 2 bytes long and can be represented as a 4-digit hex or 16 bits, with values in $[-32,768 : 32,767]$.
2. A **signed int** is 4 bytes long and can be represented as an 8-digit hex or 32 bits, with values in $[-2,147,483,648 : 2,147,483,647]$.
3. A **signed long** is 8 bytes and can be represented as an 16-digit hex or 64 bits, but they are only guaranteed to be stored in 32 bits in other systems.
4. A **signed long long** is 8 bytes and can be represented as an 16-digit hex or 64 bits, and they are guaranteed to be stored in 64 bits in other systems.

To store signed integers, it is intuitive to simply take the first (left-most) bit and have that be the sign. Therefore, we lose one significant figure but gain information about the sign. However, this has some problems: first, there are two representations of zeros: -0 and $+0$. Second, the continuity from -1 to 0 is not natural. It is best explained through an example, which doesn't lose much insight into the general case.

Example 2.2 (Problems with the Signed Magnitude)

Say that you want to develop the signed magnitude representation for 4-bit integers in C. Then, you can imagine the following diagram to represent the numbers.

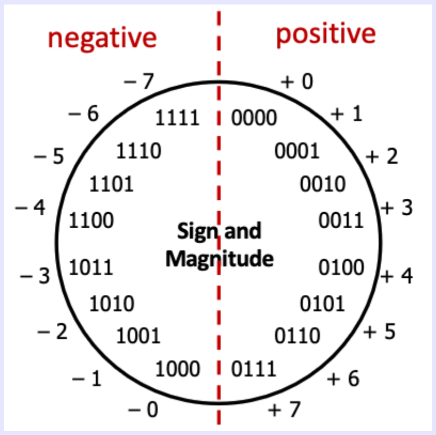


Figure 41: Signed magnitude encoding of 4-bit integers in C.

You can see that there are some problems:

1. There are two representations for 0, which is 0000 and 1000.
2. -1 (1001) plus 1 becomes -2 (1010).
3. The lowest number -7 (1111) plus 1 goes to 0 (0000) when it should go to -6 (1100).
4. The highest number 7 (0111) plus 1 goes to 0 (1000).

An alternative way is to use the two's complement representation, which solves both problems and makes it more natural.

Theorem 2.2 (Bit Representation of Signed Integers in C)

The **two's complement** representation is a way to represent signed integers in binary. It is defined as follows. Given that you want to store a decimal number p in n bits,

1. If p is positive, then take the binary expansion of that number, which should be at most $n-1$ bits (no overflow), pad it with 0s on the left.
2. If p is negative, then you can do two things: First, take the binary expansion of the positive number, flip all the bits, and add 1. Or second, represent $p = q - 2^n$, take the binary representation of q in $n-1$ bits, and add a 1 to the left.

If you have a binary number $b = b_n b_{n-1} \cdots b_1$ then to convert it to a decimal number, you simply calculate

$$q = -b_n 2^{n-1} + b_{n-1} 2^{n-2} + \cdots + b_1 \tag{4}$$

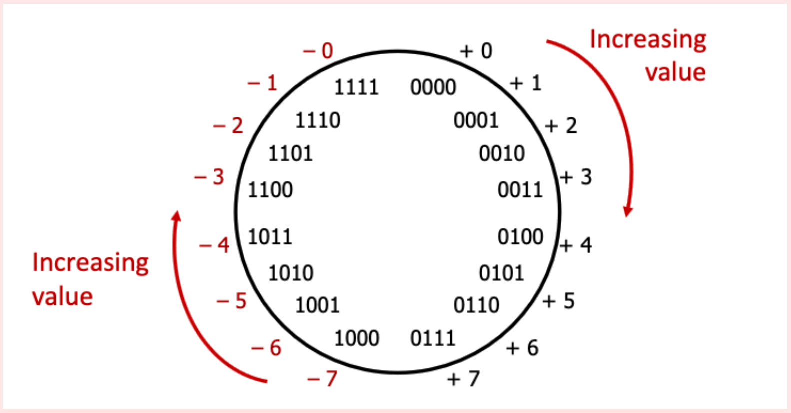


Figure 42: Two's complement encoding of 4-bit integers in C.

Example 2.3 (Bit Representation of Signed Integers in C)

We can see for ourselves how these numbers are represented in bits.

```c
int main() {

  short short_pos = 13;
  short short_neg = -25;
  int int_pos = 256;
  int int_neg = -512;

  printf("%x\n", short_pos);
  printf("%x\n", short_neg);
  printf("%x\n", int_pos);
  printf("%x\n", int_neg);

  return 0;
}
```

```
d
ffe7
100
fffffe00
.
.
.
.
.
.
.
.
.
.
```

```c
#include<stdio.h>
#include<stdbool.h>

int main() {
  printf("%lu\n", sizeof(bool));
  printf("%lu\n", sizeof(short));
  printf("%lu\n", sizeof(int));
  printf("%lu\n", sizeof(long));
  printf("%lu\n", sizeof(long long));
  return 0;
}
```

```
1
2
4
8
8
.
.
.
.
.
.
```

Figure 43: Size of various integer types in C with the `sizeof`.

2.1.1 Arithmetic Operations on Binary Numbers

Theorem 2.3 (Inversion of Binary Numbers)

Given a binary number p, to compute $-p$, simply invert the bits and add 1.

Theorem 2.4 (Addition and Subtraction of Binary Numbers)

Given two binary numbers p and q.

1. To compute $p+q$, simply add the numbers together as you would in base 10, but carry over when the sum is greater than 1.
2. To compute $p-q$, you can invert q to $-q$ and compute $p+(-q)$.

