Dangers of Asynchronous Inputs The designs of Figure 7.41 and Figure 7.43 would not be possible if the clear and load inputs were asynchronous. We use external logic to recognize the offset or cutoff state, to assert a control input that leads to an out-of-sequence new state. In these synchronous designs, the state does not change until the next rising clock edge.

If the control inputs were asynchronous, the state change would have happened as soon as the control input was asserted, independent of the clock. This violates a fundamental assumption of synchronous systems—that the state changes at clocking events and not at other times.

Let's reexamine the counter of Figure 7.43, but this time implemented with the TTL 74161 counter. This device is identical to the 74163, except that the clear signal is asynchronous.

Compare the timing diagram of Figure 7.45 with that of Figure 7.44. The $\overline{\text{CLR}}$ signal is obviously a short-duration pulse, and state 1101 is held for a much shorter time than any other state. In effect, if we consider the counter to advance from one state to the next on every rising clock edge, then the behavior of Figure 7.45 actually shows the counter advancing from 1100 to 0000, passing through 1101 in an instant. This is obviously not the behavior we desired.

Asynchronous inputs should be used only for situations like power-on reset. Never use them to implement state transitions. As a designer, you should always be aware of the behavior of the catalog parts you select. Remember to choose the right parts for the job at hand.

7.6 Random-Access Memories

In this section, we will examine memory components in more detail. In particular, we will focus on two important aspects of memory system design: the detailed timing waveforms for a static RAM component, and the design of the register and control logic that surrounds the memory subsystem, making it possible to interface the memory to the rest of the digital system. But first, we must begin with the basics.

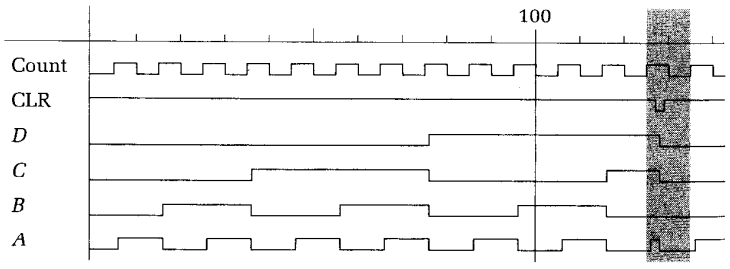


Figure 7.45 Sequence implemented with TTL 74161.

7.6.1 RAM Basics: A 1024 by Four-Bit Static RAM

We begin with a relatively simple memory component, a 1024 by 4-bit static RAM. The basic storage element of the static RAM is a six-transistor circuit, shown in Figure 7.46. The "static" storage element is provided by the cross-coupled inverters. This circuit configuration will hold a 1 or 0 as long as the system continues to receive power. There is no need for a periodic refreshing signal or a clock.

The nMOS transistors provide access to the storage element from two buses, denoted Data_j and $\overline{\text{Data}}_j$. To write the memory element, special circuitry in the RAM drives the data bit and its complement onto these lines while the word enable line is asserted. When driven in this fashion, the data bit can overwrite the previous state of the element.

To read the contents of the storage element, the word enable line is once again enabled. Instead of being driven onto the data lines (also called *bit lines*), the data are "sensed" by a different collection of special circuits. These circuits, called *sense amplifiers*, can detect small voltage differences between the data line and its complement. If Data_j is at higher voltage than $\overline{\text{Data}}_j$, the cell contained a logic 1. If the situation is reversed, the cell contained a logic 0.

RAMs are efficient in packing many bits into a circuit package for two reasons. First, only a small number of transistors are needed to implement the storage elements. And second, it is easy to arrange these elements into rows and columns. Each row of memory cells shares a common word enable line. Each column shares common bit lines. The number of columns determines the bit width of each word. Thus, you can find memory components that are 1, 4, or 8 bits wide and that read or write the bits of a single word in parallel.

Figure 7.47 shows the pin-out for our 1024 by 4-bit SRAM (static RAM). The pins can be characterized as address lines, data lines, and control lines.

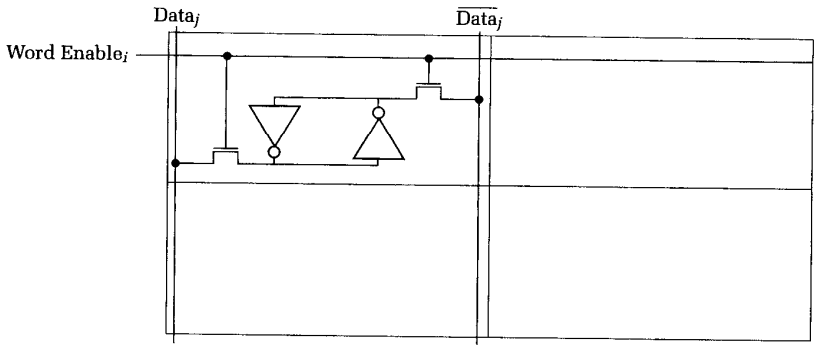


Figure 7.46 Static RAM storage elements.

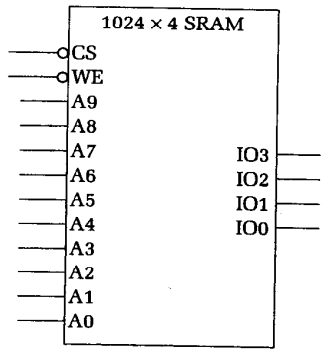


Figure 7.47 Schematic shape of the 1024 × 4-bit SRAM.

Since the RAM has 1024 words, there must be 10 lines to address th
Since each word is 4 bits wide, there are four data lines. The same pins
used for reading or writing and are called *bidirectional*. The value on
active low control signal Write Enable ($\overline{WE}$) determines their direction.

The final signal on the chip is the chip select control line ($\overline{CS}$). Wh
this signal goes low, a read or write cycle commences, depending on th
value of write enable. If write enable is also low, the data lines provide ne
values to be written into the addressed word within the RAM. If it is hig
the data lines are driven with the contents of the addressed word.

Internal Block Diagram From the preceding discussion, you might infe
that the RAM is organized as an array with 1024 words and four col
umns. In terms of performance and packaging, this is not the best inter-
nal organization. A long thin array leads to long wires, which take more
time to drive to a given logic voltage. Also, rectangular integrated cir-
cuits are more difficult to arrange on a silicon wafer for processing. A
square configuration is much more desirable.

Figure 7.48 gives a more realistic block diagram of the internal structure of a typical 1024 by 4-bit SRAM. The RAM array consists of four banks of

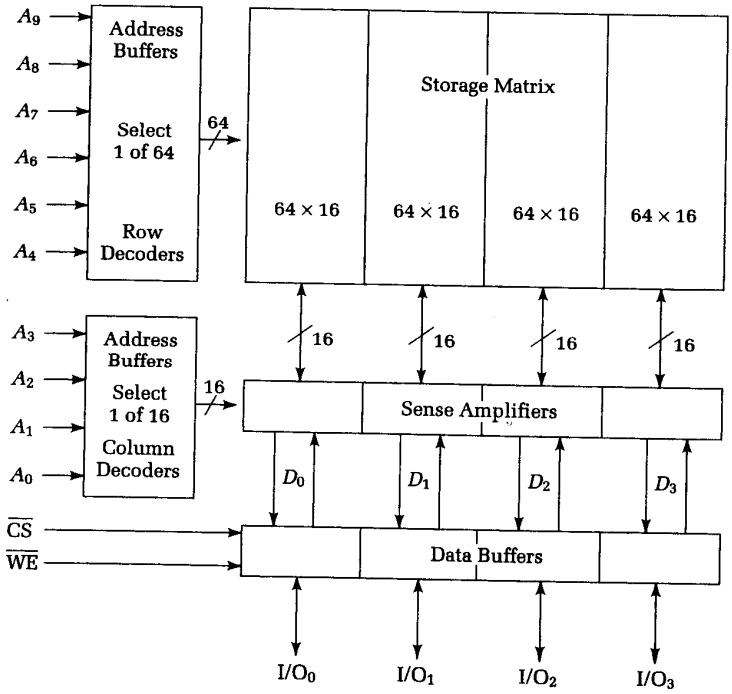


Figure 7.48 Internal block diagram of 1024 × 4-bit SRAM.

