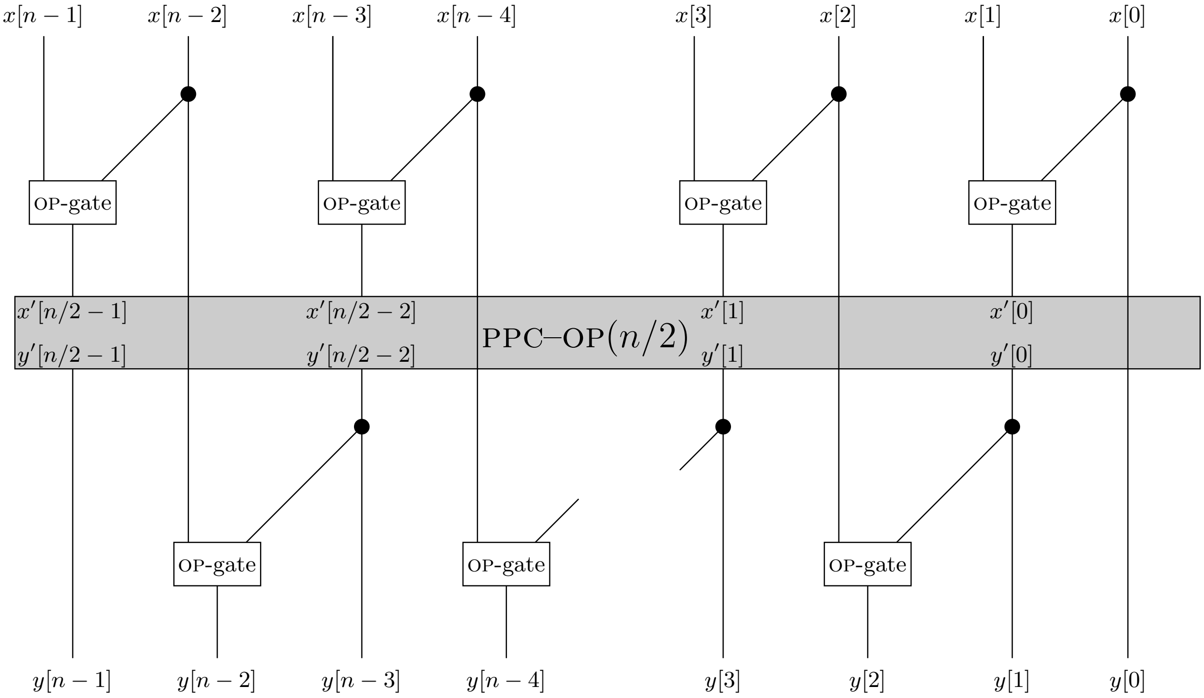
Computer Structure
Part I : combinational modules

by

Guy Even



Lecture Notes : Spring 2007
School of Electrical Engineering, Tel-Aviv University

Send comments to: *guy@eng.tau.ac.il*

Printed on February 18, 2007

Contents

Chapter 1

The digital abstraction

Contents

Preliminary Questions:

1. Can you justify or explain the saying that “computers use only zeros and ones”?
2. Can you explain the following anomaly? The design of an adder is a simple task. However, the design and analysis of a single electronic device (e.g., a single gate) is a complex task.

The term a *digital circuit* refers to a device that works in a binary world. In the binary world, the only values are zeros and ones. In other words, the inputs of a digital circuit are zeros and ones, and the outputs of a digital circuit are zeros and ones. Digital circuits are usually implemented by *electronic devices* and operate in the *real* world. In the real world, there are no zeros and ones; instead, what matters is the voltages of inputs and outputs. Since voltages refer to energy, they are continuous[1]. So we have a gap between the continuous real world and the two-valued binary world. One should not regard this gap as an absurd. Digital circuits are only an *abstraction* of electronic devices. In this chapter we explain this abstraction, called the *digital abstraction*.

In the digital abstraction one interprets voltage values as binary values. The advantages of the digital model cannot be overstated; this model enables one to focus on the digital behavior of a circuit, to ignore analog and transient phenomena, and to easily build larger more complex circuits out of small circuits. The digital model together with a simple set of rules, called *design rules*, enable logic designers to design complex digital circuits consisting of millions of gates.

1.1 Transistors

Electronic circuits that are used to build computers are mostly build of *transistors*. Small circuits, called *gates* are built from transistors. The most common technology used in VLSI chips today is called CMOS, and in this technology there are only two types of transistors: N-type and P-type. Each transistor has three connections to the outer world, called the *gate*, *source*, and *drain*. Figure 1.1 depicts diagrams describing these transistors.

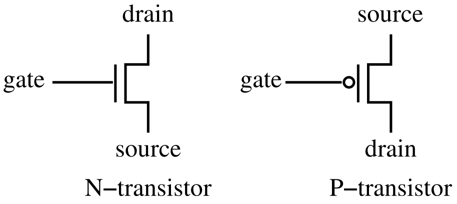


Figure 1.1: Schematic symbols of an N-transistor and P-transistor

Although inaccurate, we will refer, for the sake of simplicity, to the gate and source as inputs and to the drain as an output. An overly simple explanation of an N-type transistor in CMOS technology is as follows: If the voltage of the gate is high (i.e., above some threshold v_1), then there is little resistance between the source and the drain. Such a small resistance causes the voltage of the drain to equal the voltage of the source. If the voltage of the gate is low (i.e., below some threshold $v_0 < v_1$), then there is a very high resistance between the source and the drain. Such a high resistance means that the voltage of the drain is unchanged by the transistor (it could be changed by another transistor if the drains of the two transistors are connected). A P-type transistor is behaves in a dual manner: the resistance between drain and the source is low if the

[1] unless Quantum Physics is used.

gate voltage is below v_0. If the voltage of the gate is above v_1, then the source-to-drain resistance is very high.

Note that this description of transistor behavior implies immediately that transistors are highly non-linear. (Recall that a linear function $f(x)$ satisfies $f(a \cdot x) = a \cdot f(x)$.) In transistors, changes of 10% in input values above the threshold v_1 have a small effect on the output while changes of 10% in input values between v_0 and v_1 have a large effect on the output. In particular, this means that transistors do not follow Ohm's Law (i.e., $V = I \cdot R$).

Example 1.1 (A CMOS inverter) *Figure 1.2 depicts a CMOS inverter. If the input voltage is above v_1, then the source-to-drain resistance in the P-transistor is very high and the source-to-drain resistance in the N-transistor is very low. Since the source of the N-transistor is connected to low voltage (i.e., ground), the output of the inverter is low.*

If the input voltage is below v_0, then the source-to-drain resistance in the N-transistor is very high and the source-to-drain resistance in the P-transistor is very low. Since the source of the P-transistor is connected to high voltage, the output of the inverter is high.

We conclude that the voltage of the output is low when the input is high, and vice-versa, and the device is indeed an inverter.

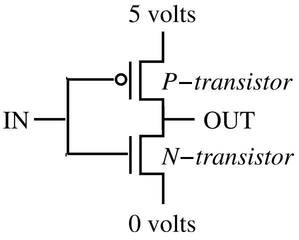


Figure 1.2: A CMOS inverter

The qualitative description in Example 1.1 hopefully conveys some intuition about how gates are built from transistors. A quantitative analysis of such an inverter requires precise modeling of the functionality of the transistors in order to derive the input-output voltage relation. One usually performs such an analysis by computer programs (e.g. SPICE). Quantitative analysis is relatively complex and inadequate for designing large systems like computers. (This would be like having to deal with the chemistry of ink when using a pen.)

1.2 From analog signals to digital signals

An *analog signal* is a real function $f : \mathbb{R} \rightarrow \mathbb{R}$ that describes the voltage of a given point in a circuit as a function of the time. We ignore the resistance and capacities of wires. Moreover, we assume that signals propagate through wires immediately[2]. Under

[2] This is a reasonable assumption if wires are short.

these assumptions, it follows that, in every moment, the voltages measured along different points of a wire are identical. Since a signal describes the voltage (i.e., derivative of energy as a function of electric charge), we also assume that a signal is a continuous function.

A *digital signal* is a function $g : \mathbb{R} \rightarrow \{0, 1, \text{non-logical}\}$. The value of a digital signal describes the *logical value* carried along a wire as a function of time. To be precise there are two logical values: zero and one. The non-logical value simply means that that the signal is neither zero or one.

How does one interpret an analog signal as a digital signal? The simplest interpretation is to set a threshold V'. Given an analog signal $f(t)$, the digital signal $dig(f(t))$ can be defined as follows.

$$dig(f(t)) \triangleq \begin{cases} 0 & \text{if } f(t) < V' \\ 1 & \text{if } f(t) > V' \end{cases} \tag{1.1}$$

According to this definition, a digital interpretation of an analog signal is always 0 or 1, and the digital interpretation is never non-logical.

There are several problems with the definition in Equation 1.1. One problem with this definition is that all the components should comply with *exactly* the same threshold V'. In reality, devices are not completely identical; the actual thresholds of different devices vary according to a tolerance specified by the manufacturer. This means that instead of a fixed threshold, we should consider a range of thresholds.

Another problem with the definition in Equation 1.1 is caused by perturbations of $f(t)$ around the threshold t. Such perturbations can be caused by *noise* or oscillations of $f(t)$ before it stabilizes. We will elaborate more on noise later, and now explain why oscillations can occur. Consider a spring connected to the ceiling with a weight w hanging from it. We expect the spring to reach a length ℓ that is proportional to the weight w. Assume that all we wish to know is whether the length ℓ is greater than a threshold ℓ_t. Sounds simple! But what if ℓ is rather close to ℓ_t? In practice, the length only tends to the length ℓ as time progresses; the actual length of the spring oscillates around ℓ with a diminishing amplitude. Hence, the length of the spring fluctuates below and above ℓ_t many times before we can decide. This effect may force us to wait for a long time before we can decide if $\ell < \ell_t$. If we return to the definition of $dig(f(t))$, it may well happen that $f(t)$ oscillates around the threshold V'. This renders the digital interpretation used in Eq. 1.1 useless.

Returning to the example of weighing weights, assume that we have two types of objects: light and heavy. The weight of a light (resp., heavy) object is at most (resp., at least) w_0 (resp., w_1). The bigger the gap $w_1 - w_0$, the easier it becomes to determine if an object is light or heavy (especially in the presence of noise or oscillations).

Now we have two reasons to introduce two threshold values instead of one, namely, different threshold values for different devices and the desire to have a gap between values interpreted as logical zero and logical one. We denote these thresholds by V_{low} and V_{high}, and require that $V_{low} < V_{high}$. An interpretation of an analog signal is depicted in Figure 1.3. Consider an analog signal $f(t)$. The digital signal $dig(f(t))$ is defined as follows.

$$dig(f(t)) \triangleq \begin{cases} 0 & \text{if } f(t) < V_{low} \\ 1 & \text{if } f(t) > V_{high} \\ \text{non-logical} & \text{otherwise.} \end{cases} \tag{1.2}$$

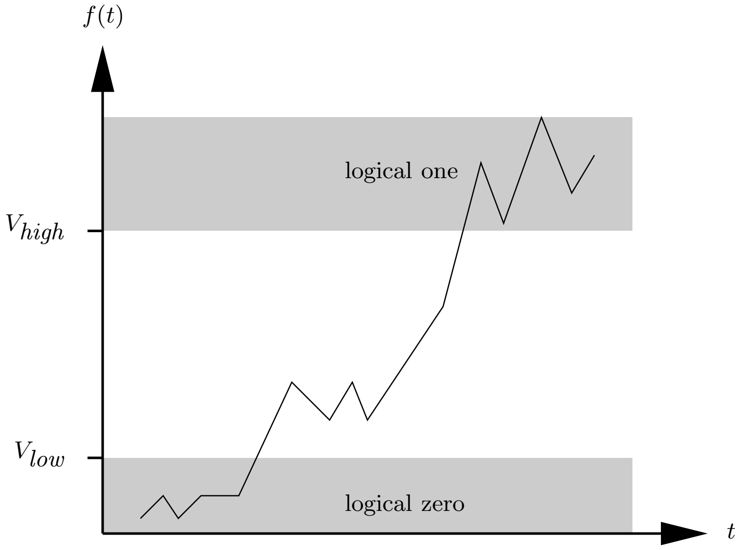


Figure 1.3: A digital interpretation of an analog signal in the zero-noise model.

We often refer to the logical value of an analog signal f. This is simply a shorthand way of referring to the value of the digital signal $dig(f)$.

It is important to note that fluctuations of $f(t)$ are still possible around the threshold values. However, if the two thresholds are sufficiently far away from each other, fluctuations of f do not cause fluctuations of $dig(f(t))$ between 0 and 1. Instead, we will have at worst fluctuations of $dig(f(t))$ between a non-logical value and a logical value (i.e., 0 or 1). A fluctuation between a logical value and a non-logical value is much more favorable than a fluctuation between 0 and 1. The reason is that a non-logical value is an indication that the circuit is still in a transient state and a "decision" has not been reached yet.

Assume that we design an inverter so that its output tends to a voltage that is bounded away from the thresholds V_{low} and V_{high}. Let us return to the example of the spring with weight w hanging from it. Additional fluctuations in the length of the spring might be caused by wind. This means that we need to consider additional effects so that our model will be useful. In the case of the digital abstraction, we need to take *noise* into account. Before we consider the effect of noise, we formulate the static functionality of a gate, namely, the values of its output as a function of its stable inputs.

Question 1.1 *Try to define an inverter in terms of the voltage of the output as a function of the voltage of the input.*

1.3 Transfer functions of gates

The voltage at an output of a gate depends on the voltages of the inputs of the gate. This dependence is called the *transfer function* (or the *voltage-transfer characteristic* - VTC). Consider, for example an inverter with an input x and an output y. To make things complicated, the value of the signal $y(t)$ at time t is not only a function of the signal x at time t since $y(t)$ depends on the history. Namely, $y(t_0)$ is a function of $x(t)$ over the interval $(-\infty, t_0]$.

Partial differential equations are used to model gates, and the solution of these equations is unfortunately a rather complicated task. A good approximation of transfer functions is obtain by solving differential equations, still a complicated task that can be computed quickly only for a few transistors. So how are chips that contain millions of chips designed if the models are too complex to be solved?

The way this very intricate problem is handled is by restricting designs. In particular, only a small set of building blocks is used. The building blocks are analyzed intensively, their properties are summarized, and designers rely on these properties for their designs.

One of the most important steps in characterizing the behavior of a gate is computing its *static transfer function*. Returning to the example of the inverter, a "proper" inverter has a unique output value for each input value. Namely, if the input $x(t)$ is stable for a sufficiently long period of time and equals x_0, then the output $y(t)$ stabilizes on a value y_0 that is a function of x_0.

Before we define what a static transfer function is, we point out that there are devices that do not have static transfer functions. We need to distinguish between two cases: (a) Stability is not reached: this case occurs, for example, with devices called oscillators. Note that oscillating devices must consume energy even when the input is stable. We point out that in CMOS technology it is easy to design circuits that do not consume energy if the input is logical, so such oscillations are avoided. (b) Stability is reached: in this case, if there is more than one stable output value, it means that the device has more than one equilibrium point. Such a device can be used to store information about the "history". It is important to note that devices with multiple equilibriums are very useful as storage devices (i.e., they can "remember" a small amount of information). Nevertheless, devices with multiple equilibriums are not "good" candidates for gates, and it is easy to avoid such devices in CMOS technology.

Example 1.2 (A device with many equillibriums) *Consider a pot that is initially filled with water. At time t, the pot is held in angle $x(t)$. A zero angle means that the pot is held upright. An angle of $180°$ means that the pot is upside down. Now, we are told that $x(t) = 0°$ for $t \geq 100$. Can we say how much water is contained in the pot at time $t = 200$? The answer, of course, depends on the history during the interval $t \in [0, 100)$, namely, whether the pot was tilted.*

We formalize the definition of a static transfer function of a gate G with one input x and one output y in the following definition.

Definition 1.1 *Consider a device G with one input x and one output y. The device G is a* gate *if its functionality is specified by a function $f : \mathbb{R} \rightarrow \mathbb{R}$ as follows: there exists a $\Delta > 0$, such that, for every x_0 and every t_0, if $x(t) = x_0$ for every $t \in [t_0 - \Delta, t_0]$, then $y(t_0) = f(x_0)$.*
Such a function $f(x)$ is called the static transfer function *of G.*

At this point we should point the following remarks:

1. Since circuits operate over a bounded range of voltages, static transfer functions are usually only defined over bounded domains and ranges (say, $[0, 5]$ volts).

2. To make the definition useful, one should allow perturbations of $x(t)$ during the interval $[t_0 - \Delta, t_0]$. Static transfer functions model physical devices, and hence, are continuous. This implies the following definition: For every $\epsilon > 0$, there exist a $\delta > 0$ and a $\Delta > 0$, such that

$$\forall t \in [t_1, t_2] \ : \ |x(t) - x_0| \leq \delta \quad \Rightarrow \quad \forall t \in [t_1 + \Delta, t_2] \ : \ |y(t) - f(x_0)| \leq \epsilon.$$

3. Note that in the above definition Δ does not depend on x_0 (although it may depend on ϵ). Typically, we are interested on the values of Δ only for logical values of $x(t)$ (i.e., $x(t) \leq V_{low}$ and $x(t) \geq V_{high}$). Once the value of ϵ is fixed, this constant Δ is called the *propagation delay* of the gate G and is one of the most important characteristic of a gate.

Question 1.2 *Extend Definition 1.1 to gates with n inputs and m outputs.*

Finally, we can now define an inverter in the zero-noise model. Observe that according to this definition a device is an inverter if its static transfer function satisfies a certain property.

Definition 1.2 (inverter in zero-noise model) *A gate G with a single input x and a single output y is an inverter if its static transfer function $f(z)$ satisfies the following the following two conditions:*

1. *If $z < V_{\text{low}}$, then $f(z) > V_{\text{high}}$.*

2. *If $z > V_{\text{high}}$, then $f(z) < V_{\text{low}}$.*

The implication of this definition is that if the logical value of the input x is zero (resp., one) during an interval $[t_1, t_2]$ of length at least Δ, then the logical value of the output y is one (resp., zero) during the interval $[t_1 + \Delta, t_2]$.

How should we define other gates such a NAND-gates, XOR-gates, etc.? As in the definition of an inverter, the definition of a NAND-gate is simply a property of its static transfer function.

Question 1.3 *Define a* NAND*-gate.*

We are now ready to strengthen the digital abstraction so that it will be useful also in the presence of bounded noise.

1.4 The bounded-noise model

Consider a wire from point A to point B. Let $A(t)$ denote the analog signal measured at point A. Similarly, let $B(t)$ denote the analog signal measured at point B. We would like to assume that wires have zero resistance, zero capacitance, and that signals propagate through a wire with zero delay. This assumption means that the signals $A(t)$ and $B(t)$ should be equal at all times. Unfortunately, this is not the case; the main reason for this discrepancy is *noise*.

There are many sources of noise. The main source of noise is heat that causes electrons to move randomly. These random movements do not cancel out perfectly, and random currents are created. These random currents create perturbations in the voltage. The difference between the signals $B(t)$ and $A(t)$ is a *noise signal*.

Consider, for example, the setting of *additive noise*: A is an output of an inverter and B is an input of another inverter. We consider the signal $A(t)$ to be a reference signal. The signal $B(t)$ is the sum $A(t) + n_B(t)$, where $n_B(t)$ is the noise signal.

The *bounded-noise model* assumes that the noise signal along every wire has a bounded absolute value. We will use a slightly simplified model in which there is a constant $\epsilon > 0$ such that the absolute value of all noise signals is bounded by ϵ. We refer to this model as the *uniformly bounded noise model*. The justification for assuming that noise is bounded is probabilistic. Noise is a random variable whose distribution has a rapidly diminishing tail. This means that if the bound is sufficiently large, then the probability of the noise exceeding this bound during the lifetime of a circuit is negligibly small.

1.5 The digital abstraction in presence of noise

Consider two inverters, where the output of one gate feeds the input of the second gate. Figure 1.4 depicts such a circuit that consists of two inverters.

Assume that the input x has a value that satisfies: (a) $x > V_{high}$, so the logical value of x is one, and (b) $y = V_{low} - \epsilon'$, for a very small $\epsilon' > 0$. This might not be possible with every inverter, but Definition 1.2 does not rule out such an inverter. (Consider a transfer function with $f(V_{high}) = V_{low}$, and x slightly higher than V_{high}.) Since the logical value of y is zero, it follows that the second inverter, if not faulty, should output a value z that is greater than V_{high}. In other words, we expect the logical value of z to be 1. At this point we consider the effect of adding noise.

Let us denote the noise added to the wire y by n_y. This means that the input of the second inverter equals $y(t) + n_y(t)$. Now, if $n_y(t) > \epsilon'$, then the second inverter is fed a non-logical value! This means that we can no longer deduce that the logical value of z is one. We conclude that we must use a more resilient model; in particular, the functionality of circuits should not be affected by noise. Of course, we can only hope to be able to cope with bounded noise, namely noise whose absolute value does not exceed a certain value ϵ.

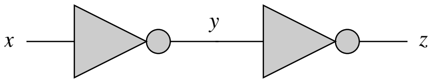


Figure 1.4: Two inverters connected in series.

1.5.1 Input and output signals

Definition 1.3 *An* input signal *is a signal that is fed to a circuit or to a gate. An* output signal *is a signal that is output by a gate or a circuit.*

For example, in Figure 1.4 the signal y is both the output signal of the left inverter and an input signal of the right inverter. If noise is not present and there is no delay, then the signal output by the left inverter always equals the signal input to the right inverter.

1.5.2 Redefining the digital interpretation of analog signals

The way we deal with noise is that we interpret input signals and output signals differently. An input signal is a signal measured at an input of a gate. Similarly, an output signal is a signal measured at an output of a gate. Instead of two thresholds, V_{low} and V_{high}, we define the following four thresholds:

- $V_{low,in}$ - an upper bound on a voltage of an input signal interpreted as a logical zero.
- $V_{low,out}$ - an upper bound on a voltage of an output signal interpreted as a logical zero.
- $V_{high,in}$ - a lower bound on a voltage of an input signal interpreted as a logical one.
- $V_{high,out}$ - a lower bound on a voltage of an output signal interpreted as a logical one.

These four thresholds satisfy the following equation:

$$V_{low,out} < V_{low,in} < V_{high,in} < V_{high,out}. \tag{1.3}$$

Figure 1.5 depicts these four thresholds. Note that the interpretation of input signals is less strict than the interpretation of output signals. The actual values of these four thresholds depend on the transfer functions of the devices we wish to use.
Consider an input signal $f_{in}(t)$. The digital signal $dig(f_{in}(t))$ is defined as follows.

$$dig(f_{in}(t)) \triangleq \begin{cases} 0 & \text{if } f_{in}(t) < V_{low,in} \\ 1 & \text{if } f_{in}(t) > V_{high,in} \\ \text{non-logical} & \text{otherwise.} \end{cases} \tag{1.4}$$

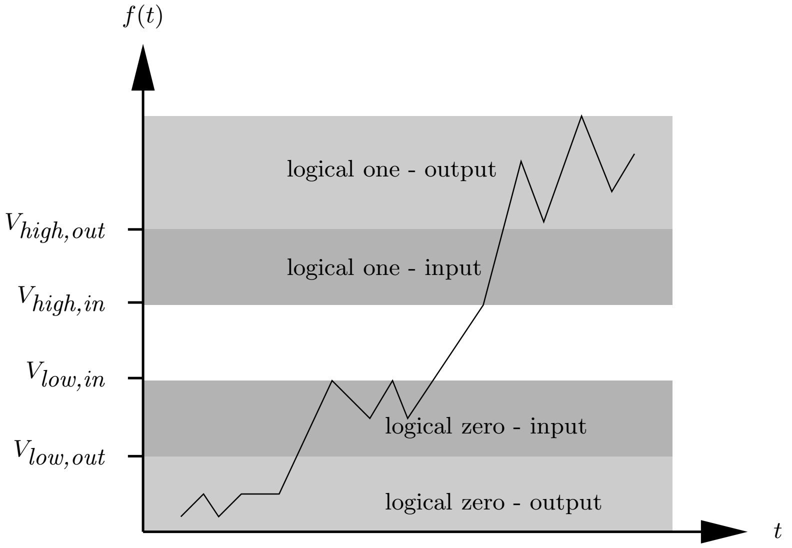


Figure 1.5: A digital interpretation of an input and output signals.

Consider an output signal $f_{out}(t)$. The digital signal $dig(f_{out}(t))$ is defined analogously.

$$dig(f_{out}(t)) \triangleq \begin{cases} 0 & \text{if } f_{out}(t) < V_{low,out} \\ 1 & \text{if } f_{out}(t) > V_{high,out} \\ \text{non-logical} & \text{otherwise.} \end{cases} \tag{1.5}$$

The differences $V_{low,in} - V_{low,out}$ and $V_{high,out} - V_{high,in}$ are called *noise margins.* Our goal is to show that if the absolute value of the noise is less than the noise margins, and if the digital value of an output signal is either zero or one, then the logical value of the corresponding input signal equals that of the output signal. In other words, if $|n(t)|$ is less than the noise margins, and $dig(f_{out})(t) \in \{0,1\}$, then $dig(f_{in})(t) = dig(f_{out})(t)$. Indeed, if the absolute value of the noise $n(t)$ is bounded by the noise margins, then an output signal $f_{out}(t)$ that is below $V_{low,out}$ will result with an input signal $f_{in}(t) = f_{out}(t) + n(t)$ that does not exceed $V_{low,in}$.

We can now fix the definition of an inverter so that bounded noise added to outputs, does not affect the logical interpretation of signals.

Definition 1.4 (inverter in the bounded-noise model) *A gate G with a single input x and a single output y is an inverter if its static transfer function $f(z)$ satisfies the following the following two conditions:*

1. *If $z < V_{\text{low,in}}$, then $f(z) > V_{\text{high,out}}$.*

2. *If $z > V_{\text{high,in}}$, then $f(z) < V_{\text{low,out}}$.*

Question 1.4 *Define a* NAND*-gate in the bounded-noise model.*

Question 1.5 *Consider the following piecewise linear function:*

$$f(x) = \begin{cases} 5 & \text{if } x \leq \frac{5}{3} \\ 0 & \text{if } x \geq \frac{10}{3} \\ -3x + 10 & \text{if } \frac{5}{3} < x < \frac{10}{3}. \end{cases}$$

Show that if $f(x)$ is the transfer function of a device C, then one can define threshold values $V_{\text{low,out}} < V_{\text{low,in}} < V_{\text{high,in}} < V_{\text{high,out}}$ so that C is an inverter according to Definition 1.4.

Question 1.6 *Consider the function $f(x) = 1 - x$ over the interval $[0, 1]$. Suppose that $f(x)$ is a the transfer function of a device C. Can you define threshold values $V_{\text{low,out}} < V_{\text{low,in}} < V_{\text{high,in}} < V_{\text{high,out}}$ so that C is an inverter according to Definition 1.4?*

Hint: Prove that $V_{\text{high,out}} \leq 1 - V_{\text{low,in}}$ and that $V_{\text{low,out}} \geq 1 - V_{\text{high,in}}$. Derive a contradiction from these two inequalities.

Question 1.7 *Consider a function $f : [0, 1] \to [0, 1]$. Suppose that: (i) $f(0) = 1$, and $f(1) = 0$, (ii) $f(x)$ is monotone decreasing, (iii) the derivative $f'(x)$ of $f(x)$ satisfies the following conditions: $f'(x)$ is continuous and there is an interval (α, β) such that $f'(x) < -1$ for every $x \in (\alpha, \beta)$. And, (iv) there exists a point $x_0 \in (\alpha, \beta)$ such that $f(x_0) = x_0$.*

Prove that one can define threshold values $V_{\text{low,out}} < V_{\text{low,in}} < V_{\text{high,in}} < V_{\text{high,out}}$ so that C is an inverter according to Definition 1.4.

Hints: (a) The derivative $f'(x)$ is continuous. Hence, there exists constants $c < -1$ and $\delta > 0$ such that, in the interval $(x_0 - \delta, x_0 + \delta)$, the derivate $f'(x)$ is less than or equal to c. (b) Set $V_{\text{low,in}} = x_0 - \delta$ and $V_{\text{high,in}} = x_0 + \delta$.

Question 1.8 ** Try to characterize transfer functions $g(x)$ that correspond to inverters. Namely, if C_g is a device, the transfer function of which equals $g(x)$, then one can define threshold values that satisfy Definition 1.4.*

1.6 Stable signals

In this section we define terminology that will be used later. To simplify notation we define these terms in the zero-noise model. We leave it to the curious reader to extend the definitions and notation below to the bounded-noise model.

An analog signal $f(t)$ is said to be *logical at time t* if *dig*$(f(t)) \in \{0, 1\}$. An analog signal $f(t)$ is said to be *stable* during the interval $[t_1, t_2]$ if $f(t)$ is logical for every $t \in [t_1, t_2]$. Continuity of $f(t)$ and the fact that $V_{low} < V_{high}$ imply the following claim.

Claim 1.1 *If an analog signal $f(t)$ is stable during the interval $[t_1, t_2]$, then one of the following holds:*

1. $\text{dig}(f(t)) = 0$*, for every* $t \in [t_1, t_2]$*, or*

2. $\mathrm{dig}(f(t)) = 1$*, for every* $t \in [t_1, t_2]$.

From this point we will deal with digital signals and use the same terminology. Namely, a digital signal $x(t)$ is *logical* at time t if $x(t) \in \{0, 1\}$. A digital signal is *stable* during an interval $[t_1, t_2]$ if $x(t)$ is logical for every $t \in [t_1, t_2]$.

1.7 Summary

In this chapter we presented the digital abstraction of analog devices. For this purpose we defined analog signals and their digital counterpart, called digital signals. In the digital abstraction, analog signals are interpreted either as zero, one, or non-logical.

We discussed noise and showed that to make the model useful, one should set stricter requirements from output signals than from input signals. Our discussion is based on the bounded-noise model in which there is an upper bound on the absolute value of noise.

We defined gates using transfer functions and static transfer functions. This functions describe the analog behavior of devices. We also defined the propagation delay of a device as the amount of time that input signals must be stable to guarantee stability of the output of a gate.

Chapter 2

Foundations of combinational circuits

Contents

Preliminary questions:

1. Does every collection of gates and wires constitute a combinational circuit?
2. Which of these tasks is easy?
   (a) Check if a circuit is combinational.
   (b) Simulate a combinational circuit.
   (c) Estimate the propagation delay of a combinational circuit.
3. Suggest criteria for comparing functionally equivalent combinational circuits.

In this chapter we define and study combinational circuits. Our goal is to prove two theorems: (i) Every Boolean function can be implemented by a combinational circuit, and (ii) every combinational circuit implements a Boolean function.

2.1 Boolean functions

Let $\{0,1\}^n$ denote the set of n-bit strings. A Boolean function is defined as follows.

Definition 2.1 *A function $B : \{0,1\}^n \rightarrow \{0,1\}^k$ is called a* Boolean function.

There are many ways to represent Boolean functions. The simplest way is by a table. The table for a function $B : \{0,1\}^n \rightarrow \{0,1\}^k$ has 2^n entries, one for every binary string $\alpha \in \{0,1\}^n$. Each entry of the table contains a binary string $\beta \in \{0,1\}^k$, where the entry corresponding to α contains the string $B(\alpha)$. The disadvantage of a table is that some Boolean functions can be represented much more efficiently. By efficiency we mean that the Boolean function can be represented by fewer bits. The table we just described requires 2^{n+k} bits.

Consider, for example, the Boolean XOR_n function (i.e., $\text{XOR}_n(\alpha) = 1$ if and only if the number of ones in α is odd). One could represent the XOR_n function more efficiently using a Boolean formula. The length of the formula is n, which is a huge improvement over the length of the table.

In this chapter, our main focus is on the representation of Boolean functions by combinational circuits, a representation that is different from tables and formulas.

2.2 Combinational gates - an analog approach

By Definition 1.1, a gate is a device whose static functionality is specified by a static transfer function. This means that the output is a function of the inputs, provided that the input values do not change for a sufficiently long amount of time.

Our goal now is to define combinational gates. The difference between a gate and a combinational gate is that we require that if the inputs are stable (and, in particular, logical), then the output is not only well defined but also logical. Hence, not only is the output a function of the present value of the inputs - the output is logical if the inputs are stable. We now formalize the definition of a combinational gate.

First, we extend the definition of the digital interpretation of an analog signal to real vectors. Let $\vec{y} \in \mathbb{R}^n$, where $\vec{y} = (y_1, y_2, \cdots, y_n)$. The function $dig_n : \mathbb{R}^n \rightarrow \{0, 1, \text{non-logical}\}^n$ is defined by

$$dig_n(y_1, y_2, \cdots, y_n) \triangleq (dig(y_1), dig(y_2), \cdots, dig((y_n))).$$

To simplify notation, we denote dig_n simply by dig when the length n of the vector is clear.

Recall that by Def. 1.1, a gate is a device that functions according to a static transfer function. We now define a combinational gate.

Definition 2.2 *Consider a gate G with n inputs (denoted by $\vec{x}$) and k outputs (denoted by $\vec{y}$). The gate G is a* combinational gate *if there exists a $\Delta > 0$, such that, for all $\vec{x}(t) \in \mathbb{R}^n$,*

$$\forall t \in [t_1, t_2]: \ \text{dig}(\vec{x(t)}) \in \{0,1\}^n \quad \Rightarrow \quad \forall t \in [t_1 + \Delta, t_2]: \ \text{dig}(y(t)) \in \{0,1\}^k. \tag{2.1}$$

The above definition requires that a stable input leads to a stable output. Note that this definition is stricter than the definition of a gate in two ways. First, we require that the static transfer function $f : \mathbb{R}^n \rightarrow \mathbb{R}^k$ satisfy

$$\forall \vec{x} : \mathit{dig}(\vec{x}) \in \{0,1\}^n \Rightarrow \mathit{dig}(f(\vec{x})) \in \{0,1\}^k. \tag{2.2}$$

Second, we allow the input $\vec{x}(t)$ to fluctuate as much as it wishes, along as it is logically stable (i.e., each component must have the same logical value during the interval $[t_1, t_2]$, but its analog value may fluctuate within the intervals $[0, V_{low,in}]$ and $[V_{high,in}, +\infty]$).

Consider a combinational gate G and let $f : \mathbb{R}^n \rightarrow \mathbb{R}^k$ denote its static transfer function. The function f induces a Boolean function $B_f : \{0,1\}^n \rightarrow \{0,1\}^k$ as follows. Given a Boolean vector $(b_1, \cdots, b_n) \in \{0,1\}^n$, define x_i as follows:

$$x_i \triangleq \begin{cases} V_{low} - \varepsilon & \text{if } b_i = 0 \\ V_{high} + \varepsilon & \text{if } b_i = 1. \end{cases}$$

The Boolean function B_f is defined by

$$B_f(\vec{b}) \triangleq \mathit{dig}(f(\vec{x})).$$

Since G is a combinational gate, it follows that every component of $\mathit{dig}(f(\vec{x}))$ is logical, and hence B_f is a Boolean function, as required.

After defining the Boolean function B_f, we can rephrase Equation 2.2 as follows:

$$\mathit{dig}(\vec{x}) \in \{0,1\}^n \Rightarrow \mathit{dig}(f(\vec{x})) = B_f(\mathit{dig}(\vec{x})).$$

The discussion so far proves the following claim.

Claim 2.1 *In a combinational gate, the relation between the logical values of the inputs and the logical values of the outputs is specified by a Boolean function.*

Recall that the propagation delay is an upper bound on the amount of time that elapses from the moment that the inputs (nearly) stop changing till the moment that the output (nearly) equals the value of the static transfer function. Hence, one must allow some time till the logical values of the outputs of a combinational gate properly reflect the value of the Boolean function. We say that a combinational gate is *consistent* if this relation holds. Formally,

Definition 2.3 *A combinational gate G with inputs $\vec{x}(t)$ and outputs $\vec{y}(t)$ is* consistent *at time t if* $\text{dig}(\vec{x}(t)) \in \{0,1\}^n$ *and* $\text{dig}(\vec{y}(t)) = B_f(\text{dig}(\vec{x}(t)))$.

2.3 Back to the digital world

In the previous section we defined combinational gates using analog signals and their digital interpretation. This approach is useful when one wishes to determine if an analog device can be used as a digital combinational gate. Here we change the approach and avoid reference to analog signals.

To simplify notation, we consider a combinational gate G with 2 inputs, denoted by x_1, x_2, and a single output, denoted by y. Instead of using analog signals, we refer only to digital signals. Namely, we denote the digital signal at terminal x_1 by $x_1(t)$. The same notation is used for the other terminals.

Our goals are to: (i) specify the functionality of combinational gate G by a Boolean function, (ii) define when a combinational gate G is consistent, and (iii) define the propagation delay of G.

We use a looser definition of the propagation delay. Recall that we decided to refer only to digital signals. Hence, we are not sensitive to the analog value of the signals. This means that a (logically) stable signal is considered to have a fixed value, and the analog values of inputs may change as long as they remain with the same logical value.

