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.

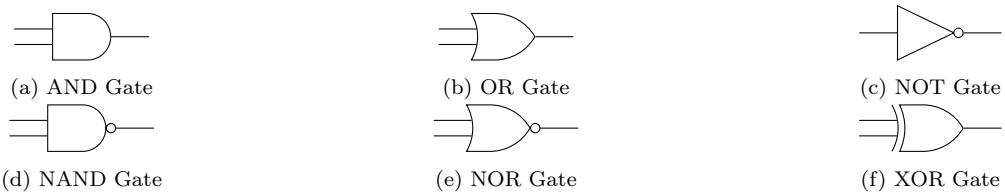
You first make silicon wafers.

1.2 Doping

1.3 Implementation of NAND

2 Logic Gates and Hardware Description Languages

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

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.

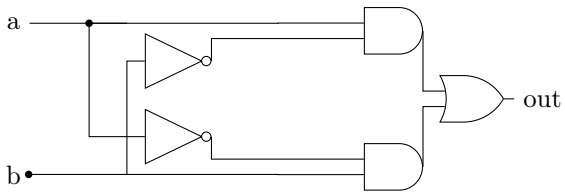


Figure 2: XOR Chip from AON gates.

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.[1]

[1]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).

Definition 2.1 (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.

2.1 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 3: 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 2.2 (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 4

Definition 2.3 (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 5

We will assume that the nand gate is always implemented.

Definition 2.4 (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 6: 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 2.5 (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 7: Gate level and dataflow level implementations in Verilog.

