Introduction to Digital Computers

Lecture Notes

by

Guy Even

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

Spring 2002

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

Printed on June 23, 2002

Contents

Chapter 1

The digital abstraction

Digital circuits, implemented by electronic devices, operate in the *real* world. What is the connection between input and output voltages of devices and the binary (i.e. zero-one) interpretation of these voltages?

In this chapter we present a model called the digital abstraction. This model enables one to interpret 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 combine circuits together. The digital model together with a simple set of *design rules* (used to guarantee the validity of the digital abstraction) enable logic designers to design complex digital circuits consisting of millions of gates.

1.1 From analog signals to digital signals

An *analog signal* is a real function $f : \mathbb{R} \to \mathbb{R}$ 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. Under these assumptions it follows that the voltage along a wire is identical at all times. Since a signal describes the voltage, we also assume that a signal is a continuous function.

A *digital signal* is a function $g : \mathbb{R} \to \{0, 1, \text{non-logical}\}$. 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 is 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 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 several 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 useless.

Returning to the example of weighing weights, assume that we have two types of objects: light and heavy. The weight of a light (heavy) object is at most (at least) w_0 (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 (tolerance of threshold values and the desire to have a gap between values interpreted as logical zero and logical one). We denote these threshold by V_{low} and V_{high}, and require that $V_{low} < V_{high}$. An interpretation of an analog signal is depicted in Figure 1.1. Consider an analog signal $f(t)$. The digital signal $dig(f(t))$ is defined as follows.

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

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 that are sufficiently far away from each other, fluctuations of $dig(f(t))$ are not between 0 and 1, but 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 since a non-logical value is an indication that 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

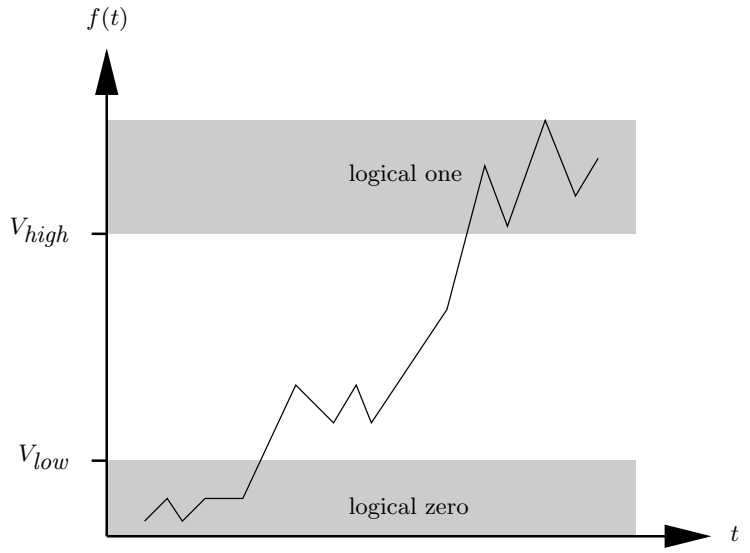


Figure 1.1: A digital interpretation of an analog signal in the zero-noise model.

consider the effect of noise, we formulate the static functionality of a gate, namely, the limit values of its output.

Question 1 *Try to define an inverter.*

1.2 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. The value of the signal $y(t)$ at time t is a function of the signal x in the interval $(-\infty, t]$.

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 one equilibrium 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. (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 consume energy even when the input is stable. We point out that in CMOS (a VLSI technology) it is easy to design circuits that do not consume

energy if the input is logical. (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". Again, it is easy to avoid such devices in CMOS technology, and even though such devices are highly useful, they are not "good" candidates as gates.) 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 *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 $f : \mathbb{R} \to \mathbb{R}$ 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 that, for every $\epsilon > 0$, there exists an $\delta > 0$, such that $|x(t) - x_0| \leq \delta$ implies $|f(x(t)) - f(x_0)| \leq \epsilon$. Therefore, the definition of a static transfer function does allow perturbations, if we also allow perturbations of the output $y(t)$.

3. We should also allow the output $y(t)$ to perturb. A reasonable modification is to require uniform convergence of $y(t)$ to $f(x_0)$. Namely, for every $\epsilon > 0$, there exist an $\delta > 0$ and a $\Delta > 0$, such that

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

4. Uniform convergence means that Δ does not depend on x_0 (although it may depend on ϵ). Typically, we are interested on the vlaues 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 2 *Extend Definition 1 to gates with x inputs and y 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 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 (one) during an interval $[t_1, t_2]$ of length at least Δ, then the logical value of the output y is one (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 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.

1.3 The bounded-noise model

Consider a wire from point A to point B. Let $A(t)$ ($B(t)$) denote the analog signal measured at point A (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 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 different 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 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.

1.4 The need for changing the digital abstraction in presence of noise

Consider two inverters, where the output of one gate feeds the input of the second gate. Figure 1.2 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 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 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 effected 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.

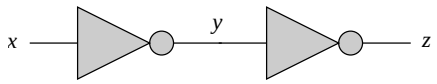


Figure 1.2: Two inverters connected in series.

1.4.1 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 1.3 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.

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 $V_{low,out}$.

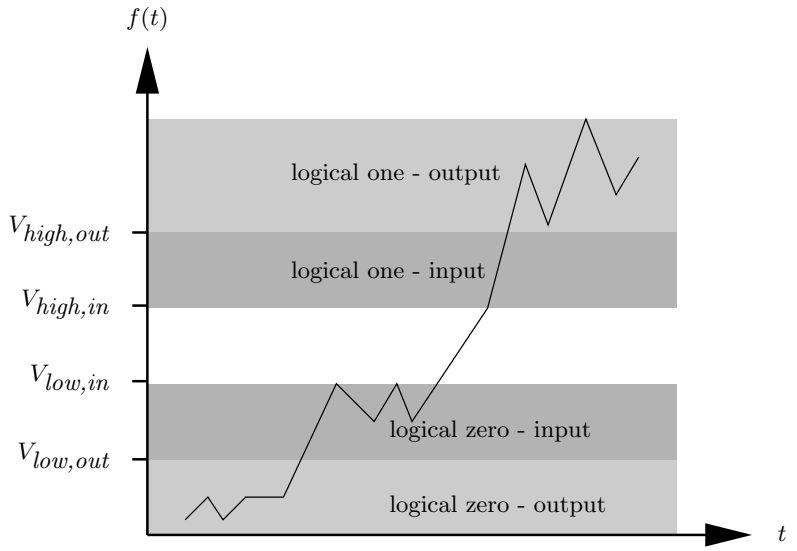


Figure 1.3: A digital interpretation of an input and output signals.

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 chain the logical values of signals.

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

Definition 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 4 *Define a* NAND*-gate in the bounded-noise model.*

Question 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 3?*

Question 6 *Consider a function $f : [0, 1] \to [0, 1]$. Suppose that $f(x)$ is monotone decreasing and that the derivative $f'(x)$ of $f(x)$ satisfies the following condition: $f'(x)$ is continuous, $f'(x)$ is strictly decreasing in the interval $[0, \alpha]$ and strictly increasing in the interval $(\alpha, 1]$. Moreover, $f'(\alpha) < -1$. 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 3?*

Question 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 so that Definition 3 is satisfied.*

1.5 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 *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 $t \in [t_1, t_2]$.

1.6 Summary

In this chapter transfer functions of gates were defined. Static transfer functions describe the output voltages of gates when the inputs are stable for a sufficiently long interval. The amount of time that input signals must be stable to guarantee stability of the output of a gate is called the propagation delay.

In the digital abstraction, analog signals are interpreted either as zero, one, or non-logical. We started with the zero-noise model in which there is no noise, and defined the digital abstraction in this model. We gave an example of a circuit in which even small amounts of noise corrupt the functionality of the circuit. We then defined the bounded-noise model. We re-defined the digital abstraction under the bounded-noise model.

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 4 *A function $f : \{0,1\}^n \to \{0,1\}^k$ 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*. 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 5 *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 6 *A gate G implements a Boolean function $f : \{0,1\}^2 \to \{0,1\}$ 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*

$$y(t) = f(\sigma_1, \sigma_2),$$

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

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 we will make the most "pessimistic" assumption. 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.

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 6 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 a gate with an input of a 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. A *net* is a subset 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 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.

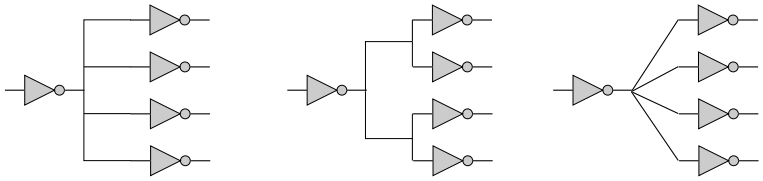


Figure 2.1: Three equivalent nets.

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 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 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, output terminals are connected via resistors either the the ground or the 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*.

Definition 7 *A net N is* simple *if (a) N is fed by exactly one output terminal, and (b) N feeds at least one input terminal.*

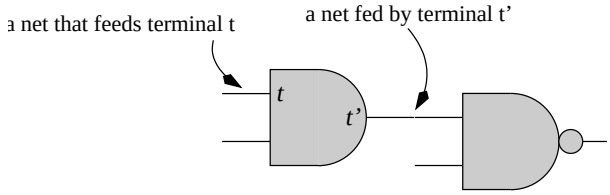


Figure 2.2: A terminal that is fed by a net and a terminal that feeds a net.

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 $t \to t_i$.

It follows that a circuit, all the nets of which are simple, may be modeled by a directed graph. The vertices of the graph are the gates and the directed edges are the wires. We denote the directed graph that models a circuit C, all the nets of which are simple, by $DG(C)$.

2.4 Combinational circuits

Question 8 *Consider the circuits depicted in Figure 2.3. Can you explain why these are not valid combinational circuits?*

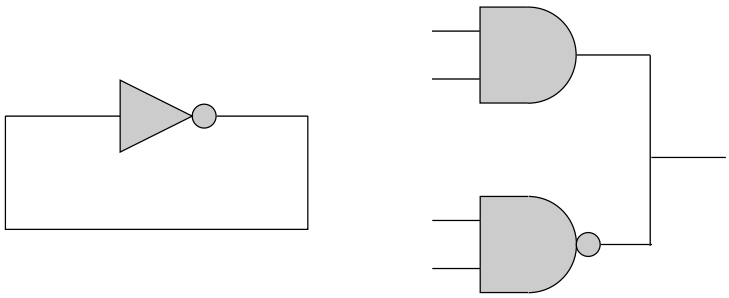


Figure 2.3: Two examples of non-combinational circuits.

Before we define combinational circuits it is helpful to define two types of special gates: an input gate and an output gate. The purpose of these gates is to avoid endpoints in nets that seem to be not connected (for example, all the nets in the circuit on the right in Figure 2.3 have endpoints that are not connected to a gate).

Definition 8 (input and output gates) *An* input gate *is a gate with zero inputs and a single input. An* output gate *is a gate with one input 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.

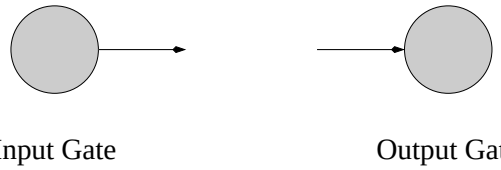


Figure 2.4: An input gate and an output gate

A circuit may contain many inverters, NAND-gates, etc. Hence, in the definition below, the set of gates $\mathcal{G}$ set is actually a multi-set (i.e. a set of repetitions). We associate with every gate $G \in \mathcal{G}$ the number of inputs, the number of outputs, and the Boolean functions that specifies its functionality.

We are now ready to present a syntactic definition of combinational circuits.

Definition 9 (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 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.*

Question 9 *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 compute the Boolean function that is implemented by a given combinational circuit.
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 the propagation delay of the circuit.

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.

Theorem 2 (Simulation theorem of combinational circuits) *Let $C = \langle \mathcal{G}, \mathcal{N} \rangle$ denote a combinational circuit that contains k input gates. Let $\{x_i\}_{i=1}^{k}$ denote the output terminals of the input gates in C. Assume that the digital signals $\{x_i(t)\}_{i=1}^{k}$ are stable during the interval $[t_1, t_2]$. Then, for every net $N \in \mathcal{N}$ there exist:*

1. *a Boolean function $f_N : \{0,1\}^k \to \{0,1\}$, 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 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 $i < j$.)

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 $\mathcal{N}_i$ precedes 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 $f_{e_i} : \{0,1\}^k \to \{0,1\}$, 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: Consider $m' = 1$. The net e_1 is fed by the gate v_1. Since the gate v_1 is not fed by any net, it follows that v_1 is an input gate. Let x_i denote the output terminal of v_1. It follows that the digital signal along e_1 always equals the digital signal $x_i(t)$. Hence we define f_{e_1} to be simply the projection on the ith component, namely $f_{e_1}(\sigma_1, \ldots, \sigma_k) = \sigma_i$. The propagation delay $t_{pd}(e_1)$ 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 (e_k) implements a Boolean function f_{e_j} (f_{e_k}) with propagation delay $t_{pd}(e_j)$ ($t_{pd}(e_k)$). 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 signals of every net in a combinational circuit is logical (if 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 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 fed the output gates of the combinational circuit. Hence, we may regard a combinational circuit as a "macro-gate". All instance 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
$$t_{pd}(e_i) \leftarrow t_{pd}(v_j) + \max\{t_{pd}(e')\}_{\{e' \text{ feeds } v_j\}}.$$

The total amount of time spend in the relaxation steps in linear, and hence the running time of this algorithm is linear.

Question 10 *Prove that the total amount of time spend in the relaxation steps in linear. 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.*

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 combinational circuit. We denote the cost of a gate G by $c(G)$.

Definition 10 *The cost of a combinational circuit $C = \langle \mathcal{G}, \mathcal{N} \rangle$ is defined by*

$$c(C) \stackrel{\triangle}{=} \sum_{G \in \mathcal{G}} c(G.$$

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

Definition 11 *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 12 *A sequence of gates $p = \{v_0, v_1, \ldots, v_k\}$ 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.

Claim 3 *The propagation delay of a combinational circuit $C = \langle \mathcal{G}, \mathcal{N} \rangle$ equals*

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

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

Question 11

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 3 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.

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

Table 2.1: Costs and delays of gates

2.6 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 combinatorial 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

Combinational modules

A combinational module is a combinational circuit that is usually part of a bigger circuit. We describe in this chapter modules that are often used in designing complex circuit such as micro-processors.

3.1 Notation

A binary string $a_0a_1 \cdots a_{n-1}$ is denoted by $a[0 : n-1]$. Indexes are often important, so we denote the string $a_{n-1}a_{n-2} \cdots a_1$ by $a[n-1 : 0]$. In VLSI-CAD tools such multiple signals are called *buses*. 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*.

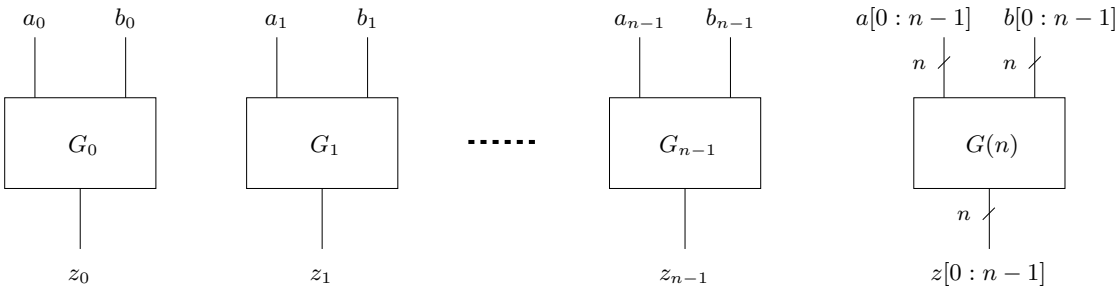
We often use the shorthand $\vec{a}$ for a binary string $a[n-1 : 0]$, provided of course that the indexes of the string $a[n-1 : 0]$ are obvious from the context.

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 3.1. 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 3.1. 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.

We often wish to feed all the second input terminals of gates in $G(n)$ with the same signal. Figure 3.2 denotes a circuit $G(n)$ in which the value b is fed to the second input terminal of all the gates.

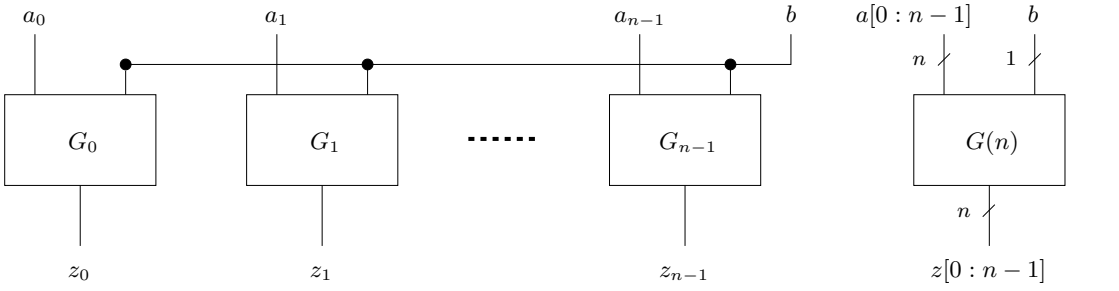
Note that the fanout of the net that carries the signal b in Figure 3.2 is linear. In practice, a large fanout increases the capacity of a net and causes an increase in the delay of the circuit. We usually ignore this phenomena in this course.

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.



(A) (B)

Figure 3.1: Vector notation: multiple instances of the same gate.



(A) (B)

Figure 3.2: Vector notation: b feeds all the gates.

Example 1
- *If* $a = 01$ *and* $b = 10$*, then* $a \cdot b = 0110$.
- *If* $a = 1$ *and* $i = 5$*, then* $a^i = 11111$.

3.2 Representation and values

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 what number is represented by a binary string.

Definition 13 *The* binary value represented by a binary string $a[n-1:0]$ *is denoted by* $\langle a[n-1:0]\rangle$*. It is defined as follows*

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

One may regard $\langle\cdot\rangle$ as a function from binary strings not including the empty string (i.e. $\{0,1\}^+$) to natural numbers (i.e. $\mathbb{N}$). The inverse function is the binary representation function.

