Chapter 5

Combinational modules

5.1 Multiplexers

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

Definition 29 *A* MUX*-gate is a combinational gate that has three inputs $D[0], D[1], S$ and one output Y. The functionality is defined by*

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

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

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

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

Functionality:

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

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

Example 6 *Let $n = 4$, $D[3:0] = 0101$, and $S[1:0] = 11$. The output Y should be 1.*

5.1.1 Implementation

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

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

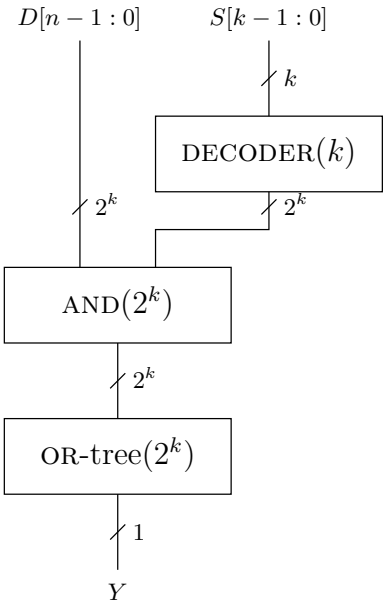

Figure 5.1: An (n:1)-MUX based on a decoder ($n = 2^k$).

Question 25 *The following question deals with the implementation of (n:1)-*MUX *suggested in Figure 5.1.*

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

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

Question 26 *Answer the same questions asked in Question 25 but this time with respect to the implementation of the (n:1)-*MUX *suggested in Figure 5.2.*

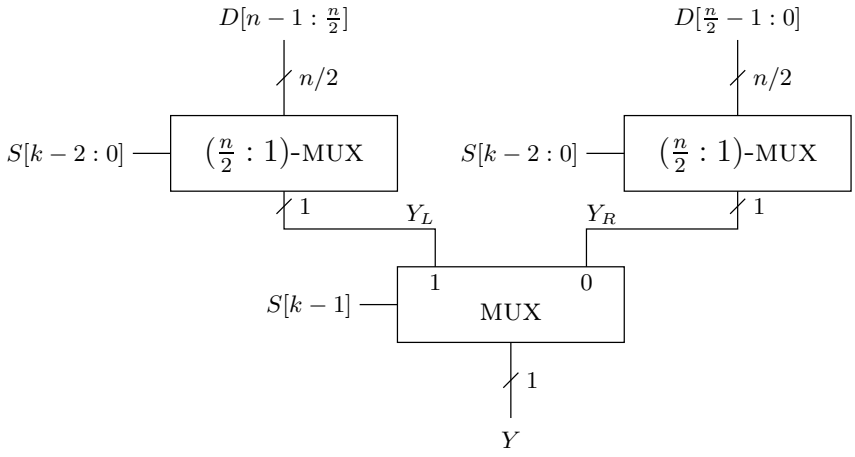

Figure 5.2: A recursive implementation of (n:1)-MUX ($n = 2^k$).

Both implementations suggested in this section are asymptotically optimal with respect to cost and delay. Which design is better? A cost and delay analysis based on the cost and delay of gates listed in Table 2.1 suggests that the tree-like implementation is cheaper and faster. Nevertheless, our model is not refined enough to answer this question sharply. On one hand, the tree-like design is simply a tree of muxes. The decoder based design contains, in addition to an OR(n)-tree with n inputs, also a line of AND-gates and a decoder. So one may conclude that the decoder based design is worse. On the other hand, OR-gates are typically cheaper and faster than MUX-gates. Moreover, fast and cheap implementations of MUX-gates in CMOS technology do not restore the signals well; this means that long paths consisting only of MUX-gates are not allowed. We conclude that the model we use cannot be used to deduce conclusively which multiplexer design is better.

Question 27 *Compute the cost and delay of both implementations of (n:1)-*MUX *based on the data in Table 2.1.*

5.2 Cyclic Shifters

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

Definition 30 *The string* $b[n-1:0]$ *is a* cyclic left shift by i positions *of the string* $a[n-1:0]$ *if*

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

Example 7 *Let* $a[3:0] = 0010$. *A cyclic left shift by one position of* $\vec{a}$ *is the string* 0100. *A cyclic left shift by* 3 *positions of* $\vec{a}$ *is the string* 0001.

