Computer Architecture

Muchang Bahng

Spring 2024

Contents

Now that we have learned the theory behind computer science, we will begin to start building a computer from scratch, and the hardware (and low level software) design of the computer is within the realm of *computer architecture*. Starting from the lowest levels allow you to both understand completely the abstractions and appreciate what they do for you.

We should start with transistors, which allows you to then physically implement basic logic gates. This then gives us a sequence of bits to work with, and to create meaningful representations, we define *encoding schemes* on them. Then, these can then be used to do Boolean arithmetic and logical operations (e.g. conditionals), which unlocks our first component of the CPU: the ALU. We must still figure out how to simulate volatile and non-volatile storage, which allows us to define registers and memory. Finally, we wish to define a very tiny language of instructions—called the *instruction set architecture*—that the CPU can understand. This allows the MMU to also interact with the memory.

Definition 0.1 (Instruction Set Architecture)

The **instruction set architecture (ISA)** of a CPU is a description of what it can do. Its scope covers the following.

1. What instructions it can execute, such as bit-length, decoding, and number of operations.
2. The performance vs power efficiency.

Example 0.1 ()

ISAs can be classified into two types.

1. The **complex instruction set computer** (CISC) is characterized by a large set of complex instructions, which can execute a variety of low-level operations. This approach aims to reduce the number of instructions per program, attempting to achieve higher efficiency by performing more operations with fewer instructions. An example is x86, which is the most common architecture for personal computers.
2. The **reduced instruction set computer** (RISC) emphasizes simplicity and efficiency with a smaller number of instructions that are generally simpler and more uniform in size and format. This approach facilitates faster instruction execution and easier pipelining, with the philosophy that simpler instructions can provide greater performance when optimized. Some examples are the ARM and RISC-V architectures.

The actual method in which a given ISA is implemented in a processor is called the *microarchitecture*.

Definition 0.2 (Microarchitecture)

The **microarchitecture**.

1 Transistors

Note that *computation* is an abstract notion (a process) that is distinct from its physical *implementations* (how the progress is run). While most modern computing devices are obtained by mapping logical gates to semiconductor-based transistors, throughout history people have computed using a huge variety of mechanisms, including mechanical systems, gas and liquid (known as fluidics), biological and chemical processes, and even living creatures.

Example 1.1 (Biological Computing)

Computation can be based on biological or chemical systems. For example the lac-operon produces the enzymes needed to digest lactose only if the conditions $x \wedge (\neg y)$ hold, where x is "lactose is present" and y is "glucose is present."

Example 1.2 (Cellular Automata and the Game of Life)

Cellular automata is a model of a system composed of a sequence of cells, each of which can have a finite state. At each step, a cell updates its state based on the states of its neighboring cells and some simple rules. As we will discuss later in this book, cellular automata such as Conway's *Game of Life* can be used to simulate computation gates.

Example 1.3 (Neural Network)

Another computation device is the brain. Even though the exact working of the brain is still not fully understood, one common mathematical model for it is a (very large) **neural network**. A neural network can be thought of as a circuit that—instead of AND/OR/NOT—uses other gates as the basic basis. One particular basis we can use are **threshold gates**, which exist through action potentials n neurons. Approximations of this simulation have been made through artificial netural networks: For every vector $w \in \mathbb{R}^k, t \in \mathbb{Z}$, the threshold function corresponding to w, t is the function

$$T_{w,t} : \{0,1\}^k \longrightarrow \{0,1\}, \qquad T_{w,t}(x) = 1 \text{ iff } \langle w, x \rangle \geq t \tag{1}$$

where $\langle \cdot, \cdot \rangle$ represents the dot product. To a first approximation, a neuron has k inputs and a single output, and the neurons "fires" or "turns on" its output when those signals pass some threshold.

A transistor can be thought of as an electric circuit with two inputs, known as the source and the gate and an output, known as the sink. The gate controls whether current flows from the source to the sink. In a standard transistor, if the gate is "ON" then current can flow from the source to the sink and if it is "OFF" then it can't. In a complementary transistor this is reversed: if the gate is "OFF" then current can flow from the source to the sink and if it is "ON" then it can't.

We can use transistors to implement various Boolean functions such as and AND, OR, and NOT. For each a two-input gate $G : \{0,1\}^2 \longrightarrow \{0,1\}$, such an implementation would be a system with two input wires x, y and one output wire z, such that if we identify high—as in passes a **threshold voltage**—voltage with 1 and low voltage with 0, then the wire z will equal to 1 if and only if applying G to the values of the wires x and y is 1.

1.1 Semiconductors

Okay, basic electronic construction and physics. Some substances are able to easily gain or lose electrons. These allow electricity to flow well, as electrical current is simply electrons moving around. These are "conductors." Other substances are highly resistant to gaining or losing electrons, which means they do not allow electricity to flow well. These are called "insulators."

There is a third kind of substance that falls in between them, that holds on to its electrons harder than

conductors but not as hard as insulators. They are called "semiconductors," of which silicon is the most important one.

Since everything that happens here is on the atomic level, it is very easy to make transistors on the small scale. A mechanical switch with copper contacts would have to be much larger than a transistor. Copper is a conductor, one of the best ones we have, so electrons can jump from one contact to another over a "decent distance." A gap of a couple millimeters is enough to break the circuit, but compared to transistors, that's a massive gulf. Plus, you need something to mechanically move the contacts. Usually an electromagnet is used. Put an electromagnet in a formation that it will cause contacts to open or close when the magnet is energized, and you have a "relay." That's what we used before transistors, and are often used today, though we no longer use them for "thinking" in electronics.

But with semiconductors, they can change from being a conductor to being an insulator very easily. The trick is to add just the right amount of impurities in just the right structure. This is called "doping," and in the world of electronics, it's a good thing. All it takes is a single atom to switch a properly doped piece of silicon from an insulator to conductor and back again. Plus the process is purely electronic. There are no moving parts, so no mechanical components are needed. All you need to do is apply an electrical current to the third leg of a transistor, and the other two legs will go from "open" to "closed." Once the current on the third leg stops, the transistor "opens" again and electricity can't pass through.

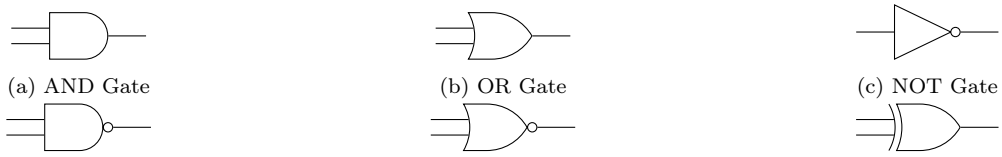
You want semiconductors since.

You first make silicon wafers.

1.2 Doping

1.3 Implementation of NAND

We have seen in our theoretical computer science notes that the NAND gate is universal, and we have implemented it with transistors in the previous chapter. Therefore using syntactic sugar, we can apply the rest of the elementary gates. The common unary and binary logic gates are listed below as a refresher.



(a) AND Gate (b) OR Gate (c) NOT Gate

(d) NAND Gate (e) NOR Gate (f) XOR Gate

Figure 1: Common Logic Gates

1.4 Propagation Delay

We have described everything as if it were instantaneous, but it is not. The *electric current*—the flow of charge carriers like electrons—travels quite slowly at the speed of millimeters per second. This is called the drift velocity of electrons in a conductor. The *electromagnetic field/signal* that propagates through the circuit travels much faster—typically at 50-99% of the speed of light, depending on the material and circuit geometry.

1.5 Clocks

Analog clocks. Digital clocks. You need an RLC circuit to make a clock, falstad. Its supposed to be consistent, so not really logical. Diode.

Definition 1.1 (Clock)

The **master clock** in a computer is an oscillator that alternates continuously between two phases—labeled 0/1, low/high,. tick/tock, etc.

1. The elapsed time between the beginning of 0 and the end of 1 is called a **cycle**, which models one discrete time unit.
2. The current **clock phase** refers to whether it is 0 or 1 now. Using the hardware circuitry, the signal is simultaneously broadcast to every sequential chip throughout the computer platform.

2 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*.

2.1 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.

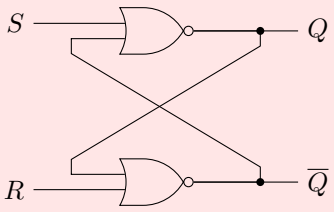
Definition 2.1 (SR Latch)

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 2.1 (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 2: 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 3: 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.

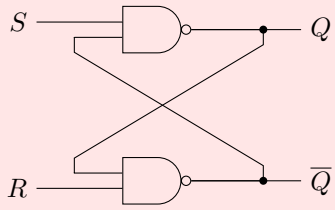
(b) If we send a signal $S = 1$, then $Q = 1$, and even if reset $S = 0$, Q is still locked at 1.

Figure 4

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 2.2 (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 5: 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 6: 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.

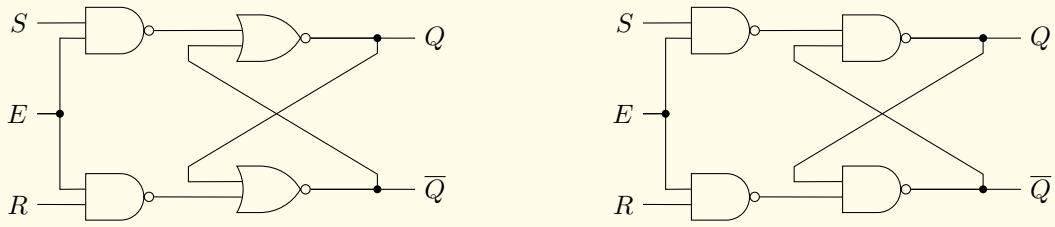
(a) (b)

Figure 7: 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 2.2 (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 8: 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 2.1 (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.

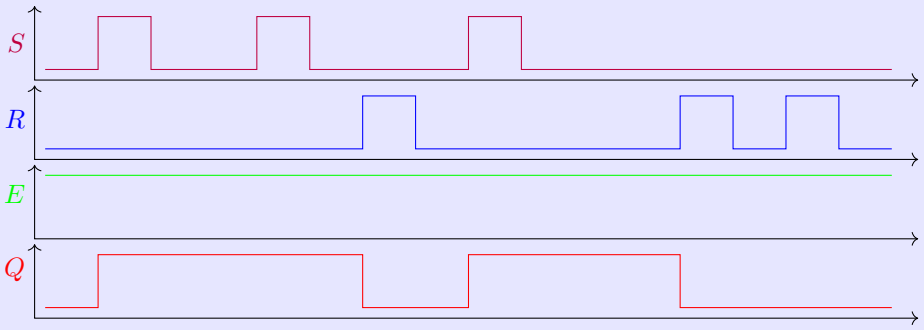


Figure 9: In here the gate is always enabled as $E = 1$ always. In the beginning $S = 1$ causing $Q = 1$, and this does not change until $R = 1$, at which point $Q = 0$. Note that the second pulse of S does not affect the state because it is already $Q = 1$. Soon after $S = 1$ again, causing $Q = 1$ and when $R = 1$ $Q = 0$.

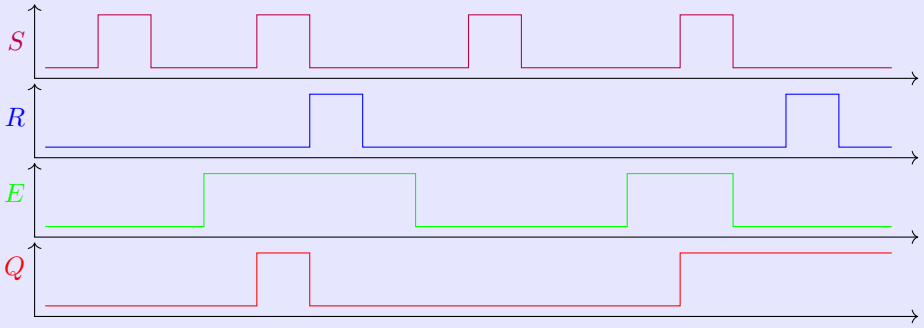


Figure 10: Now we toggle E on and off throughout. We can start off by filling in all the places where $E = 1$, where we want Q to basically copy S. At every other place, we just continue what the state Q was in.

2.2 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.

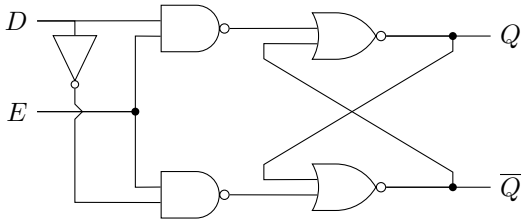
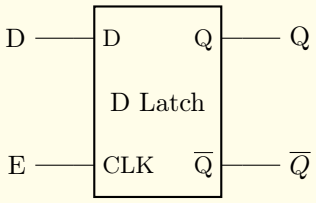


Figure 11

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

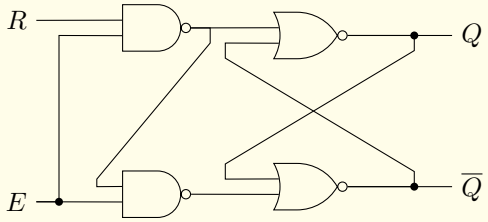
Definition 2.3 (Level Triggered D-Latch)

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

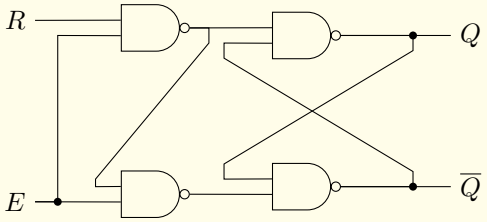


E	D	Q	$\overline{Q}$
0	0	Q_{prev}	$\overline{Q_{\text{prev}}}$
0	1	Q_{prev}	$\overline{Q_{\text{prev}}}$
1	0	0	1
1	1	1	0

Figure 12: 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 13

Example 2.2 (Level Triggered D-Latch)

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

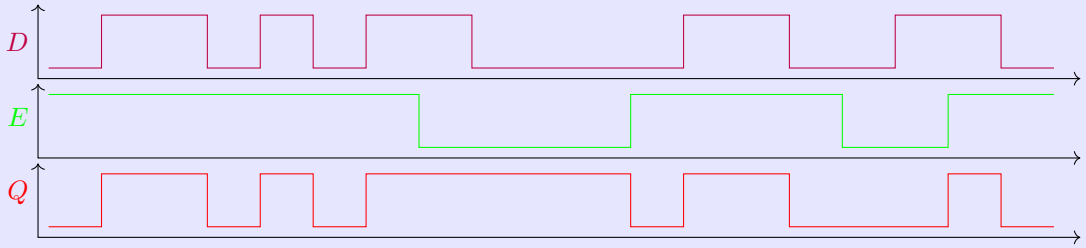


Figure 14: Again, we just let the result Q follow the input D whenever $E = 1$,and continue the rest for when $E = 0$.

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.

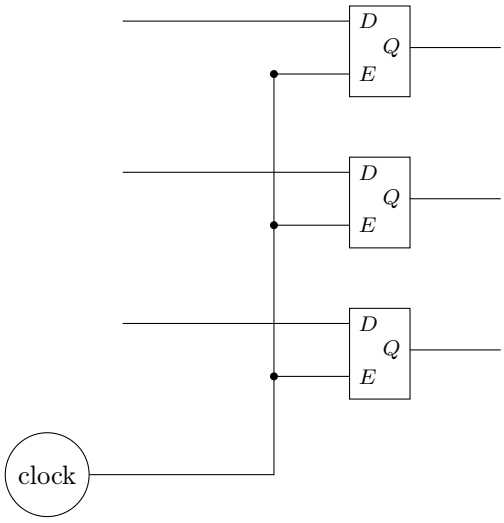


Figure 15: Multiple D-latches enabled and disabled by some external source. The system clock would be a good candidate.

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

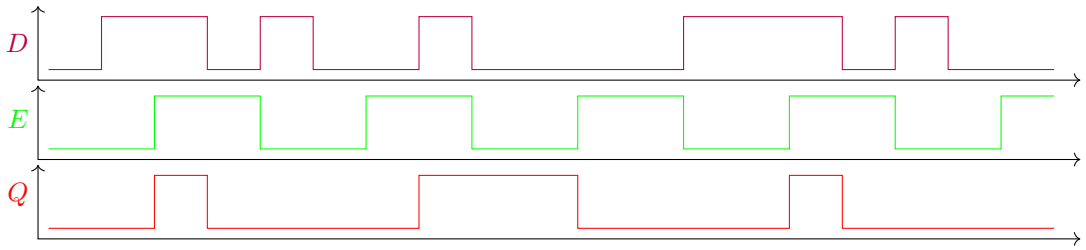


Figure 16: E is connected to a clock that ocsillates at regular intervals.

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 2.4 (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.

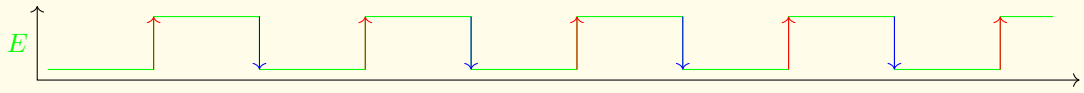
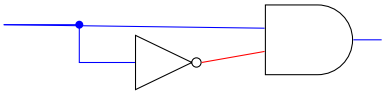
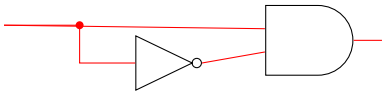


Figure 17: Rising edges are in red, falling edges in blue.