Definition 2.6 (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 8

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 2.1 (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
```

2.2 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 2.7 (Multi-Bit NOT Gate)

Definition 2.8 (Multi-Bit AND Gate)

Definition 2.9 (Multi-Bit OR Gate)

Definition 2.10 (Multi-Bit NAND Gate)

Definition 2.11 (Multi-Bit XOR Gate)

Definition 2.12 (Multi-Bit Multiplexor Gate)

Definition 2.13 (Multi-Bit Demultiplexor Gate)

2.3 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 2.14 (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 2.2 (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 9: Test bench module for multiplexor in GTKwave.

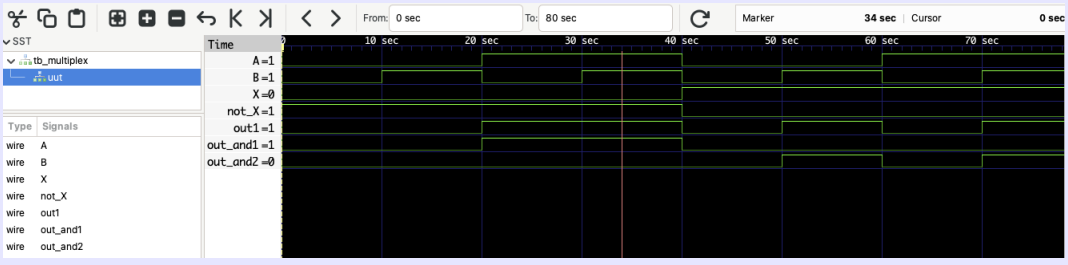


Figure 10: View in GTKwave. You want to take a signal name in the bottom left and add it to the viewer by either double clicking on it or clicking "append."

3 Encoding Schemes

The inputs of a computational program at its most fundamental level really takes in a **binary string** of 0s and 1s. Note that the choice of 0 and 1 is for convenience, but it must be binary (i.e. Boolean) in some way in order for the model to be physically implemented by transistors. Once information is in digital form, we can *compute* over it and gain insights from data that were not accessible in prior times. In fact, we can represent an unbounded variety of objects using only two symbols 0 and 1.

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.

Definition 3.1 ()

A *representation scheme* is a way to map an object x to a unique binary string $E(x) \in \{0,1\}^*$. That is, given a set of objects, E is an injective (not not necessarily surjective) map

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

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.

Definition 3.3 (Collections of Bytes)

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

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

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

3.1 Naturals/Unsigned and Integers/Signed

Definition 3.4 (Representation of the Naturals)

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

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

recursively defined as

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

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

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

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

Definition 3.5 (Representation of the Integers)

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

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

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

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

Definition 3.6 (Unsigned Integer Types in C)

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

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

Theorem 3.1 (Bit Representation of Unsigned Integers in C)

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

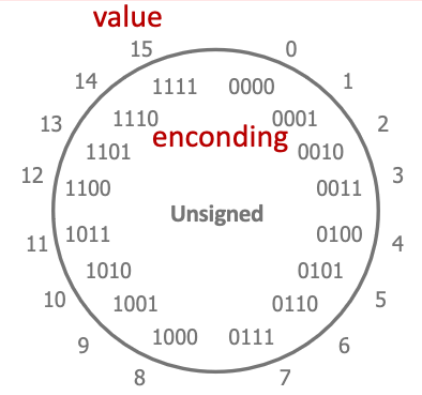


Figure 11: Unsigned encoding of 4-bit integers in C.

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 3.7 (Signed Integer Types in C)

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

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

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

Example 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.

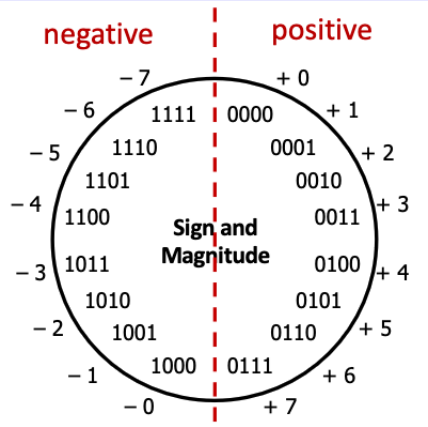


Figure 12: Signed magnitude encoding of 4-bit integers in C.

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

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

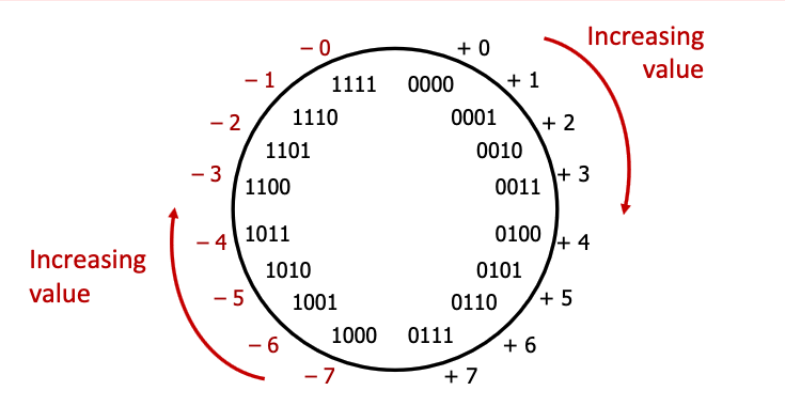


Figure 13: Two's complement encoding of 4-bit integers in C.

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 14: 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 3.8 (Representation of Rationals)

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

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

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

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

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

$$QtS \equiv p \circ q$$

Example 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

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

Proof.

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

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

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

Definition 3.9 (Floating-Point Representation)

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

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

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

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

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 3.10 (Floating Point Types in C)

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

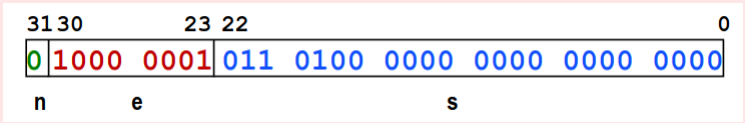
1. A **float** is 4 bytes long and can be represented as an 8-digit hex or 32 bits, with values in $[1.2 \times 10^{-38} : 3.4 \times 10^{38}]$.
2. A **double** is 8 bytes long and can be represented as an 16-digit hex or 64 bits, with values in $[2.3 \times 10^{-308} : 1.7 \times 10^{308}]$.
3. A **long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

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

Floats are actually like signed magnitude. We have

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

where



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

3.4 Characters

Definition 3.11 (Booleans in C)

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

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

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

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

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

Figure 15: 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 3.12 (ASCII)

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

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

The rest of the characters are the ASCII printable characters.

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

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

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

But even 256 characters is still not enough to cover all purposes, all languages, or even all European languages, so the emergence of *many* ASCII-derived 8-bit character sets was inevitable. Translating between these sets

(*transcoding*) is complex, especially if a character is not in both sets and was often not done, producing **mojibake** (semi-readable text resulting from text being decoded using an unintended character encoding. The result is a systematic replacement of symbols with completely unrelated ones, often from a different writing system). ASCII can also be used to create graphics, commonly called **ASCII art**.

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

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

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

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 3.13 ()

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

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

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

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

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

Definition 3.14 (Prefix)

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

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

Definition 3.15 ()

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

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

Theorem 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

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

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

Proof.

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

4 Combinational Chips

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 Addition and Subtraction

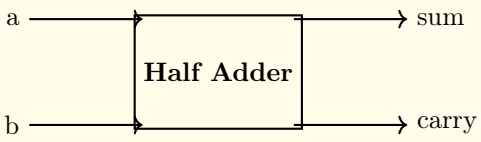
We present a hierarchy of three adders, leading to a multi-bit adder chip. Note that every single chip here represents a finite function, and so from universality of AON gates we know that an implementation is definitely possible.

