The definition of clock and system timing are integral parts of a sequential digital circuit. Data in a digital system moves from one storage device to the next by the virtue of a system clock. During its travel, data is routed in and out of different combinational logic blocks, and becomes modified to satisfy a specific functionality.

This chapter is dedicated to reviewing the basics of memory devices that store data, and sequential circuits that use memory devices to operate. The chapter begins with the introduction of two basic memory elements, the latch and the flip-flop. It then explains how data travels between memory elements using timing diagrams, and how timing violations form as a result of unexpected combinational logic delays on the data path or in the clock line. Later in the chapter, the basic sequential building blocks such as registers, shift registers and counters are examined. Moore-type and Mealy-type state machines that control data movement are also studied; their advantages and disadvantages are compared against counter-decoder type controllers in various design tasks. The concept of block memory and how it is used in a digital system is introduced at the end of this chapter. The chapter concludes with a comprehensive example which demonstrates data transfer from one memory block to another, how to build a detailed data-path during the development of the design, and how to use timing diagrams to build a controller.

## 2.1 D Latch

The D Latch is the most basic memory element in logic design. It has a data input, D, a clock input, clock, and a data output, Q, as shown in the top portion of Fig. 2.1. It contains a tri-state inverter at its input stage followed by two back-to-back inverters connected in a loop configuration, which serves to store data.
The clock signal connected to the enable input of the tri-state inverter can be set either to active-high or active-low. In Fig. 2.1, the changes at the input transmit though the memory element, and become the output during the low phase of the clock. In contrast, the changes at the input are blocked during the high phase of the clock, and no data transmits to the output. Once the data is stored in the back-to-back inverter loop, it becomes stable and does not change until different data is introduced at the input. The buffer at the output stage of the latch is used to drive multiple logic gate inputs.

The operation of the D latch is shown in Fig. 2.2. During the low phase of the clock, the tri-state inverter is enabled. The new data transmits through the tri-state inverter, overwrites the old data in the back-to-back inverter stage, and reaches the output. When the clock switches to its high phase, the input-output data transmission stops because the tri-state buffer is disabled and blocks any new data transfer. Therefore, if certain data needs to be retained in the latch, it needs to be stored some time before the rising edge of the clock. This time interval is called the set-up time, $t_S$, and it is approximately equal to the sum of delays through the tri-state inverter and the inverter in the memory element. At the high phase of the clock, the data stored in the latch can no longer change as shown in Fig. 2.2.

**Fig. 2.1** Logic and circuit diagrams of a D latch
2.2 Timing Methodology Using D Latches

Timing in logic systems is maintained by pipeline structures. A pipeline consists of combinational logic blocks bounded by memory elements as shown in the top portion of Fig. 2.3. The main purpose of pipelines is to process several data packets within the same clock cycle and maximize the data throughput.

To illustrate the concept of pipeline, latches are used as memory elements in the pipeline structure shown in Fig. 2.3. In every latch boundary, data propagates from one combinational logic stage to the next at the high and at the low phases of the clock.

The bottom part of Fig. 2.3 shows the timing diagram of a data transfer for a set of data packets ranging from D1 to D3 at the IN terminal. The first data packet, D1\(^0\), retains its original value during the high phase of the clock (Cycle 1H) at the node A. D1\(^0\) then propagates through the T1 stage, and settles at the node B in a modified form, D1\(^1\), sometime before the falling edge of the clock. Similarly, D1\(^1\) at the node C retains its value during the low phase of the clock while its processed form, D1\(^2\), propagates through the T2 stage, and arrives at the node D before the rising edge of the clock. This data is processed further in the T3 stage, and transforms into D1\(^3\) before it becomes available at the OUT terminal at the falling edge of the clock in Cycle 2L.

Similarly, the next two data packets, D2\(^0\) and D3\(^0\), are also fed into the pipeline at the subsequent negative clock edges. Both of these data propagate through the combinational logic stages, T1, T2 and T3, and become available at the OUT terminal at the falling edge of Cycle 3L and Cycle 4L, respectively.

![Operation of D latch](image)

**Fig. 2.2** Operation of D latch
The total execution time for all three data packets takes four clock cycles according to the timing diagram in Fig. 2.3. If we were to remove all the latch boundaries between nodes A and F, and wait until all three data packets, D1, D2 and D3, were processed through the sum of the three combinational logic stages, T1, T2 and T3, the total execution time would have been $3 \times 1.5 = 4.5$ clock cycles as each combinational logic stage requires half a clock cycle to process data. Therefore, pipelining can be used advantageously to process data in a shorter amount of time and increase data throughput.

**Fig. 2.3** Timing methodology using D latches (“X” marks correspond to changing data)

### 2.3 D Flip-Flop

The D flip-flop is another important timing element in logic design to maintain timely propagation of data from one combinational logic block to the next.

Similar to a latch, the flip-flop has also a data input, D, a clock input, clock, and a data output, Q, as shown in the top portion of Fig. 2.4.

The bottom part of Fig. 2.4 shows the circuit schematic of a typical flip-flop which contains two latches in series. The first latch has an active-low clock input, and it is called the master. The second latch has an active-high clock input, and it is called the slave. The
master accepts new data during the low phase of the clock, and transfers this data to the slave during the high phase of the clock.

![Logic and circuit diagrams of a D flip-flop](image)

**Fig. 2.4** Logic and circuit diagrams of a D flip-flop

Figure 2.5 shows the timing attributes of a flip-flop. The set-up time, $t_S$, is the time interval for valid data to arrive and settle in the master latch before the rising edge of the clock. Hold time, $t_H$, is the time interval after the positive edge of the clock when valid data needs to be kept steady and unchanged. The data stored in the master latch propagates through the slave latch and becomes the flip-flop output some time after the rising edge of the clock, and it is called clock-to-q delay or $t_{CLKQ}$.

