8.7 Synchronous Design Methodology

synchronous system

In a *synchronous system*, all flip-flops are clocked by the same, common clock signal, and preset and clear inputs are not used, except for system initialization. Although it's true that all the world does not march to the tick of a common clock, within the confines of a digital system or subsystem we can make it so. When we interconnect digital systems or subsystems that use different clocks, we can usually identify a limited number of asynchronous signals that need special treatment, as we'll show later, in Section 8.8.3.

Races and hazards are not a problem in synchronous systems, for two reasons. First, the only fundamental-mode circuits that might be subject to races or essential hazards are predesigned elements, such as discrete flip-flops or ASIC cells, that are guaranteed by the manufacturer to work properly. Second, even though the combinational circuits that drive flip-flop control inputs may contain static or dynamic or function hazards, these hazards have no effect, since the control inputs are sampled only *after* the hazard-induced glitches have had a chance to settle out.

Aside from designing the functional behavior of each state machine, the designer of a practical synchronous system or subsystem must perform just three well-defined tasks to ensure reliable system operation:

1. Minimize and determine the amount of clock skew in the system, as discussed in Section 8.8.1.
2. Ensure that flip-flops have positive setup- and hold-time margins, including an allowance for clock skew, as described in Section 8.1.4.
3. Identify asynchronous inputs, synchronize them with the clock, and ensure that the synchronizers have an adequately low probability of failure, as described in Sections 8.8.3 and 8.9.

Before we get into these issues, in this section we'll look at a general model for synchronous system structure and an example.

8.7.1 Synchronous System Structure

The sequential-circuit design examples that we gave in Chapter 7 were mostly individual state machines with a small number of states. If a sequential circuit has more than a few flip-flops, then it's not desirable (and often not possible) to treat the circuit as a single, monolithic state machine, because the number of states would be too large to handle.

Fortunately, most digital systems or subsystems can be partitioned into two or more parts. Whether the system processes numbers, digitized voice signals, or a stream of spark-plug pulses, a certain part of the system, which we'll call the *data unit*, can be viewed as storing, routing, combining, and generally processing "data." Another part, which we'll call the *control unit*, can be viewed as starting and stopping actions in the data unit, testing conditions, and deciding

data unit
control unit

what to do next according to circumstances. In general, only the control unit must be designed as a state machine. The data unit and its components are typically handled at a higher level of abstraction, such as:

- *Registers*. A collection of flip-flops is loaded in parallel with many bits of "data," which can then be used or retrieved together.
- *Specialized functions*. These include multibit counters and shift registers, which increment or shift their contents on command.
- *Read/write memory*. Individual latches or flip-flops in a collection of the same can be written or read out.

The first two topics above were discussed earlier in this chapter, and the last is discussed in Chapter 11.

Figure 8-79 is a general block diagram of a system with a control unit and a data unit. We have also included explicit blocks for input and output, but we could have just as easily absorbed these functions into the data unit itself. The control unit is a state machine whose inputs include *command inputs* that indicate how the machine is to function, and *condition inputs* provided by the data unit. The command inputs may be supplied by another subsystem or by a user to set the general operating mode of the control state machine (RUN/HALT, NORMAL/TURBO, etc.), while the condition inputs allow the control state machine unit to change its behavior as required by circumstances in the data unit (ZERO_DETECT, MEMORY_FULL, etc.).

command input
condition input

A key characteristic of the structure in Figure 8-79 is that the control, data, input, and output units all use the same common clock. Figure 8-80 illustrates the operations of the control and data units during a typical clock cycle:

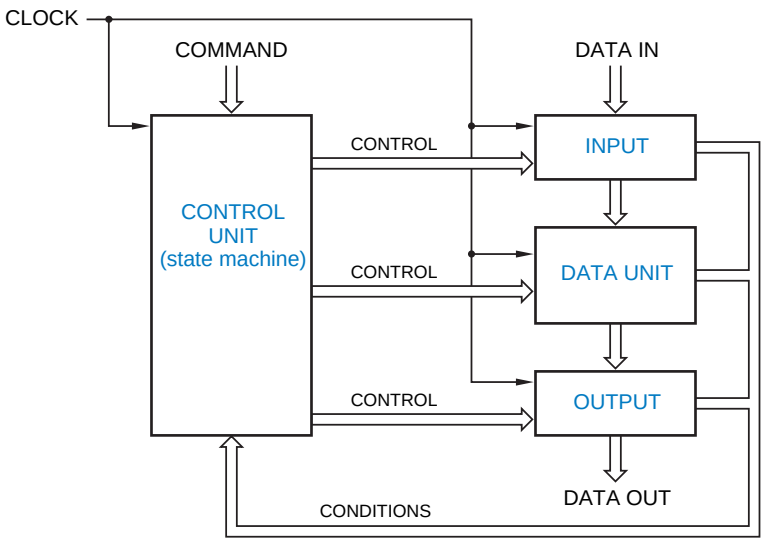

Figure 8-79
Synchronous system structure.