Definition 4.1 (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.

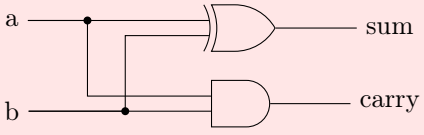


(b)

Figure 16: Chip diagram for half adder.

Theorem 4.1 (Implementation of Half-Adder)

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

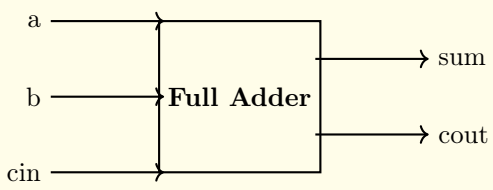
Figure 17

Definition 4.2 (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 18: Chip diagram for full adder.

Theorem 4.2 (Implementation of Full-Adder)

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

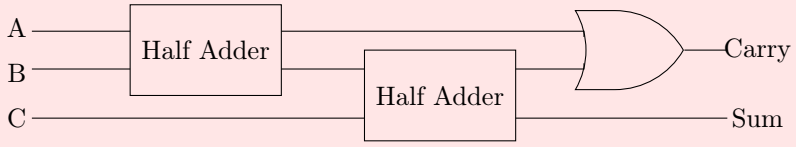


Figure 19

Definition 4.3 (N-Bit Adder)

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

Theorem 4.3 (Implementation of N-Bit Addition)

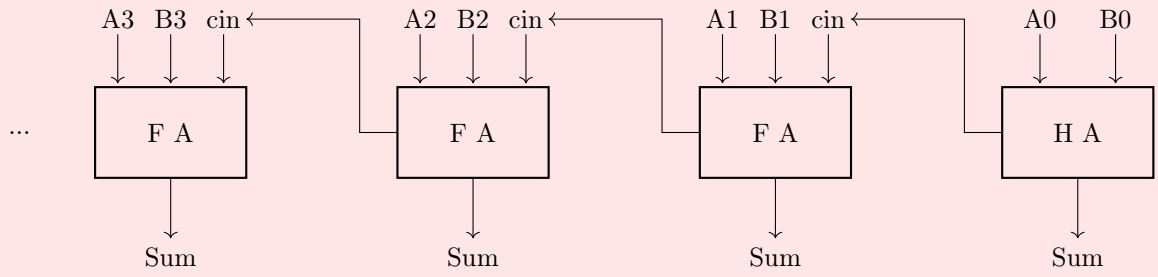


Figure 20: Ripple carry adder for the last 4 significant bits of two N-bit numbers.

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.

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.

4.2 Multiplication

Theorem 4.4 (Implementation of Bitshift Operations)

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

Theorem 4.5 (Implementation of Multiplication in Circuits)

Theorem 4.6 (Implementation of Moving Data in Circuits)

4.3 Arithmetic Logical Unit (ALU)

4.4 Conditionals

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

Theorem 4.7 (Implementation of Conditionals in Circuits)

5 Sequential Chips

So far, the Boolean/arithmetic chips we worked with are *combinational*, which compute functions that depend solely on combinations of their input values. These provide many processing functions (like the ALU), but that cannot maintain a state. Since computers must be able to not only compute values but also store and recall them, they must be equipped with memory elements that can preserve data over time. These memory elements are built from *sequential chips*.

It stores data, which we have developed encoding schemes for, plus *instructions*, which we will talk about next chapter.

The act of "remembering something" is inherently time-dependent. You remember now what has been committed to memory before. Thus, in order to build chips that remember information, we must first develop some standard means for representing the progression of time.

Definition 5.1 (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.

5.1 Latches and Flip Flops

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.[2] Therefore, given a certain signal into our circuit, the outputs remain locked—or "latched"—into a state.

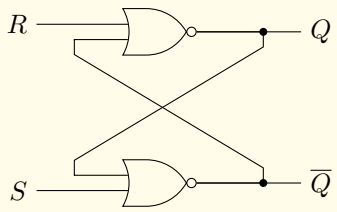
The SR latch—like all electronic circuits—require power to work, labeled with S and R. The output is really just Q, but we can add redundancy by making the inverse $\overline{Q}$ available as well.

Definition 5.2 (SR Latch)

The **set-reset (SR) latch** is a circuit to store 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. There are two implementations of an SR latch, which have symmetric behaviors.
In XOR latches,

1. The default state is $R = 0, S = 0$, i.e. they are both *low states*, and Q may be either 0 or 1. This is known as an **active high SR latch**.
2. If we send a signal $R = 1$, then $Q = 0$, and even if we reset $R = 0$, Q is still locked at 0.
3. If we send a signal $S = 1$, then $Q = 1$, and even if reset $S = 0$, Q is still locked at 1.
4. 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.

[2] Note that unlike combinational logic, this now deviates from straight-line programming.



(a) Circuit diagram.

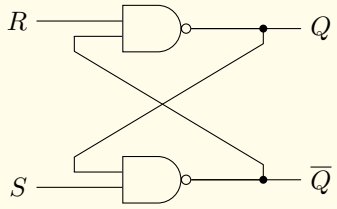
S	R	Q	$\overline{Q}$
0	0	1	0
		0	1
0	1	0	1
1	0	1	0
1	1	0	0

(b) Truth table. Note that $S = 1, R = 1$ is not a valid signal.

Figure 21: XOR SR Latch. This is

In XOR latches, this is the opposite

1. 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**.
2. If we send a signal $R = 0$, then $Q = 1$, and even if we reset $R = 1$, Q is still locked at 1.
3. If we send a signal $S = 0$, then $Q = 0$, and even if reset $S = 1$, Q is still locked at 0.
4. 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) 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

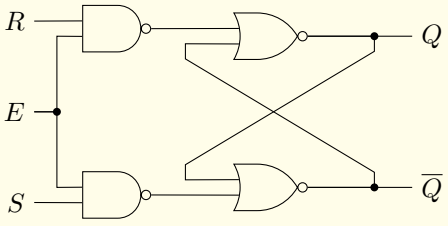
(b) Truth table. Note that $S = 0, R = 0$ is not a valid signal.

Figure 22: NAND SR Latch

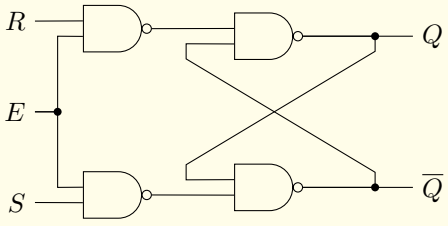
These signals may be noisy, and we might want more control over whether a latch can change states, i.e its *transparency*. This is done by adding an extra *gate* that explicitly tells us when the latch can change states.

Definition 5.3 (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 23

Example 5.1 (Gated SR Latch)

Let's run some simulations.

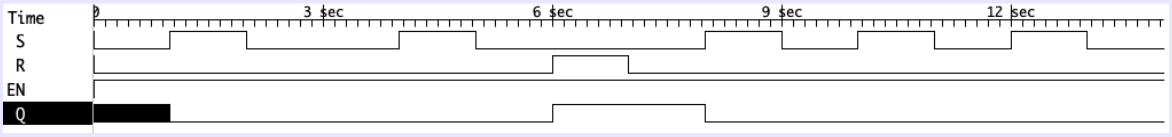


Figure 24: We keep $E = 1$ the whole time.

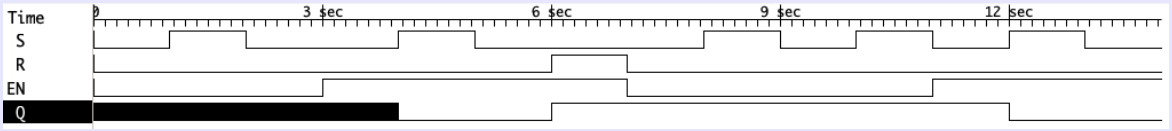


Figure 25: We enable and disable E throughout.

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.

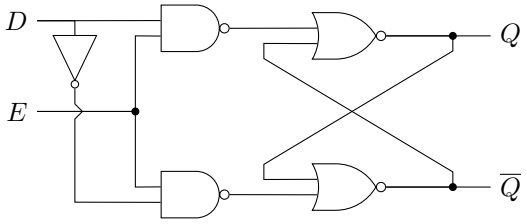
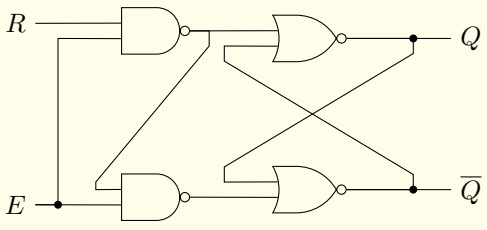


Figure 26

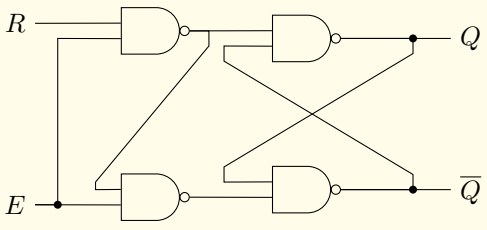
It turns out that we can simplify this circuit a bit more, giving us the gated D latch.

Definition 5.4 (Gated D Latch)

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



(a) Gated XOR SR Latch.



(b) Gated NAND SR Latch.

Figure 27

Example 5.2 (Gated D Latch)

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

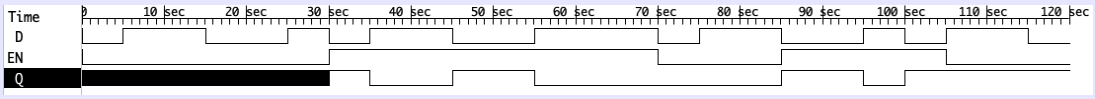


Figure 28: Gated D latch simulation.

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. So we could want something like this.

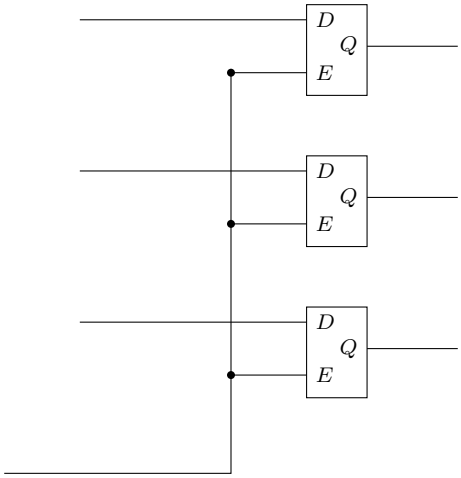


Figure 29: Multiple D-latches enabled and disabled by some external source. The system clock would be a good candidate.

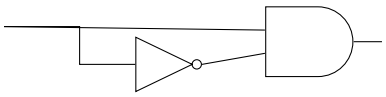


Figure 30

A clocked D latch is a D latch that is controlled by a clock signal.

Definition 5.5 (Clocked D Latch)

The most elementary chip is called a *flip-flop*, of which there are many variants. In here, we will work with the data flip-flop.

Definition 5.6 (Flip-Flop)

A **data flip-flop (DFF)** is a device whose interface consists of a single-bit data input and single-bit data output. It also has a clock input that continuously changes accordingly to the master clock's signal. Taken together, the data and the clock inputs enable to the DFF to implement the time-based

behavior

$$\text{out}(t) = \text{in}(t-1) \tag{8}$$

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

Definition 5.7 (Register)

Definition 5.8 (Counter)

5.2 Registers

To understand anything that the CPU does, we must understand assembly language. In here, everything is done within registers, and we can see how the CPU fetches, decodes, and executes instructions. So what exactly are these registers?

Definition 5.9 (Register)

A register is a small, fast storage location within the CPU. It is used to store data that is being used immediately, and is the only place where the CPU can perform operations, which is why it must move data from memory to registers before it can perform operations on it. Everything in a register is in binary, at most 8 bytes, or 64 bits.
There are very specific types of registers that you should know. All of these registers are implemented for all assembly languages and are integral to the workflow of the CPU.

1. **parameter registers** which store the parameters of a function.
2. **Return registers** which store return values of functions.
3. **stack pointers** which point to the top of the stack (at the top of the current stack frame).
4. **frame pointers** which point to the base of the current stack frame.
5. **instruction pointers** which point to the next instruction to be executed.

Definition 5.10 (32 and 64 Bit Machines)

There are two types of machines that tend to format these boxes very differently: 32-bit and 64-bit machines.

1. 32 bit machines store addresses in 32 bits, so they can have 2^{32} addresses, which is about 4 GB of memory.
2. 64 bit machines store addresses in 64 bits, so they can have 2^{64} addresses, which is about 16 EB of memory. This does not mean that the actual RAM is 16 EB, but it means that the machine can *handle* that much memory.

