Topics in Parallel and Distributed Computing:

Introducing Concurrency in Undergraduate Courses[1,2]

Chapter 5

Introducing Parallel and Distributed Computing Concepts in Digital Logic

Ramachandran Vaidyanathan Jerry L. Trahan Suresh Rai

Louisiana State University

{vaidy, jtrahan, srai}@lsu.edu

[1]How to cite this book: Prasad, Gupta, Rosenberg, Sussman, and Weems. *Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses*, 1st Edition, Morgan Kaufmann, ISBN : 9780128038994, Pages: 360.

[2]Free preprint version of the CDER book: http://grid.cs.gsu.edu/~tcpp/curriculum/?q=cedr_book.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 5

INTRODUCING PARALLEL AND DISTRIBUTED COMPUTING CONCEPTS IN DIGITAL LOGIC

Ramachandran Vaidyanathan Jerry L. Trahan Suresh Rai

Louisiana State University

{vaidy, jtrahan, srai}@lsu.edu

ABSTRACT

This chapter provides a framework for introducing parallel and distributed computing (PDC) concepts as they arise naturally in digital logic. It is directed primarily towards the instructor of a first digital logic course, although a motivated student may find it useful as well. Because a digital logic course lays the foundation for other courses such as computer organization and architecture, an early exposure to PDC concepts will have a wide reach. Moreover, the introduction of these concepts through a less conventional direction complements and enhances understanding of these ideas when taught subsequently through conventional avenues such as programming, algorithms and architecture. Thus, while the chapter primarily addresses instructors of digital logic, instructors of courses more directly associated with PDC may also benefit by connecting the concepts to topics that students learned earlier. We also anticipate that casting digital logic concepts in multiple ways will also help students better understand digital logic.

Relevant Core Courses Impacted: This chapter is primarily directed towards a first digital logic course but will indirectly impact learning in several courses such as computer

architecture and algorithms (including parallel and distributed), networking and interconnects, and computational complexity.

Relevant PDC Topics: For a course whose primary focus is outside PDC, many of the PDC topics covered would be at the lower end of the Bloom taxonomy. However, we expect the real benefit of the proposed early exposure to PDC concepts would be in subsequent courses that address PDC topics more directly. The early (indirect) exposure to PDC would build a framework to which subsequently covered PDC topics can latch on to, leading to improved learning and retention. Table 5.1 shows the expected pedagogical level for a set of PDC concepts corresponding to the digital logic topics discussed in this chapter.

Table (5.1) Pedagogical levels for PDC concepts, using K (know the term), C (comprehend so as to paraphrase or illustrate) and A (apply it in some way).

PDC Concept	Chapter Section					
	5.1	5.2	5.3	5.4	5.5	5.6
Concurrency, sequential/parallel		C	C	C	C	A
Interconnects, topologies	K	C	C			A
Performance measures (latency, scalability, efficiency, trade-offs)			K	C	C	C
Recursive decomposition, divide-and-conquer	K	C		A		
Prefix computation				C	A	
(A)synchrony			K		C	A
Pipelining					C	
Data hazards			K			
Buses, shared resources		K				K
Complexity, asymptotics	K					
Dependencies, task graphs			K			
Broadcast, multicast		K				
Reduction, convergecast		K			C	

Learning Outcomes: The main purpose of this chapter is to provide a framework for introducing PDC topics while teaching digital logic. The following set of possible

learning outcomes must be viewed in this context.

(1) Recognize basic interconnection structures.

For example: tree, bus, mesh, torus.

(2) Identify and apply parallel and sequential structures in hardware.

For example: identify critical path in a circuit; explain concurrency in a carry-look-ahead adder; identify time-size trade offs in circuits such as adders; adjust parallelism in circuits to adjust delay.

(3) Understand the concepts of recursive decomposition and divide-and-conquer.

For example: apply recursive decomposition to circuit design (such as in a multiplexer); recognize the parallelism in independent branches of a recursion.

(4) Understand the concept of a prefix computation.

For example: explain the prefix computations present in a carry look-ahead adder and in a synchronous counter.

(5) Explain the difference between synchronous and asynchronous operations.

For example: asynchronous nature of latches and ripple counters.

(6) Understand the concept of an ideal pipeline.

For example: as a generalization of a shift register; as a form of parallelism.

Context for Use: Hardware is inherently parallel and digital logic provides a fertile ground for exploring fundamental ideas in PDC at an early stage of the curriculum. It affords instructors an early opportunity to introduce key PDC concepts to which many students in electrical and computer engineering would otherwise not be exposed until much later in the academic program. Even for those students who receive an exposure to PDC concepts early in their programs, the digital logic context provides a different setting that serves to broaden their perspectives and understanding of key concepts in both PDC and digital logic.

This chapter is intended to help the instructor incorporate PDC in the coverage of standard digital logic topics, rather than budget separate instruction time specifically

for PDC. It does not seek to add new topics to the digital logic course. Rather, it suggests a framework in which PDC topics can be introduced seamlessly through the instruction of digital logic topics. The discussion of PDC topics could be in class as a side note in a lecture, or a separate exploration off-line, perhaps as part of homework or a reading assignment.

The level of the chapter assumes the digital logic instructor to have some PDC background. The goal, in most instances, is to augment standard approaches to the material and view digital logic topics from a different perspective, revealing PDC concepts already present in these topics. In some instances, the goal is to generalize a digital logic topic to a PDC application, both revealing the digital logic concept as a building block and motivating it by tangible applications. To facilitate simple integration, the material in this chapter is organized by digital logic topics (rather than by PDC concepts). Along the way, several PDC concepts will be visited, possibly multiple times and in increasing levels of complexity. This allows the instructor to select the set of PDC concepts to be discussed and the level of detail of the discussion. We also observe that in programs requiring digital logic, the material in this chapter could be utilized by instructors of mainstream PDC courses to draw analogies to (digital logic) concepts that students may have grasped earlier. The material in this chapter is meant to augment and illustrate digital logic concepts using PDC examples.

Chapter Organization: The chapter has five sections (Sections 5.1–5.5) that discuss PDC ideas in the context of specific digital logic topics. Formalism has been employed in places to convey some ideas accurately, tersely and with some generality. This is directed to the instructor. We recommend that the instruction itself use simple examples without the formalism.

As we noted earlier, the focus is on relating digital logic concepts to PDC ideas. Consequently, we do not discuss the PDC ideas themselves in much detail, often just offering an alternative view of the digital logic topic. A basic understanding of many of these

PDC ideas can be obtained from resources on the Internet; nevertheless, we suggest some references that the reader may find useful. Each section also lists applications where the PDC concepts discussed could be used.

At the end of each section, we offer suggestions for incorporating the ideas discussed in the instruction of digital logic. Typically, this picks each of the broad topics of a section and lists (in order of increasing complexity) various aspects of the topics to which students can be exposed. The instructor can tailor this to suit the class.

Finally, Section 5.6 outlines idea threads for several other digital logic settings that the instructor could elaborate upon as desired.

5.1 Number Representation

One of the first topics introduced in digital logic is the binary representation of integers. Consider an exercise where each student picks a number u from the set $\{0, 1, 2, \cdots, 15\}$ (abbreviated here as $[0, 15]$). The purpose of the exercise is to derive the 4-bit binary representation of u. For this discussion, we will refer to the set $[0, 15]$ as the *level* 4 *set of* u. As a first step, determine whether the selected number u is in the bottom half, $[0, 7]$, or top half, $[8, 15]$, of the set $[0, 15]$ of possible numbers. If $u \in [0, 7]$, record a "`0`" to indicate the bottom half; similarly if $u \in [8, 15]$, record a "`1`" to indicate the top half. Denote the symbol (`0` or `1`) as the level 3 symbol of u and the set ($[0, 7]$ or $[8, 15]$) that u belongs to as the level 3 set of u. For the running example with $u = 13$, the level-3 symbol is `1` and the level 3 set is $[8, 15]$.

Next, record the level 2 symbol of u as `0` or `1` depending on whether u is in the bottom or top half of its level 3 set. For the running example, the level-2 symbol is `1` as 13 is in the top half of its level 3 set $[8, 15]$; the level 2 set of 13 is $[12, 15]$. Proceeding similarly, we find the level 1 and level 0 symbols of u and obtain a string $b_3b_2b_1b_0$, where $b_i \in \{\texttt{0}, \texttt{1}\}$ is the level i symbol of u. For the running example, the level 1 and level 0 symbols of 13 are `0` and `1`, respectively; the string for 13 is `1101`.

We have presented the binary representation of u as a sequence of decisions about which half of a range u belongs to. The binary decision tree of Figure 5.1 represents all possible outcomes of these decisions for set $[0, 15]$.

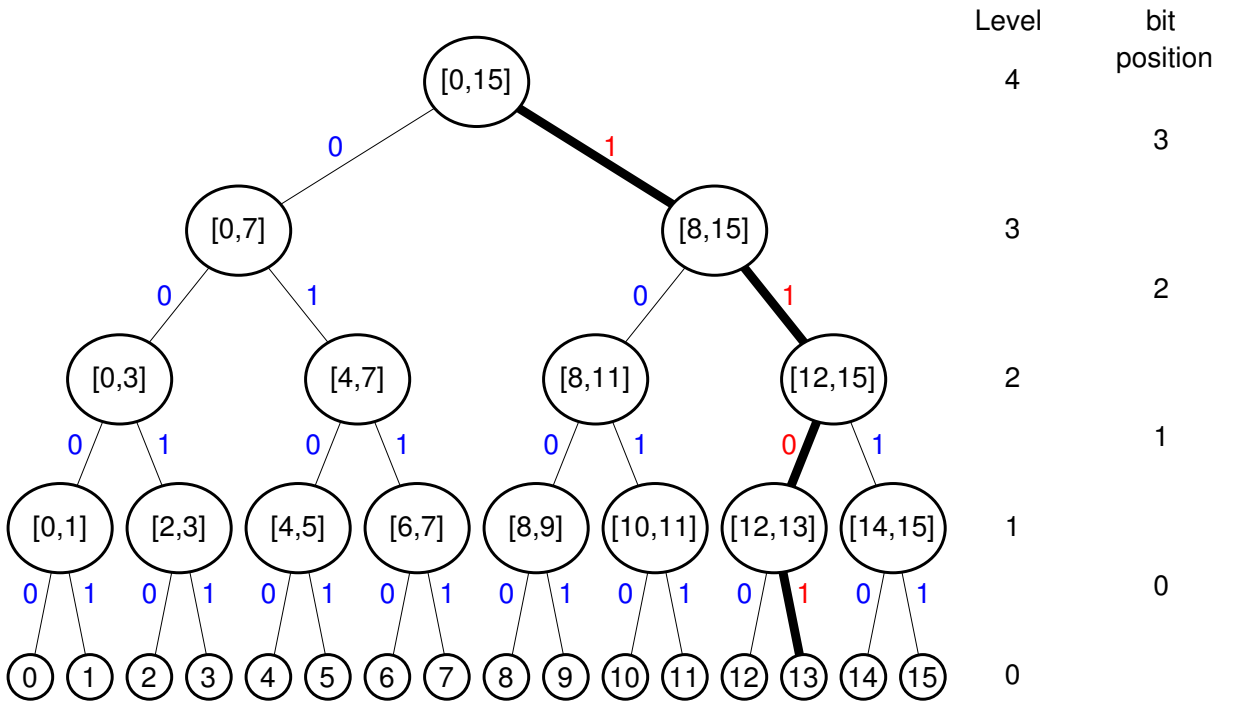


Figure (5.1) A decision tree representing the 4-bit numbers in the range $[0, 15]$. The representation for 13 is emphasized. The figure also shows the node levels and bit positions corresponding to edge labels.

The tree representation described here can be used as a basis for several concepts in digital logic, PDC, and some fundamental to computing itself.

Binary Numbers: The above tree representation can be used to augment standard discussion of binary numbers. Consider any leaf u; we assume u to denote both the node and its value. Observe that the sequence of edge labels (decision outcomes) in a path from the root to leaf u is the binary representation of the edges in the path. Figure 5.1 shows the path to leaf 13 in bold; the corresponding sequence of edge labels 1101 is the binary representation of 13.

In general, observe that nodes at level ℓ represent the level ℓ sets and an edge between nodes at levels ℓ and $\ell + 1$ is labeled with the level ℓ symbol. Let u be a leaf and let the

labels on edges in the path from the root to the leaf u be $b_{n-1}, b_{n-2}, \cdots, b_0$. Observe that the value of a leaf is the number of leaves to its left in the tree. For example, there are thirteen leaves (labeled 0 to 12) to the left of leaf 13 in Figure 5.1. Thus obtaining the binary representation of a number u is an exercise in determining the number of leaves to the left of u in the corresponding binary tree.

