Computer Sturcture
&
Introduction to Digital Computers

Lecture Notes

by

Guy Even

Dept. of Electrical Engineering - Systems, Tel-Aviv University.
Spring 2003

Printed on May 20, 2003

Contents

Chapter 1

The digital abstraction

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. Hence, 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.

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

[1]unless Quantum Physics is used.

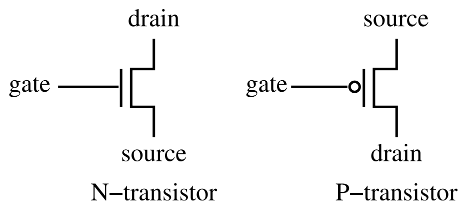


Figure 1.1: Schematic symbols of an N-transistor and P-transistor

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 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.

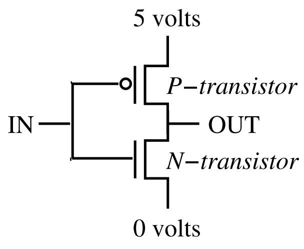


Figure 1.2: A CMOS inverter

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 these assumptions, it follows that the voltage along a wire is identical at all times. 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 the 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

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

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.

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

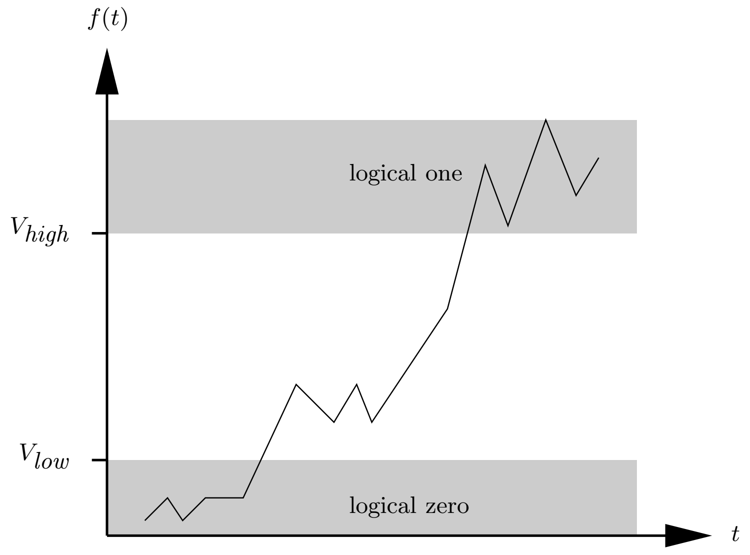


Figure 1.3: A digital interpretation of an analog signal in the zero-noise model.

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 $(-\infty, t_0]$.

Transfer functions are solved by modeling gates with partial differential equations, 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?

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 point 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.[3] 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 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] If this were not the case, then we need to distinguish between two cases: (a) Stability is not reached: this case occurs, for example, with devices called oscillators. Note that such 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..

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 $A(t)$ (resp., $B(t)$) denote the analog signal measured at point A (resp., 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 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 of a wire. 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 *uniform 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.

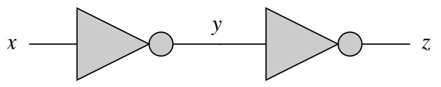


Figure 1.4: Two inverters connected in series.

1.5.1 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.

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 noise whose absolute value is less than the noise margin will not change the logical value of an output signal. 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 $V_{low,in}$ will result with an input signal $f_{in}(t) = f_{out}(t) + n(t)$ that does not exceed $V_{low,out}$.

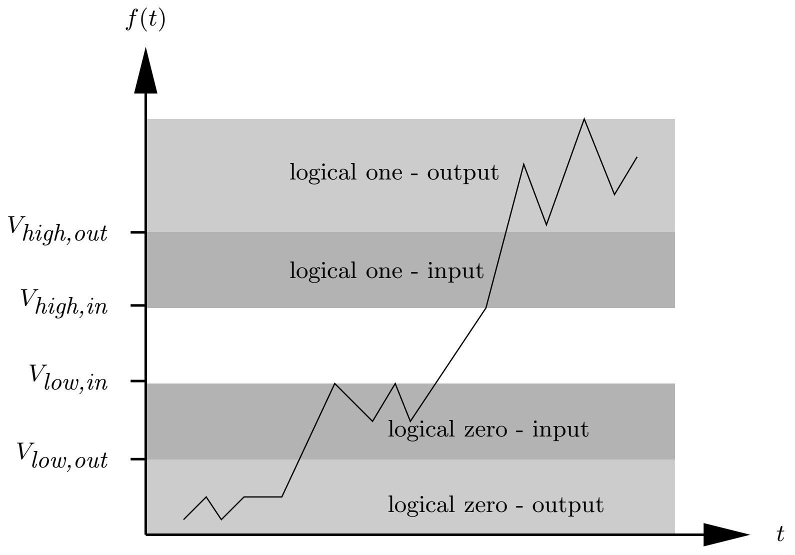


Figure 1.5: A digital interpretation of an input and output signals.

Consider an input signal $f_{in}(t)$. The digital signal $dig(f_{in}(t))$ is defined as follows.

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

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

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

Observe that sufficiently large noise margins imply that noise will not change the logical values of signals.

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

Definition 1.3 (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 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 1.3?*

Question 1.6 *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 1.3?

Hint: consider a $\delta > 0$ and set $V_{\text{low,in}} = x_0 - \delta$ and $V_{\text{high,in}} = x_0 + \delta$. What is the largest value of δ one can use?

Question 1.7 *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 1.3.*

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. $\text{dig}(f(t)) = 1$*, 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

In this chapter we define and study combinational circuits. Our goal is to prove two theorems: (A) Every Boolean function can be implemented by a combinational circuit, and (B) 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 $f : \{0,1\}^n \rightarrow \{0,1\}^k$ is called a* Boolean function.

2.2 Gates as implementations of Boolean functions

A gate is a device that has inputs and outputs. The inputs and outputs of a gate are often referred to as *terminals*, *ports*, or even *pins*. In combinational gates, the relation between the logical values of the outputs and the logical values of the inputs is specified by a Boolean function. It takes some time till the logical values of the outputs of a gate properly reflect the value of the Boolean function. We say that a gate is *consistent* if this relation holds. To simplify notation, we consider a gate G with 2 inputs, denoted by x_1, x_2, and a single output, denoted by y. We denote the digital signal at terminal x_1 by $x_1(t)$. The same notation is used for the other terminals. Consistency is defined formally as follows:

Definition 2.2 *A gate G is* consistent with a Boolean function f at time t *if the input values are digital at time t and*

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

The propagation delay is the amount of time that elapses till a gate becomes consistent. The following definition defines when a gate implements a Boolean function with propagation delay t_{pd}.

Definition 2.3 *A gate G implements a Boolean function $f : \{0,1\}^2 \rightarrow \{0,1\}$ 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

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

The following remarks should be well understood before we continue:

1. The above definition can be stated in a more compact form. Namely, a gate G implements a Boolean function $f : \{0,1\}^n \to \{0,1\}$ with propagation delay t_{pd} if stability of the inputs of G in the interval $[t_1, t_2]$ implies that the 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 gate must be stable for at least a period of t_{pd}, otherwise, the gate need not reach consistency.

3. Assume that the 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 do not assume that the output of G remains stable after t_2. The *contamination delay* of a gate is 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 will assume that the contamination delay is zero.

4. If a gate G implements a Boolean function $f : \{0,1\}^n \to \{0,1\}$ with propagation delay t_{pd}, then G also implements a Boolean function $f : \{0,1\}^n \to \{0,1\}$ with propagation delay t', for every $t' \geq t_{pd}$. It follows that it is legitimate to use upper bounds on the actual propagation delay. Pessimistic assumptions should not render a circuit incorrect.

   In fact, the actual exact propagation delay is very hard to compute. It depends on $x(t)$ (i.e how fast does the input change?). This is why we resort to upper bounds on the propagation delays.

2.3 Building blocks

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

Gates. A gate, as seen in Definition 2.3 is a device that implements a Boolean function. The *fan-in* of a gate G is the number of inputs terminals of G (i.e. the number of bits in the domain of the Boolean function that specifies the functionality of G). The basic gates that we will be using as building blocks for combinational circuits have a constant fan-in (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 2.4 *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 connected by N.

The issue of drawing nets is a bit confusing. Figure 2.1 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.

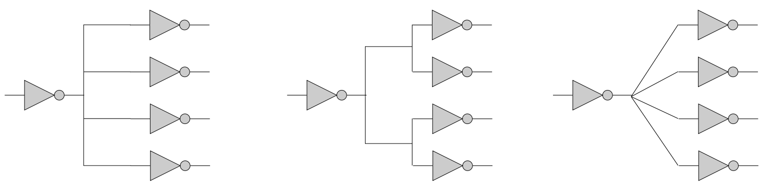


Figure 2.1: Three equivalent nets.

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 analog signals at different terminals of the net. Namely, $N(t)$ is zero (one) if the digital values of all the analog signals along the net are zero (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 2.2 depicts a 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. From an electronic point of view, in "pure" CMOS gates, output terminals are connected via resistors either to the ground or to the power. Input terminals are connected only to capacitors.

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

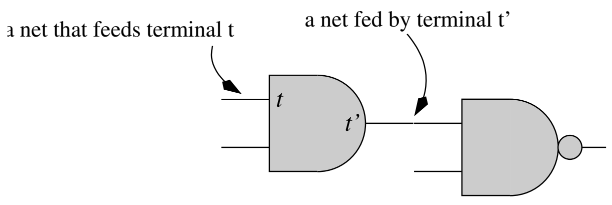


Figure 2.2: A terminal that is fed by a net and a terminal that feeds a net.

Definition 2.5 *A net N is* simple *if (a) N is fed by exactly one output terminal, and (b) 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 2.6 *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 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 of 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.

2.4 Combinational circuits

Question 2.1 *Consider the circuits depicted in Figure 2.3. Can you explain why these are not valid 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, all the nets in the circuit on the right in Figure 2.3 have endpoints that are not connected to a gate).

Definition 2.7 (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 2.4 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.

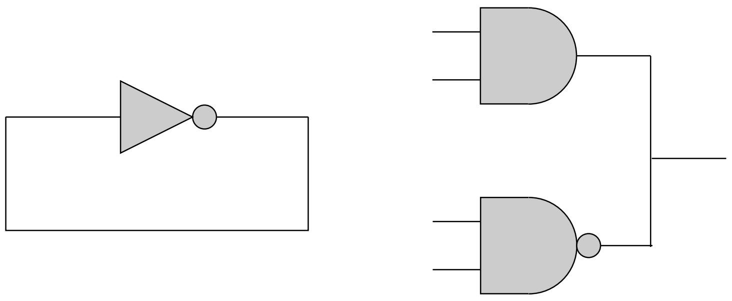


Figure 2.3: Two examples of non-combinational circuits.

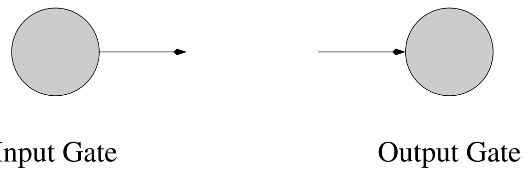


Figure 2.4: An input gate and an output gate

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.

Let $\mathcal{G}$ denote the set of gates in a circuit. Every gate $G \in \mathcal{G}$ is an *instance* of a gate 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 2.8 (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 2.8 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 2.3?*

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 f, there exists a combinational circuit that implements f. 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 in linear time. 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 2.1 (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 $f_N : \{0,1\}^k \rightarrow \{0,1\}$, and*

2. *a propagation delay $t_{pd}(N)$*

such that

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

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

We can simplify the statement of Theorem 2.1 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.

We use $\vec{x}(t)$ to denote the vector $x_1(t), \ldots, x_k(t)$.

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 $v_1, \ldots, v_k$ is the set of input gates.

Let $\mathcal{N}_i$ denote the subset of nets in $\mathcal{N}$ that are fed by gate v_i. Note that if v_i is an output gate, then $\mathcal{N}_i$ is empty. Let $e_1, e_2, \ldots, e_m$ denote an ordering of the nets in $\mathcal{N}$ such that nets in $\mathcal{N}_i$ precede nets in $\mathcal{N}_{i+1}$, for every $i < n$. In other words, we first list the nets fed by gate v_1, followed by a list of the nets fed by gate v_2, etc.

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

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

Induction Basis: We prove the induction basis for $m' = k$. Consider an $i \leq k$. Note that, for every $i \leq k$, e_i is fed by the input gate v_i. Let x_i denote the output terminal of v_i. It follows that the digital signal along e_i always equals the digital signal $x_i(t)$. Hence we define f_{e_i} to be simply the projection on the ith component, namely $f_{e_1}(\sigma_1, \ldots, \sigma_k) = \sigma_i$. The propagation delay $t_{pd}(e_i)$ is zero. The induction basis follows.

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 terminals that are fed by the nets e_j and e_k, respectively. The ordering of the nets guarantees that $j, k \leq m'$. By the induction hypothesis, the net e_j (resp., e_k) implements a Boolean function f_{e_j} (resp., f_{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 f_{v_i} with propagation delay $t_{pd}(v_i)$. It follows that the output of gate v_i equals

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

during the interval

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

We define $f_{e_{m'+1}}$ to be the Boolean function obtained by the composition of Boolean functions $f_{e_{m'+1}}(\vec{\sigma}) = f_{v_i}(f_{e_j}(\vec{\sigma}), f_{e_k}(\vec{\sigma}))$. 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 digital abstraction allows us to assume that the signal corresponding to every net in a combinational circuit is logical (provided that the time that elapses since the inputs become stable is at least the propagation delay). This justifies the convention of identifying a net with the digital value of the net.

The proof of Theorem 2.1 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
$$f_{v_j}(\vec{I}_{v_j}),$$
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.

2. The propagation delay of the gate that feeds e_i is set to
$$t_{pd}(e_i) \leftarrow t_{pd}(v_j) + \max\{t_{pd}(e')\}_{\{e' \text{ feeds } v_j\}}.$$

If the number of outputs of each gate is constant, then the total amount of time spend in the relaxation steps is linear, and hence the running time of this algorithm is linear. (Note that the input length is the number of gates plus the sum of the sizes of the nets.)

Question 2.3 *Prove that the total amount of time spent in the relaxation steps is linear 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. Can you still prove linear running time if the fan-in of the gates is not constant but the number of outputs of each gate is constant?

Can you suggest a slight weakening of this restriction which still maintains a linear running time?

2.5 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 2.9 *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 2.10 *The propagation delay of a combinational circuit $C = \langle \mathcal{G}, \mathcal{N} \rangle$ is defined by*

$$t_{pd}(C) \stackrel{\triangle}{=} \max_{N \in \mathcal{N}} t_{pd}(N).$$

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

Definition 2.11 *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 2.1.

Claim 2.2 *The propagation delay of a combinational circuit $C = \langle \mathcal{G}, \mathcal{N} \rangle$ equals*

$$t_{pd}(C) = \max_{\text{paths } p} t_{pd}(p)$$

Paths, the delay of which equals the propagation delay of the circuit, are called *critical paths.*

Question 2.4

1. *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?*
2. *In Claim 2.2 the propagation delay of a combinational circuit is defined to be 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 the following 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 figures as Motorola and Venus. Table 2.1 summarizes the normalized costs and delays in these technologies.

2.6 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,

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

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 of 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.

2.7 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 f. 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 $f(\vec{x})$ 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

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 $f : \{0,1\}^2 \to \{0,1\}$ is an associative Boolean function, then*

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

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

Question 3.2 *Show that the set of functions $f_n(x_1, \ldots, x_n)$ that are induced by associative Boolean functions $f : \{0,1\}^2 \to \{0,1\}$ is*

$$\{\textit{constant } 0, \textit{constant } 1, x_1, x_n, \text{AND}, \text{OR}, \text{XOR}, \text{NXOR}\}.$$

The implication of Question 3.2 is that there are only four non-trivial functions f_n (which?). In the rest of this section we will only consider the Boolean function OR. 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 rooted binary 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.

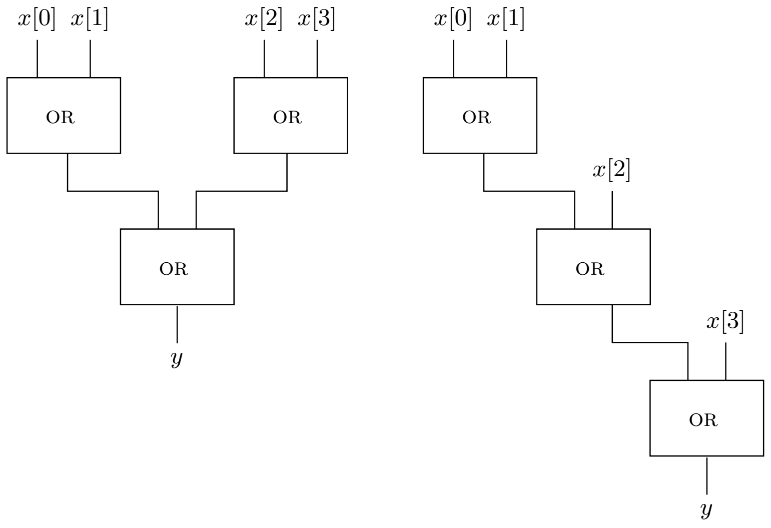


Figure 3.1: Two implementations of an OR-tree(n) with $n = 4$ inputs.

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

1. *Basis: a single* OR*-gate is an* OR*-tree(2).*
2. *Step: an* OR*(n)-tree is a circuit in which*
   (a) *the output is computed by an* OR*-gate, and*
   (b) *the inputs of this* OR*-gate are the outputs of* OR*-tree(n_1) & OR-tree(n_2), where $n = n_1 + n_2$.*

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

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

Output: y

Functionality:

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

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 and delay 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 the fact that all OR-trees have the same cost.

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

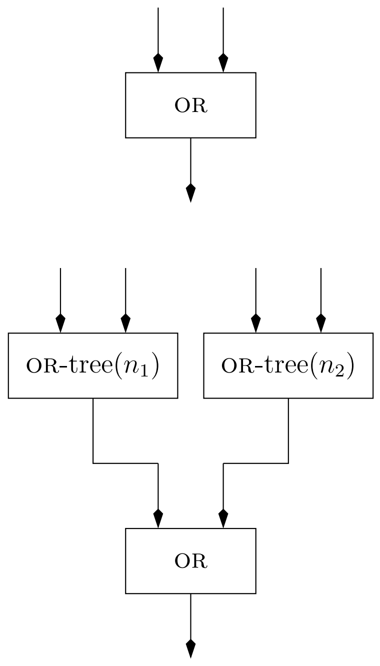


Figure 3.2: A recursive definition of an OR-tree(n).

Proof: The proof is by induction on n. The induction basis, for $n = 2$, follows because OR-tree(2) contains a single OR-gate. 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. □

The following question shows that the delay of an OR-tree(n) can be $\lceil \log_2 n \rceil \cdot t_{pd}(\text{OR})$, if a balanced tree is used.

Question 3.4 *This question deals with different ways to construct balanced trees. The goal is to achieve a depth of* $\lceil \log_2 n \rceil$.

1. *Prove that if T_n is a rooted binary tree with n leaves, then the depth of T_n is at least* $\lceil \log_2 n \rceil$.

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

3. *Prove that for every $n > 2$ there exists a pair of positive integers a, b such that (1) $a + b = n$, and (2)* $\max\{\lceil \log_2 a \rceil, \lceil \log_2 b \rceil\} \leq \lceil \log_2 n \rceil - 1$.

4. *Consider the following recursive algorithm for constructing a binary tree with $n \geq 2$ leaves.*

   (a) *The case that $n \leq 2$ is trivial (two leaves connected to a root).*

   (b) *If $n > 2$, then let a, b be* any *pair of positive integers such that (i) $n = a + b$ and (ii)* $\max\{\lceil \log_2 a \rceil, \lceil \log_2 b \rceil\} \leq \lceil \log_2 n \rceil - 1$. *(Such a pair exists by the previous item.)*

   (c) *Compute trees T_a and T_b. Connect their roots to a new root to obtain T_n.*

   Prove that the depth of T_n is at most $\lceil \log_2 n \rceil$.

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 and using the inputs to feed 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 3.5 (restricted Boolean functions) *Let $f : \{0,1\}^n \to \{0,1\}$ denote a Boolean function. Let $\sigma \in \{0,1\}$. The Boolean function $g : \{0,1\}^{n-1} \to \{0,1\}$ defined by*

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

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

Definition 3.6 (cone of a Boolean function) *A boolean function $f : \{0,1\}^n \to \{0,1\}$* depends *on its ith input if*

$$f_{\restriction x_i=0} \not\equiv f_{\restriction x_i=1}.$$

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

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

The following claim is trivial.

Claim 3.3 *The Boolean function* OR_n *depends on all its inputs, namely*

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

Example 3.1 *Consider the following Boolean function:*

$$f(\vec{x}) = \begin{cases} 0 & \text{if } \sum_i x[i] < 3 \\ 1 & \text{otherwise.} \end{cases}$$

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!

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

Claim 3.4 $\text{cone}(f) = \emptyset \Longleftrightarrow f$ *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 3.5 *Let $C = \langle \mathcal{G}, \mathcal{N} \rangle$ denote a combinational circuit that implements a Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$. Let $g_i \in \mathcal{G}$ denote the input gate that feeds the ith 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 2.1) 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 3.6 (Linear Cost Lower Bound Theorem) *Let C denote a combinational circuit that implements a Boolean function f. Then*

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

Before we prove Theorem 3.6 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 3.7 *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 3.3 and Theorem 3.6. □

We prove Theorem 3.6 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 3.3 for an example.

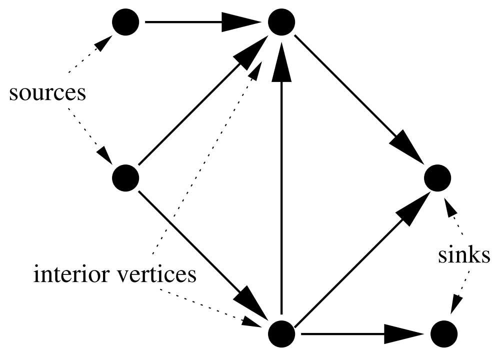


Figure 3.3: A DAG with two sources, two interior vertices, and two sinks.

Proof of Theorem 3.6: If the underlying graph of $DG(C) = (V, E)$ is a rooted binary tree, then in this tree we we know that

$$|\text{interior vertices}| \geq |\text{leaves}| - 1.$$

Recall that leaves are input gates and that every interior vertex is a non-trivial gate. Hence, the case of a tree follows.

If the underlying graph of $DG(C)$ is not a tree, then we construct a DAG $T = (V', E')$ that is a subgraph of $DG(C)$ such that (i) the sources in V' are all the input gates that feed inputs x_i such that $i \in \mathit{cone}(f)$, (ii) the output gate is the sink, and (iii) the underlying graph of T is a rooted binary tree.

The DAG T is constructed as follows. Pick a source $v \in V$ that feeds an input x_i such that $i \in \mathit{cone}(f)$. By Claim 3.5, 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 $i \in \mathit{cone}(f)$. 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 $|\text{interior vertices}| \geq |\text{leaves}| - 1$. The interior vertices of T are also interior vertices of $DG(C)$, and the theorem follows. □

Question 3.5 *State and prove a generalization of Theorem 3.6 for the case that the fan-in of every gate is bounded by a constant c.*

We now turn to proving a lower bound on the delay of a combinational circuit that implements OR_n. Again, we will use a general technique. Again, we will 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.

Theorem 3.8 (Logarithmic Delay Lower Bound Theorem) *Let $C = \langle \mathcal{G}, \mathcal{N} \rangle$ 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 $\mathcal{G}$ is at most c, then the delay of C is at least* $\log_c |\text{cone}(f)|$.

Before we prove Theorem 3.8, we show that the theorem implies a lower bound on the delay of combinational circuits that implement OR_n.

