Chapter 4

Decoders and Encoders

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

4.1 Notation

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

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

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

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

Consider a gate G with two input terminals a and b and one output terminal z. The

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

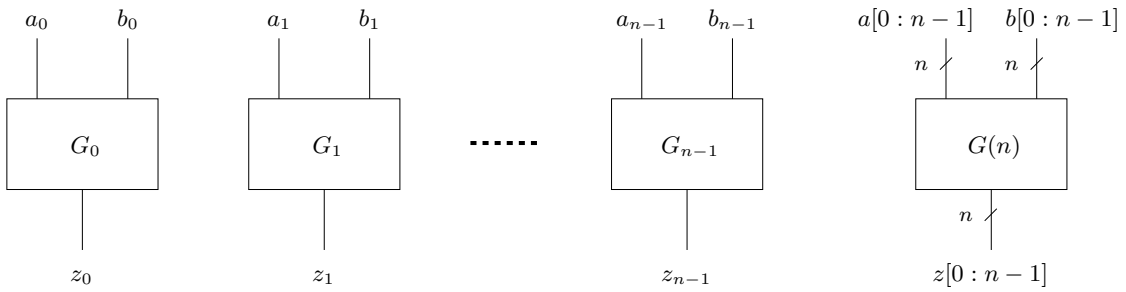

Figure 4.1: Vector notation: multiple instances of the same gate.

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

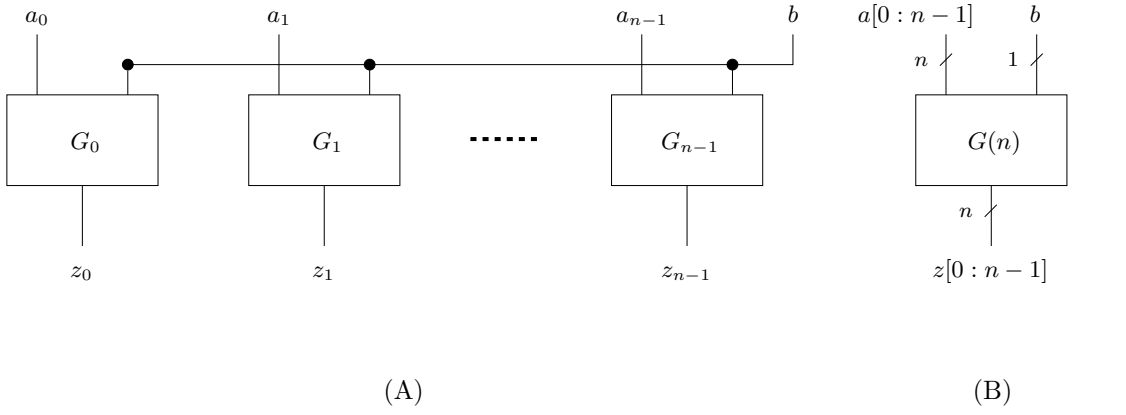

Figure 4.2: Vector notation: b feeds all the gates.

Note that the fanout of the net that carries the signal b in Figure 4.2 is linear. In practice, a large fanout increases the capacity of a net and causes an increase in the delay of the circuit. We usually ignore this phenomena in this course.

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

Example 3 *Consider the following examples of string concatenation:*

- *If* $a = 01$ *and* $b = 10$*, then* $a \cdot b = 0110$*.*
- *If* $a = 1$ *and* $i = 5$*, then* $a^i = 11111$*.*
- *If* $a = 01$ *and* $i = 3$*, then* $a^i = 010101$*.*

4.2 Values represented by binary strings

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

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

Definition 21 *The* value represented in binary representation *by a binary string* $a[n-1:0]$ *is denoted by* $\langle a[n-1:0]\rangle$*. It is defined as follows*

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

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

Definition 22 Binary representation *using* n*-bits is a function* $\mathrm{bin}_n : \{0,1,\ldots,2^n-1\} \to \{0,1\}^n$ *that is the inverse function of* $\langle\cdot\rangle$*. Namely, for every* $a[n-1:0] \in \{0,1\}^n$*,*