Consider an edge labeled b_ℓ in the path from the root to leaf u; this is an edge between a node α (say) at level $\ell + 1$ and its child β (say) at level ℓ; let the other child of α be γ (also at level ℓ). If $b_\ell = 1$, then all 2^ℓ leaves of the subtree rooted at γ are to the left of u. If $b_\ell = 0$, on the other hand, then all leaves of γ are to the right of u; one cannot tell at this point how many leaves of α are to the left of u. Extending this to all levels, the value of the leaf (number of other leaves to its left) is $u = \sum_{i=0}^{n-1} b_i 2^i$. This is the basis for converting a binary number to its (decimal) value.

To see the correspondence in the other direction, we first label each internal node slightly differently. Let u be a node at level ℓ. Then label u by a string of length n of the form

$$b_{n-1}b_{n-2}\cdots b_{n-\ell} \underbrace{**\cdots*}_{n-\ell \texttt{ times}},$$

where $b_{n-1}, b_{n-2}, \cdots, b_{n-\ell}$ is the sequence of bits in the path from the root to node u and $*$ is a wild-card symbol. For example, the nodes corresponding to ranges $[12, 15]$ and $[12, 13]$ are represented by the strings $11{**}$ and $110{*}$, respectively. Notice that the binary representations of all elements of $[12, 15]$ have 11 in the most significant two bits; the corresponding string $11 * *$ reflects this. Similarly the string $110*$ for $[12, 13]$ reflects that fact that the binary representations of 12 and 13 have 110 in their most significant bits and differ only in the least significant bit.

We now get back to relating the tree of Figure 5.1 to the standard method of converting a (decimal) number to its binary representation. This standard conversion method calls for repeated division of the given number by 2 and recording the remainders. These produce the binary number from the least significant bit (lsb) to the most significant bit (msb) or from

the leaves to the root in the binary tree. Consider the example of $u = 13$ (see Table 5.2). The last column of the table has a 4-bit string, the non-wild-card bits of which constitute

Table (5.2) Converting 13 to binary.

bit					quotient	remainder	string			
0 (lsb)	13	$\div$	2	=	6	1	1	1	0	$*$
1	6	$\div$	2	=	3	0	1	1	$*$	$*$
2	3	$\div$	2	=	1	1	1	$*$	$*$	$*$
3 (msb)	1	$\div$	2	=	0	1	$*$	$*$	$*$	$*$

the binary representation of the quotient. As described above, these strings also identify the nodes $[12, 13], [12, 15], [8, 15], [0, 15]$ on the path from 13 to the root.

These ideas could also be further explored in the context of an r-ary tree for a general radix r; $r = 8, 16$ (octal and hexadecimal) may be of particular interest in digital logic. At a later stage (particularly in input, state or output assignment in finite state machine design), one could use an n-bit number to represent fewer than 2^n distinct "values." Here one could also introduce heaps and unbalanced trees by considering subsets of $\{0, 1, \cdots, 2^n - 1\}$.

Recursive Thinking: We constructed the tree of Figure 5.1 by dividing the initial range $[0, 2^n - 1]$ into the bottom and top halves, which formed the left and right subtrees. The subtrees themselves were constructed the same way (recursively). This manner of expressing a large instance of a problem (for the range $[0, 2^n-1]$) in terms of smaller instances of the same problem (ranges $[0, 2^{n-1}-1]$ and $[2^{n-1}, 2^n-1]$) is an easy way to introduce recursive thinking and the divide-and-conquer paradigm. The fact that the left subtree can be constructed completely independently of the right subtree can also be used to introduce the idea of parallelism.

Trees and Interconnection Networks: The systematic partitioning of the range $[0, r^n-1]$ into a hierarchy of ranges represented in the r-ary tree can be used to introduce the notion of a tree itself and its use in representing hierarchical structures. Familiar examples include family trees and organizational hierarchical structures. Less familiar, perhaps, a phylogenetic tree [1212] captures evolutionary relationships between organisms, and vascular trees [3] model

the proliferation of blood vessel in organs. In the computing arena itself (including in PDC), trees are central to numerous topics. One example is that of an interconnection network [456456456]. In the current context, an interconnection network uses a tree structure to connect several elements that could communicate with each other. For instance, a set of computer terminals or telephone lines in an office building may be connected to each other and to the outside through a (minimum spanning) tree. A (minimum Steiner) tree [7] is often used to connect elements inside an integrated circuit (IC). These forms of connecting elements use the fact that a tree is a minimally connected graph (that connects its nodes with the smallest number of edges). Another interconnection network that is based on a tree is the fat-tree network [8], which has been used to connect large multiprocessor systems.

The Difference between Unary and Binary: Students may have observed that the depth (or maximum level of nodes) of a balanced binary tree for the set $\{0, 1, \cdots, N-1\}$ is $\lceil \log_2 N \rceil$. The depth of a balanced ternary tree (for radix 3 representation) is $\lceil \log_3 N \rceil$ which is smaller only by about a factor of $\log_2 3 = 1.58$. In general for radix $r \geq 2$, the balanced r-ary tree has depth $\lceil \log_r N \rceil$ which is about $\log_2 r$ times smaller than that of a balanced binary tree with the same number of nodes. Observe that the ratio of the depths is independent of N. On the other hand, a unary tree (consisting of a chain of nodes) has a depth of N (assuming internal nodes can represent elements of $\{0, 1, \cdots, N-1\}$). This depth is considerably larger than r-ary trees for $r \geq 2$.

This observation carries much import for PDC. Often a large problem is divided recursively into r independent subproblems, and each subproblem solved in parallel. Moving from $r = 1$ to $r = 2$ has a substantial impact on the time for the computation, but not so much for $r = k \geq 2$ to $k+1$. Cormen *et al.* [9] discuss the effect of input encoding (unary/binary) on computational complexity (Section 34.1).

Instruction Notes for Section 5.1: The discussion on binary numbers could start with the (in-class) exercise described earlier (in which each student picks a number u and makes decisions about the bottom and top halves of sets to derive the binary representation). It could also

be explored as an off-line activity in a flipped-classroom setting[1].

The exercise described at the start of this section uses symbols `0` and `1` to denote the bottom and top halves of a range. The instructor could point out that other symbols would work equally well, for example `f` and `t` (for "false" and "true") to denote the answer to the question of whether the number belongs to the top half. This could help students see that a bit has logical significance (over and above the arithmetic significance of a radix-2 representation).

The discussion on recursive thinking should first construct the tree emphasizing that the recipe for each node to expand into its children is exactly the same (divide into top half and bottom half). Then it can be cast as solving a large problem by solving smaller problems of the same type. The idea of parallelism and divide and conquer could be emphasized by first splitting the root into its children ($[0, 15]$ into $[0, 7]$ and $[8, 15]$ in our example). Then have a think-pair-share[2] activity to (recursively) draw the trees for the two halves and put them together.

At this stage it would be useful to introduce the term "binary tree" and to introduce the need to connect modules in a digital system (this can be expanded to the ideas of graphs (state diagrams) and interconnection networks later).

The portion on the difference between unary and binary should start with emphasizing that numbers expressed with respect to larger bases have shorter representations; this will also help justifying octal and hexadecimal number systems that will be covered at a later stage. The large difference between a unary (linear) and binary (logarithmic) representations can then be illustrated with an example.

5.2 Logic Gates

In this section we will use fan-out, fan-in and tristate gates to illustrate PDC concepts.

[1] In a *flipped-classroom* setting, students understand the basics of a topic outside class, and class time is devoted to problem solving and the discussion of advanced concepts.

[2] In a *think-pair-share* activity, students in a small group individually solve a problem, then collectively discuss their solutions to come to a final answer.

5.2.1 Fan-out and Fan-in of Gates

A fan-out of f restricts the number of places to which a gate's output can be connected to be at most f. If a signal output by a gate needs to be delivered to N other gate inputs, then it is customary to use a set of buffers to boost the signal appropriately as shown in Figure 5.2. Here a source signal x, output for example by an OR gate, is ultimately fanned

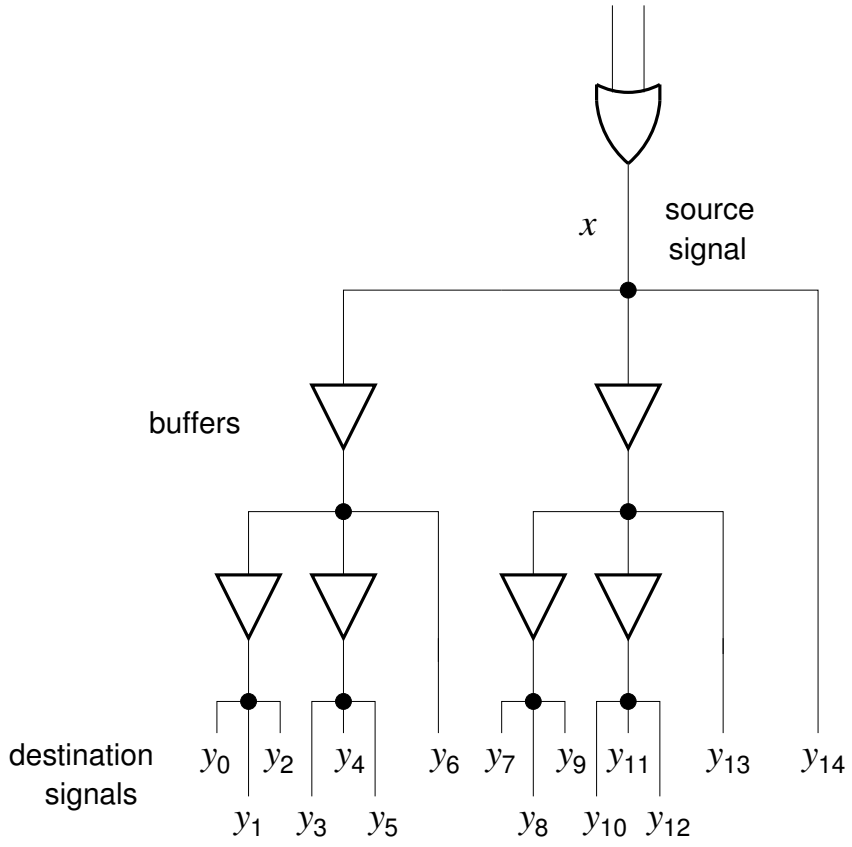


Figure (5.2) Fanning out a signal to 15 places using gates of fan-out 3.

out to 15 destinations. For this illustration a fan-out of 3 is used to construct a ternary tree. In general, fanning the signal out to N places with buffers of fanout f constructs an f-ary tree of buffers of depth $\lceil \log_f N \rceil$.

Similarly, a restriction in fan-in of a gate (number of input bits that a gate may receive) can be circumvented by a tree of gates. A gate with fan-in g can accept g inputs. For associative operations such as AND, OR, EX-OR, max or min, a g-ary tree is used to apply

the operation on $N > g$ inputs. Figure 5.3 illustrates the fanning in of 15 logic values to one

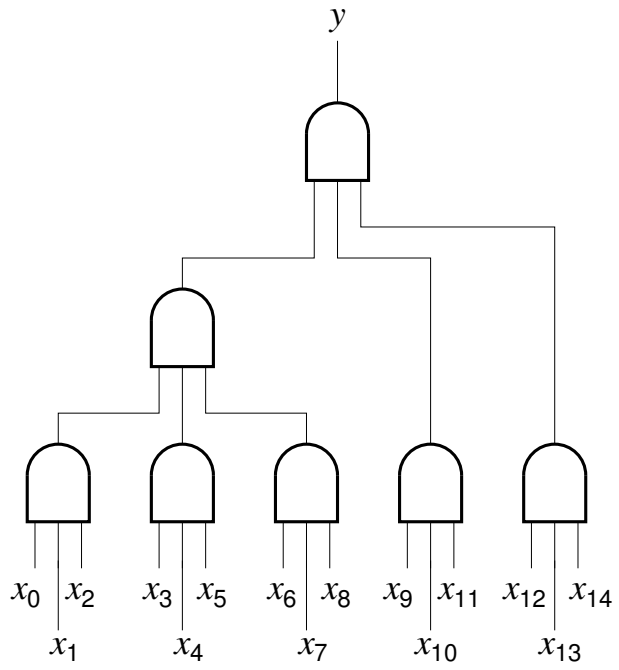


Figure (5.3) ANDing 15 bits using gates of fan-in 3.

value using seven 3-input AND gates.