Corollary 3.9 *Let C_n denote a combinational circuit that implements OR_n. Let c denote the maximum fan-in of a gate in C_n. Then*

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

Proof: The corollary follows directly from Claim 3.3 and Theorem 3.8. □

Proof of Theorem 3.8: The proof deals only with the graph $DG(C)$ and shows that there must be a path with at least $\log_c |cone(f)|$ 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. We attach to every vertex v in the DAG a subset of sources $cone(v)$ that is the set of sources from which v is reachable. Let $d(v)$ denote the maximum number of interior vertices along a path from a source in $cone(v)$ to v including v. We now prove that, for every vertex v,

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

Claim 3.5 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

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

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 $d(v) = i + 1$. We first assume that v is an interior vertex. There are at most c edges that enter v. Denote the vertices that precede v by $v_1, \ldots, v_c'$, where $c' \leq c$. Namely, the edges that enter v are $v_1 \rightarrow v, \ldots, v_{c'} \rightarrow v$. Since v is an interior vertex, it follows by definition that

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

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

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

Hence

$$\begin{aligned}|cone(v)| &\leq \sum_{i=1}^{c'} |cone(v_i)| \\ &\leq c' \cdot \max\{|cone(v_i)|\}_{i=1}^{c'}\}. \end{aligned} \tag{3.4}$$

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

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

But,

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

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 $d(v) \geq \log_c |cone(v)|$, as required, and the theorem follows. □

Question 3.6 *The proof of the Theorem 3.8 dealt with the longest path from an input vertex to an output vertex. In the proof, we strengthened this statement by considering every vertex instead of only the sink. In this question we further strengthen the conditions at the expense of a slightly weaker lower bound. We consider the longest shortest path from a set of vertices U to a given vertex r (i.e. $\max_{u\in U} \text{dist}(u,r)$).*

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 $\text{dist}(u,r) \geq \Omega(\log_c |U|)$, where c denotes the maximum degree of 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 started by considering associative Boolean functions. We showed how associative dyadic functions are extended 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 term $cone(f)$ for 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 2, 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

In this chapter we present two important combinational modules called decoders and encoders. These modules are often used as part of bigger circuits.

4.1 Notation

In VLSI-CAD tools one often uses indexes to represent multiple nets. For example, instead of naming nets $a, b, c, \ldots$, one uses the the names $a[1], a[2], a[3]$. Such indexing is very useful if the nets are connected to the same modules (i.e. "boxes"). A collection of indexed nets is often called a *bus*. Indexing of buses in such tools is often a cause of great confusion. For example, assume that one side of a bus is called $a[0:3]$ and the other side is called $b[3:0]$. Does that mean that $a[0] = b[0]$ or does it mean that $a[0] = b[3]$? Our convention will be that "reversing" does not take place unless stated explicitly. However, will often write $a[0:3] = b[4:7]$, meaning that $a[0] = b[4], a[1] = b[5]$, etc. Such a re-assignment of indexes often called *hardwired shifting.*

To summarize, 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 $b[i] \leftarrow a[i], \ldots, b[j] \leftarrow a[j]$. Assignments in which the index ranges are shifted, such as: $b[i+5:j+5] \leftarrow a[i:j]$, mean $b[i+5] \leftarrow a[i], \ldots, b[j+5] \leftarrow a[j]$. This attempt to make bus assignments unambiguous is bound to fail, so we will still need to resort to common sense (or just ask).

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 multiple wires. The number of wires is written next to the short segment.

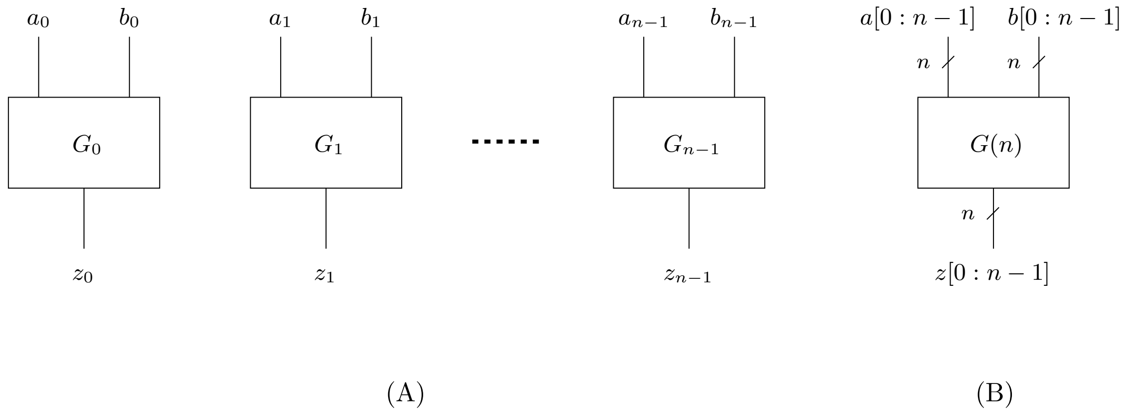


Figure 4.1: Vector notation: multiple instances of the same gate.

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.

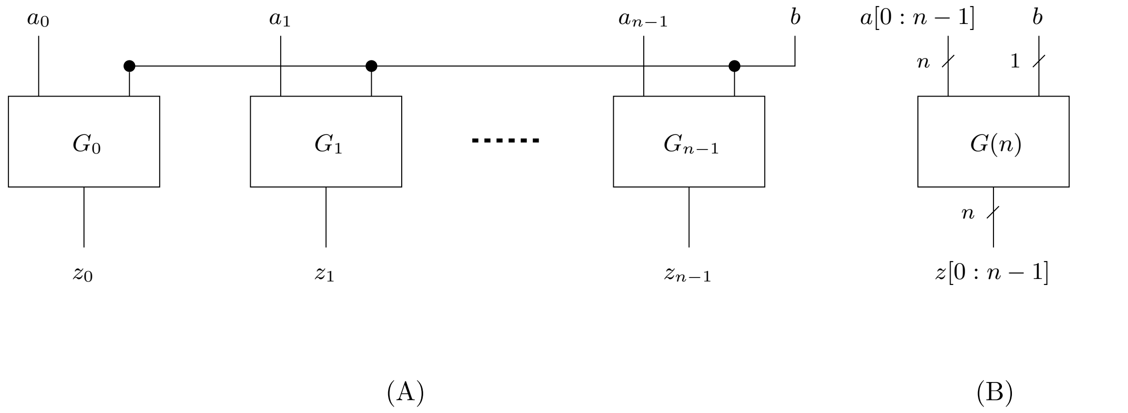


Figure 4.2: Vector notation: b feeds all the gates.

Note that the fanout of the net that carries the signal b in Figure 4.2 is linear. In practice, a large fanout increases the capacity of a net and causes an increase in the delay of the circuit. We usually ignore this phenomena 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 $a \cdot b = 0110$.*
- *If $a = 1$ and $i = 5$, then $a^i = 11111$.*
- *If $a = 01$ and $i = 3$, then $a^i = 010101$.*

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.

The following definition defines the number that is represented by a binary string.

Definition 4.1 *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 4.2 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 $0 \le k \le n-1$. Let q and r denote the quotient and remainder, respectively, when dividing i by 2^k. Namely, $i = 2^k \cdot q + r$, where $0 \le r < 2^k$.*

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

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

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

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 $\text{DECODER}(n)$.

Example 4.2 *Consider a decoder* $\text{DECODER}(3)$. *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 a circuit for computing $y[i]$. Let $b[n-1:0]$ denote the binary representation of i (i.e. $b[n-1:0] = bin(i)$).
Define $z_0[n-1:0]$ and $z_1[n-1:0]$ as follows:

$$z_1[n-1:0] \triangleq x[n-1:0] \qquad\qquad z_0[n-1:0] \triangleq \text{INV}(x[n-1:0]).$$

Note that $\vec{z}_0$ and $\vec{z}_1$ are simply vectors that correspond to inverting and not-inverting $\vec{x}$, respectively.
The following claim implies a simple circuit for computing $y[i]$.

Claim 4.2

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

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 $x[i] = 1$ if $b[i] = 1$ and $\text{INV}(x[i]) = 1$ if $b[i] = 0$. □

The brute force decoder circuit consists of (i) n inverters used to compute $\vec{z}_0$, and (ii) an $\text{AND}(n)$-tree for every output $y[i]$. The delay of the brute force design is $t_{pd}(\text{INV}) + \lfloor \log_2 t_{pd}(\text{AND}) \rfloor$. 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 DECODER(n) using recursion on n. We start with the trivial task of designing a DECODER(n) with $n = 1$. We then proceed by designing a DECODER(n) based on "smaller" decoders.

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

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 computed by the designer.

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).

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.*

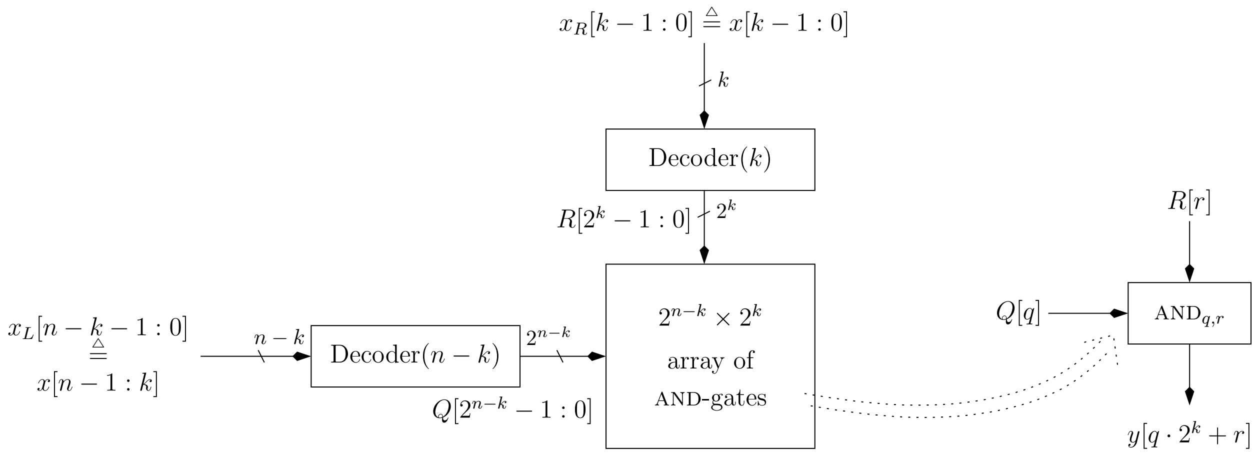


Figure 4.3: A recursive implementation of DECODER(n).

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

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

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 $r \in [2^k - 1 : 0]$.

By Claim 4.1,

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

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 $\langle x_L[n-k-1:0] \rangle = q$. This implies that

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

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 $c(n) = O(2^n)$. The obvious choice is to choose $k = n/2$. If n is odd, then we choose $k = \lfloor n/2 \rfloor$. Assume that n is a power of n, 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 DECODER(n) satisfies the following recurrence equation:

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

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

Question 4.1 *Prove that* DECODER(n) *is asymptotically optimal with respect to cost and delay.*

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 4.4 *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* $\text{wt}(\vec{a})$*.Formally,*

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

We define the encoder partial function as follows.

Definition 4.5 *The function* $\text{ENCODER}_n : \{\vec{y} \in \{0,1\}^{2^n} : \text{wt}(\vec{y}) = 1\} \rightarrow \{0,1\}^n$ *is defined as follows:* $\langle \text{ENCODER}_n(\vec{y}) \rangle$ *equals the index of the bit of* $\vec{y}$ *that equals one. Formally,*

$$\text{wt}(y) = 1 \Longrightarrow y[\langle \text{ENCODER}_n(\vec{y}) \rangle] = 1.$$

Definition 4.6 *An* encoder *with input length* 2^n *and output length* n *is a combinational circuit that implements the Boolean function* 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:

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

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 $\text{ENCODER}(n)$.

Example 4.3 *Consider an encoder* $\text{ENCODER}(3)$. *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 $\text{ENCODER}'(n)$. We then show how to modify $\text{ENCODER}'(n)$ 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 encoders $\text{ENCODER}'(n/2)$ (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.

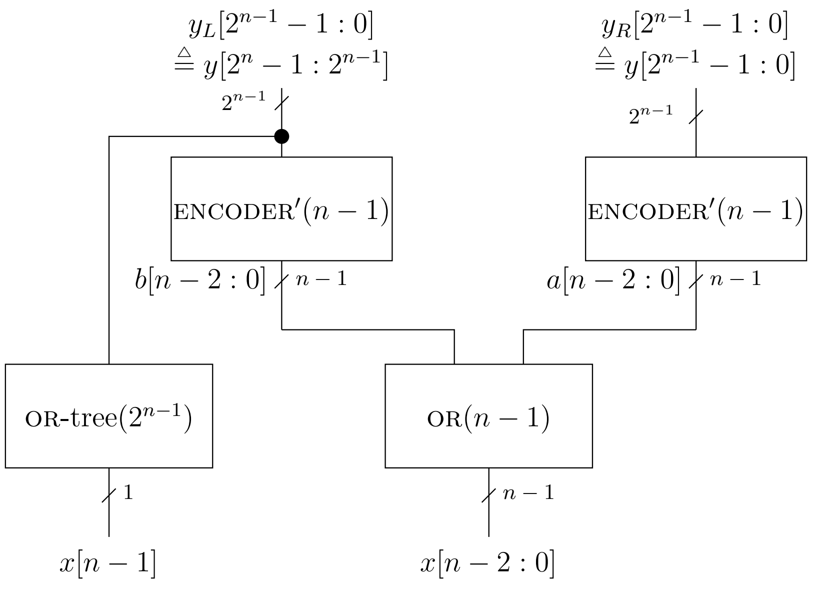


Figure 4.4: A recursive implementation of ENCODER$'(n)$.

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.

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 $\text{ENCODER}'(n)$ design is that it is too costly. We summarize the cost of $\text{ENCODER}'(n)$ in the following question.

Question 4.2 *This question deals with the cost and delay of* $\text{ENCODER}'(n)$.

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* $\text{ENCODER}'(n)$ *design have over the trivial design in which every output bit is computed by a separate circuit?*

Question 4.3 *An* $\text{ENCODER}'(n)$ *contains an* OR*-tree*(2^n) *and an* $\text{ENCODER}^*(2^{n-1})$ *that are fed by the same input. 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* $\text{ENCODER}'(n)$ *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 4.2 suggests that apart from fan-out considerations, the $\text{ENCODER}'(n)$ design is a costly 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). Assume that $y_R[i] = 1$. 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 $\text{ENCODER}^*(n)$ obtained from $\text{ENCODER}'(n)$ after commuting the OR and the $\text{ENCODER}(n-1)$ operations. We do not need to prove the correctness of the $\text{ENCODER}^*(n)$ circuit "from the beginning". Instead we rely on the correctness of $\text{ENCODER}'(n)$ and on Claim 4.5 that shows that $\text{ENCODER}'(n)$ and $\text{ENCODER}^*(n)$ are functionally equivalent.

Question 4.4 *Provide a direct correctness proof for the* $\text{ENCODER}^*(n)$ *design (i.e. do not rely on the correctness of* $\text{ENCODER}'(n)$*).*

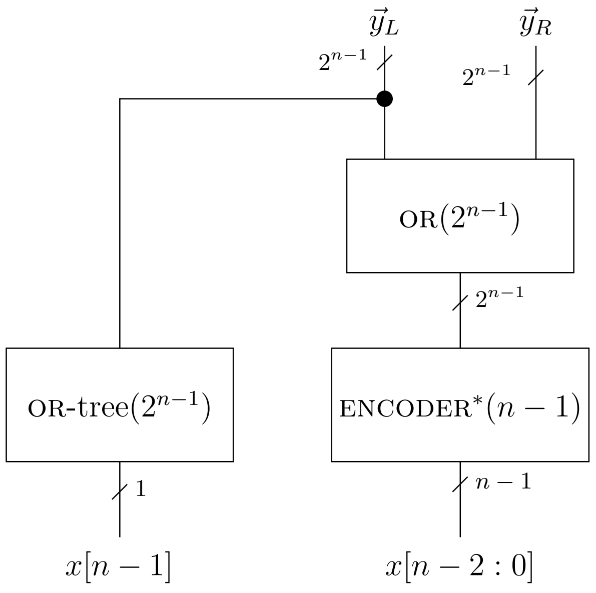


Figure 4.5: A recursive implementation of ENCODER$^*(n)$.

The following questions are based on the following definitions:

Definition 4.7 *A binary string $x'[n-1:0]$* dominates *the binary string $x''[n-1:0]$ if*

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

Definition 4.8 *A Boolean function f is* monotone *if x' dominates x'' implies that $f(x')$ dominates $f(x'')$.*

Question 4.5 *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 4.6 *The designs* ENCODER$'(n)$ *and* ENCODER$^*(n)$ *lack inverters, and hence are monotone circuits. However, the Boolean function corresponding to an encoder is not monotone. Can you resolve this contradiction?*

4.4.2 Cost and delay analysis

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

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

We expand this recurrence to obtain:

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

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

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

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

$$d(\text{ENCODER}^*(n)) = \Theta(n).$$

The following question deals with lower bounds for an encoder.

Question 4.7 *Prove that the cost and delay of* ENCODER$^*(n)$ *are asymptotically optimal.*

4.4.3 Yet another encoder

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

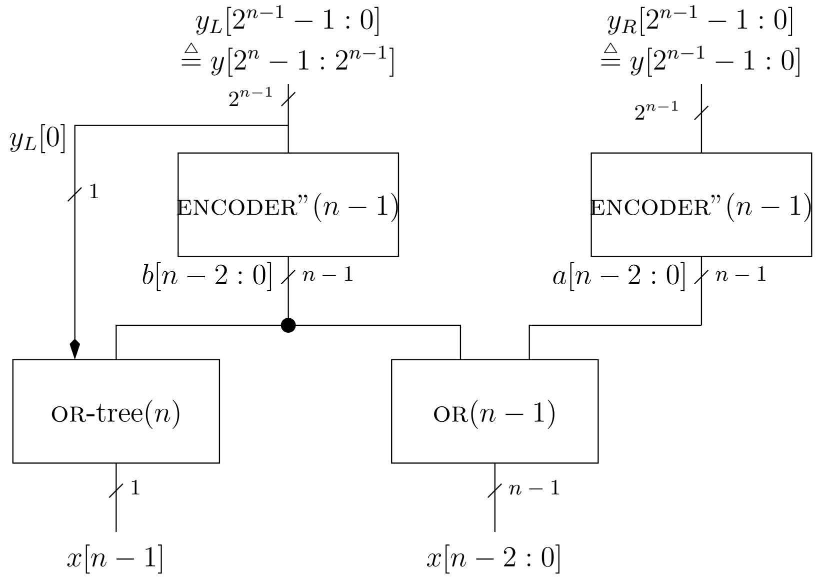


Figure 4.6: A recursive implementation of ENCODER$''(n)$.

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

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

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 a smaller, asymptotically optimal, cost.

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 use these restrictions to 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

Combinational modules

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* 1.

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.

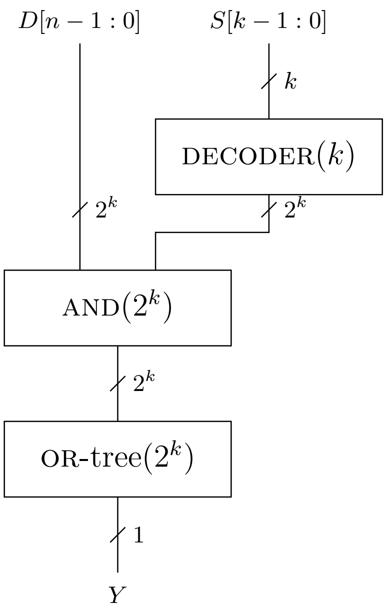


Figure 5.1: An (n:1)-MUX based on a decoder ($n = 2^k$).

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.*

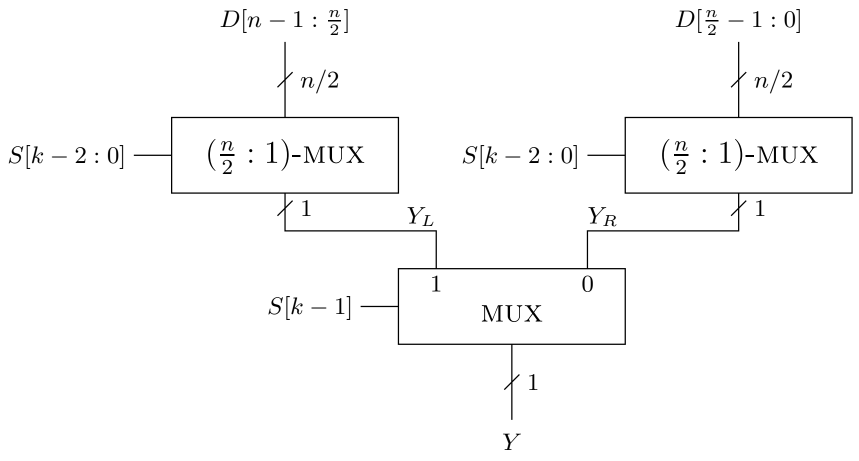


Figure 5.2: A recursive implementation of (n:1)-MUX ($n = 2^k$).

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 muxes. 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. 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.*

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.

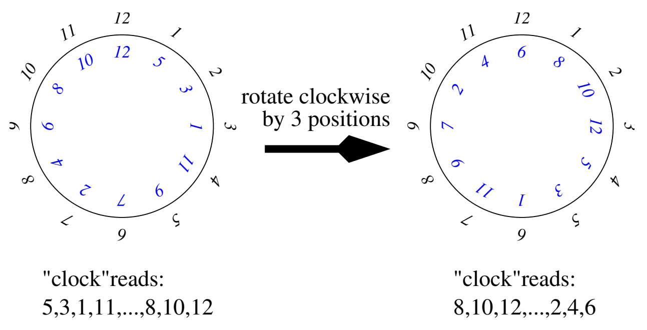


Figure 5.3: An example of a cyclic shift. The clock "reads" the numbers stored in each clock notches in clockwise order starting from the one o'clock notch.

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

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

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

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 a barrel shifter into smaller sub-tasks of shifting by powers of two. We define this sub-task formally as follows.
A 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* 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 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 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 CLS$(4, 1)$. It is

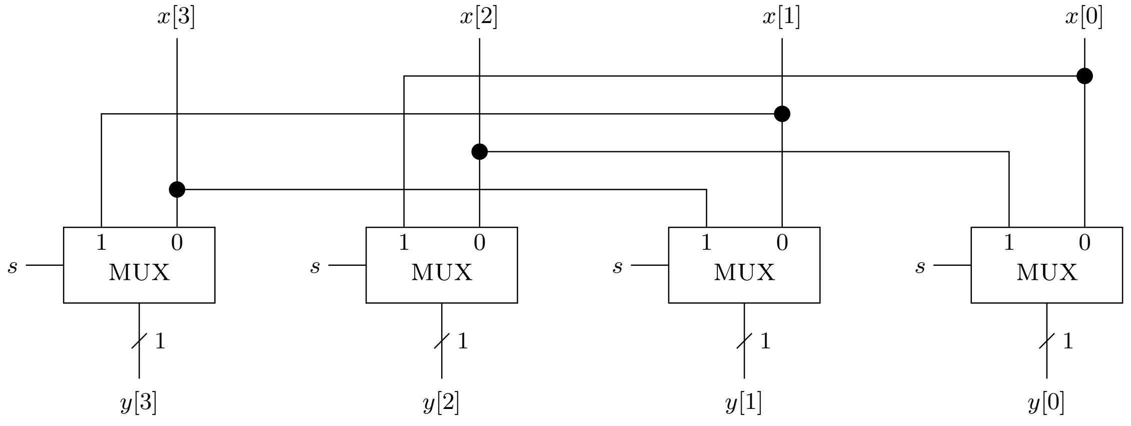


Figure 5.4: A row of multiplexers implement a $\text{CLS}(4,1)$.

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.

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

$$\begin{aligned} y_0[n-1:0] &\leftarrow \text{CLS}_{n,0}(x[n-1,0], sa[0]) \\ y_{i+1}[n-1:0] &\leftarrow \text{CLS}_{n,i+1}(y_i[n-1,0], sa[i+1]) \end{aligned}$$

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 $i=1$, holds because of the definition of $\text{CLS}(2,0)$.

The induction step is proved as follows.

