Asynchronous
Circuit Design
Asynchronous Circuit Design

Chris J. Myers
Contents

1 Introduction ........................................... 1
  1.1 Problem Specification .......................... 1
  1.2 Communication Channels ......................... 2
  1.3 Communication Protocols ....................... 4
  1.4 Graphical Representations ...................... 7
  1.5 Delay Insensitive Circuits ...................... 10
  1.6 Huffman Style Synthesis ......................... 13
  1.7 Muller Style Synthesis ........................ 15
  1.8 Timing Analysis and Optimization .............. 16
  1.9 Performance Analysis and Verification ......... 19
  1.10 Sources ........................................ 20
      Problems ....................................... 20

References .............................................. 23

Index .................................................. 25
1

Introduction

Wine is bottled poetry. —Robert Louis Stevenson

Wine gives courage and makes men more apt for passion. —Ovid

I made wine out of raisins so I wouldn’t have to wait for it to age. —Steven Wright

This chapter uses a simple example to give an informal introduction to many of the concepts and design methods that will be covered in this book. Each of the topics in this chapter will be more formally addressed in much more detail in subsequent chapters.

1.1 PROBLEM SPECIFICATION

In a small town in Southern Utah, there’s a little winery with a wine shop nearby. Being a small town in a community who thinks prohibition still exists, there is only one wine patron. The wine shop has a single small shelf capable of holding only a single bottle of wine. Each hour, on the hour, the shopkeeper receives a freshly made bottle of wine from the winery which he places on the shelf. At half past each hour, the patron arrives to purchase the wine, making space for the next bottle of wine. Now, the patron has learned that it is very important to be right on time. When he has arrived early, he has found
an empty shelf, making him quite irate. When he has arrived late, he has
found that the shopkeeper drank the last bottle of wine to make room
for the new bottle. The most frustrating experience was when he arrived at just
the same time that the shopkeeper was placing the bottle on the shelf. In
his excitement, he and the shopkeeper collided sending the wine bottle to the
floor shattering so no one got to partake of that lovely bottle of wine.

This synchronous method of wine shopping went on for some time with
all parties being quite happy. Then one day (in the mid 1980s), telephone
service arrived in this town. This was a glorious invention which got the town
really excited. The patron got a wonderful idea. He knew the winery could
operate faster, if only he had a way to purchase the wine faster. Therefore,
he suggested to the shopkeeper, “Say, why don’t you give me a call when the
wine arrives?” This way he could avoid showing up too early, frustrating his
fragile temperament. Shortly after the next hour, he received a call to pick
up the wine. He was so excited that he ran over to the store. On his way
out, he suggested to the shopkeeper, “Say, why don’t you give the folks at
the winery a call to tell them you have room for another bottle of wine?” This is
exactly what the shopkeeper did; and wouldn’t you know it, the wine patron
got another call just 10 minutes later that a new bottle had arrived. This
continued throughout the hour. Sometimes it would take 10 minutes to get
a call while other times it would take as long as 20 minutes. There was even
one time he got a call just 5 minutes after leaving the shop (fortunately, he
lived very close by). At the end of the hour, he realized that he had drunk 5
bottles of wine in just one hour!

At this point, he was feeling a bit woozy, so he decided to take a little
snooze. An hour later, he woke up suddenly quite upset. He realized the
phone had been ringing off the hook. “Oh my gosh, I forgot about the wine!”
He rushed over expecting to find that the shopkeeper had drunk several bottles
of his wine, but to his dismay, he saw one bottle on the shelf with no empties
lying around. He asked the shopkeeper, “Why did they stop delivering wine?”
The shopkeeper said, “Well, when I did not call, they decided that they had
better hold up delivery until I had a place on my shelf.”

From that day forward, this asynchronous method of wine shopping became
the accepted means of doing business. The winery was happy as they sold
more wine (on average). The shopkeeper’s wife was happy as the shopkeeper
never had to drink the excess. The patron was extremely happy as he could
now get wine faster, and whenever he felt a bit overcome from his drinking
indiscretion, he could rest easy knowing that he would not miss a single bottle
of wine.

1.2 COMMUNICATION CHANNELS

One day a VLSI engineer stopped by this small town’s wine shop, and he got
to talking with the shopkeeper about his little business. Business was good,
but his wife kept bugging him to take that vacation to Maui he had been promising for years and years. He really did not know what to do as he did not trust anyone to run his shop for him while he was away. Also, he was a little afraid that if he wasn’t a little careful that the winery and patron may realize that they really did not need him and could deal with each other directly. He really could not afford that.

