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:

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

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 $0 \leq k \leq n$.

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:

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

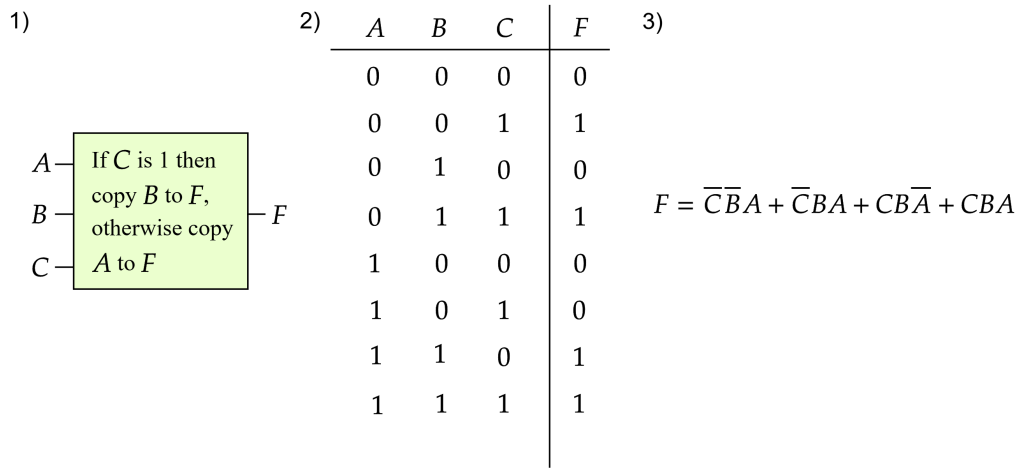
These inequalities indicate that as n grows larger, each function on the right-hand side grows faster than the ones to its left.

— 1 —

Circuits and Logic

1.0.1 Combinational Logic

Truth tables quickly grow as n inputs lead to 2^n rows in the truth table. So instead, we model functions using boolean algebra (e.g., $F = A \cdot B + \overline{C}$), and then implement them using logic gates.

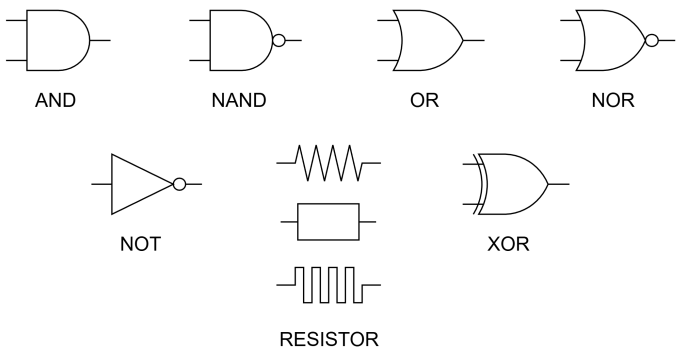


A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 1.1: 1) Shows the desired circuit function, 2) the corresponding truth table, and 3) the boolean logic. From the truth table, we can pick a row, substituting the inputs A, B, and C into the boolean expression, we expect to see the output corresponding to F. For example, row 3, $F = \overline{C}\,\overline{B}A + \overline{C}BA + CB\overline{A} + CBA$, is, $0 = (\overline{0} \cdot 1 \cdot 0) + (\overline{0} \cdot 1 \cdot 0) + (0 \cdot 1 \cdot 0) + (0 \cdot 1 \cdot 0)$.

Definition 0.1: Logic Gates

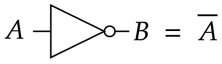
Below are the common logic gates with the addition of the resistor which limits current:



The resistor has many symbols; However, we focus purely on logic gates in this text, and how we can make more complex logical systems from these basic building blocks.

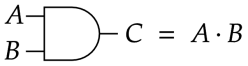
For example, lets consider just NOT, AND, and OR gates and their truth tables:

1) NOT



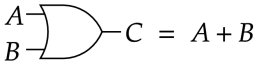
A	B
0	1
1	0

2) AND



A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

3) OR



A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

Figure 1.2: 1) NOT gate 2) AND gate 3) OR gate. Recall that inside these gates is a combination of PUNs and PDNs to achieve the desired logic, which here we have abstracted away.

$$F = \overline{C}\,\overline{B}A + \overline{C}BA + CB\overline{A} + CBA$$

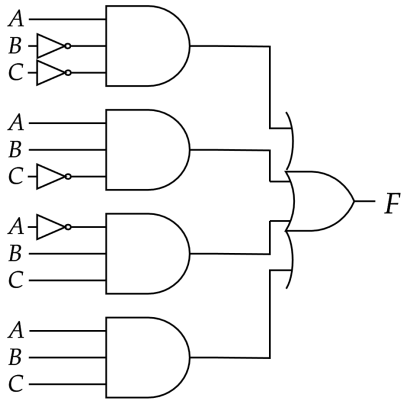
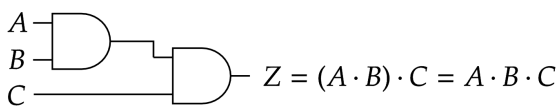


Figure 1.3: Modeling $F = \overline{C}\,\overline{B}A + \overline{C}BA + CB\overline{A} + CBA$ with logic gates. Note, we see that the ANDS and ORS have more than 2 inputs; They operate the same way as their 2-input counterparts, just with more inputs.

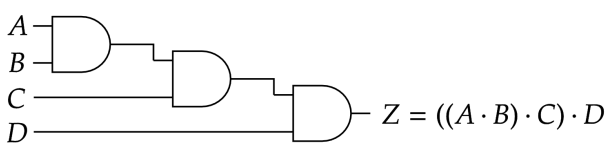
Theorem 0.1: Increasing Gate Input Size

Above in Figure 1.3, we see AND and OR gates with more than 2 inputs. There are multiple ways to implement these larger gates, observe below:

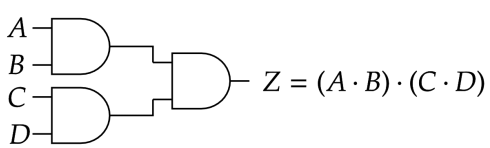
1)



2)



3)



1) is a 3 input gate, and 2) is a 4 input gate; This method of attaching the gates in sequential order is called **chaining**. 3) takes a different approach and instead uses a **tree method**.

In terms of t_{PD}, assuming all gates have the same delay, **chaining grows linearly** with the number of inputs, and the **tree method grows logarithmically** with the number of inputs.

However, if the gates were to have **different delays**, introduces **bottlenecks**; In the case of the tree method, getting an input like D introduces a longer path, while in the sequential method, D has less impact on the overall delay.

Theorem 0.2: NAND AND NOR - Gate Increase Problem & Efficiency

NAND and NOR are **not associative** like AND and OR. This means that we cannot use the chaining or tree method to increase the number of inputs for NAND and NOR gates.

Additionally, NAND and NOR gates are more efficient in CMOS logic, as CMOS is naturally inverting with PUNs and PDNs. Thus, their implementations are simpler than AND and OR gates.

To further illustrate that NAND and NOR are functionally complete, observe the below diagram:

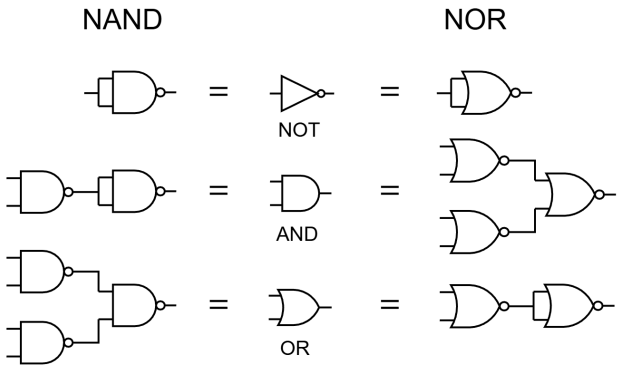
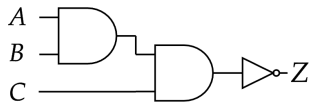


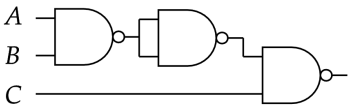
Figure 1.4: This diagram shows that NOT, AND, and OR gates can be constructed purely from NAND (left) or NOR (right) gates.

Theorem 0.3: Increasing NAND Gate Input Size with 2-input NANDs

Start with the desired function, e.g., $\overline{A \cdot B \cdot C}$, for clarity $\texttt{NAND}(A, B, C)$. We attempt to group inputs into pairs of 2: $\texttt{NAND}(A, B, C) = \texttt{NOT}(\texttt{AND}(A, B, C))$. Now we group: $\texttt{NOT}(\texttt{AND}(A, B, C)) = \texttt{NOT}(\texttt{AND}(\texttt{AND}(A, B), C))$, each AND has 2 inputs:



To only use NAND gates, we bring the NOT and AND back together: $\texttt{NAND}(\texttt{AND}(A, B), C)$. To replace A and B's AND gate, we substitute with NANDs using Figure (1.4):



Note: As we've mentioned before, chaining vs. tree methods vary differently on t_{PD}. Additionally notice that the size of our circuit can differ based on how we decide to break down our boolean expression. In the above Theorem (0.3), if we assume AND gates are slower than NAND gates in this system, the simplified NAND version is faster, and smaller, assuming AND gates are composed of multiple NANDs.