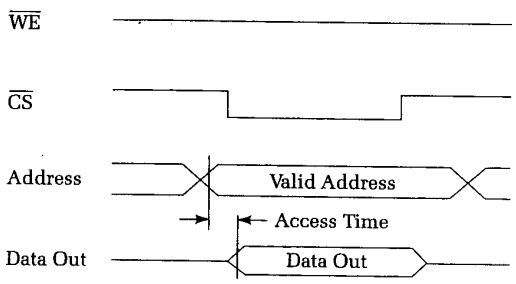


Figure 7.49 Read cycle logical sequencing.

64 words by 16 bits each. This makes the array square. Let's consider a read operation. The high-order 6 bits of the address select one of 64 words. Four groups of 16 bits each emerge from the storage array, one group for each of the possible data bits. The four low-order address bits select one of 16 bits from each of the four groups to form the 4-bit data word. Writes are similar, except with data flowing in the opposite direction.

This form of two-dimensional decode, with row and column decoders, is used universally in memory components. Not only does it keep the memory array square, it also limits the longest lines in the decoders.

Simplified Read Cycle and Write Cycle Timing Controlling the function of a RAM chip requires precise sequencing of the address pins and control signals. Figure 7.49 gives a simplified logic timing diagram for the RAM read cycle (we defer a more precise description of RAM timing to Section 7.6.2). First, a valid address must be set up on the address lines. Then the chip select ($\overline{CS}$) line is taken low while the write enable ($\overline{WE}$) stays high. The *memory access time* is the time it takes for new data to be ready to appear at the output. It is measured from the last change in the address lines, although the output is not visible off-chip unless the chip select is low. Once the chip select line goes high again, deselecting the chip, the output on the data lines will no longer be valid.

Figure 7.50 gives the write cycle sequencing. Because an erroneous write could have destructive consequences, we must be especially careful during the sequencing of the write signals. To be conservative, $\overline{WE}$ should be brought low and the address and data lines should be stable before $\overline{CS}$ goes low. A similar sequence occurs in reverse to end the write cycle.

While conceptually correct, this specification is more restrictive than it needs to be. Technically speaking, the write cycle begins when both $\overline{WE}$ and $\overline{CS}$ go low. It ends when $\overline{WE}$ goes high. The only absolute requirement is that the address is stable a setup time before both signals go low and satisfies a hold time constraint after the first one goes high. The data setup and hold times are also measured from the first control signal to rise.

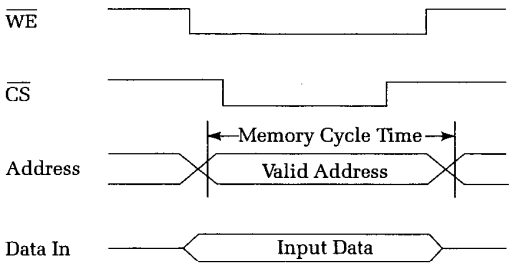


Figure 7.50 Write cycle logical sequencing.

Another important metric for RAMs is the *memory cycle time*. This is the time between subsequent memory operations. In general, the access time is less than or equal to the memory cycle time.

7.6.2 Dynamic RAM

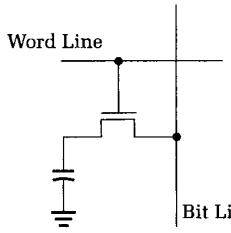


Figure 7.51 One-transistor memory cell.

Static RAMs are the fastest memories (and the easiest to interface with), but the densest memories are dynamic RAMs (DRAMs). Their high capacity is due to an extremely efficient memory element: the one-transistor (1-T) memory cell.

The 1-T memory cell, consisting of a single access transistor and a capacitor, works as follows (see Figure 7.51). The word line and bit line provide exactly the same function as in the SRAM. To write the memory cell, the bit line is charged to a logic 1 or 0 voltage while the word line is asserted. This enables the access transistor, making it possible to charge up the storage capacitor with the desired logic voltage.

The read operation takes place by asserting the word line. The access transistor is turned on, sharing the voltage on the capacitor with the bit line. Sensitive amplifier circuits detect small changes on the bit line to determine whether a 1 or 0 was stored in the selected memory element.

The destructiveness of the read operation makes DRAMs complex. To read the contents of the storage capacitor, we must discharge it across the bit line. Thus, external circuitry in the DRAM must buffer the values that have been read out and then write them right back.

The second problem with DRAMs, and the most significant one from your viewpoint as a system designer, is that their contents decay over time. Every once in a while (measured in milliseconds), the charge on the storage capacitors leaks off. To counteract this, the DRAM must be refreshed. Periodically, the memory elements must be read and written back to their storage locations.

To make this operation reasonably efficient, the DRAM's memory array is a two-dimensional matrix organized along the lines of the SRAM block diagram of Figure 7.48. Figure 7.52 shows the block diagram of a 4096 by

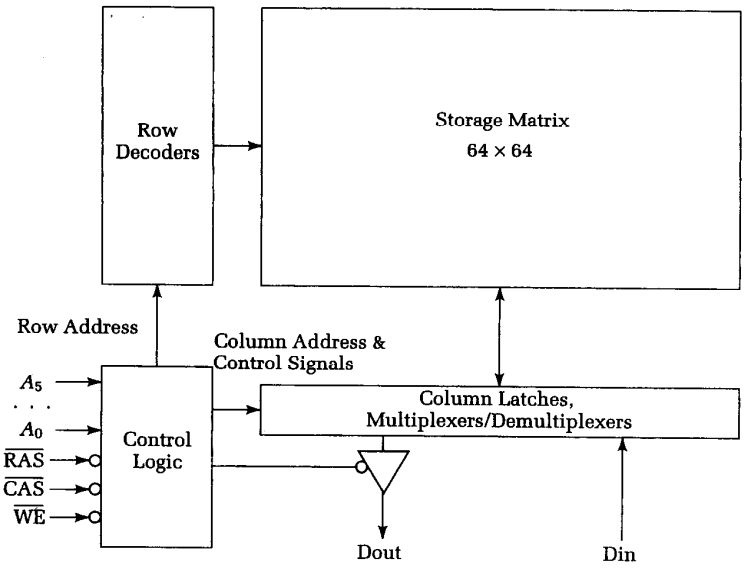


Figure 7.52 4096 × 1-bit DRAM block diagram.

1-bit DRAM. Rather than refresh individual bits, a refresh cycle reads out and writes back an entire row. This happens about once every few microseconds. Just as in the SRAM, the row is typically a multiple of the DRAM's word size. In this case, it is 64 bits wide. The refresh cycles are generated by an external memory controller.

DRAM Access with Row and Column Address Strobes Every time a single-bit word is accessed within the memory array of Figure 7.52, the DRAM actually accesses an entire row of 64 bits. The column latches select one of the 64 bits for reading or writing. The DRAM often accesses adjacent words in sequence, so it is advantageous if access is rapid.

Memory chip designers have developed clever methods to provide rapid access to the DRAM. The key is to provide separate control lines for DRAM row access, *RAS* (*row address strobe*), and column access, *CAS* (*column address strobe*). Normally, access involves specifying a row address followed by a sequence of column addresses. This has the extra advantage of reducing the number of address pins needed. A single (smaller) set of address lines are multiplexed over time to specify the row and column to be accessed. This becomes a critical issue as memory chips exceed 1 million bits (20 address pins).

In Figure 7.52 we have done away with the chip select signal and replaced it with two signals: $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$. The address lines, normally 12 for a 4096-bit memory, can now be reduced to 6. Memory access consists

of a RAS cycle followed by a CAS cycle. During the RAS cycle, the six address lines specify which of the 64 rows to access. In the following CAS cycle, the address lines select the column to access.

Figure 7.53 shows the RAS/CAS timing for a memory read. Throughout this sequencing, the $\overline{WE}$ line is held high. First the row address is provided on the address lines. When the $\overline{RAS}$ line is brought low, the row address is saved in a latch within the DRAM and the memory access begins. Meanwhile, the address lines are replaced by a column address. When $\overline{CAS}$ goes low, the column address is latched. At this point the output is enabled, although it is valid only after a propagation delay. When $\overline{RAS}$ goes high again, the accessed row is written back to the memory array, restoring its values. When $\overline{CAS}$ goes high, the output returns to the high-impedance state.