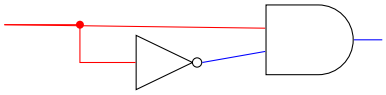
As a summary, for level-triggered (transparent) latches, we have active high and active low. Analogously, for edge-triggered latches, we have rising edge and falling edge. We want to build a D-latch that will respond to changes in D *at the rising edge*, with subsequent changes in D being ignored until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



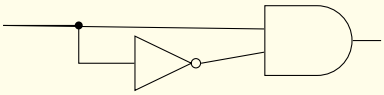
(c) The signal goes through the NOT gate, turning the AND output back to 0.

Figure 18: An edge detection device. Note that if we want to delay the signal even further, we can put an arbitrary amount of NAND gates.

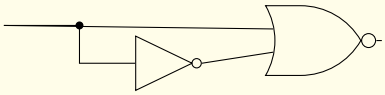
We take this idea to build an edge detection device.

Definition 2.5 (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 19

It has the following waveform.

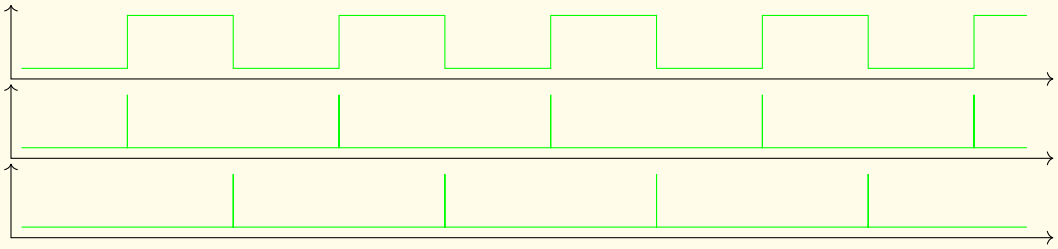


Figure 20: The clock cycle (top). Positive edge detection device (middle). Negative edge detection device (bottom).

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 2.6 (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.

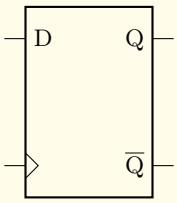


Figure 21: A clocked D latch. Note that the triangle is used to indicate that the clock is inputted.

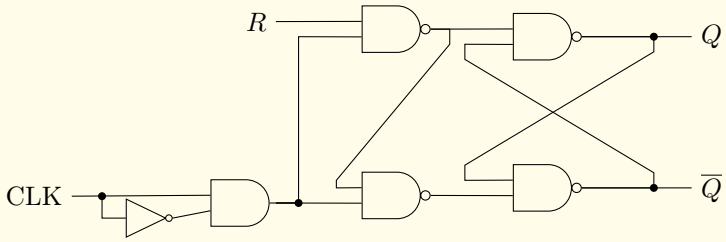


Figure 22

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

Example 2.3 (Edge-Triggered D-Latch Waveforms)

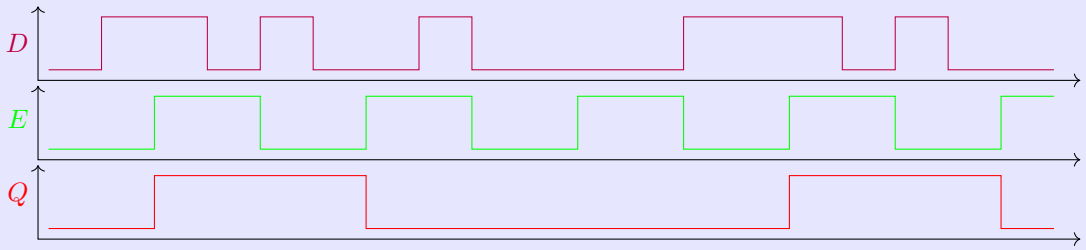


Figure 23: Again, we just let the result Q follow the input D whenever $E = 1$,and continue the rest for when $E = 0$.

Definition 2.7 (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.

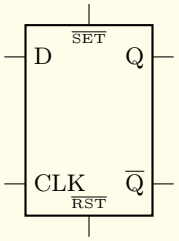


Figure 24: D latch with asynchronous set/reset.

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

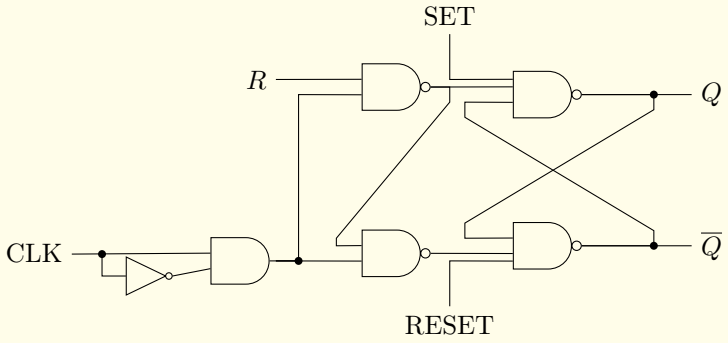


Figure 25

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.

2.3 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.

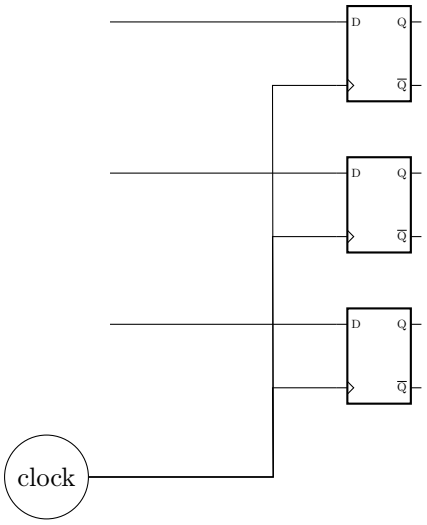


Figure 26

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 2.8 (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

$$\text{out}(t) = \text{in}(t-1) \tag{2}$$

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

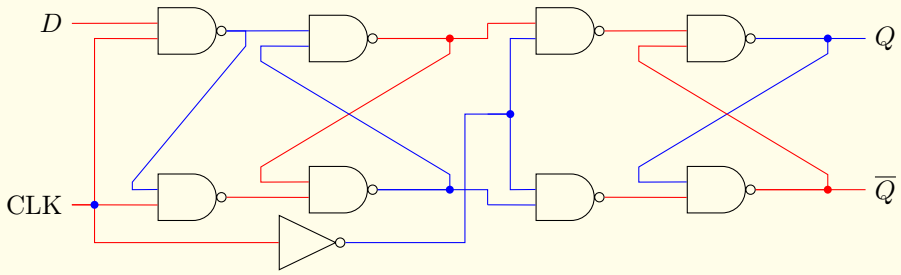


Figure 27: The master reads the input value D when the clock signal is high (or more specifically, the rising edge of the clock cycle) and latches onto it. Meanwhile, the slave is disabled, so the new output from the flip flop is not available just yet.

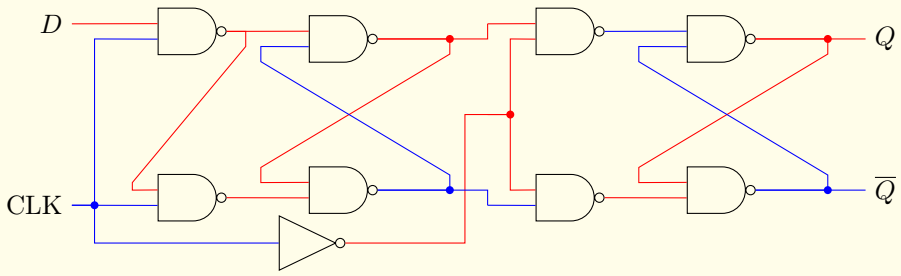


Figure 28: When the clock cycle falls to low, the slave is enabled, making this an edge triggered device. Data is passed from the master to the slave and is therefore available at the output.

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 2.4 (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.

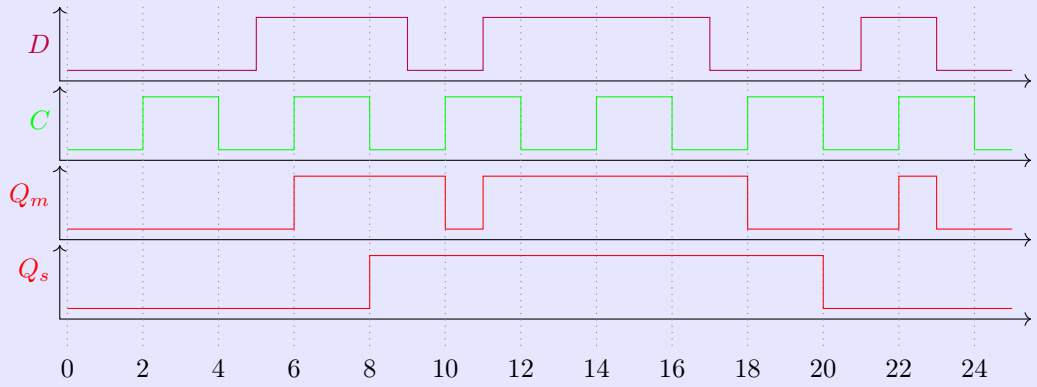


Figure 29

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 $Q = 1$.

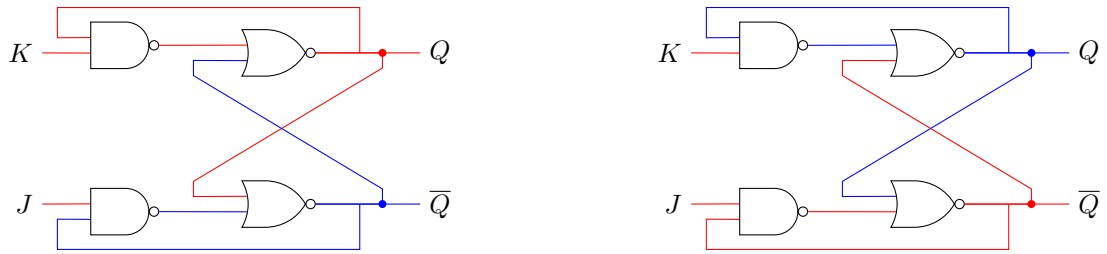


Figure 30: JK active high latch. When you set $J = K = 1$, the latch oscillates between $Q = 0$ and $Q = 1$ very fast, but this eliminates the possibility where Q is both 1 or both 0.

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 2.9 (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**.

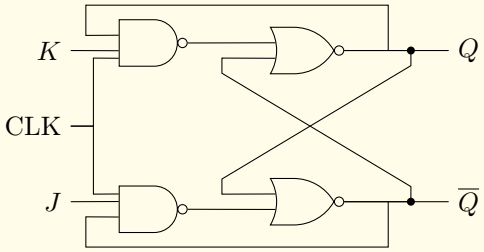


Figure 31: Level Triggered JK Flip Flop.

Example 2.5 (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.

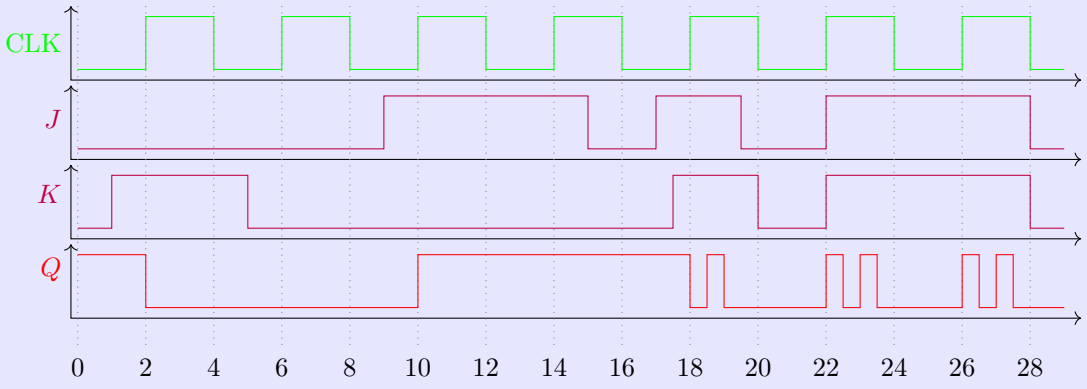


Figure 32

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 2.10 (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**.

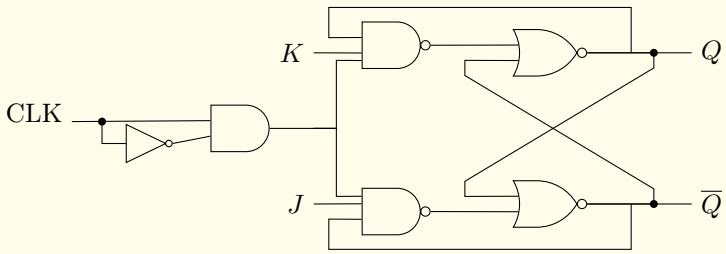


Figure 33: Edge Triggered JK Flip Flop.

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

Example 2.6 (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 $t = 18, 22, 26$).

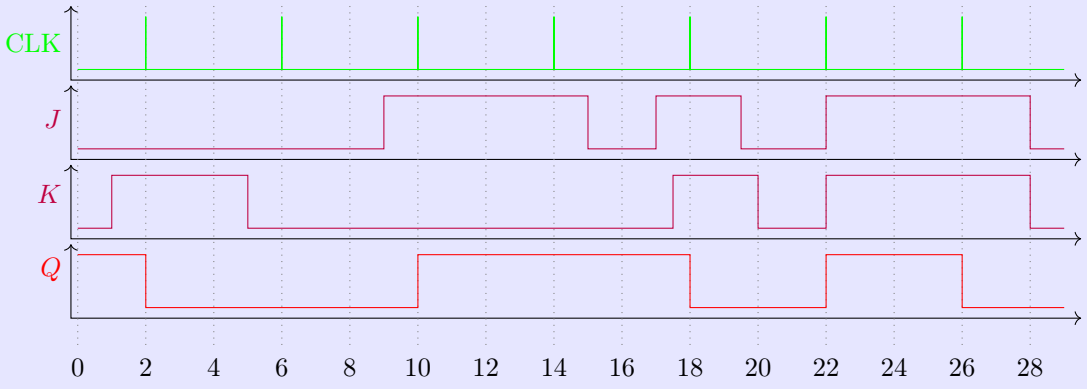


Figure 34

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 2.11 (Toggle Flip Flip)

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

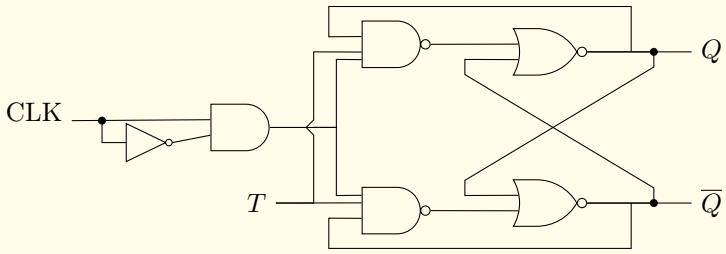


Figure 35: Toggle flip flop.

2.4 Registers

The DFF then unlocks the our first type of memory.

Definition 2.12 (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.*

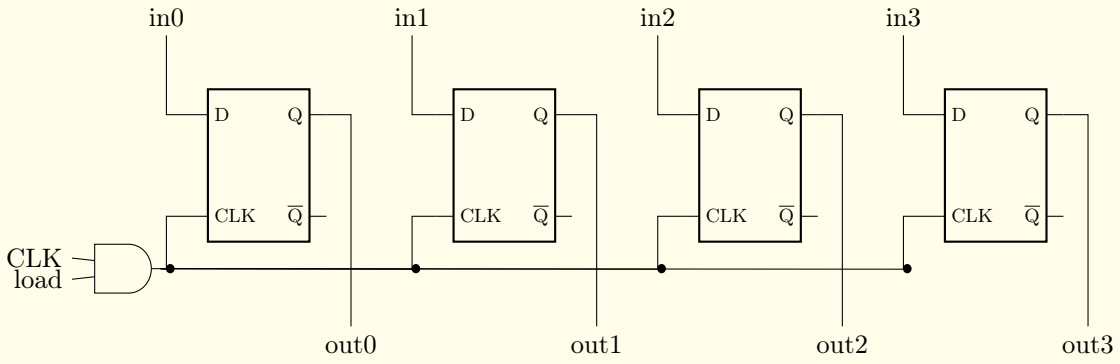


Figure 36: 4-bit register. Since $\overline{Q}$ is redundant we do not consider it as an output in our register.

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

Definition 2.13 (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 of Byte, word, quad, long, etc. Using hexadecimal notation makes them easier to read.

2.5 Applications

Definition 2.14 (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.

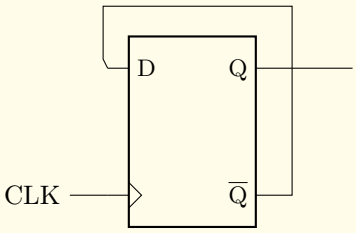


Figure 37: The output $\overline{Q}$ gets rewired into the input D, causing the frequency to slow down.

Example 2.7 (Waveforms of Divide by 2 Latch)

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

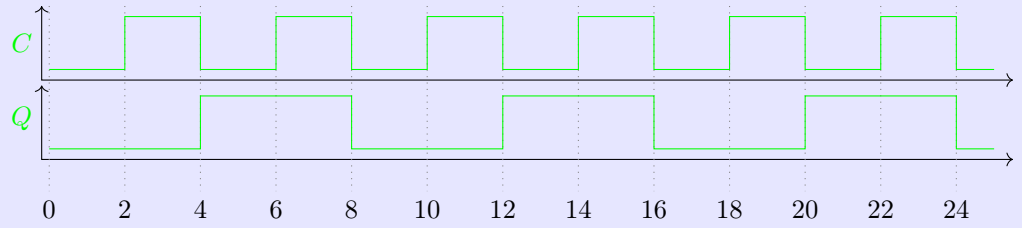


Figure 38

This actually gives us a pretty surprising circuit.