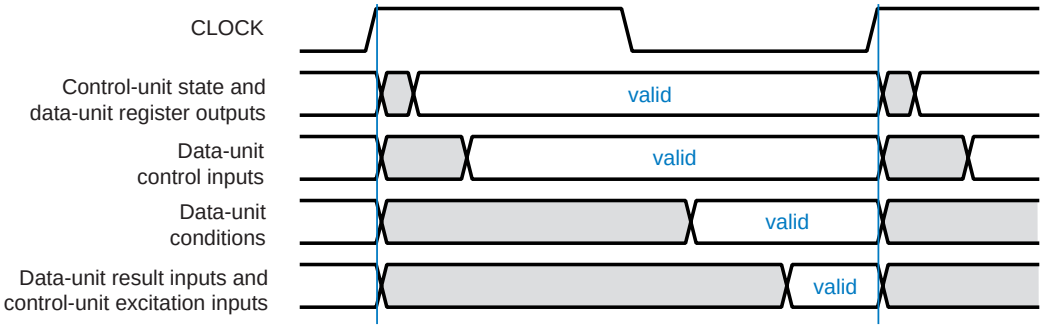

Figure 8-80 Operations during one clock cycle in a synchronous system.

1. Shortly after the beginning of the clock period, the control-unit state and the data-unit register outputs are valid.
2. Next, after a combinational logic delay, Moore-type outputs of the control-unit state machine become valid. These signals are control *inputs* to the data unit. They determine what data-unit functions are performed in the rest of the clock period, for example, selecting memory addresses, multiplexer paths, and arithmetic operations.
3. Near the end of the clock period, data-unit condition outputs such as zero- or overflow-detect are valid, and are made available to the control unit.
4. At the end of the clock period, just before the setup-time window begins, the next-state logic of the control-unit state machine has determined the next state based on the current state and command and condition inputs. At about the same time, computational results in the data unit are available to be loaded into data-unit registers.
5. After the clock edge, the whole cycle may repeat.

Data-unit control inputs, which are control-unit state-machine outputs, may be of the Moore, Mealy, or pipelined Mealy type; timing for the Moore type was shown in Figure 8-80. Moore-type and pipelined-Mealy-type outputs control the data unit's actions strictly according to the current state and past inputs, which do not depend on *current* conditions in the data unit. In contrast, Mealy-type outputs may select different actions in the data unit according to *current* conditions in the data unit. This increases flexibility, but typically also

PIPELINED MEALY OUTPUTS

Some state machines have pipelined Mealy outputs, discussed in Section 7.3.2. In Figure 8-80, pipelined Mealy outputs would typically be valid early in the cycle, at the same time as control-unit state outputs. Early validity of these outputs, compared to Moore outputs that must go through a combinational logic delay, may allow the entire system to operate at a faster clock rate.

increases the minimum clock period for correct system operation, since the delay path may be much longer. Also, Mealy-type outputs must not create feedback loops. For example, a signal that adds 1 to an adder's input if the adder output is nonzero causes an oscillation if the adder output is –1.

8.7.2 A Synchronous System Design Example

To give you an overview of several elements of synchronous system design, this subsection presents a representative example of a synchronous system. The example is a *shift-and-add multiplier* for unsigned integers using the algorithm of Section 2.8. Its data unit uses standard combinational and sequential building blocks, and its control unit is described by a state diagram

shift-and-add multiplier

Figure 8-81 illustrates data-unit registers and functions that are used to perform an 8-bit multiplication:

MPY/LPROD A shift register that initially stores the multiplier, and accumulates the low-order bits of the product as the algorithm is executed.

HPROD A register that is initially cleared, and accumulates the high-order bits of the product as the algorithm is executed.

MCND A register that stores the multiplicand throughout the algorithm.

F A combinational function equal to the 9-bit sum of HPROD and MCND if the low-order bit of MPY/LPROD is 1, and equal to HPROD (extended to 9 bits) otherwise.