1.0.2 Karnaugh Maps

To help us find simpler boolean expressions, we can change how we represent our truth table data:

Definition 0.2: Karnaugh Maps

A Karnaugh map (K-map) represents truth table data with a format called **Gray Code**. In Gray Code, columns and rows differ by only one bit of information:

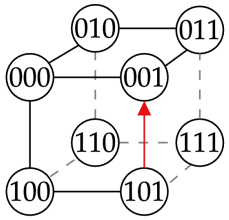
1)

C	B	A	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

2)

$C \setminus AB$	00	01	11	10
0	0	0	1	1
1	0	1	1	0

3)



1) Shows the truth table for inputs A, B, and C with output F. 2) The corresponding K-map (left-most column, C, top row, AB respectively). **Note:** in 2) columns and rows are cyclical, meaning the leftmost and rightmost columns are adjacent, as are the top and bottom rows. Hence with 3), we can represent our truth table as a 3D cube (Formatted ABC). The red arrow in both 2) and 3) indicate adjacency. We extend this to 4 inputs:

$CD \setminus AB$	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

For inputs 5 and 6, we need an additional dimension, building a $4 \times 4 \times 4$ k-map. Anything beyond 6 inputs becomes difficult to visualize, so algorithms become more integral in the simplification process.

Now to actually use this k-map to simplify we introduce the following method:

Definition 0.3: Simplifying with Implicants & Prime Implicants

An **implicant** is a set of adjacent 1s in a k-map, whose width and length are a power of 2 (i.e., 1, 2, 4, 8, ...). An implicant that is not a proper subset of another implicant (i.e., not contained within another implicant) is called a **prime implicant**.

Name of the game: Find the smallest set of prime implicants that cover all 1s in the k-map. Then translate each member of such set into a product term (e.g., top row $AB := 01 \to \overline{A}B$). Then sum all found product terms to yield the simplified boolean expression:

1)

$C \setminus AB$	00	01	11	10
0	0	0	1	1
1	1	0	0	0

2)

$C \setminus AB$	00	01	11	10
0	1	0	0	1
1	1	1	0	1

1) The red singleton implicant translates to $\overline{A}\overline{B}C$. The blue pair translates to $A\overline{C}$, yielding the simplified expression $F = \overline{A}\overline{B}C + A\overline{C}$. **Note:** In the second term we dropped B, as despite its value switching, it did not affect the output.

2) Red group: $\overline{A}C$, blue group: $\overline{B}$, yielding $F = \overline{A}C + \overline{B}$, which demonstrates that finding **larger groups** lead to smaller terms, and thus, a simpler expression.

Theorem 0.4: Prime Implicants & Uniqueness

Though we look for the smallest set of prime implicants, there may be multiple such sets (i.e., same size, different members) that cover all 1s in the k-map.

Tip: Think of our 1 sets as capturing the truth table in its truth states (i.e,. when A is this, and B is this, the output is true!). We combine all such states to reflect the overall behavior (i.e., if this state or this state is true, the output is true!).

Recall from discrete math, this is called the disjunctive normal form (DNF) of a boolean function. We can do the same with 0s, called the conjunctive normal form (CNF) of a boolean function, where we specify which states we don't want; Though we won't explore CNF here.

Let us explore an example with 4 inputs:

1)

$CD \setminus AB$	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

2)

$CD \setminus AB$	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

Figure 1.5: 1) and 2) A 4-input k-map with 4 prime implicants highlighted. 1) Yields $F = \overline{B}C + D + B\overline{C} + \overline{C}A$. 2) Yields $F = \overline{B}C + D + B\overline{C} + A\overline{B}$.

Theorem 0.5: Glitches from Implicant Hopping

Moving between two prime implicants (product terms) which do not overlap causes glitches; Because each is solely driving the output. So switching one off one and the other on, briefly exhibits undefined behavior, due to the t_{PD} (uncertainty period).

$C \setminus AB$	00	01	11	10
0	0	0	1	1
1	0	1	1	0

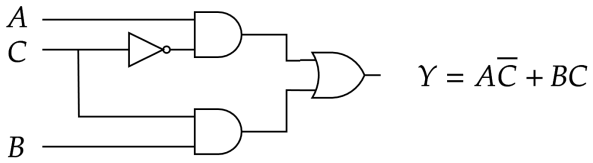


Figure 1.6: To the left shows the k-map of a function consisting of two prime implicants (circled red). The right shows the logic gate implementation and its resulting sum of products expression.

We deviate from our game of smallest sets, adding redundancy:

Theorem 0.6: Resolving Implicant Glitches with Redundancy

To resolve glitches caused by gaps between prime implicants, we can add another prime implicant that fills that gap to help hold the output steady during transitions.

To further illustrate the glitch we visualize the voltage over time:

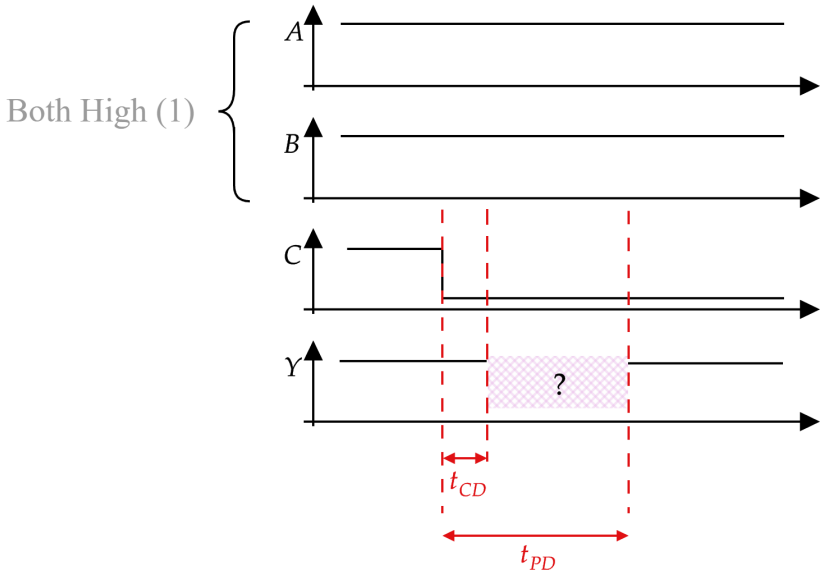
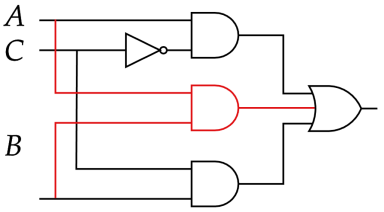


Figure 1.7: A and B inputs remain steady high (1), while C transitions from high (1) to low (0). Based on Figure 1.6, the output Y should remain high (1); However, due to both products relying on C, the output is briefly undefined for t_{PD}.

$C \setminus AB$	00	01	11	10
0	0	0	1	1
1	0	1	1	0



$$Y = A\overline{C} + BC + AB$$

Figure 1.8: By adding the redundant prime implicant (in red), we ensure steady output during transitions. In particular, the output isn't solely reliant on C during its transition, thus avoiding the glitch.

1.0.3 Branching Logic: Multiplexers & Decoders

Below observe the table we've used in our k-map Definition (0.2):

1)

S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

2)

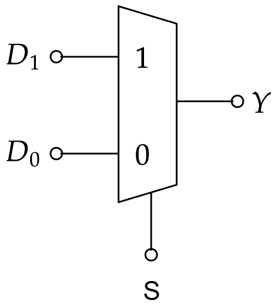


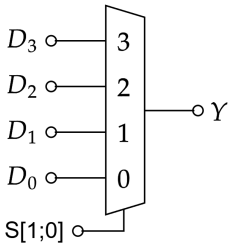
Figure 1.9: 1) A three input truth table for when S is 0 D_0 is the output, and when S is 1, D_1 is the output. 2) A diagram simplification of the truth table: S has two controls 1 and 0, which chooses between data inputs D_1 and D_0 respectively.

This new combination device we've created has a name:

Definition 0.4: Multiplexer

A **multiplexer** (MUX) is a combinational logic device that selects one of many data inputs based on control inputs, outputting the selected data input. In Figure (1.9), S the control input and D_0, D_1 the data inputs.

More generally, a MUX with n control inputs can select from 2^n data inputs. Vice-versa a MUX with m data inputs, has $\log_2(m)$ control inputs.



E.g., a 4-to-1 MUX: 2 control inputs S_1 and S_0 selecting from 4 data inputs.