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

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 CLS(n, i) we are ready to implement a BARREL-SHIFTER(n). Figure 5.5 depicts an implementation of a BARREL-SHIFTER(n). The implementation is based on k levels of CLS(n, i), for $i \in [k-1:0]$, where the ith level is controlled by $sa[i]$.

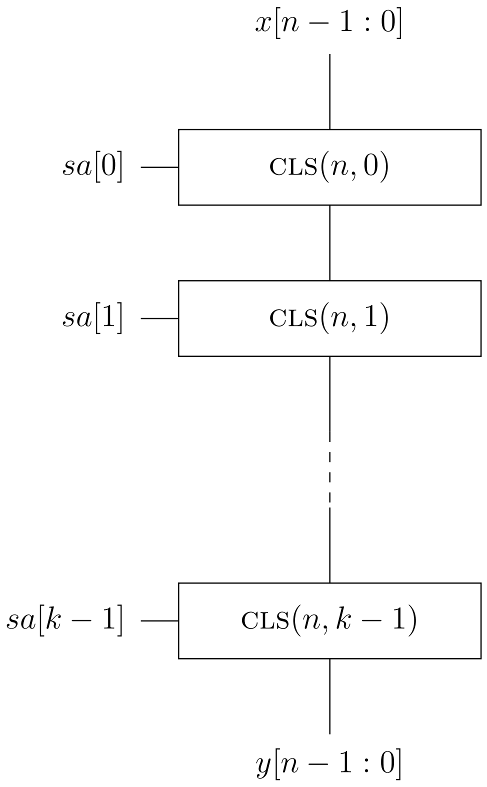


Figure 5.5: A BARREL-SHIFTER(n) built of k levels of CLS(n, i) $(n = 2^k)$.

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 Priority Encoders

Consider a binary string $x[0 : n-1]$. If $\vec{x} \neq 0^n$, then the *leading one* in the string $x[0 : n-1]$ is the non-zero bit $x[i]$ with the smallest index. A priority encoder is a combinational circuit that computes the index of the leading one. We consider two types of priority encoders: A unary priority encoder outputs the index of the leading one in unary representation. A binary priority encoder outputs the index of the leading one in binary representation.

Example 5.3 *Consider the string* $x[0 : 6] = 0110100$*. The leading one is the bit* $x[1]$*. Note that indexes are in ascending order and that* $x[0]$ *is the leftmost bit.*

Note that in the context of the representation of integers using binary representation, the binary string is written with the most significant bit on the left (i.e. $x_{n-1}x_{n-2}\cdots x_0$). This can cause some confusion, because then the term leading one refers to the non-zero bit with the highest index. If one is interested in computing the highest index of a non-zero bit, one could reverse the indexes and then find the index of the leading one according to our definition. This reversing is a purely notational issue, and requires no delay or cost. The reason for using the convention of the leading one being the non-zero bit $x[i]$ with the smallest index (rather than the highest index) is that it makes handling indexes slightly easier.

Definition 5.5 *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*.*

Remark 5.1 *A binary string such as* 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.*

We denote unary priority encoder by $\textsc{u-penc}(n)$ and define it as follows.

Definition 5.6 *A unary priority encoder* $\textsc{u-penc}(n)$ *is a combinational circuit specified as follows.*

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

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

Functionality:

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

Note that if $\vec{x} \neq 0^n$, then $y[0 : n-1] = 0^j \cdot 1^{n-j}$, where $j = \min\{i \mid x[i] = 1\}$. Hence $\textsc{inv}(\vec{y})$ is a unary representation of the position of the leading one of the string $\vec{x} \cdot 1$.
We denote binary priority encoder by $\textsc{b-penc}(n)$ and define it as follows.

Definition 5.7 *A binary priority encoder* $\textsc{b-penc}(n)$ *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:

$$\langle \vec{y} \rangle = \begin{cases} \min\{i \mid x[i] = 1\} & \text{if } \vec{x} \neq 0^n \\ n & \text{if } \vec{x} = 0^n. \end{cases}$$

Example 5.4 *Given input* $x[0 : 5] = 000101$, *a* U-PENC(6) *outputs* $y[0 : 5] = 000111$, *and* B-PENC(6) *outputs* $y[2 : 0] = 011$.

5.3.1 Implementation of U-PENC(n)

As in the case of decoders, we can design a brute force unary priority encoder by 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. Instead, we will present here a design based on divide-and-conquer.

The method of divide-and-conquer is applicable for designing a U-PENC(n). We apply divide-and-conquer in the following recursive design. If $n = 1$, then U-PENC(1) is the trivial design in which $y[0] \leftarrow x[0]$. A recursive design of U-PENC(n) for $n > 1$ that is a power of 2 is depicted in Figure 5.6. Proving the correctness of the proposed U-PENC(n) design is a simple exercise in associativity of OR$_n$, so we leave it as a question.

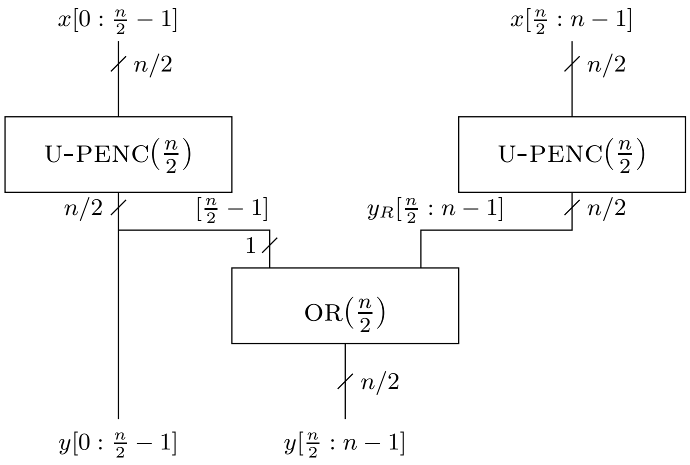


Figure 5.6: A recursive implementation of U-PENC(n).

Question 5.5 *This question deals with the design of the priority encoder* U-PENC(n) *depicted in Figure 5.6.*

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 U-PENC(n) depicted in Figure 5.6 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 optimality of the U-PENC(n) design depicted in Figure 5.6.

Question 5.6 *Prove a lower bound on the cost of a unary priority encoder.*

In the chapter on parallel prefix computation we will present a cheaper implementation of U-PENC(n).

5.3.2 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 U-PENC

Consider an input $x[0 : n-1]$ to a priority encoder (unary or binary). 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) satisfies $\langle \vec{y} \rangle = j$. Our design is based on the observation that the output $u[0 : n-1]$ of a U-PENC(n) satisfies $\vec{u} = 0^j \cdot 1^{n-j}$. In Figure 5.7 we depict a reduction from the task of computing $\vec{y}$ to the task of computing $\vec{u}$.

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

1. The input $\vec{x}$ is fed to a U-PENC(n) which 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:

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

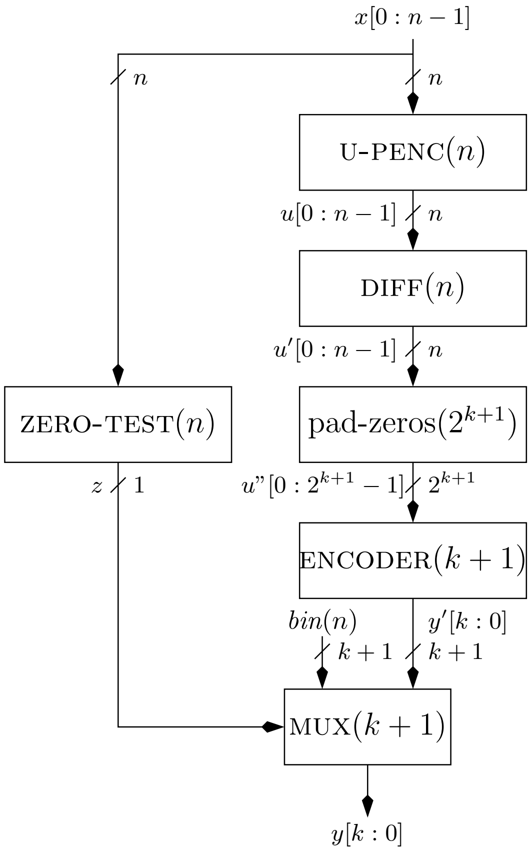


Figure 5.7: A binary priority encoder based on a unary priority encoder.

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 we need to pad $\vec{u'}$ by zeros. The string $u''[0 : 2^{k+1} - 1]$ is obtained from $u'[0 : n - 1]$ by padding zeros as follows:
$$u''[0 : n-1] \leftarrow u'[0 : n-1] \qquad\qquad u''[n : 2^{k+1} - 1] \leftarrow 0^{2^k - n}.$$
(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. If $\vec{x} \neq 0^n$, then the vector $\vec{y'}$ equals the final output. However, we need to deal also with 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 $y[k - 1 : 0]$ should satisfy $\langle \vec{y} \rangle = n$. This selection is performed by the multiplexer that selects between $bin(n)$ and $\vec{y'}$ according to the value of z.

Cost and delay analysis. The cost of the binary priority encoder depicted in Figure 5.7 satisfies:
$$\begin{aligned} c(n) &= c(\textsc{u-penc}(n)) + c(\textsc{diff}(n)) + c(\textsc{encoder}(k)) + c(\textsc{mux}(k)) + c(\textsc{zero-test}(n)) \\ &= c(\textsc{u-penc}(n)) + \Theta(n). \end{aligned}$$
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 U-PENC(n) then we would have a linear cost B-PENC(n).

The delay of the binary priority encoder depicted in Figure 5.7 satisfies:
$$\begin{aligned} d(n) &= \max\{d(\textsc{u-penc}(n)) + d(\textsc{diff}(n)) + d(\textsc{encoder}(k)), d(\textsc{zero-test}(n))\} + d(\textsc{mux}) \\ &= d(\textsc{u-penc}(n)) + \Theta(\log n). \end{aligned}$$
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 5.8 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 $y_L[k - 1] = y_R[k - 1] = 1$, $y[k-1] = \textsc{and}(y_L[k-1], \textsc{inv}(y_R[k-1]))$, 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.

Claim 5.2 *The design depicted in Figure 5.8 is a binary priority encoder.*

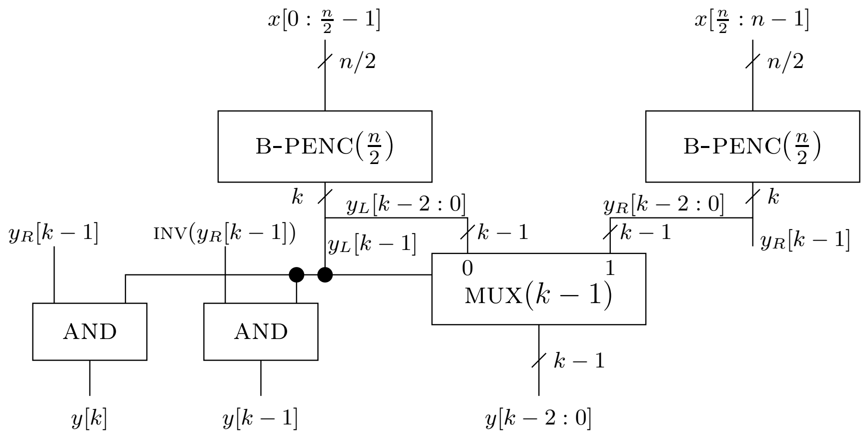


Figure 5.8: A recursive implementation of a binary priority encoder.

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 $y[k-2:0] = 0^{k-2}$, 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 5.8 satisfies (recall that $n = 2^k$):

$$c(\text{B-PENC}(n)) = \begin{cases} c(\text{NOR}) & \text{if n=2} \\ 2 \cdot c(\text{B-PENC}(n/2)) + 2 \cdot c(\text{AND}) + (k-1) \cdot c(\text{MUX}) & \text{otherwise.} \end{cases}$$

This recurrence is identical to the recurrence in Equation 4.1 if one substitutes $k = \log n$ for n. Hence $c(\text{B-PENC}(n)) = \Theta(n)$, which implies the asymptotic optimality of the design.

The delay of the binary priority encoder depicted in Figure 5.8 satisfies:

$$d(\text{B-PENC}(n)) = \begin{cases} t_{pd}(\text{NOR}) & \text{if n=2} \\ d(\text{B-PENC}(n/2)) + \max\{d(\text{MUX}) + d(\text{AND})\} & \text{otherwise.} \end{cases}$$

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

5.4 Half-Decoders

In this section 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 are used for implementing logical right shift.

Definition 5.8 *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 5.5

- *Consider a half-decoder* H-DEC(3)*. On input $x = 101$, the output $y[0:7]$ equals* 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 5.2 *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 slow.

Question 5.7 *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 optimal with respect to cost or delay?*

5.4.1 Preliminaries

In this section we present a few claims that are used in the design of optimal half-decoders. The following claim follows trivially from the definition of a half-decoder.

Claim 5.3

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

Assume that the binary string $z[0:n-1]$ represents a number in unary representation. 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 5.4 *For $i \in [0 : n-1]$:*

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

Claim 5.4 gives a simple recipe for a "comparison box". A comparison box, denoted by $\text{COMP}(\vec{z}, i)$, is a combinational circuit that compares $wt(\vec{z})$ and i and 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 $z[i]$).

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 5.5

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

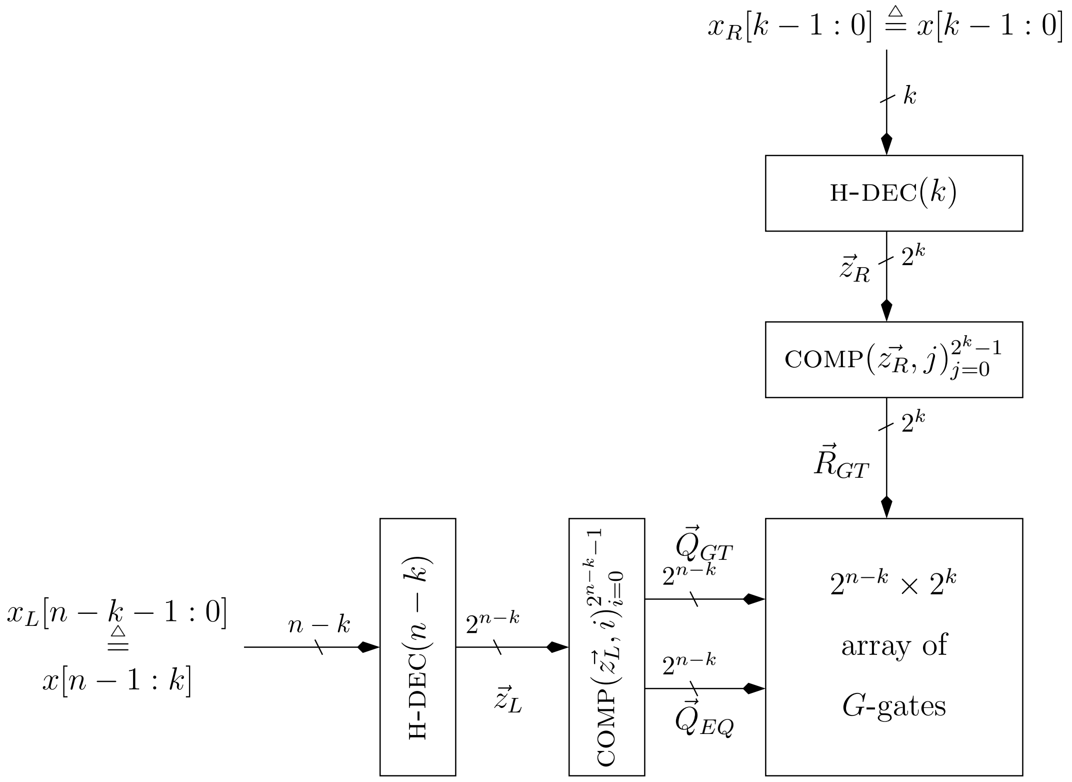
The interpretation of the above claim in terms of "blocks" and "offsets" is the following. The number $\langle \vec{x} \rangle$ is a number in the range $[2^n - 1 : 0]$. The index of the block this number belongs to is $\langle \vec{x}_L \rangle$. The offset of this number within its block is $\langle \vec{x}_R \rangle$. 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.

5.4.2 Implementation

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

A $\text{H-DEC}(n)$, for $n = 1$ is the trivial circuit $y[0] \leftarrow x[0]$. We now proceed with the recursion step. Figure 5.9 depicts a recursive implementation of a $\text{H-DEC}(n)$. The parameter k equals $k = \lceil \frac{n}{2} \rceil$ (in fact, to minimize delay one needs to be a bit more precise). 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 $y[q \cdot 2^k + r]$ which is defined by

$$y[q \cdot 2^k + r] \triangleq \text{OR}(Q_{GT}[q], \text{AND}(Q_{EQ}[q], R_{GT}[r])). \tag{5.1}$$



$x_R[k-1:0] \triangleq x[k-1:0]$
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 5.9: 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.

Example 5.6 *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 5.3, $y[6] = 1$ iff $\langle x[3:0] \rangle > 6$. By*

Claim 5.5, $\langle \vec{x} \rangle > 6$ 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 $y[6] = R_{GT}[2]$.

5.4.3 Correctness

Claim 5.6 *The design depicted in Figure 5.9 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 5.3 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 5.5,

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

The induction hypothesis implies that:

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

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 $\textsc{and}(\textsc{inv}(z_L[q], z_L[q-1])$, 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. 5.1, $y[i] = 1$ iff $\textsc{or}(Q_{GT}[q], \textsc{and}(Q_{EQ}[q], R_{GT}[r]))$. Hence $y[i]$ is correct, and the claim follows. □

5.4.4 Cost and delay analysis

The cost of $\textsc{h-dec}(n)$ satisfies the following recurrence equation:

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

The cost of computing the EQ signals is $c(\textsc{inv}) + c(\textsc{and})$. The cost of a G-gate is $c(\textsc{and}) + c(\textsc{or})$. It follows that

$$c(\textsc{h-dec}(n)) = c(\textsc{h-dec}(k)) + c(\textsc{h-dec}(n-k)) + \Theta(2^n)$$

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

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

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

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

$$\begin{aligned} d(\textsc{h-dec}(n)) &= d(\textsc{h-dec}(n/2)) + \Theta(1) \\ &= \Theta(\log n). \end{aligned}$$

It follows that the delay of $\textsc{h-dec}(n)$ is asymptotically optimal since all the inputs belong to the cone of a half-decoder.
The following question deals with the optimal cost of a half-decoder design.

Question 5.8 *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 which 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?

5.5 Logical Shifters

To be completed.

5.6 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 then defined cyclic shifting, and presented an implementation of a barrel-shifter.

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 presented a divide-and-conquer design for a unary priority encoder. The delay of this design is optimal, but its cost is not. We will present an optimal unary priority encoder as soon as we learn about parallel prefix computation.

We presented two designs for a binary priority encoder. The first design is based on a unary priority encoder. The overhead in cost is linear and the overhead in delay is logarithmic. Hence, this design leads to an optimal binary priority encoder, provided that an optimal unary priority encoder is used. The second design is a divide-and-conquer design with linear cost and logarithmic delay. This design is optimal.

The last section in 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 6

Addition

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.

6.1 Definition of a binary adder

Definition 6.1 *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:

$$\langle \vec{S} \rangle + 2^n \cdot C[n] = \langle \vec{A} \rangle + \langle \vec{B} \rangle + C[0] \tag{6.1}$$

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, and the output $C[n]$ is often called the *carry-out bit.*

Question 6.1 *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 6.1.*

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

6.2 Ripple Carry Adder

Ripple Carry Adders are adders that are built of a chain 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.

Definition 6.2 (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 of a chain of Full-Adders. An RCA(n) is depicted in Figure 6.1. 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.

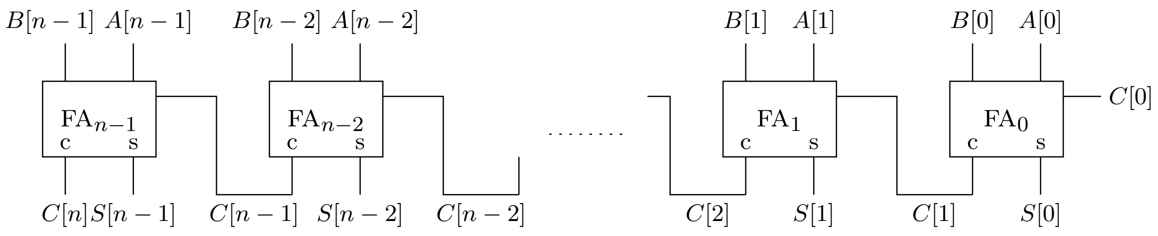


Figure 6.1: A Ripple Carry Adder RCA(n).

6.2.1 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 6.2.

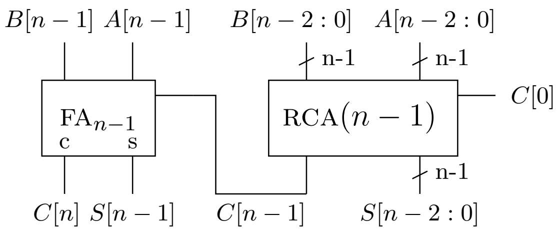


Figure 6.2: A recursive description of RCA(n).

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

Claim 6.1 $\text{RCA}(n)$ *is a correct implementation of* $\text{ADDER}(n)$.

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

$$\langle A[n-2:0]\rangle + \langle B[n-2:0]\rangle + C[0] = 2^{n-1}\cdot C[n-1] + \langle S[n-2:0]\rangle. \tag{6.2}$$

The definition of a Full-Adder states that

$$A[n-1] + B[n-1] + C[n-1] = 2\cdot C[n] + S[n-1]. \tag{6.3}$$

Multiply Equation 6.3 by 2^{n-1} to obtain

$$2^{n-1}\cdot A[n-1] + 2^{n-1}\cdot B[n-1] + 2^{n-1}\cdot C[n-1] = 2^n\cdot C[n] + 2^{n-1}\cdot S[n-1]. \tag{6.4}$$

Note that $2^{n-1}\cdot A[n-1] + \langle A[n-2:0]\rangle = \langle A[n-1:0]\rangle$. By adding Equations 6.2 and 6.4 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. $\square$

6.2.2 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 6.2 implies that the 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. 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.

6.3 Carry bits

The easiest way to define the carry bit associated with an addition

$$\langle A[n-1:0]\rangle + \langle B[n-1:0]\rangle + C[0] = \langle S[n-1:0]\rangle + 2^n\cdot C[n] \tag{6.5}$$

is by considering the internal signals $C[n-1:1]$ of an $\text{RCA}(n)$. We extend this definition of the carry bits to the input $C[0]$ and the output $C[n]$. Hence there are $n+1$ carry-bits $C[n:0]$ associated with the addition defined in Equation 6.5.

We discuss a few issues related to the definition of the carry bits and binary addition.

1. Consider the functionality of an adder:
$$\langle A[n-1:0]\rangle + \langle B[n-1:0]\rangle + C[0] = 2^n \cdot C[n] + \langle S[n-1:0]\rangle.$$
Let $x = \langle \vec{A}\rangle + \langle \vec{B}\rangle + C[0]$. The above equality means that the sum x admits two representations. The representation of x on the right hand side is unique (this is the binary representation). Namely, given x, the bit $C[i+1]$ and the string $S[i:0]$ are uniquely determined. We refer to such a representation as a *non-redundant representation*. One characteristic of non-redundant representation is that if one wishes to compare two numbers x, y represented in the same non-redundant representation by X and Y, respectively, then it suffices to compare their representations X and Y.

   The left hand side represents the same value x but uses 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 $C[0]$. 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, inequality of the representations does not imply inequality of the represented values.

   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.

2. The correctness proof of RCA(n) implies that, for every $0 \leq i \leq n-1$,
$$\langle A[i:0]\rangle + \langle B[i:0]\rangle + C[0] = 2^{i+1} \cdot C[i+1] + \langle S[i:0]\rangle. \tag{6.6}$$
This equality means that the cone of the outputs $C[i+1]$ and $S[i:0]$ is a subset of the inputs $A[i:0] \bigcup B[i:0] \bigcup C[0]$. In fact, this is the cone of $S[i]$ and $C[i+1]$.

