XSbb: Sequential Building-Block Examples

Just about every real digital system is a sequential circuit—all it takes is one feedback loop, latch, or flip-flop to make a circuit's present behavior depend on its past inputs. However, if we were to design or analyze a typical digital system as a single sequential circuit, we would end up with an enormous state table. For example, strictly speaking, a PC with only 16 Mbytes of main memory is a sequential circuit with well over $2^{128,000,000}$ states!

We noted in other chapters that we typically deal with digital systems in smaller chunks, for example, partitioning them into data paths, registers, and control units (Section 8.7.1). In fact, a typical system has multiple functional units with well-defined interfaces and connections between them (as supported by VHDL and Verilog), and each functional unit may contain a hierarchy with several layers of abstraction (as supported by both HDLs and typical schematic drawing packages). Thus, we can deal with sequential circuits in much smaller chunks.

After all of this build-up, I have to admit that the design of complex, hierarchical digital systems is beyond the scope of this text. However, the heart of any system or any of its subsystems is typically a state machine or other sequential circuit, and that's something we *can* study here. So, this and other sequential-circuit example sections will try to reinforce your understanding of sequential circuit and state-machine design.

Early digital designers and many designers through the 1980s wrote out state tables by hand and built corresponding circuits using the synthesis methods that we described in Section 7.4. However, hardly anyone does that anymore. Nowadays, most state tables are specified with a hardware description language (HDL) such as ABEL, VHDL, or Verilog. The HDL compiler then performs the equivalent of our synthesis methods and realizes the specified machine in a PLD, CPLD, FPGA, ASIC, or other target technology.

This section gives a few sequential-circuit examples using MSI building blocks. Three other sequential-example sections use HDLs ABEL, VHDL, and Verilog, respectively.

XSbb.1 The World's Biggest Shift-Register Application

The most common application of shift registers is to convert parallel data into serial format for transmission or storage, and to convert serial data back to parallel format for processing or display (see Section 2.16.1). The most common example of serial data transmission, one that you almost certainly take part in every day, is in *digital telephony.*

digital telephony

For years, TPCs (The Phone Companies) have been installing digital switching equipment in their central offices (*COs*). Most home phones have a two-wire analog connection to the central office. However, an analog-to-digital converter samples the analog voice signal 8,000 times per second (once every

CO

125 μs) when it enters the CO and produces a corresponding sequence of 8,000 8-bit bytes representing the sign and amplitude of the analog signal at each sampling point. Subsequently, your voice is transmitted digitally on 64-Kbps *serial channels* throughout the phone network, until it is converted back to an analog signal by a digital-to-analog converter at the far-end CO.

serial channel

The 64-Kbps bandwidth required by a single digital voice signal is far *less* than can be obtained on a single digital signal line or switched by digital ICs. Therefore most digital telephone equipment *multiplexes* many 64-Kbps channels onto a single wire, saving both wires and digital ICs for switching. In the next subsection, we show how 32 channels can be processed by a handful of MSI chips; and these chips could be easily integrated into a single CPLD. This is a classic example of a *space/time trade-off* in digital design—by running the chips faster, you can accomplish a larger task with fewer chips. Indeed, this is the main reason that the telephone network has "gone digital."

multiplex

space/time trade-off

XSbb.2 Serial/Parallel Conversion

A typical application of serial data transfer between two modules (possibly in a piece of CO switching equipment) is shown in Figure XSbb-1. Three signals are normally connected between a source module and a destination module to accomplish the transfer:

- *Clock.* The clock signal provides the timing reference for transfers, defining the time to transfer one bit. In systems with just two modules, the clock may be part of control circuits located on the source module as shown. In larger systems, the clock may be generated at a common point and distributed to all of the modules.
- *Serial data.* The data itself is transmitted on a single line.
- *Synchronization.* A *synchronization pulse* (or *sync pulse*) provides a reference point for defining the data format, such as the beginning of a byte or word in the serial data stream. Some systems omit this signal and instead use a unique pattern on the serial data line for synchronization.