Trees Again: While the use of trees for fan-out and fan-in is folklore in digital logic, it provides a convenient basis for reinforcing ideas from the previous section and to introduce new ideas. First it must be noted that the idea of a tree introduced in the context of number representations has a direct application in digital logic (for fan-out and fan-in). Observe that while the logic values in the circuits in Figures 5.2 and 5.3 move from one-to-many and many-to-one, respectively, the underlying tree structures (with its edge directions interpreted differently) is abstract enough to capture both situations. Indeed, one could easily draw a fan-out tree corresponding to Figure 5.3 and a fan-in tree corresponding to Figure 5.2. This also illustrates that many such trees are possible for the same problem, and, done correctly, all of them for the example in question will have seven gates arranged in three levels.

Broadcast, Multicast and Convergecast: In many applications, a value needs to be *broadcast* from a source to all other nodes, for example, an emergency bulletin that needs to be sent

to all. This is a fan-out of a value to many places. Broadcasting is a special case of a more common mechanism called a *multicast* in which a value is sent to a subset of nodes. For example, a webinar may require the source to send the same content to several users (a very small subset of the Internet). Other similar examples include the multicasting of news items and stock quotes that may be viewed by several consumers simultaneously. In all of these cases, it would be unnecessarily expensive to send a separate feed to each user. For example, a live World Cup Soccer video may be fed to several regional hubs and local stations before it is streamed into individual devices. This requires the construction of a *multicast tree* through which the source can send content to a subset of nodes. A node in the interior of this tree may or may not be a destination for the multicast (see Figure 5.4 for an example).

At this point it may be useful to generalize trees into graphs. That is, explain to students that a graph is a set of nodes connected by a set of edges. The Internet, for example, may be represented as a graph [10]. In this context, a multicast tree is a subgraph of the Internet graph that includes all the nodes of interest (source and destinations), and which is typically optimized with respect to some cost (such as latency, quality of service or traffic). Figure 5.4 shows a (multicast) subtree of a larger graph.

In a multicast, the flow of information is from a source to several destinations (as in fan-out). *Convergecast* is the collection of information from several sources into one sink (as in the fan-in tree). Convergecasting is a fundamental operation of distributed systems and is commonly used in sensor networks in which the information from several sensors must be aggregated into a node. Again the idea is similar to a multicast, in that a tree is employed, this time to move information from the leaves to the root. The tree in Figure 5.4 can be used to collect information from the destination nodes into the source node.

Parallelism: Consider a game (such as "Monopoly") that can be played with three players, in which one player wins a 3-player match. A tournament pits fifteen players in groups of three in a knock-out tournament. For our discussion, let us assume that each match lasts a day. One way to conduct the tournament is with seven matches as shown in Figure 5.5; the last six players have the highest rankings and get to play one less game than the rest of

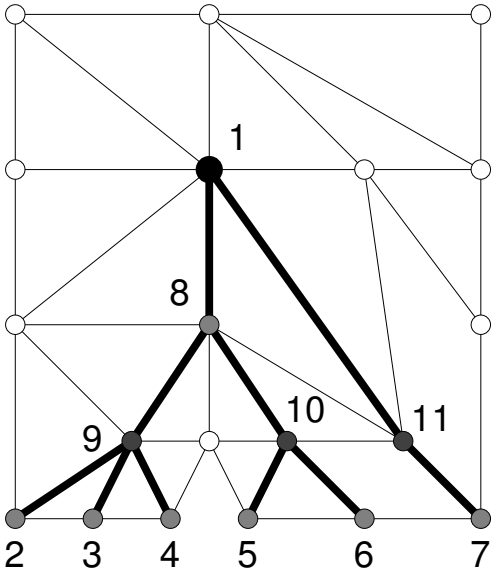


Figure (5.4) An example of a multicast tree. Node 1 is the source node. Nodes 2–8 are destination nodes, and 9–11 are intermediate nodes (that are part of the multicast tree, but are not destination nodes); observe that a non-leaf node, such as node 8, can be a destination node.

the field. Clearly, the trees in Figures 5.3 and 5.5 are topologically equivalent. The tree of Figure 5.5 shows the dependencies among the matches in the tournament (in the same way that Figure 5.3 shows the connections among gates). The gate delay and the match duration are also analogous quantities. We now indicate how the dependencies in the tree can be used to illustrate *speed-up* and *efficiency*, two important concepts in parallel processing.

A *feasible schedule* for the tournament is an arrangement of matches in time (possibly in parallel) such that the dependencies shown in the tree of Figure 5.5 are respected. For example, matches 1 and 2 can be held simultaneously, whereas matches 1 and 6 cannot, as the result of match 1 is needed before match 6 can be held. We now examine different schedules for the tournament.

An obvious schedule for the seven matches is to hold them one at a time. While this schedule would need only one venue (there are no simultaneous matches), the tournament would last a week. One could recognize that all matches at the same level of the tree in Figure 5.5 can be held in parallel, and this leads to a second schedule with a 3-day

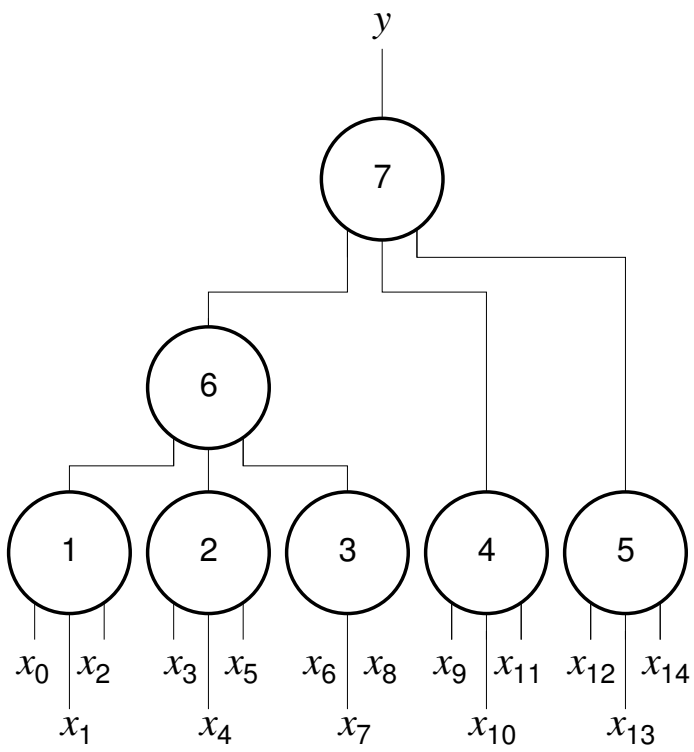


Figure (5.5) A 7-game tournament among 15 players. Each circle represents a match. Players are denoted by x_0–x_{14} and y represents the champion.

tournament (one day per level of the tree). Specifically, matches 1–5 are held on day 1, match 6 on day 2 and match 7 on day 3. This schedule requires five venues, however, as the first day has five concurrent matches. Observe in Figure 5.5 that matches 4, 5 and 6 are independent of each other and can, therefore, be held in parallel. This results in a third schedule shown in Figure 5.6 that corresponds to a three-day tournament requiring only three venues; matches 1–3 are held on day 1, matches 4–6 are held on day 2 and match 7 is held on day 3. Note also that a first round match (one of matches 1–3) for players x_0–x_8, their second round match (match 6) and the finals (match 7) all have to be held one after the other (sequentially); therefore, the tournament cannot be conducted in fewer than three days. Moreover since there are seven matches over three days, we will need a minimum of $\lceil \frac{7}{3} \rceil = 3$ venues. So the third schedule is optimal.

As this example illustrates, a tree may be used to model the hierarchical relationship between a set of elements (including computational tasks). The relative positions of these

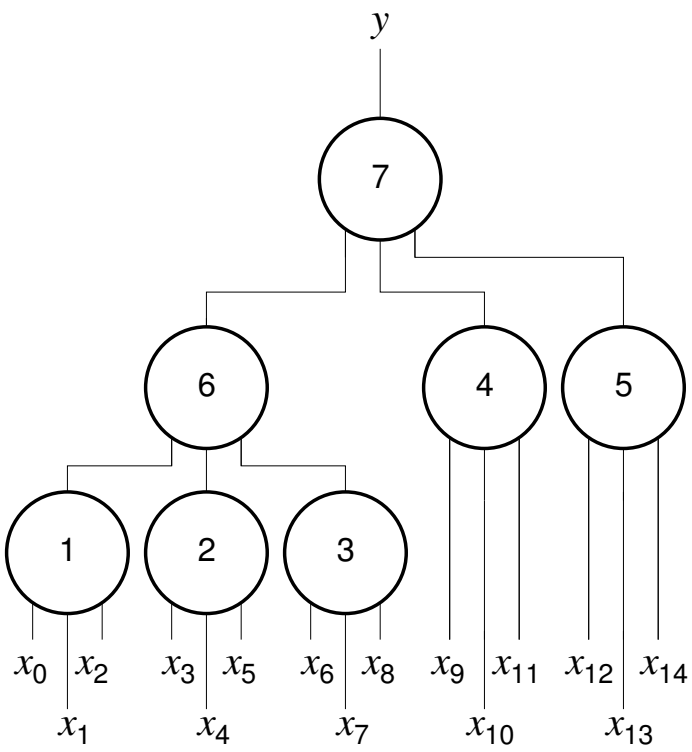


Figure (5.6) The schedule tree of Figure 5.5 redrawn to balance the number of nodes per level.

elements within the tree could be used to determine which tasks can run in parallel and which are (sequentially) dependent on each other. In a computational setting, the number of venues may correspond to the number of processors and the number of days needed for the tournament corresponds to the time needed for the computation. Compared to the sequential 7-day solution, the parallel 3-day solution finishes $\frac{7}{3}$ faster. In the setting of parallel computing, this quantity is called the *speed-up* of the computation.

Continuing with the Monopoly example, suppose that each venue rents for one unit per day and that the venues must be reserved for the entire tournament. Then the sequential 7-day solution costs 7 units. The first 3-day, 5-venue schedule has a cost of $3 \times 5 = 15$ and the second 3-day, 3-venue schedule has a cost of 9. The 7-day tournament has $\frac{7}{9}$ the cost of our 3-day, 3-venue tournament. This quantity is sometimes called the *efficiency* of the parallel solution.

We note that one of the first areas in which parallelism was explored was circuit com-

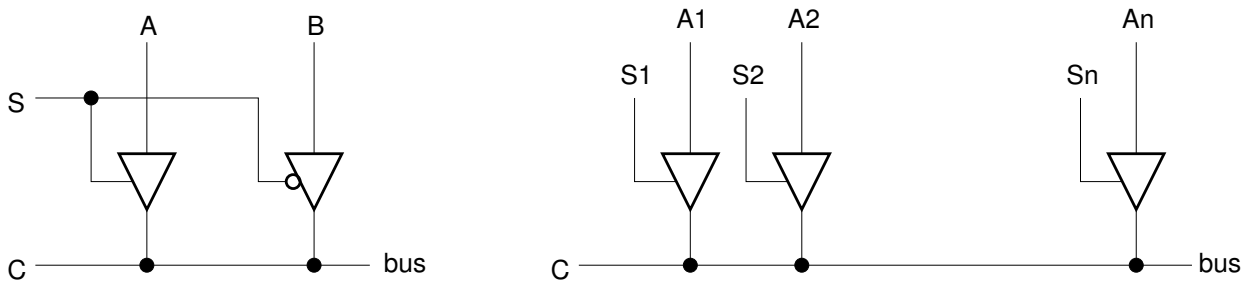
plexity [11]. The simplicity of circuits as a model was a key reason, as was the relation of the circuit model to physical circuits. Limiting parameters such as fan-in, types of gates, and uniformity of construction allowed investigating different computational complexity questions. The primary measures were circuit size and depth. Further, circuit complexity strongly relates to the the complexity of parallel computation (for example circuit depth and size roughly correspond to the time and work needed for a parallel computation).

Recursive Decomposition: The fan-in and fan-out problems may also been viewed recursively. For example, consider the fan-in problem of Figure 5.6. Suppose $F(N)$ represents the fan-in solution for N inputs using gates of fan-in 3. The tree of Figure 5.6 corresponds to recursively decomposing $F(15)$ (that is, fanning in 15 inputs) into three subproblems, two $F(3)$'s and one $F(9)$ and combining the outputs of the three subproblems into the solution to $F(15)$. This can also be first explained in terms of a balanced ternary tree for $N = 3^k$, where k is an integer; then it makes sense to use $15 = 3^1 + 3^1 + 3^2$. The fan-out tree of Figure 5.2 similarly reduces the problem of sending a signal to 15 places to that of two instances of sending the signal to seven places, and sending the signal to one place. The broadcasting to seven places can similarly be recursively viewed in terms of that of two instances of sending to three places and one instance of sending to one place.

These examples further reinforce the idea of trees and recursive decomposition, illustrating the use of the same model (trees in this case) in two different scenarios.

