Part One

Computer Engineering

COPYRIGHTED MATERIAL

Chapter 1

Digital Logic and Microprocessor Design

This chapter focuses on the fundamentals of digital logic and design, with numerous examples from both computer hardware design and "everyday life" events to demonstrate that digital logic is not confined to designing computers. My objective is to equip the engineer or student with sufficient knowledge of design principles to be able to design a digital computer. In addition, I integrate the important role that software plays in modern computer systems with the hardware design principles. Numerous design examples and solved problems are provided to support learning objectives.

MICROPROCESSOR DESIGN

Functions

Using its arithmetic logic unit(ALU), a microprocessor can perform mathematical and logic operations like addition, subtraction, multiplication, division, and comparison. Modern microprocessors contain complete floating-point processors that can perform extremely sophisticated operations on large variable-length numbers. In addition, a microprocessor can perform the following functions:

Move data from one memory location to another.

Make decisions and jump to a new set of computer program instructions based on those decisions.

Use an RD (read) and WR (write) line to tell the memory whether it wants to read from or write to the addressed location.

Use a clock line to transmit clock pulses (CPs) to sequence the microprocessor. For example, when numbers are added by the microprocessor, which you

Computer, Network, Software, and Hardware Engineering with Applications, First Edition. Norman F. Schneidewind.
© 2012 the Institute of Electrical and Electronics Engineers, Inc. Published 2012 by John Wiley & Sons, Inc.

will see later, addition takes place bit by bit, and the clock triggers each binary bit addition to ultimately form a decimal result.

Uses a reset line to reset the program counter to zero and restart execution.

Components

Microprocessor components are the building blocks of modern computers. These components are the following:

- ***ALU.*** Consists of accumulators, registers, and control unit.
  - The ALU executes instructions and manipulates data.
  - An 8-bit ALU can add, subtract, multiply, and divide two 8-bit numbers, while a 32-bit ALU can manipulate 8-bit, 16-bit, and 32-bit numbers.
  - An 8-bit ALU would have to execute four instructions to add two 32-bit numbers (four add instructions, each of which adds 8-bit numbers), whereas a 32-bit ALU can do it in one instruction.
- ***Accumulator.*** Holds data and instructions for processing by the ALU.
- ***Register.*** Temporary storage of instructions and data.
  - ***Program Counter*** **(PC).** Contains the address of next instruction to be executed
  - ***Instruction Register*** **(IR).** Holds address of current instruction being executed
  - ***General Registers.*** Holds operator (e.g., code for add instruction), operands (e.g., numbers to be added), and data while an instruction is executed
- ***Stack.*** Temporary storage of instructions and data, usually on a last in, first out (LIFO) basis. Also called push-down stack.
- ***Control Unit.*** Fetches and decodes instructions, generates signals for the ALU to execute instructions
- ***Busses***
  - ***Address Bus.*** Path over which addresses flow for directing memory and input/output (I/O) data transfers. An address bus may be 8, 16, or 32 bits wide that sends an address to memory or I/O for accessing memory or I/O.
  - ***Data Bus.*** Transfers data. A data bus may be 8, 16, or 32 bits wide that can send data to memory or I/O and receive data from memory or I/O. The number of address bus lines determine the amount of addressable memory (n lines = 2^n addressable words).
  - ***Control Bus.*** Communicates control and status information.
- ***Chip.*** A chip is also called an integrated circuit. Generally it is a small, thin piece of silicon onto which the transistors making up the microprocessor have been etched. A chip might be as large as an inch on a side and can contain tens of millions of transistors. Simpler processors might consist of a few

thousand transistors etched onto a chip just a few millimeters square. Microns are the width of the smallest wire on the chip. For comparison, a human hair is 100 µm thick. As the feature size on the chip goes down, the number of transistors rises.

Characteristics

Microprocessor characteristics govern the speed and functionality of computer operations. Important characteristics include the following presented in the succeeding paragraphs.

Smaller microprocessors can be combined into a larger one (four 4-bit microprocessors combined into one 16-bit microprocessor).

A crystal-controlled clock sequences the operations of a microprocessor (e.g., the sequence of computer program instruction execution) by generating CPs. Clock speed is specified in cycles per second, where 1 MHz is equal to 1 million cycles per second. Clock speed is the maximum speed of the chip.

Instructions require one or more clock cycles to execute the following, depending on its complexity: fetch instruction from memory, decode the operation code, fetch operands from memory, execute the instruction, and store the result in memory. In addition to clock speed, an important performance metric is the number of floating-point operations per second or flops.

***Complex instruction set computing* (CISC).** A single instruction can perform several operations. This design simplifies programming because, for example, a single instruction can fetch instruction from memory, decode the operation code, fetch operands from memory, execute the instruction, and store the result in memory. However, the downside is the relatively slow speed of the computer [RAF05].

***Reduced instruction set computing* (RISC).** Several operations are required to execute a single instruction. This design provides high speed, for example, well suited to real-time applications that must meet deadlines, but at the expense of relatively complex programming.

Performance

One measure of the computing power of a microprocessor is its clock speed, measured in millions of cycles per second (MHz). It usually takes from one to seven cycles of a microprocessor's internal clock to fully process an instruction. The faster the internal clock, the more instructions can be processed per unit of time. For the microprocessors in laptop and desktop computers, clock speeds are usually greater than 100 MHz. The fastest microprocessors can run at a speed of 2 GHz. From a user standpoint, the most important performance metric is program execution time, defined as [HAR07]:

$$\text{Program execution time} = (\text{Number of instructions in program}) * (\text{Clock cycles per instruction}) * (\text{Time per clock cycle}).$$

Another measure of performance is the number of instructions that can be processed per second, referred to as MIPS, for million instructions per second. The MIPS rating of a microprocessor depends on both the clock speed and the number of instructions that can be executed per clock cycle. Simple microprocessors can execute a maximum of one instruction per clock cycle. Advanced microprocessors can execute up to six or eight instructions per clock cycle. The relationship between clock speed and MIPS is not straightforward, however, because some instructions may take more than one clock cycle to execute, depending on the program. The product of clock speed and the number of instructions that can be executed per cycle may be greater than MIPS. The maximum clock speed is a function of the manufacturing process and delays within the chip. MIPS is proportional to the clock speed and inversely proportional to the number of clock cycles per instruction.

Another indication of microprocessor speed is the word length, as measured by the number of bits of information that can be transferred simultaneously. Long words allow the microprocessor to handle data and perform complex tasks more efficiently. The number of bits per word has been steadily increasing with the growth of circuit technology. Thus 4-, 8-, 16-, 32-, and 64-bit microprocessors are now common. Some personal computers use 32-bit microprocessors. More powerful computers use 64-bit microprocessors. The 4-, 8-, or 16-bit devices are usually employed in simple embedded applications, such as microwave ovens, electric shavers, and televisions. Figure 1.1 shows the microprocessor architecture.

Pipeline Systems

An important aid to performance is the pipeline system. The purpose of a pipeline system is to reduce delay caused by the computer processor having to wait for instructions to complete. With a pipeline design, the processor begins the execution of the next instruction while the current instruction is executing. Thus, various phases of instruction execution are overlapped. The concept is to keep the pipeline full, with as many execution sequences as possible. For example, due to overlapped instruction execution, each instruction overlaps during $(n - 1)$ clock cycles, and each of $m = 4$ instructions requires one clock cycle, yielding $(n - 1) + m = 7$ clock cycles, total, as shown in Figure 1.2.

Problem: How is the *increase in speed*, obtained by a pipelined system over a conventional system, computed?