synchronization pulse
sync pulse

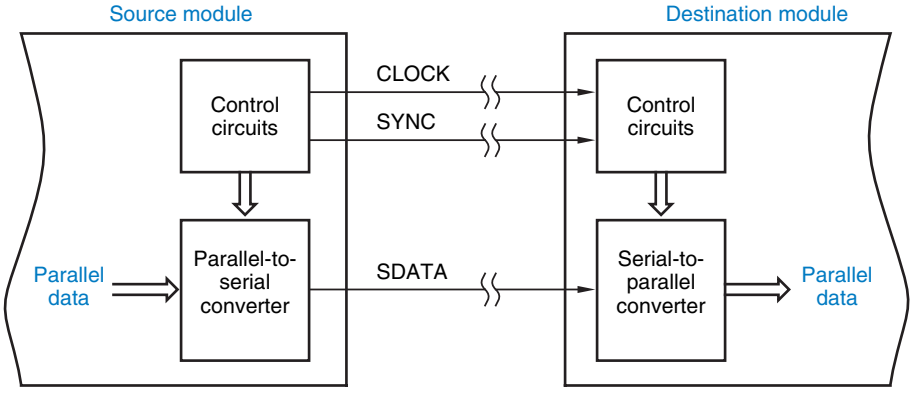


Figure XSbb-1 A system that transmits data serially between modules.

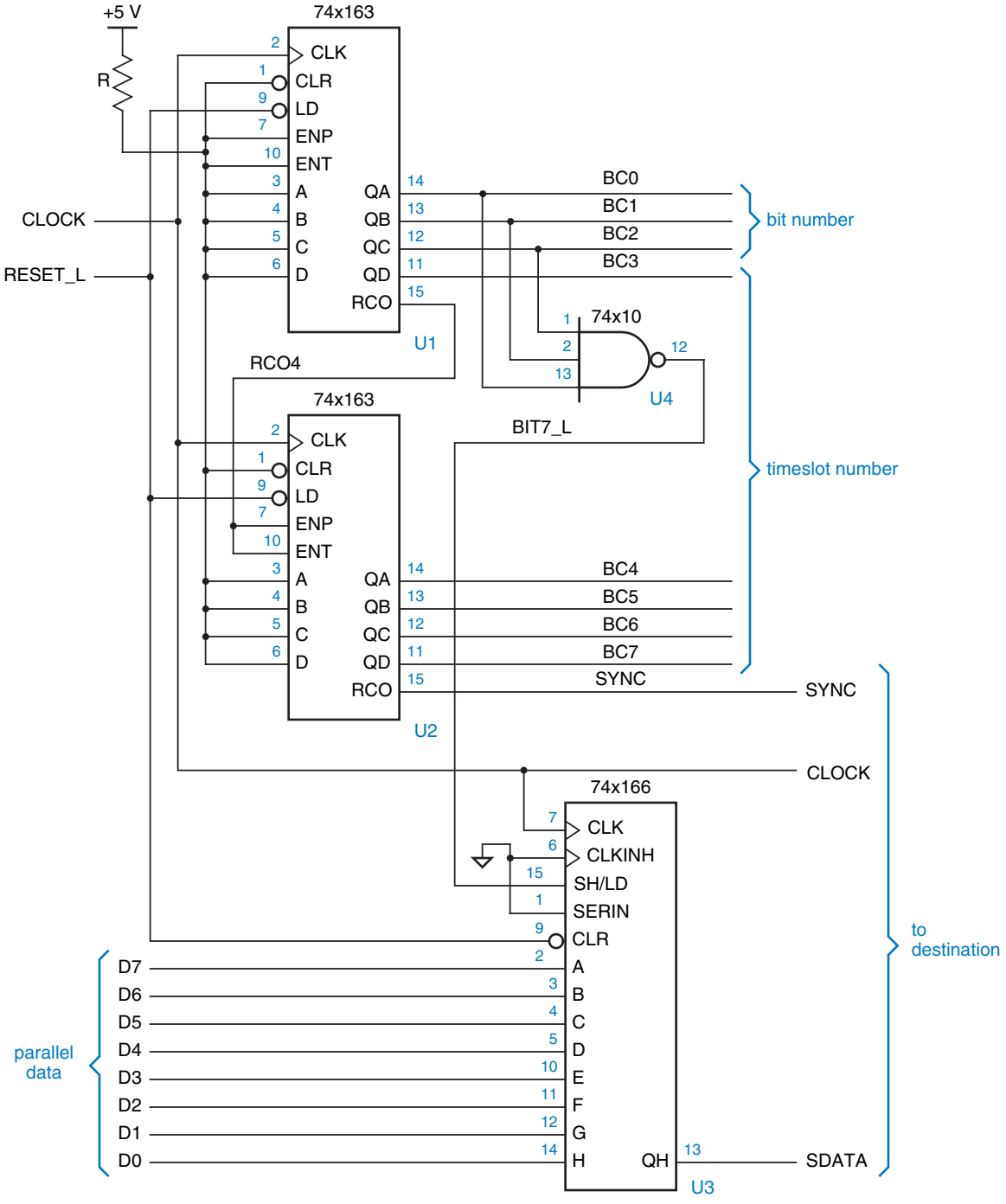


Figure XSbb-2 Parallel-to-serial conversion using a parallel-in shift register.