Figure 7.54 shows the RAS/CAS sequencing for a memory write. The signaling begins as before: the row address appears on the address lines and is internally latched when $\overline{RAS}$ goes low. The row is now read out from the memory array. While the address lines are changing to the column

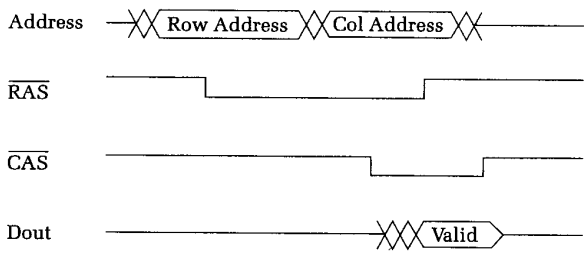


Figure 7.53 RAS/CAS sequencing for a memory read.

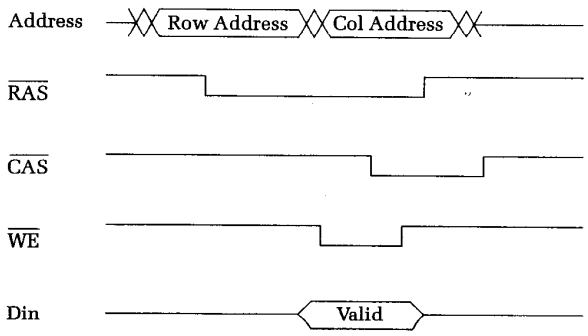


Figure 7.54 RAS/CAS sequencing for a memory write.

address, valid data is placed on the data input line and $\overline{WE}$ is taken low. Once the address lines are stable, $\overline{CAS}$ can also be taken low. This latches the column address and also replaces the selected bit with the DIN within the column latches. At this point, $\overline{WE}$ can be driven high. When $\overline{RAS}$ goes high again, the entire row, including the replaced bit, is written back to the memory array. Finally, $\overline{CAS}$ can be driven high, and another memory cycle can commence.

Refresh Cycle The storage capacitor at the heart of a DRAM memory cell is not perfect. Over time, it leaks away the charge it is meant to hold. Thus DRAMs must undergo periodic *refresh cycles* to maintain their state. In its simplest form, a refresh cycle looks like an abbreviated read cycle: data is extracted from the storage matrix and then immediately written back without appearing at the output pin.

Suppose every DRAM word must be refreshed once every 4 ms. This means that the 4096-word RAM would require a refresh cycle once every 976 ns. Assuming the cycle time is 120 ns, approximately one in every eight DRAM accesses would be a refresh cycle!

Fortunately, the two-dimensional organization of the storage matrix makes it possible to refresh an entire row at a time. Since the DRAM of Figure 7.52 has 64 rows, we can refresh the rows in sequence once every 62.5 μs, still meeting the overall 4-ms requirement. This is approximately one refresh cycle every 500 accesses. Larger-capacity DRAMs usually have 256 to 512 rows and require a refresh cycle once every 8 to 16 μs.

The RAS-only refresh cycle provides a simple form of refresh. It looks very much like the read timing with the CAS phase deleted. The row address is placed on the address lines and $\overline{RAS}$ is taken low. This causes the row to be read out of the storage matrix into the column latches. When $\overline{RAS}$ goes high again, the column latches are written back, refreshing the row's contents.

This refresh cycle requires an external memory controller to keep track of the last row to be refreshed. To simplify the memory controller design, some DRAMs have a refresh row pointer in the memory chip. A special CAS-before-RAS signaling convention implements the refresh. If $\overline{CAS}$ goes low before $\overline{RAS}$, the chip recognizes this as a refresh cycle. The indicated row is read, written back, and the internal indicator points to the next row to be refreshed.

7.6.3 DRAM Variations

We have described the basic internal functioning of a DRAM with RAS/CAS addressing, but have not shown how such an organization can improve DRAM performance. Variations on the basic DRAM model take advantage of the row-wide access to the storage matrix to reduce the time to access bits in the same DRAM row. These are called page mode,

static column mode, and nibble mode DRAMs. All three support conventional RAS/CAS addressing. They differ in how they specify accesses to additional bits in the same row.

Page mode DRAMs can read or write a bit within the last accessed row without repeating the RAS cycle. The first time a bit within a row is accessed, the controller sequences through a RAS followed by a CAS cycle, as described earlier. To access a subsequent bit in the row, the controller simply changes the column address and pulses the CAS strobe ($\overline{RAS}$ is held low throughout). CAS pulsing can be repeated several times to access a sequence of bits in the row. The result is much faster access than is possible with complete RAS/CAS cycling.

Static column mode DRAMs provide a similar function but present a slightly simpler interface to the memory controller. Changing the column address bits accomplishes a static column read, eliminating multiple strobes on the $\overline{CAS}$ line altogether. Writes are a little more complicated. To protect against accidentally writing the wrong memory location, either $\overline{CAS}$ or $\overline{WE}$ must be driven high before the column address can be changed.

Nibble mode DRAMs are yet another variation on page mode. Most memory locations are accessed in sequence, and the DRAM can take advantage of this to reduce the complexity of the control sequencing. After the first RAS/CAS cycle, a subsequent CAS pulse accesses the next bit in sequence. This can be done three times, yielding 4 bits in sequence, before a RAS/CAS cycle is needed again. Thus the sequence is RAS/CAS, CAS, CAS, CAS, RAS/CAS, CAS, CAS, CAS, etc.

Video RAMs (*VRAMs*) are DRAMs that can be used as frame buffers for computer displays. A *frame buffer* is a display memory that allows new data to be written to storage without affecting how the screen is being refreshed from the old data. A VRAM has a conventional DRAM storage matrix and four serial-access memories (SAMs). Its signaling convention allows a data row to be transferred from the storage matrix to the SAMs. Once in a SAM, the data can be read out a bit at a time at a high rate even while new data is being written into the storage matrix. In this way, VRAMs support a kind of dual-port access to memory: one from the standard read/write interface and one from the serial memories.

In addition, some VRAMs support logical operations, such as XOR, between the current contents of the memory and the bit that is overwriting it. This is useful for certain graphics-oriented operations, such as moving items around smoothly on the display.

7.6.4 Detailed SRAM Timing

Here we expand on the discussion of SRAM components begun in the previous section. We will describe the detailed timing of a 1024 by 4-bit static RAM, the National 2114. We have already shown the basic pin-out

in Figure 7.47: 10 address lines, 4 data input/output lines, and active low chip select ($\overline{CS}$) and write enable ($\overline{WE}$) control signals. The generic read and write cycle sequencings were shown in Figures 7.49 and 7.50.

Read Cycle Let's reexamine the read cycle timing in more detail. The following discussion assumes that $\overline{WE}$ is held high throughout the read operation. Any change on the address lines causes new data to be extracted from the storage matrix, independent of the condition of the chip select. Once $\overline{CS}$ goes low, the output buffers become enabled, latching the data from the storage array and driving the output pins.

An important metric of a memory component is the *access time*, t_A. This is the time it takes for an address change to cause new data to appear at the output pins (the memory has already been selected by driving CS low). The MM2114 has an access time of 200 ns, which is relatively slow by today's standards. High-speed static RAMs, such as the Cypress CY2148, have an access time of 35 ns.

An equally important metric is the *read cycle time*, t_{RC}. This is the time between the start of one read operation and a subsequent read operation. In the case of the MM2114, it is also 200 ns. In modern SRAM components, the access time and cycle time are usually the same. However, this is not the case for DRAMs, where cycle times are often longer than access times.

Figure 7.55 shows the read cycle timing waveform. It shows two back-to-back read cycles, the first commencing before $\overline{CS}$ goes low and the second while $\overline{CS}$ is low.

