# Computer Science Fundamentals:
Intro to Algorithms, Systems, & Data Structures

Christian J. Rudder

October 2024

## Contents

*This page is left intentionally blank.*

## Preface

Big thanks to **Christine Papadakis-Kanaris**
for teaching Intro. to Computer Science II,
**Dora Erdos** and **Adam Smith**
for teaching BU CS330: Introduction to Analysis of Algorithms
With contributions from:
**S. Raskhodnikova, E. Demaine, C. Leiserson, A. Smith, and K. Wayne**,
at Boston University

*Please note:* These are my personal notes, and while I strive for accuracy, there may be errors. I encourage you to refer to the original slides for precise information.
Comments and suggestions for improvement are always welcome.

Prerequisites

**Theorem 0.1: Common Derivatives**

| | |
|---|---|
| Power Rule: For $n \neq 0$ | $\frac{d}{dx}(x^n) = n \cdot x^{n-1}$ . E.g., $\frac{d}{dx}(x^2) = 2x$ |
| Derivative of a Constant: | $\frac{d}{dx}(c) = 0$ . E.g., $\frac{d}{dx}(5) = 0$ |
| Derivative of ln: | $\frac{d}{dx}(\ln x) = \frac{1}{x}$ |
| Derivative of $\log_a$: | $\frac{d}{dx}(\log_a x) = \frac{1}{x \ln a}$ |
| Derivative of $\sqrt{x}$: | $\frac{d}{dx}(\sqrt{x}) = \frac{1}{2\sqrt{x}}$ |
| Derivative of function $f(x)$: | $\frac{d}{dx}(x) = 1$ . E.g., $\frac{d}{dx}(5x) = 5$ |
| Derivative of the Exponential Function: | $\frac{d}{dx}(e^x) = e^x$ |

**Theorem 0.2: L'Hopital's Rule**

Let $f(x)$ and $g(x)$ be two functions. If $\lim_{x\to a} f(x) = 0$ and $\lim_{x\to a} g(x) = 0$, or $\lim_{x\to a} f(x) = \pm\infty$ and $\lim_{x\to a} g(x) = \pm\infty$, then:

$$\lim_{x\to a} \frac{f(x)}{g(x)} = \lim_{x\to a} \frac{f'(x)}{g'(x)}$$

Where $f'(x)$ and $g'(x)$ are the derivatives of $f(x)$ and $g(x)$ respectively.

**Theorem 0.3: Exponents Rules**

For $a, b, x \in \mathbb{R}$, we have:

$$x^a \cdot x^b = x^{a+b} \text{ and } (x^a)^b = x^{ab}$$

$$x^a \cdot y^a = (xy)^a \text{ and } \frac{x^a}{y^a} = \left(\frac{x}{y}\right)^a$$

**Note:** The := symbol is short for "is defined as." For example, $x := y$ means $x$ is defined as $y$.

**Definition 0.1: Logarithm**

Let $a, x \in \mathbb{R}$, $a > 0$, $a \neq 1$. Logarithm $x$ base $a$ is denoted as $\log_a(x)$, and is defined as:

$$\log_a(x) = y \Longleftrightarrow a^y = x$$

Meaning *log* is inverse of the exponential function, i.e., $\log_a(x) := (a^y)^{-1}$.

**Tip:** To remember the order $log_a(x) = a^y$, think, "base $a$," as $a$ is the base of our *log* and $y$.

**Theorem 0.4: Logarithm Rules**

For $a, b, x \in \mathbb{R}$, we have:

$$\log_a(x) + \log_a(y) = \log_a(xy) \text{ and } \log_a(x) - \log_a(y) = \log_a\left(\frac{x}{y}\right)$$

$$\log_a(x^b) = b\log_a(x) \text{ and } \log_a(x) = \frac{\log_b(x)}{\log_b(a)}$$

**Definition 0.2: Permutations**

Let $n \in \mathbb{Z}^+$. Then the number of distinct ways to arrange $n$ objects in order is $n! := n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$. When we choose $r$ objects from $n$ objects, it's Denoted:

$$^nP_r := \frac{n!}{(n-r)!}$$

Where $P(n,r)$ is read as "$n$ permute $r$."

**Definition 0.3: Combinations**

Let $n$ and $k$ be positive integers. Where order doesn't matter, the number of distinct ways to choose $k$ objects from $n$ objects is it's *combination.* Denoted:

$$\binom{n}{k} := \frac{n!}{k!(n-k)!}$$

Where $\binom{n}{k}$ is read as "$n$ choose $k$.", and $\binom{\cdot}{\cdot}$, the *binomial coefficient.*

**Theorem 0.5: Binomial Theorem**

Let $a$ and $b$ be real numbers, and $n$ a non-negative integer. The binomial expansion of $(a+b)^n$ is given by:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$$

which expands explicitly as:

$$(a+b)^n = \binom{n}{0}a^n + \binom{n}{1}a^{n-1}b + \binom{n}{2}a^{n-2}b^2 + \cdots + \binom{n}{n-1}ab^{n-1} + \binom{n}{n}b^n$$

where $\binom{n}{k}$ represents the binomial coefficient, defined as:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

for $0 \le k \le n$.

**Theorem 0.6: Binomial Expansion of $2^n$**

For any non-negative integer $n$, the following identity holds:

$$2^n = \sum_{i=0}^{n} \binom{n}{i} = (1+1)^n.$$

**Definition 0.4: Well-Ordering Principle**

Every non-empty set of positive integers has a least element.

**Definition 0.5: "Without Loss of Generality"**

A phrase that indicates that the proceeding logic also applies to the other cases. i.e., For a proposition not to lose the assumption that it works other ways as well.

**Theorem 0.7: Pigeon Hole Principle**

Let $n, m \in \mathbb{Z}^+$ with $n < m$. Then if we distribute $m$ pigeons into $n$ pigeonholes, there must be at least one pigeonhole with more than one pigeon.

**Theorem 0.8: Growth Rate Comparisons**

Let $n$ be a positive integer. The following inequalities show the growth rate of some common functions in increasing order:

$$1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!$$

These inequalities indicate that as $n$ grows larger, each function on the right-hand side grows faster than the ones to its left.

# — 1 —
# Circuits and Logic

### 1.0.1 Sequential Logic: Building Memory Latches

If we want to have a circuit that can **store** information, say "Do $x$ if the previous input was $y$" (E.g., traffic lights).

To start building intuition lets start by seeing what happens when we connect gate outputs to other gate inputs in a loop, creating a **feedback loop** [2]:

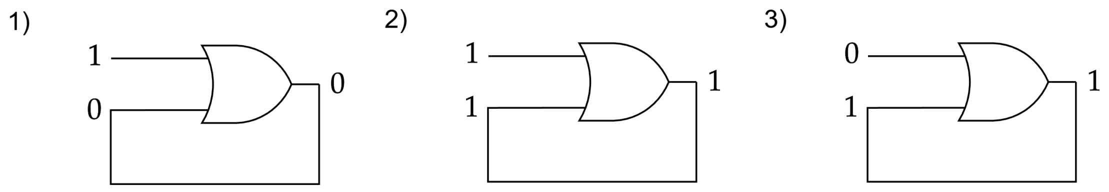


Figure 1.1: A simple feedback loop using OR gates. 1) Initially both inputs are zero, then the free input is set to 1. 2) The output becomes 1, switching the feedback input to 1. 3) Now even if the free input is set back to 0, the output remains 1, since one of the OR inputs is still 1.

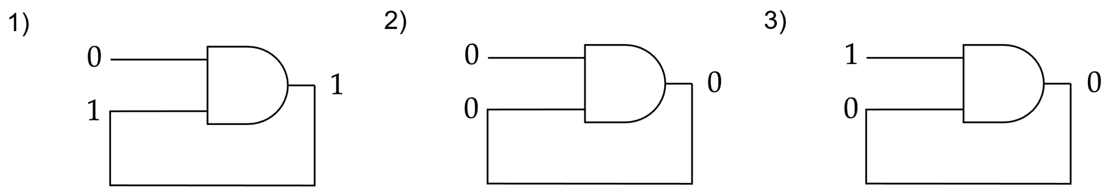


Figure 1.2: A simple feedback loop using AND gates. 1) Initially both inputs are one, then the free input is set to 0. 2) The output becomes 0, switching the feedback input to 0. 3) Now even if the free input is set back to 1, the output remains 0, since one of the AND inputs is still 0.