Definition 2.15 (Counter Chip)

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

$$Q(t) = Q(t-1) \tag{3}$$

```
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.

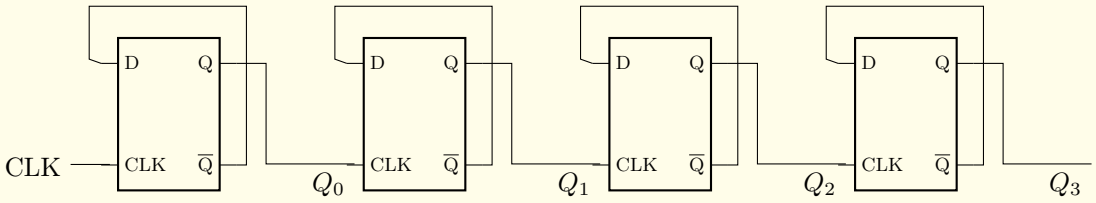


Figure 39: 4-bit register. Since $\overline{Q}$ is redundant we do not consider it as an output in our register. To reset it, we can use the set-reset latches connected to one power source to initialize it to 0.

Example 2.8 (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!

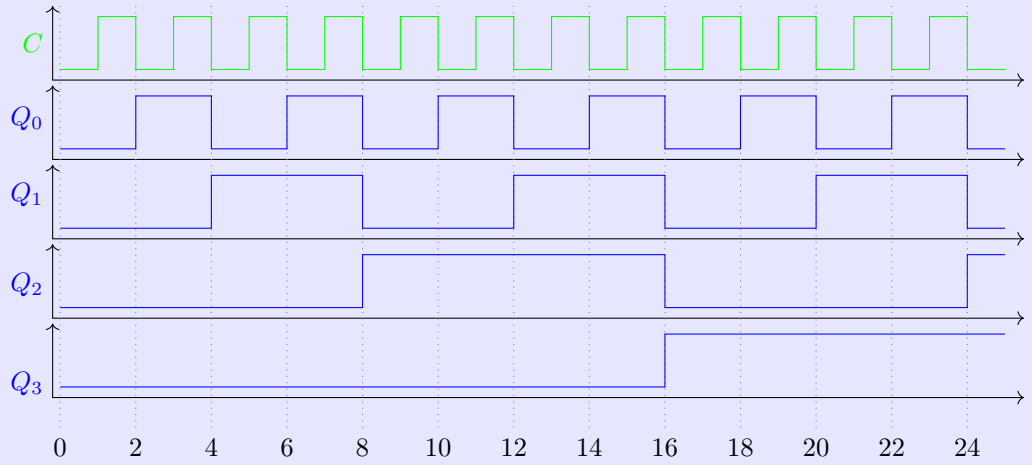


Figure 40: From $t = 0$ to $t = 1$, we have $Q = Q_3Q_2Q_1Q_0 = 0000$. The next time period, we have $Q = 0001$, and so on. This is precisely a counter.

This is extremely useful in almost every situation. First, we can now implement a program counter. Second, when multitasking in an operating system on a single core, we can have a scheduler that allows us to switch from one application to another.

If we connect the latches a bit differently, we can get a shift register, which allows us to do bit-shift operations.

Definition 2.16 (Shift Registers)

A **shift register** takes in a serial input and gives a serial output. It essentially does a bitshift.

$$Q(t+1, \text{in}) = [Q_3(t+1), Q_2(t+1), Q_1(t+1), Q_0(t+1)] = [\text{in}, Q_3(t), Q_2(t), Q_1(t)] \tag{4}$$

$$\text{out} = Q_0(t) \tag{5}$$

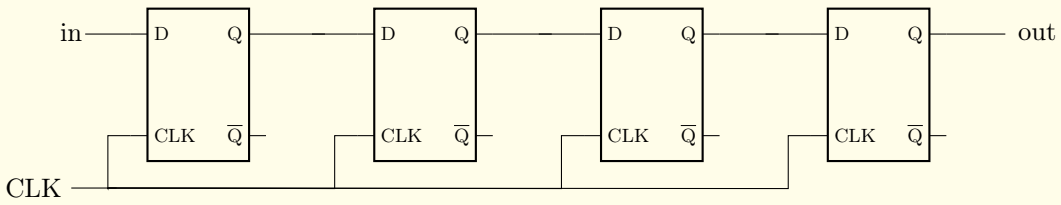


Figure 41: 4-bit register. Since $\overline{Q}$ is redundant we do not consider it as an output in our register.

Example 2.9 (4-Bit Shift Register)

Say that we start off with a buffer of 0000, and we input in a 1 every time interval. Then, with the clock, the flip-flops will make sure to update the shift register consistently. Note that D_{in} does not have to be perfectly initialized to 1 at exactly the rising or falling edge. From our construction we have flexibility.

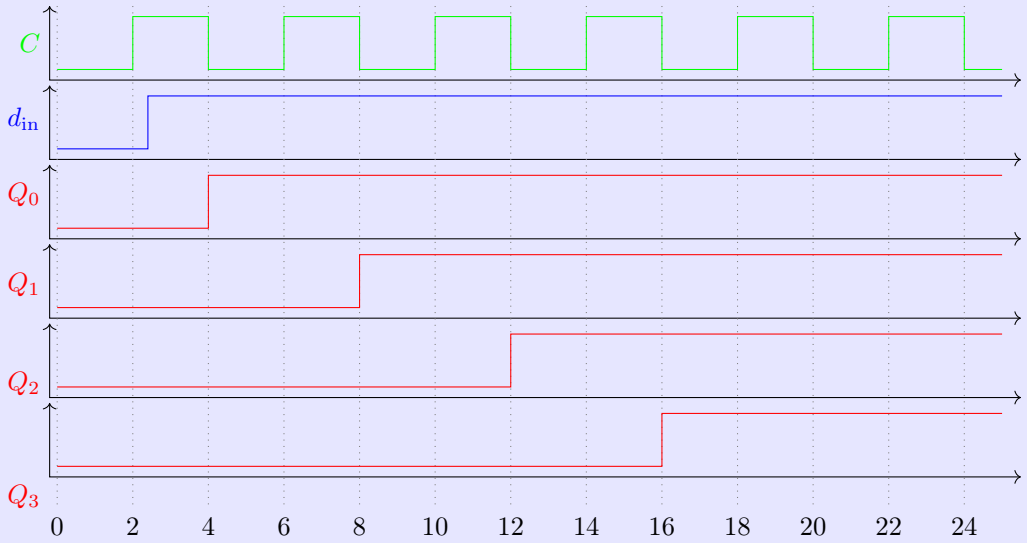


Figure 42: From $t = 0$ to $t = 4$, we have $Q = 0000$. From $t = 4$ to $t = 8$, we have $Q = 1000$. From $t = 8$ to $t = 12$, we have $Q = 1100$. From $t = 12$ to $t = 16$, we have $Q = 1110$. From $t = 16$ to $t = 20$, we have $Q = 1111$.

3 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 3.1 (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

$$E : X \longrightarrow \{0,1\}^* \tag{6}$$

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 3.2 (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.

3.1 Naturals/Unsigned and Integers/Signed

Definition 3.3 (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

$$NtS : \mathbb{N} \longrightarrow \{0,1\}^* \quad (NtS = \text{ "Naturals to Strings"}) \tag{7}$$

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 3.4 (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 3.5 (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 3.1 (Bit Representation of Unsigned Integers in C)

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

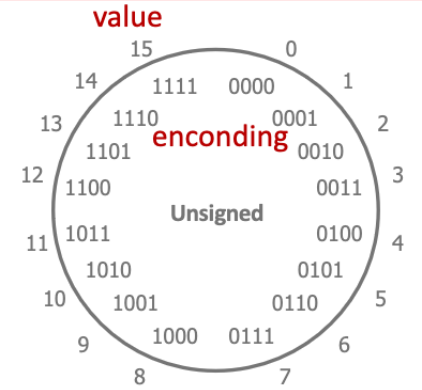


Figure 43: Unsigned encoding of 4-bit integers in C.

Example 3.1 (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.

```c
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 3.6 (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 3.2 (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.

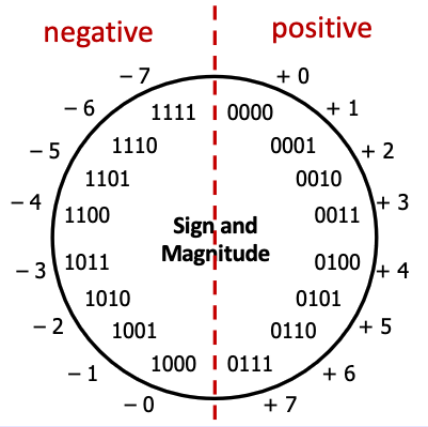


Figure 44: Signed magnitude encoding of 4-bit integers in C.

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 3.2 (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

$$q = -b_n 2^{n-1} + b_{n-1} 2^{n-2} + \cdots + b_1 \tag{8}$$

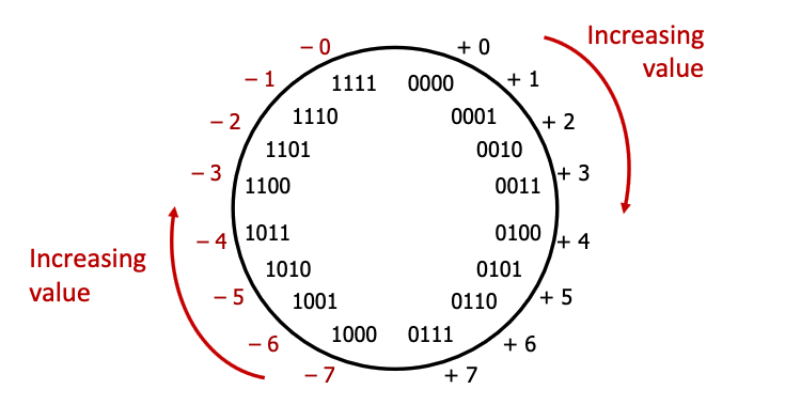


Figure 45: Two's complement encoding of 4-bit integers in C.

Example 3.3 (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 46: Size of various integer types in C with the `sizeof`.

3.1.1 Arithmetic Operations on Binary Numbers

Theorem 3.3 (Inversion of Binary Numbers)

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

Theorem 3.4 (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 $p + (-q)$.

3.2 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 3.7 (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 3.4 ()

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 3.5 (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 3.1 (Representation of Graphs)

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

Theorem 3.6 (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.

3.3 Floats

Theorem 3.7 (Representation of Reals)

There exists no representation of the reals

$$NtR : \mathbb{R} \longrightarrow \{0,1\}^* \tag{9}$$

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 3.8 (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

$$1.b_0b_1b_2...b_k = 1 + \frac{b_1}{2} + \frac{b_2}{4} + ... + \frac{b_k}{2^k}, \quad b_i \in \{0,1\}$$

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 3.9 (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 3.8 (Bit Representation of Floating Point Types in C)

Floats are actually like signed magnitude. We have

$$(-1)^n \times 2^{e-127} \times 1.s \tag{10}$$

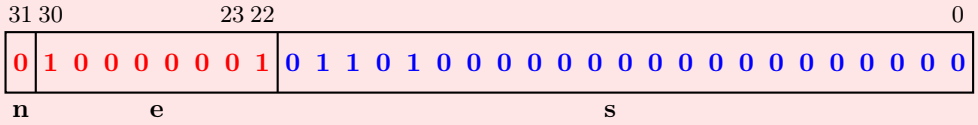


Figure 47: 32-bit representation of floats.

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

3.4 Characters

Definition 3.10 (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 48: 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.

3.4.1 ASCII

Definition 3.11 (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).

3.4.2 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.

3.4.3 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.

$$1110\mathbf{0110}\ 10\mathbf{110001}\ 10\mathbf{001001}$$

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

$$11110\mathbf{000}\ 10\mathbf{011111}\ 10\mathbf{100010}\ 10\mathbf{000110}$$

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.

3.4.4 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.

3.5 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 3.12 ()

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 3.13 (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 3.14 ()

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 3.9 ()

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 3.1 ()

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

$$|\overline{E}(o)| \leq 2|E(o)| + 2 \tag{11}$$

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.)

4 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.

4.1 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 4.1 (Multi-Bit NOT Gate)

Definition 4.2 (Multi-Bit AND Gate)

Definition 4.3 (Multi-Bit OR Gate)

Definition 4.4 (Multi-Bit NAND Gate)

Definition 4.5 (Multi-Bit XOR Gate)

Definition 4.6 (Multi-Bit Multiplexor Gate)

Definition 4.7 (Multi-Bit Demultiplexor Gate)

4.2 Multiplexer

Multiplexors are good for conditionals and implementing hierarchical functions.

Definition 4.8 (Multiplexer)

A $2^n : 1$ **multiplexer** is a gate that takes in
1. n **control bits** as inputs, denoted $s = (s_0, \dots, s_n)$.
2. 2^n **input channels**, denoted x_s for $s \in \{0,1\}^n$.

and outputs the designated input channel according to s.

$$y(s, x) = x_s \tag{12}$$

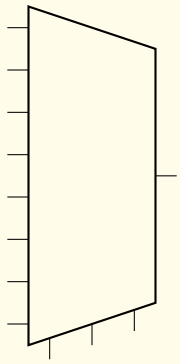


Figure 49: General diagram of a multiplexer with $n = 3$ control bits, $2^n = 8$ input channels, and 1 output channel.

Theorem 4.1 (Implementation of 2:1 Multiplexer)

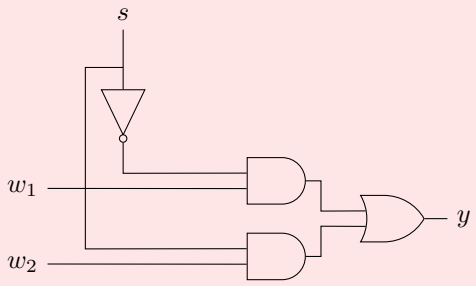


Figure 50: The truth table grows exponentially large with n and does not provide much value, so it is omitted here.

Theorem 4.2 (Implementation of 4:1 Multiplexer)

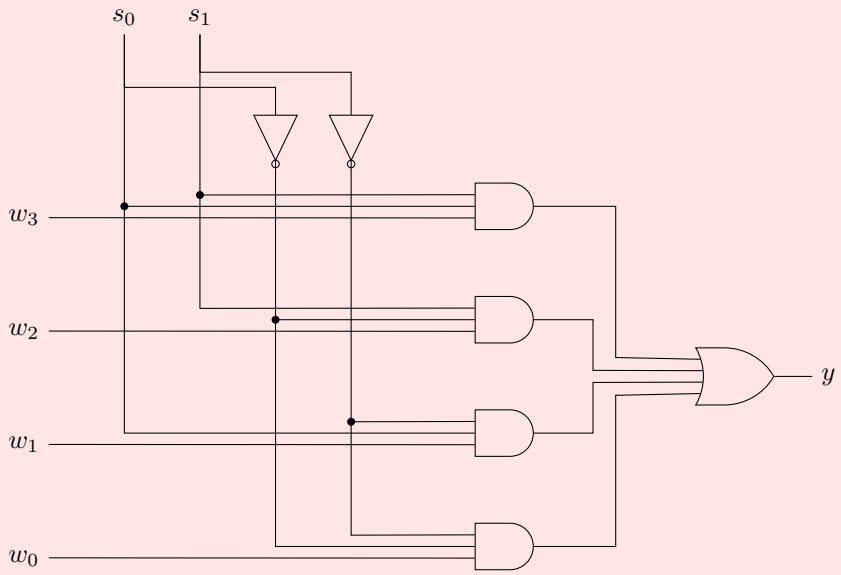


Figure 51: Implementation of 4:1 multiplexer.

There is a general method to build larger multiplexers, and so implementing for n control bits is easy. Consider the simpler diagram for a 4:1 multiplexer.

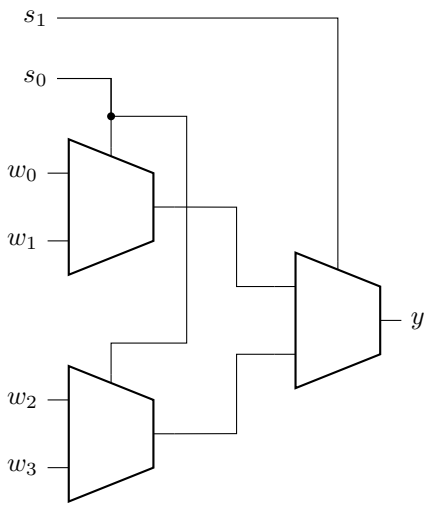
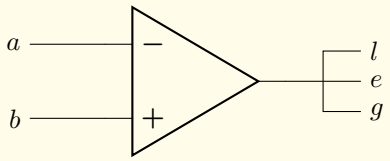


Figure 52: With this pattern, we can build arbitrarily large multiplexors.

4.3 Comparator

Definition 4.9 (1-Bit Comparator)

A **1-bit (magnitude) comparator** simply takes in 2 1-bit numbers $a, b \in \{0, 1\}$ and outputs whether $a > b$, $a = b$, or $a < b$.



(a)

a	b	aEQb	aGTb	bGTa
0	0	1	0	0
0	1	0	0	1
1	0	0	1	0
1	1	1	0	0

(b) Truth table.

The implementation is quite simple.