3. Equation 6.6 implies that, for every $0 \leq i \leq n-1$,
$$\langle S[i:0]\rangle = \text{mod}(\langle A[i:0]\rangle + \langle B[i:0]\rangle + C[0], 2^{i+1}).$$

4. The correctness of RCA(n) implies that, for every $0 \leq i \leq n-1$,
$$S[i] = \text{XOR}(A[i], B[i], C[i]). \tag{6.7}$$
This immediately implies that, for every $0 \leq i \leq n-1$,
$$C[i] = \text{XOR}(A[i], B[i], S[i]). \tag{6.8}$$

5. Equations 6.7 and 6.8 imply constant-time linear-cost reductions between the problems of computing the sum bits $S[n-1:0]$ and computing the carry bits $C[n-1:0]$. The task of computing the sum bits is the task of an adder. We will later design an optimal adder that will first compute the carry bits, and then obtain the sum bits from the carry-bit by applying bit-wise XOR.

6.4 Conditional Sum Adder

A Conditional Sum Adder is an adder design that is based on divide-and-conquer.

6.4.1 Motivation

Imagine a situation in which Alice positioned on Earth holds the strings $A[k-1:0], B[k-1:0], C[0]$. Bob, 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. How many bits must Alice send to Bob? It suffices to send $C[k]$ to Bob. Now, 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. How what should Bob do during the second it takes $C[k]$ to travel to the Moon? Since the message may be either 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. This story captures the idea behind a conditional sum adder.

6.4.2 Implementation

A Conditional Sum Adder is designed recursively using divide-and-conquer. A csa(1) is simply a Full-Adder. A csa(n), for $n > 1$ is depicted in Figure 6.3. 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 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 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 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 6.3 *Prove the correctness of the* csa(n) *design.*

6.4.3 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 $d(\text{FA})$ denote the delay of a Full-Adder. The delay of a csa(n) satisfies the following recurrence:

$$d(\text{CSA}(n)) = \begin{cases} d(\text{FA}) & \text{if } n = 1 \\ d(\text{CSA}(n/2)) + d(\text{MUX}) & \text{otherwise.} \end{cases}$$

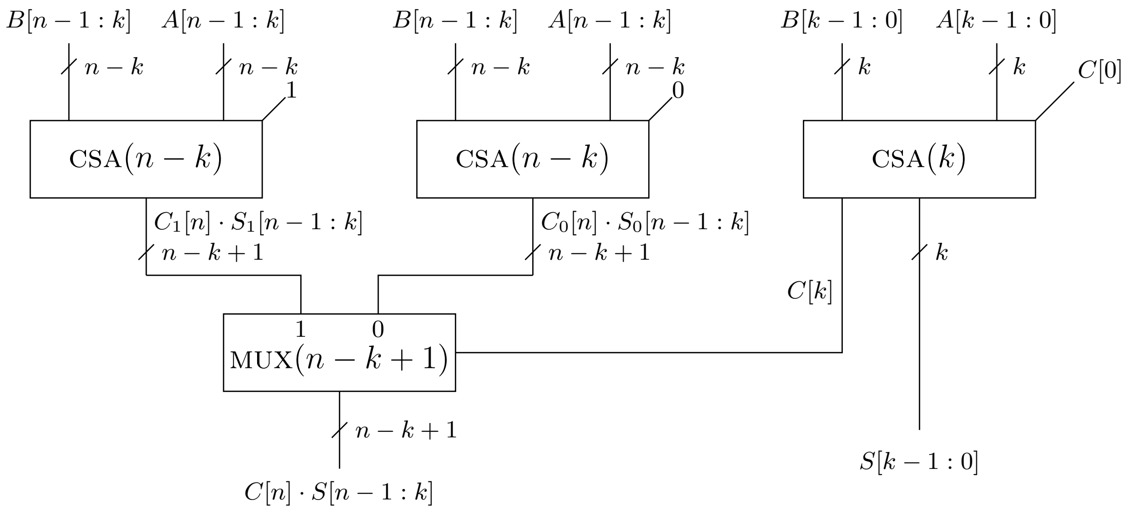


Figure 6.3: A Conditional Sum Adder CSA(n).

It follows that the delay of a CSA(n) is

$$\begin{aligned} d(\text{CSA}(n)) &= \ell \cdot d(\text{MUX}) + d(\text{FA}) \\ &= O(\log n). \end{aligned}$$

Let $c(\text{FA})$ denote the cost of a Full-Adder. The cost of a 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 will solve this recurrence from scratch.
We open two steps of the recurrence to get a feeling of how the different terms grow.

$$\begin{aligned} c(n) &= 3 \cdot c\left(\frac{n}{2}\right) + c\,(\text{MUX}) \cdot \left(\frac{n}{2}+1\right) \\ &= 3 \cdot \left(3 \cdot c\left(\frac{n}{4}\right) + c(\text{MUX}) \cdot \left(\frac{n}{4}+1\right)\right) + c(\text{MUX}) \cdot \left(\frac{n}{2}+1\right) \\ &= 3^2 \cdot c\left(\frac{n}{4}\right) + c(\text{MUX}) \cdot \frac{n}{2} \cdot \left(1+\frac{3}{2}\right) + (1+3)\cdot c\,(\text{MUX}) \end{aligned}$$

We now have a good guess (which can be proved by induction) that

$$c(n) = 3^\ell \cdot c\left(\frac{n}{2^\ell}\right) + c(\text{MUX}) \cdot \frac{n}{2} \cdot \left(1 + \frac{3}{2} + \cdots \left(\frac{3}{2}\right)^{\ell-1}\right) + (1+3+\cdots+3^{\ell-1}) \cdot c(\text{MUX}).$$

Since $\ell = \log_2 n$, it follows that

- $3^\ell = n^{\log_2 3}$

- $(1 + 3 + \cdots + 3^{\ell-1}) < 3^\ell/2 = \frac{1}{2} \cdot n^{\log_2 3}$
- $\frac{n}{2} \cdot \left(1 + \frac{3}{2} + \cdots \left(\frac{3}{2}\right)^{\ell-1}\right) < n^{\log_2 3}$.

We conclude that

$$\begin{aligned} c(n) &< n^{\log_2 3} \cdot \left(c(\text{FA}) + \frac{3}{2} \cdot c(\text{MUX})\right) \\ &\approx \Theta\left(n^{1.58}\right). \end{aligned}$$

We conclude that a $\text{CSA}(n)$ 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 $\text{CSA}(n)$ design does allow us to use three half-sized adders (i.e. addends are $n/2$ bits long) in order to implement a full sized adder (i.e. addends are n bits long).

Question 6.4 (effect of fanout on $\text{CSA}(n)$) *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 $\text{CSA}(n)$?*

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 $\text{CSA}(n)$?

6.5 Compound Adder

The Conditional Sum Adder 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. Such an adder is called a Compound Adder and is defined as follows.

Definition 6.3 *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 $\text{COMP-ADDER}(n)$.

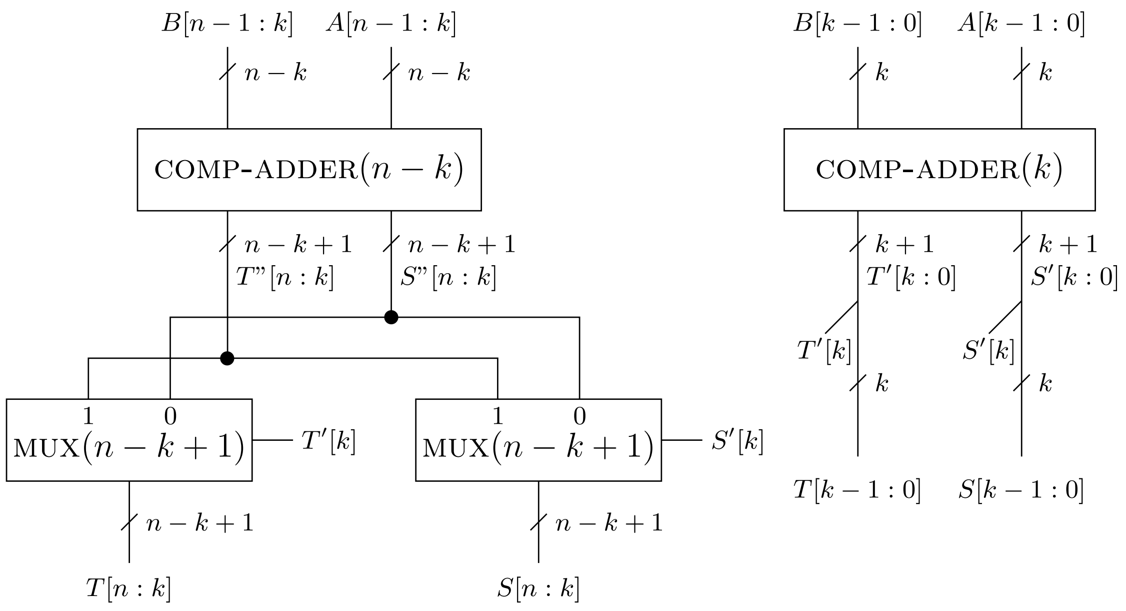


Figure 6.4: A Compound Adder COMP-ADDER(n).

6.5.1 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). The design for $n > 1$ is depicted in Figure 6.4.

Example 6.1 *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 6.5 *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?*

6.5.2 Correctness

We prove the correctness of COMP-ADDER(n).

Claim 6.2 *The* COMP-ADDER(n) *design depicted in Figure 6.4 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 $T[n:0]$ is proved in a similar fashion.

The induction hypothesis implies that

$$\langle S'[k:0]\rangle = \langle A[k-1:0]\rangle + \langle B[k-1:0]\rangle. \tag{6.9}$$

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

$$\begin{aligned}\langle S''[n:k]\rangle &= \langle A[n-1:k]\rangle + \langle B[n-1:k]\rangle\\ \langle T''[n:k]\rangle &= \langle A[n-1:k]\rangle + \langle B[n-1:k]\rangle + 2^k.\end{aligned} \tag{6.10}$$

It follows from Equations 6.9 and 6.10 that

$$\langle S''[n:k]\rangle + \langle S'[k:0]\rangle = \langle A[n-1:0]\rangle + \langle B[n-1:0]\rangle \tag{6.11}$$

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 6.11 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 6.11 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. □

6.5.3 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

$$\begin{aligned}d(\text{COMP-ADDER}(n)) &= \ell \cdot d(\text{MUX}) + d(\text{FA})\\ &= \Theta(\log n).\end{aligned}$$

This recurrence is identical to that of $d(\text{CSA}(n))$, hence $d(\text{COMP-ADDER}(n)) = \Theta(\log n)$.

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 CSA(n), the delay is actually $\Theta(\log^2 n)$.

The cost of a COMP-ADDER(n) satisfies the following recurrence:

$$c(\text{COMP-ADDER}(n)) = \begin{cases} c(\text{FA}) + c(\text{HA}) & \text{if } n = 1 \\ 2 \cdot c(\text{COMP-ADDER}(n/2)) + (n/2+1) \cdot c(\text{MUX}) & \text{otherwise.} \end{cases}$$

We are already familiar with such a recurrence and conclude that $c(\text{COMP-ADDER}) = \Theta(n \log n)$.

6.6 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 (CSA(n)). The 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 (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 COMP-ADDER(n) is much cheaper asymptotically than a CSA(n)! You should make sure that you understand the rational behind this magic. Moreover, by adding a line of multiplexers, one can obtain an ADDER(n) from a COMP-ADDER(n). So the design of a COMP-ADDER(n) is a real improvement over the 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 optimal design is not affected by considering fanout.

Chapter 7

Fast Addition

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.

7.1 Reduction: sum-bits $\longmapsto$ carry-bits

In this section we review the reduction (presented in Section 6.3) of the task of computing the sum-bits to the task of computing the carry-bits.

The correctness of $\textsc{rca}(n)$ implies that, for every $0 \leq i \leq n-1$,

$$S[i] = \textsc{xor}_3(A[i], B[i], C[i]).$$

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 $\textsc{xor}$ of $C[i]$ with $\textsc{xor}(A[i], B[i])$. The cost of this reduction is $2n$ times the cost of a $\textsc{xor}$-gate and the delay is $2 \cdot d(\textsc{xor})$.

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.

7.2 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 $\textsc{fa}_i$ in an $\textsc{rca}(n)$. The functionality of a Full-Adder implies that $C[i+1]$ satisfies:

$$C[i+1] = \begin{cases} 0 & \text{if } A[i] + B[i] + C[i] \leq 1 \\ 1 & \text{if } A[i] + B[i] + C[i] \geq 2. \end{cases} \tag{7.1}$$

The following claim follows directly from Equation 7.1.

Claim 7.1 *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 7.1 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 7.1 *The string* $\sigma[n-1:-1] \in \{0,1,2\}^{n+1}$ *is defined as follows:*

$$\sigma[i] \triangleq \begin{cases} 2 \cdot C[0] & \textit{if } i=-1 \\ A[i]+B[i] & \textit{if } i \in [0,n-1]. \end{cases}$$

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 $\{k,p,g\}$.)

7.2.1 Carry-Lookahead Adders

Carry-Lookahead adders are hierarchical designs in which addends are partitioned into blocks. The number of levels in the hierarchy in this family of adders ranges from two-three levels to $O(\log n)$ levels.

In this section we focus on the computation in a block in the topmost level. We explain the theory behind the design of a block. We then analyze the cost and delay associated with such a block. 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 common block size is limited to a small constant (i.e. 4 bits).

The correctness of Carry-Lookahead Adders is based on the following claim that characterizes when the carry bit $C[i+1]$ equals 1.

Claim 7.2 *For every* $-1 \leq i \leq n-1$,

$$C[i+1]=1 \qquad \Longleftrightarrow \qquad \exists j \leq i \;:\; \sigma[i:j]=1^{i-j}\cdot 2.$$

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 $A[i]+B[i]=\sigma[i]=2$. Hence $C[i+1]=1$.

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 $\sigma[i] = 1$, we conclude that

$$\underbrace{A[i] + B[i]}_{\sigma[i]=1} + C[i] = 2.$$

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 $\sigma[-2] = 2$. 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.
- If $\sigma[i] = 2$, then we set $j = i$.
- If $\sigma[i] = 1$, then $C[i] = 1$.

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 wish to show that Claim 7.2 lays the foundation for Carry-Lookahead Adders. For this purpose, 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 a 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. In this case, we would need to add a bit to binary representation to signify whether $\sigma[i] = 0$. This adds the cost and delay of a NOR-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.

Computation of $C[i+1]$ in a Carry-Lookahead Adder. Figure 7.1 depicts how the carry-bit $C[i+1]$ is computed in a Carry-Lookahead Adder. 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. 7.1. 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 of the form $\sigma[i:j]$ with $1^{i-j} \cdot 2$ are fed to an OR-tree. The OR-tree outputs the carry-bit $C[i+1]$.

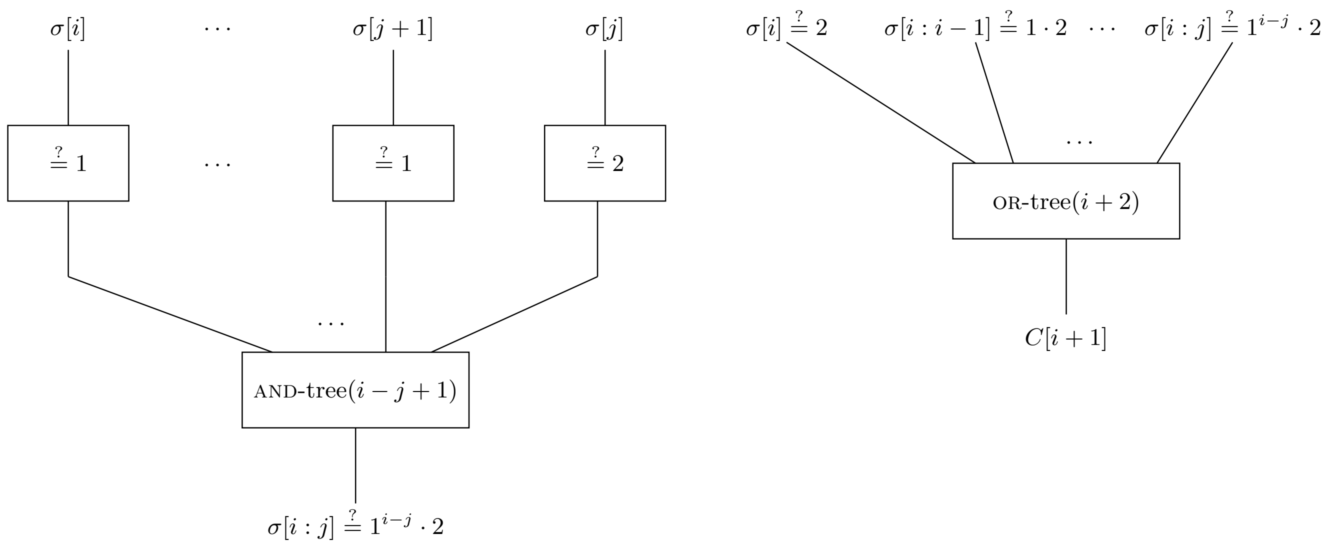

$\sigma[i]$
$\cdots$
$\sigma[j+1]$
$\sigma[j]$
$\overset{?}{=} 1$
$\overset{?}{=} 1$
$\overset{?}{=} 2$
AND-tree$(i-j+1)$
$\sigma[i:j] \overset{?}{=} 1^{i-j} \cdot 2$
$\sigma[i] \overset{?}{=} 2$
$\sigma[i:i-1] \overset{?}{=} 1 \cdot 2$
$\sigma[i:j] \overset{?}{=} 1^{i-j} \cdot 2$
OR-tree$(i+2)$
$C[i+1]$

Figure 7.1: Computing the carry-bit $C[i+1]$ in a Carry-Lookahead Adder.

Note that the computation of the carry bits in a Carry-Lookahead Adder is done in this fashion only within a block (hence n here denotes the size of a block).

Cost and delay analysis. The delay associated with computing $C[i+1]$ in this fashion is clearly logarithmic. The cost of computing $C[i+1]$ in the fashion depicted in Fig 7.1 is

$$\begin{aligned}\sum_{i=1}^{n} c(\text{lookahead } C[i+1]) &= \sum_{i=1}^{n}\left(\sum_{j=-1}^{i} c(\text{AND-tree}(i-j+1)) + c(\text{OR-tree}(i+2))\right)\\ &= \sum_{i=1}^{n}\left(\sum_{j=-1}^{i} O(i-j)\right)\\ &= \sum_{i=1}^{n} O(i^2)\\ &= O(n^3).\end{aligned}$$

We conclude that the cost per block is cubic in the length of the block.

7.2.2 Reduction to prefix computation

Definition 7.2 *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 7.3 *For every* $a \in \{0,1,2\}$*:*

$$\begin{aligned} 0 * a &= 0 \\ 1 * a &= a \\ 2 * a &= 2. \end{aligned}$$

Claim 7.4 *The function* $*$ *is associative. Namely,*

$$\forall a, b, c \in \{0,1,2\} : (a * b) * c = a * (b * c).$$

Question 7.1 *(i) Prove claim 7.4. (ii) Is the function* $*$ *commutative?*

We refer to the outcome of applying the $*$ function by the $*$-product. We also use the notation

$$\prod[j : i] \triangleq \sigma[j] * \cdots * \sigma[i].$$

Associativity of $*$ implies that for every $i \leq j < k$:

$$\prod[k : i] = \prod[k : j+1] * \prod[j : i].$$

The reduction of the computation of the carry-bits to a prefix computation is based on the following claim.

Claim 7.5 *For every* $-1 \leq i \leq n-1$,

$$C[i+1] = 1 \qquad \Longleftrightarrow \qquad \prod[i : -1] = 2.$$

Proof: From Claim 7.2, it suffices to prove that

$$\exists j \leq i \ : \ \sigma[i:j] = 1^{i-j} \cdot 2 \qquad \Longleftrightarrow \qquad \prod[i:-1] = 2.$$

($\Rightarrow$) Assume that $\sigma[i:j] = 1^{i-j} \cdot 2$. It follows that

$$\prod[i:j] = 2.$$

If $j = -1$ we are done. Otherwise, by Claim 7.3

$$\begin{aligned}\prod[i:-1] &= \underbrace{\prod[i:j]}_{=2} * \prod[j-1:-1] \\ &= 2.\end{aligned}$$

($\Leftarrow$) Assume that $\prod[i:-1] = 2$. If, for every $\ell \leq i$, $\sigma[\ell] \neq 2$, then $\prod[i:-1] \neq 2$, a contradiction. Hence

$$\{\ell \in [-1,i] \ : \ \sigma[\ell] = 2\} \ \neq \ \emptyset.$$

Let

$$j \triangleq \max\{\ell \in [-1,i] \ : \ \sigma[\ell] = 2\}.$$

By Claim 7.3, $\prod[j:-1] = 2$. We claim that $\sigma[\ell] = 1$, for every $j < \ell \leq i$.

By the definition of j, $\sigma[\ell] \neq 2$, for every $j < \ell \leq i$. If $\sigma[\ell] = 0$, for $j < \ell \leq i$, then $\prod[i:\ell] = 0$, and then $\prod[i:-1] = \prod[i:\ell] * \prod[\ell-1:-1] = 0$, a contradiction.

Since $\sigma[i:j+1] = 1^{i-j}$, we conclude that $\sigma[i:j] = 1^{i-j} \cdot 2$, and the claim follows. □

A prefix computation problem is defined as follows.

Definition 7.3 *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 7.5 implies a reduction of the problem of computing the carry-bits $C[n:1]$ to the prefix computation problem of computing the prefixes $\prod[0:-1], \ldots, \prod[n:-1]$.

7.3 Parallel prefix computation

In this section we present a general asymptotically optimal circuit for the prefix computation problem.

As in the previous section, let $\text{OP} : \Sigma^2 \longrightarrow \Sigma$ 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 $\text{OP}(a, b)$.

Definition 7.4 *A* Parallel Prefix Circuit*,* $\text{PPC–OP}(n)$*, 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 7.1 *A Parallel Prefix Circuit with (i) the alphabet* $\Sigma = \{0, 1\}$ *and (ii) the function* $\text{OP} = \text{OR}$ *is exactly the Unary Priority Encoder,* $\text{U-PENC}(n)$.

Example 7.2 *Consider* $\Sigma = \{0, 1, 2\}$ *and* $\text{OP} = *$*. The Parallel Prefix Circuit with (i) the alphabet* $\Sigma = \{0, 1, 2\}$ *and (ii) the function* $\text{OP} = *$ *can be used (according to Claim 7.5) to compute the carry-bits.*

Our goal is to design a $\text{PPC–OP}(n)$ using only OP-gates. The cost of the design is the number of OP-gates used. The delay is the maximum number of OP-gates along a directed path from an input to an output.

Question 7.2 *Design a* $\text{PPC–OP}(n)$ *circuit with linear delay and cost.*

Question 7.3 *Design a* $\text{PPC–OP}(n)$ *circuit with logarithmic delay and quadratic cost.*

Question 7.4 *Assume that a design* $C(n)$ *is a* $\text{PPC–OP}(n)$*. This means that it is comprised only of* OP*-gates and works correctly for every alphabet* Σ *and associative function* $\text{OP} : \Sigma^2 \rightarrow \Sigma$*. Can you prove a lower bound on its cost and delay?*

7.3.1 Implementation

In this section we present a linear cost logarithmic delay $\text{PPC–OP}(n)$ design. The design is recursive and uses a technique that we name "odd-even".

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 $y[1] \leftarrow \text{OP}(x[0], x[1])$. The recursion step is depicted in Figure 7.2. Adjacent inputs are paired and fed to an OP-gate. Observe that wires carrying the inputs with even indexes are sent over the $\text{PPC–OP}(n/2)$ so that the even indexed outputs can be computed. The $n/2$ outputs of the OP-gates are fed to a $\text{PPC–OP}(n/2)$. The outputs of the $\text{PPC–OP}(n/2)$ circuit are directly connected to the odd indexed outputs $y[1], y[3], \ldots, y[n-1]$. The even indexed outputs are obtained as follows: $y[2i] \leftarrow \text{OP}(x[2i], y[2i-1])$.

