Contents

1 Crash Course on Digital Logic Design 1
1.1 Overview 1
1.2 Digital Operation and Building Blocks 2
1.2.1 Building blocks 2
1.3 Graphs 4
1.4 Combinational Circuits 5
1.4.1 Equivalence of Combinational Circuits and Boolean Circuits 6
1.4.2 Modularization 6
1.5 Synchronous/Clocked Circuits and Finite State Machines 9
1.5.1 Syntactic Definition 9
1.5.2 Finite state machines 10
1.5.3 Equivalence of FSMs and Synchronous Circuits 10
1.5.4 Addition 12
1.6 Complexity Measures: Cost and Delay 15
1.7 Lower Bounds 15
1.8 Conclusion 16
1.9 Further Reading 17
Bibliography 19

1 Crash Course on Digital Logic Design

Chapter Contents

1.1 Overview 1
1.2 Digital Operation and Building Blocks 2
1.3 Graphs 4
1.4 Combinational Circuits 5
1.5 Synchronous/Clocked Circuits and Finite State Machines 9
1.6 Complexity Measures: Cost and Delay 15
1.7 Lower Bounds 15
1.8 Conclusion 16
1.9 Further Reading 17

1.1 Overview

This chapter is aimed for grad students which have seen logic design, either from the EE perspective or the CS one. The goal of this chapter is to "synchronize" the logic-design language that is going to be used throughout this book, so that both EE and CS can speak the same language!

Naturally, a full logic-design course requires much more than a single chapter. Hence, in this chapter we focus more on functionality, while overlooking some more elaborate models and presenting simplified and idealized circuits. This emphasis on simplicity allows for a quick yet concise hands-on on logic design. We will go all the way from simple gates to complex circuits that can implement finite state machines. In that journey we will see how to define a circuit that can add and how to implement it. This adder will be slow - we will get back to designing a better on later on in the book. We will also set some abstract complexity measures, and argue regarding the best possible circuit for a given task. Some of the detailed will be left open for you to close at the inline exercises and in the exercises at the end of this chapter.

This chapter is based on a Monograph [1] by Guy Even on how to teach hardware (more specifically on how to teach fast adder designs), and a logic design Book [2].

1.2 Digital Operation and Building Blocks

In this chapter, we assume that inputs and outputs of devices (gates, circuits, memory components, etc) are always either zero or one, i.e., a bit. As we will see in Chapter 3, this assumption is unrealistic due to the fact that the digital value is obtained by rounding an analog value that changes continuously (e.g., voltage). There is a gap between analog values that are rounded to zero and analog values that are rounded to one. When the analog value is in this gap, its digital value is neither zero or one.

The advantage of this assumption is that it simplifies the task of designing digital hardware. We do need to take precautions to make sure that this unrealistic assumption will not render our designs useless. We set strict design rules for designing circuits to guarantee well defined functionality.

We often use the term *signal*. A signal is simply zero or one value that is output by a gate, input to a gate, or delivered by a wire.

1.2.1 Building blocks

The first issue we need to address is: "what are our building blocks?" The building blocks are:

- combinational gates,
- flip-flops,
- and wires.

We briefly describe these objects.

Combinational gates. A *combinational gate* (or gate, in short) is a device that implements a Boolean function. What does this mean? Consider a Boolean function $f : \{0,1\}^k \rightarrow \{0,1\}^\ell$. Now consider a device G with k inputs and ℓ outputs. We say that G *implements* f if the outputs of G equal $f(\alpha) \in \{0,1\}^\ell$ when the input equals $\alpha \in \{0,1\}^k$. Of course, the evaluation of $f(\alpha)$ requires time and cannot occur instantaneously. This is formalized by requiring that the inputs of G remain stable with the value α for at least d units of time. After d units of time elapse, the output of G stabilizes on $f(\alpha)$. The amount of time d that is required for the output of G to stabilize on the correct value (assuming that the inputs are stable during this period) is called the *propagation delay* of a gate.

Typical gates are inverters (i.e., a NOT gate) and gates that compute the Boolean OR, AND, or XOR of two bits. The functionality of these Boolean functions are as follows: given two bits $x, y \in \{0, 1\}$

$$\begin{aligned} \text{NOT}(x) &= 1 - x, \\ \text{OR}(x, y) &= \max\{x, y\}, \\ \text{AND}(x, y) &= \min\{x, y\}, \text{ and} \\ \text{XOR}(x, y) &= x + y \mod 2 . \end{aligned}$$

It is not hard to verify that the OR, AND, and XOR are associative operators. We will see, in Chapter 21 that associativity is a very useful property.

E1.1 We say that an operator $\circ : A \times A \longrightarrow A$ is associative if $a \circ (b \circ c) = (a \circ b) \circ c$ for every $a, b, c, \in A$. Show that OR, AND, and XOR are indeed associative operators. List all the associative operators in $\{0, 1\}^2 \longrightarrow \{0, 1\}$.