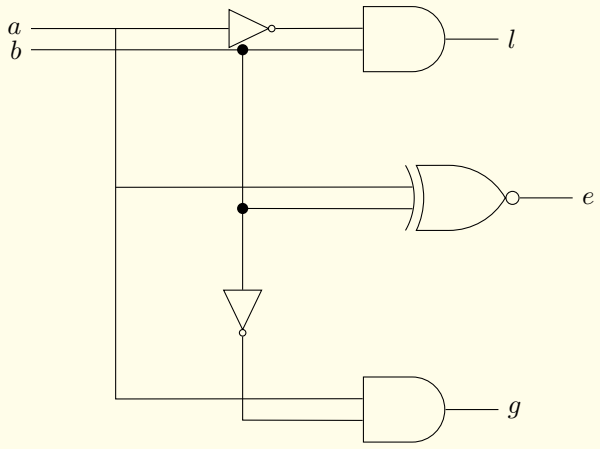


Figure 54

Now that we have a 1-bit comparator, given two n-bit inputs, all we have to do is apply the 1-bit comparator starting from the most significant bit to the least significant bit, and this gives us an n-bit comparator. However, this is pretty slow since it is sequential, but implementing one-shot n-bit comparators requires an exponentially large circuit. Therefore, we compromise between the two schemes. The general idea is that we implement 2-bit and 4-bit one-shot comparators, and then use them to sequentially compare bigger—e.g. 32-bit—numbers.

Definition 4.10 (2-Bit Comparator)

A 2-bit comparator is implemented as such.

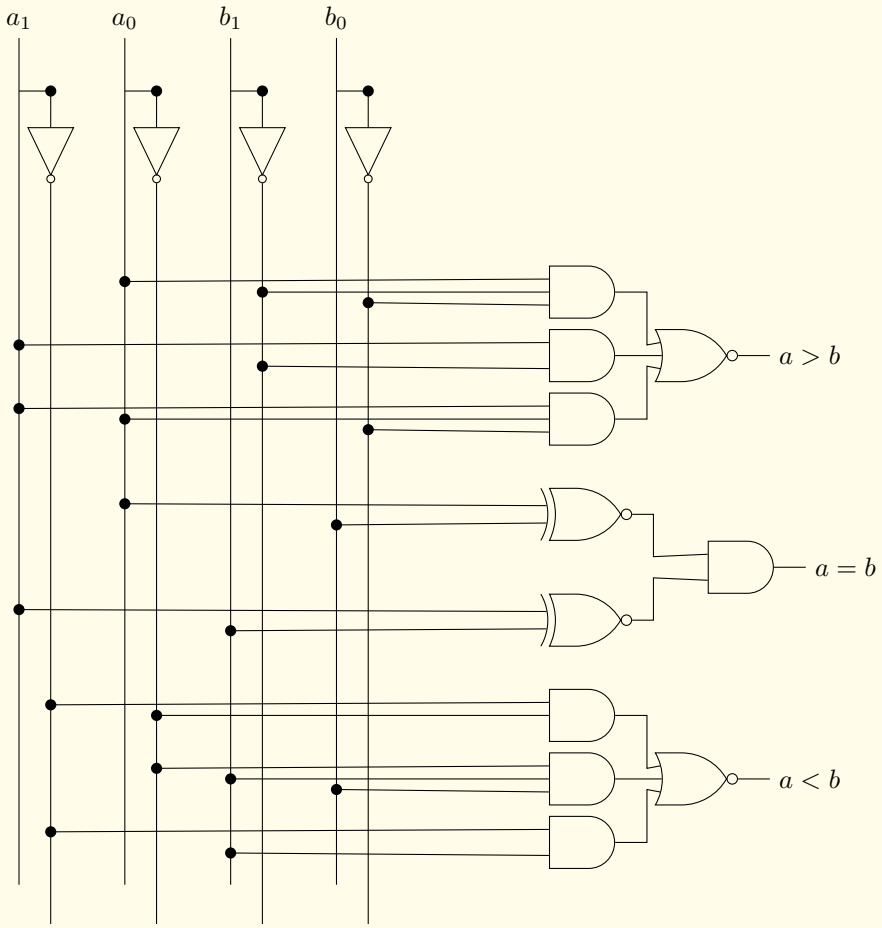


Figure 55: Credits to GeeksForGeeks.

Definition 4.11 (4-Bit Comparator)

A 4-bit comparator is implemented as such.

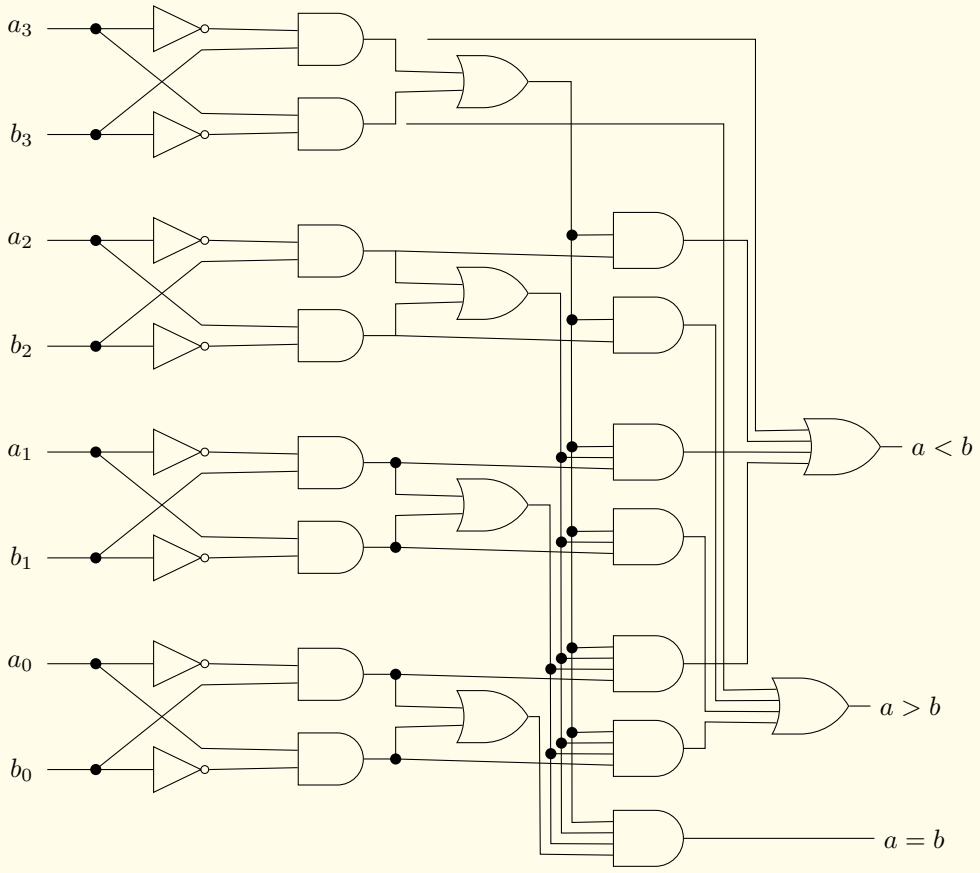


Figure 56: Implementation of 4-bit magnitude comparator.

Therefore,

Definition 4.12 (n-Bit Comparator)

A n-bit comparator is implemented sequentially with 4-bit comparators.

4.4 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 4.13 (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

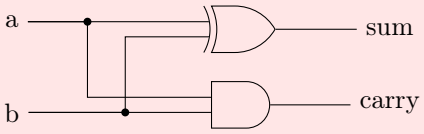
(a) Truth table for half adder.

(b)

Figure 57: Chip diagram for half adder.

Theorem 4.3 (Implementation of Half-Adder)

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



(a)

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

(b) HDL implementation.

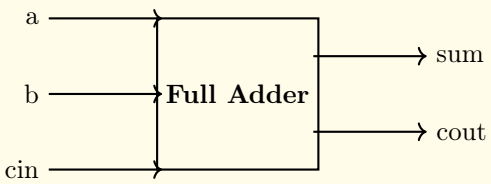
Figure 58

Definition 4.14 (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 59: Chip diagram for full adder.

Theorem 4.4 (Implementation of Full-Adder)

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

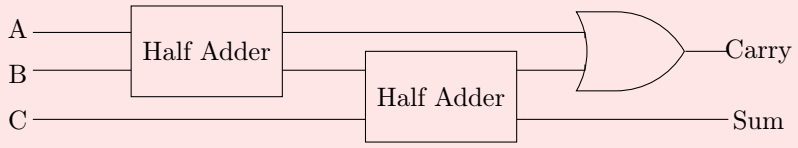


Figure 60

Definition 4.15 (N-Bit Adder)

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

Theorem 4.5 (Implementation of N-Bit Addition)

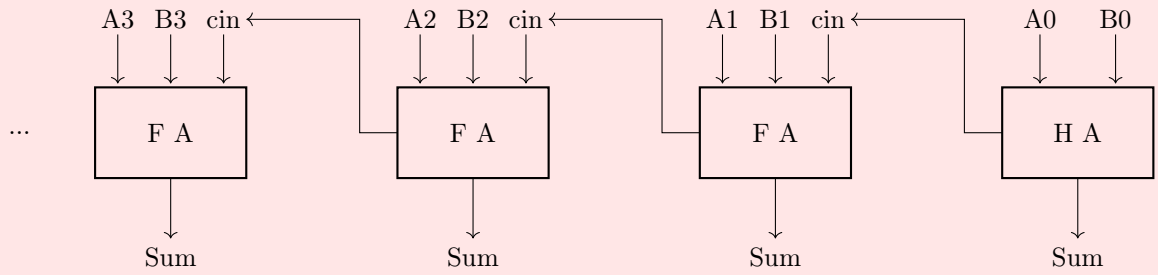


Figure 61: Ripple carry adder for the last 4 significant bits of two N-bit numbers.

Corollary 4.1 (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 4.1 (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.

4.5 Multiplication

Booths algo to do multiplication fast with bitshifts and addition

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

Theorem 4.6 (Implementation of Multiplication in Circuits)

Theorem 4.7 (Implementation of Moving Data in Circuits)

4.6 Arithmetic Logical Unit (ALU)

4.7 Control Unit

The CPU also has a **control unit**, which is responsible for fetching instructions from memory through the **databus**, which is literally a wire connecting the CPU and RAM, and executing them.

In total, we can summarize the role of the CPU as such: the CPU executes instructions from memory one at a time and executes them, known as the **fetch-execute cycle**. It consists of 4 main operations.

1. **Fetch**: The **program counter**, which holds the memory address of the next instruction to be executed, tells the control unit to fetch the instruction from memory through the databus.
2. **Decode**: The fetched data is passed to the **instruction decoder**, which figures out what the instruction is and what it does and stores them in the registers.
3. **Execute**: The arithmetic and logic unit then carries out these operations.
4. **Store**: Then it puts the results back on the databus, and stores them back into memory.

5 Memory Banks

Now that we have established registers, we can use them to build arbitrary sets of memory. For example, by abstracting away the circuits and sticking a bunch of 1-bit registers together, we can get a sequence of them. The location of a register in this sequence is called the **memory address** of the register.

Addresses	Values
...	...
0b0010	1
0b0011	1
0b0100	0
0b0101	1
0b0110	0
0b0111	0
0b1000	0
0b1001	1
0b1010	1
...	...

Figure 62: You can visualize memory as a long array of boxes of bits, similar to PO boxes. 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.

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 architectures, some systems are **word-addressable**, meaning that the memory is addressed by words, which are 4 bytes.[2]

[2]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.

Byte Address	Values
...	...
0x120	10010010 = 0x92
0x121	00000000 = 0x00
0x122	01101111 = 0x6F
0x123	10110000 = 0xB0
0x124	10010110 = 0x96
0x125	10010111 = 0x97
0x126	00010001 = 0x11
0x127	10011001 = 0x99
0x128	11111110 = 0xFE
...	...

Values	Word Address
...	...
0x92006FB0	0x48
0x96971199	0x49
0xFE...	0x4A

Figure 63: 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.

Definition 5.1 (Memory Bank)

A **memory bank** with respect to an ISA is the smallest unit of addressable memory units.

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 5.2 (Endian Architecture)

The **endianness** of an ISA refers to the byte order in which data is stored in memory.

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.

Say that we have an integer of value `0xA1B2C3D4` (4 bytes). Then,

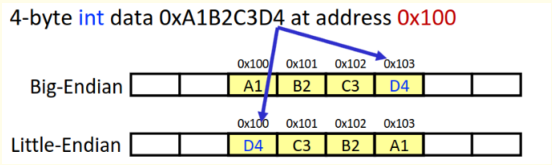


Figure 64: The big vs little endian architectures.

Note that endianness is not a property of memory, but a property of the ISA.

Example 5.1 ()

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.

Let's clarify the differences between registers and memory banks. Memory is an overloaded term that is thrown as an umbrella term for any type of data storage, or as RAM. Memory is addressed by an unsigned integer while registers have names as we will see later (e.g. `%rsi`). RAM is much bigger at several GB, while the total register space is much smaller at around a few KB at most. The memory is much slower than registers, which is usually on a sub-nanosecond timescale. The memory is dynamic and can grow as needed while the registers are static and cannot grow.

5.1 Data Buses

RAM gives us a large pool of memory to work with, albeit slow.

5.2 Fetching and Writing

6 Hardware Description Languages

Look into ternary operators, which is structural. Also for loops. In gtkwave, look at edit/data,color format. Select a bunch of wires and hit g to make a group.

Now that we know about chips, perhaps we are ready to mass produce them. Consider the following scenario where you are a hardware engineer with three boxes full of AND, OR, NOT gates. You need to ship an order of 1000 XOR gates. How would you do this? To construct one XOR gate, you can follow the example below.

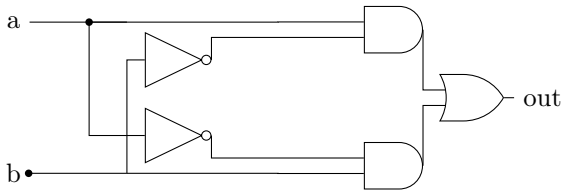


Figure 65: XOR Chip from AON gates.

We would take two AND gates, two NOT gates, and one OR gate, mount them on a board according to the figure's layout, and connect the chips to one another by running copper wires among them and soldering the wire ends to the respective input/output pins. After this, we will have 3 exposed wire ends—two inputs and one output. We then solder a pin to each one of these wire ends, seal the entire device (except for the three pins) in a plastic encasing, and label it as XOR. Do this 1000 times and you're done.

There's a lot of problems with this, with the foremost being that this might be error-prone, especially in more complex chips. There is guarantee that the given chip diagram is correct. Although we can prove correctness in simple cases like XOR, we cannot do so in realistically complex chips. Thus, we need to empirically test the chip, i.e. connect it to a power supply, activate/deactivate the input pins in various configurations, and hope that the chip outputs will agree with its specifications.

Even this debugging process can be quite time-intensive if we endlessly tinker with wires and circuits. Therefore, engineers simulate the construction and testings of these circuits with **hardware simulators**. Remember that we have established that *straight-line programs* are an equivalent model of finite computation, and so we can use lexical programs to model boolean circuits. These programs are called **hardware description languages (HDL)** (analogous to software language) and are used to model and design these digital systems. Once you have written a script in some HDL, you can use a **hardware simulator** (analogous to compiler) to test the circuit. We will use the Verilog language along with the Icarus Verilog hardware simulator.[3]

Definition 6.1 (Module)

A **module** represents some sort of class.

1. **Ports** represent the inputs and outputs of a gate, represented with the `input` and `output` keywords. You might see a convention to put the outputs first and then the inputs.
2. **Wires** are used for connecting different elements, like physical wires between gates. You can think of them as signals, which can be read (is current flowing?) or assigned, but no values get stored in them. They are automatically updated when input changes are specified with the `wire` keyword.
3. **Regs** are like variables that store values—similar to physical registers in CPUs. They are specified with the `reg` keyword.
4. The output value is determined by some logic using the `assign` keyword.

[3]Historically the *VHDL* language was created as a military project—and is still in use—but is a bit ugly. Then, *Verilog* became the most dominant, but it has been largely replaced by *SystemVerilog*. Regardless, both of these are a superset of Verilog, and we will begin with this. Given the Verilog language, *Icarus Verilog* is its corresponding open-source hardware simulator that runs on all platforms (Windows, Mac, Linux).

6.1 Structural and Behavioral Modeling

There are two paradigms of writing Verilog. **Structural modeling** refers to writing code in which we describe the *structure*—i.e. each component—of our circuit. There are two main types.

```
module nand(
  input x1, x2,
  output y
);
  wire z;
  and and1(z, x1, x2);
  not not1(y, z);
endmodule
```

(a) Gate level implementation.

```
module nand(
  input x1, x2,
  output y
);
  assign y = ~(x1 & x2);


endmodule
```

(b) Dataflow level implementation.

Figure 66: Two different implementations of NAND. This example is just to show the difference between the two types of structural modeling. We will assume that NAND is the fundamental operator.

Definition 6.2 (Gate Level Implementation)

Gate level implementation is a functional paradigm similar to a straight line program. Here we use built-in **primitive gates** to work with bits, where the syntax is

```
gate gatename(*output, *input);
```

Here are some sample input signals for demonstration.

```
reg a, b, c;
wire out1, out2, out3, out4, out5, out6, out7, out8;
wire out9, out10, out11, out12, out13, out14, out15;
```

```
and  gate1(out1, a, b);
or   gate2(out2, a, b);
not  gate3(out3, a);
nand gate4(out4, a, b);
nor  gate5(out5, a, b);
xor  gate6(out6, a, b);
```

(a) Basic logic gates.

```
and  gate8(out8, a, b, c);
or   gate9(out9, a, b, c);
nand gate10(out10, a, b, c);
nor  gate11(out11, a, b, c);
// XOR limited to 2 inputs
.
```

(b) Multiple input gates.