In the looser definition of propagation delay we only ask about the time that elapses from the moment the inputs are stable till the gate is consistent.

Definition 2.4 *A combinational gate G is* consistent with a Boolean function B at time t *if the input values are logical at time t and*

$$y(t) = B(x_1(t), x_2(t)).$$

Note that $y(t)$ must be also logical since $x_1(t), x_2(t) \in \{0,1\}$ and B is a Boolean function.

The following definition defines when a combinational gate implements a Boolean function with propagation delay t_{pd}.

Definition 2.5 *A combinational gate G implements a Boolean function $B : \{0,1\}^2 \rightarrow \{0,1\}$ with propagation delay t_{pd} if the following holds.*

For every $\sigma_1, \sigma_2 \in \{0,1\}$, if $x_i(t) = \sigma_i$, for $i = 1,2$, during the interval $[t_1, t_2]$, then

$$\forall t \in [t_1 + t_{pd}, t_2] \ : \ y(t) = B(\sigma_1, \sigma_2).$$

The following remarks should be well understood before we continue:

1. The above definition can be stated in a more compact form. Namely, a combinational gate G implements a Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ with propagation delay t_{pd} if stability of the inputs of G in the interval $[t_1, t_2]$ implies that the combinational gate G is consistent with f in the interval $[t_1 + t_{pd}, t_2]$.

2. If $t_2 < t_1 + t_{pd}$, then the statement in the above definition is empty. It follows that the inputs of a combinational gate must be stable for at least a period of t_{pd}, otherwise, the combinational gate need not reach consistency.

3. Note that the propagation delay is an upper bound on the amount of time that elapses till a combinational gate becomes consistent (provided that its inputs are stable). The actual amount of time that passes till a combinational gate is consistent is very hard to compute, and in fact it is random. It depends on $x(t)$ during the interval $(-\infty, t)$ (i.e., how fast does the input change?), noise, and manufacturing variance. This is why upper bounds are used for propagation delays rather than the actual times.

   Suppose that a combinational gate G implements a Boolean function $B : \{0,1\}^n \rightarrow \{0,1\}$ with propagation delay t_{pd}. Assume that $t' \geq t_{pd}$. Then G also implements the Boolean function $B(x)$ with propagation delay t'. It is legitimate to use upper bounds on the actual propagation delay, and pessimistic assumptions should not render a circuit incorrect. Timing analysis, on the the other hand, depends on the upper bounds we use; the tighter the bounds, the more accurate the timing analysis is.

Assume that the combinational gate G is consistent at time t_2, and that at least one input is not stable in the interval (t_2, t_3). We can not assume that the output of G remains stable after t_2. However, in practice, an output may remain unchanged for a short while after an input becomes instable. We formalize this as follows.

Definition 2.6 *The* contamination delay *of a combinational device is a lower bound on the amount of time that the output of a consistent gate remains stable after its inputs stop being stable.*

Throughout this course, unless stated otherwise, we will make the most “pessimistic” assumption about the contamination delay. Namely, we do not rely on an output remaining stable after an input becomes instable. Formally, we will assume that the contamination delay is zero.

Figure 2.1 depicts the propagation delay and the contamination delay. The outputs become stable at most t_{pd} time units after the inputs become stable. The outputs remain stable at least t_{cont} time units after the inputs become instable.

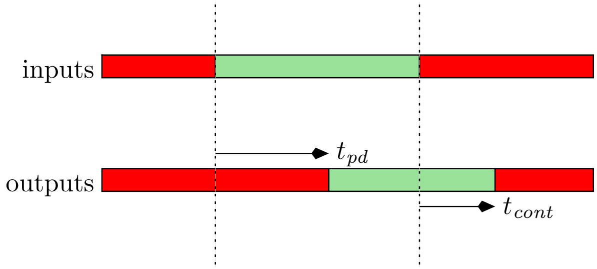


Figure 2.1: The propagation delay and contamination delay of a combinational gate. The x-axis corresponds to time. The dark (or red) segments signify that the signal is not guaranteed to be logical; the light (or green) segments signify that the signal is guaranteed to be stable.

Example 2.1 (Timing analysis & inferring output values based on partial inputs) *Consider an* AND*-gate with inputs* $x_1(t)$ *and* $x_2(t)$ *and an output* $y(t)$*. Suppose that the propagation delay of the gate is* $t_{\text{pd}} = 2$ *seconds. (All time units are in seconds in this example, so units will not be mentioned anymore in this example).*

- *Assume that the inputs equal* 1 *during the interval* $[100, 109]$ *. Since* $t_{pd} = 2$*, it follows that* $y(t) = 1$ *during the interval* $[102, 109]$*. It may very well happen that* $y(t) = 1$ *before* $t = 102$*, however, we are not certain that this happens. During the interval* $[100, 102)$*, we are uncertain about the value of* $y(t)$*; it may be* 0*,* 1*, or non-logical, and it may fluctuate arbitrarily between these values.*

- *Assume that* $x_1(t) = 1$ *during the interval* $[109, 115]$*,* $x_2(t) =$ *non-logical during the interval* $(109, 110)$*, and* $x_2(t) = 0$ *during the interval* $[110, 115]$*.*

  During the interval $(109, 110)$ *we know nothing about the value of the output* $y(t)$ *since* $x_2(t)$ *is non-logical. The inputs are stable again starting* $t = 110$*. Since* $t_{pd} = 2$*, we are only sure about the value of* $y(t)$ *during the interval* $[112, 115]$ *(during the interval* $[112, 115]$*,* $y(t) = 0$*). Hence, we are uncertain about the value of* $y(t)$ *during the interval* $(109, 112)$*.*

- *Assume that* $x_2(t)$ *remains stable during the interval* $[110, 120]$*,* $x_1(t)$ *becomes non-logical during the interval* $(115, 116)$*, and* $x_1(t)$ *equals* 1 *again during the interval* $[116, 120]$*.*

  Since $x_2(t)$ *is stable during the interval* $[110, 120]$*, we conclude that it equals* 0 *during this interval. The truth-table of an* AND*-gate implies that if one input is zero, then the output is zero. Can we conclude that that* $y(t) = 0$ *during the interval* $[110, 120]$*?*

  There are some technologies in which we could draw such a conclusion. However, our formalism does not imply this at all! As soon as $x_1(t)$ *becomes non-logical (after* $t = 115$*), we cannot conclude anything about the value of* $y(t)$*. We remain uncertain for two seconds after both inputs stabilize. Both inputs stabilize at* $t = 116$*. Therefore, we can only conclude that* $y(t) = 0$ *during the interval* $[118, 120]$*.*

  The inability to determine the value of $y(t)$ *during the interval* $(115, 118)$ *is a shortcoming of our formalism. For example, in a CMOS* NAND*-gate, one can determine that the output is zero if one of the outputs is one (even if the other input is non-logical). The problem with using such deductions is that timing becomes dependent on the values of the signals. On one hand, this improves the estimates computed by timing analysis. One the other hand, timing analysis becomes a very hard computational problem. In particular, instead of a task that can be computed in linear time, it becomes an NP-hard task (i.e., a task that is unlikely to be solvable in polynomial time).*

2.4 Building blocks

The building blocks of combinational circuits are combinational gates and wires. In fact, we will need to consider *nets* that are generalizations of wires.

Combinational gates or gates. A combinational gate, as defined in Definition 2.5 is a device that implements a Boolean function. We refer to a combinational gate, in short, as a gate.

The inputs and outputs of a gate are often referred to as *terminals*, *ports*, or even *pins*. The *fan-in* of a gate G is the number of input terminals of G (i.e., the number of bits in the domain of the Boolean function that specifies the functionality of G). The fan-in of the basic gates that we will be using as building blocks for combinational circuits is constant (i.e., at most $2 - 3$ input ports). The basic gates that we consider are: inverter (NOT-gate), OR-gate, NOR-gate, AND-gate, NAND-gate, XOR-gate, NXOR-gate, multiplexer (MUX).

The input ports of a gate G are denoted by the set $\{in(G)_i\}_{i=1}^{n}$, where n denotes the fan-in of G. The output ports of a gate G are denoted by the set $\{out(G)_i\}_{i=1}^{k}$, where k denotes the number of output ports of G.

Wires and nets. A wire is a connection between two terminals (e.g., an output of one gate and an input of another gate). In the zero-noise model, the signals at both ends of a wire are identical.

Very often we need to connect several terminals (i.e., inputs and outputs of gates) together. We could, of course, use any set of edges (i.e., wires) that connects these terminals together. Instead of specifying how the terminals are physically connected together, we use nets.

Definition 2.7 *A* net *is a subset of terminals that are connected by wires.*

In the digital abstraction we assume that the signals all over a net are identical (why?). The *fan-out* of a net N is the number of input terminals that are contained in N.

The issue of drawing nets is a bit confusing. Figure 2.2 depicts three different drawings of the same net. All three nets contain an output terminal of an inverter and 4 input terminals of inverters. However, the nets are drawn differently. Recall that the definition of a net is simply a subset of terminals. We may draw a net in any way that we find convenient or aesthetic. The interpretation of the drawing is that terminals that are connected by lines or curves constitute a net.

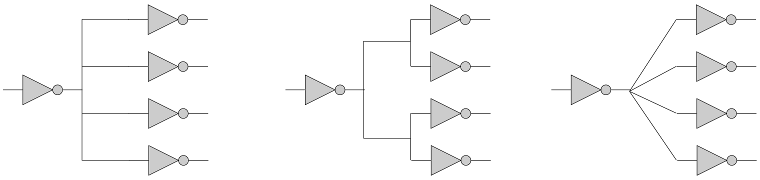


Figure 2.2: Three equivalent nets.

Consider a net N. We would like to define the digital signal $N(t)$ for the whole net. The problem is that due to noise (and other reasons) the analog signals at different terminals of the net might not equal each other. This might cause the digital interpretations of analog signals at different terminals of the net to be different, too. We solve this problem by defining $N(t)$ to logical only if there is a consensus among all the digital interpretations of the analog signals at all the terminals of the net. Namely, $N(t)$ is zero (respectively, one) if the digital values of all the analog signals in the net are zero (respectively, one). If there is no consensus, then $N(t)$ is non-logical. Recall that, in the bounded-noise model, different thresholds are used to interpret the digital values of the analog signals measured in input and output terminals.

We say that a net N *feeds* an input terminal t if the input terminal t is in N. We say that a net N is *fed* by an output terminal t if t is in N. Figure 2.3 depicts an output terminal that feeds a net and an input terminal that is fed by a net. The notion of feeding and being fed implies a direction according to which information "flows"; namely, information is "supplied" by output terminals and is "consumed" by input terminals. Direction of signals along nets is obtained in "pure" CMOS gates as follows. Output terminals are fed via resistors either by the ground or by the power. Input terminals, one the other hand, are connected only to capacitors.

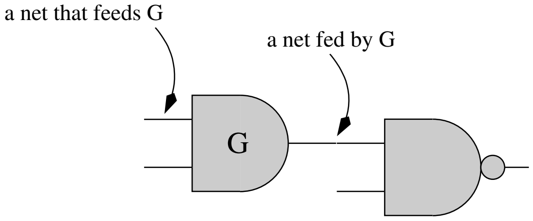


Figure 2.3: A terminal that is fed by a net and a terminal that feeds a net.

The following definition captures the type of nets we would like to use. We call these nets *simple*.

Definition 2.8 *A net N is* simple *if (i) N is fed by exactly one output terminal, and (ii) N feeds at least one input terminal.*

A simple net N that is fed by the output terminal t and feeds the input terminals $\{t_i\}_{i\in I}$, can be modeled by wires $\{w_i\}_{i\in I}$. Each wire w_i connects t and t_i. In fact, since information flows in one direction, we may regard each wire w_i as a directed edge $t \rightarrow t_i$.

It follows that a circuit, all the nets of which are simple, may be modeled by a directed graph. We define this graph in the following definition.

Definition 2.9 *Let C denote a circuit, all the nets of which are simple. The directed graph $DG(C)$ is defined as follows. The vertices of the graph $DG(C)$ are the gates of C. The directed edges correspond to the wires as follows. Consider a simple net N fed by an output terminal t that feeds the input terminals $\{t_i\}_{i\in I}$. The directed edges that correspond to N are $u \rightarrow v_i$, where u is the gate that contains the output terminal t and v_i is the gate that contains the input terminal t_i.*

Note that the information regarding which terminal is connected to each wire is not maintained in the graph $DG(C)$. One could of course label each endpoint of an edge in $DG(C)$ with the name of the terminal the edge is connected to.

2.5 Combinational circuits

Question 2.1 *Consider the circuits depicted in Figure 2.4. Can you explain why these are not valid combinational circuits?*

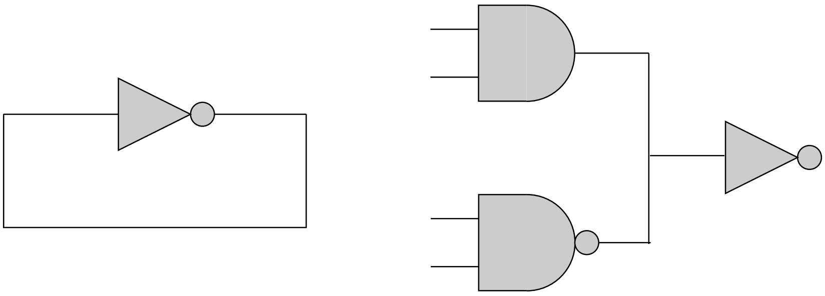


Figure 2.4: Two examples of non-combinational circuits.

Before we define combinational circuits, it is helpful to define two types of special gates: an input gate and an output gate. The purpose of these gates is to avoid endpoints in nets that seem to be not connected. (For example, consider the the circuit on the right side in Figure 2.4. Every net in this circuit has one endpoint that is connected to a gate and one endpoint that "hangs" without a connection.)

Definition 2.10 (input and output gates) *An* input gate *is a gate with zero inputs and a single output. An* output gate *is a gate with one input and zero outputs.*

Figure 2.5 depicts an input gate and an output gate. Inputs from the "external world" are fed to a circuit via input gates. Similarly, outputs to the "external world" are fed by the circuit via output gates.

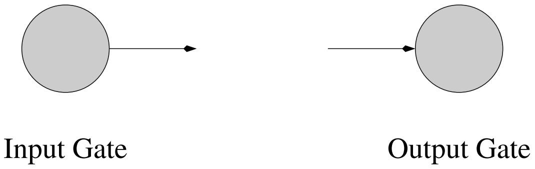


Figure 2.5: An input gate and an output gate

Consider a fixed set of gate-types (e.g.,inverter, NAND-gate, etc.); we often refer to such a set of gate-types as a *library*. We associate with every gate-type in the library the number of inputs, the number of outputs, and the Boolean function that specifies its functionality.

Every gate G in a circuit C is an *instance* of a gate-type from the library. Formally, the *gate-type* of a gate G indicates the library element that corresponds to G (e.g.,"the gate-type of G is an inverter"). To simplify the discussion, we simply refer to a gate G as an inverter instead of saying that its gate-type is an inverter.

We now present a syntactic definition of combinational circuits.

Definition 2.11 (syntactic definition of combinational circuits) *A* combinational circuit *is a pair $C = \langle \mathcal{G}, \mathcal{N} \rangle$ that satisfies the following conditions:*

1. *$\mathcal{G}$ is a set of gates.*
2. *$\mathcal{N}$ is a set of nets over terminals of gates in $\mathcal{G}$.*
3. *Every terminal t of a gate $G \in \mathcal{G}$ belongs to exactly one net $N \in \mathcal{N}$.*
4. *Every net $N \in \mathcal{N}$ is simple.*
5. *The directed graph $DG(C)$ is acyclic.*

Note that Definition 2.11 is independent of the gate types. One need not even know the gate-type of each gate to determine whether a circuit is combinational. Moreover, the question of whether a circuit is combinational is a purely topological question (i.e., are the interconnections between gates legal?).

Question 2.2 *Which conditions in the syntactic definition of combinational circuits are violated by the circuits depicted in Figure 2.4?*

We list below a few properties that explain why the syntactic definition of combinational circuits is so important. In particular, these properties show that the syntactic definition of combinational circuits implies well defined semantics.

1. Completeness: for every Boolean function B, there exists a combinational circuit that implements B. We leave the proof of this property as an exercise for the reader.
2. Soundness: every combinational circuit implements a Boolean function. Note that it is NP-Complete to decide if the Boolean function that is implemented by a given combinational circuit with one output ever gets the value 1.
3. Simulation: given the digital values of the inputs of a combinational circuit, one can simulate the circuit efficiently (the running time is linear in the size of the circuit). Namely, one can compute the digital values of the outputs of the circuit that are output by the circuit once the circuit becomes consistent.
4. Delay analysis: given the propagation delays of all the gates in a combinational circuit, one can compute in linear time an upper bound on the propagation delay of the circuit. (Moreover, computing tighter upper bounds is again NP-Complete.)

The last three properties are proved in the following theorem by showing that in a combinational circuit every net implements a Boolean function of the inputs of the circuit.

Theorem 2.2 (Simulation theorem of combinational circuits) *Let $C = \langle \mathcal{G}, \mathcal{N} \rangle$ denote a combinational circuit that contains k input gates. Let $\{x_i\}_{i=1}^k$ denote the output terminals of the input gates in C. Assume that the digital signals $\{x_i(t)\}_{i=1}^k$ are stable during the interval $[t_1, t_2]$. Then, for every net $N \in \mathcal{N}$ there exist:*

1. *a Boolean function $B_N : \{0,1\}^k \rightarrow \{0,1\}$, and*
2. *a propagation delay $t_{pd}(N)$*

such that

$$N(t) = B_N(x_1(t), x_2(t), \ldots, x_k(t)),$$

for every $t \in [t_1 + t_{pd}(N), t_2]$.

We can simplify the statement of Theorem 2.2 by considering each net $N \in \mathcal{N}$ as an output of a combinational circuit with k inputs. The theorem then states that every net implements a Boolean function with an appropriate propagation delay. To simplify notation, we use $\vec{x}(t)$ to denote the vector $x_1(t), \ldots, x_k(t)$. To simplify the proof, we assume that every gate (except an output gate) has a single output, hence every gate feeds a single net.

Proof: Let n denote the number of gates in $\mathcal{G}$ and m the number of nets in $\mathcal{N}$. The directed graph $DG(C)$ is acyclic. It follows that we can topologically sort the vertices of $DG(C)$. Let $v_1, v_2, \ldots, v_n$ denote the set of gates $\mathcal{G}$ according to the topological order. (This means that if there is a directed path from v_i to v_j in $DG(C)$, then $i < j$.) We assume, without loss of generality, that the inputs are ordered first, namely, $v_1, \ldots, v_k$ is the set of input gates, where the input gate v_i outputs x_i.

Let e_i denote the net in $\mathcal{N}$ that is fed by the output of gate v_i (assuming v_i is not an output gate). Note that if v_i is an input gate then $e_i(t) = x_i(t)$. Let $e_1, e_2, \ldots, e_m$ denote an ordering of the nets in $\mathcal{N}$ such that the net e_i precedes the net e_{i+1}, for every $i < n$. In other words, we first list the net fed by gate v_1, followed by the net fed by gate v_2, etc. Since we assumed that $v_1, \ldots, v_k$ are the input gates, it follows that $e_1 = x_1, \ldots, e_k = x_k$.

Having defined a linear order on the gates and on the nets, we are now ready to prove the theorem by induction on m (the number of nets).

Induction hypothesis: For every $i \leq m'$ there exist:

1. a Boolean function $B_{e_i} : \{0,1\}^k \rightarrow \{0,1\}$, and
2. a propagation delay $t_{pd}(e_i)$

such that the network e_i implements the Boolean function B_{e_i} with propagation delay $t_{pd}(e_i)$.

Induction Basis: Recall that the first k nets are the input signals, hence $e_i(t) = x_i(t)$, if $i \leq k$. Hence we define B_{e_i} to be simply the projection on the ith component, namely $B_{e_1}(\sigma_1, \ldots, \sigma_k) = \sigma_i$. The propagation delay $t_{pd}(e_i)$ is zero. The induction basis follows for $m' = k$.

Induction Step: Assume that the induction hypothesis holds for $m' < m$. We wish to prove that it also holds for $m' + 1$. Consider the net $e_{m'+1}$. Let v_i denote the gate that feeds the net $e_{m'+1}$. To simplify notation, assume that the gate v_i has two input terminals that are fed by the nets e_j and e_k, respectively. The ordering of the nets guarantees that $j, k < m' + 1$. By the induction hypothesis, the net e_j (resp., e_k) implements a Boolean function B_{e_j} (resp., B_{e_k}) with propagation delay $t_{pd}(e_j)$ (resp., $t_{pd}(e_k)$). This implies that both inputs to gate v_i are stable during the interval

$$[t_1 + \max\{t_{pd}(e_j), t_{pd}(e_k)\}, t_2].$$

Gate v_i implements a Boolean function B_{v_i} with propagation delay $t_{pd}(v_i)$. It follows that the output of gate v_i equals

$$B_{v_i}(B_{e_j}(\vec{x}(t)), B_{e_k}(\vec{x}(t)))$$

during the interval

$$[t_1 + \max\{t_{pd}(e_j), t_{pd}(e_k)\} + t_{pd}(v_i), t_2].$$

We define $B_{e_{m'+1}}$ to be the Boolean function obtained by the composition of Boolean functions $B_{e_{m'+1}}(\vec{\sigma}) = B_{v_i}(B_{e_j}(\vec{\sigma}), B_{e_k}(\vec{\sigma}))$. We define $t_{pd}(e_{m'+1})$ to be $\max\{t_{pd}(e_j), t_{pd}(e_k)\} + t_{pd}(v_i)$, and the induction step follows. □

The proof of Theorem 2.2 leads to two related algorithms. One algorithm simulates a combinational circuit, namely, given a combinational circuit and a Boolean assignment to the inputs $\vec{x}$, the algorithm can compute the digital signal of every net after a sufficient amount of time elapses. The second algorithm computes the propagation delay of each net. Of particular interest are the nets that feed the output gates of the combinational circuit. Hence, we may regard a combinational circuit as a "macro-gate". All instances of the same combinational circuit implement the same Boolean function and have the same propagation delay.

The algorithms are very easy. For convenience we describe them as one joint algorithm. First, the directed graph $DG(C)$ is constructed (this takes linear time). Then the gates are sorted in topological order (this also takes linear time). This order also induced an order on the nets. Now a sequence of *relaxation* steps take place for nets $e_1, e_2, \ldots, e_m$. In a relaxation step the propagation delay of a net e_i two computations take place:

1. The Boolean value of e_i is set to
$$B_{v_j}(\vec{I}_{v_j}),$$
where v_j is the gate that feeds the net e_i and $\vec{I}_{v_j}$ is the binary vector that describes the values of the nets that feed gate v_j. We usually assume that the Boolean function B_{v_j} is efficiently computable (i.e., its value is computable in a time that is linear in the number of inputs plus outputs).

2. The propagation delay of the gate that feeds e_i is set to

$$t_{pd}(e_i) \leftarrow t_{pd}(v_j) + \max\{t_{pd}(e')\}_{\{e' \text{ feeds } v_j\}}.$$

We define the circuit size to be the number of gates plus the sum of the sizes of the nets. This implies that the total amount of time spent in the relaxation steps is linear in the circuit size, and hence the running time of the algorithm for computing the propagation delay is linear. Simulation requires computing the Boolean function implemented by each gate. To maintain linear running time, we must assume that each Boolean function is computable in time that is linear in the number of inputs plus outputs.

Question 2.3 *Prove that the total amount of time spent in the relaxation steps is linear in the number nodes if the fan-in of each gate is constant (say, at most* 3*).*

Note that it is not true that each relaxation step can be done in constant time if the fan-in of the gates is not constant.

Prove linear running time in the number of nodes if (i) every net feeds a single input terminal and (ii) the number of outputs of each gate is constant. (You may not assume that the fan-in of every gate is constant.)

2.6 Cost and propagation delay

In this section we define the cost and propagation delay of a combinational circuit.

We associate a cost with every gate. We denote the cost of a gate G by $c(G)$.

Definition 2.12 *The cost of a combinational circuit* $C = \langle \mathcal{G}, \mathcal{N} \rangle$ *is defined by*

$$c(C) \triangleq \sum_{G \in \mathcal{G}} c(G).$$

The following definition defined the propagation delay of a combinational circuit.

Definition 2.13 *The propagation delay of a combinational circuit* $C = \langle \mathcal{G}, \mathcal{N} \rangle$ *is defined by*

$$t_{pd}(C) \triangleq \max_{N \in \mathcal{N}} t_{pd}(N).$$

We often refer to the propagation delay of a combinational circuit as its *depth* or simply its *delay*.

Definition 2.14 *A sequence* $p = \{v_0, v_1, \ldots, v_k\}$ *of gates from* $\mathcal{G}$ *is a* path *in a combinational circuit* $C = \langle \mathcal{G}, \mathcal{N} \rangle$ *if* p *is a path in the directed graph* $DG(C)$*.*

The propagation delay of a path p is defined as

$$t_{pd}(p) = \sum_{v \in p} t_{pd}(v).$$

The proof of the following claim follows directly from the proof of Theorem 2.2.

Claim 2.3 *The propagation delay of a combinational circuit $C = \langle \mathcal{G}, \mathcal{N} \rangle$ equals*

$$t_{pd}(C) = \max\{t_{pd}(p) \mid p \textit{ is a path in } DG(C)\}$$

Paths, the delay of which equals the propagation delay of the circuit, are called *critical paths.* We may assume that critical paths always end in output gates.

Question 2.4 (Exponentially many paths in circuits) *Describe a combinational circuit with n gates that has at least $2^{n/2}$ paths. Can you describe a circuit with 2^n different paths?*

Question 2.5 *In Claim 2.3 the propagation delay of a combinational circuit is claimed to equal the maximum delay of a path in the circuit. The number of paths can be exponential in n. How can we compute the propagation delay of a combinational circuit in linear time?*

Müller and Paul compiled a table of costs and delays of gates. These figures were obtained by considering ASIC libraries of two technologies and normalizing them with respect to the cost and delay of an inverter. They referred to these technologies as Motorola and Venus. Table 2.1 summarizes the normalized costs and delays in these technologies according to Müller and Paul.

Gate	Motorola		Venus	
	cost	delay	cost	delay
INV	1	1	1	1
AND,OR	2	2	2	1
NAND, NOR	2	1	2	1
XOR, NXOR	4	2	6	2
MUX	3	2	3	2

Table 2.1: Costs and delays of gates

2.7 Syntax and semantics

In this chapter we have used both explicitly and implicitly the terms *syntax* and *semantics.* These terms are so fundamental that they deserve a section.

The term semantics (in our context) refers to the function that a circuit implements. Often, the semantics of a circuit is referred to as the *functionality* or even the *behavior* of the circuit. In general, the semantics of a circuit is a formal description that relates the outputs of the circuit to the inputs of the circuit. In the case of combinational circuits, semantics are described by Boolean functions. Note that in non-combinational circuits, the output depends not only on the current inputs, so semantics cannot be described simply by a Boolean function.

The term syntax refers to a formal set of rules that govern how "grammatically correct" circuits are constructed from smaller circuits (just as sentences are built by combining words). In the syntactic definition of combinational circuits, the functionality (or gate-type) of each gate is not important. The only part that matters is that the rules for connecting gates together are followed. Following syntax in itself does not guarantee that the resulting circuit is useful. Following syntax is, in fact, a restriction that we are willing to accept so that we can enjoy the benefits of well defined functionality, simple simulation, and simple timing analysis. The restriction of following syntax rules is a reasonable choice since every Boolean function can be implemented by a syntactically correct combinational circuit.

2.8 Summary

Combinational circuits were formally defined in this chapter. We started by considering the basic building blocks: gates and wires. Gates are simply implementations of Boolean functions. The digital abstraction enables a simple definition of what it means to implement a Boolean function B. Given a propagation delay t_{pd} and stable inputs whose digital value is $\vec{x}$, the digital values of the outputs of a gate equal $B(\vec{x})$ after t_{pd} time elapses.

Wires are used to connect terminals together. Bunches of wires are used to connect multiple terminals to each other and are called nets. Simple nets are nets in which the direction in which information flows is well defined; from output terminals of gates to input terminals of gates.

The formal definition of combinational circuits turns out to be most useful. It is a syntactic definition that only depends on the topology of the circuit, namely, how the terminals of the gates are connected. One can check in linear time whether a given circuit is indeed a combinational circuit. Even though the definition ignores functionality, one can compute in linear time the digital signals of every net in the circuit. Moreover, one can also compute in linear time the propagation delay of every net.

Two quality measures are defined for every combinational circuit: cost and propagation delay. The cost of a combinational circuit is the sum of the costs of the gates in the circuit. The propagation delay of a combinational is the maximum delay of a path in the circuit.

Chapter 3

Trees

Contents

Preliminary questions:

1. Which Boolean functions are suited for implementation by tree-like combinational circuits?
2. In what sense are tree-like implementations optimal?

In this chapter we deal with combinational circuits that have a topology of a tree. We begin by considering circuits for associative Boolean function. We then prove two lower bounds; one for cost and one for delay. These lower bounds do not assume that the circuits are trees. The lower bounds prove that trees have optimal cost and balanced trees have optimal delay.

3.1 Trees of associative Boolean gates

In this section, we deal with combinational circuits that have a topology of a tree. All the gates in the circuits we consider are instances of the same gate that implements an associative Boolean function.

3.1.1 Associative Boolean functions

Definition 3.1 *A Boolean function $f : \{0,1\}^2 \rightarrow \{0,1\}$ is* associative *if*

$$f(f(\sigma_1, \sigma_2), \sigma_3) = f(\sigma_1, f(\sigma_2, \sigma_3)),$$

for every $\sigma_1, \sigma_2, \sigma_3 \in \{0,1\}$.

Question 3.1 *List all the associative Boolean functions $f : \{0,1\}^2 \rightarrow \{0,1\}$.*

A Boolean function defined over the domain $\{0,1\}^2$ is often denoted by a dyadic operator, say $\odot$. Namely, $f(\sigma_1, \sigma_2)$ is denoted by $\sigma_1 \odot \sigma_2$. Associativity of a Boolean function $\odot$ is then formulated by

$$\forall \sigma_1, \sigma_2, \sigma_3 \in \{0,1\} \ : \ (\sigma_1 \odot \sigma_2) \odot \sigma_3 = \sigma_1 \odot (\sigma_2 \odot \sigma_3).$$

This implies that one may omit parenthesis from expressions involving an associative Boolean function and simply write $\sigma_1 \odot \sigma_2 \odot \sigma_3$. Thus we obtain a function defined over $\{0,1\}^n$ from a dyadic Boolean function. We formalize this composition of functions as follows.

Definition 3.2 *Let $f : \{0,1\}^2 \rightarrow \{0,1\}$ denote a Boolean function. The function $f_n : \{0,1\}^n \rightarrow \{0,1\}$, for $n \geq 2$ is defined by induction as follows.*

1. *If $n = 2$ then $f_2 \equiv f$ (the sign $\equiv$ is used instead of equality to emphasize equality of functions).*

2. *If $n > 2$, then f_n is defined based on f_{n-1} as follows:*

$$f_n(x_1, x_2, \ldots x_n) \stackrel{\triangle}{=} f(f_{n-1}(x_1, \ldots, x_{n-1}), x_n).$$

If $f(x_1, x_2)$ is an associative Boolean function, then one could define f_n in many equivalent ways, as summarized in the following claim.

Claim 3.1 *If $f : \{0,1\}^2 \rightarrow \{0,1\}$ is an associative Boolean function, then*

$$f_n(x_1, x_2, \dots x_n) = f(f_k(x_1, \dots, x_k), f_{n-k}(x_{k+1}, \dots, x_n)),$$

for every $k \in [2, n-2]$.

Question 3.2 *Prove that each of the following functions $f : \{0,1\}^n \rightarrow \{0,1\}$ is associative:*

$$f \in \left\{\textit{constant } 0, \textit{constant } 1, x_1, x_n, \text{AND}_n, \text{OR}_n, \text{XOR}_n, \text{NXOR}_n\right\}.$$

The implication of Question 3.2 is that there are only four non-trivial associative Boolean functions f_n (which?). In the rest of this section we will only consider the Boolean function OR_n. The discussion for the other three non-trivial functions is analogous.

3.1.2 OR-trees

Definition 3.3 *A combinational circuit $C = \langle \mathcal{G}, \mathcal{N} \rangle$ that satisfies the following conditions is called an* OR*-tree(n).*

1. ***Input:*** $x[n-1:0]$.
2. ***Output:*** $y \in \{0,1\}$
3. ***Functionality:*** $y = \text{OR}(x[0], x[1], \cdots, x[n-1])$.
4. ***Gates:*** *All the gates in $\mathcal{G}$ are* OR*-gates.*
5. ***Topology:*** *The underlying graph of $DG(C)$ (i.e., undirected graph obtained by ignoring edge directions) is a tree.*

Consider the binary tree T corresponding to the underlying graph of $DG(C)$, where C is an OR-tree(n). The root of T corresponds to the output gate of C. The leaves of T correspond to the input gates of C, and the interior nodes in T correspond to OR-gates in C.

Claim 3.1 provides a "recipe" for implementing an OR-tree using OR-gates. Consider a rooted binary tree with n leaves. The inputs are fed via the leaves, an OR-gate is positioned in every node of the tree, and the output is obtained at the root. Figure 3.1 depicts two OR-tree(n) for $n = 4$.

One could also define an OR-tree(n) recursively, as follows.

Definition 3.4 *An* OR*-tree(n) is defined recursively as follows (see Figure 3.2):*

1. *Basis $(n = 1)$: An input gate that directly feeds an output gate is an* OR*-tree with one input.*
2. *Step $(n > 1)$: Let T_{n_1} and T_{n_2} denote* OR*-trees with n_1 and n_2 inputs where $n_1, n_2 > 0$ and $n_1 + n_2 = n$. Remove the output gate from each tree, and connect the corresponding nets to the inputs of a (new)* OR*-gate. Connect the output of the new gate to an output gate.*

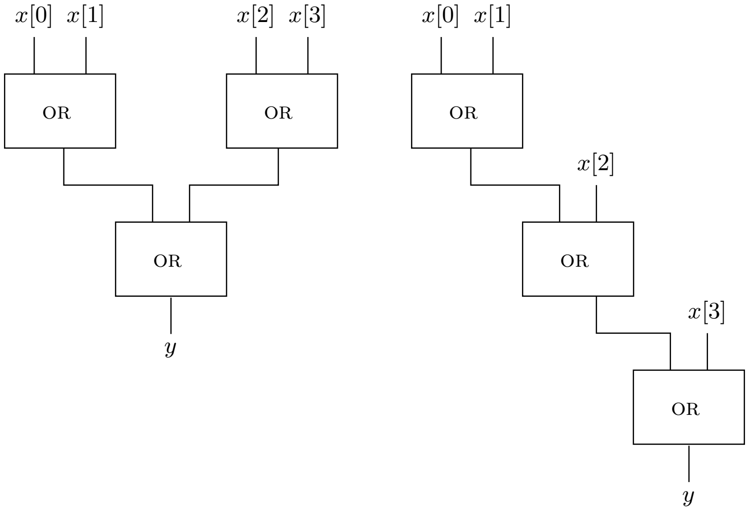


Figure 3.1: Two implementations of an OR-tree(n) with $n = 4$ inputs.

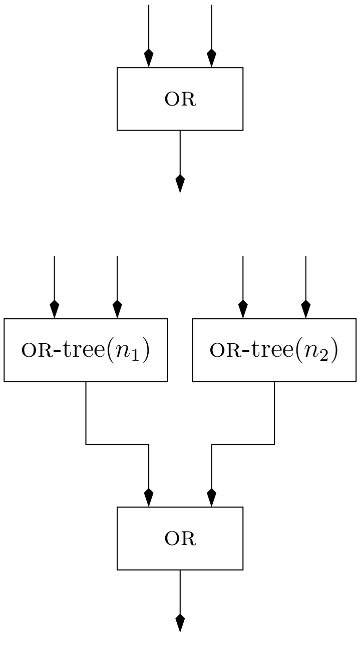


Figure 3.2: A recursive definition of an OR-tree(n).

Question 3.3 *Design a zero-tester defined as follows.*

Input: $x[n-1:0]$.

Output: y

Functionality:

$$y = 1 \;\; \textit{iff} \;\; x[n-1:0] = 0^n.$$

1. *Suggest a design based on an* OR*-tree.*
2. *Suggest a design based on an* AND*-tree.*
3. *What do you think about a design based on a tree of* NOR*-gates?*

3.1.3 Cost analysis

You may have noticed that both OR-trees depicted in Figure 3.1 contain three OR-gates. However, their delay is different. The following claim summarizes the fact that all OR-trees have the same cost.

Claim 3.2 *The cost of every* OR*-tree*(n) *is* $(n-1) \cdot c(\text{OR})$.