The first cycle begins with a change on the address lines. Valid data cannot appear on the output lines before an access time, 200 ns, has

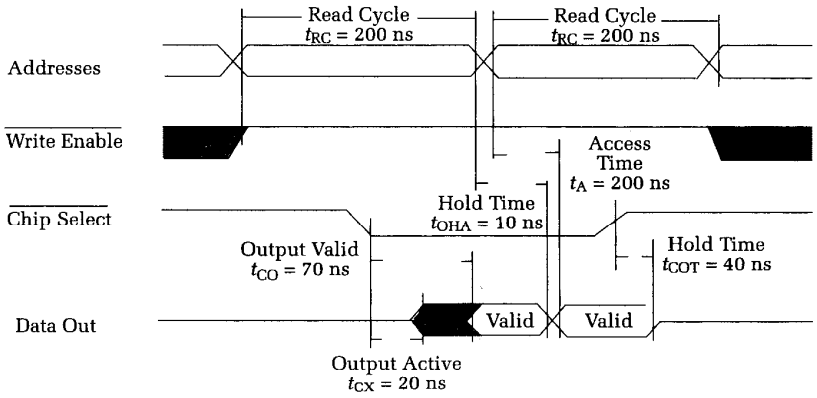


Figure 7.55 MM2114 read cycle timing.

expired. The timing waveform assumes that the limiting condition is not the access time, but rather the time from when the chip is selected. T_{CX}, the time from chip select to output active, is 20 ns. This means that the memory chip will begin to drive its outputs no sooner than 20 ns after chip select goes low, although the data is not yet valid. Any other components driving the same wires as the RAM's output lines must become tri-state within this time.

T_{CO}, the time from chip select to output valid, is 70 ns. Thus, the time to data valid is determined by which is longer: 200 ns from the last address line change or 70 ns from when the chip is selected.

Once the address lines begin to change for the next read cycle, the outputs are guaranteed to remain valid for another 10 ns, the output-hold-from-address-change time, t_{OHA}. External logic latching the output of the RAM must factor this into its timing before it can allow the addresses to change.

Once the address lines are stable, it takes another 200 ns for valid data to appear at the outputs. If a third read cycle were to commence now, by changing the address lines, the output data would remain valid for 10 ns. However, the read cycle is terminated in the waveform by deselecting the chip. In this case, the hold time is determined by t_{COT}, the time from chip select to output tri-state. For the MM2114, t_{COT} is a minimum of 0 ns and a maximum of 40 ns. Thus external logic must wait at least 40 ns before it can begin to drive the wires at the RAM's I/O pins.

Write Cycle As in all RAMs, the write cycle timing is more complex and requires more careful design. The write operation is enabled whenever $\overline{CS}$ and $\overline{WE}$ are simultaneously driven low. This is called the *write pulse*. To guard against incorrect writes, one or both of the signals must be driven high before the address lines can change.

Figure 7.56 gives the timing waveform for the write operation. Once the address lines are stable, we must wait an address setup time before driving the last of the chip select and write enable signals low. This is the address-to-write setup time, t_{AW}, and is 20 ns. The write cycle time, t_{WC}, is defined from address change to address change and must be at least 200 ns for this component. The write pulse width, t_{WP}, is at least 100 ns.

The write cycle ends when the first of the two control signals goes high. Thus, data setup and hold times are measured with respect to this event. The data setup time, t_{DS}, is at least 100 ns. The data hold time, t_{DH}, is 0 ns.

The write recovery time—the time between the end of the write pulse and when the address lines are allowed to change—is denoted by t_{WR}. For this RAM component, the recovery time is 0 ns.

The bottommost waveform in Figure 7.56, labeled Data Out, illustrates the interaction between read and write cycles. The writing circuitry must realize that the outputs could take as long as 40 ns to be tri-stated, so this

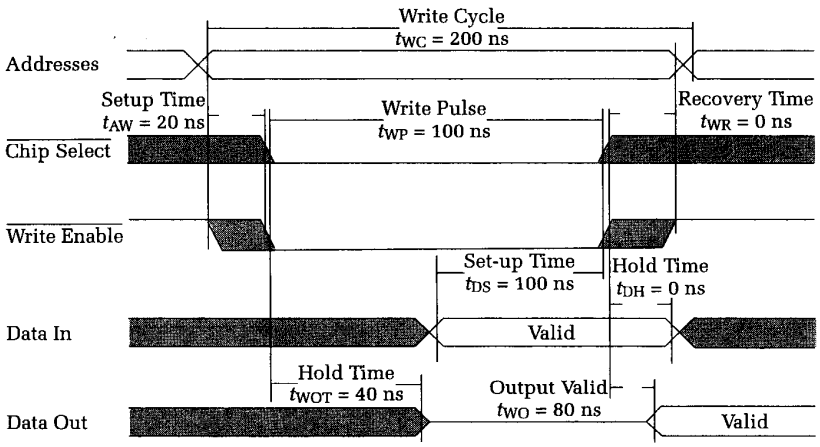


Figure 7.56 MM2114 write cycle timing.

amount of time must pass before the data to be written can be placed on the I/O pins. This is indicated by t_{WOT}, the time from write enable to output tri-state.

The final timing specification, t_{WO}, the write-enable-to-output-valid time, is the time between the end of the write cycle and when the RAM turns around to drive the output lines with the data just written (write enable high). In this case, it is 80 ns.

7.6.5 Design of a Simple Memory Controller

At the heart of most digital systems is a data path consisting of one or more interconnection pathways (called *buses*) and several registers, arithmetic circuits, and memory attached to some or all of these interconnections. This is the "switchyard," which routes data items from memory to a unit that executes some operation on them and then back into memory.

In this subsection, we will design a simple memory controller, using the TTL register and counter components introduced in this chapter, as well as a 2114 RAM chip. We will develop control circuitry for sequencing through the write enable and chip select signals to implement read and write operations.

Memory Subsystem Data Path Figure 7.57 gives a block diagram view of the data and address paths of a simple memory subsystem. To keep it simple, the address and data paths are just 4 bits wide. We can read data from four input switches and store them in the 2114 RAM when the tri-state buffer is enabled. Data stored in the RAM can be read out, latched into a

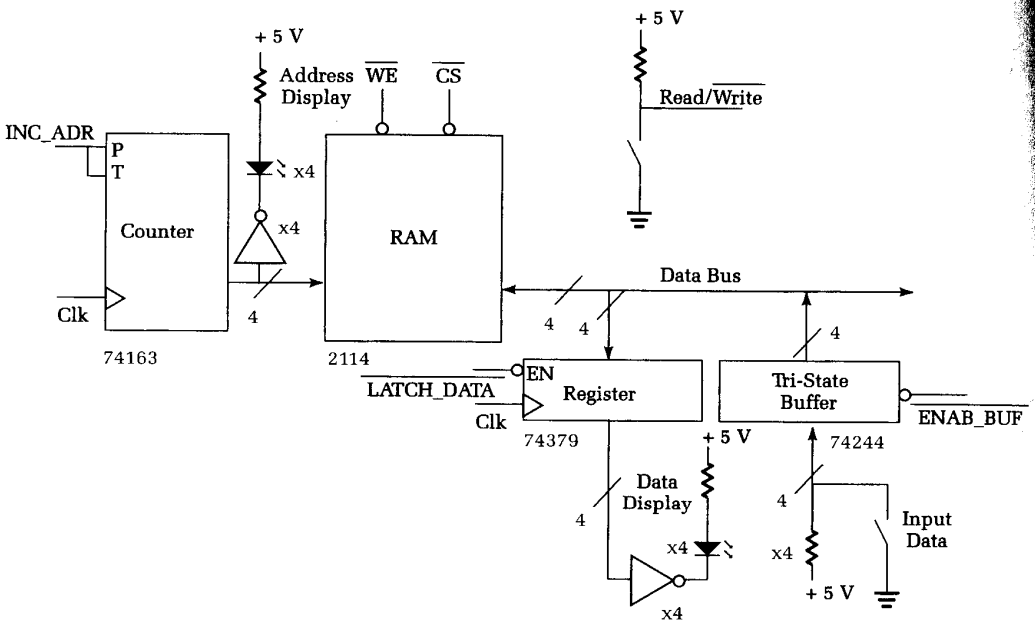


Figure 7.57 Data path of simple memory subsystem.

register, and then displayed on LEDs attached to the register's outputs (inverter drivers are used to buffer the LEDs). We use a 4-bit binary counter to access locations in memory sequentially for reading or writing. We also use LEDs to monitor memory addresses.

Figure 7.58 provides the schematic representation for the data path, using TTL components to implement the logic blocks of Figure 7.57. We implement the address counter with a 74163 four-bit binary up-counter. For the tri-state buffers and output latches we use one-half of a 74244 and a 74379 component, respectively. The 74379 is a 4-bit version of the 74377 introduced in Figure 7.3(a). For the purposes of the schematic, we have replaced the output LEDs by symbols for hexadecimal displays and the input switches by a hex keypad.