We depict gates by boxes; the functionality is written in the box, or by a special symbol.

Flip-Flops. A flip-flop is a memory device. Here we use only a special type of flip-flops called *edge triggered D-flip-flops*. What does this mean? We assume that time is divided into intervals, each interval is called a *clock cycle*. Namely, the ith clock cycle is the interval $[t_i, t_{i+1})$. A flip-flop has one input (denoted by D), and one output (denoted by Q). The output Q during clock cycle $i + 1$ equals the value of the input D at time t_i (end of clock cycle i). We denote a flip-flop by FF.

This functionality is considered as memory because the input D is sampled at the end of clock cycle i. The sampled value is stored and output during the next clock cycle (i.e., clock cycle $i + 1$). Note that D may change during clock cycle $i + 1$, but the output Q must stay fixed.

We remark that three issues are ignored in this description: (i) Initialization. What does a flip-flop output during the first clock cycle (i.e., clock cycle zero)? We assume that it outputs a zero. (ii) Timing - we ignore timing issues such as setup-time and hold-time. (iii) The clock signal is missing. (The role of the clock signal is to mark the beginning of each clock cycle). We elaborate on these three points later in Chapter 2. In fact, every flip-flop has an additional input for a global signal called the clock signal. We assume that the clock signal is used only for feeding these special inputs, and that all the clock inputs of all the flip-flops are fed by the same clock signal. Hence, we may ignore the clock signal.

We will spend a lot of time on clocks in the rest of the book. In future chapters, we will have several clocks, so specifying which clock is feeding the flip-flops will be required, again, this is not the case in our current chapter.

Wires. The idea behind connecting a wire between two components is to take the output of one component and use it as input to another component. We refer to an input or an output of a component as a *port*. A wire can connect exactly one output port and at least one input port. We assume also that a wire can deliver only a single bit.

We will later see that there are strict rules regarding wires. To give an idea of these rules, note that it makes little sense to connect two outputs ports to each other. However, it does make sense to feed multiple input ports by the same output port. We, therefore, allow different wires to be connected to the same output port. However, we do not allow more than one wire to be connected to the same input port. The reason is that multiple wires feeding the same input port could cause an ambiguity in the definition of the input value.

The number of inputs ports that are fed by the same output port is called the *fanout* of that output port.

1.3 Graphs

We will shortly see that it is useful to consider the underlying graph of a circuit, as follows.

We model a circuit C using a directed graph $G(C)$. This graph is called the *netlist* of C. We assign a vertex for every gate and a directed edge for every wire. The orientation of the edge is from the output port to the input port (we will see in Section 1.4 that we allow wire connections between outputs to inputs only).

Given these terminology, the outdegree of a vertex v corresponds to the number input ports the corresponding gate feeds. An outdegree of a vertex is also referred to as the *fanout* (recall that our basic gates have only a single output port) of the corresponding gate. The *fanout of a circuit* is the maximum fanout over all the circuit's output ports. Similarly, the indegree of a vertex corresponds to the number of the corresponding gate's number of input ports, e.g., zero, one or two for our basic gates.

One can also consider introducing weights to this graph in order to capture a more elaborate model. For example, vertex weights can capture different gate delays, and edge weights can capture wire delays, etc.

1.4 Combinational Circuits

In this section we define an important class of circuits: combinational circuits.

One can build complex circuits from gates by connecting wires between gates. However, only a small subset of such circuits are *combinational*. To guarantee well defined functionality, strict design rules are defined regarding the connections between gates. Only circuits that abide these rules are called combinational circuits. We now describe these rules.

"Output-to-Input" Rule. The first rule says: *the starting point of every connection must be an output port and the end point must be an input port*. There is a problem with this rule; namely, how do we feed the external inputs to input ports? We encounter a similar problem with external outputs. Instead of setting exceptions for external inputs and outputs, perhaps the best way to solve this problem is by defining special gates for external inputs and outputs. We define an *input gate* as a gate with a single output port and no input port. An input gate simply models an external signal that is fed to the circuit. Similarly, we define an *output gate* as a gate with a single input port and no output port. Let us recap. The gates that we encountered so far are: NOT, OR, AND, XOR, input gate, and an output gate.

"Feel the (single) bit" Rule. The second rule says: *feed every input port exactly once*. This means that there must be exactly one connection to every input port of every gate. We do not allow input ports that are not fed by some signal (do not get confused with an input gate). The reason is that an unconnected input may violate the digital abstraction (namely, we cannot decide whether it feeds a zero or a one). We do not allow multiple connections to the same input port. The reason is that different connections to the same input port may deliver different values, and then the input value is not well defined. (Note that we do allow the same output port to be connected to multiple inputs and we also allow unconnected output ports.)