Answer: Using Figure 1.2 as an example, the increase is computed as follows:

The number of clock cycles required in conventional system is $mn = 4 * 4 = 16$ in the example of Figure 1.2. Thus, the decrease in number of clock cycles for a pipelined system is:

$$mn - ((n - 1) + m) = 16 - 7 = 9,$$

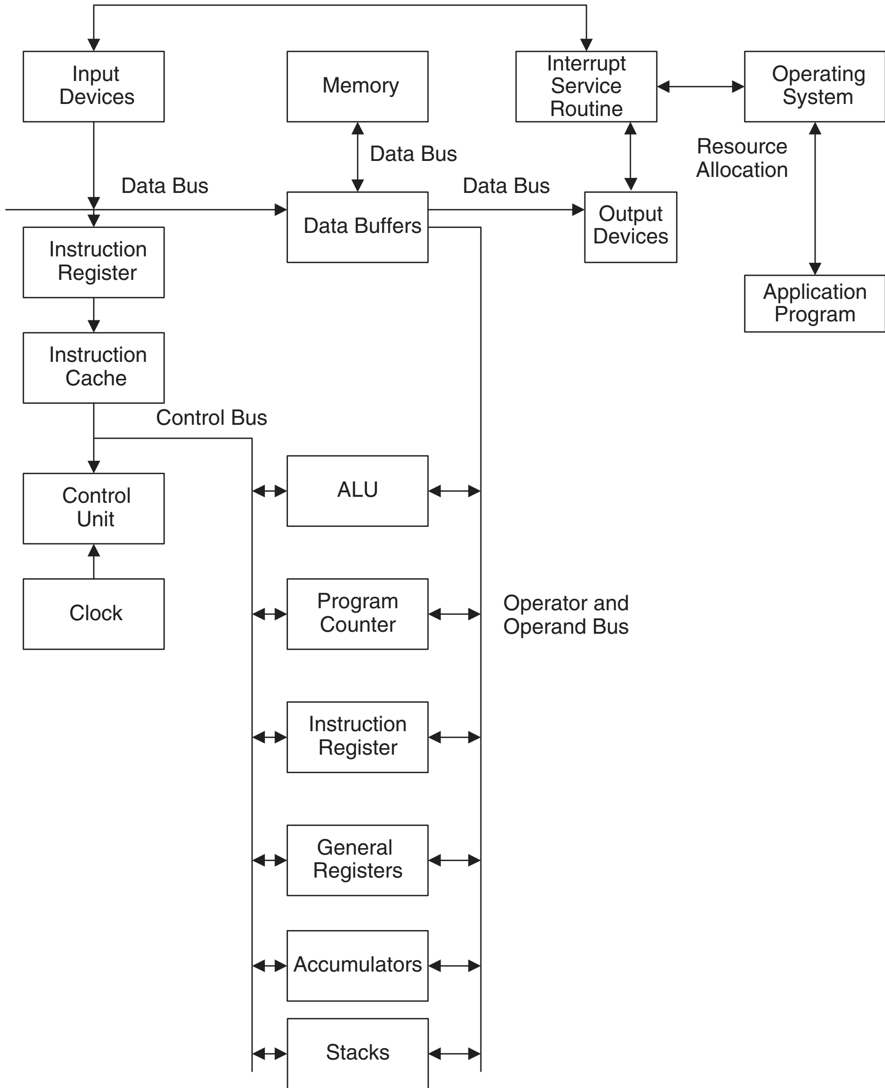


Figure 1.1 Microprocessor architecture.

and the *increase in speed* (number of clock cycles required in conventional system/ number of clock cycles required in a pipelined system) is:

$$(mn)/((n-1)+m) = n/(((n-1)/m)+1) = 16/7 = 2.286.$$

If m is large, the increase in speed approaches n clock cycles per instruction—maximum speed increase.

The pipeline *throughput* is defined as the *number of instructions*, m, per *total clock cycle time* required to process m instructions:

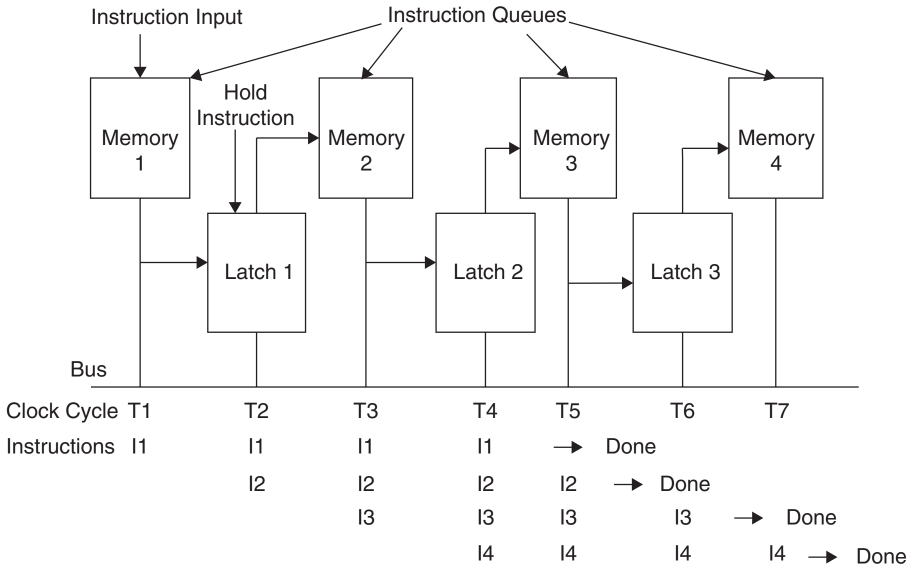


Figure 1.2 Pipelined system. n, clock cycle per instruction; m, instructions, each requiring one clock cycle; (n − 1) + m = 7 clock cycles (each instruction overlaps for [n − 1] clock cycles).

$$\frac{\text{m instructions}}{\text{Number of clock cycles per instruction} * \text{Time per clock cycle}} = \frac{\text{m}}{\text{m} + (\text{n} - 1)\text{T}},$$

where T is clock cycle time per instruction.

Problem: Compute the throughput of the pipeline microprocessor in Figure 1.2.

Answer: For a clock speed of 10 Mhz (10^7 clock cycles per second), T = $1/10^7$ seconds, the throughput is:

$$\text{m} / ((\text{m} + \text{n} - 1)\text{T}) = 4 / ((7)(1/107)) = (4)(107)/7 = 5.71 \text{ MIPS.}$$

Pipeline efficiency is computed as: speed increase/maximum speed increase (n = 4 clock cycles per instruction) = 2.286/4 = 0.5715.

Pipeline System Delay

When a pipeline instruction is unable to complete on the scheduled clock cycle, then

- Finish the earlier instructions on schedule and
- Delay the later instructions
- This is called stalling the pipeline

*Structural hazard*s are pipeline hardware delays.

Example: Memory does not respond to a request as fast as it is expected.

Data hazards arise when data are not ready in a pipeline at the time they are needed.

Example: An instruction needs data in a register that a previous instruction is still modifying.

Control hazards arise when the central processing unit (CPU) needs to manage a pipeline but instead must increment the program counter.

Example: Nonpipelined conditional branch instruction jumps to a pipelined instruction.

Problem: Delay in a pipelined operation is illustrated in this problem that compares the clock cycle delay for nonjump instructions with that of jump instructions.

If a jump instruction is executed in the pipelined CPU in Figure 1.2, what is the clock cycle delay?

Answer: Since the target of the jump instruction (another instruction) cannot be decoded (i.e., program counter updated) until the jump instruction is executed, there is a delay of three clock cycles.