Memory Controller The following signals control the data path:

INC_ADR	Add one to address.
$\overline{\text{WE}}$	Write Enable on 2114.
$\overline{\text{CS}}$	Chip Enable on 2114.
$\overline{\text{LATCH\_DATA}}$	Latch valid data on data bus in display register during read cycle.

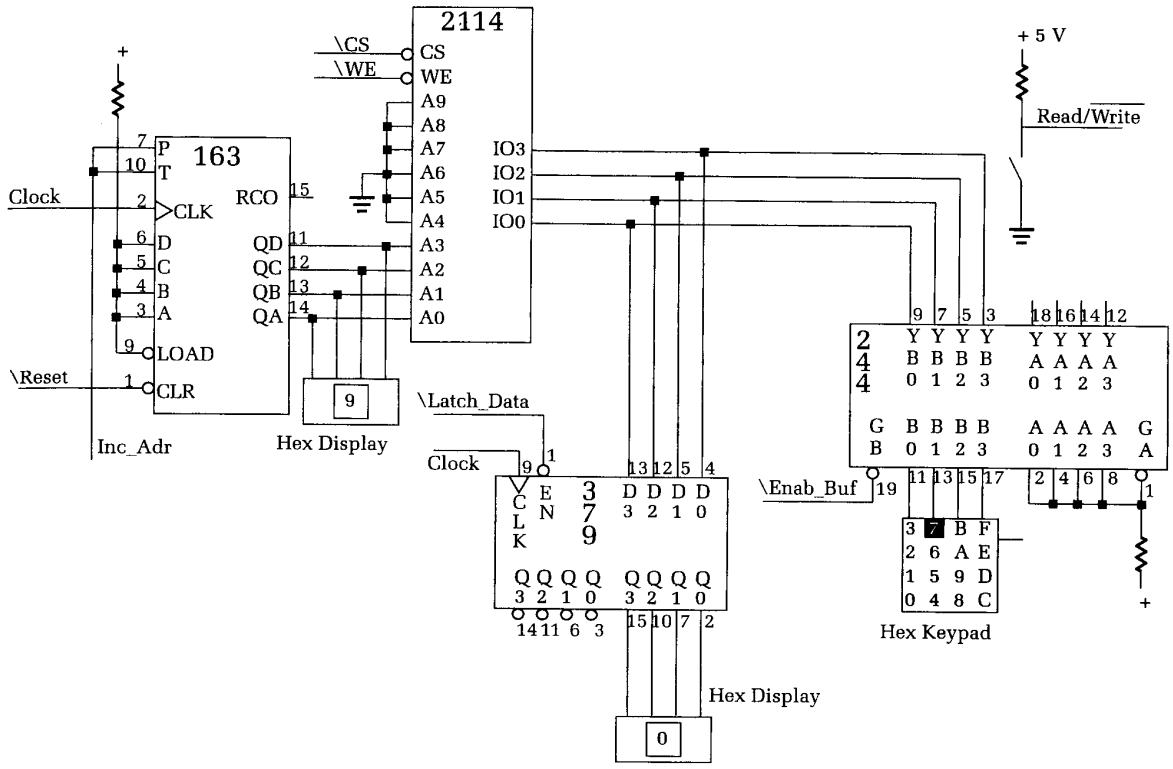


Figure 7.58 Schematic for memory subsystem data path.

$\overline{\text{ENAB\_BUF}}$	Enable buffer to put switch data on data bus during write cycle.
READ/$\overline{\text{WRITE}}$	User input to select read or write mode.

In addition, we need a global reset signal to force the counter to the 0000 state.

The memory controller reads from or writes to the current address, then increments the counter to point to the next address. First, a sequence of write cycles fills the RAM with data. Then the controller is reset, setting the address counter back to 0. A sequence of read cycles then views the data that has been stored in the RAM.

Figure 7.59 shows a skeletal sequencer circuit diagram. The timing waveform it generates is given in Figure 7.60. Pressing the momentary push-button switch generates a $\overline{\text{GO}}$ signal that lasts for one clock cycle. The $\overline{\text{GO}}$ signal enables the 74194 shift register, which shifts right, generating overlapping clock signals Φ_1, Φ_2, Φ_3, Φ_4. The circuit halts when

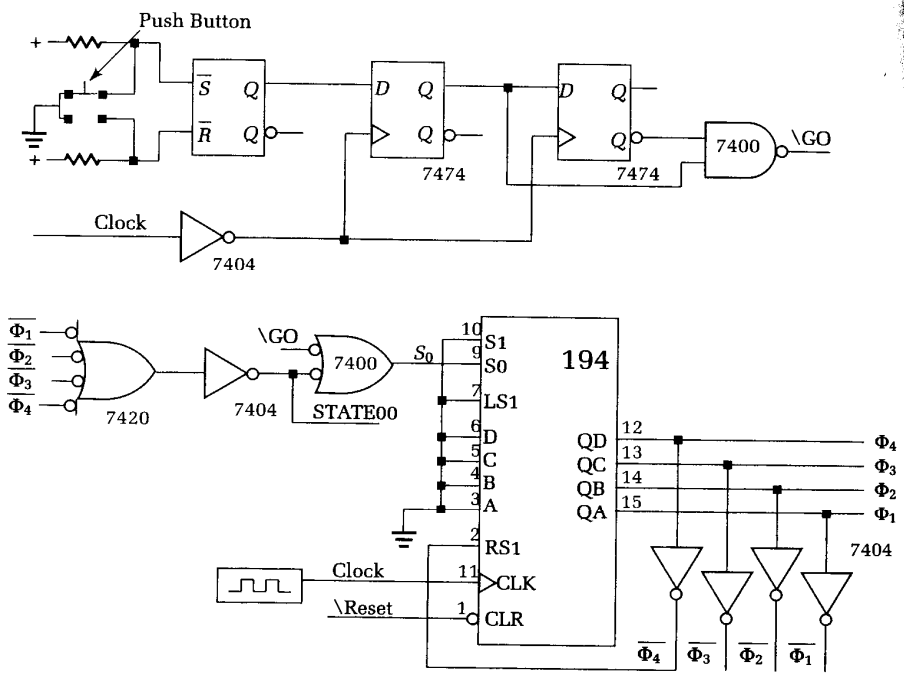


Figure 7.59 Multiphase clock for sequencer.

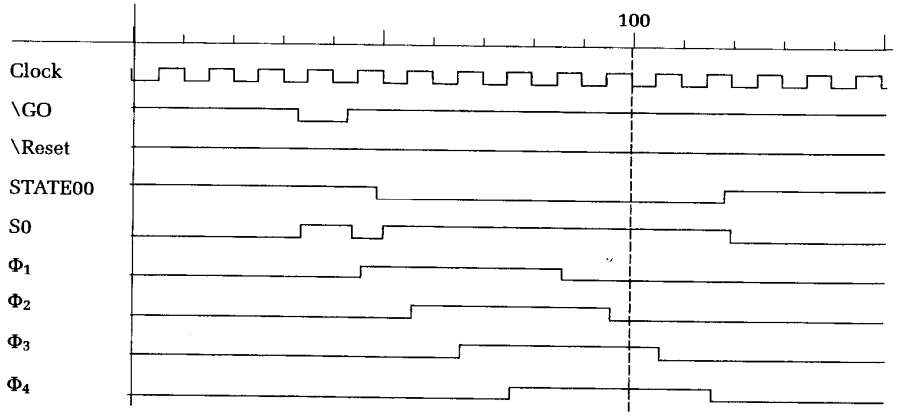


Figure 7.60 Timing waveforms for sequencer.

all of the clock signals return to 0. This sequencer could be driven with a slow clock, such as the 555 timer described in the last chapter.

We can use simple combinational logic to derive pulses of the correct start time and length for the various control signals from the multiphase clock. Figure 7.61 shows how this is accomplished.