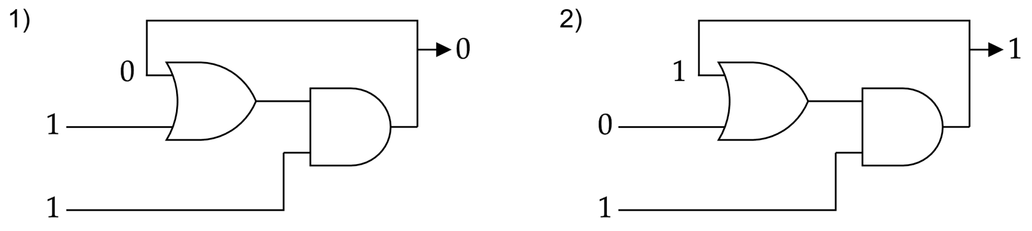


Figure 1.3: Combining the OR and AND feedback loops from Figures (1.1) and (1.2), we get the above. 1) Initially both inputs are 0, output 0, then inputs both are set to 1, resulting in a a 1 output. 2) The output is now 1, and the first input is set to 0, but the output remains 1, as the second input is still 1, driving the AND gate.

In the above Figure (1.3), we can see that turning on the second input, effectively resets the output to 0. Lets see what happens when we keep the second input on with an inverter:
This circuit is called an:

**Definition 0.1: AND-OR Latch**

An **AND-OR Latch** is a basic memory element that can store one bit of information. It has two inputs, labeled 'S' (Set) and 'R' (Reset), and a single output 'Q'.

However, this design has a critical flaw: if both 'S' and 'R' are set to 1 simultaneously creates an **invalid state**, and the output hangs on 0, ignoring the 1 that's being "set".

Next we create a more sophisticated latch that avoids this invalid state.

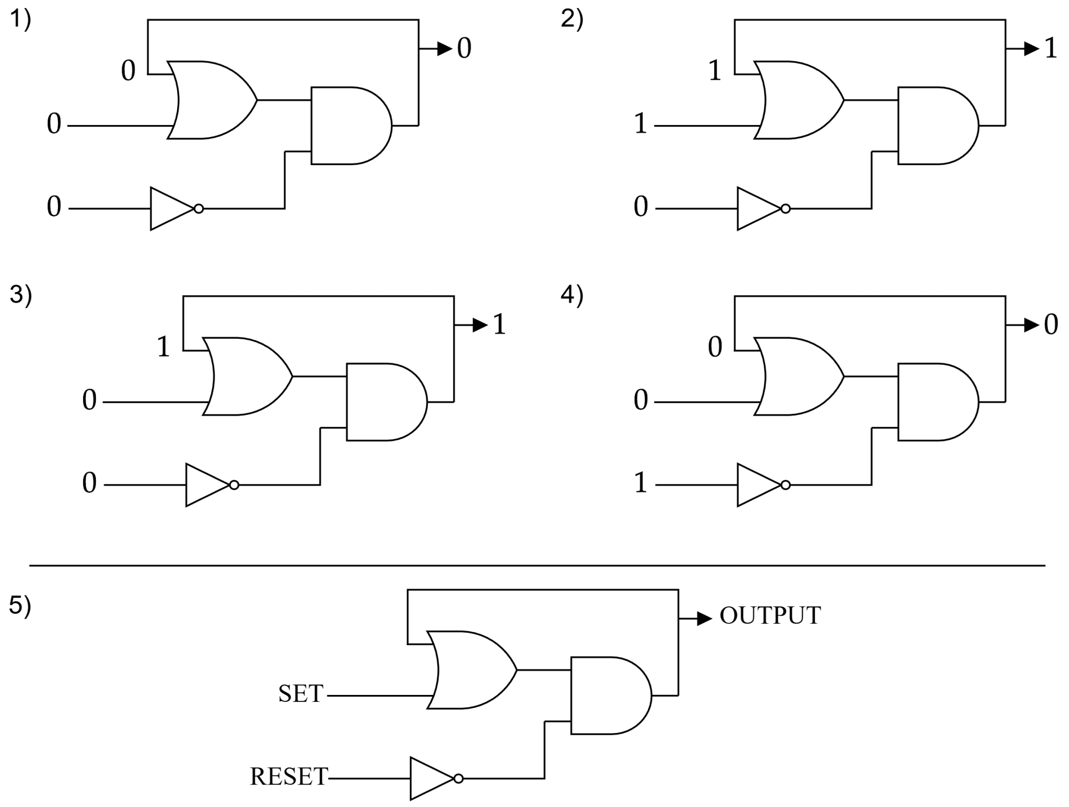


Figure 1.4: 1) Initially all inputs are 0, output 0. 2) First input is set to 1, output becomes 1. 3) First input is set back to 0, but output remains 1. 4) Second input is set to 1, output becomes 0. 5) We characterize the first input as the 'SET' and the second input as the 'RESET'.

Let's work through improving the design of the AND-OR latch step by step:

**Definition 0.2: Gated Latch**

A **Gated Latch** is a memory element that stores one bit of information. It has three inputs: 'Data', 'Write Enable', and a single output 'Q'. When 'Write Enable' is high, the value on the 'Data' input is stored in the latch and reflected on the output 'Q'.

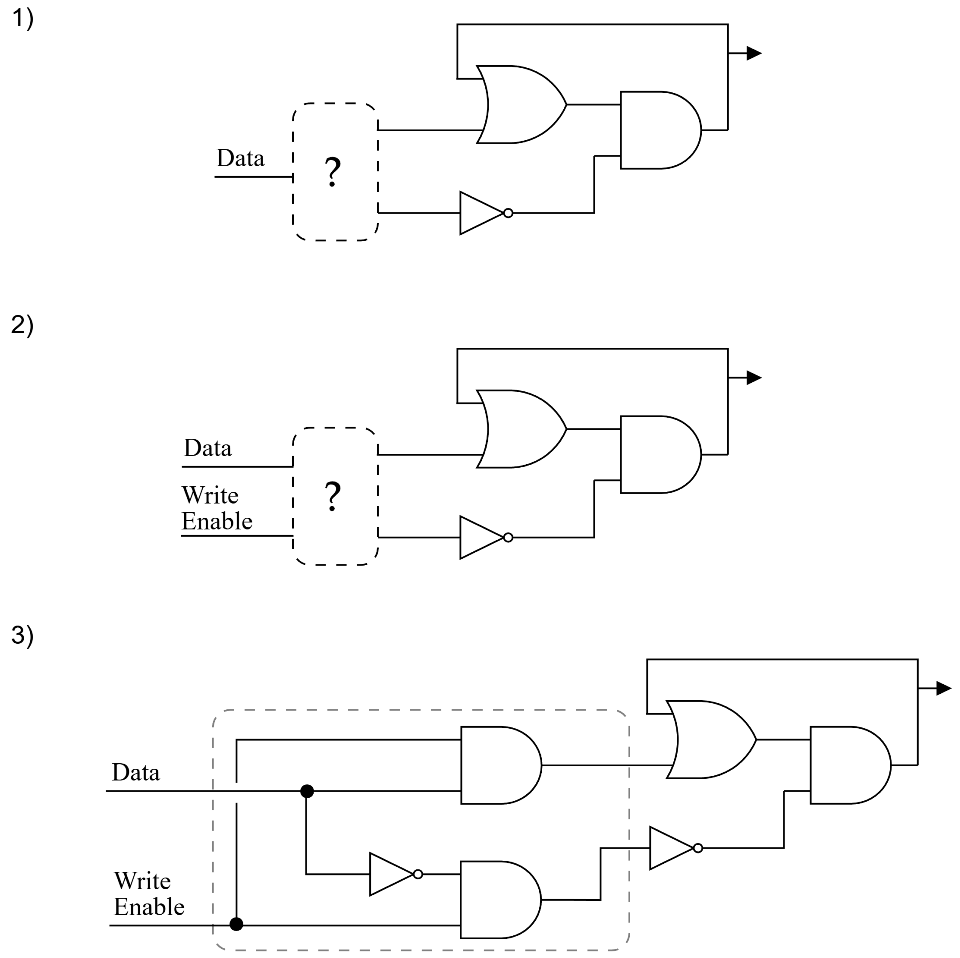


Figure 1.5: 1) We attempt to combine the SET and RESET lines to drive our ‘Data’; However, this doesn’t store information, as the output immediately follows the input. 2) We introduce a ‘Write Enable’ line. Now the data input only is written when we drive Write Enable high. 3) Is the combinational logic we were abstracting—Test it out!