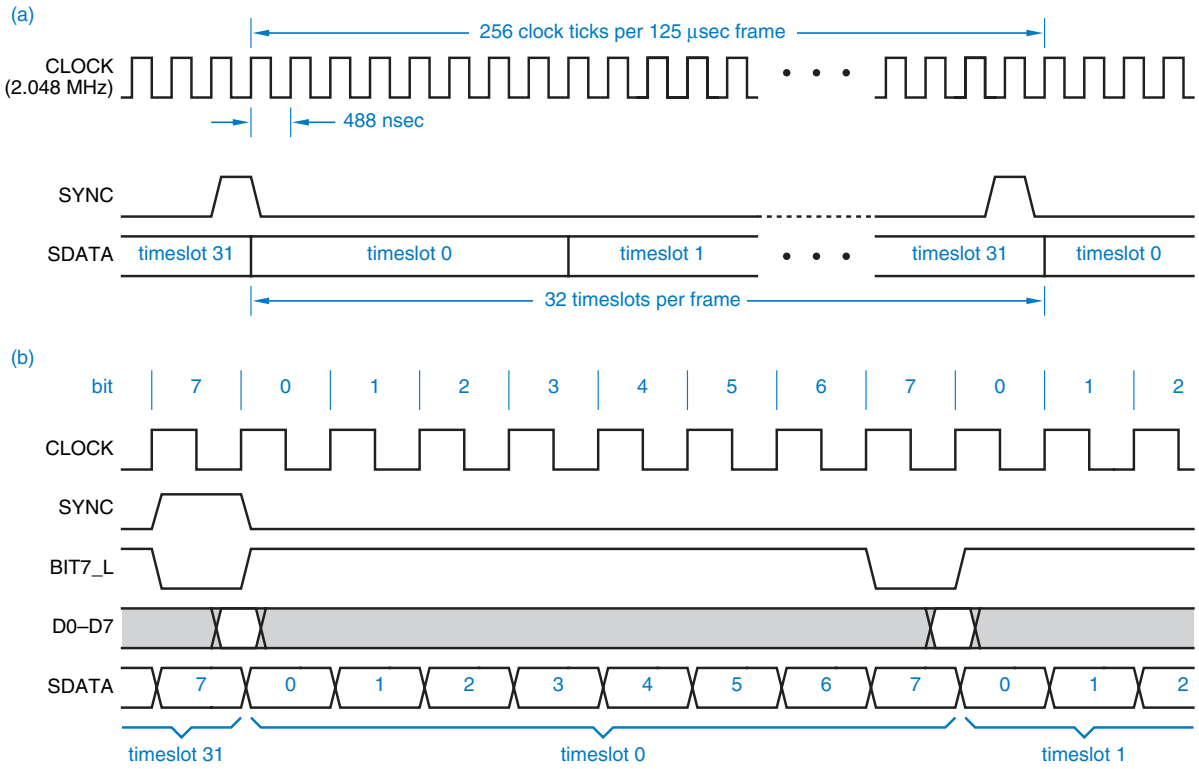


Figure XSbb-3 Timing diagram for parallel-to-serial conversion: (a) a complete frame; (b) one byte at beginning of frame.

The general timing characteristics of these signals in a typical digital telephony application are shown in Figure XSbb-3(a). The CLOCK signal has a frequency of 2.048 MHz to allow the transmission of $32 \times 8{,}000$ 8-bit bytes per second. The 1-bit-wide pulse on the SYNC signal identifies the beginning of a 125-μs interval called a *frame*. A total of 256 bits are transmitted on SDATA during this interval, which is divided into 32 *timeslots* containing eight bits each. Each timeslot carries one digitally encoded voice signal. Both timeslot numbers and bit positions within a timeslot are located relative to the SYNC pulse.

frame
timeslot

Figure XSbb-2 shows a circuit that converts parallel data to the serial format of Figure XSbb-3(a), with detailed timing shown in (b). Two 74x163

WHICH BIT FIRST? Most real serial links for digitized voice actually transmit bit 7 first, because this is the first bit generated by the analog-to-digital converter that digitizes the voice signal. However, to simplify our examples, we transmit bit 0 first so that counter state equals bit number.