"No-Cycles" Rule. The third rule says: *a connection is forbidden if it closes a directed cycle*. The no-cycles rule simply does not allow directed cycles in the directed graph $G(C)$ (also referred to as a *Directed Acyclic Graph or DAG*).

Note that combinational circuits do not include flip-flops as one of its building blocks. Hence, there is no clock signal that "times" the computation the circuit performs. We will elaborate shortly on how to evaluate the output of a circuit, given its inputs.

E1.2 Recall the design rules above. Come up with a circuit which is combinational and one that is not. Draw these circuits, and map these circuit drawings to directed graphs. How will you test in linear time (in the size of each graph) that these circuits are indeed combinational?

1.4.1 Equivalence of Combinational Circuits and Boolean Circuits

In this section we show that every combinational circuit implements some Boolean function. We also argue that every Boolean function has a combinational circuit (this will be left for you as an exercise).

The definition we use for combinational circuits is syntactic; namely, we only require that the connections between the components follow some simple rules. Our focus was syntax and we did not say anything about functionality. The syntactic definition of combinational circuits has two main advantages: (i) It is easy to check if a circuit is indeed combinational. (ii) The functionality of every combinational circuit is well defined. The task of determining the output values given the input values is referred to as *logical simulation*; it can be performed as follows. Given the input values of the circuit, one can scan the circuit starting from the inputs and determine all the values of the inputs and outputs of gates. When this scan ends, the output values of the circuit are known. The order in which the gates should be scanned is called *topological order*, and this order can be computed in linear time [3, Sec 1.6 & 6.5]. It follows that combinational circuits implement Boolean functions just as gates do. The difference is that functions implemented by combinational circuits are bigger (i.e., have more inputs/outputs).

E1.3 We now saw how to evaluate the output of a combinational circuit, given stable and logic values at its inputs. Assume that each of the gates has delay of a single time unit, i.e., the time it takes for a gate to output a signal which equals to the corresponding Boolean function applied to its inputs. Extend the above logical simulation to also compute the time required for a given circuit to output a logical signal which corresponds to the circuit's Boolean function applied to the circuit's inputs.

1.4.2 Modularization

Since every combinational circuit implements some Boolean function, it is useful to treat a circuit as a "big" gate, e.g., when designing a large circuit. In order to have a clean modularization we need to attach to this "new" gate the same parameters and interfaces a basic gate has, as follows. The cost and the delay of this new gate is, naturally, the cost and the delay of the corresponding circuit (note that now the large circuit has (complex) gates with non uniform delay and cost as before). The input and output ports of this new gates correspond to the input and output gates of the circuit, respectively.

We now introduce two useful (combinational) modules: (1) a *Multiplexer*, and (2) a *Full-Adder*.

1.4.2.1 Full-Adder. One of the basic arithmetic operations that we need for our computer to work is an adder: a combinational circuit that can add two "numbers". How do numbers look in our world of logic design? there are several conventions for encoding (or representing) natural (and even rational) numbers. In this chapter we introduce *the Binary Representation* which is formally defined as follows.

Definition 1.1 (Binary Representation)**.** *Let $x \in \{0,1\}^n$. The natural number that is represented by x in the Binary Representation equals to*

$$\sum_{i=0}^{n-1} x[i] \cdot 2^i \, .$$

We denote the one-to-one mapping between binary strings in $\{0,1\}^n$ to $\{0,\ldots,2^n-1\}$ by $\langle\cdot\rangle_n$. When n is clear from the context we remove the subscript from the $\langle\cdot\rangle_n$ operator. The bit $x[0]$ has weight 1 in the binary representation, and is referred to as the least significant bit. *The bit $x[n-1]$ has weight 2^{n-1} in the representation and is referred to as the* most significant bit.

Example 1.2. $\langle 110\rangle_3 = 0\cdot 2^0 + 1\cdot 2^1 + 1\cdot 2^2 = 6$

E1.4 (1) Show that a natural number is even iff its least significant bit is 0, (2) show that "shifting" the binary representation of a given natural number a to the left, while inserting a 0 bit (which results with a binary string which is longer by one) represents the natural number $2a$, and (3) why is the $\langle\cdot\rangle_n$ mapping is one-to-one?

Throughout this book we will consider other kind of number representations (or encodings) such as Unary or Reflected-Gray Code in Chapter 19.

Before we add two binary strings, we will see soon that it is useful to deal with adding 3 bits first - very much like you did in the first grades in your school, i.e., we learned how to add multiple digits by first adding two (or three). The Boolean function that corresponds to adding 3 bits is referred to as *Full-Adder*. The combinational circuit that implements it is, conveniently, referred to as a FADDER. The specification of the Full-Adder is as follows.

Input: $\{x, y, z\} \in \{0,1\}$,
Output: $\{S, C\} \in \{0,1\}$,
Functionality: $x + y + z = 2\cdot C + S$.