```
...
0x00007FFF7FBFF860 --> 0b00000000000000000000000011111111111
                       11110111111111011111111111100001100000
0x00007FFF7FBFF861 --> 0b00000000000000000000000011111111111
                       11110111111111011111111111100001100001
0x00007FFF7FBFF862 --> 0b00000000000000000000000011111111111
                       11110111111111011111111111100001100010
0x00007FFF7FBFF863 --> 0b00000000000000000000000011111111111
                       11110111111111011111111111100001100011
0x00007FFF7FBFF864 --> 0b00000000000000000000000011111111111
                       11110111111111011111111111100001100100
```

```
...
```

The numbers typically mean the size of the type that the machine works best with, so all memory addresses will be 32 or 64 bits wide. Most machines are 64-bits, and so everything in this notes will assume that we are working with a 64 bit machine. As we will later see, this is why pointers are 8 bytes long, i.e. 64 bits. This is because the memory addresses are 64 bits long, though all of them are not used.

5.3 Memory Banks

Definition 5.11 (Memory)

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

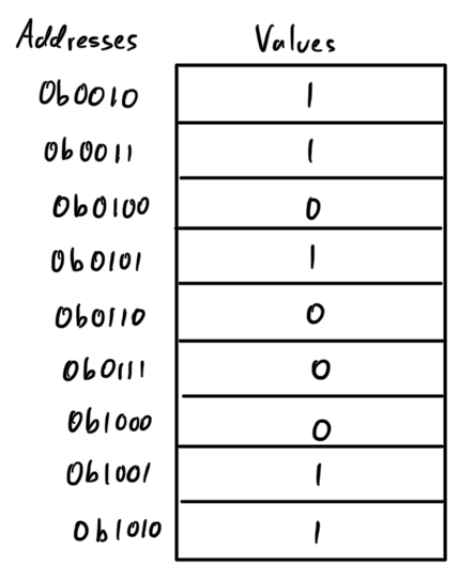


Figure 31: Visualization of memory as a long array of boxes of bits.

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

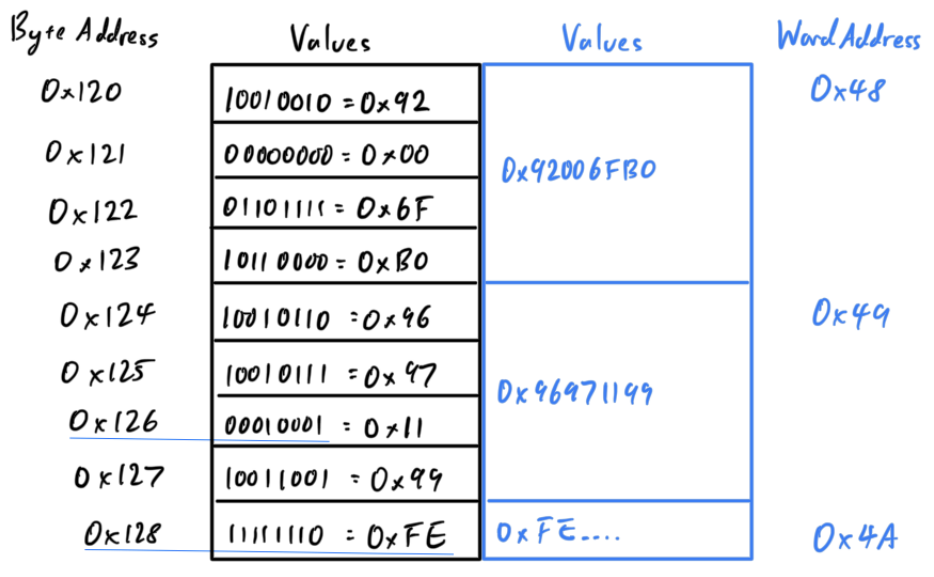


Figure 32: Visualization of memory as a long array of boxes of bytes. Every address is a byte and its corresponding value at that address is also a byte, though we represent it as a 2-digit hex.

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

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

Definition 5.12 (Endian Architecture)

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

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

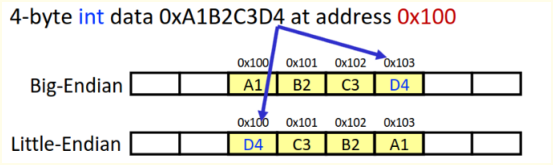


Figure 33: The big vs little endian architectures.

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

Example 5.3 (Endianness of C Int in x86-64)

To do.

Example 5.4 (Endianness of C Int in ARM64)

To do.

5.4 Counter Chips

5.5 Memory Management Unit

6 Instruction Language

6.1 Addressing Modes

Registers being 8 bytes mean that we can store memory addresses, and if we can store memory addresses, we can access memory, i.e. the values at those memory addresses. There are 4 ways to do this, called **addressing modes**: immediate, normal, displacement, and indexed. When we parse an instruction, its operands are either

1. Constant (literal) values
2. Registers
3. Memory forms

Definition 6.1 (Immediate Addressing)

Immediate addressing is when the operand is a constant value, used with a $ sign.

