Faculty of Engineering

Digital System Design

Semester 1 2023-2024

Jose Alberto ESPINO PITTI, *UID:3035946813*

2023

Course Code

ELEC

3342

Class Contents

Introduction

The concept of system is extremely relevant in the course because we are handling **systems** — which can get very complex as we increase the number of components. When you design such systems, you need to consider all relevant trade–offs and make sure that the system is **correct** and **optimal**. In summary, a digital system is an electronic system that operates on a digital abstraction, as opposed to an analog system. This means that the system operates on discrete values instead of continuous ones. In the physical world, quantities occur mostly as continuous analog values, thus building systems that follow this convention is intuitive. The issue with this is that not only are the values these systems handle continuous, but the time they are processed in is also continuous. Since the exact value of a signal is needed, any degradation of the signal will be reflected in the output. This is why analog systems are very sensitive to noise, such as interference from outside the system (e.g. radio frequency interference), noise within the system (thermo noise, shot noise, etc.), and noise from non–ideal electronic components (e.g. resistors, capacitors, degradation of materials, etc.). On top of that, it is difficult to store and process any value in continuous time, especially due to the effect previous signals have on the current signal (see echo cancellation, reverb...). Lastly, it is a challenge to transport continuous signals because they are sensitive to degradation over time and space.
Digital systems, on the other hand, operate on **discrete values**, which are represented by a finite number of bits. This means that the system operates on a finite number of states, and the values are processed in **discrete time**. This means that the exact value of a signal is not needed, but rather the range of values that the signal can take. This is why digital systems are less sensitive to noise, and can be made arbitrarily accurate by increasing the number of bits. On top of that, it is easy to store and process any value. These systems work with discrete values and discrete time.

Check out Quantization and sampling

Analog data is turned discreet through **quantization**. This is done by **sampling** (obtaining data at a discrete time or space) the analog signal at a certain rate, and then rounding the value to the nearest discrete value. A way to see this is that sampling is making the x–axis (time) discrete, whereas quantization is the process of making the y–axis (value) discrete.

It is possible to work with analog systems, but digital ones are preferred because:

- Discrete values are easy to store and transport, especially since they don't face degradation over time and space
- Discrete values are easy to process, since they can be represented by a finite number of bits
- These values enable very powerful and complicated processing of the input (e.g. encryption, compression, ...)
- These values are less sensitive to interference and noise

However, these systems have issues:

- Garbage in, garbage out: if the input is not correct, the output will not be correct. That is why the way **sampling and quantization** are performed is extremely important
- Relatively slower than analog systems in standard circuit implementations
- Possible loss of information when sampling data

Check out Discrete systems

Systems that operate on discrete values can function on either discrete or continuous time unit. If it is the former, it tends to operate exclusively on combinational logic. If it is the latter, it tends to operate on sequential logic.

Combinational Circuits

Logic gates are the building blocks of combinational circuits. Essentially, logic gates operate on binary values, perform a simple operation, and output a **single** binary value.

Check out Binary Values

In this course, we will work with a *binary* digital abstraction, usually denoted as `TRUE` and `FALSE`, high and low, and 1 and 0. We can define the first element in any way we want as long as its value does not overlap with the second element. Binary abstractions are manipulable through *Boolean algebra*.

Let us look at the most common logic gates in more detail:

- `AND`:
  Its output is high only when all the inputs are high. Its truth table is shown in Table 1.1.

A	B	$A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

Table 1.1: Truth table for `AND` gate

The `AND` operation is performed like an ordinary multiplication in linear algebra (assuming the only possible values are 1 and 0). The `AND` gate is also called a **product** gate. Normally it is represented by a dot, as $y = a \cdot b$.

- `OR`:
  Its output is high when at least one of the inputs is high. Its truth table is shown in Table 1.2.

A	B	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

Table 1.2: Truth table for `OR` gate

The `OR` operation is performed like an ordinary addition in linear algebra (assuming the only possible values are 1 and 0). The `OR` gate is also called a **sum** gate. Normally it is represented by a plus sign, as $y = a + b$.

- `NOT`:
  Its output is the opposite of the input. Its truth table is shown in Table 1.3.

A	$\overline{A}$
0	1
1	0

Table 1.3: Truth table for `NOT` gate

The `NOT` gate is also called an **inverter**. Normally it is represented by a bar over the input, as $y = \overline{a}$.

- `NAND`:
  Its output is the opposite of the `AND` gate. Its truth table is shown in Table 1.4.

A	B	$\overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

Table 1.4: Truth table for `NAND` gate

The `NAND` gate is also called a **negative product** gate. Normally it is represented by a dot with a bar over it, as $y = \overline{a \cdot b}$.

- `NOR`:
  Its output is the opposite of the `OR` gate. Its truth table is shown in Table 1.5.

A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

Table 1.5: Truth table for `NOR` gate

The `NOR` gate is also called a **negative sum** gate. Normally it is represented by a plus sign with a bar over it, as $y = \overline{a + b}$.

- `XOR`:
  Its output is high when the inputs are different. Its truth table is shown in Table 1.6.

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Table 1.6: Truth table for `XOR` gate

The `XOR` gate is also called an **exclusive OR** gate. Normally it is represented by a plus sign with a circle around it, as $y = a \oplus b$.

These gates are represented in circuits as as:

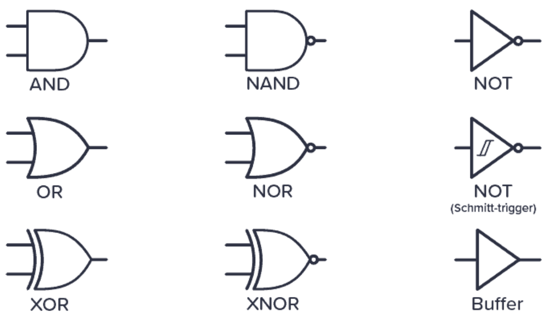


Figure 1.1: Schematics of logic gates, sourced from this website

Check out Extending the gates to more than two inputs

The gates can be, for the most part, intuitively extended to more than two inputs. For instance, the `AND` gate can be extended only returning positive if all n inputs are positive. The `OR` gate can be extended only returning positive if at least one of the n inputs is positive. The debate arises when it comes to the gate `XOR`: should it return positive if an odd number of inputs is positive, or if exactly one input is positive? A lot has been talked about this topic, as this website attests. As a fun fact, this was part of an assignment question on this iteration of the course, and the solution expected of us was the one where the `XOR` gate returns positive if and only if an odd number of inputs is positive.

Using these gates, we can build **combinational functions**, which are functions that take binary inputs and produce a binary output. For example, the function $y = a \cdot b + \overline{a} \cdot \overline{b}$. These functions must meet some conditions, such as

- All enclosing functions are combinational
- There is no loop in the connection
- Every node is either an input or connects to **exactly one** output

The three representations that we previously saw are the **truth table**, the **Boolean expression**, and the **schematic**. They are equivalent and can be converted from one to the other. Truth table is the only **unique** representation, which means that there is only one truth table for each function. However, there are multiple possible Boolean expressions and schematics for a single function.
Converting from schematics to Boolean expressions is straightforward:

1. Label all inputs
2. Repeat until all nodes labeled:
   - For all gates G with all inputs labelled:
     compute and label output of G

Converting from a boolean expression to schematic is also direct: it is the inverse of the previous process. You can use the following algorithm:

1. Make the entire expression the output of a gate
2. Until you have reached the single letter inputs, do:
   - IF:
     There are any plus signs, make each of the terms separated by the sign the inputs of an `OR` gate
     ELSE IF:
     There are any dots or products, make each of the terms multiplied the inputs of an `AND` gate
   - Each input of these extra gates you have created is the new entire expression you have to deconstruct

Check out Precedence

The precedence of the operators is as follows:

1. `()`
2. `NOT`
3. `AND`
4. `OR`

This means that if you have an expression such as $y = a + b \cdot c$, you should first compute $b \cdot c$ and then add a to it. This is because the `AND` operator has precedence over the `OR` operator. When deconstructing an expression, this precendece is inversed.