Proof: The proof is by induction on n. The induction basis, for $n = 2$ follows because OR-tree(2) contains a single OR-gate. (What about the case $n = 1$?)

We now prove the induction step. Let C denote an OR-tree(n), and let g denote the OR-gate that outputs the output of C. The gate g is fed by two wires e_1 and e_2. The recursive definition of OR-gate(n) implies the following. For $i = 1, 2$, the wire e_i is the output of C_i, where C_i is an OR-tree(n_i). Moreover, $n_1 + n_2 = n$. The induction hypothesis states that $c(C_1) = (n_1 - 1) \cdot c(\text{OR})$ and $c(C_2) = (n_2 - 1) \cdot c(\text{OR})$. We conclude that

$$\begin{aligned} c(C) &= c(g) + c(C_1) + c(C_2) \\ &= (1 + n_1 - 1 + n_2 - 1) \cdot c(\text{OR}) \\ &= (n-1) \cdot c(\text{OR}), \end{aligned}$$

and the claim follows. □

Claim 3.2 is re-statement of the well known relationship between the number of leaves and interior nodes in rooted binary trees. To make sure that the connection is well understood, we describe this relationship in the following lemma. We begin with trees that are not rooted.

Lemma 3.3 *Let* $T = (V, E)$ *denote a tree. A* leaf *is a vertex of degree* 1*. An* interior *vertex is a vertex that is not a leaf. We denote the set of leaves in* V *by* $leaves(V)$ *and the set of interior vertices in* V *by* $interior(V)$.

If the degree of every vertex in T *is at most three, then*

$$|interior(V)| \geq |leaves(V)| - 2.$$

Proof: The proof is by induction on $|V|$. The induction basis for $|V| \leq 2$ trivially holds because the right hand side is nonpositive, and the left hand side is nonnegative. The induction step is proved as follows. Pick an arbitrary vertex r and call it the root. Now assign directions to all the edges as follows: an edge (u, v) is assigned the direction $u \rightarrow v$ if u belongs to the path from r to v (check that the direction of each edge is well defined!). We refer to v as a *child* of u if $(u, v) \in E$ and the edge is assigned the direction $u \rightarrow v$.

Consider a vertex v that is furthest away from the root r. It follows that v must be a leaf (why?). Let u denote the vertex that appears before v along the path from r to v. Namely, v is a child of u. Since v is furthest away from the root r, it follows that every child of u is a leaf.

Let $T' = (V', E')$ denote the tree obtained from T by deleting the children of u and the edges incident to the children of u. The induction hypothesis, applied to T', gives

$$|\text{interior}(V')| \geq |\text{leaves}(V')| - 2.$$

But, $|\text{interior}(V')| = |\text{interior}(V)| - 1$ and $|\text{leaves}(V')| = |\text{leaves}(V)| - |\text{children}(u)| + 1$. Since u has at most two children (recall that its degree is at most 3), the induction step follows. □

In a binary tree, the degree of every vertex is at most three. Hence Lemma 3.3 is a lemma about binary trees. You may be used to the statement that the number of leaves is one plus the number of interior nodes. Note, however, that this statement is about rooted trees in which the root does not count as a leaf (even if its degree is one).

We now connect Claim 3.2 with Lemma 3.3. Let C denote an OR-tree(n). Let $DG(C)$ denote the directed acyclic graph that corresponds to C. Let $G = (V, E)$ denote the underlying graph that corresponds to $DG(C)$ (i.e., the undirected graph obtained by ignoring edge directions). Since C is an OR-tree(n), it follows that $G = (V, E)$ satisfies the following properties: (1) G is a tree, (2) the degree of every vertex in G is either one or three, (3) the leaves of G correspond to the input gates and the output gate of C (hence G has $n + 1$ leaves), and (4) the interior vertices of G correspond to OR-gates in C. In Claim 3.2 we have equality whereas in Lemma 3.3 we only have a lower bound on the number of gates. The explanation is that in Lemma 3.3 we did not assume that the degree of every interior node is 3.

3.1.4 Delay analysis

The delay of an OR-tree(n) is simply the number of OR-gates along the longest path from an input to an output times the delay of an OR-gate. The depth of a rooted tree is the length of the longest path from the root to a leaf. We refer to a rooted tree as a *minimum depth tree* if its depth is minimum among all the rooted trees with the same number of leaves.

Since all trees have the same cost, we may focus on minimizing the depth of the tree without worrying about the cost. The natural candidates to minimize delay are balanced trees. One can easily verify that for n that is a power of 2 there is a unique minimum depth tree, namely, the complete binary tree with $\log_2 n$ levels.

On the other hand, if n is not a power of 2, then there can be more than one minimum depth tree, as demonstrated in the following example.

Example 3.1 *Consider the two trees that are depicted in Figure 3.3, each with* 6 *inputs. One tree is obtained from two binary trees with three leaves each. The second tree is obtained from one binary tree with four leaves and one with two leaves. Although both these trees have six leaves, they are quite different. On the other hand, their depth is the same. Are these minimum depth trees?*

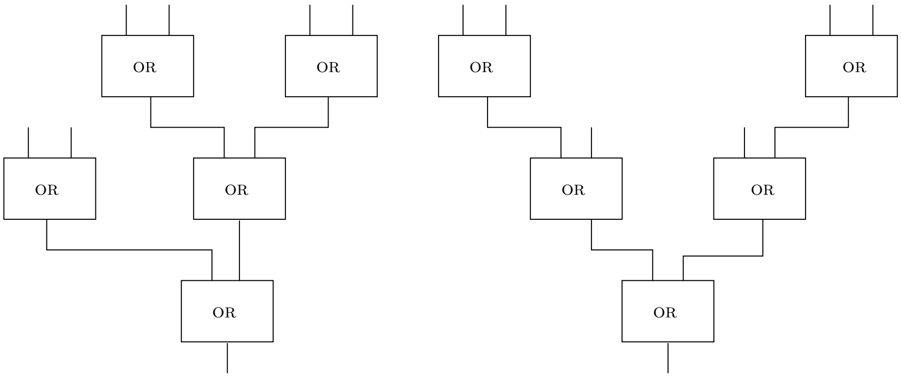


Figure 3.3: Two trees with six inputs.

Our goal is to prove that the depth of every rooted tree with n leaves is at least $\lceil \log_2 n \rceil$. Moreover, we wish to show that this bound can be obtained rather easily.

Claim 3.4 *If T_n is a rooted binary tree with n leaves, then the depth of T_n is at least $\lceil \log_2 n \rceil$.*

Question 3.4 *Prove Claim 3.4. (Hint: Start with the root, and keep picking the child that has more leaves hanging from it. Prove that this path is long.)*

Question 3.5 *Assume that n is a power of* 2. *Prove that the depth of a complete binary tree with n leaves is* $\log_2 n$.

We now show that for every n, we can construct a minimum depth tree T_n of depth $\lceil \log_2 n \rceil$. In fact, if n is not a power of 2, then there are many such trees.

We start with a simple rule for determining how to split the leaves between the subtrees hanging from the root.

Definition 3.5 *Two positive integers a, b are a* balanced partition *of n if: (1) $a+b=n$, and (2)* $\max\{\lceil \log_2 a \rceil, \lceil \log_2 b \rceil\} \leq \lceil \log_2 n \rceil - 1$.

Question 3.6 *An* even partition *is a balanced partition in which $|a-b|$ is minimized. Prove that $a = \lceil n/2 \rceil$ and $b = n - a$ is an even partition. (Hint: It is easy to show that $|a-b|$ is minimized. Now prove that the partition is balanced for an even n. For an odd n, prove that $\lceil \log_2 a \rceil > \lceil \log_2 n \rceil - 1$ iff n is a power of two. Hence, for an odd n, this cannot happen.)*

Question 3.7 *Prove that if $n = 2^k - r$, where $0 \leq r < 2^{k-1}$, then any partition in which $2^{k-1} - r \leq a \leq 2^{k-1}$ and $b = n - a$ is a balanced partition.*

The following question deals with the construction of minimum depth trees. $\lceil \log_2 n \rceil$.

Question 3.8 *Consider the following recursive algorithm for constructing a binary tree T_n with $n \geq 2$ leaves. Prove that the depth of T_n is $\lceil \log_2 n \rceil$.*

1. *The case that $n \leq 2$ is trivial (two leaves connected to a root).*
2. *If $n > 2$, then let a, b be balanced partition of n.*
3. *Compute trees T_a and T_b. Connect their roots to a new root to obtain T_n.*

3.2 Optimality of trees

In this section we deal with the following questions: What is the best choice of a topology for a combinational circuit that implements the Boolean function OR_n? Is a tree indeed the best topology? Perhaps one could do better if another implementation is used? (Say, using other gates or connecting an input x_i to more than one gate.)

We attach two measures to every design: cost and delay. In this section we prove lower bounds on the cost and delay of every circuit that implements the Boolean function OR_n. These lower bounds imply the optimality of using balanced OR-trees.

3.2.1 Definitions

In this section we present a few definitions related to Boolean functions.

Definition 3.6 (restricted Boolean functions) *Let $f : \{0,1\}^n \rightarrow \{0,1\}$ denote a Boolean function. Let $\sigma \in \{0,1\}$. The Boolean function $g : \{0,1\}^{n-1} \rightarrow \{0,1\}$ defined by*

$$g(w_0, \ldots, w_{n-2}) \stackrel{\triangle}{=} f(w_0, \ldots, w_{i-1}, \sigma, w_i, \ldots, w_{n-2})$$

is called the restriction *of f with $x_i = \sigma$. We denote it by $f_{\restriction x_i = \sigma}$.*

Example 3.2 *Consider the Boolean function $f(\vec{x}) = \text{XOR}_n(x_1, \ldots, x_n)$. The restriction of f with $x_n = 1$ is the Boolean function*

$$\begin{aligned} f_{\restriction x_n = 1}(x_1, \ldots, x_{n-1}) &\stackrel{\triangle}{=} \text{XOR}_n(x_1, \ldots, x_{n-1}, 1) \\ &= \text{INV}(\text{XOR}_{n-1}(x_1, \ldots, x_{n-1})). \end{aligned}$$

Definition 3.7 (cone of a Boolean function) *A Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$* depends *on its ith input if*

$$f_{\restriction x_i = 0} \neq f_{\restriction x_i = 1}.$$

The cone *of a Boolean function f is defined by*

$$\text{cone}(f) \stackrel{\triangle}{=} \{i : f_{\restriction x_i = 0} \neq f_{\restriction x_i = 1}\}.$$

The following claim is trivial.

Claim 3.5 *The Boolean function* OR_n *depends on all its inputs, namely*

$$|\text{cone}(\text{OR}_n)| = n.$$

Example 3.3 *Consider the following Boolean function:*

$$f(\vec{x}) = \begin{cases} 0 & \textit{if } \sum_i x_i < 3 \\ 1 & \textit{otherwise.} \end{cases}$$

Suppose that one reveals the input bits one by one. As soon as 3 *ones are revealed, one can determine the value of* $f(\vec{x})$. *Nevertheless, the function* $f(\vec{x})$ *depends on all its inputs, and hence,* $\text{cone}(f) = \{1, \ldots, n\}$.

The following trivial claim deals with the case that $cone(f) = \emptyset$.

Claim 3.6 $\text{cone}(f) = \emptyset \iff f$ *is a constant Boolean function.*

3.2.2 Lower bounds

The following claim shows that, if a combinational circuit C implements a Boolean function f, then there must be a path in $DG(C)$ from every input in $cone(f)$ to the output of f.

Claim 3.7 *Let* $C = \langle \mathcal{G}, \mathcal{N} \rangle$ *denote a combinational circuit that implements a Boolean function* $f : \{0,1\}^n \to \{0,1\}$. *Let* $g_i \in \mathcal{G}$ *denote the input gate that feeds the* i*th input. If* $i \in \text{cone}(f)$, *then there is a path in* $DG(C)$ *from* g_i *to the output gate of* C.

Proof: If $DC(C)$ lacks a path from the input gate g_i that feeds an input $i \in cone(f)$ to the output y of C, then C cannot implement the Boolean function f. Consider an input vector $w \in \{0,1\}^{n-1}$ for which $f_{\restriction x_i=0}(w) \neq f_{\restriction x_i=1}(w)$. Let w' (resp., w'') denote the extension of w to n bits by inserting a 0 (resp., 1) in the ith coordinate. The proof of the Simulation Theorem of combinational circuits (Theorem 2.2) implies that C outputs the same value when given the input strings w' and w'', and hence C does not implement f, a contradiction. □

The following theorem shows that every circuit, that implements the Boolean function OR_n in which the fan-in of every gate is bounded by two, must contain at least $n-1$ non-trivial gates (a trivial gate is an input gate, an output gate, or a gate that feeds a constant). We assume that the cost of every non-trivial gate is at least one, therefore, the theorem is stated in terms of cost rather than counting non-trivial gates.

Theorem 3.8 (Linear Cost Lower Bound Theorem) *Let* C *denote a combinational circuit that implements a Boolean function* f. *If the fan-in of every gate in* $\mathcal{C}$ *is at most* 2, *then*

$$c(C) \geq |\text{cone}(f)| - 1.$$

Before we prove Theorem 3.8 we show that it implies the optimality of OR-trees. Note that it is very easy to prove a lower bound of $n/2$. The reason is that every input must be fed to a non-trivial gate, and each gate can be fed by at most two inputs.

Corollary 3.9 *Let C_n denote a combinational circuit that implements* OR_n *with input length n. Then*

$$c(C_n) \geq n - 1.$$

Proof: Follows directly from Claim 3.5 and Theorem 3.8. □

We prove Theorem 3.8 by considering the directed acyclic graph (DAG) $DG(C)$. We use the following terminology for DAGs: The *in-degree* (resp., *out-degree*) of a vertex is the number of edges that enter (resp., emanate from) the vertex. A *source* is a vertex with in-degree zero. A *sink* is a vertex with out-degree zero. An *interior vertex* is a vertex that is neither a source or a sink. See Figure 3.4 for an example.

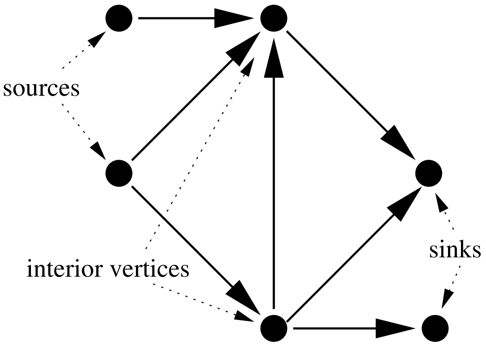


Figure 3.4: A DAG with two sources, two interior vertices, and two sinks.

Proof of Theorem 3.8: We first suppose that the underlying graph $G = (V, E)$ of $DG(C)$ is a tree. By Lemma 3.3 it follows that

$$|\text{interior}(V)| \geq |\text{leaves}(V)| - 2.$$

Recall that leaves correspond to the input gates and the output gate. Hence, the number of leaves is $n + 1$, and the right hand side equals $n - 1$. Moreover, every interior vertex corresponds to a non-trivial gate. Hence, it follows that there are at least $n-1$ non-trivial gates, as required.

If the underlying graph of $DG(C)$ is not a tree, then we construct a "spanning" subgraph $T = (V', E')$ of $DG(C)$ that satisfies the following properties: (i) the sources in V' are all the input gates that feed inputs x_i such that $i \in cone(f)$, (ii) the output gate is the sink, (iii) the underlying graph of T is a tree, (iv) the in-degree of every node in T is at most two.

The subgraph T is constructed as follows. Pick a source $v \in V$ that feeds an input x_i such that $i \in cone(f)$. By Claim 3.7, there is a path in $DG(C)$ from v to the output gate. Add all the edges and vertices of this path to T. Now continue in this manner by picking, one by one, sources that feed inputs x_i such that $i \in cone(f)$. Each time consider a path

p that connects the source to the output gate. Add the prefix of the path p to T up to the first vertex that is already contained in T. We leave it as an exercise to show that T meets the three required conditions.

In the underlying graph of T we have the inequality $|\text{interior vertices}| \geq |\text{leaves}| - 2$. The interior vertices of T are also interior vertices of $DG(C)$, and the theorem follows. □

Question 3.9 *State and prove a generalization of Theorem 3.8 for the case that the fan-in of every gate is bounded by a constant k.*

We now turn to proving a lower bound on the delay of a combinational circuit that implements OR_n. Again, we use a general technique and rely on all gates in the design having a constant fan-in.

The following theorem shows a lower bound on the delay of combinational circuits that is logarithmic in the size of the cone. We assume that the delay of every nontrivial gate is at least one.

Theorem 3.10 (Logarithmic Delay Lower Bound Theorem) *Let C denote a combinational circuit that implements a non-constant Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$. If the fan-in of every gate in C is at most k, then the delay of C is at least* $\log_k |\text{cone}(f)|$.

Before we prove Theorem 3.10, we show that the theorem implies a lower bound on the delay of combinational circuits that implement OR_n.

Corollary 3.11 *Let C_n denote a combinational circuit that implements OR_n. Let k denote the maximum fan-in of a gate in C_n. Then*

$$t_{pd}(C_n) \geq \lceil \log_k n \rceil .$$

Proof: The corollary follows directly from Claim 3.5 and Theorem 3.10. □

Proof of Theorem 3.10: The proof deals only with the graph $DG(C)$ and shows that there must be a path with at least $\log_k |cone(f)|$ interior vertices in $DG(C)$. Note that input/output gates and constants have zero delay, so we have to be careful not to count them. However, zero delay vertices can appear only as end-points of a path; this is why we count interior vertices along paths.

The proof involves strengthening the theorem to every vertex v as follows. Following the notion of cones, we denote by $cone(v)$ the subset of sources from which v is reachable. Let $d(v)$ denote the maximum number of interior vertices of $DG(C)$ along a path from a source in $cone(v)$ to v (including v). We now prove that, for every vertex v,

$$d(v) \geq \log_k |cone(v)|. \tag{3.1}$$

Claim 3.7 implies there must be a path in $DG(C)$ from every input $x_i \in cone(f)$ to the output y of C. Hence $|cone(f)| \leq |cone(y)|$. Therefore, Equation 3.1 implies the theorem.

We prove Equation 3.1 by induction on $d(v)$. The induction basis, for $d(v) = 0$, is trivial since $d(v) = 0$ implies that v is a source. The cone of a source v consists v itself, and $\log 1 = 0$.

The induction hypothesis is

$$d(v) \leq i \quad \Longrightarrow \quad d(v) \geq \log_k |cone(v)|. \tag{3.2}$$

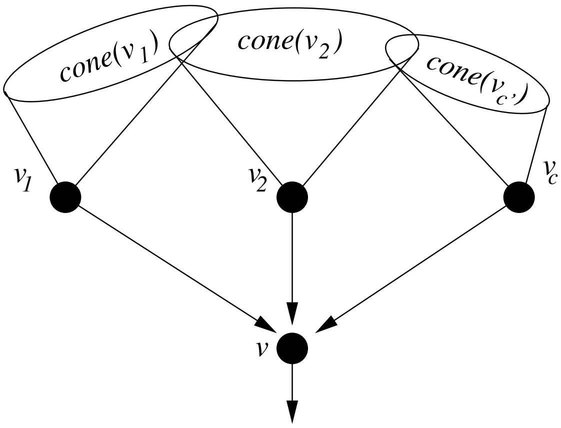


Figure 3.5: The induction step in the proof of Theorem 3.10

In the induction step, we wish to prove that the induction hypothesis implies that Equation 3.2 holds also if $d(v) = i + 1$. Consider a vertex v with $d(v) = i + 1$ (see Figure 3.5 for a depiction of the induction step). We first assume that v is an interior vertex. There are at most k edges that enter v. Denote the vertices that precede v by $v_1, \ldots, v_k'$, where $k' \leq k$. Namely, the edges that enter v are $v_1 \rightarrow v, \ldots, v_{k'} \rightarrow v$. Since v is an interior vertex, it follows by definition that

$$d(v) = \max\{d(v_i)\}_{i=1}^{k'} + 1. \tag{3.3}$$

Since v is not a source, it follows by definition that

$$cone(v) = \bigcup_{i=1}^{k'} cone(v_i).$$

Hence

$$\begin{aligned} |cone(v)| &\leq \sum_{i=1}^{k'} |cone(v_i)| \\ &\leq k' \cdot \max\{|cone(v_1)|, \ldots, |cone(v_{k'})|\}. \end{aligned} \tag{3.4}$$

Let v' denote a predecessor of v that satisfies $|cone(v')| = \max\{|cone(v_i)|\}_{i=1}^{k'}$. The induction hypothesis implies that

$$d(v') \geq \log_k |cone(v')|. \tag{3.5}$$

But,

$$\begin{aligned} d(v) &\geq 1 + d(v') && \text{by Eq. 3.3} \\ &\geq 1 + \log_k |cone(v')| && \text{by Eq. 3.5} \\ &\geq 1 + \log_k |cone(v)|/k' && \text{by Eq. 3.4} \\ &\geq \log_k |cone(v)|. \end{aligned}$$

To complete the proof we consider the case the v is an output gate. In this case, v has a unique predecessor v' that satisfies: $d(v) = d(v')$ and $cone(v) = cone(v')$. The induction step applies to v', and therefore we also get $d(v) \geq \log_k |cone(v)|$, as required, and the theorem follows. □

Question 3.10 *Prove the following statement. Let $U \subseteq V$ denote a subset of vertices of a directed graph $G = (V, E)$, and let $r \in V$. There exists a vertex $u \in U$ such that* $\text{dist}(u, r) \geq \log_k |U|$*, where k denotes the maximum in-degree in G. (*$\text{dist}(u, r)$ *denotes the length of the shortest path from u to r; if there is no such path, then the distance is infinite.)*

3.3 Summary

In this chapter we focused on combinational circuits that have a topology of a tree and are built from instances of identical gates. Such circuits are especially suited for computing associative Boolean functions (why?).

We began this chapter by extending associative dyadic functions to n arguments. We argued that there are only four non-trivial associative Boolean functions; and we decided to focus on OR_n. We then defined an OR-tree(n) to be a combinational circuit that implements OR_n using a topology of a tree.

Although it is intuitive that OR-trees are the cheapest designs for implementing OR_n, we had to work a bit to prove it. It is also intuitive that balanced OR-trees are the fastest designs for implementing OR_n, and again, we had to work a bit to prove that too.

We will be using the lower bounds that we proved in this chapter also in the next chapters. To prove these lower bounds, we introduced the $cone(f)$ of a Boolean function f. The cone of f is the set of inputs the function f depends on.

If all the gates have a fan-in of at most two and the cost and delay of non-trivial gates is at least one, then the lower bounds are as follows. The first lower bound states that the number of gates of a combinational circuit implementing a Boolean function f must be at least $|cone(f)| - 1$. The second lower bound states that the propagation delay of a circuit implementing a Boolean function f is at least $\log_2 |cone(f)|$.

Chapter 4

Decoders and Encoders

Contents

Preliminary questions:

1. Suggest methods for designing combinational circuits that implement nontrivial Boolean functions.
2. How do we check if a design of a combinational circuit is correct?

In this chapter we present two important combinational circuits called decoders and encoders. Decoders and encoders are often used as part of bigger circuits. Along the way we will define buses and develop notation for buses.

4.1 Notation

We begin this section by describing what buses are. Consider two blocks in a circuit (such blocks are often called "modules") that are connected by many wires. A good example is an adder and a register (a memory device) in which the output of the adder is input to the register (namely, we wish to store the sum). Suppose that the adder outputs a binary number with 8 bits. This means that there are 8 different nets that are fed by the adder and feed inputs of the register. Since the nets are different, we must assign a different name to each net. Instead of naming the nets $a, b, c, \ldots$, one uses the names $a[0], a[1], \ldots, a[7]$.

Definition 4.1 *A* bus *is a set of nets that are connected to the same modules. The* width *of a bus is the number of nets in the bus.*

In VLSI-CAD tools and hardware description languages (such as VHDL), one often uses indexes to represent buses. Indexing of buses is often a cause of confusion. For example, assume that the terminals on one side of a bus are called $a[0:3]$ and the terminals on the other side of the bus are called $b[3:0]$. Does that mean that $a[0]$ is connected to $b[0]$ or does it mean that $a[0]$ is connected to $b[3]$? Obviously, naming rules are well defined in hardware description languages, but these rules are too strict for our purposes (for example, negative indexes are not allowed, and connections are not implied).

Our convention regarding indexing of terminals and their connection by buses is as follows:

1. Connection of terminals is done by assignment statements. For example, the terminals $a[0:3]$ are connected to the terminals $b[0:3]$ by the statement $b[0:3] \leftarrow a[0:3]$. This statement is meaningful if $a[0:3]$ are output terminals and $b[0:3]$ are input terminals.

2. "Reversing" of indexes does not take place unless explicitly stated. Hence, unless stated otherwise, assignments of buses in which the index ranges are the same or reversed, such as: $b[i:j] \leftarrow a[i:j]$ and $b[i:j] \leftarrow a[j:i]$, simply mean $b[i] \leftarrow a[i], \ldots, b[j] \leftarrow a[j]$.

3. "Shifting" is done by default. For example, will often write $a[0:3] \leftarrow b[4:7]$, meaning that $a[0] \leftarrow b[4], a[1] \leftarrow b[5]$, etc. Similarly, assignments in which the index ranges are shifted, such as: $b[i+5:j+5] \leftarrow a[i:j]$, mean $b[i+5] \leftarrow a[i], \ldots, b[j+5] \leftarrow a[j]$. We refer to such an implied re-assignment of indexes as *hardwired shifting.*

Recall that we denote the (digital) signal on a net N by $N(t)$. This notation is a bit cumbersome in buses, e.g., $a[i](t)$ means the signal on the net $a[i]$. To shorten notation, we will often refer to $a[i](t)$ simply as $a[i]$. Note that $a[i](t)$ is a bit (this is true only after the signal stabilizes). So, according to our shortened notation, we often refer to $a[i]$ as a bit meaning actually "the stable value of the signal $a[i](t)$". This establishes the somewhat confusing convention of referring to buses (e.g., $a[n-1:0]$) as binary strings (e.g., the binary string corresponding to the stable signals $a[n-1:0](t)$).

We will often use an even shorter abbreviation for signals on buses, namely, vector notation. We often use the shorthand $\vec{a}$ for a binary string $a[n-1:0]$ provided, of course, that the indexes of the string $a[n-1:0]$ are obvious from the context.

Consider a gate G with two input terminals a and b and one output terminal z. The combinational circuit $G(n)$ is simply n instances of the gate G, as depicted in part (A) of Figure 4.1. The ith instance of gate G in $G(n)$ is denoted by G_i. The two input terminals of G_i are denoted by a_i and b_i. The output terminal of G_i is denoted by z_i. We use shorthand when drawing the schematics of $G(n)$ as depicted in part (B) of Figure 4.1. The short segment drawn across a wire indicates that the line represents a bus. The bus width is written next to the short segment.

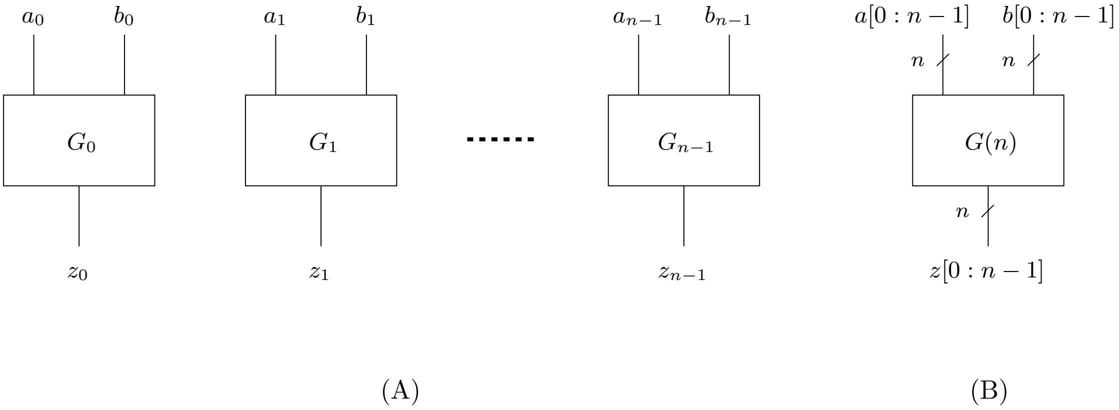


Figure 4.1: Vector notation: multiple instances of the same gate.

We often wish to feed all the second input terminals of gates in $G(n)$ with the same signal. Figure 4.2 denotes a circuit $G(n)$ in which the value b is fed to the second input terminal of all the gates.

Note that the fanout of the net that carries the signal b in Figure 4.2 is n. In practice, the capacity of a net increases linearly with the fanout, hence large fanout increases the amount of time that is needed to stabilize the signal along a net. To keep our delay model simple, we usually ignore this important phenomenon in this course.

The binary string obtained by concatenating the strings a and b is denoted by $a \cdot b$. The binary string obtained by i concatenations of the string a is denoted by a^i.

Example 4.1 *Consider the following examples of string concatenation:*

- *If $a = 01$ and $b = 10$, then $a \cdot b = 0110$.*

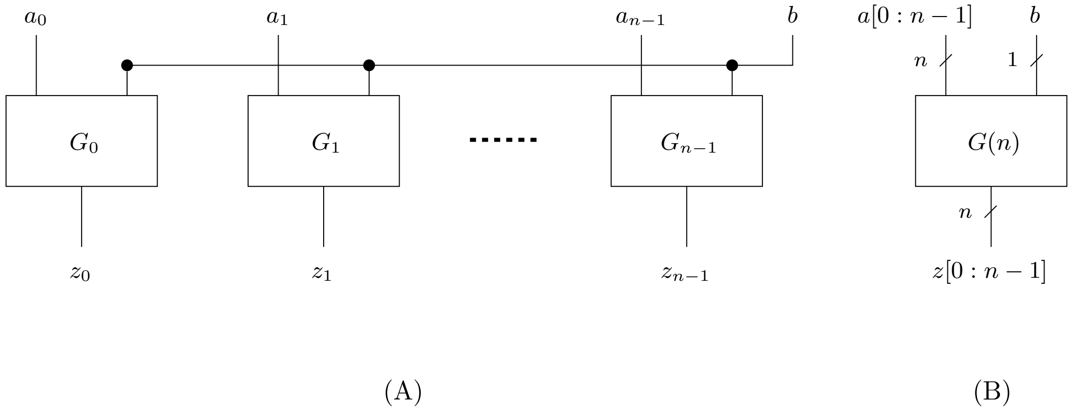


Figure 4.2: Vector notation: b feeds all the gates.

- *If* $a = 1$ *and* $i = 5$, *then* $a^i = 11111$.
- *If* $a = 01$ *and* $i = 3$, *then* $a^i = 010101$.
- *We denote an the zeros string of length* n *by* 0^n *(hopefully, there is no confusion between exponentiation and concatenation of the binary string* 0*).*

4.2 Values represented by binary strings

There are many ways to represent the same value. In binary representation, the number 6 is represented by the binary string 101. The unary representation of the number 6 is 111111. Formal definitions of functionality (i.e., specification) become cumbersome without introducing a simple notation to relate a binary string with the value it represents.

We now define the number that is represented by a binary string.

Definition 4.2 *The* value represented in binary representation *by a binary string* $a[n-1:0]$ *is denoted by* $\langle a[n-1:0]\rangle$. *It is defined as follows*

$$\langle a[n-1:0]\rangle \triangleq \sum_{i=0}^{n-1} a_i \cdot 2^i.$$

One may regard $\langle\cdot\rangle$ as a function from binary strings in $\{0,1\}^n$ to natural numbers in the range $\{0,1,\ldots,2^n-1\}$. We omit the parameter n, since it is not required for defining the value represented in binary representation by a binary string. However, we do need the parameter n in order to define the inverse function, called the binary representation function.

Definition 4.3 Binary representation *using* n*-bits is a function* $\text{bin}_n : \{0,1,\ldots,2^n-1\} \rightarrow \{0,1\}^n$ *that is the inverse function of* $\langle\cdot\rangle$. *Namely, for every* $a[n-1:0] \in \{0,1\}^n$,

$$\text{bin}_n(\langle a[n-1:0]\rangle) = a[n-1:0].$$

One advantage of binary representation is that it is trivial to divide by powers of two as well as compute the remainders. We summarize this property in the following claim.

Claim 4.1 *Let $x[n-1:0] \in \{0,1\}^n$ denote a binary string. Let i denote the number represented by $\vec{x}$ in binary representation, namely, $i = \langle x[n-1:0] \rangle$. Let k denote an index such that $0 \leq k \leq n-1$. Let q and r denote the quotient and remainder, respectively, when dividing i by 2^k. Namely, $i = 2^k \cdot q + r$, where $0 \leq r < 2^k$.*

Define the binary strings $x_R[k-1:0]$ and $x_L[n-1:n-k-1]$ as follows.

$$\begin{aligned} x_R[k-1:0] &\leftarrow x[k-1:0] \\ x_L[n-k-1:0] &\leftarrow x[n-1:k]. \end{aligned}$$

Then,

$$\begin{aligned} q &= \langle x_L[n-k-1:0] \rangle \\ r &= \langle x_R[k-1:0] \rangle. \end{aligned}$$

4.3 Decoders

In this section we present a combinational module called a decoder. We start by defining decoders. We then suggest an implementation, prove its correctness, and analyze its cost and delay. Finally, we prove that the cost and delay of our implementation is asymptotically optimal.

Definition 4.4 *A* decoder with input length n *is a combinational circuit specified as follows:*

Input: $x[n-1:0] \in \{0,1\}^n$.

Output: $y[2^n-1:0] \in \{0,1\}^{2^n}$

Functionality:

$$y[i] = 1 \Longleftrightarrow \langle \vec{x} \rangle = i.$$

Note that the number of outputs of a decoder is exponential in the number of inputs. Note also that exactly one bit of the output $\vec{y}$ is set to one. Such a representation of a number is often termed *one-hot encoding* or 1*-out-of-k encoding.*

We denote a decoder with input length n by DECODER(n).

Example 4.2 *Consider a decoder* DECODER(3)*. On input $x = 101$, the output y equals* 00100000.

4.3.1 Brute force design

The simplest way to design a decoder is to build a separate circuit for every output bit $y[i]$. We now describe a circuit for computing $y[i]$. Let $b[n-1:0]$ denote the binary representation of i (i.e., $b[n-1:0] = bin(i)$).
Let $z_{i,j}$ be defined by $z_{i,j} = 1$ iff $x[j] = b[j]$. An easy way to compute $z_{i,j}$ is as follows:

$$z_{i,j} \triangleq \begin{cases} x[j] & \text{b[j]=1} \\ \text{INV}(x[j]) & \text{b[j]=0.} \end{cases}$$

Note that for every i, the bits of b are known (no need to compute them). Hence we need not XOR or NXOR the bits $x[j]$ and $b[j]$. Instead we simply add terminals that are supposed to be fed by $z_{i,j}$ to the nets fed by $x[j]$ or $\text{INV}(x[j])$ according to the value of $b[j]$.
The following claim implies a simple circuit for computing $y[i]$.

Claim 4.2

$$y[i] = \text{AND}(z_{i,0}, z_{i,1}, \ldots, z_{i,n-1}).$$

Proof: By definition $y[i] = 1$ iff $\langle \vec{x} \rangle = i$. Now $\langle \vec{x} \rangle = i$ iff $\vec{x} = \vec{b}$. We compare $\vec{x}$ and $\vec{b}$ by requiring that $x[j] = 1$ if $b[j] = 1$ and $\text{INV}(x[j]) = 1$ if $b[j] = 0$. □

The brute force decoder circuit consists of (i) n inverters used to compute $\text{INV}(\vec{x})$, and (ii) an $\text{AND}(n)$-tree for every output $y[i]$. The delay of the brute force design is $t_{pd}(\text{INV}) + t_{pd}(\text{AND}(n)\text{-tree})$. The cost of the brute force design is $\Theta(n \cdot 2^n)$, since we have an $\text{AND}(n)$-tree for each of the 2^n outputs.

Intuitively, the brute force design is wasteful because, if the binary representation of i and j differ in a single bit, then the corresponding AND-trees share all but a single input. Hence the AND of $n-1$ bits is computed twice. In the next section we present a systematic way to share hardware between different outputs.

4.3.2 An optimal decoder design

We design a $\text{DECODER}(n)$ using recursion on n. We start with the trivial task of designing a $\text{DECODER}(n)$ with $n = 1$. We then proceed by designing a $\text{DECODER}(n)$ based on "smaller" decoders.

$\text{DECODER}(1)$: The circuit $\text{DECODER}(1)$ is simply one inverter where: $y[0] \leftarrow \text{INV}(x[0])$ and $y[1] \leftarrow x[0]$.

$\text{DECODER}(n)$: We assume that we know how to design decoders with input length less than n, and design a decoder with input length n.