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

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

Claim 13 *Let* $x[n-1:0] \in \{0,1\}^n$ *denote a binary string. Let* i *denote the number represented by* $\vec{x}$ *in binary representation, namely,* $i = \langle x[n-1:0]\rangle$*. Let* k *denote an index such that* $0 \le k \le n-1$*. Let* q *and* r *denote the quotient and remainder, respectively, when dividing* i *by* 2^k*. Namely,* $i = 2^k \cdot q + r$*, where* $0 \le r < 2^k$*.*

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

$$x_R[k-1:0] \leftarrow x[k-1:0]$$
$$x_L[n-k-1:0] \leftarrow x[n-1:k].$$

Then,

$$q = \langle x_L[n-k-1:0]\rangle$$
$$r = \langle x_R[k-1:0]\rangle.$$

4.3 Decoders

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

Definition 23 *A* decoder with input length n *is a combinational circuit specified as follows:*

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

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

Functionality:

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

Note that the number of outputs of a decoder is exponential in the number of inputs. Note also that exactly one bit of the output $\vec{y}$ is set to one. Such a representation of a number is often termed *one-hot encoding* or 1*-out-of-k encoding*.
We denote a decoder with input length n by DECODER(n).

Example 4 *Consider a decoder* DECODER(3)*. On input* $x = 101$*, the output* y *equals* 00100000*.*

4.3.1 Brute force design

The simplest way to design a decoder is to build a separate circuit for every output bit $y[i]$. We now describe a a circuit for computing $y[i]$. Let $b[n-1:0]$ denote the binary representation of i (i.e. $b[n-1:0] = bin(i)$).
Define $z_0[n-1:0]$ and $z_1[n-1:0]$ as follows:

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

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

Claim 14

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

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

The brute force decoder circuit consists of (i) n inverters used to compute $\vec{z}_0$, and (ii) an AND(n)-tree for every output $y[i]$. The delay of the brute force design is $t_{pd}(\text{INV}) + \lfloor \log_2 t_{pd}(\text{AND}) \rfloor$. The cost of the brute force design is $\Theta(n \cdot 2^n)$, since we have an AND(n)-tree for each of the 2^n outputs.

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

4.3.2 An optimal decoder design

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

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

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

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

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

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

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

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

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

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

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

4.3.3 Correctness

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

Claim 15 *The* DECODER(n) *design is a correct implementation of a decoder.*

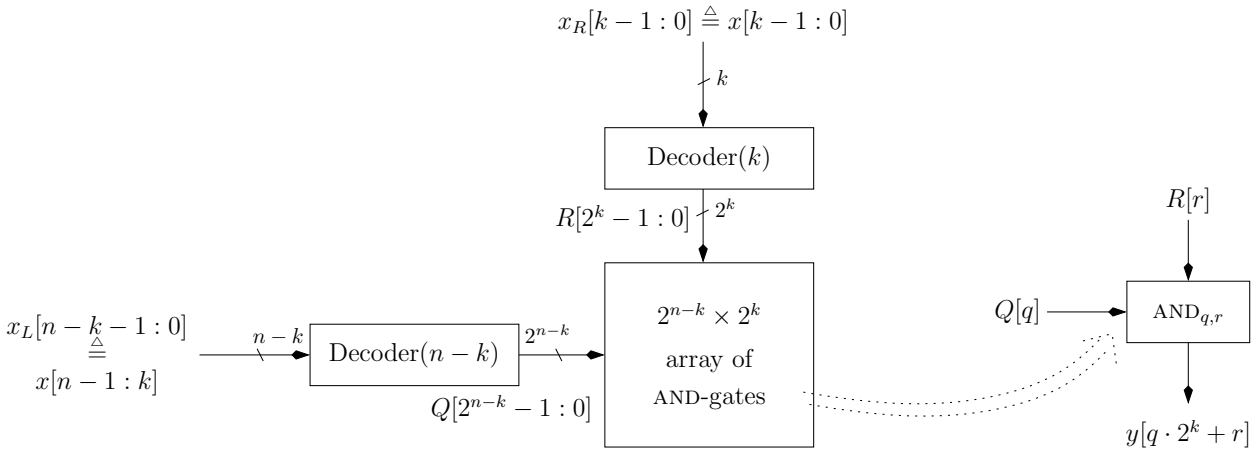

Figure 4.3: A recursive implementation of DECODER(n).