Figure 67

Definition 6.3 (Dataflow Modeling)

Dataflow modeling models more of the flow of data, similar to mathematical notation. Here we use built-in **operators** rather than primitive gates. Here are some sample inputs for demonstration.

```
reg [3:0] a = 4'b1010;
reg [3:0] b = 4'b1100;
reg [3:0] c;
```

There are many different types of operators one can use.

```
wire [3:0] and  = a & b;
wire [3:0] or   = a | b;
wire [3:0] not  = ~a;
wire [3:0] xor  = a ^ b;
wire [3:0] nand = ~(a & b);
wire [3:0] nor  = ~(a | b);
wire [3:0] xnor = ~(a ^ b);
```

(a) Logical bitwise operators act on an array of bits and outputs an array.

```
wire and  = &a;
wire or   = |a;
// not behaves identically
wire xor  = ^a;
wire nand = ~&a;
wire nor  = ~|a;
wire xnor = ~^a;
```

(b) Reduction operators reduce an entire array to 1 bit, e.g. AND over n inputs.

```
wire [4:0] add_result = a + b;
wire [3:0] sub_result = a - b;
wire [7:0] mul_result = a * b;
wire [3:0] div_result = a
wire [3:0] mod_result = a % b;

wire logical_and = a && b;
wire logical_or  = a || b;
wire logical_not = !a;
```

(c) Arithmetic and logical operators

```
wire less_than         = a < b;
wire less_equal        = a <= b;
wire greater_than      = a > b;
wire greater_equal     = a >= b;

wire logical_equal     = a == b;
wire logical_not_equal = a != b;
wire case_equal        = a === b;
wire case_not_equal    = a !== b;
```

(d) Relational and equality operators.

Figure 68

We will assume that the nand gate is always implemented.

Definition 6.4 (Structural Implementation of AON)

```
module and(
  input x1, x2
  output y
);
  wire z1;

  nand nand1(z1, x1, x2);
  nand nand2(y, z1, z1);

endmodule
```

(a) Gate Level AND

```
module or(
  input x1, x2,
  output y
);
  wire z1, z2;

  nand nand1(z1, x1, x1);
  nand nand2(z2, x2, x2);
  nand nand3(y, z1, z2);
endmodule
```

(b) Gate Level OR

```
module not(
  input x,
  output y
);
  nand nand1(y, x, x);



endmodule
```

(c) Gate Level NOT

```
module (
  input x1, x2,
  output y
);
  assign y = x1&x2;
endmodule
```

(d) Dataflow Level AND

```
module (
  input x1, x2,
  output y
);
  assign y = x1|x2
endmodule
```

(e) Dataflow Level OR

```
module (
  input x,
  output y
);
  assign y = ~x;
endmodule
```

(f) Dataflow Level NOT

Figure 69: Gate level implementations of elementary gates with NAND in Verilog (top). Notice that these look like straight line programs. Dataflow implementations make things more concise, but less readable.

Definition 6.5 (Structural Implementation of NOR, XOR)

```

```

(a) Gate Level NOR

```

```

(b) Gate Level XOR

```
module nor(
  input x1, x2,
  output y
);
  assign y = ~(x1 | x2);
endmodule
```

(c) Dataflow Level NOR

```
module xor(
  input x1, x2,
  output y
);
  assign y = x1 ^ x2;
endmodule
```

(d) Dataflow Level XOR

Figure 70: Gate level and dataflow level implementations in Verilog.

Definition 6.6 (Structural Implementation of Multiplexor)

```
module mux(a, b, x, out1);
  input a, b, x;
  output out1;

  wire not_x;
  wire out_and1, out_and2;

  not not1(not_x, x);
  and and1(out_and1, not_x, a);
  and and2(out_and2, x, b);
  or or1(out1, out_and1, out_and2);
endmodule
```

(a)

```
module multiplex_gatel(a, b, x, out1);
  input a, b, x;
  output out1;

  assign out1 = (~x & a)|(b & x);
endmodule
```

(b)

Figure 71

Can be efficient but a bit cryptic. So we really want to describe the behavior of the circuit rather than what the circuit actually is. So we do not have to worry about the implementation details, and we trust that the compiler will take care of it.

Example 6.1 (Behavioral Level Implementation of Multiplexor)

In here, we don't care what the circuit looks like, and we model the behavior.

```
module multiplex_gate_level(A, B, X, out1);
  input A, B, X;
  output out1;

  always @(*)
  begin
    if(X==0)
```

```
      out1 = A;
    else
      out1 = B;
  end
endmodule
```

6.2 Test Benching

We've seen how to construct certain gates/chips in Verilog, but we don't know if the circuits actually do what we want. For this, we need to set up *test bench modules*. With these, we can select a predetermined set of inputs and test the signals through each intermediate wire and the output for each.

Definition 6.7 (Test Bench Module)

A **testbench module** represents a suite of inputs that you want to test. The **device under test (dut)** connects the testbench signals to the DUT ports using named port connections.

```
module nand_gate_tb;
  reg a, b; // registers that hold states
  wire y;

  // Instantiate device under test
  nand_gate dut(.a(a), .b(b), .y(y));

  initial begin
    // Enable waveform dumping
    $dumpfile("nand_gate.vcd");
    $dumpvars(0, nand_gate_tb);

    // Test all input combinations
    a = 0; b = 0; #10;
    a = 0; b = 1; #10;
    a = 1; b = 0; #10;
    a = 1; b = 1; #10;

    $display("Test complete");
    $finish;
  end

  // Monitor changes
  initial
    $monitor("At time %t: a=%b, b=%b, y=%b", $time, a, b, y);
endmodule
```

Example 6.2 (Test Benching Multiplexor)

Here we show a testbench module that takes a predetermined set of inputs (all 8) and shows the signals traveling through each wire.

```
module tb_multiplex;
  reg A, B, X;
  wire out1;

  multiplex_gate uut(A, B, X, out1);

  initial begin
    // Test all combinations
    A = 0; B = 0; X = 0; #10;
    A = 0; B = 1; X = 0; #10;
    A = 1; B = 0; X = 0; #10;
    A = 1; B = 1; X = 0; #10;
    A = 0; B = 0; X = 1; #10;
    A = 0; B = 1; X = 1; #10;
    A = 1; B = 0; X = 1; #10;
    A = 1; B = 1; X = 1; #10;
    $finish;
  end

  initial begin
    $dumpfile("waves.vcd");
    $dumpvars(0, tb_multiplex);
  end
endmodule
```

Figure 72: Test bench module for multiplexor in GTKwave.

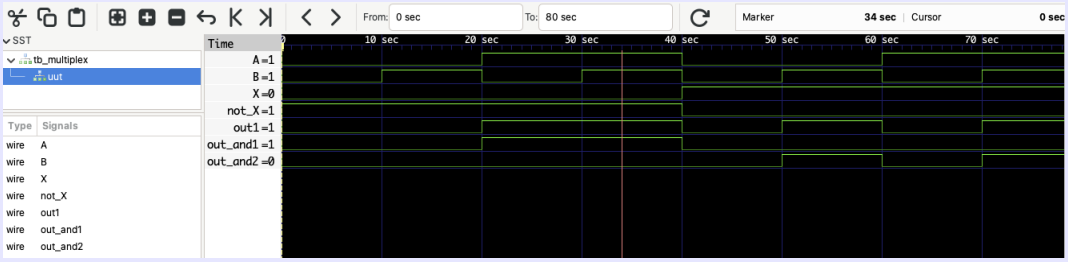


Figure 73: View in GTKwave. You want to take a signal name in the bottom left and add it to the viewer by either double clicking on it or clicking "append."

7 Instruction Sets

Let's review what we have so far. From only gates, we have constructed the three main components.

1. A CPU that can do various arithmetic operations with the ALU and multiplexors.
2. A larger memory bank, called RAM, where the CPU can load and store to.
3. Since a CPU cannot directly do operations in RAM, a set of registers is built into the CPU and is the only place where the CPU can perform operations.[4]

While this is physically implemented in hardware, it is probably not the most efficient to take pieces of wires and directly send electrical signals by physically tapping the pins on the processor. What we first need is some *interface* to efficiently communicate with our machine—along with it some sort of standard that both the machine and the human can understand. This naturally introduces an *instruction set*, which is a more human-friendly abstraction above the hardware level.

Note that the instruction set is the border between the hardware and software level. Another name for instruction sets is *assembly*, which has many different types of languages depending on the type of CPU. The specific implementations of assembly languages—notably x86, ARM, and RISC-V—will be detailed in my separate assembly notes. This section is to describe the general operations that are considered essential, and we will talk about specific implementations in occasional examples. Note that instructions are simply another layer of abstraction that maps binary sequences to human readable words.

Definition 7.1 (Instruction Set Architecture)

The **instruction set architecture (ISA)** of a CPU is basically a description of what it can do. It specifies the following.

1. A predetermined set of **instructions**—also known as **opcodes**—that describes the operations that the CPU supports.
2. The instruction length and decoding, along with its complexity.
3. The performance vs power efficiency.

```
OPCODE1 arg1 arg2 arg3
OPCODE2 arg1 arg2
...
OPCODEn arg1
```

(a) The syntax may differ across ISAs and opcodes, but it is usually the opcode followed by its arguments/operands. The specific opcodes available are usually documented in the ISA's provider page online.

```
0100 0010 0110 1111
0001 000001 010100
...
1101 1100 00000000
```

(b) Every instruction (opcode and arguments) has a direct binary representation which can entirely fit into a register. In here, we consider a register of 16 bits, and unused bits are zero-padded.

Note that it only makes sense to talk about the instruction set architecture of some processing unit (e.g. CPU, GPU), and nothing more/less. It does not make sense to talk about the ISA of a computer, and it is usually implied that we are talking about the CPU residing *in* the computer. The ISA is also a subset of the more general *computer architecture*.

Example 7.1 (CISC vs RISC)

ISAs can be classified into two types.

1. The **complex instruction set computer** (CISC) is characterized by a large set of complex instructions, which can execute a variety of low-level operations. This approach aims to reduce the number of instructions per program, attempting to achieve higher efficiency by performing more operations with fewer instructions.

[4] which is why it must move data from memory to registers before it can perform operations on it.

2. The **reduced instruction set computer** (RISC) emphasizes simplicity and efficiency with a smaller number of instructions that are generally simpler and more uniform in size and format. This approach facilitates faster instruction execution and easier pipelining, with the philosophy that simpler instructions can provide greater performance when optimized.

We should first start off with describing the high level categories of an opcode and an operand. Like higher level programming languages, we can perform operations, do comparisons, and jump to different parts of the code.

Definition 7.2 (Types of Instructions)

There are generally three types of instructions.

1. **Data Movement**: These instructions move data between memory and registers or between the registery and registery. Memory to memory transfer cannot be done with a single instruction.

```
%reg = Mem[address]     # load data from memory into register
Mem[address] = %reg     # store register data into memory
```

2. **Arithmetic Operation**: Perform arithmetic operation on register or memory data.

```
%reg = %reg + Mem[address]     # add memory data to register
%reg = %reg - Mem[address]     # subtract memory data from register
%reg = %reg * Mem[address]     # multiply memory data to register
%reg = %reg / Mem[address]     # divide memory data from register
```

3. **Control Flow**: What instruction to execute next both unconditional and conditional (if statements) ones. With if statements, loops can then be defined.

```
jmp label     # jump to label
je label      # jump to label if equal
jne label     # jump to label if not equal
jg label      # jump to label if greater
jl label      # jump to label if less
call label    # call a function
ret           # return from a function
```

Definition 7.3 (Types of Operands)

These are equivalently determined by their **mode of access**:

1. **Immediate addressing** is denoted with a `$` sign, e.g. a constant integer data `$1`.
2. **Register addressing** is denoted with a `%` sign with the following register name, e.g. `%rax`.
3. **Memory addressing** is denoted with the hexadecimal address in memory, e.g. `0x034AB`.

7.1 Data Movement Operations

When we parse an instruction, its operands are one of three things: literals, registers, and memory forms. The method in which we access each type of data is known as a **mode of access**. These three types are pretty universal, but the syntax in the modes of access vary quite a bit.

Definition 7.4 (Literals)

A **literal** is a constant value, which we will label as `L` or some other capital letter when specified.

Definition 7.5 (Immediate Addressing)

A literal is accessed through **immediate addressing modes**, when the operand of an instruction is a literal `L`.[a]

[a]This seems a bit repetitive, saying that `L` is accessed by `L`. In our syntax it is, but in other languages, like x86, literals are prepended with `$`, e.g. `$1` or `$0x13`.

Every ISA has a predetermined set of registers.

Definition 7.6 (Registers)

The set of registers of a n-bit ISA correspond to the physical registers on the CPU of bit length n. There are two types.

1. *General-purpose registers*—labeled as `R1`, `R2`, ..., `Ri`—can be used to store any value that the coder wants.
2. *Special-purpose registers* are reserved for storing special values and should not in general be used. We will introduce special purpose registers and their notations along the way.

One example of a special purpose register is the stack pointer, which always keep track of the next memory location for the CPU to retrieve. We will cover this later, but note that if we were to override the stack pointer (which we have the power to do), then the program will most likely break. Another example is the *frame pointer*, which points to the base of the current stack frame, and *instruction pointers*, which point to the next instruction to be executed.

Furthermore, there are general conventions even for general-purpose registers. This is to set some standard that programmers can work with, given that assembly is hard enough to learn on its own. For example, certain general registers are generally known to store the parameters of a functions. *Return registers* store return values of functions. In order to see which registers are on or off limits, you must refer to the documentation of each language.

Example 7.2 (Types of Registers)

Here is a snippet from the Apple Developer Docs for ARM64 assembly: The ARM standard delegates certain decisions to platform designers. Apple platforms adhere to the following choices:

- The platforms reserve register x18. Don't use this register.
- The frame pointer register (x29) must always address a valid frame record. Some functions—such as leaf functions or tail calls—may opt not to create an entry in this list. As a result, stack traces are always meaningful, even without debug information.

Definition 7.7 (Normal/Register Addressing)

The value residing in a register `Ri` is accessed through **normal/register addressing modes**, in the following syntax.

```
Reg[Ri]
```
(13)

Finally, we have talked about accessing data on memory banks. By definition, a memory bank must be accessed through its memory address, and fortunately, we have designed our architecture so that all memory addresses can be indexed by a number that fits in our n-bit width register. We can refer to this address plus its value at the address as a *memory form*.

Definition 7.8 (Memory Form)

A **memory form** refers to some representation of a memory address.

Therefore the next addressing mode should be very natural.

Definition 7.9 (Direct Addressing)

In **direct accessing modes**, an instruction contains the memory address to access. We will denote it as

$$\texttt{Mem[L]} \tag{14}$$

Note that the literal `L` was not interpreted as a value itself, but rather as a memory address. Therefore, if we store `L` in a register, then we can interpret the contents of a register as sort of like a pointer to some other location in memory. This introduces us the familiar concept in low-level programming languages. There can be multiple levels of indirection here. A register may point to a memory address and a memory bank may itself point to another memory address.

Definition 7.10 (Pointer)

A memory device that stores a memory address is said to be a **pointer**. To access the value at a location in memory, we **dereference** the pointer as follows.

$$\texttt{Mem[Reg[Ri]]}, \quad \texttt{Mem[Mem[L]]} \tag{15}$$

Above, `Ri` is a pointer since it is a register that contains `Reg[Ri]`, an address to another memory. Similarly, `Mem[L]` is a pointer since the value at this location in memory points to another memory address.

Therefore, there is a duality. A literal `L` can be interpreted both as a value or a pointer. This allows us to use syntactic sugar to compute offset memories through a technique called **pointer arithmetic**. The following addressing modes are just fancy names for computing offsetting addresses.

Definition 7.11 (Offset Addressing)

Take a base memory address `B`, an offset `D`, and a scale factor `S`.[a] We can access the offset memory address

$$\texttt{Mem[B + S * D]} \tag{16}$$

in the following ways.[b]

1. If the base address is a literal and `D = Reg[Ri]` is stored in a register, i.e. `Ri` is a pointer, then this is called **indexed addressing**.

$$\texttt{Mem[B + S * Reg[Ri]]} \tag{17}$$

2. If the base address `B = Reg[Rb]` and offset is a literal, then this is called **displacement addressing**.

$$\texttt{Mem[Reg[Rb] + S * D]} \tag{18}$$

3. If the base address `B = Reg[Rb]` and offset is also a pointer `D = Reg[Ri]`, then this is called **indirect addressing**.

$$\texttt{Mem[Reg[Rb] + S * Reg[Ri]]} \tag{19}$$

[a] We need a scale factor since the word size—the bit length of memory banks—may be several Bytes.

[b] The names are very interchangeable depending on context, so rather than just memorizing these, just know that you can always replace literals with addresses, and use them to calculate offsets.

Note that this is just syntax, and the `+` and `*` signs do not mean anything yet.

Now that we have developed syntax for accessing data, the next thing to do is to know how to move data around. Let's define a simple operator for this.

Definition 7.12 (`mov`)

The move operation has the following syntax. It is more of a copy rather than a move.

```
mov (source) (destination)
```
(20)

and supports the following combination of operands.

```
mov L       Ri        // move literal L to register Ri
mov L       Mem[A]    // move literal L to memory address at A
mov Ri      Rj        // copy value at register Ri to register Rj
mov Ri      Mem[A]    // move value at register Ri to memory address at A
mov Mem[A]  Ri        // move value at memory address A to register Ri
mov Mem[A1] Mem[A2]   // move value at memory address A1 to address A2
```

Note the following details on the syntax.

1. We cannot set a literal as the destination, and so there are no operands that support this.
2. The syntax for moving the value at register `Ri` to `Rj` is not `mov Reg[Ri] Rj`.

Note that when a move instruction is called, the opcode `mov` along with its operands will get translated into binary. Consider a 16-bit machine with 6-bit opcodes and 5-bit operands. If the opcode for `mov` is `111000` and say that we look for the two cases.