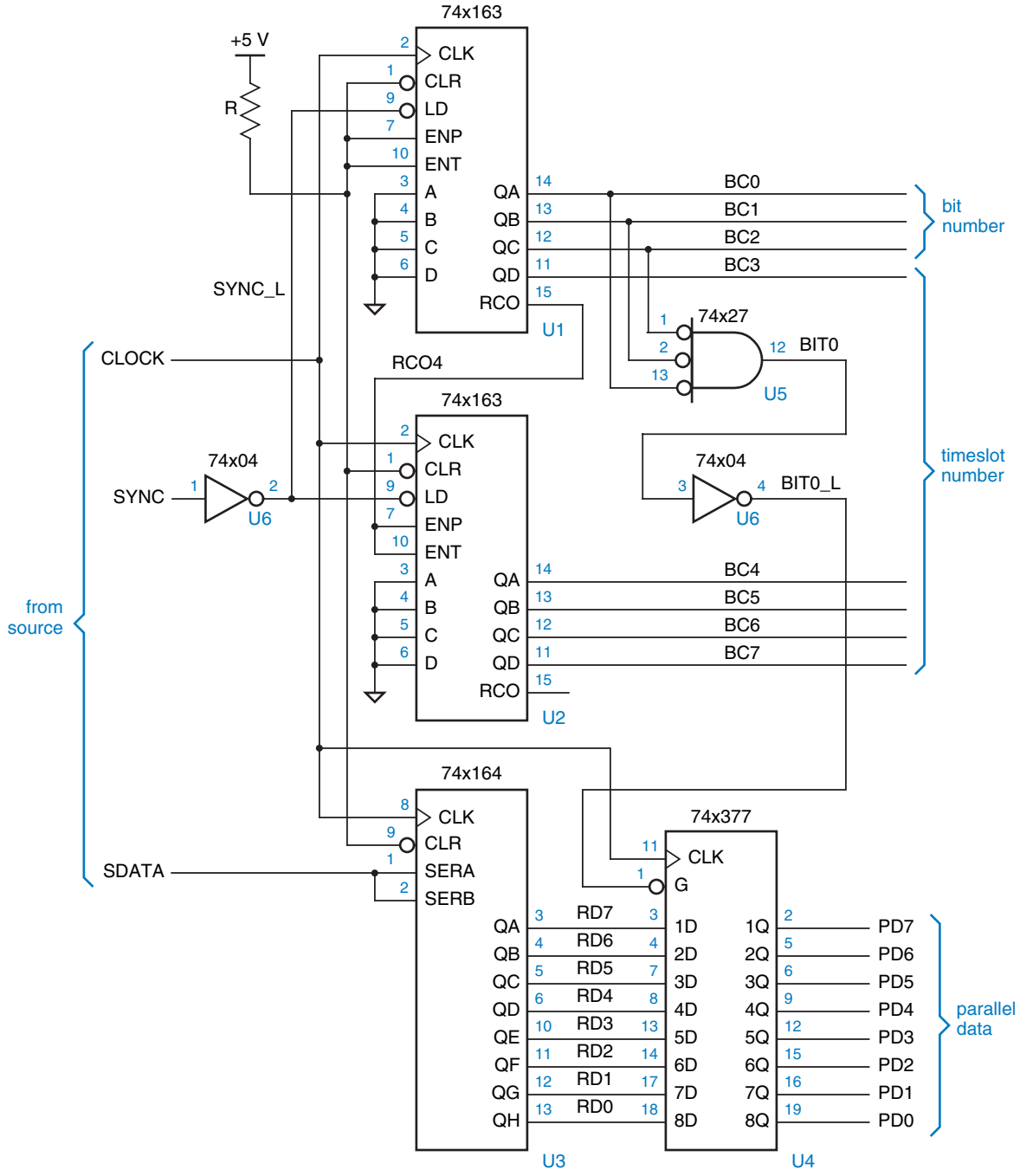


Figure XSbb-4 Serial-to-parallel conversion using a parallel-out shift register.

THE NATION'S CLOCK

Believe it or not, in the phone network, a very precise 8-KHz clock is generated in St. Louis and distributed throughout the U.S.! The clock signal that is distributed in a particular piece of local CO equipment is normally derived from the national clock. For example, the 2.048-MHz clock in this section's example could be derived by a phase-locked loop circuit that multiplies the national clock frequency by 256.