Definition 14 *Binary representation is a function* $\mathrm{bin} : \mathbb{N} \to \{0,1\}^+$ *that is the inverse function of* $\langle\cdot\rangle$*. Namely, for every* $n \geq 1$ *and every* $a[n-1:0] \in \{0,1\}^n$*,*

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

Definition 15 *A binary string* $x[n-1:0]$ represents a number in unary representation *if* $\vec{x} = 0^* \cdot 1^*$*. Namely, the string* x *consists of a block of zeros followed by a block of zeros. The* weight *of a binary string* $x[n-1:0]$ *is the number of ones in it. We denote the weight of* $\vec{x}$ *by* $\mathrm{wt}(\vec{x})$*. Formally,*

$$\mathrm{wt}(\vec{x}) = \sum_{i=1,\ldots,n} x[i].$$

The value represented by a binary string $x[n-1:0]$ *in unary representation is* $\mathrm{wt}(\vec{x})$*, provided that* $\vec{x}$ *represents a number in unary representation.*

Remark 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-zeros string with an all-ones string represents a number in unary representation.*

3.3 Trees of associative Boolean gates

In this section we deal with combinational circuits with topologies that are trees and consist of instances of the same gate, where the gate implements an associative Boolean function. We first discuss extensions of associative Boolean functions.

3.3.1 Associative Boolean functions

Definition 16 *A Boolean function* $f : \{0,1\}^2 \to \{0,1\}$ *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 12 *List all the associative Boolean functions* $f : \{0,1\}^2 \to \{0,1\}$.

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

$$(\sigma_1 \odot \sigma_2) \odot \sigma_3 = \sigma_1 \odot (\sigma_2 \odot \sigma_3)$$

for every $\sigma_1, \sigma_2, \sigma_3 \in \{0,1\}$. This implies that one may omit parenthesis from expressions involving an associative Boolean function and simply write $\sigma_1 \odot \sigma_2 \odot \sigma_3$.

We now build Boolean functions defined over $\{0,1\}^n$ from Boolean functions defined over $\{0,1\}^2$.

Definition 17 *The function* $f_n : \{0,1\}^n \to \{0,1\}$, *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 4 *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 13 *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*

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

The implication of Question 13 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.

3.3.2 OR-trees

An OR-tree is a combinational circuit defined as follows:

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

Output: $y \in \{0,1\}$

Functionality: $y = x[0] \vee x[1] \cdots \vee x[n-1]$. (where $a \vee b$ denotes $\text{OR}(a,b)$).

Claim 4 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 3.3 depicts two OR-trees for $n = 4$.

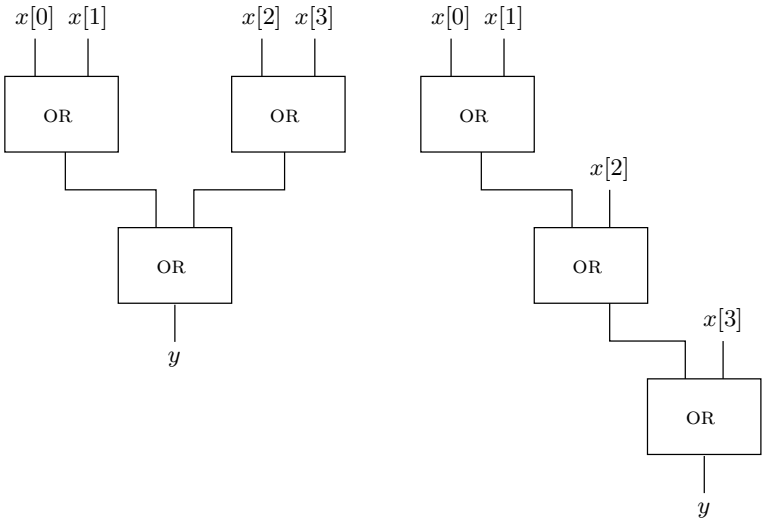


Figure 3.3: Two implementations of an OR-tree with $n = 4$ inputs.

Question 14 *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?*

3.3.3 Cost and delay analysis

You may have noticed that both OR-trees in depicted in Figure 3.3 contain 3 OR-gates. However, their delay is different. The following claim summarizes the the fact that all OR-trees have the same cost.

Claim 5 *The cost of every* OR*-tree with the topology of a rooted binary tree is* $(n-1)\cdot c(\text{OR})$.

Proof: The number of interior nodes of a binary tree equals the number of leaves minus one. □

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

Question 15 *This question deals with different ways to construct balanced trees. The goal is to achieve* $log_2 n$ *depth.*

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 exist 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* $\lceil \log_2 n \rceil$.

3.3.4 Lower bounds

In this section we deal with the following questions: What is the best choice of a topology for a combinational circuit that implements an OR-tree? Why is a tree 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.) Why is a tree the best topology?

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 an OR-tree. We will be able to prove lower bounds for circuits that implement OR-trees that will imply the optimality of using balanced trees.

Definition 18 *Let $f : \{0,1\}^n \to \{0,1\}$ denote a Boolean function. Let $\sigma \in \{0,1\}$. The Boolean function $g : \{0,1\}^{n-1} \to$ defined by*

$$g(x_0, \ldots, x_{n-2}) = f(x_0, \ldots, x_{i-1}, \sigma, x_i, \ldots, x_{n-2})$$

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

Definition 19 *A boolean function $f : \{0,1\}^n \to \{0,1\}$* depends *on its ith input if*

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

The set of input indexes that a Boolean function f depends on is called the *cone*. The cone of f is denoted by *cone*(f).
A Boolean function $f : \{0,1\}^n \to \{0,1\}^m$ is a vector of m Boolean functions $f_i : \{0,1\}^n \to \{0,1\}$, where

$$f(\vec{x}) = (f_1(\vec{x}), \ldots, f_m(\vec{x})).$$

Example 2 *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 6 $\text{cone}(f) = \emptyset \iff f$ *is a constant Boolean function.*

Frame 1: More on Boolean functions.

Before we proceed, we need a few more definitions regarding Boolean functions. These definitions appear in Frame 1.

The following claim is trivial.

Claim 7 *The Boolean function OR_n depends on all its inputs, namely*

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

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 8 *Let $C = \langle \mathcal{G}, \mathcal{N} \rangle$ denote a combinational circuit that implements a Boolean function $f : \{0,1\}^n \to \{0,1\}$. Let $g_i \in \mathcal{G}$ denote an input gate of 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 $x \in \{0,1\}^{n-1}$ for which $f_{\restriction x_i=0}(x) \neq f_{\restriction x_i=1}(x)$. Let x' (x'') denote the extension of x to n bits by inserting a 0 (1) in the ith coordinate. The proof of the Simulation Theorem of combinational circuits (Theorem 2) implies that C outputs the same value when given the input strings x' and x'', and hence C does not implement f, a contradiction. □

The following claim shows that every circuit, that implements an OR-tree in which the fan-in of every gate is bounded by two, must contain at least $n-1$ gates. Hence, an implementation based on a tree of OR-gates is optimal up to constant factors. 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.

Theorem 9 (Linear Cost Lower Bound Theorem) *Let C denote a combinatorial circuit that implements a Boolean function f. Then*

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

Proof: Let $n = |cone(f)|$. Without loss of generality, we may assume that C has n input gates. Moreover, we may assume that there is a path from every input gate to the output gate. This assumption is justified by the following reduction: Let v denote the output gate. Mark all the gates u such that there is a path from u to v. Remove all the gates that are not marked, and leave only the nets that are connected only to marked nodes. We leave it as an exercise to show that the resulting sub-circuit C' also implements the function f. It suffices to prove that $c(C') \geq n-1$. To simplify notation, we assume that $C = C'$.

Sort the the gates of C in topological order (as in the proof of the Simulation Theorem of combinational circuits 2). Let $v_1, v_2 \ldots, v_s$ denote the gates of C sorted in topological order. We may assume that the n input gates appear first in this order, and that the output gate appears last. Namely, $v_1, \ldots, v_n$ are all the input gates and v_s is the output gate.

Define cut_i to be a set of nets defined as follows. A net that contains a port both in the prefix $\{v_1, \ldots, v_{i-1}\}$ and in the suffix $\{v_i, v_{i+1}, \ldots, v_s\}$ is said to belong to cut_i. Let n_i denote the number of nets that belong to cut_i. We say that a net is cut first (last) in cut_i if it does not belong to a cut cut_j for $j < i$ ($j > i$). Note that $n_1 = 0$ since the prefix is empty, and $n_s = 0$ because the suffix is empty.

It follows from the definition of n_{i+1} that it satisfies the following equation

$$n_{i+1} = n_i - |\{\text{nets cut last in } cut_i\}| + |\{\text{nets cut first in } cut_{i+1}\}|.$$

This implies that

$$n_{i+1} \geq \begin{cases} n_i + 1 & \text{if } v_i \text{ is an input gate or a constant} \\ n_i - 1 & \text{otherwise} \end{cases}$$

If v_i is that an input gate or a constant gate, then its in-degree is zero. It follows that a net cannot be cut last in cut_i. However, the net that is fed by v_i is cut first in cut_{i+1}. Hence $n_{i+1} = n_i + 1$.

If v_i is an output gate, then it is possible that at most one net "ends" at v_i hence a decrease of at most one in the cut size. Finally, if v_i is a "real" gate, then since its fan-in is at most two, it follows that at most two nets can be cut last in cut_i. However, the net that is fed by v_i is cut first in cut_{i+1}. Hence a net decrease of at most one in the cut size.

By summing, it follows that

$$n_s \geq n_0 + n - 1 - |\{v_i \mid v_i \text{ is a non-trivial gate}\}|$$

The reason that n input gates contribute n to n_s. The output gate v_s causes a decrease of 1, and every non-trivial gate causes a decrease of at most 1. Since $n_0 = n_s = 0$, it follows that the number of non-trivial gates is at least $n - 1$. The cost of every non-trivial gate is at least 1, and the theorem follows. □

The following corollary proves that every implementation of OR_n requires $n-1$ non-trivial gates, hence using trees is optimal with respect to cost (up to constant factors).

Corollary 10 *Let C_n denote a combinatorial circuit that implements an* OR*-tree with input length n. Then*

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

Proof: Follows directly from Claim 7 and Theorem 9. □

Question 16 *State and prove a generalization of Theorem 9 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 (e.g. the number of inputs of every gate in the design is at most 2).

The following theorem shows a lower bound on the delay of combinational circuits that is logarithmic in the size of the cone.

Theorem 11 (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 $\log_c |\text{cone}(f)|$.*

Proof: 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.

Claim 8 implies there must be a path in $DG(C)$ from every input $x_i \in cone(f)$ to the output y of C. We show that every directed acyclic graph (DAG) with k inputs (i.e. nodes with in-degree zero and out-degree one), one output (i.e. node with in-degree one and out-degree zero), and an in-degree bounded by c, must have a path containing at least $\log_c k$ interior vertices. This is a standard argument in trees, and is easily extendible to graphs. Here we prove it for a DAG.

Observe that if a DAG has exactly one output, then the output is reachable from every input. Simply, pick a maximal path that starts in an input; it can only stop at the output vertex.

The involves strengthen the claim on DAGs it in a manner similar to the strengthening that we used in the proof of the Simulation Theorem of combinational circuits (Theorem 2). We attach to every edge e in the DAG a subset of inputs $cone(e)$ which is the set of input vertices from which the edge e is reachable. Let $d(e)$ denote the maximum number of interior vertices along a path from input vertex in $cone(e)$ to e. We now prove that, for every edge e, $d(e) \geq \log_c |cone(e)|$.

We sort the vertices of the DAG in topological order. Let $e_1, e_2, \ldots$ denote the edges sorted in an order induced by the topological ordering of the vertices. We the claim on DAGs by induction on the index of an edge. The induction hypothesis is trivial since e_1 must emanate from an input vertex, hence $cone(e_1) = 1$, in which case the claim is trivial since $\log 1 = 0$. We now proceed to prove the induction step. Consider edge e_i. To simplify indexes, assume that e_i emanates from a vertex v, where $\{e_{i-c}, \ldots, e_{i-1}\}$ is the set of edges that enter v. (Exercise: modify the proof so that it holds also if the number of edges that enter v is strictly less than c.) By definition

$$cone(e_i) = \bigcup_{j=1}^{c} cone_(e_{i-j}).$$

Hence

$$|cone(e_i)| \leq \sum_{j=1}^{c} |cone_(e_{i-j})|. \tag{3.1}$$

Consider the following expansion:

$$\begin{aligned} d(e) &= 1 + \max_{j=1\ldots c}\{d(e_{i-j})\} \\ &\geq 1 + \max_{j=1\ldots c}\{\log_c |cone(e_{i-j})|\} \\ &\geq 1 + \log_c \frac{\sum_{j=1}^{c} |cone(e_{i-j})|}{c} \\ &\geq 1 + \log_c \frac{cone(e_i)}{c} \\ &= \log_c |cone(e_i)|. \end{aligned}$$

The first line follows by the definition of $d(e)$. The second line follows by the induction hypothesis. The third line follows by $\max_{j=1\ldots c}\{\log a_i\} = \log\left(\max_{j=1\ldots c}\{a_i\}\right) \geq \log \frac{\sum_{j=1}^{c} a_i}{c}$. The fourth line follows from Equation 3.1. This concludes the proof of the claim on DAGs, and the theorem follows. □

The following corollary proves a lower bound on the delay of combinational circuits that implement OR_n.

Corollary 12 *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*

$$d(C_n) \geq \lceil \log_c n \rceil .$$

Proof: The corollary follows directly from Claim 7 and Theorem 11. □

Question 17 *The proof of the Theorem 11 dealt with the longest path from an input vertex to an output vertex. In the proof, we strengthened this statement by considering edges instead of output vertices. 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.* $\max_{u \in U} \mathrm{dist}(u, r)$*).*

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 $\mathrm{dist}(u, r) \geq \Omega(\log_c |U|)$*, where c denotes the maximum degree of G. (*$\mathrm{dist}(u, r)$ *denotes the length of the shortest path from u to r.)*

3.4 Half-Decoders

In this section we deal with the design and analysis of a half-decoder. Apart from its application in implementing a logical right shifter, half-decoders serve as an example of unary representation.

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

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

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

Functionality: The output string y consists of a block of zeros followed by a block of ones. The length of the block of ones equals $\langle x[n-1:0] \rangle$. Formally,

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

The following simple claim describes an equivalent definition of the functionality of a half-decoder.

Claim 13

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

We denote a half-decoder with input length n by H-DEC(n). Figure 3.4 depicts the schematics we use to depict a half-decoder. Note that the n does not denote n instances of a circuit as in Figure 3.1; instead, n denotes the length of the input of the decoder.

Example 3 *Consider a half-decoder* H-DEC(3)*. On input $x = 101$, the output y equals* 00011111*.*

Remark 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.*

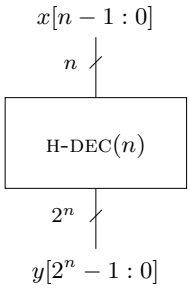


Figure 3.4: Schematic notation of H-DEC(n).

3.4.1 Implementation

We design an H-DEC(n) using recursion. We start by designing an H-DEC(n) with $n = 1$. We then proceed by designing an H-DEC(n) based on "smaller" half-decoders.

An H-DEC(1) is simply 2 wires where: $y[0] \leftarrow x[0]$ and $y[1] \leftarrow 0$. Note the use of a constant value 0 is this design. We will assume that a designer can always connect a 1 or a 0 to an input terminal, if needed.

Figure 3.5 depicts a recursive implementation of an H-DEC(n). All the inputs, except for the most-significant bit $x[n-1]$, are fed to an H-DEC($n-1$). The output of this half-decoder is denoted by $z[2^{n-1}-1:0]$. The lower half of the output is obtained or-ing the bits of $\vec{z}$ with $x[n-1]$. The upper half of the output is obtained by and-ing the bits of $\vec{z}$ with $x[n-1]$.

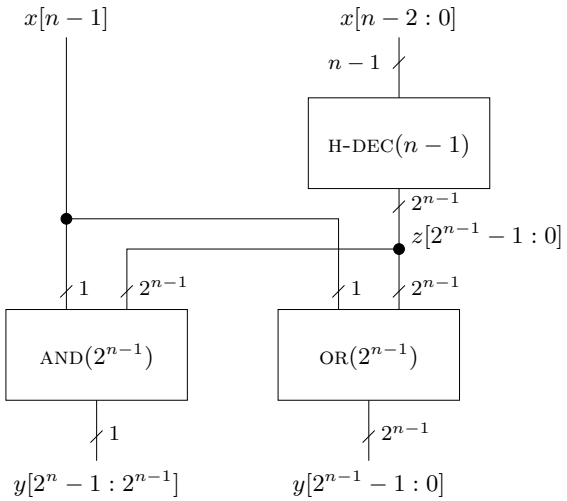


Figure 3.5: A recursive implementation of H-DEC(n).

3.4.2 Correctness

Claim 14 *The design depicted in Figure 3.5 is a correct implementation of a half-decoder.*

Proof: The proof is by induction on n. The induction basis, for $n = 1$, follows simply by considering the two possible values of $x[0]$.

We introduce the following notation:

$$\begin{array}{lll} \vec{y} = y[2^n - 1 : 0] & \vec{y}_R = y[2^{n-1} - 1 : 0] & \vec{y}_L = y[2^n - 1 : 2^{n-1}] \\ \vec{x} = x[n-1:0] & \vec{x}' = x[n-2:0] & \vec{z} = z[2^{n-1} - 1 : 0] \end{array}$$

Note that $\vec{y}_L$ ($\vec{y}_R$) corresponds to the left (right) half of $\vec{y}$.

The induction hypothesis is that the module marked as H-DEC$(n-1)$ in Figure 3.5 is correct. This means that

$$\vec{z} = 0^{2^{n-1} - \langle \vec{x}' \rangle} \cdot 1^{\langle \vec{x}' \rangle}.$$

We wish to prove that

$$\vec{y} = 0^{2^n - \langle \vec{x} \rangle} \cdot 1^{\langle \vec{x} \rangle}. \tag{3.2}$$

We consider two cases: (1) $x[n-1] = 0$, and (2) $x[n-1] = 1$.