```
$val
```
(9)

Definition 6.2 (Normal Addressing)

Normal addressing is when the operand is a register, used with a % sign and the following syntax. The parentheses are used to dereference the memory address like dereferencing a pointer in C.

```
(R) = Mem[Reg[R]]
```
(10)

where `R` is the register name, `Reg[R]` is the value in the register, and `Mem[Reg[R]]` is the value in the memory address pointed to by the register.

Definition 6.3 (Displacement Addressing)

When we have a memory address stored in a register, we can add an offset to it to access a different memory address.

```
D(R) = Mem[Reg[R] + D]
```
(11)

where `R` is the register name and `D` is a constant displacement that specifies offset.

Definition 6.4 (Indexed Addressing)

Indexed addressing gives us more flexibility, allowing us to multiply the value in the register by a constant and add it to the value in another register. The general formula is shown as the top, but there are special cases:

```
D(Rb, Ri, S)  = Mem[Reg[Rb] + S*Reg[Ri] + D]
   D(Rb, Ri)  = Mem[Reg[Rb] + Reg[Ri] + D]
 (Rb, Ri, S)  = Mem[Reg[Rb] + S*Reg[Ri]]
    (Rb, Ri)  = Mem[Reg[Rb] + Reg[Ri]]
   (, Ri, S)  = Mem[S*Reg[Ri]]
```

where `D` is a constant displacement of 1, 2, or 4 bytes, `Rb` is the base register (can be any of 8 integer registers), `Ri` is the index register (can be any register except `rsp`), and `S` is the scale factor (1, 2, 4, or 8).

6.2 Instructions

Now let's talk about how functions work on a deeper level. When we write a command, like `int x = 4`, we are manually looking for an address (in the stack, global, or heap) and rewriting the bits that are at that address. Functions are just an automated way to do this, and all these modifications and computations are done by the CPU.

Definition 6.5 (Central Processing Unit)

The CPU is responsible for taking instructions (data) from memory and executing them.

1. The CPU is composed of **registers** (different from the cache), which are small, fast storage locations. These registers can either be **general purpose** (can be used with most instructions) or **special purpose** (can be accessed through special instructions, or have special meanings/uses, or are simply faster when used in a specific way).
2. The CPU also has an **arithmetic unit** and **logic unit**, which is responsible for performing arithmetic and logical operations.
3. 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.

It 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.

The CPU's **clock cycle** is the time it takes for the CPU to execute one instruction. More specifically, the clock cycle refers to a single oscillation of the clock signal that synchronizes the operations of the processor and the memory (e.g. fetch, decode, execute, store), and decent computers have clock cycles of at least 2.60GHz (2.6 billion clock cycles per second).

Therefore, 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.

To clarify, let us compare registers and memory. Memory is addressed by an unsigned integer while registers have names like `%rsi`. Memory is much bigger at several GB, while the total register space is much smaller at around 128 bytes (may differ depending on the architecture). 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.

The specific structure/architecture of the CPU is determined by the instruction set architecture (ISA), which can be thought of as a subset of the general computer architecture.

Definition 6.6 (Instruction Set Architecture)

The **ISA** or just **architecture** of a CPU is a high level description of what it can do. Some differences are listed here:

1. What instructions it can execute.
2. The instruction length and decoding, along with its complexity.
3. The performance vs power efficiency.

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.
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.

6.3 Instructions

Now that we've gotten a sense of what these registers are and some commonalities between them, let's do some operations on them with instructions.

Definition 6.7 (Instruction)

An instruction is a single line of assembly code. It consists of some instruction followed by its (one or more) operands. The instruction is a mnemonic for a machine language operation (e.g. `mov`, `add`, `sub`, `jmp`, etc.). The **size specifier** can be appended to this instruction mnemonic to specify the size of the operands.

1. **b** (byte) for 1 byte
2. **w** (word) for 2 bytes
3. **l** (long) for 4 bytes
4. **q** (quad word) for 8 bytes

Note that due to backwards compatibility, word means 2 bytes in instruction names. Furthermore, the maximum size is 8 bytes since that is the size of each register in x86_64. An operand can be of 3 types, 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`.

Like higher level programming languages, we can perform operations, do comparisons, and jump to different parts of the code. Instructions can be generally categorized into three types:

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
```