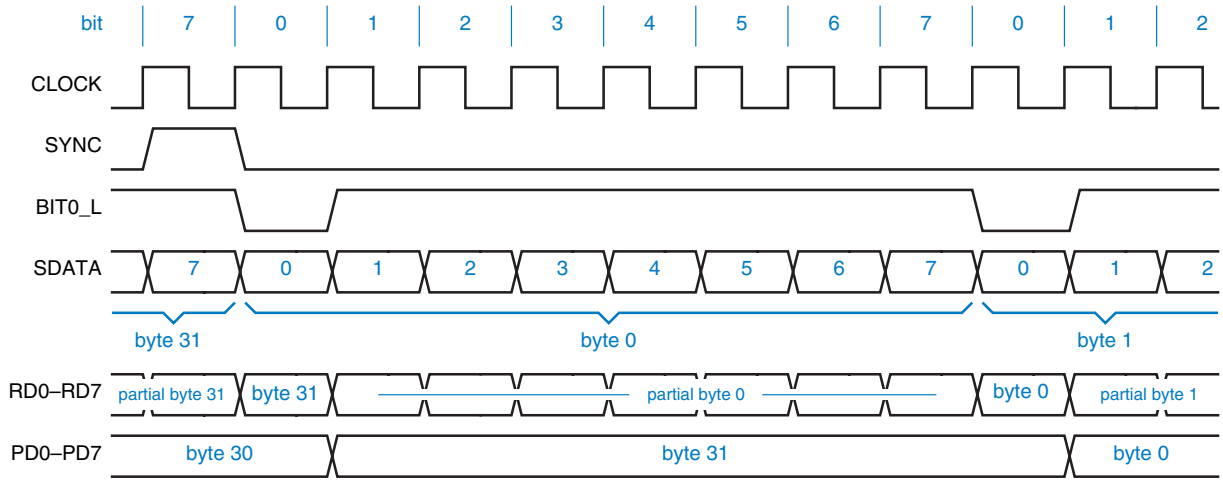
counters are wired as a free-running modulo-256 counter to define the frame. The five high-order and three low-order counter bits are the timeslot number and bit number, respectively.

A 74x166 parallel-in shift register performs the parallel-to-serial conversion. Bit 0 of the parallel data (D0–D7) is connected to the '166 input closest to the SDATA output, so bits are transmitted serially in the order 0 through 7.

During bit 7 of each timeslot, the BIT7_L signal is asserted, which causes the '166 to be loaded with D0–D7. The value of D0–D7 is irrelevant except during the setup- and hold-time window around the clock edge on which the '166 is loaded, as shown by shading in the timing diagram. This leaves open the possibility that the parallel data bus could be used for other things at other times (see Exercise XSbb.2).

A destination module can convert the serial data back into parallel format using the circuit of Figure XSbb-4 on the preceding page. A modulo-256 counter built from a pair of '163s is used to reconstruct the timeslot and bit numbers. Although SYNC is asserted during state 255 of the counter on the source module, SYNC loads the destination module's counter with 0 so that both counters go to 0 on the same clock edge. The counter's high-order bits (timeslot number) are not used in the figure, but they may be used by other circuits in the

Figure XSbb-5 Timing diagram for serial-to-parallel conversion.



LITTLE ENDIANS AND BIG ENDIANS

At one point in the evolution of digital systems, the choice of which bit or byte to transmit first was a religious issue. In a famous article on the subject, "On Holy Wars and a Plea for Peace" (*Computer*, October 1981, pp. 48–54), Danny Cohen described the differences in bit- and byte-ordering conventions and the havoc that could be (and now has been) wrought as a result.

A firm standard was never established, so that today there are some popular computer systems (such as IBM-compatible PCs) that transmit or number the low-order byte of a 32-bit word first, and others (such as Apple Macintosh computers) that transmit or number the high-order byte first. Following Cohen's nomenclature, people refer to these conventions as "Little Endian" and "Big Endian," respectively, and talk about "endianness" as if it were actually a word.

destination module to identify the byte from a particular timeslot on the parallel data bus (PD0–PD7).

Figure XSbb-5 shows detailed timing for the serial-to-parallel conversion circuit. A complete received byte is available at parallel output of the 74x164 shift register during the clock period following the reception of the last bit (7) of the byte. The parallel data in this example is *double-buffered*—once it is fully received, it is transferred into a 74x377 register, where it is available on PD0-PD7 for eight full clock periods until the next byte is fully received. The BIT0_L signal enables the '377 to be loaded at the proper time. More registers and decoding could be provided to load the byte from each timeslot into a different register, making each byte available for 125 μs (see Exercise XSbb.4).

double-buffered data

Once the received data is in parallel format, it can easily be stored or modified by other digital circuits; we'll give examples in Section 9.1.6. In digital telephony, the received parallel data is converted back into an analog voltage that is filtered and transmitted to an earpiece or speaker for 125 μs, until the next voice sample arrives.

I STILL DON'T KNOW

ISDN (Integrated Services Digital Network) technology was developed in the late 1980s to extend full-duplex 144-Kbps serial digital channels to home phones. The idea was to carry two 64-Kbps voice conversations plus a 16-Kbps control channel on a single pair of wires, thus increasing the capacity of installed wiring.