1. We want to move the value at register `Ri`, which may translate to the binary encoding `11111`, to the value at register `Rj`, which may have the binary encoding `00000`.
2. We want to move the literal `11111` to the value at register `Rj` with binary encoding `00000`.

Both scenarios give us the instruction encoding as

```
111000 11111 00000
```
(21)

The CPU—only able to interpret the instruction as a sequence of bits, will not be able to differentiate the two. The solution is to have *different* binary encodings for each version of the `mov` operation.

Example 7.3 (`mov` **Opcodes in x86)**

The `mov` instruction in x86_64 has the following opcodes. Note that in here, the first operand is the destination

Opcode	Description
0x88	mov r/m8, r8 (byte register to memory/register)
0x89	mov r/m32/64, r32/64 (32/64-bit register to memory/register)
0x8A	mov r8, r/m8 (memory/register to byte register)
0x8B	mov r32/64, r/m32/64 (memory/register to 32/64-bit register)
0xB0-0xB7	mov r8, imm8 (immediate byte to 8-bit register)
0xB8-0xBF	mov r32/64, imm32/64 (immediate to 32/64-bit register)
0xC6	mov r/m8, imm8 (immediate byte to memory/register)
0xC7	mov r/m32/64, imm32 (immediate to memory/register)

Figure 75: Opcodes for x86.

We can see this in the object dump file.

```
> objdump -d exit

exit:     file format elf64-x86-64

Disassembly of section .text:

0000000000401000 <_start>:
  401000:   b8 01 00 00 00          mov    $0x1,%eax
  401005:   bb 12 00 00 00          mov    $0x12,%ebx
  40100a:   cd 80                   int    $0x80
```

Figure 76

One final note is that in actual implementation, the load (`ldr`) operation is actually used to move data from memory to registers, while `mov` is used to move from registers to registers/memory. Memory to memory transfer cannot be done with a single execution.

7.2 Arithmetic and Logical Operations

Now we start abstracting away the implementations of combinational circuits into a language.

Definition 7.13 (Flag Register)

The `flg` register keep tracks of all this stuff. The most common flags are as follows.

Flag	Name	Description
Z	Zero flag	Indicates that the result of an arithmetic or logical operation (or, sometimes, a load) was zero.
C	Carry flag	Enables numbers larger than a single word to be added/subtracted by carrying a binary digit from a less significant word to the least significant bit of a more significant word as needed. It is also used to extend bit shifts and rotates in a similar manner on many processors (sometimes done via a dedicated X flag).
S / N	Sign flag Negative flag	Indicates that the result of a mathematical operation is negative. In some processors, the N and S flags are distinct with different meanings and usage: One indicates whether the last result was negative whereas the other indicates whether a subtraction or addition has taken place.
V / O / W	Overflow flag	Indicates that the signed result of an operation is too large to fit in the register width using two's complement representation.

Figure 77: Most common CPU status register flags, implemented in almost all modern processors.

Some less common flags are shown below.

Flag	Name	Description
H / A / DC	Half-carry flag Auxiliary flag Digit carry Decimal adjust flag	Indicates that a bit carry was produced between the nibbles (typically between the 4-bit halves of a byte operand) as a result of the last arithmetic operation. Such a flag is generally useful for implementing BCD arithmetic operations on binary hardware.
P	Parity flag	Indicates whether the number of set bits of the last result is odd or even.
I	Interrupt flag	On some processors, this bit indicates whether interrupts are enabled or masked. If the processor has multiple interrupt priority levels, such as the PDP-11, several bits may be used to indicate the priority of the current thread, allowing it to be interrupted only by hardware set to a higher priority. On other architectures, a bit may indicate that an interrupt is currently active, and that the current thread is part of an interrupt handler.
S	Supervisor flag	On processors that provide two or more protection rings, one or more bits in the status register indicate the ring of the current thread (how trusted it is, or whether it must use the operating system for requests that could hinder other threads). On a processor with only two rings, a single bit may distinguish Supervisor from User mode.

Figure 78

Example 7.4 (Flag Registers in x86 and ARM)

Here are some common flag register implementations.

1. In x86, the `FLAGS` register contains.
2. In ARM, the `NZCV` register stores the sign, zero, carry, and overflow flags.
3. In RISC-V, comparison instructions store the comparison result in a general-purpose register, and conditional branches act based on the value in the general purpose register.
4. In MIPS, we do not use a dedicated flag register.

With the flag registers, we can have a complete implementation of the following.

Definition 7.14 (Addition)

The addition operation performs arithmetic addition and stores the result in the destination.

```
add (source) (destination)
```
(22)

and supports the following combination of operands.

```
add L       Ri        // add literal L to register Ri
add L       Mem[A]    // add literal L to value at memory address A
add Ri      Rj        // add value at register Ri to register Rj
add Ri      Mem[A]    // add value at register Ri to value at memory address A
add Mem[A]  Ri        // add value at memory address A to register Ri
add Mem[A1] Mem[A2]   // add value at memory address A1 to value at address A2
```

Note the following details on the syntax.

1. The destination operand is modified to contain the sum of source and destination.
2. We cannot set a literal as the destination, and so there are no operands that support this.
3. The operation affects processor flags including zero, carry, overflow, and sign flags.

Definition 7.15 (Addition with Carry)

Definition 7.16 (Subtraction)

The subtraction operation performs arithmetic subtraction and stores the result in the destination.

```
sub (source) (destination)
```
(23)

and supports the following combination of operands.

```
sub L       Ri        // subtract literal L from register Ri
sub L       Mem[A]    // subtract literal L from value at memory address A
sub Ri      Rj        // subtract value at register Ri from register Rj
sub Ri      Mem[A]    // subtract value at register Ri from value at memory address A
sub Mem[A]  Ri        // subtract value at memory address A from register Ri
sub Mem[A1] Mem[A2]   // subtract value at memory address A1 from value at address A2
```

Note the following details on the syntax.

1. The destination operand is modified to contain the difference (destination - source).
2. We cannot set a literal as the destination, and so there are no operands that support this.
3. The operation affects processor flags including zero, carry, overflow, and sign flags.

Definition 7.17 (Multiplication)

The multiplication operation performs arithmetic multiplication and stores the result in the destination.

```
mul (source) (destination)
```
(24)

and supports the following combination of operands.

```
mul L       Ri        // multiply register Ri by literal L
mul L       Mem[A]    // multiply value at memory address A by literal L
mul Ri      Rj        // multiply register Rj by value at register Ri
mul Ri      Mem[A]    // multiply value at memory address A by value at register Ri
mul Mem[A]  Ri        // multiply register Ri by value at memory address A
mul Mem[A1] Mem[A2]   // multiply value at address A2 by value at address A1
```

Note the following details on the syntax.

1. The destination operand is modified to contain the product of source and destination.
2. We cannot set a literal as the destination, and so there are no operands that support this.
3. The operation may produce results larger than register width, affecting overflow flags.

Definition 7.18 (Division)

The division operation performs arithmetic division and stores the result in the destination.

```
div (source) (destination)
```
(25)

and supports the following combination of operands.

```
div L       Ri        // divide register Ri by literal L
div L       Mem[A]    // divide value at memory address A by literal L
div Ri      Rj        // divide register Rj by value at register Ri
div Ri      Mem[A]    // divide value at memory address A by value at register Ri
div Mem[A]  Ri        // divide register Ri by value at memory address A
div Mem[A1] Mem[A2]   // divide value at address A2 by value at address A1
```

Note the following details on the syntax.
1. The destination operand is modified to contain the quotient (destination $\div$ source).
2. We cannot set a literal as the destination, and so there are no operands that support this.
3. Division by zero typically triggers an exception or sets error flags.

Definition 7.19 (Increment, Decrement)

The increment and decrement operations add or subtract 1 from the operand.

```
inc (operand)                    dec (operand)
```

and support the following operands.

```
inc Ri        // increment register Ri by 1
inc Mem[A]    // increment value at memory address A by 1
dec Ri        // decrement register Ri by 1
dec Mem[A]    // decrement value at memory address A by 1
```

Note the following details on the syntax.
1. These are unary operations that modify the operand in place.
2. Literals cannot be used as operands since they cannot be modified.
3. The operations affect flags but typically preserve the carry flag.

Definition 7.20 (Negation)

The negation operation performs two's complement negation of the operand.

```
neg (operand)
```
(26)

and supports the following operands.

```
neg Ri        // negate value in register Ri
neg Mem[A]    // negate value at memory address A
```

Note the following details on the syntax.
1. This is a unary operation that modifies the operand in place.
2. Literals cannot be used as operands since they cannot be modified.
3. The operation sets flags based on the result, including overflow for minimum values.

Definition 7.21 (Bitwise Operations)

The bitwise operations perform logical operations on operands bit by bit.

```
and (source) (destination)
or (source) (destination)
xor (source) (destination)
```

and support the following combination of operands.

```
and L       Ri        // bitwise AND register Ri with literal L
or  L       Mem[A]    // bitwise OR value at memory address A with literal L
xor Ri      Rj        // bitwise XOR register Rj with value at register Ri
and Ri      Mem[A]    // bitwise AND value at memory address A with register Ri
or  Mem[A]  Ri        // bitwise OR register Ri with value at memory address A
xor Mem[A1] Mem[A2]   // bitwise XOR value at address A2 with value at address A1
```

Note the following details on the syntax.

1. The destination operand is modified to contain the result of the logical operation.
2. We cannot set a literal as the destination, and so there are no operands that support this.
3. These operations clear the carry and overflow flags, setting zero and sign flags.

Definition 7.22 (Bitshift)

The bitshift operations shift bits left or right by a specified number of positions.

$$\texttt{shl (count) (operand)} \quad \texttt{shr (count) (operand)} \tag{27}$$

and support the following combination of operands.

```
shl L       Ri        // shift register Ri left by L positions
shr L       Mem[A]    // shift value at memory address A right by L positions
shl Ri      Rj        // shift register Rj left by value in register Ri
shr Ri      Mem[A]    // shift value at memory address A right by value in register Ri
```

Note the following details on the syntax.

1. The operand is modified to contain the shifted result.
2. Left shift (shl) multiplies by powers of 2; right shift (shr) divides by powers of 2.
3. The carry flag receives the last bit shifted out of the operand.

Definition 7.23 (Unconditional Jumps)

The unconditional jump operation transfers control to a specified address.

$$\texttt{jmp (target)} \tag{28}$$

and supports the following target specifications.

```
jmp L         // jump to literal address L
jmp Ri        // jump to address stored in register Ri
jmp Mem[A]    // jump to address stored at memory address A
jmp Label     // jump to labeled instruction
```

Note the following details on the syntax.

1. This operation unconditionally transfers control, updating the program counter.
2. Labels are resolved to addresses during assembly or linking.
3. No processor flags are affected by the jump operation itself.

Definition 7.24 (Compare)

The compare operation performs subtraction but only sets flags without modifying operands.

```
cmp (source) (destination)
```
(29)

and supports the following combination of operands.

```
cmp L       Ri        // compare register Ri with literal L
cmp L       Mem[A]    // compare value at memory address A with literal L
cmp Ri      Rj        // compare register Rj with value at register Ri
cmp Ri      Mem[A]    // compare value at memory address A with value at register Ri
cmp Mem[A]  Ri        // compare register Ri with value at memory address A
cmp Mem[A1] Mem[A2]   // compare value at address A2 with value at address A1
```

Note the following details on the syntax.

1. Neither operand is modified; only processor flags are set based on (destination - source).
2. We cannot set a literal as the destination, and so there are no operands that support this.
3. Sets zero, carry, overflow, and sign flags for use by conditional jump instructions.

Definition 7.25 (Conditional Jump)

The conditional jump operation transfers control to a specified address based on processor flags.

```
jcc (target)
```
(30)

where `cc` represents condition codes, and supports the following target specifications.

```
je  L         // jump if equal (zero flag set)
jne L         // jump if not equal (zero flag clear)
jl  L         // jump if less than (sign != overflow)
jg  L         // jump if greater than (zero clear and sign == overflow)
jc  L         // jump if carry set
jo  L         // jump if overflow set
js  L         // jump if sign set
```

Note the following details on the syntax.

1. Control transfers only if the specified condition is met; otherwise execution continues.
2. Conditions are based on flags set by previous operations like compare or arithmetic.
3. Common conditions include equality, inequality, and signed/unsigned comparisons.

7.3 Code and Data Segments

Okay, now that we are familiar with the syntax, let's step back to consider how a series of bits are actually converted into code. Given a program, it must work with data, either to store its own code or to access data from memory itself. Whenever there is data, there exists memory, and whenever there is memory, there is the *address* where it is located in.

Definition 7.26 (Address Space)

An **address space** is the range of memory addresses available to a program or a process. Each address space of a program is also called a *segment*.

The program itself—i.e. the list of instructions—is also data, and so it will be stored in some address space. We have a name for this.

Definition 7.27 (Text/Code Segment)

The **text/code segment** is the address space where the program instructions are stored. It is readable, executable, and of fixed size. It is not writable.

Okay, so assume that we have the binary encodings of instructions in the text segment, along with the relevant data in the data segment. The CPU is apparently supposed to execute each line of instruction, but

Definition 7.28 (Instruction Pointer)

The **instruction pointer**, which we will denote `ip`, is a register that stores the address of the next instruction to execute.

Example 7.5 (Execution of Simple Program)

Definition 7.29 (Data Segment)

The **data segment** is the address space where static global variables are stored. It is readable and writable, but not executable.

Definition 7.30 (Read Only Data Segment)

The **read only data segment**, also known as **rodata**, is the address space that stores read-only data, i.e. constants.

These two address spaces are so important that every script written in assembly must specify this. Having no data segment implies that the data segment is empty.

```
.section .text
.globl _start

_start:
  movl $1, %eax
  movl $0, %ebx
  int $0x80
```

(a) x86 assembly

(b) ARMv8 assembly

Figure 79: Every script must have a text section.

At this point, we introduced the text and data segments as simply some blocks of memory that are allocated before a program starts executing. We will see the full significance of this segments soon.

7.4 Stack Memory

Say you are working with an architecture consisting of 32 8-bit registers. You might be working on a program that—during runtime—works with a lot more than just 32 numbers. To solve this, you take advantage of the large pool of RAM that you have already built for yourself. Say you want to store `0x18` originally in register `r1` into memory. If you do it, then you will have to store the address of `0x18` *somewhere* in one of your registers. Therefore you have simply replaced the value with its address and have not saved any space

at all! In fact, it is counterproductive since now you have to wait for slow access times into the memory. [5]

Designing the memory so that it works like a stack data structure turns out to solve this problem, along with other ones such as function call traces. In fact, the stack is so important that is pretty much universal across all computer architectures. The general idea is to simulate intermediate storage by pushing and popping temporary variables onto the stack. The stack pointer is *not* a fundamental part of memory since memory is just an array of Bytes. It is a part of the ISA.

Definition 7.31 (Stack Memory/Segment)

The **stack memory** is an address space that stores
1. *Intermediate calculations*
2. *Return addresses*
3. *Local*

The main API of the stack is at the top address, which is at the *stack pointer*.

Definition 7.32 (Stack Pointer)

The **stack pointer** is a register—denoted `sp`—that stores the address of the top of the stack.

Definition 7.33 (Push, Pop)

The push and pop operations manage data on the stack using the stack pointer.

$$\texttt{push (operand)} \quad \texttt{pop (operand)} \tag{31}$$

and support the following operands.

```
push L        // push literal L onto stack
push Ri       // push value in register Ri onto stack
push Mem[A]   // push value at memory address A onto stack
pop Ri        // pop top of stack into register Ri
pop Mem[A]    // pop top of stack into memory address A
```

Note the following details on the syntax.
1. Push decrements the stack pointer then stores the value; pop retrieves then increments.
2. We cannot pop into a literal since literals cannot be modified.
3. Stack grows downward in memory (higher addresses to lower addresses).

Definition 7.34 (Base Pointer)

Definition 7.35 (Call, Return)

The call and return operations manage function invocation and return.

$$\texttt{call (target)} \quad \texttt{ret} \tag{32}$$

and support the following target specifications for call.

```
call L        // call function at literal address L
```

[5] If you read up on the implementation of arrays in my DSA notes, you might think of putting them into a list, storing the front of the array in memory, and having a special terminating character indicating the end of an array. But not all data structures can—nor should—be nicely formatted in this list.

```
call Ri         // call function at address stored in register Ri
call Mem[A]     // call function at address stored at memory address A
call Label      // call labeled function
ret             // return from current function
```

Note the following details on the syntax.

1. Call pushes the return address onto the stack then jumps to the target.
2. Return pops the return address from the stack and jumps to it.
3. The stack pointer must be properly maintained for correct return behavior.

7.5 Functions and Control Transfer

These are really the three basic functions needed to do anything in assembly, but let's talk about an important implementation called the **control transfer**. Say that you want to compute a function.

1. Then we must retrieve the data from the memory.
2. We must load it into our registers in the CPU and perform some computation.
3. Then we must store the data back into memory.

Let's begin with a refresher on how the call stack is managed. Recall that `%rsp` is the stack pointer and always points to the top of the stack. The register `%rbp` represents the base pointer (also known as the frame pointer) and points to the base of the current stack frame. The stack frame (also known as the activation frame or the activation record) refers to the portion of the stack allocated to a single function call. The currently executing function is always at the top of the stack, and its stack frame is referred to as the active frame. The active frame is bounded by the stack pointer (at the top of stack) and the frame pointer (at the bottom of the frame). The activation record typically holds local variables for a function.