Circuit components can be grouped into sub–circuits known as **blocks**, which can simplify the circuit and make it more readable. In a logic circuit, each function is represented by a block (also known as module) and reused multiple times.
The process of converting from a truth table to a boolean expression is somewhat more complicated. Firstly, it is important to introduce the term **minterm**, which is a product that involves all the input variables (regardless of there being a negation or not). In a truth table, each row represents a minterm. The boolean expression will be the sum of all the minterms that have a 1 in the output, something called **sum of products**.
Converting from a boolean expression to a truth table itself is trivial: given an expression in its canonical sum of products form, you just have to put a 1 in each row that corresponds to a minterm. The tricky aspect is getting the expression into the SOP form in the first place.

Check out Canonical Expressions

Boolean expressions can be expressed in many ways. Canonical expressions eliminate this ambiguity by imposing a standard form. There are main forms of canonical expressions:

- **Sum of products (SOP)**: a sum of products of literals. Canonically, it may be negated but it should have no parentheses.
- **Product of sums (POS)**: a product of sums of literals.

Canonical expressions must not contain any parentheses.

1.0.1 Boolean Algebra

Boolean algebra is a set of rules that allow us to manipulate Boolean expressions. This branch of algebra is based on a series of axioms, mainly: T

Axiom	Dual	Description
$B = 0$ if $B \neq 1$	$B = 1$ if $B \neq 0$	Binary Field
$\overline{0} = 1$	$\overline{1} = 0$	Complement/NOT
$0 \cdot 0 = 0$	$1 + 1 = 1$	Annihilator/ AND OR
$1 \cdot 1 = 1$	$0 + 0 = 0$	Identity/ AND OR
$0 \cdot 1 = 0$	$1 + 0 = 1$	Null/ AND OR

Table 1.7: Important axioms of Boolean algebra

If you replace the `AND` operator with the `OR` operator and replace every 1 with a 0, the axiom still holds true. This is what is known as the **dual** of the axiom.
Similarly, there are certain useful theorems we apply when handling operations involving a single variable:

Theorem	Dual	Description
$B \cdot 1 = B$	$B + 0 = B$	Identity
$B \cdot 0 = 0$	$B + 1 = 1$	Null
$B \cdot B = B$	$B + B = B$	Idempotent
$B \cdot \overline{B} = 0$	$B + \overline{B} = 1$	Complement

Table 1.8: Important theorems of Boolean algebra

Check out De Morgan's Laws

De Morgan's laws are a set of rules that allow us to manipulate Boolean expressions. They are as follows:

- $\overline{A + B} = \overline{A} \cdot \overline{B}$
- $\overline{A \cdot B} = \overline{A} + \overline{B}$

In short, **break the bar, change the operator**

Now, let us look at some **multivariable** theorems:

Theorem	Dual	Description
$B \cdot C = C \cdot B$	$B + C = C + B$	Commutative
$(B \cdot C) \cdot D = B \cdot (C \cdot D)$	$(B + C) + D = B + (C + D)$	Associative
$B \cdot (C + D) = (B \cdot C) + (B \cdot D)$	$B + (C \cdot D) = (B + C) \cdot (B + D)$	Distributive
$B \cdot (B + C) = B$	$B + (B \cdot C) = B$	Absorption
$(B \cdot C) + (B \cdot \overline{C}) = B$	$(B + C) \cdot (B + \overline{C}) = B$	Combining
$(B \cdot C) + (\overline{B} \cdot D) + (C \cdot D) = B \cdot C + \overline{B} \cdot D$	$(B + C) \cdot (\overline{B} + D) \cdot (C + D) = B + C \cdot (\overline{B} + D)$	Consensus

Table 1.9: Important theorems of Boolean algebra - Multivariable

1.0.2 Circuit Schematics

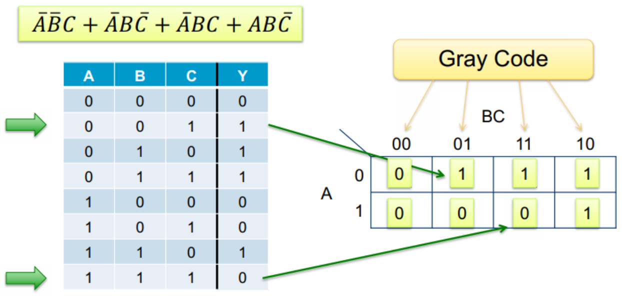
Circuit schematics are a way to represent circuits in a more visual way. They are composed of **logic gates** and **wires**. Logic gates are represented by their symbols, and wires are represented by lines. Usually, some conventions are followed:

- Inputs on the left
- Outputs on the right
- Gates flow from left to right
- Straight wires are preferred
- Wires always connect at a T junction
- Wires crossing without a dot **do not connect**

If possible, it is ideal to **minimise** logic circuits, since the higher the number of logic gates, the larger the area, the power consumption, and the delay of the circuit. This minimisation is usually achieved either via **Boolean algebra** or via **Karnaugh maps**. The latter refers to a method of representing the truth values of a function in a wat that allows us to identify the minterms of an expression. Based on those, we can simplify the expression.
The following steps allow us to express a truth table as a Karnaugh map:

1. Draw a grid with the number of rows and columns being the number of variables
2. Label the rows and columns with the binary values of the variables
3. Fill in the grid with the output values of the function
4. Group the 1s in the grid in groups of 2^n cells, where n is a positive integer. Note:
    - The groups can be horizontal or vertical
    - The groups can wrap around the grid
    - The groups can overlap
    - Make the largest groups possible
5. For each group, write the product of the variables that are constant in the group
6. Sum all the products
7. Simplify the expression

Here is an example of this method:



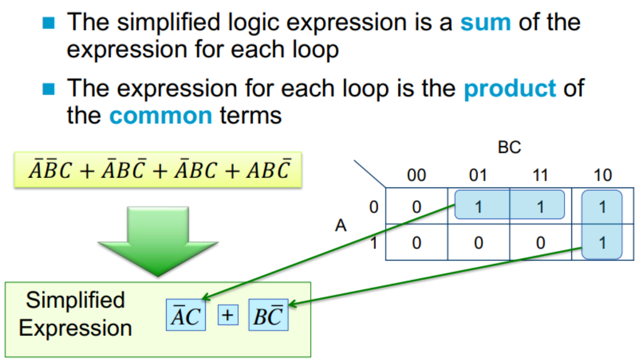
$\bar{A}\bar{B}C + \bar{A}B\bar{C} + \bar{A}BC + AB\bar{C}$

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

A \ BC	00	01	11	10
0	0	1	1	1
1	0	0	0	1

Figure 1.2: Example of Karnaugh map, sourced from the slides on Combinational Logic by Professor Hayden So of the University of Hong Kong

Here is how the simplification is carried out:



- The simplified logic expression is a **sum** of the expression for each loop
- The expression for each loop is the **product** of the **common** terms

$\bar{A}\bar{B}C + \bar{A}B\bar{C} + \bar{A}BC + AB\bar{C}$

A \ BC	00	01	11	10
0	0	1	1	1
1	0	0	0	1

Figure 1.3: Example of Karnaugh map simplification, sourced from the slides on Combinational Logic by Professor Hayden So of the University of Hong Kong

Combinational Building Blocks & Arithmetic Circuits

When building a system, we can follow one of two approaches:

- Top–down: we start with the overall system requirements and break the system down into smaller and smaller subsystems until we reach the level of individual components (unbreakable components).
- Bottom–up: we start with the individual components and build them up into larger and larger subsystems until we reach the level of the overall system.

These unbreakable components are called **building blocks**, the basic components that we use to build our system. Let us look at some of the most common circuit blocks:

- **Multiplexers**: a multiplexer (mux) is a circuit that has multiple inputs and a single output. It selects one of its inputs to output through a set of **select** inputs. The number of select inputs determines the number of inputs the multiplexer has. The number of select inputs is n, the number of inputs is 2^n, and the number of outputs is 1.