Problem: What cam be done in a pipeline system to maintain performance when a *structural hazard* occurs?

Answer: More resources can be employed, if available, or the pipeline can be stalled (i.e., no instructions executed until needed hardware is available).

Problem: Is the microprocessor architecture in Figure 1.1 a pipeline computer?

Answer: No, it is not because only one instruction can be executed at a time.

Problem: What determines the clock cycle frequency of a pipeline system?

Answer: The clock cycle frequency of a *pipeline system* is governed by the *pipeline* with the slowest processing time. For example, whichever pipeline queue in Figure 1.2 experiences the slowest processing determines clock cycle frequency.

Operating System

The operating system contains the software necessary to manage the resources of a computer system. An example is a signal called an interrupt that is used to indicate to the microprocessor that an I/O device needs attention (i.e., data input or data output) or that there is an error condition (e.g., attempted divide by zero). The interrupt service routine is shown in Figure 1.1. In addition to managing resources, the operating system is responsible for allocating resources, for example, allocating memory to the application program, as depicted in Figure 1.1.

Memory

Because computer performance depends on the characteristics of memory systems in addition to the microprocessor architecture, it is important to consider the former

[HAR07]. Two important types of memory systems are main memory (random access memory, RAM) and secondary memory (hard disk, USB flash). Main memory can be divided between a relatively slow RAM for program and data access and a fast cache memory for accessing recently used instructions and data. In addition, secondary memory can be classified as virtual, meaning that pages on a hard disk can be mapped to main memory locations under the control of a memory management unit. A microprocessor may be equipped with special hardware, called direct memory access (DMA), which allows I/O devices to communicate directly with memory rather than using intermediate devices (such as data buffers in Fig. 1.1).

RAM

RAM contains bytes of information that the microprocessor can read or write, depending on whether the RD or WR line is activated. One problem with RAM chips is that they are volatile; the RAM contents are lost once the power goes off. That is why the microprocessor needs read-only memory (ROM).

ROM

All microprocessors contain ROM. A ROM chip is programmed with a permanent collection of preset bytes. The address bus tells the ROM chip which byte to read and place on the data bus. The RD line signal causes the ROM chip to transfer the selected byte to the data bus. On a personal computer, the program in the ROM is called the BIOS (basic input/output system). When the microprocessor starts, it begins executing instructions it finds in the BIOS. The BIOS instructions test the hardware, and then control is transferred to the hard disk to fetch the boot sector. The boot sector is another small program that the BIOS stores in RAM after reading it from the disk. The microprocessor then begins executing the boot sector's instructions from RAM. The boot sector program will tell the microprocessor to fetch more instructions from the hard disk into RAM, which the microprocessor then executes, and so on. This is how the microprocessor loads and executes the entire operating system.

Read/Write (R/W) Control Line

This single wire is driven by the microprocessor to control memory functions. If the R/W control line is asserted as a logical 1 (i.e., true), then the microprocessor performs a read operation. If it is asserted as a logic 0 (i.e., false), then the microprocessor performs a write operation. The relationship between logic level and voltage level can vary, depending on the implementation. For example, a logical 0 corresponds to a voltage of 0 V, and a logical 1 corresponds to a voltage of 5 V. Figure 1.3 is a block diagram of the microprocessor and memory, showing the R/W control line.

Address Bus

These wires are controlled by the microprocessor to select a particular location in memory for reading or writing. The microprocessor in Figure 1.3 uses a memory chip that has 15 address wires.

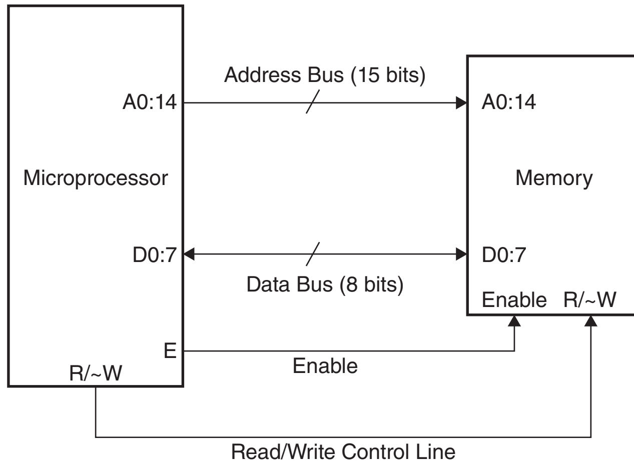


Figure 1.3 Diagram of microprocessor and memory.

Problem: How many locations can be addressed in Figure 1.3?

Answer: Since each wire has two states (it can be a digital 1 or a 0), $2^{15} = 32{,}768$ locations are possible. Thus, the system is said to have 32K of memory (1K = 1024 bytes).

Data Bus

These wires are used to pass data between the microprocessor and the memory. When data are written to the memory, the microprocessor drives the data bus; when data are read from the memory, memory drives the bus. In the example, in Figure 1.3, there are eight data wires (or bits). These wires can transfer one of 2^8 or 256 different binary values per transfer. The data size of 8 bits is commonly referred to as a byte. A data size of 4 bits is frequently referred to as a nibble.

Memory Enable Control Line

This wire, called the Enable line, connects to the enable circuitry of the memory in Figure 1.3. When the memory is enabled, it performs either a read or write operation as determined by the status of the R/W line.

Memory System Performance

Memory system performance is computed by considering hit and miss rates and the order of accessing memory components: cache memory, main memory, and hard disk. These rates are related to whether the instructions or data that are required by a program are available, first, in the cache memory, or second, in the main memory. If the instructions or data are in the cache, the access is scored as a cache hit; otherwise, the access is scored as a cache miss. Similarly, if the instructions or data

are not in the cache but are in main memory, the access is scored as a main memory hit; otherwise, the access is scored as a main memory miss because the instructions or data are only available on the hard disk [HAR07]. Thus, hit and miss rates are computed as follows:

$$\text{Cache hit rate (CHR)} = \frac{\text{Number of cache hits}}{\text{Total number of memory accesses}},$$

$$\text{Cache miss rate (CMR)} = \frac{\text{Number of cache misses}}{\text{Total number of memory accesses}},$$

$$\text{Main memory hit rate (MMHR)} = \frac{\text{Number of main memory hits}}{\text{Total number of memory accesses}},$$

$$\text{Main memory miss rate (MMMR)} = \frac{\text{Number of main memory misses}}{\text{Total number of memory accesses}},$$

$$\begin{aligned}\text{Number of hard disk accesses (HAD)} &= \text{Total number of memory accesses} \\ &- (\text{Number of cache memory hits} + \text{Number of main memory hits} \\ &+ \text{Number of main memory misses}).\end{aligned}$$

Note that when there is a cache memory miss, the main memory access is attempted. Thus, it is not necessary to count cache memory misses in the foregoing computation:

$$\text{Hard disk access rate (HDAR)} = \text{HAD} / \text{Total number of memory accesses.}$$

Problem: For example, consider the following case:

4000 total number of memory accesses

1200 cache accesses are hits and 800 are misses

Of the 800 cache misses that require access to the main memory, 200 are hits and 600 are misses

Compute CHR, CMR, MMHR, MMMR, HAD, and HDAR.