Now unlike compiled languages, which are translated into machine code by a compiler, assembly code is translated into machine code through a two-step process. First, we **assemble** the assembly code into an **object file** by an **assembler**, and then we **link** the object file into an executable by a **linker**. Some common assemblers are **NASM** (Netwide Assembler) and **GAS/AS** (GNU Assembler), and common linkers are **ld** (GNU Linker) and **lld** (LLVM Linker), both installable with **sudo pacman -S nasm ld**.

6.3.1 Moving and Arithmetic

Again, it is more important to have a general feel of what instructions every assembly language should and get the ideas down rather than the syntax. We list them here, beginning with simply moving.

Definition 6.8 (Moving)

Next we want to have some sort of arithmetic to do calculations and to compare values.

Definition 6.9 (Arithmetic Operations)

6.3.2 Conditionals

Definition 6.10 (Conditionals)

6.3.3 Control Transfer on Stack

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.

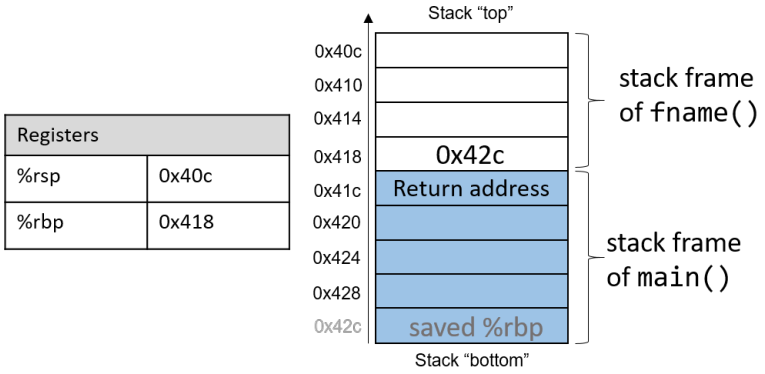


Figure 34: The current active frame belongs to the callee function (fname). The memory between the stack pointer and the frame pointer is used for local variables. The stack pointer moves as local values are pushed and popped from the stack. In contrast, the frame pointer remains relatively constant, 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 6.11 (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 6.1 (Control Transfer Example)

We show this with a minimal example with psuedocode.

6.3.4 Multiple Functions

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

Example 6.2 (Multiple Functions Example)

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 Instruction Set Architectures

7.1 Control Unit

7.2 x86

7.3 ARM

7.4 RISC-V

8 Storage Hierarchy

8.1 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 8.1 (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.

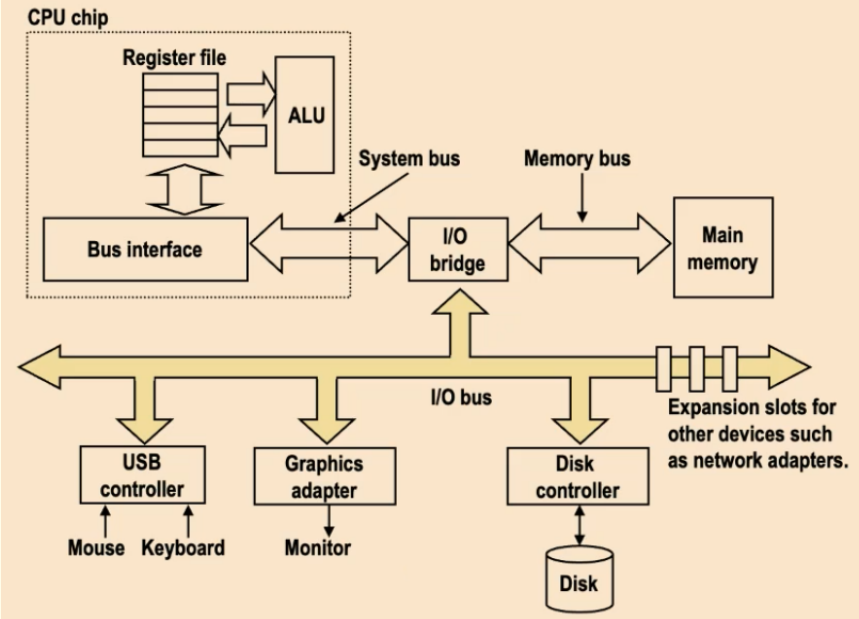


Figure 35: Diagram of the IO bus.

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 8.2 (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 8.3 (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.