Let’s do the same thing but with MUXs:

**Definition 0.3: Data Latch (D-Latch)**

A **Data Latch** (D-Latch) is a memory element that stores one bit of information. It has one select line ‘Gate’, a ‘Data’ input, a feedback input which store the last output value.

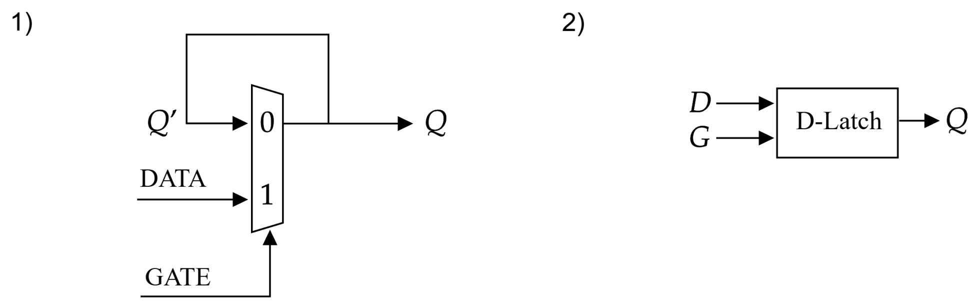


Figure 1.6: 1) Shows our MUX configuration. 'Gate' selects which line to output: 'Data' or $Q'$. Data contains incoming data, while $Q'$ is the feedback line (stores the last output). 2) Is an abstracted interface for what we call a **Data Latch** (D-Latch). [3]

Now we can build interesting circuits with combinational logic and memory:

**Definition 0.4: Sequential Circuit**

A **Sequential Circuit** combines combinational logic with memory devices (like D-Latches) to perform operations that depend on both current past inputs. The memory devices write enable line is often controlled by an oscillating clock signal (high and low). [3]

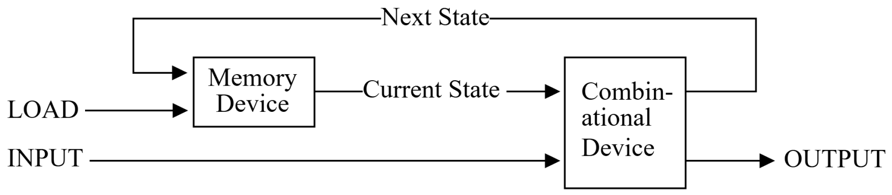


Figure 1.7: A High-level idea of a sequential circuit; 'LOAD' possibly controlled by a clock, 'INPUT' perhaps a new button press.

Ignoring our one-bit storage limitation for a moment, say we wanted to load a 1 into a D-latch controlled integer-sum circuit (initially 0):

**Note:** Timing is everything in sequential logic circuits. Though we will address this glaring issue shortly, timing electrical charges throughout an entire circuit is a complex issue. We will not address every single nuance when it comes to electrical charges, but rather focus on the

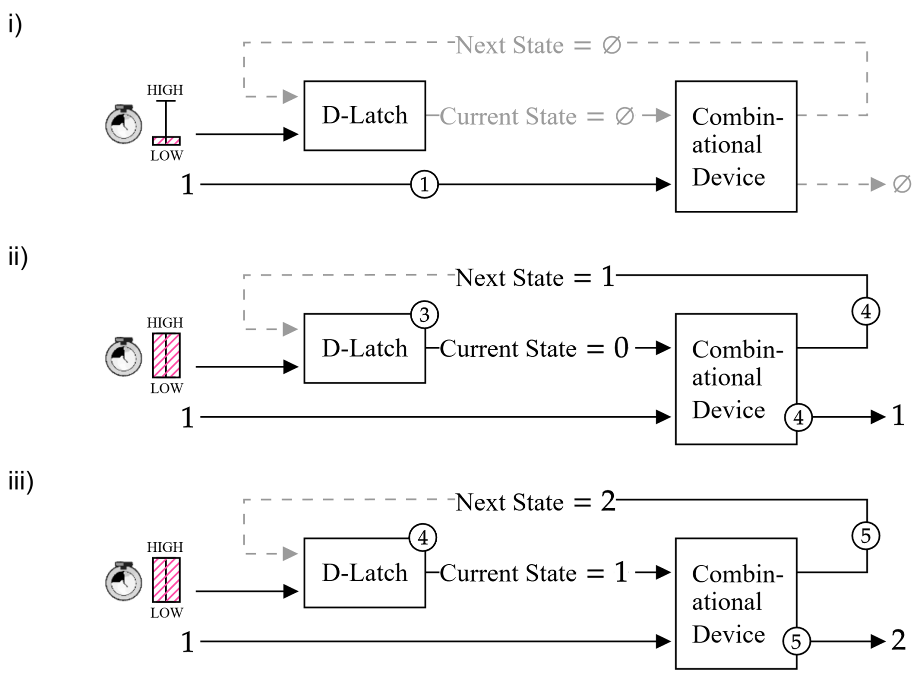


Figure 1.8: A D-latch (DL) controlled integer-sum circuit. The ∅ symbolizes no input/output (off), and the bubbled numbers indicate order of events. i) Our starting state, we input 1, and since the DL isn't enabled yet, there is no output from the combinational device (CD). ii) The clock goes high, enabling the DL. The CD outputs 1 while sending it back to the DL. iii) Again, the clock is still high, so the DL writes 1 back to the CD, outputting 2 while sending 2 back to the DL. This would continue indefinitely, **causing an infinite loop**.

high level ideas, which could be extended to other systems that perhaps don't utilize the same finicky electrical properties; For example, these systems could be built one-to-one in a popular sandbox game called Minecraft, or even with water and pipes.

Our goal is to only register the input value once, i.e., when we turn it on, i.e., the rising edge. Now just as a ticket booth person needs a bouncer to only let one person in at a time to register their ticket, we need a buffer to stop the signal from propagating through the circuit multiple times. What if we created our digital "bouncer" with another D-Latch?