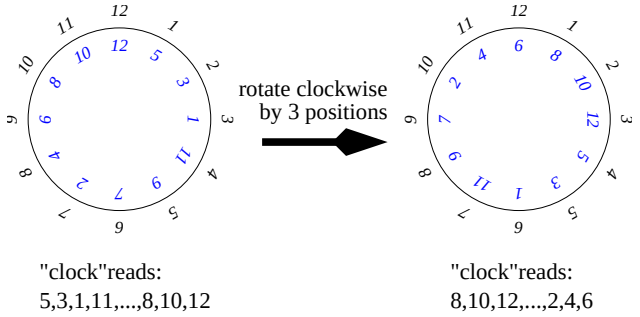

Figure 5.3: An example of a cyclic shift. The clock "reads" the numbers stored in each clock notches in clockwise order starting from the one o'clock notch.

Definition 31 *A* BARREL-SHIFTER(n) *is a combinational circuit defined as follows:*

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

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

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

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

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

5.2.1 Implementation

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

Definition 32 *A* CLS(n, i) *is a combinational circuit defined as follows:*

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

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

Functionality:

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

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

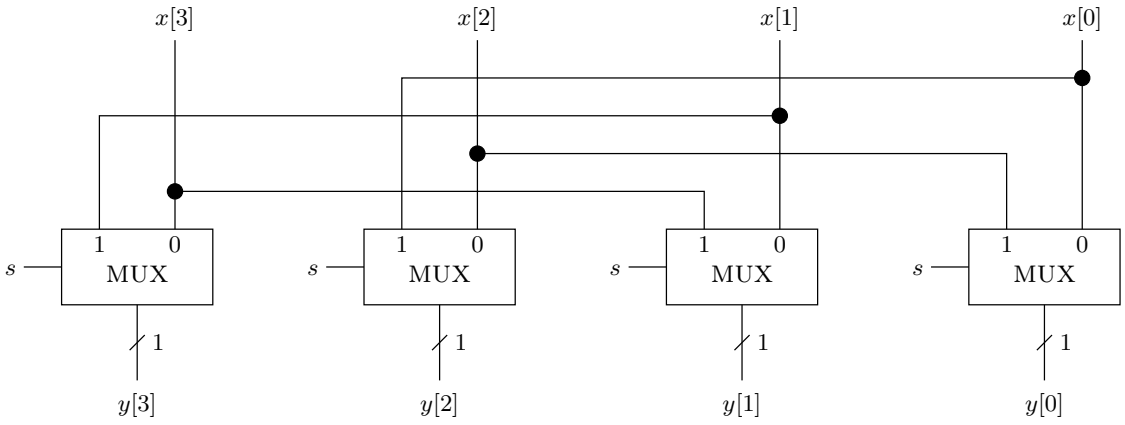

Figure 5.4: A row of multiplexers implement a CLS$(4, 1)$.

self-evident that the main complication with the design of CLS$(n.i)$ is routing (i.e. drawing the wires).

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

Claim 18 *Define the strings $y_i[n-1:0]$, for $0 \leq i \leq k-1$, recursively as follows:*

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

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

Proof: The proof is by induction on i. The induction basis, for $i = 1$, holds because of the definition of CLS$(2, 0)$.

The induction step is proved as follows.

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

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

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

Note that

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

Therefore

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

and the claim follows. □

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

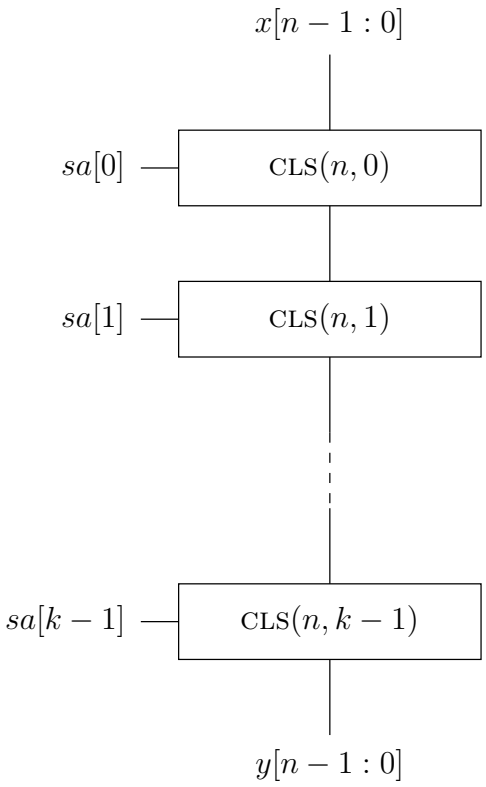

Figure 5.5: A BARREL-SHIFTER(n) built of k levels of CLS(n, i) $(n = 2^k)$.