1. $x[n-1] = 0$: Since AND$(A, 0) = 0$ and OR$(A, 0) = A$, it follows that $\vec{y}_L = 0^{2^{n-1}}$ and $\vec{y}_R = \vec{z}$. Therefore,

$$\begin{aligned} \vec{y} &= \vec{y}_L \cdot \vec{y}_R \\ &= 0^{2^{n-1}} \cdot \vec{z} \\ &= 0^{2^{n-1}} \cdot 0^{2^{n-1} - \langle \vec{x}' \rangle} \cdot 1^{\langle \vec{x}' \rangle} \\ &= 0^{2^n - \langle \vec{x}' \rangle} \cdot 1^{\langle \vec{x}' \rangle} \\ &= 0^{2^n - \langle \vec{x} \rangle} \cdot 1^{\langle \vec{x} \rangle}. \end{aligned}$$

   The third line follows from the induction hypothesis. The fourth line follows by grouping together the block of zeros (i.e. the concatenation operation over strings is associative). The fifth line follows from $\langle \vec{x} \rangle = \langle \vec{x}' \rangle$ since $x[n-1] = 0$.

2. $x[n-1] = 1$: Since AND$(A, 1) = A$ and OR$(A, 1) = 1$, it follows that $\vec{y}_L = \vec{z}$ and $\vec{y}_R = 1^{2^{n-1}}$. Therefore,

$$\begin{aligned} \vec{y} &= \vec{y}_L \cdot \vec{y}_R \\ &= \vec{z} \cdot 1^{2^{n-1}} \\ &= 0^{2^{n-1} - \langle \vec{x}' \rangle} \cdot 1^{\langle \vec{x}' \rangle} \cdot 1^{2^{n-1}} \\ &= 0^{2^n - \langle \vec{x} \rangle} \cdot 1^{\langle \vec{x} \rangle}. \end{aligned}$$

   The third line follows from the induction hypothesis. The fourth line follows by grouping together the block of zeros and applying $\langle \vec{x} \rangle = \langle \vec{x}' \rangle + 2^{n-1}$ since $x[n-1] = 1$.

We showed that in both cases Equation 3.2 holds, and hence the claim follows. □

3.4.3 Cost and delay analysis

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

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

It follows that

$$\begin{aligned} c(\text{H-DEC}(n)) &= (c(\text{AND}) + c(\text{OR})) \cdot (2^{n-1} + 2^{n-2} + \cdots + 2) \\ &= (c(\text{AND}) + c(\text{OR})) \cdot (2^n - 2) \\ &= \Theta(2^n). \end{aligned}$$

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

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

It follows that

$$\begin{aligned} d(\text{H-DEC}(n)) &= (n-2) \cdot \max\{d(\text{AND}), d(\text{OR})\} \\ &= \Theta(n). \end{aligned}$$

3.4.4 Lower bounds

Is the design of a half-decoder suggested in Section 3.4.1 a good design? What makes a design good? 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 a half-decoder. At this point, let us focus on the asymptotic cost and delay since the constants are rather small (e.g. the constant in the cost analysis is $c(\text{AND}) + c(\text{OR})$).

The following claim proves that the cost of every half-decoder design must be $\Omega(2^n)$. Claim 15 is a rather strong statement because it is not limited to a single design but to every possible design one could come up with. In fact, the claim also resolves our question about the optimality of the cost of the half-decoder design suggested in Section 3.4.1. It shows that the presented design is in fact of minimum cost (up to constant factors).

Claim 15 *Let H_n denote a combinational circuit that implements a half-decoder with input length n. Then*

$$c(H_n) \geq 2^n - 2.$$

Proof: The proof uses the following argument. Suppose we could prove the following statements:

1. $y[i]$ is not constant, for every $i \in \{0, \ldots, 2^n - 2]|$. Namely there exists an input for which $y[i] = 0$, and there exists an input for which $y[i] = 1$.

2. $y[i] \neq y[j]$, for every $i \neq j$. Namely, there exists an input for which $y[i] \neq y[j]$. This implies that the nets that feed the output gates must be distinct (we regard two nets that are fed by the same output port of the same gate as identical). Let N_i denote the net that feeds the output gate $y[i]$.

3. $y[i] \neq x[j]$, for every $i \in \{0, \ldots, 2^n - 2\}$ and every $j \in \{0, \ldots, (n-1)\}$. This means that the nets N_i cannot be fed by input gates (whose cost is zero).

It follows that every output $y[i]$ (except for $i = 2^n - 1$) must be fed by a non-trivial gate (namely not a constant and not an input gate). Moreover, the gates that feed the nets $\{N_i\}_{i=1}^{2^n-2}$ must be distinct. It follows that the circuit must contain at least $2^n - 1$ non-trivial gates. The cost of every gate is at least 1, and so the claim follows.

Statement (1) follows simply by considering the all-ones and all-zeros input strings.

We leave it as an exercise to show that, for every $i \neq j$, there exists an input $\vec{x}$ such that the corresponding output satisfies $y[i] \neq y[j]$. It follows that Statement (2) holds.

Unfortunately, Statement (3) is not quite true! We leave it to the reader to prove that $y[2^{n-1} - 1] = x[n-1]$. Luckily, this is the only "counter-example", and we prove it below. (The curious reader should now understand why the statement of the claim is that the cost is at least $2^n - 2$ and not $2^n - 1$.)

Our goal is to show that, for every i (except $i = 2^n - 1$ and $i = 2^{n-1} - 1$), $y[i]$ does not simply equal one of the input bits. We now show that, for every output $y[i]$ (except $i = 2^n - 1$ and $i = 2^{n-1} - 1$) and every $j \in \{0, \ldots, (n-1)\}$, there exist two input strings $\vec{x'}$ and $\vec{x''}$ such that

1. $x'[j] = x''[j]$, and

2. Let $\vec{y'}$ ($\vec{y''}$) denote output string corresponding to $\vec{x'}$ ($\vec{x''}$). Then $y'[i] \neq y''[i]$.

The implication is that a circuit in which $y[i]$ is fed directly by $x[j]$ is not a correct half-decoder (it would err with the input string $\vec{x'}$ or with the input string $\vec{x''}$).

Given $i \in \{0, \ldots 2^n - 2\} - \{2^{n-1}\}$, let $x_0[n-1:0] = bin(i)$ and $x_1[n-1:0] = bin(i+1)$. By Claim 13 it follows that when fed by $\vec{x_0}$ ($\vec{x_1}$), the output $y[i]$ equals 0 (1). We consider now the following three cases:

1. There exists an index $k \neq j$ such that $x_0[k] = 0$. In this case "flipping" the kth bit of $\vec{x_0}$ (from 0 to 1) causes $y[i]$ to flip to 1. Namely, Let $x'_0[n-1:0]$ be defined by $x'_0[\ell] = x_0[\ell]$ if $\ell \neq k$, and $x'_0[k] = 1$. Since $\langle \vec{x_0} \rangle \geq i + 1$, the input string $\vec{x'_0}$ produces an output for which $y[i] = 1$. Hence the binary strings $\vec{x_0}$ and $\vec{x'_0}$ are exactly what we are looking for to prove Statement (3).

2. There exists an index $k \neq j$ such that $x_1[k] = 1$. In this case "flipping" the kth bit of $\vec{x_1}$ (from 1 to 0) causes $y[i]$ to flip to 1. Again, we have found two strings as desired.

3. If neither case (1) or case (2) hold, then $\vec{x_0} = 1^{n-(j+1)} \cdot 0 \cdot 1^j$ and $\vec{x_1} = 0^{n-(j+1)} \cdot 1 \cdot 0^j$ (why?). By definition, we also know that $\langle \vec{x_0} \rangle + 1 = \langle \vec{x_1} \rangle$. Hence, $j = n - 1$ (why?). It follows that this case can occur only if $i + 1 = 2^{n-1}$, which we already ruled out.

We showed that Statement (3) holds for all nets, except $N_{2^{n-1}-1}$, and the claim follows. □

We proved a lower bound on cost of a half-decoder that matches (up to constant factors) the cost of the design suggested in Section 3.4.1. This implies that the design in Section 3.4.1 is optimal! One cannot come up with a cheaper design.

We now turn to proving a lower bound on the delay of a half-decoder. We again rely on the Logarithmic Delay Lower Bound Theorem (Theorem 11).

The following corollary proves a lower bound on the delay of half-decoders.

Corollary 16 *Let H_n denote a combinational circuit that implements a half-decoder with input length n. Let c denote the maximum fan-in of a gate in H_n. Then*

$$d(H_n) \geq \lceil \log_c n \rceil .$$

Proof: In light of Theorem 11, all we need to show is that one of the outputs $y[i]$ of the half-decoder implements a Boolean function f_i that satisfies $cone(f_i) = n$. We leave it to the reader to verify that $y[0]$ depends on all the input bits, hence, the corollary follows. □

3.5 An asymptotically optimal half-decoder design

In this section we present a half-decoder design with minimum cost and delay (up to constant factors).

3.5.1 Comparison

Claim 13 implies that in order to determine whether $y[i] = 1$, we need to tell whether $\langle \vec{x} \rangle > i$. We therefore discuss how one can conduct such a comparison efficiently.

Note that it is easy to compare the unary value represented by a binary string $z[2^n - 1]$ with a fixed number $i \in [2^n - 1 : 0]$. (By easy we mean that it requires constant cost and delay.)

Claim 17 *Let* $\mathrm{wt}(\vec{z})$ *denote the value represented by $\vec{z}$ in unary representation. For $i > 0$:*

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

Question 18 *Present a combinational circuit of constant cost and delay that is given a binary string $z[2^n - 1 : 0]$ that represents a number in unary representation and compares it with a fixed i. The circuit has three outputs, denoted by $>, <$, and $=$, where exactly one of them should equal 1. (The outputs may behave arbitrarily if $\vec{z}$ does not represent a number in unary representation.)*

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

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

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 one could compare i and $\langle \vec{x} \rangle$ given $q, r, \langle \vec{x_L} \rangle$, and $\langle \vec{x_R} \rangle$.

Claim 18 *One can compare i and $\langle \vec{x} \rangle$ as follows:*

1. *If $\langle \vec{x}_L \rangle > q$, then $\langle \vec{x} \rangle > i$.*
2. *If $\langle \vec{x}_L \rangle < q$, then $\langle \vec{x} \rangle < i$.*
3. *If $\langle \vec{x}_L \rangle = q$, then the relation between i and $\langle \vec{x} \rangle$ is identical to the relation between r and $\langle \vec{x}_R \rangle$ (e.g. $\langle \vec{x}_R \rangle < r$ implies $\langle \vec{x} \rangle < i$).*

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.

In the next section we present a half-decoder design which is based on computing binary strings that represent $\langle \vec{x_L} \rangle$ and $\langle \vec{x_R} \rangle$ in unary representation. Claim 18 in conjunction with Question 18 are then used to compute each of the outputs $y[i]$ with constant delay and cost.

3.5.2 Implementation

In this section we present an optimal implementation of a half-decoder. We denote this design by H-DEC$^*(n)$. As in Section 3.4.1 our design is recursive. The design for $n = 1$ is identical to H-DEC(1).

Figure 3.5 depicts a recursive implementation of an H-DEC$^*(n)$. Note that the design uses an additional parameter $k \in [n-1:0]$, so formally the design depends on k as well. We omit k since a reasonable choice for the value of this parameter is $k = \lceil \frac{n-1}{2} \rceil$ (this will become clearer when we analyze the delay). 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$^*(k)$. We denote the outputs of the half-decoders by $z_L[2^{n-k}-1:0]$ and $z_R[2^k-1:0]$, respectively. The string $z_R[2^k-1:0]$ is shifted by one position to the left to obtain the string $z'_R[2^k-1:0]$. This shifting pads in a 1 and discards $z_R[2^k-1]$ (hence $z'_R[0] = 1$ and $z'_R[j+1] = z_R[j]$).

The indexes of the output $\vec{y}$ is partitioned into blocks $\{B_q\}_{q=0}^{2^{n-k}}$ as follows:

$$B_q \triangleq \begin{cases} [2^k - 2 : 0] & \text{if } q = 0 \\ [q \cdot 2^k + 2^k - 2 : q \cdot 2^k - 1] & \text{if } q > 0. \end{cases}$$

Note that the length of block B_0 is $2^k - 2$. To unify notation we may define $y[-1] = 1$, and then redefine $B_0 = [2^k - 2 : -1]$. Note also that $[2^n - 1]$ does not belong to any block; this is no cause of trouble since $y[2^n - 1]$ is always 0 (so one should connect $y[2^n - 1]$ to zero directly). We now discuss how the values of $y[i]$ for indexes i that belong to block B_q are computed. Note that i belongs to block B_q iff $i + 1 = q \cdot 2^k + r$, where $r \in [0 : 2^k - 1]$.

Compare $\langle \vec{z}_L \rangle$ and q. Let the result of the comparison be denoted by a symbol $\gamma_q \in \{<, =, >\}$, where $\gamma_q =<$ means that $wt(\vec{z}_L) < q$. (We ignore at this point the issue of how the symbol γ is actually represented by a binary string.) Now γ_q controls a "multiplexer" $\text{MUX}_q(2^k)$ with three data inputs marked by $in_>, in_=, in_>$. Given a control signal γ_q, the output of MUX_q equals the signal that is fed via the input in_{γ_q}. Finally, the signals connected to the 3 data inputs of $\text{MUX}_q(2^k)$ are: 0 is fed to $in_<$, 1 is fed to $in_>$, and $\vec{z'}_R$ is fed to $in_=$. This computation applied for every block B_q. Hence there are 2^{n-k} different instances of the comparator and $\text{MUX}_q(2^k)$. Each comparator is fed only by two bits of $\vec{z_L}$ (which explains the junction depicted by a filled circle).

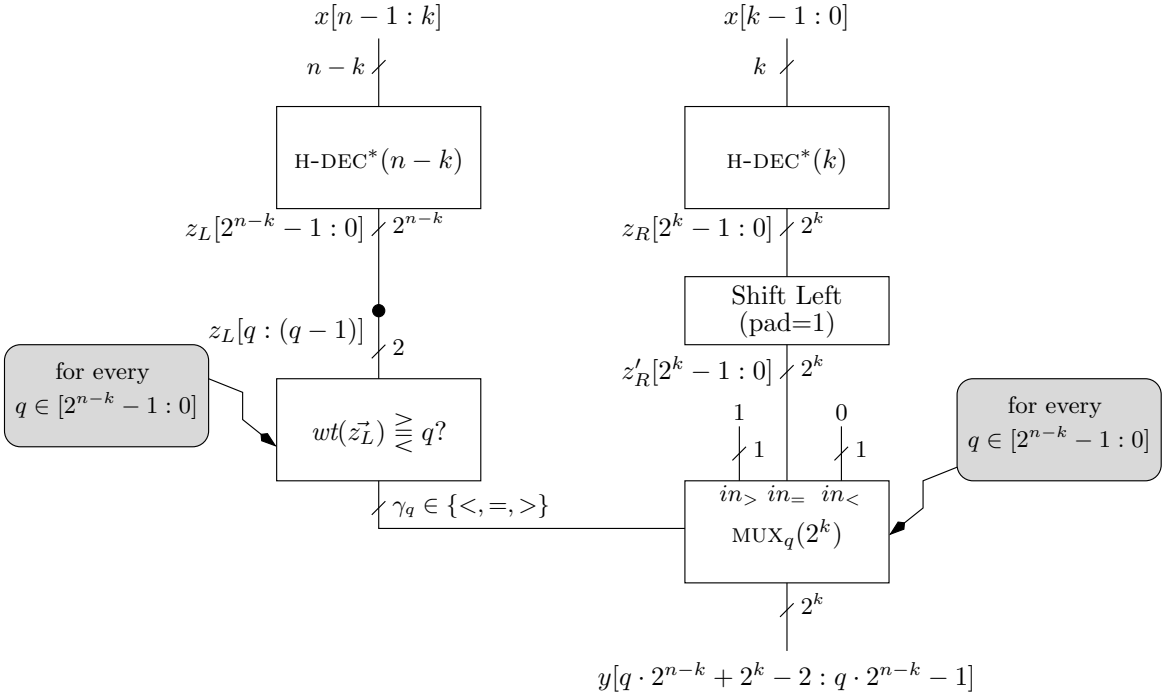


Figure 3.6: A recursive implementation of H-DEC$^*(n)$.

Example 4 *Let $n = 4$ and $k = 2$. The output $y[15:0]$ is partitioned into 4 blocks: $y[14:11], y[10:7], y[6:3], y[2:0]$.*

Consider $i = 6$. Since $i + 1 = 1 \cdot 4 + 3$, when divided by 4 the quotient is 1 and the remainder is 3. By Claim 13, $y[6] = 1$ iff $\langle \vec{x} \rangle > 6$. By Claim 18, $\langle \vec{x} \rangle > 6$ iff ($\langle \vec{x}_L \rangle > 1$) or ($\langle \vec{x}_L \rangle = 1$ and $\langle \vec{x}_R \rangle \geq 3$). It follows that if γ_1 equals $>$, then $y[6] = 1$. If γ_1 equals $=$, then $y[6] = z_R[2]$. Finally, if γ_1 equals $<$, then $y[6] = 0$.

Remark 3 *The design depicted in Figure 3.5 is a special case of the the design depicted in Figure 3.6, when $k = n - 1$. In this case there are just two blocks. The comparator is degenerated to $x[n-1]$ (e.g. indicating "less than" for the right block). The multiplexer is degenerated to an* OR*-gate or to an* AND*-gate.*

The following question shows that a shift by one position to the left incurs zero cost and zero delay.

Question 19 *Show how to implement the "Shift Left" box depicted in Figure 3.6 with zero cost and zero delay.*

Question 20 *Suggest a combinational circuit that computes $y[i]$ from $z_L[q], z_L[q-1]$ and $z_R[r-1]$.*

3.5.3 Correctness

Claim 19 *The design depicted in Figure 3.6 is a correct implementation of a half-decoder.*

Proof: The proof is by induction on n. The induction basis is identical to that in the proof of Claim 14. Before we proceed to the induction step we observe two properties:

1. $i \in B_q$ iff $i + 1 = q \cdot 2^k + r$, for $r \in [0 : 2^k - 1]$. In fact, we defined the blocks exactly so that this property holds.