2.2 Rationals and Countable Sets

When representing rational numbers, we cannot simply concatenate the numerator and denominator as such

$$a/b \mapsto ZtS(a)\, ZtS(b)$$

since this map is not surjective (and may overlap with other integers).

Definition 2.8 (Representation of Rationals)

To represent a rational number a/b, we create a separator symbol $|$ and map the rational number as below in the alphabet $\{0,1,|\}$.

$$q : a/b \mapsto ZtS(a)|ZtS(b)$$

Then, we use a second map that goes through each digit in z and is defined

$$p : \{0,1,|\} \longrightarrow \{00,11,01\} \subset \{0,1\}^2,\ p(n) = \begin{cases} 00 & n=0 \\ 11 & n=1 \\ 01 & n=| \end{cases}$$

Therefore, p maps the length n string $z \in \{0,1\}^*$ to the length $2n$ string $\omega \in \{0,1\}^*$. The representation scheme for $\mathbb{Q}$ is simply

$$QtS \equiv p \circ q$$

Example 2.4 ()

Given the rational number $-5/8$,

$$\frac{-5}{8} \mapsto 1101|01000 \mapsto 11110011010011000000$$

This same idea of using separators and compositions of injective functions can be used to represent arbitrary n-tuples of strings (since a finite Cartesian product of countable sets is also countable).

Theorem 2.5 (Representation of Vectors)

All vectors, matrices, and tensors over the field $\mathbb{Q}$ are representable.

Proof.

For vectors, we can simply create another separator symbol $\cdot$ and have the initial mapping q map to a string over the alphabet $\{0, 1, |, \cdot\}$, which injectively maps to $\{00, 01, 10, 11\}$. For tensors, create more separator symbols and map them to a sufficiently large set (which can be extended arbitrarily). For example, to perhaps $\{000, 001, ..., 111\}$.

Corollary 2.1 (Representation of Graphs)

Directed graphs, which can be represented with their adjacency matrices, can therefore be represented with binary strings.

Theorem 2.6 (Representation of Images)

Every finite-resolution image can be represented as a binary number.

Proof.

Since we can interpret each image as a matrix where each element (a pixel) is a color, and since each color can be represented as a 3-tuple of rational numbers corresponding to the intensities of red, green, and blue (for humans, we can restrict it to three primary colors), all images can eventually be decomposed into binary strings.

2.3 Floats

Theorem 2.7 (Representation of Reals)

There exists no representation of the reals

$$NtR : \mathbb{R} \longrightarrow \{0, 1\}^* \tag{5}$$

Proof.

By Cantor's theorem, the reals are uncountable. That is, there does not exist a surjective function $NtR : \mathbb{N} \longrightarrow \mathbb{R}$. The implies the nonexistence of an injective inverse; that is, there does not exist an injective function

$$RtS : \mathbb{R} \longrightarrow \{0, 1\}^*$$

However, since $\mathbb{Q}$ is dense in $\mathbb{R}$, we can approximate every real number x by a rational number a/b to arbitrary accuracy. There are multiple ways to construct these approximations (decimal approximation up to kth digit, finite continued fractions, truncated infinite series, etc.), but computers use the *floating-point approximation*.

Definition 2.9 (Floating-Point Representation)

The **floating-point representation scheme** of a real number $x \in \mathbb{R}$ is its approximation as a number of the form

$$\sigma b \cdot 2^e$$

where $\sigma \in \{0, 1\}$ determines the sign of the representation of x, e is a (potentially negative) integer, and b is a rational number between 1 and 2 expressed as a binary fraction

$$1.b_0 b_1 b_2 ... b_k = 1 + \frac{b_1}{2} + \frac{b_2}{4} + ... + \frac{b_k}{2^k}, \quad b_i \in \{0, 1\}$$

where the number k is fixed (determined by the desired accuracy; greater k implies more digits and better accuracy). The $\sigma b \cdot 2^e$ closest to x is the *floating-point representation*, or *approximation*, of x. We can think of σ determining the sign, e the order of magnitude (in base 2) of x, and b the value of the number scaled down to a value in $[1, 2)$, called the *mantissa*.

Definition 2.10 (Floating Point Types in C)

In C, there are several floating point types. We use this hierarchical method to give flexibility to the programmer on the size of the integer and whether it is signed or not.

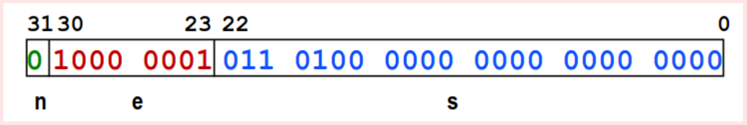
1. A **float** is 4 bytes long and can be represented as an 8-digit hex or 32 bits, with values in $[1.2 \times 10^{-38} : 3.4 \times 10^{38}]$.
2. A **double** is 8 bytes long and can be represented as an 16-digit hex or 64 bits, with values in $[2.3 \times 10^{-308} : 1.7 \times 10^{308}]$.
3. A **long double** is 8 bytes and can be represented as an 16-digit hex or 64 bits, but they are only guaranteed to be stored in 80 bits in other systems.

Theorem 2.8 (Bit Representation of Floating Point Types in C)

Floats are actually like signed magnitude. We have

$$(-1)^n \times 2^{e-127} \times 1.s \tag{6}$$

where



Doubles encode 64 bits, so not we have exponent having 11 bits (so bias is not 1023) and 52 bits for mantissa.

2.4 Characters

Definition 2.11 (Booleans in C)

The most basic type is the boolean, which is simply a bit. In C, it is represented as `bool`, and it is either `true` (1) or `false` (0).