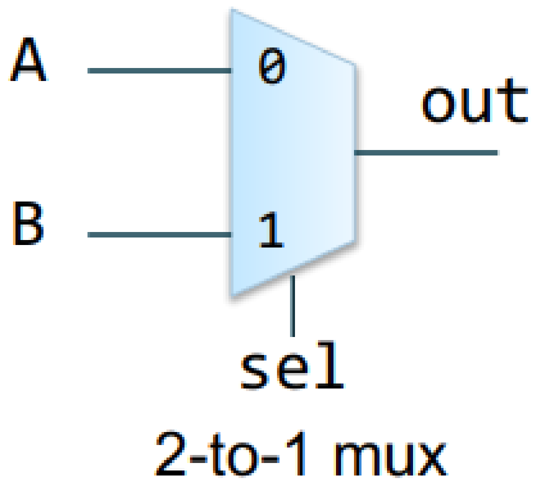


Figure 1.4: Schematic of a multiplexer, sourced from the slides on Combinational Logic by Professor Hayden So of the University of Hong Kong

A way to implement a mux is to treat it as simple combinational logic; you determine its truth table and then use logic gates to implement such truth table.
When there are more than one select inputs, we usually represent them through **bus notation**; this means that we represent a group of wire signals as a single multiple–bit wire signal. For example, a 32 bit signal X would be represented as `X(31 downto 0)` or `X[31:0]`. If we want to refer to a particular bit of the bus, we can use the notation `X[2:0]` (including bits 1, 2, 3) or `X[0]`. In this course, we denote the **leftmost** bit with the largest index in a bus. When it comes to **4–to–1** muxes, they can be implemented in two ways:

  - As a big combinational circuit, where you express the desired function as a truth table and then proceed to implement it using logic gates

– Using smaller subcircuits (2–to–1 muxes), such as in

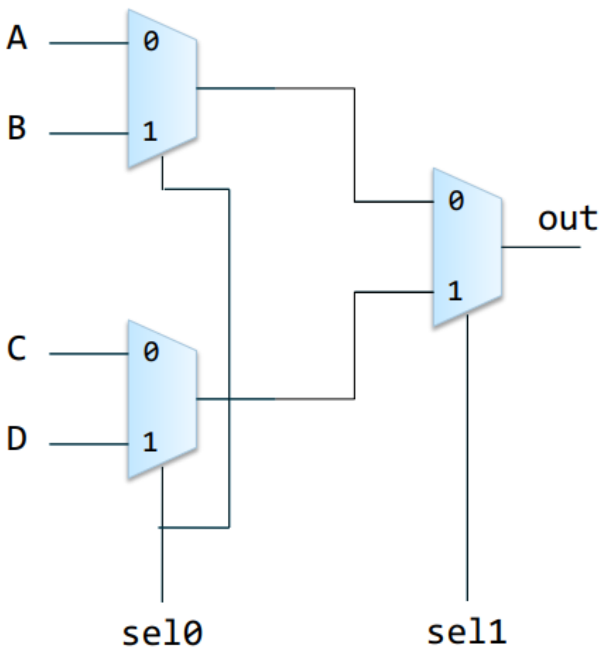


Figure 1.5: Schematic of a 4–to–1 mux, sourced from the slides on Combinational Logic by Professor Hayden So of the University of Hong Kong

Check out VHDL Muxes

VHDL, also known as VHSIC Hardware Description Language, is a language used to describe digital and mixed–signal systems as well as their behaviour. It is the main language that we will use in this course. If you want to express a mux in VHDL, you should express it in its expected form:

```
architecture behav of mux2 is
begin
    Y <= A when (S = '0') else B;
end behav;

```

Similarly, buses are expressed as **arrays of bits**:

```
signal X: std_logic_vector(31 downto 0)
--refer to individual bits as
X(0) --bit 0
X(18 downto 0) --bits 0 to 18

```

4:1 muxes can be expressed in many ways, such as:

```
--using a big combinational circuit
process (sel, A, B, C, D)
begin
    case sel is
        when "00" => Y <= A;
        when "01" => Y <= B;
        when "10" => Y <= C;
```

```
                when "11" => Y <= D;
                when others => Y <= A;
            end case;
        end process;
        --using smaller subcircuits
        architecture behav of mux4 is
            component mux2
                port (A, B, S: in std_logic;
                      Y: out std_logic);
            end component;
        begin
            mux2_1: mux2 port map (A, B, S(0), Y(0));
            mux2_2: mux2 port map (C, D, S(0), Y(1));
            mux2_3: mux2 port map (Y(0), Y(1), S(1), Y);
        end behav;
```

- **Demultiplexers**: a demultiplexer (demux) is a circuit that has a single input and multiple outputs. It selects one of its outputs to output through a set of **select** inputs. The number of select inputs determines the number of outputs the demultiplexer has. The number of select inputs is n, the number of outputs is 2^n, and the number of inputs is 1.

1.0.3 Binary Numbers

In Computer Engineering, two numbering schemes appear frequently: **binary**, which represents numbers in **base 2** and **hexadecimal**, which represents numbers in **base 16**. The conversion between these two is easy, so hexadecimal is often used as a *shorthand* for the binary system.
In order to convert from decimal to binary/hexadecimal, we can use the following algorithm:

1. Divide the number by the base (2 or 16)
2. End when the remainder is smaller than the base
3. The remainders form the number in the resulting system when counted from the bottom

When we want to convert from binary/hex to decimal, we use the formula

$$\sum_{i=0}^{n-1} B^i d_i$$

where B is the base, n is the number of digits, and d_i is the value of the digit at position i. The right–most digit is at position 0.

Check out Signed and unsigned numbers

Binary numbers can be either signed or unsigned. Unsigned numbers are simply non–negative binary numbers using their natural binary representations. They represent **equally spaced** integers on the number line. Following this scheme, an n–bit string can represent the numbers in the range $[0, 2^n - 1]$. For example, an 8–bit string can represent the numbers in the range $[0, 255]$. However, this scheme is unable to represent negative numbers. This shortcoming is addressed by the **signed** number representation known as **2's complement**. In this scheme, the left–most bit is used to represent the sign of the number. If the bit is 0, the number is positive. If the bit is 1, the number is negative. The remaining bits represent the magnitude of the number. In digital circuits, the negative of any number can be found by flipping all the bits and adding a 1. For instance, 388 in binary is 0000000110000100. Flipping all the bits and adding a 1 gives us 1111111001111100, which is the binary representation of -388. The range of an n–bit string using this scheme is of $[-2^{n-1}, 2^{n-1} - 1]$ — which is asymmetric! The following formula returns the value of any 2's complement bitstring:

$$-2^{n-1} + \sum_{i=0}^{n-2} 2^i d_i$$

The conversion between binary and hexadecimal is straightforward once we know the fact that **from right to left, each group of four bits in binary form one digit in hex**.

1.0.4 Arithmetic Circuits

Binary addition is no different to regular addition—two positive integers can be added via long addition. When implementing this operation in a circuit, we need to mind the fact that when we have two ones being added together, there will be a **carry bit**, which is then added to the next column. The **half adder** is a simple circuit that performs a basic addition of two 1–bit values. It generates a carry out to the next bit if the result is 2. The truth table for this circuit is shown in Table 1.10.

A	B	Co	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Table 1.10: Truth table for half adder

Where Co stands for carry over and S stands for sum. Algebraically speaking, $S = a \oplus b$ and $Co = a \cdot b$.
When we are handling more than one bit, we need to consider the possibility of a carry over bit from the bit to the right. Thus, we design a so–called **full adder**, which is a three–input function as per Table 1.11.

Ci	A	B	Co	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

Table 1.11: Truth table for full adder

Here, $S = a \oplus b \oplus Ci$ and $Co = a \cdot b + Ci \cdot (a + b)$.
What if we need to add two bits — the previous carryover and the carryover from the present operation?

Since those instances are not infrequent, we need **multi–bit adders**. These adders are the result of connecting the carry output from a full adder to the carry input of the next full adder. The first full adder has a carry input of 0. The resulting circuit is called a **ripple carry adder**.

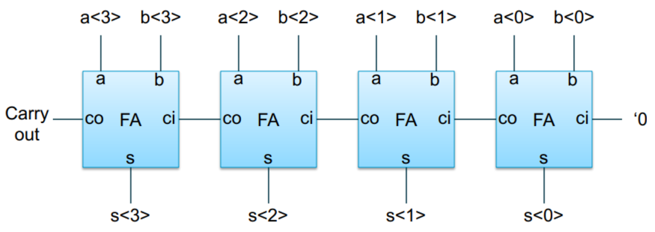


Figure 1.6: Schematic of a ripple carry adder, sourced from the slides on Combinational Logic by Professor Hayden So of the University of Hong Kong

As you can see in the image, the carry bit will be rippled across the adder from right to left. Once we have figured this out, representing **subtraction** is a relatively trivial task. We can simply use the fact that $a - b = a + (-b)$, where $-b$ is the two's complement of b — as long as we can find the negative value of a number, we can subtract it. Well, we know how to do this! We just need to simply negate all the bits and add 1. Bit flippers can be implemented using **XOR** gates.