![Timing attributes of a D flip-flop](image)

**Fig. 2.5** Timing attributes of a D flip-flop

The operation of the flip-flop in two different phases of the clock is shown in Fig. 2.6. During the low phase of the clock, new data enters the master latch, and it is stored. This data cannot propagate beyond the master latch because the tri-state inverter in the slave latch acts as an open circuit during the low phase of the clock. The flip-flop output reveals only the old data stored in the slave latch. When the clock goes high, the new data stored in the master
latch transmits through the slave and arrives at the output. One can approximate values of $t_S$ and $t_{CLKQ}$ using the existing gate delays in the flip-flop.

**Fig. 2.6** Operation of D flip-flop

### 2.4 Timing Methodology Using D Flip-Flops

Data propagation through a pipeline with D flip-flops is shown in Fig. 2.7. The bottom part of Fig. 2.7 shows the timing diagram of a data transfer for a set of data packets ranging from D1 to D3 at the IN terminal.

The first data packet, D1$^0$, at the IN terminal has to be steady and valid during the set-up and hold periods of the flip-flop, but it is free to change during the remaining part of the clock period as shown by oscillatory lines. Once the clock goes high, the valid D1$^0$ starts to propagate through the combinational logic block of T1 and reaches the second flip-flop boundary. The processed data, D1$^1$, has to arrive at the second flip-flop input, B, no later than the set-up time of the flip-flop. Otherwise, the correct data cannot be latched. D1$^1$ propagates through the second (T2) and third (T3) combinational logic stages, and becomes D1$^2$ and D1$^3$, respectively, before exiting at the OUT terminal as shown in the timing diagram in Fig. 2.7.

The subsequent data packets, D2$^0$ and D3$^0$, are similarly fed into the pipeline stage from the IN terminal following D1$^0$. They are processed and modified by the T1, T2 and T3 combinational logic stages as they propagate through the pipeline, and emerge at the OUT terminal.

The total execution time for three input data packets, D1$^0$, D2$^0$ and D3$^0$, takes six clock cycles, including the initial three clock cycle build-up period before D1$^3$ emerges at the OUT terminal. If we were to remove all the flip-flop boundaries between the nodes A and F, and
wait for these three data packets to be processed without any pipeline structure, the total execution time would have been $3 \times 3 = 9$ clock cycles, assuming each T1, T2 or T3 logic stage imposes one clock cycle delay.

Once again, the pipelining technique considerably reduces the overall processing time and increase data throughput whether the timing methodology is latch-based or flip-flop-based.

The advantage of using latches as opposed to flip-flops is to be able to borrow time from neighboring stages. For example, the propagation delay in the T1 stage in Fig. 2.3 can be extended at the expense of shortening the propagation delay in the T2 stage. This flexibility does not exist in a flip-flop based design in Fig. 2.7.

**Fig. 2.7** Timing methodology using D flip-flops (“X” marks correspond to changing data)

### 2.5 Timing Violations

Although pipelining scheme helps reducing the overall data processing time, we still need to watch out possible timing violations because of the unexpected delays in the data-path and the clock network.

Therefore, this section examines the set-up and hold timing violations in flip-flop controlled pipelines, and proposes possible solutions to eliminate them.
Figure 2.8 shows a section of a pipeline where a combinational logic block with a propagation delay of $T_{\text{COMB}}$ is sandwiched between two flip-flop boundaries. At the rising edge of the clock, the valid data that meets the set-up and hold time requirements is introduced at the IN terminal. After $t_{\text{CLKQ}}$ delay, the data emerges at the node A and propagates through the combinational logic block as shown in the timing diagram. However, the data arrives at the node B too late and violates the allocated set-up time of the flip-flop. This is called the set-up violation. The amount of violation is dependent on the clock period and is calculated as follows:

Set-up violation $= t_S - [T_C - (t_{\text{CLKQ}} + T_{\text{COMB}})]$

![Timing Diagram](image)

**Fig. 2.8** Setup violation

Figure 2.9 describes the hold time violation where the clock shifts by $T_{\text{CLK}}$ due to an unexpected delay in the clock line. In the timing diagram, the valid data is introduced to the pipeline at the IN terminal, and it arrives at the node B after a short delay equal to $(t_{\text{CLKQ}} + T_{\text{COMB}})$. The shifted clock, on the other hand, creates a substantial set-up time slack equal to $(T_C + T_{\text{CLK}} - t_S - t_{\text{CLKQ}} - T_{\text{COMB}})$, but it also produces a hold time violation at the delayed clock edge. The amount of violation is dependent on the clock delay and is calculated as follows:

Hold violation $= (T_{\text{CLK}} + t_H) - (t_{\text{CLKQ}} + T_{\text{COMB}})$
Set-up violations can be recovered simply by increasing the clock period, $T_C$. However, there is no easy way to fix hold violations as they need to be searched at each flip-flop input. When they are found, buffer delays are added to the combinational logic block, $T_{COMB}$, in order to avoid the violation.

The schematic in Fig. 2.10 examines the timing ramifications of two combinational logic blocks with different propagation delays merging into a single block. The data arrives at the node C much earlier than the node D as shown in the timing diagram. The data at the nodes C and D propagate through the last combinational block and arrive at the node E. This scenario creates minimum and maximum delay paths at the node E. We need to focus on the maximum path, ($T_2 + T_3$), when examining the possibility of a set-up violation and the minimum path, ($T_1 + T_3$), when examining the possibility of a hold violation at the next flip-flop boundary.

**Fig. 2.9** Hold violation

Set-up violations can be recovered simply by increasing the clock period, $T_C$. However, there is no easy way to fix hold violations as they need to be searched at each flip-flop input. When they are found, buffer delays are added to the combinational logic block, $T_{COMB}$, in order to avoid the violation.