In the above Definition (0.4), we simply showed a 4-1 MUX; However, they are much more complicated devices:

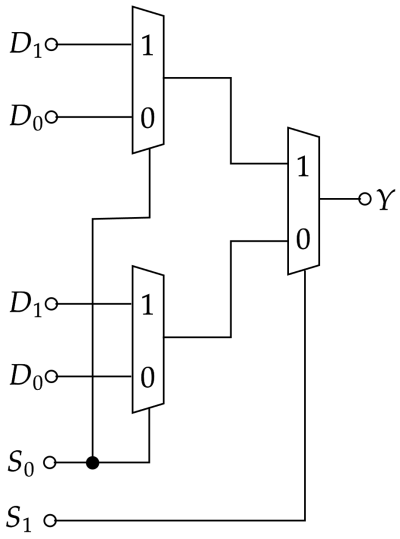


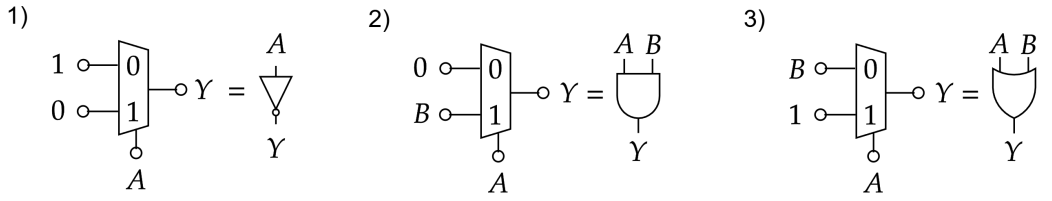
Figure 1.10: This shows that under the hood is composed of a tree of 2-to-1 MUXes.

Theorem 0.7: Building Larger MUXes from 2-to-1 MUXes

Any larger MUX can be built from a tree of 2-to-1 MUXes. In particular, an n-to-1 MUX requires $n-1$ of 2-to-1 MUXes to build. [3]

Theorem 0.8: MUX as Boolean Function Implementers

MUXes are functionally complete, i.e., can implement any boolean function.



E.g., 1) shows a NOT, 2) an AND, and 3) an OR gate implemented with a 2-to-1 MUX.

To be clear, Logic gates and MUXes may both be used to implement boolean functions, it's a matter of which use-case is more appropriate:

Theorem 0.9: MUX vs. Logic Gates

Logic gates help with basic/dedicated functions, such as needing a AND, OR, NOT gate, or a simple routine (algorithm). MUXes help with selecting between multiple data inputs, i.e., large if-else/branching logical statements.

Depending on the function implementation's t_{PD} and size requirements, one may be more appropriate than the other based on the schematics at hand.

Note: In real world applications, MUXes carry significant overhead compared to logic gates. They are complex devices for simple selection use cases, meaning they are not as flexible as logic gates. This is an important consideration when it comes to the design costs/efficiency of a circuit.

We discussed a device (MUX) that selects the row of a truth table where a specific piece of data lives. This is good for one output; But to allow for n outputs, we'll add another set of n lines (columns) to each row. Each column leads to an output:

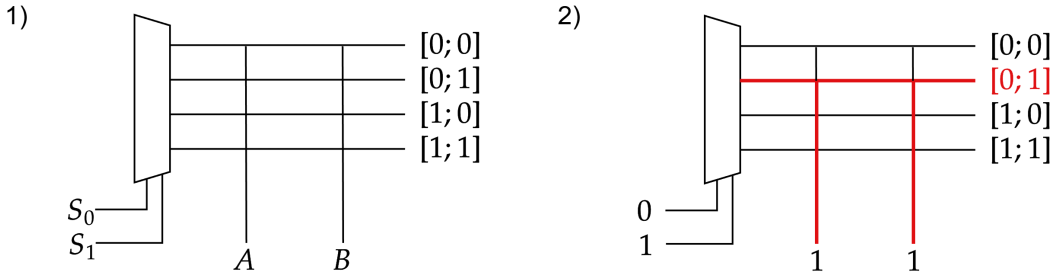


Figure 1.11: 1) Shows an altered 4-to-1 MUX from Definition (0.4) with the addition of 2 columns (outputs) A and B per row. 2) Shows $S_0 = 0$ and $S_1 = 1$, thus selecting row 2, outputting $A = 1$ and $B = 1$.

This new device also has a name:

Definition 0.5: Decoder

A **decoder** is a combinational logic device that takes n control inputs and decodes them to 2^n possible unique outputs (explained on the next page). E.g., Figure (1.11) shows a 2-to-4 decoder, 2 inputs leading to 4 outputs.

This differs from the MUX naming convention where n select inputs lead to 1 output (2^n-to-1 MUX) vs. (n-to-2^n Decoder).

In Figure (1.11), we see that we'll always get 1 from each input, which isn't very interesting;

Theorem 0.10: Decoder PDN-line Manipulation

To achieve 2^n output on a decoder, PDNs are attached to output-lines driving high (1) lines to low (0). Each row mix-and-matches PDNs to each intersection just how a truth table would with 0s and 1s for unique combinations.
Note: PDNs only effect their intersection, not the entire column.

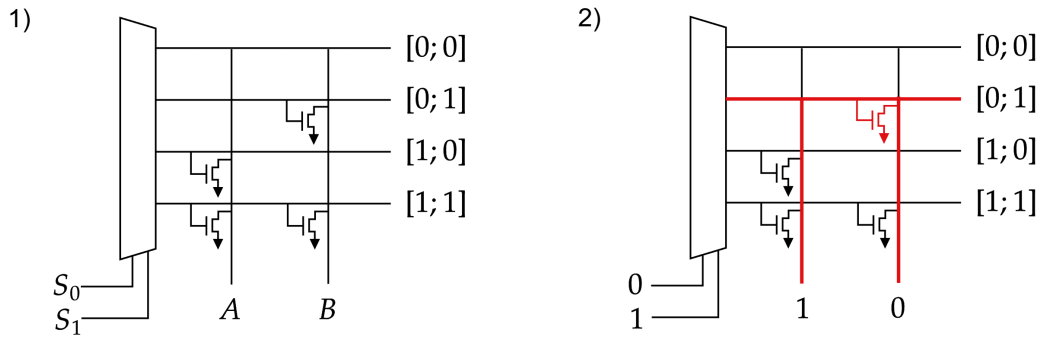


Figure 1.12: 1) Shows the 2-to-4 decoder from Figure (1.11) now with PDNs: $[0;0]$ has 0 PDNs, $[0;1]$ and $[1;0]$ have 1 PDN each, and $[1;1]$ has 2 PDNs. 2) Shows $S_0 = 0$ and $S_1 = 1$, thus selecting row 2, outputting $A = 1$ and $B = 0$ because of it's PDN. **Notice:** That A's line isn't effected by the other PDNs below, thus it's output remains high.

1.0.4 Creating the ALU: Addition & Subtraction Circuits

We now know enough to create a combinational logic circuit that performs addition. Recall the information provided in Theorem (**??**) about binary addition. We now build intuition:

Theorem 0.11: 32-bit Full Adder Strategy (Logic Gates)

We approach the problem truth-table first. To add two binary numbers A and B, we need at least 2 inputs; However, we need to account for carry-in bits C_{in} from previous additions (3 inputs). We also need to pick between sum bit S and carry-out bit C_{out} (2 outputs).

Logically, we only have a carry if at least 2 of the 3 inputs are 1; Hence, $C_{out} = (A \wedge B) \vee (A \wedge C_{in}) \vee (B \wedge C_{in})$. For the sum bit, at most 1 input can be 1; Hence, $S = A \oplus B \oplus C_{in}$ (XOR).

This only adds 1 bit; To extend to 32-bits, we chain 32 of these full adders (FA) together: Break A and B into 32 individual bits, A_0 to A_{31} and B_0 to B_{31}. The carry-out of each FA becomes the carry-in of the next FA. We may use the last carry-out C_{out} as an overflow bit.

Let's implement the above strategy:

A	B	C_i	$C_o = (A \wedge B) \vee (A \wedge C_i) \vee (B \wedge C_i)$	$S = A \oplus B \oplus C$
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

Figure 1.13: Full Adder (FA) truth table, A, B, and C_{in}, with outputs C_{out} and S.

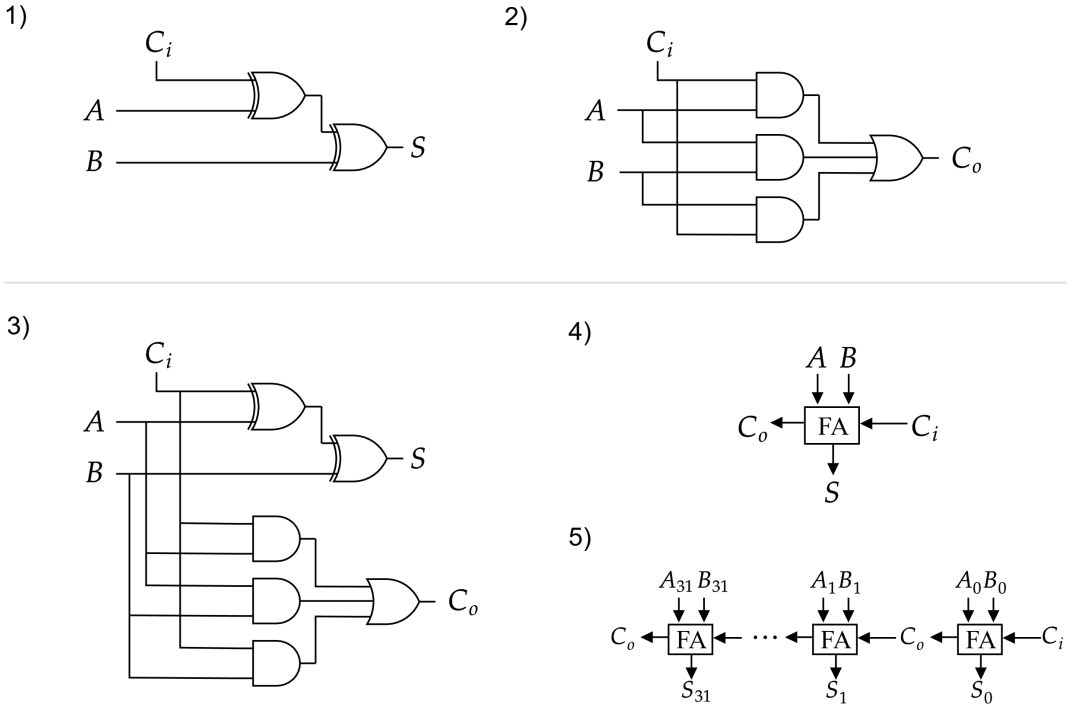


Figure 1.14: 1) S logic gates, 2) C_{out} logic gates, 3) combining S and C_{out} logic gates to form a full adder. 4) diagram representation of FA, 5) chaining 32 FAs to create a 32-bit adder. [2]