Answer: CHR = 1200/4000 = 30%

CMR = 800/4000 = 20%

MMHR = 200/4000 = 5%

MMMR = 600/4000 = 1%

HAD = 4000 − (1200 + 200 + 600) = 2000

HDAR = 1200/4000 = 50%

Another memory performance metric is average access time (AAT), which is computed as follows:

$$\begin{aligned}\text{AAT} &= \text{CHR} * (\text{cache access time}) \\ &+ \text{MMHR} * (\text{main memory access time}) + \text{HDAR} * (\text{hard disk access time}).\end{aligned}$$

Problem: For the following typical access times: cache = 2 ns, main memory = 60 ns, and hard disk = 35 ms, and using the above hit and miss access rates, compute the AAT.

Answer: AAT = (0.30)(2) + (0.04)(60) + (0.50)($35 * 10^6$) ns = $20.50 * 10^6$ ns (of course, hard disk access time dominates).

Multiplexing Data and Address Signals

On the Motorola 68HC11 microprocessor, in Figure 1.4, the 8-bit address/data bus takes turns acting as an address bus and a data bus. When a memory location is accessed (for reading or writing), the bus first acts as an address bus, transmitting the 8 lower-order bits of the address. Then the bus functions as a data bus, either transmitting a data byte (for a write cycle) or receiving a data byte (for a read cycle). This kind of split-personality bus is referred to as a multiplexed address and data bus. The support needed by the memory is provided by an 8-bit latch (a device that can store an address), using a multiplexed address/data bus. This chip (HC373) performs the function of latching the lower 8 address bits, when combined with the upper 7 address bits from the microprocessor, will provide the full 15-bit address for reading or writing data.

Figure 1.4 shows how the latch is wired. The upper 7 address bits run directly from the microprocessor to the memory. The lower 8 *address bits* are multiplexed with 8 *data bits*. When an *address* appears on the wires AD: 07, the latch connects the address bits of the microprocessor to the memory. On the other hand, when *data* appears on the wires AD0:7, the latch connects the data bits of the microprocessor to the memory. An additional signal, the address strobe (AS) output of the

Figure 1.4 Block diagram of microprocessor and memory with latch.

microprocessor, tells the latch when to obtain the address bits from the address/data bus. When the full 15-bit address is available to the memory (upper 7 bits direct from the microprocessor (wires A8: 14) and lower 8 bits from the latch (wires AD: 07), the read or write access can occur. Because the address/data bus is also wired directly to the memory, data can flow in either direction between the memory and the microprocessor. The entire process is managed by the microprocessor. The Enable (E) clock, the R/W line, and the AS line perform in tight synchronization to make sure these operations happen in the correct sequence and within the timing capacities of the microprocessor hardware.

Memory Mapping the RAM

Memory mapping refers to allocating blocks of memory to different functions, such as the operating system and the application program. If a microprocessor has 15 address bits, it has 32,728 (32K bytes) of addressable locations that can be mapped. This address space would be used by the 32K memory chip in Figure 1.5. The technique used to map the memory is fairly simple. Whenever the microprocessor's A15 (the highest order address bit) is logic 1, the high-order address bit is selected. The other 15 address bits (A0 through A14) determine the address within that 32K block. If A15 is logic 0, the 32K block is not selected.

A NAND gate (actually a portion of a programmable logic device called a PAL) is used to enable the memory when A15 and the E Clock equal 1 in Figure 1.5. (See the "Digital Logic" section below for the explanation of NAND and other gates).

The E Clock controls the timing of the chip enable line. Some memory chips use an active low (sometimes called "active false") signal to enable inputs, meaning that they are enabled when the enable input is 0. The method for denoting an input that is active low (i.e., 0) is shown in Figure 1.5, where the chip enable input connects to a circle; this circle indicates an active low input. Also, the name for the signal, CE, is prefixed with a ~ symbol.

Interrupt Handling

The microprocessor has a bank of interrupt vectors, as shown in Figure 1.5, which are hardware-defined locations in the memory address space where the microproces-

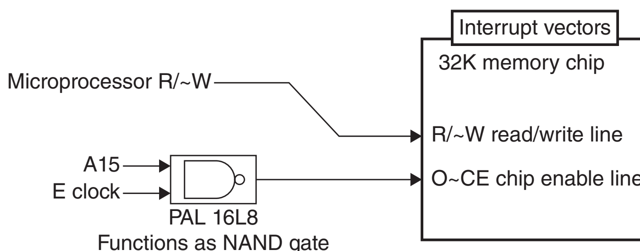


Figure 1.5 Enabling the memory.

Table 1.1 NOT Truth Table

Input	Output
A	$\overline{A}$
0	1
1	0

sor expects to find pointers to interrupt handling routines, for processing input and output data, arithmetic overflow, and so on. Also, when the microprocessor is reset, it finds the reset vector to determine where it should begin running a program. These vectors are located in the address space of the memory.

DIGITAL LOGIC

The fundamental logic operations of a microprocessor are performed by the following circuits. The results of those operations are represented in truth tables, where the binary value 0 is considered "low" (e.g., low voltage) and the binary value 1 is considered "high" (e.g., high voltage). While digital logic is used in the design of microprocessors, "everyday" examples are provided to show that the logic operations are not restricted to microprocessors.

NOT: represented in Table 1.1 and implemented with an inverter in Figure 1.6.

Application: The application is to complement the input A, producing the output $\overline{A}$.

Microprocessor example: the binary bit input was caused by an arithmetic overflow condition, so it is ignored and *not* used in the computation.

Everyday example: if we are to leave on an automobile trip, where A = 1 represents leaving at 1000, $\overline{A} = 0$ represents all times *not* equal to 1000.

OR: represented in Table 1.2 and implemented with OR gate in Figure 1.6.

Application: The application is to produce a 1 output if *any* or *both* of the inputs are 1.

Microprocessor example: the inputs are binary bits from memory stick or hard disk, so the microprocessor can accept *either* or *both* to perform a computation, depending on the current computer program instruction.

Everyday example: if A = 1 represents the decision to purchase a house and B = 1 represents the decision to purchase an automobile, Z = 1 represents the decision to purchase a house *or* an automobile *or* both.

AND: represented n Table 1.3 and implemented with an AND gate in Figure 1.6.

Application: The application is to produce a 1 output if *all* inputs are 1.

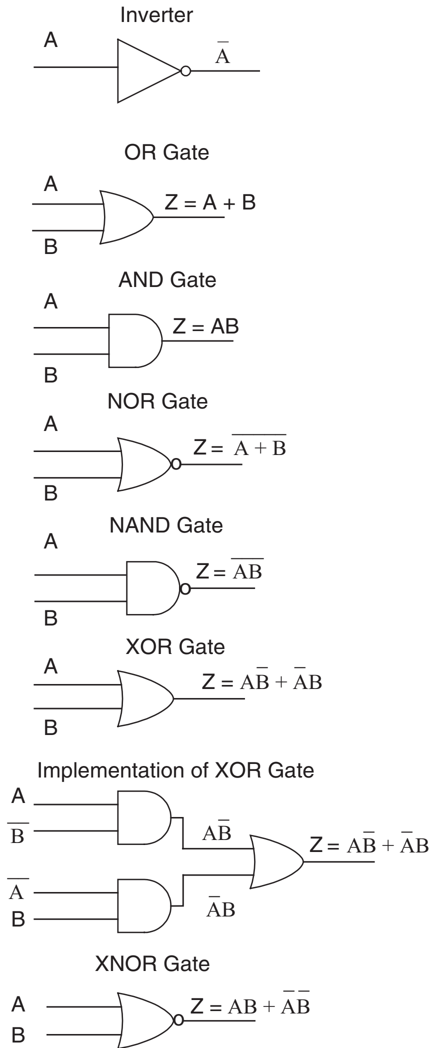


Figure 1.6 Logic operations.