The schematic in Fig. 2.10 examines the timing ramifications of two combinational logic blocks with different propagation delays merging into a single block. The data arrives at the node C much earlier than the node D as shown in the timing diagram. The data at the nodes C and D propagate through the last combinational block and arrive at the node E. This scenario creates minimum and maximum delay paths at the node E. We need to focus on the maximum path, ($T_2 + T_3$), when examining the possibility of a set-up violation and the minimum path, ($T_1 + T_3$), when examining the possibility of a hold violation at the next flip-flop boundary.
To further illustrate the timing issues involving multiple combinational logic blocks, an example is given in Fig. 2.11 where two combinational logic blocks merge into a single block. The adder is bypassed with the inclusion of a 2-1 MUX which selects either the output of the adder or the bypass path, by a selector input, SEL.

The propagation delays of the inverter, $T_{INV}$, and the two-input NAND gate, $T_{NAND2}$, are given as 100 ps and 200 ps, respectively. The set-up, hold and clock-to-q delays are also given as 100 ps, 0 ps and 300 ps, respectively.

**Fig. 2.10** A timing example combining two independent data-paths

To further illustrate the timing issues involving multiple combinational logic blocks, an example is given in Fig. 2.11 where two combinational logic blocks merge into a single block. The adder is bypassed with the inclusion of a 2-1 MUX which selects either the output of the adder or the bypass path, by a selector input, SEL.

The propagation delays of the inverter, $T_{INV}$, and the two-input NAND gate, $T_{NAND2}$, are given as 100 ps and 200 ps, respectively. The set-up, hold and clock-to-q delays are also given as 100 ps, 0 ps and 300 ps, respectively.
Both the one-bit full adder and the 2-1 MUX are decomposed into basic logic gates, such as inverters and two-input NAND gates, as shown in Fig. 2.12. We obtain a total of seven propagation paths all merging at the node R. However, we only need to search for the maximum and the minimum delay paths to locate possible set-up and hold violations.

The maximum delay path consists of the inverter 1, and the series combination of four two-input NAND gates numbered as 1, 3, 4 and 6 shown in the schematic. This path results in a total delay of 900 ps. The minimum delay path, on the other hand, contains two two-input NAND gates numbered as 5 and 6, and it produces a delay of 400 ps. Placing these delays in the timing diagram in Fig. 2.13 yields a set-up slack of 100 ps at the node R when a clock period of 1400 ps is used. There is no need to investigate hold violations because there is no shift in the clock edge. However, if there were a shift in the clock line beyond \( t_{\text{CLKQ}} = 300 \) ps, then we would have a hold violation, and it would require an additional combinational logic delay in the data-path to proportionally shift the valid data at the node R to compensate the hold violation. This, however, would also eliminate the 100 ps set-up slack in Fig. 2.13.

**Fig. 2.11** An example with multiple propagation paths

---

- \( T_{\text{INV}} = 100 \) ps
- \( T_{\text{NAND2}} = 200 \) ps
- \( t_s = 100 \) ps
- \( t_h = 0 \) ps
- \( t_{\text{CLKQ}} = 300 \) ps
Fig. 2.12 Logic circuit of Fig. 2.11 showing maximum and minimum paths

Fig. 2.13 Timing diagram of the circuit in Fig. 2.12
2.6 Register

While the flip-flop holds data for only one clock cycle until new data arrives at the next clock edge, the register can hold the same data perpetually until the power is turned off.

Figure 2.14 shows the circuit diagram of a one-bit register composed of a flip-flop and a 2-1 MUX. The Write Enable pin, WE, is a selector input to the 2-1 MUX and transfers new data from the IN terminal to the flip-flop input when WE = 1. If the WE input is at logic 0, any attempt to write new data to the register is blocked. The old data stored in the flip-flop simply circulates around the feedback loop from one clock cycle to the next.

The timing diagram at the bottom of Fig. 2.14 describes the operation of the one-bit register. The data at the IN terminal is blocked until the WE input becomes logic 1 in the middle of the second clock cycle. At this point, the new data is allowed to pass through the 2-1 MUX, and it renews the contents of the register at the beginning of the third clock cycle. The WE input transitions to logic 0 before the end of the third clock cycle, and causes the register output, OUT, to stay at logic 1 during the fourth clock cycle.

![One-bit register and a sample timing diagram](image)

Fig. 2.14 One-bit register and a sample timing diagram

A 32-bit register shown in Fig. 2.15 is composed of 32 one-bit registers. All 32 registers have a common clock and WE input. Therefore, any new 32-bit data introduced at the register input changes the contents of the register at the rising edge of the clock if the WE input is set to logic 1.
2.7 Shift Register

The shift register is a particular version of an ordinary register, and it specializes in shifting data to the right or to the left according to the design needs.

Figure 2.16 shows the circuit schematic of a four-bit shift register that shifts serial data at the IN terminal to the left if enabled.

The operation of this shift register is explained in the timing diagram in Fig. 2.17. In cycle 1, SHIFT = 0. Therefore, the change at the IN terminal during this cycle does not affect the register outputs. However, when the SHIFT input transitions to logic 1 in the middle of cycle 2, it allows IN = 1 to pass to the least significant output bit, OUT[0], at the beginning of the third clock cycle. From the middle of cycle 2 to the middle of cycle 13, SHIFT is kept at logic 1. Therefore, any change at the IN node directly transmits to the OUT[0] node at the positive edge of each clock cycle. The other outputs, OUT[1], OUT[2] and OUT[3], produce delayed outputs one clock cycle apart from each other because the output of a lesser significant bit is connected to the input of a greater significant bit in the shift register.