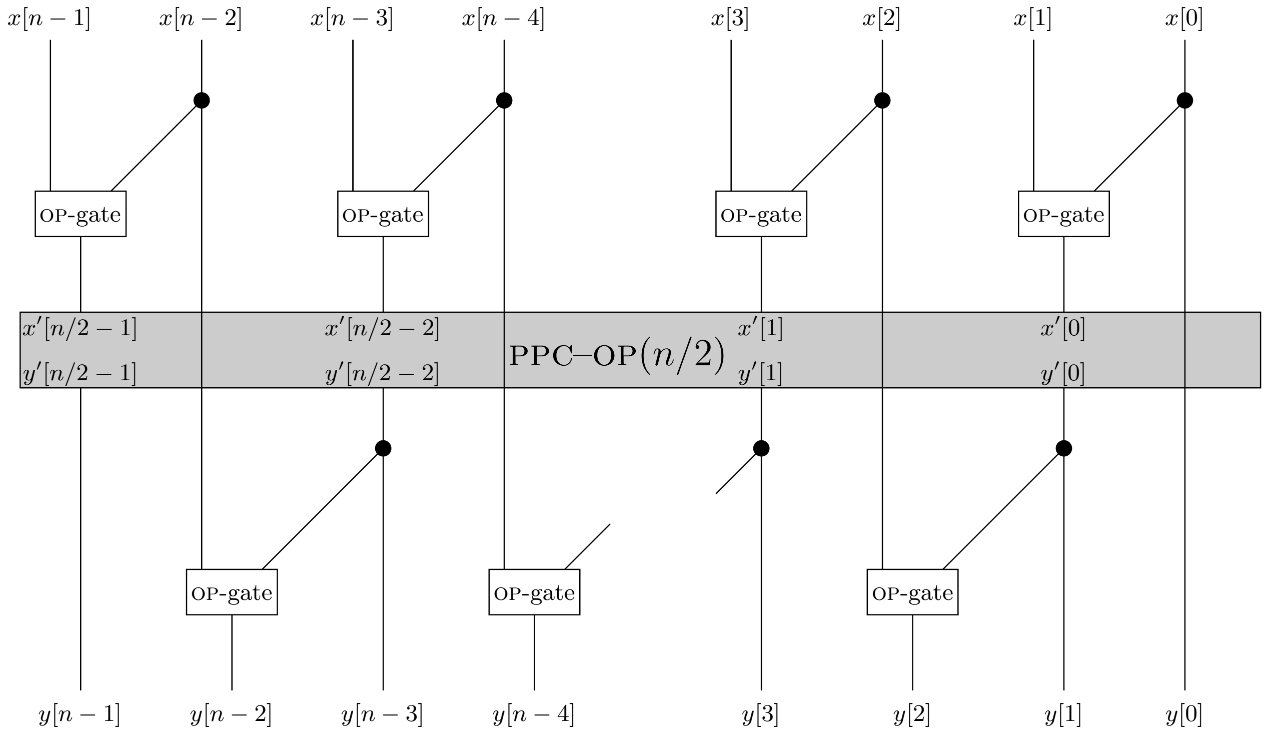


Figure 7.2: A recursive design of PPC–OP(n).

7.3.2 Correctness

Claim 7.6 *The design depicted in Fig. 7.2 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 PPC–OP($n/2$) used in a PPC–OP(n). Let $x'[n/2 - 1 : 0]$ and $y'[n/2 - 1 : 0]$ denote the inputs and outputs of the PPC–OP($n/2$), respectively. The ith input $x'[i]$ equals $\textsc{op}(x[2i + 1], x[2i])$. By the induction hypothesis, the ith output $y'[i]$ satisfies:

$$\begin{aligned} y'[i] &= \textsc{op}_{i+1}(x'[i], \ldots, x'[0]) \\ &= \textsc{op}_{2i+2}(x[2i + 1], \ldots, x[0]). \end{aligned}$$

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. □

7.3.3 Delay and cost analysis

The delay of the 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

$$\begin{aligned} c(\textsc{ppc–op}(n)) &= \sum_{i=2}^{k} (2^i - 1) \cdot c(\textsc{op}\text{-gate}) + c(\textsc{op}\text{-gate}) \\ &= (2n - 4 - (k-1) + 1) \cdot c(\textsc{op}\text{-gate}) \\ &= (2n - \log n - 2) \cdot c(\textsc{op}\text{-gate}). \end{aligned}$$

Corollary 7.7 *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 7.7 implies that $\textsc{ppc–op}(n)$ with $\Sigma = \{0, 1\}$ and $\textsc{op} = \textsc{or}$ is an asymptotically optimal $\textsc{u-penc}(n)$. It also implies that we can compute the carry-bit corresponding to an addition in linear cost and logarithmic delay.

Remark 7.1 *This analysis ignores fanout effects. Note, however, that if we insert a buffer in every branching point of the* $\textsc{ppc–op}(n)$ *design (such branching points are depicted by filled circles), then the fanout is constant. Such an insertion only affects the constants in the analysis of the cost and delay.*

Question 7.5 *What is the maximum fanout in the* $\textsc{ppc–op}(n)$ *design. Analyze the effect of inserting buffers to the cost and delay of* $\textsc{ppc–op}(n)$.

7.4 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 $O(1)$.

PPC– $*$ (n): In this step the products $\prod[i:-1]$ 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 7.5, it follows that $C[i+1]=1$ iff $\prod[i:-1]=2$. In this stage we compare each product $\prod[i:-1]$ 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

$$S[i] = \text{XOR}_3(A[i], B[i], C[i]).$$

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 7.8 *The adder based on parallel prefix computation is asymptotically optimal; its cost is linear and its delay is logarithmic.*

Remark 7.2 *There are representations of* $\{0,1,2\}$ *in which* $\text{XOR}(A[i], B[i])$ *is computed in the stage in which* $\sigma[i]$ *is computed. In this case, we can use this value again in the last stage when the sum-bits are computed.*

7.5 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 prefix computation problem.

A prefix computation problem is the problem of computing $\text{OP}_i(x[i-1:0])$, for $0 \leq i \leq n-1$, where OP is an associative operation. We present a linear cost logarithmic delay circuit for the prefix computation problem. We refer to this circuits as PPC–OP(n). Using a PPC–OP(n) we were also able to design an optimal Unary Priority Encoder.

In Section 7.2.1 we describe the top level of Carry-Lookahead Adders. It is possible to design asymptotically optimal adders based on Carry-Lookahead Adders, however, the design is less systematic and is less general.

Chapter 8

Signed Addition

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.

8.1 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 8.1 *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 8.2 *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 8.3 *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:

$$[A[n-1:0]] \triangleq -2^{n-1} \cdot A[n-1] + \langle A[n-2:0] \rangle.$$

We often use the term "a two's complement number $A[n-1]$". This term refers to the pair consisting of the binary string $A[n-1:0]$ and the number $[A[n-1:0]]$. This term is useful because it attaches a meaning to a binary string and it spares us from having to use the rather long 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.

8.2 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 8.1

$$T_n \triangleq \left\{-2^{n-1}, -2^{n-1}+1, \ldots, 2^{n-1}-1\right\}.$$

Question 8.1 *Prove Claim 8.1*

The following claim deals with negating a value represented in two's complement representation.

Claim 8.2

$$-\left[A[n-1:0]\right] = \left[\textsc{inv}(A[n-1:0])\right] + 1.$$

Proof: Note that $\textsc{inv}(A[i]) = 1 - A[i]$. Hence,

$$\begin{aligned}
\left[\textsc{inv}(A[n-1:0])\right] &= -2^{n-1} \cdot \textsc{inv}(A[n-1]) + \langle \textsc{inv}(A[n-2:0]) \rangle \\
&= -2^{n-1} \cdot (1 - A[n-1]) + \sum_{i=0}^{n-2} (1 - A[i]) \cdot 2^i \\
&= \underbrace{-2^{n-1} + \sum_{i=0}^{n-2} 2^i}_{=-1} + \underbrace{2^{n-1} \cdot A[n-1] - \sum_{i=0}^{n-2} A[i])}_{=-[A[n-1:0]]} \\
&= -1 - \left[A[n-1:0]\right].
\end{aligned}$$

□

In Figure 8.1 we depict a design for negating numbers based on Claim 8.2. 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 $0^{n-1} \cdot 1$.

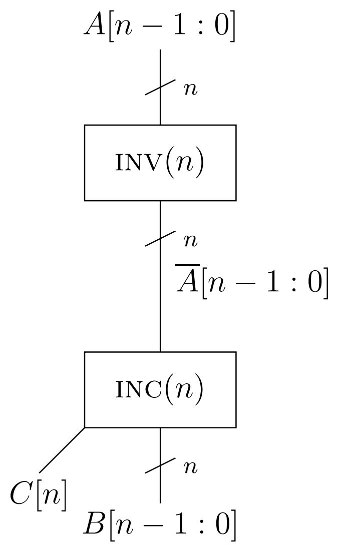


Figure 8.1: A circuit for negating a value represented in two's complement representation.

We would like to claim that the circuit depicted in Fig. 8.1 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 8.2 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 8.2 requires that we compute

$$\left[\overline{A}[n-1:0]\right] + 1.$$

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 8.7. (Question 8.12 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 8.2. 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 8.2 *Propose circuits for negation with respect to other two representations of signed numbers: sign-magnitude and one's-complement.*

8.3 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. The importance of this claim 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 following claim.

Claim 8.3 *For every $A[n-1:0] \in \{0,1\}^n$*

$$\text{mod}(\langle \vec{A} \rangle, 2^n) = \text{mod}(\left[\vec{A}\right], 2^n).$$

Question 8.3 *Prove Claim 8.3.*

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 8.4

$$[A[n-1:0]] < 0 \quad \Longleftrightarrow \quad A[n-1] = 1.$$

Question 8.4 *Prove Claim 8.4.*

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.

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 8.5 *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 8.6

$$[A[n-1]^* \cdot A[n-1:0]] = [A[n-1:0]] .$$

Question 8.5 *Prove Corollary 8.6.*

8.4 Reduction: two's complement addition to binary addition

In Section 8.2 we tried 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

$$\langle A[n-1:0]\rangle + \langle B[n-1:0]\rangle + C[0] = \langle C[n] \cdot S[n-1:0]\rangle. \tag{8.1}$$

Namely, $\vec{A}, \vec{B}$, and $C[0]$ are fed to a binary adder ADDER(n). The theorem addresses the following questions:

- When does the output $S[n-1:0]$ satisfy:

$$\left[\vec{S}\right] = [A[n-1:0]] + [B[n-1:0]] + C[0]? \tag{8.2}$$

- How can we know that Equation 8.2 holds?

Theorem 8.7 *Let $C[n-1]$ denote the carry-bit in position $[n-1]$ associated with the binary addition described in Equation 8.1 and let*

$$z \triangleq [A[n-1:0]] + [B[n-1:0]] + C[0].$$

Then,

$$\begin{aligned}
C[n] - C[n-1] = 1 \quad &\Longrightarrow \quad z < -2^{n-1} && (8.3)\\
C[n-1] - C[n] = 1 \quad &\Longrightarrow \quad z > 2^{n-1} - 1 && (8.4)\\
z \in T_n \quad &\Longleftrightarrow \quad C[n] = C[n-1] && (8.5)\\
z \in T_n \quad &\Longrightarrow \quad z = [S[n-1:0]]\,. && (8.6)
\end{aligned}$$

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

$$A[n-1] + B[n-1] = 2C[n] - C[n-1] + S[n-1]. \tag{8.7}$$

We now expand z as follows:

$$\begin{aligned}
z &= [A[n-1:0]] + [B[n-1:0]] + C[0]\\
&= -2^{n-1} \cdot (A[n-1] + B[n-1]) + \langle A[n-2:0]\rangle + \langle B[n-2:0]\rangle + C[0]\\
&= -2^{n-1} \cdot (2C[n] - C[n-1] + S[n-1]) + \langle C[n-1] \cdot S[n-2:0]\rangle.
\end{aligned}$$

The last line is based on Equation 8.7 and on

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

This 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

$$\begin{aligned}
z &= -2^{n} + [S[n-1:0]]\\
&\leq -2^{n} + 2^{n-1} - 1 = -2^{n-1} - 1.
\end{aligned}$$

   Hence Equation 8.3 follows.

2. If $C[n] - C[n-1] = -1$, then
$$\begin{aligned} z &= 2^n + [S[n-1:0]] \\ &\geq 2^n - 2^{n-1} = 2^{n-1}. \end{aligned}$$
Hence Equation 8.4 follows.

3. If $C[n] = C[n-1]$, then $z = [S[n-1:0]]$, and obviously $z \in T_n$.

Equation 8.5 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 8.6 follows from the third case as well, and the theorem follows. □

8.4.1 Detecting overflow

Overflow occurs when the sum of signed numbers is not in T_n. Using the notation of Theorem 8.7, overflow is defined as follows.

Definition 8.4 *Let* $z \triangleq [A[n-1:0]] + [B[n-1:0]] + C[0]$. *The signal* OVF *is defined as follows:*
$$\text{OVF} \triangleq \begin{cases} 1 & \text{if } z \notin T_n \\ 0 & \text{otherwise.} \end{cases}$$

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 8.7, 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 8.2 holds. Hence, we have a simple way to answer both questions raised before the statement of Theorem 8.7. 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 8.8
$$\text{XOR}(C[n-1], C[n]) = \text{XOR}_4(A[n-1], B[n-1], S[n-1], C[n]).$$

Proof: Recall that
$$C[n-1] = \text{XOR}_3(A[n-1], B[n-1], S[n-1]).$$
□

Question 8.6 *Prove that*

$$\text{OVF} = \text{OR}(\text{AND}_3(A[n-1], B[n-1], \text{INV}(S[n-1])), \text{AND}_3(\text{INV}(A[n-1]), \text{INV}(B[n-1]), S[n-1])).$$

8.4.2 Determining the sign of the sum

How do we determine the sign of the sum z? Obviously, if $z \in T_n$, then Claim 8.4 implies that $S[n-1]$ indicates whether z is negative. However, if overflow occurs, this is not true.

Question 8.7 *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 8.5 *The signal* NEG *is defined as follows:*

$$\text{NEG} \triangleq \begin{cases} 1 & \text{if } z < 0 \\ 0 & \text{if } z \geq 0. \end{cases}$$

A brute force method based on Theorem 8.7 for computing the NEG signal is as follows:

$$\text{NEG} = \begin{cases} S[n-1] & \text{if no overflow} \\ 1 & \text{if } C[n] - C[n-1] = 1 \\ 0 & \text{if } C[n-1] - C[n] = 1. \end{cases} \tag{8.8}$$

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 8.9

$$\text{NEG} = \text{XOR}_3(A[n-1], B[n-1], C[n]).$$

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 8.5), 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 8.7

$$\left[\tilde{S}[n:0]\right] = \left[\tilde{A}[n:0]\right] + \left[\tilde{B}[n:0]\right] + C[0].$$

We conclude that $z = \left[\tilde{S}[n:0]\right]$. It follows that $\text{NEG} = \tilde{S}[n]$. However,

$$\begin{aligned} \tilde{S}[n] &= \text{XOR}_3(\tilde{A}[n], \tilde{B}[n], \tilde{C}[n]) \\ &= \text{XOR}_3(A[n-1], B[n-1], C[n]), \end{aligned}$$

and the claim follows. □

Question 8.8 *Prove that* $\text{NEG} = \text{XOR}(\text{OVF}, S[n-1])$.

8.5 A two's-complement adder

In this section we define and implement a two's complement adder.

Definition 8.6 *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* $\text{NEG}, \text{OVF} \in \{0,1\}$.

Functionality: *Define z as follows:*

$$z \triangleq [A[n-1:0]] + [B[n-1:0]] + C[0].$$

The functionality is defined as follows:

$$\begin{aligned} z \in T_n &\quad\Longrightarrow\quad [S[n-1:0]] = z \\ z \in T_n &\quad\Longleftrightarrow\quad \text{OVF} = 0 \\ z < 0 &\quad\Longleftrightarrow\quad \text{NEG} = 1. \end{aligned}$$

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 8.2 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 $C[0]$.

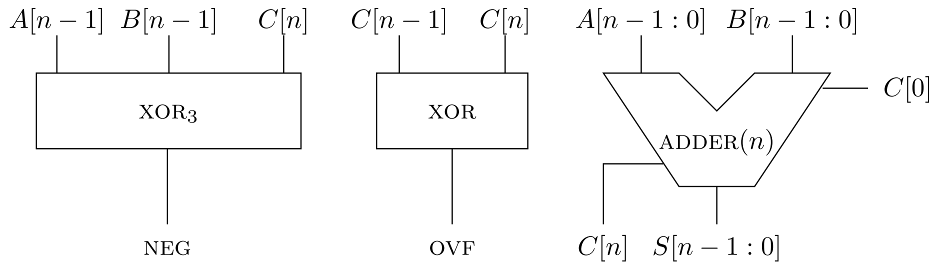


Figure 8.2: A two's complement adder S-ADDER(n)

2. The output OVF is simply XOR($C[n-1], C[n]$) if $C[n-1]$ is available. Otherwise, we apply Claim 8.8, namely, OVF $=$ XOR$_4(A[n-1], B[n-1], S[n-1], C[n])$.
3. The output NEG is compute according to Claim 8.9. 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 8.9 *Prove the correctness of the implementation of* S-ADDER(n) *depicted in Figure 8.2.*

Question 8.10 *Is the design depicted in Figure 8.3 a correct* S-ADDER($2n$)*?*

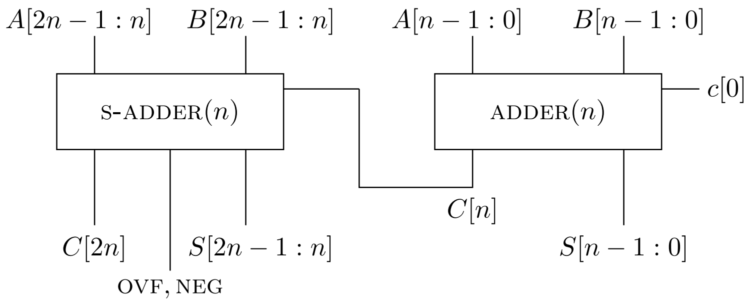


Figure 8.3: Concatenating an S-ADDER(n) with an ADDER(n).

8.6 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 8.7 *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* sub $\in \{0,1\}$.

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

$$\begin{aligned} z \in T_n &\implies [S[n-1:0]] = z \\ z \in T_n &\iff \text{OVF} = 0 \\ z < 0 &\iff \text{NEG} = 1. \end{aligned}$$

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 8.4. The implementation is based on a two's complement adder S-ADDER(n) and Claim 8.2.

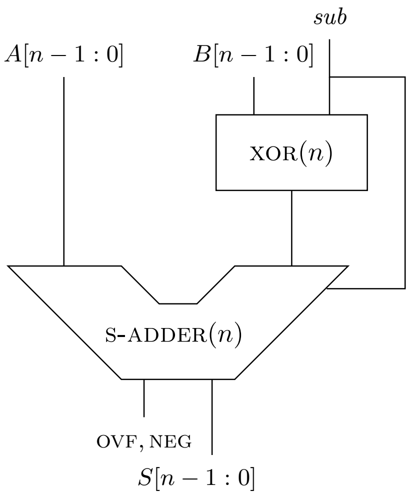


Figure 8.4: A two's-complement adder/subtracter ADD-SUB(n).

Claim 8.10 *The implementation of* ADD-SUB(n) *depicted in Figure 8.4 is correct.*

Question 8.11 *Prove Claim 8.10.*

Question 8.12 (back to the negation circuit) *Consider the negation circuit depicted in Figure 8.1.*

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 8.13 (wrong implementation of ADD-SUB(n)**)** *Find a input for which the circuit depicted in Figure 8.5 errs. Can you list all the inputs for which this circuit outputs a wrong output?*

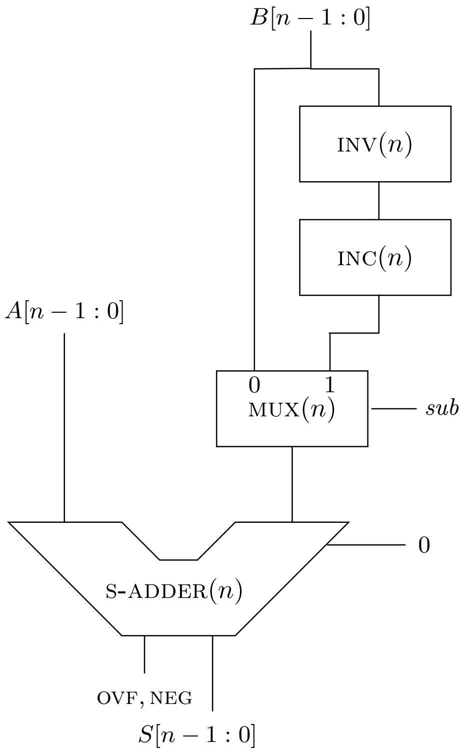


Figure 8.5: A wrong implementation of ADD-SUB(n).

8.7 Additional questions

Question 8.14 (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 8.15 (bi-directional cyclic shifting) *The goal in this question is to design a bi-directional barrel-shifter.*

Definition 8.8 *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 8.16 (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:

$$\begin{array}{ccc} \left[\vec{A}\right] > \left[\vec{B}\right] & \Longleftrightarrow & GT = 1 \\ \left[\vec{A}\right] = \left[\vec{B}\right] & \Longleftrightarrow & EQ = 1 \\ \left[\vec{A}\right] < \left[\vec{B}\right] & \Longleftrightarrow & LT = 1. \end{array}$$

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 8.17 (one's complement adder/subtracter) *Design an adder/subtracter with respect to one's complement representation.*

Question 8.18 (sign-magnitude adder/subtracter) *Design an adder/subtracter with respect to sign-magnitude representation.*

8.8 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 8.3 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 8.4. We reduce the task of two's complement addition to binary addition. Theorem 8.7 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 8.5 we present an implementation of a circuit that adds two's complement numbers. Finally, in Section 8.6 we present an implementation of a circuit that can add and subtract two's complement numbers.

Chapter 9

Flip-Flops

9.1 The clock

Synchronous circuits depend on a special signal called the *clock*. In practice, the clock is generated by rectifying and amplifying a signal generated by special non-digital devices (i.e. crystal oscillators). Since our course is about digital circuits, we use the following abstraction to describe the clock.

Definition 9.1 *A* clock *is a periodic logical signal that oscillates instantaneously between logical one and logical zero. There are two instantaneous transitions in every clock period: (i) in the beginning of the clock period, the clock transitions instantaneously from zero to one; and (ii) at some time in the interior of the clock period, the clock transitions instantaneously from one to zero.*

Figure 9.1 depicts a clock signal. We use the convention that the clock rise occurs in the beginning of the clock period. Note that we assume that the transitions of the clock signal are instantaneous; this is obviously impossible in practice. We show later how we get around this unrealistic assumption.

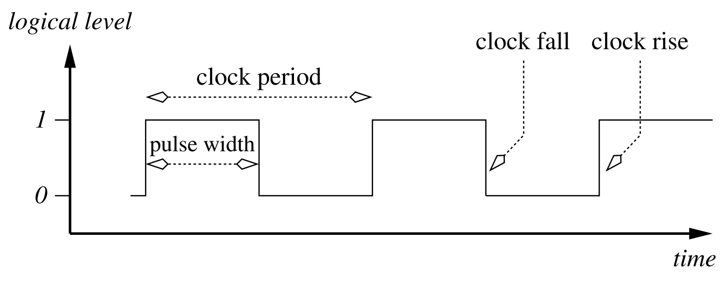


Figure 9.1: A clock signal.

Notation. We denote the clock signal by CLK. We refer to the period of time within a clock period during which the clock equals one as the *clock pulse* (see Fig. 9.1). We denote the clock period by $\varphi(\text{CLK})$. We denote the duration of the clock pulse by CLK_{pw}. A clock CLK is *symmetric* if $\text{CLK}_{pw} = \varphi(\text{CLK})/2$. A clock is said to have *narrow pulses* if $\text{CLK}_{pw} < \varphi(\text{CLK})/2$. A clock is said to have *wide pulses* if $\text{CLK}_{pw} > \varphi(\text{CLK})/2$. See Figure 9.2 for three examples.