The VLSI engineer listened to him attentively, and when he was done announced, “I can solve all your problems. Let me design you a circuit!” At first, the shopkeeper was quite skeptical when he learned that this circuit would be powered with electricity (a new magical force that the locals had not completely accepted yet). The engineer announced, “It is really quite simple actually.” The engineer scribbled a little picture on his napkin (see Figure 1.1). This picture shows two channels of communication which must be kept synchronized. One is between the winery and the shop, and another is between the shop and the patron. When the winery receives a request from the shop over its communication channel, it sends over a bottle of wine. This can be specified as follows:

\[
\text{Winery: process} \\
\text{begin} \\
\text{send(WineryShop, bottle);} \\
\text{end process;}
\]

The wine patron when requested by the shop over its communication channel comes to receive a bottle of wine which is specified as follows:

\[
\text{Patron: process} \\
\text{begin} \\
\text{receive(ShopPatron, bottle);} \\
\text{end process;}
\]

Now, what the shopkeeper does as the middle-man (besides mark-up the price) is provide a buffer for the wine to allow the winery to start preparing its next bottle of wine. This is specified as follows:

\[
\text{Shop: process} \\
\text{begin} \\
\text{receive(WineryShop, bottle);} \\
\text{send(ShopPatron, bottle);} \\
\text{end process;}
\]

These three things together form a specification. The first two processes describe the types of folks the shopkeeper deals with (i.e., his environment). The last process describes the behavior of the shop.
1.3 COMMUNICATION PROTOCOLS

After deriving a channel-level specification, it is then necessary to determine a communication protocol that implements the communication. For example, the shopkeeper calls the winery to ‘request’ a new bottle of wine. After some time, the new bottle arrives ‘acknowledging’ the request. Once the bottle has been safely shelved, the shopkeeper can call the patron to ‘request’ him to come purchase the wine. After some time, the patron arrives to purchase the wine which ‘acknowledges’ the request. This can be described as follows:

Shop: process
begin
    req_wine;   -- call winery
    ack_wine;   -- wine arrives
    req_patron; -- call patron
    ack_patron; -- patron buys wine
end process;

To build a VLSI circuit, it is necessary to assign signal wires to each of the four operations above. Two of the wires go to a device to place the appropriate phone call. These are called outputs. Another wire will come from a button that the wine delivery boy presses when he delivers the wine. Finally, the last wire comes from a button pressed by the patron. These two signals are inputs. Since this circuit is digital, these wires can only be in one of two states either '0' (a low voltage state) or '1' (a high voltage state). Let us assume that the actions above are signalled by the corresponding wire changing to '1'. This can be described as follows:

Shop: process
begin
    assign(req_wine,'1');   -- call winery
    guard(ack_wine,'1');   -- wine arrives
    assign(req_patron,'1'); -- call patron
    guard(ack_patron,'1'); -- patron buys wine
end process;

The function assign used above sets a signal to a value. The function guard waits until a signal attains a given value. There is a problem with the specification given above in that when the second bottle of wine comes req_wine will already be '1'. Therefore, we need to reset these signals before looping back.

Shop_2Phase: process
begin
    assign(req_wine,'1');   -- call winery
    guard(ack_wine,'1');   -- wine arrives
    assign(req_patron,'1'); -- call patron
    guard(ack_patron,'1'); -- patron buys wine
    assign(req_wine,'0');   -- call winery
    guard(ack_wine,'0');   -- wine arrives
assign(req_patron,'0'); -- call patron
guard(ack_patron,'0'); -- patron buys wine
end process;

When req_wine changes from '0' to '1', a phone call is placed, and when it changes again from '1' to '0', another call is placed. We call this transition signalling. It is also known as two-phase or two-cycle signalling for obvious reasons. Another alternative is given below:

Shop_4Phase: process
begin
assign(req_wine,'1'); -- call winery
guard(ack_wine,'1'); -- wine arrives
assign(req_wine,'0'); -- reset req_wine
guard(ack_wine,'0'); -- ack_wine resets
assign(req_patron,'1'); -- call patron
guard(ack_patron,'1'); -- patron buys wine
assign(req_patron,'0'); -- reset req_patron
guard(ack_patron,'0'); -- ack_patron resets
end process;

This protocol is called level signalling because a call is placed when the request signal is '1'. It is also called four-phase or four-cycle signalling. While this protocol may appear to be a little more complex in that it requires twice as many transitions of signal wires, it often leads to simpler circuitry.

There are still more options. In the original protocol, the shop makes the calls to the winery and the patron. In other words, the shop is the active participant in both communications. The winery and the patron are the passive participants. They simply wait to be told when to act. Another alternative would be for the winery to be the active participant and call you when a bottle of wine is ready as shown below:

Shop_PA: process
begin
guard(req_wine,'1'); -- winery calls
assign(ack_wine,'1'); -- wine is received
guard(req_wine,'0'); -- req_wine resets
assign(ack_wine,'0'); -- reset ack_wine
assign(req_patron,'1'); -- call patron
guard(ack_patron,'1'); -- patron buys wine
assign(req_patron,'0'); -- reset req_patron
guard(ack_patron,'0'); -- ack_patron resets
end process;

Similarly, the patron could be active as well and call when he has finished his last bottle of wine, and requires another. Of course in this case, the shopkeeper needs to install a second phone line.