Again, remember, we can always make a MUX/Decoder version of whatever logic gate version we create:

1)

A	B	C_i	C_o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

2)

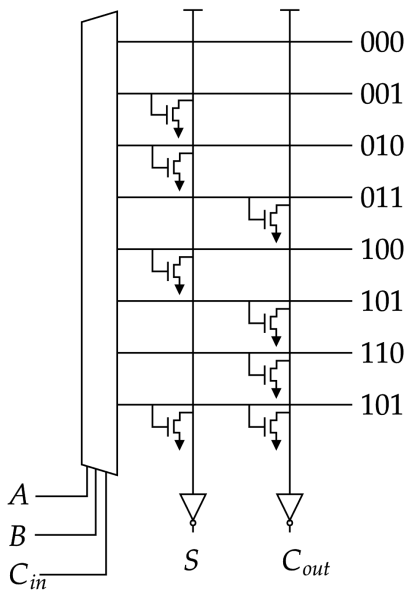


Figure 1.15: 1) Shows our desired logic (truth-table) 2) Shows a 3-to-8 Decoder, with select lines A, B, and C_{in}, driving 8 outputs, leading to outputs S and C_{out} via OR gates. **Note:** The inverters just before the outputs; This flips our PDNs low outputs to high for our desired logic.

We can reuse the same technique for subtraction with a small twist:

Theorem 0.12: 32-bit Subtraction Strategy via 2's Complement

Recall, $3 - 2 = 3 + (-2)$; Thus, altering our 32-bit adder and negating, say B, allows us to perform subtraction. We negate using Definition (**??**), two's complement: Invert every bit in the number, then add 1. I.e.,

$$A - B = A + (-B) = A + (\overline{B} + 1)$$

To get the $+1$, we set the initial carry-in C_{in} to 1.

Instead of building a separate subtraction circuit, we use our initial C_{in} as a control line choosing between addition (0) and subtraction (1); Connect the control line to all outgoing B lines replacing the ORs with XOR gates to invert B when subtracting.

The following is our addition/subtraction circuit:

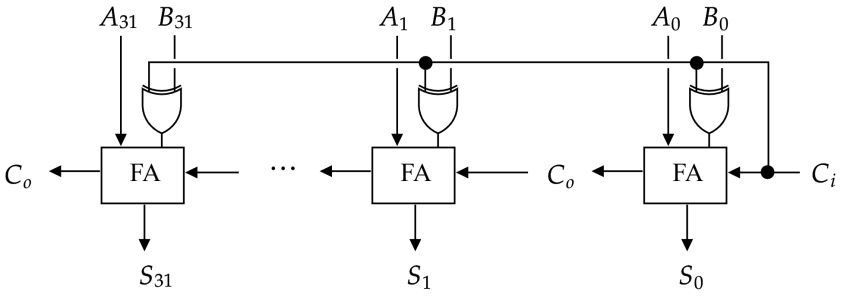


Figure 1.16: Here shows a similar FA string from Figure (1.14), with the addition of XOR gates after each B input supplemented by the initial C_{in} line (control line). [5]

Theorem 0.13: n-bit AND and OR Operations

To create n-bit AND and OR operations, we chain n individual AND and OR gates respectively to each bit from A and B. E.g., for AND, gate 0 takes in A_0 and B_0 and so on.

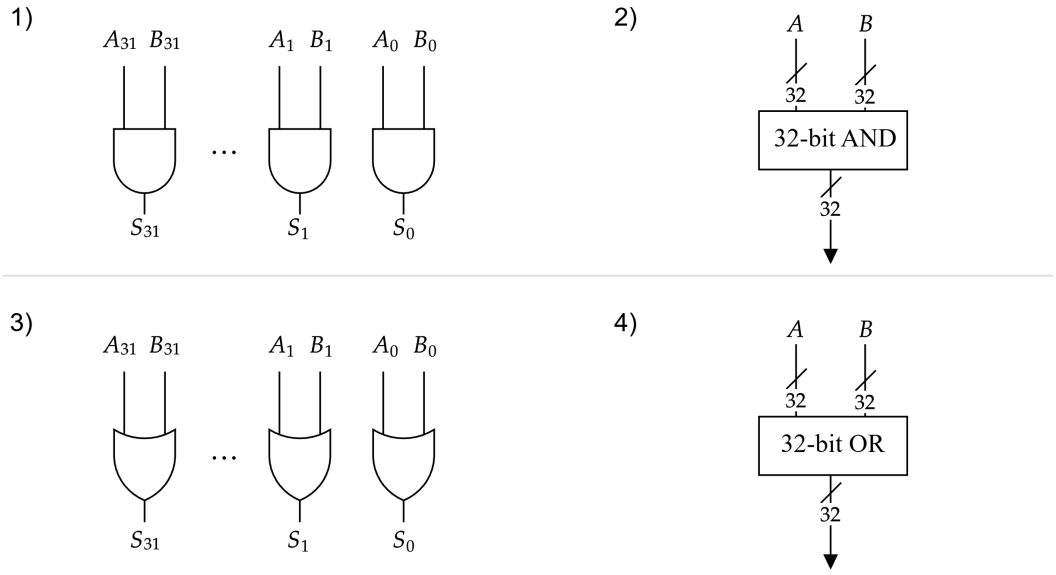


Figure 1.17: (1) and (3) both demonstrate the AND and OR chain discussed in Theorem (0.13) respectively. 2) shows an abstraction or **'interface'** of the AND circuit; The slash ("/") followed by the 32 indicates that the line is actually 32 individual lines (bits). [5]

Combining all our techniques so far, we can create a simple ALU:

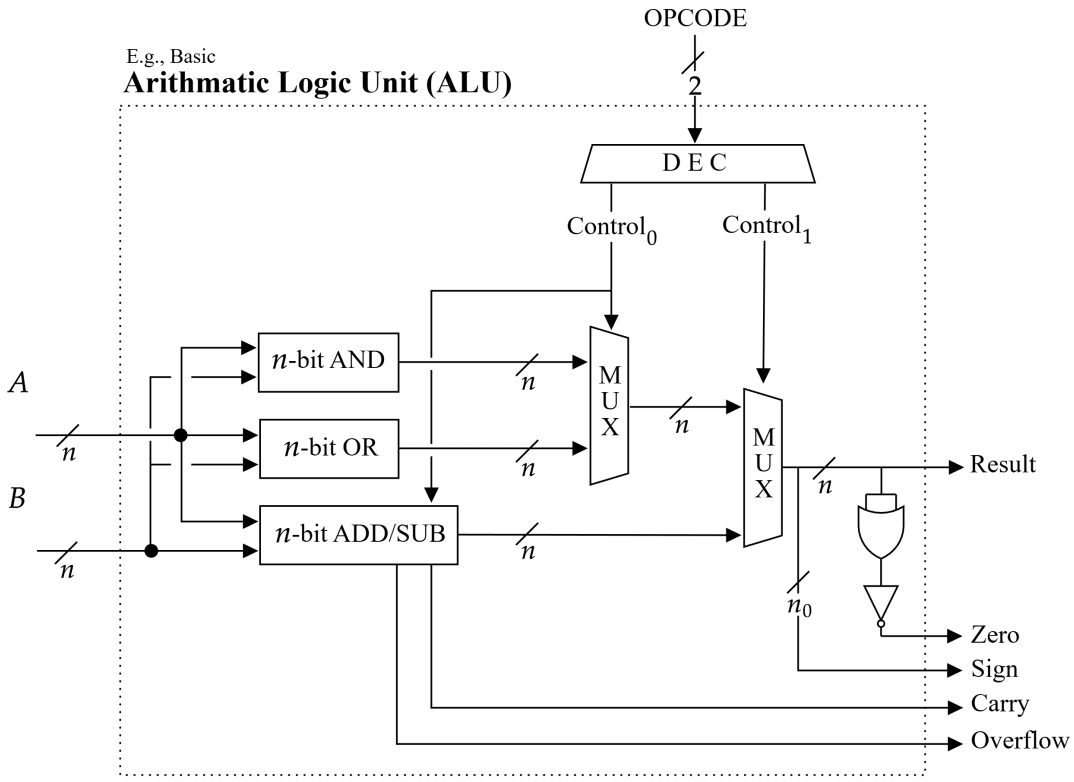


Figure 1.18: Here shows a basic ALU that can perform addition, subtraction, AND, and OR operations. Starting from the left, we have 2 inputs A and B both of variable bit-length n. At the top we have our 2-bit OPCODE. The OPCODE selects which operation to perform via a 2-4 decoder (DEC), leading to 2 control lines (Control_0 and Control_1). Control_0 (C_0), if $C_0 = 0$ then (AND and ADD) are selected, else (OR and SUB) are selected Control_1 (C_1), if $C_1 = 0$ then the boolean operations (AND or OR) is the Result; Otherwise (ADD or SUB) is the Result. The ADD and SUB operations share the same 32-bit adder/subtractor circuit from Figure (1.16) E.g., OPCODE = 10, $C_0 = 1$ and $C_1 = 0$, performs subtraction. We have additional outputs, **'Zero'**, which determines if the output is zero; 1 (true) and 0 (false). Then **'Sign'**, which relies on two's complement formatting to acquire the integer sign n_0 the left-most bit. Finally our 'Carry' and 'Overflow' from our modified FA. [5]