5.2.2 Tristate Gates and Buses

Tristate gates open the opportunity of introducing a bus (to which several modules could connect through exclusively enabled tristate gates). For a start, something along the lines of Figure 5.7(a) is a good way to illustrate the operation of a tristate gate; this figure may also be useful in explaining the functionality of a multiplexer. The figure shows two data inputs A, B connected to a bus C. In general, the control input S could be independently generated for each of the two tristate gates (for example, to disconnect both of them from the bus). One could also easily generalize this to connect several data inputs to the bus (see



(a) (b)

Figure (5.7) Part (a) shows a multiplexer with data inputs A, B output through a bus C. Control input S selects a data input to connect to the bus. Part (b) shows the structure of an n-input bus in which each control bit S_i independently decides if data input A_i is connected to the bus.

Figure 5.7(b)).

In Section 5.1, we introduced the idea of an interconnection network to connect communicating elements. The bus is an important interconnection network in modern computing systems; the PCI bus [12], Ethernet [13], FireWire (IEEE 1394) [14] and USB [15] are commonly used standards. The term "bus" is used today to mean a network with the functionality illustrated in Figure 5.7. The term *multidrop bus* is often used to mean a bus constructed by connecting elements to a wire as in Figure 5.7. Buses also model other communication systems with shared resources (such as wireless systems in which a single frequency may be shared by multiple users).

Instruction Notes for Section 5.2: The discussion on fan-in and fan-out circuits directly pertains to digital logic, so much of this could be an in-class activity. The instructor could further reinforce the idea of trees and PDC, possibly introduced earlier. One could also illustrate an unbalanced higher depth tree (as a bad choice for fan-in or fan-out) and explain why it entails a higher gate delay; the prefix circuit of Figure 5.15 illustrates a high delay fan-in. This ties in to circuit delay (in logic), algorithmic time and network latency in very similar ways.

At this point, the generalization of trees to graphs should, in our view, be only with the purpose of emphasizing the acyclic and sparse nature of trees. Later these ideas can

generalize to directed acyclic graphs (for reconvergent circuits) and directed graphs (state diagrams).

The ideas of broadcast and multicast are, in principle, analogous to signal fan-out; convergecasting is analogous to fan-in. A modest investment of time here could motivate students to see the wider-than-expected scope of the principles studied in digital logic (and, in general, other courses).

If recursive decomposition has been explored earlier (for example in binary number systems), then fan-in and fan-out trees present a simple way to further reinforce the idea.

The discussion on parallelism is perhaps best done as an off-line exercise. The purpose of this discussion could be to lay the groundwork for subsequent discussion (in sequential circuits and Verilog). Some points that could be emphasized at this stage include that, in general, both delay and efficiency decrease with increased parallelism. While delay is easily appreciated in combinational circuits, efficiency requires reuse of combinational blocks; sequential circuits will provide a way for this reuse (see "Bit Serial Adder" of Section 5.6.2).

In many digital logic texts, tristate gates are introduced in the context of a bus. The discussion typically explains how the gates can effectively connect or disconnect a signal source from the bus. It is easy to use this to further build on the idea of interconnection networks. Another important PDC point to note here is that of exclusive access to a shared resource. While the mechanism for providing this exclusive access may differ across environments, the underlying ideas and issues have many similarities whether the environment is a shared bus, a shared frequency or wavelength in a multiaccess system, or a memory module in a shared memory system.

5.3 Combinational Logic Synthesis and Analysis

In this section we will consider two broad topics: timing analysis and Karnaugh maps.

5.3.1 Timing Analysis

This part of the discussion should be introduced only after students have been exposed to an idea of propagation delay of gates and circuit delay. The Monopoly tournament example of Section 5.2.1 (see page 147), touched on these ideas. Here we build on this initial exposure to introduce directed acyclic graphs (DAGs) [16171617] that capture task dependencies and play an important role in several scheduling and resource allocation problems. We also use the setting to discuss data hazards and synchronization.

Consider the multiplexer circuit implementing the function $C = AS + BS'$ shown in Figure 5.8.

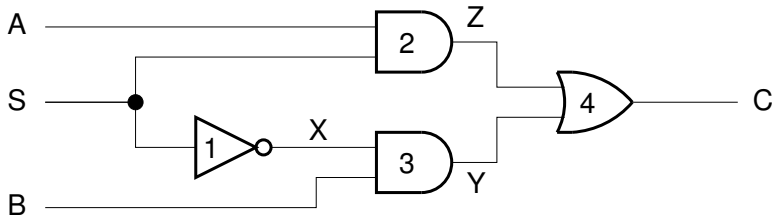


Figure (5.8) A 2-to-1 multiplexer circuit.

Directed Acyclic Graphs: If the propagation delay of gate i (for $1 \leq i \leq 4$) is t_i, then after any change in the input, the values of Z and Y are settled (in steady state) in at most t_2 and $t_1 + t_3$ time, respectively. Thus, the multiplexer output is settled in at most $t_4 + \max\{t_2, t_1 + t_3\}$ time. Clearly this has some of the ideas introduced in the Monopoly example. The new idea here is that of reconvergent paths that are not present in trees (here input S flows through two paths to get to output C).

If each gate (or module) of a combinational circuit corresponds to a node and a connection between them corresponds to a directed edge, then the circuit generates a graph. Because combinational logic is inherently without feedback, its graph is acyclic and forms a directed acyclic graph (DAG). In fact, the tree topology that we discussed in Section 5.2.1 is a special case of a DAG. The DAG of the multiplexer example of Figure 5.8 is shown in Figure 5.9. The node named "A" in the DAG represents the module generating the signal

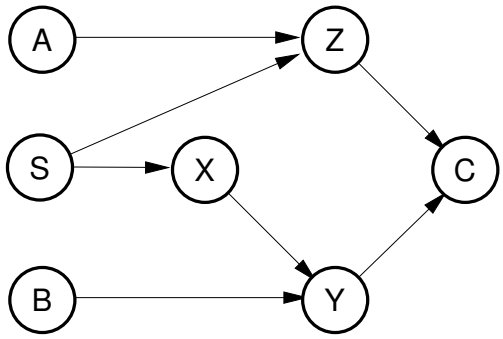


Figure (5.9) The DAG corresponding to the circuit of Figure 5.8.

named "A" in the circuit of Figure 5.8; other nodes of Figures 5.9 and 5.8 have a similar correspondence.. Each node in the DAG may represent a task and the DAG edges could represent dependencies. Therefore a node named α in a DAG cannot be "executed" unless nodes with incoming edges to α have been executed. If node α requires t_α time, the time needed to execute the DAG of Figure 5.9 (once input nodes A, S and B have been obtained) is at least $t_C + \max\{t_Z, t_X + t_Y\}$, which, not surprisingly, closely corresponds to the time for the circuit of Figure 5.8.

In general, a DAG could be used to represent dependencies in a variety of computations. Consider, for example, a finite impulse response (FIR) filter of the first order that transforms an input signal $s(t)$ at time t to an output $c(t)$ given by

$$c(t) = a \cdot s(t) + b \cdot s(t-1).$$

If $x(t) = s(t-1)$, $z(t) = a \cdot s(t)$ and $y(t) = b \cdot x(t)$, then $c(t) = y(t) + z(t)$. Clearly, the computation that this FIR filter represents corresponds to the DAG of Figure 5.9.

Typically, a faster circuit or algorithm uses more resources (for example area, power, memory or processors). Therefore, there is a benefit to slowing down a part of the circuit or algorithm, if possible. As noted earlier, the DAG's execution time is $t_C + \max\{t_Z, t_X + t_Y\}$. Suppose $t_Z < t_X + t_Y$, then the path S, X, Y, C is the *critical path* of the DAG; the time to execute the nodes on a critical path is the minimum time to execute the entire DAG. All

nodes outside the critical path can be suitably slowed down for potential improvements in resource usage. For the example, the execution of nodes X, Y and Z should be adjusted so that (ideally) $t_Z = t_X + t_Y$. This could entail using a slower (and cheaper) execution at node Z than at nodes X and Y. This idea is central in digital design and is discussed further in Section 5.4.1 in the context of fast adders.

It should also be pointed out that DAGs have several applications in computer science and engineering [16171617]. One important application is in modern compilers for parallel systems (including multicore environments). A given piece of (typically sequential) code is analyzed by the compiler to construct a DAG that establishes dependencies among threads in the code. The compiler then translates this information into code that can execute in parallel, ensuring that dependencies are respected and data hazards are prevented, while striving to reduce resources used (time, power) [17].

Data Hazards and Synchronization: Static logic hazards in digital circuits can be used to illustrate synchronization and data hazards. Consider the circuit of Figure 5.8 with inputs $A = B = S = 1$. Clearly, the output is $C = 1$. Changing S from 1 to 0 should not change C, unless the inputs to gate 4 settle at different times. Suppose $t_2 \ll t_1 + t_3$, then the output of gate 2 (signal Z) could settle much earlier than that of gate 3 (signal Y). Consequently, gate 4 could first react to the change of Z, and then to the change of Y. In the example at hand when S changes from 1 to 0, Z may change to 0 before Y changes to 1. Consequently both inputs to gate 4 are 0 and the output $C = 0$. When X then Y ultimately change to 1, C goes back to 1. This glitch in the output, if wide enough, could cause elements using C as an input to produce incorrect results. It is also possible that $t_2 \gg t_1 + t_3$ in which case a change of S from 0 to 1 causes a glitch. In short the problem arises due to gate 4 using information that is not yet current.

This situation has a counterpart in parallel and distributed systems. Suppose that processes Y and Z of Figure 5.9 are generating data for a third process C and passing this data through shared memory locations M_y and M_z. Process C should ensure that the values in M_y and M_z are updated before it proceeds to use them; otherwise the computation could

produce an incorrect result. This situation could be particularly problematic if, say, processor 1 is sequentially executing processes Y and C and processor 2 is executing process Z. If one is not careful, processor 1 could, after executing process Y, move on to process C, without ensuring that processor 2 has completed process Z. This problematic situation is an example of a *data hazard* [17], one with which most parallel and distributed systems must grapple.

Coming back to the multiplexer circuit, one solution to the problem described above is to ensure that gate 4 receives the new values of Z and Y only after both have reached steady state. This can be done by inserting a a 2-bit register before gate 4. (As a practical matter since gate 4 itself does not cause an erroneous behavior of consequence, a 1-bit register can instead be placed after gate 4.)

The corresponding solution for processes Y, Z, C is to place a *barrier* or a synchronizing stage before process C can start. A barrier [18], which causes a set of processes (or threads) to stop until all processes in the set have completed, is a common method of synchronization in parallel and distributed systems. This form of synchronization is also used in other everyday situations, such as not starting a game until all players are present or not counting (or announcing) election results until voting has closed in all precincts.

5.3.2 Karnaugh Maps

Karnaugh maps, a staple in digital logic, can serve as much more than a tool to minimize the number of prime implicants in a Boolean expression. In this section we detail how a Karnaugh map can be used to introduce the torus, mesh and possibly the hypercube topologies. The cells of a Karnaugh map are arranged so that adjacent cells (to within wraparound and across dimensions) can be combined into a prime implicant. Specifically, define two cells to be *adjacent* if and only if their corresponding expressions (minterms) differ in only one literal. For example in a three-variable map, the cell (minterm) corresponding to xyz is adjacent to $x'yz, xy'z, xyz'$, but not to $x'y'z$. Define the *adjacency graph* of a Karnaugh map to be a graph with cells as vertices and an edge connecting each vertex pair

corresponding to a pair of adjacent cells. This adjacency graph is a hypercube [6]. This is because two Karnaugh map cells are adjacent if and only if their expressions differ in exactly one variable. The corresponding condition for two nodes of a hypercube to be connected is for the binary addresses of the two nodes to differ in exactly one bit. Karnaugh maps flatten and slice the hypercube to fit a 2-dimensional depiction.