2. If γ_q equals "=", then $y[i] = z'_R[\mathrm{mod}((i+1), 2^k)]$. The reason is that MUX_q connects $\vec{z'}_R$ to block B_q of the output. By definition, $B_q = [q \cdot 2^k + (2^k - 2) : q \cdot 2^k - 1]$ and the indexes of $\vec{z'}_R$ are $[2^k - 1 : 0]$. This means that $y[q \cdot 2^k + j - 1] = z_R[j]$, which is exactly what is claimed.

We now prove the induction step. By Claim 13 it suffices to show that $y[i] = 1$ iff $\langle \vec{x} \rangle \geq i + 1$, for every $i \in [2^n - 1 : 0]$. Fix an index i and let $i + 1 = q \cdot 2^k + r$. By Property (1), $i \in B_q$. By Claim 18,

$$y[i] = 1 \Longleftrightarrow (\langle \vec{x}_L \rangle > q) \text{ OR } ((\langle \vec{x}_L \rangle = q) \text{ AND } (\langle \vec{x}_R \rangle \geq r)). \tag{3.3}$$

We now consider three cases:

1. $\langle \vec{x}_L \rangle > q$. In this case $\gamma_q = ''>''$, hence the multiplexer MUX_q sets $y[i] = 1$, as required by Eq. 3.3.

2. $\langle \vec{x}_L \rangle < q$. In this case $\gamma_q = ''<''$, hence the multiplexer MUX_q sets $y[i] = 0$, as required by Eq. 3.3.

3. $\langle \vec{x}_L \rangle = q$. In this case $\gamma_q = ''='' $, hence By Property (2), $y[i] = z'_R[\text{mod}((i+1), 2^k)]$. We distinguish now between two sub-cases:

   (a) $r = 0$. $z'_R[r] = 1$, and hence $y[i] = 1$. This agrees with Eq. 3.3 since $\langle \vec{x}_R \rangle$ is always non-negative.

   (b) $r > 0$. In this case $z'_R[r] = z_R[r-1]$. The induction hypothesis implies that $z_R[r-1] = 1$ iff $\langle \vec{x}_L \rangle \geq r$. Hence in this sub-case $y[i] = 1$ iff $\langle \vec{x}_L \rangle \geq r$, which agrees with Eq. 3.3.

   The above case analysis covered all the possible cases for the right-hand side of Eq. 3.3. Hence both directions of the equation are proved, and the claim follows.

□

3.5.4 Cost and delay analysis

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

$$c(\text{H-DEC}^*(n)) = \begin{cases} 0 & \text{if n=1} \\ c(\text{H-DEC}^*(k)) + c(\text{H-DEC}^*(n-k)) & \text{otherwise.} \\ +2^{n-k} \cdot \big(c(\text{compare}) + 2^k \cdot c(\text{3:1-MUX})\big) & \end{cases}$$

The cost of a 3:1-MUX is constant. From Question 18, it follows that the cost of each comparator is also constant. It follows that

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

This recurrence is minimized when k and $n-k$ are as equal as possible. If $k = \lceil \frac{n-1}{2} \rceil$, then the recurrence degenerates to

$$c(\text{H-DEC}^*(n)) = 2 \cdot c(\text{H-DEC}^*(n/2)) + \Theta(2^n) \tag{3.4}$$

In the following question the reader is required to show the solution to this recurrence.

Question 21 *Show that the solution to the recurrence in Eq. 3.4 is* $c(\text{H-DEC}^*(n)) = \Theta(2^n)$.

It follows that the asymptotic cost of H-DEC$^*(n)$ is optimal.

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

$$d(\text{H-DEC}^*(n)) = \begin{cases} 0 & \text{if n=1} \\ \max\{d(\text{H-DEC}^*(k)), d(\text{H-DEC}^*(n-k)) + d(\text{compare})\} & \text{otherwise.} \\ +d(\text{3:1-MUX}) & \end{cases}$$

Again the delay of a comparator is constant as well as a delay of a 3:1-MUX. Set $k = \lceil \frac{n-1}{2} \rceil$, then the recurrence degenerates to

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

It follows that the delay of H-DEC$^*(n)$ is optimal, as promised.

3.6 Decoders

A decoder with input length n is a combinational circuit defined as follows:

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

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

Functionality:

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

Note 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 of length n by DECODER(n).

Example 5 *Consider a decoder* DECODER(3). *On input* $x = 101$*, the output* y *equals* 00100000.

3.6.1 Implementation

We design a DECODER(n) using recursion. We start by designing a DECODER(n) with $n = 1$. We then proceed by designing a DECODER(n) based on "smaller"decoders. The circuit DECODER(1) is simply one inverter where: $y[0] \leftarrow$ INV$(x[0])$ and $y[1] \leftarrow x[0]$.

Figure 3.7 depicts a recursive implementation of an DECODER(n). As in the definition of H-DEC$^*(n)$, the design uses an additional parameter $k \in [n-1:0]$, so formally the design depends on k as well. We omit k since, as in the H-DEC$^*(n)$ design, a reasonable choice for the value of this parameter is $k = \lceil \frac{n-1}{2} \rceil$. The input string $x[n-1:0]$ is divided into two strings $x_L[n-k-1:0] = x[n-1:k]$ and $x_R[k-1:0] = x[k-1:0]$. These strings are fed to decoders DECODER$(n-k)$ and DECODER(k). We denote the outputs of the decoders by $Q[2^{n-k}-1:0]$ and $R[2^k-1:0]$, respectively. Consider an index $i \in [2^n-1]$. Divide i by 2^k to obtain

$$i = 2^k \cdot q + r,$$

where $r \in [2^k-1:0]$ is the remainder and q is the quotient. The output bit $y[i]$ is computed by an AND-gate AND$_i$ that is fed by $Q[q]$ and $R[r]$.

3.6.2 Correctness

Claim 20 *The design depicted in Figure 3.7 is a correct implementation of a decoder.*

Proof: The proof is by induction on n. The induction basis is trivial. We proceed directly to the induction step. We need to show that $y[i] = 1$ iff $\langle \vec{x} \rangle = i$, for every $i \in [2^n-1:0]$. Fix an index i and let $i = q \cdot 2^k + r$, where $r \in [2^k-1:0]$. By Claim 18,

$$\langle \vec{x} \rangle = i \Longleftrightarrow (\langle \vec{x}_L \rangle = q) \text{ AND } (\langle \vec{x}_L \rangle = r). \tag{3.5}$$

The induction basis applied to DECODER(k) (DECODER$(n-k)$) implies that $R[j] = 1$ iff $\langle \vec{x}_R \rangle = j$ ($Q[j] = 1$ iff $\langle \vec{x}_L \rangle = j$). Since $y[i] =$ AND$(Q[q], R[r])$, it follows that $y[i] = 1$ iff $\langle \vec{x} \rangle = 1$, and the claim follows. □

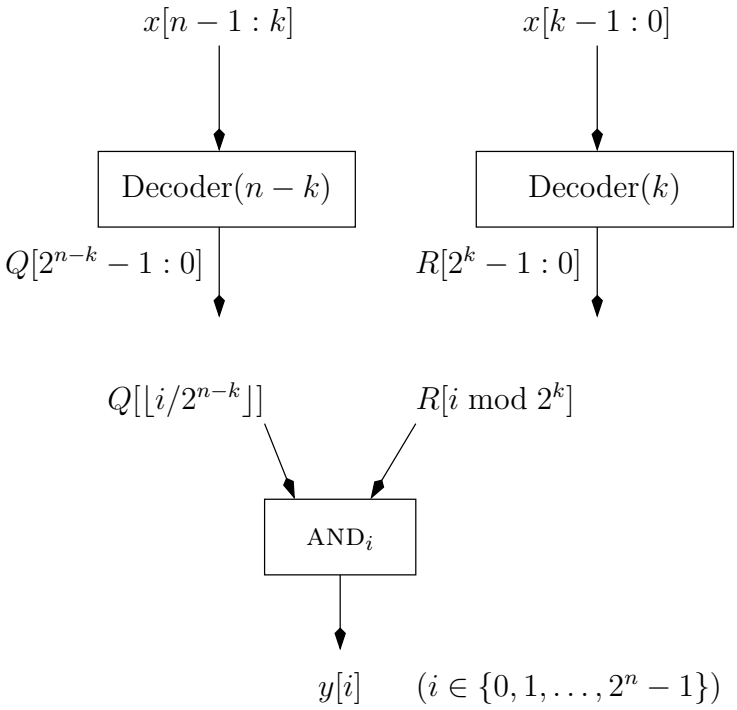


Figure 3.7: A recursive implementation of DECODER(n).

3.6.3 Cost and delay analysis

The cost of DECODER(n) satisfies the following recurrence equation:

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

It follows that

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

As in the analysis of the cost of H-DEC$^*(n)$, the cost is minimized when k and $n-k$ are as equal as possible. If $k = \lceil \frac{n-1}{2} \rceil$, then $c(\text{DECODER}(n)) = \Theta(2^n)$.

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

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

Set $k = \lceil \frac{n-1}{2} \rceil$, and it follows that $d(\text{DECODER}(n)) = \Theta(\log n)$.

Question 22 *Prove that* DECODER(n) *is asymptotically optimal with respect to cost and delay.*

3.7 Encoders

An encoder implements the inverse Boolean function of a decoder. Note however, that the Boolean function, DECODER$_n$, implemented by a decoder is not surjective. In fact, the range of DECODER$_n$ 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).

An encoder with input length 2^n and output length n is a combinational circuit defined as follows:

Input: $y[2^n - 1 : 0]$, with $wt(\vec{y} = 1)$ (i.e. exactly one bit equals 1 in $\vec{y}$).

Output: $x[n - 1 : 0]$

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

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

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 with input length 2^n and output length n by ENCODER(n).

Example 6 *Consider an encoder* ENCODER(3)*. On input* 00100000*, the output equals* 101*.*

3.7.1 Implementation

In this section we present a step by step implementation of an encoder. We start with a rather costly design, and show how it can be transformed to an optimal one.

As in the design of a decoder, out design is recursive. The design for $n = 1$, is simply $y[0] \leftarrow x[1]$. We proceed with the design for $n > 1$.

Figure 3.8 depicts an implementation of ENCODER(n). Our design methodology is "divide-and-conquer" (which, if fact, we already employed in the designs H-DEC$^*(n)$ and DECODER(n)). We partition the input $\vec{y}$ into two equals strings

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

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

An important problem with the design at this stage is that if $\vec{y}$ is a "legal" input (namely, $wt(\vec{y}) = 1$), one of inputs $\vec{y}_L$ or $\vec{y}_R$ is not. This renders our approach useless. To fix this problem we extend the functionality of an encoder so that

$$wt(\vec{y}) = 0 \implies wt(\vec{x}) = 0.$$

Namely, an all-zeros output is computed when the input is all-zeros.

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.

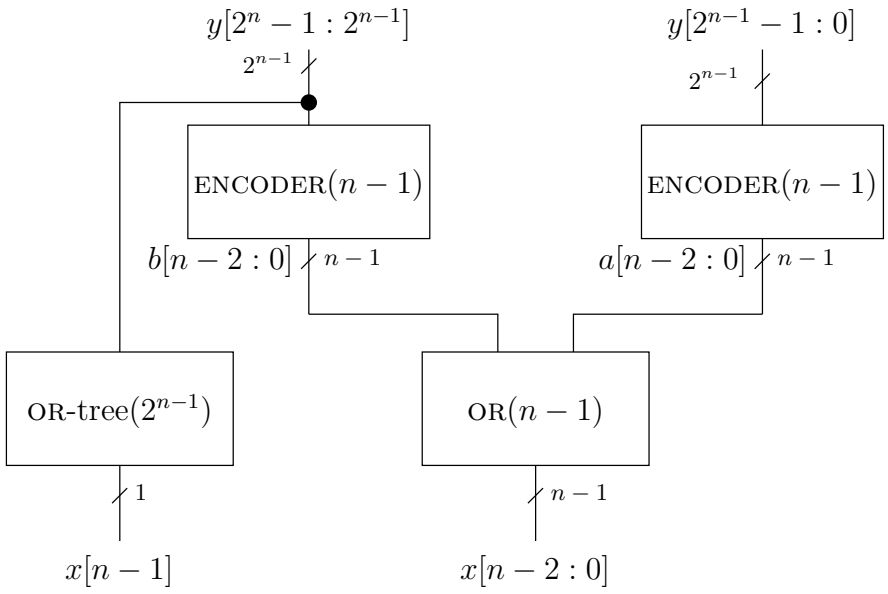


Figure 3.8: A recursive implementation of ENCODER(n).

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 desired output is $\vec{x} = 0 \cdot \vec{a}$.

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 desired output is $\vec{x} = 1 \cdot \vec{b}$.

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

We can unite the last three cases simply by setting the output $\vec{x}$ as follows:

$$x[n-1] = \text{OR}_{2^{n-1}}(\vec{y}_L)$$
$$x[n-2:0] = \text{OR}(\vec{a}, \vec{b}),$$

where $\text{OR}(\vec{a}, \vec{b})$ denotes bit-wise or-ing, namely, the binary vector $\vec{z}$ whose i bit satisfies $z[i] = \text{OR}(a[i], b[i])$. The reason that this unification is correct is that in all three cases one of the vectors $\vec{a}$, $\vec{b}$ is an all zeros vector. We conclude that the design ENCODER(n) is correct.

Claim 21 *The circuit* ENCODER(n) *depicted in Figure 3.8 is a correct implementation of an encoder with input length* 2^n.

The problem with this design is its cost as summarized in the following question.

Question 23 *This question deals with the cost and delay of* ENCODER(n).

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

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

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

Question 23 suggests that apart from fan-out considerations, the 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 22 *Let* ENCODER$_n$ *denote the Boolean function defined over binary strings of length* 2^n *whose weight is at most* 1. *If* $\mathrm{wt}(y[2^n-1:0]) \leq 1$, *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 3.9 depicts the design ENCODER$^*(n)$ obtained from ENCODER(n) after commuting the OR and the ENCODER$(n-1)$ operations. The correctness follows from the correctness of ENCODER(n) and Claim 22.

Question 24 *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?*

3.7.2 Cost and delay analysis

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

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

This recurrence equation is similar to that of the half-decoder H-DEC(n), hence

$$c(\text{ENCODER}^*(n)) = (2 \cdot 2^n - 2) \cdot c(\text{OR})) = \Theta(2^n).$$

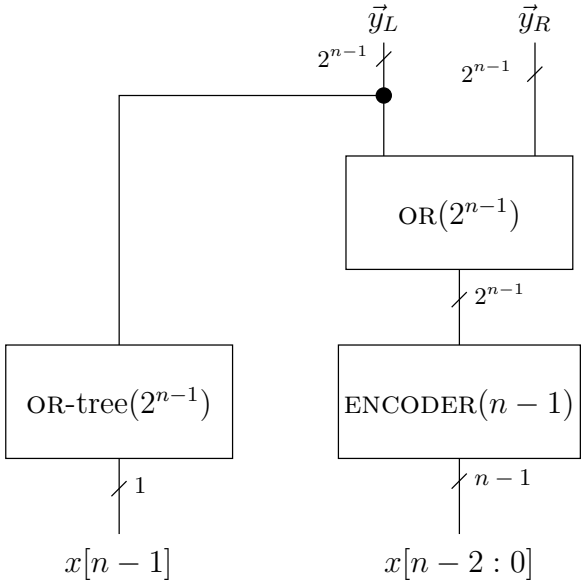


Figure 3.9: A recursive implementation of ENCODER$^*(n)$.

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

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

Again the delay of a comparator is constant as well as a delay of a 3:1-MUX. Set $k = \lceil \frac{n-1}{2} \rceil$, then the recurrence degenerates to

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

It follows that the delay of H-DEC$^*(n)$ is optimal, as promised.

The following question deals with lower bounds for an encoder.

Question 25 *Proof the cost and delay of* ENCODER$^*(n)$ *are asymptotically optimal.*

3.8 Multiplexers

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:

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

The input $\vec{S}$ encodes the index of the input bit 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 7 *Let $n = 4$, $D[3:0] = 0100$, and $S[1:0] = 11$. The output Y should be 1.*

3.8.1 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.

Figure 3.10 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.

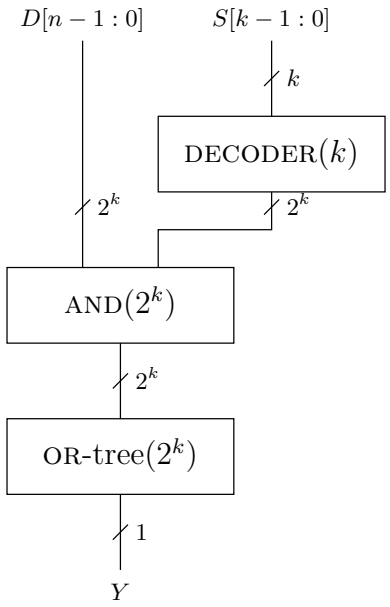


Figure 3.10: An (n:1)-MUX based on a decoder ($n = 2^k$).

Question 26 *The following question deals with the implementation of (n:1)-MUX suggested in Figure 3.10.*

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 second implementation of (n:1)-MUX is a recursive tree-like implementation. The design for $n = 2$ is simply a MUX. The design for $n = 2^k$ is depicted in Figure 3.11. 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 $S[k-1]$.

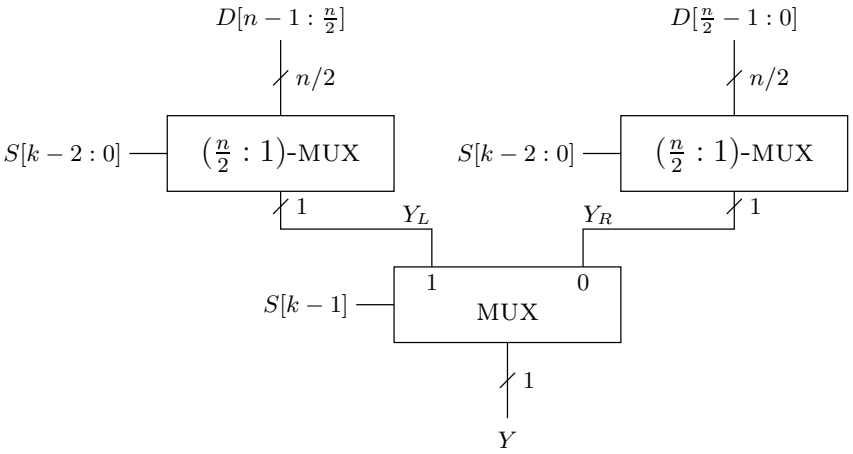


Figure 3.11: A recursive implementation of (n:1)-MUX ($n = 2^k$).