1.0.5 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** [1]:

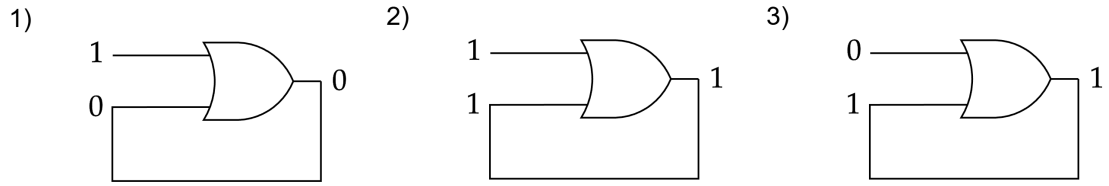


Figure 1.19: A simple feedback loop using OR gates. 1) Initially both inputs are zero, then the free input is set to 1. 2) The output becomes 1, switching the feedback input to 1. 3) Now even if the free input is set back to 0, the output remains 1, since one of the OR inputs is still 1.

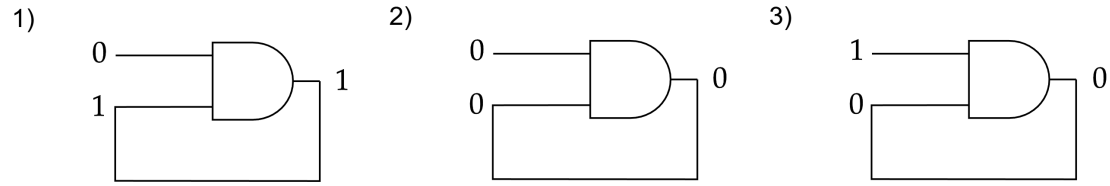


Figure 1.20: A simple feedback loop using AND gates. 1) Initially both inputs are one, then the free input is set to 0. 2) The output becomes 0, switching the feedback input to 0. 3) Now even if the free input is set back to 1, the output remains 0, since one of the AND inputs is still 0.

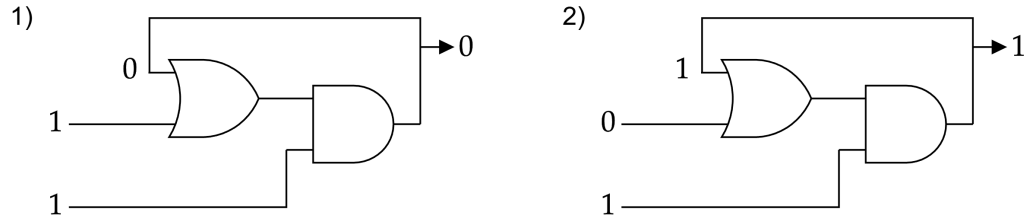


Figure 1.21: Combining the OR and AND feedback loops from Figures (1.19) and (1.20), we get the above. 1) Initially both inputs are 0, output 0, then inputs both are set to 1, resulting in a a 1 output. 2) The output is now 1, and the first input is set to 0, but the output remains 1, as the second input is still 1, driving the AND gate.

In the above Figure (1.21), 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:

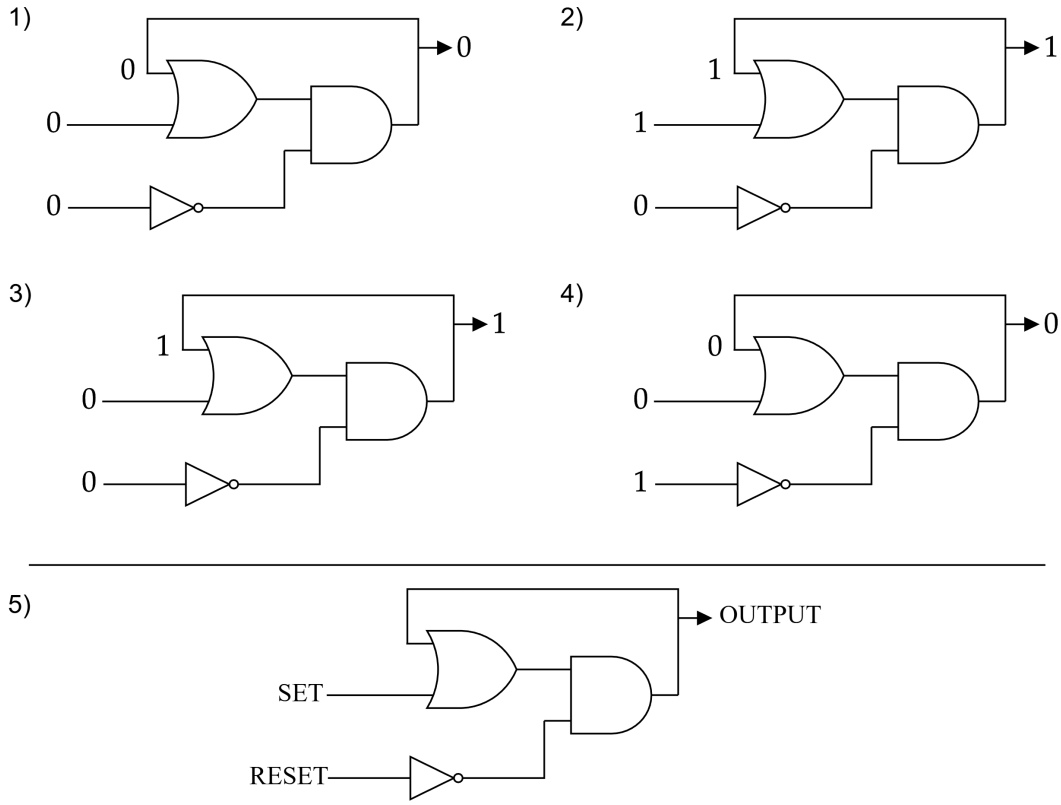


Figure 1.22: 1) Initially all inputs are 0, output 0. 2) First input is set to 1, output becomes 1. 3) First input is set back to 0, but output remains 1. 4) Second input is set to 1, output becomes 0. 5) We characterize the first input as the 'SET' and the second input as the 'RESET'.

This circuit is called an:

Definition 0.6: 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.

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

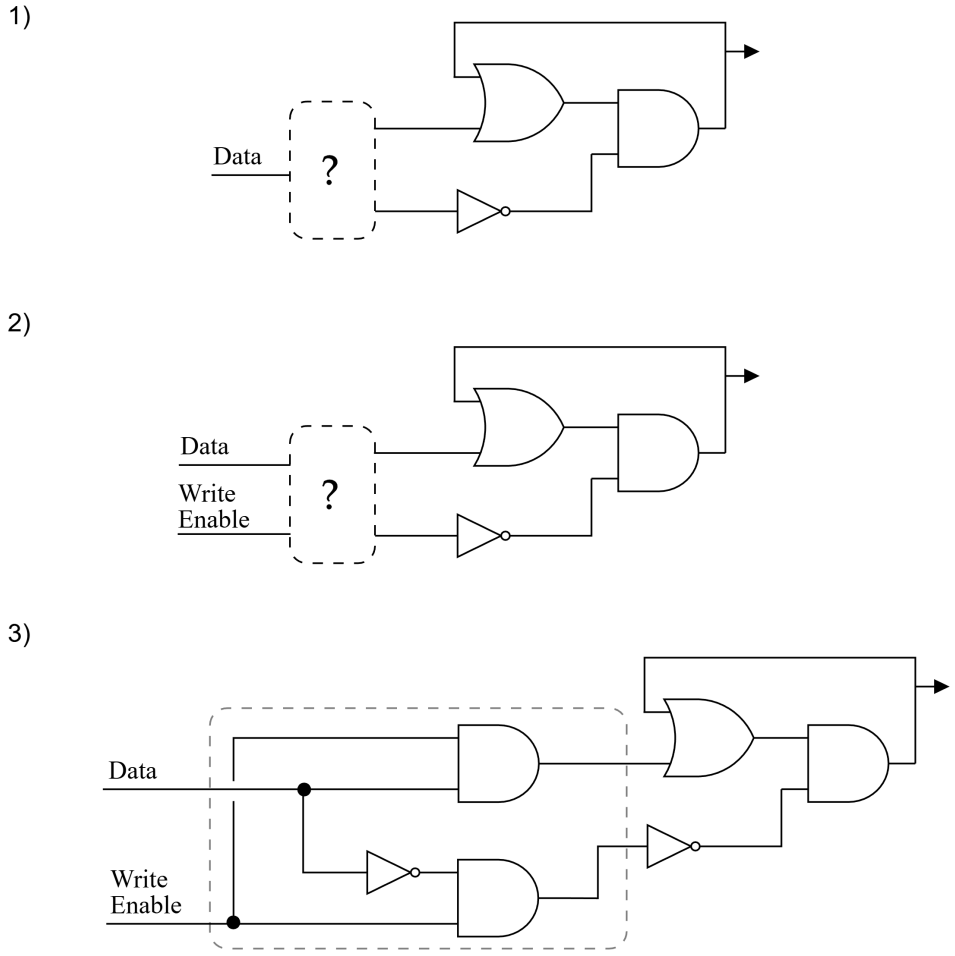


Figure 1.23: 1) We attempt to combine the SET and RESET lines to drive our 'Data'; However, this doesn't store information, as the output immediately follows the input. 2) We introduce a 'Write Enable' line. Now the data input only is written when we drive Write Enable high. 3) Is the combinational logic we were abstracting—Test it out!