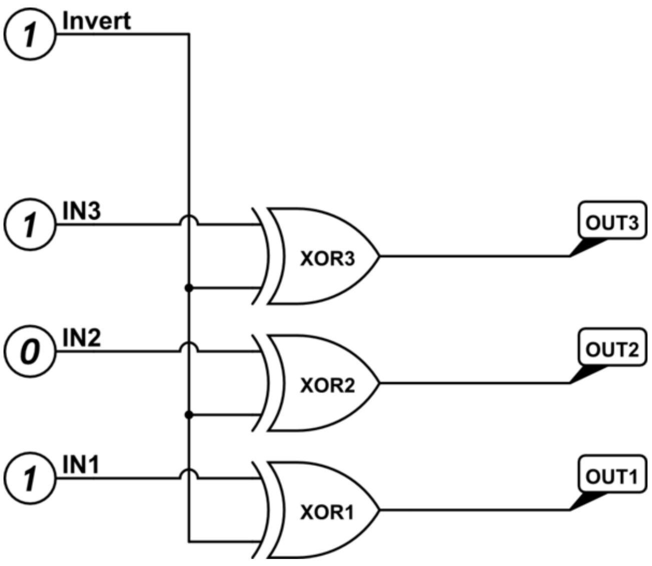


Figure 1.7: Schematic of a ripple carry subtractor (|1 = isNegative), sourced from this website

The other essential arithmetic blocks are the **comparators**, which determine the relationship between two binary numbers.

- **Equality**: It produces a single input indicating whether A is equal to B. It is normally implemented with a series of **XOR** gates as follows:

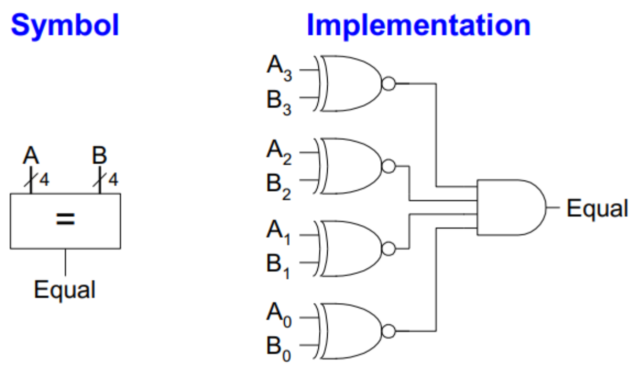


Figure 1.8: Schematic of an equality comparator, sourced from the slides on Combinational Logic by Professor Hayden So of the University of Hong Kong

- **Less Than**: Its output is positive only if $A < B$. A way this is performed is by computing $A - B$ and looking at the sign of the result. The symbol for this comparator is the one in Figure 1.9.

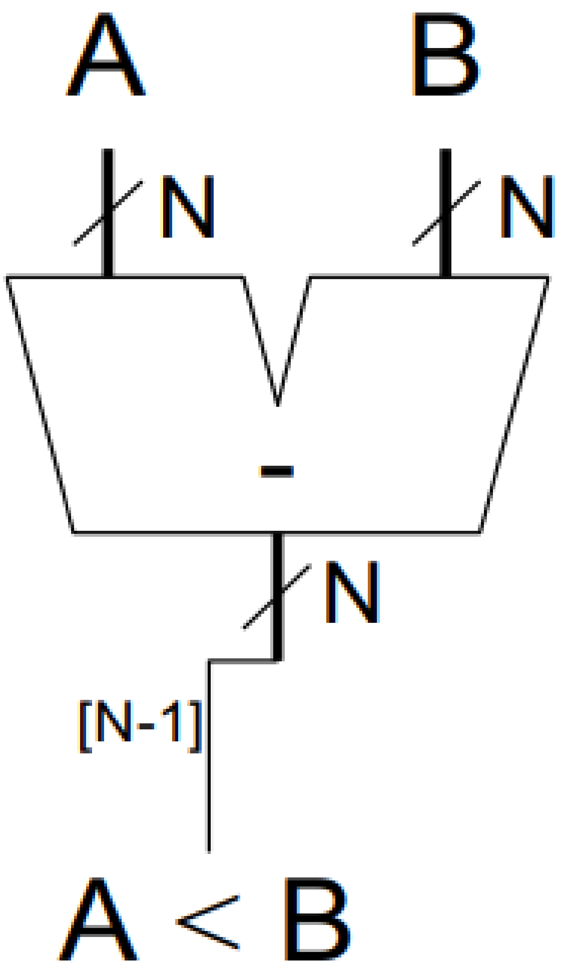


Figure 1.9: Schematic of a less than comparator, sourced from the slides on Combinational Logic by Professor Hayden So of the University of Hong Kong

- **Less Than or Equal To**: This can be trivially implemented by making the output of an equality unit and that of a less than unit the inputs of an `OR` gate.

Then, we have **shifters** and **rotators**. Shifters are circuits that shift the bits of a binary number to the left or to the righ, whereas rotators are circuits that rotate the bits of a binary number to the left or to the right. Let us look at them in detail:

- **Logical Shifter**: It shifts the bits of a binary number to the left or right and fills all empty spaces with zero. For instance, `11001 >> 2` means shift the number two bits to the right and results in `00110` If it were `11001 << 2` the result would be `00100`. Figure 1.10 shows the design of this shifter, where `shamt` means shift amount.

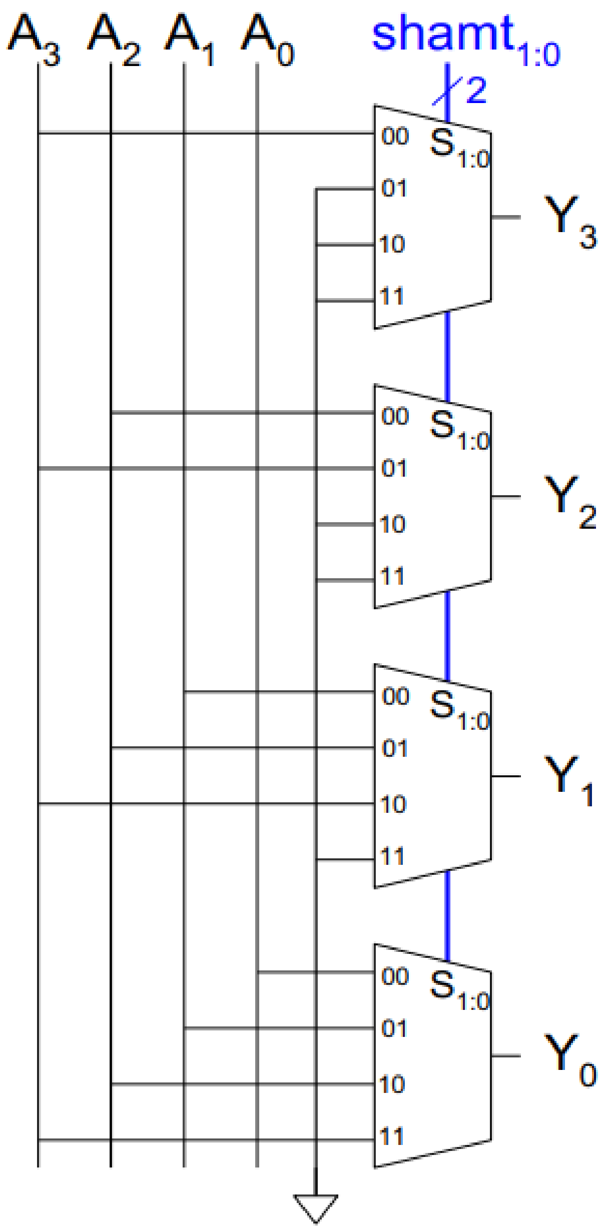


Figure 1.10: Schematic of a logical shifter, sourced from the slides on Combinational Logic by Professor Hayden So of the University of Hong Kong

- **Arithmetic Shifter**: Same as the previous shifter, with the exception that on right shift, it fills empty spaces with the previous most significant bit. For instance, `11001 >> 2` means shift the number two bits to the right and results in `11110` If it were `11001 << 2` the result would still be `00100`.