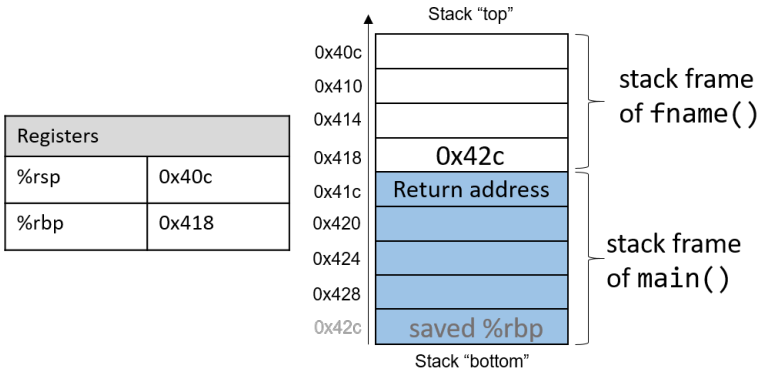


Figure 80: The current active frame belongs to the callee function (fname). The memory between the stack pointer and the frame pointer is used for local variables. The stack pointer moves as local values are pushed and popped from the stack. In contrast, the frame pointer remains relatively constant, pointing to the beginning (the bottom) of the current stack frame. As a result, compilers like GCC commonly reference values on the stack relative to the frame pointer. In Figure 1, the active frame is bounded below by the base pointer of fname, which is stack address 0x418. The value stored at address 0x418 is the "saved" `%rbp` value (0x42c), which itself is an address that indicates the bottom of the activation frame for the main function. The top of the activation frame of main is bounded by the return address, which indicates where in the main function program execution resumes once the callee function fname finishes executing.

Once we have done this we are really done. Formally, this is called Turing complete (?).

Definition 7.36 (Control Transfers)

We list some.
1. Push
2. Pop
3. Call to call a function
4. Return to return from a function
5. Continue
6. Get out of stack with leave.

Example 7.6 (Control Transfer Example)

We show this with a minimal example with psuedocode.

Example 7.7 (Multiple Functions Example)

Now what happens if there are multiple functions calling each other? Take a look at the following example with two functions.

There is a bit of a concern here from the previous example. The main function had two functions that returned two values. As the subfunction stack frame is removed from the stack, the return value is stored in the `%rax` register. If another function is called right after, then the return value of the second function will overwrite that of the previous one. This was not a problem in the previous example since the return value of the `assign` function was not used. However, if it was, then the return value of the `adder` function would have overwritten it. This is known as register saving.

1. For **caller-saved registers**, the caller function is responsible for saving the value of the register before calling a function and restoring it after the function returns. The caller should save values in its stack frame before calling the callee function, e.g. by pushing all the return values of each callee in the caller stack frame. Then it will restore values after the call.

   Therefore, if we have a set of registers {`%reg`}, the caller must take everything and push them in the caller stack frame. Then it will restore them after the call.

2. For **callee-saved registers**, it is the callee's responsibility to save any data in these registers before using the registers.

   Therefore, if we have a set of registers {`%reg`}, then inside the callee stack frame, the callee must take everything and push them in the callee stack frame. Once it computes the final return value, then it will restore all the saved register values from the callee stack frame back into the registers for the caller to use.

Ideally, we want *one* calling convention to simply separate implementation details between caller and callee. In general, however, neither is best. If the caller isn't using a register, then caller-save is better, and if callee doesn't need a register, then callee-save is better. If we do need to save, then callee save generally makes smaller programs, so we compromise and use a combination of both caller-save and callee-save.

7.6 Heap Memory

7.7 Assembling and Linking

The conversion of assembly code into a full binary executable is done through a two step process. First, the code is *assembled* into a set of *object files*, and then these object files are *linked* into a binary executable.[6]

[6]Common assemblers are `gas`, `as` and common linkers are `ld` (GNU linker) or `lld` (LLVM linker).

8 Caches

In order to actually do computations on the data stored in the memory, the CPU must first fetch the data, perform the computations, and then store the results back into memory. This can be done in two ways.

1. Load and Store Operations: CPUs use load instructions to move data from memory to registers (where operations can be performed more quickly) and store instructions to move the modified data back into memory.
2. If the data is too big to fit into the registers, the CPU will use the **cache** to store the data, and in worse cases, the actual memory itself. Compilers optimize code by maximizing the use of registers for operations to minimize slow memory access. This is why you often see assembly code doing a lot in registers.

8.1 Locality

So far, we have abstracted away most of these memory types as a single entity with nearly instantaneous access, but in practice this is not the case. The most simple way is to simply have RAM and our CPU registers, but by introducing more intermediate memory types, we can achieve greater efficiency.

Definition 8.1 (Locality)

Locality is a principle that generally states that a program that accesses a memory location n at time t is likely to access memory location $n + \epsilon$ at time $t + \epsilon$. This principle motivates the design of efficient caches.

1. **Temporal locality** is the idea that if you access a memory location, you are likely to access it again soon.
2. **Spatial locality** is the idea that if you access a memory location, you are likely to access nearby memory locations soon.

This generally means that if you access some sort of memory, the values around that address is also likely to be accessed and therefore it is wise to store it closer to your CPU. In CPUs, both the instructions and the data are stored in the cache, which exploits both kinds of locality (repeated operations for temporal and nearby data for spatial).

Example 8.1 (Locality)

Consider the following code.

```
int sum_array(int *array, int len) {
  int i;
  int sum = 0;

  for (i = 0; i < len; i++) {
    sum += array[i];
  }

  return sum;
}
```

1. **Temporal Locality**
   (a) We cycle through each loop repeatedly with the same add operation, exploiting temporal locality.
   (b) The CPU accesses the same memory (stored in variables `i`, `len`, `sum`, `array`) within each iteration and therefore at similar times.
2. **Spatial Locality**

(a) The spatial locality is exploited when the CPU accesses memory locations from each element of the array, which are contiguous in memory.
(b) Even though the program accesses each array element only once, a modern system loads more than one `int` at a time from memory to the CPU cache. That is, accessing the first array index fills the cache with not only the first integer but also the next few integers after it too. Exactly how many additional integers get moved depends on the cache's **block size**. For example, a cache with a 16 byte block size will store `array[i]` and the elements in `i+1`, `i+2`, `i+3`.

We can see the differences in spatial locality in the following example.

Example 8.2 ()

One may find that simply changing the order of loops can cause a significant speed up in your program. Consider the following code.

```c
float averageMat_v1(int **mat, int n) {
  int i, j, total = 0;

  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      // Note indexing: [i][j]
      total += mat[i][j];
    }
  }
  return (float) total / (n * n);
}
```

```c
float averageMat_v2(int **mat, int n) {
  int i, j, total = 0;

  for (j = 0; j < n; j++) {
    for (i = 0; i < n; i++) {
      total += mat[i][j];
    }
  }
  return (float) total / (n * n);
}
.
```

Figure 81: Two implementations of taking the total sum of all elements in a matrix.

It turns out that the left hand side of the code executes about 5 times faster than the second version. Consider why. When we iterate through the `i` first and then the `j`, we access the values `array[i][j]` and then by spatial locality, the next few values in the array, which are `array[i][j+1]`, ... are stored in the cache.

1. In the left hand side of the code, these next stored values are exactly what is being accessed, and the CPU can access them in the cache rather than having to go into memory.
2. In the right hand side of the code, these next values are *not* being accessed since we want to access `array[i+1][j]`, ... Unfortunately, this is not stored in the cache and so for every n^2 loops we have to go back to the memory to retrieve it.

8.2 Caches

In theory, a cache should know which subsets of a program's memory it should hold, when it should copy a subset of a program's data from main memory to the cache (or vice versa), and how it can determine whether a program's data is present in the cache. Let's talk about the third point first. It all starts off with a CPU requesting some memory address, and we want to determine whether it is in the cache or not. To do this, we need to look a little deeper into memory addresses.

Definition 8.2 (Portions of Memory Addresses)

A memory address is a m-bit number.[a] It is divided up into three portions.

1. The **tag** field with t bits at the beginning.

2. The **index** field with i bits in the middle.
3. The **offset** field with o bits at the end.

The tag plus the index together refers to the **block number**.

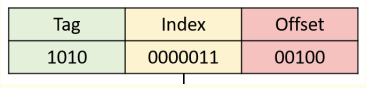


Figure 82: Portions of a 16 bit memory address with $t = 4, i = 7, o = 5$.

[a] 64 in 64-bit machines.

Before we see why we do this, we should also define the portions of a CPU.

Definition 8.3 (CPU Cache)

A **CPU cache** divides its storage space as follows. A cache is essentially an array of sets, where S is the number of sets. Each set is divided into E units called **cache lines/rows**, with each cache line independent of all others and contains two important types of information.

1. The **cache block** stores a subset of program data from main memory, of size 2^o.[a] Sometimes, the block is referred to as the cache line. Note that is the cache block size is 2^o bytes, then the block offset field has length $\log_2 2^o = o$.
2. The **metadata** stores the **valid bit** (which tells us if the actual data in memory is valid), and the **tag** of length t (the same as the tag length of the memory address) which tells us the memory address of the data in the cache.

Therefore, the **cache size** is defined to be $C = S \cdot E \cdot B$ (the metadata is not included).

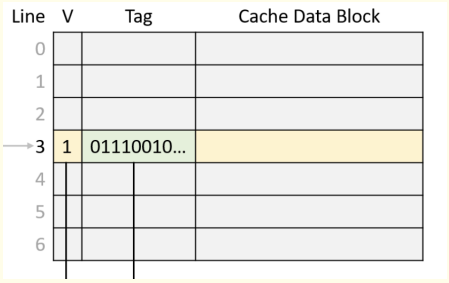


Figure 83: Diagram of a direct-mapped cache.

CPU caches are built-in fast memory (SRAM) that stores stuff. There are two types:

1. **i-cache** stores copies of instructions.
2. **d-cache** stores copies of data from commonly referenced locations.

We saw that caches come in different levels, they all just hold words retrieved from a higher level of memory.

1. CPU registers hold words retrieved from L1 cache.
2. L1 holds cache lines retrieved from L2 cache.
3. L2 cache holds cache lines retrieved from L3 cache or the main memory.
4. Main memory holds disk blocks retrieved from local disks.
5. Local disks hold blocks retrieved from remote disks or network servers.

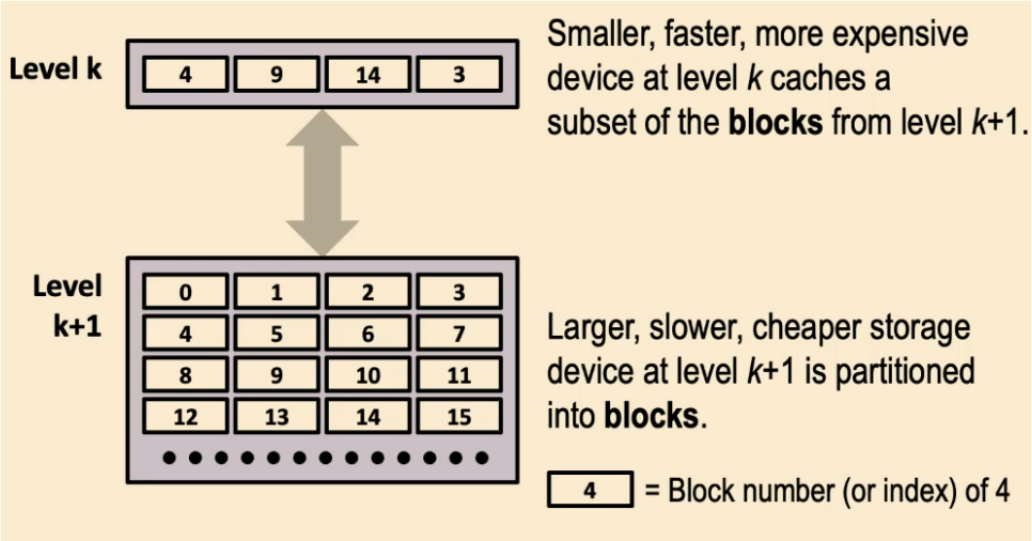


Figure 84: How caches retrieve data from higher levels of memory.

[a] In Intel computers, it is typically 64 bytes long and for Mac Silicon, it is 128 bytes.

Example 8.3 (Simple Calculations)

Given a direct-mapped cache specified by a block size of 8 bytes and a cache capacity of 4 KB,

1. the cache can hold 512 blocks.
2. the block offset field is $\log_2 8 = 3$ bits wide.
3. the address `0x1F = 0b00011111` is in block number 3 since the last three bits are the offset, and whatever is left (passed through the hashamp, which is simply modulo), is the block number.

In **I/O caches**, software keeps copies of cached items in memory, indexed by name via a hash table.

At the lowest level, registers are explicitly program-controlled, but when accessing any sort of higher memory, the CPU doesn't know whether some data is in the cache, memory, or the disk.

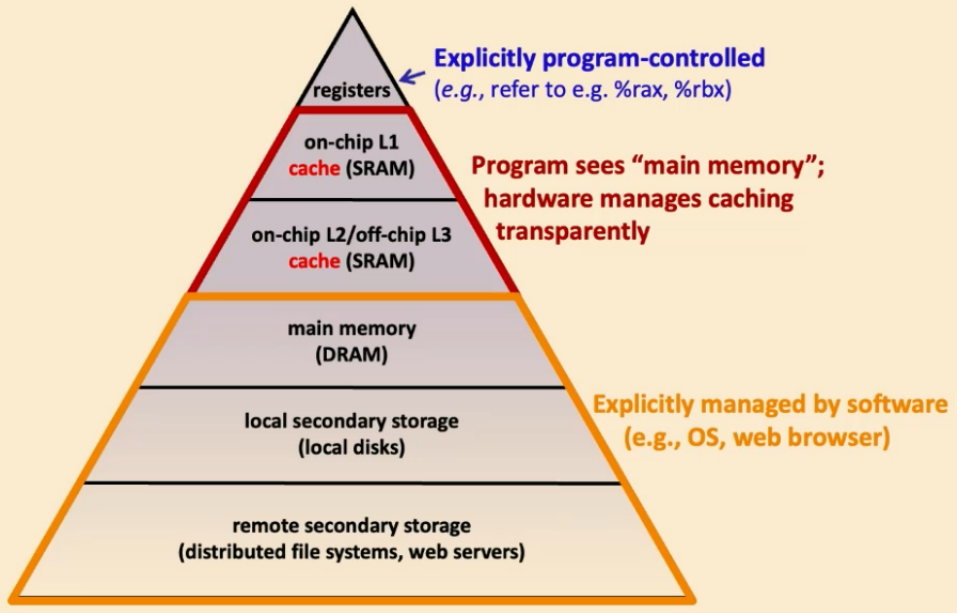


Figure 85

Finally, let's compare software vs hardware caches.

Definition 8.4 (Software Caches)

When implementing caches in software, there are large time differences (DRAM vs disk, local vs remote), and they can be tailored to specific uses cases. They also have flexible and sophisticated approaches with data structures (like trees) and can perform complex computation.

Theoretically, when implementing hash tables, you never actually have to evict something. You can have the values of the table to be a linked list where we add to the head. If there is unlimited chaining, we have a full associative cache, and if we have limited chaining (e.g. 5), it is like a 5-way set associative cache. If it goes out of bound, we can implement LRU by removing the tail of the linked list.

Definition 8.5 (Hardware Caches)

In hardware caches, there are smaller time differences, needs to be as fast as possible, and parallelization is emphasized.

There are slightly different implementations of caching, and for each implementation, we will describe

1. how to load data from memory into the cache,
2. how to retrieve data from the cache,
3. how to write data to the cache.

8.2.1 Direct Mapped Cache

A direct mapped cache is a caching implementation when we assume that $E = 1$, which means that for any given memory address, there is only one possible cache line that can store this data at that memory address. That is, the cache is really just a bunch of sets with one cache line each, and each cache line is completely isolated from the others. Whether we load data from memory into cache or try to retrieve data from the cache, it's really the same process.

Theorem 8.1 (Placement)

To load data from memory into the cache, which happens when there is a **cache miss**, we do the following.

1. The CPU requests a memory address $M = (T, I, O)$.
2. There exists a hashmap H that maps the index I to a cache line.
3. At line $H(I)$, we can get a cache miss and must load from memory into this cache.
4. We wait until the memory has retrieved the data from the portion of the memory. i.e. we wait for the 2^o bytes located at addresses $(T, I, 0 \dots 0)$ to $(T, I, 1 \dots 1)$. Call this data D.
5. The 2^o byte string D is stored in the cache data block at line $M(I)$,ready to be used.

Theorem 8.2 (Lookup)

To see whether a requested memory address is in the cache, we do the following.

1. The CPU requests a memory address $M = (T, I, O)$.
2. There exists a hashmap H that maps the index I to a cache line.
3. At line $H(I)$, check the cache line's valid bit. If it is not valid, then this is a cache miss and we must go to the memory to retrieve the data, leading to the above process.
4. Since there could be multiple I that maps to the same cache line, there will be overlap. But this is where the tag portion comes in. At cache line $H(I)$, the CPU checks the cache tag to see if it matches the memory tag T.
5. If it does, then we have just found a way to identify the first $t + i$ bits of the requested memory address, and we have gotten a cache hit. Now, we know that the cache's data block holds the data that the program is looking for. We use the low-order offset bits of the address to extract the program's desired data from the stored block.