When the SHIFT input becomes logic 0 from the middle of cycle 13 to cycle 17, the shift register becomes impervious to any change at the IN terminal, and retains the old values from the beginning of cycle 13 to cycle 18 as seen in Fig. 2.17. From the middle of cycle 17 onwards, the SHIFT input becomes logic 1 again, and the shift register distributes all new data entries at the IN terminal to its outputs.
2.8 Counter

The counter is a special form of a register which is designed to count up (or down) at each rising edge of the clock.

The circuit schematic in Fig. 2.18 shows a typical 32-bit up-counter with two control inputs, COUNT and LOAD. The COUNT = 1 entry enables the counter to count upwards at the rising edge of each clock cycle, and the LOAD = 1 entry loads new data to the counter from its IN [31:0] terminal. Once loaded, the counter output, OUT[31:0], increments by one at the positive edge of each clock cycle until all the output bits become logic 1. The next increment

Fig. 2.16 Four-bit shift register

Fig. 2.17 A sample timing diagram of the four-bit shift register in Fig. 2.16

2.8 Counter

The counter is a special form of a register which is designed to count up (or down) at each rising edge of the clock.

The circuit schematic in Fig. 2.18 shows a typical 32-bit up-counter with two control inputs, COUNT and LOAD. The COUNT = 1 entry enables the counter to count upwards at the rising edge of each clock cycle, and the LOAD = 1 entry loads new data to the counter from its IN [31:0] terminal. Once loaded, the counter output, OUT[31:0], increments by one at the positive edge of each clock cycle until all the output bits become logic 1. The next increment
automatically resets the counter output to logic 0. When LOAD = COUNT = 0, the counter neither loads new data nor is able to count upwards; it stalls and repeats its old output value.

The sample timing diagram at the bottom of Fig. 2.18 illustrates its operation. Prior to the first clock edge, the LOAD input is at logic 1 which allows an input value, IN = 3, to be stored in the counter. This results in OUT[31:0] = 3 at the positive edge of the first clock cycle. The LOAD = 0 and COUNT = 1 entries before the end of the first clock cycle start the up-count process, and the contents of the output, OUT[31:0] = 3, is subsequently incremented by one. The result, 3 + 1 = 4, passes through the C-port of the 3-1 MUX and arrives at the flip-flop inputs. At the positive edge of the second clock cycle, this new value overwrites the old registered value, and the OUT[31:0] node becomes equal to 4. In the next cycle, the counter goes through the same process and increments by one again. However, in the same cycle, the COUNT input also transitions to logic 0, and turns on the I-port of the 3-1 MUX. This input value prevents any new data from entering the up-counter, and it keeps the old data in the following clock cycles. As a result, the counter output stops incrementing and stalls at the value of OUT[31:0] = 5.

Fig. 2.18 A 32-bit counter and a sample timing diagram
2.9 Moore Machine

A state machine can be formed as soon as a flip-flop output is connected to a flip-flop input. Therefore, an overall state machine topology consists of flip-flops, feedback loops from flip-flop outputs to flip-flop inputs, and combinational logic blocks connected to flip-flop outputs and embedded in feedback loops.

Figure 2.19 shows the Moore-type state machine topology consisting of a flip-flop and a feedback loop. In this configuration, the feedback loop includes a combinational logic block that accepts both the flip-flop output and external inputs. If there are multiple flip-flops, the combination of all flip-flop outputs constitutes the “present” state of the machine. The combination of all flip-flop inputs is called the “next” state because at the positive edge of the clock these inputs become the flip-flop outputs, and form the present state. Flip-flop outputs are processed further by an additional combinational logic block, forming the present state outputs.

The basic state diagram of a Moore machine, therefore, includes the present state (PS) and the next state (NS) as shown in Fig. 2.19. The machine can transition from the PS to the NS if the required present state inputs are supplied. The outputs of the Moore machine are solely generated by the present state, and they emerge only from the current states as shown in the basic state diagram.

![Fig. 2.19 Block diagram and state representation of Moore machine](image)

The state diagram in Fig. 2.20 shows an example of a Moore-type machine with four states. Note that every state-to-state transition in the state diagram requires a valid present state input entry, and every node generates one present state output.

The state 0, S0, produces a present state output, \( \text{OUT} = 1 \), regardless of the value of the present state input, IN. When \( \text{IN} = 1 \), the state S0 transitions to the next state S1. Otherwise, it circulates back to itself. The state S1 produces \( \text{OUT} = 2 \); its next state becomes S1 if \( \text{IN} = 0 \), or it becomes S2 if \( \text{IN} = 1 \). The state S2 also produces a present state output, \( \text{OUT} = 3 \), and transitions to the state S3 if \( \text{IN} = 1 \). The state S2 remains unchanged if \( \text{IN} = 0 \). In the fourth and the final state, the present state output from S3 becomes 4. The machine stays in this state if \( \text{IN} \) stays at 0; otherwise, it goes back to the state S1.
The present state inputs and outputs of this Moore machine and its states can be tabulated in a table called the “state table” given in Fig. 2.21. In this table, the first column under the PS entry lists all the possible present states in the state diagram in Fig. 2.20. The middle two columns contain the next state entries for $IN = 0$ and $IN = 1$. The last column lists the present state outputs, one for each present state.

### Fig. 2.20 State diagram of a Moore machine with four states

<table>
<thead>
<tr>
<th>PS</th>
<th>NS</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S0</td>
<td>1</td>
</tr>
<tr>
<td>S1</td>
<td>S1</td>
<td>2</td>
</tr>
<tr>
<td>S2</td>
<td>S2</td>
<td>3</td>
</tr>
<tr>
<td>S3</td>
<td>S3</td>
<td>4</td>
</tr>
</tbody>
</table>

### Fig. 2.21 State table of the Moore machine in Fig. 2.20
The binary state assignment is performed according to Fig. 2.22 where only one bit is changed between adjacent states.