Example 1.3. • *Let $x = y = z = 1$, then the* FADDER *outputs $S = C = 1$, since $3 = 2\cdot 1 + 1$. Recall that $\langle 11\rangle = 3$.*

- *Let* $x = y = 1, z = 0$*, then the* FADDER *outputs* $C = 1, S = 0$*, since* $2 = 2 \cdot 1 + 0$.
- *Let* $x = 1, y = z = 0$*, then the* FADDER *outputs* $C = 0, S = 1$*, since* $1 = 0 \cdot 1 + 1$.

How does the combinational circuit that implement a Full-Adder look like? We leave this as an exercise for the reader (see Exercise 2). What we can say, even without implementing an FADDER is that since there are 3 inputs and 2 outputs, then every reasonable implementation of this function has a "constant" number of gates and a "constant" depth.

E1.5 Make sure that you have a concrete argument to why there are implementation of constant cost and delay. Draw the circuit and its corresponding DAG.

To conclude: we have just created our first meaningful module - a Full-Adder. From now on, you can treat it as a (complex) gate with 3 inputs and 2 outputs, and with a constant delay and a constant cost.

1.4.2.2 A Multiplexer. Another basic operation that is widely used in logic design is the ability to select a bit out of a string of input bits, and to output this bit. Again, before confronting this problem (actually, you will solve it yourself on Exercise 4) we will deal with a more basic problem: how to choose one bit our of two input bits? This Boolean function is called a *Multiplexer* and its corresponding combinational circuit is denoted by MUX. The specification of a MUX is as follows.

Input: $\{a, b, s\} \in \{0, 1\}$,
Output: $\{o\} \in \{0, 1\}$,
Functionality: $\begin{cases} a, & \text{if } s = 0\ , \\ b, & \text{o.w. }\ . \end{cases}$

Example 1.4. • *Let* $a = b = s = 1$*, then the* MUX *outputs* 1*. Actually, it does not matter what is the value of s here, the* MUX *should always output 1 in this case.*
- *Let* $a = s = 1, b = 0$*, then the* MUX *outputs 0.*

As in the Full-Adder implementation, the number of inputs and outputs of this Boolean function are constant, hence, any reasonable implementation of it has constant depth and cost. One implementation of a MUX is as follows: Connect the input gate that feeds the a input to an AND gate where the second input port of that gate is connected to the negation of the input gate feeding s. Similarity, connect the input gate that feeds the b input to another AND gate where the second input port of that gate is connected to input gate feeding s.

Now, connect the two output ports of the two AND gates to the input ports of an OR gate, the output of which is the output of our MUX.

E1.6 Convince yourself (by a short proof) that the above MUX design is correct. Draw the MUX circuit and its corresponding DAG.

You will deepen your knowledge on multiplexers in the Exercises section where you will consider a generalized definition of a multiplexer, and realize that it is a "powerful" module.

1.5 Synchronous/Clocked Circuits and Finite State Machines

In this section we discuss another family of circuits: synchronous circuits. The main difference between combinational circuit and synchronous ones is that the latter has flip-flops as a new building block (which also means that synchronous circuits are clocked!). We will, again, start with a syntactic definition of synchronous circuits. We will then show that synchronous circuits are equivalent to finite state machines, hence settling the functionality of synchronous circuits.

1.5.1 Syntactic Definition

Synchronous circuits are built from gates and flip-flops. As in the case of combinational circuits, the definition of synchronous circuits is syntactic.

Consider a circuit C that is simply a set of gates and flip-flops connected by wires. We assume that the first two rules of combinational circuit are satisfied: Namely, wires connect outputs ports to inputs ports and every input port is fed exactly once.

We produce a circuit C' from C by removing its flip-flops, as follows. For every flip-flop in C, we remove the flip-flop and add an output-gate instead of the input D of the flip-flop. Similarly, we add an input-gate instead of the output Q of the flip-flop.

Now, we say that C is a synchronous circuit if C' is a combinational circuit.

E1.7 Prove that every directed cycle in a synchronous circuit contains at least one flip-flop.

What about the functionality of a synchronous circuit? The functionality of a synchronous circuit can be modeled by a finite state machine, defined below. We show that FSMs and Synchronous Circuits are equivalent (in the sense that one can be simulated by the other), and hence conclude the synchronous functionality discussion.

1.5.2 Finite state machines

A finite state machine (also known as a finite automaton with outputs or a transducer) is an abstract machine that transcribes an input to an output string. We follow the definition of a finite state machine given by Mealy [4], also known as a *Mealy Machine*.