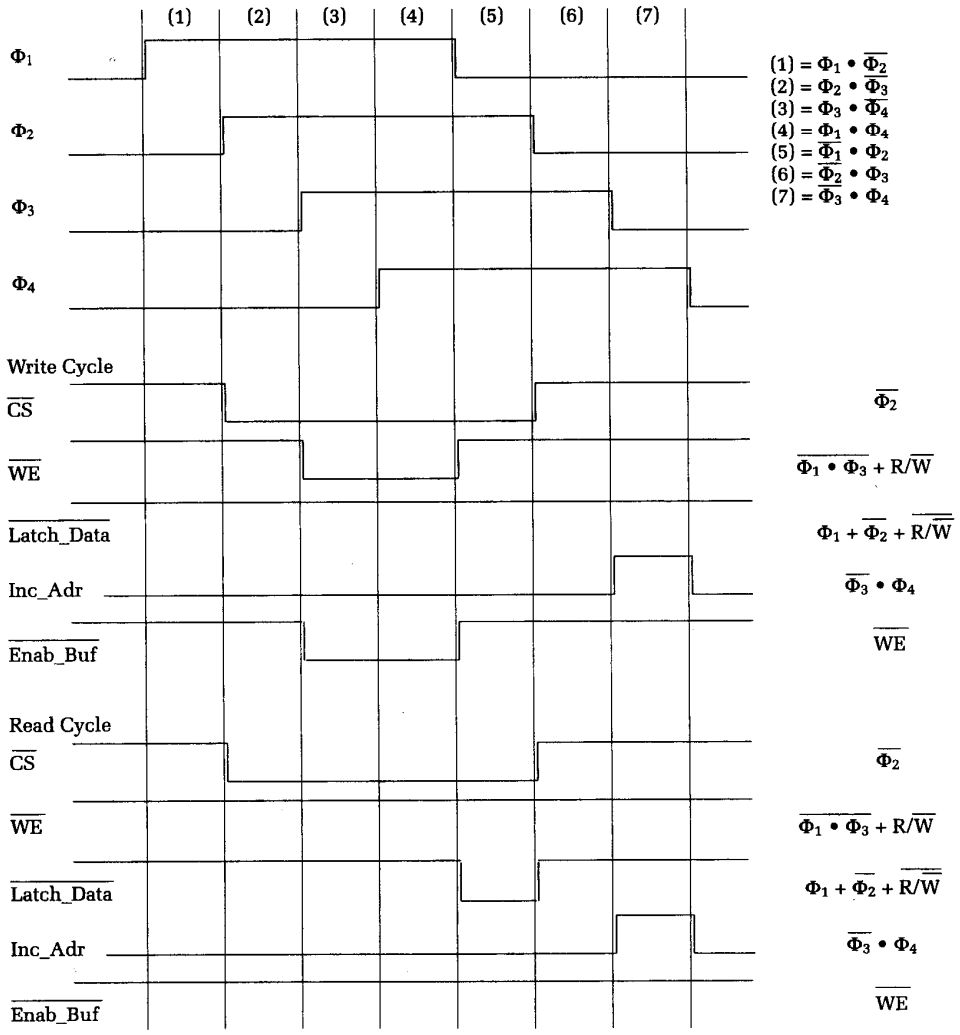


Figure 7.61 Generation of control signals from overlapping clocks.

We start by partitioning the overlapping clocks into seven periods each of which is defined as a unique function of two of the clock phases. By combining these functions, we can obtain equations for the individual control signals. For example, we choose to implement $\overline{CS}$ simply as the inversion of clock phase Φ_2. To be safe, we will design the high-to-low-to-high transitions on the $\overline{WE}$ signal to be properly nested within the $\overline{CS}$ transitions.

$\overline{WE}$ should be low exactly during the time that clock phases Φ_1 and Φ_3 overlap (periods 3 and 4 in Figure 7.61). This is easy to generate with combinational logic:

$$WE = \Phi_1 \bullet \Phi_3 \bullet \overline{R/W} \qquad \overline{WE} = (\overline{\Phi_1 \bullet \Phi_3}) + R/\overline{W}$$

The write enable signal should be asserted whenever clock phases Φ_1 and Φ_3 are asserted simultaneously and the READ/$\overline{WRITE}$ signal is low. Since $\overline{WE}$ is active low, we invert this logic to obtain the current sense of the signal.

We assert $\overline{LATCH\_DATA}$ during period 5, when Φ_1 is low and Φ_2 is high. Since the signal is active low, the implementation for the control signal becomes

$$\overline{\overline{\Phi_1} \bullet \Phi_2 \bullet R/\overline{W}} = \Phi_1 + \overline{\Phi_2} + \overline{R/\overline{W}}$$

INC_ADR is active during period 7, so its implementation becomes $\overline{\Phi_3} \bullet \Phi_4$. Finally, $\overline{ENAB\_BUF}$ is identical to the signal $\overline{WE}$.

Of course, alternative implementations are possible as long as they lead to valid sequencing of the control signals. Also, the logic must be designed so that the sequence meets the setup and hold time requirements for the 2114 RAM.

Chapter Review

In this chapter, we have focused on a variety of sequential circuits used mostly as storage elements: registers, counters, and RAMs. We saw how to construct registers for storage and shifting from simple *D*-type flip-flops sharing common clock lines.

Random-access memories provide a large number of storage elements in a single integrated circuit package. Static RAMs are based on a storage element constructed from cross-coupled inverters. These devices tend to be very fast. Dynamic RAMs are based on a single transistor storage element. They have higher capacity than SRAMs but longer access times.

Counters are an important class of sequential circuits, not only because they count events but also because they can be used to generate a periodic sequence. This has important implications for controller design, which we will examine in more detail in the next chapter.

We introduced a design procedure for mapping a state transition diagram, describing the count sequence, into actual hardware. The steps involved deriving the state diagram from the written specification for the counter, deducing a state table from the diagram which tabulates current and next states, expressing each next-state bit as a combinational logic function of the current-state bits, and remapping these next-state functions according to the kind of flip-flop chosen as a storage element. Excitation tables provide the information for remapping the next-state truth tables into truth tables for the selected flip-flops' inputs.

In general, *J-K* flip-flops yield counter implementations with the fewest gates. However, we often choose *D* flip-flops because of the simplified design procedure. No remapping step is necessary, and fewer wires are needed to control a *D* flip-flop.

Because real hardware does not necessarily come up in a known state, we described the concept of self-starting counters. Self-starting design methods can be used to get a counter (eventually) into a known, valid state.

We presented counters without a global clock, asynchronous counters, and pointed out that their use should be avoided. We also demonstrated the advantages of counters with synchronous load and clear inputs, especially for the design of counters with starting or ending offsets.

In the final section, we looked at the timing behavior of a typical SRAM component in more detail. The important performance metrics are the memory access time and the memory cycle time. In addition, we introduced a simple memory controller design to illustrate the "glue" logic that must surround a RAM subsystem to interface it to the rest of the digital system.

We are now ready to study the design of more general circuits that follow a prescribed sequence other than the counters we have studied up to now. These *finite-state machines* will be the topic of the next chapter.

Further Reading

Just about any textbook on digital design will describe register and counter components and design strategies that employ them. It is more difficult to find good descriptions of memory components. J. Wakerly, *Digital Design: Principles and Practices*, Prentice-Hall, Englewood Cliffs, NJ, 1990 and Johnson and Karim, *Digital Design: A Pragmatic Approach*, PWS Publishers, Boston, 1987, are notable exceptions.

Perhaps the best way to learn about digital components is to study the relevant data books. For example, Cypress Semiconductor's *CMOS/BiCMOS Data Book*, San Jose, CA, 1989, describes their product line of very fast CMOS SRAMs in considerable detail.

An excellent survey and tutorial on memory system design using general-purpose and special-purpose RAMs can be found in Jean-Daniel Nicoud's paper "Video RAMs: Structure and Applications," which

appeared in *IEEE MICRO* (February 1988), pp. 8–27. Nicoud describes how to apply video RAMs in the design of graphical frame buffers and compares the designs with those employing conventional RAMs.

For a system-level perspective on interfacing hardware to memory systems, see M. Slater's textbook *Microprocessor-Based Design*, published by Prentice-Hall in 1989.

Exercises

...ift Register Design) Design the basic cell of a universal shift register to the following specifications. The internal storage elements will be positive edge-triggered D flip-flops. Besides the clock, the shifter stage has two external control inputs, S_0 and S_1, and three external data inputs, SR, SL, and DI. SR is input data being shifted into the cell from the right, SL is data being shifted from the left, and DI is parallel load data. The current value of the flip-flop will be replaced according to the following settings of the control signals: $S_0 = S_1 = 0$: replace D with DI; $S_0 = 0$, $S_1 = 1$: replace D with SL; $S_0 = 1$, $S_1 = 0$: replace D with SR; $S_0 = S_1 = 1$: hold the current state. Draw a schematic for this basic shifter cell.