Figure 5.10 shows the adjacency graph for 3- and 4-variable Karnaugh maps. Figure 5.11

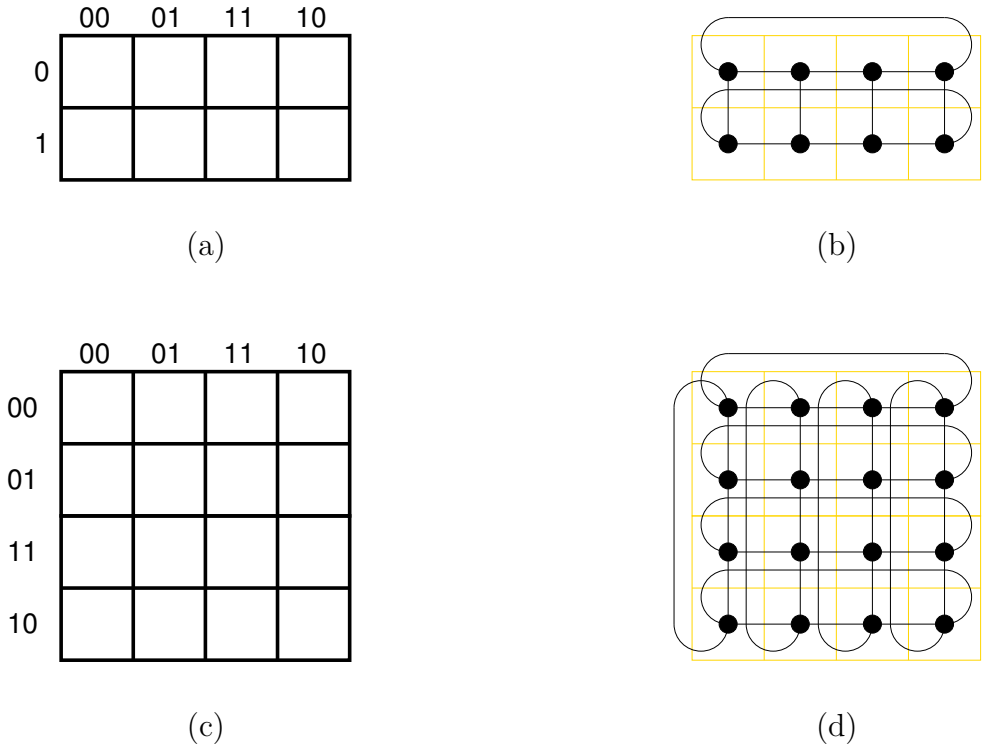
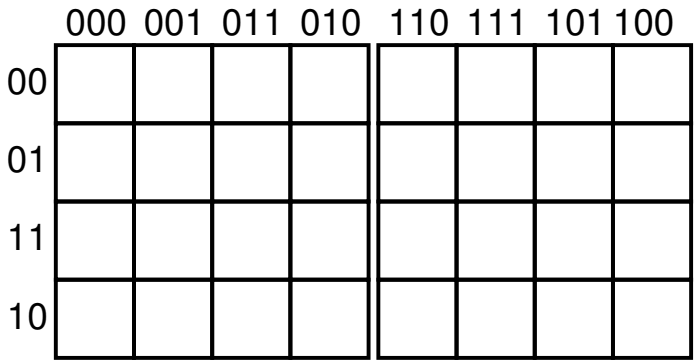
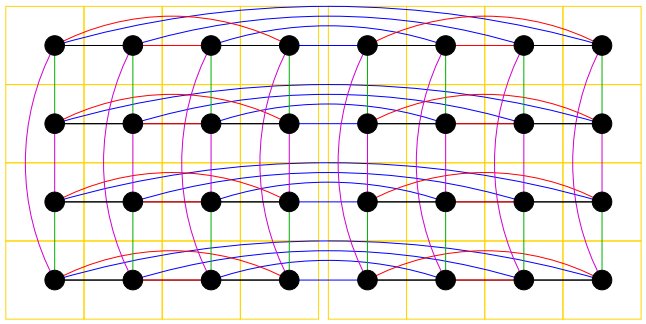


Figure (5.10) Cell layouts (a) and (c) and adjacency graphs (b) and (d) of 3- and 4-variable Karnaugh maps.

shows a 5-variable Karnaugh map and the corresponding 5-dimensional hypercube. It is not difficult to observe that the adjacency graphs of the 3- and 4-variable Karnaugh maps are, in fact, 2×4 and 4×4 tori. (Observe that these are the same graphs as 3- and 4-dimensional hypercubes.) One could now easily extend the small tori to a general $R \times C$ two-dimensional torus and an $R \times C$ two-dimensional mesh (torus without the wraparound connections). An extension to higher dimensional meshes and tori is also possible. The mesh and torus



(a)



(b)

Figure (5.11) Cell layout (a) and adjacency graph (b) of a 5-variable Karnaugh map.

are arguably among the most widely used topologies [6]. The IBM Blue Gene/L has a 3-dimensional torus [19] connecting its processors. The Cray Seastar interconnect is also a 3-dimensional torus [20] and the Fujitsu K Computer uses a 6-dimensional torus [21]. The planar structure of the 2-dimensional mesh also makes mesh-like interconnects common in IC chips (for example, in FPGAs [22232223]). Ring based topologies (1-dimensional torus) are also common in local area networks and distributed computing frameworks [24252425].

Instruction Notes for Section 5.3: The relation of fan-in and fan-out circuits to directed trees easily generalizes to the relation of logic circuits to directed graphs. One could observe here that a main difference between combinational and sequential logic is the lack of feedback

cycles in the former. This allows for the introduction of DAGs. However, introducing this term would be much more meaningful if one could tie it to the critical path, data hazards and synchronization ideas. While this would probably require too much time for an in-class exposition, it holds the potential to motivate and clarify slightly advanced digital logic ideas such as design optimization principles and fault tolerance; some of these may be explored in a design course/lab setting for digital logic.

Synchronization could be presented here as a method to keep signals with vastly different or unpredictable delays on the same page. This is in line with the barrier examples (game and voting) cited earlier, to which students would probably relate easily. Later in the context of sequential circuits, it would be beneficial to draw upon this intuitive understanding of synchronization to fully appreciate the advantage of clocked circuits.

Karnaugh maps should be presented as simply a different way of drawing a truth table, a way that allows us to see spatial relationships easily. An initial (obvious) reflection of this spatial relationship (for up to 4-variable Karnaugh maps) is through a 2-dimensional mesh (without the wraparound connections). The notion of adjacency (with the purpose of minimizing a function) for cells in these Karnaugh maps requires wraparound connections and this naturally leads to a torus. All along the way the ideas of graphs and interconnection networks can also be easily reinforced.

The Karnaugh maps of Figures 5.10 and 5.11 number rows and columns according to the reflected Gray code sequence. This could be used to further reinforce recursive thinking.

Extension to a hypercube is difficult in a first digital logic course, but not much more than explaining adjacencies in 5- and 6-variable Karnaugh maps. In this context, we note that some texts view, for example, a 5-variable Karnaugh map in terms of two 4-variable maps, but without reflecting the indices of the second 4-variable map. This will alter the edges along the fifth dimension in the corresponding hypercube of Figure 5.11(b), but does not change the underlying idea.

5.4 Combinational Building Blocks

In this section we will use two commonly discussed combinational structures, adders and multiplexers, to illustrate several important concepts, including parallel prefix, and a variant of Amdahl's Law.

5.4.1 Adders

Adders are one of the most commonly discussed combinational circuits in a course on digital logic and offer unique opportunities for exploring PDC topics. Consider the standard ripple carry adder illustrated in Figure 5.12. The bottleneck of the ripple-carry adder's speed

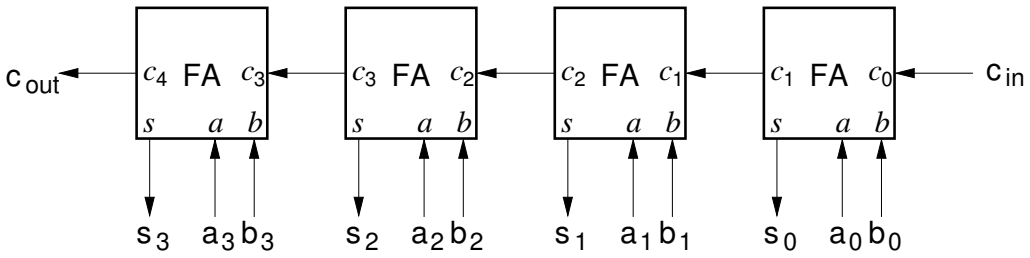


Figure (5.12) A 4-bit ripple carry adder. Each box labeled "FA" is a full-adder accepting data bits a_i, b_i, carry-in bit c_i, and generating the sum bit s_i and the carry-out bit c_{i+1}.

is the sequential generation of carry bits. That is, the longest path in the circuit traverses the carry lines. Students should note the important design principle of improving overall performance of a module by reducing its bottleneck (or critical path). This principle can also be seen as being somewhat similar to Amdahl's Law [26272627] that states that the part of an algorithm that is inherently sequential is the speed bottleneck in parallelizing this algorithm. For the ripple-carry adder (all of whose component full-adders run in parallel) the carry generation used reflects an inherently sequential task. Here the speed of the circuit is bottlenecked by this sequential part (namely ripple carry generation).

In the normal course of explaining a ripple-carry adder, one would note that for $i \geq 0$, the $(i+1)^{\text{th}}$ carry c_{i+1} is given by the following recurrence in which "+" and "$\cdot$" represent

the logical OR and AND operations.

$$c_{i+1} = a_i \cdot b_i + (a_i + b_i) \cdot c_i = g_i + p_i \cdot c_i. \tag{5.1}$$

Here a_i, b_i are the i^{th} bits of the numbers to be added and g_i, p_i are called the i^{th} *carry-generate* and *carry-propagate* bits. Given this first order recurrence in which c_{i+1} is expressed using c_i, a student in a first digital logic class (typically a freshman or sophomore) may incorrectly assume that carry computation (or for that matter any recurrence) is inherently sequential and that the adder circuit cannot overcome the delay of a ripple-carry adder. This presents the opportunity to use a carry look-ahead adder (Figure 5.13) to show how seemingly sequential operations can be parallelized. In particular, one could introduce the idea of parallel prefix, a versatile PDC operation, used in numerous applications ranging from polynomial evaluation and random number generation to the N-body problem and genome sequence alignment [28292829].

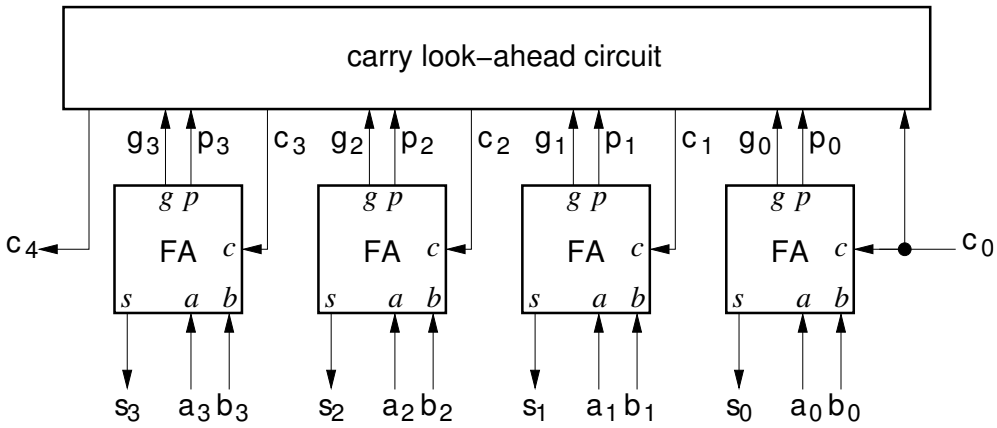


Figure (5.13) A 4-bit carry look-ahead adder. Each box labeled "FA" is a modified full-adder accepting data bits a_i, b_i, carry-in bit c_i, and generating the sum bit s_i and the carry generate and propagate bits g_i, p_i.

Prefix Computation: Let $\otimes$ be any associative operation. A *prefix computation* with respect

to $\otimes$ on inputs $\alpha_0, \alpha_1, \cdots, \alpha_{n-1}$ produces outputs $\beta_0, \beta_1, \cdots \beta_{n-1}$, where

$$\beta_i = \bigotimes_{j=0}^{i} \alpha_j = \alpha_0 \otimes \alpha_1 \otimes \cdots \otimes \alpha_i, \quad \text{for any } 0 \leq i < n.$$

A prefix computation is a special case of a first order recurrence in that $\beta_i = \beta_{i-1} \otimes \alpha_i$. We now develop the relationship in the other direction and indicate how the first order recurrence for the carry in Equation (5.1) can be expressed as a prefix computation.

Let x_1, y_1, x_2, y_2 be four binary signals. Define a binary operation $\odot$ on two doublets (pairs) $\langle x_1, y_1 \rangle$ and $\langle x_2, y_2 \rangle$ to produce the doublet $\langle x, y \rangle = \langle x_1, y_1 \rangle \odot \langle x_2, y_2 \rangle$ defined as follows:

$$x = x_2 + x_1 y_2 \quad \text{and} \quad y = y_1 \cdot y_2,$$

where $+$ and $\cdot$ denote the OR and AND operations. The operation $\odot$ can be proved to be associative. It can be shown that for $j \geq 0$ and doublets $\langle x_j, y_j \rangle$,

$$\bigodot_{j=0}^{i} \langle x_j, y_j \rangle \;=\; \langle G_i, P_i \rangle, \quad \text{where} \quad G_i = \sum_{j=0}^{i} x_j \left(\prod_{k=j+1}^{i} y_k \right) \quad \text{and} \quad P_i = \prod_{j=0}^{i} y_j. \qquad (5.2)$$

Notice that this represents a prefix computation on the doublets $\langle x_i, y_i \rangle$ with respect to the operation $\odot$.