Question 27 *Answer the same questions asked in Question 26 but this time with respect to the implementation of the (n:1)-*MUX *suggested in Figure 3.11.*

Both implementations suggested in this section are asymptotically optimal with respect to cost and delay. Which design is better? Our model is not refined enough to answer this question sharply. 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.

Question 28 *Compute the cost and delay of both implementations of (n:1)-*MUX *based on the data in Table 2.1.*

3.9 Cyclic Shifters

A cyclic left shift by i positions of a binary string $a[n-1:0]$ is the string $b[n-1:0]$ where $b[j] = a[\mathrm{mod}(i+j, n)]$.

Example 8 *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*.*

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 $\langle s\vec{a} \rangle$ positions. Formally,

$$y[j] = x[\mathrm{mod}(j + \langle s\vec{a} \rangle, n)],$$

for every $j \in [n-1:0]$.

To simplify the discussion, we will assume in this section that n is a power of 2, namely, $n = 2^k$.

3.9.1 Implementation

We break the task of cyclic left shifting by $s\vec{a}$ positions into separate 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 defined as follows:

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

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

Functionality:

$$y[j] = x[\mathrm{mod}(j + s \cdot 2^i, n)],$$

for every $j \in [n-1:0]$.

A CLS(n, i) is quite simple to implement since $y[j]$ is either $x[j]$ or $x[\mathrm{mod}(j + 2^i, n)]$. So all one needs is a MUX to select between the two depending 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 3.12 depicts an implementation of a CLS$(4, 1)$. It is self-evident that the main complication with the design of CLS$(n.i)$ is routing (i.e. drawing the wires).

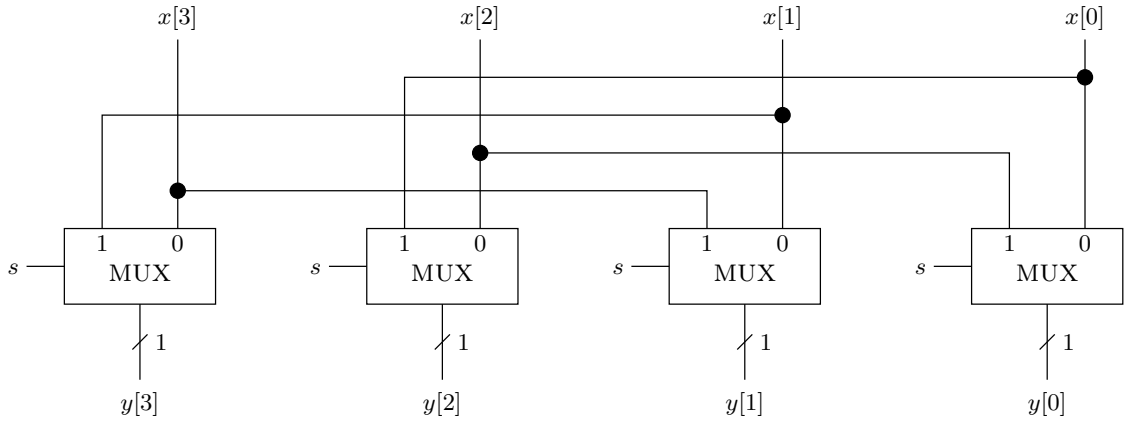


Figure 3.12: A row of multiplexers implement a CLS$(4, 1)$.

Having designed a CLS(n, i) we are all set for implementing a BARREL-SHIFTER(n). Figure 3.13 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 $sa[i]$.

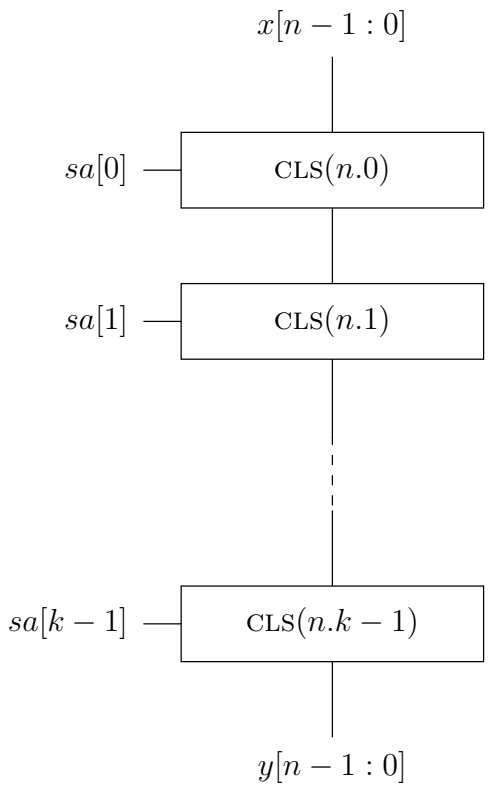


Figure 3.13: A BARREL-SHIFTER(n) built of k levels of CLS(n, i) $(n = 2^k)$.

Question 29 *This question deals with the design of the* BARREL-SHIFTER(n) *depicted in Figure 3.13.*

1. *Prove the correctness of the design.*
2. *Analyze the cost and delay of the design.*
3. *Prove the asymptotic optimality of the delay of the design.*
4. *Prove a lower bound on the cost of a combinational circuit that implements a cyclic shifter.*

3.10 Priority Encoders

The *leading one* in a binary string that is not all zeros is the left most bit that equals 1. A priority encoder is a combinational circuit that computes the index of the leading one. Formally, PENC(n) is a combinational circuit defined as follows.

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

Output: $y[k-1:0]$, where $k = \lceil \log_2 n \rceil$.

Functionality:

$$\langle \vec{y} \rangle = \max\{i \mid x[i] = 0\},$$

where we use the notation that the maximum over an empty set is zero.

To simplify the discussion, we will assume in this section that n is a power of 2, namely, $n = 2^k$.

Example 9 *Given input* $\vec{x} = 010111$*, a* PENC(6) *outputs* $y[2:0] = 100$*, since* $x[4]$ *is the leading one in* $\vec{x}$*.*

3.10.1 Implementation

We apply divide-and-conquer in the recursive design of a priority encoder. A PENC(1) is a trivial circuit since $y[0] = 0$ (recall that $\langle \vec{y} \rangle = 0$ if $\vec{x} = 0^n$). A recursive design of PENC(n) for $n > 1$ that is a power of 2 is depicted in Figure 3.14.

Question 30 *This question deals with the design of the priority encoder* PENC(n) *depicted in Figure 3.14.*

1. *Prove the correctness of the design.*
2. *Analyze the cost and delay of the design.*
3. *Prove the asymptotic optimality of the delay of the design.*

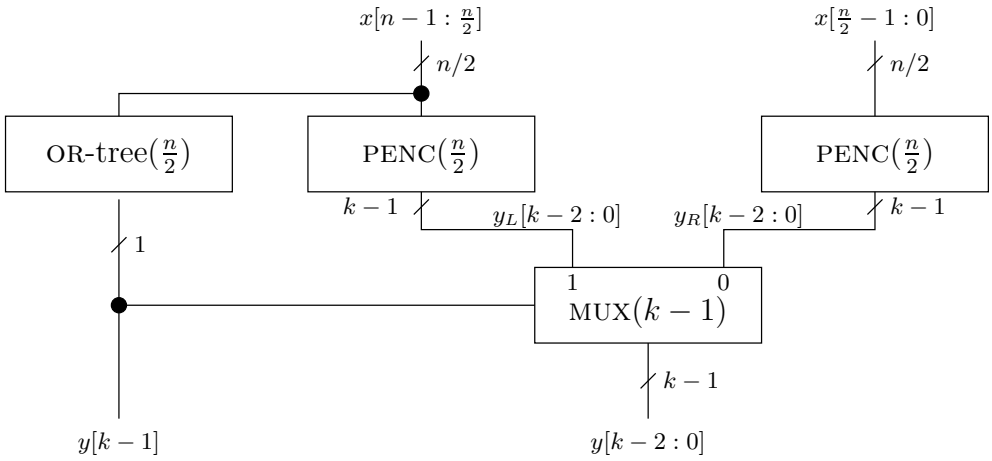


Figure 3.14: A recursive implementation of PENC(n).

4. *Prove a lower bound on the cost of a combinational circuit that implements a priority encoder.*

The cost of PENC(n) depicted in Figure 3.14 is $\Theta(n \log n)$. Can we reduce the cost to $\Theta(n)$? The following question deals with this issue (a somewhat more "systematic" linear cost design can be obtained by parallel prefix computation).

Question 31 *Modify the* PENC(n) *design depicted in Figure 3.14 so that the cost is* $\Theta(n)$. *(Hint: use a single "global"* OR*-tree*(n). *The internal signals of the* OR*-tree are fed to the recursive instances* PENC$(\frac{n}{2})$.*)*

Chapter 4

Addition

Wolfgang Paul suggested that I ask students (before teaching how adders are designed) to try to define what a binary adder is. When I pose this question in class I often get answers like: "sure, it is a circuit that adds numbers!". When I insist that I am after a formal answer, I can eventually get an answer that has some reasonable ingredients, but is never quite there. Lack of ability to formally specify what a circuit should does hit back when one tries to design somewhat more complicated circuits. Without formal specification there is never confidence or evidence that modular design methodology can be successful; separately, modules function properly, but when "glued" together, strange things happen.

Perhaps one of the least understood topics is how subtraction is done via addition. No wonder, except for books of Wolfgang Paul and his co-authors, no formal specification or proof is provided as to what an adder is, and how it can be used to perform subtraction in two's complement representation.

4.1 What is a binary adder?

A binary adder ADDER(n) is a combinational circuit defined as follows.

Input: $A[n-1:0], B[n-1:0]$, and $C[0]$.

Output: $S[n-1:0]$ and $C[n]$.

Functionality:

$$\langle \vec{S} \rangle + 2^n \cdot C[n] = \langle \vec{A} \rangle + \langle \vec{B} \rangle + C[0] \tag{4.1}$$

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 32 *Verify that the functionality of* ADDER(n) *is well defined. Show that the set of numbers that can be represented by sums* $\vec{A} + \vec{B} + C[0]$ *equals the set of numbers that can be represented by sums* $\vec{S} + C[n]$.

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

Question 33 *Prove lower bounds on the cost and delay of combinational circuits that implement an* ADDER(n).

4.2 Ripple-carry adders

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 defined as follows.

Definition 20 (Full-Adder) *A* full-adder *is a combinational circuit with* 3 *inputs* $x, y, z \in \{0,1\}$ *and* 2 *outputs* $s[1:0]$ *that satisfies:*

$$\langle s[1:0] \rangle = x + y + z.$$

The output $s[0]$ of a full-adder is often called *the sum output*, and The output $s[1]$ of a full-adder is often called *the carry-out output*. We denote a full-adder by FA.

A ripple-carry adder, RCA(n), built of a chain of full-adders is depicted in Figure 4.1. The carry-out output of the ith full is denoted by $c[i+1]$. One can readily notice that an RCA(n) adds numbers in the same fashion that we a taught to add in elementary school.

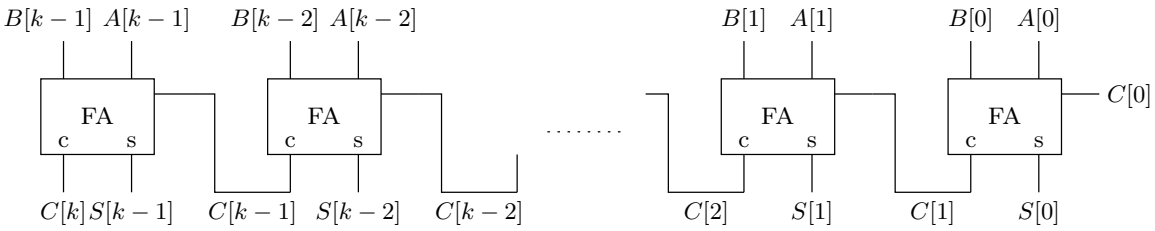


Figure 4.1: A ripple-carry adder RCA(n).

4.2.1 Correctness proof

To facilitate proving the correctness of an RCA(n) we first present an equivalent recursive definition. An RCA(1) is simply a full adder. A recursive description of RCA(n), for $n \geq 1$, is depicted in Figure 4.2. The following claim deals with the correctness of RCA(n).

Claim 23 RCA(n) *is a correct implementation of* ADDER(n).

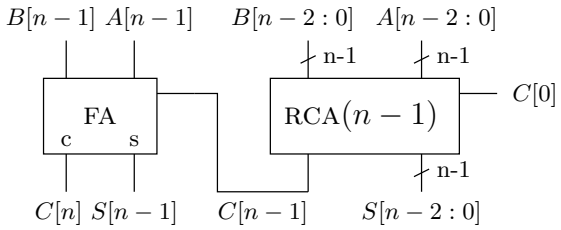


Figure 4.2: A recursive description of RCA(n).

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

$$\langle A[n-2:0]\rangle + \langle B[n-2:0]\rangle + C[0] = 2^{n-1} \cdot C[n-1] + \langle S[n-2:0]\rangle. \tag{4.2}$$

The definition of a full-adder states that

$$A[n-1] + B[n-1] + C[n-1] = 2 \cdot C[n] + S[n-1]. \tag{4.3}$$

Multiply Equation 4.3 by 2^{n-1}. Note that $2^{n-1} \cdot A[n-1] + \langle A[n-2:0]\rangle = \langle A[n-1:0]\rangle$. By adding with Equation 4.2 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. □

4.2.2 Delay and cost analysis

The cost of an RCA(n) satisfies:

$$c(\text{RCA}(n)) = n \cdot c(\text{FA}) = \Theta(n).$$

The delay of an RCA(n) satisfies

$$d(\text{RCA}(n)) = n \cdot d(\text{FA}) = \Theta(n).$$

The answer to Question 33 implies that the cost of 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 ADDER(n) designs.

TO BE COMPLETED

Chapter 5

Introduction to Interrupts

5.1 Computational models

By now you are familiar with two models of a "computer": (A) A DLX CPU with main memory that can execute programs written in DLX machine language. (B) A computer system that can execute various programs. These programs are often written in a high level language such as C. Moreover, computers are capable of executing multiple programs simultaneously (or at least give users sometimes the impression that programs are being executed simultaneously). How is abstraction B obtained from abstraction A?

Every complex system is built hierarchically: each level in the hierarchy provides service to the level above it and receives service from the level below it. Good engineering practice is expressed by concise and precise definitions of each level and of the interactions between the levels. In our context, the DLX CPU abstraction is obtained by a digital circuit, which in turn is based on the digital abstraction. The abstraction of a computer system is based on the abstraction of a CPU with main memory (as well as many other components). This chapter deals with how the abstraction of a computer system is obtained.

Let us start with a relatively simple issue. How are programs written in C executed on a DLX CPU? This question encapsulates many issues, the first issue is the issue of language; it is required to translate the program written in C to the DLX machine language. Translation is performed by a program called a *compiler*. A somewhat over-simplified definition of a compiler is that it is a program that is given as input a file that contains a program written in a high level language and outputs its translation to a program written in machine language. However, translation is not enough to bridge between the two abstractions (i.e. high-level programming and a computer running programs).

A computer can be viewed as a list of resources: the CPU, main memory, secondary memory (disks), keyboard, mouse, monitor, printer, modem, etc. (Just look at computer ads if you are curious about more components that are often part of a computer system). A program that is executed by a computer uses these resources. The main resource consumed by a running program is the time of the CPU, namely the right to run commands of the program on the CPU. Other resources used by a program are storing the program and the data in the memory, receiving input from the use via the mouse and keyboard, displaying output on the monitor. The main problem is in managing such a complex set of resources

so that they function in a coordinated fashion. Moreover, a typical programmer does not concern herself with the fact that some of the resources are being used by other programs. Magically a typical programmer works under the illusion that all the resources are available upon request, and yet even though this assumption is false things "work out eventually".

Question 34 *Points to think about:*

1. *Does a programmer know the physical addresses in the main memory in which the program is stored when it is executed? Can you describe a situation in which the address is determined in advance and a situation in which it is not determined in advance?*

2. *What happens from the moment you click on an icon of an application till the application actually starts running?*

3. *What happens when a computer is being boot?*

5.2 The operating system

In this section we describe the main gaols of the operating system and how these goals are achieved. We refer to the operating system as the OS.

5.2.1 Gaps between models

As pointed out earlier, a computer is a list of resources. Programs are usually designed under the assumption that all the resources are available upon request. How can one justify this assumption?

We list below a few important problems in computer systems. These problems point out gaps between the model of a CPU plus main memory and the model of a computer system.

1. Suppose that a user is running two applications: an editor and a browser. The two programs are stored in the main memory and are executed according to some schedule (for example, the user types some sentences using her editor, and then reads some pages using the browser). Each program is designed under the assumption that all resources belong to it. But the main memory is a resource that is used simultaneously by both. It is absolutely crucial that the editor not over-write information stored in the memory space of the browser, and vice-versa. This means that each program should have a well defined "territory" in the main memory that can be changed only by the program. Mechanisms should be devised to allocate such territories to running programs, to protect these territories, and finally to return these territories to the "public pool" when the programs halts.

2. Suppose the editor is faulty; in particular, assume that the editor runs into an endless loop. Executing the loop requires CPU cycles. Does that mean that the editor takes over the CPU? Does that mean that the computer should be "stuck" since now the CPU is busy executing the instructions of the endless loop? Obviously, the CPU is

not conscious of the fact that the execution of instructions of an endless loop is a futile task. How is control of the CPU taken away from the editor?

This question is a crucial one, since the CPU is where the main activity takes place. There is a crucial difference between a program printing an endless number of pages and a program taking over the CPU. In the case of an endless printout, other programs may not be able to print, but they could be executed. In the case of a program taking over the CPU, other programs are disabled, and often the whole system is rendered useless.