7.2 *(Shift Register Design)* Shifters are normally used to shift data in a *circular* pattern (the data that shifts out at one end of the shifter is shifted back into the other end), or as a *logic* shift (fill the shifted positions with 0's) or an *arithmetic* shift (propagate the high-order sign bit to the right or shift in 0's to the left). For example, if a 4-bit register contains the data 1110, the effects of the six kinds of shifts are the following:

Circular shift right: 1110 becomes 0111.

Circular shift left: 1110 becomes 1101.

Logical shift right: 1110 becomes 0111.

Logical shift left: 1110 becomes 1100.

Arithmetic shift right: 1110 becomes 1111.

Arithmetic shift left: 1110 becomes 1100.

Show how to wire up a 4-bit universal shift register (TTL component 74194) to perform the following kinds of shifts:

a. Circular shift right

b. Circular shift left

c. Logic shift right

d. Logic shift left

e. Arithmetic shift right

f. Arithmetic shift left

7.3 *(Shift Register Design)* Your task is to design a shift register subsystem based on the TTL 74194 component that can implement the six kinds of shifts described in Exercise 7.2. The subsystem has three control inputs, S_2, S_1, S_0 that are interpreted as follows: S_2, S_1, $S_0 = 000$ is hold; 001 is circular shift right; 010 is circular shift left; 011 is logic shift right; 100 is logic shift left; 101 is arithmetic shift right; 110 is arithmetic shift left; 111 is parallel load.

a. Show the data path for the shifter subsystem. You may use multiplexers at the shift inputs to the 74194.

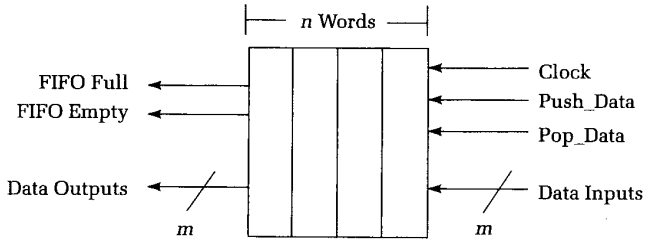
b. Show the combinational logic (equations or schematics) to decode the global S_2, S_1, S_0 control inputs into the appropriate detailed control signals for the 74194 shifter and the external data path logic for handling the serial shift inputs.

7.4 *(Register Design)* A FIFO (first in, first out) queue is a special-purpose register file n words deep and m bits wide that operates as follows (see the block diagram in Figure Ex7.4(a)). When a PUSH_DATA control input is asserted, new data at the inputs at the right is read into the end of the queue. When a POP_DATA control input is asserted, existing data at the head of the queue becomes available at the outputs at the left. Since the FIFO has finite capacity, two status outputs indicate whether the FIFO is empty or full. PUSH_DATA is inhibited in a full FIFO, while POP_DATA is inhibited in an empty FIFO. On reset, the FIFO should be set to empty.

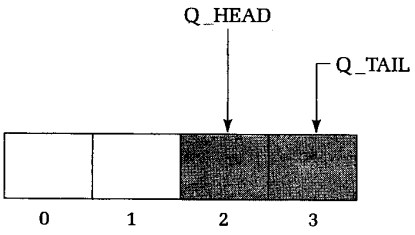
a. A "flow-through" FIFO is the simplest form of this kind of device. The FIFO must fill up with data before any data can be removed. Furthermore, the FIFO must be completely emptied before new data can be placed in it. Using only shift register components and combinational logic, design a flow-through 4 word by 4 bit FIFO. Consider carefully how to represent the empty/full status of the FIFO. (*Hint*: Consider adding an (m + 1)st bit to the FIFO to indicate whether the FIFO word is valid.)

b. A more flexible version of the FIFO can be implemented with counter and register components. Two counters represent the Q_HEAD and Q_TAIL respectively. These can be used as an index to a register file or an array of registers for reading from

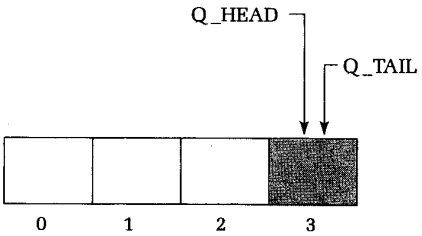
the head of the FIFO or writing to its tail. (b) shows one way to implement the PUSH_DATA and POP_DATA operations. POP_DATA delivers $word_2$ to the outputs and advances the Q_HEAD pointer. PUSH_DATA advances the tail pointer and writes the input to $word_0$. Notice how the queue wraps around on itself. Your task is to implement a 4 word by 4 bit FIFO using two 2-bit counters for the Q_HEAD and Q_TAIL. Think carefully about how you determine whether the FIFO is empty or full. Don't forget to consider the starting configuration of the head and tail pointers on FIFO reset.



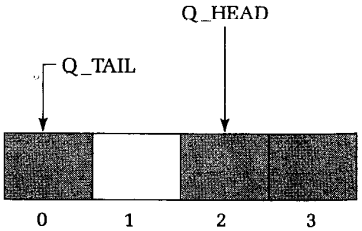
(a) FIFO block diagram



Starting Configuration



Result of POP_DATA Operation



Result of PUSH_DATA Operation

(b) PUSH_DATA and POP_DATA FIFO operations

Figure Ex7.4

7.5 *(Register Design)* A LIFO (last in, first out) stack is similar in concept to the FIFO queue, except that the most recently pushed data is the first to be popped. The block diagram is identical to the FIFO, except that the data inputs and outputs are the same lines. Design a 4 word by 4 bit LIFO stack using shift registers and combinational logic only. Draw your schematic, indicating the components used. How do you distinguish between a full stack and an empty stack in this implementation?

7.6 *(Register Design)* Repeat Exercise 7.5, this time using registers and counters. Draw the schematic. How do you distinguish between a full stack and an empty stack? What is the initial condition of the pointers at reset?

a. *(Register Design)* One way to compute the twos complement of a number is examine the number bit by bit from the lowest-order bit to the highest-order bit. In scanning from right to left, find the first bit that is one. All bits to the left of this should now be complemented to form the twos complement number. For example, the twos complement of 0010 is formed as follows:

$$\begin{array}{lll} 0010 & \longrightarrow 1101 & \text{(ones complement)} \\ \downarrow & \quad +1 & \\ 1110 & 1110 & \text{(twos complement)} \end{array}$$

b. Your task is to draw a schematic for a 4-bit register with parallel inputs and outputs and the synchronous control signals HOLD, CLEAR, LOAD, and COMPLEMENT. When the complement signal is asserted, the register's contents will be replaced by its twos complement.

c. *(Shifter Application)* Figure 7.9 shows the partial implementation of a bit-serial transmission subsystem. This subsystem is unidirectional: it can transmit only from the left shift registers to the right.

i. Extend this data path implementation to make it bidirectional. Under external control, include logic (such as multiplexers) thatComposition 2 allows the subsystem to transmit from right to left as well as left to right.

ii. The subsystem also needs a counter that can raise a signal after 8 bits have been transmitted. Show how a counter such as the TTL 74163 synchronous up-counter can be used to perform this function.

7.7 *(Shifter Application)* In this exercise, you will generate control logic for use with the data path you developed in Exercise 7.6. You are given the control signals DIRECTION (0 = left to right, 1 = right to left), LOAD (load parallel data), XMIT (transmit bits in the indicated direction). External logic first sets the DIRECTION, then asserts LOAD to load parallel data, and finally asserts XMIT to begin the data transfer. Your logic will use these inputs to (a) parallel load the appropriate shifter registers, (b) shift 8 bits in the correct direction, and (c) place the receiving registers in the hold state when the transfer is complete. State all assumptions, and show your equations or logic schematics.