Definition 1.5. *[Mealy Machine] A Mealy machine is a 6-tuple $T = (S, s_o, \Sigma, \Lambda, t, o)$, where*

- *S is a finite set of states,*
- *$s_0 \in S$ is a starting state,*
- *Σ is the input alphabet,*
- *Λ is the output alphabet,*
- *$t : S \times \Sigma \to S$ is the state transition function, and*
- *$o : S \times \Sigma \to \Lambda$ is the output function.*

The machine T operates in steps. In each step of the machine, an input symbol is fed to the machine, a transition to a next state occurs, and a symbol is output. Formally, in the ith step the input symbol $\sigma_i \in \Sigma$ is fed to T, where $i \in \mathbb{N}$. The state $s_i \in S$ after i such steps is inductively defined as $s_i \triangleq t(s_{i-1}, \sigma_i)$, where s_0 is the initial state. The output $o_i \in \Lambda$ after i such steps is inductively defined as $o_i \triangleq o(s_{i-1}, \sigma_i)$.

1.5.3 Equivalence of FSMs and Synchronous Circuits

In this section we show how to simulate FSMs by a Synchronous circuits and vice a versa.

Synchronous Circuit to FSM (aka Analysis). Let C denote a synchronous circuit, and let C' denote the combinational circuit obtained by removing the flip-flops in C (see Section 1.5.1).

Let k denote the number of flip-flops in C. Every output gate in C is either feeding a flip-flop or feeding an output gate. Let OUT denote the set of output gates in C' that correspond to an output gate of C, and let $m \triangleq |OUT|$. Note, that the number of output gates in C' that correspond to an input to a flip-flop is k (why?) - we denote this set of output gates by NS. Let IN denote the set of input gates in C, and $n \triangleq |NS|$. Again, note that the number of input gates in C that are *not* input gates in C is exactly k (why?) - we denote this set of inputs by CS. Let $FF = \{F_1, \dots, F_k\}$ denote the set of flip-flops of C, and let $F_\ell(i)$ denote the output of the ℓth flip-flop at the ith clock cycle. Given this notation, you should have in your mind now, a mental picture of the synchronous circuit which looks like an “island” of combinational circuits that are fed by flip-flops via the CS (for “current state”) inputs and from the outer-world inputs IN. This combinational island also feeds the flip-flops via the NS (for "next state") output gates, and feeds the outer-world with m output gates OUT. Your intuition is probably telling you that the flip-flops are storing the state of the soon to come equivalent FSM.

Up to here, we gave some notation that describe the circuit C. Now, we start "building" the equivalent FSM. We abuse notation and refer to the signals that are fed by input gates, and to signals that output ports feed by their names, e.g., NS is both the set of input gates, and the set of logical values fed by them. Recall that for defining an FSM we need to specify all 6 objects in its 6-tuple (see Definition 1.5), as follows:

- $S = \{0, 1\}^k$,
- $s_0 = 0^k$ is set to initialization of the flip-flops. Recall that we assume in this chapter that flip-flops are initialized to zero.
- $\Sigma = \{0, 1\}^n$,
- $\Lambda = \{0, 1\}^m$,
- $t(CS, IN) = NS$, and
- $o(CS, IN) = OUT$.

E1.8 Is this FSM well defined? convince yourself that the functions t and o matches the definition of an FSM according to Definition 1.5.

Is that all? of course not. We need to show that the obtained FSM simulates the synchronous circuit C, that is, we will show that for every clock cycle i the output of the synchronous circuit $C(i)$ equals to o_i. The proof of the following by induction. The idea is that the FSM follows the same state sequence as the sequence of values stores in C's flip-flops, and since the functions o and t compute the Boolean function that C' computes.

Theorem 1.6. *The obtained FSM simulates the synchronous circuit C. Moreover, for every $i \in \mathbb{N}$ it holds that $CS_i = s_{i-1}$.*

In Exercise 6 you will show how to simulate an FSM by a synchronous circuit (aka Synthesis). This is done in a similar way to the simulation of the other direction shown above.

We conclude that FSMs and synchronous circuits are equivalent, that is, every synchronous circuit implements an FSM, and every FSM can be implemented by a synchronous circuit.

Feasibility and Minimum Clock-cycle in our Simplified Model One thing which is left open in Exercise 6 is at what clock frequency does the obtained circuit actually simulates the FSM? In fact, not all clock frequencies enable a correct operation of the synchronous circuit. This topic will be discussed in more depth in the next chapter. Briefly, let C and C' denote, as before, a synchronous circuit and its corresponding set of combinational circuits. Since every flip-flop in C is fed by a combinational circuit in C', and feeds another one (see Inline Exercise 1.5.1) the clock cycle should be long enough to allow for the feeding combinational circuit to "finish" its computation before the sampling of the corresponding flip-flop. In a more elaborate model, one should also consider the delay of the flip-flop itself, e.g., since it feeds some combinational circuit. Hence, a feasible clock cycle should be at least as long as the maximum combinational delay of combinational circuits in C'. Since we are interested in fast circuits, we will take the minimum feasible clock cycle. This boils down to the following simple task: (1) compute C', (2) compute the longest path in the corresponding graph, (3) set the minimum clock period to be this length.