We can manually check the size of the boolean type in C with the following code.

```c
#include<stdio.h>
#include<stdbool.h>

int main() {
  printf("%lu\n", sizeof(bool));
  return 0;
}
```

```
1
.
.
.
.
.
.
```

Figure 44: We can verify the size of various primitive data types in C with the `sizeof` operator.

Note that **it does not make sense to have a string without knowing what encoding it uses**. We can't just assume that every plaintext is in ASCII, since there are hundreds of extended ASCII encodings.

If you have a string, in memory, in a file, or in an email message, you have to know what encoding it is in or you cannot interpret it or display it to users correctly.

For example, when you are sending an email, Gmail is the only client that automatically converts your text to UTF-8, regardless of what you set in the header. The browser also uses a certain encoding, which can be accessed (and changed) under the "view" tab.

2.4.1 ASCII

Definition 2.12 (ASCII)

The **ASCII** (also called US-ASCII) code, which stands for American Standard Code for Information Interchange is a 7 bit character code where every single bit represents a unique character. ASCII codes represent text in computers, telecommunications equipment, and other devices. Most modern character-encoding schemes are based on ASCII, although they support many additional characters. The first 32 characters are called the *control characters*: codes originally intended not to represent printable information, but rather to control devices (such as printers) that make use of ASCII, or to provide meta-information about data streams. For example, character 10 (decimal) represents the "line feed" function (which causes a printer to advance its paper) and character 8 represents "backspace." Except for the control characters that prescribe elementary line-oriented formatting, ASCII does not define any mechanism for describing the structure or appearance of text within a document.

Dec	Oct	Hex	Bin	Symbol	Description
0	000	00	0000000	NULL	Null char
1	001	01	0000001	SOH	Start of Heading
2	002	02	0000010	STX	Start of Text
3	003	03	0000011	ETX	End of Text
4	004	04	0000100	EOT	End of Transmission
5	005	05	0000101	ENQ	Enquiry
6	006	06	0000110	ACK	Acknowledgement
7	007	07	0000111	BEL	Bell
8	010	08	0001000	BS	Back Space
9	011	09	0001001	HT	Horizontal Tab
10	012	0A	0001010	LF	Line Feed
11	013	0B	0001011	VT	Vertical Tab
12	014	0C	0001100	FF	Form Feed
13	015	0D	0001101	CR	Carriage Return
14	016	0E	0001110	SO	Shift Out/X-On
15	017	0F	0001111	SI	Shift In/X-Off
16	020	10	0010000	DLE	Data Line Escape
17	021	11	0010001	DC1	Device Control 1
18	022	12	0010010	DC2	Device Control 2
19	023	13	0010011	DC3	Device Control 3
20	024	14	0010100	DC4	Device Control 4
21	025	15	0010101	NAK	Negative Acknowledgement
22	026	16	0010110	SYN	Synchronous Idle
23	027	17	0010111	ETB	End of Transmit Block
24	030	18	0011000	CAN	Cancel
25	031	19	0011001	EM	End of Medium
26	032	1A	0011010	SUB	Substitute
27	033	1B	0011011	ESC	Escape
28	034	1C	0011100	FS	File Separator
29	035	1D	0011101	GS	Group Separator
30	036	1E	0011110	RS	Record Separator
31	037	1F	0011111	US	Unit Separator

The rest of the characters are the ASCII printable characters.