Definition 0.7: 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'.

Let's do the same thing but with MUXs:

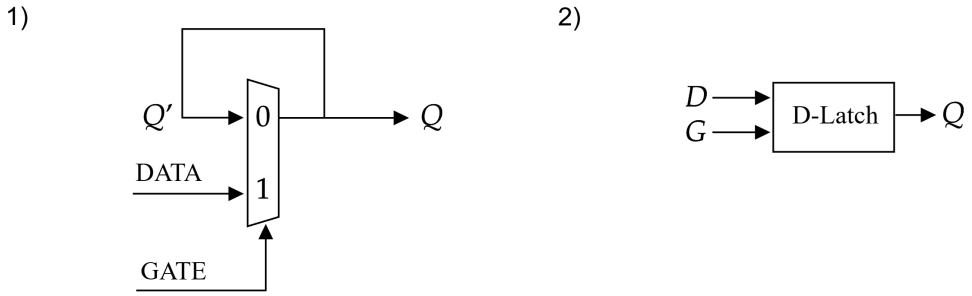


Figure 1.24: 1) Shows our MUX configuration. 'Gate' selects which line to output: 'Data' or Q'. Data contains incoming data, while Q' is the feedback line (stores the last output). 2) Is an abstracted interface for what we call a **Data Latch** (D-Latch). [4]

Definition 0.8: 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.

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

Definition 0.9: 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). [4]

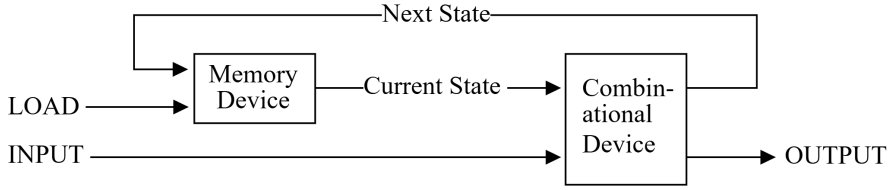


Figure 1.25: A High-level idea of a sequential circuit; 'LOAD' possibly controlled by a clock, 'INPUT' perhaps a new button press.

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):

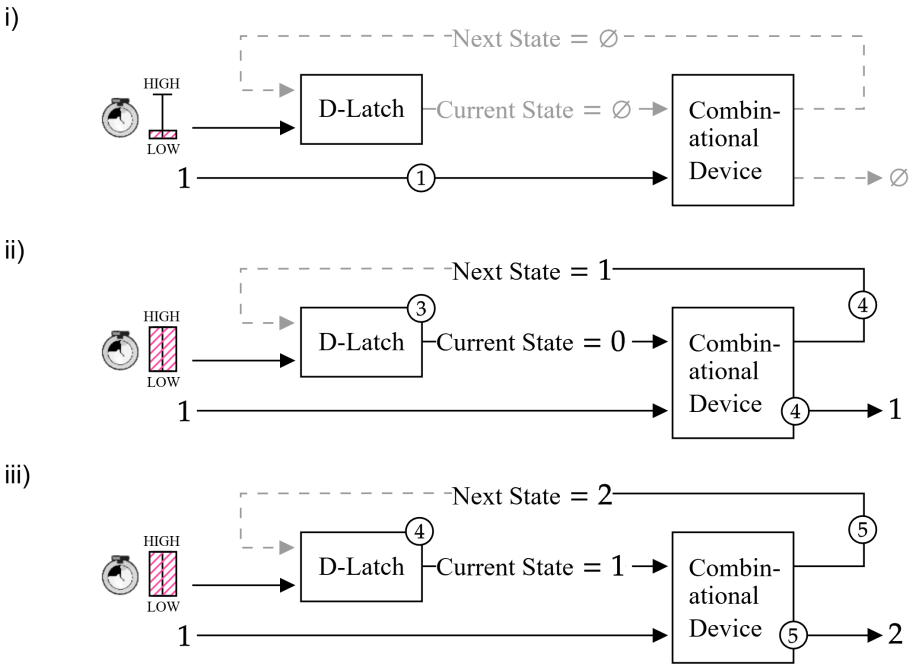


Figure 1.26: A D-latch (DL) controlled integer-sum circuit. The ∅ symbolizes no input/output (off), and the bubbled numbers indicate order of events. i) Our starting state, we input 1, and since the DL isn't enabled yet, there is no output from the combinational device (CD). ii) The clock goes high, 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**.

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 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?

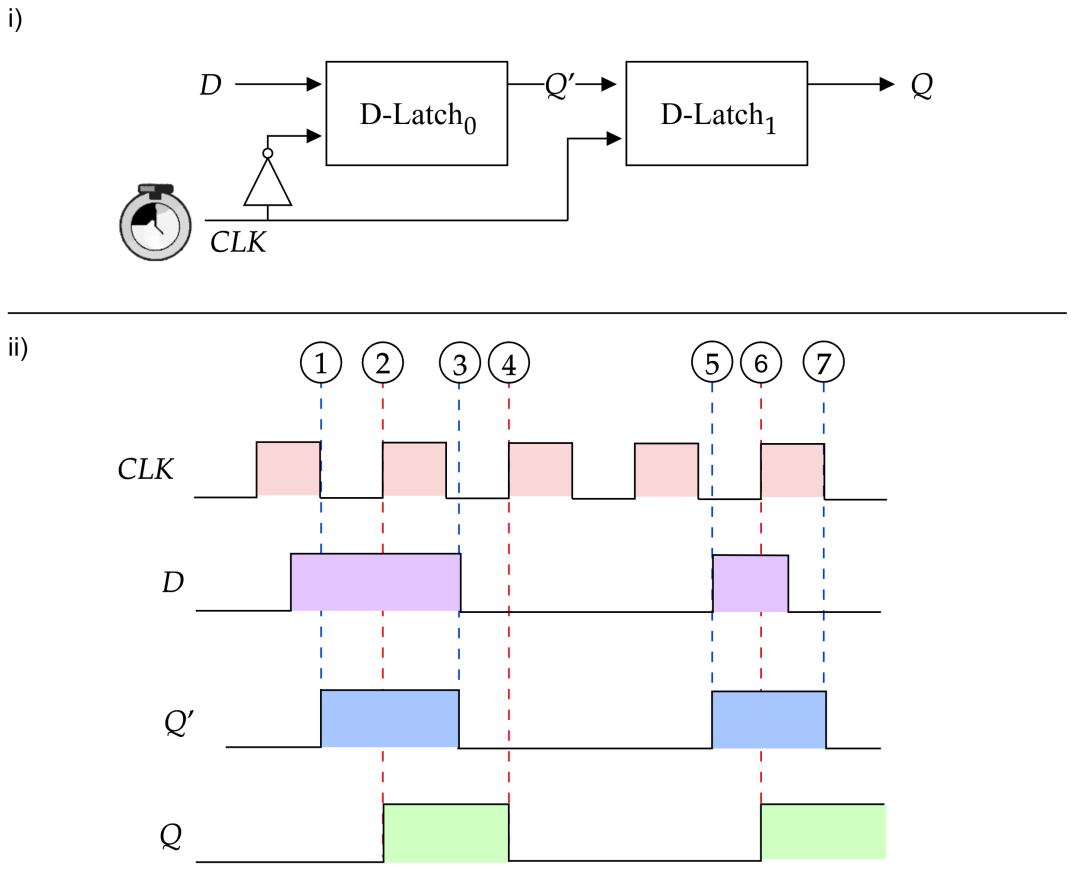


Figure 1.27: i) Shows 2 D-latches attached to one another, D-Latch$_1$ and D-Latch$_2$, which we'll call $D1$ and $D2$ respectively. $D1$ will act as our "bouncer". So when the clock is low, $D1$ prepares the data D for $D2$. Once the clock goes high, $D1$ blocks new data, serving Q' (the last written 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 (1.27) is called a:

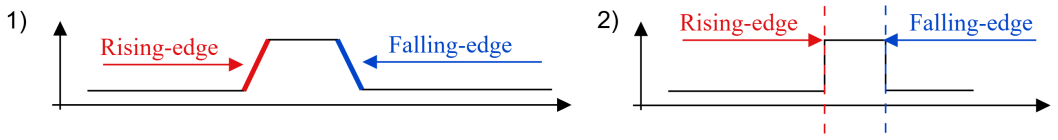
Definition 0.10: Data Flip-Flop

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 (1.27) is how signals are triggered:

Definition 0.11: 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,

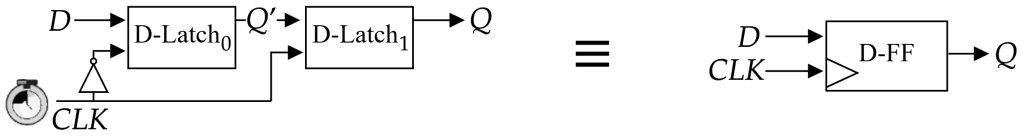


Figure 1.28: On the left is the circuit diagram for a D Flip-Flop, and on the right is the abstracted interface; Many texts use the triangle from clock to indicate edge-triggered behavior. [4]

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.