What is simpler in our model? A short answer to this questions is: flip-flops. The longer one is, as follows. Every flip-flop comes with a set of four parameter: setup-time t_{su}, holds-time t_{hold}, propagation delay t_{pd}, and contamination delay t_{cont}. When we defined flip-flops we required that the output Q during clock cycle $i + 1$ equals the value of the input D at time t_i. This is equivalent to assuming that the setup-time equals the clock period, i.e., $t_{i+1} - t_i$, and that $t_{\text{hold}} = t_{\text{cont}} = t_{\text{pd}} = 0$. [1]

In real life, one has to take into considerations these four parameters without our assumption on the behaviour of a flip-flop, and with wire delays, non-uniform gate delays, etc. More about this, including a more elaborate discussion on how to compute the minimum clock period (while taking into account all of flip-flop's parameters) that ensures the correct functionality of the FSM is discussed in chapter 2 (and in Sec. 19.6 on [2]). Moreover, in this chapter we dealt with a single FSM and its transformation to a synchronous circuit. What happens if there are several such FSMs that communicate with each other - how do we translate them to a circuit where each FSM is transformed into a different circuit while also implementing some communication mechanism between them? this discussion appear on Chapter 2

1.5.4 Addition

In Section 1.4.2.1 we defined a FADDER, as a first step towards "really" adding two numbers represented in the Binary representation. We define an adder, we

[1] There is even a more extreme model where the clock-period is assumed to be 1. This model is called the "Zero Delay Model" [2, Sec. 18.1]. The Zero Delay Model is only used for specifying and simulating the functionality of circuits with flip-flops.

then define a "timed" version of it - a Serial Adder. We will implement this serial adder by a small synchronous circuit, and finally will show how to transform this synch. circuit to a combinational adder which is referred to by Ripple carry adder. Other than introducing binary addition, this chapter serves as an elaborate example for the reader of several concepts we have seen in the chapter.

Notation. This is a good place to introduce vector notation for signals in our circuits. Given a binary string A of length n we denote it (also) by $A[n-1:0]$ emphasizing that the indices of the bits going from right to left. This emphasis on indexing is somewhat important when dealing with binary strings that represent numbers, i.e., the least significant bit is indexed by 0 and the most significant bit is indexed by $n-1$. We denote the ith bit of A by $A[i]$. We denote the string coarctation operator by simply writing them one next to the other, e.g., C[2n-1:0] = A[n-1:0]B[n-1:0] is a string of length $2n$ where the bottom n bits are B's and the upper ones are of A. In our synchronous circuits our notations need to capture time when referring to a certain signal in the circuit. We denote the value of a (stable) signal A in the tth clock cycle by $A(t)$. Similarly to the vector notation before we write $A(t:0)$ to denote the binary string $A(t)A(t-1)\cdots A(0)$. Again, the order of the bits in the string is defined by the notation: the signal is going "back in time" as you go right in the string $A(t:0)$.

We are now ready to define a *combinational adder*, as follows.

Definition 1.7. *A* combinational binary adder *is a combinational circuit defined as follows.*

Input: $A[n-1:0], B[n-1] \in \{0,1\}^n$

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

Functionality: $\langle A[n-1:0]\rangle + \langle B[n-1:0]\rangle = \langle CS[n-1:0]\rangle$.

E1.9 Can all pairs of numbers that their binary representation is of length n be added this way to produce the output S and C?

1.5.4.1 Serial Addition. Before jumping into the deep waters of designing a combinational adder let us consider the following version of serial addition. In this version, the bits of inputs A and B are "coming" in pairs, one pair in each clock cycle. Very much like calculating the sum of to numbers "by hand": we inspect two digits at a time, sum them and some times we need to carry a 1 to the next pair of digits and sometimes we need to sum three digits (where one of the was carried to us from the previous "cycle").

Although the functionality of serial addition (we will soon see in its definition) is defined w.r.t. the entire "history" of summation, but we will show a synchronous circuit that only need to remember s *single* bit.

The definition is as follows.

Definition 1.8. *A* serial adder *is a synchronous circuit defined as follows.*

Input: $a, b \in \{0, 1\}$
Output: $S \in \{0, 1\}$.
Functionality: $\langle a(0:t)\rangle + \langle b(0:t)\rangle = \langle S(0:t)\rangle \mod 2^{t+1}$.

The synchronous circuit that implements this specification is quite simple. We will need a FADDER and a flip-flop: Connect the two inputs a and b to the input ports x and y of the FADDER (see Section 1.4.2.1). Connect the S output port of the FADDER to the output gate s of the circuit. The C output port of the FADDER is connected to the D input port of the flip-flop, and the Q output port of the flip-flop is connected to the z input port of the FADDER.