Question 28 *This question deals with the design of the* BARREL-SHIFTER(n) *depicted in Figure 5.5.*

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

5.3 Priority Encoders

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

Example 8 *Consider the string* $x[0:6] = 0110100$. *The leading one is the bit* $x[1]$. *Note that indexes are in ascending order and that* $x[0]$ *is the leftmost bit.*

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

Definition 33 *A binary string* $x[0:n-1]$ represents a number in unary representation *if* $x[0:n-1] \in 1^* \cdot 0^*$. *The* value represented in unary representation *by the binary string* $1^i \cdot 0^j$ *is* i.

Remark 1 *A binary string such as* 01001011 *does not represent a number in unary representation. Only a string that is obtained by concatenating an all-ones string with an all-zeros string represents a number in unary representation.*

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

Definition 34 *A unary priority encoder* U-PENC(n) *is a combinational circuit specified as follows.*

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

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

Functionality:

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

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

Definition 35 *A binary priority encoder* B-PENC(n) *is a combinational circuit specified as follows.*

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

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

Functionality:

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

Example 9 *Given input* $x[0:5] = 000101$, *a* U-PENC(6) *outputs* $y[0:5] = 000111$, *and* B-PENC(6) *outputs* $y[2:0] = 011$.

5.3.1 Implementation of U-PENC(n)

As in the case of decoders, we can design a brute force unary priority encoder by a separate OR-tree for each output bit. The delay of such a design is $O(\log n)$ and the cost is $O(n^2)$. The issue of efficiently combining these trees will be discussed in detail when we discuss parallel prefix computation. Instead, we will present here a design based on divide-and-conquer.

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

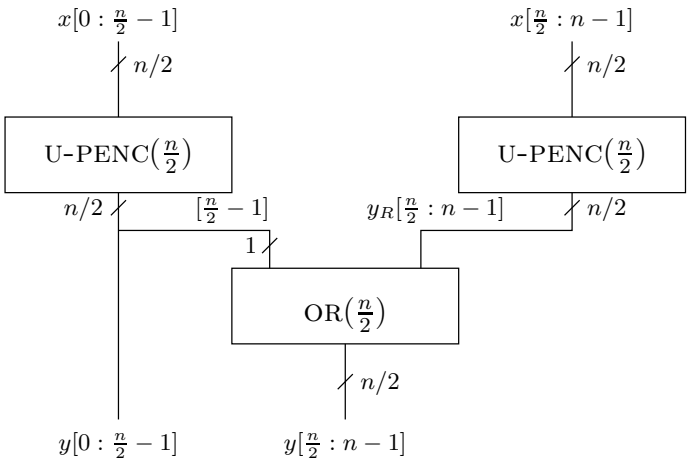

Figure 5.6: A recursive implementation of U-PENC(n).

Question 29 *This question deals with the design of the priority encoder* U-PENC(n) *depicted in Figure 5.6.*

1. *Prove the correctness of the design.*
2. *Extend the design for values of* n *that are not powers of* 2.

3. *Analyze the delay of the design.*

4. *Prove the asymptotic optimality of the delay of the design.*

Cost analysis. The cost $c(n)$ of the U-PENC(n) depicted in Figure 5.6 satisfies the following recurrence equation.

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

It follows that

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

The following question deals with the optimality of the U-PENC(n) design depicted in Figure 5.6.