In the first edition of this book, we noted that delays in ISDN deployment had led some people in the industry to rename it "Imaginary Services Delivered Nowhere." In the mid-1990s, ISDN finally took off in the U.S., but it was deployed not so much to carry voice as to provide "high-speed" connections to the Internet.

Unfortunately for TPCs, the growth in ISDN was cut short first by the deployment of inexpensive 56-Kbps analog modems and later by the growing availability of very high-speed connections (160 Kbps to 2 Mbps or higher) using newer DSL (Digital Subscriber Line) and cable-modem technologies.

XSbb.3 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 XSbb-6 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 XSbb-7 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.

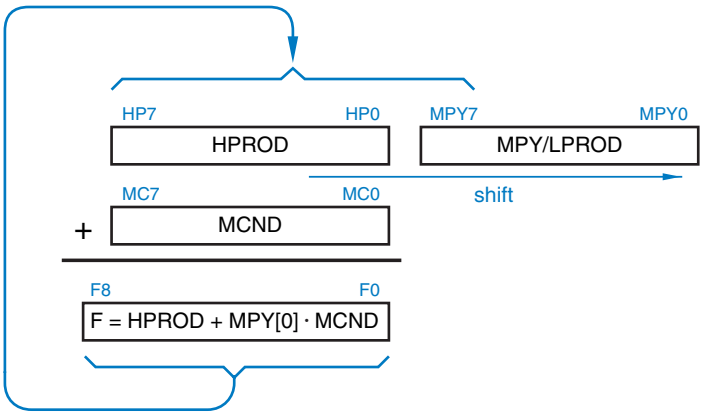


Figure XSbb-6 Registers and functions used by the shift-and-add multiplication algorithm.