The MPY/LPROD shift register serves a dual purpose, holding both yet-to-be-tested multiplier bits (on the right) and unchanging product bits (on the left) as the algorithm is executed. At each step it shifts right one bit, discarding the multiplier bit that was just tested, moving the next multiplier bit to be tested to the rightmost position, and loading into the leftmost position one more product bit that will not change for the rest of the algorithm.

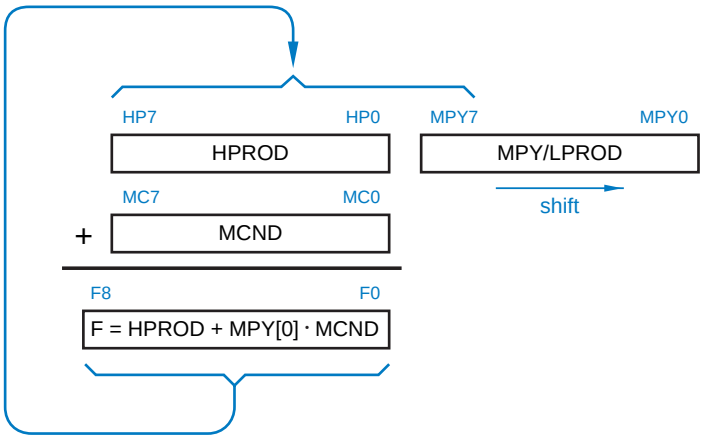

Figure 8-81
Registers and functions used by the shift-and-add multiplication algorithm.