- **Rotator**: Rotates the bits. For example, `11001 ROR 2` will result in `01110`.

Check out Multipliers and Dividers

Shifters can be used to implement **multipliers** and **dividers**. For instance, to multiply a number by 2^n, we can simply shift it n bits to the left (e.g. `00001 << 2 == 1 * 2^2`). Similarly, to divide a number by 2^n, we can simply shift it n bits to the right. Multipliers form **partial products** by multiplying a single digit of the multiplier with a multiplicand. Then, the shifted partial products are **summed** to form the result. This comes together in the design in Figure 1.11.

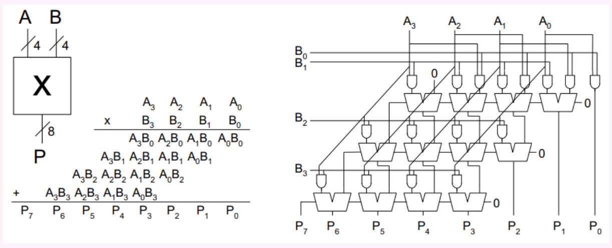


Figure 1.11: Schematic of a 4x4 multiplier, sourced from the slides on Combinational Logic by Professor Hayden So of the University of Hong Kong

In this design, the multiplier receives the multiplicand (A) and the multiplier (B) to produce the product (P). Each partial product is a single multiplier bit ($B3, B2, B1, B0$) `AND` the multiplicand bits ($A3, A2, A1, A0$). The partial products are then shifted and added together to produce the final product.

VHDL

As mentioned before, VHDL is a hardware description language. It refers to both the **conectivity** and the **function** of each module in hardware. However, it is important to notice that it does **not** describe **how** to implement each function — this is the job of the synthesis tool. When using HDL, we need to keep in mind that we are describing **hardware**, not software. This means that we need to think of the system in terms of hardware components, such as combinational logic, combinational building blocks, registers, finite state machines... The design procedure usually follows these steps:

1. Sketch a block design of the system. Focus on **what blocks are needed** and **how these blocks are connected**

2. Implement each block **one by one**

3. Put blocks together into a **working system**

Usually, any VHDL script contains three components:

- **Entity**:
  It defines the interface of a module. Each module has exactly one entity. It also must indicate a **port mode** for every signal it handles. Some common port modes are `in`, `out`, `inout`...

- **Architecture**:
  It defines the **function** of an entity, the thing it is supposed to do. It is uncommon, but one entity

may have multiple architectures; this allows the entity to have different behaviours depending on what the tools want to do. A declaration statement must follow a specific syntax:

```
        architecture <arch_name> of <entity_name> is
        declarations; --optional
        begin
            concurrent_statements;
            concurrent_statements;
        end <arch_name>;

```

It is important to note that the statements within the architecture body run **concurrently**, not sequentially. This means that the order of the statements does not matter. In this course, you will encounter three main types of concurrent statements:

– **Signal Assignment**:
  Simple assignment refers to assigning a signal to a value. For example,

```
            architecture ex1 of foo is
            begin
                x <= ( a and not b ) and c ;
                y <= ( a and not b ) and not c;
            end ex1;

```

  You can also instantiate signals that only exist within the architecture, such as

```
             architecture ex2 of foo is
                 signal tmp: STD_LOGIC;
                 --this signal is a NODE in the schematic
             begin
                 tmp <= ( a and not b );
                 x <= tmp and c ;
                 y <= tmp and not c;
             end ex2;

```

– **Component Instantiation**:
  This action places an instance of the component in the design, which in turns allows the design hierarchy to be built. The same component can be instantiated multiple times; however, each instance must have a unique name. Importantly, the component must be declared before it is instantiated. The syntax for this is as follows:

```
             --instantiate the component
             architecture ex3 of foo is
             -- declare the existence of the component
                 component sample is
                     port ( r : in STD_LOGIC;
                            s : in STD_LOGIC;
                            t : out STD_LOGIC);
                 end sample;
                 --notice the similarity to entity declaration
                 --except for keyword component
                 signal tmp: STD_LOGIC;
             begin
                 G1: sample port map (a, b, tmp);
                 --this is the instantiation of the component
                 -- the nodes are connected in the order they are declared
                 -- thus a connected to r, b connected to s, tmp to t.

```

– **Process**:
A process is a block of an arbitrarily long list of sequential statements. Statements within a process describe the function of the process with sequential statements, but as a whole, the process will be a non–sequential block of code. Synthesis tools will examine the function described and synthesises the corresponding implementation. Importantly, every process must declare its **sensitivity list**, which lists all the signals that will cause the process to execute; which is to say that the process will not execute unless one of the signals in the sensitivity list changes value. Take a look of this sample process:

```
            architecture ex4 of foo is
            begin
                process ( A, B, C ) --sensitivty list
                begin
                    Y <= ( A and not B ) and ( B or C );
                    -- Y <= C;
                end process;
            end ex4;

```

Note that if we were to uncomment the line `Y <= C;`, the value stored in `Y` would be `C`, because within the process, the statements are executed sequentially.
Variables can be declared inside a process; they have no physical correspondence (unlike signals) but are temporary storage for values during the execution of the process. They are declared as follows:

```
            architecture ex5 of foo is
            begin
                process ( A, B, C ) --sensitivty list
                variable tmp: STD_LOGIC;
                begin
                    tmp := ( A and not B ) and ( B or C );
                    Y <= tmp;
                end process;
            end ex5;

```

- **Library Packages**:
The `library` keyword loads predefined libraries while the `use` keyword specifies which libraries to use. Some common libraries are `STD_LOGIC_1164` for logic operations, `NUMERIC_STD` for signed and unsigned arithmetic, among others.

This library contains two common data types, the single bit `std_logic` and the multi–bit `std_logic_vector`. The former is used to represent a single bit, whereas the latter is used to represent a bus of bits. This library models the behaviour of **real wire** and provides values such as 0 (forces a 0), 1 (forces a 1), z (high impedance), x (unknown), u (uninitialized), – (don't care).
Here is a sample program:

```
    --Full adder

    --Libraries
    library IEEE;
    use IEEE.std_logic_1164.all;

    --Entity
    entity FA is
        port (A, B, Ci: in std_logic;
              Co, S: out std_logic);
```

```
end FA;

--One or more architectures per entity
architecture rtl of FA is
begin
    S <= A xor B xor Ci;
    Co <= (A and B) or (Ci and (A xor B));
end rtl;
```

Sequential Logic

Combinational logic has no memory of the past; the output is only a function of the present input. In contrast to this, **sequential logic** is influenced by the history of the inputs — something known as the memory effect. Sequential logic ciruits contain **state elements**, which are circuits that can hold a value over time. All the state elements collectively store the current **state** of the circuit.
Sequential circuits are unique because they provide **order** to the operation of the circuit—some operations will only compute once the inputs are ready. Furthermore, these kinds of circuits allow us to **coordinate** the operation of different parts of a circuit. The main objective of this coordination is that each part must operate on the correct set of data. Sequential circuits are composed of **state elements**, which are circuit components able to store a 1 or 0 internally **regardless of the input**. The most fundamental building block of state elements is the **bistable circuit**. This circuit has two outputs, Q and $\overline{Q}$. This circuit (Figure 1.12) creates a feedback loop that allows the circuit to store a value indefinitely. The only issue it has is that it has no input — thus, there is no way to change its state.