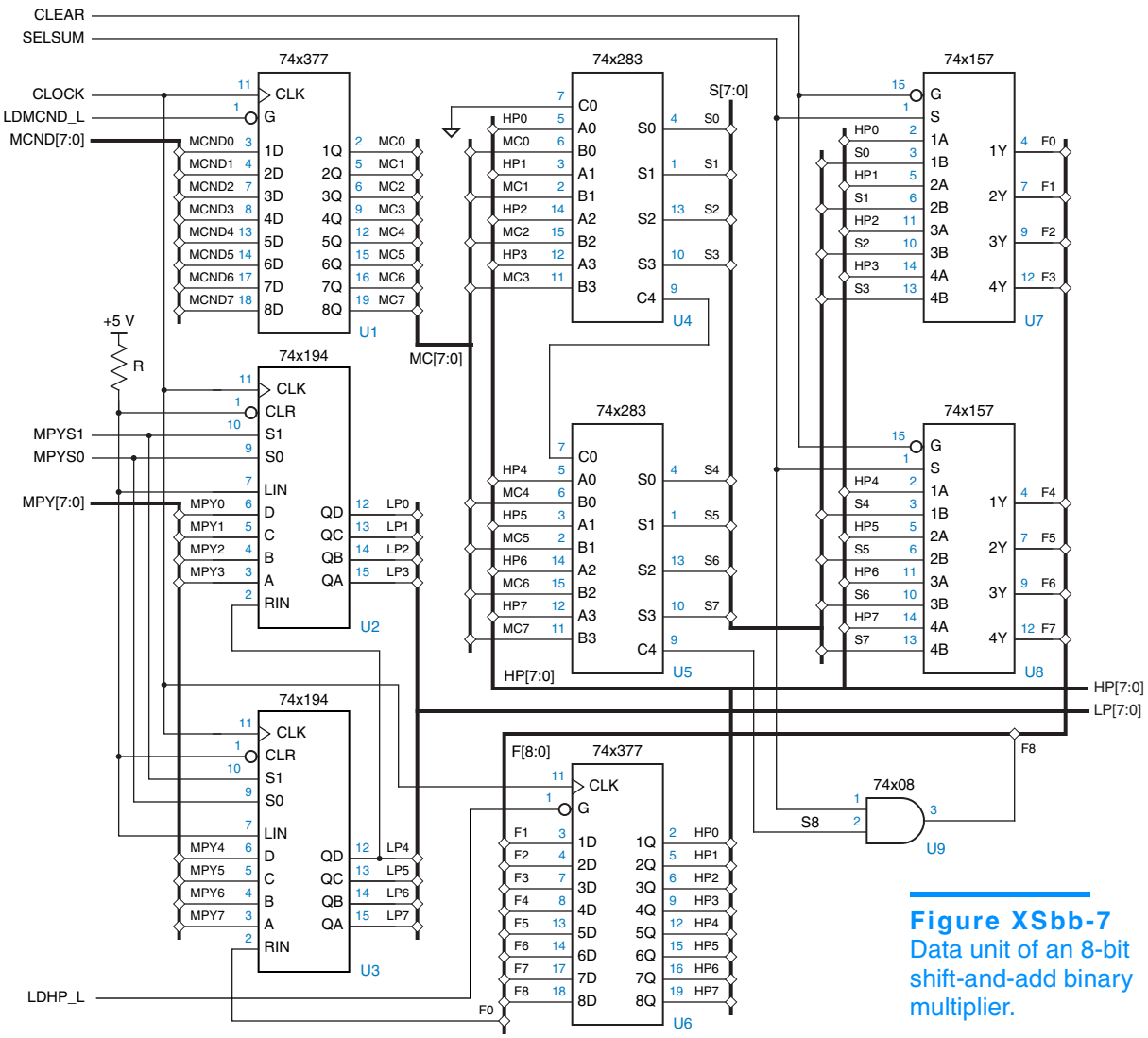


Figure XSbb-7
Data unit of an 8-bit shift-and-add binary multiplier.

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 they are 00 at other times to preserve the register's contents.

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.

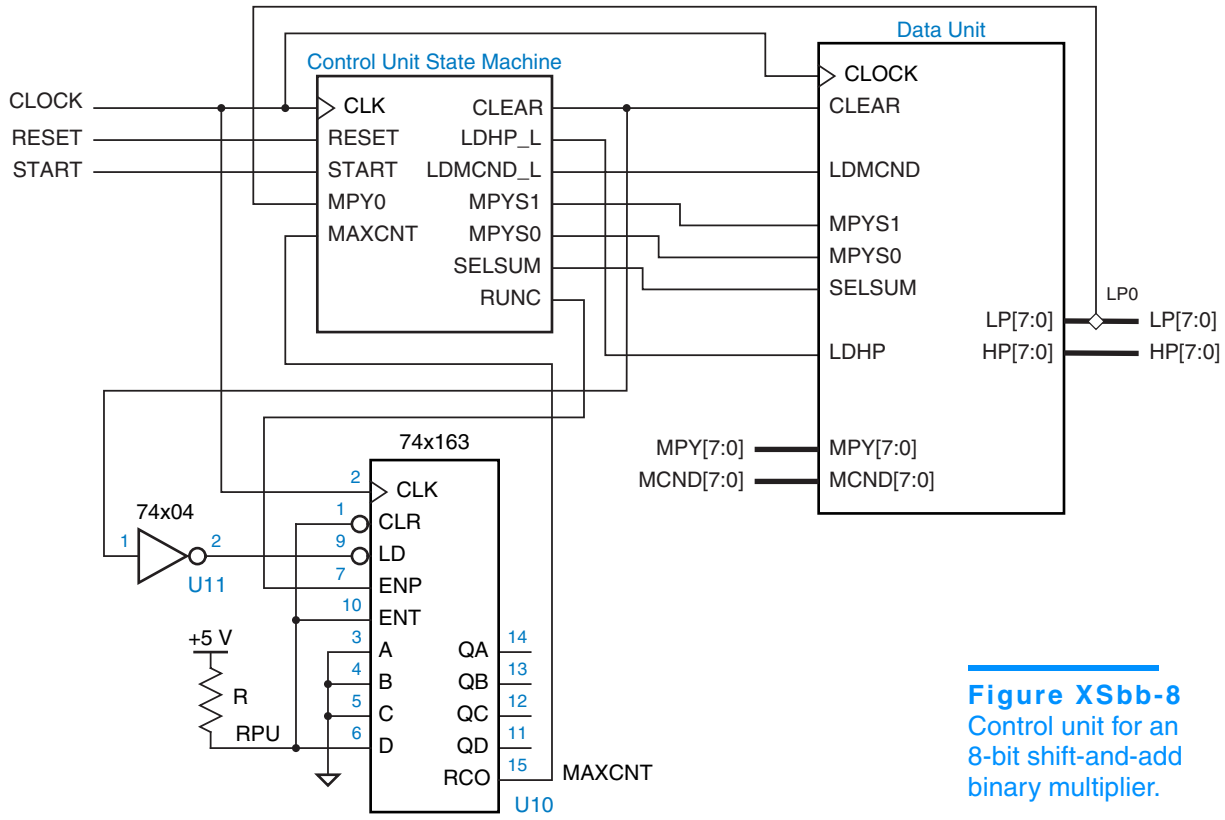


Figure XSbb-8
Control unit for an 8-bit shift-and-add binary multiplier.