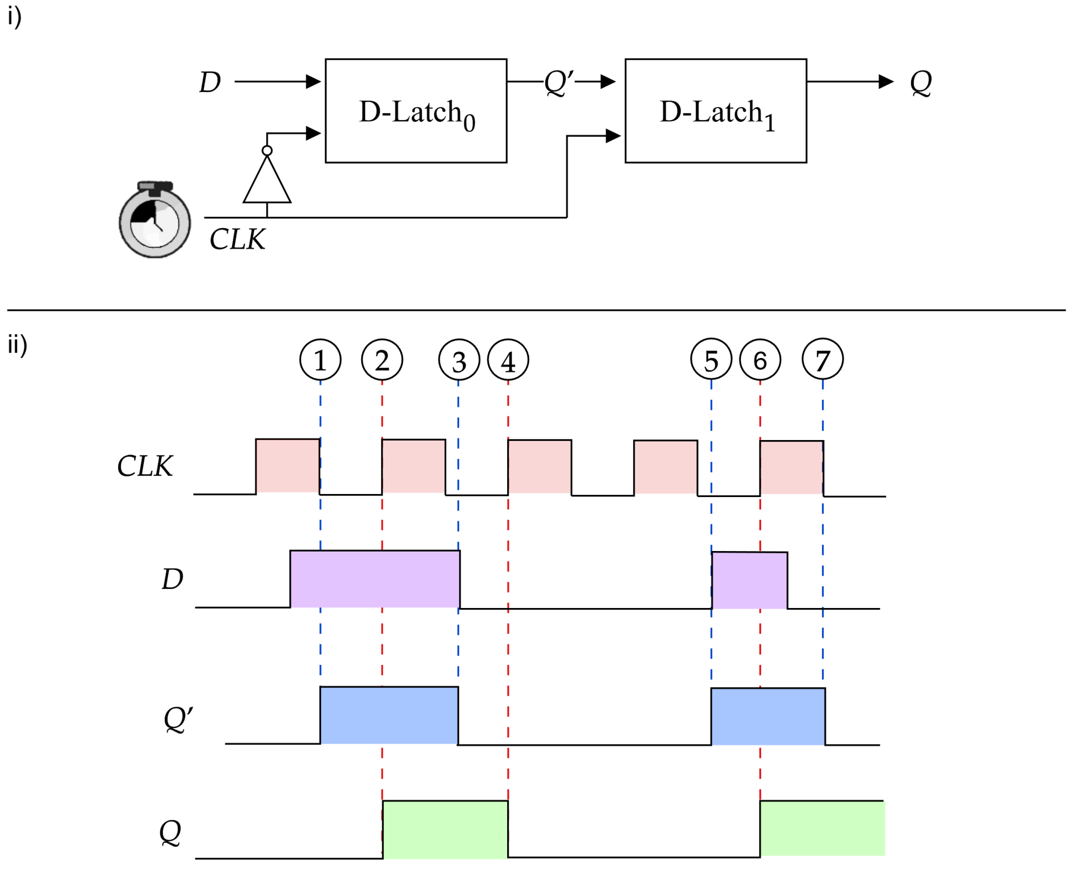


Figure 1.9: i) Shows 2 D-latches attached to one another, D-Latch$_1$ and D-Latch$_2$, which we'll call $D1$ and $D2$ respectively. $D1$ will act as our "bouncer". So when the clock is low, $D1$ prepares the data $D$ for $D2$. Once the clock goes high, $D1$ blocks new data, serving $Q'$ (the last written $D$) to $D2$, which then writes $Q'$ to its output $Q$. ii) Is a the signal timing diagram for the circuit in (i). Notice the bubbled numbers and their intersections: 1) The $CLK$ goes low, loading data $D$ into $Q'$. 2) $CLK$ goes high, $Q'$ is written to $Q$. 3) $CLK$ goes low again, loading new data $D$ into $Q'$. 4) $CLK$ goes high again, $Q'$ is written to $Q$. 5) $D$ changes while $CLK$ is low, so $Q'$ updates. 6) $CLK$ goes high, writing $Q'$ to $Q$. 7) $D$ had changed while $CLK$ was high, $Q'$ remained steady until $CLK$ went low again.

The circuit in Figure (1.9) is called a:

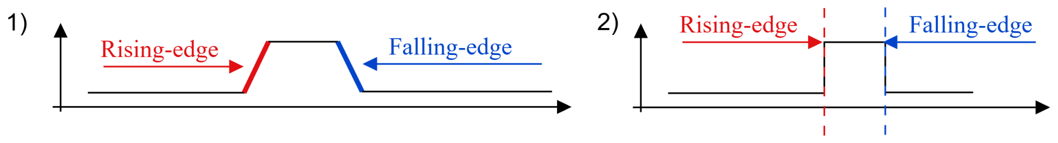
**Definition 0.5: Data Flip-Flop**

A **Data Flip-Flop** (D flip-flop) is a memory element that stores one bit of information. It consists of two D-Latches connected in series. The first D-Latch (D1) acts as a buffer, while the second D-Latch (D2) stores the final output. The D Flip-Flop captures the input data on the rising edge of the clock signal (a transition from low to high), ensuring that the output only changes once per clock cycle.

Another convention we must point out from Figure (1.9) is how signals are triggered:

**Definition 0.6: Edge-Triggered**

A signal is said to be **edge-triggered** if it responds to changes in the signal level, specifically the transition from low to high (rising edge) or high to low (falling edge):



1) Shows the real world electrical signal with visible slopes. 2) Shows an idealized discrete digital signal (which we have used most of this text).

Finally,

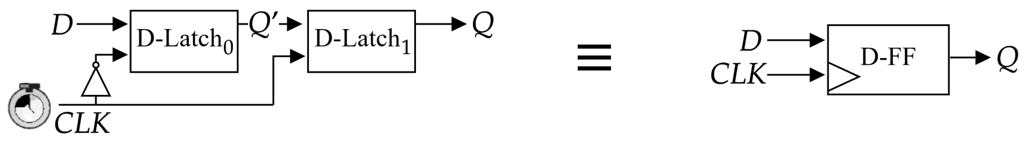


Figure 1.10: On the left is the circuit diagram for a D Flip-Flop, and on the right is the abstracted interface; Many texts use the triangle from clock to indicate edge-triggered behavior. [3]

**Note:** There are many ways to build these latches, we have shown one way with Logic Gates and MUXs. The design is not unique, so you may see different designs in other texts.

### 1.0.2 Creating The Stack: Random Access Memory (RAM)

Now that we can store one bit of information, let's see if we can string our one-bit storage devices together to create larger memory units:

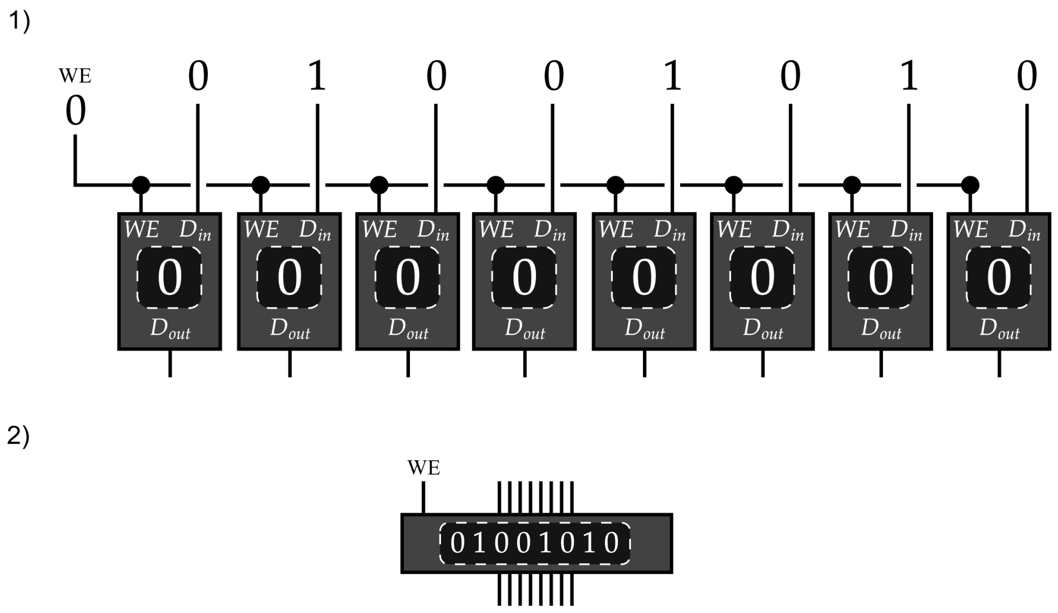


Figure 1.11: 1) Shows 8 gated latches connected in parallel, their 'Write Enable' (WE) lines are connected to a common load line. 2) Shows the abstracted diagram after the write from (1) goes through. This device is a called an 8-bit **Register**.

**Definition 0.7: Register**

A **Register** is a memory element that stores $n$-bits of information. It consists of multiple single bit memory devices of some configuration sharing a common write enable line, allowing simultaneous writing of all bits to represent larger data values.

However, in an electrical circuits, registers require constant power to maintain their state, i.e., they are **volatile** (temporary) memory devices.

In modern computers we might often see 32-bit or 64-bit registers, but if we want to store larger values it becomes quite expensive/inefficient to keep adding more bit lines.