1.0.6 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:

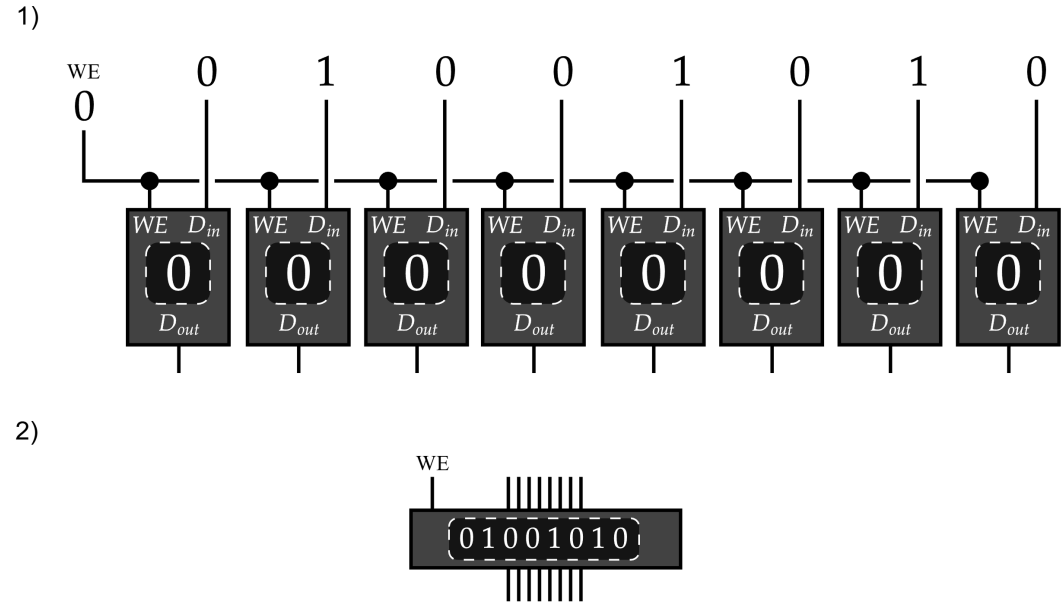


Figure 1.29: 1) Shows 8 gated latches connected in parallel, their 'Write Enable' (WE) lines are connected to a common load line. 2) Shows the abstracted diagram after the write from (1) goes through. This device is a called an 8-bit **Register**.

Definition 0.12: 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:

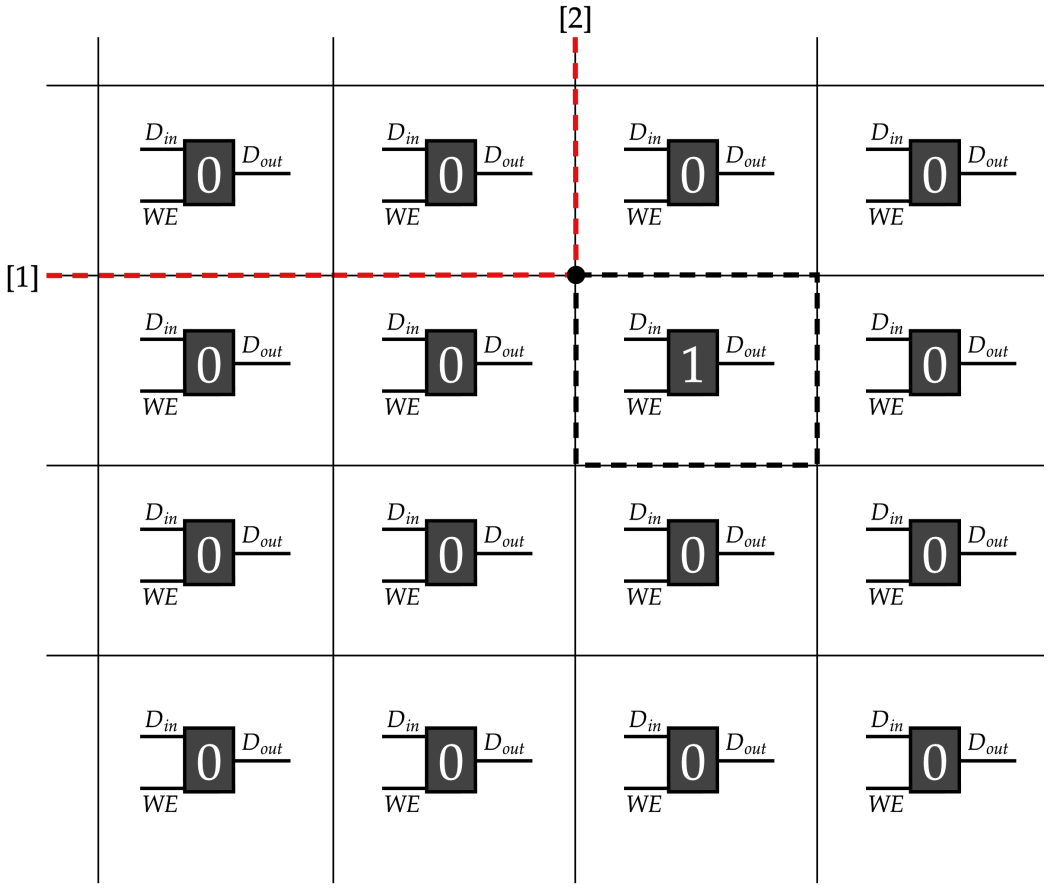


Figure 1.30: A 4x4 grid of latches (16 locations total). Here we write '1' to row 1 and column 2.

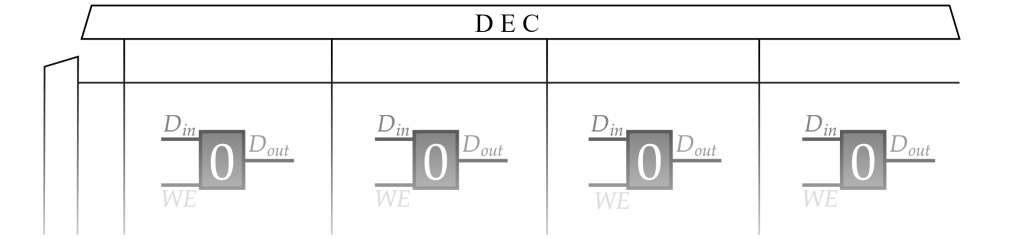


Figure 1.31: To achieve the row and column selection, we use decoders for both rows and columns.

Let's see this in action:

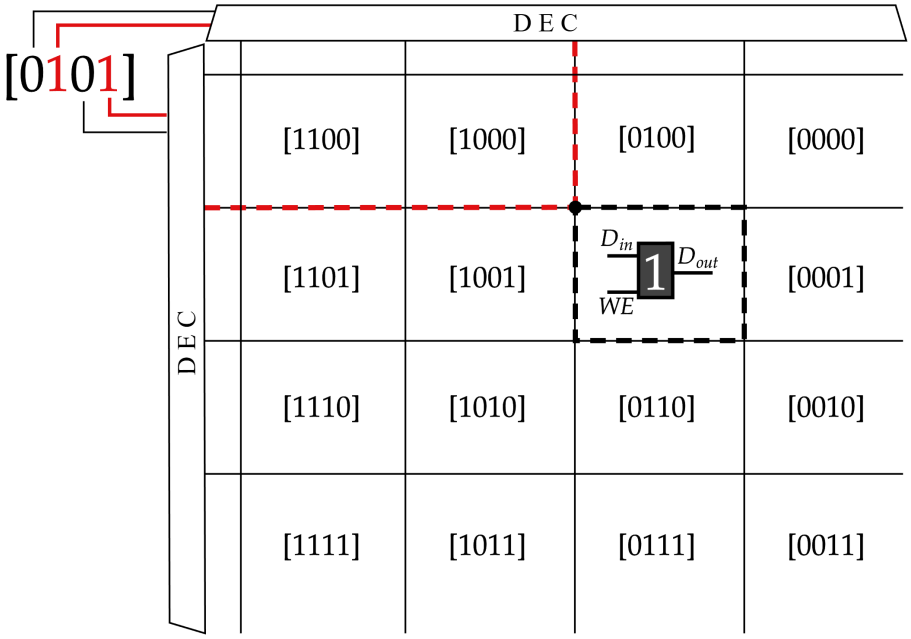


Figure 1.32: Just how houses are arranged in a grid with streets and avenues, we give each latch an **address** based on its row and column ($[0001], [0010], \ldots, [1111]$). Here we activate the same latch in Figure (1.30) by selecting [0101]. The address is split into two halves: The higher order 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):