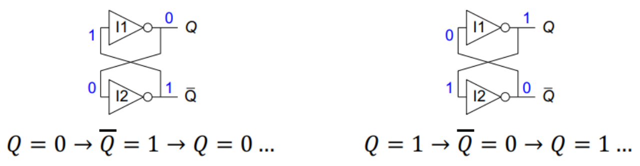

I1
I2
0
1
Q
$\overline{Q}$
$Q = 0 \rightarrow \overline{Q} = 1 \rightarrow Q = 0 \dots$
$Q = 1 \rightarrow \overline{Q} = 0 \rightarrow Q = 1 \dots$

Figure 1.12: Schematic of a bistable circuit, sourced from the slides on Combinational Logic by Professor Hayden So of the University of Hong Kong

Here is where **flip–flops** come into play. These circuits are bistable circuits with an input. They are also known as **edge–triggered** circuits, because they only change their state when a **clock** signal is received. The clock signal is a signal that oscillates between 0 and 1 at a fixed frequency and it is usually denoted as `clk` or `clock`. When the clock is not changing, the flip–flop will ignore the input. In this class, we will only use **rising edge triggered flip–flops**, which are flip–flops that change their state when the clock signal goes from 0 to 1. There are different types of flip–flops, but the most common one is the **D Flip–flop**. This type of FF has one data input port D, a single output port Q, and a clock input port clk. Optionally, this FF may contain a reset port R and an enable port En. At the rising edge of the clock signal, the value of the input D is stored. This data is outputed to Q after a small delay, represented as $T_{clk \rightarrow Q}$
D Flip–flops are often used as **delay lines** or **shift registers**, whose function is to delay the input signal by a certain amount of time. This is achieved by connecting the output of one FF to the input of the next FF in a chain of n length that will delay the input signal by n clock cycles. This works

because in hardware, all the parts of the circuit operate in parallel, so the output of the first FF will be available at the input of the second FF at the next clock cycle, and so on.
A **register** is a parallel composition of DFFs, where an n–bit register contains n DFFs. The difference between registers and singular FFs is the capacity the former have of storing **multi–bit** values, as you can see on Figure 1.13.

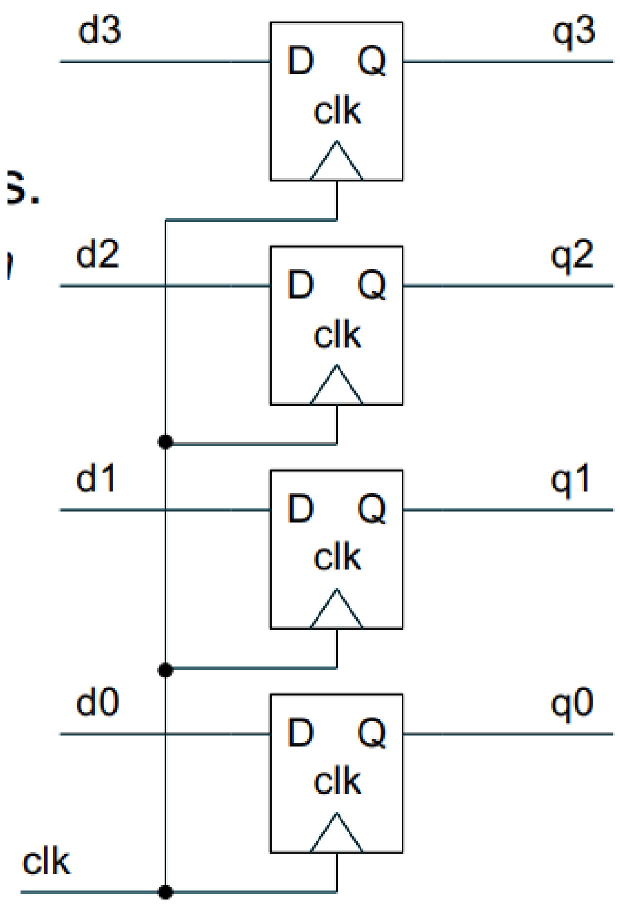


Figure 1.13: Schematic of a register, sourced from the slides on Combinational Logic by Professor Hayden So of the University of Hong Kong

If a FF has an En port, it is called an **enabled** FF. This means that on every clock edge, the DFF will be updated with D only if En is 1. If En is 0, the DFF will be "updated" with the previous Q. If a FF has an R port, it is called a **resettable** FF. This means that when R is 1, Q will be reset to 0 regardless of the clock signal.

Check out Reset vs Clear

There are two common definitions of reset. The first one is the **synchronous reset**, which can be implemented using logic gates. This reset will make the output Q 0 on the **next clock edge**. The second one is the **asynchronous reset**, which requires special hardware. This is an immediate change — as soon as clear is asserted, the output Q will be 0.

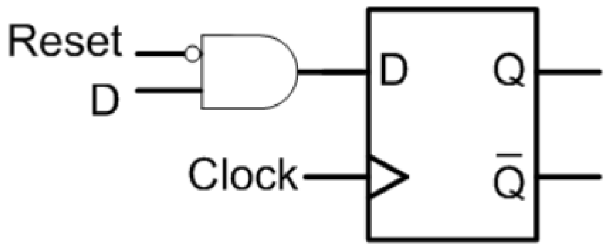


Figure 1.14: Schematic of a D Flip–flop with reset, sourced from the slides on Combinational Logic by Professor Hayden So of the University of Hong Kong

Let us understand better how to use DFFs. **Toggle flip–flops** toggle the output at a rising clock edge when the toggle signal T is 1; otherwise, the output remains unchanged. We can implement this with a DFF as follows:

1. Define a signal `nextQ` that denotes the output value of Q after the next clock edge
2. Make `nextQ` a function of Q and T This can be easily expressed in a truth table:

Q	T	$nextQ$
0	0	0
0	1	1
1	0	1
1	1	0

Table 1.12: Truth table for toggle flip–flop

3. Then, use a DFF to implement the function of storing `nextQ` into Q at the next clock edge

1.0.5 Synchronous Sequential Circuits

In a **synchronous** circuit, every state element is updated synchronously according to a single clock signal. In this kind of circuit, every circuit element is either a register or a combinational circuit, there must be **at least one register**, all the registers receive the **same clock signal**, and every cyclic path in the circuit must contain at least one register.

Check out Clock Signal

The clock signal is a signal that toggles between 0 and 1 periodically. The frequency of toggling determines the maximum speed of the circuit, which is measured via $\frac{1}{\text{clock period}} =$ clock frequency. Unless creating an advanced circuit, you should never connect clock signals to a normal port or connect normal signals into a clock port.

1.0.6 Sequential Circuits and VHDL

The hardware you describe using VHDL can be synthesised by certain tools, a process called **inferring circuits**. To implement state elements, you may use the function `rising_edge(clk)`, which returns true if the clock signal has just risen. This function is defined within the `std_logic_1164` library. For instance,

```vhdl
architecture ex6 of foo is
begin
    proc_reg: process (clk)
    begin
        if rising_edge(clk) then
            q <= d;
        end if;
    end process;
end ex6;
-------if we want to use synchronous reset:
architecture ex7 of foo is
begin
    proc_reg: process (clk, rst)
    begin
        if rst = '1' then
            q <= '0';
        elsif rising_edge(clk) then
            q <= d;
        end if;
    end process;
end ex7;
---alternatively
architecture ex8 of foo is
begin
proc_reg: process (clk)
begin
    if ( clk'event and clk = '1' ) then
        if ( reset - '1' ) then
            q <= '0';
        else
            q <= d;
        end if;
    end if;
end process;
end ex8;
```

Finite State Machine (FSM)

A **finite state machine** is an abstraction of computation that can be used to model many computing tasks. It is used to describe complex behaviour in circuits and systems, such as decision making, network communication, microprocessor control, and others. Each FSM defines a **finite** number of **states** that the machine can be in, as well as the conditions under which the machine will go from one state to the other. At any moment, the machine can only exist in **one** of the defined states. Similarly, its output will depend on the state the machine is in (and optionally on the input as well). FSM are generated with the help of **state transition diagrams**, a visual tool to describe the behaviour of the machines. Here, we represent each state as a block, and the transitions between states as directed edges. The edges are labelled with the format `condition / [output]` Here, the output is optional and represents the machine's output specifically during that transition.

Check out Moore Machine & Mealy Machine

When the output of a FSM depends only on the current state, we call it a **Moore Machine**; thus, this kind of Machine has no combinational paths between input and output of a state machine. On the other hand, when the output of a FSM depends on both the current state and the input, we call it a **Mealy Machine**. This kind of machine requires less states to implement the same function as a Moore Machine, but it is more complex to implement.