3. How are keyboard and mouse movements interpreted? This problem is of particular importance when more than one application is being executed. For example, when you press the "enter" key, how is this input sent to the right application?

4. How can the same program be executed simultaneously multiple times? I often have many windows of a browser application running simultaneously. Is that like having many separate applications (each running as if it were the only application)? Or, is there some "smart" way of letting the multiple instances of the same program share resources. It seems rather wasteful to store the many copies of the same instructions in the main memory.

5. How can multiple users use the same computer? Is there a difference between programs of different users and programs of the same user? How can different user execute the same program simultaneously?

5.2.2 Processes

A key notion in providing the abstraction of a computer system is the notion of a *process*.

Definition 21 *A* process *is an instance of an execution of a program.*

Note that a program may be run simultaneously more than once (by the same user or even by different users). Each such execution is called an *instance*.

The goal of the OS is to provide each process with the abstraction of a *virtual computer*. Namely, both the user and programmer of a program assume that the program is the only program that runs on the computer, and that all the resources are available upon request. The virtual computer is then a concept that captures the idea that a program has a memory space, the ability to receive input from the user, the ability to display output on the screen, etc.

The OS manages multiple processes and allocates the resources to the processes. In particular, our abstraction of a CPU is that the CPU can execute one instruction at a time. This means that the operating system must schedule the processes: processes are given the right to execute instructions on the CPU, and that right may be taken away before the process terminates. We refer to a process whose instructions are being executed by the CPU as *active*. Inactive processes are often referred to as sleeping processes. It is the task of the OS to be able to swap between an active process and a sleeping process.

To be able to provide the abstraction of the virtual computer to all the processes, the OS must deal with the following issues:

- Management of memory: every process is limited to its memory space. Correct management has many implications such as: privacy (a process may not read the contents of the memory in addresses out of its memory space), security (a process may not write to addresses out of its memory space), and consistency (if processes share memory, then they must not violate consistency).
- Fairness: resources must be allocated in a fair manner.
- Avoiding deadlock. An example of deadlock is a situation in which Process A is holding resource X and needs resource Y to proceed. On the other hand, Process B is holding resource Y and needs resource X to proceed. Neither process releases a resource not proceeds, and the system is stuck.

It is important to note that the OS is itself a process (or a set of processes). This leads for the need for at least two levels of privileges: a regular user privilege, and a system privilege. A process with system privileges may allocate resources or capture a resource. The system privileges are given to the OS processes so that they can provide the abstraction of the virtual computer to user processes.

5.2.3 Process context

The implementation of the idea of a process requires that a running user program be encapsulated by various data that enables the OS to provide the abstraction of the virtual computer. This "wrapping" is constructed when the processes is initiated and is managed by the OS. We refer to this wrapping as the *process context*. The context of a process consists of the following parts:

- A memory space: the memory space contains the program, data, the program stack, and the program's heap.
- The state of the CPU: the contents of the general purpose registers and the special registers.
- Resources allocated to the process: file handles, buffers, etc.

5.2.4 Context switching

Our abstraction of the CPU is that the CPU can execute a single instruction during every cycle. The execution of an instruction may require several clock cycles. Even pipelined CPU's and CPU's with parallelism usually function equivalently to a CPU that runs instruction in a serial fashion.

In order to run multiple processes, the OS must be able to switch between an active process and an inactive process. Such an event is called *swapping* of processes or *context switching*. When should the OS decide to switch contexts? We list a few cases that require context switching.

- There is an error in the active process: division by zero, a mis-aligned access to the memory, an illegal instruction.
- A page fault occurred.
- The active process is requesting service from the OS. For example, the process wishes to open a file, close a file, read from a file, etc.
- Too much time has passed: to guarantee fairness, the CPU should be allocated to other processes.
- Hardware faults. For example, a memory access failed or, in general, a device is reporting an error.
- Reset: the user pressed the reset button.

A mechanism must be devised to enable context switching. Except for the case that the active process requests service from the OS, in all the other examples the execution of the active process is preempted. Namely, the active process is stopped without request.

5.3 Interrupts

Interrupts play a key role in enabling the CPU and OS to provide the abstraction of the virtual computer.

Definition 22 *An* interrupt *is a change in the sequence of executed instructions that is not caused by a jump or branch instruction.*

Note that the definition of an interrupt assumes that there exists some sequence of instructions that should be executed in the absence of interrupts. In this sequence the typical successive instruction I_{j+1} to instruction I_j is simply the instruction stored in address $PC+4$ (or $PC+1$ if the memory is word addressable rather than byte addressable).

5.3.1 List of common interrupts

We list the various types of interrupts below:

1. Reset: A reset interrupt is caused by the user pressing the reset button. A privileged user (or process) may also initiate a reset by a special command. One should also consider the situation shortly after the computer is turned on. We will present a unified way for dealing with all these cases, and for simplicity focus on the case that the reset button is pressed.
2. Illegal instruction: Recall that the contents of the PC register determines the next fetched instruction. The fetched instruction is stored in the IR register. The IR environment in conjunction with the control attempt to decode the instruction. However, not all 32-bit strings are valid encodings of instructions. An illegal instruction interrupt

is the event that the decoding of an instruction fails. If a machine language program is generated by a compiler, then we do not expect an illegal instruction interrupt to occur. It is still possible if a jump is made to a memory address that does not store an instruction. Another possibility is that a store instruction stored a non-instruction in the address pointed now by the PC.

3. Misaligned access: The main memory allows to access single bytes (i.e.8 bits), half-words (i.e. 16 bits), or words (i.e. 32 bits). However, when accessing a word, the address must be divisible by 4 (i.e. the two least significant bits must equal zero). Similarly, when accessing a half-word, the address must be even. A memory access that does not satisfy this condition causes an misaligned memory access. The main reason for this rule is the organization of the RAM.

4. Page fault: Explaining page-faults requires some explaining of a method called virtual memory (which is part of the abstraction of the virtual computer). Very briefly, virtual memory is an abstraction that decouples the memory space of a process from the physical main memory. A process is designed under the abstraction that there is a rather large memory space at its disposal. The virtual memory can be even much bigger than the actual amount of memory in the main memory. The virtual memory is divided into pages (the length of each page is typically a few kilo-bytes). Instead of attempting to store all the pages of the virtual memory in the main memory (an impossible task if the virtual memory is bigger than the physical memory), the OS manages the virtual memory by storing only a subset of the pages in the physical memory. The management of the virtual memory has a few aspects. First, the address of every memory access must be translated to the corresponding physical address. We do not describe this mechanism here, but point out that this is achieved by managing a table with the list of stored pages and their physical addresses. A second issue, is what should be done if an accessed memory address belongs to a page that has not been copied to the main. This is exactly the event of a page-fault. In this event, the OS determines which page should be evacuated. The space of the evacuated page is used to store the page that contains the accessed address. Once the page has been loaded, the memory access can be completed. The chapter of virtual memory is typically a major chapter in OS books and computer architecture books. We strongly suggest that you read such a chapter.

5. A system call: The abstraction of the virtual memory not only provides the illusion that all the resources are available to the process. The virtual computer abstraction also dictates that the usage of resources be done by calling procedures of the OS. Every "printf" instruction in a C program is translated to calling an OS procedure. A process may not access the screen, the keyboard buffer, etc. directly. All such accesses are mediated by the OS. System calls are implemented by a special DLX instruction, called "trap".

6. Overflow: overflow is an interrupt that occur when an execution of an arithmetic instruction (i.e. add or subtract) results with an overflow. Divide by zero is a similar type of interrupt.

7. Input/Output interrupts: Devices such as a keyboard, mouse, hard disk, printer, modem, etc., are input/output devices. Most of these devices are characterized by being much slower than the CPU and the memory. For example, printing a page can take several seconds. The common method of dealing with a slow device, such as a hard disk, is to give the disk a task (read or write a block). When the task is completed, the disk signals this fact, and eventually the CPU returns to deal with the disk. There is no good reason for the CPU to remain idle while the disk is completing its task. The event that an input/output signals that the CPU it needs the "attention" of the CPU is an input/output interrupt.

5.3.2 Characteristics of interrupts

To be completed:

1. internal/external
2. maskable/non-maskable
3. type: abort/repeat/continue
4. priority

5.4 Handling interrupts

To be completed

1. ISR
2. comparison with invoking procedures

5.5 Main problems

To be completed

1. Simultaneous occurrence of interrupts. Which interrupt should be served first?
2. Nesting of interrupts.

Chapter 6

DLX with interrupt handling

This description of interrupt handling in a DLX processor is based on Chapter 8 in the book of Müller and Paul [**?**]. It is highly recommended that you read that chapter.

6.1 List of interrupts and their properties

Table 6.1 lists the interrupts that we consider; each interrupt is given a priority, and three additional attributes: internal/external, maskable/non-maskable, and type (abort, repeat, continue). The interrupts are numbered from 0 to 31. Interrupt 0 has the highest priority and interrupt 31 has the lowest priority.

Name	Priority	Internal/External	Maskable/Non-maskable	Type
reset	0	external	non-maskable	abort
illegal instruction	1	internal	non-maskable	abort
misaligned access	2	internal	non-maskable	abort
page fault on fetch	3	internal	non-maskable	repeat
page fault on load/store	4	internal	non-maskable	repeat
trap	5	internal	non-maskable	continue
arithmetic overflow	6	internal	maskable	continue
input/output	$7-31$	external	maskable	continue

Table 6.1: List of interrupts

Question 35 *List, for each interrupt, the state of the control of the DLX when the interrupt occurs and the instruction that is executed when the interrupt occurs. Note that, for some interrupts, the control must be in a specific subset of states, whereas, for others, the control may be in any state.*

6.2 Signaling an occurrence of an interrupt

The DLX with hardware support has an event signal for every possible interrupt. The event signal for interrupt i is denoted by $evn[i]$. The occurrence of interrupt i is signaled by a signal $evn[i]$. There is a difference between the signaling of internal interrupts and external interrupts. The following assumptions are used for signaling of interrupts:

External Interrupts: We assume that $evn[i]$ is output by a register that belongs to an external device outside of the processor. The external device activates $evn[i]$ to signal that it would like to receive service from the OS. The $evn[i]$ signal remains active until the the Interrupt Service Routine (ISR) instructs the external device that the interrupt has been served and that $evn[i]$ should be deactivated[1].

The reasons for this signaling mechanism are: (a) The control of the DLX may be in any state when $evn[i]$ is activated. We would like the control to be in a unique state (i.e. "fetch") when the decision is made whether an interrupt should be served. (b) The task of remembering that an external interrupt is pending is delegated to the external devices (this also simplifies timing issues).

Internal Interrupts: The occurrence of an unmaskable internal interrupt is signaled by a condition of the form:

$$evn[i] = States(i) \ \wedge \ Condition(i)$$

where: (a) $States(i)$ denotes the set of control states in which interrupt i can occur; and (b) $Condition(i)$ denotes the required condition for an occurrence of interrupt i.

Examples:

1. An illegal instruction can occur if: (a) the control is in the "decode" state; and (b) the control reports that the instruction is illegal.
2. A misaligned memory access can occur if: (a) the control is in the states: "fetch", "load" or "store"; and (b) the memory environment reports a misaligned memory access. How does the memory environment determine if a misaligned memory access occurred? The answer is that it must know what type of access is made (word, half-word, or byte). It then checks the 2-LSBs of the address to see if the address is properly aligned.

The occurrence of a maskable internal interrupt is signaled by a condition of the form:

$$evn[i] = States(i) \ \wedge \ Condition(i) \ \wedge \ SR(i)$$

where $SR(i)$ denotes whether interrupt i is enabled.

[1]The ISR communicates with the external device simply by loading and storing words in a dedicated area in the memory. For more details, see the topic of *memory mapped I/O* in hardware books.

Example: An overflow interrupt is signaled if: (a) the control is in an "alu" or "alui" state; (b) the ALU environment is signaling an overflow and the executed instruction is a signed add or subtract instruction; and (c) the overflow interrupt is enabled.

The occurrence of an internal interrupt is signaled only during one clock cycle. The DLX must store this signal so that it can be treated. We will see that the control is modified so that an occurrence of an internal interrupt causes a transition to a special state.

Effect of masking. Interrupt masking has a different effect on internal interrupts and on external interrupts: An occurrence of a disabled internal interrupt is ignored altogether (namely, the occurrence is "lost" forever); an occurrence of a disabled external interrupt is ignored until the interrupt is enabled (i.e. the $evn[i]$ signal remains active).

6.3 Hardware and Software Support

Interrupt handling requires both hardware and software support. The software support is done by a program of the operating system called the *Interrupt Service Routine* (ISR - in short).

Hardware support

Hardware support for handling interrupts consists of four components: special registers, new instructions, extension of the datapath, and extension of the control.

In practice, some of the new instructions are restricted and may only be used by routines that are part of the operating system. This is enforced by defining two modes for the processor: a user mode and a privileged mode. A user program is allowed to use a subset of the instructions that can be used during privileged mode. This means that additional hardware mechanisms need to be introduced to enable changing the mode and checking that restricted instructions are used only in privileged mode. The DLX architecture does not support multiple modes; we simply assume that user programs do not contain restricted instructions.

Special registers. In general, the access to the special registers is restricted and only the operating system is allowed to execute instructions that modify the special registers.

The interrupt handling mechanism uses the following special registers:

1. Status Register $SR[31:0]$. Interrupt i is enabled when $SR[i] = 1$. A user can change the contents of the SR register in two ways: (1) Use a system call to enable/disable maskable interrupts. The operating system (i.e. the ISR) can use a privileged instruction "movei2s" to set the contents of the SR. Of course, the system verifies that the request is "legal" before changing the mask. (2) During the execution of an instruction in which an overflow should be signaled (i.e. signed add, signed subtract), the overflow interrupt is enabled.

2. Cause Register $CA[31:0]$. Interrupt i is pending if $\mathrm{CA}(i) = 1$. The functionality of the cause register CA is given by

$$CA[i](t+1) = \begin{cases} reset \vee pup \vee (CA[0](t) \wedge not(\mathrm{clear\_CA}(t)) & \text{if } i = 0 \\ evn[i](t) & \text{if } i \in [31:7] \\ evn[i](t) \vee (CA[i](t) \wedge not(\mathrm{clear\_CA}(t))) & \text{if } i \in [6:1]. \end{cases}$$

   Note that $i = 0$ means a reset interrupt, $i \in [31:7]$ means an external input/output interrupt, and $i \in [6:1]$ means an internal interrupt.

   We will later address the following questions: Why and when is the CA register cleared? Who is allowed to clear the CA register?

3. The Exception Program Counter EPC. Holds the return address for the ISR before jumping to the ISR.

4. The Exception Status Register ESR. Holds the contents of the SR before jumping to the ISR.

5. The Exception Cause Register ECA. Holds the list of pending unmasked interrupts when jumping to the ISR. When jumping to the ISR, the ECA register satisfies:

$$ECA[i](t+1) = \begin{cases} CA[i](t) & \text{if } i \leq 6 \\ CA[i](t) \wedge SR[i](t) & \text{otherwise} \end{cases}$$

   Note that $i \leq 6$ means a non-maskable interrupt.

6. The Exception Memory Address Register EMAR. When jumping to the ISR, the contents of the MAR are copied to the EMAR. This enables the ISR that deals with page faults to know which address caused the page fault.

New instructions. Four instructions are used for interrupt handling:

1. `movei2s RS1 SA` : copies the contents of $R_{\langle RS1 \rangle}$ to $Special\_R_{\langle SA \rangle}$.

2. `moves2i RD SA` : copies the contents of $Special\_R_{\langle SA \rangle}$ to $R_{\langle RD \rangle}$.

3. `trap type` : a request for a system call. Note that the ISR extracts the type of the system call from the word in the main memory whose address is stored in the EPC (the IR is not backed up). The trap instruction causes some side effects that are common to the all jumps to the ISR; we will specify these side effects later.

4. `rfe` : return from exception. Restores the SR and PC registers as follows:

$$\begin{aligned} SR(t+1) &= ESR(t) \\ PC(t+1) &= EPC(t) \end{aligned}$$

Extension of the datapath. Handling interrupts requires a few modifications in the datapath.

1. PC environment. Previously, the PC register sampled either the new address from the D-Bus or was reset to zero during power-up. The situation now is very similar: during power-up or jump-to-ISR the PC is reset to starting address of the ISR; otherwise, the PC samples the D-bus. Question: when should the PCce signal be active?

2. Memory environment. The write signals are activated only during write accesses which are properly aligned - it is critical that memory is protected from store instructions that are misaligned! A similar demand holds for page faults; namely, during a page fault the memory is not changed. The memory environment outputs the event signals for misaligned access, page fault during fetch, and page fault during load or store. The busy signal indicates, as before, whether a successful memory access is not completed yet.

3. Special purpose register environment. Consists of 5 registers (the CA register is considered part of the control). Supports the movei2s and moves2i instructions as well as parallel copying of registers described later.

Extension of the control The following states are added to the control:

1. movs2i, movi2s: these state deal with the execution phase of the corresponding instructions.

2. JISR(-1), JISR1, JISR2: these states deal with part of snap-shooting the state of the processor. We elaborate on these states in Sec. 6.4.

3. RFE1, RFE2: these states deal with the execution phase of the rfe instruction. We elaborate on these states in Sec. 6.4

External interrupts may occur in any state of the control. The decision to serve them takes place only during the "fetch" state.

Internal interrupts occur in specific states of the control (see Question 35). An occurrence of an internal interrupt in a state in which such an interrupt may occur causes a transition to one of the states JISR(-1) or JISR1. For example: an occurrence of a page fault during "load" causes a transition to JISR(-1), an occurrence of misaligned access during "store" causes a transition to JISR(-1), an occurrence of an overflow during an "alu" state causes a transition from the write-back states to JISR1. We describe in Section 6.4 why and when a transition to JISR(-1) is needed.

The illegal instruction interrupt is an exception to this rule: instead of adding a transition to JISR1, a transition was added to "fetch". The reason for this is simplification of control at the cost of a unified structure (think twice before you apply such tricks...).