The method we apply for our design is called "divide-and-conquer". Consider a parameter k, where $0 < k < n$. We partition the input string $x[n-1:0]$ into two strings as follows:

1. The right part (or lower part) is $x_R[k-1:0]$ and is defined by $x_R[k-1:0] = x[k-1:0]$.

2. The left part (or upper part) is $x_L[n-k-1:0]$ and is defined by $x_L[n-k-1:0] = x[n-1:k]$. (Note that hardwired shift is applied in this definition, namely, $x_L[0] \leftarrow x[k], \ldots, x_L[n-k-1] \leftarrow x[n-1]$.)

We will later show that, to reduce delay, it is best to choose k as close to $n/2$ as possible. However, at this point we consider k to be an arbitrary parameter such that $0 < k < n$.

Figure 4.3 depicts a recursive implementation of an DECODER(n). Our recursive design feeds $x_L[n-k-1:0]$ to DECODER($n-k$). We denote the output of the decoder DECODER($n-k$) by $Q[2^{n-k}-1:0]$. (The letter 'Q' stands "quotient".) In a similar manner, our recursive design feeds $x_R[k-1:0]$ to DECODER(k). We denote the output of the decoder DECODER(k) by $R[2^k-1:0]$. (The letter 'R' stands for "remainder".)

The decoder outputs $Q[2^{n-k}-1:0]$ and $R[2^k-1:0]$ are fed to a $2^{n-k} \times 2^k$ array of AND-gates. We denote the AND-gate in position (q,r) in the array by $\text{AND}_{q,r}$. The rules for connecting the AND-gates in the array are as follows. The inputs of the gate $\text{AND}_{q,r}$ are $Q[q]$ and $R[r]$. The output of the gate $\text{AND}_{q,r}$ is $y[q \cdot 2^k + r]$.

Note that we have defined a routing rule for connecting the outputs $Q[2^{n-k}-1:0]$ and $R[2^k-1:0]$ to the inputs of the AND-gates in the array. This routing rule (that involves division with remainder by 2^k) is not computed by the circuit; the routing rule defines the circuit and must be followed by the person implementing the design.

In Figure 4.3, we do not draw the connections in the array of AND-gates. Instead, connections are inferred by the names of the wires (e.g., two wires called $R[5]$ belong to the same net).

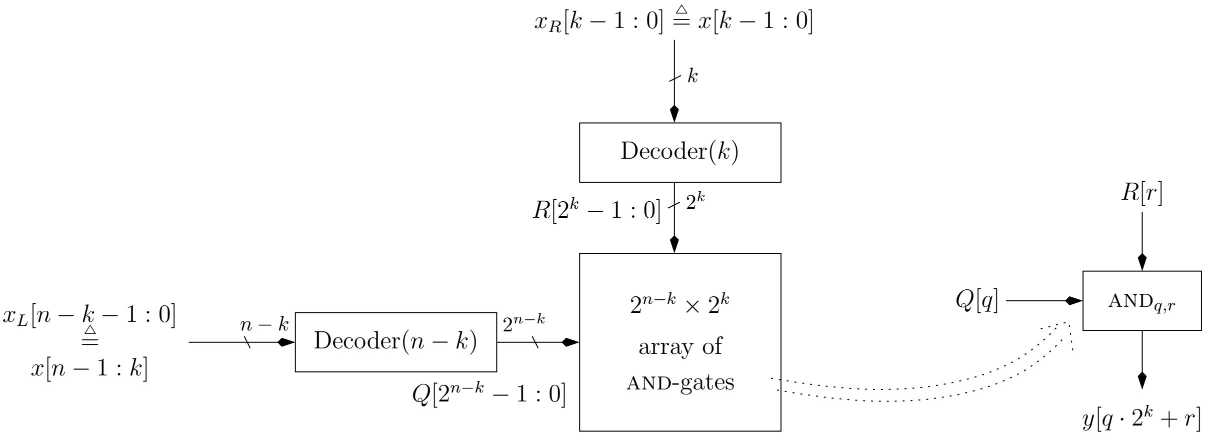


Figure 4.3: A recursive implementation of DECODER(n).

4.3.3 Correctness

In this section we prove the correctness of the DECODER(n) design.

Claim 4.3 *The* DECODER(n) *design is a correct implementation of a decoder.*

Proof: Our goal is to prove that, for every n and every $0 \leq i < 2^n$, the following holds:

$$y[i] = 1 \quad \Longleftrightarrow \quad \langle x[n-1:0] \rangle = i.$$

The proof is by induction on n. The induction basis, for $n = 1$, is trivial. We proceed directly to the induction step. Fix an index i and divide i by 2^k to obtain $i = q \cdot 2^k + r$, where $r \in [2^k - 1 : 0]$.

We apply the induction hypothesis to DECODER(k) to conclude that $R[r] = 1$ iff $\langle x_R[k-1:0] \rangle = r$. Similarly, the induction hypothesis when applied to DECODER$(n-k)$ implies that $Q[q] = 1$ iff $\langle x_L[n-k-1:0] \rangle = q$. Since $i = q \cdot 2^k + r$, this implies that

$$\begin{aligned} y[i] = 1 &\Longleftrightarrow R[r] = 1 \text{ and } Q[q] = 1 \\ &\Longleftrightarrow \langle x_R[k-1:0] \rangle = r \text{ and } \langle x_L[n-k-1:0] \rangle = q. \\ &\Longleftrightarrow \langle x[n-1:0] \rangle = i, \end{aligned}$$

and the claim follows. □

4.3.4 Cost and delay analysis

In this section we analyze the cost and delay of the DECODER(n) design. We denote the cost and delay of DECODER(n) by $c(n)$ and $d(n)$, respectively.
The cost $c(n)$ satisfies the following recurrence equation:

$$c(n) = \begin{cases} c(\text{INV}) & \text{if n=1} \\ c(k) + c(n-k) + 2^n \cdot c(\text{AND}) & \text{otherwise.} \end{cases}$$

It follows that

$$c(n) = c(k) + c(n-k) + \Theta(2^n)$$

Obviously, $c(n) = \Omega(2^n)$ (regardless of the value of k), so the best we can hope for is to find a value of k such that $c(n) = O(2^n)$. In fact, it can be shown that $c(n) = O(2^n)$, for every choice of $1 \leq k < n$. We show below that $c(n) = O(2^n)$ if $k = n/2$. (If n is odd, then we choose $k = \lfloor n/2 \rfloor$.) To simplify, we assume that n is a power of 2, namely, $n = 2^\ell$. If $k = n/2$ we open the recurrence to get:

$$\begin{aligned} c(n) &= 2 \cdot c(n/2) + \Theta(2^n) \\ &= 4 \cdot c(n/4) + \Theta(2^n + 2 \cdot 2^{n/2}) \\ &= 8 \cdot c(n/8) + \Theta(2^n + 2 \cdot 2^{n/2} + 4 \cdot 2^{n/4}) \\ &= n \cdot c(1) + \Theta(2^n + 2 \cdot 2^{n/2} + 4 \cdot 2^{n/4} + \cdots + n \cdot 2^{n/n}) \\ &= \Theta(2^n). \end{aligned}$$

The last line is justified by

$$\begin{aligned} 2^n + 2 \cdot 2^{n/2} + 4 \cdot 2^{n/4} + \cdots + n \cdot 2^{n/n} &\leq 2^n + 2(\log_2 n) \cdot 2^{n/2} \\ &\leq 2^n \left(1 + \frac{2 \log_2 n}{2^{n/2}}\right) \\ &= 2^n (1 + o(1)). \end{aligned}$$

The delay of $\textsc{decoder}(n)$ satisfies the following recurrence equation:

$$d(n) = \begin{cases} d(\textsc{inv}) & \text{if n=1} \\ \max\{d(k), d(n-k)\} + d(\textsc{and}) & \text{otherwise.} \end{cases}$$

Set $k = n/2$, and it follows that $d(n) = \Theta(\log n)$.

Question 4.1 *Prove that $c(n) = O(2^n)$, for every choice of $1 \leq k < n$.*

Question 4.2 *Prove that* $\textsc{decoder}(n)$ *is asymptotically optimal with respect to cost and delay. Namely, prove that, for every decoder G of input length n, the following hold:*

$$\begin{aligned} c(G) &\geq \Omega(2^n) \\ d(G) &\geq \Omega(\log n). \end{aligned}$$

4.4 Encoders

An encoder implements the inverse Boolean function of a decoder. Note however, that the Boolean function implemented by a decoder is not surjective. In fact, the range of the decoder function is the set of binary vectors in which exactly one bit equals 1. It follows that an encoder implements a partial Boolean function (i.e., a function that is not defined for every binary string).

We first define the (Hamming) weight of binary strings.

Definition 4.5 *The weight of a binary string equals the number of non-zero symbols in the string. We denote the weight of a binary string $\vec{a}$ by $\mathrm{wt}(\vec{a})$.Formally,*

$$\mathrm{wt}(a[n-1:0]) \triangleq |\{i : a[i] \neq 0\}|.$$

We define the encoder partial function as follows.

Definition 4.6 *The function $\textsc{encoder}_n : \{\vec{y} \in \{0,1\}^{2^n} : \mathrm{wt}(\vec{y}) = 1\} \rightarrow \{0,1\}^n$ is defined as follows: $\langle \textsc{encoder}_n(\vec{y}) \rangle$ equals the index of the bit of $\vec{y}$ that equals one. Formally,*

$$\mathrm{wt}(y) = 1 \Longrightarrow y[\langle \textsc{encoder}_n(\vec{y}) \rangle] = 1.$$

Definition 4.7 *An* encoder *with input length 2^n and output length n is a combinational circuit that implements the Boolean function $\textsc{encoder}_n$.*

An encoder can be also specified as follows:

Input: $y[2^n - 1 : 0] \in \{0,1\}^{2^n}$.

Output: $x[n-1:0] \in \{0,1\}^n$.

Functionality: If $wt(\vec{y}) = 1$, let i denote the index such that $y[i] = 1$. In this case $\vec{x}$ should satisfy $\langle \vec{x} \rangle = i$. Formally:

$$wt(\vec{y}) = 1 \implies y[\langle \vec{x} \rangle] = 1.$$

If $wt(\vec{y}) \neq 1$, then the output $\vec{x}$ is arbitrary.

Note that the functionality is not uniquely defined for all inputs $\vec{y}$. However, if $\vec{y}$ is output by a decoder, then $wt(\vec{y}) = 1$, and hence an encoder implements the inverse function of a decoder. We denote an encoder with input length 2^n and output length n by ENCODER(n).

Example 4.3 *Consider an encoder* ENCODER(3)*. On input* 00100000*, the output equals* 101*.*

4.4.1 Implementation

In this section we present a step by step implementation of an encoder. We start with a rather costly design, which we denote by ENCODER$'(n)$. We then show how to modify ENCODER$'(n)$ to an asymptotically optimal one.

As in the design of a decoder, our design is recursive. The design for $n = 1$, is simply $x[0] \leftarrow y[1]$. Hence, for $n = 1$, the cost and delay of our design are zero. We proceed with the design for $n > 1$.

Again, we use the divide-and-conquer method. We partition the input $\vec{y}$ into two strings of equal length as follows:

$$y_L[2^{n-1} - 1 : 0] = y[2^n - 1 : 2^{n-1}] \qquad y_R[2^{n-1} - 1 : 0] = y[2^{n-1} - 1 : 0].$$

The idea is to feed these two parts into two encoders ENCODER$'(n-1)$ (see Figure 4.4). However, there is a problem with this approach. The problem is that even if $\vec{y}$ is a "legal" input (namely, $wt(\vec{y}) = 1$), then one of the strings $\vec{y}_L$ or $\vec{y}_R$ is all zeros, which is not a legal input. An "illegal" input can produce an arbitrary output, which might make the design wrong.

To fix this problem we augment the definition of the ENCODER$_n$ function so that its range also includes the all zeros string 0^{2^n}. We define

$$\text{ENCODER}_n(0^{2^n}) \triangleq 0^n.$$

Note that ENCODER$'(1)$ also meets this new condition, so the induction basis of the correctness proof holds.

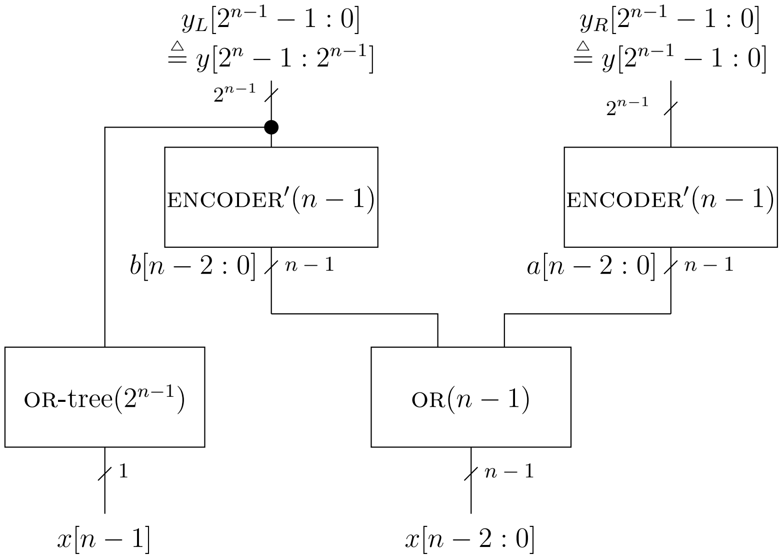


Figure 4.4: A recursive implementation of ENCODER$'(n)$.

Let $a[n-2:0]$ (resp., $b[n-2:0]$) denote the output of the ENCODER$'(n-1)$ circuit that is fed by $\vec{y}_R$ (resp., $\vec{y}_L$).

Having fixed the problem caused by inputs that are all zeros, we proceed with the "conquer" step. We distinguish between three cases, depending on which half contains the bit that is lit in $\vec{y}$, if any.

1. If $wt(\vec{y}_L)=0$ and $wt(\vec{y}_R)=1$, then the induction hypothesis implies that $\vec{b}=0^{n-1}$ and $y_R[\langle\vec{a}\rangle]=1$. It follows that $y[\langle\vec{a}\rangle]=1$, hence the required output is $\vec{x}=0\cdot\vec{a}$. The actual output equals the required output, and correctness holds in this case.

2. If $wt(\vec{y}_L)=1$ and $wt(\vec{y}_R)=0$, then the induction hypothesis implies that $y_L[\langle\vec{b}\rangle]=1$ and $\vec{a}=0^{n-1}$. It follows that $y[2^{n-1}+\langle\vec{b}\rangle]=1$, hence the required output is $\vec{x}=1\cdot\vec{b}$. The actual output equals the required output, and correctness holds in this case.

3. If $wt(\langle\vec{y}\rangle)=0$, then the required output is $\vec{x}=0^n$. The induction hypothesis implies that $\vec{a}=\vec{b}=0^{n-1}$. The actual output is $\vec{x}=0^n$, and correctness follows.

We conclude that the design ENCODER$'(n)$ is correct.

Claim 4.4 *The circuit* ENCODER$'(n)$ *depicted in Figure 4.4 implements the Boolean function* ENCODER$_n$.

The problem with the ENCODER$'(n)$ design is that it is too costly. We summarize the cost of ENCODER$'(n)$ in the following question.

Question 4.3 *This question deals with the cost and delay of* ENCODER$'(n)$.

1. *Prove that* $c(\text{ENCODER}'(n))=\Theta(n\cdot 2^n)$.

2. *Prove that* $d(\text{ENCODER}'(n)) = \Theta(n)$.

3. *Can you suggest a separate circuit for every output bit* $x[i]$ *with cost* $O(2^n)$ *and delay* $O(n)$*? If so then what advantage does the* ENCODER$'(n)$ *design have over the trivial design in which every output bit is computed by a separate circuit?*

Question 4.4 (hard) *An* ENCODER$'(n)$ *contains an* OR*-tree*(2^n) *and an* ENCODER$'(n-1)$ *that are both fed by* $y_L[2^{n-1}-1:0]$*. This implies that if we "open the recursion" we will have a chain of* OR*-trees, where small trees are sub-trees of larger trees. This means that an* ENCODER$'(n)$ *contains redundant duplications of* OR*-trees. Analyze the reduction in cost that one could obtain if duplicate* OR*-trees are avoided. Does this reduction change the asymptotic cost?*

Question 4.3 suggests that the ENCODER$'(n)$ design is not better than a brute force design. Can we do better? The following claim serves as a basis for reducing the cost of an encoder.

Claim 4.5 *If* $\text{wt}(y[2^n-1:0]) \leq 1$*, then*

$$\text{ENCODER}_{n-1}(\text{OR}(\vec{y}_L, \vec{y}_R)) = \text{OR}(\text{ENCODER}_{n-1}(\vec{y}_L), \text{ENCODER}_{n-1}(\vec{y}_R)).$$

Proof: The proof in case $\vec{y} = 0^{2^n}$ is trivial. We prove the case that $wt(\vec{y}_L) = 0$ and $wt(\vec{y}_R) = 1$ (the proof of other case is analogous). Hence,

$$\begin{aligned}\text{ENCODER}_{n-1}(\text{OR}(\vec{y}_L, \vec{y}_R)) &= \text{ENCODER}_{n-1}(\text{OR}(0^{2^{n-1}}, \vec{y}_R)) \\ &= \text{ENCODER}_{n-1}(\vec{y}_R)).\end{aligned}$$

However,

$$\begin{aligned}\text{OR}(\text{ENCODER}_{n-1}(\vec{y}_L), \text{ENCODER}_{n-1}(\vec{y}_R)) &= \text{OR}(\text{ENCODER}_{n-1}(0^{2^{n-1}}), \text{ENCODER}_{n-1}(\vec{y}_R)) \\ &= \text{OR}(0^{n-1}, \text{ENCODER}_{n-1}(\vec{y}_R)) \\ &= \text{ENCODER}_{n-1}(\vec{y}_R),\end{aligned}$$

and the claim follows. □

Figure 4.5 depicts the design ENCODER$^*(n)$ obtained from ENCODER$'(n)$ after commuting the OR and the ENCODER$(n-1)$ operations. We do not need to prove the correctness of the ENCODER$^*(n)$ circuit "from the beginning". Instead we rely on the correctness of ENCODER$'(n)$ and on Claim 4.5 that shows that ENCODER$'(n)$ and ENCODER$^*(n)$ are functionally equivalent.

Question 4.5 *Provide a direct correctness proof for the* ENCODER$^*(n)$ *design (i.e., do not rely on the correctness of* ENCODER$'(n)$*). Does the correctness of* ENCODER$^*(n)$ *require that* ENCODER$^*(n-1)$ *output an all-zeros string when the input is an all-zeros string?*

The following questions are based on the following definitions:

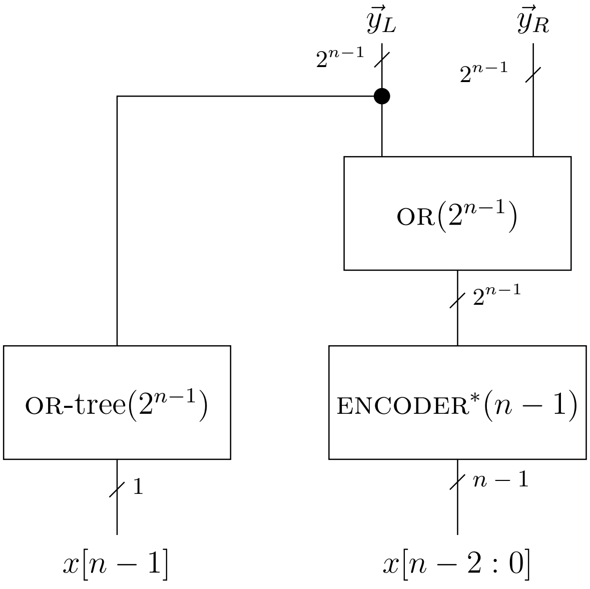


Figure 4.5: A recursive implementation of ENCODER$^*(n)$.

Definition 4.8 *A binary string* $x'[n-1:0]$ dominates *the binary string* $x''[n-1:0]$ *if*

$$\forall i \in [n-1:0]: \quad x''[i]=1 \Rightarrow x'[i]=1.$$

Definition 4.9 *A Boolean function f is* monotone *if x' dominates x'' implies that $f(x')$ dominates $f(x'')$.*

Question 4.6 *Prove that if a combinational circuit C contains only gates that implement monotone Boolean functions (e.g., only* AND*-gates and* OR*-gates, no inverters), then C implements a monotone Boolean function.*

Question 4.7 *The designs* ENCODER$'(n)$ *and* ENCODER$^*(n)$ *lack inverters, and hence are monotone circuits. Is the Boolean function* ENCODER$_n$ *a monotone Boolean function? Suppose that G is an encoder and is a monotone combinational circuit. Suppose that the input y of G has two ones (namely,* $\text{wt}(y)=2$*). Can you immediately deduce which outputs of G must equal one?*

4.4.2 Cost and delay analysis

The cost of ENCODER$^*(n)$ satisfies the following recurrence equation:

$$c(\textsc{encoder}^*(n)) = \begin{cases} 0 & \text{if n=1} \\ c(\textsc{encoder}^*(n-1)) + 2^n \cdot c(\textsc{or}) & \text{otherwise.} \end{cases}$$

We expand this recurrence to obtain:

$$\begin{aligned} c(\textsc{encoder}^*(n)) &= c(\textsc{encoder}^*(n-1)) + 2^n \cdot c(\textsc{or}) \\ &= (2^n + 2^{n-1} + \ldots + 4) \cdot c(\textsc{or}) \\ &= (2 \cdot 2^n - 3) \cdot c(\textsc{or})) \\ &= \Theta(2^n). \end{aligned}$$

The delay of $\textsc{encoder}^*(n)$ satisfies the following recurrence equation:

$$d(\textsc{encoder}^*(n)) = \begin{cases} 0 & \text{if n=1} \\ \max\{d(\textsc{or}\text{-tree}(2^{n-1}), d(\textsc{encoder}^*(n-1) + d(\textsc{or}))\} & \text{otherwise.} \end{cases}$$

Since $d(\textsc{or}\text{-tree}(2^{n-1}) = (n-1) \cdot d(\textsc{or})$, it follows that

$$d(\textsc{encoder}^*(n)) = \Theta(n).$$

The following question deals with lower bounds for an encoder.

Question 4.8 *Prove that the cost and delay of* $\textsc{encoder}^*(n)$ *are asymptotically optimal.*

4.4.3 Yet another encoder

In this section we present another encoder design. We denote this design by $\textsc{encoder}''(n)$. The design is a variation of the $\textsc{encoder}'(n)$ design and saves hardware by exponentially reducing the cost of the $\textsc{or}$-tree. Figure 4.6 depicts a recursive implementation of the $\textsc{encoder}''(n)$ design.

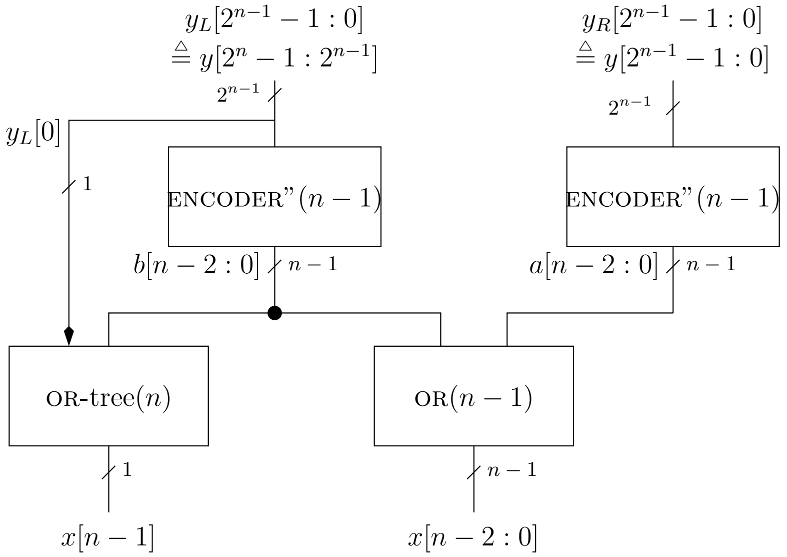


Figure 4.6: A recursive implementation of $\textsc{encoder}''(n)$.

Question 4.9 *Prove the correctness of the* ENCODER$''(n)$ *design. Hint: the idea behind this design is to check if* $\vec{y_L} = 0^{2^{n-1}}$ *by checking if* $b[n-2:0] = 0^{n-1}$ *and* $y_L[0] = 0$. *Note that* $b[n-2:0] = 0^{n-1}$ *implies that* $y_L[2^{n-1}-1:1] = 0^{2^{n-1}-1}$. *So we only need to check if* $y_L[0] = 0$.

The advantage of the ENCODER$''(n)$ design is that it has an optimal asymptotic cost. In particular its cost satisfies the recurrence:

$$c(\text{ENCODER}''(n)) = \begin{cases} 0 & \text{if n=1} \\ 2 \cdot c(\text{ENCODER}''(n-1)) + 2 \cdot (n-1) \cdot c(\text{OR}) & \text{otherwise.} \end{cases} \tag{4.1}$$

It follows that

$$\begin{aligned} c(\text{ENCODER}''(n)) &= c(\text{OR}) \cdot (2 \cdot 2^{n-2} + 4 \cdot 2^{n-3} + \cdots + 2 \cdot (i-1) \cdot 2^{n-i} + \cdots + 2 \cdot (n-1)) \\ &= c(\text{OR}) \cdot 2^n \cdot \left(\frac{1}{2} + \frac{2}{4} + \cdots + \frac{i-1}{2^{(i-1)}} + \cdots + \frac{n-1}{2^{(n-1)}} \right) \\ &\leq c(\text{OR}) \cdot 2^n \cdot 2. \end{aligned}$$

The disadvantage of the ENCODER$''(n)$ design is that is it slow.

Question 4.10 1. *Write the recurrence equation for the delay of the* ENCODER$''(n)$ *design.*

2. *Prove that* $d(\text{ENCODER}''(n)) \leq n \cdot \log_2 n$.

3. *Prove that* $d(\text{ENCODER}''(n)) \geq \ln 2 \cdot (n-1) \cdot (\ln(n-1) - 1) + \ln 2$. *Hint:* $\sum_{i=2}^{n-1} \ln i \geq \int_1^{n-1} (\ln x) dx$. *(Any other* $\Omega(n \log n)$ *lower bound is fine too).*

4.5 Summary

In this chapter, we introduced bus notation that is used to denote indexed signals (e.g., $a[n-1:0]$). We also defined binary representation. We then presented decoder and encoder designs using divide-and-conquer.

The first combinational circuit we described is a decoder. A decoder can be viewed as a circuit that translates a number represented in binary representation to a 1-out-of-2^n encoding. We started by presenting a brute force design in which a separate AND-tree is used for each output bit. The brute force design is simple yet wasteful. We then presented a recursive decoder design with asymptotically optimal cost and delay.

There are many advantages in using recursion. First, we were able to formally define the circuit. The other option would have been to draw small cases (say, $n = 3, 4$) and then argue informally that the circuit is built in a similar fashion for larger values of n. Second, having recursively defined the design, we were able to prove its correctness using induction. Third, writing the recurrence equations for cost and delay is easy. We proved that our decoder design is asymptotically optimal both in cost and in delay.

The second combinational circuit we described is an encoder. An encoder is the inverse circuit of a decoder. We presented a naive design and proved its correctness. We then reduced the cost of the naive design by commuting the order of two operations without changing the functionality. We proved that the final encoder design has asymptotically optimal cost and delay.

Three main techniques were used in this chapter.

- Divide & Conquer. We solve a problem by dividing it into smaller sub-problems. The solutions of the smaller sub-problems are "glued" together to solve the big problem.

- Extend specification to make problem easier. We encountered a difficulty in the encoder design due to an all zeros input. We bypassed this problem by extending the specification of an encoder so that it must output all zeros when input an all zeros. Adding restrictions to the specification made the task easier since we were able to employ these restrictions with smaller encoders in our recursive construction.

- Evolution. We started with a naive and correct design. This design turned out to be too costly. We improved the naive design while preserving its functionality to obtain a cheaper design. The correctness of the improved design follows from the correctness of the naive design and the fact that it is functionally equivalent to the naive design.

Chapter 5

Selectors and Shifters

Contents

Preliminary questions:

1. Which types of shifts are you familiar with in your favorite programming language? What is the differences between these shifts? Why do we need different types of shifts?

2. How are these shifts executed in a microprocessor?

3. Should shifters be considered to be combinational circuits? After all, they simply "move bits around" and do not compute "new bits".

In this chapter we present combinational circuits that manipulate the input bits. These manipulations are referred to as *shifts*. There are different types of shifts: cyclic shifts, logical shifts, and arithmetic shifts. We present shifter designs that are based on multiplexers.

5.1 Multiplexers

In this section we present designs of $(n : 1)$-multiplexers. Multiplexers are often also called *selectors*.
We first review the definition of a MUX-gate (also known as a $(2 : 1)$-multiplexer).

Definition 5.1 *A* MUX*-gate is a combinational gate that has three inputs* $D[0], D[1], S$ *and one output* Y*. The functionality is defined by*

$$Y = \begin{cases} D[0] & \text{if } S = 0 \\ D[1] & \text{if } S = 1. \end{cases}$$

Note that we could have used the shorter expression $Y = D[S]$ to define the functionality of a MUX-gate.
An (n:1)-MUX is a combinational circuit defined as follows:

Input: $D[n-1:0]$ and $S[k-1:0]$ where $k = \lceil \log_2 n \rceil$.

Output: $Y \in \{0, 1\}$.

Functionality:

$$Y = D[\langle \vec{S} \rangle].$$

We often refer to $\vec{D}$ as the *data input* and to $\vec{S}$ as the *select input*. The select input $\vec{S}$ encodes the index of the bit of the data input $\vec{D}$ that should be output. To simplify the discussion, we will assume in this section that n is a power of 2, namely, $n = 2^k$.

Example 5.1 *Let* $n = 4$*,* $D[3:0] = 0101$*, and* $S[1:0] = 11$*. The output* Y *should be* 0*.*

5.1.1 Implementation

We describe two implementations of (n:1)-MUX. The first implementation is based on translating the number $\langle \vec{S} \rangle$ to 1-out-of-n representation (using a decoder). The second implementation is basically a tree.

A decoder based implementation. Figure 5.1 depicts an implementation of a (n:1)-MUX based on a decoder. The input $S[k-1:0]$ is fed to a DECODER(k). The decoder outputs a 1-out-of-n representation of $\langle \vec{S} \rangle$. Bitwise-AND is applied to the output of the decoder and the input $D[n-1:0]$. The output of the bitwise-AND is then fed to an OR-tree to produce Y.

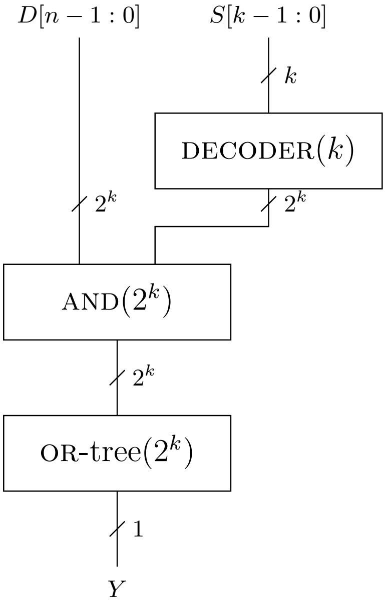


Figure 5.1: An (n:1)-MUX based on a decoder ($n = 2^k$).

Question 5.1 *The following question deals with the implementation of (n:1)-*MUX *suggested in Figure 5.1.*

1. *Prove the correctness of the design.*
2. *Analyze the cost and delay of the design.*
3. *Prove that the cost and delay of the design are asymptotically optimal.*

A tree-like implementation. A second implementation of (n:1)-MUX is a recursive tree-like implementation. The design for $n = 2$ is simply a MUX-gate. The design for $n = 2^k$ is depicted in Figure 5.2. The input $\vec{D}$ is divided into two parts of equal length. Each part is fed to an $(\frac{n}{2} : 1)$-MUX controlled by the signal $S[k-2:0]$. The outputs of the $(\frac{n}{2} : 1)$-MUXs are Y_L and Y_R. Finally a MUX selects between Y_L and Y_R according to the value of $S[k-1]$.

Question 5.2 *Answer the same questions asked in Question 5.1 but this time with respect to the implementation of the (n:1)-*MUX *suggested in Figure 5.2.*

Both implementations suggested in this section are asymptotically optimal with respect to cost and delay. Which design is better? A cost and delay analysis based on the cost and delay of gates listed in Table 2.1 suggests that the tree-like implementation is cheaper and faster. Nevertheless, our model is not refined enough to answer this question sharply. On one hand, the tree-like design is simply a tree of multiplexers. The decoder based design contains, in addition to an OR(n)-tree with n inputs, also a line of AND-gates and a decoder. So one may conclude that the decoder based design is worse.

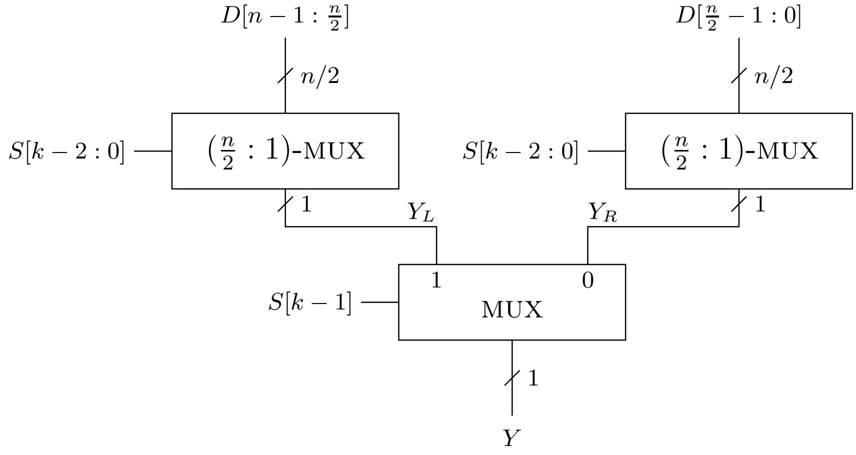


Figure 5.2: A recursive implementation of (n:1)-MUX ($n = 2^k$).

On the other hand, OR-gates are typically cheaper and faster than MUX-gates. Moreover, fast and cheap implementations of MUX-gates in CMOS technology do not restore the signals well; this means that long paths consisting only of MUX-gates are not allowed. We conclude that the model we use cannot be used to deduce conclusively which multiplexer design is better.

Question 5.3 *Compute the cost and delay of both implementations of (n:1)-*MUX *based on the data in Table 2.1 for various values of n (e.g., $n = 4, 8, 16, 32$).*

5.2 Cyclic Shifters

We explain what a cyclic shift is by the following example. Consider a binary string $a[1:12]$ and assume that we place the bits of a on a wheel. The position of $a[1]$ is at one o'clock, the position of $a[2]$ is at two o'clock, etc. We now rotate the wheel, and read the bits in clockwise order starting from one o'clock and ending at twelve o'clock. The resulting string is a cyclic shift of $a[1:12]$. Figure 5.2 depicts an example of a cyclic shift.

Definition 5.2 *The string $b[n-1:0]$ is a* cyclic left shift by i positions *of the string $a[n-1:0]$ if*

$$\forall j: \quad b[j] = a[\text{mod}(j-i, n)].$$

Example 5.2 *Let $a[3:0] = 0010$. A cyclic left shift by one position of $\vec{a}$ is the string 0100. A cyclic left shift by 3 positions of $\vec{a}$ is the string 0001.*

Definition 5.3 *A* BARREL-SHIFTER(n) *is a combinational circuit defined as follows:*

Input: *$x[n-1:0] \in \{0,1\}^n$ and $sa[k-1:0] \in \{0,1\}^k$ where $k = \lceil \log_2 n \rceil$.*

Output: *$y[n-1:0] \in \{0,1\}^n$.*

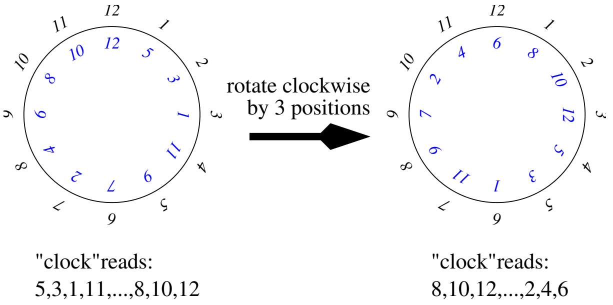


Figure 5.3: An example of a cyclic shift. The clock "reads" the numbers stored in each clock notch in clockwise order starting from the one o'clock notch.

Functionality: *$\vec{y}$ is a cyclic left shift of $\vec{x}$ by $\langle \vec{sa} \rangle$ positions. Formally,*

$$\forall j \in [n-1:0]: \quad y[j] = x[\text{mod}(j - \langle \vec{sa} \rangle, n)].$$

We often refer to the input $\vec{x}$ as the *data input* and to the input $\vec{sa}$ as the *shift amount input*. To simplify the discussion, we will assume in this section that n is a power of 2, namely, $n = 2^k$.