Shop_PP: process
begin
guard(req_wine,'1'); -- winery calls
assign(ack_wine,'1');  -- wine is received
assign(ack_wine,'0');  -- reset ack_wine
guard(req_wine,'0');  -- req_wine resets
guard(req_wine,'1');  -- req_wine calls
assign(ack_wine,'0');  -- patron calls
assign(ack_wine,'0');  -- sells wine
assign(ack_wine,'0');  -- reset ack_wine

end process;

Unfortunately, none of these specifications can be transformed into a circuit as is. Let’s return to the initial 4-phase protocol (i.e., the one labelled Shop\(_4\)Phase). Initially, all the signal wires are set to '0', and the circuit is supposed to call the winery to request a bottle of wine. After the wine has arrived, and the signal req\(_\text{wine}\) and ack\(_\text{wine}\) have been reset, the state of the signal wires is again all '0'. The problem is that in this case the circuit must call the patron. In other words, when all signal wires are set to '0', the circuit is in a state of confusion. Should the winery or the patron be called at this point? We need to determine some way to clarify this. Considering again the initial 4-phase protocol, this can be accomplished by reshuffling the order in which these signal wires change. While it is important that the wine arrives before the patron is called, exactly when the handshaking wires reset is less important. Rearranging the protocol as shown below allows the circuit to always be able to tell what to do. Also, the eager patron gets a call sooner. On top of that, it results in the very simple circuit shown in Figure 1.2.

\(\text{Shop}_{\text{AA}}\text{\_reshuffled: process}\)

\begin{verbatim}
begin
  assign(req_wine,'1');  -- call winery
  guard(ack_wine,'1');  -- wine arrives
  assign(req_wine,'0');  -- call patron
  guard(ack_wine,'0');  -- wine arrives
  assign(req_wine,'0');  -- reset req_wine
  guard(ack_wine,'0');  -- reset ack_wine
  assign(req_wine,'0');  -- call patron
end process;
\end{verbatim}

\begin{figure}[h]
\centering
\begin{tikzpicture}
  \node (req_wine) at (0,0) {req_wine};
  \node (ack_wine) at (1,0) {ack_wine};
  \node (ack_patron) at (2,0) {ack_patron};
  \node (req_patron) at (3,0) {req_patron};
  \draw [->] (req_wine) -- (ack_wine);
  \draw [->] (ack_wine) -- (ack_patron);
  \draw [->] (ack_wine) -- (req_patron);
\end{tikzpicture}
\caption{Circuit for active/active shop.}
\end{figure}

Alternatively, we could have rescheduled the protocol in which the shop passively waits for the winery to call, but still actively calls the patron as shown below. The resulting circuit is shown in Figure 1.3.

\(\text{Shop}_{\text{PA}}\text{\_reshuffled: process}\)
begin
  guard(req_wine,'1'); -- winery calls
  assign(ack_wine,'1'); -- receives wine
  guard(ack_patron,'0'); -- ack_patron resets
  assign(req_patron,'1'); -- call patron
  guard(req_wine,'0'); -- req_wine resets
  assign(ack_wine,'0'); -- reset ack_wine
  guard(ack_patron,'1'); -- patron buys wine
  assign(req_patron,'0'); -- reset req_patron
end process;

req-wine

ack_wine

req_patron

ack_patron

Fig. 1.3 Circuit for passive/active shop.

The gate with a 'C' in the middle is called a 'Muller C-element'. When both of its inputs are '1', its output goes to '1'. Similarly, when both of its inputs are '0', its output goes to '0'. Otherwise, it retains its old value. Another curious thing about this protocol is that it waits for ack_patron to be '0', but it is '0' to begin with. Waits in which their expression is satisfied simply pass straight through. We call that first wait vacuous because it does nothing. However, the second time around, ack_patron may actually have not reset at that point. Postponing this wait until this point increases the concurrency and thus potentially the performance of the system.

1.4 GRAPHICAL REPRESENTATIONS

Before describing how these circuits are derived, let us first consider an alternative way of looking at these specifications using graphs. The first method is to use an Asynchronous Finite State Machine (AFSM). As an example, consider the active/active protocol from above (see Shop-AA_reshuffled), it can be represented as an AFSM as shown in Figure 1.4(a) or in a tabular form called a Huffman flow table shown in Figure 1.4(b). In the state ma-
machine, each node in the graph represents a state, and each arc represents a state transition. The state transition is labelled with the value of the inputs needed to make a state transition (these are the numbers to the left of the ‘/’). The numbers to the right of the ‘/’ represent what the outputs do during the state transition. Starting in state 0, if both \texttt{ack\_wine} and \texttt{ack\_patron} are '0', as is the case initially, the output \texttt{req\_wine} is set to '1' and the machine moves into state 1. In state 1, the machine waits until \texttt{ack\_wine} goes to '1', and then it sets \texttt{req\_patron} to '1' and moves to state 2. The same behavior is illustrated in the Huffman flow table in which the rows are the states and the columns are the input values. Each entry is labelled with the next state and next value of the outputs for a given state and input combination. When the next state equals the current state, it is circled to indicate that it is stable.