<table>
<thead>
<tr>
<th>States</th>
<th>NS1</th>
<th>NS0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>S2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>S3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Fig. 2.22** Bit representations of states S0, S1, S2 and S3

The binary form of the state table in Fig. 2.21 is reconstructed in Fig. 2.23 according to the state assignment in Fig. 2.22. This table is called the “transition table”, and it includes the binary representation of the next state and the present state outputs.

<table>
<thead>
<tr>
<th>PS1</th>
<th>PS0</th>
<th>IN = 0</th>
<th>IN = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>NS1 NS0</td>
<td>NS1 NS0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Fig. 2.23** Transition table of the Moore machine in Fig. 2.20

Forming this machine’s K-maps for the NS0, NS1, OUT0, OUT1 and OUT2 requires grouping all the input terms, PS1, PS0 and IN, according to the table in Fig. 2.23. The K-maps and their corresponding Sum of Products (SOP) expressions are shown in Fig. 2.24.
The next step is to generate the circuit diagram that produces all five outputs of the Moore machine according to these SOP expressions in Fig. 2.24. This circuit diagram is given in Fig. 2.25.

In order to generate this circuit, individual combinational logic blocks for NS0 and NS1 must be formed first in terms of PS0, PS1 and IN. Then, each NS0 and NS1 output is connected to the corresponding flip-flop input, producing the feedback loops for this state machine. The logic blocks for OUT0, OUT1 and OUT2 are generated directly from PS0 and PS1.

**Fig. 2.24** K-maps and SOP expressions for the Moore machine in Fig. 2.20

The next step is to generate the circuit diagram that produces all five outputs of the Moore machine according to these SOP expressions in Fig. 2.24. This circuit diagram is given in Fig. 2.25.

In order to generate this circuit, individual combinational logic blocks for NS0 and NS1 must be formed first in terms of PS0, PS1 and IN. Then, each NS0 and NS1 output is connected to the corresponding flip-flop input, producing the feedback loops for this state machine. The logic blocks for OUT0, OUT1 and OUT2 are generated directly from PS0 and PS1.

**Fig. 2.25** Logic circuit of the Moore machine in Fig. 2.20
2.10 Mealy Machine

The Mealy machine shares the same circuit topology with the Moore machine. The machine configuration also contains flip-flops and feedback loops as shown in Fig. 2.26. However, the present state outputs are generated from the combinational logic block in the feedback loop rather than from the present states as in the Moore-type machines.

As a result of this topology, the basic state diagram of a Mealy machine includes the present state, the next state and the input condition that makes the state-to-state transition possible as shown in Fig. 2.26. The present state output(s) does not emerge from the present state; instead, it is a function of the present state input(s) and the present state.

![Diagram of Mealy Machine](image)

**Fig. 2.26** Block diagram and state representation of Mealy machine

The Mealy state diagram in Fig. 2.27 exhibits similar characteristics compared to the Moore state diagram in Fig. 2.20, and all the state names and the state-to-state transitions in this diagram are kept the same for comparison purposes. However, each arrow connecting one state to the next carries the value of the present state output as a function of the present state input and the present state as indicated in Fig. 2.26. As a result, the Mealy state table in Fig. 2.28 contains two separate columns that tabulate the values of NS and OUT for IN = 0 and IN = 1. The binary state assignment is the same as in Fig. 2.22, which results in a transition table in Fig. 2.29.
Fig. 2.27 State diagram of a Mealy machine with four states

<table>
<thead>
<tr>
<th>PS</th>
<th>IN = 0</th>
<th>IN = 1</th>
<th>OUT = 0</th>
<th>OUT = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S0</td>
<td>S1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>S1</td>
<td>S1</td>
<td>S2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>S2</td>
<td>S2</td>
<td>S3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>S3</td>
<td>S3</td>
<td>S1</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Fig. 2.28 State table of the Mealy machine in Fig. 2.27
The K-maps for NS0, NS1, OUT0, OUT1 and OUT2 are formed according to the table in Fig. 2.29 and shown in Fig. 2.30 with the corresponding SOP expressions. Figure 2.31 shows the circuit diagram of this machine according to the expressions in Fig. 2.30. The methodology used to construct this circuit diagram is identical to the methodology used in the circuit diagram for the Moore machine in Fig. 2.25.

**Fig. 2.29** Transition table of the Mealy machine in Fig. 2.27

The K-maps for NS0, NS1, OUT0, OUT1 and OUT2 are formed according to the table in Fig. 2.29 and shown in Fig. 2.30 with the corresponding SOP expressions. Figure 2.31 shows the circuit diagram of this machine according to the expressions in Fig. 2.30. The methodology used to construct this circuit diagram is identical to the methodology used in the circuit diagram for the Moore machine in Fig. 2.25.

**Fig. 2.30** K-maps and SOP expressions for the Mealy machine in Fig. 2.27
2.11 Controller Design: Moore Machine Versus Counter-Decoder Scheme

Both Mealy and Moore-type state machines have practical implementation limits when it comes to design. A large ring-style state machine composed of N states such as in Fig. 2.32 may have multiple outputs attached to each state, making its implementation nearly impossible with conventional state machine implementation techniques. However, these types of designs are excellent candidates for the counter-decoder type of designs where each state in the state diagram is associated with a counter output value. Therefore, as the counter increments, present state outputs for each state can simply be generated by a set of decoders connected to the output of the counter.

Fig. 2.31 Logic circuit of the Mealy machine in Fig. 2.27
To illustrate this theory, a controller that generates the timing diagram in Fig. 2.33 will be implemented using both the Moore-type state machine and the counter-decoder approach.

From the timing diagram below, this state machine generates a single active-high output, Out = 1, once in every 8 cycles as long as Stop = 0. When Stop = 1, however, the machine stalls and it retains its current state.