Coming back to the carry recurrence, let $x_0 = g_0$, $y_0 = c_0 p_0$ and for all $i > 0$, let $x_i = g_i$, $y_i = p_i$. Let bits G_i and P_i be obtained by a prefix computation as described in Equation (5.2). It can be shown that the i^{th} carry is $c_i = G_i + P_i$. In other words, the carry bits can be computed by a prefix computation.

The ripple carry circuit corresponds to a very slow prefix computation. A carry-look-ahead adder employs a fast prefix computation circuit to generate the carry bits. In general, by adopting different prefix circuits for carry generation, one could create adders with different cost-performance trade-offs. In Section 5.5.1 we will revisit prefix computations in the context of counters. It should also be noted that, in general, any linear recurrence can be converted to a prefix computation [30313031].

5.4.2 Multiplexers

Earlier, we introduced recursive thinking in the context of number representations (Section 5.1), expressed fan-in and fan-out recursively (Section 5.2.1), and discussed carry look-ahead in terms of a recurrence relation (Equation (5.1)). We will show in Section 5.5.1 that prefix computation circuits (that compute a recurrence relation) also lend themselves to recursive decomposition. In this section, we will use multiplexers to reinforce recursive decomposition. It is well known that large multiplexers can be constructed using smaller multiplexers (see Figure 5.14). However, the construction is often not cast recursively or in terms of a divide-and-conquer strategy. For example, the multiplexer of Figure 5.14 can be viewed as follows. The problem is to select one input out of a set of 64. Partition the 64-input problem into four subproblems, each with 16 inputs. Then, recursively solve these four subproblems to obtain one (intermediate) selection from each subproblem. Finally, select one of the 4 intermediate selections using a 4-to-1 multiplexer. Similar decompositions can be illustrated through one-hot decoders and priority encoders (that are typically taught with multiplexers).

These ideas can be generalized to the task of recursively constructing a 2^n-to-1 multiplexer using 2^r-to-1 multiplexers for $r < n$. Observe that this recursive decomposition corresponds to a 2^r-ary tree and should be related to the ideas in Section 5.1. It should also be observed that this recursive decomposition is an algorithmic notion. It is applied here to a piece of hardware, but can equally apply to software. For example, merge sort [9] provides a classic illustration of such a recursive decomposition in a divide-and-conquer algorithm. In the context of PDC, the merge sort example can also be expressed in terms of a DAG with an initial divide phase to run smaller sorting instances on individual processors, then merging the sorted sequences in the conquer phase; this merge can also illustrate synchronization.

The fact that important algorithmic paradigms like divide-and-conquer apply broadly is an important abstraction for computer science and engineering students.

Instruction Notes for Section 5.4: Normally, instruction on adders will cover the recurrence

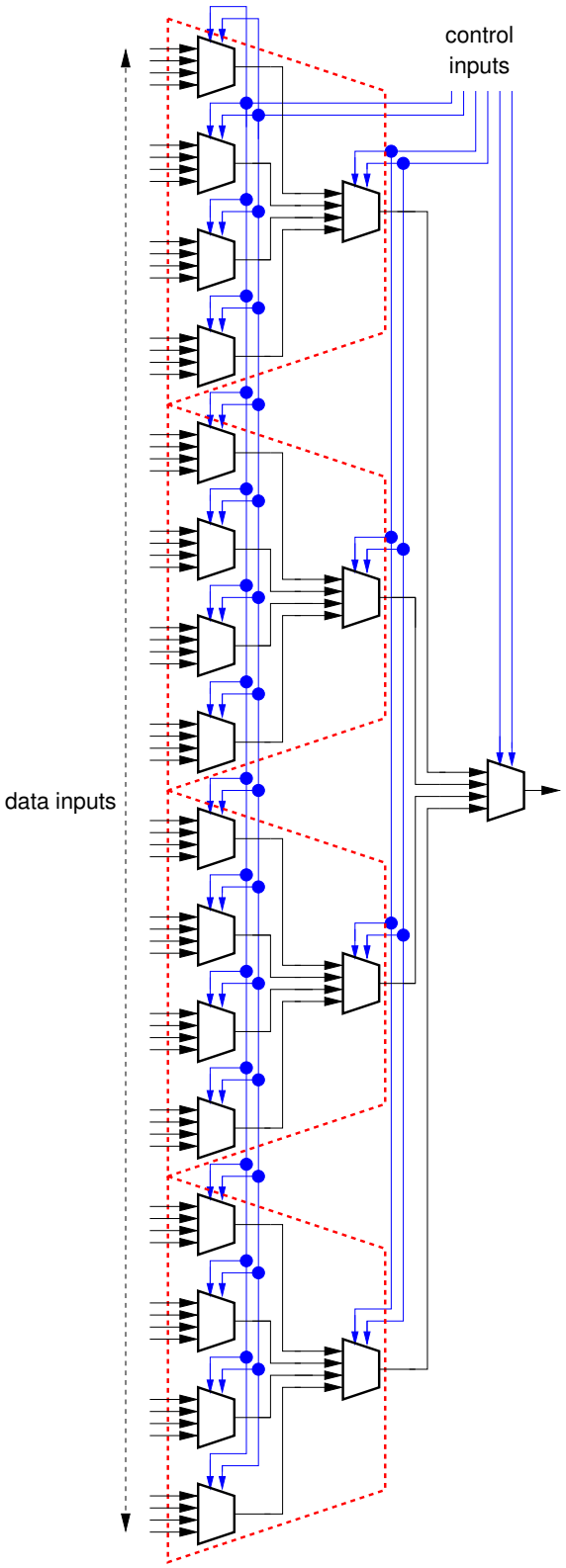


Figure (5.14) A 64-to-1 multiplexer constructed out of twenty-one 4-to-1 multiplexers. Portions outlined in dashed lines indicate recursively constructed 16-to-1 multiplexers.

relation of Equation (5.1) and the linear delay of ripple-carry adders. It is easy to include in this a discussion of how the carry propagation bottlenecks the performance of the entire adder. Before getting to the carry-look-ahead adder, a brief detour into prefix computations may be worthwhile. A simple example of prefix AND has several benefits: (1) by representing it as a recurrence, its structural similarity to carry-propagation can be appreciated, (2) by providing the simple circuit for prefix AND (for example from Figure 5.15), students can see the similarity to ripple-carry, (3) from Figure 5.16, students can see that the seemingly sequential nature of a recurrence can be misleading, (4) these prefix circuits can be used later in the context of counters in Section 5.5.1. Although one could go into additional details, it may suffice to indicate that the carry propagation recurrence can be cast as a prefix computation. Some of these details would surface in the course of discussing carry-look-ahead adders.

The recursive decomposition in multiplexers and decoders is covered in the normal course of a digital logic course. One could point students to notice that the solution to the larger problem is in terms of smaller instances of the same problem. One could use other circuits, such as a barrel shifter, to similar effect.

5.5 Counters and Registers

In this section we use counters to reinforce prefix computation and shift registers to illustrate pipelining.

5.5.1 Counters

Counters are an important sequential building block, not just from their utility in logic circuits, but also as a vehicle to explain concepts in digital logic (for example, a counter's operation lends itself to a simple and intuitive introduction to state diagrams and transition expressions). Here we cast counters in a direction that we expect will help the student grasp both counters and prefix computations better. We will also use asynchronous (ripple) counters to illustrate the benefits and pitfalls of synchronous and asynchronous computation

environments.

Prefix Computations in Counters: An n-bit binary counter gets to the next state by incrementing (modulo 2^n) the binary representation of the current state. This boils down to complementing bits up to the least significant 0 in the current state (and leaving the remaining higher bits unchanged). For instance, let the current state of an 8-bit counter be 1011$\underline{0}$111. Here the least significant 0 (underlined) is in position 3. The next state

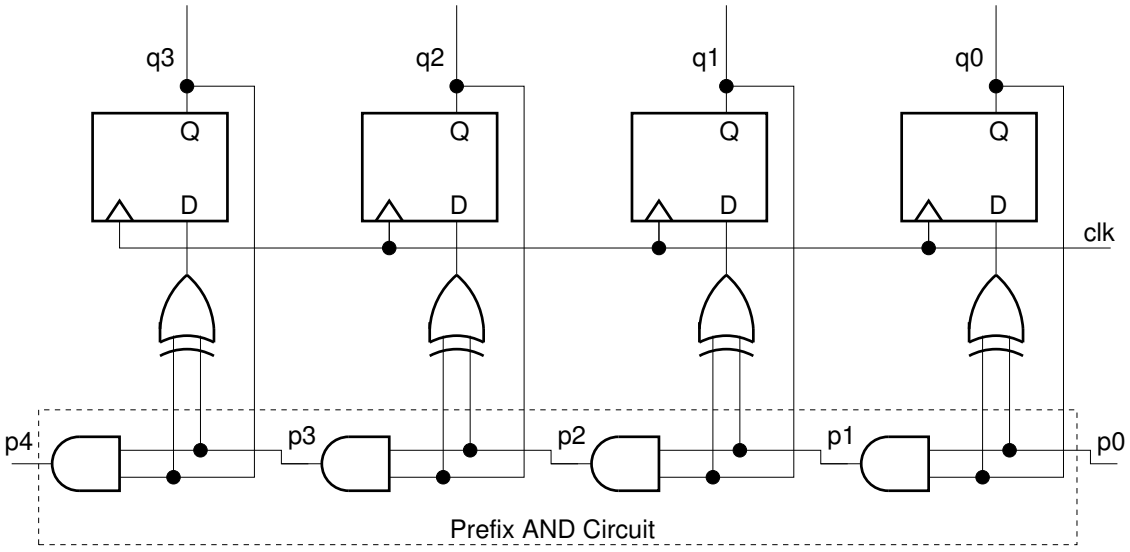


Figure (5.15) A 4-bit counter. Typically p_0 is the counter-enable signal and p_4 is the "ripple carry-out" signal used for counter expansion.

is $10110111 + 1 = \underbrace{1011}_{\texttt{unchanged}}\ \overbrace{1000}^{\texttt{complemented}}$. Determining the position of the least significant 0 reduces to computing the prefix AND of the bits. Specifically, let the current state (or count) be $Q_t = q_{n-1}q_{n-2}\cdots q_1q_0$ (in binary). We now construct an n-bit "prefix" sequence $P = \langle p_{n-1}, p_{n-2}, \cdots, p_1, p_0\rangle$ in which $p_0 = 1$ and for $0 < i < n$, we have $p_i = q_0 \cdot q_1 \cdot \cdots \cdot q_{i-1} = p_{i-1} \cdot q_{i-1}$, where "$\cdot$" denotes the logical AND operation. Let $0 \le i_0 \le n$ be the position of the least significant 0 ($i_0 = n$ indicates that $q_i = 1$ for all $0 \le i < n$). Then $p_i = 1$ if and only if $i \le i_0$. It is easy to show that the result of incrementing $Q_t = q_{n-1}q_{n-2}\cdots q_{i_0+1}q_{i_0}q_{i_0-1}\cdots q_1q_0 = q_{n-1}q_{n-2}\cdots q_{i_0+1}01\cdots 11$ is

$Q_{t+1} = \underbrace{q_{n-1}q_{n-2}\cdots q_{i_0+1}}_{\texttt{unchanged}}\overbrace{q'_{i_0}q'_{i_0-1}\cdots q'_1q'_0}^{\texttt{complemented}} = q_{n-1}q_{n-2}\cdots q_{i_0+1}10\cdots00$. Noting that for the exclusive-OR operation $\oplus$ and any variable v we have $v \oplus 1 = v'$ and $v \oplus 0 = v$, one could say that $Q_{t+1} = Q_t \oplus P$ (using bitwise exclusive-OR operations). That is, given the current state Q_t, the next state is easily computed if P is available. Figure 5.15 shows a standard 4-bit counter cast in this mold.