Table 1.2 OR Truth Table

Input	Input	Output
A	B	Z = A + B
0	0	0
0	1	1
1	0	1
1	1	1

Table 1.3 AND Truth Table

Input	Input	Output
A	B	Z = AB
0	0	0
0	1	0
1	0	0
1	1	1

Table 1.4 NOR Truth Table

Input	Input	Output
A	B	$Z = \overline{A} + \overline{B}$
0	0	1
0	1	0
1	0	0
1	1	0

Table 1.5 NAND Truth Table

Input	Input	Output
A	B	$Z = \overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0

Microprocessor example: the microprocessor uses a signal Z = 1 to tell it that an interrupt has occurred on input line A *and* signifying that data input occurs on B, which the microprocessor will transfer to its memory.

Everyday example: if A = 1 represents a gas station and B = 1 represents a restaurant, we would stop our automobile at location Z, if Z has *both* a gas station *and* a restaurant.

NOR: represented in Table 1.4 and implemented with NOR gate in Figure 1.6.

Application: The application is to produce a 1 output if all inputs are 0.

Microprocessor example: the microprocessor Z = 1 output is recognized as interrupt code AB = 00.

Everyday example: if A = 0 represents the decision to *not* purchase a home and B = 0 represents the decision *not* to purchase an automobile, then Z = 1 represents the decision to *neither* purchase a home *nor* purchase an automobile.

NAND: represented in Table 1.5 and implemented with NAND gate in Figure 1.6.

Application: The application is to produce a 1 output if all inputs are *not* 1.

Microprocessor example: the microprocessor program produces the complement of the product of binary bits. This would be the case, for example, when Z = 1 signals that 0s occur on *either or both* of two input channels.

Everyday example: if A = 1 represents a gas station and B = 1 represents a restaurant, we would stop our automobile at location Z, if Z has only a gas station, or has only a restaurant, or has neither (i.e., rest stop).

Exclusive OR (XOR): represented in Table 1.6 and implemented with EXCLUSIVE OR gate in Figure 1.6. The figure also shows how the gate can be implemented, using AND and OR gates.

Application: The application is to produce a 1 output if *any* of the inputs is 1, but *not all* inputs are 1, and *not all* inputs are 0.

Microprocessor example: the main microprocessor receives a signal Z = 1 from the output of the I/O microprocessor that a binary bit A = 1 from a memory stick *or* B = 1 from a hard disk, and is ready for input, but these inputs are *not concurrent*.

Everyday example: if A = 1 represents the decision to purchase a house and B = 1 represents the decision to purchase an automobile, Z = 1 represents the decision to purchase a house *or* an automobile, but *not both at the same time*.

Exclusive NOR (XNOR): represented in Table 1.7 and implemented with XNOR gate in Figure 1.6. The *NOR* gate is the negation of the *XOR* gate from Table 1.6, as indicated in Table 1.7.

Table 1.6 EXCLUSIVE OR Truth Table

Input	Input	Output
A	B	$Z = A\overline{B} + \overline{A}B$
0	0	0
0	1	1
1	0	1
1	1	0

Table 1.7 EXCLUSIVE NOR (XNOR) Truth Table

Input	Input	Output
A	B	$Z = \overline{A\overline{B} + \overline{A}B} = (\overline{A\overline{B}})(\overline{\overline{A}B}) = (\overline{A} + B)(A + \overline{B}) = \overline{A}A + AB + \overline{A}\overline{B} + B\overline{B} = AB + \overline{A}\overline{B}$
0	0	1
0	1	0
1	0	0
1	1	1

Application: The application is to produce a 1 output if all inputs are 0 *or* all inputs are 1.

Microprocessor example: Two hard drives are identified as A = 0 and A = 1; two flash memories are identified as B = 0, and B = 1. The microprocessor is programmed to input data from a hard drive and a flash memory *concurrently*. Therefore, it reads A = 0 *and* B = 0 *or* A = 1 *and* B = 1.

Everyday example: if A = 1 represents a gas station and B = 1 represents a restaurant, we would stop our automobile at location Z, if Z has *neither* a gas station *nor* a restaurant (i.e., rest stop) *or* has *both* a gas station and restaurant (i.e., get gas and eat).

De Morgan's theorem [GRE80] is used to simplify complex logic equations and the resultant digital logic. The theorem is used to simplify relatively simple expressions, as contrasted with Karnaugh maps (K-maps), described in the next section. The application of this theorem is shown in the following example:

$$\text{Theorem: } \overline{A+B}=\overline{A}\,\overline{B} \text{ and } \overline{AB}=\overline{A}+\overline{B}.$$

Suppose it is required to simplify $F=\overline{((\overline{AB})(\overline{AB}))}$.

Applying the theorem:

$$\overline{AB}=\overline{A}+\overline{B},(\overline{AB})(\overline{AB})=(\overline{A}+\overline{B})(\overline{A}+\overline{B})$$

$$=\overline{A}\ \overline{A}+\overline{A}\ \overline{B}+\overline{A}\ \overline{B}+\overline{B}\ \overline{B}=\overline{A}+\overline{A}\ \overline{B}+\overline{B}=\overline{A}+(\overline{A}+1)\overline{B}=\overline{A}+\overline{B}$$

$$F=\overline{(\overline{A}+\overline{B})}(\overline{A}+\overline{B}=(\overline{\overline{A}}+\overline{\overline{B}})+(\overline{\overline{A}}+\overline{\overline{B}})=AB+AB=B.$$

Then, use Table 1.8 to demonstrate the equivalence between $\overline{((\overline{AB})(\overline{AB}))}$ and AB.

K-MAPS

A K-map in Table 1.9 is used to minimize a complex Boolean expression [RAF05]. Each square of a K-map represents a minterm (i.e., product terms). The process proceeds by listing the binary equivalents of the terms A and BC on the axes of Table 1.9, ordering them so that there is only a 1-bit difference between adjacent cells. Then, the minimum number of cells is enclosed. Next, minterms are identified

Table 1.8 Truth Table to Demonstrate Equivalence between F and AB

A	B	$\overline{AB}$	$\overline{AB}\,\overline{AB}$	$F=\overline{((\overline{AB})(\overline{AB}))}$	AB
0	0	1	1	0	0
0	1	1	1	0	0
1	0	1	1	0	0
1	1	0	0	1	1

according to terms that are common to all cells in the enclosure. Last, the product terms are summed. Notice what a clever method this is. Minimization is achieved by noting the combination of terms that yields the minimum difference!

Example: Simplify $F=\overline{A}\ \overline{B}\ \overline{C}+A\ \overline{B}\ \overline{C}+\overline{A}\ \overline{B}\ C+A\ \overline{B}\ C$.

Table 1.9 K-Map for $F=\overline{A}\overline{B}\overline{C}+A\overline{B}\overline{C}+\overline{A}\overline{B}C+A\overline{B}C$

		$\overline{B}\,\overline{C}$	$\overline{B}C$	BC	$B\overline{C}$
		00	01	11	10
$\overline{A}$	0	1	1		
A	1	1	1		

In minterm form, $F=\overline{A}\ \overline{B}\ \overline{C}+A\ \overline{B}\ \overline{C}+\overline{A}\ \overline{B}\ C+A\ \overline{B}\ C=\overline{B}$

In the K-map, $\overline{B}$ is common to the enclosed minterms. Therefore, $F=\overline{B}$. Table 1.10 demonstrates this result. The considerable reduction from the original function would result in significant savings in circuitry to implement the function.

Prime Implicant

A prime implicant is the *product term* obtained by enclosing the *maximum* number of adjacent cells in a K-map. For example, in the K-map of Table 1.9, $F=\overline{B}$ is a prime implicant. The prime implicant is only useful for providing a name for the maximum enclosure in a K-map.