Dec	Oct	Hex	Bin	Sym	Description	Dec	Oct	Hex	Bin	Sym	Description
32	040	20	0100000		Space	80	120	50	1010000	P	Uppercase P
33	041	21	0100001	!	Exclamation	81	121	51	1010001	Q	Uppercase Q
34	042	22	0100010	"	Double quotes	82	122	52	1010010	R	Uppercase R
35	043	23	0100011	#	Number	83	123	53	1010011	S	Uppercase S
36	044	24	0100100	$	Dollar	84	124	54	1010100	T	Uppercase T
37	045	25	0100101	%	Per cent sign	85	125	55	1010101	U	Uppercase U
38	046	26	0100110	&	Ampersand	86	126	56	1010110	V	Uppercase V
39	047	27	0100111	’	Single quote	87	127	57	1010111	W	Uppercase W
40	050	28	0101000	(	Open paren.	88	130	58	1011000	X	Uppercase X
41	051	29	0101001	)	Closed paren.	89	131	59	1011001	Y	Uppercase Y
42	052	2A	0101010	*	Asterisk	90	132	5A	1011010	Z	Uppercase Z
43	053	2B	0101011	+	Plus	91	133	5B	1011011	[	Opening bracket
44	054	2C	0101100	,	Comma	92	134	5C	1011100	\	Backslash
45	055	2D	0101101	-	Hyphen	93	135	5D	1011101	]	Closing bracket
46	056	2E	0101110	.	Period	94	136	5E	1011110	^	Caret
47	057	2F	0101111	/	Slash	95	137	5F	1011111	_	Underscore
48	060	30	0110000	0	Zero	96	140	60	1100000	‘	Grave accent
49	061	31	0110001	1	One	97	141	61	1100001	a	Lowercase a
50	062	32	0110010	2	Two	98	142	62	1100010	b	Lowercase b
51	063	33	0110011	3	Three	99	143	63	1100011	c	Lowercase c
52	064	34	0110100	4	Four	100	144	64	1100100	d	Lowercase d
53	065	35	0110101	5	Five	101	145	65	1100101	e	Lowercase e
54	066	36	0110110	6	Six	102	146	66	1100110	f	Lowercase f
55	067	37	0110111	7	Seven	103	147	67	1100111	g	Lowercase g
56	070	38	0111000	8	Eight	104	150	68	1101000	h	Lowercase h
57	071	39	0111001	9	Nine	105	151	69	1101001	i	Lowercase i
58	072	3A	0111010	:	Colon	106	152	6A	1101010	j	Lowercase j
59	073	3B	0111011	;	Semicolon	107	153	6B	1101011	k	Lowercase k
60	074	3C	0111100	<	Less than	108	154	6C	1101100	l	Lowercase l
61	075	3D	0111101	=	Equals	109	155	6D	1101101	m	Lowercase m
62	076	3E	0111110	>	Greater than	110	156	6E	1101110	n	Lowercase n
63	077	3F	0111111	?	Question mark	111	157	6F	1101111	o	Lowercase o
64	100	40	1000000	@	At symbol	112	160	70	1110000	p	Lowercase p
65	101	41	1000001	A	Uppercase A	113	161	71	1110001	q	Lowercase q
66	102	42	1000010	B	Uppercase B	114	162	72	1110010	r	Lowercase r
67	103	43	1000011	C	Uppercase C	115	163	73	1110011	s	Lowercase s
68	104	44	1000100	D	Uppercase D	116	164	74	1110100	t	Lowercase t
69	105	45	1000101	E	Uppercase E	117	165	75	1110101	u	Lowercase u
70	106	46	1000110	F	Uppercase F	118	166	76	1110110	v	Lowercase v
71	107	47	1000111	G	Uppercase G	119	167	77	1110111	w	Lowercase w
72	110	48	1001000	H	Uppercase H	120	170	78	1111000	x	Lowercase x
73	111	49	1001001	I	Uppercase I	121	171	79	1111001	y	Lowercase y
74	112	4A	1001010	J	Uppercase J	122	172	7A	1111010	z	Lowercase z
75	113	4B	1001011	J	Uppercase K	123	173	7B	1111011	{	Opening brace
76	114	4C	1001100	L	Uppercase L	124	174	7C	1111100	|	Vertical bar
77	115	4D	1001101	M	Uppercase M	125	175	7D	1111101	}	Closing brace
78	116	4E	1001110	N	Uppercase N	126	176	7E	1111110	~	Tilde
79	117	4F	1001111	O	Uppercase O	127	177	7F	1111111		Delete

The **Extended ASCII** (EASCII or high ASCII) character encodings are 8-bit or larger encodings that include the standard 7-bit ASCII characters, plus additional characters. Note that this does not mean that the standard ASCII coding has been updated to include more than 128 characters nor does it mean that there is an universal extension to the original ASCII coding. In fact, there are several (over 100) extended ASCII encodings.

With the creation of the 7-bit ASCII format, increased need for more letters and symbols (such as characters in other languages or more punctuation/mathematical symbols). With better computers and software, it became obvious that they could handle text that uses 256-character sets at almost no additional cost in programming or storage. The 8-bit format would allow ASCII to be used unchanged and provide 128 more characters.

But even 256 characters is still not enough to cover all purposes, all languages, or even all European languages, so the emergence of *many* ASCII-derived 8-bit character sets was inevitable. Translating between these sets (*transcoding*) is complex, especially if a character is not in both sets and was often not done, producing **mojibake** (semi-readable text resulting from text being decoded using an unintended character encoding. The result is a systematic replacement of symbols with completely unrelated ones, often from a different writing system). ASCII can also be used to create graphics, commonly called **ASCII art**.

But ASCII isn't enough. We have lots of languages with lots of characters that computers should ideally display. Unicode assigns each character a unique number, or code print. Computers deal with such numbers as bytes: 8-bit computers would treat an 8-bit byte as the largest numerical unit easily represented on the hardware, 16-bit computers would expand that to 2 bytes, and so forth. Old character encodings like ASCII are from the (pre-) 8-bit era, and try to cram the dominant language in computing at the time, i.e. English, into numbers ranging from 0 to 127 (7 bits). When ASCII got extended by an 8th bit for other non-English languages, the additional 128 numbers/code points made available by this expansion would be mapped to different characters depending on the language being displayed. The **ISO-8859** standards are the most common forms of this mapping:

1. **ISO-8859-1**
2. **ISO-8859-15**, also called **ISO-Latin-1**

But that's not enough when you want to represent characters from more than one language, so cramming all available characters into a single byte just won't work. The following shows ways to do this (that is compatible with ASCII).

2.4.2 ISO-10646, UCS

We can simply expand the value range by adding more bits. The UCS-2 uses 2 bytes (or 16 bits) and UCS-4 uses 4 bytes (32 bits). However, these codings suffer from inherently the same problem as ASCII and ISO-8859 standards, as their value range is still limited, even if the limit is vastly higher. Note that these encode from the ISO-10646, which defines several character encoding forms for the Universal Coded Character Set.

1. UCS-2 can store $2^{16} = 65,536$ characters.
2. UCS-4 can store $2^{32} = 4,294,967,296$ characters.

Notice that UCS encoding has a fixed number of bytes per character, which means that UCS-2 stores each character in 2 bytes, and UCS-4 stores each character in 4 bytes. This is different from **UTF-8** encoding.

ISO 10646 and Unicode have an identical repertoire and numbers—the same characters with the same numbers exist on both standards, although Unicode releases new versions and adds new characters more often. Unicode has rules and specifications outside the scope of ISO 10646. ISO 10646 is a simple character map, an extension of previous standards like ISO 8859. In contrast, Unicode adds rules for collation, normalization of forms, and the bidirectional algorithm for right-to-left scripts such as Arabic and Hebrew. For interoperability between platforms, especially if bidirectional scripts are used, it is not enough to support ISO 10646; Unicode must be implemented.