![AFSM and Huffman flow table](image)

Not all protocols can be described using an AFSM. The AFSM model assumes that inputs change followed by output and state changes in sequence. In the second design (see \texttt{Shop\_PA\_reshuffled}), however, inputs and outputs can change concurrently. For example, \texttt{req\_wine} may be set to '0' while \texttt{req\_patron} is being set to '1'. Instead, we can use a different graphical method called a Petri-net (PN), to illustrate the behavior of the second design as shown in Figure 1.5(a).

In a Petri-net, the nodes of the graph represent signal transitions. For example, \texttt{req\_wine+} indicates that \texttt{req\_wine} changes from '0' to '1'. Similarly, \texttt{req\_wine-} indicates that \texttt{req\_wine} changes from '1' to '0'. The arcs in this graph represent causal relationships between transitions. For example, the arc between \texttt{req\_wine+} and \texttt{ack\_wine+} indicates that \texttt{req\_wine} must be set to '1' before \texttt{ack\_wine} can be set to '1'. The little balls are called tokens, and a collection of tokens is called a \textit{marking} of the Petri-net. The initial marking is shown in Figure 1.5(a).
Fig. 1.5 PN for passive/active shop. (a) Initial marking. (b) after req\_wine goes to '1', and (c) after ack\_wine goes to '1'.

For a signal transition to occur, it must have tokens on all of its incoming arcs. Therefore, the only transition that may occur in the initial marking is req\_wine may be set to '1'. After req\_wine changes value, the tokens are removed from the incoming arcs and new tokens are placed on each outgoing arc. In this case, the token on the arc between ack\_wine− and req\_wine+ would be removed, and a new token would be put on the arc between req\_wine+ and ack\_wine+ as shown in Figure 1.5(b). In this marking, ack\_wine can now be set to '1', and no other signal transition is possible. After ack\_wine becomes '1', tokens are removed from its two incoming arcs and tokens are placed on its two outgoing arcs as shown in Figure 1.5(c). In this new marking, there are two possible next signal transitions. Either req\_patron will be set to '1' or req\_wine will be set to '0'. These two signal transitions can occur in either order. The rest of the behavior of this circuit can be determined by similar analysis.

It takes quite a bit of practice to come up with a Petri-net model from any given word description. Another graphical model called the timed event/level
(TEL) structure has a more direct correspondence with the word description. The TEL structure for the Shop_PA_reshuffled protocol is shown in Figure 1.6. The first one describes the behavior of the winery, the second one describes the behavior of the patron, and the last one represents the behavior of the shop. The difference between a TEL structure and a Petri-net is the ability to specify signal levels on the arcs. Each level expression corresponds to a guard and each signal transition, or event, corresponds to an assign statement in the word description. There are four guards and four assign statements in the shop process which correspond to four levels and four events in the graph. Note that the dashed arcs represent the initial marking.

![Fig. 1.6 TEL structure for passive/active shop.](image)

### 1.5 DELAY INSENSITIVE CIRCUITS

Let’s go back now and look at those circuits from before. How do we know they work correctly? Let’s look again at our first circuit redrawn in Figure 1.7 as a complete circuit, that is with circuit elements for the environment. Let’s also add delay elements (cigar shaped boxes labeled '[0,inf]') to represent an unbounded delay. Now, recall that in the initial state all signal wires are set to '0'. In this state, the only thing that can happen is that req_wine can be set to '1'. This is due to the '0' at the input of the inverter which sets its output to '1'. We assume that this happens instantaneously. The delay element now randomly select a delay \(d\) between 0 and infinity. Let’s say that it picks 5 seconds. The signal req_wine now changes to '1' after 5 seconds have elapsed. At that point, the output of the attached buffer also changes to '1', but this change does not affect ack_wine until a random delay later due to the attached delay element. If you play with this for awhile, you should be able to convince yourself that regardless of what delay values you choose the circuit always behaves as you originally specified it. We call such a circuit a delay-insensitive circuit. That is one in which its correctness is independent of
both the delays of the gates and the wires even if these delays are unbounded. This mean that even during a strike of the grape mashers at the winery or when the patron is sleeping one off, the circuit still operates correctly. It is extremely robust.

![Diagram](image)

**Fig. 1.7** Complete circuit for active/active shop.

Let’s now look at the second circuit redrawn with its environment in Figure 1.8. Again, in the initial state, the only transition which can occur is that req\_wine can be set to ’1’ after some arbitrary delay. After req\_wine is ’1’, the next transition which can occur is that ack\_wine can be set to ’1’ after some delay. When ack\_wine is ’1’ though, there are two possible next transitions. Either req\_wine can go ’0’, or req\_patron can go to ’1’ depending on the choices made for the delay values in each delay element along the corresponding path. If you look at this circuit long enough, you can likely convince yourself that the circuit will behave as specified for whichever delay values are chosen each time around. This circuit is also delay-insensitive.