Quine–McCluskey Method

This method is an alternative to the K-map for minimizing a Boolean function. The method is illustrated in Table 1.11 by minimizing the function $F=\overline{A}\ \overline{B}\ \overline{C}+A\ \overline{B}\ \overline{C}+\overline{A}\ \overline{B}\ C+A\ \overline{B}\ C$, where these minterms are placed in Table

Table 1.10 F Function Truth Table

A	B	C	$F=\overline{A}\ \overline{B}\ \overline{C}+A\ \overline{B}\ \overline{C}+\overline{A}\ \overline{B}\ C+A\ \overline{B}\ C$	$\overline{F}=\overline{B}$
0	0	0	1	1
0	0	1	1	1
0	1	0	0	0
0	1	1	0	0
1	0	0	1	1
1	0	1	1	1
1	1	0	0	0
1	1	1	0	0

Table 1.11 Quine–McCluskey Method for $F = \overline{A}\,\overline{B}\,\overline{C} + A\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}\,C + A\,\overline{B}\,C = \overline{B}$

			Difference of 1		Difference of 1		
Minterm		ABC	Minterms		Minterms	Minterms	Prime implicant
0	$\overline{A}\,\overline{B}\,\overline{C}$	000	0,1	00-	0,1,4,5	-0-	$\overline{B}$
1	$\overline{A}\,\overline{B}\,C$	001					
4	$A\,\overline{B}\,\overline{C}$	100	4,5	10-			
5	$A\,\overline{B}\,C$	101					

Table 1.12 One-Bit Adder Truth Table

A	B	Q	CO
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

1.11. This method is used to represent a difference of 1 between two adjacent minterms, such as $\overline{A}\,\overline{B}\,\overline{C}$ and $\overline{A}\,\overline{B}\,C$, yielding $\overline{A}\,\overline{B}$- = 00- . The symbol - is placed where there is a difference in minterm bit values, such as between 00- and 10- in Table 1.11, yielding -0-. This process continues until the four minterms 0, 1, 4, and 5 show a difference of 1 (00- compared with 10-), yielding prime implicant $\overline{B}$(-0-). The same result is obtained as was the case using the K-map in Table 1.9. Of the two methods, the K-map is easier to apply.

COMBINATIONAL CIRCUITS

These are circuits that use logic gates to produce outputs at any time that are only dependent on the *current* values of the inputs, meaning that it is not necessary to use a CP to trigger outputs [HAR07]. A typical combinational circuit is the adder.

One-Bit Adder with Carry Out

A and B are added, producing Q output and CO (carry out). Q and CO are implemented according to the truth table shown in Table 1.12.

Two-bit Adder with Carry In and CO

What if you want to add two 8-bit bytes? This becomes slightly harder. In this case, you need to create a full binary adder. The difference between a full adder and the

Table 1.13 Two-Bit Adder Truth Table

					Q = 1	CO = 1
CI	A	B	Q	CO	Minterms	Minterms
0	0	0	0	0		
0	0	1	1	0	$\overline{CI}\ \overline{A}\ B$	
0	1	0	1	0	$\overline{CI}\ A\ \overline{B}$	
0	1	1	0	1		$\overline{CI}\ A\ B$
1	0	0	1	0	$CI\ \overline{A}\ \overline{B}$	
1	0	1	0	1		$CI\ \overline{A}\ B$
1	1	0	0	1		$CI\ A\ \overline{B}$
1	1	1	1	1	CI A B	CI A B

Q Product Terms: $\overline{CI}\ \overline{A}\ B + \overline{CI}\ A\ \overline{B} + CI\ \overline{A}\ \overline{B} + CIAB$

$Q = \overline{CI}\ (\overline{A}\ B + A\ \overline{B}) + CI\ (\overline{A}\ \overline{B} + AB)$

CO Product Terms: $\overline{CI}A\ B + CI\ \overline{A}\ B + CI\ A\ \overline{B} + CI\ A\ B = AB\ (\overline{CI} + CI) + CI(\overline{A}\ B + A\ \overline{B})$

$CO = AB + CI\ (\overline{A}\ B + A\ \overline{B})$

Table 1.14 K-Map for $Q = \overline{CI}\ \overline{A}\ B + \overline{CI}\ A\ \overline{B} + CI\ \overline{A}\ \overline{B} + CIAB = \overline{CI}(\overline{A}\ B + A\ \overline{B}) + CI(\overline{A}\ \overline{B} + AB)$

	AB			
CI	00	01	11	10
0		1		1
0	1		1	

$CI\overline{A}\,\overline{B}$ $\overline{CI}\,\overline{A}\,B$ $CIAB$ $\overline{CI}A\,\overline{B}$

1-bit adder is that a full adder accepts A and B inputs plus a carry-in (CI) input. Once you have a full adder, you can string eight of them together to create a byte-wide adder and cascade the carry bit from one adder to the next. The truth table for a full adder is slightly more complicated than the previous truth table because now there are 3 input bits.

A combinational circuit minterm is represented by a product in a row of the truth table as shown in Table 1.13, corresponding to a 1 in the Q or CO output columns; for example, the fourth row for CO and the second row for Q in Table 1.13 [GIB80]. The values of Q and CO product terms are obtained by ORing the products in each row of Table 1.13 where Q = 1 or CO = 1, and then summing these terms, followed by simplifying the expressions, as demonstrated in Table 1.13. Further simplification *may* be possible by using a K-map.

As can be seen in Table 1.14, the adder output Q cannot be simplified by using a K-map because there are no adjacent cells. However, simplification is achieved

Table 1.15 K-Map for Carry Out $(CO) = \overline{CI}AB + CI\overline{A}B + CIA\overline{B} + CIAB = AB + CI(\overline{A}B + A\overline{B})$

	AB			
CI	00	01	11	10
0			1	
1		1	1	1

$CI\,\overline{A}\,B$ $\overline{CI}AB$ $CIAB$ AB $CIA\,\overline{B}$

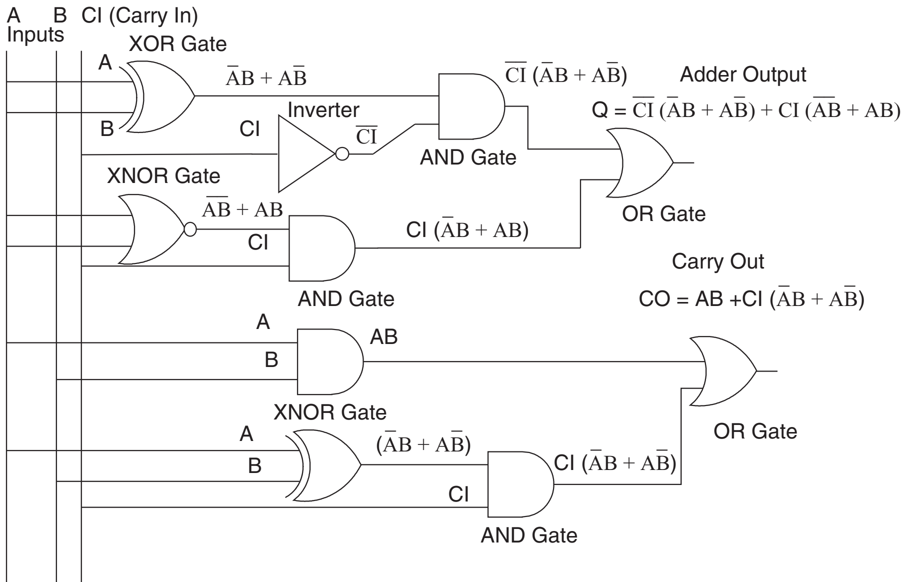


Figure 1.7 Adder circuit.

for CO, as shown in Table 1.15, producing $CO = AB + CI(\overline{A}B + A\overline{B})$. The relevant minterm cells in Table 1.15 that comprise the minimized function are outlined in red. Minterm logic is called *sum of products*. The full adder logic that corresponds to the minterms in Table 1.13 is shown in Figure 1.7, showing the adder output Q and the CO.