To compact our design, we attempt to arrange our latches in a grid:

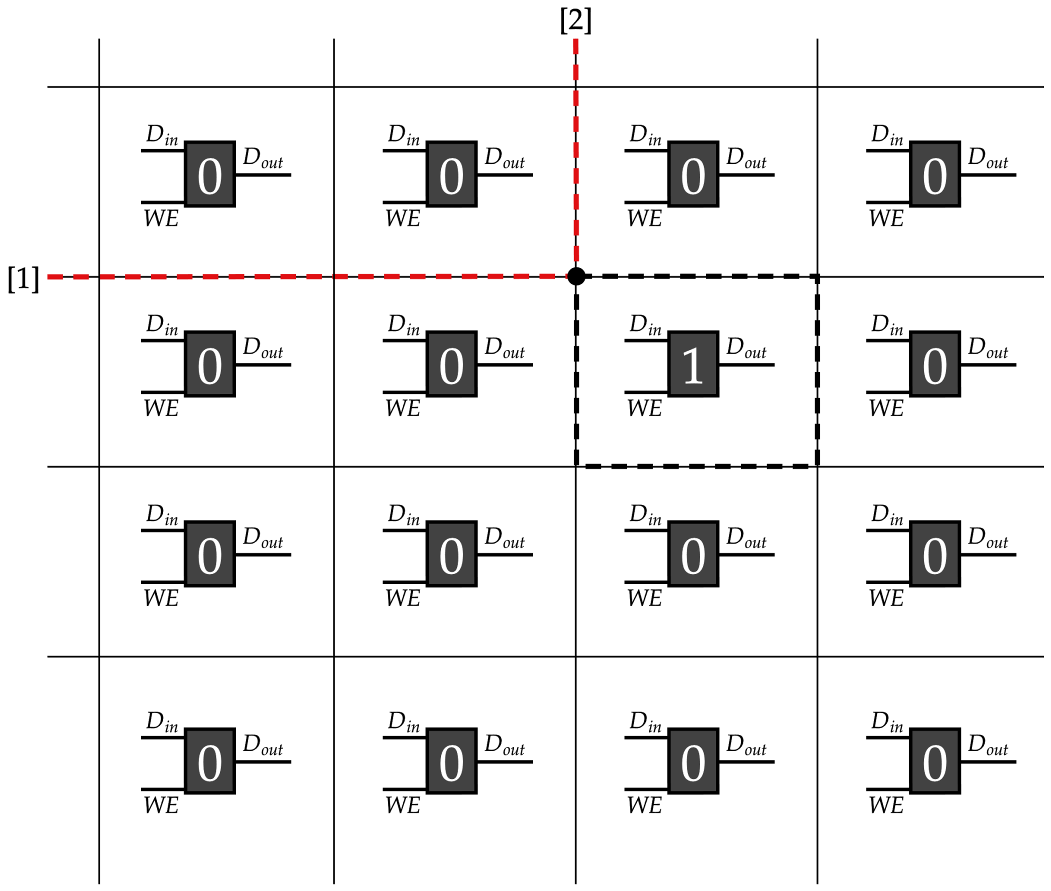


Figure 1.12: A 4x4 grid of latches (16 locations total). Here we write ‘1’ to row 1 and column 2.

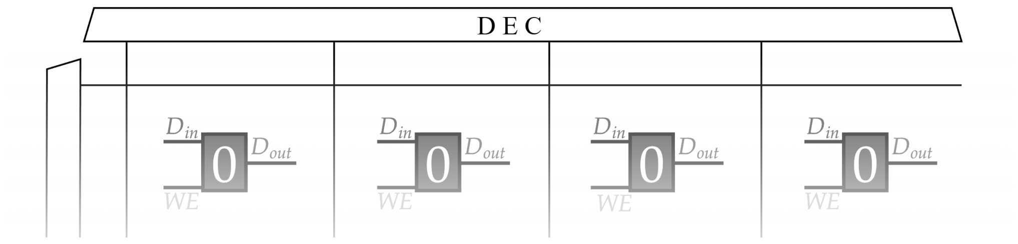


Figure 1.13: To achieve the row and column selection, we use decoders for both rows and columns.

Let's see this in action:

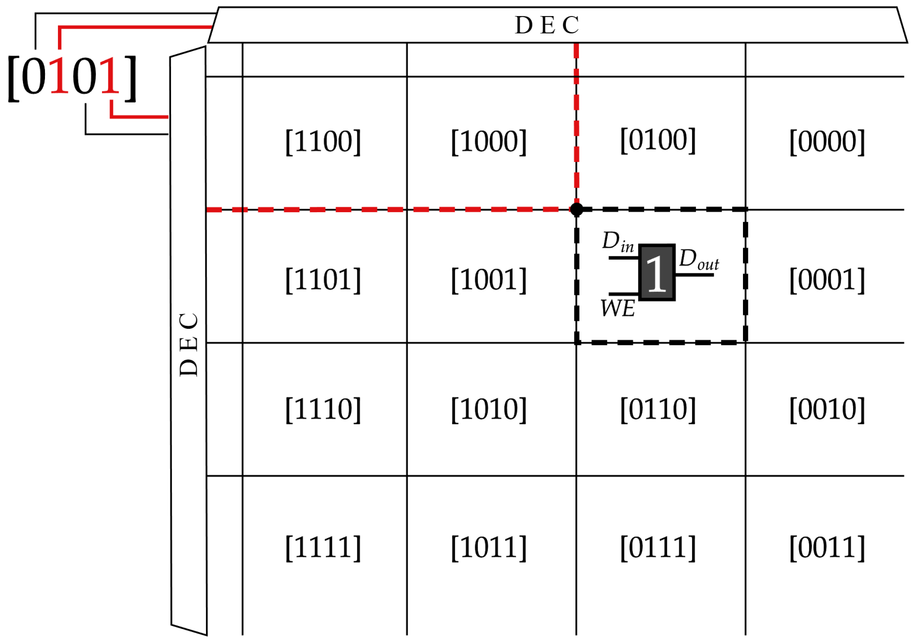


Figure 1.14: Just how houses are arranged in a grid with streets and avenues, we give each latch an **address** based on its row and column ([0001], [0010], ..., [1111]). Here we activate the same latch in Figure (1.12) by selecting [0101]. The address is split into two halves: The higher order bits (first 2) select the row, and the lower order bits (last 2) select the column.

We continue to condense our design, combining wires for Write Enable ($WE$) and Read Enable ($RE$):

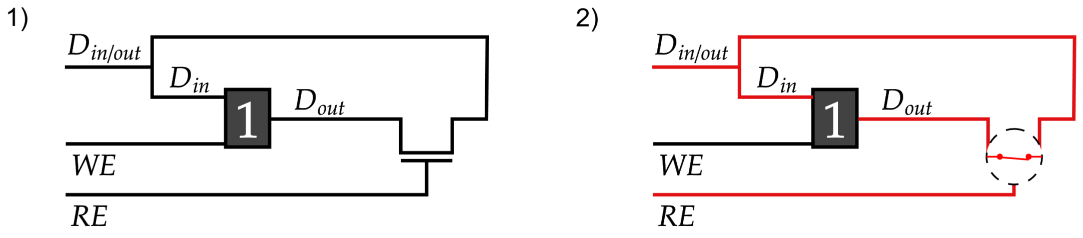


Figure 1.15: 1) Combines $D_{in}$ and $D_{out}$ lines, for a $D_{in/out}$ line. Where $D_{out}$ is only active when $RE$ is high via an NFET gate. 2) Shows that NFET gate turned on when $RE$ is high. Note $D_{in}$ is not affected as $WE$ is low (vice-versa).