The following control signals determine whether jump to the ISR takes place (i.e. whether a transition to JISR(-1) or JISR1 takes place):

1. Interrupt handling is invoked from the "fetch" control state by a signal *int* defined by

$$int = \bigvee_{i=0}^{3} CA[i] \quad \vee \quad \bigvee_{i=7}^{31} (CA[i] \wedge SR[i])$$

   Note that interrupts 0-6 are non-maskable. An illegal instruction (interrupt 1) is detected during "decode", causes the control to return to "fetch" (the default transition in the control is to go to "fetch"), and then $CA[1]$ is checked. Interrupts 4-6 are considered during other control states.

2. From the "load" and "store" control states, interrupt handling is invoked by the signal *intm*, defined by:

$$intm = CA[2] \vee CA[4] \qquad \text{(misaligned access or page fault on load/store).}$$

3. From write-back control states, interrupt handling is invoked when an unmasked overflow occurs:

$$int\_ovf = CA[6].$$

   Note that the masking of the overflow interrupt is already considered when $evn[6]$ is computed.

4. From the "trap" control state, the event signal $evn[5]$ is set, which causes $CA[5]$ to be set as well. The next transition is always to JISR1.

Software support

Software support consists of the Interrupt Service Routine (ISR). The ISR is a DLX program. Correct functionality is obtained by requiring certain properties from the ISR. These requirements are:

1. Termination: every uninterrupted execution of the ISR terminates.
2. Critical sections of the ISR may be interrupted only by a reset interrupt.
3. Obeying priority: an ISR for interrupt $i > 0$ is not interrupted by an interrupt j for which $j \geq i$.

How are these requirements satisfied?

1. Termination is satisfied by correct programming. This is not a trivial task and it is even harder to prove that a program is correct than to write a correct program. Unfortunately, we can offer very limited guidelines how to write correct programs.
2. Critical sections are not interrupted by maskable interrupts simply by disabling maskable interrupts upon entering the critical section. Some non-maskable interrupts can be avoided again by correct programming (misaligned access, illegal instruction, and trap can be avoided, especially during short program segments).

Page faults are avoided during the critical sections by insuring that the program segments of the critical sections and the interrupt stack is stored on a permanent page of memory. A permanent page is never swapped out by the virtual memory management system, and hence, access to the interrupt stack or fetching of instructions in the critical sections do not create page faults.

3. Priority is obeyed by (a) disabling maskable interrupts of lower priority, and (b) avoiding internal interrupts in ISRs of internal interrupts (using the same ideas used to avoid interrupts during the execution of the critical sections).

6.4 Interrupt Handling

Interrupt handling is divided into three stages, as follows.

Save Status

The first two steps of saving the status are implemented in hardware, the rest are implemented in software.

1. Save the PC of the running program. This PC serves as the return address of the ISR. There are two control states the are involved with this step: JISR1 and JISR(-1), and their actions are:

$$\begin{aligned} \text{JISR(-1)} &: PC(t+1) \leftarrow PC(t) - 4 \\ \text{JISR1} &: EPC(t+1) \leftarrow PC(t) \end{aligned}$$

   There is a subtle issue regarding the return address. Recall that the PC is incremented during the "decode" control state. This means that if the "decode" control state has been already executed, then the PC points to the "next" instruction. If the "decode" control state has not been already executed, then the PC points to the "current" instruction. This observation together with whether the interrupt is of type abort/repeat/continue determines which return address should be used, and is reflected in the control in the states JISR1 and JISR(-1).

   In the JISR1 control state, the PC is simply copied to the EPC. However, in the JISR(-1) state, the PC is decremented before it is copied to the EPC (i.e. $EPC \leftarrow PC - 4$) to reverse the increment performed during the "decode" control state so that the return address is the "current" instruction. (Question: check which interrupts go through JISR1 and which interrupts go through JISR(-1)).

2. The JISR1 state is followed by the JISR2 state. During the JISR2 control state three actions take place simultaneously (i.e. during the same clock cycle): enter a critical section, snapshot the status, and jump to the ISR.

   (a) Enter the critical section. This step consists of two actions:

i. Disable the maskable interrupts (i.e interrupts 6-31) by setting

$$SR[31:6](t+1) \leftarrow 0^{26}.$$

ii. Clear the CA: Set clear_CA to 1. This has the effect of setting $CA(t+1)$ as follows:

$$CA(t+1) \leftarrow evn[31:7] \cdot 0^6 \cdot (reset \vee pup).$$

This clears the CA and insures that pending internal interrupts 1-6 will not invoke the interrupt handling mechanism during the critical section.

(b) Snapshot the status:

$$\begin{aligned} ESR(t+1) &\leftarrow SR(t) \\ ECA(t+1) &\leftarrow CA(t) \wedge SR(t) \\ EMAR(t+1) &\leftarrow MAR(t) \end{aligned}$$

(c) Jump to the ISR:

$$PC(t+1) \leftarrow SISR$$

where $SISR$ is the fixed address of the beginning of the ISR. (Question: Where does $SISR$ appear in the DLX?)

3. The ISR starts running, and performs the following steps (Note, this is software - not hardware! Everything below is implemented by regular DLX instructions.):

(a) Allocate a new frame for the interrupt on the interrupt stack (in the main memory). This is done by adding $35 \cdot 4$ (i.e 35 words) to the interrupt stack pointer (recall: there are 4 special registers with backup information and 31 general purposes registers). We need an interrupt stack to support nesting of interrupts.

An interesting issue is where is the interrupt stack pointer stored? There are two options: (a) in a special register - in which case we need to add it to the DLX, or (b) in the data segment of the ISR program. I prefer the second option although it requires loading the interrupt stack pointer. (Can you think of reasons why using a general purpose register is not such a good idea?)

A reset interrupt cannot be interrupted (it has the highest priority), and therefore, we don't have to create a new frame if a reset interrupt is being served. I suspect that allocating a new frame in the interrupt stack is a good idea even in the case of an reset interrupt.

(b) Copy the special registers EPC, ESR, ECA, EMAR to the new frame on the interrupt stack. This is done by using one of the general purpose registers as a "channel" (why?). The instruction movs2i is used to copy a special register to a general purpose register. (How is the value of the general purpose register that is used as a "channel" preserved?). We do this step even if a reset interrupt occurred (why?).

(c) Compute the pending interrupt with the highest priority:

$$\ell = \min\{j : ECA[j] = 1\}$$

(where do we store this value?)

(d) Exit the critical section. Disable all maskable interrupts with priority lower than ℓ:

$$SR[i] = \begin{cases} 1 & \text{if } i < \max\{6, \ell\} \\ 0 & \text{otherwise} \end{cases}$$

We are now ready to serve interrupt ℓ. (Question: do you know how to program these steps using the DLX instruction set?)

Serve the Interrupt

This stage is part of the ISR. The ISR executes the following steps and may be interrupted by an unmasked interrupt.

1. Jump to the segment of the ISR that handles interrupt ℓ.
2. Save the general purpose registers that will be used by the ISR in the interrupt stack.
3. Serve the interrupt.
4. Restore the general purpose registers from the interrupt stack.

Restore Status

This stage is also part of the ISR.

1. Enter a critical section by setting: $SR[31:6](t+1) \leftarrow 0^{26}$.
2. Restore the EPC and the ESR from the interrupt stack. (We will be using a general purpose register for this, and need to restore it from the interrupt stack as well).
3. Free the top frame from the interrupt stack (i.e. subtract $35 \cdot 4$ from the interrupt stack pointer).
4. Exit the critical section. Use the instruction RFE (return from exception) which does the following:

$$\begin{array}{lrcl} RFE1: & SR(t+1) & \leftarrow & ESR(t) \\ RFE2: & PC(t+1) & \leftarrow & EPC(t) \end{array}$$

The next instruction which is fetched is the instruction which should be executed when returning from handling the interrupt (this can be an instruction of a user's program or an instruction of an ISR of an interrupt of a lower priority).

An important point to remember is that the critical sections are not interrupted, except for reset interrupts which cannot be "avoided" since they are external. The occurrence of an interrupt of priority 1-5 during a critical section is a severe error in the ISR.

Question 36 *Why isn't "saving of the general purpose registers" part of "save status"? Similarly, why isn't "restoring the general purpose registers" part of "restore status"?*

Question 37 *Why isn't the cause register restored during Restore Status?*

Question 38 *Design the environment of the special registers. Describe how this environment connects to the busses of the DLX, describe the control signals (and when they are active), and show how this environment can support the functionality needed to support interrupts.*

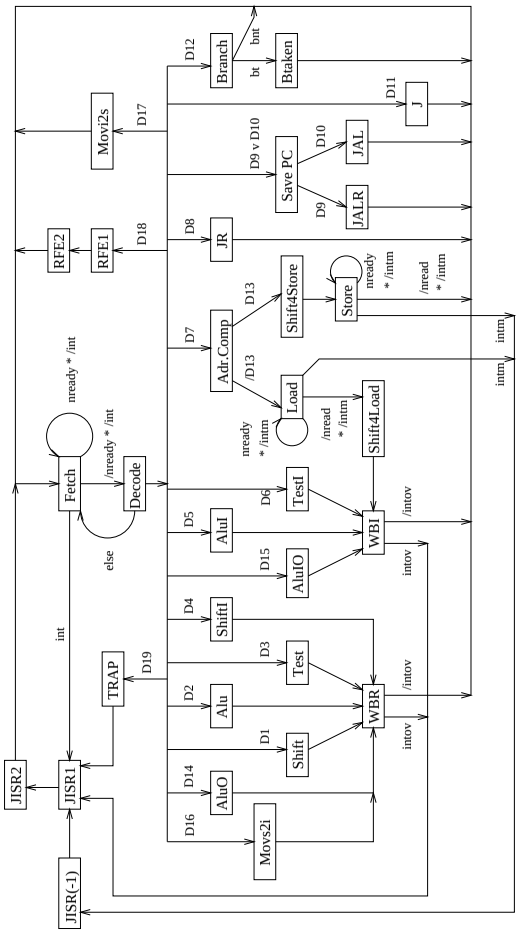


Figure 4: FSD of the DLX design supporting interrupts ($k = 32$ states)

Figure 6.1: The state-diagram of the DLX with support of interrupts from the book of Mueller and Paul

Chapter 7

Correctness of The DLX Interrupt Handling Mechanism

7.1 Terminology

Clock cycle t. We number the clock cycles starting with clock cycle 0.

Which instruction is executed in clock cycle t? This is presumably a trivial question. The DLX executes the instructions one by one. However, interrupts disturb the "normal" execution of instructions. For example, when handling an internal interrupt, the control transitions to the JISR states rather than continue with the execution of the instruction. The following definitions resolve this ambiguity.

Definition 23 *The control* enters *the "fetch" state in clock cycle t if in cycle t the control is in the "fetch" state and in cycle $t-1$ the control is not in the "fetch" state.*

The instruction executed in cycle t, denoted by $I(t)$, is defined as follows:

Definition 24 *Given clock cycle t, define* last_entrance_fetch(t) *as follows:*

$$\text{last\_entrance\_fetch}(t) = \max\{t' \leq t : \textit{ the control enters "fetch" is cycle } t'\}.$$

$I(t)$, the instruction executed in cycle t, is defined as the instruction pointed to by the PC in cycle last_entrance_fetch(t).

Note that *last_entrance_fetch*(t) may not be well defined! If the control does not enter the "fetch" state in the cycles $[0, t]$, then *last_entrance_fetch*(t) is not well defined and neither is $I(t)$. This is not an impossible situation; it occurs shortly after power-up. Such a situation can be avoided by changing the control so that power-up has two effects: (a) the control transitions to the "fetch" state, and (b) the PC is set to SISR (i.e. the starting address of the ISR).

Let $t_1 < t_2$ denote successive entrances to the "fetch" state, then $I(t)$ is constant in the interval $[t_1, t_2)$. The definition of $I(t)$ partitions the clock cycles into intervals, the boundaries of which are entrances into the "fetch" state. Within each such interval, the current instruction $I(t)$ is fixed and well defined.

The next instruction. The next instruction in an execution of a program that is not interrupted is defined by the semantics of the DLX instruction set (note, however, that the address of the next instruction is not always $PC+4$). We denote the instruction that follows instruction I by I^+.

Note that: (a) If I is a trap instruction, then I^+ is the instruction in address $PC+4$ - not the first instruction in the ISR. The reason is that the abstraction of the virtual computer regards a system call as a single instruction (even though we know that a system call requires many instructions). (b) If I is an illegal instruction or its fetch causes a misaligned access, then I^+ is not well defined.

Admissible ISRs The ISR is *admissible* if:

1. Every uninterrupted execution of the ISR terminates.
2. The critical sections are not interrupted by interrupts 1-5.
3. ISR[0]-ISR[5] are not interrupted by interrupts 1-5.

How do we guarantee that interrupts 1-5 do not occur during the critical sections?

- Illegal instructions and misaligned memory accesses are avoided if the critical sections are programmed properly. This is a relatively easy task.
- Page faults during fetch are avoided by requiring that the program segments of the critical sections are stored on a permanent page of memory.
- Page faults during load and store are avoided by requiring that the interrupt stack, as well as other data segments accessed by the critical sections are stored on a permanent page of memory.
- Trap interrupts are avoided during the execution of ISRs of internal interrupts and the execution of the ISR of the reset interrupt. This is done simply by not using the trap instruction in these program segments. (Note that when programming these parts of the ISR the programmer may not use print statement to monitor the values of variables.)

Similar requirements suffice to insure that ISR[1]-ISR[5] are not interrupted by interrupts 1-5.

We summarize which memory addresses are stored on a permanent page of memory.

1. The interrupt stack,
2. The critical sections of the ISR, and
3. The ISRs for interrupts 0-5.

Note that these conditions could be relaxed to allow page faults during ISR(5) that (the segment that handles the trap interrupt).

The current interrupt. Just as we defined the current instruction, we would like to define the current interrupt that is being handled. For this purpose we consider the user's program as interrupt 32. The current interrupt is defined by the *interrupt level*, denoted by $il(t)$.

The interrupt level is defined inductively, starting from the first entrance to the "fetch" state, as follows:

1. Basis: If the interrupt stack is empty in cycle t and the control enters the "fetch" state in cycle t, then $il(t) = 32$.

2. "Push": If the control is in state $JISR2$ in cycle t and in "fetch" state in cycle $t+1$, then
$$il(t+1) = \min\{i : ECA[i](t+1) = 1\}$$
Recall that at JISR2 the ECA register is set to $CA \wedge SR$, hence, ECA holds the information about which ISR should be executed.

3. "Pop": If (a) the control is in state $RFE2$ in cycle t and in "fetch" state in cycle $t+1$; and (b) the interrupt stack is not empty in cycle $t+1$, then
$$il(t+1) = \min\{i : eca[i](t) = 1\}$$
where $eca[i]$ is the backup of the ECA stored in top frame in the interrupt stack. Note that if the interrupt stack is empty, then the "basis" case is chosen (i.e. $il(t+1) = 32$).

4. In all other cases: $il(t+1) = il(t)$.

Note that the interrupt level changes only during an entrance to the "fetch" state from states $JISR2$ and $RFE2$. Hence, the interrupt level during the execution of instruction I is fixed. During the period between power-up and the entrance to "fetch" with $PC = SISR$ the interrupt level is not well defined (one reason is that the interrupt stack is not in a well defined state). We are not concerned about this situation because we prove later that this undefined period is bounded.

Waiting for service.

Definition 25 *1. Interrupt i* occurs *in clock cycle t if $evn[i](t) = 1$ and $evn[i](t-1) = 0$.*

2. Interrupt i is caught *in clock cycle t if $CA[i](t+1) = 1$ (namely, $CA[i]$ sampled a 1 in the end of cycle t and outputs a 1 in the beginning of cycle $t+1$).*

3. An interrupt j is served *in cycle t if*
$$il(t) = j \text{ or } il(t) = i \text{ for an interrupt } i \text{ of type "abort"}$$

4. An interrupt j is pending *(i.e. waiting for service) in cycle t if (a) it was caught in cycle $t_1 \leq t$; and (b) it was not served in the interval $[t_1, t]$.*

Machine State. The machine's state refers to the contents of the registers that effect functionality from the programmer's point of view. Therefore, the machine's state is defined as follows.

Definition 26 *The state of the DLX is the contents of the general purpose registers* $R1-R31$ *and the registers* PC *and* SR.

Note that the CA and MAR are not part of the state, although the MAR and CA are backed-up; they are backed-up to help the ISR, but they are not restored later.

Wait states (WS). An access to the memory may take more than one cycle. Consider, for example, fetching of an instruction. The control remains in the state "fetch" until the IR register is loaded with the instruction to be executed. The number of consecutive self-loop transitions "fetch" $\rightarrow$ "fetch" is called the number of *wait states*.

We use the same definition for the self-loops in the states "load" and "store". Namely, the number of wait states is the number of consecutive transitions $s \rightarrow s$, where s is a a state $s \in \{$"fetch","load", "store"$\}$.

We denote the number of wait states by WS. For simplicity, we assume that WS is constant.

Note that unless interrupts are detected during fetch/load/store, the control will stay in these states for $WS + 1$ cycles. If $WS = 0$, then the control stays in the fetch/load/store states for only one clock cycle, namely, the self-loop is not never used.

7.2 Specification of the interrupt handling mechanism

Before proving correctness, we need to define what we expect the interrupt handling mechanism to do. In fact, we should have specified the mechanism before we described it. Due to the complexity of the interrupt handling mechanism we first described what we would like it to do and how this is done. Now we formally specify properties of this mechanism. We regard a mechanism that satisfies these specifications as correct.

Response time. How many cycles elapse from the cycle in which an interrupt occurs till it is served? We consider 3 cases:

1. A reset interrupt that occurs in cycle t is served in cycle $t' \leq t + 9 + WS$.

2. Consider an external input/output interrupt $i > 6$ (not reset) that is caught in cycle t. Let t_2 denote the closest entrance to "fetch", namely,

$$t_2 = \min\{t' \geq t : \text{state}(t') = \text{"fetch"}\}.$$

   We demand that if interrupt i is enabled in cycle t_2 (i.e. $SR[i](t_2) = 1$), then there exists an interrupt $j \leq i$ that is served in cycle $t' \leq t + 9 + WS$.