![Diagram](image)

**Fig. 1.8** Complete circuit for passive/active shop.

You may at this point begin to believe you can always build a delay-insensitive circuit. Unfortunately, this is not the case. We are actually pretty fortunate with these two circuit designs. In general, if you only use single-output gates, this class of circuits is severely limited. In particular, you can
only use buffers, inverts, and Muller C-elements to build delay-insensitive circuits. As an example, consider this circuit shown in Figure 1.9. It is a circuit implementation of a slightly modified version of our original four-phase protocol where we have added a state variable to get rid of the state coding problem. The protocol is given below:

```vhd
process state_variable
begin
assign(req_wine,'1'); -- call winery
guard(ack_wine,'1'); -- wine arrives
assign(x,'1'); -- set state variable
assign(req_wine,'0'); -- reset req_wine
guard(ack_wine,'0'); -- ack_wine resets
assign(req_patron,'1'); -- call patron
guard(ack_patron,'1'); -- patron buys wine
assign(x,'0'); -- reset state variable
assign(req_patron,'0'); -- reset req_patron
guard(ack_patron,'0'); -- ack_patron resets
end process;
```

Fig. 1.9 Another complete circuit for active/active shop.

Trace the following sequence of actions through the circuit:

req_wine+, ack_wine+, x+, req_wine−, ack_wine−, req_patron+, ack_patron+.

At this point, u2 and u6 are enabled to go to '0', but assume that the delay of u2− is shorter than that of u6−. After u2 becomes '0', x can go to '0', and assume it does so before u6 goes to '0'. After x becomes '0', u4 can go to '1'. If this happens before u6−, then req_wine can be enabled to go to '1'. Now, if u6 finally changes to '0' then req_wine no longer is enabled to change to '1'. In a real circuit what may happen is that req_wine may experience a small pulse which we call a hazard or glitch. Depending on the duration of the pulse, it may or may not be perceived by the winery as a request for wine. If it is perceived as a request, then when the true request for wine comes after req_patron and ack_patron have been reset, this will be perceived as a second request. This may lead to a second bottle of wine being delivered before the first has been sold. This potentially catastrophic chain of events would surely
spoil the shopkeeper’s holiday in Maui. The engineer discusses this problem with his colleagues Dr. Huffman and Dr. Muller over dinner.

1.6 Huffman Style Synthesis

The first person to arrive at dinner was Dr. Huffman. The first thing he did was redraw the original specification as an AFSM and a Huffman flow table as shown in Figure 1.10. Note that in the flow table the changing output in each unstable state is made a ‘don’t care’ which will help us out later.

![AFSM and Huffman flow table](image)

Fig. 1.10 (a)AFSM and (b) Huffman flow table for active/active shop (input/output vector is \( \langle \text{ack}_\text{wine}, \text{ack}_\text{patron} \rangle / \langle \text{req}_\text{wine}, \text{req}_\text{patron} \rangle \)).

The first thing he noticed is that there are more states than necessary. States 0 and 1 one are compatible. In each entry, either the next states and outputs are the same or in one is a ‘don’t care’. Similarly, states 2 and 3 are compatible. When states are compatible, they can be combined to reduce the number of states in the state machine. This process is called state minimization. The new AFSM and flow table are shown in Figure 1.11.

![Reduced AFSM and Huffman flow table](image)

Fig. 1.11 Reduced (a) AFSM and (b) Huffman flow table for the active/active shop.
Next, we must choose a state assignment for our two states. This is a unique binary value for each state. Now, care must be taken when doing this for an asynchronous design, but for this simple design only a single bit is needed. Therefore, encoding state 0 with '0' and state 1 with '1' will suffice here.

At this point, it is a simple matter to derive the circuit by creating and solving three Karnaugh maps. There is one for each output and one for the state signal given in Figure 1.12. The circuit implementation is shown in Figure 1.13.

\begin{tabular}{|c|c|c|c|c|}
\hline
\text{ack\_wine}/\text{ack\_patron} & \text{ack\_wine}/\text{ack\_patron} & \text{ack\_wine}/\text{ack\_patron} \\
\hline
\text{req\_wine} & \text{req\_patron} & \text{x} \\
\hline
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 \\
\hline
\end{tabular}

\textbf{Fig. 1.12} Karnaugh maps for the active/active shop.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{circuit_diagram}
\caption{Huffman's circuit for the active/active shop.}
\end{figure}

Let's compare this circuit with the previous one shown in Figure 1.9. The logic for \text{req\_wine} and \text{req\_patron} is identical. The logic for \text{x} is actually pretty similar to the previous circuit. Except that Huffman sticks with simple AND and OR gates. "Muller's C-element is cute, but it is really hard to find in any of the local electronics shops", says Dr. Huffman. Also, all of Huffman's delay elements have an upper bound, \(U\). Huffman assumes a \textit{bounded gate and wire delay model}. Huffman's circuit is also not closed. That is, he does not explicitly show the relationships between the outputs and inputs which is not necessary in his method. Huffman also splits the feedback into an output signal, \(X\), and an input signal, \(x\).