We now connect all these three lines together in each cell of our grid:

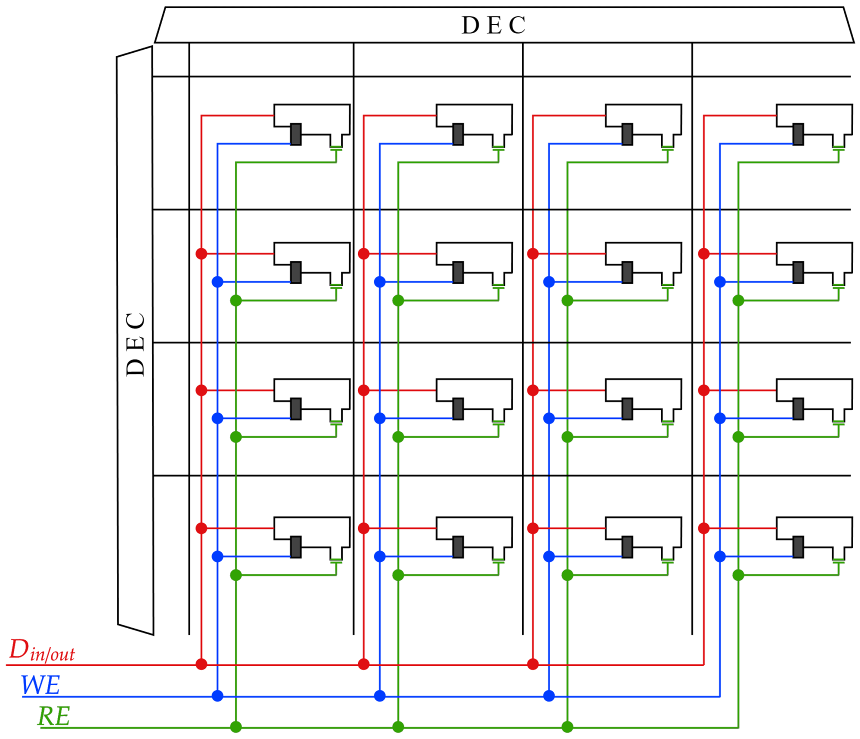


Figure 1.16: Here $D_{in/out}$ (Red), $WE$ (Blue), and $RE$ (Green) lines are combined in each cell of the grid; **However**, this design writes and reads to **all** cells.

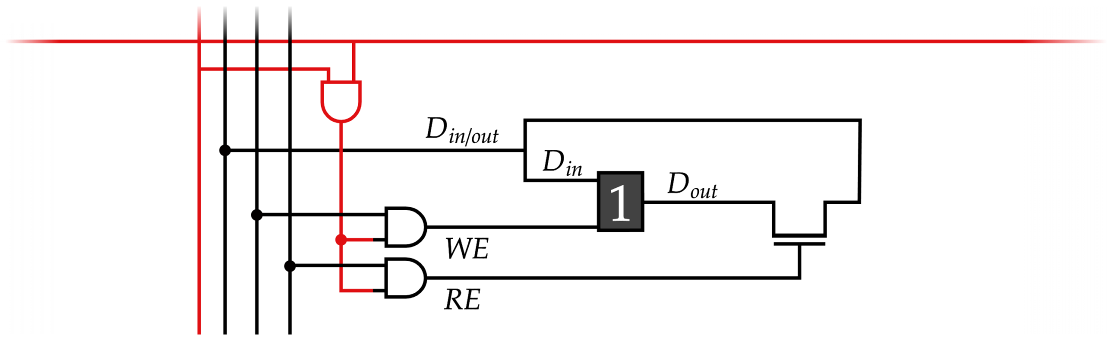


Figure 1.17: Using our addressing method in Figure (1.14), if we set an AND gate to the row and column select lines, feeding into $WE$ and $RE$, with their AND control lines, we ensure only one cell is written to/read from at a time. Here this cell is selected (red lines), however, we haven't chosen whether to read or write yet.

However, thus far we've achieved storing 1 bit of information per matrix. We can string multiple matrices together to create larger memory units, say, store 4-bits per address:

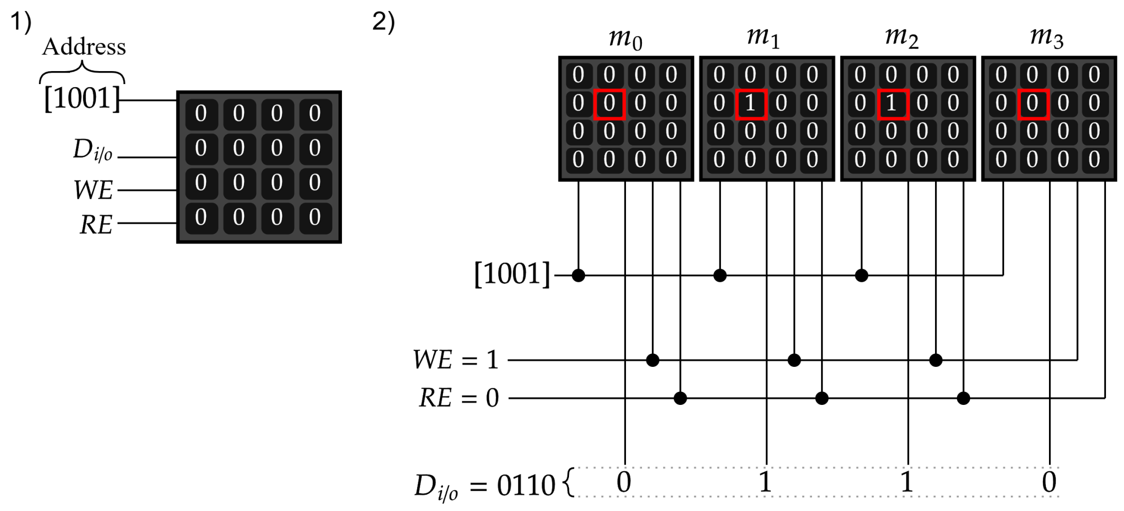


Figure 1.18: 1) Shows the abstracted 4x4 latch matrix with it's address line, $D_{i/o}$ (Data in/out), $WE$ (Write Enable), and $RE$ (Read Enable) lines. 2) Shows 4 of these matrices ($m_0$, $m_1$, $m_2$, $m_3$) connected in parallel, $WE$ and $RE$ lines are shared, while $D_{i/o}$ lines are now 4-bits wide. To write we write each bit to its respective matrix at the same address ($m_0$=0, $m_1$=1, $m_2$=1, $m_3$=0), while enabling $WE$. We chose the address [1001] to write to (boxed in red). To read this value, we disable $WE$ and enable $RE$ at the same address; Then $D_{i/o}$ lines output the stored 4-bit value.

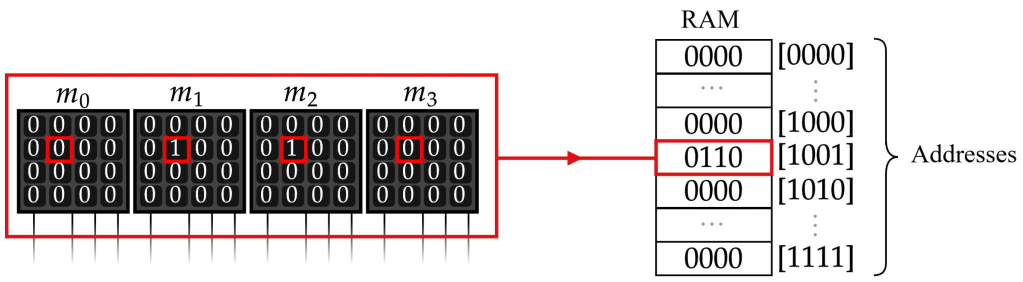


Figure 1.19: Continuing from Figure (1.18), we abstract even further. We instead **stack** each address vertically. Now we can at random choose any address to read from or write to; Hence we call it—**Random Access Memory** (RAM).

We make some final notes:

**Definition 0.8: Address-bus & Data-Bus**

An address is composed of multiple bits, the **width**, i.e., number of bits. To make note of this width, we call the address a **Address-bus** of width $n$.

Similarly, the data input/output lines are called a **Data-bus** of width $m$, which scales 1-to-1 with the number of latch matrices stacked together.

**Theorem 0.1: RAM Bottle-neck**

For every $n$-bit Address-bus, $2^n$ unique addresses can be accessed. Thus, in a 32-bit system, $2^{32} = 4,294,967,296$ (Bytes) unique addresses can be accessed. I.e,. 4GB of RAM.

In this section we created static RAM:

**Theorem 0.2: Static vs Dynamic RAM**

**Static RAM** (SRAM) uses latches to store each bit of data, making it faster and more reliable, but also more expensive and power-consuming. **Dynamic RAM** (DRAM) uses a MOSFET and a capacitor (stores charge), which are cheaper and can achieve higher densities, but the capacitor constantly needs to be charged to maintain data integrity, making it slower and less reliable.

This concludes our discussion on memory devices. In the next section (next page), we'll create the CPU to run programs.

### 1.0.3 Creating a CPU

In the following page we'll put our RAM and ALU together to create a simple CPU. We'll first show the design (i.e., architecture), then explain each component through various figures. So don't worry about understanding what all the variables mean at once.