E1.10 Make sure you know how to draw this circuit to yourself.

You will prove that this synchronous circuit implements the serial adder in Exercise 7. The cost of this serial adder is constant and the clock period is also constant (since all the paths in the corresponding DAG traverse through the FADDER which has a constant depth).

1.5.4.2 Ripple-Carry Adder. Now, we are ready to play a little "Thought experiment" that its result will be a combinational adder.

Let us assume the stream of bits that are input to the serial adder are of length n. For each clock cycle we have a copy of our serial adder where we removed its flip-flops (i.e., a FADDER). Now, the C output of the FADDER copy of the first clock cycle is input to the z input of second copy of the FADDER (which corresponds to the second clock cycle), and so on. In this circuit, we do not have any flip-flops and no directed cycles - it is a combinational circuit! This combinational circuit is usually called a ripple-carry adder.

E1.11 Convince yourself that this thought experiment yields a combinational circuit.

To conclude: in this section we saw what is a serial adder, and how to apply a "space-time" transformation (essentially our "Thought experiment") to obtain an adder - a ripple carry adder. This technique of mapping a synchronous circuit to a combinational one can be "glued" with the ability to transform FSMs to a corresponding synchronous circuit. Naturally, the input is different in the combinational version and in the synch. one: in one the entire input is “ready” where for the latter the input arrives bit by bit. Using these two transformations we get that, given an FSM we obtain a linear cost and liner depth adder. In a future chapter we will see how to transform an FSM to an optimal combinational circuit!

1.6 Complexity Measures: Cost and Delay

We are interested in cheap and fast circuits. Hence, design (correct) circuits while minimizing the number of gates, and also minimizing their depth. The *depth* of a combinational circuit is the longest path from an input gate to an output gate. The *cost* of a combinational circuit is simply the number of gates that are used in the circuit. Naturally, in a more elaborate model, one would measure the cost as the costs of its gates, e.g., different gates are implemented with different number of transistors. Similarly, instead of depth, one would measure the delay (or “weighted” depth), e.g., each gate has different propagation delay due its implementation. In this case one would need to compute the heaviest path instead of the longest one, i.e., the difference between weighted to unweighed graphs (see Section 1.3).

For synchronous circuit, the cost is defined analogously (while some count the flip-flop severalty). Instead of minimizing the delay, we would like to minimize the clock cycle length - more on that on Chapter 2. Modularizing synchronous circuits is also useful (very much like its parallel for combinational circuits). The discussion on how to specify the parameters of the new synchronous module requires a discussion on the more elaborate set of flip-flop parameters, and the more accurate calculation of a minimum feasible clock - all of this is also on Chapter 2.

1.7 Lower Bounds

We have seen by now that every Boolean function can be implemented by a combinational circuit. Each such circuit has its cost and depth. What is the best cost and depth possible for a given Boolean function? Lower bounds are always a “sneaky thing” - how do you prove something on an object that you do not “hold in your hand”? for example, we have an implementation of a combinational adder. We can say many things on that adder, but what can we say on *any* circuit that implements the combinational adder Boolean function?

In this section we give an idea on how to extract structural properties of any circuit that implements a given function. These properties will allow as us to argue what is the best cost and best depth possible for that function. Note, that we can say that we have an optimal circuit if the claimed lower bounds meet the bounds of the designed circuit. If this is not the case, then either we need to strengthen the lower bound for that specific Boolean function or we need to dig deeper and improve our design (or both).

As before, we assume that our gates have a constant indegree (actually it is 1 or 2 in this chapter), and we disregard the effect of high fanout of a gate. We also assume that all the gates have a uniform cost, i.e., one.

Here we will give some intuition of why the following lower bounds are true. The rigourous and complete proofs require some more arguments, e.g., as in [2, Chap. 12].

Depth Lower Bound. We begin with a structural property that every combinational circuit has. Let us fix one of the circuit's outputs. now, consider the number of inputs n that are connected by a directed paths to that input. What is the most "compact" graph topology that you know of that can span these n inputs? A balanced binary tree! Hence, the circuit at hand has a depth of $\Omega(logn)$ w.r.t. the fixed output.

Now, we look on a Boolean function f and define a notion of a dependency of an output on a (subset) of its inputs. We say that an output bit of f, y_j depends on a specific input bit x_i if there is an input string to f such that if this input bit x_i is flipped then the output of fm y_j changes as well.

For simplicity, let us assume that f has a single output. Any circuit that implements f has to have a path from the output to the inputs that the output depends on (this requires a proof).

When combining these two arguments we get that any circuit that implements a Boolean function that depends on n inputs has a depth of $\Omega(logn)$.