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

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

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

By Claim 13,

$$q = \langle x_L[n-k-1:0] \rangle$$
$$r = \langle x_R[k-1:0] \rangle.$$

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

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

and the claim follows. □

4.3.4 Cost and delay analysis

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

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

It follows that

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

Obviously, $c(n) = \Omega(2^n)$ (regardless of the value of k), so the best we can hope for is to find a value of k such that $c(n) = O(2^n)$. The obvious choice is to choose $k = n/2$. If n is odd, then we choose $k = \lfloor n/2 \rfloor$. Assume that n is a power of n, namely, $n = 2^\ell$. If $k = n/2$ we open the recurrence to get:

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

The last line is justified by

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

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

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

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

Question 18 *Prove that* DECODER(n) *is asymptotically optimal with respect to cost and delay.*

4.4 Encoders

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

Definition 24 *The weight of a binary string equals the number of non-zero symbols in the string. We denote the weight of a binary string $\vec{a}$ by* $\text{wt}(\vec{a})$*.Formally,*

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

We define the encoder partial function as follows.

Definition 25 *The function* $\text{ENCODER}_n : \{\vec{y} \in \{0,1\}^{2^n} : \text{wt}(\vec{y}) = 1\} \to \{0,1\}^n$ *is defined as follows:* $\langle \text{ENCODER}_n(\vec{y}) \rangle$ *equals the index of the bit of* $\vec{y}$ *that equals one. Formally,*

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

Definition 26 *An* encoder *with input length* 2^n *and output length* n *is a combinational circuit that implements the Boolean function* ENCODER_n*.*

An encoder can be also specified as follows:

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

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

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

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

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

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

Example 5 *Consider an encoder* $\text{ENCODER}(3)$*. On input* 00100000*, the output equals* 101*.*

4.4.1 Implementation

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

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

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

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

The idea is to feed these two parts into encoders $\text{ENCODER}'(n/2)$ (see Figure 4.4). However, there is a problem with this approach. The problem is that even if $\vec{y}$ is a "legal" input (namely, $wt(\vec{y}) = 1$), then one of the strings $\vec{y}_L$ or $\vec{y}_R$ is all zeros, which is not a legal input. An "illegal" input can produce an arbitrary output, which might make the design wrong.

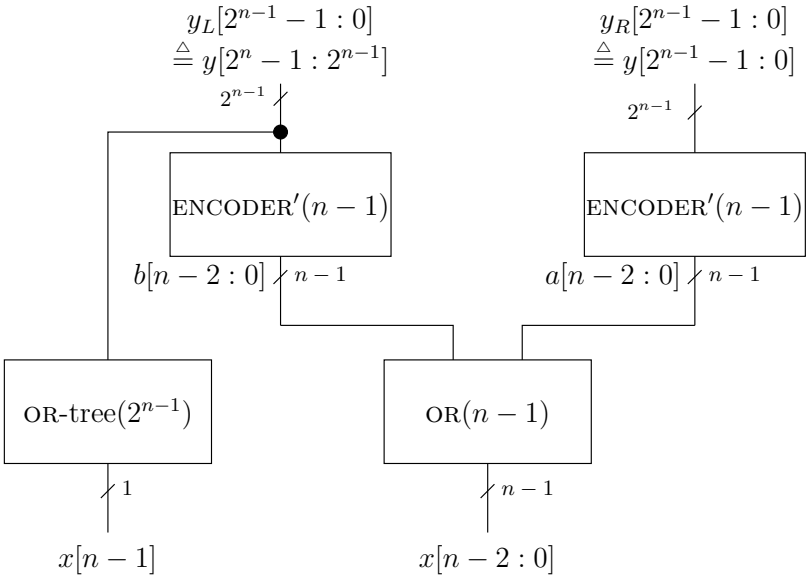

Figure 4.4: A recursive implementation of ENCODER$'(n)$.