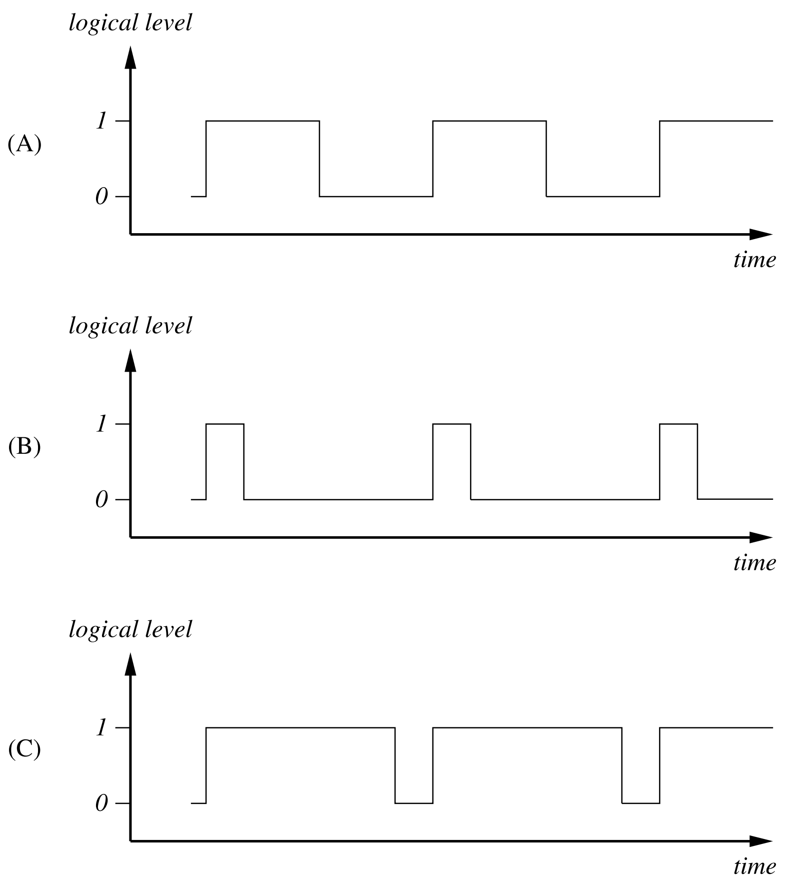


Figure 9.2: (A) A symmetric clock (B) A clock with narrow pulses (C) A clock with wide pulses.

Clock cycles. A clock partitions time into discrete intervals. Throughout this chapter we denote the starting time of the ith clock periods by t_i. We refer to the half-closed interval $[t_i, t_{i+1})$ as *clock cycle i*.

9.2 Edge-triggered Flip-Flop

In this section we define edge-triggered flip-flops.

Definition 9.2 *An edge-triggered flip-flop is defined as follows.*

Inputs: *A digital signal $D(t)$ and a clock* CLK.

Output: *A digital signal $Q(t)$.*

Parameters: *Four parameters are used to specify the functionality of a flip-flop:*

- Setup-time *denoted by* t_{su}*,*
- Hold-time *denoted by* t_{hold}*,*
- Contamination-delay *denoted by* t_{cont}*, and*
- Propagation-delay *denoted by* t_{pd}*.*

These parameters satisfy $-t_{\text{su}} < t_{\text{hold}} < t_{\text{cont}} < t_{\text{pd}}$*. We refer to the interval* $[t_i - t_{\text{su}}, t_i + t_{\text{hold}}]$ *as the* critical segment C_i *and to the interval* $[t_i + t_{\text{cont}}, t_i + t_{\text{pd}}]$ *as the* instability segment A_i*. See Figure 9.3 for a depiction of these parameters.*

Functionality: *If* $D(t)$ *is stable during the critical segment* C_i*, then* $Q(t) = D(t_i)$ *during the interval* $(t_i + t_{\text{pd}}, t_{i+1} + t_{\text{cont}})$*.*

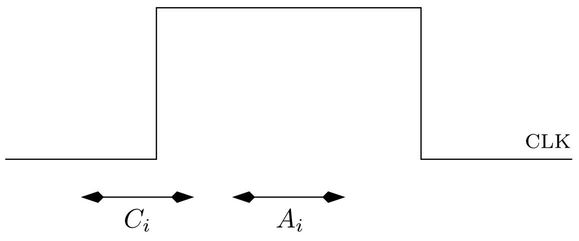


Figure 9.3: The critical segment $C_i = [t_i - t_{su}, t_i + t_{hold}]$ and instability segment $A_i = [t_i + t_{cont}, t_i + t_{pd}]$ corresponding the clock period starting at t_i.

This is a rather complicated definition, so we elaborate.

1. The assumption $-t_{su} < t_{hold} < t_{cont} < t_{pd}$ implies that the critical segment C_i and the instability segment A_i are disjoint.

2. If $D(t)$ is stable during the critical segment C_i, then the value of $D(t)$ during the critical segment C_i is well defined and equals $D(t_i)$.

3. The flip-flop *samples* the input signal $D(t)$ during the critical segment C_i. The sampled value $D(t_i)$ is output during the interval $[t_i + t_{pd}, t_{i+1} + t_{cont}]$. Sampling is successful only if $D(t)$ is stable while it is sampled. This is why we refer to C as a critical segment.

4. If the input $D(t)$ is stable during the critical segments $\{C_i\}_i$, then the output $Q(t)$ is stable in between the instability segments $\{A_i\}_i$.

5. The stability of the input $D(t)$ during the critical segments depends on the clock period. We will later see that slowing down the clock (i.e. increasing the clock period) helps in achieving a stable $D(t)$ during the critical segments.

Figure 9.4 depicts a schematic of an edge-triggered flip-flop. Note the special "arrow" that marks the clock-port. We refer to an edge-triggered flip-flop in short as a flip-flop.

Question 9.1 *Prove that an edge-triggered flip-flop is not a combinational circuit.*

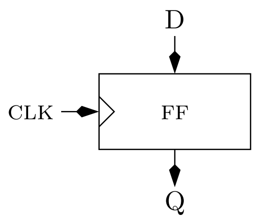


Figure 9.4: A schematic of an edge-triggered flip-flop

9.3 Arbitration

Arbitration is the problem of deciding which event occurs first. For the sake of simplicity we focus on the task of determining which of two signals reaches first the value one. We formally define arbitration as follows.

Definition 9.3 *An* arbiter *is a circuit defined as follows.*

Inputs: *Non-decreasing analog signals $A_0(t), A_1(t)$ defined for every $t \geq 0$.*

Output: *An analog signal $Z(t)$.*

Functionality: *Assume that $A_0(0) = A_1(0) = 0$. Define T_i, for $i = 0, 1$, as follows:*

$$T_i \triangleq \inf\{t \mid \text{dig}(A_i(t)) = 1\}.$$

Let $t' \triangleq 10 + \max\{T_0, T_1\}$. The output $Z(t)$ must satisfy, for every $t \geq t'$,

$$\text{dig}(Z(t)) = \begin{cases} 0 & \textit{if } T_0 < T_1 - 1 \\ 1 & \textit{if } T_1 < T_0 - 1 \\ 0 \textit{ or } 1 & \textit{otherwise.} \end{cases}$$

Note that if T_0 or T_1 equals infinity, then t' equals infinity, and there is no requirement on the output $Z(t)$. The idea is that the arbiter circuit is given 10 time units starting from $\max\{T_0, T_1\}$ to determine if $T_0 < T_1$ or $T_1 < T_0$. We refer to the case in which $|T_0 - T_1| \leq 1$ as a "tie". The arbiter is not required to make a specific decision of a tie occurs. However, even in the case of a tie, the arbiter must make some decision on time and its output $Z(t)$ must have a logical value.

Arbiters are very important in many applications since an arbiter determines the order between events. For example, an arbiter can determine which message arrived first in a network switch.

We will show in this chapter that, under very reasonable assumptions, arbiters do not exist. Moreover, we will show that a flip-flop with an empty critical segment can be used to implement an arbiter. The lesson is that without critical segments flip-flops do not exist.

9.4 Arbiters - an impossibility result

In this section we prove that arbiters do not exist.

Claim 9.1 *There does not exist a circuit C that implements an arbiter.*

Proof: Let C denote a circuit with inputs $A_0(t), A_1(t)$ and output $Z(t)$. Define $A_0(t)$ to be the analog signal that rises linearly in the interval $[0, 100]$ from 0 to $V_{high,in}$, and for every $t \geq 100$, $A_0(t) = V_{high,in}$. Let x denote a parameter that defines $A_1(t)$ as follows: $A_1(t)$ rises linearly in the interval $[0, 100 + x]$ from 0 to $V_{high,in}$, and for every $t \geq 100 + x$, $A_1(t) = V_{high,in}$. Let $f(x)$ denote the function that describes the value of $Z(200)$ (i.e. the value of $Z(t)$ at time $t = 200$) when fed by the signals $A_0(t)$ and $A_1(t)$. We study the function $f(x)$ in the interval $x \in [-2, 2]$. We make the following observations:

1. $f(-2) \geq V_{high,out}$. The reason is that if $x = -2$, then $T_0 = 100$ and $T_1 = 98$. Hence $A_1(t)$ "wins", and by time $t = 200$, the arbiter's output should have stabilized on the logical value 1.

2. $f(2) \leq V_{low,out}$. The reason is that if $x = 2$, then $T_0 = 100$ and $T_1 = 102$. Hence $A_0(t)$ "wins", and $dig(Z(200)) = 0$.

3. $f(x)$ is continuous in the interval $[-2, 2]$. This is not a trivial statement and its formal proof is not within the scope of this course. We provide an intuitive proof of this fact. The idea of the proof of the continuity of $f(x)$ is that the output $Z(200)$ depends on the following: (i) The initial state of the device C at time $t = 0$. We assume that the device C is in a stable state and that the charge is known everywhere. (ii) The signal $A_i(t)$ in the interval $[0, 200]$, for $i = 0, 1$.

   An infinitesimal change in x affects only $A_1(t)$ (i.e. the initial state of the circuit and $A_0(t)$ are not affected by x). Moreover, the difference in energy of $A_1(t)$ corresponding to two very close values of x is infinitesimal. Hence, we expect the difference in $Z(200)$ for two very close values of x to be also infinitesimal. If this were not the case, then noise would cause uncontrollable changes in $Z(t)$ and the circuit C would not be useful anyhow.

By the Mean Value Theorem, it follows that, for every $y \in [V_{low,out}, V_{high,out}]$, there exists an $x \in [-2, 2]$ such that $f(x) = y$. In particular, choose a value y for which $dig(y)$ is not logical. We conclude that circuit C is not a valid arbiter since its output can be forced to be non-logical way past the time it should be logical. □

Claim 9.1 and its proof are very hard to grasp at first. It seems to imply some serious flaw in our perception. Among other things, the claim implies that there does not exist a perfect judge who can determine the winner in a 100-meters dash. This statement remains true even in the presence of high speed cameras located at the finish line and the runners run slowly. This statement remains true even if we allow the judge several hours to decide. This statement remains true even if we allow the judge to decide arbitrarily if the running times of the winner and runner-up are within a second! Does this mean that races are pointless since, for every judge, there exist two runners whose running times are such that the judge still hangs after an hour?

Our predicament can be clarified by the following example depicted in Figure 9.5. Consider a player whose goal is to throw a ball past an obstacle so that it rolls past point P. If the ball is rolled at a speed above v', then it will pass the obstacle and then roll past point P. If the ball is thrown at a speed below v' it will not pass the obstacle. The judge is supposed to announce her decision 24 hours after the player throws the ball. The judge's decision must be either "passed" or "did not pass". Seems like an easy task. However, if the player throws the ball at speed v', then the ball reaches the tip of the obstacle and may remain there indefinitely long! If the ball remains on the obstacle's tip 24 hours past the throw, then the judge cannot announce her decision.

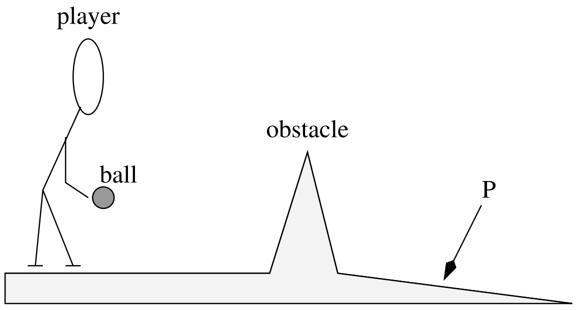


Figure 9.5: A player attempting to roll a ball so that it passes point P.

We refer to the state of the ball when resting on the tip of the obstacle as a *meta-stable* state of equilibrium. Luckily, throwing the ball so that it rests on the tip of the obstacle is a very hard task. Suppose there is some probability distribution for the speed of the ball when thrown. Unless this probability distribution is pathologic, the probability of obtaining a meta-stable state is small. Moreover, the probability of meta-stability occurring can be made even smaller by sharpening the tip of the obstacle or giving the arbiter more time to decide. This ability to control the probability of the event that a decision cannot be made plays a crucial role in real life. In VLSI chips, millions of transistors transition from one state to another millions of times per second. If even one transistor is "stuck" in a meta-stable state, then the chip might output a wrong value. By reducing the probability of meta-stability, one can estimate that meta-stability will not happen during the life-time of the chip (a lightening will hit the chip before meta-stability happens).

The consequence of this discussion is that Claim 9.1 does not make judges unemployed just as a coin toss is not likely to end up with the coin standing on its perimeter (but bear in mind that it could!). The moral of Claim 9.1 is that: (i) Certain tasks are not achievable with probability 1. If we consider the random nature of noise, we should not be surprised at all. In fact, noise could be big enough to cause the digital value of a signal to flip from zero to one. If the noise margin is large enough, then such an event is not likely to occur. However, there is always a positive probability that such an error will occur. (ii) Increasing the amount of time during which the arbiter is allowed to reach a decision (significantly) decreases the chances of meta-stability. As time progresses, even if the ball is resting on the tip of the obstacle, it is likely to fall to one of the sides. Note, however, that increasing the clock rate means that "decisions" must be made faster (i.e. within a clock period) and the chance of meta-stability increases.

Question 9.2 *Does the proof of Claim 9.1 hold only if the signals $A_i(t)$ rise gradually? Prove the claim with respect to non-decreasing signals $A_i(t)$ such that the length of the interval during which* $\text{dig}(A_i(t))$ *is non-logical equals ε.*

9.5 Necessity of critical segments

In this section we present a reduction from flip-flops without critical segments to arbiters. Since arbiters do not exist, the implication of this reduction is that flip-flops without critical segments do not exist as well.

We define a flip-flop without a critical segment as a flip-flop in which the setup-time and hold-time satisfy $t_{su} = t_{hold} = 0$. The functionality is defined as follows: For every i, $Q(t)$ is logical (either zero or one) during the interval $t \in (t_i + t_{pd}, t_{i-1} + t_{cont})$ regardless of whether $D(t_i)$ is logical. If $D(t_i)$ is logical, then $Q(t) = D(t_i)$ during the interval $t \in (t_i + t_{pd}, t_{i-1} + t_{cont})$.

The definition of a flip-flop without a critical segment is similar to an arbiter. Just as the arbiter's decision is free if a tie occurs, the flip-flop is allowed to output zero or one if $D(t_i)$ is not logical. However, the output of the flip-flip must be logical once the instability segment ends.

Consider the circuit depicted in Figure 9.6 in which the flip-flop is without a critical segment. Assume that the parameters t_{cont} and t_{pd} are significantly smaller than one time unit (e.g. at most 10^{-9} second, where one time unit equals one second). Assume also that the intervals during which the inputs $A_0(t)$ and $A_1(t)$ are non-logical are also very short (e.g. 10^{-9} second).

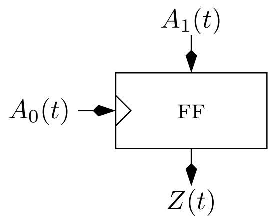


Figure 9.6: An arbiter based on a flip-flop without a critical segment.

Note that the signal $A_0(t)$ is input as a clock to the flip-flop. Our requirements from $A_0(t)$ are somewhat weaker than the requirements from a clock. Instead of periodic instantaneous transitions from zero to one and back, $A_0(t)$ is non-decreasing. The claim assumes only one "tick of the clock", so we may regard $A_0(t)$ as a clock with a very long period. On the other hand, we do not rely on $A_0(t)$ rising slowly; the claim holds regardless of the rate of change of $A_0(t)$.

Claim 9.2 *The circuit depicted in Figure 9.6 is an arbiter.*

Proof: If $T_0 < T_1 - 1$, then we claim that $dig(A_1(T_0)) = 0$. The reason that since $T_0 < T_1$, it follows that $dig(A_1(T_0))$ is either zero or non-logical. If it is non-logical, then the assumption on the fast transition of $dig(A_1(t))$ from zero to one implies that $dig(A_1(T_0 + 10^{-9})) = 1$, and hence, $T_1 \leq T_0 + 10^{-9}$. But then we have a contradiction to the assumption that $T_1 > T_0 + 1$.

It follows that if $T_0 < T_1 - 1$, then $dig(A_1(T_0)) = 0$, and hence, $dig(Z(t)) = 0$, for every $t \geq T_0 + t_{pd}$.

If $T_1 < T_0 - 1$, then $dig(A_1(T_0)) = 1$, and hence, $dig(Z(t)) = 1$, for every $t \geq T_0 + t_{pd}$.

Since the flip-flop's output $Z(t)$ is always logical at time $T_0 + t_{pd}$, it follows that the circuit is an arbiter, and the claim follows. □

Claims 9.1 and 9.2 imply that a flip-flop without a critical segment does not exist. In other words, for every flip-flop, if there is no critical segment requirement, then there exist input signals that can cause it to output a non-logical value outside of the instability segment.

Corollary 9.3 *There does not exist an edge-triggered flip-flop without a critical segment.*

9.6 An example

Figure 9.7 depicts a circuit consisting of two identical flip-flops and a combinational circuit C in between. A simplified timing diagram of this circuit is depicted in Figure 9.8. Instead of drawing the clock signal, only the times t_i and t_{i+1} are marked on the time axis. In addition the critical segment and instability segment are depicted for each clock period. The digital signals $D_0(t), Q_0(t), D_1(t), Q_1(t)$ are depicted using a simplified timing diagram. In this diagram, intervals during which a digital signal is guaranteed to be stable are marked by a white block. On the other hand, intervals during which a digital signal is possibly non-logical are marked by a gray block.

In this example, we assume that the signal $D_0(t)$ is stable only during the critical segments. As a result, the signal $Q_0(t)$ is stable in the complement of the instability segments. The signal $D_1(t)$ is output by the combinational circuit C. The signal $D_1(t)$ becomes instable as soon as $Q_0(T)$ (the input of C) becomes instable. We denote the propagation delay of C by $d(C)$. The signal $D_1(t)$ stabilizes at most $d(C)$ time units after $Q_0(t)$ stabilizes. Note that we do not assume that the the contamination delay of C is positive (often combinational devices do have guarantees for positive contamination delays, but we do not rely on it in this course). The signal $D_1(t)$ is stable during the critical segment C_{i+1}, and therefore, $Q_1(t)$ is stable during the complement of the instability segments.

From a functional point of view, stability of $D_0(t)$ during the critical segments implies that $D_0(t_i)$ is logical. We denote $D_0(t_i)$ by $\sigma \in \{0,1\}$. During the interval $[t_i + t_{pd}, t_{i+1} + t_{cont}]$ the flip-flop's output $Q_0(t)$ equals σ. The circuit C outputs a logical value $\sigma' \in \{0,1\}$ which is a Boolean function of σ. The value σ' is output by C during the interval $[t_i + t_{pd} + d(C), t_{i+1} + t_{cont}]$. It follows that $Q_1(t)$ equals σ' during the interval $[t_{i+1} + t_{pd}, t_{i+2} + t_{cont}]$.

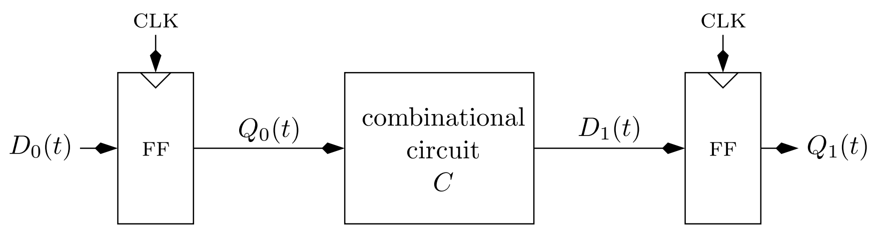


Figure 9.7: A circuit with two identical flip-flips and a combinational circuit in between.

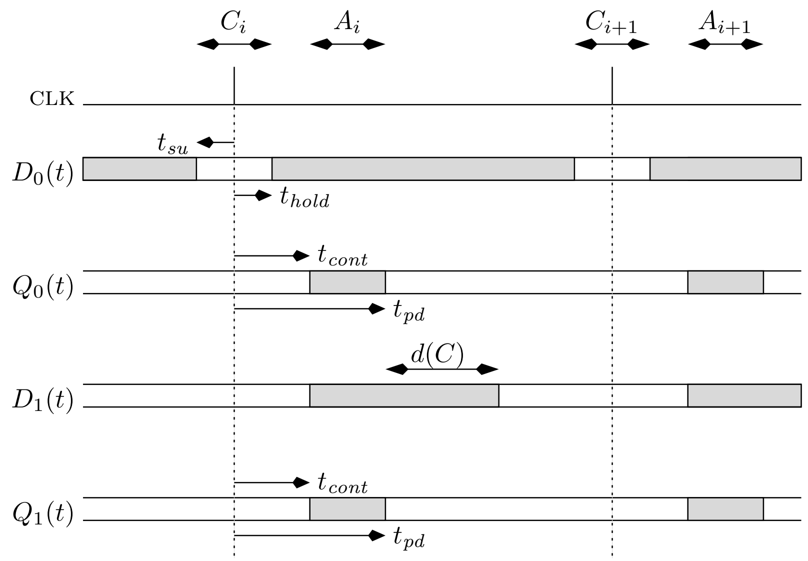


Figure 9.8: A simplified timing diagram of circuit depicted in Fig. 9.7. Gray areas denote potential instability of a signal, and white areas denote guaranteed stability of a signal.

9.6.1 Non-empty intersection of C_i and A_i

The above analysis fails if the critical segment C_i and the instability segment intersect, namely,

$$C_i \cap A_i \neq \emptyset.$$

This could happen, if $t_{hold} > t_{cont}$ (in contradiction to Definition 9.2).

We now explain why this can cause the circuit to fail (see Figure 9.9). The period during which $D_1(t)$ is guaranteed to be stable is $[t_i + t_{pd} + d(C), t_{i+1} + t_{cont}]$. However, if $t_{cont} < t_{hold}$, then $D_1(t)$ is not guaranteed to be stable during the critical segment C_{i+1}. This is a violation of the assumptions we require in order to guarantee correct functionality.

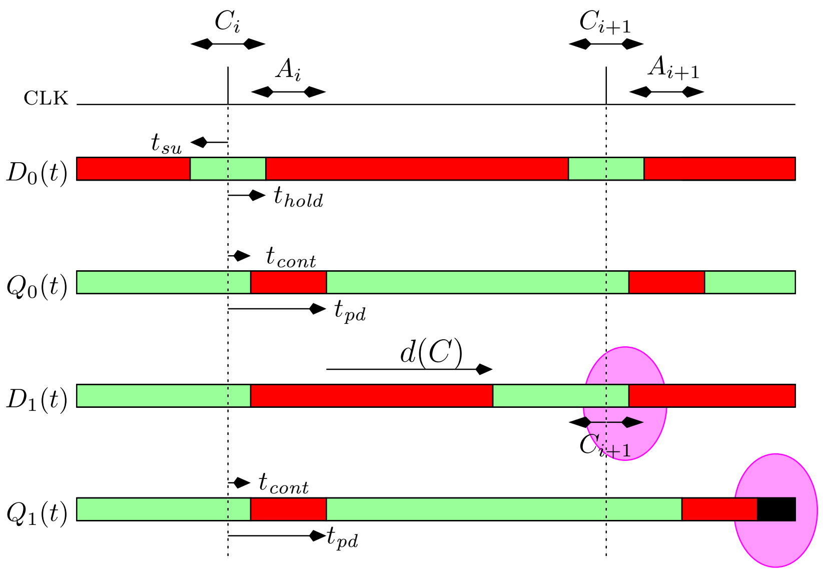


Figure 9.9: The simplified timing diagram in the case that $A_i \cap C_i \neq \emptyset$.