2.4.3 Unicode, UTF-8

Unicode is the universal character encoding, maintained by Unicode Consortium, and it covers the characters for all the writing systems of the world, modern and ancient. It also includes technical symbols, punctuation, and many other characters used in writing text. As of Unicode Version 13.0, the Unicode standard contains 143,859 characters, stored in the format U+****, where **** is a number in hexadecimal notation. Notice that these ones are not fixed in the number of bits; that is,

$$\text{U+27BD and U+1F886}$$

are perfectly viable representations of characters in Unicode. Even though only 143,859 characters are in use, Unicode currently allows for 1,114,112 ($16^5 + 16^4$) code values, and assigns codes covering nearly all modern text writing systems, as well as many historical ones and for many non-linguistic characters such as printer's dingbats, mathematical symbols, etc.

*Note that Unicode, along with ISO-10646, is a standard that assigns a name and a value (**Character Code** or **Code-Point**) to each character in its repertoire.* However, the Unicode format must be encoded in a binary format for the computer to understand. When you save a document, the text editor has to explicitly set its encoding to be UTF-8 (or whatever other format) the user wants it to be. Also, when a text editor program reads a file, it needs to select a text encoding scheme to decode it correctly. Even further, when you are typing and entering a letter, the text editor needs to know what scheme you use so that it will save it correctly. Therefore, *UTF-8 encoding is a way to represent these characters digitally in computer memory.* The way that **UTF-8** encodes characters is with the following format:

```
1st Byte    2nd Byte    3rd Byte    4th Byte    Number of Free Bits
0xxxxxxx                                                7
110xxxxx    10xxxxxx                                 (5+6)=11
1110xxxx    10xxxxxx    10xxxxxx                   (4+6+6)=16
11110xxx    10xxxxxx    10xxxxxx    10xxxxxx    (3+6+6+6)=21
```

From this, we can see that UTF-8 uses a variable number of bytes per character. All UTF encodings work in roughly the same manner: you choose a unit size, which for UTF-8 is 8 bits, for UTF-16 is 16 bits, and for UTF-32 is 32 bits. The standard then defines a few of these bits as *flags* (e.g. the 0, 110, 1110, 11110, ...). If they're set, then the next unit in a sequence of units is considered part of the same character. If they're not set, this unit represents one character fully. Thus, the most common (English) characters only occupy one byte in UTF-8 (two in UTF-16, 4 in UTF-32), but other language characters can occupy more bytes. We can see that UTF-8 can encode up to (and slightly more than) $2^{21} = 2,097,152$ characters. UTF-8 is by far the most common encoding for the World Wide Web, accounting for 96.0% of all web pages, and up to 100% for some languages, as of 2021.

For example, let's take a random character, say with the Unicode value to be U+6C49. Then, we convert this to binary to get

01101100 01001001

But we can't just store this because this isn't a prefix-free notation. This is when UTF-8 is needed. Using the chart above, we need to prefix our character with some headers/flags. The binary Unicode value of the character is 16 bits long, so we can store it in 3 bytes (in the format of the third row) as it provides enough space. The headers are not bolded, while the binary values added are.

1110**0110** 10**110001** 10**001001**

We can take another example of a character with the Unicode value U+1F886. Converting to binary gets

0001 1111 1000 1000 0110

There are 20 bits, so we will need to store it in 4 bytes (in the format of fourth row) as it provides enough space (21). We convert the 20-bit-long binary Unicode value to a 21-bit-long value (so that it is compatible with the 21 free bits) to get

0 0001 1111 1000 1000 0110

Encoding it in UTF-8 in 4 bytes gives

11110**000** 10**011111** 10**100010** 10**000110**

There is no need to go beyond 4 bytes since every Unicode value will have at most 5 hexadecimal digits (since $16^5 = 1,048,576$, which is far more than the number of characters there are). There is also another, obsolete, encoding used called the **UTF-7**.

Both the UCS and UTF standards encode the code points as defined in Unicode. In theory, those encodings could be used to encode any number (within the range the encoding supports) - but of course these encodings were made to encode Unicode code points. Windows handles so-called "Unicode" strings as UTF-16 strings, while most UNIXes default to UTF-8 these days. Communications protocols such as HTTP tend to work best with UTF-8, as the unit size in UTF-8 is the same as in ASCII, and most such protocols were designed in the ASCII era. On the other hand, UTF-16 gives the best average space/processing performance when representing all living languages.

While UTF-7, 8, 16, and 32 all have the nice property of being able to store *any* code point correctly, there are hundreds of encodings that can only store a set amount of characters. If there's no equivalent for the Unicode code point you're trying to represent in the encoding you're trying to represent it in, you usually get a little question mark: ? For example, trying to store Russian or Hebrew letters in these encodings results in a bunch of question marks.

2.4.4 Text Files

The ASCII character set is the most common compatible subset of character sets for English-language text files, and is generally assumed to be the default file format in many situations.