Question 30 *Prove a lower bound on the cost of a unary priority encoder.*

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

5.3.2 Implementation of B-PENC

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

Reduction to U-PENC

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

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

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

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

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

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

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

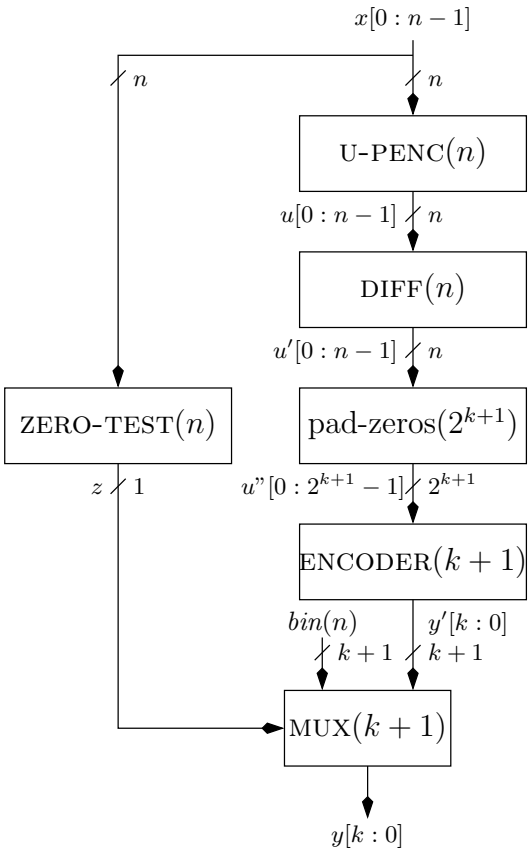

Figure 5.7: A binary priority encoder based on a unary priority encoder.

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

3. If n is not a power of 2, then we need to pad $\vec{u'}$ by zeros. The string $u''[0 : 2^{k+1} - 1]$ is obtained from $u'[0 : n - 1]$ by padding zeros as follows:

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

(Recall that $k = \lfloor \log_2 n \rfloor$.) Note that the cost and delay of padding zeros is zero.

4. The string $u''[0 : 2^{k+1} - 1]$ is input to an ENCODER$(k + 1)$. The encoder outputs the string $y'[k : 0]$ that satisfies $\langle \vec{y'} \rangle$ equals the index of the bit in $\vec{u''}$ that equals one. If $\vec{x} \neq 0^n$, then the vector $\vec{y'}$ equals the final output. However, we need to deal also with an all-zeros input.

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

Cost and delay analysis. The cost of the binary priority encoder depicted in Figure 5.7 satisfies:

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

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

The delay of the binary priority encoder depicted in Figure 5.7 satisfies:

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

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

A divide-and-conquer implementation

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

Figure 5.8 depicts a recursive design for a binary priority encoder. Consider an input $x[0 : n - 1]$. We partition it into two halves: the left part $x[0 : \frac{n}{2} - 1]$ and the right part $x[\frac{n}{2} : n - 1]$. Each of these two parts is fed to a binary priority encoder with $n/2$ inputs. We denote the outputs of these binary priority encoders by $y_L[k - 1 : 0]$ and $y_R[k - 1 : 0]$. The final output $y[k : 0]$ is computed as follows: $y[k] = 1$ iff $y_L[k - 1] = y_R[k - 1] = 1$, $y[k - 1] = \text{AND}(y_L[k - 1], \text{INV}(y_R[k - 1]))$, and $y[k - 2 : 0]$ equals $y_L[k - 2 : 0]$ if $y_L[k - 1] = 0$ and $y_R[k - 2 : 0]$ otherwise. We now prove the correctness of the binary priority encoder design based on divide-and-conquer. Note that we omitted a description for $n = 2$. We leave the recursion basis as an exercise.

Claim 19 *The design depicted in Figure 5.8 is a binary priority encoder.*

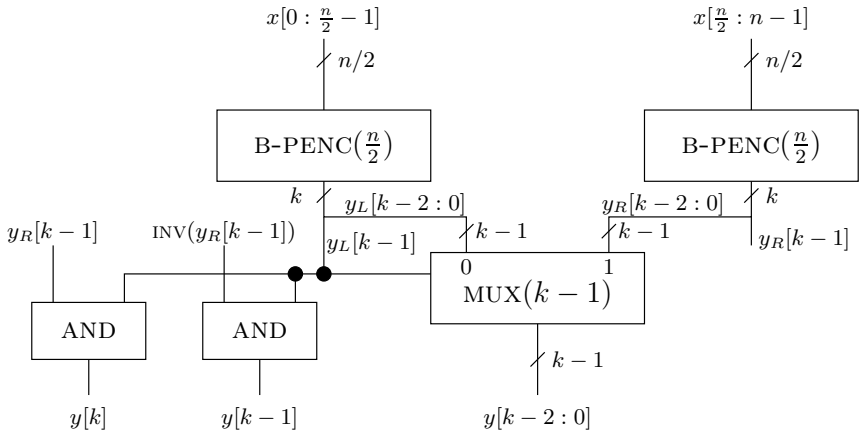

Figure 5.8: A recursive implementation of a binary priority encoder.