5.2.1 Implementation

We break the task of designing a barrel shifter into smaller sub-tasks of shifting by powers of two. We define this sub-task formally as follows.
A $\text{CLS}(n, i)$ is a combinational circuit that implements a cyclic left shift by zero or 2^i positions depending on the value of its select input.

Definition 5.4 *A $\text{CLS}(n, i)$ is a combinational circuit defined as follows:*

Input: *$x[n-1:0]$ and $s \in \{0, 1\}$.*

Output: *$y[n-1:0]$.*

Functionality:

$$\forall j \in [n-1:0]: \quad y[j] = x[\text{mod}(j - s \cdot 2^i, n)].$$

A $\text{CLS}(n, i)$ is quite simple to implement since $y[j]$ is either $x[j]$ or $x[\text{mod}(j - 2^i, n)]$. So all one needs is a MUX-gate to select between $x[j]$ or $x[\text{mod}(j - 2^i, n)]$. The selection is based on the value of s. It follows that the delay of $\text{CLS}(n, i)$ is the delay of a MUX, and the cost is n times the cost of a MUX. Figure 5.4 depicts an implementation of a $\text{CLS}(4, 1)$. It is self-evident that the main complication with the design of $\text{CLS}(n.i)$ is routing (i.e., drawing the wires).

The following claim shows how to design a barrel shifter using $\text{CLS}(n, i)$ circuits. In the following claim we refer to $\text{CLS}_{n,i}$ as the Boolean function that is implemented by a $\text{CLS}(n, i)$ circuit.

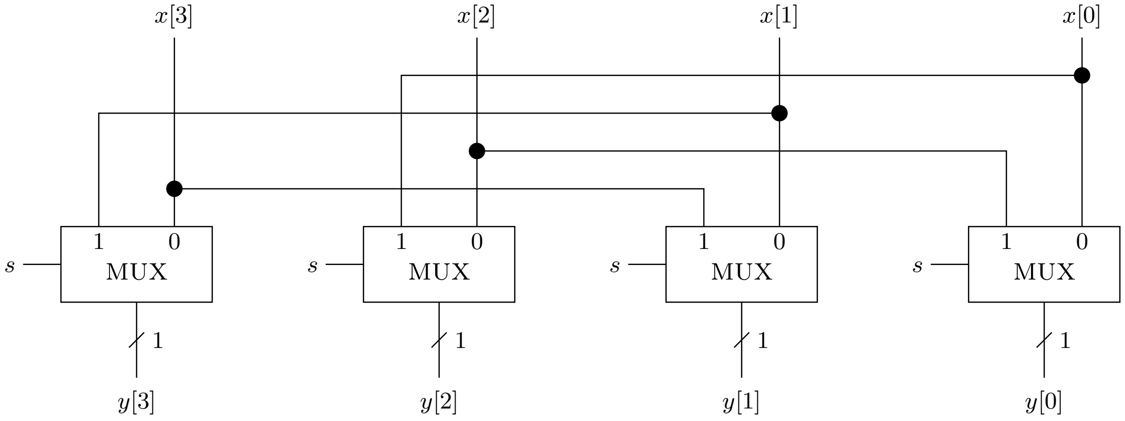


Figure 5.4: A row of multiplexers implement a $\textsc{cls}(4, 1)$.

Claim 5.1 *Define the strings $y_i[n-1:0]$, for $0 \leq i \leq k-1$, recursively as follows:*

$$y_0[n-1:0] \leftarrow \textsc{cls}_{n,0}(x[n-1,0], sa[0])$$
$$y_{i+1}[n-1:0] \leftarrow \textsc{cls}_{n,i+1}(y_i[n-1,0], sa[i+1])$$

The string $y_i[n-1:0]$ is a cyclic left shift of the string $x[n-1:0]$ by $\langle sa[i:0]\rangle$ positions.

Proof: The proof is by induction on i. The induction basis, for $i = 0$, holds because of the definition of $\textsc{cls}(2, 0)$.

The induction step is proved as follows.

$$\begin{aligned} y_i[j] &= \textsc{cls}_{n,i}(y_{i-1}[n-1,0], sa[i])[j] && \text{(by definition of } y_i) \\ &= y_{i-1}[\text{mod}(j - 2^i \cdot sa[i], n)] && \text{(by definition of } \textsc{cls}_{n,i}). \end{aligned}$$

Let $\ell = \text{mod}(j - 2^i \cdot sa[i], n)$. The induction hypothesis implies that

$$y_{i-1}[\ell] = x[\text{mod}(\ell - \langle sa[i-1:0]\rangle, n)].$$

Note that

$$\begin{aligned} \text{mod}(\ell - \langle sa[i-1:0]\rangle, n) &= \text{mod}(j - 2^i \cdot sa[i] - \langle sa[i-1:0]\rangle, n) \\ &= \text{mod}(j - \langle sa[i:0]\rangle, n). \end{aligned}$$

Therefore

$$y_i[j] = x[\text{mod}(j - \langle sa[i:0]\rangle, n)],$$

and the claim follows. □

Having designed a $\textsc{cls}(n, i)$ we are ready to implement a $\textsc{barrel-shifter}(n)$. Figure 5.5 depicts an implementation of a $\textsc{barrel-shifter}(n)$. The implementation is based on k levels of $\textsc{cls}(n, i)$, for $i \in [k-1:0]$, where the ith level is controlled by $sa[i]$.

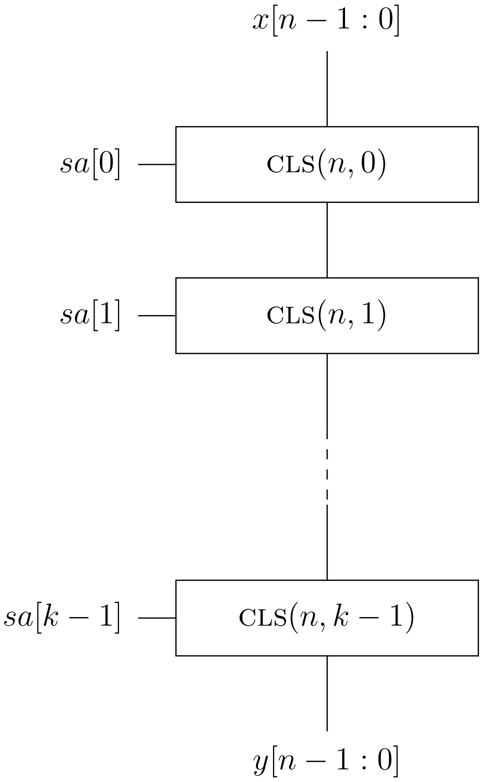


Figure 5.5: A BARREL-SHIFTER(n) built of k levels of CLS(n,i) $(n=2^k)$.

Question 5.4 *This question deals with the design of the* BARREL-SHIFTER(n) *depicted in Figure 5.5.*

1. *Prove the correctness of the design.*
2. *Is the functionality preserved if the order of the levels is changed?*
3. *Analyze the cost and delay of the design.*
4. *Prove the asymptotic optimality of the delay of the design.*
5. *Prove a lower bound on the cost of a combinational circuit that implements a cyclic shifter.*

5.3 Logical Shifters

Logical shifting is used for shifting binary strings that represent unsigned integers in binary representation. Shifting to the left by s positions corresponds to multiplying by 2^s followed by modulo 2^n. Shifting to the right by s positions corresponds to division by 2^s followed by truncation.

A bi-directional logical shifter is defined as follows.

Definition 5.5 *A* LOG-SHIFT(n) *is a combinational circuit defined as follows:*

Input:

- $x[n-1:0] \in \{0,1\}^n$,
- $sa[k-1:0] \in \{0,1\}^k$, *where* $k = \lceil \log_2 n \rceil$, *and*
- $\ell \in \{0,1\}$.

Output: $y[n-1:0] \in \{0,1\}^n$.

Functionality: *The output $\vec{y}$ is a logical shift of $\vec{x}$ by $\langle \vec{sa} \rangle$ positions. The direction of the shift is determined by ℓ. Formally, If $\ell = 1$, then perform a logical left shift as follows:*

$$y[n-1:0] \triangleq x[n-1-\langle \vec{sa} \rangle : 0] \cdot 0^{\langle \vec{sa} \rangle}.$$

If $\ell = 0$, then perform a logical right shift as follows:

$$y[n-1:0] \triangleq 0^{\langle \vec{sa} \rangle} \cdot x[n-1 : \langle \vec{sa} \rangle].$$

Example 5.3 *Let $x[3:0] = 0010$. If $sa[1:0] = 10$ and $\ell = 1$, then* LOG-SHIFT(4) *outputs $y[3:0] = 1000$. If $\ell = 0$, then the output equals $y[3:0] = 0000$.*

5.3.1 Implementation

As in the case of cyclic shifters, we break the task of designing a logical shifter into sub-tasks of logical shifts by powers of two.

Loosely speaking, an LBS(n, i) is a logical bi-directional shifter that outputs one of three possible strings: the input shifted to the left by 2^i positions, the input shifted to the right by 2^i positions, or the input without shifting. We now formally define this circuit.

Definition 5.6 *An* LBS(n, i) *is a combinational circuit defined as follows:*

Input: $x[n-1:0]$ *and* $s, \ell \in \{0,1\}$.

Output: $y[n-1:0]$.

Functionality: *Define* $x'[n-1+2^i : -2^i] \in \{0,1\}^{n+2\cdot 2^i}$ *as follows:*

$$x'[j] \triangleq \begin{cases} x[j] & \text{if } n > j \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

The value of the output $y[n-1:0]$ *is specified by*

$$\forall j \in [n-1:0]: \quad y[j] = x'[j + (-1)^{\ell} \cdot s \cdot 2^i]. \tag{5.1}$$

Note that the vector $\vec{x}'$ is an extension of the input $\vec{x}$ in which 2^i zeros are padded to the left and to the right of $\vec{x}$. The indexes of the bit string $\vec{x}'$ are also negative; this non-standard indexing is very useful in the context of logical shifters. It enables us to very easily express the fact that, in a logical shift, zeros are padded to one of the sides of the output.

The role of ℓ in Equation 5.1 is to determine if the shift is a left shift or a right shift. If $\ell = 1$ then $(-1)^{\ell} = -1$, and the shift is a left shift (since increasing indexes from $j - 2^i$ to j has the effect of a left shift). If $\ell = 0$, then $(-1)^{\ell} = 1$, and decreasing indexes from $j + 2^i$ to j has the effect of a right shift.

The role of s in Equation 5.1 is to determine if a shift (in either direction) takes place at all. If $s = 0$, then $y[j] = x[j]$, and no shift takes place. If $s = 1$, then the direction of the shift is determined by ℓ.

A bit-slice of an implementation of an LBS(n, i) is depicted in Figure 5.6. By the term "bit-slice" we mean that the figure depicts only how a single output bit $y[j]$ is computed. The whole circuit is obtained by combining such circuits for every output bit $y[j]$. We do not depict the whole circuit to avoid a messy figure with lots of wires that are hard to follow.

Figure 5.6 depicts a circuit for computing $y[j]$ that consists of two blocks.

1. The first block is a $(3:1)$-MUX. Since 3 is not a power of 2, such a multiplexer can be constructed by a "pruned" tree-like construction. On the other hand, many libraries contain a $(3:1)$-MUX as a basic gate; in such cases we can simply consider a $(3:1)$-MUX as a basic gate. In either case, a $(3:1)$-MUX is a simple circuit with a constant number of inputs and outputs, so the best option can be easily determined based on the technology at hand.

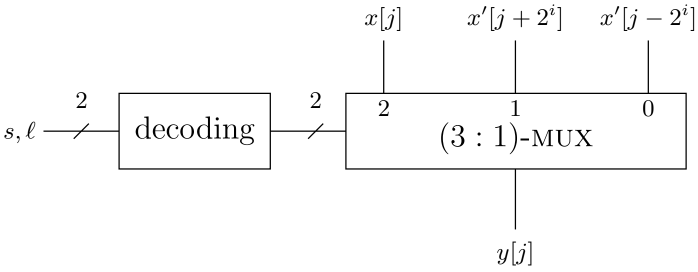


Figure 5.6: A bit-slice of an implementation of LBS(n, i).

2. The second block is a decoding circuit. This circuit is not a decoder as defined in Chapter 4. Instead, this is a simple circuit that is input two bits (ℓ and s) and outputs two bits. The output of the decoding circuit controls the selection performed by the $(3:1)$-mux. The decoding must cause the $(3:1)$-MUX to select the correct input based on the values of s and ℓ. We leave it as an exercise to design the decoding circuit.

Question 5.5 *This question deals with various aspects and details concerning the design of a logical shifter.*

1. *Design a "pruned" tree-like* $(3:1)$-MUX.
2. *Design the decoding box depicted in Figure 5.6.*
3. *Show how* LBS(n, i) *circuits can be cascaded to obtain a* LOG-SHIFT(n). *Hint: follow the design of a* BARREL-SHIFTER(n).

5.4 Arithmetic Shifters

Arithmetic shifters are used for shifting binary strings that represent signed integers in two's complement representation. Since left shifting is the same in logical shifting and in arithmetic shifting, we discuss only right shifting (i.e., division by a power of 2).

An arithmetic right shifter is defined as follows.

Definition 5.7 *An* ARITH-SHIFT(n) *is a combinational circuit defined as follows:*

Input: $x[n-1:0] \in \{0,1\}^n$ *and* $sa[k-1:0] \in \{0,1\}^k$, *where* $k = \lceil \log_2 n \rceil$.

Output: $y[n-1:0] \in \{0,1\}^n$.

Functionality: *The output* $\vec{y}$ *is a (sign-extended) arithmetic right shift of* $\vec{x}$ *by* $\langle \vec{sa} \rangle$ *positions. Formally,*

$$y[n-1:0] \triangleq x[n-1]^{\langle \vec{sa} \rangle} \cdot x[n-1:\langle \vec{sa} \rangle].$$

Example 5.4 *Let $x[3:0] = 1001$. If $sa[1:0] = 10$, then* ARITH-SHIFT(4) *outputs $y[3:0] = 1110$.*

Question 5.6 *Consider the definitions of* CLS(n, i) *and* LBS(n, i)*. Suggest an analogous definition* ARS(n, i) *for arithmetic right shift (i.e., modify the definition of $\vec{x}'$ and use* $(2:1)$-MUX*s). Suggest an implementation of an arithmetic right shifter based on cascading* ARS$(n.i)$ *circuits.*

5.5 Summary

We began this chapter by defining $(n : 1)$-multiplexers. We presented two optimal implementations. One implementations is based on a decoder, the other implementation is based on a tree of multiplexers.

We continued by defining three types of shifts: cyclic, logical, and arithmetic shifts. The method we propose for designing such shifters is to cascade a logarithmic number of shifters (with parameter i) that either perform a shift by 2^i positions or no shift at all.

Chapter 6

Priority encoders

Contents

Preliminary questions

1. Suppose that many devices wish to transmit data along a shared bus. How does the bus controller decide which device gets to use the bus?

2. Consider the binary fraction 0.000010101. In many cases, an arithmetic unit is supposed to shift the fraction so that its value is in the range $[1/2, 1)$ (this is often called *normalization*). How is the shift amount computed?

In this chapter we present designs of an important combinational module called priority encoders. Priority encoders have many uses, among them: (1) allocating usage of a shared resource according to a policy that is defined by priorities, and (2) determining the required amount of left shift needed to normalize a fraction so that it represents a number in the interval $[1/2, 1)$.

We begin this chapter with a discussion of the confusion that may be caused by interchanging bit orders in which the least significant bit appears first and orders in which the most significant bit appears first.

6.1 Big Endian vs. Little Endian

A long standing source of confusion is the order of bits in binary strings. This issue is very important when strings of bits are serially communicated or stored in memories. Consider the following two scenarios.

In the first setting Alice wishes to send to Bob a binary string $a[n-1:0]$. The channel that Alice and Bob use for communication is a serial channel. This means that Alice can only send one bit at a time. Now Alice has two "natural" choices:

- She can send $a[n-1]$ first and $a[0]$ last. Namely, she can send the bits in descending index order. This order is often referred to as *most significant bit first* or just MSB first.

- She can send $a[0]$ first and $a[n-1]$ last. Namely, she can send the bits in ascending index order. This order is often referred to as *least significant bit first* or just LSB first.

In the second setting computer words are stored in memory. A memory is a vector of storage places. We denote this vector by $M[0], M[1], \ldots$. Suppose that each storage place is capable of storing a byte (i.e., 8 bits). The typical word size in modern computers is 32 bits (and even 64 bits). This means that a word is stored in 4 memory slots. The question is how do we store a word $a[31:0]$ in 4 memory slots?

Obviously, it is a good idea to store the word in 4 consecutive slots, say $M[i:i+3]$. There are two "natural" options. In the first option storage is as follows:

$$\begin{aligned} M[i] &\leftarrow a[31:24] \\ M[i+1] &\leftarrow a[23:16] \\ M[i+2] &\leftarrow a[15:8] \\ M[i+3] &\leftarrow a[7:0]. \end{aligned}$$

This option is referred to as *Big Endian.*

In the second option storage is as follows:

$$\begin{aligned} M[i] &\leftarrow a[0:7] \\ M[i+1] &\leftarrow a[8:15] \\ M[i+2] &\leftarrow a[16:23] \\ M[i+3] &\leftarrow a[24:31]. \end{aligned}$$

This option is referred to as *Little Endian.* Note that, for the sake of aesthetics, we used increasing bit indexes in the second option.

It is highly recommended that you read the treatise "On holy wars and a plea for peace" by Danny Cohen (URLs have short lives, so I hope that you can find Cohen's treatise in `http://www.networksorcery.com/enp/ien/ien137.txt`). It was Danny Cohen who coined the terms Big Endian and Little Endian.

Each of these options has certain advantages and disadvantages. For example, if an architecture supports multiple word lengths, then it is convenient to have the most significant bit (MSB) stored in a fixed position relative to the address of the word (in our example we can see that in Big Endian the MSB is stored in $M[i]$ regardless of the number of bytes in $\vec{a}$.) On the other hand, if multiple word lengths are supported and we wish to add a half word (i.e., two-byte string) with a word (i.e., four-byte string), then Little Endian may simplify the task of aligning the two words (i.e., making sure that bits of the same weight are placed in identical offsets).

It is of no surprise that both conventions are used in commercial products. Architectures from the X86 family (such as Intel processors) use Little Endian byte ordering, while Motorola 68000 CPUs follow the Big Endian convention. Interestingly, the Power-PC supports both! Nevertheless, operating systems also follow different conventions: Microsoft operating systems follow Little Endian and Apple operating systems follow Big Endian. So a MAC with a Power-PC CPU that runs an Apple operating system runs in Big Endian mode.

This confusion spreads beyond hardware to software (e.g., Java uses Big Endian) and to file formats (e.g., GIFF uses Little Endian and JPEG uses Big Endian).

What does this story have to do with us? You might have noticed that we use both ascending indexes and descending indexes (e.g. $a[n-1:0]$ vs. $a[0:n-1]$) to denote the same string. These two conventions are simply an instance of the Big Endian vs. Little Endian controversy.

Following Oliver Swift (at the risk of not obeying Danny Cohen's plea), we use both ascending and descending bit orders according to the task we are considering. When considering strings that represent integers in binary representation, descending indexes are used (i.e., leftmost bit is the MSB). However in many parts of this chapter ascending indexes are used; the reason is to simplify handling of indexes in the text. We can only hope that this simplification does not lead to confusion.

6.2 Priority Encoders

Consider a binary string $x[0:n-1]$. (Note that we use ascending indexes here! So the index of the leftmost bit is 0.) The index of the leading one is the index of the leftmost non-zero bit in $x[0:n-1]$. We often abbreviate and refer to the index of the leading one simply as the leading one. Formally,

Definition 6.1 *The* leading one *of a binary string* $x[0:n-1]$ *is defined by*

$$\textsc{leading-one}(x[0:n-1]) \triangleq \begin{cases} \min\{i \mid x[i]=1\} & \textit{if } x[0:n-1] \neq 0^n \\ n & \textit{otherwise.} \end{cases}$$

Example 6.1 *Consider the string* $x[0:6] = 0110100$. *The leading one is the index* $[1]$. *Note that indexes are in ascending order and that* $x[0]$ *is the leftmost bit.*

The following claim follows immediately from the definition of the leading one.

Claim 6.1 *For every binary string* $\vec{a} = a[n-1:0]$

$$\textsc{leading-one}(\vec{a}) = \textsc{leading-one}(\vec{a} \cdot 1).$$

A priority encoder is a combinational circuit that computes the leading one. We consider two types of priority encoders: A unary priority encoder outputs the leading one in unary representation. A binary priority encoder outputs the leading one in binary representation.

Definition 6.2 *A binary string* $x[0:n-1]$ represents a number in unary representation *if* $x[0:n-1] \in 1^* \cdot 0^*$. *The* value represented in unary representation *by the binary string* $1^i \cdot 0^j$ *is* i.

Example 6.2 *The binary string* 01001011 *does not represent a number in unary representation. Only a string that is obtained by concatenating an all-ones string with an all-zeros string represents a number in unary representation.*

Before we define a unary priority encoder, we define a parallel prefix OR circuit.

Definition 6.3 *A parallel prefix* OR *circuit of length* n *is a combinational circuit specified as follows.*

Input: $x[0:n-1]$.

Output: $y[0:n-1]$.

Functionality:

$$y[i] = \textsc{or}(x[0:i]).$$

We denote parallel prefix OR circuit of length n by PPC–OR(n).

We denote unary priority encoder by U-PENC(n) and define it as follows. Note that the output of U-PENC(n) is simply the inversion of the outputs of PPC–OR(n).

Definition 6.4 *A unary priority encoder* U-PENC(n) *is a combinational circuit specified as follows.*

Input: $x[0:n-1]$.

Output: $y[0:n-1]$.

Functionality: *The output* $y[0:n-1]$ *is the unary representation of* LEADING-ONE($\vec{x}$), *namely,*

$$y[i] = \text{INV}(\text{OR}(x[0:i])).$$

Example 6.3 *Consider the string* $x[0:6] = 0110100$. *The output of a* PPC–OR(7) *is* 0111111. *The output of a* U-PENC(7) *is* 1000000.

Example 6.4 *If* $\vec{x} \neq 0^n$, *then the output of* U-PENC(n) *satisfies* $y[0:n-1] = 1^j \cdot 0^{n-j}$, *where* $j = \min\{i \mid x[i] = 1\}$. *If* $\vec{x} = 0^n$, *then* $\vec{y} = 1^n$ *and* $\vec{y}$ *is a unary representation of* n.

We denote binary priority encoder by B-PENC(n) and define it as follows.

Definition 6.5 *A binary priority encoder* B-PENC(n) *is a combinational circuit specified as follows.*

Input: $x[0:n-1]$.

Output: $y[k:0]$, *where* $k = \lfloor \log_2 n \rfloor$. *(Note that if* $n = 2^\ell$, *then* $k = \ell$.*)*

Functionality:

$$\langle \vec{y} \rangle = \text{LEADING-ONE}(\vec{x})$$

Note that if $n = 2^k$, then the length of the output of a B-PENC(n) is $k+1$ bits; otherwise, the number n could not be represented by the output.

Example 6.5 *Given input* $x[0:5] = 000101$, *a* U-PENC(6) *outputs* $y[0:5] = 111000$, *and* B-PENC(6) *outputs* $y[2:0] = 011$.

6.2.1 Implementation of U-PENC(n)

We can design a unary priority encoder by inverting the outputs of a parallel prefix PPC–OR(n). A brute force design of a PPC–OR(n) uses a separate OR-tree for each output bit. The delay of such a design is $O(\log n)$ and the cost is $O(n^2)$. The issue of efficiently combining these trees will be discussed in detail when we discuss parallel prefix computation in the context of fast addition. In the meantime, we present here a (non-optimal) design based on divide-and-conquer.

The method of divide-and-conquer is applicable for designing a PPC–OR(n). We apply divide-and-conquer in the following recursive design. If $n = 1$, then PPC–OR(1) is the trivial design in which $y[0] \leftarrow x[0]$. A recursive design of PPC–OR(n) for $n > 1$ that is a power of 2 is depicted in Figure 6.1. Proving the correctness of the proposed PPC–OR(n) design is a simple exercise in associativity of OR_n, so we leave it as a question.

Question 6.1 *This question deals with the recursive design of the* PPC–OR(n) *circuit depicted in Figure 6.1.*

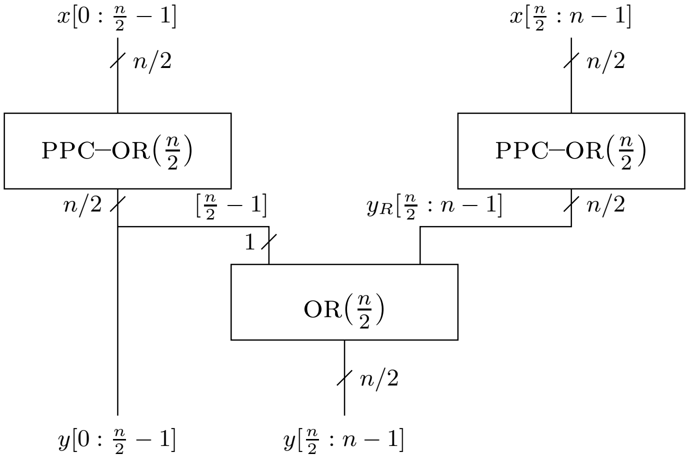


Figure 6.1: A recursive implementation of PPC–OR(n).

1. *Prove the correctness of the design.*
2. *Extend the design for values of n that are not powers of 2.*
3. *Analyze the delay of the design.*
4. *Prove the asymptotic optimality of the delay of the design.*

Cost analysis. The cost $c(n)$ of the PPC–OR(n) depicted in Figure 6.1 satisfies the following recurrence equation.

$$c(n) = \begin{cases} 0 & \text{if n=1} \\ 2 \cdot c(\frac{n}{2}) + (n/2) \cdot c(\text{OR}) & \text{otherwise.} \end{cases}$$

It follows that

$$\begin{aligned} c(n) &= 2 \cdot c(\frac{n}{2}) + \Theta(n) \\ &= \Theta(n \cdot \log n). \end{aligned}$$

The following question deals with the asymptotic optimality of the PPC–OR(n) design depicted in Figure 6.1.

Question 6.2 *Prove a lower bound on the cost of a parallel prefix computation circuit* PPC–OR(n).

In the chapter on fast addition we will present a cheaper implementation of PPC–OR(n) (with logarithmic delay).

6.2.2 Implementation of B-PENC

In this section we present two designs for a binary priority encoder. The first design is based on a reduction to a unary priority encoder. The second design is based on divide-and-conquer.

Reduction to PPC–OR

Consider an input $x[0:n-1]$ to a binary priority encoder. If $\vec{x} = 0^n$, then the output should equal $bin(n)$. If $\vec{x} \neq 0^n$, then let $j = \min\{i \mid x[i] = 1\}$. The output $\vec{y}$ of a B-PENC(n) should satisfy $\langle\vec{y}\rangle = j$. Our design is based on the observation that the output $u[0:n-1]$ of a PPC–OR(n) satisfies $\vec{u} = 0^j \cdot 1^{n-j}$. In Figure 6.2 we depict a reduction from the task of computing $\vec{y}$ to the task of computing $\vec{u}$.

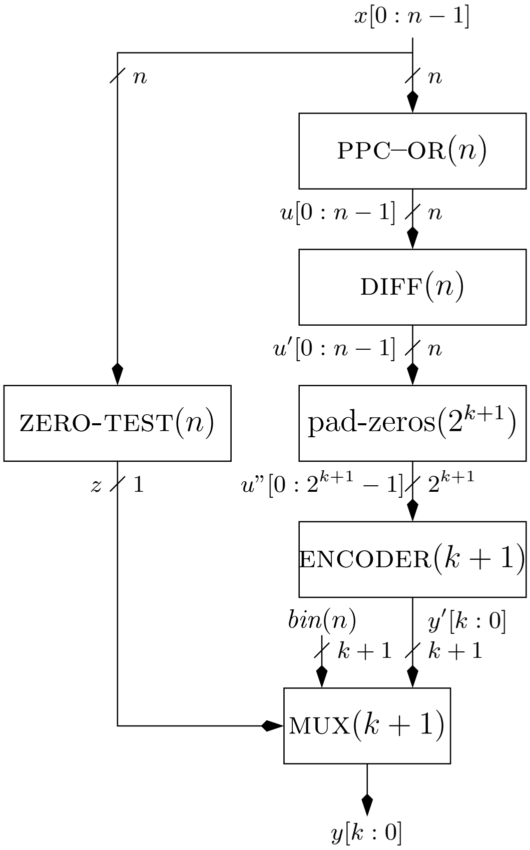


Figure 6.2: A binary priority encoder based on a unary priority encoder.

The stages of the reduction are as follows. We first assume that the input $x[n-1:0]$ is not 0^n and denote LEADING-ONE$(\vec{x})$ by j. We then deal with the case that $\vec{x} = 0^n$ separately.

1. The input $\vec{x}$ is fed to a PPC–OR(n) that outputs the string $u[0:n-1] = 0^j \cdot 1^{n-j}$.

2. A difference circuit is fed by $\vec{u}$ and outputs the string $u'[0 : n-1]$. The string $u'[0 : n-1]$ is defined as follows:

$$u'[i] = \begin{cases} u[0] & \text{if } i = 0 \\ u[i] - u[i-1] & \text{otherwise.} \end{cases}$$

   Note that the output $u'[0 : n-1]$ satisfies:

$$u'[0 : n-1] = 0^j \cdot 1 \cdot 0^{n-1-j}.$$

   Hence $\vec{u'}$ constitutes a 1-out-of-n representation of the index of the leading one (provided that $\vec{x} \neq 0^n$).

3. If n is not a power of 2, then obtain the string $u''[0 : 2^{k+1}-1]$ from $\vec{u'}$ by padding zeros from the right as follows:

$$u''[0 : n-1] \leftarrow u'[0 : n-1] \qquad\qquad u''[n : 2^{k+1}-1] \leftarrow 0^{2^k - n}.$$

   Namely, $u''[0 : 2^{k+1}-1] = 0^j \cdot 1 \cdot 0^{2^{k+1}-1}$. (Recall that $k = \lfloor \log_2 n \rfloor$.) Note that the cost and delay of padding zeros is zero.

4. The string $u''[0 : 2^{k+1}-1]$ is input to an ENCODER$(k+1)$. The encoder outputs the string $y'[k : 0]$ that satisfies $\langle \vec{y'} \rangle$ equals the index of the bit in $\vec{u''}$ that equals one. Namely, $\langle \vec{y'} \rangle = j$, which is the desired output if $\vec{x} \neq 0^n$. In the next item deal also with the case of an all-zeros input.

5. The input $x[n-1 : 0]$ is fed to a zero-tester that outputs 1 if $\vec{x} = 0^n$. In this case, the output $y[k : 0]$ should satisfy $\langle \vec{y} \rangle = n$. The selection between $bin(n)$ and $\vec{y'}$ is performed by the multiplexer according to the output of the zero-tester.

Cost and delay analysis. The cost of the binary priority encoder depicted in Figure 6.2 satisfies:

$$\begin{aligned} c(n) &= c(\text{PPC–OR}(n)) + c(\text{DIFF}(n)) + c(\text{ENCODER}(k)) + c(\text{MUX}(k)) + c(\text{ZERO-TEST}(n)) \\ &= c(\text{PPC–OR}(n)) + \Theta(n). \end{aligned}$$

Hence, the cost of the reduction from a binary priority encoder to a unary priority encoder is linear. This implies that if we knew how to implement a linear cost PPC–OR(n) then we would have a linear cost B-PENC(n).

The delay of the binary priority encoder depicted in Figure 6.2 satisfies:

$$\begin{aligned} d(n) &= \max\{d(\text{PPC–OR}(n)) + d(\text{DIFF}(n)) + d(\text{ENCODER}(k)), d(\text{ZERO-TEST}(n))\} + d(\text{MUX}) \\ &= d(\text{PPC–OR}(n)) + \Theta(\log n). \end{aligned}$$

Hence, the delay of the reduction from a binary priority encoder to a unary priority encoder is logarithmic.

A divide-and-conquer implementation

We now present a divide-and-conquer design. For simplicity, assume that $n = 2^k$.

Figure 6.3 depicts a recursive design for a binary priority encoder. Consider an input $x[0 : n-1]$. We partition it into two halves: the left part $x[0 : \frac{n}{2} - 1]$ and the right part $x[\frac{n}{2} : n-1]$. Each of these two parts is fed to a binary priority encoder with $n/2$ inputs. We denote the outputs of these binary priority encoders by $y_L[k-1 : 0]$ and $y_R[k-1 : 0]$. The final output $y[k : 0]$ is computed as follows: $y[k] = 1$ iff $y_L[k-1] = y_R[k-1] = 1$, $y[k-1] = \text{AND}(y_L[k-1], \text{INV}(y_R[k-1]))$, and $y[k-2 : 0]$ equals $y_L[k-2 : 0]$ if $y_L[k-1] = 0$ and $y_R[k-2 : 0]$ otherwise. We now prove the correctness of the binary priority encoder design based on divide-and-conquer. Note that we omitted a description for $n = 2$. We leave the recursion basis as an exercise.

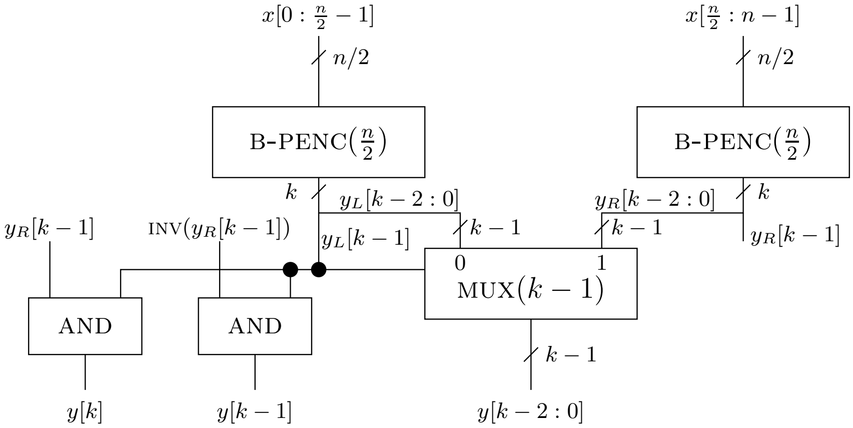


Figure 6.3: A recursive implementation of a binary priority encoder.

Claim 6.2 *The design depicted in Figure 6.3 is a binary priority encoder.*

Proof: The proof is by induction. We assume that we have a correct design for $n = 2$ so the induction basis holds. We proceed with the induction step. We consider three cases:

1. $x[0 : \frac{n}{2} - 1] \neq 0^{n/2}$. By the induction hypothesis, the required output in this case equals $0 \cdot \vec{y_L}$. Note that $y_L[k-1] = 0$ since the index of the leading one is less than $n/2$. It follows that $y[k] = y[k-1] = 0$ and $y[k-2 : 0] = y_L[k-2 : 0]$. Hence $\langle \vec{y} \rangle = \langle \vec{y_L} \rangle$, and the output equals the index of the leading one, as required.

2. $x[0 : \frac{n}{2} - 1] = 0^{n/2}$ and $x[\frac{n}{2} : n-1] \neq 0^{n/2}$. In this case the index of the leading one is $n/2 + \langle \vec{y_R} \rangle$. By the induction hypothesis, we have $y_L[k-1] = 1$ and $y_R[k-1] = 0$. It follows that $y[k] = 0$, $y[k-1] = 1$, and $y[k-2 : 0] = y_R[k-2 : 0]$. Hence $\langle \vec{y} \rangle = 2^{k-1} + \langle \vec{y_R} \rangle$, as required.

3. $x[0 : \frac{n}{2} - 1] = 0^{n/2}$ and $x[\frac{n}{2} : n-1] = 0^{n/2}$. By the induction hypothesis, we have $y_L[k-1 : 0] = y_R[k-1 : 0] = 1 \cdot 0^{k-1}$. Hence $y[k] = 1$, $y[k-1] = 0$, and $y[k-2 : 0] = 0^{k-1}$, as required.

Since the design is correct in all three cases, we conclude that the design is correct. □

Cost and delay analysis. The cost of the binary priority encoder depicted in Figure 6.3 satisfies (recall that $n = 2^k$):

$$c(\textsc{b-penc}(n)) = \begin{cases} c(\textsc{nor}) + c(\textsc{and}) + c(\textsc{inv}) & \text{if n=2} \\ 2 \cdot c(\textsc{b-penc}(n/2)) + 2 \cdot c(\textsc{and}) + (k-1) \cdot c(\textsc{mux}) & \text{otherwise.} \end{cases}$$

Consider the substitution $\gamma(k) = c(2^k)$. Then $\gamma(k)$ satisfies the recurrence

$$\gamma(k) = 2 \cdot \gamma(k-1) + \Theta(k).$$

This recurrence is identical to the recurrence in Equation 4.1. Hence $\gamma(k) = \Theta(2^k)$ and $c(\textsc{b-penc}(n)) = \Theta(n)$. This implies the asymptotic optimality of the design.