MULTIPLE OUTPUT COMBINATIONAL CIRCUITS

Combinational circuits can have multiple outputs [RAF05]. Each output is expressed as a function of the inputs, as shown in Table 1.16, where the inputs are binary-coded decimal (BCD) bits W, X, Y, and Z, corresponding to the decimal digits 0,…, 9. A

Table 1.16 Truth Table for Binary-Coded Decimal (BCD) Converter

Decimal digit	BCD input bits				Computer display segment output bits						
	W	X	Y	Z	a	b	c	d	e	f	g
0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	0
1	0	0	0	1	0	**1**	**1**	0	0	0	0
2	0	0	1	0	**1**	**1**	0	**1**	**1**	0	**1**
3	0	0	1	1	**1**	**1**	**1**	**1**	0	0	**1**
4	0	1	0	0	0	**1**	**1**	0	0	**1**	**1**
5	0	1	0	1	**1**	0	**1**	**1**	0	**1**	**1**
6	0	1	1	0	0	0	**1**	**1**	**1**	**1**	**1**
7	0	1	1	1	**1**	**1**	0	**1**	0	0	0
8	1	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**
9	1	0	0	1	**1**	**1**	**1**	0	0	**1**	**1**

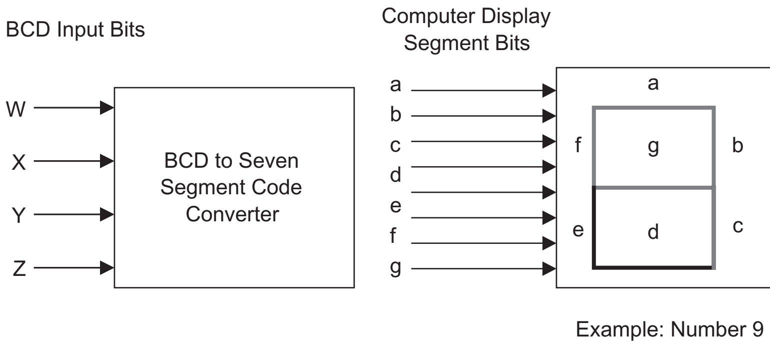


Figure 1.8 BCD to seven-segment code converter.

binary coded decimal converter is an example shown in Figure 1.8, showing how the number 9 can be displayed. The outputs are computer display segment bits a,…, g that represent the 1s necessary to generate the display decimal numbers. The code converter transforms the BCD numbers 0000,…, 1001 to display segments. The converter does not represent decimal numbers greater than 9. The K-maps use “don’t cares” = Xs in order to simplify the logic; the “don’t cares” should not be confused with the BCD bit = X in Table 1.16. The “don’t cares” are used to advantage in forming minterms, as, for example, in Tables 1.17–1.23.

In order to generate the K-maps, place a 1 in the K-map cells corresponding to the 1s that appear in Table 1.16. For example, for *segment a* in Table 1.17, a 1 is recorded in the cell WXYZ = 0000, corresponding to the **1** (bolded) in the *segment a* column in Table 1.16.

The K-maps will lead to simplifying the equations for the seven-segment computer display (Fig. 1.8). The equations will then be used to design the digital logic circuit in Figures 1.9 and 1.10.

Table 1.17 K-Map for Segment a

	YZ			
WX	00	01	11	10
00	1		1	1
01		1	1	
11	X	X	X	X
10	1	1	X	X

W $\overline{W}\,\overline{X}\,\overline{Z}$ XZ   YZ

$a = W + \overline{W}\ \overline{X}\ \overline{Z} + Z(X + Y).$

Table 1.18 K-Map for Segment b

	YZ			
WX	00	01	11	10
00	1	1	1	1
01	1		1	
11	X	X	X	X
10	1	1	X	X

$\overline{Y}\,\overline{Z}$ W $\overline{W}\,\overline{X}$ YZ

$b = W + \overline{W}\ \overline{X} + YZ + \overline{Y}\ \overline{Z}.$

Table 1.19 K-Map for Segment c

	YZ			
WX	00	01	11	10
00	1	1	1	
01	1	1		1
11	X	X	X	X
10	1	1	X	X

W $\overline{Y}$ $\overline{X}\,YZ$ $XY\,\overline{Z}$

$c = W + \overline{Y} + \overline{X}\ YZ + XY\ \overline{Z} = W + \overline{Y} + Y(\overline{X}Z + X\overline{Z}).$

Table 1.20 K-Map for Segment d

	YZ			
WX	00	01	11	10
00	1		1	1
01		1	1	1
11	X	X	X	X
10	1		X	X

$\overline{X}\,\overline{Y}\,\overline{Z}$ $X\overline{Y}Z$ Y

$d = \overline{X}\,\overline{Y}\,\overline{Z} + X\overline{Y}Z + Y = \overline{Y}(\overline{X}\,\overline{Z} + XZ) + Y.$

Table 1.21 K-Map for Segment e

	YZ			
WX	00	01	11	10
00	1			1
01				1
11	X	X	X	X
10	1		X	X

$\overline{X}\,\overline{Y}\,\overline{Z}$ $Y\overline{Z}$

$e = \overline{Z}(\overline{X}\,\overline{Y} + Y).$

Table 1.22 K-Map for Segment f

	YZ			
WX	00	01	11	10
00	1			
01	1	1		1
11	X	X	X	X
10	1	1	X	X

$\overline{Y}\,\overline{Z}$ W $X\overline{Y}$ $XY\overline{Z}$

$f = \overline{Z}(\overline{Y} + XY) + W + X\overline{Y}.$

Table 1.23 K-Map for Segment g

	YZ			
WX	00	01	11	10
00			1	1
01	1	1		1
11	X	X	X	X
10	1	1	X	X

W $\overline{W}X\overline{Y}$ $\overline{W}\,\overline{X}Y$ $Y\overline{Z}$

$g = \overline{W}(X\overline{Y} + \overline{X}Y) + W + Y\overline{Z}$.

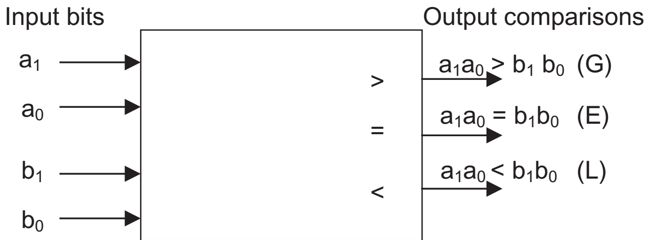


Figure 1.9 Two-bit comparator block diagram.

Comparators

A comparator is another type of combinational circuit. Its block diagram is shown in Figure 1.9 and the corresponding logic diagram is shown in Figure 1.10. For example, as Figure 1.10 shows, a comparator can be designed to compare two 2-bit quantities for greater-than (G), equal-to (E), and less-than (L) conditions. By minimizing the logic in Table 1.24, as accomplished by the K-maps in Tables 1.25–1.27, the logic circuit is designed in Figure 1.9. The K-maps are generated by recording a 1 in cells corresponding to 1s in Table 1.24; for example, placing a 1 in the cells a_1, a_0, b_1, and b_0 = 0100 for G in Table 1.24. Notice, as opposed to previous examples, there are no "don't care" conditions because all four comparator bits are relevant.