The multiplier uses a control unit, shown along with the data-unit block in Figure XSbb-8, 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 XSbb-9.

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.

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.

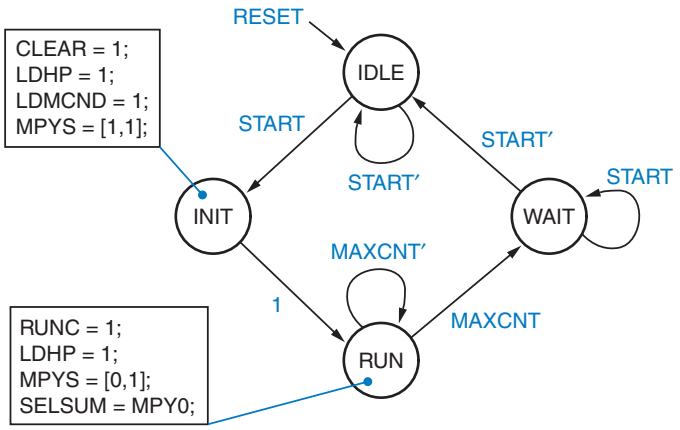


Figure XSbb-9
State diagram for the control state machine for a shift-and-add binary multiplier.

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 outputs 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 8-64 on page 760, 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.

Exercises

XSbb.1 Design a clocked synchronous state machine that checks for even parity on a serial byte-data line with timing similar to Figure XSbb-3. The circuit should have three inputs, RESET, SYNC, and DATA, in addition to CLOCK, and one Moore-type output, ERROR. The ERROR output should be asserted if any DATA byte received since reset had odd parity. Devise a state/output table that does the job using no more than four states and include a description of each state's meaning in the table. Choose a 2-bit state assignment, write transition and excitation equations for D flip-flops, and draw the logic diagram.

XSbb.2 Design a parallel-to-serial conversion circuit with eight 2.048-Mbps, 32-channel serial links and a single 2.048-MHz, 8-bit, parallel data bus that carries 256 bytes per frame. Each serial link should have the frame format defined in Figure XSbb-3. Each serial data line SDATAi should have its own sync signal SYNCi; the sync pulses should be staggered so that SYNCi + 1 has a pulse one tick after SYNCi. Show the timing of the parallel bus and the serial links, and write a table or formula that shows which parallel-bus timeslots are transmitted on which serial links and timeslots. Draw a logic diagram for the circuit using MSI parts from Chapter 8; you may abbreviate repeated elements (e.g., shift registers), showing only the unique connections to each one.

XSbb.3 Repeat Exercise XSbb.2, assuming that all serial data lines reference their data to a single, common SYNC signal. How many more chips does this design require?

XSbb.4 Show how to enhance the serial-to-parallel circuit of Figure XSbb-4 so that the byte received in each timeslot is stored in its own register for 125 μs, until the next byte from that timeslot is received. Draw the counter and decoding logic for 32 timeslots in detail, as well as the parallel data registers and connections for timeslots 31, 0, and 1. Also draw a timing diagram in the style of Figure XSbb-5 that shows the decoding and data signals associated with timeslots 31, 0, and 1.

XSbb.5 The text stated that the designer need not worry about any timing problems in the synchronous design of Figure XSbb-8. Actually, there is one slight worry. Look at the timing specifications for the 74x377 and discuss.

XSbb.6 Determine the minimum clock period for the shift-and-add multiplier circuit in Figure XSbb-8, assuming that the state machine is realized in a single GAL16V8-20 and that the MSI parts are all 74LS TTL. Use worst-case timing information from the tables in this book. For the '194, t_{pd} from clock to any output is 26 ns, and t_s is 20 ns for serial and parallel data inputs and 30 ns for mode-control inputs.

XSbb.7 Design a data unit and a control-unit state machine for multiplying 8-bit two's-complement numbers using the algorithm discussed in Section 2.8.

XSbb.8 Design a data unit and control-unit state machine for dividing 8-bit unsigned numbers using the shift-and-subtract algorithm discussed in Section 2.9.

XSbb.9 Calculate the minimum clock period of the data unit in Figure XSbb-7. Use the maximum propagation delays given in Table 6-3 for LS-TTL combinational parts. Unless you can get the real numbers from a TTL data book, assume that all registers require a 10-ns minimum setup time on inputs and have a 20-ns propagation delay from clock to outputs. Indicate any assumptions you've made about delays in the control unit.

XSbb.10 Write an ABEL, VHDL, or Verilog program corresponding to the state diagram in Figure XSbb-9 for the multiplier control unit.