In many flip-flop implementations it so happens that $t_{hold} > t_{cont}$. How are such flip-flops used? The answer is that one needs to rely on the contamination delay of the combinational circuit C. Let $cont(C)$ denote the contamination delay of C. The interval during which $D_1(t)$ is guaranteed to be stable is

$$[t_i + t_{pd} + d(C), t_{i+1} + t_{cont} + cont(C)].$$

If $t_{cont} + cont(C) > t_{hold}$, then the signal $D_1(t)$ is stable during the critical segment C_{i+1}, and correct functionality is obtained.

In this course we simplify by adopting the more restrictive assumption that the contamination delay of every combinational circuit is zero. This means that we need to be more restrictive with respect to flip-flops and require that the critical segment and the instability segments are disjoint. Note, however, that even if the contamination delay of C is positive (although we assumed it is zero), then our analysis is still valid. Hence, not relying on a positive contamination delay of combinational circuits does not introduce errors even if the contamination delay is positive.

Question 9.3 *Assume that we have an edge-triggered flip-flop* FF *in which* $t_{\text{hold}} > t_{\text{cont}}$. *Suppose that we have an inverter with a contamination delay* $\text{cont}(\text{INV}) > 0$. *Suggest how to design an edge-triggered flip-flop* FF′ *that satisfies* $t_{\text{hold}}(\text{FF}') < t_{\text{cont}}(\text{FF}')$. *What are the parameters of* FF′*?*

9.7 Other types of memory devices

Edge triggered flip-flops are not the only memory device that exist. We briefly overview some of these devices.

9.7.1 D-Latch

A D-latch, like an edge-triggered flip-flop, is characterized by two parameters t_{su}, t_{hold}. However, the critical segment is defined with respect to the falling edge of the clock. Let t'_i denote the time of the falling edge of the clock during the ith clock cycle. The critical segment of a D-latch is defined to be $[t'_i - t_{su}, t'_i + t_{hold}]$. In addition, the D-latch is characterized by a combinational delay d. The functionality of a D-latch is defined as follows.

1. During the interval $[t_i + d, t'_i)$, the output $Q(t)$ satisfies: $Q(t) = D(t)$, provided that $D(t)$ is stable during the interval $[t - d, t]$. We say that the D-latch is *transparent* during the interval $[t_i + d, t'_i)$.
2. During the interval $(t'_i + t_{hold}, t_{i+1})$, if $D(t)$ is stable during the critical segment $[t'_i - t_{su}, t'_i + t_{hold}]$, then $Q(t) = D(t'_i)$. We say that the D-latch is opaque during the interval $(t'_i + t_{hold}, t_{i+1})$.

D-latches are very important devices. They are cheaper than flip-flops, and in fact, D-latches are the building blocks of flip-flops. Moreover, using D-latches wisely leads to faster designs. However, designs based on D-latches require multiple clock phases (or at least a clock CLK and its negation $\overline{\text{CLK}}$). Although timing with multiple clock phases is an important and interesting topic, we do not deal with it in this course.

9.7.2 Clock enabled flip-flips

We use the terminology and notation of an edge-triggered flip-flop in the definition of a clock enabled flip-flop.

Definition 9.4 *A clock enabled flip-flop is defined as follows.*

Inputs: *Digital signals* $D(t), \text{CE}(t)$ *and a clock* CLK.

Output: *A digital signal* $Q(t)$.

Functionality: *If* $D(t)$ *and* $\text{CE}(t)$ *are stable during the critical segment* C_i*, then for every* $t \in (t_i + t_{\text{pd}}, t_{i+1} + t_{\text{cont}})$

$$Q(t) = \begin{cases} D(t_i) & \textit{if } \text{CE}(t_i) = 1 \\ Q(t_i) & \textit{if } \text{CE}(t_i) = 0. \end{cases}$$

We refer to the input signal CE(t) as the clock-enable signal. Note that the input CE(t) indicates whether the flip-flop samples the input $D(t)$ or maintains its previous value.

Part (A) of Figure 9.10 depicts a successful implementation of a clock enabled flip-flop. This implementation uses a MUX and an edge-triggered flip-flop. Part (B) of Figure 9.10 depicts a weak implementation of a clock enabled flip-flop.

The main weakness of the design depicted in part (B) is that the output of the AND-gate is not a clock signal. For example, the output of the AND-gate is allowed to fluctuate when CE(t) is not logical. Such fluctuations (called *glitches*) can cause the flip-flop to sample the input when not needed. In addition, the transitions of the output of the AND-gate might be slow and require increasing the hold time. Moreover, in some technologies, the flip-flop does not retain the stored bit forever. For example, consider the case in which the stored value is retained for 2-3 clock cycles. In such a case, if the clock-enable signal is low for a long period then the flip-flop's output may become non-logical.

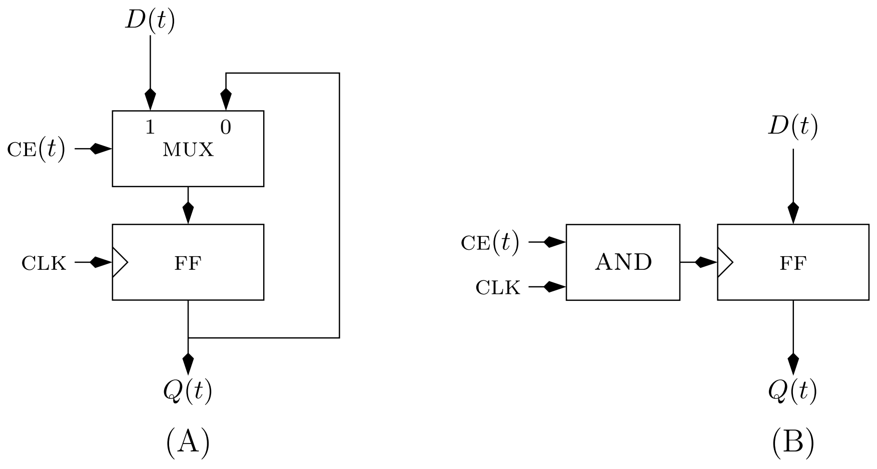


Figure 9.10: (A) a successful implementation of a clock enabled flip-flop. (B) A wrong design.

Question 9.4 *Compute the parameters of the clock-enabled flip-flop depicted in part (A) of Figure 9.10 in terms of the parameters of the edge-triggered flip-flop and the* MUX.

Question 9.5 *Define what an edge-triggered flip-flop with a clear (or reset) signal is. Suggest an implementation of an edge-triggered flip-flop with a clear signal.*

Chapter 10

Synchronous Circuits

10.1 Syntactic definition

Definition 10.1 *A* synchronous circuit *is a circuit C composed of combinational gates, nets, and flip-flops that satisfies the following conditions:*

1. *There is a net called* CLK *that carries a clock signal.*
2. *The* CLK *net is fed by an input gate.*
3. *The set of ports that are fed by the* CLK *net equals the set of clock-inputs of the flip-flops.*
4. *Define the circuit C' as follows: The circuit C' is obtained by (i) deleting the* CLK *net, (ii) deleting the input gate that feeds the* CLK *net, and (iii) replacing each flip-flip with an output gate (instead of the port D) and an input gate (instead of the port Q). We require that the circuit C' is combinational.*

We remark that in a synchronous circuit the clock signal is connected only to the clock port of the flip-flops; the clock may not feed other inputs (i.e. inputs of combinational gates or the D-port of flip-flops). Moreover, every clock-port of a flip-flop is fed by the clock signal.

Figure 10.1 depicts a synchronous circuit C and the corresponding combinational circuit C'.

Question 10.1 *Suggest an efficient algorithm that decides if a given circuit is synchronous.*

Question 10.2 *In Definition 10.1 we required that the clock signal feed only the clock ports of flip-flops. Consider the circuit depicted in Figure 10.2 that violates this requirement. Proper functioning of the D-FF (and hence the circuit) is guaranteed only if setup and hold time requirements are satisfied. Under which conditions can the signal feeding the D port of the D-FF satisfy the setup and hold time requirements?*

10.2 Timing analysis: the canonic form

In this section we analyze the timing constraints of a synchronous circuit that is given in canonic form.

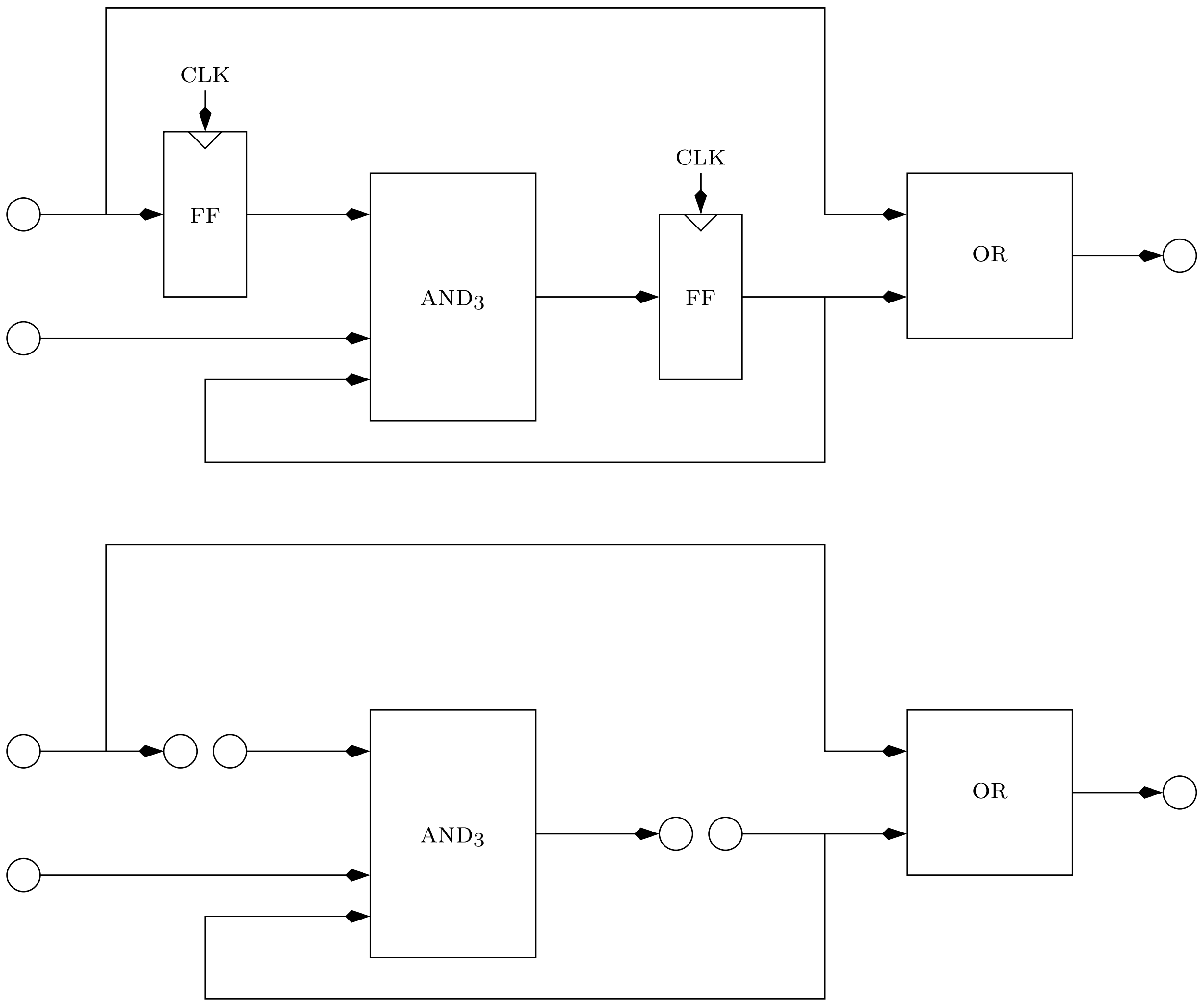


Figure 10.1: A synchronous circuit C and the corresponding combinational circuit C'.

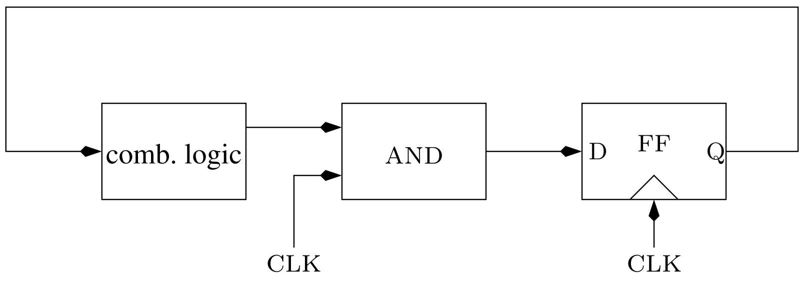


Figure 10.2: A non-synchronous circuit in which the clock feeds also a gate.

10.2.1 Canonic form of a synchronous circuit

Consider the synchronous circuit depicted in Figure 10.3. The circuit has an input IN, and output OUT, and internal signals S (for "state") and NS (for "next state"). We abuse notation and refer to the combinational circuits λ and δ by the Boolean functions that they implement. In this example, all the signals in the circuit carry single bits (as normal signals do). However, we could easily deal with the case in which IN, OUT, S, NS are buses (i.e. multiple-bit signals).

One can transform every synchronous circuit so that it fits the description in Figure 10.3. This is achieved by: (i) gathering the flip-flops into one group and (ii) duplicating the combinational circuits (if necessary) so that we can separate between the combinational circuits that produce output signals and combinational circuits that produce signals that are fed back to the flip-flops. This is why we refer to the circuit depicted in Figure 10.3 as a *canonic form* of a synchronous circuit.

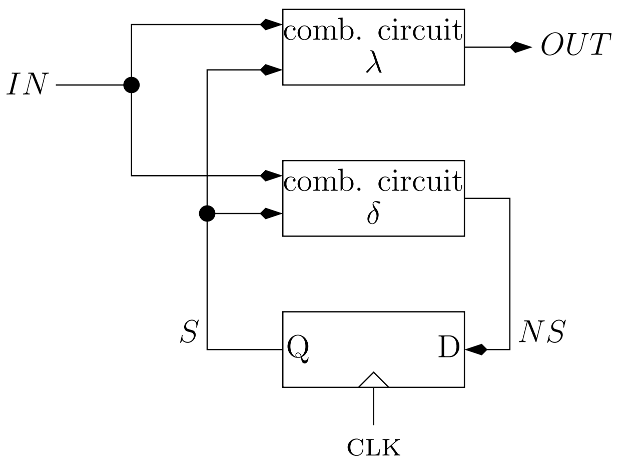


Figure 10.3: A synchronous circuit in canonic form.

10.2.2 Timing constraints

Stability interval. We associate with each signal an interval corresponding to the ith clock cycle during which the signal is stable. We refer to this interval as the *stability interval*. We denote the stability interval corresponding to the ith interval of signal X by $\mathit{stable}(X)_i$. We denote the value of $\mathit{dig}(X)$ during the interval $\mathit{stable}(X)_i$ by X_i.

Input/Output timing constraints. The input/output timing constraints formulate the timing interface between the the circuit and the "external world". The constraint corresponding to the input tells us when the input is guaranteed to be stable, and the constraint corresponding to the output tells us when the circuit's output is required to be stable. Usually the external world is also a synchronous circuit. This means that the signal IN is an output of another synchronous circuit. Similarly, the signal OUT is an input of another synchronous circuit.

1. The timing constraint corresponding to IN is defined by two parameters: $pd(IN) > cont(IN)$ as follows. The stability intervals of signal IN are guaranteed to satisfy

$$\forall i \geq 0 : \quad [t_i + pd(IN), t_{i+1} + cont(IN)] \subseteq stable(IN)_i. \tag{10.1}$$

   Recall that t_i denotes the starting time of the ith clock period. Note that if $pd(IN) \leq cont(IN)$, then the stability intervals $stable(IN)_i$ and $stable(IN)_{i+1}$ overlap. This means that IN is always stable, which is obviously not an interesting case.

2. The timing constraint corresponding to OUT is defined by two parameters: $setup(OUT)$ and $hold(OUT)$ as follows. The stability intervals of signal OUT must satisfy the following condition:

$$\forall i \geq 0 : \quad [t_{i+1} - setup(OUT), t_{i+1} + hold(OUT)] \subseteq stable(OUT)_i. \tag{10.2}$$

   Note that that timing constraint of OUT is given relative to the end of the ith cycle (i.e. t_{i+1}) .

Note that there is an asymmetry in the terminology regarding IN and OUT. The parameters associated with IN are $pd(IN)$ and $cont(IN)$, whereas the parameters associated with OUT are $setup(OUT)$ and $hold(OUT)$. This is not very aesthetic if OUT is itself an input to another synchronous circuit. The reason for this asymmetric choice is that it is useful to regard IN as an output of a flip-flip and OUT as an input of a flip-flop (even if they are not).

Timing constraints of internal signals. The only constraint we have for an internal signal is that NS is stable during the critical segments. Namely,

$$\forall i \geq 0 : \quad C_{i+1} \subseteq stable(NS)_i.$$

Note that, as in the case of the output signal, the timing constraint of NS corresponding to clock cycle i is relative to the end of the ith clock cycle (i.e. the critical segment C_{i+1}).

10.2.3 Sufficient conditions

We associate a contamination delay $cont(x)$ and a propagation delay $pd(x)$ with each combinational circuit x. The following claim follows directly from the propagation delay and contamination delay of the combinational circuits λ and δ.

Claim 10.1 *If*

$$[t_i + t_{\text{pd}}, t_{i+1} + t_{\text{cont}}] \subseteq \text{stable}(S)_i, \tag{10.3}$$

then the stability intervals of the signals OUT and NS satisfy:

$$[t_i + \max\{t_{\text{pd}}, \text{pd}(IN)\} + pd(\lambda), t_{i+1} + \min\{t_{\text{cont}}, \text{cont}(IN)\} + \text{cont}(\lambda)] \subseteq \text{stable}(OUT)_i \tag{10.4}$$

$$[t_i + \max\{t_{\text{pd}}, \text{pd}(IN)\} + pd(\delta), t_{i+1} + \min\{t_{\text{cont}}, \text{cont}(IN)\} + \text{cont}(\delta)] \subseteq \text{stable}(NS)_i. \tag{10.5}$$

The following claim provides a sufficient condition so that the timing constraint of *OUT* holds.

Claim 10.2 *If*

$$[t_i + t_{\text{pd}}, t_{i+1} + t_{\text{cont}}] \subseteq \text{stable}(S)_i$$
$$\max\{t_{\text{pd}}, \text{pd}(IN)\} + pd(\lambda) + \text{setup}(OUT) \leq t_{i+1} - t_i \tag{10.6}$$
$$\min\{t_{\text{cont}}, \text{cont}(IN)\} + \text{cont}(\lambda) \geq \text{hold}(OUT), \tag{10.7}$$

then the timing constraint of OUT corresponding to cycle i holds, namely,

$$[t_{i+1} - \text{setup}(OUT), t_{i+1} + \text{hold}(OUT)] \subseteq \text{stable}(OUT)_i.$$

Proof: If Equation 10.6 holds, then

$$t_i + \max\{t_{pd}, pd(IN)\} + pd(\lambda) \leq t_{i+1} - setup(OUT).$$

If Equation 10.7 holds, then

$$t_{i+1} + hold(OUT) \leq t_{i+1} + \min\{t_{cont}, cont(IN)\} + cont(\lambda).$$

By Equation 10.4 it follows that

$$[t_{i+1} - setup(OUT), t_{i+1} + hold(OUT)] \subseteq stable(OUT)_i.$$

□

The following claim provides a sufficient condition so that the flip-flop's input is stable during the critical segments.

Claim 10.3 *If*

$$[t_i + t_{\text{pd}}, t_{i+1} + t_{\text{cont}}] \subseteq \text{stable}(S)_i$$
$$\max\{t_{\text{pd}}, \text{pd}(IN)\} + \text{pd}(\delta) + t_{\text{su}} \leq t_{i+1} - t_i \tag{10.8}$$
$$t_{\text{hold}} \leq \min\{t_{\text{cont}}, \text{cont}(IN)\} + \text{cont}(\delta), \tag{10.9}$$

then the signal NS is stable during the critical segment C_{i+1}.

Proof: The conditions together with Equation 10.5 imply that the critical segment $C_{i+1} \subseteq$ *stable*$(NS)_i$. □

Note that, in the proof of Claim 10.3, we showed that the critical segment corresponding to clock cycle $i+1$ (i.e. C_{i+1}) is contained in the stability interval of NS corresponding to cycle i (i.e. *stable*$(NS)_i$).

Corollary 10.4 *Assume that Equation 10.3 holds with respect to $i = 0$. Assume that Equations 10.6,10.7,10.8, and 10.9 hold. Then (i) the timing constraints of NS and OUT hold with respect to every clock cycle $i \geq 0$, and (ii) Equation 10.3 holds for every $i \geq 0$.*

Proof: The proof is by induction on i. The induction basis for $i = 0$ follows from Claims 10.2 and 10.3. The induction step for $i+1$ is proved is follows. Since NS is stable during C_{i+1}, it follows that Equation 10.3 hold for $i+1$. We then apply Claims 10.2 and 10.3 to show that NS and OUT satisfy the timing constraints with respect to clock cycle $i+1$. □

We point out that we are left with the assumption that the flip-flop is properly initialized so that S satisfies Equation 10.3 with respect to $i = 0$. We deal with issue of initialization in Section 10.2.6.

10.2.4 Satisfying the timing constraints

Our goal is to simplify the conditions in Claims 10.2 and 10.3 so that they are reduced to lower bounds on the clock period. This will guarantee well defined functionality provided that the clock period is large enough.

We first address the conditions expressed in Claim 10.2. Equation 10.6 is a lower bound on the clock period. However, Equation 10.7 may not hold. This is a serious problem that can lead to failure to meet the timing constraint of OUT.

We would like to argue that, under reasonable circumstances, Equation 10.7 does hold. In a typical situation the signal IN is the output of a combinational circuit, all the inputs of which are outputs of flip-flops. Assume, for simplicity, that all the flip-flops are identical. It follows that $cont(IN) \geq t_{cont}$. By the definition of the contamination delay of a combinational circuit it follows that $cont(\lambda) \geq 0$. Hence the right hand side of Equation 10.7 is at least t_{cont}. Similarly, the signal OUT feeds a combinational circuit that feeds a flip-flop. Hence $hold(OUT) \leq t_{hold}$. Since $t_{hold} < t_{cont}$, it follows that, under these assumptions, Equation 10.7 holds.

We now address the conditions expressed in Claim 10.3. Equation 10.8 sets a lower bound on $\varphi(\textsc{clk})$. Equation 10.9, like Equation 10.7, might not hold. However, under the same assumptions used for Equation 10.7, the right hand side of Equation 10.9 is at least t_{cont}. Since $t_{hold} < t_{cont}$, it follows that, under these assumptions, Equation 10.9 holds.

We summarize this discussion with the following claim.

Claim 10.5 *Assume that* $\text{cont}(IN) \geq t_{\text{cont}}$ *and* $\text{hold}(OUT) \leq t_{\text{hold}}$. *If*

$$\varphi(\textsc{clk}) \geq \max\{t_{\text{pd}}, \text{pd}(IN)\} + \max\{\text{pd}(\lambda) + \text{setup}(OUT), \text{pd}(\delta) + t_{\text{su}}\}$$

and Equation 10.3 holds with respect to $i = 0$, then the timing constraints of NS and OUT hold with respect to every clock cycle $i \geq 0$, and Equation 10.3 holds for every $i \geq 0$.

10.2.5 Minimum clock period

We now define the minimum clock period.

Definition 10.2 *The* minimum clock period *of a synchronous circuit C is the shortest clock period for which the timing constraints of the output signals and signals that feed the flip-flops are satisfied.*

We denote the minimum clock period of a synchronous circuit by $\varphi^*(C)$.