Finite State Machines can be implemented in hardware using **synchronous sequential circuits**. The state of the machine is stored in a register, and the state transition conditions can be combinational functions on input signals and the states. The outputs will thus simply be the output signals of the circuit. Here, the transition condition will be checked on every clock edge. The following steps can aid your design of such a machine:

1. Define the input and output signals
2. Determine how the FSM states will be represented in hardware. For instance, if there are n states, we will need $\log_2 n$ bits to represent them. Thus, if we have two FSM states, we will need one DFF.
3. Implement the state transition logic. At each cycle, you will need to determine the next state the FSM should be in the next cycle, as well as the transition conditions required. The next state logic will thus be a combinational function of the current state and the input signals. It can be found most directly through a truth table.
4. Determine the output logic, which is done in a way similar to how the next state logic is obtained.
5. Implement the circuit

Check out State Encoding

The most common way to represent abstract FSM states in hardware is through the **binary encoding scheme**, where each state is encoded using an n–bit binary number. This encoding scheme is simple and easy to implement, but it is not the most efficient. For instance, if we have a FSM with n states, we will need n DFFs to store the state. However, if we use a **one–hot encoding scheme**, we will only need n DFFs to store the state. This is because in this scheme, each state is represented by a single bit, and only one bit is 1 at any given time. This scheme is more efficient, but it is more complex to implement.
The mapping between state and encoding is arbitrary, so we usually encode the **reset state** as all zeroes for convenience.
It is also possible for your machine to end up producing **invalid** states. When this is the case, the machine can decide to ignore them, flag an error, reset the machine, or go to a default state.

1.0.7 VHDL

FSMs have 3 components, the state elements (registers), the next state logic (combinational), and the output logic (combinational). Considering this, a good approach to implementing these machines in VHDL is to use a **process** for each component. The next state logic and the output logic will be combinational processes.

1. State Registers:

```vhdl
        SYNC_PROC: process (clk)
        begin
        if (clk'event and clk = '1') then
            if ( r = '1' ) then
                state <= s_waitcard;
                -- s_waitcard is the reset state
            else
                state <= next_state;
            end if;
        end if;
        end process;

```

You can define a new custom VHDL enumerated data type called `state_type`. For example:

```vhdl
        architecture Behavioral of fsmdesign is
        -- let us tell VHDL the possible values of this data type
        type state_type is (s_waitcard, s_waitclk, s_waitdata, s_waitstop);
        -- Define two signals of this type
        signal state, next_state: state_type;
        begin

```

2. Next State:
   Since this is a combinational process, the next state will be a function of the current state and the input signals. This can be implemented using a `case` statement:

```vhdl
        NEXT_STATE_DECODE: process (state, valid, pass )
        begin
        --declare default state for next_state to avoid latches
        next_state <= state; -- default is to stay in current state
        case (state) is
            when s_waitcard =>
                if valid = '1' then
                    next_state <= s_waitpass;
                end if;
            when s_waitpass =>
                if pass = '1' then
                    next_state <= s_waitcard;
                end if;
            when others =>
                next_state <= s_waitcard;
        end case;
        end process;

```

3. Output Logic:
   This is also a combinational process, so the output will be a function of the current state and the input signals.

```vhdl
        OUTPUT_DECODE: process (state)
        begin
            if state = s_waitcard then
                motor <= '1';
            else
                motor <= '0';
            end if;
        end process;

```

1.0.8 Sequential Building Blocks

Counters are sequential circuits whose stored value increases on each clock edge. Binary counters will cycle through all binary numbers, just to wrap around after overflow. They can be implemented as in Figure 1.15.

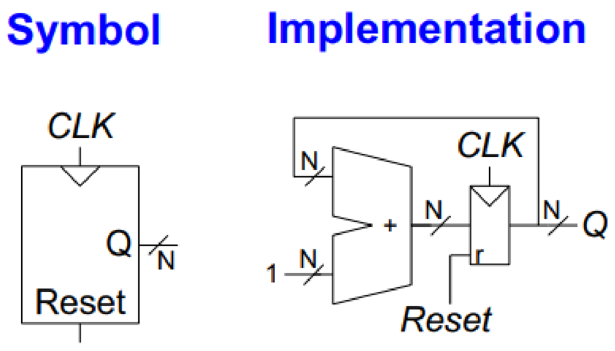


Figure 1.15: Schematic of a binary counter, sourced from the slides on Combinational Logic by Professor Hayden So of the University of Hong Kong

Using VHDL, this can be implemented as in the following sample:

```
library ieee;
use ieee.numeric_std.all;
--we need numeric_std for signals that involve
--arithmetic operations with unsigned type
architecture rtl of bcounter is
    signal cnt: unsigned( 31 downto 0 );
    --this is the counter
begin
    q <= std_logic_vector( cnt ):
    proc_cnt: process( clk )
    begin
        if (clk'event and clk = '1') then
            cnt <= cnt + 1;
        end if;
    end process;
end rtl;
--if we need to add clear, we built it as
architecture rtlr of bcounter is
    signal cnt: unsigned(31 downto 0);
begin
    q <= std_logic_vector( cnt );
    proc_cnt: process( clk, clr )
    begin
        if ( clr = '1' ) then
            cnt <= x"00000000";
        elsif ( clk'event and clk = '1' ) then
            cnt <= cnt + 1;
        end if;
    end process;
end rtlr;
```

Similarly, we have the **shift register**, which is a sequential circuit that shifts the bits of a binary number to the left or right and fills all empty spaces with zero. For instance, `11001 >> 2` means shift the number two bits to the right and results in `00110` If it were `11001 << 2` the result would be `00100`. This can be implemented as in Figure 1.16.

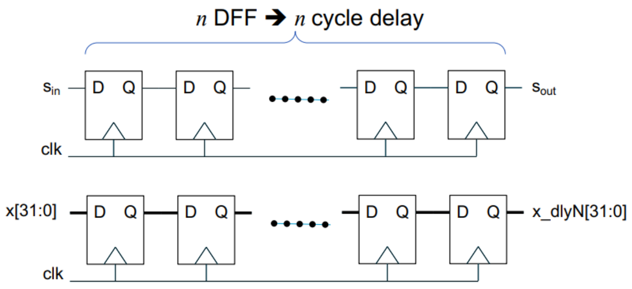


Figure 1.16: Schematic of a shift register, sourced from the slides on Combinational Logic by Professor Hayden So of the University of Hong Kong

Another important device is the **serial–to–parallel converter**. These devices are tasked with shifting a new bit in on each clock edge for N cycles. It has two modes, one where it reads the serial output and another one where it reads the parallel output. This can be implemented as in Figure 1.17.

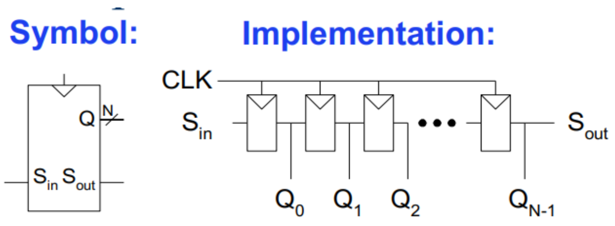


Figure 1.17: Schematic of a serial–to–parallel converter, sourced from the slides on Combinational Logic by Professor Hayden So of the University of Hong Kong

Shift registers with parallel loads can be used as serial–parallel converters. If you wish to do serial–to–parallel conversion, you need to convert from S_{in} to $Q_{0:N-1}$. Alternatively, if you wish to do a parallel–to–serial conversion, you need to convert from $D_{0:N-1}$ to S_{out}, as shown in Figure 1.18. When $load = 1$, the device acts as a normal N–bit register. When $load = 0$, the device acts as a shift register.

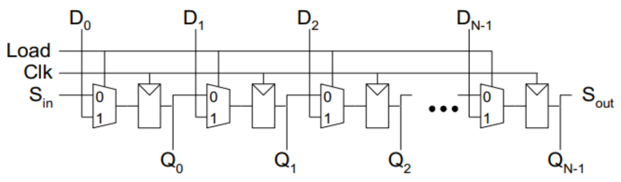


Figure 1.18: Schematic of a shift register with parallel load, sourced from the slides on Combinational Logic by Professor Hayden So of the University of Hong Kong

Memory and Related Building Blocks

Memory arrays are 2–dimensional arrays of bit cells. Each bit cell is tasked with storing one bit. They are used for storing data, logic, reorganisation of data, FIFO, and more. Memory arrays are randomly addressable, and can be found in either **RAM** (Random Access Memory) or **ROM** (Read Only Memory). The dimensions of the memory array are defined by the number of **address bits** N and the number of data bits M. The number of rows is 2^N and the number of columns is M. The number of rows is also known as the number of words or the **depth**, whereas the number of columns is also known as the size of word or the **width**. The overall array size is $2^N \times M$ bits.
Internally, most memory arrays have a **row decoder**, which selects the row to be read or written, and a **column decoder**, which selects the column to be read or written, as seen on Fig 1.19. The row decoder is a combinational circuit that takes the address bits as input and outputs a single row. The dimensions of the decoder are contingent on the size of N, which can prove problematic considering the cases when N is a large number, such as 32. This layout directly it not scallable, which is why real memory is divided into *banks* or have different column organisations.