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

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

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

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

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

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

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

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

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

Claim 16 *The circuit* ENCODER$'(n)$ *depicted in Figure 4.4 implements the Boolean function* ENCODER$_n$.

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

Question 19 *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 20 *An* ENCODER$'(n)$ *contains an* OR*-tree*(2^n) *and an* ENCODER$^*(2^{n-1})$ *that are fed by the same input. This implies that if we "open the recursion" we will have a chain of* OR*-trees, where small trees are sub-trees of larger trees. This means that an* ENCODER$'(n)$ *contains redundant duplications of* OR*-trees. Analyze the reduction in cost that one could obtain if duplicate* OR*-trees are avoided. Does this reduction change the asymptotic cost?*

Question 19 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 17 *If* $\text{wt}(y[2^n - 1 : 0]) \leq 1$*, then*

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

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

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

However,

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

and the claim follows. □

Figure 4.5 depicts the design ENCODER$^*(n)$ obtained from ENCODER$'(n)$ after commuting the OR and the ENCODER$(n-1)$ operations. We do not need to prove the correctness of the ENCODER$^*(n)$ circuit "from the beginning". Instead we rely on the correctness of ENCODER$'(n)$ and on Claim 17 that shows that ENCODER$'(n)$ and ENCODER$^*(n)$ are functionally equivalent.

Question 21 *Provide a direct correctness proof for the* ENCODER$^*(n)$ *design (i.e. do not rely on the correctness of* ENCODER$'(n)$*).*

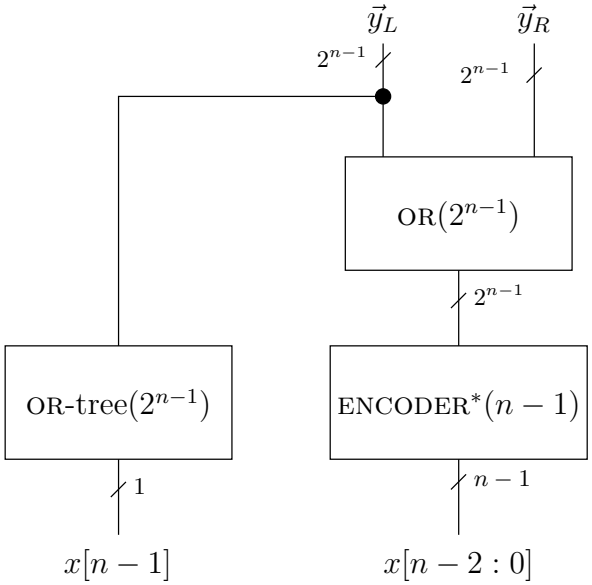

Figure 4.5: A recursive implementation of ENCODER$^*(n)$.

The following questions are based on the following definitions:

Definition 27 *A binary string* $x'[n-1:0]$ dominates *the binary string* $x''[n-1:0]$ *if*

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

Definition 28 *A Boolean function* f *is* monotone *if* x' *dominates* x'' *implies that* $f(x')$ *dominates* $f(x'')$.

Question 22 *Prove that if a combinational circuit* C *contains only gates that implement monotone Boolean functions (e.g. only* AND*-gates and* OR*-gates, no inverters), then* C *implements a monotone Boolean function.*

Question 23 *The designs* ENCODER$'(n)$ *and* ENCODER$^*(n)$ *lack inverters, and hence are monotone circuits. However, the Boolean function corresponding to an encoder is not monotone. Can you resolve this contradiction?*

4.4.2 Cost and delay analysis

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

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

We expand this recurrence to obtain:

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

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

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

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

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

The following question deals with lower bounds for an encoder.

Question 24 *Prove that the cost and delay of* $\text{ENCODER}^*(n)$ *are asymptotically optimal.*

4.5 Summary

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

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

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

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

Three main techniques were used in this chapter.

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

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

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