*Continued on next page...*

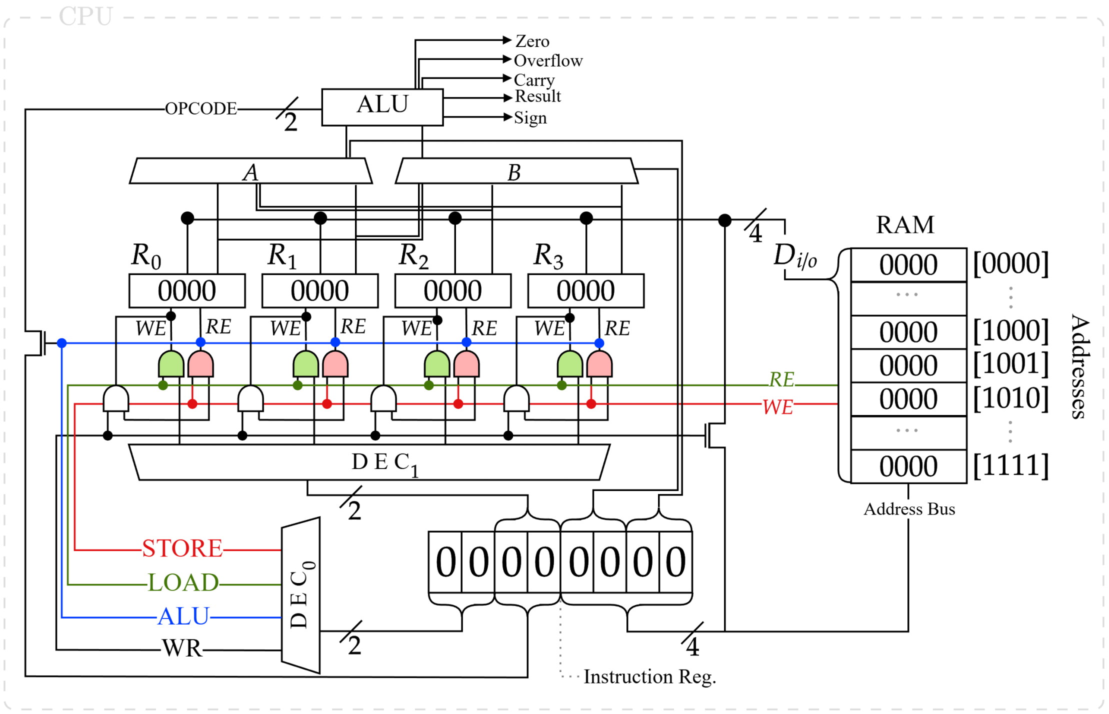


Figure 1.20: This is *a* CPU design and not unique, more to illustrate fundamental concepts.[1]

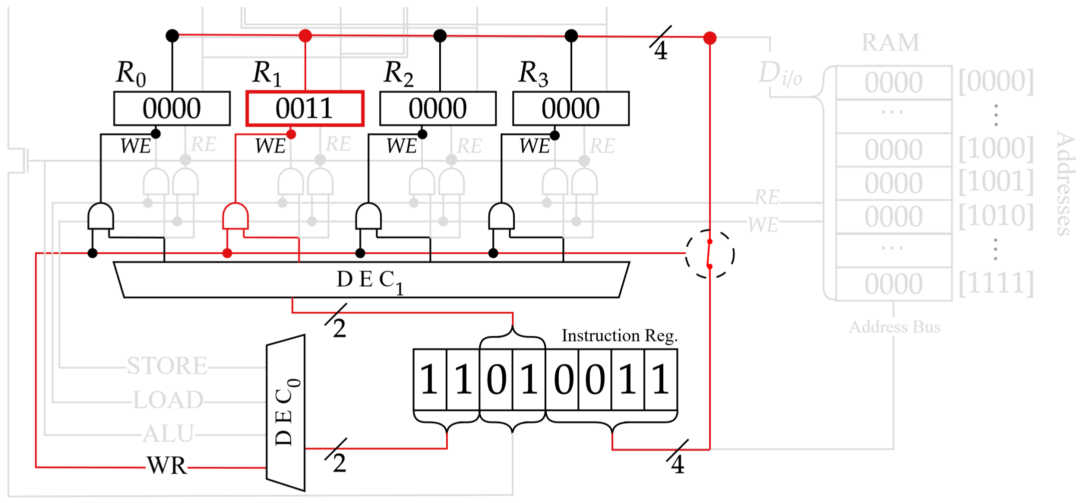


Figure 1.21: We use the first 2 bits of the instruction register (IR) to select WR (write register). The next 2 bits select which register, here $R_1$. The last 4 bits are written to that register.

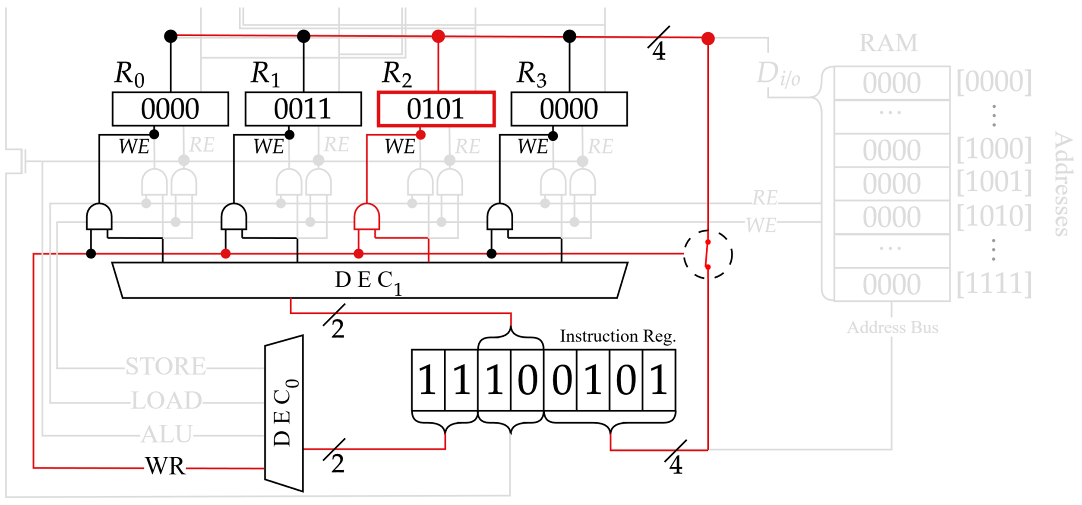


Figure 1.22: We now do the same for $R_2$, writing [0101] which is decimal 5.

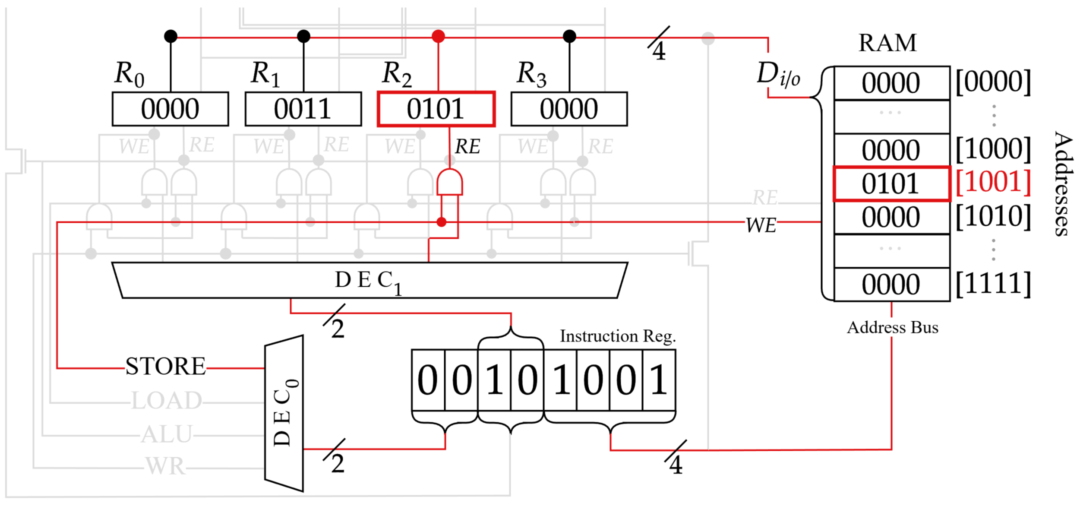


Figure 1.23: The first 2 bits now represent STORE (00), writing $R_2$ into address [1001] as RAM WE and $R_2$'s RE is high.