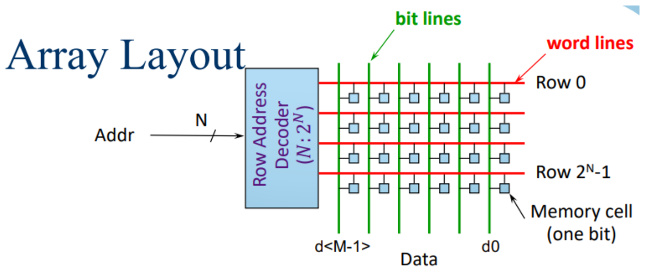


Figure 1.19: Schematic of a memory array, sourced from the slides on Combinational Logic by Professor Hayden So of the University of Hong Kong

There are two main elements involved in read/write operations, the **wordline** and the **bitline**. The former is the row decoder, whereas the latter is the column decoder. In the case of **read** operations, there is only one wordline active at a time. The row of memory array to read is selected, and then one address is read at a time. The bitline is connected to the output of the memory array, and the value of the bitline is determined by the value of the bit cell. The case of **write** operations is similar; the wordline will select the row of memory array to write, considering only one wordline can be active at a time. Then, the bitline will broadcast the data from the external source to all cells. The values broadcasted by the bitline will only be stored by the row of memory array selected by the wordline.
The **naive memory implementation** consists in mimicking the behaviour of a memory array using a DFF with enable coupled with a **tristate buffer**. Read is performed when wordline is selected, where the values will be passed through the buffer. If wordline is not selected, the output of the buffer will remain being Z. Write is achieved by using the wordline to control the enable signal. The implementation of this can be seen in Figure 1.20.

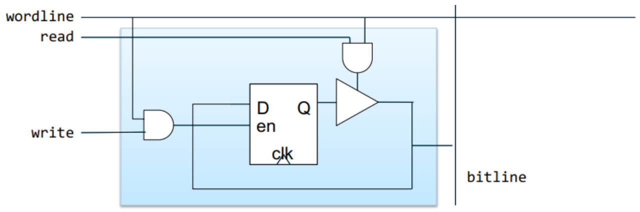


Figure 1.20: Schematic of a naive memory implementation, sourced from the slides on Combinational Logic by Professor Hayden So of the University of Hong Kong

Check out Tristate Circuit

Real world circuits have properties that are not modeled by Boolean equations: timing and Z. Some conbinational circuits can produce a **high–Z** (Z) output; this output has a value that is neither 0 or 1, but rather a high impedance state (a way for the driver to say "I'm not driving anything"). This is achieved by using a **tristate buffer**, which is a circuit that has an input, an output, and an enable signal. When the enable signal is 1, the output is the same as the input. When the enable signal is 0, the output is Z. Essentially, when the enable signal is 0, the output is physically disconnected from the input. Tristate buffers are often used to connect multiple mutually exclusive drivers to the same node: both drivers share the same enable signal, but only one of them is enabled at any given time.

The naive implementation is too complicated for large memory, considering that each bit cell requires one DFF and one tristate buffer. This is why we use different types of memory technologies to suit our needs.
The two main big categories are **ROM** (Read Only Memory) and **RAM** (Random Access Memory). ROM is a memory array that can only be read, whereas RAM is a memory array that can be read and written. RAM is further subdivided into **volatile** and **non–volatile** memory. The former loses its data when the power is turned off, whereas the latter retains its data even when the power is turned off. Let us look at some examples of these technologies.

1. **SRAM** (Static RAM):
   This memory technology is volatile and mostly designed on standard digital circuit technology. It has a simple read/write interface and is very fast. However, it is expensive and has a low density. The most common SRAM cell design is that of a simple cross–coupled inverter as in Figure 1.21. The read operation involves simply taking the value from Q or Q'. The write operation involves *forcing* values through the 2 access transistors. This is done byb setting B and B' to the desired value, enabling WE and disabling it after the write is done. After disabling WE, the cell will retain the value written.

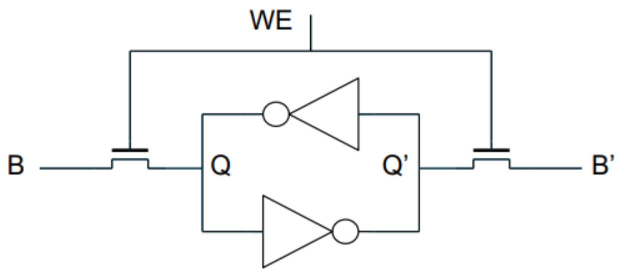


Figure 1.21: Schematic of a SRAM cell, sourced from the slides on Combinational Logic by Professor Hayden So of the University of Hong Kong

Each inverter can be implemented using two transistors; thus, the entire SRAM cell uses 6 transistors (two for each inverter and two for the access transistors). This is why SRAM is expensive and has a low density—each SRAM cell is six to ten times larger than one DRAM cell.

2. **DRAM** (Dynamic RAM):
   In DRAM data is stored in a single capacitor. Read/write access is done through a single transistor (sometimes referred to as a $1T$ cell). The read operation for DRAM is **destructive**, meaning that the data stored will be lost after read. Usually, the data is written back after read to prevent it from disappearing. This type of device also loses its data over time due to charge leakage—most modern DRAM devices have auto refresh capabilities to counter this.
   Bits are stored in two–dimensional arrays on the chip, chips having around 4 to 8 logical banks each, which can be seen on Figure 1.22.

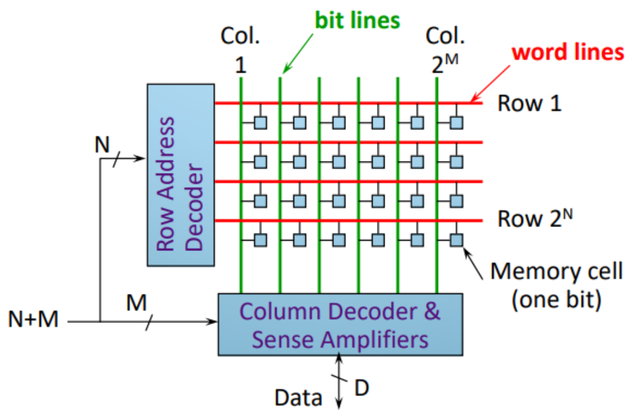


Figure 1.22: Schematic of a DRAM chip, sourced from the slides on Combinational Logic by Professor Hayden So of the University of Hong Kong

   The read operation is performed by enabling the target wordline, sensing Amps sense changes in bitline voltage, latching the results, and then selecting the desired column from the output latch. In general, DRAM has three basic operations:

   (a) Row Access: select rows depending on address. Sense amps sense very small changes in bitline level. Voltage change is small because charge stored in a small cap is shared with long bitline. Thus, sense amps restore full swing level and restore the data in the cell (refresh)
   (b) Column Access: select the desired bits out of the entire row (usually small portions of 4, 8, 16, or 32 bits). On read, the data is sent out of the package. On write, the design data is written in the sense amp latches, and the sense amp latches refresh the array with new data and the original data from unchanged bits.
   (c) Precharge: the bitline is precharged for the next operation

   In modern DRAM, each operation takes 15 to 20ns. Getting the first bit takes very long (high latency), but once the entire row of bits has been sensed, subsequent column data can be sent out of the package at high bandwidth.

In a system, DRAM is usually packaged in chips. DIMM (Dual Inline Memory Module) contains multiple chips with clock/control/address signals connected in parallel. Data pins work together to return wide words (64–bit data bus using 16×4–bit parts). DIMM is too bulky, so it is also common for the DRAM to be soldered on the back of the main board directly.