In the Mac, checking the character encoding of a text file can be done with the command

```
>>>file -I filename.txt
filename.txt: text/plain; charset=us-ascii
```

ASCII covers American English, but for the British Pound sign, the Euro sign, or characters used outside English, a richer character set must be used. In many systems, this is chosen based on the default setting on the computer it is read on. Prior to UTF-8, this was traditionally single-byte encodings (such as ISO-8859-1 through ISO-8859-16) for European languages and wide character encodings for Asian languages. However, most computers use UTF-8 as the natural extension. We can check this firsthand by inputting a non-ASCII character in filename.txt, which would result in

```
>>>file -I filename.txt
filename.txt: text/plain; charset=utf-8
```

Because encodings necessarily have only a limited repertoire of characters, often very small, many are only usable to represent text in a limited subset of human languages. Unicode is an attempt to create a common standard for representing all known languages, and most known character sets are subsets of the very large Unicode character set. Although there are multiple character encodings available for Unicode, the most common is UTF-8, which has the advantage of being backwards-compatible with ASCII; that is, every ASCII text file is also a UTF-8 text file with identical meaning. UTF-8 also has the advantage that it is easily auto-detectable. Thus, a common operating mode of UTF-8 capable software, when opening files of unknown encoding, is to try UTF-8 first and fall back to a locale dependent legacy encoding when it definitely isn't UTF-8.

Because of their simplicity, text files are commonly used for storage of information. When data corruption occurs in a text file, it is often easier to recover and continue processing the remaining contents. A disadvantage of text files is that they usually have a low entropy, meaning that the information occupies more storage than is strictly necessary. A simple text file may need no additional metadata (other than knowledge of its character set) to assist the reader in interpretation. A text file may contain no data at all, which is a case of zero-byte file.

2.5 Representation of General Sets

Let there exist some set $\mathcal{O}$ consisting of objects. Then, a representation scheme for representing objects in $\mathcal{O}$ consists of an *encoding* function that maps an object in $\mathcal{O}$ to a string, and a *decoding* function that decodes a string back to an object in $\mathcal{O}$.

Definition 2.13 ()

Let $\mathcal{O}$ be any set. A *representation scheme for* $\mathcal{O}$ is a pair of functions E, D where

$$E : \mathcal{O} \longrightarrow \{0,1\}^*$$

is an injective function, and the induced mapping D is restriction of the inverse of E to the image of E.

$$D : \text{Im}(E) \subset \{0,1\}^* \longrightarrow \mathcal{O}$$

This means that $(D \circ E)(o) = o$ for all $o \in \mathcal{O}$. E is known as the *encoding function* and D is known as the *decoding function*.

Definition 2.14 (Prefix)

For two strings y, y', y is a prefix of y' if y' "starts" with y. That is, y is a **prefix** of y' if $|y| \leq |y'|$ and for every $i < |y|$, $y'_i = y_i$.

With this, we can define the concept of prefix free encoding.

Definition 2.15 ()

Let $\mathcal{O}$ be a nonempty set and $E : \mathcal{O} \longrightarrow \{0,1\}^*$ be a function. E is **prefix-free** if $E(o)$ is nonempty for every $o \in \mathcal{O}$ and there does not exist a distinct pair of objects $o, o' \in \mathcal{O}$ such that $E(o)$ is a prefix of $E(o')$.

Being prefix-free is a nice property that we would like an encoding to have. Informally, this means that no string x representing an object o is an initial substring of string y representing a different object o. This means that we can simply represent a *list* of objects simply by concatenating the representations of all the list members and still get a valid, injective representation. We formalize this below.

Theorem 2.9 ()

Suppose that $E : \mathcal{O} \longrightarrow \{0,1\}^*$ is prefix free. Then the following map

$$\overline{E} : \mathcal{O}^* \longrightarrow \{0,1\}^*$$

over all finite length tuples of elements in $\mathcal{O}$ is injective, where for every $o_0, o_1, ..., o_{k-1} \in \mathcal{O}^*$, we define $\overline{E}$ to be the simple concatenation of the separate encodings of o_i:

$$\overline{E}(o_0, ..., o_{k-1}) \equiv E(o_0)E(o_1)...E(o_{k-1})$$

Even if the representation E of objects in $\mathcal{O}$ is prefix free, this does not imply that our representation $\overline{E}$ of *lists* of such objects will be prefix free as well. In fact, it won't be, since for example, given three objects o, o', o'', the representation of the list (o, o') will be a prefix of the representation of the list (o, o', o'').

However, it turns out that in fact we can transform *every* representation into prefix free form, and so will be able to use that transformation if needed to represents lists of lists, lists of lists of lists, and so on.

Some natural representations are prefix free. For example, every *fixed output length* representation (i.e. an injective function $E : \mathcal{O} \longrightarrow \{0,1\}^n$) is automatically prefix-free, since a string x can only be a prefix of an equal length x' if x and x' are identical. Moreover, the approach that was used for representing rational numbers can be used to show the following lemma.

Lemma 2.1 ()

Let $E : \mathcal{O} \longrightarrow \{0,1\}^*$ be a one-to-one function. Then there is a one-to-one prefix-free encoding $\overline{E}$ such that

$$|\overline{E}(o)| \leq 2|E(o)| + 2 \tag{7}$$

for every $o \in \mathcal{O}$.

Proof.

The general idea is the use the map $0 \mapsto 00$, $1 \mapsto 11$ to "double" every bit in the string x and then mark the end of the string by concatenating to it the pair 01. If we encode a string x in this way, it ensures that the encoding of x is never a prefix of the encoding of a distinct string x'. (Note that this is not the only or even the best way to transform an arbitrary representation into prefix-free form.)

3 Combinational Logic

Talk about how to construct arithmetic operations with these gates such as adding two integers or multiplying them, and not just that, but other operations that we may need in a programming language.

3.1 Multi-Bit Gates

Note that we can naturally work with multiple bits. This could mean a few things for—say, an AND gate.