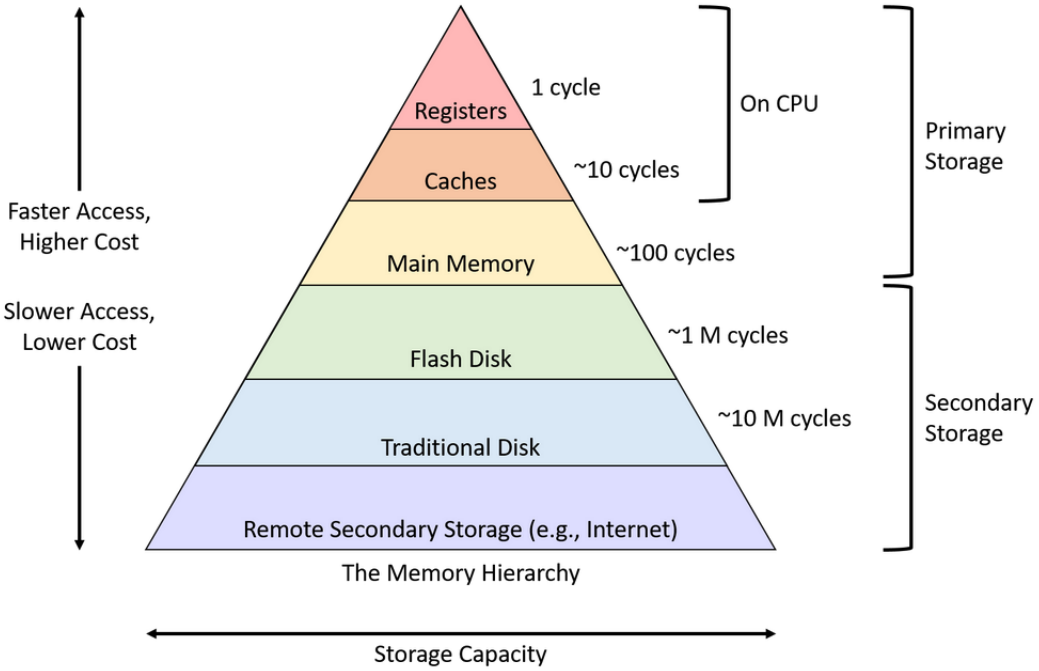


Figure 36: Memory hierarchy.

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.

8.2 Disk

Definition 8.4 (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.

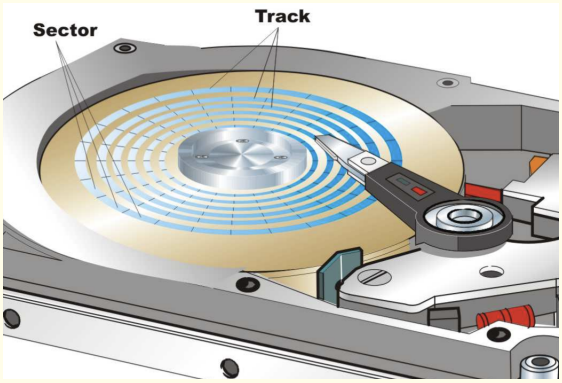


Figure 37: Visual diagram of hard disk drive with its sectors.

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 8.5 (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.

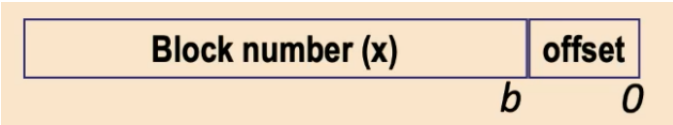


Figure 38: Block offset.

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]`.

8.3 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 8.6 (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 39: 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.

8.4 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 8.7 (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**.

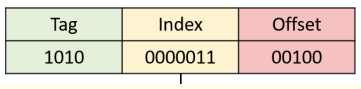


Figure 40: Portions of a 16 bit memory address with $t = 4, i = 7, o = 5$.

[a] 64 in 64-bit machines.

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

Definition 8.8 (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).

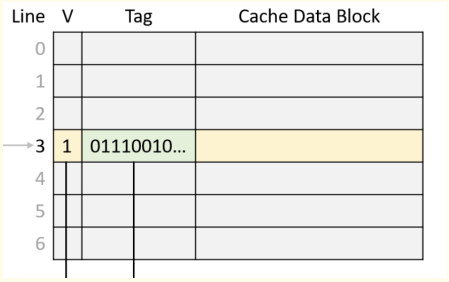


Figure 41: Diagram of a direct-mapped cache.

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.

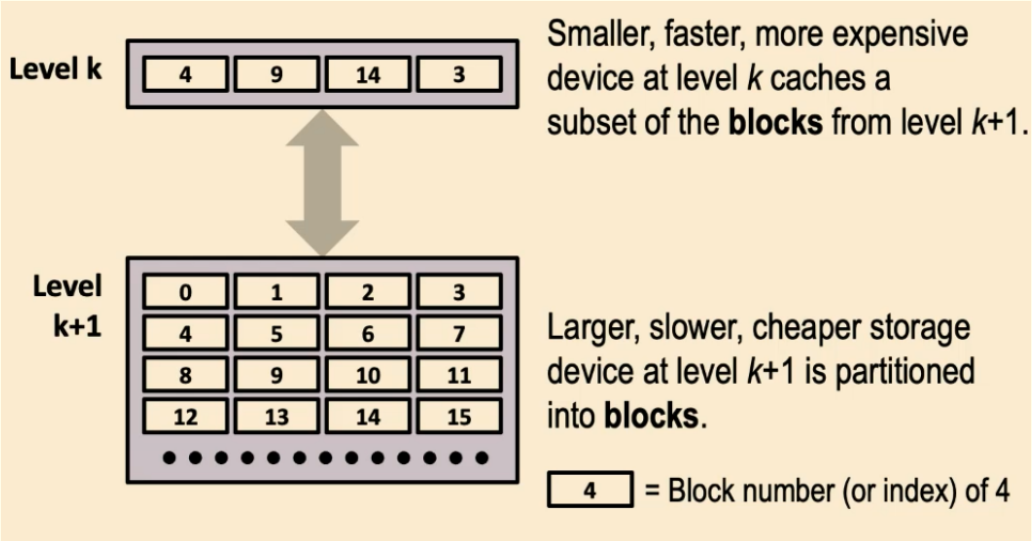


Figure 42: How caches retrieve data from higher levels of memory.