The delay of the binary priority encoder depicted in Figure 6.3 satisfies:

$$d(\textsc{b-penc}(n)) = \begin{cases} t_{pd}(\textsc{nor}) & \text{if n=2} \\ d(\textsc{b-penc}(n/2)) + \max\{d(\textsc{mux}) + d(\textsc{and})\} & \text{otherwise.} \end{cases}$$

Hence, the delay is logarithmic, and the design is asymptotically optimal also with respect to delay.

6.3 Summary

In this section we presented designs of priority encoders. Priority encoders are circuits that compute the position of a leading one in a binary string. We considered two types of priority encoders: unary and binary. The difference between these two types of priority encoders is in how the position of the leading one is represented (i.e. in binary representation or in unary representation).

We first considered unary priority encoders. In the unary case, it is more convenient to consider parallel prefix OR circuits. The only difference between these two circuits is the inversion of the outputs. We presented a divide-and-conquer design for a parallel prefix OR circuit PPC–(OR)(n). When we discuss fast addition we will return to this circuit and present an even cheaper design.

We then considered binary priority encoders. Two designs were presented. The first one is simply a reduction to PPC–(OR)(n). The overhead in cost is linear and the overhead in delay is logarithmic. Hence, this design leads to an asymptotically optimal binary priority encoder provided that a linear cost and logarithmic delay PPC–(OR)(n) design is used.

The second binary priority encoder design is a divide-and-conquer design. The cost of this design is linear cost and the delay is logarithmic, hence this design is asymptotically optimal.

Chapter 7

Half-Decoders

Contents

Preliminary questions

1. How can we reduce the task of a bi-directional logical shift to a cyclic left shift?
2. Is it easy to compare a number x represented in unary representation with a constant i?

In this chapter we deal with the design of half-decoders. Recall that a decoder is a combinational circuit that computes a 1-out-of-2^n representation of a given binary number. A half-decoder computes a unary representation of a given binary number. Half-decoders can be used for reducing the task of a logical shift to a cyclic shift.

7.1 Specification

Definition 7.1 *A half-decoder with input length n is a combinational circuit defined as follows:*

Input: $x[n-1:0]$.

Output: $y[0:2^n-1]$

Functionality:

$$y[0:2^n-1] = 1^{\langle x[n-1:0]\rangle} \cdot 0^{2^n - \langle x[n-1:0]\rangle}.$$

We denote a half-decoder with input length n by H-DEC(n).

Example 7.1

- *Consider a half-decoder* H-DEC(3). *Given $x[2:0]=101$,* H-DEC(3) *outputs $y[0:7]=11111000$.*
- *Given $\vec{x}=0^n$,* H-DEC(n) *outputs $\vec{y}=0^{2^n}$.*
- *Given $\vec{x}=1^n$,* H-DEC(n) *outputs $\vec{y}=1^{2^n-1}\cdot 0$.*

Remark 7.1 *Observe that $y[2^n-1]=0$, for every input string. One could omit the bit $y[2^n-1]$ from the definition of a half-decoder. We left it in to make the description of the design slightly simpler.*

The next question deals with designing a half-decoder using a decoder and a unary priority encoder. The delay of the resulting design is too big.

Question 7.1 *Suggest an implementation of a half-decoder based on a decoder and a unary priority encoder. Analyze the cost and delay of the design. Is it asymptotically optimal with respect to cost or delay?*

7.2 Preliminaries

In this section we present a few claims that are used in the design of asymptotically optimal half-decoders.

The following claim follows trivially from the definition of a half-decoder.

Claim 7.1

$$y[i]=1 \Longleftrightarrow i < \langle \vec{x}\rangle.$$

Assume that the binary string $z[0:n-1]$ represents a number in unary representation (namely, $\vec{z} = 1^j \cdot 0^{n-j}$). Let $wt(\vec{z})$ denote the value represented by $\vec{z}$ in unary representation. The following claim shows that it is easy to compare $wt(\vec{z})$ with a fixed constant $i \in [0, n-1]$. (By easy we mean that it requires constant cost and delay.)

Claim 7.2 *For $i \in [0 : n-1]$:*

$$\begin{aligned}
\mathrm{wt}(\vec{z}) < i &\iff z[i-1] = 0\\
\mathrm{wt}(\vec{z}) > i &\iff z[i] = 1\\
\mathrm{wt}(\vec{z}) = i &\iff z[i] = 0 \text{ and } z[i-1] = 1.
\end{aligned}$$

Question 7.2 *Prove Claim 7.2.*

Claim 7.2 gives a simple recipe for a "comparison box". A comparison box, denoted by $\textsc{comp}(\vec{z}, i)$, is a combinational circuit that compares $wt(\vec{z})$ and i. It has two inputs: $z[i-1]$ and $z[i]$, and it has three outputs GT, EQ, LT. The outputs have the following meaning: GT - indicates whether $wt(\vec{z}) > i$, EQ - indicates whether $wt(\vec{z}) = i$, and LT - indicates whether $wt(\vec{z}) < i$. In the sequel we will only need the GT and EQ outputs. (Note that the GT output simply equals $z[i]$ and the LT output simply equals $\textsc{inv}(z[i-1])$).

We now follow the technique of division with a remainder (see Claim 4.1). Consider a partitioning of a string $x[n-1:0]$ according to a parameter k into two sub-strings of length k and $n-k$. Namely

$$x_L[n-k-1:0] = x[n-1:k] \qquad \text{and} \qquad x_R[k-1:0] = x[k-1:0].$$

Binary representation implies that

$$\langle \vec{x} \rangle = 2^k \cdot \langle \vec{x}_L \rangle + \langle \vec{x}_R \rangle.$$

Namely the quotient when dividing $\langle \vec{x} \rangle$ by 2^k is $\langle \vec{x}_L \rangle$, and the remainder is $\langle \vec{x}_R \rangle$.

Consider an index i, and divide it by 2^k to obtain $i = 2^k \cdot q + r$, where $r \in \{0, \ldots, 2^k - 1\}$. (The quotient of this division is q, and r is simply the remainder.) Division by 2^k can be interpreted as partitioning the numbers into blocks, where each block consists of numbers with the same quotient. This division divides the range $[2^n - 1 : 0]$ into 2^{n-k} blocks, each block is of length 2^k. The quotient q can be viewed as an index of the block that i belongs to. The remainder r can be viewed as the offset of i within its block.

The following claim shows how to easily compare i and $\langle \vec{x} \rangle$ given $q, r, \langle \vec{x}_L \rangle$, and $\langle \vec{x}_R \rangle$.

Claim 7.3

$$i < \langle \vec{x} \rangle \iff q < \langle \vec{x}_L \rangle \text{ or } (q = \langle \vec{x}_L \rangle \text{ and } r < \langle \vec{x}_R \rangle)$$

The interpretation of the above claim in terms of "blocks" and "offsets" is the following. The number $\langle \vec{x} \rangle$ is a number in the range $[2^n - 1 : 0]$. The index of the block this number belongs to is $\langle \vec{x}_L \rangle$. The offset of this number within its block is $\langle \vec{x}_R \rangle$. Hence, comparison of $\langle \vec{x} \rangle$ and i can be done in two steps: compare the block indexes, if they are different, then the number with the higher block index is bigger. If the block indexes are identical, then the offset value determines which number is bigger.

7.3 Implementation

In this section we present an asymptotically optimal half-decoder design. Our design is a recursive divide-and-conquer design.

A H-DEC(n), for $n = 1$ is the trivial circuit $y[0] \leftarrow x[0]$. We now proceed with the recursion step. Figure 7.1 depicts a recursive implementation of a H-DEC(n). The parameter k equals $k = \lceil \frac{n}{2} \rceil$ (in fact, to minimize delay one needs to be a bit more precise since the paths may not be perfectly balanced for $k = \lceil \frac{n}{2} \rceil$). The input string $x[n-1:0]$ is divided into two strings $x_L[n-k-1:0] = x[n-1:k]$ and $x_R[k-1:0] = x[k-1:0]$. These strings are fed to a half-decoders H-DEC$(n-k)$ and H-DEC(k), respectively. We denote the outputs of the half-decoders by $z_L[2^{n-k}-1:0]$ and $z_R[2^k-1:0]$, respectively. Each of these string are fed to comparison boxes. The "rows" comparison box is fed by $\vec{z_L}$ and compares $\vec{z_L}$ with $i \in [0 : 2^{n-k}-1]$. The "columns" comparison box is fed by $\vec{z_R}$ and compares $\vec{z_R}$ with $j \in [0 : 2^k - 1]$. Note that we only use the GT and EQ outputs of the rows comparison box, and that we only use the GT output of the columns comparison box. (Hence the columns comparison box is trivial and has zero cost and delay.) We denote the outputs of the rows comparison box by $Q_{GT}[0 : 2^{n-k}-1]$ and $Q_{EQ}[0 : 2^{n-k}-1]$. We denote the output of the columns comparison box by $R_{GT}[0 : 2^k - 1]$. Finally, the outputs of the comparison boxes fed an array of $2^{n-k} \times 2^k$ G-gates. Consider the G-gate $G_{q,r}$ positioned in row q and in column r. The gate $G_{q,r}$ outputs $y[q \cdot 2^k + r]$ that is defined by

$$y[q \cdot 2^k + r] \stackrel{\triangle}{=} \text{OR}(Q_{GT}[q], \text{AND}(Q_{EQ}[q], R_{GT}[r])). \tag{7.1}$$

Example 7.2 *Let $n = 4$ and $k = 2$. Consider $i = 6$. The quotient and remainder of i when divided by 4 are 1 and 2, respectively. By Claim 7.1, $y[6] = 1$ iff $\langle x[3:0] \rangle > 6$. By Claim 7.3, $\langle \vec{x} \rangle > 6$ iff ($\langle x[3:2] \rangle > 1$) or ($\langle x[3:2] \rangle = 1$ and $\langle x[1:0] \rangle > 2$). It follows that if $Q_{GT}[1] = 1$, then $y[6] = 1$. If $Q_{EQ}[1] = 1$, then $y[6] = R_{GT}[2]$.*

7.4 Correctness

Claim 7.4 *The design depicted in Figure 7.1 is a correct implementation of a half-decoder.*

Proof: The proof is by induction on n. The induction basis, for $n = 1$, is trivial. We now prove the induction step. By Claim 7.1 it suffices to show that $y[i] = 1$ iff $i < \langle \vec{x} \rangle$, for every $i \in [2^n - 1 : 0]$. Fix an index i and let $i = q \cdot 2^k + r$. By Claim 7.3,

$$i < \langle \vec{x} \rangle \iff (q < \langle \vec{x}_L \rangle) \text{ or } ((q = \langle \vec{x}_L \rangle) \text{ and } (r < \langle \vec{x}_R \rangle)).$$

The induction hypothesis implies that:

$$\begin{aligned} q < \langle \vec{x}_L \rangle &\iff z_L[q] = 1 \\ q = \langle \vec{x}_L \rangle &\iff z_L[q] = 0 \text{ and } z_L[q-1] = 1 \\ r < \langle \vec{x}_R \rangle &\iff z_R[r] = 1. \end{aligned}$$

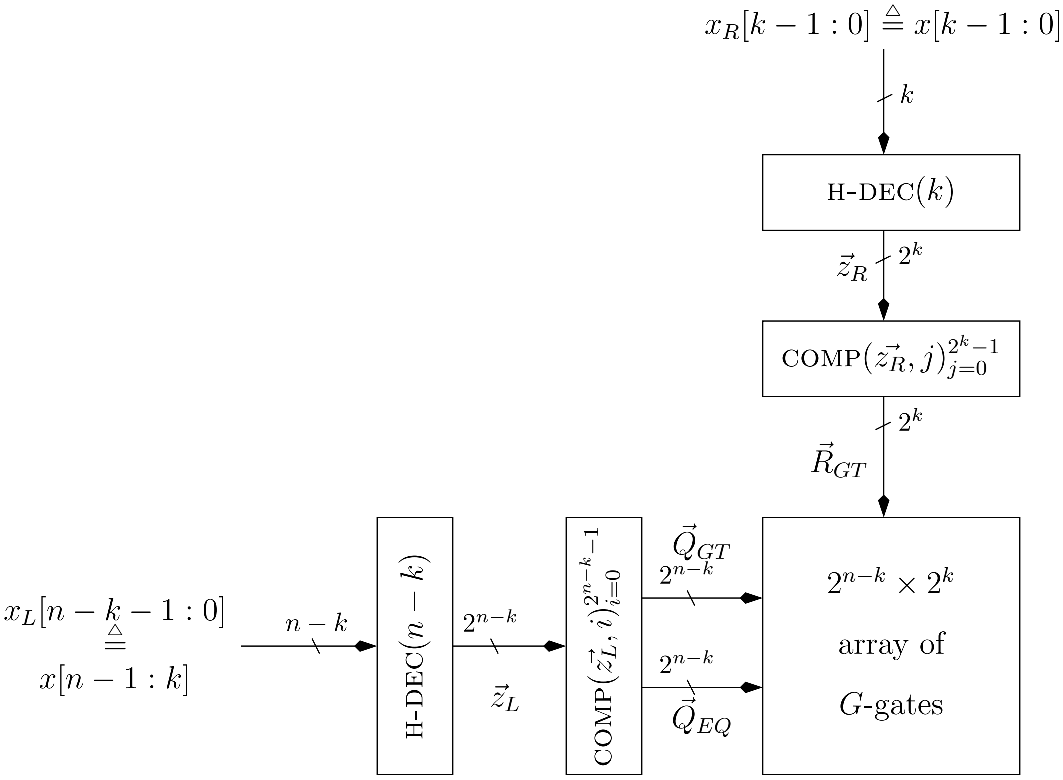

$x_R[k-1:0] \triangleq x[k-1:0]$
k
H-DEC(k)
$\vec{z}_R$
2^k
COMP$(\vec{z}_R, j)_{j=0}^{2^k-1}$
2^k
$\vec{R}_{GT}$
$x_L[n-k-1:0] \triangleq x[n-1:k]$
$n-k$
H-DEC$(n-k)$
2^{n-k}
$\vec{z}_L$
COMP$(\vec{z}_L, i)_{i=0}^{2^{n-k}-1}$
$\vec{Q}_{GT}$
2^{n-k}
2^{n-k}
$\vec{Q}_{EQ}$
$2^{n-k} \times 2^k$ array of G-gates

Figure 7.1: A recursive implementation of H-DEC(n). Note that the comparison boxes COMP$(\vec{z}_R, j)$ are trivial, since we only use their GT outputs.

Observe that:

- The signal $Q_{GT}[q]$ equals $z_L[q]$, and hence indicates if $q < \langle \vec{x}_L \rangle$.
- The signal $Q_{EQ}[q]$ equals $\text{AND}(\text{INV}(z_L[q]), z_L[q-1])$, and hence indicates if $q = \langle \vec{x}_L \rangle$.
- The signal $R_{GT}[r]$ equals $z_R[r]$, and hence indicates if $r < \langle \vec{x}_R \rangle$.

Finally, by Eq. 7.1, $y[i] = 1$ iff $\text{OR}(Q_{GT}[q], \text{AND}(Q_{EQ}[q], R_{GT}[r]))$. Hence $y[i]$ is correct, and the claim follows. □

7.5 Cost and delay analysis

The cost of H-DEC(n) satisfies the following recurrence equation:

$$c(\text{H-DEC}(n)) = \begin{cases} 0 & \text{if n=1} \\ c(\text{H-DEC}(k)) + c(\text{H-DEC}(n-k)) & \\ +2^{n-k} \cdot c(EQ) + 2^n \cdot c(G) & \text{otherwise.} \end{cases}$$

The cost of computing the EQ signals is $c(\text{INV}) + c(\text{AND})$. The cost of a G-gate is $c(\text{AND}) + c(\text{OR})$. It follows that

$$c(\text{H-DEC}(n)) = c(\text{H-DEC}(k)) + c(\text{H-DEC}(n-k)) + \Theta(2^n)$$

We already solved this recurrence in the case of decoders and showed that $c(\text{H-DEC}(n)) = \Theta(2^n)$.

The delay of H-DEC(n) satisfies the following recurrence equation:

$$d(\text{H-DEC}(n)) = \begin{cases} 0 & \text{if n=1} \\ \max\{d(\text{H-DEC}(k)), d(\text{H-DEC}(n-k)) + d(EQ)\} & \\ +d(G) & \text{otherwise.} \end{cases}$$

The delay of computing EQ as well as the delay of a G-gate is constant. Set $k = \lceil \frac{n}{2} \rceil$, then the recurrence degenerates to

$$\begin{aligned} d(\text{H-DEC}(n)) &= d(\text{H-DEC}(n/2)) + \Theta(1) \\ &= \Theta(\log n). \end{aligned}$$

It follows that the delay of H-DEC(n) is asymptotically optimal since all the inputs belong to the cone of a half-decoder.

The following question deals with the asymptotically optimal cost of a half-decoder design.

Question 7.3 *Prove that every implementation of a half-decoder design must contain at least $2^n - 2$ non-trivial gates. (Here we assume that every non-trivial gate has a single output, and we do not have any fan-in or fan-out restrictions).*

Hint: The idea is to show that all the outputs must be fed by distinct non-trivial gates. Well, we know that $y[2^n - 1] = 0$, so that rules out one output. What about the other outputs? We need to show that:

1. *The other outputs are not constant - so they can't be fed by a trivial constant gate.*

2. *The other outputs are distinct - so every two outputs can't be fed by the same gate.*

3. *The other outputs do no equal the inputs - so they can't be directly fed from input gates (which are trivial gates).*

It is not hard to prove the first two items. The third item is simply false! There does exist an output bit that equals one of the input bits. Can you prove which output bit this is? Can you prove that it is the only such output bit?

7.6 Application

In the following question half-decoders are used to implement shifters that support all types of shifts.

Question 7.4 *CPUs often support all three types of shifting: cyclic, logical, and arithmetic shifting.*

1. *Write a complete specification of a shifter that can perform all three types of shifts.*

2. *Propose an implementation of such a shifter.*

7.7 Summary

This chapter deals with the design of optimal half-decoders. A half-decoder outputs a unary representation of a binary number. Our divide-and-conquer design is a variation of a decoder design. It employs the fact that comparison with a constant is easy in unary representation.

Chapter 8

Addition

Contents

Preliminary questions

1. What is the definition of an adder?
2. What is the smallest possible delay of an adder? Do you know of an adder that achieves this delay?
3. Can you prove the correctness of the addition algorithm taught in elementary school?
4. Suppose you are given the task of adding very long numbers. Could you share this work with friends so that you could work on it simultaneously to speed up the computation?

In this chapter we define binary adders. We start by considering a Ripple Carry Adder. This is a design with linear delay and linear cost. We then present designs with logarithmic delay but super-linear cost.

8.1 Definition of a binary adder

Definition 8.1 *A* binary adder *with input length n is a combinational circuit specified as follows.*

Input: $A[n-1:0], B[n-1:0] \in \{0,1\}^n$, *and* $C[0] \in \{0,1\}$.

Output: $S[n-1:0] \in \{0,1\}^n$ *and* $C[n] \in \{0,1\}$.

Functionality:

$$\langle \vec{S} \rangle + 2^n \cdot C[n] = \langle \vec{A} \rangle + \langle \vec{B} \rangle + C[0] \tag{8.1}$$

We denote a binary adder with input length n by ADDER(n). The inputs $\vec{A}$ and $\vec{B}$ are the binary representations of the addends. The input $C[0]$ is often called the *carry-in bit.* The output $\vec{S}$ is the binary representation of the sum (more precisely, $\vec{S}$ is the binary representation of the sum modulo 2^n), and the output $C[n]$ is often called the *carry-out bit.*

Question 8.1 *Verify that the functionality of* ADDER(n) *is well defined. Namely, for every $A[n-1:0], B[n-1:0]$, and $C[0]$ there exist $S[n-1:0]$ and $C[n]$ that satisfy Equation 8.1.*

Hint: Show that the set of numbers that can be represented by sums $\langle \vec{A} \rangle + \langle \vec{B} \rangle + C[0]$ equals the set of numbers that can be represented by sums $\langle \vec{S} \rangle + 2^n \cdot C[n]$.

There are many ways to implement an ADDER(n). In this chapter we present a few ADDER(n) designs.

Question 8.2 *Prove lower bounds on the cost and delay of combinational circuits that implement an* ADDER(n).

8.2 Ripple Carry Adder

Ripple Carry Adders are built by chaining a row of Full-Adders. We denote a Ripple Carry Adder that implements an ADDER(n) by RCA(n). A Full-Adder is a combinational circuit that adds three bits and represents their sum in binary representation.

We do not discuss here how to build a Full-Adder from gates. Since a Full-Adder has a constant number of inputs and outputs, every (reasonable) implementation has constant cost and delay. Optimizing these constants is a technology dependent issue and is not within the scope of our discussion.

Definition 8.2 (Full-Adder) *A* Full-Adder *is a combinational circuit with* 3 *inputs* $x, y, z \in \{0, 1\}$ *and* 2 *outputs* $c, s \in \{0, 1\}$ *that satisfies:*

$$2c + s = x + y + z.$$

The output s of a Full-Adder is often called the *sum output*. The output c of a Full-Adder is often called the *carry-out output*. We denote a Full-Adder by FA.

A Ripple Carry Adder, RCA(n), is built by chaining a row of n Full-Adders. An RCA(n) is depicted in Figure 8.1. Note that the carry-out output of the ith Full-Adder is denoted by $c[i+1]$. The weight of $c[i+1]$ is 2^{i+1}. This way, the weight of every signal is two to the power of its index. One can readily notice that an RCA(n) adds numbers using the same addition algorithm that we use for adding numbers by hand.

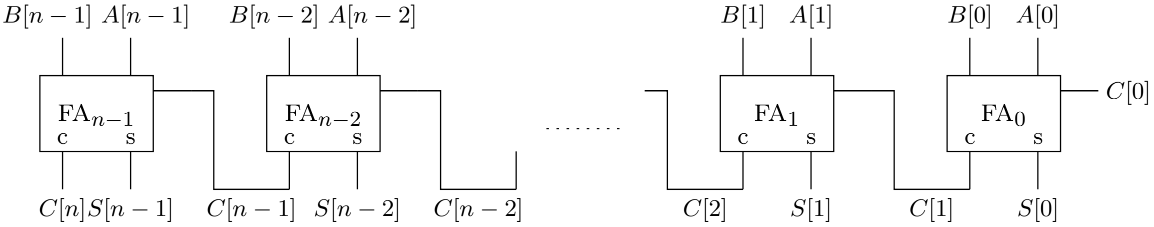


Figure 8.1: A Ripple Carry Adder RCA(n).

8.2.1 Correctness proof

In this section we prove the correctness of an RCA(n). To facilitate the proof, we use an equivalent recursive definition of RCA(n). The recursive definition is as follows.

Basis: an RCA(1) is simply a Full-Adder. Step: a recursive description of RCA(n), for $n \geq 1$, is depicted in Figure 8.2.

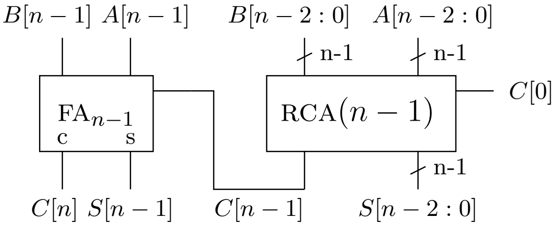


Figure 8.2: A recursive description of RCA(n).

The following claim deals with the correctness of RCA(n).

Claim 8.1 RCA(n) *is a correct implementation of* ADDER(n).

Proof: The proof is by induction on n. The induction basis, for $n = 1$, follows directly from the definition of a Full-Adder. The induction step is proved as follows.

The induction hypothesis, for $n - 1$, is

$$\langle A[n-2:0]\rangle + \langle B[n-2:0]\rangle + C[0] = 2^{n-1} \cdot C[n-1] + \langle S[n-2:0]\rangle. \tag{8.2}$$

The definition of a Full-Adder states that

$$A[n-1] + B[n-1] + C[n-1] = 2 \cdot C[n] + S[n-1]. \tag{8.3}$$

Multiply Equation 8.3 by 2^{n-1} to obtain

$$2^{n-1} \cdot A[n-1] + 2^{n-1} \cdot B[n-1] + 2^{n-1} \cdot C[n-1] = 2^n \cdot C[n] + 2^{n-1} \cdot S[n-1]. \tag{8.4}$$

Note that $2^{n-1} \cdot A[n-1] + \langle A[n-2:0]\rangle = \langle A[n-1:0]\rangle$. By adding Equations 8.2 and 8.4 we obtain:

$$2^{n-1} \cdot C[n-1] + \langle A[n-1:0]\rangle + \langle B[n-1:0]\rangle + C[0] = 2^n \cdot C[n] + 2^{n-1} \cdot C[n-1] + \langle S[n-1:0]\rangle.$$

Cancel out $2^{n-1} \cdot C[n-1]$, and the claim follows. □

8.2.2 Delay and cost analysis

The cost of an $\text{RCA}(n)$ satisfies:

$$c(\text{RCA}(n)) = n \cdot c(\text{FA}) = \Theta(n).$$

The delay of an $\text{RCA}(n)$ satisfies

$$d(\text{RCA}(n)) = n \cdot d(\text{FA}) = \Theta(n).$$

The answer to Question 8.2 implies that the asymptotic cost of $\text{RCA}(n)$ is optimal, but its delay is exponentially far away from the optimum delay. The clock rates in modern microprocessors correspond to the delay of 15-20 gates (in more aggressive designs, the critical paths are even shorter). Most microprocessors easily add 32-bit numbers within one clock cycle. Obviously, adders in such microprocessors are not Ripple Carry Adders. In the rest of the chapter we present faster $\text{ADDER}(n)$ designs.

8.3 Carry bits

We now define the carry bits associated with the addition

$$\langle A[n-1:0]\rangle + \langle B[n-1:0]\rangle + C[0] = \langle S[n-1:0]\rangle + 2^n \cdot C[n]. \tag{8.5}$$

Our definition is based on the values of the signals $C[n-1:1]$ of an $\text{RCA}(n)$. This definition is well defined in light of the Simulation Theorem of combinational circuits.

Definition 8.3 *The carry bits $C[n:0]$ corresponding to the addition in Eq. 8.5 are defined as the values of the stable signals $C[n:0]$ in an* RCA(n).

Note that there are $n+1$ carry-bits associated with the addition defined in Equation 8.5; these bits are indexed from zero to n. The first carry bit $C[0]$ is an input, the last carry bit $C[n]$ is an output, and the remaining carry bits $C[n-1:0]$ are internal signals.

We now discuss a few issues related to the definition of the carry bits and binary addition.

8.3.1 Redundant and non redundant representation

Consider Eq. 8.5 and let $x = \langle\vec{A}\rangle + \langle\vec{B}\rangle + C[0]$. Equation 8.5 means that the sum x admits two representations. The representation of x on the right hand side is the standard binary representation. This representation is *non-redundant*. This means that every number that is representable by $n+1$ bits has a unique representation. (Note that we need to restrict ourselves to $n+1$ bits, otherwise leading zeros create multiple representations. For example: 1, 01, and 001 are different representations of the same number).

One nice characteristic of non-redundant representation is that comparison is easy. Suppose that $X[n-1:0]$ is a binary representation of x and that $Y[n-1:0]$ is a binary representation of y. If we wish to check if $x=y$, all we need to do is check if the binary strings $\vec{X}$ and $\vec{Y}$ are identical.

The left hand side represents the same value represented by $C[n]$ and $S[n-1:0]$. However, on the left hand side we have two binary strings and a carry-in bit. Given x, there are many possible combinations of values of $\langle\vec{A}\rangle$, $\langle\vec{B}\rangle$ and $C[0]$ that represent x. For example: $8 = 4+3+1$ and also $8 = 5+3+0$.

We refer to such a representation as *redundant representation*. Comparison of values represented in redundant representation is not as easy as it is with non-redundant representation. For example, assume that

$$x = \vec{A} + \vec{B}$$
$$x' = \vec{A'} + \vec{B'}.$$

It is possible that $x = x'$ even though $A \neq A'$ and $B \neq B'$. Namely, in redundant representation inequality of the representations does not imply inequality of the represented values.

Some of you might wonder at this point whether redundant representations are useful at all. We just saw that redundant representation makes comparison non-trivial. The answer is that redundant representation is most useful. Probably the most noted application of redundant representation is fast multiplication. In fast multiplication, redundant representation is used for fast (redundant) addition.

We summarize this discussion by noting that an alternative way to interpret an RCA(n) (or an ADDER(n), in general) is to say that it translates a redundant representation to a non-redundant binary representation.

8.3.2 Cone of adder outputs

The correctness proof of $\textsc{rca}(n)$ implies that, for every $0 \leq i \leq n-1$,

$$\langle A[i:0]\rangle + \langle B[i:0]\rangle + C[0] = 2^{i+1} \cdot C[i+1] + \langle S[i:0]\rangle. \tag{8.6}$$

Equation 8.6 implies that, for every $0 \leq i \leq n-1$,

$$\langle S[i:0]\rangle = \mathrm{mod}(\langle A[i:0]\rangle + \langle B[i:0]\rangle + C[0], 2^{i+1}).$$

These equations imply that the cone of each of the signals $C[i+1]$ and $S[i]$ is the set of inputs corresponding to $A[i:0] \bigcup B[i:0] \bigcup C[0]$ (see the following question).

Question 8.3 *Prove that the cone of each of the signals $C[i+1]$ and $S[i]$ consists of $2i+1$ inputs corresponding to $A[i:0] \bigcup B[i:0] \bigcup C[0]$.*

8.3.3 Reductions between sum and carry bits

The correctness of $\textsc{rca}(n)$ implies that, for every $0 \leq i \leq n-1$,

$$S[i] = \textsc{xor}(A[i], B[i], C[i]). \tag{8.7}$$

This immediately implies that, for every $0 \leq i \leq n-1$,

$$C[i] = \textsc{xor}(A[i], B[i], S[i]). \tag{8.8}$$

Equations 8.7 and 8.8 imply constant-time linear-cost reductions between the problems of computing the sum bits $S[n-1:0]$ and computing the carry bits $C[n-1:0]$. (This reduction uses the addends $\vec{A}$ and $\vec{B}$.) The task of computing the sum bits is the task of an adder. In an $\textsc{rca}(n)$, the carry bit $C[i]$ is computed first, and then the sum bit $S[i]$ is computed according to Eq. 8.7. We will later design an asymptotically optimal adder that first computes all the carry bits and then obtains the sum bits from the carry-bits by applying Eq. 8.7.

Question 8.4 *Prove Equation 8.8.*

8.4 Conditional Sum Adder

A Conditional Sum Adder is a recursive adder design that is based on divide-and-conquer. One often uses only one "level" of recursion. Namely, three adders with input length $n/2$ are used to construct one adder with input size n.

8.4.1 Motivation

The following “story” captures the main idea behind a conditional sum adder.

Imagine a situation in which Alice, who is positioned on Earth, holds the strings $A[k-1:0], B[k-1:0], C[0]$. Bob, who is stationed on the Moon, holds the strings $A[n-1:k], B[n-1:k]$. The goal of Alice and Bob is to jointly compute the sum $\langle A[n-1:0]\rangle + \langle B[n-1:0]\rangle + C[0]$. They don’t care who holds the sum bits and $C[n]$, as long as one of them does. Now, sending information from Alice to Bob is costly. The first question we pose is: how many bits must Alice send to Bob? After a short thought, Alice figures out that it suffices to send $C[k]$ to Bob. Alice is happy since she only needs to pay for sending a single bit (which is a big savings compared to sending her $2k+1$ bits!).

Unfortunately, sending information from Alice to Bob takes time. Even at the speed of light, it takes a second, which is a lot compared to the time it takes to compute the sum. Suppose Bob wants to finish his task as soon as possible after receiving $C[k]$ from Alice. The second question we pose is: what should Bob do during the second it takes $C[k]$ to reach him? Since the message has only two possible values (one or zero), an industrious Bob will compute two sums; one under the assumption that $C[k] = 0$, and one under the assumption that $C[k] = 1$. Finally, when $C[k]$ arrives, Bob only needs to select between the two sums he has pre-computed.

8.4.2 Implementation

A Conditional Sum Adder is designed recursively using divide-and-conquer. A $\textsc{csa}(1)$ is simply a Full-Adder. A $\textsc{csa}(n)$, for $n > 1$ is depicted in Figure 8.3. The input is partitioned into a lower part consisting of the bits in positions $[k-1:0]$ and an upper part consisting of the bits in positions $[n-1:k]$. The lower part (handled by Alice in our short tale) is fed to a $\textsc{csa}(k)$ to produce the sum bits $S[k-1:0]$ and the carry bit $C[k]$. The upper part (handled by Bob) is fed to two $\textsc{csa}(n-k)$ circuits. The first one is given a carry-in of 0 and the second is given a carry-in of 1. These two $\textsc{csa}(n-k)$ circuits output $n-k+1$ bits each. A multiplexer selects one of these outputs according to the value of $C[k]$ which arrives from the lower part.

Question 8.5 *Prove the correctness of the* $\textsc{csa}(n)$ *design.*

8.4.3 Delay and cost analysis

To simplify the analysis we assume that $n = 2^\ell$. To optimize the cost and delay, we use $k = n/2$.

Let $d(\textsc{fa})$ denote the delay of a Full-Adder. The delay of a $\textsc{csa}(n)$ satisfies the following recurrence:

$$d(\textsc{csa}(n)) = \begin{cases} d(\textsc{fa}) & \text{if } n = 1 \\ d(\textsc{csa}(n/2)) + d(\textsc{mux}) & \text{otherwise.} \end{cases}$$

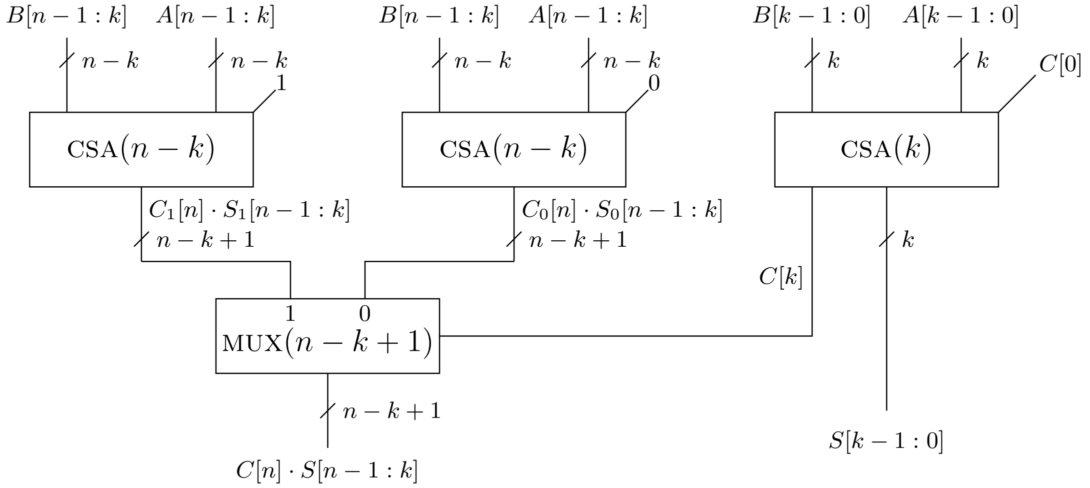


Figure 8.3: A Conditional Sum Adder $\text{CSA}(n)$.

It follows that the delay of a $\text{CSA}(n)$ is

$$\begin{aligned} d(\text{CSA}(n)) &= \ell \cdot d(\text{MUX}) + d(\text{FA}) \\ &= \Theta(\log n). \end{aligned}$$

Let $c(\text{FA})$ denote the cost of a Full-Adder. The cost of a $\text{CSA}(n)$ satisfies the following recurrence:

$$c(\text{CSA}(n)) = \begin{cases} c(\text{FA}) & \text{if } n = 1 \\ 3 \cdot c(\text{CSA}(n/2)) + (n/2+1)\cdot c(\text{MUX}) & \text{otherwise.} \end{cases}$$

Those of you familiar with the master theorem for recurrences can use it to solve this recurrence. We solve this recurrence from scratch.