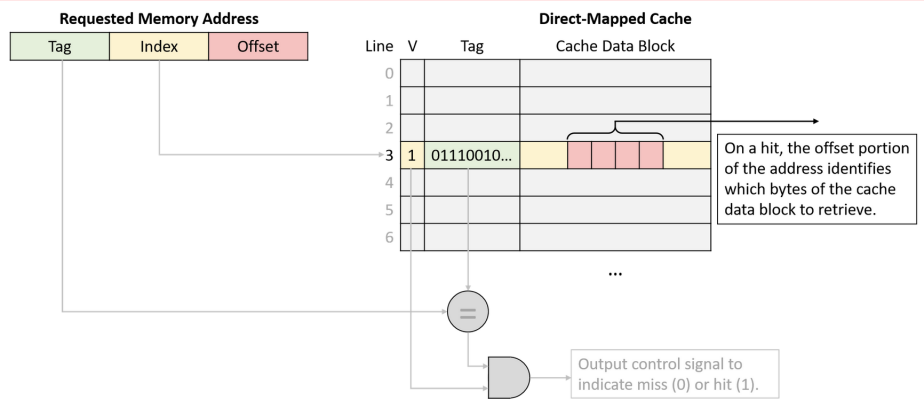


Figure 86: Diagram of a cache request. Note that since the entire data in the memory block stored in the cache, we can take advantage of spatial locality.

So far, we've talked about reading operations, but what about writing to the cache? It is generally implemented in two ways.

Definition 8.6 (Write-Through, Write-Back Cache)

Note that when we write data to cache, it does not need to be immediately written to memory, but rather it can be flushed to memory at a later time. This is efficient since if we have repeated operations on a single memory address, we don't have to go back and forth between the CPU and memory.

1. In a **write-through cache**, a memory write operation modifies the value in the cache and

simultaneously writes the value to the corresponding location in memory. It is always synchronized.

2. In a **write-back cache**, a memory write operation modifies the value stored in the cache's data block, but does *not* update main memory. Instead, the cache sets a **dirty bit** in the metadata to indicate that the cache block has been modified. The modified block is only written back to memory when the block is replaced in the cache.

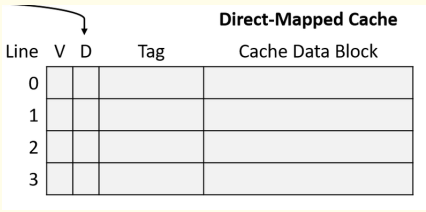


Figure 87: A dirty bit is a one bit flag that indicates whether the data stored in a cache line has been modified. When set, the data in the cache line is out o sync with main memory and must be written back (flushed) back to memory before eviction.

As usual, the difference between the designs reveals a trade-off. Write-through caches are less complex than write-back caches, and they avoid storing extra metadata in the form of a dirty bit for each line. On the other hand, write-back caches reduce the cost of repeated writes to the same location in memory.

Theorem 8.3 (Replacement)

Replacement occurs exactly the same way as if we just did a placement and is trivial. We retrieve the data block from the memory and store it in the cache. Direct-mapping conveniently determines which cache line to evict when loading new data. Given new memory $M = (T, I, O)$, you *must* evict the cache line at $H(I)$.

8.2.2 N way Set-Associative Cache

Note that for both examples, given a fixed hashmap H it is not possible to store data in two memory addresses M_1 and M_2 where both $H(I_1) = H(I_2)$. Therefore, the choice of hashing must be done so that it minimizes the number of collisions. So far, we have only considered memory read operations for which a CPU performs lookups on the cache. Caches must also allows programs to store values. However, there is a better way to do this: just construct it so that each set has more than one cache line, and so data in index portions of different memory addresses can be stored in different cache lines.

In here, we deal with $E \neq 1$, and so there are multiple set each with multiple lines. This means that the cache is more like a 2D array, and when we want to retrieve an index, we must look through the $H(I)$th line in *each* set to see if the tag matches.

Theorem 8.4 (Lookup)

To see whether a requested memory address is in the cache, we do the following.

1. The CPU requests a memory address $M = (T, I, O)$.
2. We iterate through each of the S sets in the cache, looking at cache line $M(I)$.
3. For each line, we check if it is valid and if so, whether the line tag matches the memory tag. If we get a hit, then we have found the data in the cache.

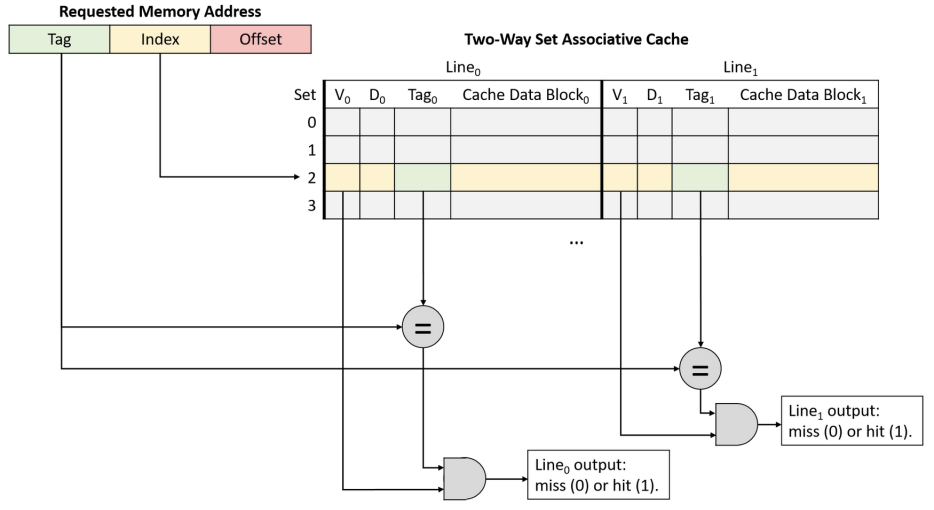


Figure 88: Diagram of a 2 set-associative cache.

If you have a **fully associative cache**, then you have one set with $E = C/B$ lines. Therefore, you can really put any memory address data in any cache line. There is a clear tradeoff here. As we increase N, we can get more flexibility in using all of our cache space, but the time complexity of retrieving and writing data scales linearly. In fact, this linear scan is too slow for a cache, which is why you need to implement some parallel tag search, but this turns out to be quite expensive to build.[7]

Though we have a more robust implementation with associative mapping, placement and replacement now face the problem of *which* set to place the data in or evict existing data.

Theorem 8.5 (Placement)

To load data from memory into the cache this is trivial since we can just go through the sets, find one where the valid bit is 0, and just place the data there.

In replacement, this is a bit trickier, but using the principle of temporal locality, we can try and replace the least recently used cache. This tries to minimize cache misses, but not slow down the lookup too much.

Theorem 8.6 (Replacement)

To replace data on the cache, we use the **least recently used (LRU)** algorithm. This matches temporal locality, but it also requires some additional state to be kept.

8.2.3 Types of Cache Misses

There are three types of cache misses.

Definition 8.7 (Cold (Compulsory) Miss)

A **cold miss** occurs when the cache is empty and the CPU requests a memory address. This is the first time the CPU is requesting this memory address, and so it must go to the memory to retrieve the data.

[7] You have to copy the request tag with a circuit and compare it to all the tags in the cache, which turns out to be a much larger circuit.

Definition 8.8 (Capacity Miss)

A **capacity miss** occurs when the cache is full and the CPU requests a memory address that is not in the cache. This is because the cache is full and so the CPU must evict some data to make space for the new data.

Definition 8.9 (Conflict Miss)

A **conflict miss** occurs from premature eviction of a warm block.

Valgrind's cachegrind mode.

9 Input Output

There are 3 file descriptors: STDIN, STDOUT, STDERR

Memory mapped IO. Port mapped IO.

10 Disk

10.1 Expanding on von Neumann Architecture

So far, our model of the computer has been a simple von Neumann architecture which consists of a CPU and memory. However, there are many other intricacies that are extremely important in practice, and we'll expand on each one by one.

Definition 10.1 (Computer Architecture)

In our elaborated computer architecture, a computer consists of the components.

1. A **CPU** that consists of an arithmetic logic unit (ALU), registers, and a **bus interface** that controls the input and output.
2. The **IO bridge** that handles communication between everything.
3. The **system bus** that connects the CPU to the IO bridge.
4. The **memory bus** that connects the memory to the IO bridge.
5. The **IO bus** that connects the IO devices and disk to the IO bridge.
6. **IO devices** like mouse, keyboard, and monitor.
7. The **disk controller and disk** that stores data.

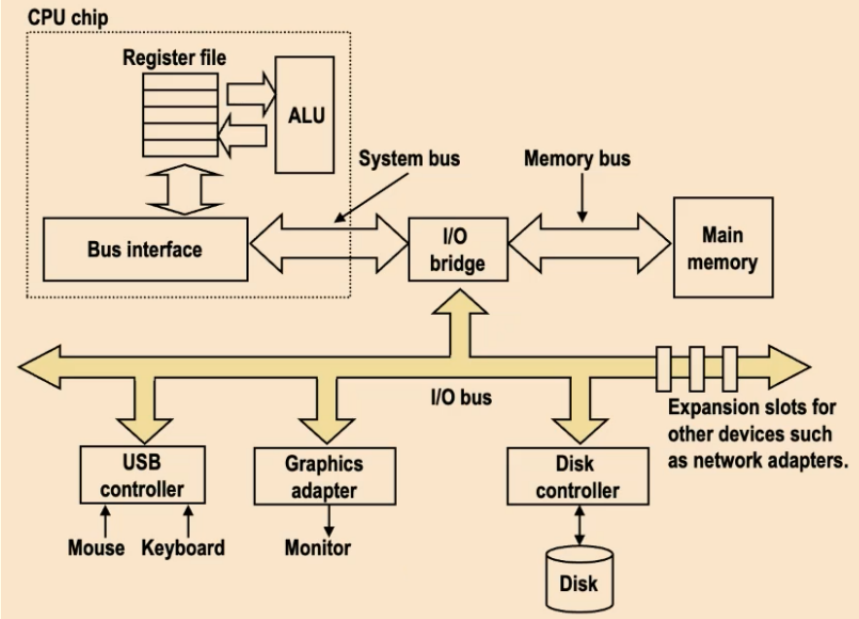


Figure 89: Diagram of the IO bus.

We can see from the diagram above that the CPU can directly access registers (since it's in the CPU itself) and the main memory (since it's connected to the memory bus). However, to access something like the disk, it must go through the disk controller. This gives us our first categorization of memory.

Definition 10.2 (Primary Storage)

Primary storage devices are directly accessible by the CPU and are used to store data that is currently being processed. This includes CPU registers, cache memory, and RAM. In memory, the basic storage unit is normally a **cell** (one bit per cell), which is the physical material that holds information. A **supercell** has address and data widths (number of bits), which is analogous to a lock number and the lock capacity, respectively. It is called random access since it takes approximately the same amount of time to access any cell in memory. There are two primary ways that this is

implemented:

1. **Static RAM (SRAM)** stores data in small electrical circuits (e.g. latches) and is typically the fastest type of memory. However, it is more expensive to build, consumers more power, and occupies more space, limiting the SRAM storage.
2. **Dynamic RAM (DRAM)** stores data using electrical components (e.g. capacitors) that hold an electrical charge. It is called *dynamic* because a DRAM system must frequently refresh the charge of its capacitors to maintain a stored value. It also requires error correction which introduces redundancy.

Device	**Capacity**	**Approx. latency**	**RAM type**
Register	4 - 8 bytes	$<$ 1 ns	SRAM
CPU cache	1 - 32 megabytes	5 ns	SRAM
Main memory	4 - 64 gigabytes	100 ns	DRAM

Table 1: Memory hierarchy characteristics

Definition 10.3 (Secondary Storage)

Secondary storage devices are not directly accessible by the CPU and are used to store data that is not currently being processed. This includes hard drives, SSDs, and magnetic tapes. There are two primary ways:

1. **Spinning disks** store data on a magnetic surface that spins at high speeds.
2. **Solid state drives (SSDs)** store data on flash memory chips.

There are three key components of memory that we should think about:

1. The **capacity**, i.e. amount of data, it can store (how large the water tank is).
2. The **latency**, i.e. amount of time it takes for a device to respond with data after it has been instructed to perform a data retrieval operation (how fast the data flows).
3. The **transfer rate** or **thoroughput**, i.e. amount of data that can be moved between the device and main memory (how wide the pipe is). Naively, with one channel and sequential transfer the transfer rate is one over the latency.

We must provide a good balance of these three qualities, and also note that there are some physical limitations (i.e. latency cannot be faster than speed of light), and this is more effectively done through a hierarchical memory system.

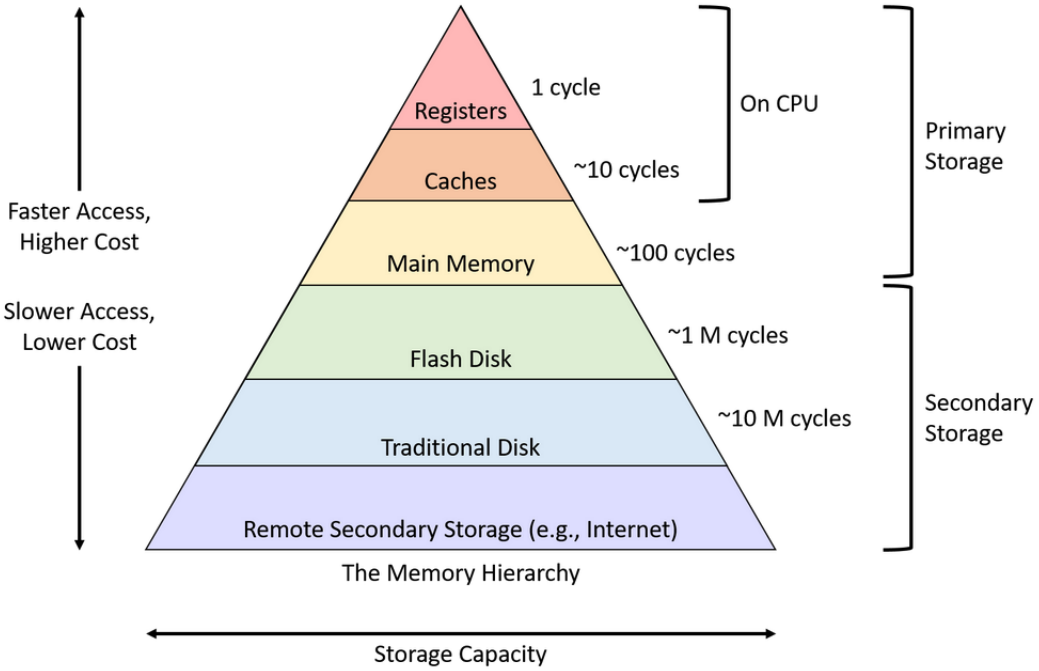


Figure 90: Memory hierarchy.

For example when we want to read from the disk, the CPU must request to the bus interface, which travels through the bus interface, I/O bridge, I/O bus, disk controller, and to the disk itself. Then the data goes back through the disk controller, I/O bus, I/O bridge, through the memory bus, and resides in the main memory. Note that disks are block addressed, so it will transfer the entire block of data into the memory. It must specify a **destination memory address (DMA)**. When the DMA completes, the disk controller notifies the CPU with an *interrupt* (i.e. asserts a special interrupt pin on the CPU), letting it know that the operation has finished. This signal goes through the disk controller to the IO bridge to the CPU. From now on, the CPU knows that there is memory that it can access to run an application loaded in memory.

10.2 Disk

Definition 10.4 (Hard Disk Drives)

Back then, there were **hard disk drives (HDDs)** that literally had a spinning wheel and a needle head that read the data.

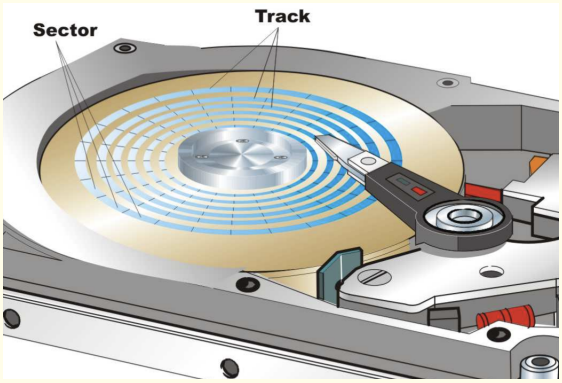


Figure 91: Visual diagram of hard disk drive with its sectors.

1. HDDs are not random access since the data must be sequentially read. This was disadvantageous since the spinning wheel had to spin to the correct location, which took time. The needle also had to move to the correct location, which also took time and therefore read and write speeds were dominated by the time it took to move the needle.
2. The smallest unit of data that can be read is a complete disk sector (not a single byte like RAM).

Definition 10.5 (Solid State Drives)

Now, we have **solid state drives (SSDs)** that store data on flash memory chips. This is advantageous since there are no moving parts, so the latency is much lower and the latency is not dominated by the time it takes to move the needle.

1. SSDs are random access.
2. The smallest unit of data is a **page**, which is usually 4KB and maybe for high scale computers 2-4 MB (but on "Big Data" applications big but computers, it can be up to 1GB).
3. A collection of pages, usually 128 pages, is called a **block**, making is 512KB.

While virtually all RAM and primary storage devices are **byte addressable** (i.e. you can access any byte in memory), secondary storage devices are **block addressable** (i.e. you can only access a block of memory at a time). Therefore, to access a single byte in secondary storage, you must first load the entire block into memory, calculate which byte from that block you want, and then access it. Therefore, you need both the block number x and the offset o to access a byte in secondary storage, which is why it is even slower than accessing RAM.

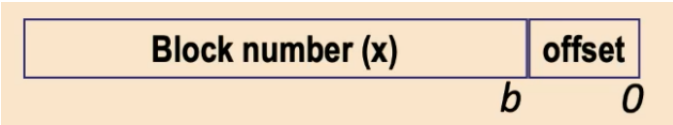


Figure 92: Block offset.

Therefore, you can think of raw data in units of blocks of size 2^b for some b bits.

1. Take the low order b bits of a byte address as an integer, which is the offset of the addressed byte in the block.
2. THe rest of the bits are the block number x, which is an unsigned long.
3. You request the block number x, receive the block contents, and then extract the requested byte at

offset in x i.e. calculate `block[x][offset]`.