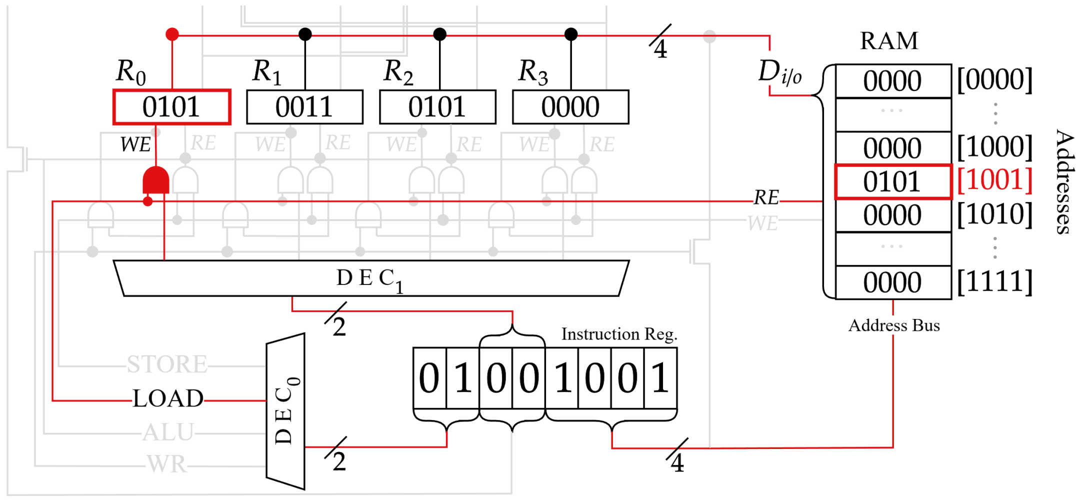


Figure 1.24: The first 2 bits now represent LOAD (01), reading from address [1001] into $R_0$ as RAM RE and $R_0$'s WE is high.

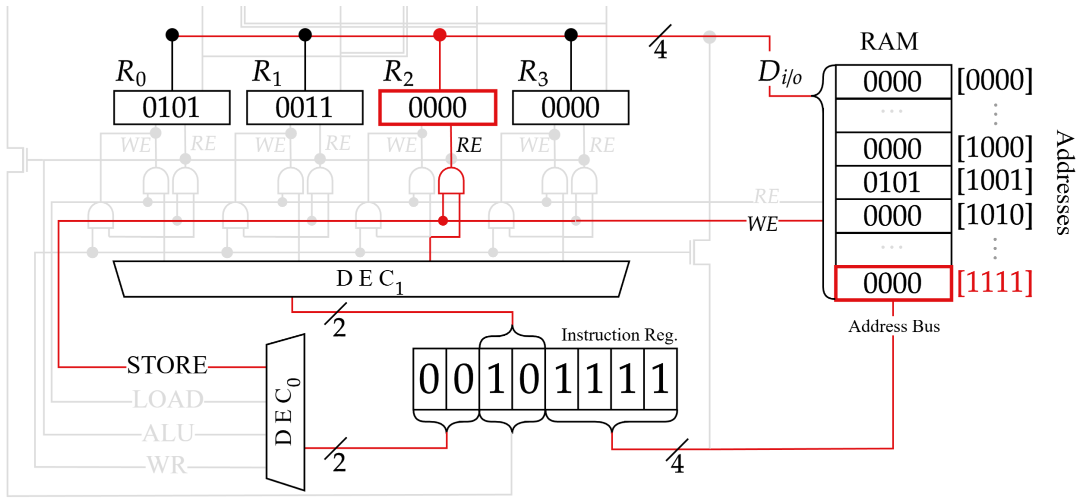


Figure 1.25: We swap back to STORE (00), writing address [1111] into $R_2$.

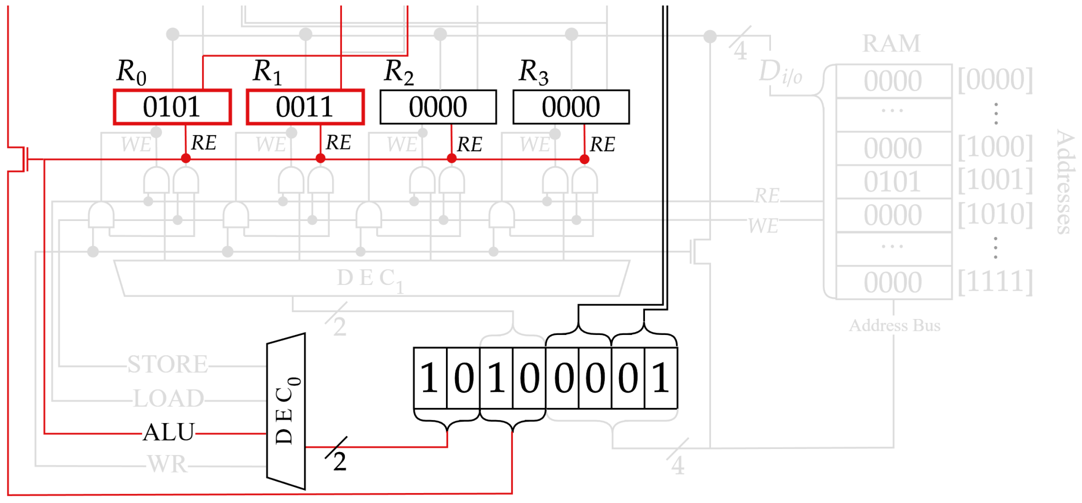


Figure 1.26: We change the first 2 bits to 10, the ALU. Now the next 2 bits select the operation, 10 for SUB; The above decoder line ($DEC_1$) isn't active as it requires the AND from a LOAD/STORE signal. The next 2 bits select $B$, here $R_0$, and the last 2 bits select $A$, here $R_1$ ($A$ and $B$ MUX components off screen).

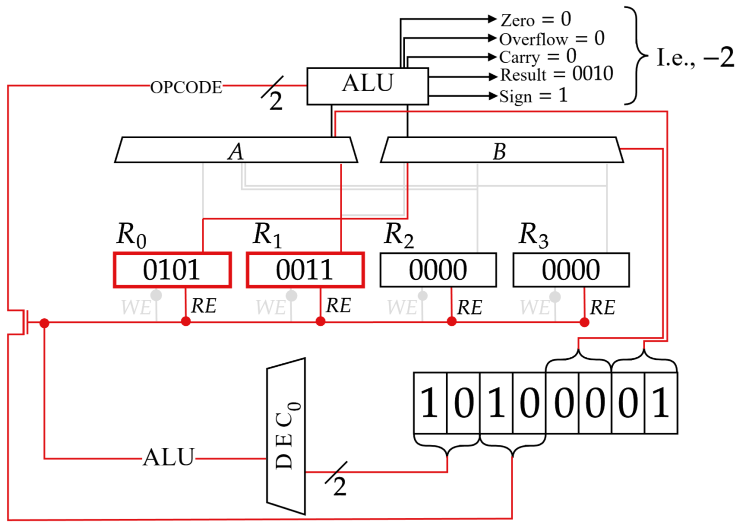


Figure 1.27: Continuing from Figure (1.26), shrinking our diagram to see the whole picture for the ALU operation; Here we perform $A - B = 0011 - 0101 = 3 - 5 = -2$.

Next steps may include finding a way to reroute the ALU output back to one of the $R_i$ general purpose registers, or map alphanumeric strings, e.g., 'SUB'$\rightarrow$ 1010, creating a programming language; However we stop here. This should give the general idea to build even more complex systems.

# Bibliography

[1] Core Dumped. Crafting a cpu to run programs.

[2] Core Dumped. How transistors remember data. YouTube, `https://youtu.be/rM9BjciBLmg?si=jbhhJZrP07Z2Pb3B`, 2024. Accessed: 2026-01-07.

[3] Chris Terman. 6.004 computation structures, 2017. Undergraduate course, Spring 2017.