To simplify notation, we ignore the exact constants and solve the recurrence

$$c(n) = \begin{cases} 3\cdot c\left(\frac{n}{2}\right) + n & \text{if } n > 1 \\ 1 & \text{if } n = 1. \end{cases} \tag{8.9}$$

We open two steps of the recurrence to get a feeling of how the different terms grow.

$$\begin{aligned} c(n) &= 3 \cdot c\left(\frac{n}{2}\right) + n \\ &= 3 \cdot \left(3 \cdot c\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n \\ &= 3^2 \cdot c\left(\frac{n}{4}\right) + n \cdot \left(1 + \frac{3}{2}\right) \end{aligned}$$

We now have a good guess (which can be proved by induction) that

$$c(n) = 3^\ell \cdot c\left(\frac{n}{2^\ell}\right) + n \cdot \left(1 + \frac{3}{2} + \cdots + \left(\frac{3}{2}\right)^{\ell-1}\right).$$

The recursion ends when $\ell = \log_2 n$. It follows that

$$\begin{aligned} c(n) &= 3^{\log_2 n} + n \cdot \frac{\left(\frac{3}{2}\right)^{\ell} - 1}{\frac{3}{2} - 1} \\ &= \Theta(n^{\log_2 3}). \end{aligned}$$

Since $\log_2 3 \approx 1.58$, we conclude that a CSA(n) is rather costly - although, for the time being, this is the only adder we know whose delay is logarithmic. We do point out that the CSA(n) design does allow us to use three half-size adders (i.e., adders with input length $n/2$) to implement a full-size adder (i.e., input length n).

Question 8.6 (effect of fanout on CSA(n)) *The fanout of the carry-bit $C[k]$ is $n/2+1$ if $k = n/2$. Suppose that we associate a delay of $\log_2(f)$ with a fanout f. How would taking the fanout into account change the delay analysis of a* CSA(n)*?*

Suppose that we associate a cost $O(f)$ with a fanout f. How would taking the fanout into account change the cost analysis of a CSA(n)*?*

8.5 Compound Adder

The Conditional Sum Adder is a divide-and-conquer design that uses two adders in the upper part, one with a zero carry-in and one with a one carry-in. This motivates the definition of an adder that computes both the sum and the incremented sum. Surprisingly, this augmented specification leads to an asymptotically cheaper design. We refer to such an adder as a Compound Adder.

Definition 8.4 *A* Compound Adder *with input length n is a combinational circuit specified as follows.*

Input: $A[n-1:0], B[n-1:0] \in \{0,1\}^n$.

Output: $S[n:0], T[n:0] \in \{0,1\}^{n+1}$.

Functionality:

$$\begin{aligned} \langle \vec{S} \rangle &= \langle \vec{A} \rangle + \langle \vec{B} \rangle \\ \langle \vec{T} \rangle &= \langle \vec{A} \rangle + \langle \vec{B} \rangle + 1. \end{aligned}$$

Note that a Compound Adder does not have carry-in input. To simplify notation, the carry-out bits are denoted by $S[n]$ for the sum and by $T[n]$ for the incremented sum. We denote a compound adder with input length n by COMP-ADDER(n).

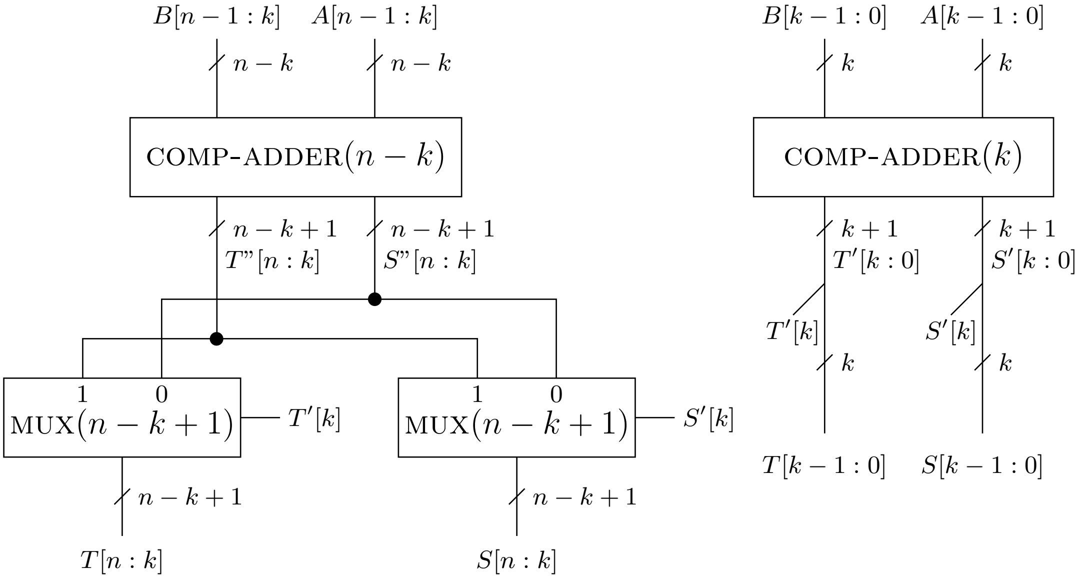


Figure 8.4: A Compound Adder COMP-ADDER(n).

8.5.1 Implementation

We apply divide-and-conquer to design a COMP-ADDER(n). For $n = 1$, we simply use a Full-Adder and a Half-Adder (one could optimize this a bit and combine the Half-Adder and the Full-Adder to reduce the constants). The design for $n > 1$ is depicted in Figure 8.4.

Example 8.1 *Consider a* COMP-ADDER(4) *with input* $A[3:0] = 0110$ *and* $B[3:0] = 1001$. *The lower part computes* $S'[2:0] = 011$ *and* $T'[2:0] = 100$. *The two lower bits of the outputs are simply* $S[1:0] = S'[1:0] = 11$ *and* $T[1:0] = T'[1:0] = 00$. *The upper part computes* $S''[4:2] = 011$ *and* $T''[4:2] = 100$. *The output* $S[4:2]$ *is selected to be* $S''[4:2]$ *since* $S'[2] = 0$. *The output* $T[4:2]$ *is selected to be* $T''[4:2]$ *since* $T'[2] = 1$. *Hence* $S[4:0] = 01111$ *and* $T[4:0] = 10000$.

Question 8.7 *Present an example for* COMP-ADDER(4) *in which* $T[4:2]$ *is selected to be* $S''[4:2]$. *Is it possible that* $S'[k] = 1$ *and* $T'[k] = 0$*? Which combinations of* $S'[k]$ *and* $T'[k]$ *are possible?*

8.5.2 Correctness

We prove the correctness of COMP-ADDER(n).

Claim 8.2 *The* COMP-ADDER(n) *design depicted in Figure 8.4 is a correct adder.*

Proof: The proof is by induction on n. The case of $n = 1$ follows from the correctness of a Full-Adder and a Half-Adder. We prove the induction step for the output $S[n : 0]$; the correctness of $T[n : 0]$ can be proved in a similar fashion and is left as an exercise.

The induction hypothesis implies that

$$\langle S'[k : 0]\rangle = \langle A[k-1 : 0]\rangle + \langle B[k-1 : 0]\rangle. \tag{8.10}$$

Note that (i) the output $S[k-1 : 0]$ equals $S'[k-1 : 0]$, and (ii) $S'[k]$ equals the carry bit $C[k]$ corresponding to the addition $\langle A[k-1 : 0]\rangle + \langle B[k-1 : 0]\rangle$.

The induction hypothesis implies that

$$\begin{aligned}\langle S''[n : k]\rangle &= \langle A[n-1 : k]\rangle + \langle B[n-1 : k]\rangle\\ \langle T''[n : k]\rangle &= \langle A[n-1 : k]\rangle + \langle B[n-1 : k]\rangle + 2^k.\end{aligned} \tag{8.11}$$

It follows from Equations 8.10 and 8.11 that

$$\langle S''[n : k]\rangle + \langle S'[k : 0]\rangle = \langle A[n-1 : 0]\rangle + \langle B[n-1 : 0]\rangle \tag{8.12}$$

We consider two cases of the carry bit $C[k]$: $C[k] = 0$ and $C[k] = 1$.

1. If $C[k] = 0$, then $S'[k] = 0$. Equation 8.12 then reduces to

$$\begin{aligned}\langle A[n-1 : 0]\rangle + \langle B[n-1 : 0]\rangle &= \langle S''[n : k]\rangle + \langle S'[k-1 : 0]\rangle\\ &= \langle S[n : k]\rangle + \langle S[k-1 : 0]\rangle = \langle S[n : 0]\rangle.\end{aligned}$$

2. If $C[k] = 1$, then $S'[k] = 1$. Equation 8.12 then reduces to

$$\begin{aligned}\langle A[n-1 : 0]\rangle + \langle B[n-1 : 0]\rangle &= \langle S''[n : k]\rangle + 2^k + \langle S'[k-1 : 0]\rangle\\ &= \langle T''[n : k]\rangle + \langle S[k-1 : 0]\rangle = \langle S[n : 0]\rangle.\end{aligned}$$

In both cases, the output $S[n : 0]$ is as required, and the claim follows. □

8.5.3 Delay and cost analysis

To simplify the analysis we assume that $n = 2^\ell$. To optimize the cost and delay, we use $k = n/2$.

The delay of a COMP-ADDER(n) satisfies the following recurrence:

$$d(\text{COMP-ADDER}(n)) = \begin{cases} d(\text{FA}) & \text{if } n = 1\\ d(\text{COMP-ADDER}(n/2)) + d(\text{MUX}) & \text{otherwise.}\end{cases}$$

It follows that the delay of a COMP-ADDER(n) is

$$\begin{aligned} d(\text{COMP-ADDER}(n)) &= \ell \cdot d(\text{MUX}) + d(\text{FA})\\ &= \Theta(\log n).\end{aligned}$$

Note that the fanout of $S'[k]$ and $T'[k]$ is $n/2+1$. If the effect of fanout on delay is taken into account, then, as in the case of $\textsc{csa}(n)$, the delay is actually $\Theta(\log^2 n)$.

The cost of a $\textsc{comp-adder}(n)$ satisfies the following recurrence:

$$c(\textsc{comp-adder}(n)) = \begin{cases} c(\textsc{fa}) + c(\textsc{ha}) & \text{if } n = 1 \\ 2 \cdot c(\textsc{comp-adder}(n/2)) + (n/2+1) \cdot c(\textsc{mux}) & \text{otherwise.} \end{cases}$$

We are already familiar with such a recurrence and conclude that $c(\textsc{comp-adder}) = \Theta(n \log n)$.

8.6 Summary

We started by defining binary addition. We reviewed the Ripple Carry Adder. We proved its correctness rigorously and used it to define the carry bits associated with addition.

We showed that the problems of computing the sum bits and the carry bits are equivalent modulo a constant-time linear-cost reduction. Since the cost of every adder is $\Omega(n)$ and the delay is $\Omega(\log n)$, we regard the problems of computing the sum bits and the carry bits as equivalently hard.

We presented an adder design called Conditional Sum Adder ($\textsc{csa}(n)$). The $\textsc{csa}(n)$ design is based on divide-and-conquer. Its delay is asymptotically optimal (if fanout is not taken into account). However, its cost is rather large, approximately $\Theta\left(n^{1.58}\right)$.

We then considered the problem of simultaneously computing the sum and incremented sum of two binary numbers. We presented a design called Compound Adder ($\textsc{comp-adder}(n)$). This design is also based on divide-and-conquer. The asymptotic delay is also logarithmic, however, the cost is $\Theta(n \cdot \log n)$.

This result is rather surprising: a $\textsc{comp-adder}(n)$ is much cheaper asymptotically than a $\textsc{csa}(n)$! You should make sure that you understand the rational behind this magic. Moreover, by adding a line of multiplexers controlled by the carry-in bit $C[0]$, one can obtain an $\textsc{adder}(n)$ from a $\textsc{comp-adder}(n)$. So the design of a $\textsc{comp-adder}(n)$ is a real improvement over the $\textsc{csa}(n)$.

In the next chapter we present an adder design that is asymptotically optimal both with respect to delay and with respect to cost. Moreover, the asymptotic delay and cost of this asymptotically optimal design is not affected by considering fanout.

Chapter 9

Fast Addition

Contents

Preliminary questions

1. Is the task of computing the sum bits harder than the task of computing the carry bits?

2. Consider a bit-serial adder implemented by a finite state machine with two states. How can one "parallelize" the computation of such a finite state machine? What about a bit serial finite state machine that computes the OR of sequence of bits?

In this chapter we present an asymptotically optimal adder design. The method we use is called parallel prefix computation. This is a quite general method and has many applications besides fast addition.

9.1 Reduction: sum-bits $\longmapsto$ carry-bits

In this section we review the reduction (presented in Section 8.3.3) of the task of computing the sum-bits to the task of computing the carry-bits.

The correctness of $\text{RCA}(n)$ implies that, for every $0 \leq i \leq n-1$,

$$S[i] = \text{XOR}_3(A[i], B[i], C[i]).$$

This implies a constant-time linear-cost reduction of the task of computing $S[n-1:0]$ to the task of computing $C[n-1:0]$. Given $C[n-1:0]$ we simply apply a bit-wise XOR of $C[i]$ with $\text{XOR}(A[i], B[i])$. The cost of this reduction is $2n$ times the cost of a XOR-gate and the delay is $2 \cdot d(\text{XOR})$.

We conclude that if we know how to compute $C[n-1:0]$ with $O(n)$ cost and $O(\log n)$ delay, then we also know how to add with $O(n)$ cost and $O(\log n)$ delay.

9.2 Computing the carry-bits

In this section we present a reduction of the problem of computing the carry-bits to a prefix computation problem (we define what a prefix computation problem is in the next section).

Consider the Full-Adder FA_i in an $\text{RCA}(n)$. The functionality of a Full-Adder implies that $C[i+1]$ satisfies:

$$C[i+1] = \begin{cases} 0 & \text{if } A[i] + B[i] + C[i] \leq 1 \\ 1 & \text{if } A[i] + B[i] + C[i] \geq 2. \end{cases} \tag{9.1}$$

The following claim follows directly from Equation 9.1.

Claim 9.1 *For every* $0 \leq i \leq n-1$,

$$\begin{aligned} A[i] + B[i] = 0 &\implies C[i+1] = 0 \\ A[i] + B[i] = 2 &\implies C[i+1] = 1 \\ A[i] + B[i] = 1 &\implies C[i+1] = C[i]. \end{aligned}$$

Claim 9.1 implies that it is easy to compute $C[i+1]$ if $A[i] + B[i] \neq 1$. It is the case $A[i] + B[i] = 1$ that creates the effect of a carry rippling across many bit positions.

The following definition is similar to the “kill, propagate, generate” signals that are often described in literature.

Definition 9.1 *The string $\sigma[n-1:-1] \in \{0,1,2\}^{n+1}$ is defined as follows:*

$$\sigma[i] \triangleq \begin{cases} 2 \cdot C[0] & \text{if } i = -1 \\ A[i] + B[i] & \text{if } i \in [0, n-1]. \end{cases}$$

Note that $\sigma[i] = 0$ corresponds to the case that the carry is "killed"; $\sigma[i] = 1$ corresponds to the case that the carry is "propagated"; and $\sigma[i] = 2$ corresponds to the case that the carry is "generated". Instead of using the letters k, p, g we use the letters $0, 1, 2$. (One advantage of our notation is that this spares the need to define addition over the set $\{k, p, g\}$.)

The definition of $\sigma[-1]$ might seem awkward at first. A simple way to interpret it is that we could replace a carry-in $C[0] = 1$ by adding two halves in the addends, namely, $A[-1] = B[-1] = 1$. The contribution of $A[-1]$ to the sum is one half, and the same holds for $B[-1]$. So instead of $C[0]$ we have $A[-1]$ and $B[-1]$. Now, we simply write $\sigma[-1] = A[-1] + B[-1]$.

9.2.1 Carry-Lookahead Adders

Carry-Lookahead adders are hierarchical designs in which addends are partitioned into blocks. Loosely speaking, for each block there is a separate carry-lookahead generator that is input the corresponding block of $\vec{\sigma}$ and a carry-in bit. This generator computes the carry bits corresponding to the block as well as a "combined" σ of this block (i.e., a (G, P) pair for the block).

Carry-lookahead Adders are constructed in a "tree-like" manner by building a hierarchy of such carry-lookahead generators. Most texts describe only two levels of such a hierarchy although it is possible to build $O(\log n)$ levels (to obtain a linear-cost logarithmic-delay adder). Since information flows both from the leaves to the root and from the root to the leaves, block diagrams of Carry-lookahead Adders contain directed cycles. Only a more detailed look at the dependencies between inputs and outputs in each block shows that these cycles do not translate to cycles in realizations of Carry-lookahead Adders. (Hence these are combinational circuits after all.)

To avoid this complication, we do not discuss here how the carry-lookahead generators are organized in a Carry-Lookahead Adders. Instead, we focus on the computation in single carry-lookahead generator. We prove that (i) the delay is logarithmic in the block size (if fanout is not considered), and (ii) the cost grows cubically as a function of the block size. This is why the block size in Carry-Lookahead Adders is limited to a small constant (i.e., 4 bits).

Before we specify carry-lookahead generators, we state the following claim that characterizes when the carry bit $C[i+1]$ equals 1.

Claim 9.2 *For every $-1 \leq i \leq n-1$,*

$$C[i+1] = 1 \qquad \Longleftrightarrow \qquad \exists j \leq i \ : \ \sigma[i:j] = 1^{i-j} \cdot 2.$$

Example 9.1 *Consider the addition with inputs* $A[3:0] = 0101$*,* $B[3:0] = 0110$*, and* $C[0] = 1$*. The vector* $\sigma[3:-1]$ *that corresponds to these inputs is* $\sigma[3:-1] = 02112$*. The vector of carry bits that corresponds to this addition is* $C[3:0] = 1111$*. Consider the bit* $C[2]$*. According to Claim 9.2* $C[2] = 1$ *simply because* $\sigma[1:-1] = 112$ *(here* $i = 1$ *and* $j = -1$*). Now consider the bit* $C[3]$*. According to the claim* $C[3] = 1$ *simply because* $\sigma[2] = 2$ *(here* $i = j = 2$*).*

Proof: We first prove that $\sigma[i:j] = 1^{i-j} \cdot 2 \Longrightarrow C[i+1] = 1$. The proof is by induction on $i - j$. In induction basis, for $i - j = 0$ is proved as follows. Since $i = j$, it follows that $\sigma[i] = 2$. We consider two cases:

- If $i = -1$, then, by the definition of $\sigma[-1]$, it follows that $C[0] = 1$.
- If $i \geq 0$, then $A[i] + B[i] = 2$. Hence, by Claim 9.1 $C[i+1] = 1$.

The induction step is proved as follows. Note that $\sigma[i:j] = 1^{i-j} \cdot 2$ implies that $\sigma[i-1:j] = 1^{i-j-1} \cdot 2$. We apply the induction hypothesis to $\sigma[i-1:j]$ and conclude that $C[i] = 1$. Since $\sigma[i] = 1$, by Claim 9.1, $C[i+1] = C[i]$, and hence, $C[i+1] = 1$, as required.

We now prove that $C[i+1] = 1 \Rightarrow \exists j \leq i : \sigma[i:j] = 1^{i-j} \cdot 2$. The proof is by induction on i. The induction basis, for $i = -1$, is proved as follows. If $C[0] = 1$, then $\sigma[-1] = 2$. We set $j = i$, and satisfy the requirement.
The induction step is proved as follows. Assume $C[i+1] = 1$. Hence,

$$\underbrace{A[i] + B[i]}_{\sigma[i]} + C[i] \geq 2.$$

We consider three cases:

- If $\sigma[i] = 0$, then we obtain a contradiction (since $C[i]$ is not greater than 1).
- If $\sigma[i] = 2$, then we set $j = i$.
- If $\sigma[i] = 1$, then $C[i]$ must equal 1.

We conclude that

$$\begin{aligned} C[i] = 1 \quad & \overset{\text{Ind. Hyp.}}{\Longrightarrow} \quad \exists j \leq i : \sigma[i-1:j] = 1^{i-j-1} \cdot 2 \\ & \overset{\sigma[i]=1}{\Longrightarrow} \quad \exists j \leq i : \sigma[i:j] = 1^{i-j} \cdot 2. \end{aligned}$$

This completes the proof of the claim. □

We are now ready to specify carry-lookahead generators.

Definition 9.2 *A* carry-lookahead generator *with block length* n *is a combinational circuit specified as follows.*

Input: $\sigma[n-1:-1] \in \{0,1,2\}^{n+1}$,

Output: $C[n:1] \in \{0,1\}^n$.

Functionality:

$$C[i+1] = 1 \qquad \Longleftrightarrow \qquad \exists j \leq i \;:\; \sigma[i:j] = 1^{i-j} \cdot 2. \tag{9.2}$$

At this point, many readers might be anxious to see a concrete definition of how $\sigma[i] \in \{0,1,2\}$ is represented. There are, of course, many ways to represent elements in $\{0,1,2\}$ using bits; all such reasonable methods use $2-3$ bits and incur constant cost and delay. We describe a few options for representing $\{0,1,2\}$.

1. One could, of course, represent $\sigma[i]$ by the pair $(A[i], B[i])$, in which case the incurred cost and delay are zero.

2. Binary representation could be used. In this case $\sigma[i]$ is represented by a pair $x[1:0]$. Since the value 3 is not allowed, we conclude that $x[0]$ signifies whether $\sigma[i] = 1$. Similarly, $x[1]$ signifies whether $\sigma[i] = 2$. The cost and delay in computing the binary representation of $\sigma[i]$ from $A[i]$ and $B[i]$ is the cost and delay of a Half-Adder.

3. 1-out-of-3 representation could be used. This requires a NOR-gate, a XOR-gate, and an AND-gate.

Regardless of the representation that one may choose to represent $\sigma[i]$, note that one can compare $\sigma[i]$ with 1 or 2 with constant delay and cost. In fact, in binary representation and 1-out-of-3 representation, such a comparison incurs zero cost and delay.

Implementation. Figure 9.1 depicts how the carry-bit $C[i+1]$ is computed in a carry-lookahead generator. For every $-1 \leq j \leq i$, one compares the block $\sigma[i:j]$ with $1^{i-j} \cdot 2$. This comparison is depicted in the left hand side of Fig. 9.1. Each symbol $\sigma[i], \ldots, \sigma[j+1]$ is compared with 1 and the symbol $\sigma[j]$ is compared with 2. The results of these $i-j+1$ comparisons are fed to an AND-tree. The output is denoted by $\sigma[i:j] \overset{?}{=} 1^{i-j} \cdot 2$.

The outcomes of the $i+2$ comparisons of blocks $\sigma[i:j]$ with $1^{i-j} \cdot 2$ (for $j = -1, \ldots, i$) are fed to an OR-tree. The OR-tree outputs the carry-bit $C[i+1]$.

Cost and delay analysis. The delay associated with computing $C[i+1]$ in this fashion is clearly logarithmic. Assume that the comparisons are for free. It follows that the cost

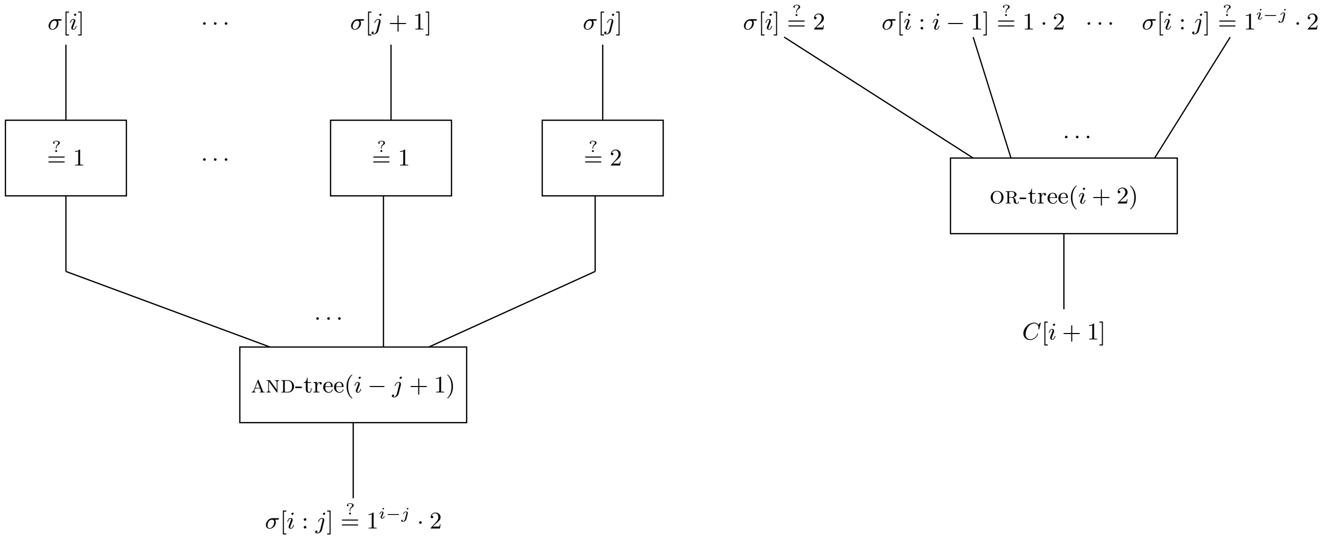


Figure 9.1: Computing the carry-bit $C[i+1]$ in a carry-lookahead generator.

of computing the carry bits $C[n:1]$ in the fashion depicted in Fig 9.1 is

$$\begin{aligned}\sum_{i=1}^{n} c(\text{lookahead } C[i+1]) &= \sum_{i=1}^{n}\left(\sum_{j=-1}^{i} c(\textsc{or}\text{-tree}(i+2)) + c(\textsc{and}\text{-tree}(i-j+1))\right)\\ &= \sum_{i=1}^{n}\left(\sum_{j=-1}^{i}\Theta(i-j)\right)\\ &= \sum_{i=1}^{n}\Theta(i^2)\\ &= \Theta(n^3).\end{aligned}$$

We conclude that the cost per block is cubic in the length of the block.

9.2.2 Reduction to prefix computation

The main disadvantage of the carry-lookahead generator implementation depicted in Fig. 9.1 is that disjoint circuits are used for computing each carry bit. For example, if $\sigma[15:2] = 1^{13}\cdot 2$, then we can immediately conclude that $\sigma[i:2] = 1^{i-2}\cdot 2$, for every $2 \leq i \leq 15$. Nevertheless, in a carry-lookahead generator, separate circuits "repeat" this computation. Our goal is therefore to reduce cost by sharing. For this purpose we start by presenting a simplified form of the problem of computing the carry bits.

Definition 9.3 *The dyadic operator* $* : \{0,1,2\}\times\{0,1,2\} \longrightarrow \{0,1,2\}$ *is defined by the following table.*

*	0	1	2
0	0	0	0
1	0	1	2
2	2	2	2

We use the notation $a * b$ to denote the result of applying the function $*$ to a and b.

Claim 9.3 *For every $a \in \{0,1,2\}$:*

$$\begin{aligned} 0 * a &= 0 \\ 1 * a &= a \\ 2 * a &= 2. \end{aligned}$$

Claim 9.4 *The function $*$ is associative. Namely,*

$$\forall a,b,c \in \{0,1,2\} : (a * b) * c = a * (b * c).$$

Question 9.1 *(i) Prove claim 9.4. (ii) Is the function $*$ commutative?*

We refer to the outcome of applying the $*$ function by the $*$-product. We also use the notation

$$\pi[i:j] \triangleq \sigma[i] * \cdots * \sigma[j].$$

Note that if $i = j$ then $\pi[i:j] \triangleq \sigma[i]$. Also if $i < j$, then $\pi[i:j] \triangleq 1$.

Associativity of $*$ implies that for every $i > j \geq k$:

$$\pi[i:k] = \pi[i:j+1] * \pi[j:k].$$

The reduction of the computation of the carry-bits to a prefix computation is based on the following claim.

Claim 9.5 *For every $-1 \leq i \leq n-1$,*

$$C[i+1] = 1 \qquad \Longleftrightarrow \qquad \pi[i:-1] = 2.$$

Proof: From Claim 9.2, it suffices to prove that

$$\exists j \leq i \ : \ \sigma[i:j] = 1^{i-j} \cdot 2 \qquad \Longleftrightarrow \qquad \pi[i:-1] = 2.$$

($\Rightarrow$) Assume that $\sigma[i:j] = 1^{i-j} \cdot 2$. It follows that

$$\pi[i:j] = 2.$$

If $j = -1$ we are done. Otherwise, by associativity and by Claim 9.3 it follows that

$$\begin{aligned} \pi[i:-1] &= \underbrace{\pi[i:j]}_{=2} * \pi[j-1:-1] \\ &= 2. \end{aligned}$$

($\Leftarrow$) Assume that $\pi[i:-1]=2$. If, for every $\ell \leq i$, $\sigma[\ell] \neq 2$, then $\pi[i:-1] \neq 2$, a contradiction. Hence

$$\{\ell \;:\; \sigma[\ell]=2 \text{ and } -1 \leq \ell \leq i\} \;\neq\; \emptyset.$$

Let

$$\ell^* \triangleq \max\{\ell \in [-1,i] \;:\; \sigma[\ell]=2\}.$$

If $\ell^*=i$, then $\sigma[i]=2$ and we are done. So we assume that that $\ell^*<i$. We claim that $\sigma[j]=1$, for every $\ell^*<j\leq i$.

By the definition of ℓ^*, $\sigma[j]\neq 2$, for every $\ell^*<j\leq i$. If $\sigma[j]=0$, for $\ell^*<j\leq i$, then $\pi[i:j]=0$, and then $\pi[i:-1]=\pi[i:j]*\pi[j-1:-1]=0$, a contradiction.

Since $\sigma[i:\ell^*+1]=1^{i-\ell^*}$, we conclude that $\sigma[i:j]=1^{i-j}\cdot 2$, and the claim follows. □

A prefix computation problem is defined as follows.

Definition 9.4 *Let Σ denote a finite alphabet. Let* OP $:\Sigma^2 \longrightarrow \Sigma$ *denote an associative function. A* prefix computation *over Σ with respect to* OP *is defined as follows.*

Input $x[n-1:0]\in\Sigma^n$.

Output: $y[n-1:0]\in\Sigma^n$ *defined recursively as follows:*

$$\begin{aligned} y[0] &\leftarrow x[0] \\ y[i+1] &= \text{OP}(x[i+1], y[i]). \end{aligned}$$

Note that $y[i]$ can be also expressed simply by

$$y_i = \text{OP}_{i+1}(x[i], x[i-1], \ldots, x[0]).$$

Claim 9.5 implies a reduction of the problem of computing the carry-bits $C[n:1]$ to the prefix computation problem over $\{0,1,2\}$ with respect to the associative operator $*$.

9.3 Parallel prefix computation

In this section we present a general asymptotically optimal circuit for the prefix computation problem.

As in the previous section, let OP $:\Sigma^2 \longrightarrow \Sigma$ denote an associative function. We do not address the issue of how values in Σ are represented by binary strings. We do assume that some fixed representation is used. Moreover, we assume the existence of a OP-gate that given representations of $a,b\in\Sigma$ outputs a representation of OP(a,b).

Definition 9.5 *A* Parallel Prefix Circuit, PPC–OP(n), *is a combinational circuit that computes a prefix computation. Namely, given input $x[n-1:0]\in\Sigma^n$, it outputs $y[n-1:0]\in\Sigma^n$, where*

$$y_i = \text{OP}_{i+1}(x[i], x[i-1], \ldots, x[0]).$$

Example 9.2 *A Parallel Prefix Circuit with (i) the alphabet $\Sigma = \{0,1\}$ and (ii) the function* $\textsc{op} = \textsc{or}$ *is exactly the* $\textsc{ppc–or}(n)$ *circuit from Definition 6.3.*

Example 9.3 *Consider $\Sigma = \{0,1,2\}$ and* $\textsc{op} = *$. *The Parallel Prefix Circuit with (i) the alphabet $\Sigma = \{0,1,2\}$ and (ii) the function* $\textsc{op} = *$ *can be used (according to Claim 9.5) to compute the carry-bits.*

Our goal is to design a $\textsc{ppc–op}(n)$ using only $\textsc{op}$-gates. The cost of the design is the number of $\textsc{op}$-gates used. The delay is the maximum number of $\textsc{op}$-gates along a directed path from an input to an output.

Question 9.2 *Design a* $\textsc{ppc–op}(n)$ *circuit with linear delay and cost.*

Question 9.3 *Design a* $\textsc{ppc–op}(n)$ *circuit with logarithmic delay and quadratic cost.*

Question 9.4 *Assume that a design $C(n)$ is a* $\textsc{ppc–op}(n)$. *This means that it is comprised only of* $\textsc{op}$*-gates and works correctly for every alphabet Σ and associative function* $\textsc{op} : \Sigma^2 \rightarrow \Sigma$. *Can you prove a lower bound on its cost and delay?*

9.3.1 Implementation

In this section we present a linear-cost logarithmic-delay $\textsc{ppc–op}(n)$ design. The design is recursive and uses a technique that we name "odd-even" since even indexed inputs and outputs are handled differently than odd indexed inputs and outputs.

The design we present is a recursive design. For simplicity, we assume that n is a power of 2. The design for $n = 2$ simply outputs $y[0] \leftarrow x[0]$ and $y[1] \leftarrow \textsc{op}(x[0], x[1])$. The recursion step is depicted in Figure 9.2. Adjacent inputs are paired and fed to an $\textsc{op}$-gate. The $n/2$ outputs of the $\textsc{op}$-gates are fed to a $\textsc{ppc–op}(n/2)$. The outputs of the $\textsc{ppc–op}(n/2)$ circuit are directly connected to the odd indexed outputs, namely, $y[2i+1] \leftarrow y'[i]$. Observe that wires carrying the inputs with even indexes are drawn (or routed) over the $\textsc{ppc–op}(n/2)$ box; these "even indexed" wires are not part of the $\textsc{ppc–op}(n/2)$ design. The even indexed outputs (for $i > 0$) are obtained as follows: $y[2i] \leftarrow \textsc{op}(x[2i], y'[i-1])$.

9.3.2 Correctness

Claim 9.6 *The design depicted in Fig. 9.2 is correct.*

Proof: The proof of the claim is by induction. The induction basis holds trivially for $n = 2$. We now prove the induction step. Consider the $\textsc{ppc–op}(n/2)$ used in a $\textsc{ppc–op}(n)$. Let $x'[n/2-1:0]$ and $y'[n/2-1:0]$ denote the inputs and outputs of the $\textsc{ppc–op}(n/2)$, respectively. The ith input $x'[i]$ equals $\textsc{op}(x[2i+1], x[2i])$. By associativity and the induction hypothesis, the ith output $y'[i]$ satisfies:

$$\begin{aligned}
y'[i] &= \textsc{op}_{i+1}(x'[i], \ldots, x'[0]) \\
&= \textsc{op}_{i+1}(\textsc{op}(x[2i+1], x[2i]), \ldots, \textsc{op}(x[1], x[0])) \\
&= \textsc{op}_{2i+2}(x[2i+1], \ldots, x[0]).
\end{aligned}$$

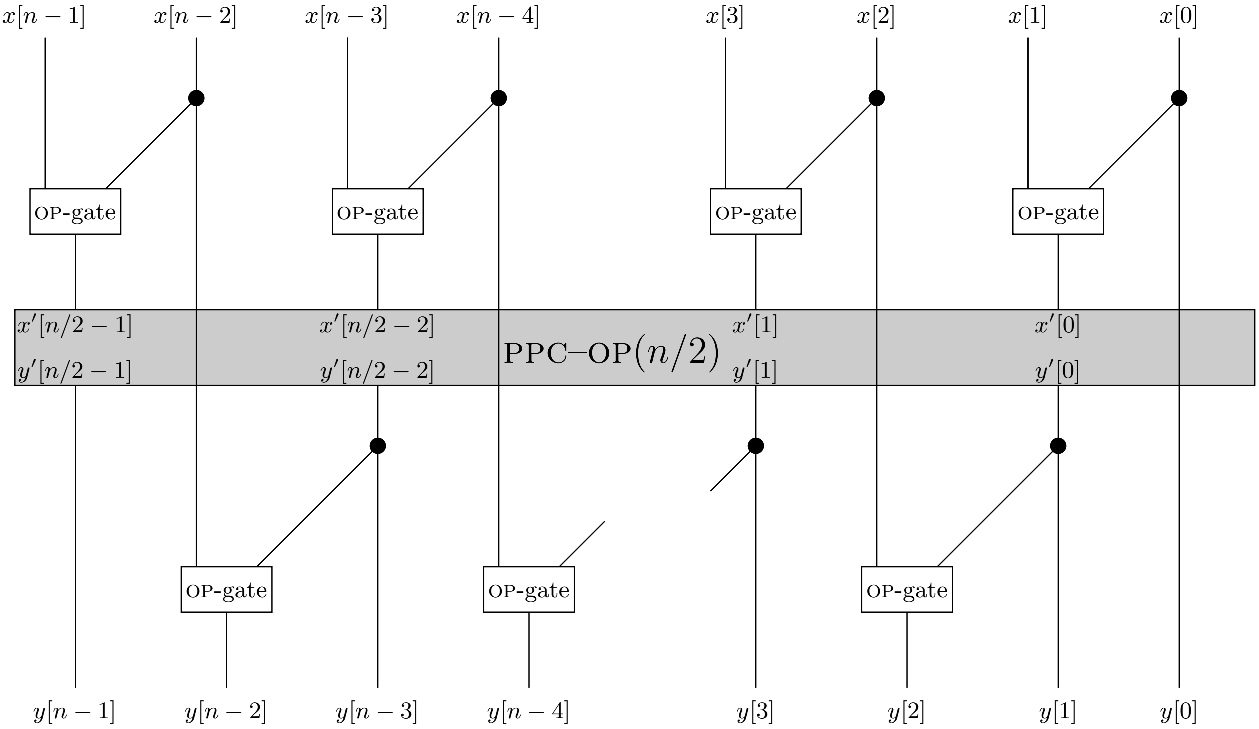


Figure 9.2: A recursive design of PPC–OP(n). (The even indexed wires x[0],x[2],... are pass above the PPC–OP($n/2$) box only to simplify the drawing. These wires are not inputs/outputs of the PPC–OP($n/2$) sub-circuit.)