![State diagram of a counter with N states](image)

**Fig. 2.32** State diagram of a counter with N states

Once the state assignments are made for each clock cycle in Fig. 2.33, the state diagram for a Moore-type state machine emerges in Fig. 2.34.

The first and the second clock cycles in the timing diagram are assigned to the S0 and the S1 states, respectively. The third clock cycle is assigned to the S2 state where Out = 1. The fourth clock cycle corresponds to the S3 state. The machine stays in the S3 state as long as...
Stop = 1. This ranges from the fourth to the sixth clock cycle in the timing diagram. The state assignments from the seventh to the tenth clock cycle become the S4, S5, S6 and S7 states, respectively. The eleventh clock cycle returns to the S0 state.

**Fig. 2.34** Moore representation of the timing diagram in Fig. 2.33

Implementing the state diagram in Fig. 2.34 follows a lengthy process of producing state tables, transition tables, and K-maps, resulting in a total of four outputs (three flip-flop outputs due to eight states and one output for Out). However, using a counter-decoder approach minimizes this design task considerably and reveals a rather explicit circuit implementation.

When the timing diagram in Fig. 2.33 is redrawn to implement the counter-decoder design approach, it yields a simple three-bit counter which counts from zero to seven as shown in Fig. 2.35. The counter output, CountOut, is included in this figure to show the relationships between the state assignments, the input (Stop) and the output (Out). The figure also shows the clock cycle where the counter resets itself when its output reaches seven.
The first task for the design is to construct a three-bit up-counter as shown in Fig. 2.36. The counter in this figure is derived from a general counter topology, and it consists of a three-bit adder, three 2-1 MUXes and three flip-flops. A three-input AND gate is used as a decoder at the counter output to implement Out = 1 when the CountOut node becomes 2. Therefore, this method follows a simple, step-by-step design approach in producing the final circuit that does not require implicit logic design techniques.

![Fig. 2.35 Timing diagram of a three-bit counter with a single input, stop, and a single output](image)

![Fig. 2.36 Counter-decoder representation of the timing diagram in Fig. 2.35](image)
2.12 Memory

Small memory blocks can be assembled from one-bit registers in a variety of configurations as shown in Fig. 2.14. For example, a 32-bit wide, 16-bit deep memory block shown in Fig. 2.37 can be built by stacking 16 rows of 32-bit registers on top of each other. Each 32-bit register contains tri-state buffers at its output to prevent logic contention during read as shown in Fig. 2.38.

The inputs to each column of the memory block in Fig. 2.37 are connected together to write data to a selected row. For example, the input terminal, IN[0], is connected to the In[0] pins of all 32-bit registers between row 0 to row 15 to be able to write a single bit at a selected row. The same is true for the remaining inputs, IN[1] to IN[31].

Similarly, all outputs of each column in Fig. 2.37 are connected together to read data from the memory block. For example, the output pin, OUT[0], is connected to the Out[0] pin of every 32-bit register from row 0 to row 15 to be able to read one bit from a selected row. The same is true for the remaining output pins, OUT[1] through OUT[31].

Every row of the memory block in Fig. 2.37 is accessed by an individual Write Enable (WE) and Read Enable (RE) signal for writing or reading data, respectively.

---

**Fig. 2.37** A 32x16 memory and the truth table of its address decoder
In order to generate the WE inputs, WE[0] to WE[15], an address decoder is used. This decoder enables only one row while deactivating all the other rows using a four-bit address, Address[3:0], and a single WE input according to the truth table in Fig. 2.39. For example, a 32-bit data is written to row 0 if WE = 1 and Address[3:0] = 0000 at the decoder input. However, WE = 0 blocks writing data to all rows of the memory block regardless of the input address as shown in the truth table in Fig. 2.40.

The RE inputs, RE[0] to RE[15], use address decoders similar to Figs. 2.39 and 2.40 to read a block of data from a selected row. The read operation is achieved with a valid input address and RE = 1 according to the truth table in Fig. 2.41. The RE = 0 entry disables reading data from any row regardless of the input address as shown in Fig. 2.42.

Therefore, a valid input address along with the RE and WE command inputs must be provided to the memory in order to perform a read or a write operation, respectively. The WE = 0, RE = 1 combination reads data from the selected row. Similarly, the WE = 1 and RE = 0 combination writes data to a selected row. The WE = 0 and RE = 0 combination disables both reading and writing to the memory block. The control input entry, WE = 1 and RE = 1, is not allowed, and it should be interpreted as memory read.

**Fig. 2.38** A 32-bit register slice at every row of Fig. 2.37
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>0 0 0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>0 0 0</td>
<td>0 1 0</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0 0 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>0 1 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>1 0 0</td>
<td>0 0 0</td>
</tr>
</tbody>
</table>

**Fig. 2.39** The address decoder for the 32x16 memory in Fig. 2.37 when WE = 1

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
</tbody>
</table>

**Fig. 2.40** The address decoder for the 32x16 memory in Fig. 2.37 when WE = 0

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>0 0 0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>0 0 0</td>
<td>0 1 0</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0 0 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>0 1 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>1 0 0</td>
<td>0 0 0</td>
</tr>
</tbody>
</table>

**Fig. 2.41** The address decoder for the 32x16 memory in Fig. 2.37 when RE = 1
This design example combines the data-path and controller design concepts described in this chapter and in Chap. 1. It also introduces the use of important sequential logic blocks such as flip-flop, register, counter and memory in the same design.

Every design starts with gathering small or large logic blocks to meet the functional specifications of the design and to construct a data-path for proper data-flow. Once the data-path is set, then the precise data movements from one logic block to the next are described using timing diagrams. Any architectural change in the data-path should follow a corresponding change in the timing diagram or vice versa.