[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.

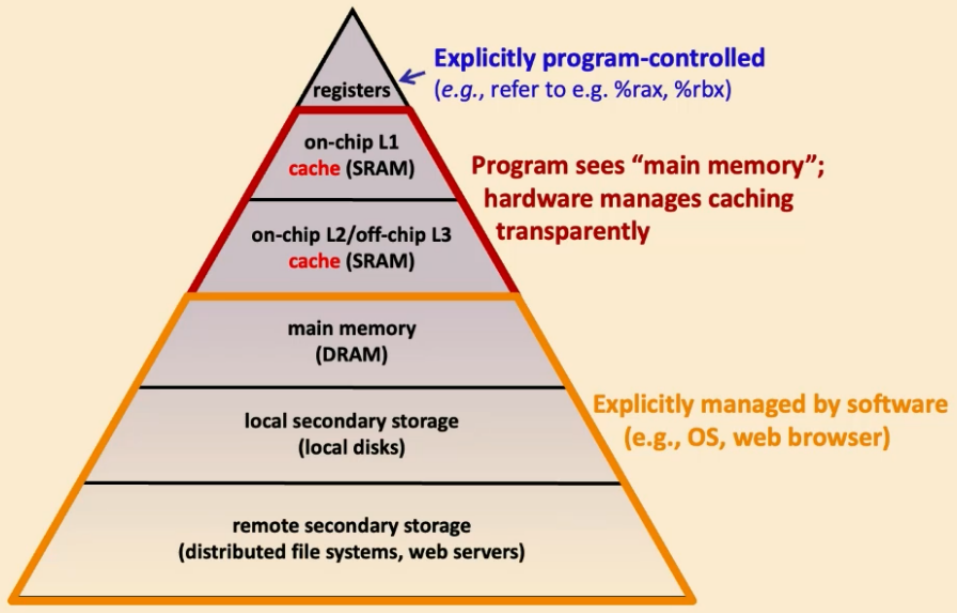


Figure 43

Finally, let's compare software vs hardware caches.

Definition 8.9 (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 8.10 (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.

8.4.1 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.

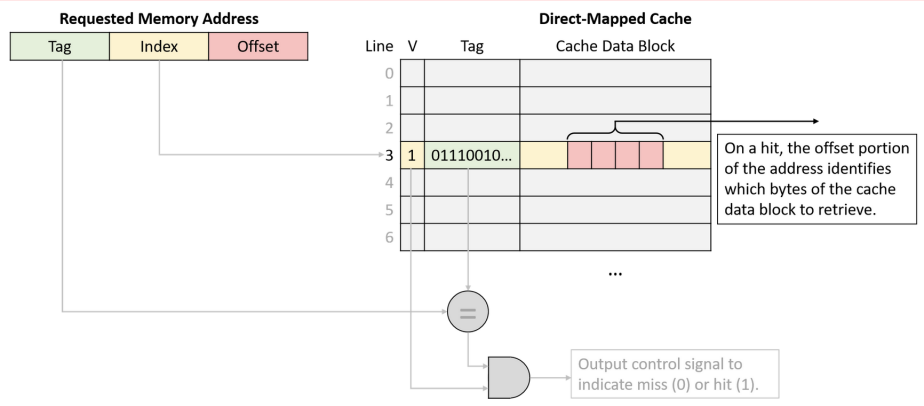


Figure 44: Diagram of a cache request. Note that since the entire data in the memory block stored in the cache, we can take advantage of spatial locality.

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

Definition 8.11 (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.

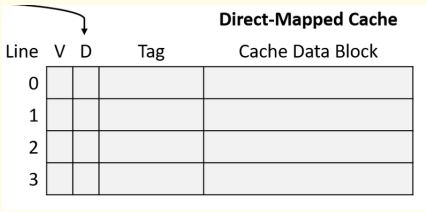


Figure 45: A dirty bit is a one bit flag that indicates whether the data stored in a cache line has been modified. When set, the data in the cache line is out o sync with main memory and must be written back (flushed) back to memory before eviction.

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 $H(I)$.

8.4.2 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.

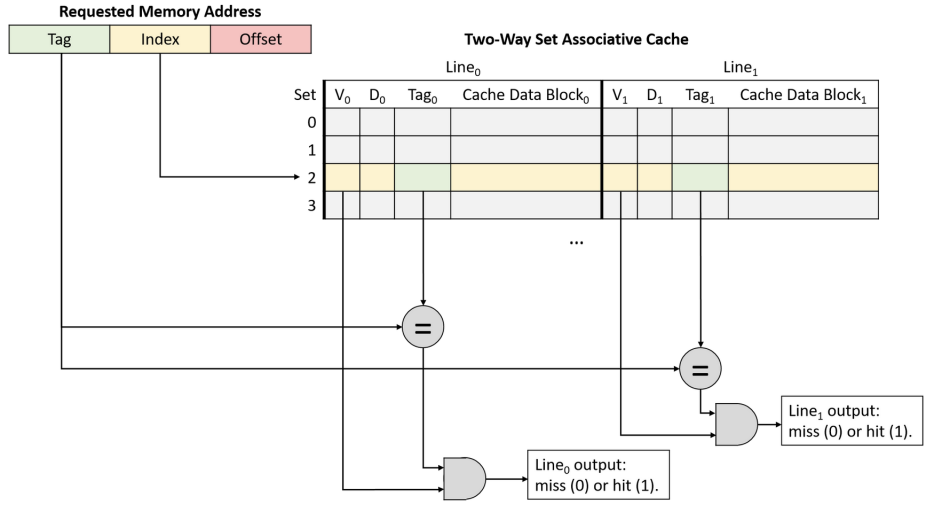


Figure 46: Diagram of a 2 set-associative cache.

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.[3]

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.

8.4.3 Types of Cache Misses

There are three types of cache misses.

Definition 8.12 (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.

[3]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 8.13 (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 8.14 (Conflict Miss)

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

Valgrind's cachegrind mode.