Since $y[2i+1]$ equals $y'[i]$, it follows that the odd indexed outputs $y[1], y[3], \ldots, y[n-1]$ are correct. Finally, $y[2i]$ equals $\textsc{op}(x[2i], y'[i-1])$, and hence $y[2i] = \textsc{op}(x[2i], y[2i-1])$. It follows that the even indexed outputs are also correct, and the claim follows. □

9.3.3 Delay and cost analysis

The delay of the $\textsc{ppc–op}(n)$ circuit satisfies the following recurrence:

$$d(\textsc{ppc–op}(n)) = \begin{cases} d(\textsc{op}\text{-gate}) & \text{if } n = 2 \\ d(\textsc{ppc–op}(n/2)) + 2 \cdot d(\textsc{op}\text{-gate}) & \text{otherwise.} \end{cases}$$

If follows that

$$d(\textsc{ppc–op}(n)) = (2\log n - 1) \cdot d(\textsc{op}\text{-gate}).$$

The cost of the $\textsc{ppc–op}(n)$ circuit satisfies the following recurrence:

$$c(\textsc{ppc–op}(n)) = \begin{cases} c(\textsc{op}\text{-gate}) & \text{if } n = 2 \\ c(\textsc{ppc–op}(n/2)) + (n-1) \cdot c(\textsc{op}\text{-gate}) & \text{otherwise.} \end{cases}$$

Let $n = 2^k$, it follows that

$$\begin{aligned} c(\textsc{ppc–op}(n)) &= \sum_{i=2}^{k} (2^i - 1) \cdot c(\textsc{op}\text{-gate}) + c(\textsc{op}\text{-gate}) \\ &= ((2n - 1 - (1+2)) - (k-1) + 1) \cdot c(\textsc{op}\text{-gate}) \\ &= (2n - \log n - 2) \cdot c(\textsc{op}\text{-gate}). \end{aligned}$$

Corollary 9.7 *If the delay and cost of an* $\textsc{op}$*-gate is constant, then*

$$\begin{aligned} d(\textsc{ppc–op}(n)) &= \Theta(\log n) \\ c(\textsc{ppc–op}(n)) &= \Theta(n). \end{aligned}$$

Corollary 9.7 implies that $\textsc{ppc–op}(n)$ with $\Sigma = \{0,1\}$ and $\textsc{op} = \textsc{or}$ is an asymptotically optimal $\textsc{ppc–or}(n)$. It also implies that we can compute the carry-bit corresponding to an addition in linear cost and logarithmic delay.

Question 9.5 *In this question we consider fanout in the* $\textsc{ppc–op}(n)$ *design and suggest a way to reduce the fanout so that it is at most two.*

- *What is the maximum fanout in the* $\textsc{ppc–op}(n)$ *design?*
- *Show that if a buffer is inserted in every branching point of the* $\textsc{ppc–op}(n)$ *design (such branching points are depicted by filled circles), then the fanout is constant. (A buffer is a combinational circuit that implements the identity function. A buffer is often implemented by cascading two inverters.)*
- *By how much does the insertion of buffers increase the cost and delay?*

9.4 Putting it all together

In this section we assemble an asymptotically optimal adder. The stages of the construction are as follows.

Compute $\sigma[n-1:-1]$: In this step the symbols $\sigma[i] \in \{0,1,2\}$ are computed. This step can be regarded as an encoding step; the sum $A[i]+B[i]$ is encoded to represent the corresponding value in $\{0,1,2\}$. The cost and delay of this step depend on the representation used to represent values in $\{0,1,2\}$. In any case, the cost and delay is constant per $\sigma[i]$, hence, the total cost is $O(n)$ and the total delay is $O(1)$.

$\textsc{ppc}-*(n)$: In this step the products $\pi[i:-1]$ are computed from $\sigma[i:-1]$, for every $i \in [n-1:0]$. The cost and delay of this step are $O(n)$ and $O(\log n)$, respectively.

Extraction of $C[n:1]$: By Claim 9.5, it follows that $C[i+1]=1$ iff $\pi[i:-1]=2$. In this stage we compare each product $\pi[i:-1]$ with 2. The result of this comparison equals $C[i+1]$. The cost and delay of this step is constant per carry-bit $C[i+1]$. It follows that the cost of this step is $O(n)$ and the delay is $O(1)$.

Computation of sum-bits: The sum bits are computed by applying

$$S[i] = \textsc{xor}_3(A[i], B[i], C[i]).$$

The cost and delay of this step is constant per sum-bit. It follows that the cost of this step is $O(n)$ and the delay is $O(1)$.

By combining the cost and delay of each stage we obtain the following result.

Theorem 9.8 *The adder based on parallel prefix computation is asymptotically optimal; its cost is linear and its delay is logarithmic.*

Remark 9.1 *A careful reader may notice that cost can be reduced since it may not be needed to compute $\textsc{xor}(A[i], B[i])$ in the last stage. The reason is that in certain representations of $\sigma[i]$ (which?) this xor is is computed in the first stage.*

9.5 Summary

In this chapter we presented an adder with asymptotically optimal cost and delay. The adder is based on a reduction of the task of computing the sum-bits to the task of computing the carry bits. We then reduce the task of computing the sum bits to a parallel prefix computation problem.

A parallel prefix computation problem is the problem of computing $\textsc{op}_i(x[i-1:0])$, for $0 \le i \le n-1$, where $\textsc{op}$ is an associative operation. We present a linear cost logarithmic delay circuit for the parallel prefix computation problem. We refer to this circuit as $\textsc{ppc}–\textsc{op}(n)$. This design has two implications:

- This design computes the product prefixes of the "carry generate-propagate-kill" signals corresponding to an addition. These product prefixes are translated to sum bits in constant time and linear cost. Hence, an asymptotically optimal cost and delay adder is obtained.

- This design is used to design an optimal cost and delay PPC–OR(n) circuit. This in turn leads to optimal priority encoder designs (both unary and binary).

It is possible to design asymptotically optimal adders based on Carry-Lookahead Adders. We chose not to describe it because the design is less systematic and is less general.

Chapter 10

Signed Addition

Contents

Preliminary questions

1. How are signed integers represented in a computer?
2. How are signed integers added and subtracted in a computer?
3. Can we use the same circuitry for adding unsigned and signed integers?

In this chapter we present circuits for adding and subtracting signed numbers that are represented by two's complement representation. Although the designs are obtained by very minor changes of a binary adder designs, the theory behind these changes requires some effort.

10.1 Representation of negative integers

We use binary representation to represent non-negative integers. We now address the issue of representing positive and negative integers. Following programming languages, we refer to non-negative integers as *unsigned numbers* and to negative and positive numbers as *signed numbers*.

There are three common methods for representing signed numbers: sign-magnitude, one's complements, and two's complement.

Definition 10.1 *The number represented in* sign-magnitude *representation by* $A[n-1:0] \in \{0,1\}^n$ *and* $S \in \{0,1\}$ *is*

$$(-1)^S \cdot \langle A[n-1:0] \rangle .$$

Definition 10.2 *The number represented in* one's complement *representation by* $A[n-1:0] \in \{0,1\}^n$ *is*

$$-(2^{n-1}-1) \cdot A[n-1] + \langle A[n-2:0] \rangle .$$

Definition 10.3 *The number represented in* two's complement *representation by* $A[n-1:0] \in \{0,1\}^n$ *is*

$$-2^{n-1} \cdot A[n-1] + \langle A[n-2:0] \rangle .$$

We denote the number represented in two's complement representation by $A[n-1:0]$ as follows:

$$[A[n-1:0]] \stackrel{\triangle}{=} -2^{n-1} \cdot A[n-1] + \langle A[n-2:0] \rangle .$$

We often use the term "a two's complement number $A[n-1]$" as an abbreviation of the longer phrase "the number represented by $A[n-1:0]$ in two's complement representation".

The most common method for representing signed numbers is two's complement representation. The main reason is that adding, subtracting, and multiplying signed numbers represented in two's complement representation is almost as easy as performing these computations on unsigned (binary) numbers.

10.2 Negation in two's complement representation

We denote the set of signed numbers that are representable in two's complement representation using n-bit binary strings by T_n.

Claim 10.1

$$T_n \triangleq \left\{-2^{n-1}, -2^{n-1}+1, \ldots, 2^{n-1}-1\right\}.$$

Question 10.1 *Prove Claim 10.1*

The following claim deals with negating a value represented in two's complement representation.

Claim 10.2

$$-\left[A[n-1:0]\right] = \left[\text{INV}(A[n-1:0])\right] + 1.$$

Proof: Note that $\text{INV}(A[i]) = 1 - A[i]$. Hence,

$$\begin{aligned}
\left[\text{INV}(A[n-1:0])\right] &= -2^{n-1} \cdot \text{INV}(A[n-1]) + \langle \text{INV}(A[n-2:0]) \rangle \\
&= -2^{n-1} \cdot (1 - A[n-1]) + \sum_{i=0}^{n-2} (1 - A[i]) \cdot 2^i \\
&= \underbrace{-2^{n-1} + \sum_{i=0}^{n-2} 2^i}_{=-1} + \underbrace{2^{n-1} \cdot A[n-1] - \sum_{i=0}^{n-2} A[i])}_{=-[A[n-1:0]]} \\
&= -1 - \left[A[n-1:0]\right].
\end{aligned}$$

□

In Figure 10.1 we depict a design for negating numbers based on Claim 10.2. The circuit is input $\vec{A}$ and is supposed to compute the two's complement representation of $-\left[\vec{A}\right]$. The bits in the string $\vec{A}$ are first inverted to obtain $\overline{A}[n-1:0]$. An increment circuit outputs $C[n] \cdot B[n-1:0]$ such that

$$\langle C[n] \cdot B[n-1:0] \rangle = \langle \overline{A}[n-1:0] \rangle + 1.$$

Such an increment circuit can be implemented simply by using a binary adder with one addend string fixed to $0^{n-1} \cdot 1$.

We would like to claim that the circuit depicted in Fig. 10.1 is correct. Unfortunately, we do not have yet the tools to prove the correctness. Let us try and see the point in which we run into trouble.

Claim 10.2 implies that all we need to do to compute $-\left[\vec{A}\right]$ is invert the bits of $\vec{A}$ and increment. The problem is with the meaning of increment. The increment circuit computes:

$$\langle \overline{A}[n-1:0] \rangle + 1.$$

However, Claim 10.2 requires that we compute

$$\left[\overline{A}[n-1:0]\right] + 1.$$

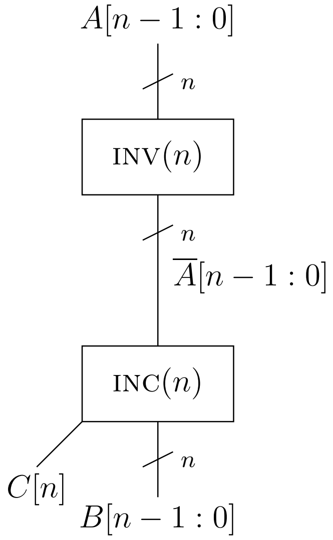


Figure 10.1: A (wrong) circuit for negating a value represented in two's complement representation.

Now, let $C[n] \cdot B[n-1:0]$ denote the output of the incrementor. We know that

$$\langle C[n] \cdot B[n-1:0] \rangle = \langle \overline{A}[n-1:0] \rangle + 1.$$

One may suspect that if $C[n] = 1$, then correctness might fail due to the "lost" carry-bit. Assume we are "lucky" and $C[n] = 0$. In this case,

$$\langle B[n-1:0] \rangle = \langle \overline{A}[n-1:0] \rangle + 1.$$

Why should this imply that

$$[B[n-1:0]] = \left[\overline{A}[n-1:0]\right] + 1?$$

At this point we leave this issue unresolved. We prove a more general result in Theorem 10.7. (Question 10.12 deals with the correctness of the circuit for negating two's complement numbers.) Note, however, that the circuit errs with the input $A[n-1:0] = 1 \cdot 0^{n-1}$. The value represented by $\vec{A}$ equals -2^{n-1}. Inversion yields $\overline{A}[n-1:0] = 0 \cdot 1^{n-2}$. Increment yields $C[n] = 0$ and $B[n-1:0] = 1 \cdot 0^{n-2} = A[n-1:0]$. This, of course, is not a counter-example to Claim 10.2. It is an example in which an increment with respect to $\langle \overline{A}[n-1:0] \rangle$ is not an increment with respect to $\left[\overline{A}[n-1:0]\right]$. This is exactly the point which concerned us. A more careful look at this case shows that every circuit must err with such an input. The reason is that $-\left[\vec{A}\right] \notin T_n$. Hence, the negated value cannot be represented using an n-bit string, and negation had to fail.

Interestingly, negation is easier with respect to the other two representations of signed numbers.

Question 10.2 *Propose circuits for negation with respect to other two representations of signed numbers: sign-magnitude and one's-complement. (Make sure your designs correctly invert zero!)*

10.3 Properties of two's complement representation

Alternative definition of two's complement representation. The following claim follows immediately from the definition of two's complement representation.

Claim 10.3 *For every* $A[n-1:0] \in \{0,1\}^n$

$$\text{mod}(\langle \vec{A} \rangle, 2^n) = \text{mod}(\left[\vec{A}\right], 2^n).$$

The importance of Claim 10.3 is that it provides an explanation for the definition of two's complement representation. In fact, one could define two's complement representation based on the claim. Namely, represent $x \in [-2^{n-1}, 2^{n-1}-1]$ by $x' \in [0, 2^n - 1]$, where $\text{mod}(x, 2^n) = \text{mod}(x', 2^n)$.

Question 10.3 *Prove Claim 10.3.*

Sign bit. The most significant bit $A[n-1]$ of a string $A[n-1:0]$ that represents a two's complement number is often called the *sign-bit* of $\vec{A}$. The following claim justifies this term.

Claim 10.4

$$[A[n-1:0]] < 0 \quad \Longleftrightarrow \quad A[n-1] = 1.$$

Question 10.4 *Prove Claim 10.4.*

Do not be misled by the term sign-bit. Two's complement representation is not sign-magnitude representation. In particular, the prefix $A[n-2:0]$ is not a binary representation of the magnitude of $[A[n-1:0]]$. Computing the absolute value of a negative signed number represented in two's complement representation involves inversion of the bits and an increment (as suggested by Claim 10.2).

Sign extension. The following claim is often referred to as "sign-extension". It basically means that duplicating the most significant bit does not affect the value represented in two's complement representation. This is similar to padding zeros from the left in binary representation.

Claim 10.5 *If* $A[n] = A[n-1]$, *then*

$$[A[n:0]] = [A[n-1:0]].$$

Proof:

$$\begin{aligned}
[A[n:0]] &= -2^n \cdot A[n] + \langle A[n-1:0]\rangle \\
&= -2^n \cdot A[n] + 2^{n-1} \cdot A[n-1] + \langle A[n-2:0]\rangle \\
&= -2^n \cdot A[n-1] + 2^{n-1} \cdot A[n-1] + \langle A[n-2:0]\rangle \\
&= -2^{n-1} \cdot A[n-1] + \langle A[n-2:0]\rangle \\
&= [A[n-1:0]] \,.
\end{aligned}$$

□

We can now apply arbitrarily long sign-extension, as summarized in the following Corollary.

Corollary 10.6

$$[A[n-1]^* \cdot A[n-1:0]] = [A[n-1:0]] \,.$$

Question 10.5 *Prove Corollary 10.6.*

10.4 Reduction: two's complement addition to binary addition

In Section 10.2 we tried (and partly failed) to use a binary incrementor for incrementing a two's complement signed number. In this section we deal with a more general case, namely computing the two's complement representation of

$$\left[\vec{A}\right] + \left[\vec{B}\right] + C[0].$$

The following theorem deals with the following setting. Let

$$\begin{aligned}
A[n-1:0], B[n-1:0], S[n-1:0] &\in \{0,1\}^n \\
C[0], C[n] &\in \{0,1\}
\end{aligned}$$

satisfy

$$\langle A[n-1:0]\rangle + \langle B[n-1:0]\rangle + C[0] = \langle C[n] \cdot S[n-1:0]\rangle. \tag{10.1}$$

Namely, $\vec{A}, \vec{B}$, and $C[0]$ are fed to a binary adder ADDER(n) and $\vec{S}$ and $C[n]$ are output by the adder. The theorem addresses the following questions:

- When does the output $S[n-1:0]$ satisfy:

$$\left[\vec{S}\right] = [A[n-1:0]] + [B[n-1:0]] + C[0]? \tag{10.2}$$

- How can we know that Equation 10.2 holds?

Theorem 10.7 *Let $C[n-1]$ denote the carry-bit in position $[n-1]$ associated with the binary addition described in Equation 10.1 and let*

$$z \triangleq [A[n-1:0]] + [B[n-1:0]] + C[0].$$

Then,

$$\begin{aligned} C[n] - C[n-1] = 1 &\quad\Longrightarrow\quad z < -2^{n-1} && (10.3)\\ C[n-1] - C[n] = 1 &\quad\Longrightarrow\quad z > 2^{n-1} - 1 && (10.4)\\ z \in T_n &\quad\Longleftrightarrow\quad C[n] = C[n-1] && (10.5)\\ z \in T_n &\quad\Longrightarrow\quad z = [S[n-1:0]]\,. && (10.6) \end{aligned}$$

Proof: Recall that the definition of the functionality of FA_{n-1} in a Ripple-Carry Adder $\text{RCA}(n)$ implies that

$$A[n-1] + B[n-1] + C[n-1] = 2C[n] + S[n-1].$$

Hence

$$A[n-1] + B[n-1] = 2C[n] - C[n-1] + S[n-1]. \tag{10.7}$$

We now expand z as follows:

$$\begin{aligned} z &= [A[n-1:0]] + [B[n-1:0]] + C[0]\\ &= -2^{n-1}\cdot(A[n-1] + B[n-1]) + \langle A[n-2:0]\rangle + \langle B[n-2:0]\rangle + C[0]\\ &= -2^{n-1}\cdot(2C[n] - C[n-1] + S[n-1]) + \langle C[n-1]\cdot S[n-2:0]\rangle, \end{aligned}$$

where the last line is based on Equation 10.7 and on

$$\langle A[n-2:0]\rangle + \langle B[n-2:0]\rangle + C[0] = \langle C[n-1]\cdot S[n-2:0]\rangle.$$

Commuting $S[n-1]$ and $C[n-1]$ implies that

$$\begin{aligned} z &= -2^{n-1}\cdot(2C[n] - C[n-1] - C[n-1]) + [S[n-1]\cdot S[n-2:0]]\\ &= -2^{n}\cdot(C[n] - C[n-1]) + [S[n-1:0]]\,. \end{aligned}$$

We distinguish between three cases:

1. If $C[n] - C[n-1] = 1$, then

$$\begin{aligned} z &= -2^n + [S[n-1:0]]\\ &\le -2^n + 2^{n-1} - 1 = -2^{n-1} - 1. \end{aligned}$$

   Hence Equation 10.3 follows.

2. If $C[n] - C[n-1] = -1$, then

$$\begin{aligned} z &= 2^n + [S[n-1:0]]\\ &\ge 2^n - 2^{n-1} = 2^{n-1}. \end{aligned}$$

   Hence Equation 10.4 follows.

3. If $C[n] = C[n-1]$, then $z = [S[n-1:0]]$, and obviously $z \in T_n$.

Equation 10.5 follows from the fact that if $C[n] \neq C[n-1]$, then either $C[n]-C[n-1] = 1$ or $C[n-1] - C[n] = 1$. In both these cases $z \notin T_n$. Equation 10.6 follows from the third case as well, and the theorem follows. □

10.4.1 Detecting overflow

Overflow occurs when the sum of signed numbers is not in T_n. Using the notation of Theorem 10.7, overflow is defined as follows.

Definition 10.4 *Let* $z \triangleq [A[n-1:0]] + [B[n-1:0]] + C[0]$. *The signal* OVF *is defined as follows:*

$$\text{OVF} \triangleq \begin{cases} 1 & \textit{if } z \notin T_n \\ 0 & \textit{otherwise.} \end{cases}$$

Note that overflow means that the sum is either too large or too small. Perhaps the term "out-of-range" is more appropriate than "overflow" (which suggests that the sum is too big). We choose to favor tradition here and follow the common term overflow rather than introduce a new term.

By Theorem 10.7, overflow occurs iff $C[n-1] \neq C[n]$. Namely,

$$\text{OVF} = \text{XOR}(C[n-1], C[n]).$$

Moreover, if overflow does not occur, then Equation 10.2 holds. Hence, we have a simple way to answer both questions raised before the statement of Theorem 10.7. The signal $C[n-1]$ may not be available if one uses a "black-box" binary-adder (e.g., a library component in which $C[n-1]$ is an internal signal). In this case we detect overflow based on the following claim.

Claim 10.8

$$\text{XOR}(C[n-1], C[n]) = \text{XOR}_4(A[n-1], B[n-1], S[n-1], C[n]).$$

Proof: Recall that

$$C[n-1] = \text{XOR}_3(A[n-1], B[n-1], S[n-1]).$$

□

Question 10.6 *Prove that*

$$\text{OVF} = \text{OR}(\text{AND}_3(A[n-1], B[n-1], \text{INV}(S[n-1])), \text{AND}_3(\text{INV}(A[n-1]), \text{INV}(B[n-1]), S[n-1])).$$

10.4.2 Determining the sign of the sum

How do we determine the sign of the sum z? Obviously, if $z \in T_n$, then Claim 10.4 implies that $S[n-1]$ indicates whether z is negative. However, if overflow occurs, this is not true.

Question 10.7 *Provide an example in which the sign of z is not signaled correctly by $S[n-1]$.*

We would like to be able to know whether z is negative regardless of whether overflow occurs. We define the NEG signal.

Definition 10.5 *The signal* NEG *is defined as follows:*

$$\textsc{neg} \triangleq \begin{cases} 1 & \textit{if } z < 0 \\ 0 & \textit{if } z \geq 0. \end{cases}$$

A brute force method based on Theorem 10.7 for computing the NEG signal is as follows:

$$\textsc{neg} = \begin{cases} S[n-1] & \text{if no overflow} \\ 1 & \text{if } C[n] - C[n-1] = 1 \\ 0 & \text{if } C[n-1] - C[n] = 1. \end{cases} \tag{10.8}$$

Although this computation obviously signals correctly whether the sum is negative, it requires some further work if we wish to obtain a small circuit for computing NEG that is not given $C[n-1]$ as input.

Instead pursuing this direction, we compute NEG using a more elegant method.

Claim 10.9

$$\textsc{neg} = \textsc{xor}_3(A[n-1], B[n-1], C[n]).$$

Proof: The proof is based on playing the following "mental game". We extend the computation to $n+1$ bits. We then show that overflow does not occur. This means that the sum bit in position n indicates correctly the sign of the sum z. We then express this sum bit using n-bit addition signals.
Let

$$\tilde{A}[n:0] \triangleq A[n-1] \cdot A[n-1:0]$$
$$\tilde{B}[n:0] \triangleq B[n-1] \cdot B[n-1:0]$$
$$\langle \tilde{C}[n+1] \cdot \tilde{S}[n:0] \rangle \triangleq \langle \tilde{A}[n:0] \rangle + \langle \tilde{B}[n:0] \rangle + C[0].$$

Since sign-extension preserves value (see Claim 10.5), it follows that

$$z = \left[\tilde{A}[n:0]\right] + \left[\tilde{B}[n:0]\right] + C[0].$$

We claim that $z \in T_{n+1}$. This follows from

$$\begin{aligned} z &= [A[n-1:0]] + [B[n-1:0]] + C[0] \\ &\leq 2^{n-1} - 1 + 2^{n-1} - 1 + 1 \\ &\leq 2^n - 1. \end{aligned}$$

Similarly $z \geq 2^{-n}$. Hence $z \in T_{n+1}$, and therefore, by Theorem 10.7

$$\left[\tilde{S}[n:0]\right] = \left[\tilde{A}[n:0]\right] + \left[\tilde{B}[n:0]\right] + C[0].$$

We conclude that $z = \left[\tilde{S}[n:0]\right]$. It follows that $\textsc{neg} = \tilde{S}[n]$. However,

$$\begin{aligned} \tilde{S}[n] &= \textsc{xor}_3(\tilde{A}[n], \tilde{B}[n], \tilde{C}[n]) \\ &= \textsc{xor}_3(A[n-1], B[n-1], C[n]), \end{aligned}$$

and the claim follows. □

Question 10.8 *Prove that* $\textsc{neg} = \textsc{xor}(\textsc{ovf}, S[n-1])$.

10.5 A two's-complement adder

In this section we define and implement a two's complement adder.

Definition 10.6 *A* two's-complement adder *with input length n is a combinational circuit specified as follows.*

Input: $A[n-1:0], B[n-1:0] \in \{0,1\}^n$, *and* $C[0] \in \{0,1\}$.

Output: $S[n-1:0] \in \{0,1\}^n$ *and* $\textsc{neg}, \textsc{ovf} \in \{0,1\}$.

Functionality: *Define z as follows:*

$$z \triangleq [A[n-1:0]] + [B[n-1:0]] + C[0].$$

The functionality is defined as follows:

$$\begin{aligned} z \in T_n &\quad\Longrightarrow\quad [S[n-1:0]] = z \\ z \in T_n &\quad\Longleftrightarrow\quad \textsc{ovf} = 0 \\ z < 0 &\quad\Longleftrightarrow\quad \textsc{neg} = 1. \end{aligned}$$

Note that no carry-out $C[n]$ is output. We denote a two's-complement adder by s-adder(n). The implementation of an s-adder(n) is depicted in Figure 10.2 and is as follows:

1. The outputs $C[n]$ and $S[n-1:0]$ are computed by a binary adder adder(n) that is fed by $A[n-1:0], B[n-1:0]$, and $C[0]$.

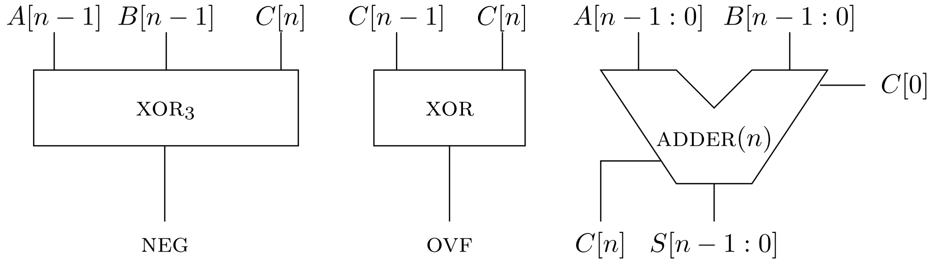


Figure 10.2: A two's complement adder S-ADDER(n)

2. The output OVF is simply XOR($C[n-1], C[n]$) if $C[n-1]$ is available. Otherwise, we apply Claim 10.8, namely, OVF $=$ XOR$_4$($A[n-1], B[n-1], S[n-1], C[n]$).

3. The output NEG is compute according to Claim 10.9. Namely, NEG $=$ XOR$_3$($A[n-1], B[n-1], C[n]$).

Note that, except for the circuitry that computes the flags OVF and NEG, a two's complement adder is identical to a binary adder. Hence, in an arithmetic logic unit (ALU), one may use the same circuit for signed addition and unsigned addition.

Question 10.9 *Prove the correctness of the implementation of* S-ADDER(n) *depicted in Figure 10.2.*

Question 10.10 *Is the design depicted in Figure 10.3 a correct* S-ADDER($2n$)*?*

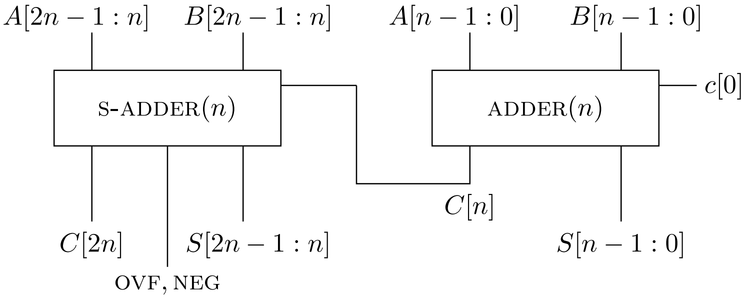


Figure 10.3: Concatenating an S-ADDER(n) with an ADDER(n).

10.6 A two's complement adder/subtracter

In this section we define and implement a two's complement adder/subtracter. A two's complement adder/subtracter is used in ALUs to implement addition and subtraction of signed numbers.

Definition 10.7 *A* two's-complement adder/subtracter *with input length n is a combinational circuit specified as follows.*

Input: $A[n-1:0], B[n-1:0] \in \{0,1\}^n$, *and* $\text{sub} \in \{0,1\}$.

Output: $S[n-1:0] \in \{0,1\}^n$ *and* $\text{NEG}, \text{OVF} \in \{0,1\}$.

Functionality: *Define z as follows:*

$$z \triangleq [A[n-1:0]] + (-1)^{\text{sub}} \cdot [B[n-1:0]] .$$

The functionality is defined as follows:

$$\begin{aligned} z \in T_n &\quad\Longrightarrow\quad [S[n-1:0]] = z \\ z \in T_n &\quad\Longleftrightarrow\quad \text{OVF} = 0 \\ z < 0 &\quad\Longleftrightarrow\quad \text{NEG} = 1. \end{aligned}$$

We denote a two's-complement adder/subtracter by ADD-SUB(n). Note that the input *sub* indicates if the operation is addition or subtraction. Note also that no carry-in bit $C[0]$ is input and no carry-out $C[n]$ is output.

An implementation of a two's-complement adder/subtracter ADD-SUB(n) is depicted in Figure 10.4. The implementation is based on a two's complement adder S-ADDER(n) and Claim 10.2.

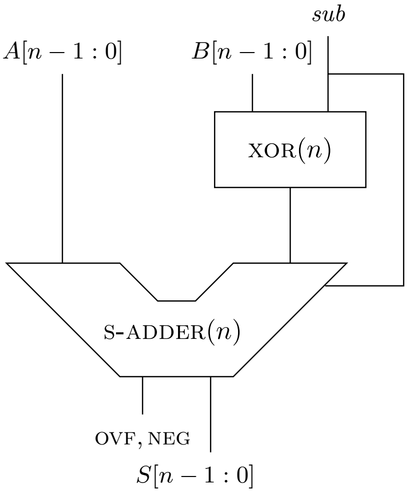


Figure 10.4: A two's-complement adder/subtracter ADD-SUB(n).

Claim 10.10 *The implementation of* ADD-SUB(n) *depicted in Figure 10.4 is correct.*

Question 10.11 *Prove Claim 10.10.*

Question 10.12 (back to the negation circuit) *Consider the negation circuit depicted in Figure 10.1.*

1. *When is the circuit correct?*

2. *Suppose we wish to add a signal that indicates whether the circuit satisfies* $\left[\vec{B}\right] = -\left[\vec{A}\right]$. *How should we compute this signal?*

Question 10.13 (wrong implementation of ADD-SUB(n)) *Find a input for which the circuit depicted in Figure 10.5 errs. Can you list all the inputs for which this circuit outputs a wrong output?*

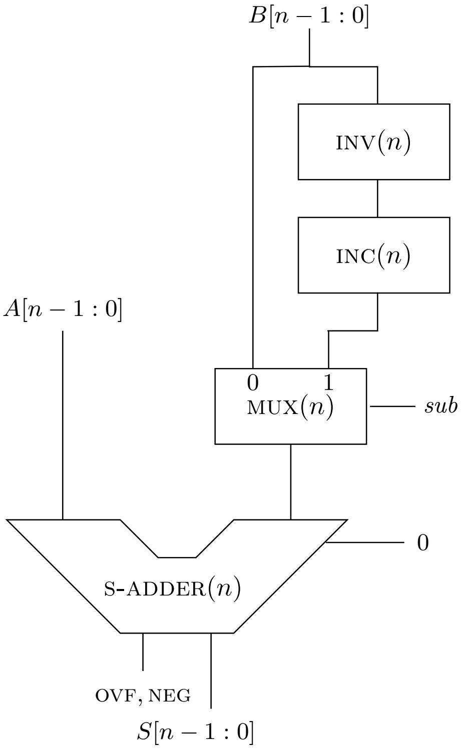


Figure 10.5: A wrong implementation of ADD-SUB(n).

10.7 Additional questions

Question 10.14 (OVF and NEG flags in high level programming) *High level programming languages such as C and Java do not enable one to see the value of the* OVF *and* NEG *signals (although these signals are computed by adders in all microprocessors).*

1. *Write a short program that deduces the values of these flags. Count how many instructions are needed to recover these lost flags.*

2. *Short segments in a low level language (Assembly) can be integrated in C programs. Do you know how to see the values of the* OVF *and* NEG *flags using a low level language?*

Question 10.15 (bi-directional cyclic shifting) *The goal in this question is to design a bi-directional barrel-shifter.*

Definition 10.8 *A bi-directional barrel-shifter* BI-BARREL-SHIFTER(n) *is a combinational circuit defined as follows:*

Input: $x[n-1:0]$*,* $dir \in \{0,1\}$*, and* $sa[k-1:0]$ *where* $k = \lceil \log_2 n \rceil$*.*

Output: $y[n-1:0]$*.*

Functionality: *If* $dir = 0$ *then* $\vec{y}$ *is a cyclic left shift of* $\vec{x}$ *by* $\langle \vec{sa} \rangle$ *positions. Formally,*

$$\forall j \in [n-1:0]: \quad y[j] = x[\text{mod}(j + \langle \vec{sa} \rangle, n)].$$

If $dir = 1$ *then* $\vec{y}$ *is a cyclic right shift of* $\vec{x}$ *by* $\langle \vec{sa} \rangle$ *positions. Formally,*

$$\forall j \in [n-1:0]: \quad y[j] = x[\text{mod}(j - \langle \vec{sa} \rangle, n)].$$

1. *Suggest a reduction of right cyclic shifting to left cyclic shifting for* $n = 2^k$*. (Hint: shift by* x *to the right is equivalent to shift by* $2^k - x$ *to the left.)*
2. *If your reduction includes an increment, suggest a method that avoids the logarithmic delay associated with incrementing.*

Question 10.16 (Comparison) *Design a combinational circuit* COMPARE(n) *defined as follows.*

Inputs: $A[n-1:0, B[n-1:0] \in \{0,1\}^n$*.*

Output: $LT, EQ, GT \in \{0,1\}$*.*

Functionality:

$$\begin{array}{ccc} [\vec{A}] > [\vec{B}] & \iff & GT = 1 \\ [\vec{A}] = [\vec{B}] & \iff & EQ = 1 \\ [\vec{A}] < [\vec{B}] & \iff & LT = 1. \end{array}$$

1. *Design a comparator based on a two's complement subtracter and a zero-tester.*
2. *Design a comparator from scratch based on a* PPC–OP(n) *circuit.*

Question 10.17 (one's complement adder/subtracter) *Design an adder/subtracter with respect to one's complement representation.*

Question 10.18 (sign-magnitude adder/subtracter) *Design an adder/subtracter with respect to sign-magnitude representation.*

10.8 Summary

In this chapter we presented circuits for adding and subtracting two's complement signed numbers. We started by describing three ways for representing negative integers: sign-magnitude, one's-complement, and two's complement. We then focused on two's complement representation.

The first task we consider is negating. We proved that negating in two's complement representation requires inverting the bits and incrementing. The claim that describes negation was insufficient to argue about the correctness of a circuit for negating a two's complement signed number. We also noticed that negating the represented value is harder in two's complement representation than in the other two representations.

In Section 10.3 we discussed a few properties of two's complement representation: (i) We showed that the values represented by the same n-bit string in binary representation and in two's complement representation are congruent module 2^n. (ii) We showed that the most-significant bit indicates whether the represented value is negative. (iii) Finally, we discussed sign-extension. Sign-extension enables us to increase the number of bits used to represent a two's complement number while preserving the represented value.

The main result of this chapter is presented in Section 10.4. We reduce the task of two's complement addition to binary addition. Theorem 10.7 also provides a rule that enables us to tell when this reduction fails. The rest of this section deals with: (i) the detection of overflow - this is the case that the sum is out of range; and (ii) determining the sign of the sum even if an overflow occurs.

In Section 10.5 we present an implementation of a circuit that adds two's complement numbers. Finally, in Section 10.6 we present an implementation of a circuit that can add and subtract two's complement numbers. Such a circuit is used in arithmetic logic units (ALUs) to implement signed addition and subtraction.