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

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

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

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

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

Cost and delay analysis. The cost of the binary priority encoder depicted in Figure 5.8 satisfies (recall that $n = 2^k$):

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

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

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

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

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

5.4 Half-Decoders

In this section we deal with the design of half-decoders. Recall that a decoder is a combinational circuit that computes a 1-out-of-2^n representation of a given binary number. A half-decoder computes a unary representation of a given binary number. Half-decoders are used for implementing logical right shift.

Definition 36 *A half-decoder with input length n is a combinational circuit defined as follows:*

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

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

Functionality:

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

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

Example 10

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

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

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

Question 31 *Suggest an implementation of a half-decoder based on a decoder and a unary priority encoder. Analyze the cost and delay of the design. Is it optimal with respect to cost or delay?*

5.4.1 Preliminaries

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

Claim 20

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

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

Claim 21 *For $i \in [0 : n-1]$:*

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

Claim 21 gives a simple recipe for a "comparison box". A comparison box, denoted by COMP$(\vec{z}, i)$, is a combinational circuit that compares $wt(\vec{z})$ and i and has three outputs GT, EQ, LT. The outputs have the following meaning: GT - indicates whether $wt(\vec{z}) > i$, EQ - indicates whether $wt(\vec{z}) = i$, and LT - indicates whether $wt(\vec{z}) < i$. In the sequel we will only need the GT and EQ outputs. (Note that the GT output simply equals $z[i]$).

Consider a partitioning of a string $x[n-1:0]$ according to a parameter k into two sub-strings of length k and $n-k$. Namely

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

Binary representation implies that

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

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

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

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

Claim 22

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

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

5.4.2 Implementation

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

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

input string $x[n-1:0]$ is divided into two strings $x_L[n-k-1:0] = x[n-1:k]$ and $x_R[k-1:0] = x[k-1:0]$. These strings are fed to a half-decoders H-DEC$(n-k)$ and H-DEC(k), respectively. We denote the outputs of the half-decoders by $z_L[2^{n-k}-1:0]$ and $z_R[2^k-1:0]$, respectively. Each of these string are fed to comparison boxes. The "rows" comparison box is fed by $\vec{z_L}$ and compares $\vec{z_L}$ with $i \in [0:2^{n-k}-1]$. The "columns" comparison box is fed by $\vec{z_R}$ and compares $\vec{z_R}$ with $j \in [0:2^k-1]$. Note that we only use the GT and EQ outputs of the rows comparison box, and that we only use the GT output of the columns comparison box. (Hence the columns comparison box is trivial and has zero cost and delay.) We denote the outputs of the rows comparison box by $Q_{GT}[0:2^{n-k}-1]$ and $Q_{EQ}[0:2^{n-k}-1]$. We denote the output of the columns comparison box by $R_{GT}[0:2^k-1]$. Finally, the outputs of the comparison boxes fed an array of $2^{n-k} \times 2^k$ G-gates. Consider the G-gate $G_{q,r}$ positioned in row q and in column r. The gate $G_{q,r}$ outputs $y[q \cdot 2^k + r]$ which is defined by