When the data-path design and the timing diagram fully associate with each other, and each describes identical data movements, the next step in the design process is to build the controller that governs the flow of data. To define the states of the controller, the clock periods that generate different sets of outputs are separated from each other and named individually as distinct states. Similarly, the clock periods revealing identical outputs are grouped together under the same state name. The controller can be Moore-type or Mealy-type state machine according to the design needs. The design methodology of building the data-path, timing diagram and controller shown here will be repeated in every design throughout this book, especially when designing peripherals for a computer system in Chap. 7.

The example design in this section reads two eight-bit data packets from an 8x8 source memory (memory A), processes them and stores the result in an 8x4 target memory (memory B). The processing part depends on the relative contents of each data packet: if the contents

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

Fig. 2.42 The address decoder for the 32x16 memory in Fig. 2.37 when RE = 0
of the first data packet are larger than the second, the contents of the data packets are added. Otherwise, they are subtracted from each other before the result is stored.

The block diagram in Fig. 2.43 demonstrates the data-path required for this memory-to-memory data transfer as described above. The timing diagram in Fig. 2.44 needs to accompany the data-flow in Fig. 2.43 since it depicts precise data movements and values in each clock cycle.

To be able to write data to a memory address in Fig. 2.37, a valid data and address must be available within the same clock cycle. In a similar fashion, data is read from the memory core a cycle after a valid address is introduced.

Initially, counter A generates the addresses, 0 to 7, for memory A and writes the data packets, A0 to A7, through DataInA[7:0] port. This is shown in the timing diagram in Fig. 2.44 from clock cycles 1 through 8. When this task is complete, counter A resets and reads the first data packet A0 from AddrA[2:0] = 0 in clock cycle 9. In the next clock cycle, A0 becomes available at DOut1, and the counter A increments by one. In cycle 11, AddrA[2:0] becomes 2, the data packet A1 is read from DOut1[7:0], and the data packet A0 transfers to DOut2[7:0]. In this cycle, the contents of the data packets A0 and A1 are compared with each other by subtracting A1 (at DOut1) from A0 (at DOut2). If the contents of A0 are less than A1, then the sign bit, Sign, of (A0 – A1) becomes negative. Sign = 1 selects (A0 + A1) at ADDOut[7:0] and routes this value to DataInB[7:0]. However, if the contents of A0 are greater than A1, (A0 – A1) becomes positive. Sign = 0 selects (A0 – A1) and routes this value from SUBOut[7:0] to DataInB[7:0]. The result at DataInB[7:0] is written at AddrB[1:0] = 0 of memory B at the positive edge of clock cycle 12. In the same cycle, A1 is transferred to DOut2[7:0], and A2 becomes available at DOut1[7:0]. A comparison between A1 and A2 takes place, and either (A1 + A2) or (A1 – A2) is prompted to be written to memory B depending on the value of the Sign node. However, this is an unwarranted step in the data transfer process because the design requirement states that the comparison has to be done only once between two data packets from memory A. Since A1 is used in an earlier comparison with A0, A1 cannot be used in a subsequent comparison with A2, and neither (A1 + A2) nor (A1 – A2) should be written to memory B. The remaining clock cycles from 13 through 18 compare the values of A2 with A3, A4 with A5, and A6 with A7, and write the added or subtracted results into memory B. After clock cycle 19, all operations on this data-path suspend, the counters are reset and all writes to the memory core are disabled.
Fig. 2.43 Data-path of a memory transfer example
Fig. 2.44 Timing diagram for the memory transfer data-path in Fig. 2.43
To govern the data-flow in Fig. 2.44, a Moore-type state machine (or a counter-decoder-type controller) is used. A Mealy-type state machine for a controller design is usually avoided because present state inputs of this type of a state machine may change during the clock period and may cause jittery outputs to form.

The inclusion of the controller in Fig. 2.45 identifies the necessary control signals to be able to guide the data flow in Fig. 2.44 properly. These signals increment the counters A and B (with IncA and IncB), and enable writes to memory A or B (with WEA and WEB) when necessary. Thus, the timing diagram in Fig. 2.44 is expanded to include these control signals in Fig. 2.46, and this provides a complete picture of the data transfer process from memory A to memory B in contrast to the earlier timing diagram in Fig. 2.44.

**Fig. 2.45** Compete block diagram of the memory transfer example with controller
The controller in Fig. 2.45 can be implemented either by a Moore-type state machine in Fig. 2.47 or by a counter-decoder-type design in Fig. 2.48.

In the Moore type design, the states from S1 through S18 are assigned to each clock cycle of the timing diagram in Fig. 2.46. The values of the present state outputs, WEA, IncA, WEB and IncB, in each clock cycle are read from the timing diagram and attached to each state in Fig. 2.47. The reset state, S0, is included in the Moore machine in case the data-path receives an external reset signal to interrupt an ongoing data transfer process. Whichever state the state machine may be in, a Reset = 1 entry always forces the current state to transition back to the S0 state at the positive edge of the clock. These transitions are not included in Fig. 2.47 for simplicity.

The counter-decoder style design in Fig. 2.48 consists of a five-bit counter and four decoders to generate WEA, IncA, WEB and IncB control signals. To show the operation of this design to generate WEA, for example, this particular decoder includes eight five-input AND gates, one for each clock cycle from cycle 1 to cycle 8 in order to keep WEA = 1 in Fig. 2.46. The five-bit counter implicitly receives a reset signal from its output when it reaches clock cycle 18, and resets counter A, counter B and the rest of the system in Fig. 2.45.
Fig. 2.46 The complete timing diagram for the memory transfer in Fig. 2.45
Fig. 2.47  Moore representation of the controller unit in Fig. 2.45
Fig. 2.48 Counter-decoder representation of the controller unit in Fig. 2.45
**Review Questions**

1. Implement the following Moore machine:

   ![Moore Machine Diagram](image1)

2. Implement the following Moore machine using a timer. The timer is initiated when \( \text{In} = 1 \). With this input, the state machine goes to the A state and stays there for 10 cycles. In the tenth cycle, the state machine transitions to the B state and stays in this state for only one cycle before switching to the IDLE state. One implementation scheme is to construct a four-bit up-counter to generate the timer. When the counter output reaches 9, the decoder at the output of the counter informs the state machine to switch from the A state to the B state.