Notice that the clocking rate for an n-bit counter constructed along the lines of Figure 5.15 must accommodate at least $n-1$ AND gate delays. Clearly, this does not scale for large n (in much the same way that a ripple-carry adder does not scale for large addends). Standard textbook solutions include using larger-input AND gates such as the design shown in Figure 5.16 for an 8-bit counter. Here the delay is that of an n-input AND gate and

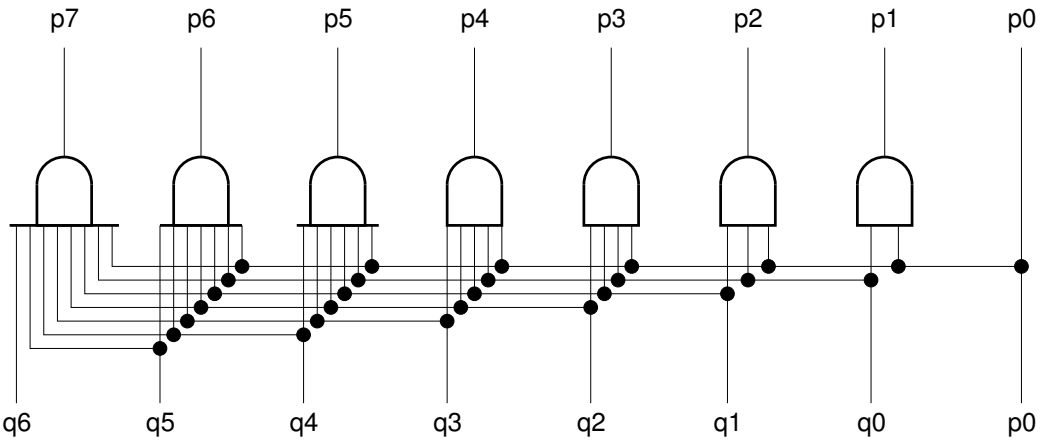
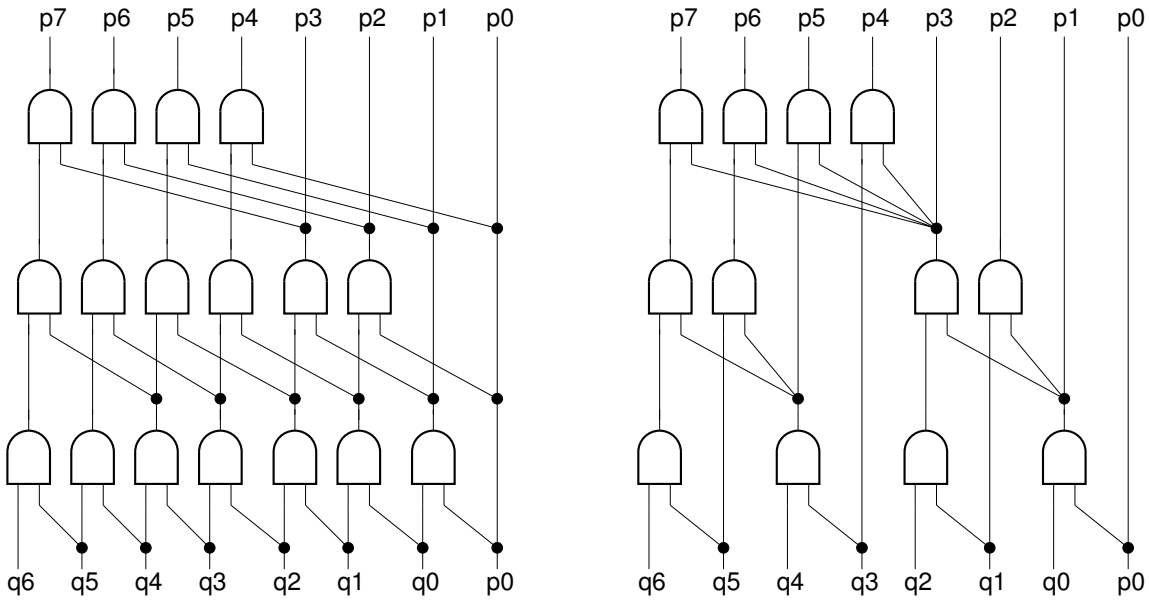


Figure (5.16) A prefix AND circuit for an 8-bit counter. For clarity, the ripple carry-out signal p_8 has not been generated.

the delay for fanning a signal out to n places; this delay typically grows sublinearly with n. One could replace the circuit of Figure 5.16 by any other prefix circuit (see, for example, Figure 5.17).

A combination of one of these circuits with the method of Figure 5.15 to accommodate the availability of high fan-in gates may provide an optimal middle ground for large values of n.



(a) The Kogge-Stone circuit [32] (b) The Ladner-Fischer circuit [33]

Figure (5.17) Example of prefix AND circuits for an 8-bit counter. These circuits exhibit trade-offs between the number of gates used and their fan-out. These circuits can be used with any associative operation, including one for carry propagation.

Ripple Counters: It is well known that ripple counters (Figure 5.18) can be very cheap to implement and quite fast for small sized counters. However, as the size of the counter grows, flip-flop delays accumulate, necessitating a reduction in the clock speed. As shown in Figure 5.18, the clock of an n-bit ripple counter should be wide enough to accommodate n flip-flop delays.

The clock in synchronous counters of the form illustrated in Figure 5.15 also needs to accommodate n gate delays. However, there is a significant difference in the two situations.

In general a synchronous circuit must use a clock that is wide enough to accommodate delays to flip-flop inputs. Although this worst-case clock considerably simplifies circuit design and ensures correct flip-flop state changes, it fails to exploit the speed of a typical case behavior, while an asynchronous circuit can (albeit at the cost of more complex design procedures and potentially unreliable operation).

These trade-offs present significant design choices for parallel and distributed systems. The Internet (or a computer network, in general) is typically distributed over a large geo-

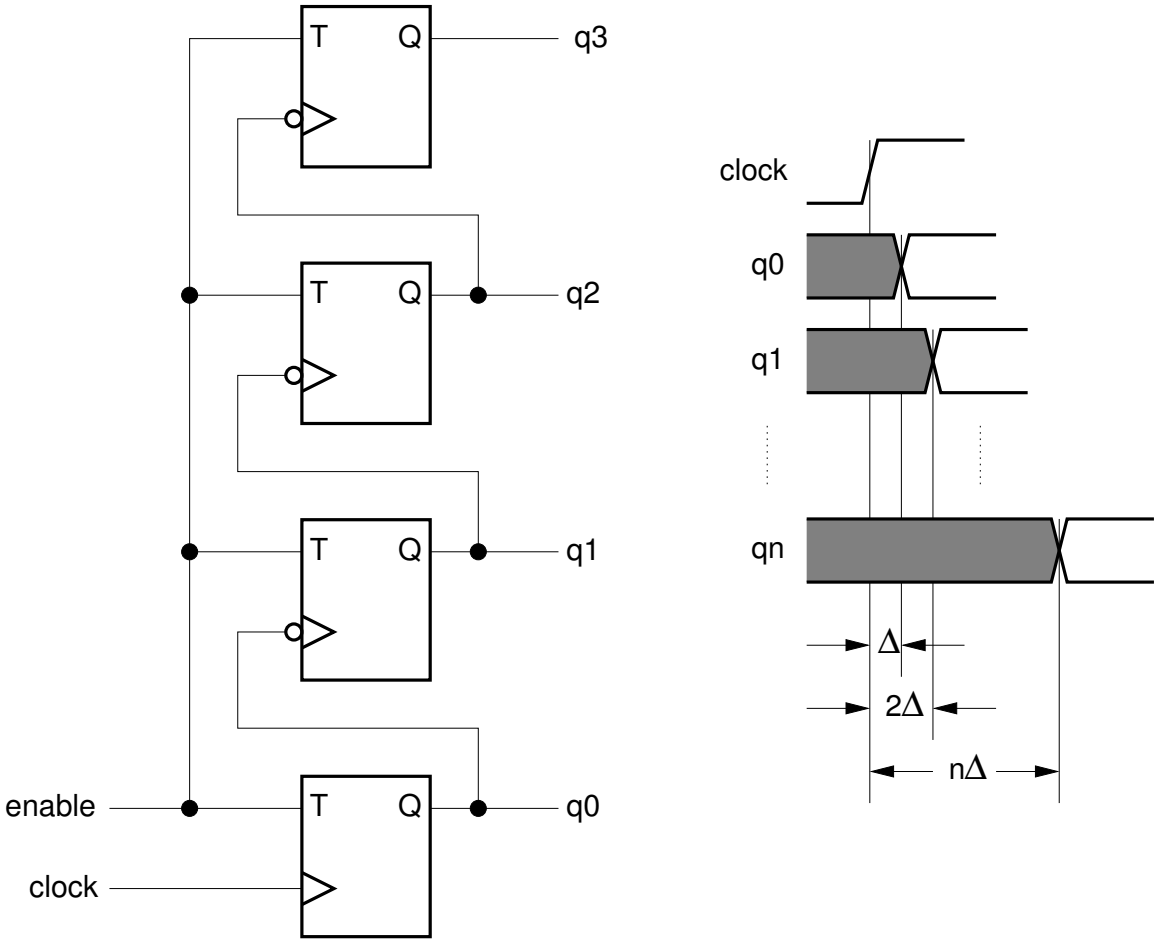


Figure (5.18) Structure of a ripple counter and its timing diagram; flip-flop propagation delay is denoted by Δ.

graphic area. Consequently there are large (variations in) delays associated with communication between nodes in the network. Clearly, a synchronous system is not suitable here, as that would entail very long waits to accommodate all delays. In other cases, the nature of the problem or computing environment induces large variations or uncertainties among communicating elements; again an asynchronous environment is better here. Consider a school of sardines that collectively swim as a school, but each fish individually takes its cues from its immediate neighborhood. If all the sardines need to proceed in lock step, the school's response would not be fast enough to evade a predator. Such event- and neighborhood-driven responses are typical in distributed computing systems that operate asynchronously. Other

situations, however, call for synchronous operation. An an example, many medium access control (MAC) protocols or leader election algorithms use a slotted (synchronous) mode to resolve contention. As opposed to the previous situations where all entities need to react quickly, contention resolution tries to systematically reduce attempts to secure a shared resource (such as a wireless channel), ideally to one contender. While the slotted mode reduces the speed of operation of individual nodes, it facilitates a quicker resolution of contention [34353435]. In most modern parallel computing environments, a suitable mix of synchronous and asynchronous operations is used. In general, processes may proceed asynchronously (to fully leverage local speeds) and then explicitly synchronize when they reach a point where all (or some) processes need to be on the same page.

5.5.2 Shift Registers

A standard use of a shift register (with serial or parallel load) is to illustrate the conversion of data between serial and parallel formats. This clearly reinforces the difference between serial and parallel. The standard portrayal of a shift register can also introduce (ideal) pipelining.

Consider the shift register of Figure 5.19. With new bits a–e coming in, Table 5.3(a)

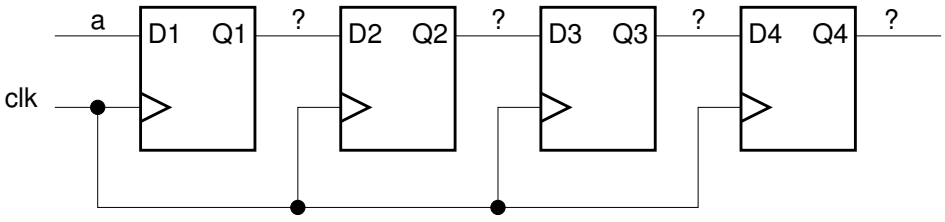


Figure (5.19) A 4-bit shift register. The input D_1 has a binary value a. The question marks at flip-flop outputs indicate don't care values.

shows the movement of these bits through the shift register over clock cycles. The first point to note (even in the understanding of the shift register) is that with each clock tick, all bits shift simultaneously by one position; for example at clock tick 5, e moves from D_1 to Q_1 at the same time as d moves from Q_1 to Q_2 and so on. Therefore, despite the seemingly serial

Table (5.3) An illustration of pipelining. Part (a) shows the shifting of bits through a 4-bit shift register. Part (b) shows the movement of inputs through a 4-stage pipeline. A "–" indicates a don't care value

clock	D_1	Q_1	Q_2	Q_3	Q_4
0	a	–	–	–	–
1	b	a	–	–	–
2	c	b	a	–	–
3	d	c	b	a	–
4	e	d	c	b	a
5	–	e	d	c	b
6	–	–	e	d	c
7	–	–	–	e	d
8	–	–	–	–	e

(a)

time	Input x	$F_1(x)$	$F_2(x)$	$F_3(x)$	$F_4(x)$
0	a	–	–	–	–
Δ	b	a_1	–	–	–
2Δ	c	b_1	a_2	–	–
3Δ	d	c_1	b_2	a_3	–
4Δ	e	d_1	c_2	b_3	a_4
5Δ	–	e_1	d_2	c_3	b_4
6Δ	–	–	e_2	d_3	c_4
7Δ	–	–	–	e_3	d_4
8Δ	–	–	–	–	e_4

(b)

structure of the shift register, the movement of bits is inherently parallel, a fact that many students initially miss. Now consider the 4-stage ideal pipeline of Figure 5.20 (in which each stage requires the same time, Δ, to process any input). For any input x and $1 \leq i \leq 4$,

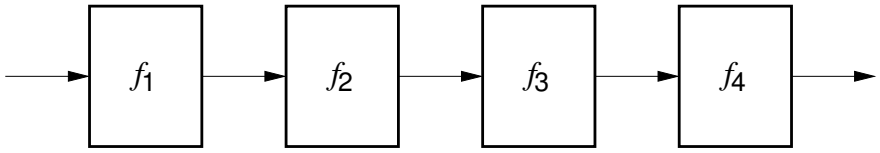


Figure (5.20) A 4-stage pipeline. Stage i computes a function f_i.

let $x_i = f_i(f_{i-1}(\cdots(f_1(x))\cdots)) = F_i(x)$ (say), then Table 5.3(b) represents the flow of data through the pipeline. The similarity between the two tables is unmistakable. While the quantities moving though the pipeline may be different, the pattern of their movement is the same.