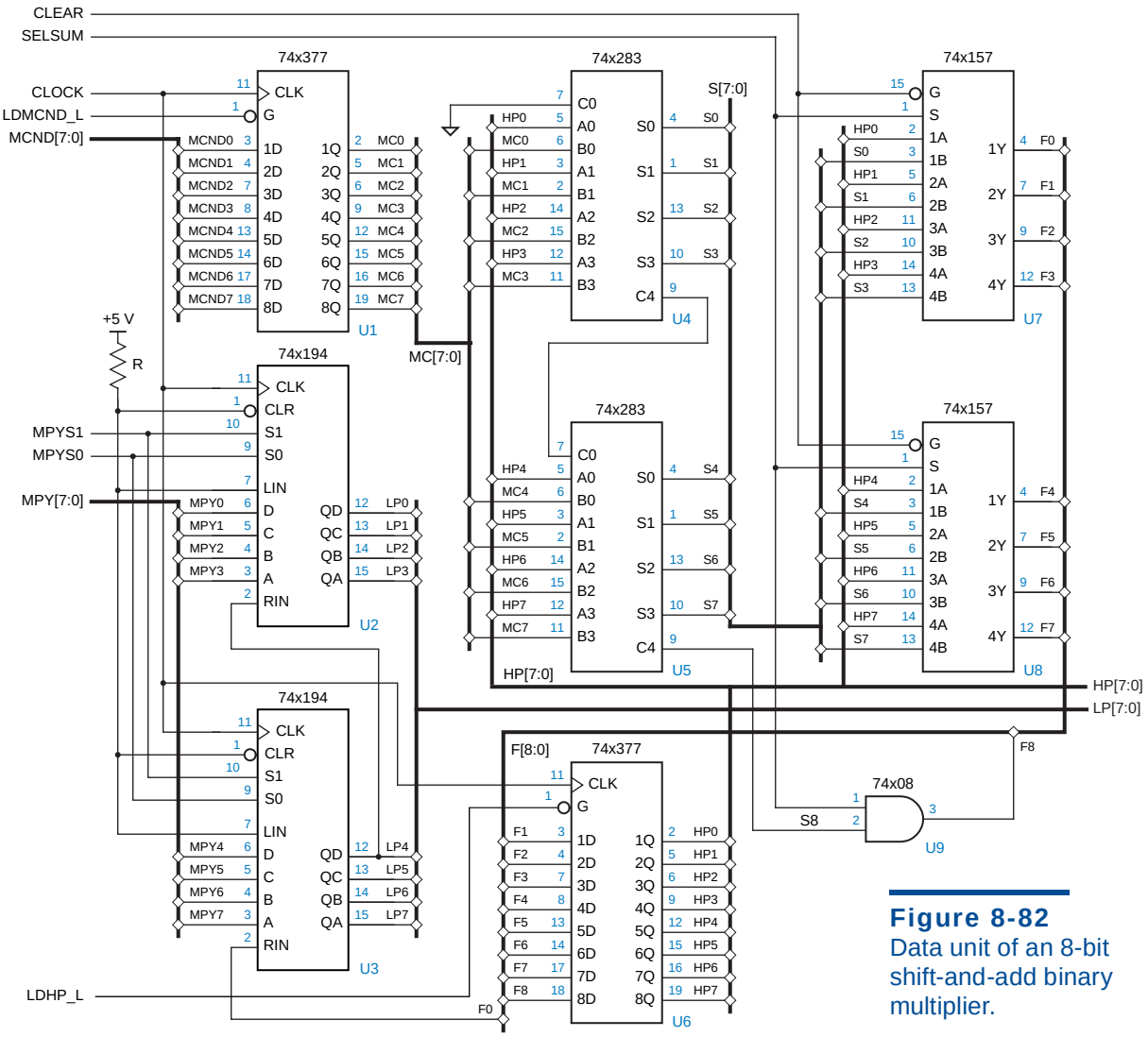

Figure 8-82
Data unit of an 8-bit shift-and-add binary multiplier.

Figure 8-82 is an MSI design for the data unit. The multiplier, MPY[7:0], and the multiplicand, MCND[7:0], are loaded into two registers before a multiplication begins. When the multiplication is completed, the product appears on HP[7:0] and LP[7:0]. The data unit uses the following control signals:

LDMCND_L When asserted, enables the multiplicand register U1 to be loaded.

LDHP_L When asserted, enables the HPROD register U6 to be loaded.

MPYS[1:0] When 11, these signals enable the MPY/LPROD register U2 and U3 to be loaded at the next clock tick. They are set to 01 during the multiplication operation to enable the register to shift right, and are 00 at other times to preserve the register's contents.