1. AND can take in multiple gates.
2.

Definition 3.1 (Multi-Bit NOT Gate)

Definition 3.2 (Multi-Bit AND Gate)

Definition 3.3 (Multi-Bit OR Gate)

Definition 3.4 (Multi-Bit NAND Gate)

Definition 3.5 (Multi-Bit XOR Gate)

Definition 3.6 (Multi-Bit Multiplexor Gate)

Definition 3.7 (Multi-Bit Demultiplexor Gate)

3.2 Addition and Subtraction

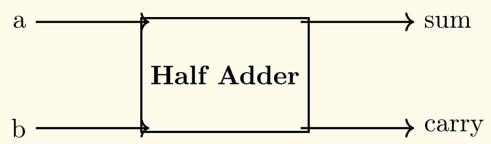
We present a hierarchy of three adders, leading to a multi-bit adder chip. Note that every single chip here represents a finite function, and so from universality of AON gates we know that an implementation is definitely possible.

Definition 3.8 (Half-Adder Chip)

A **half-adder** is designed to add two bits.

Inputs		Outputs	
a	**b**	**carry**	**sum**
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(a) Truth table for half adder.

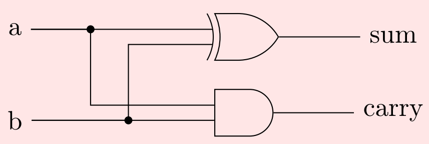


(b)

Figure 45: Chip diagram for half adder.

Theorem 3.1 (Implementation of Half-Adder)

To construct this chip, note that the way sum and carry acts on a, b is identical to the standard XOR(a, b) and AND(a, b) functions.



(a)

```
module half_adder(
  input a, b,
  output s, c
);
  assign s = a ^ b;
  assign c = a & b;
endmodule
```

(b) HDL implementation.

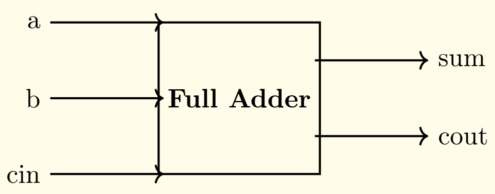
Figure 46

Definition 3.9 (Full-Adder)

Now that we know how to add two bits, a **full-adder chip** allows us to add 3 bits.

Inputs			Outputs	
a	**b**	**cin**	**cout**	**sum**
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

(a) Truth table for full adder.



(b) Block diagram for full adder.

Figure 47: Chip diagram for full adder.

Theorem 3.2 (Implementation of Full-Adder)

We can implement a full adder with 2 half-adders and an OR gate.

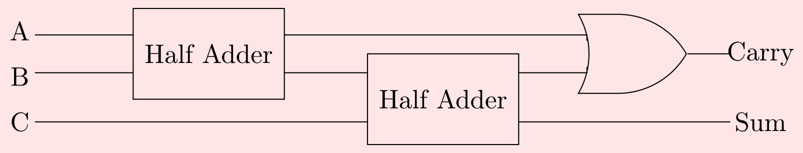


Figure 48

Definition 3.10 (N-Bit Adder)

Usually, N is $16, 32, 64$, or 128.

Theorem 3.3 (Implementation of N-Bit Addition)

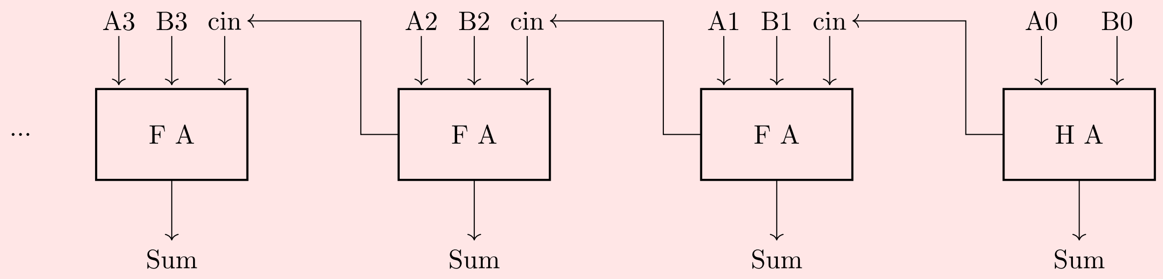


Figure 49: Ripple carry adder for the last 4 significant bits of two N-bit numbers.

Corollary 3.1 (Implemention for N-Bit Subtraction)

This is a standard construction and a goods start, but there are a few pitfalls of this. First, this does not detect nor handle overflows after adding. This will be handled at the operating system level. Second, addition is limited in that we can only apply it for precisely 2 arguments.

Ripple carry, carry select, carry look ahead adder to make it parallel (lecture 4)

Example 3.1 (More Arguments for Binary Addition)

Note that the full adder—which takes in 3 bits—was designed so that there is enough space for each digit of the 2 inputs, plus a potential carry. If there were 3 inputs, then the full adder would need to support 4 inputs. Even worse, if we have $1 + 1 + 1 + 1 = 100$, then our carry digit will be greater than 1 digit, which messes things up even more.

Finally, note that this is not a very efficient way to add because there are delays as the carry bit propagates from the least significant to the most significant bit pair. We can improve this with carry look-ahead techniques.

3.3 Comparator

3.4 Multiplication

Booths algo to do multiplication fast with bitshifts and addition

Theorem 3.4 (Implementation of Bitshift Operations)

The reason bitshift is introduced first is that in binary, bit-shifting is equivalent to multiplication!

Theorem 3.5 (Implementation of Multiplication in Circuits)

Theorem 3.6 (Implementation of Moving Data in Circuits)

3.5 Arithmetic Logical Unit (ALU)

3.6 Conditionals

We also want some sort of conditionals. This then can be used to implement loops by checking some conditional.