7.8 *(Counter Design)* Design a 2-bit counter that behaves according to the two control inputs I_0 and I_1 as follows: I_0, $I_1 = 0$, 0: stop counting; I_0, $I_1 = 0$, 1: count up by one; I_0, $I_1 = 1$, 0: count down by one; I_0, $I_1 = 1$, 1: count by two.

a. Draw the state diagram and state transition table.

b. Implement the counter using *T* flip-flops, *D* flip-flops, and *J-K* flip-flops.

c. Which choice of flip-flops leads to the minimum gate count? Assume that only two-input NAND, NOR, XOR, and XNOR gates are available. Draw the schematic for your minimum gate count implementation.

d. Which choice of flip-flops leads to the minimum wire count? The same kinds of gates are available as in part (c). Draw the schematic for your minimum wire count implementation. Indicate how you have counted the interconnections.

7.9 *(Counter Design)* Design a three flip-flop counter that counts in the following sequence: 000, 010, 111, 100, 110, 011, 001, and repeat. Design the counter using toggle flip-flops. Verify that your implementation is self-starting.

7.10 *(Counter Design)* Consider the design of a 4-bit BCD counter that counts in the following sequence: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, and then back to 0000, 0001, etc.

a. Draw the state diagram and next-state table.

b. Implement the counter using *D* flip-flops, toggle flip-flops, *S-R* flip-flops, and *J-K* flip-flops.

c. Implement the counter making it self-starting just for the *D* flip-flop case.

7.11 *(Counter Design)* Consider the design of a 4-bit Gray code counter (that is, only one of the state bits changes for each transition) that counts in the following sequence: 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000, and then back to 0000, 0001, 0011, etc.

a. Draw a state diagram and next-state table.

b. Implement the counter using *D* flip-flops, toggle flip-flops, *R-S* flip-flops, and *J-K* flip-flops.

c. Do you have to worry about self-starting? Why or why not?

7.12 *(Counter Design)* The 4-bit Johnson counter advances through the sequence 0000, 1000, 1100, 1110, 1111, 0111, 0011, 0001, and repeat. Using the standard counter design process, show how to implement this count sequence using (a) *D* flip-flops and (b) *T* flip-flops. How does your solution compare with the *J-K* implementation of Figure 7.10, in terms of gates and wiring complexity?

7.13 *(Self-Starting Counters)* Analyze your solutions to Exercise 7.12(a) and (b) to check whether or not they are self-starting. Draw complete state diagrams and show all states and transitions implied by your implementation.

7.14 *(Reverse Engineering)* What is the counter state diagram implied by the flip-flop implementation of Figure Ex7.14?

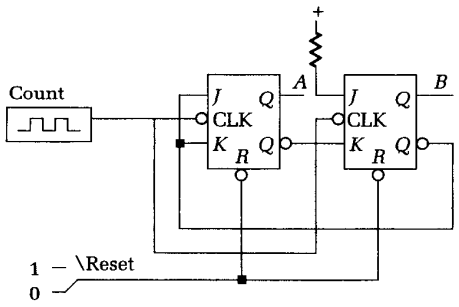


Figure Ex7.14 Counter implementation to reverse engineer in Exercise 7.14.

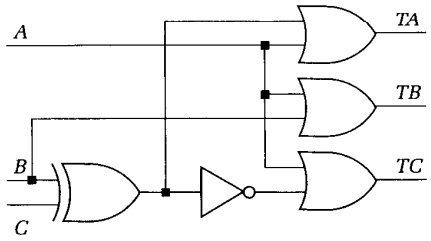


Figure Ex7.15 Next state function to be reversed engineered in Exercise 7.15.

7.15 *(Reverse Engineering)* Consider a counter implemented with three toggle flip-flops, labeled *C*, *B*, *A* from highest-order bit to lowest. The next-state function is implemented by the logic in Figure Ex7.15. Assuming the counter can be reset to state 000, what is the state diagram for this counter?

7.16 *(Ripple Counters)* Figure 7.37 shows a 3-bit ripple up-counter. What simple change can you make to this circuit to convert it to a down-counter? Show your logic circuit and associated timing diagram. Explain why your circuit operates correctly.

7.17 *(Asynchronous Inputs)* Can you think of any cases in which an asynchronous load or reset signal is acceptable in a digital design?

7.18 *(Offset Counters)* Figure 7.41 shows a possible implementation for a counter with a beginning offset. The counter will eventually enter the correct sequence, but in some applications this may not be acceptable. How might you add a reset signal that will force the counter into a valid state when it is asserted? What assumptions must be made about the duration of the reset signal?

7.19 *(Offset Counters)* Use two cascaded synchronous up-counters (for example, a TTL 74163 component) to implement a 6-bit offset counter that counts from 000010 to 110011 and repeats. Make sure the counter begins in state 000010 when the external reset signal is asserted.

7.20 *(Self-Starting Counters)* Determine whether the counter implementation shown in the K-maps of Figure 7.34 is self-starting. Show the complete state diagram including the illegal states.

7.21 *(Self-Starting Counters)* Determine whether the counter implementation indicated by the logic schematic of Figure 7.36 is self-starting. Show the complete state diagram including the invalid states. How does it compare with the complete state diagram of Exercise 7.20?

7.22 *(Counter/Register Applications)* Consider the design of a bit-serial adder. This circuit uses a single full adder to add two binary numbers presented in serial fashion, 1 bit at a time.

a. Draw the schematic for a 4-bit version of this circuit. Two 4-bit shift registers are loaded with the data to be added in parallel. These are shifted out a bit at a time, starting with the lowest-order bit, into the *A* and *B* inputs of the full adder. The partial sum is shifted into a third register. How should the carry out be handled between subsequent bits?

b. Define your control signals for the bit-serial adder subsystem. Draw a timing diagram that illustrates the sequencing of these signals to implement the 4-bit addition. What happens on reset?

7.23 *(Counter/Register Applications)* Flip-flops, registers, and counters can be used to implement a variety of useful clocking functions.

a. Design a system that will generate a single clock pulse one period long each time a push-button is pressed (you may assume that an external reference clock is available).

b. Design a system using an MSI counter that will assert a signal for exactly 13 clock pulses each time a push-button is pressed (once again, an external reference clock is available).

c. In many applications, it is desirable to single step the clock as well as to halt a free-running clock signal and later restart it. Design a circuit that will generate a single clock pulse each time a STEP push-button is pressed. Provide a separate circuit, independent of STEP, that will cleanly turn the clock on when a RUN switch is in the ON position and off when RUN is in its off position. No marginal/partial clock outputs are allowed.

7.24 *(Counter/Register Applications)* Design a 2-bit binary up-counter to the following specification. The counter has five inputs (not including the clock) and three outputs. The inputs are CLR, LOAD, COUNT, L_B, and L_A. CLR takes precedence over LOAD, which in turn takes precedence over COUNT. The outputs are *B*, *A*, and RCO.

7.25 When CLR is asserted, the flip-flops of the counter in Exercise 7.24 are reset to 0. If LOAD is asserted (when CLR is not asserted), the flip-flops' contents are replaced by the L_B and L_A inputs. If COUNT is asserted (when CLR and LOAD are not asserted), the flip-flops' contents are incremented: 00 becomes 01, 01 becomes 10, 10 becomes 11, and 11 becomes 00. When the counter is in state 11, RCO is asserted.

7.26 Implement the counter in Exercise 7.24 using *D* flip-flops and as few logic components as possible. *Hint*: Use 4-to-1 multiplexers to implement your next-state function. You may also use XOR gates, as well as standard AND, OR, and NOT gates.

7.27 *(Random-Access Memories)* A microprocessor with an 8-bit-wide data bus uses RAM chips of 4096 by 1-bit capacity. How many chips are needed and how should their address lines be connected to provide a memory capacity of 16 Kbytes (1 byte = 8 bits).

7.28 *(Random-Access Memories)* Consider a 1-megabit dynamic memory component. The memory is organized into 512 rows of 2048 bits each. Assume that every bit must be refreshed within 4 ms. How frequently should a row refresh operation be scheduled? If the memory has an 80 ns access time, approximately what fraction of memory accesses must be dedicated for refresh?

7.29 *(Random-Access Memories)* Consider the memory controller design described in Section 7.6.5. Show an alternative implementation of the control signals INC_ADR, $\overline{\text{LATCH\_DATA}}$, and $\overline{\text{ENAB\_BUF}}$ that leads to a different detailed sequencing of the signals that is still logically correct.

7.30 *(Random-Access Memories)* Consider the read and write timing of the 2114 memory component in Figures 7.55 and 7.56. What is the minimum clock width for the overlapping clocks generated by the memory controller that will still meet the memory's timing specification? Justify your answer.