   ![State Machine Diagram](image2)
3. The following truth table needs to be implemented using two-input NAND gates and inverters.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ T_{NAND} \text{ (two-input NAND gate delay)} = 500 \text{ ps} \]
\[ T_{INV} \text{ (inverter delay)} = 500 \text{ ps} \]
\[ t_{clk-q} \text{ (clock-to-q delay)} = 200 \text{ ps} \]
\[ t_{su} \text{ (setup time)} = 200 \text{ ps} \]
\[ t_{h} \text{ (hold time)} = 300 \text{ ps} \]

(a) Implement this truth table between two flip-flop boundaries.
(b) Find the maximum clock frequency using a timing diagram.
(c) Shift the clock by 500 ps at the receiving flip-flop boundary. Show whether or not there is a hold violation using a timing diagram.

4. A block diagram is given below:

Block A contains only two flip-flops. Block B contains a one-bit adder with SUM and COUT outputs connected to two flip-flops as shown below.
(a) Using the logic gates with propagation delays listed below, determine the setup time for A, B, and CIN with respect to clkshift.

- 100ps
- 200ps
- 200ps
- 400ps
- 400ps

(b) Assuming $T = 0$ ns and $T_{CLK}$ (clock period) = 5 ns, if data at A, B and CIN become valid and stable 4 ns after the positive edge of clkshift, will there be any timing violations? Assume $t_H$ (hold time) = 3 ns for the flip-flop.

(c) How can you eliminate the timing violations? Show your calculations and draw a timing diagram with no timing violations.

5. A schematic is given below:
(a) If tsu (setup time) = 200 ps, th (hold time) = 200 ps and tclk-q (clock-to-q delay) = 300 ps for the flip-flop, and TA = 1000 ps, TB = 100 ps for the internal logic blocks on the schematic, show if there is any timing violation or timing slack in a detailed timing diagram if TC = 0 ps.

(b) What happens if TC = 400 ps? Show it in a separate timing diagram.

6. The state diagram of a Moore machine is given below:

The assignment of the states A, B and C are indicated as follows:

<table>
<thead>
<tr>
<th>states</th>
<th>PS[1]</th>
<th>PS[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(a) Implement this state machine using inverters, two-input and three-input AND gates and two-input OR gates.

(b) Find the maximum operating frequency of the implementation in part (a) if the following timing assignments are applied:

- tsu (setup time) = 100 ps, th (hold time) = 100 ps, tclk-q (clock-to-q delay) = 200 ps,
- TINV (inverter delay) = 200 ps, TAND2 (two-input AND gate delay) = 300 ps, TAND3 (three-input AND gate delay) = 400 ps, TOR2 (two-input OR gate delay) = 400 ps.

7. Data is transferred from Memory Tx to Memory Rx starting from the address 0x00 and ending at the address 0x0F as shown below. Once a valid address is produced for Memory Tx, the data is read from this address at the next positive clock edge. On the other hand, data is written to Memory Rx at the positive edge of the clock when a valid address is available. The operating clock frequency of Memory Tx is twice the clock frequency of Memory Rx.
(a) Assuming address generators for Memory Tx and Memory Rx start generating valid addresses at the same positive clock edge, show which data is actually stored in Memory Rx using a timing diagram. Indicate all the address and data values for Memory Tx and Memory Rx in the timing diagram.

(b) Now, assume that the operating clock frequency of Memory Tx is four times higher than the clock frequency of Memory Rx, and an actual write to Memory Rx takes place at the negative edge of the clock when a valid address is present. Redraw the timing diagram indicating all address and data values transferred from Memory Tx to Memory Rx.

8. Serial data is transferred to program four eight-bit registers. The start of the transfer is indicated by a seven-bit sequence = \{1010101\} immediately followed by the address of the register (two bits) and the data (eight bits). The transfer stops after programming the last register. After this point, all other incoming bits to the serial input are ignored. Design this interface by developing a data-path and a timing diagram simultaneously. Implement the state diagram. Can this controller be implemented by a counter-decoder scheme?
Projects

1. Implement the one-bit register in Fig. 2.14 and verify its functionality using Verilog. Use timing attributes in the flip-flop and the multiplexer to create gate propagation delays. Change the clock frequency until set-up time violation is produced.

2. Implement the four-bit shift register in Fig. 2.16 and verify its functionality using Verilog. Use timing attributes for flip-flops and the multiplexers to create gate propagation delays. Examine the resulting timing diagram.

3. Implement the 32-bit counter in Fig. 2.18 and verify its functionality using Verilog.

4. Implement the four-state Moore-type state machine in Fig. 2.20 and verify its functionality using Verilog.

5. Implement the four-state Mealy-type state machine in Fig. 2.27 and verify its functionality using Verilog.

6. Implement the three-bit counter-decoder in Fig. 2.36 and verify its functionality using Verilog and examining the resulting timing diagram.

7. Implement the 32x16 memory block in Fig. 2.37 using Verilog. How can this memory be verified functionally?

8. Implement the memory-to-memory transfer circuit in Fig. 2.45 and verify its functionality using Verilog.
Fundamentals of Computer Architecture and Design
Bindal, A.
2017, XIV, 533 p. 531 illus., Hardcover
ISBN: 978-3-319-25809-6