These are the obvious differences. There are also some not so obvious ones. Namely, Huffman’s assumptions on how this circuit will be operated. First of all, Huffman assumes that the circuit is operated in what is called single-input change fundamental-mode. What this means is that the environment will apply only a single input change at a time, then it will not apply another
until the circuit has stabilized. In other words, the operation of the circuit is as follows: an input changes, outputs and next state signals change, the next state is fed back to become the current state, and then a new input can arrive. To maintain this order, it may not only be necessary to slow down the environment, but it may also be necessary to delay the state signal change from being feedback too soon by adding a delay between $X$ and $x$.

Consider again the bad trace. Again, the following sequence can happen: \texttt{req\_wine+}, \texttt{ack\_wine+}, $X+$, $x+$, \texttt{req\_wine-}, \texttt{ack\_wine-}, \texttt{req\_patron+}, \texttt{ack\_patron+}. At this point, $u2$ and $u6$ are again enabled to go to '0', and lets assume that the delay of $u2-$ is faster than that of $u6-$. After $u2$ becomes '0', $X$ can go to '0' before $u6$ goes to '0'. However, in this case, we have added sufficient delay in the feedback path such that we do not allow $x$ to change to '0' until we have insured that the circuit has stabilized. In other words, as long as the delay in the feedback path is greater than $U$, the hazard is no longer there.

1.7 MULLER STYLE SYNTHESIS

"What kind of crazy ideas have you been putting in this poor man's head?", "Well, if it isn't Dr. Muller. You're late as usual", announced Dr. Huffman.

"You know me, I don't believe in bounding time. Let me take a look at what you got there", said Dr. Muller. He stared at the napkin and made some marks on it. "Now, that's much better. Take a look at this" (see Figure 1.14).

![Figure 1.14 Muller's circuit for active/active shop.](image)

Muller's circuit looks similar to Huffman's, but there are some important differences in Muller's model. First, notice that he removed the delay elements from the wires, and he left only a single delay element on each output and next state signal. He also changed the upper bound of this delay element to infinity. Finally, he changed the signal $X$ to $x$. In Muller's circuit, he has not put any restrictions on the order that inputs, outputs, and state signals change except they must behave according to the protocol. This model is called an
unbounded gate delay model. Note that in this model, Muller assumes that wire delays are negligible. This means that whenever a signal changes value, all gates it is connected to will see that change immediately. For example, \texttt{ack\_wine} and \texttt{ack\_patron} each fork to two other gates while \texttt{x} forks to three. These forks are called isochronic forks meaning they have no delay. This delay model is called speed-independent. A similar model where only certain forks are isochronic is called quasi-delay insensitive.

Consider again the bad trace. The following sequence can still happen:
\texttt{req\_wine\+, ack\_wine\+, x+, req\_wine\-, ack\_wine\-, req\_patron\+, ack\_patron\+.} At this point, both the gate for \texttt{req\_wine} and \texttt{x} will see the change in \texttt{ack\_patron} at the same time due to the isochronic fork. Therefore, when \texttt{x} goes to '0', the effect of \texttt{ack\_patron} being '1' will already be felt by \texttt{req\_wine} so it will not glitch to '1'. The circuit is hazard-free under Muller’s model as well, and he did not need to determine any sort of delay for the state variable feedback path.

"Those isochronic forks may be tricky to design though", insisted Dr. Huffman.

This is true. Both models require some special design. By the way, not all the forks actually need to be isochronic. In particular, you can put different delays on each of the branches of the wire fork for \texttt{x}, and the circuit still operates correctly.

1.8 TIMING ANALYSIS AND OPTIMIZATION

The engineer returned from the bar where he had been talking to the wine patron and the head of production from the winery. He learned a few interesting items which can be used to optimize the circuit. First, he learned that with the winery’s new wine production machine, they are always well ahead of schedule. In fact, since they are only 1 block away from the wine shop, they have guaranteed delivery within 2 to 3 minutes after being called. The patron lives about 5 blocks away and it takes him at least 5 minutes sprinting to get to the shop after being called. If he is busy sleeping one off, it may take him even longer. Finally, the circuit delays will all be very small, certainly less than 1 minute. Using this delay information and doing a bit of reshuffling, he came up with this description of the circuit and its environment:

\begin{verbatim}
Shop\_AA\_timed: process
begin
  assign(req\_wine,'1',0,1); -- call winery
  assign(req\_patron,'1',0,1); -- call patron
  -- wine arrives and patron arrives
  guard\_and(ack\_wine,'1',ack\_patron,'1');
  assign(req\_wine,'0',0,1);
  assign(req\_patron,'0',0,1);
  -- wait for ack\_wine and ack\_patron to reset
\end{verbatim}
guard_and(ack_wine, '0', ack_patron, '0');
end process;

winery: process
begin
  guard(req_wine, '1');  -- wine requested
  assign(ack_wine, '1', 2, 3);  -- deliver wine
  guard(req_wine, '0');
  assign(ack_wine, '0', 2, 3);
end process;

patron: process
begin
  guard(req_patron, '1');  -- shop called
  assign(ack_patron, '1', 5, inf);  -- buy wine
  guard(req_patron, '0');
  assign(ack_patron, '0', 5, 7);
end process;

The assignment function now takes two additional parameters which are
the lower and upper bound on the delay in which the assignment takes to
complete. The guard_and function requires multiple signals to attain a given
value before the circuit can progress. Notice that in this protocol, the patron
is called after the wine is requested, but before the wine arrives. This improves
the performance of the system by allowing more concurrency. As we will see,
timing makes sure that the patron does not arrive too early.

The TEL structure for this specification is shown in Figure 1.15. In the
TEL structure, there are timing annotations on each arc. For example, rep-
wine must go to '1' within 0 to 1 minute from the initial state. Once req_wine
has gone to '1', ack_wine can go to '1' since its level expression is now true.
However, it must wait at least 2 minutes and will change within 3 minutes.
The signal req_patron is also enabled to change and will do so within 0 to 1
minute. Therefore, we know that req_patron will change first. After req_patron
has gone to '1', ack_wine is now enabled to change to '1', but it must wait
at least 5 minutes. Therefore, we know that ack_wine changes next. In other
words, we know the wine will arrive before the patron arrives. A very
important property of the system.

From this TEL structure, it is possible to derive the state graph in Figure
1.16. In the initial state, all signals are '0', but req_wine is labelled with
an 'R' to indicate that it is enabled to 'rise'. Once req_wine rises, we move to
a new state where req_patron and ack_wine are both enabled to rise. However,
as mentioned before the only possible next state transition is on req_patron
rising.

To get a circuit, a Karnaugh map is created for each output with columns
for each input combination and rows for each output combination. A '1' is
placed in each entry corresponding to a state where the output is either 'R' or
'1', a '0' in each entry where the output is 'F' or '0', and a '-' in the remaining
entries. From the maps shown in Figure 1.17, the simple circuit in Figure 1.18
is derived.
Fig. 1.15 TEL structure for an active/active shop.

Fig. 1.16 State graph for a active/active shop (with state vector \{ack\_wine, ack\_patron, req\_wine, req\_patron\}).

Fig. 1.17 Karnaugh maps for active/active shop.
1.9 PERFORMANCE ANALYSIS AND VERIFICATION

With all these design alternatives, how does one know which one to choose? First, you may want to choose the fastest. In a synchronous design, you simply determine the worst-case delay, and set your clock cycle to match it. Then, you count how many clock cycles it takes to do your job. In an asynchronous circuit, however, it is not that simple. You need to determine the average performance. Let us assume that the delays are those used in the timed circuit, and they are uniformly distributed over their range, except that the patron’s delay in buying the wine actually ranges uniformly between 5 and 10 minutes (it may be more, but that is pretty unlikely). If this is the case, we find that the protocol used by Huffman and Muller has a cycle time of 21.5 minutes on average while our original one (ShopAA_reshuffled) had a cycle time of 20.6 minutes. However, the timed circuit’s cycle time is only 15.8 minutes. This means that on average the patron will get about one more bottle of wine every hour using the timed circuit!

The timed circuit made some pretty aggressive assumptions. How does one know if the circuit operates correctly? The first thing to do is to simulate the circuit for some example situations and check that all parties end up happy. This process is validation. However, to be really sure, one must exhaustively check all possible situations. This process is formal verification. First, one can check that the circuit always does what it is specified to do. This type of checks if a circuit conforms to its specification. In other words, the circuit should follow the protocol. However, this may not be sufficient for the shopkeeper to rest easy on the Maui beaches. For example, in the modified protocol, if the timing assumptions are violated, then it is possible for the patron to arrive before the wine. Knowing how irate he gets when this happens, this is clearly not acceptable. Therefore, one should enumerate the properties that the circuit should have and check that the protocol with the timing assumptions satisfies these properties. This process is called model checking. A couple such properties might be these:

1. The wine arrives before the patron.
2. When the wine is requested, it eventually arrives.

Both of these properties hold for all states in Figure 1.16.

"Anyway, this is enough talk about work for now. What do you say that
we order some dinner?", said the engineer.

"This is just getting interesting. I would like to learn more.", spoke the
shopkeeper.