Theorem 3.7 (Implementation of Conditionals in Circuits)

4 Memory Banks

Definition 4.1 (Memory)

The **memory** is where the computer stores data and instructions, which can be though of as a giant array of memory addresses, with each containing a byte. This data consists of graphical things or even instructions to manipulate other data. It can be visualized as a long array of boxes that each have an **address** (where it is located) and **contents** (what is stored in it).
Memory simply works as a bunch of bits in your computer with each bit having some memory address, which is also a bit. For example, the memory address `0b0010` (2) may have the bit value of `0b1` (1) stored in it.

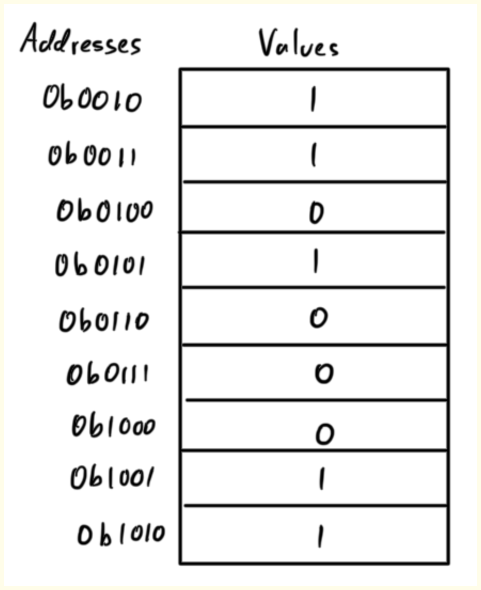


Figure 50: Visualization of memory as a long array of boxes of bits.

However, computers do not need this fine grained level of control on the memory, and they really work at the Byte level rather than the bit level. Therefore, we can visualize the memory as a long array of boxes indexed by *Bytes*, with each value being a byte as well. In short, the memory is **byte-addressable**. In certain arthitectures, some systems are **word-addressable**, meaning that the memory is addressed by words, which are 4 bytes.[a]

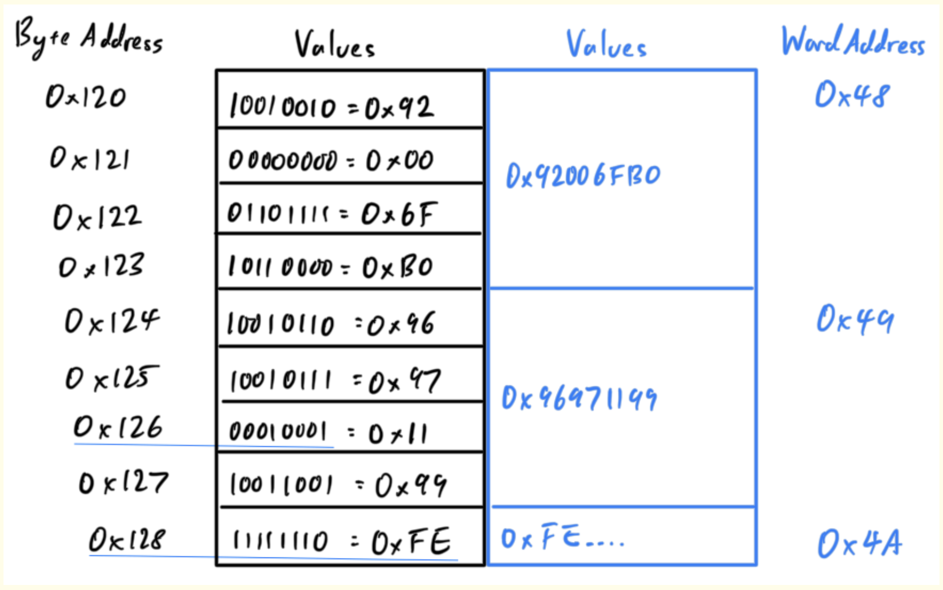


Figure 51: Visualization of memory as a long array of boxes of bytes. Every address is a byte and its corresponding value at that address is also a byte, though we represent it as a 2-digit hex.

[a]Note that in here the size of a word is 2 bytes rather than 4 as stated above. This is just how it is defined in some `x86` architectures.

It is intuitive to think that given some multi-byte object like an `int` (4 bytes), the beginning of the int would be the lowest address and the end of the int would be the highest address, like how consecutive integers are stored in an array. However, this is not always the case (almost always not the case since most computers are little-endian).

Definition 4.2 (Endian Architecture)

Depending on the machine architecture, computers may store these types slightly differently in their *byte* order. Say that we have an integer of value `0xA1B2C3D4` (4 bytes). Then,

1. A **big-endian architecture** (e.g. SPARC, z/Architecture) will store it so that the least significant byte has the highest address.
2. A **little-endian architecture** (e.g. x86, x86-64, RISC-V) will store it so that the least significant byte has the lowest address.
3. A **bi-endian architecture** (e.g. ARM, PowerPC) can specify the endianness as big or little.

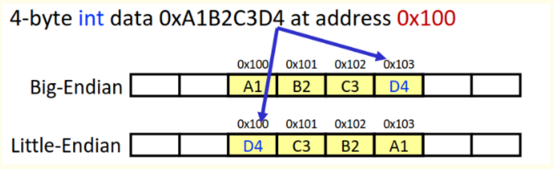


Figure 52: The big vs little endian architectures.

We can simply print out the hex values of primitive types to see how they are stored in memory, but it does not provide the level of details that we want on which bytes are stored where. At this point, we must use certain **debuggers** to directly look at the memory. For x86 architectures, we can use `gdb` and for ARM architectures, we can use `lldb`. At this point, we need to understand assembly to look through debuggers, so we will provide the example here.