3. An internal interrupt i that occurs in cycle t is served in cycle $t' \leq t + 4$. (Note that an internal interrupt must be enabled to occur.)

Obeying priority. When can the execution of $ISR(i)$ be interrupted? Loosely speaking, we allow $ISR(i)$ to be interrupted only by an interrupt $j < i$. We make this requirement more detailed as follows:

1. $ISR(0)$ can be interrupted only by a reset interrupt.
2. $ISR(i)$ for $i \in \{1, \ldots, 5\}$, can only be interrupted by a reset interrupt. Recall that admissibility implies that an interrupt $j \in \{1, \ldots, 5\}$ does not occur while $ISR(i)$ is running.
3. $ISR(i)$ for $i \geq 6$ can be interrupted only by an enabled interrupt j, for which $j < i$.

Finite memory. The interrupt stack never contains more than 32 frames (the size of each frame is 35 words). This condition is important because we require that the interrupt stack be stored in a page that is not swapped out. Hence, we want to verify that the interrupt stack is not too big.

Preciseness. Preciseness refers to the state of the DLX after the interrupt is served. Suppose that interrupt j is served in the interval $[t+1, t')$. Let t_0 denote the cycle in which the execution of instruction $I(t)$ started. Figure 1 depicts the required state of the DLX in cycle t' after the return from exception. The intuition is that one can "cut" the segment between the two cycles in which the DLX state is S and obtain an uninterrupted execution.

Termination. Assume that only a finite number of interrupts occur. An execution of $ISR(i)$ that serves interrupt i either terminates or is aborted due to the occurrence of an interrupt of type "abort" while it is running.

Completeness. Assume that only a finite number of interrupts occur. Every internal interrupt is served eventually. A pending external interrupt is served eventually if it is enabled long enough.

7.3 Correctness Proof

In this section we prove that the interrupt handling mechanism described in Chapter 6 is correct; namely, it satisfies the specifications.

7.3.1 response time

Let state(t) denote the state of the control in cycle t. We first prove the following claim.

Claim 24 *Let $t_1 < t_2$ denote two cycles of successive entrances to the fetch state. If state$(t_2 - 1) \neq$ JISR2, then*

$$t_2 - t_1 \leq 6 + 2 \cdot WS.$$

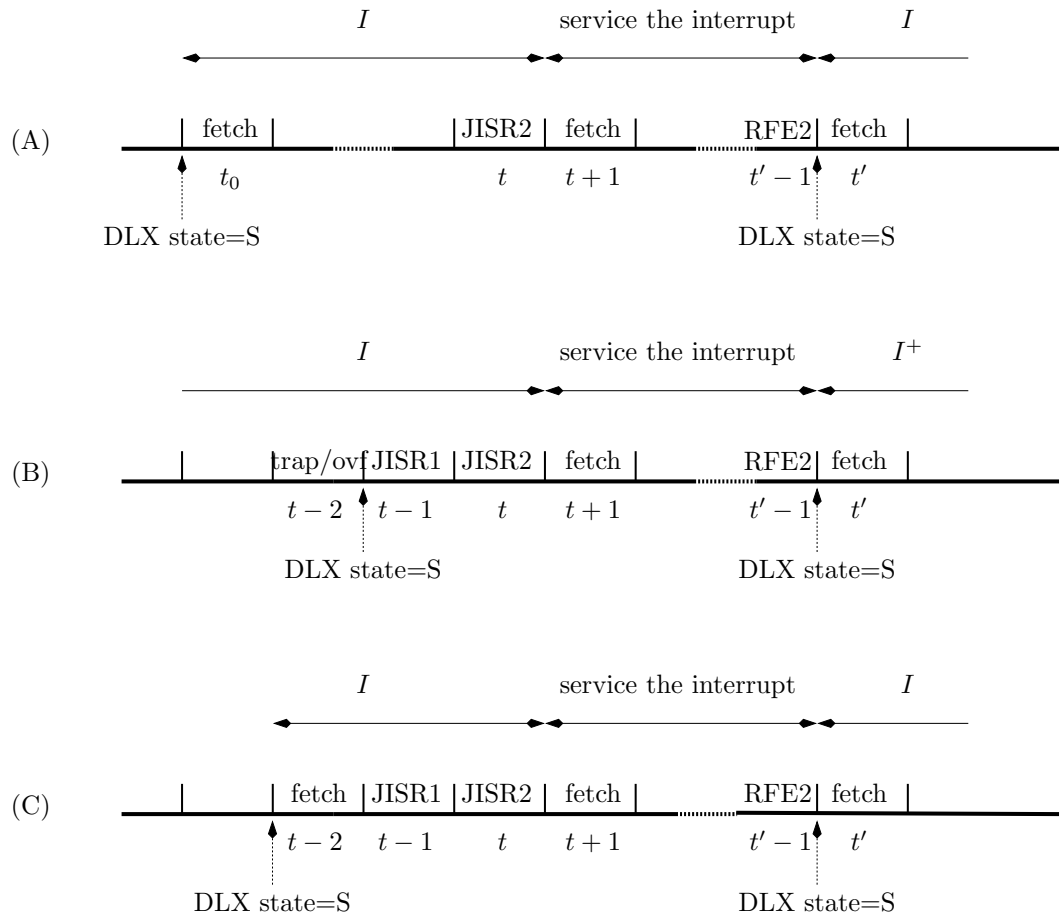


Figure 7.1: Preciseness requirement on state after RFE: (A) internal interrupt of type "repeat" (B) internal interrupt of type "continue" (C) external interrupt (not reset)

Proof: Compute the longest simple cycle from fetch to fetch in the state diagram of the control after removing the state $JISR2$. We denote the number of cycles x the control stays in state A before moving to state B by $A \xrightarrow{x} B$. The longest simple cycle is as follows:

$$\text{fetch} \overset{WS+1}{\rightarrow} \text{decode} \overset{1}{\rightarrow} \text{Adr. Comp.} \overset{1}{\rightarrow} \text{load} \overset{WS+1}{\rightarrow} \text{shift4store} \overset{1}{\rightarrow} \text{WBI} \overset{1}{\rightarrow} \text{fetch}$$

□

Corollary 25 *Assume that state$(t) \neq$ "fetch". Let $t_1 < t_2$ denote two cycles of successive entries to the fetch state for which $t_1 < t < t_2$. If state$(t_2 - 1) \neq$ JISR2, then*

$$t_2 - t \leq WS + 5.$$

Note that Coro. 25 holds even if t_1 is not defined because of power-up. In this case, power-up occurs in cycle t, and the machine enters fetch for the first time in cycle t_2 (but this entrance is not from JISR2).

The following claim addresses the rather special case that (a) a reset interrupt is pending, (b) the control enters state "fetch" from state JISR2, and (c) the interrupt level is not zero upon entry to "fetch".

Claim 26 *Assume that a reset interrupt occurs in cycle t (namely, $reset(t) = 1$). Let t_2 denote the first cycle in which the control enters the "fetch" state after cycle t (namely, $t < t_2$). If $il(t_2) \neq 0$ and $state(t_2 - 1) = JISR2$, then $t = t_2 - 1$.*

Proof: If $il(t_2) \neq 0$, then $ECA[0](t_2) = 0$. Since the state of the control is JISR2 in cycle $t_2 - 1$, it follows that $CA[0](t_2 - 1) = 0$. But $reset(t) = 1$ implies that $CA[0](t + 1) = 1$. We would like to argue that the only way $CA[0](t_2 - 1) = 0$ and $CA[0](t + 1) = 1$ is that $t + 1 > t_2 - 1$. If this is true, then $t_2 > t > t_2 - 2$, and hence, $t = t_2 - 1$, as required.

Assume that $t + 1 \leq t_2 - 1$. The value of $CA[0]$ does not change in cycle $(t + 1)$ from 1 to 0 in cycle $t_2 - 1$ unless a *clear_CA* is active in a cycle $t' \in [t + 1, t_2 - 2]$. A *clear_CA* is active only in the JISR2 state. The only cycle in the interval $[t + 1, t_2]$ in which the control can be in the JISR2 state is cycle $t_2 - 1$. Hence the control is not in the JISR2 state during the interval $[t + 1, t_2 - 2]$, a contradiction, and the claim follows. □

How does the process of servicing an interrupt begin? There are two cases: (1) The control is in the "fetch" state and the *int* signal is raised causing a transition to JISR1, JISR2, and then upon entry to "fetch" again an interrupt is served. (2) The control is in "load", "store", "WBI", or "WBR" states and the corresponding internal interrupt is caught. This interrupt causes a transition to JISR(-1) or JISR1, and then upon entry to "fetch" an interrupt is served.

We analyze the response times for the various types of interrupts as follows:

1. A reset interrupt. First, observe that the occurrence of a reset interrupt is always caught in the CA register, even during a JISR2 cycle (in which the CA register is cleared). Second, observe that before the CA register is cleared, it is copied to the ECA register. So an already caught reset interrupt occurrence is not lost.

   Suppose that the reset interrupt occurs in cycle $t \in [t_1, t_2)$, where $t_1 < t_2$ denote two successive entries to the fetch state. (If power-up occurs in cycle t, then t_1 might not be defined.) We consider the following cases:

   (a) $t < t_2 - 1$, or equivalently, $state(t + 1) \neq$"fetch". We consider the following sub-cases:

   - $il(t_2) = 0$, namely, the reset interrupt is served in cycle t_2. We can bound $t_2 - (t+1)$ by observing that state$(t+1) \neq$ "fetch". By Coro. 25, The longest path in the control from a non-"fetch" state to the "fetch" state is $WS + 5$ cycles. Hence,

     $$t_2 - t = 1 + (t_2 - (t + 1)) \leq 1 + WS + 5 = WS + 6.$$

   - $il(t_2) \neq 0$ and $state(t_2-1) \neq$"JISR2". According to Corollary 25, $t_2-(t+1) \leq 5+WS$. Since $CA[0](t_2) = 1$ (i.e. the reset interrupt is pending) the following transitions are made (starting from cycle t_2):

$$\text{fetch} \xrightarrow{1} \text{JISR1} \xrightarrow{1} \text{JISR2} \xrightarrow{1} \text{fetch}$$

Therefore, in cycle t_2+3, the reset interrupt is served, and $(t_2+3)-t \leq 9+WS$, as required.

- $il(t_2) \neq 0$ and $state(t_2 - 1) =$"JISR2". Claim 26 implies that $t = t_2 - 1$. But this contradicts our assumption that $t < t_2 - 1$.

(b) Suppose that $t = t_2 - 1$, namely, $state(t + 1) =$"fetch". Then in cycle $t_2 = t + 1$ the control signal *int* equals 1, and the following transitions are made (starting from cycle $t + 1$):

$$\text{fetch} \xrightarrow{1} \text{JISR1} \xrightarrow{1} \text{JISR2} \xrightarrow{1} \text{fetch}$$

Therefore, in cycle $t + 4$ the reset interrupt is served.

2. An external interrupt (not reset). The proof follows the same type of arguments; the main difference is that we require that some interrupt be served instead of a specific interrupt.

   The following claim is the analogue of Claim 26 (try to prove it!).

   Claim 27 *Assume that interrupt i, for $i \geq 7$, occurs in cycle t (namely, $evn[0](t) = 1$). Let t_2 denote the first cycle in which the control enters the "fetch" state after cycle t (namely, $t < t_2$). If $il(t_2) = 32$ and $state(t_2 - 1) = JISR2$, then $t = t_2 - 1$.*

   The proof now follows the same cases (make sure you can fill the proof): (a) $t < t_2 - 1$ and (i) $il(t) < 32$; (ii) $il(t) = 32$ and $state(t_2 - 1) \neq JISR2$; or (iii) $il(t) = 32$ and $state(t_2 - 1) = JISR2$. (b) $t = t_2 - 1$.

3. An enabled internal interrupt. If an enabled internal interrupt i occurs in cycle t, then in cycle $t + 1$ the control transitions to state JISR1 or JISR(-1).

   An exception is the illegal instruction interrupt, in which the control moves to the "fetch" state and from there to the JISR1 state. Therefore, in time $t_2 = t + 3$ (or $t_2 = t + 4$), the control enters the fetch state and serves an interrupt j.

   The easy case is that $i = j$ or interrupt j is an aborting interrupt. Is it possible that $i \neq j$ and j is non-aborting? We claim that this is impossible. This could only happen if $i, j \in \{3, 4, 5, 6\}$ (i.e. pff,pfls,trap,ovf). But interrupts 3-6 occur in different control states, and therefore, cannot occur simultaneously and cannot be pending together.

7.3.2 priority

Consider interrupts $0 - 5$. During the service of interrupt $i \in \{0, \ldots, 5\}$, all the maskable interrupts are disabled, so they can't interrupt $ISR[i]$. Since the ISR's are admissible, interrupts 1-5 do not occur during $ISR[i]$, this completes the first two parts.

Consider a maskable interrupt j. The critical section starts by disabling all the maskable interrupts and ends by disabling all the interrupts with priority less than or equal to j. This masking is kept until the service terminates, so if $ISR[j]$ is interrupted by $ISR[i]$, then $i < j$ and interrupt i is enabled, and we are done.

7.3.3 finite space

Nesting of interrupts implies that the interrupt levels increases. So there can be at most one frame for each interrupt in the range 31-0, which implies at most 32 frames in the interrupt stack. The size of each frame is 35 words. A more careful argument is to observe that interrupts $1-5$ never have frames simultaneously in the interrupt stack.

7.3.4 preciseness

We consider the three cases:

1. An internal interrupt i of type repeat. Interrupt i is a page fault either on fetch or on load/store. In both cases the general-purpose registers and the SR register are not changed by the current execution of the instruction. The PC is not changed in page fault on fetch, and therefore, JISR1 copies the PC. The PC is incremented in page fault on load/store (during the decode state) and therefore, JISR(-1) decrements the PC back before it is copied.

2. An internal interrupt of type continue. This interrupt occurs after the PC is incremented and after the general purpose register is updated (in overflow). Therefore, a transition to JISR1 ensures that the CPU state after executing the current instruction is copied.

3. An external interrupt of type continue. The control checks for an external interrupt only in the fetch state in cycle t. The instruction $I(t)$, the execution of which starts when the control enters the fetch state, is not executed at all because the control moves to JISR1,JISR2,fetch, and servicing of the interrupt starts in cycle $t+3$. This means that when the CPU returns from the ISR, the CPU state should be restored to the CPU state in cycle t and instruction $I(t)$ should be restarted. The JISR1 and JISR2 states in cycles $t+1$ and $t+2$ copy the state of the CPU in cycle t, as required.

7.3.5 termination

We wish to prove that $ISR(j)$ terminates or that the program is aborted.

The proof is by double induction! First, we claim that $ISR(i)$ terminates or is interrupted by an aborting interrupt. This claim is proved by induction on i. If $i=0$, then obeying priorities implies that $ISR(i)$ can only be interrupted by another reset interrupt. Assuming that the number of interrupts is finite, the last reset interrupt is not interrupted and terminates, as required. A similar argument holds for the other interrupts of type abort (i.e., $i=1,2$).

The (first) induction hypothesis is that $ISR(j)$, for $j \leq i$, terminates or is interrupted by an aborting interrupt. The induction step is to prove that the same holds for $ISR(i+1)$. The induction step is proved by induction on n - the number of interrupts that have not occurred yet (again, we are assuming that only a finite number of interrupts occur).

The induction basis, for $n=0$, is easy because $ISR(i+1)$ is not interrupted at all, so it terminates. The (second) induction hypothesis is that if at most n interrupts have not

occurred yet when $ISR(i+1)$ is running, then $ISR(i+1)$ either terminates or is interrupted by an aborting interrupt. The induction step for $n+1$ is as follows: the next interrupt that occurs has priority ℓ, and we consider two cases:

1. $\ell \geq i+1$ or interrupt ℓ is disabled. In this case, our claim on priorities implies that interrupt ℓ does not interrupt $ISR(i+1)$, and hence, the remaining number of interrupts decreases by one, and we apply the induction hypothesis (on n), to conclude that $ISR(i+1)$ terminates or is interrupted by an interrupt of type abort.

2. $\ell < i+1$ and interrupt ℓ is enabled. In this case, interrupt ℓ interrupts $ISR(i+1)$. If interrupt ℓ is of type abort, then we are done. Otherwise, the first induction hypothesis implies that $ISR(\ell)$ either terminates or is interrupted by an interrupt of type abort. If $ISR(\ell)$ terminates, then $ISR[i+1]$ is resumed with at most n remaining interrupts, and the second induction hypothesis (on n) implies that $ISR(i+1)$ terminates or is interrupted by an interrupt of type abort. If $ISR(\ell)$ is interrupted by an interrupt of type abort, then so is $ISR(i+1)$, and we are done.

7.3.6 Completeness

A reset interrupt or an enabled internal interrupt is served after at most $WS+9$ cycles.

We prove that an external interrupt i that occurs in cycle t is served after a finite time if it is enabled long enough. The proof is by induction on n, where n denotes the number of pending interrupts in cycle t plus the number of interrupts that occur during cycle t or after it. The induction basis for $n=1$ is easy because after $WS+9$ cycles an interrupt is served, and the interrupt that is served must be i.

The induction step is proved as follows. If the interrupt ℓ that is served after $WS+9$ cycles does not serve to interrupt i, then $ISR[\ell]$ terminates. After $ISR[\ell]$ terminates, there are less than n interrupts left, and the induction hypothesis implies that interrupt i is served after a finite time.

7.4 Power-up

Can we guarantee that the state of the DLX is in a pre-defined state shortly after power-up? The response time specification implies that, at most $9+WS$ cycles after power-up, the reset signal is served. Assume that this execution of $ISR(0)$ is not interrupted (it can be interrupted only by a reset interrupt). This means that the state of the DLX can be fixed by $ISR(0)$, except for the content of the cause register CA (external interrupts may occur all the time). The amount of time it takes to start servicing the reset interrupt upon power-up can be shortened by causing a reset of the state of the control to the fetch state whenever $pup = 1$.

Our assumption that $ISR(0)$ is not interrupted could be somewhat relaxed to deal with a finite (and small) number of reset interrupts. This can happen if, for example, the reset button causes a burst of power-up signals.

Note that executing $ISR(0)$ upon power-up makes sense only if part of the memory (including $SISR$) is non-volatile read-only memory.