Cost Lower Bound. Similar arguments to the one made in the depth lower bound, can be made here: a balanced tree with n leaves has $\Omega(n)$ vertices. Hence, every combinational circuit that implements a given Boolean function as cost of $\Omega(n)$.

1.8 Conclusion

In this chapter we have seen how to construct complex clocked circuits from simple gates, flip-flops and wires. These circuits, which are called synchronous circuits, are powerful enough to implement an FSM. Our lap-top's processor (or an abstract form of it) can be represented as such an FSM. We overlooked all kinds of more "physical" parameters (or considerations) so that the discussion on the functionality of such circuits will be consider. Finally, our circuits live in a single clock domain - they are all clocked by the same clock signal. We have not dealt with how to distribute this signal, and have not discussed how a large circuit that is fed by multiple clock signal work. More about the overlooked parameters and clocks in the rest of the book!

1.9 Further Reading

For more elaborate discussion, the curious reader might want to Parts II (Combinational Circuits) and III (Synchronous Circuits) and the many exercises within of [2].

Problems and Assignments

1. **From a Boolean function to a Circuit:** In Section 1.4.1 we showed that every combinational circuit implements some Boolean function. Prove that the other direction is also true: for every Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}^m$ there is a combinational circuit C such that for all $x \in \{0,1\}^n$ it holds that $C(x) = f(x)$. What is the propagation delay of this circuit? Can you argue that it is optimal w.r.t. its depth or cost? How many kinds of gates have you used? Can you use less? in case you can, improve your construction, o.w., please explain why this is minimum set of gates needed.
2. Design a Full-Adder (FADDER). Prove that your design is correct.
3. Revisit your answer to Exercise 1. Can you replace all your gates with a MUX and the constant 0 and 1? Give a short argument that justified your answer.
4. **$(n : 1)$ – MUX**: In Section 1.4.2.2 we have seen the definition of the Multiplexer Boolean function, and how to implement it with a combinational circuit. In this exercise we consider an extended definition of a Multiplexer, an n to 1 multiplexer. In this generalized definition there are two kinds of inputs: *select* and *data*. The select bits represent an index of a bit from the data input that is the output of the extended multiplexer. We denote the corresponding combinational circuit as a $(n : 1) - \text{MUX}$.
   - Implement such a $(n : 1) - \text{MUX}$ circuit.
   - Consider the underlying graph of your circuit. Is there a special structure to it? what is it?
   - What is the cost and delay of your circuit? Can you argue that you design is optimal? explain why.
5. Prove Theorem 1.6.
6. **From an FSM to a Circuit**: In this question we will show that one can simulate an FSM by a Synchronous circuit. That is given a Mealy machine $T = (S, s_o, \Sigma, \Lambda, t, o)$ design a synchronous circuit C, such that for every step $i \in \mathbb{N}$, it holds that $C(i) = o(i)$. For simplicity, assume

that $\Sigma = \{0,1\}^k$, and $\Lambda = \{0,1\}^\ell$. Also assume that we operate in the Zero delay model, and that all gates have delay of 1 time unit. Guidance: "Store" the state of the FSM and update it according to t. Draw a canonical design and prove that is indeed simulating the FSM T.

7. Recall the definition of a serial adder (see Definition 1.8). Prove by induction on the clock-cycle t that the suggested implementation in Section 1.5.4.1 is correct.
8. Consider the suggested synchronous circuit of a serial adder in Section 1.5.4.1. Specify the corresponding FSM by applying transformation in Section 1.5.3 aka analysis. Synthesize the FSM that you obtained and verify that you get the serial-adder that you started with.
9. Recall the ripple-carry adder design in Section 1.5.4.2. Prove that this adder is correct, that is, that it implements an adder (see Definition 1.7. Prove that the cost and depth of a ripple-carry adder are both linear in n.
10. **Clock-enabled Flip-Flop:** Design a new flip-flop with the following new property. There is a new input signal ce. If $ce(t) = 1$ in the tth clock cycle then the new flip-flop behaves as before, and if $ce(t) = 0$, then the flip-flop does not "sample" the input and retains the bit that is already stored. Suggest a design that *does not* mask the clock signal.
11. Fill the missing detailed for the lower bounds proofs in Section 1.7.
12. Rewrite the functionality of a flip-flop under the Zero Delay Model.

Bibliography

[1] Even, Guy. 2006. On teaching fast adder designs: Revisiting Ladner & Fischer. In *Theoretical computer science*, 313–347. Springer.

[2] Even, Guy, and Moti Medina. 2012. *Digital logic design: a rigorous approach*. Cambridge University Press. http://hyde.eng.tau.ac.il/Even-Medina/index.html.

[3] Even, Shimon. 2011. *Graph algorithms*. Cambridge University Press.

[4] Mealy, George H. 1955. A method for synthesizing sequential circuits. *The Bell System Technical Journal* 34 (5): 1045–1079.