Claim 10.5 deals with a synchronous circuit in canonic form, and states that, under reasonable conditions,

$$\varphi^*(C) = \max\{t_{pd}, pd(IN)\} + \max\{pd(\lambda), pd(\delta)\} + t_{su}. \tag{10.10}$$

The timing analysis of synchronous circuits in canonic form is overly pessimistic. The problem is that each of the combinational circuits λ and δ is regarded as a "gate" with a propagation delay. In practice it may be the case, for example, that the accumulated delay from the input IN to the output OUT is significantly different than the accumulated delay from S to the output OUT. The situation is even somewhat more complicated in the case of multi-bit signals. In the next section we deal with the general case.

10.2.6 Initialization

Meeting the timing constraints relies on the circuit being properly initialized. Specifically, we require that

$$[t_0 + t_{pd}, t_1 + t_{cont}] \subseteq \mathit{stable}(S)_0. \tag{10.11}$$

If we consider the state of a circuit (shortly) after power is turned on, then the definition of a flip-flop does not guarantee anything about the values of S (the flip-flop's output).

The natural solution to the problem of initialization is to introduce a reset signal. There are other situations where resetting the circuit is desirable. For example, a human user presses a reset button or the operating system decides to reset the system. However, the situation after power-up is more complicated.

Here we are confronted with a boot-strapping problem: How is a reset signal generated? Why does a reset signal differ from the signal S? After all, the reset signal might be stuck in a non-logical value. How do we guarantee that the reset signal is stable for a long enough period so that it can be used to initialize the circuit?

Not surprisingly, there is no solution to this problem within the digital abstraction. The reason is that a circuit attempting to generate a reset signal (or any digital signal) may be in a meta-stable state. All we can try to do is reduce the probability of such an event.

We have already discussed two methods to reduce the probability of meta-stability: (i) allow slow decisions and (ii) increase the "slope" (i.e. the derivative of the energy). Slowing down the decision is achieved by using a slow clock (e.g. clock period of 10^{-3} seconds) in the circuit that generates the reset signal. Increasing the slope is achieved by cascading (i.e connecting in series) edge-triggered flip-flops. In practice, a special circuit, often called a reset controller, generates a reset signal that is guaranteed to be stable during the critical segment of the flip-flop. We then use a flip-flop with a reset signal as depicted in Figure 10.4.[1]

[1]We must take into account the possibility that the signal NS is not logical or stable during reset. The implementation of the MUX that selects between the initial state (a constant string) and NS should be such that if $\mathit{reset} = 1$, then the MUX outputs the initial state even if the input NS is not logical. Implementation based on drivers has this property, while implementation based on combinational gates may not have this property.

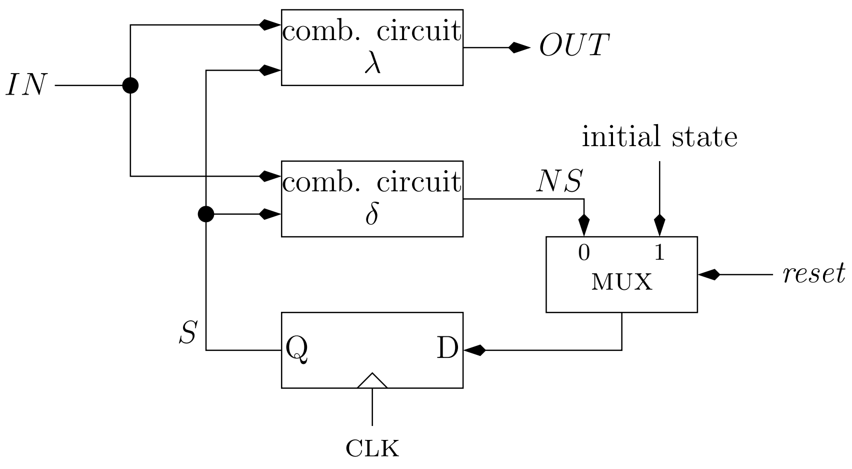


Figure 10.4: A synchronous circuit in canonic form with reset.

10.2.7 Functionality

We denote the logical value of a signal X during the stability interval $\mathit{stable}(X)_i$ by X_i. The following corollary implies that if the clock period is sufficiently large, then the functionality of the circuit is well defined. As in the case of combinational circuits (i.e. the Simulation Theorem of Combinational Circuits), we are able to derive well-defined functionality from syntax.

Corollary 10.6 *Under the premises of Claim 10.5, the following relations hold for every $i \geq 0$:*

$$\begin{aligned} NS_i &= \delta(IN_i, S_i) \\ OUT_i &= \lambda(IN_i, S_i) \\ S_{i+1} &= NS_i. \end{aligned}$$

Question 10.3 *Prove Corollary 10.6.*

Finite State Machines. We now show that Corollary 10.6 states that synchronous circuits implement *finite state machines.*

Definition 10.3 *A* finite state machine *(FSM) is a* 6*-tuple* $\mathcal{A} = \langle Q, \Sigma, \Delta, \delta, \lambda, q_0 \rangle$*, where*

- Q *is a set of* states.
- Σ *is the alphabet of the input.*
- Δ *is the alphabet of the output.*
- $\delta : Q \times \Sigma \to Q$ *is a* transition function.
- $\lambda : Q \times \Sigma \to Q$ *is an* output function.

- $q_0 \in Q$ *is an* initial state*.*

Other terms for a finite state machine are a *finite automaton with outputs* and *transducer*. In the literature, an FSM according to Definition 10.3 is often called a *Mealy Machine*. Another type of machine, called *Moore Machine*, is an FSM in which the output function λ is only a function of the state and does not depend on the input.

An FSM is an abstract machine that operates as follows. The input is a sequence $\{x_i\}_{i=0}^{n-1}$ of symbols over the alphabet Σ. The output is a sequence $\{y_i\}_{i=0}^{n-1}$ of symbols over the alphabet Δ. An FSM transitions through the sequence of states $\{q_i\}_{i=0}^{n}$. The state q_i is defined recursively as follows:

$$q_{i+1} \triangleq \delta(q_i, x_i)$$

The output y_i is defined as follows:

$$y_i \triangleq \lambda(q_i, x_i).$$

State Diagrams. FSMs are often depicted using state diagrams.

Definition 10.4 *The* state diagram *corresponding to an FSM $\mathcal{A}$ is a directed graph $G = (V, E)$ with edge labels $(x, y) \in \Sigma \times \Delta$. The vertex set V equals the state set S. The edge set E is defined by*

$$E \triangleq \{(q, \delta(q, x)) : q \in Q \text{ and } x \in \Sigma\}.$$

An edge $(q, \delta(q, x))$ is labeled $(x, \lambda(q, x))$.

The vertex q_0 corresponding to the initial state of an FSM is usually marked in an FSM by a double circle. Figure 10.5 depicts a state diagram of an FSM that outputs y if the weight of the input so far is divisible by 4, and n otherwise.

10.3 Timing analysis: the general case

In this section we present the timing constraints of a synchronous circuit that is not in canonic form. We present an algorithm that, given a synchronous circuit C, computes the minimum clock period $\varphi^*(C)$. We also present an algorithm that decides whether the timing constraints are feasible (i.e. whether there exists a minimum clock period); the conditions used by this algorithm are less restrictive than the conditions used in Claim 10.5.

10.3.1 Timing constraints

The timing constraints of general synchronous circuits are identical to those of the canonic form. For completeness, we list them below:

Input constraints: For every input signal IN, it is guaranteed that the stability intervals of IN satisfy:

$$\forall i \geq 0 : \quad [t_i + pd(IN), t_{i+1} + cont(IN)] \subseteq stable(IN)_i. \tag{10.12}$$

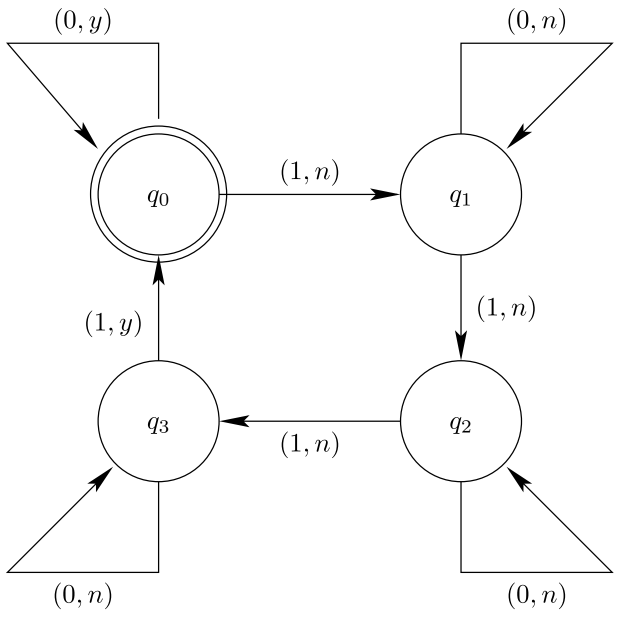


Figure 10.5: A state diagram of an FSM that counts $\text{mod}(\cdot, 4)$.

Output constraints: For every output signal OUT, it is required that the stability intervals of OUT satisfy:

$$\forall i \geq 0 : \quad [t_{i+1} - \mathit{setup}(OUT), t_{i+1} + \mathit{hold}(OUT)] \subseteq \mathit{stable}(OUT)_i. \tag{10.13}$$

Critical segments: For every signal NS that feeds a D-port of a flip-flop, it is required that NS is stable during the critical segments, namely:

$$C_{i+1} \subseteq \mathit{stable}(NS)_i. \tag{10.14}$$

10.3.2 Algorithm: minimum clock period

We now present an algorithm that computes the minimum clock period of a synchronous circuit. The correctness of the algorithm is based on the same assumptions used in Claim 10.5 to insure that the timing constraints are feasible.

The input of the algorithm consists of (i) a description of the circuit C, (ii) $\mathit{pd}(IN)$ for every input signal IN, (iii) $\mathit{setup}(OUT)$ for every output signal OUT, and (iv) a propagation delay $\mathit{pd}(v)$ for every combinational gate v. For simplicity, we assume that all the flips-flops are identical and have the same parameters (i.e $t_{su}, t_{hold}, t_{cont}, t_{pd}$).

The algorithm proceeds as follows:

1. Let C' denote the combinational circuit $C' = \langle \mathcal{G}, \mathcal{N} \rangle$ obtained from C by deleting the clock net and replacing each flip-flop with a pair of input/output gates. (See Definition 10.1.)

2. Attach a delay $d(v)$ to every input gate v of C' as follows:

$$d(v) \triangleq \begin{cases} pd(IN) & \text{if } v \text{ is an input gate of } C \text{ and } v \text{ feeds the input signal } IN \\ t_{pd} & \text{if } v \text{ corresponds to a } Q\text{-port of a flip-flop.} \end{cases}$$

3. Attach a delay $d(v)$ to every output gate v of C as follows:

$$d(v) \triangleq \begin{cases} setup(OUT) & \text{if } v \text{ is an output gate of } C \text{ and } v \text{ is fed by the output signal } OUT \\ t_{su} & \text{if } v \text{ corresponds to a } D\text{-port of a flip-flop.} \end{cases}$$

4. Attach a delay $d(v) = pd(v)$ to every combinational gate v of C.

5. Let $DG(C')$ denote the directed acyclic graph (DAG) that corresponds to C'. Let p' denote the longest path in $DG(C')$ with respect to the delays $d(v)$. Return $\varphi^*(C) = d(p')$.

The algorithm reduces the problem of computing the minimum clock period to the problem of computing a longest path in a DAG. Since a longest path in a DAG is computable in linear time, the algorithm runs in linear time as well.

10.3.3 Algorithm: correctness

Our goal is to prove that the algorithm computes the minimum clock period. Since the minimum clock period does not always exist, we need to add some conditions to guarantee its existence.

We first define delays $c(v)$ to non-sink vertices in $DG(C')$ as follows.

$$c(v) \triangleq \begin{cases} cont(IN) & \text{if } v \text{ is an input gate of } C \text{ and } v \text{ feeds the input signal } IN. \\ t_{cont} & \text{if } v \text{ corresponds to a } Q\text{-port of a flip-flop.} \\ cont(v) & \text{if } v \text{ is a combinational gate in } C. \end{cases}$$

We do not assign delays $c(v)$ to sinks because we do not need them in the following lemma that proves when signals are stable in C.

Lemma 10.7 *Consider a combinational gate, an input gate, or a flip-flop v in the synchronous circuit C. Let $\mathcal{P}_v$ denote the set of all directed paths in the directed acyclic graph $DG(C')$ that begin at a source and end in v. If the output of every flip-flop is stable in the interval $[t_i + t_{\text{pd}}, t_{i+1} + t_{\text{cont}}]$, then every output N of v satisfies*

$$[t_i + \max_{p \in \mathcal{P}_v} d(p), t_{i+1} + \min_{p \in \mathcal{P}_v} c(p)] \subseteq \text{stable}(N)_i. \tag{10.15}$$

Proof: Let $\{v_0, \ldots, v_{n-1}\}$ denote an ordering of the vertices of $DG(C')$ in topological order. Let $v = v_j$. We prove Equation 10.15 by induction on j. The induction basis, for $j = 0$, has two cases: (i) If v is an input gate, then Equation 10.15 is simply the input constraint corresponding to v. (ii) If v is a flip-flop, then the induction basis is exactly the assumption on the output of flip-flops.

The induction step is proved as follows. If v is an input gate or a flip flop, then the proof is identical to the proof of the induction basis. We are left to deal with the case that v is a combinational gate. If every input N' of v_{j+1} satisfies Equation 10.15, then every output N of v_{j+1} satisfies: (i) N becomes stable at most $d(v_{j+1})$ time units after its last input becomes stable, and (ii) N remains stable at least $c(v_{j+1})$ time units after its first input becomes instable. The lemma follows. □

The following claim and corollary proves that the clock period $\varphi^*(\text{CLK})$ computed by the algorithm is indeed the minimum clock period.

The following claim shows that if the outputs of the flip-flops are stable during the ith clock cycle, then clock period computed by the algorithm is the smallest clock period that insures that the timing constraints of the ith clock cycle are satisfied.

Claim 10.8 *Suppose that: (i) for every signal fed by a Q-port of a flip-flop,* $[t_i + t_{\text{pd}}, t_{i+1} + t_{\text{cont}}] \subseteq \text{stable}(S)_i$*, (ii) for every input* IN*,* $\text{cont}(IN) \geq t_{\text{cont}}$*, and (iii) for every output* OUT*,* $\text{hold}(OUT) \leq t_{\text{hold}}$*. Then,*

1. *For every clock period* $\varphi(\text{CLK}) \geq \varphi^*(\text{CLK})$*, the signals feeding D-ports of flip-flops are are stable during the critical segment* C_{i+1}*.*

2. *For every clock period* $\varphi(\text{CLK}) \geq \varphi^*(\text{CLK})$*, the output timing constraints corresponding to cycle* i *are satisfied.*

3. *For every clock period* $\varphi(\text{CLK}) < \varphi^*(\text{CLK})$*, a violation of the timing constraints is possible.*

Proof: Assume that $\varphi(\text{CLK}) \geq \varphi^*(\text{CLK})$. We first prove that the signals feeding the D-ports are stable during the critical segments. Consider a D-port of a flip-flop v that is fed by u. By Lemma 10.7, the output of u (i.e. the signal feeding the D-port of v) is stable during the interval

$$[t_i + \max_{p \in \mathcal{P}_u} d(p), t_{i+1} + \min_{p \in \mathcal{P}_u} c(p)].$$

Since

$$\begin{aligned}\varphi(\text{CLK}) &\geq \max_{p \in \mathcal{P}_v} d(p) \\ &= d(v) + \max_{p \in \mathcal{P}_u} d(p)\end{aligned}$$

and $d(v) = t_{su}$, we conclude that

$$t_{i+1} - t_i = \varphi(\text{CLK}) \geq t_{su} + \max_{p \in \mathcal{P}_u} d(p).$$

This implies that the signal feeding the D-port of v is stable starting at

$$t_i + \max_{p \in \mathcal{P}_u} d(p) \leq t_{i+1} - t_{su}. \tag{10.16}$$

Hence, the setup-time constraint is satisfied.

The signal feeding the D-port of v is stable until $t_{i+1} + \min_{p \in \mathcal{P}_u} c(p)$. However, every path $p \in \mathcal{P}_u$ begins at a source. A source may correspond to an input gate in C or a Q-port of a flip flop. Since $cont(IN) \geq t_{cont}$, we conclude that $c(s) \geq t_{cont}$, for every source s. It follows that

$$\min_{p \in \mathcal{P}_u} c(p) \geq t_{cont} > t_{hold}. \tag{10.17}$$

Hence the signal feeding the D-port of v is stable during the critical segment C_i, as required.

We now prove that the output constraints are satisfied. Let OUT denote an output gate. The same argumentation as in Equation 10.16 gives

$$t_i + \max_{p \in \mathcal{P}_u} d(p) \leq t_{i+1} - setup(OUT).$$

The same argumentation as in Equation 10.16 gives $\min_{p \in \mathcal{P}_u} c(p) > t_{hold}$. Since $hold(OUT) \leq t_{hold}$, it follows that the output constraints are satisfied as well.

To prove the second part, assume that $\varphi(\text{CLK}) < \varphi^*(\text{CLK})$. Let p denote a longest path in $DG(C')$ with respect to lengths $d(v)$. Without loss of generality, p begins at a source and ends in a sink v. Let p' denote the path obtained from p by omitting the sink v. It follows that

$$t_i + d(p') > t_{i+1} - d(v).$$

If the actual propagation delays along p are maximal, then the signal feeding v is not stable at time $t_{i+1} - d(v)$. If v is a flip-flop, then its input is not stable during the critical segment. If v is an output gate, then its input does not meet the output constraint. The claim follows.

□

The following corollary shows that if the circuit is properly initialized, then the clock period computed by the algorithm is the shortest clock period that satisfies all the timing constraints for all clock cycles i, for $i \geq 0$.

Corollary 10.9 *Suppose that: (i) for every signal S fed by a Q-port of a flip-flop, $[t_0 + t_{\text{pd}}, t_1 + t_{\text{cont}}] \subseteq \text{stable}(S)_0$, (ii) for every input IN, $\text{cont}(IN) \geq t_{\text{cont}}$, and (iii) for every output OUT, $\text{hold}(OUT) \leq t_{\text{hold}}$. Then,*

1. *For every clock period $\varphi(\text{CLK}) \geq \varphi^*(\text{CLK})$, the signals feeding D-ports of flip-flops are are stable during every critical segment C_{i+1}, for $i \geq 0$.*
2. *For every clock period $\varphi(\text{CLK}) \geq \varphi^*(\text{CLK})$, the output timing constraints corresponding to cycle i are satisfied, for every $i \geq 0$.*
3. *For every clock period $\varphi(\text{CLK}) < \varphi^*(\text{CLK})$, a violation of the timing constraints is possible.*

Proof: Proof is by induction on the clock cycle i. □

10.3.4 Algorithm: feasibility of timing constraints

In Claim 10.8 reasonable assumptions are made so that it is guaranteed that a minimum clock period exists. It is possible that these assumptions do not hold although the timing constraints are feasible. In this section we present an algorithm that verifies whether the timing constraints are feasible.

Lemma 10.7 gives a recipe for checking the feasibility of the timing constraints. For every non-sink v in C', the guaranteed stability interval of the signals that are output by v is:

$$[t_i + \max_{p \in \mathcal{P}_v} d(p), t_{i+1} + \min_{p \in \mathcal{P}_v} c(p)].$$

The algorithm for computing $\varphi^*(C)$ deals with making sure that each such interval does not start too late (e.g. if the signal feeds a flip-flop, then it is stable not later than $t_{i+1} - t_{su}$). We need to show that each such interval does not end too early. Namely, we need to show that:

1. For every u that feeds a D-port of a flip-flop, require
$$\min_{p \in \mathcal{P}_u} c(p) \geq t_{hold}. \tag{10.18}$$
2. For every u that generates an output signal OUT, require
$$\min_{p \in \mathcal{P}_u} c(p) \geq hold(OUT). \tag{10.19}$$

If Equations 10.18 and 10.19 do not hold, then the timing constraints are infeasible, and a minimum clock period does not exist. If these equations hold, then we may omit the assumptions used in Claim 10.8.

Lemma 10.7 also suggests an algorithm to check whether Equations 10.18 and 10.19 hold. The algorithm simply computes for every non-sink $v \in DG(C')$ the value $\min_{p \in \mathcal{P}_v} c(p)$. This is simply computing a shortest path in a DAG and can be done in linear time (e.g. using depth first search). After these values are computed for all the non-sinks, the algorithm simply checks Equation 10.18 for every D-port and Equation 10.19 for every output. If a violation is found, then the timing constraints are infeasible.

10.4 Functionality

We started with a syntactic definition of a synchronous circuit. We then attached timing constraints to the inputs and outputs of synchronous circuit. For a given synchronous circuit C with input/output timing constraints, we differentiate between two cases:

- The timing constraints are infeasible. If this happens, then one cannot guarantee well defined functionality of C. For example, if the timing constraints are not met, then inputs of flip-flops might not be stable during the critical segments, and then the flip-flop output is not guaranteed to be even logical.

- The timing constraints are feasible. If this happens, then we know that the functionality is well defined provided that the clock period satisfies $\varphi(\text{CLK}) \geq \varphi^*(\text{CLK})$.

In this section we deal with the functionality of synchronous circuits when the timing constraints are feasible. We present a trivial timing model called the *zero delay model.* In this model, time is discrete and in each clock cycle, the circuit is reduced to a combinational circuit. The advantage of this model is that it decouples timing issues from functionality and enables simple logical simulations.

10.4.1 The zero delay model

In the zero delay model we assume that all the parameters of all the components are zero (i.e. $t_{su} = t_{hold} = t_{cont} = t_{pd} = 0$, $pd(IN) = cont(IN) = setup(OUT) = hold(OUT) = 0$, and $d(G) = 0$, for every combinational gate G). Under this unrealistic assumption, the timing constraints are feasible.

By Lemma 10.7, it follows that, in the zero delay model, the stability interval of every signal is $[t_i, t_{i+1})$. Following Corollary 10.6, we conclude that, for every signal X, X_i is well defined (recall, that X_i equals $dig(X)$ during the interval $stable(X)_i$).

10.4.2 Simulation

Simulation of synchronous circuit during cycles $i = 0, \ldots, n-1$ in the zero propagation model proceeds as follows:

We assume that the flip-flops are initialized. Let S_0 denote the ordered set of values initially stored by the flip-flops.

1. Construct the combinational circuit C' that corresponds to C.

2. For $i = 0$ to $n-1$ do:

   (a) Simulate the combinational circuit C' with input values corresponding to S_i and IN_i. Namely, every input gate in C feeds a value according to IN_0, and every Q-port of a flip-flop feeds a value according to S_0

   (b) For every output OUT, let y denote the value that is fed to y. We set $OUT_i = y$.

   (c) For every D-port NS of a flip-flop, let y denote the value that is fed to the flip-flop. We set $NS_i = y$.

   (d) For every Q-port S of a flip-flop, define $S_{i+1} \leftarrow NS_i$, where NS denotes the D-port of the flip-flop.

10.5 Summary

In this chapter we started by defining synchronous circuits. We then turned to analyze synchronous circuits in canonic form. Every synchronous circuits can be transformed to the canonic form. However, from a timing point of view, the canonic form is not the most general.

Our analysis of the canonic form included the definition of timing constraints and the formulation of sufficient conditions for satisfying the timing constraints. These conditions are simplified by relying on the assumption that the input originates from a flip-flop and the output is eventually fed to a flip-flop.

We defined the minimum clock period of a synchronous circuit. The minimum clock period exists only if the timing constraints are feasible. We then turned to the issue of initializing the values stored in the flip-flops so that the computation of the synchronous circuit could properly begin. We then showed that a synchronous circuit in canonic form with feasible time constraints implements a finite state machine.

In Section 10.3 we turned to the more general case of synchronous circuit that are not in canonic form. Two algorithms are presented. The first algorithm works under the assumption that the timing constraints are feasible. The algorithm computes the minimum clock period. The second algorithm verifies whether the timing constraints are feasible. Both algorithms are very simple and run in linear time.

Finally, in Section 10.4, we present the zero delay model. In this simplified delay model, time is discrete and functionality is decoupled from timing issues. We conclude with a simulation algorithm that works under the zero delay model.