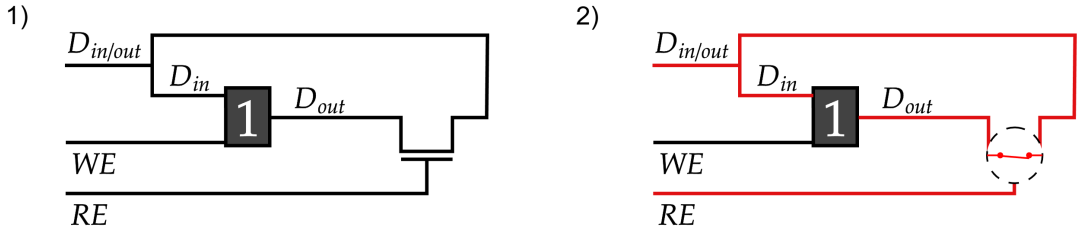


Figure 1.33: 1) Combines D_{in} and D_{out} lines, for a $D_{in/out}$ line. Where D_{out} is only active when RE is high via an NFET gate. 2) Shows that NFET gate turned on when RE is high. Note D_{in} is not affected as WE is low (vice-versa).

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

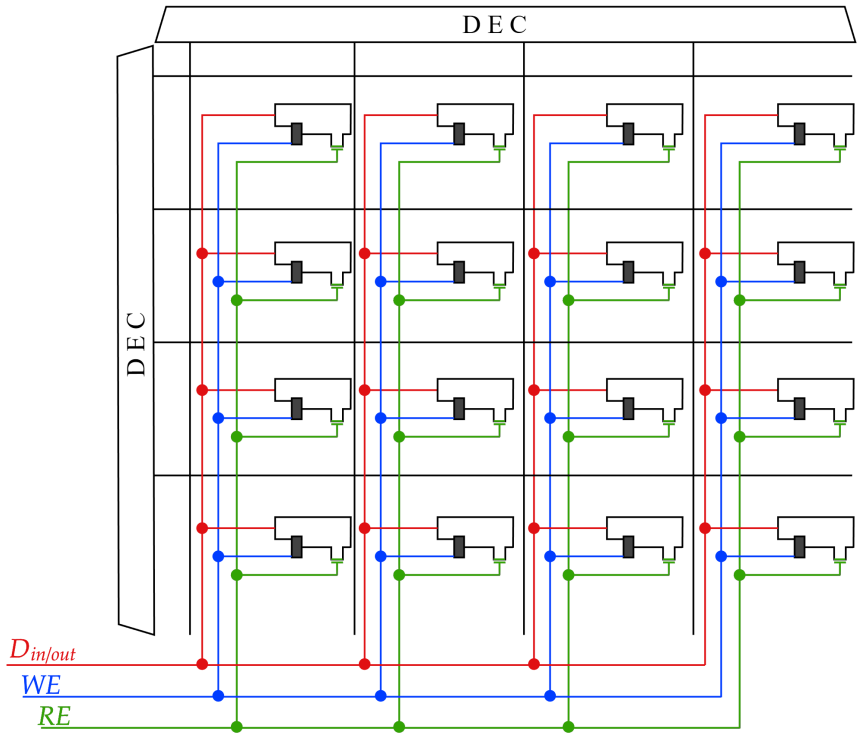


Figure 1.34: Here $D_{in/out}$ (Red), WE (Blue), and RE (Green) lines are combined in each cell of the grid; **However**, this design writes and reads to **all** cells.

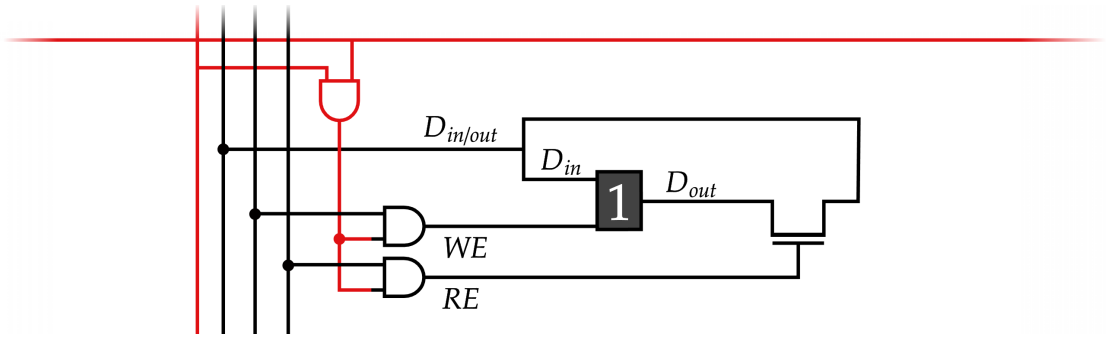


Figure 1.35: Using our addressing method in Figure (1.32), if we set an AND gate to the row and column select lines, feeding into WE and RE, with their AND control lines, we ensure only one cell is written to/read from at a time. Here this cell is selected (red lines), however, we haven't chosen whether 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:

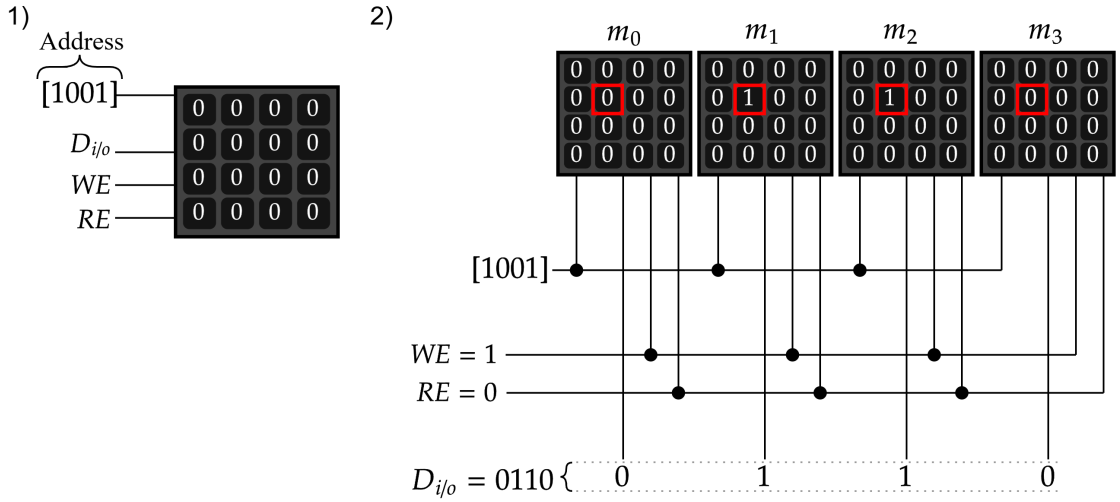


Figure 1.36: 1) Shows the abstracted 4x4 latch matrix with it's address line, $D_{i/o}$ (Data in/out), WE (Write Enable), and RE (Read Enable) lines. 2) Shows 4 of these matrices (m_0, m_1, m_2, m_3) connected in parallel, WE and RE lines are shared, while $D_{i/o}$ lines are now 4-bits wide. To write 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.

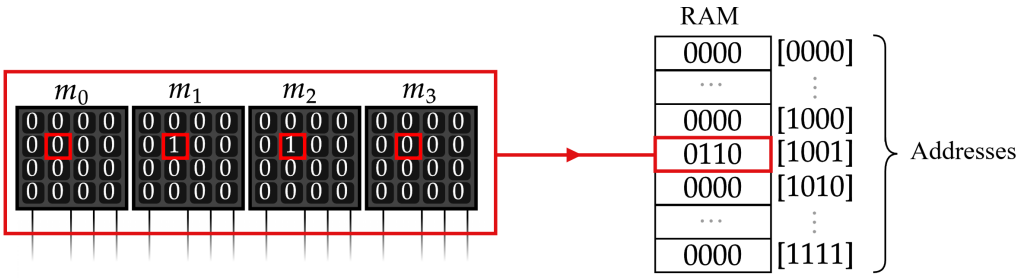


Figure 1.37: Continuing from Figure (1.36), we abstract even further. We instead **stack** each address vertically. Now we can at random choose any address to read from or write to; Hence we call it—**Random Access Memory** (RAM).

Bibliography

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

[2] Padraic Edgington. 7-1. building a 32-bit adder. YouTube, https://youtu.be/-nAg1TFgDjQ?si=U_TvV1cxAVgynnD9, 2017. Accessed: 2026-01-07.

[3] Quora. How do you design a 12-to-1 multiplexer from 2-to-1 multiplexers only? https://www.quora.com/How-do-you-design-a-12-to-1-multiplexer-from-2-to-1-multiplexers-only, 2025. Accessed: 2025-12-23.

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

[5] Warren Toomey and Mirela Damian. Logic gates - building an alu. http://www.csc.villanova.edu/~mdamian/Past/csc2400fa13/assign/ALU.html. Accessed: 2026-01-07.