"Why don’t you attend our seminar at the local university", replied the
engineer.

Did the shopkeeper attend the seminar? Did he and his wife ever get that
trip to Maui? Did they ever get their dinner? To learn more, you must read
the next chapter.

1.10 SOURCES

Detailed references on the material in this chapter is given at the end of
the subsequent chapters when these topics are discussed later. The idea
of modeling communicating processes through send and receive actions on
channels originated with Hoare’s Communicating Sequential Processes (CSP)
[Hoa78, Hoa85], and it was adapted to the specification of asynchronous
circuits by Martin [Mar90b]. The Huffman flow table is described in [Huf54]
(later republished in [Huf64]), Petri-nets were introduced in [Pet66], and they
were first adapted to the specification of asynchronous circuits by Seitz [Sei71].
TEL structures were introduced by Belhomini [BM97].

In [Mar90a], Martin proved that when the gate-library is restricted to
single-output gates, the class of circuits that can be built using a purely delay
insensitive model is severely limited. Namely, these circuits can only utilize
inverters and Muller C-elements. This result and the examples contained in
this paper inspired much of the discussion in this chapter. Techniques using
the fundamental-mode assumption originated with Huffman [Huf54, Huf64].
The speed-independent model was first proposed by Muller [MB59], and it
was used in the design of the ILLIAC and ILLIAC II [Bre65]. In [MM92],
Myers introduced the first synthesis method which utilized the timed circuit
model to optimize the implementation. Earlier, however, most practical asyn-
chronous designs certainly utilized timing assumptions in an ad hoc fashion
to improve the implementation.

Problems

1.1 Describe in words the behavior of the channel level shop process given
below.

    Shop: process
    begin
    receive(WineryShop, bottle1);
receive(WineryShop, bottle2);
send(ShopPatron, bottle2);
send(ShopPatron, bottle1);
end process;

1.2 In this problem, we will design a Huffman circuit for the shop when the
winery is passive while the patron is active.
1.2.1. Write a process for the 4-phase protocol for an active/passive shop.
1.2.2. A reshuffled version of an active/passive shop is shown below. Give
an AFSM and Huffman flow table that describes its behavior.

Shop_Ap_reshuffled: process
begin
  guard(req_pataon,'1'); -- patron calls
  assign(req_wine,'1'); -- call winery
  guard(ack_wine,'1'); -- wine arrives
  assign(req_wine,'0'); -- reset req_wine
  guard(ack_wine,'0'); -- ack_wine resets
  assign(ack_pataon,'1'); -- sells wine
  guard(req_pataon,'0'); -- req_paton resets
  assign(ack_pataon,'0'); -- reset ack_paton
end process;

1.2.3. Combine compatible rows in the flow table and make a state
assignment.
1.2.4. Use Karnaugh maps to find a Huffman circuit implementation.

1.3 In this problem, we will design a Muller circuit for the shop when the
winery is passive while the patron is active.
1.3.1. Draw a TEL structure for the reshuffled active/passive shop shown
below.

Shop_Ap_reshuffled: process
begin
  assign(req_wine,'1'); -- call winery
  guard(ack_wine,'1'); -- wine arrives
  guard(req_paton,'1'); -- patron calls
  assign(ack_paton,'1'); -- sells wine
  assign(req_wine,'0'); -- reset req_wine
  guard(ack_wine,'0'); -- ack_wine resets
  guard(req_paton,'0'); -- req_paton resets
  assign(ack_paton,'0'); -- reset ack_paton
end process;

1.3.2. Find the state graph from the TEL structure.
1.3.3. Use Karnaugh maps to derive a Muller circuit implementation.
References


Index

Communicating Sequential Processes (CSP), 20
Huffman flow table, 7
Karnaugh maps, 14
Muller C-element, 7
Petri-net (PN), 8
Active communication, 5
Asynchronous finite state machine (AFSM), 7
Asynchronous timing, 2
Bounded delay, 14
Channel, 3
Compatible states, 13
Complete circuit, 10
Conformance, 19
Delay elements, 10
Delay-insensitive circuit, 10
Environment, 3
Events, 10
Formal verification, 19
Four-cycle signalling, 5
Four-phase signalling, 5
Fundamental-mode, 14
Handshaking, 2
Hazard, 12
Hold time, 2
Inputs, 4
Isochronic forks, 16
Level signalling, 5
Levels, 10
Marking, 8
Model checking, 19
Outputs, 4
Passive communication, 5
Protocol, 4
Quasi-delay insensitive, 16
Reshuffling, 6
Setup time, 2
Specification, 3
Speed-independent, 16
State assignment, 14
State minimization, 13
State variable, 12
Synchronization failure, 2
Synchronous timing, 2
Tokens, 8
Transition signalling, 5
Two-cycle signalling, 5
Two-phase signalling, 5
Unbounded delay, 16
Vacuous, 7
Validation, 19