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

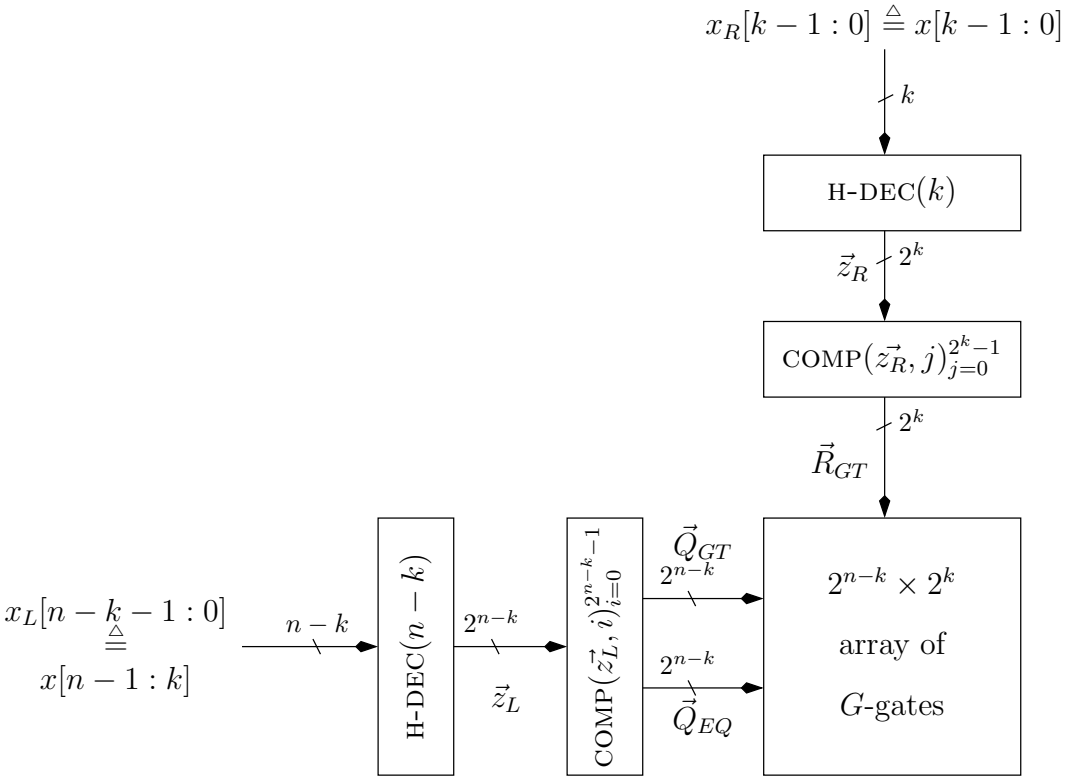

Figure 5.9: A recursive implementation of H-DEC(n). Note that the comparison boxes COMP$(\vec{z_R}, j)$ are trivial, since we only use their GT outputs.

Example 11 *Let $n = 4$ and $k = 2$. Consider $i = 6$. The quotient and remainder of i when divided by 4 are 1 and 2, respectively. By Claim 20, $y[6] = 1$ iff $\langle x[3:0] \rangle > 6$. By Claim 22,*

$\langle \vec{x} \rangle > 6$ iff ($\langle x[3:2] \rangle > 1$) or ($\langle x[3:2] \rangle = 1$ and $\langle x[1:0] \rangle > 2$). It follows that if $Q_{GT}[1] = 1$, then $y[6] = 1$. If $Q_{EQ}[1] = 1$, then $y[6] = R_{GT}[2]$.

5.4.3 Correctness

Claim 23 *The design depicted in Figure 5.9 is a correct implementation of a half-decoder.*

Proof: The proof is by induction on n. The induction basis, for $n = 1$, is trivial. We now prove the induction step. By Claim 20 it suffices to show that $y[i] = 1$ iff $i < \langle \vec{x} \rangle$, for every $i \in [2^n - 1 : 0]$. Fix an index i and let $i = q \cdot 2^k + r$. By Claim 22,

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

The induction hypothesis implies that:

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

Observe that:

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

Finally, by Eq. 5.1, $y[i] = 1$ iff $\text{OR}(Q_{GT}[q], \text{AND}(Q_{EQ}[q], R_{GT}[r]))$. Hence $y[i]$ is correct, and the claim follows. □

5.4.4 Cost and delay analysis

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

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

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

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

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

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

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

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

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

It follows that the delay of H-DEC(n) is asymptotically optimal since all the inputs belong to the cone of a half-decoder.
The following question deals with the optimal cost of a half-decoder design.

Question 32 *Prove that every implementation of a half-decoder design must contain at least $2^n - 2$ non-trivial gates. (Here we assume that every non-trivial gate has a single output, and we do not have any fan-in or fan-out restrictions).*

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

1. *The other outputs are not constant - so they can't be fed by a trivial constant gate.*
2. *The other outputs are distinct - so every two outputs can't be fed by the same gate.*
3. *The other outputs do no equal the inputs - so they can't be directly fed from input gates (which are trivial gates).*

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

5.5 Summary

We began this chapter by defining n:1-multiplexers. We presented two optimal implementations. One implementations is based on a decoder, the other implementation is based on a tree of multiplexers. We then defined cyclic shifting, and presented an implementation of a barrel-shifter.

Priority encoders are circuits that compute the position of a leading one in a binary string. We considered two types of priority encoders: unary and binary. The difference between these two types of priority encoders is in how the position of the leading one is represented (i.e. in binary representation or in unary representation). We presented a divide-and-conquer design for a unary priority encoder. The delay of this design is optimal,

but its cost is not. We will present an optimal unary priority encoder as soon as we learn about parallel prefix computation.

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

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