It could also be noted that many practical pipelines are not ideal, suffering stalls, timing imbalances and requiring flushing of partly processed inputs. However, the fundamental (and preferred) operation of the pipeline is embodied in the flow of bits in a shift register. Pipelines are common in processors [36]. A classic RISC pipeline includes stages for instruction fetch, decode, execute, memory access and register writeback. Image and video pipelines [37]

are standard in cameras. The idea of pipelining also applies to software, for example, in out-of-order execution of instructions in loops [17].

Instruction Notes for Section 5.5: The standard discussion on synchronous counters uses a circuit similar to that of Figure 5.15. Expressing the counter in terms of prefix AND would serve to explain why the circuit works as a counter; for a first digital logic course, these ideas could be expressed through examples rather than general expressions. If the prefix AND circuits were discussed earlier in the context of adders (see Section 5.4.1), then relatively little time is needed to explain the "slow" and "fast" counters associated with Figures 5.15 and 5.16. One could go further and note that prefix computations have many applications besides counters and there are several methods for performing prefix computations (such as those shown in Figure 5.17). On a more advanced note, it could be observed that the Ladner-Fischer circuit (Figure 5.17(b)) translates to a simple recursive algorithm on the CREW PRAM, a model of parallel computation on which numerous parallel algorithms, algorithmic techniques and complexity results have been developed [31].

The discussion with ripple counters should aim to first distinguish synchronous and asynchronous circuits in basic terms. For example, one could emphasize that for flip-flops (or latches), synchronous operation amounts to triggering at the same time and for correct operation no flip-flop triggers until all flip-flops are ready to trigger. This has similarities to the data hazards and synchronization example of Section 5.3.1. The larger discussion of the advantages and limitations of synchronous/asynchonous design should be placed in the context of most sequential logic in a first digital logic course being synchronous.

A key common point of shift registers and pipelines is that the flow of "data" is parallel. This observation (without reference to pipelines) may help students understand shift registers better. Next, one could consider a (simplified) 4-stage ideal video pipeline (consisting of, for example, A-to-D conversion, noise filtering, color correction, compression) and illustrate how successive frames move through these stages. If one wishes to take this further, the idea of a processor pipeline could also be introduced without too many details. The key points to note here may be that a program is a sequence of instructions stored in memory outside

the processor. The processor must first get (fetch) the instruction from the memory, then it must determine what the instruction means (decode). Only after that can the processor do what the instruction calls for (execute). While instruction 3 is being fetched, instruction 2 could be decoded and instruction 1 executed.

5.6 Other Digital Logic Topics

We have discussed a selection of digital logic topics that can serve to introduce several PDC concepts. However, digital logic is replete with examples and instances that can illustrate PDC concepts. In this section we touch on some of them.

5.6.1 Latches and Flip-Flops

Latches and flip-flops clearly show the difference between asynchronous and synchronous systems. A simple analysis of a latch can also be used to show how different delays on different paths can result in different flip-flop states. (One standard example of that of metastability when an SR-latch inputs change simultaneously from $SR = 11$ to $SR = 00$. The output state depends on the gate delays in the paths to the outputs.) This illustration can generalize to situations in which the order of an execution (used in a distributed algorithm sense) can affect the outcome of an algorithm. Read/write hazards and cache consistency are examples [36].

5.6.2 Finite State Machines

State diagrams of finite state machines (FSMs) can be used to reinforce the idea of a graph introduced in Section 5.2.1 (see page 147). In addition, FSM design examples can also convey PDC ideas. In this section we identify three such commonly used examples.

Bit Serial Adder: A bit-serial adder is a ripple carry adder with only one full adder. An additional flip-flop conveys the carry out of one bit position to the carry-in of the next bit position. This design, which uses few components, but which is slower and has synchronization requirements on the inputs and outputs, illustrates the cost performance trade-offs

inherent in parallelism.

Bus Arbiter: An arbiter selects one out of a set of contenders according to some rule (that can be simple or as complex as the instructor wishes). When placed in the context of the earlier discussion of the bus, this illustrates the overheads inherent in sharing resources.

Pattern Recognizer: A pattern recognizer accepts a stream of input symbols and outputs a 1 whenever the last k inputs coincide with a fixed reference string of length k. The reference string passes through the state diagram unimpeded from the initial state to the ACCEPT state. Any deviation from the reference string sets the progress back a little further away from the ACCEPT state. This is similar to a complex pipeline in which inputs move in tandem until something like a missed prediction causes the pipeline to start over.

5.6.3 Verilog

Most modern offerings of digital logic also include an introduction to a hardware description language such as Verilog. Despite its similarity to C (in syntax and some semantics), Verilog code works very differently from (sequential) C code. This difference could be useful to illustrate the difference between sequential and parallel systems. Verilog code is inherently parallel. For example, two or more instantiations of a module in Verilog result in two or more pieces of hardware that have the potential to operate simultaneously. On the other hand, multiple calls to a C procedure run the procedure sequentially multiple times. Blocking and non-blocking assignments in Verilog also illustrate parallelism and (a)synchrony.

5.6.4 PLDs to FPGAs

Almost all digital logic texts cover programmable logic devices (PLDs) and some devote pages to more advanced devices such as complex-PLDs (CPLDs) and field-programmable gate arrays (FPGAs). FPGAs, in particular, have assumed a mainstream presence in parallel computing as accelerators and in embedded systems. Some modern FPGAs also have embedded processor cores. Depending on the depth of coverage in digital logic, FPGAs could be used to inform students of the wide scope of digital logic in PDC, and perhaps

to introduce concepts ranging from interconnect structures to the role of special purpose accelerators and reconfigurable computing in general [22232223].

5.6.5 Practical Considerations

Clock distribution and skew can reinforce ideas of scaling (seen in the context of fan-in, fan-out, adders and counters). A clock distribution tree can also be used to revisit trees, interconnects, and broadcasting. The tree to depict binary numbers (Figure 5.1) can be extended to explain prefix codes.

REFERENCES

[1] I. Letunic and P. Bork, "Interactive Tree of Life (iTOL): An Online Tool for Phylogenetic Tree Display and Annotation," *Bioinformatics*, vol. 23, pp. 127–128, January 2007.

[2] iTOL: Interactive Tree of Life. [Online]. Available: http://itol.embl.de/

[3] A. Espinoza-Valdez, R. Femat, and F. C. Ordaz-Salazar, "A Model for Renal Arterial Branching Based on Graph Theory," *Mathematical Bioscience*, vol. 225, pp. 36–43, May 2010.

[4] W. J. Dally and B. Towles, *Principles and Practices of Interconnection Networks.* San Francisco, CA, USA: Morgan Kaufman Publishers, 2004.

[5] J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks.* San Francisco, CA, USA: Morgan Kaufman Publishers, 2003.

[6] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* San Mateo, CA, USA: Morgan Kaufman Publishers, 1992.

[7] M. Brazil, R. Graham, D. Thomas, and M. Zachariasen, "On the history of the euclidean steiner tree problem," *Archive for History of Exact Sciences*, vol. 68, no. 3, pp. 327–354, 2014. [Online]. Available: http://dx.doi.org/10.1007/s00407-013-0127-z

[8] C. E. Leiserson, "Fat-trees: Universal Networks for Hardware-Efficient Supercomputing," *IEEE Transactions on Computers*, vol. C-34, pp. 892–901, October 1985.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms.* Cambridge, MA, USA: MIT Press, 2009.

[10] IPv4 AS Core: Visualizing IPv4 and IPv6 Internet Topology at a Macroscopic Scale in 2013. [Online]. Available: http://www.caida.org/research/topology/as_core_network/

[11] L. Stockmeyer and U. Vishkin, "Simulation of Parallel Random Access Machines by Circuits," *Siam Journal on Computing*, vol. 13, pp. 409–422, 1984.

[12] PCI-SIG Specifications. [Online]. Available: http://www.pcisig.com/specifications/

[13] IEEE Standard 802.3-2012: Ethernet. [Online]. Available: http://standards.ieee.org/findstds/standard/802.3-2012.html

[14] IEEE Standard 1394-2008: High Performance Serial Bus. [Online]. Available: http://standards.ieee.org/findstds/standard/1394-2008.html

[15] Universal Serial Bus (USB). [Online]. Available: http://www.usb.org/home

[16] Y.-K. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys*, vol. 31, pp. 406–471, December 1999.

[17] S. Pande and D. P. Agrawal, *Compiler Optimizations for Scalable Parallel Systems: Languages, Compilation Techniques, and Run Time Systems.* Berlin, Germany: Springer verlag, 2001.

[18] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming.* Waltham, MA, USA: Morgan Kaufman Publishers, 2013.

[19] N. R. Adiga, M. A. Blumrich, D. Chen, P. Coteus, A. Gara, M. E. Giampapa, P. Heidelberger, S. Singh, B. D. Steinmacher-Burow, T. Takken, M. Tsao, and P. Vranas, "Blue Gene/L Torus Interconnection Network," *IBM Journal of Research and Development*, vol. 49, pp. 265–276, March/May 2005.

[20] The Cray XT5 System Highlights. [Online]. Available: www.cray.com/Assets/PDF/products/xt/CrayXT5Brochure.pdf

[21] Y. Akima, Y. Takagi, T. Inoue, S. Hiramoto, and T. Shimizu, "The Tofu Interconnect," in *Proceedings of 19th IEEE Symposium on High Performance Interconnects*, 2011, pp. 87–94.

[22] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Burlington, MA, USA: Morgan Kaufman Publishers, 2008.

[23] R. Vaidyanathan and J. L. Trahan, *Dynamic Reconfiguration: Architectures and Algorithms*. New York, NY, USA: Kluwer Academic Publishers, 2004.

[24] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. Hoboken, NJ, USA: John Wiley Interscience, March 2004.

[25] N. A. Lynch, *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[26] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," *IEEE Computer*, vol. 41, pp. 33–38, July 2008.

[27] M. A. Suleman, Y. N. Patt, E. Sprangle, A. Rohillah, A. Ghuloum, and D. Carmean, "Asymmetric Chip Multiprocessors: Balancing Hardware Efficiency and Programmer Efficiency (TR-HPS-2007-001)," http://hps.ece.utexas.edu/pub/TR-HPS-2007-001.pdf, February 2007.

[28] S. Aluru, N. Futamura, and K. Mehrotra, "Parallel Biological Sequence Comparison using Prefix Computations," *Journal of Parallel and Distributed Computing*, vol. 63, pp. 264–272, March 2003.

[29] G. E. Blelloch, "Prefix Sums and their Applications," in *Synthesis of Parallel Algorithms*, J. H. Reif, Ed. Morgan Kaufman Publishers, 1993, pp. 35–60.

[30] H. P. Dharmasena and R. Vaidyanathan, "The Mesh with Binary Tree Networks: An Enhanced Mesh with Low Bus-Loading," *Journal of Interconnection Networks*, vol. 05, pp. 131–150, May 2004.

[31] J. Jájá, *An Introduction to Parallel Algorithms*. Reading, MA, USA: Addison Wesley Publishing Co., 1992.

[32] P. M. Kogge and H. S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE Transactions on Computers*, vol. C-22, pp. 786–793, August 1973.

[33] R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation," *Journal of the ACM*, vol. 27, pp. 831–838, October 1980.

[34] N. Abramson, "The ALOHA System–Another Alternative for Computer Communications," in *Proceedings of the Fall Joint Computer Conference*. AFIPS Press, 1970, pp. 281–285. [Online]. Available: http://en.wikipedia.org/wiki/ALOHAnet

[35] D. R. Kowalski and A. Pelc, "Leader Election in Ad Hoc Radio Networks: A Keen Ear Helps," *Journal of Computer and System Sciences*, vol. 79, pp. 1164–1180, November 2013.

[36] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Waltham, MA, USA: Morgan Kaufman Publishers, 2011.

[37] R. Vaidyanathan, P. Vinukonda, and A. C. Lessing, "Pipelined Execution of Windowed Image Computations," *Journal of Networking and Computing*, vol. 3, pp. 75–97, January 2013.