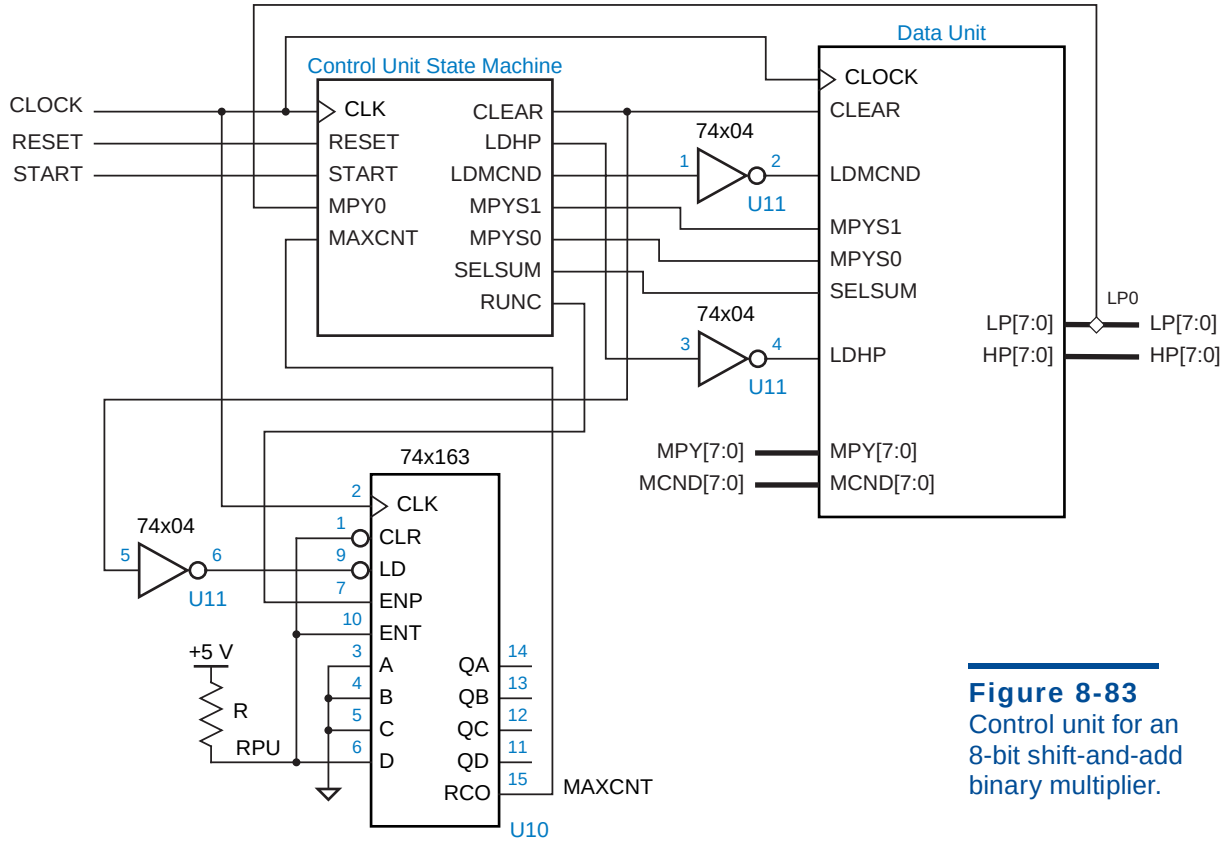

Figure 8-83
Control unit for an 8-bit shift-and-add binary multiplier.

SELSUM When this is asserted, the multiplexers U7 and U8 select the output of the adders U4 and U5, which is the sum of HPROD and the multiplicand MC. Otherwise, they select HPROD directly.

CLEAR When asserted, the output of multiplexers U7 and U8 is zero.

The multiplier uses a control unit, shown along with the data-unit block in Figure 8-83, to initialize the data unit and step through a multiplication. The control unit is decomposed into a counter (U10) and a state machine with the state diagram shown in Figure 8-84.

The state machine has the following inputs and outputs:

RESET A reset input that is asserted at power-up.

START An external command input that starts a multiplication.

MPY0 A condition input from the data unit, the next multiplier bit to test.

CLEAR A control output that zeroes the multiplexer output and initializes the counter.

LDMCND A control output that enables the MCND register to be loaded.

LDHP A control output that enables the HPROD register to be loaded.

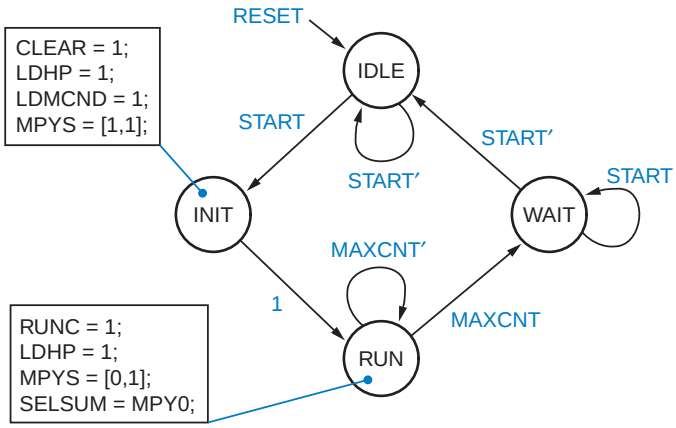

Figure 8-84
State diagram for the control state machine for a shift-and-add binary multiplier.

RUNC A control output that enables the counter to count.

MPYS[1:0] Control outputs for MPY/LPROD shifting and loading.

SELSUM A control output that selects between the shifted adder output or shifted HPROD to be loaded back into HPROD.

The state diagram can be converted into a corresponding state machine using any of a variety of methods, from turn-the-crank (a.k.a. hand-crafted) design to automatic synthesis using a corresponding ABEL or VHDL description. The state machine has mostly Moore-type outputs; SELSUM is a Mealy-type output. Two boxes in the state diagram list outputs that are asserted in the INIT and RUN states; all ouptuts are negated at other times. The machine is designed so that asserting RESET in any state takes it to the IDLE state.

After the START signal is asserted, a multiplication begins in the INIT state. In this state, the counter is initialized to 1000_2, the multiplier and multiplicand are loaded into their respective registers, and HPROD is cleared. The RUN state is entered next, and the counter is enabled to count. The state machine stays in the RUN state for eight clock ticks, to execute the eight steps of the 8-bit shift-and-add algorithm. During the eighth tick, the counter is in state 1111_2, so MAXCNT is asserted and the state machine goes to the WAIT state. The machine waits there until START is negated, to prevent a multiplication from restarting until START is asserted once again.

The design details of the data and control units are interesting, but the most important thing to see in this example is that all of the sequential circuit elements for both data and control are edge-triggered flip-flops clocked by the same, common CLOCK signal. Thus, its timing is consistent with the model in Figure 8-80, and the designer need not be concerned about races, hazards, and asynchronous operations. Unless the state machine realization is very slow, the overall circuit's maximum clock speed will be limited mainly by the propagation delays through the data unit.