Decoders

A decoder is a combinational circuit that, when enabled, selects one of 2^n inputs and produces a 1 output, where n is the number of input bits, as shown in Figure 1.11. After this block diagram is displayed, the truth table (Table 1.28), is formulated, showing the relationship between inputs and outputs, where an output term 1 is

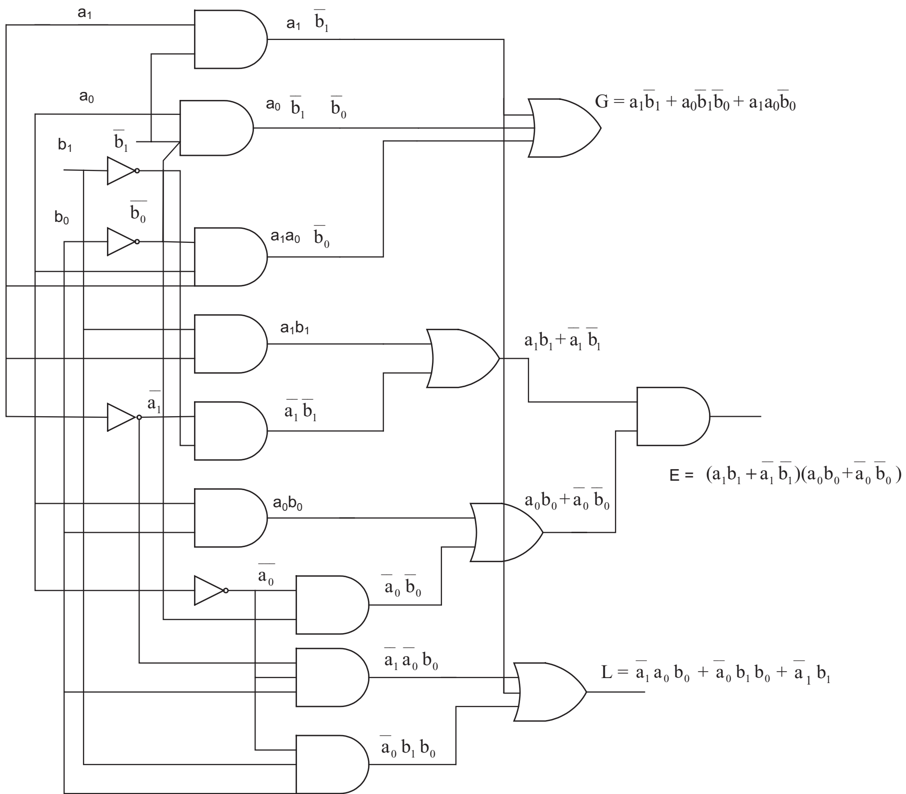


Figure 1.10 Two-bit comparator logic diagram.

generated according to the appearance of 0s and 1s in the inputs columns; for example, $d_3 = E\,\bar{x}_1\,\bar{x}_0 = 1$ for $E\,\bar{x}_1\,\bar{x}_0 = 100$.

Finally, Table 1.28 is used to design the logic diagram in Figure 1.11. Applying K-maps to minimize the logic of the truth table is not necessary because there is only a single 1 output for each combination of inputs in Table 1.28. However, the truth table is used to generate the output equations, which will lead to the design of the logic diagram in Figure 1.11. An application of the decoder is to select an operand (i.e., 4-bit output $d_0d_1d_2d_3$) in a computer instruction, based on the operation code (i.e., 2-bit input x_1x_0) in the instruction, when the instruction execution enable is high (E = 1).

Encoders

Encoders produce n output bits in accordance with the value of 2^n input bits, as shown in the block diagram of Figure 1.12. Like the decoder, it is not necessary to develop K-maps of the outputs as a function of the inputs because of the inherent simplicity of the circuit logic in Figure 1.12. Equations that emerge from the

Table 1.24 Truth Table for Two-Bit Comparator

Inputs				Outputs		
a_1	a_0	b_1	b_0	G: $a_1a_0 > b_1b_0$	E: $a_1a_0 = b_1b_0$	L: $a_1a_0 < b_1b_0$
0	0	0	0	0	**1**	0
0	0	0	1	0	0	**1**
0	0	1	0	0	0	**1**
0	0	1	1	0	0	**1**
0	1	0	0	**1**	0	0
0	1	0	1	0	**1**	0
0	1	1	0	0	0	**1**
0	1	1	1	0	0	**1**
1	0	0	0	**1**	0	0
1	0	0	1	**1**	0	0
1	0	1	0	0	**1**	0
1	0	1	1	0	0	**1**
1	1	0	0	**1**	0	0
1	1	0	1	**1**	0	0
1	1	1	0	**1**	0	0
1	1	1	1	0	**1**	0

Table 1.25 K-Map for Output G:$a_1a_0 > b_1b_0$

Inputs		Inputs b_1b_0			
		00	01	11	10
a_1a_0	00				
	01	1			
	11	1	1		1
	10	1	1		

$a_0\overline{b_1}\,\overline{b_0}$ $a_1\overline{b_1}$ $a_1a_0\overline{b_0}$

$G = a_0\overline{b_1}\,\overline{b_0} + a_1\overline{b_1} + a_1a_0\overline{b_0}.$

Table 1.26 K-Map for Output E:$a_1a_0 = b_1b_0$

Inputs		Inputs b_1b_0			
		00	01	11	10
a_1a_0	00	1			
	01		1		
	11			1	
	10				1

$\overline{a_1}\,\overline{a_0}\,\overline{b_1}\,\overline{b_0}$ $\overline{a_1}a_0\overline{b_1}b_0$ $a_1a_0b_1b_0$ $a_1\overline{a_0}b_1\overline{b_0}$

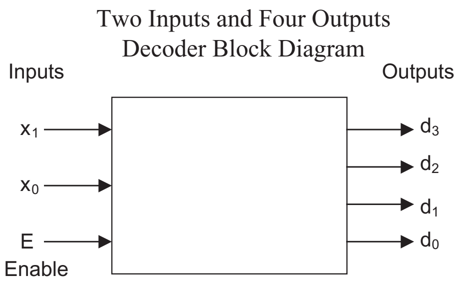
$$E = \overline{a_1}\,\overline{a_0}\,\overline{b_1}\,\overline{b_0} + \overline{a_1}a_0\overline{b_1}b_0 + a_1a_0b_1b_0 + a_1\overline{a_0}b_1\overline{b_0}$$
$$= \overline{a_1}\,\overline{b_1}(a_0b_0 + \overline{a_0}\,\overline{b_0}) + a_1b_1(a_0b_0 + \overline{a_0}\,\overline{b_0}) = (a_1b_1 + \overline{a_1}\,\overline{b_1})(a_0b_0 + \overline{a_0}\,\overline{b_0}).$$

Table 1.27 K-Map for Output $L: a_1a_0 < b_1b_0$

Inputs		Inputs b_1b_0			
		00	01	11	10
a_1a_0	00		1	1	1
	01			1	1
	11				
	10			1	

$\bar{a}_1 \bar{a}_0 b_0$ $\bar{a}_0 b_1 b_0$ $\bar{a}_1 b_1$

$$L = \bar{a}_1\bar{a}_0 b_0 + \bar{a}_0 b_1 b_0 + \bar{a}_1 b_1.$$



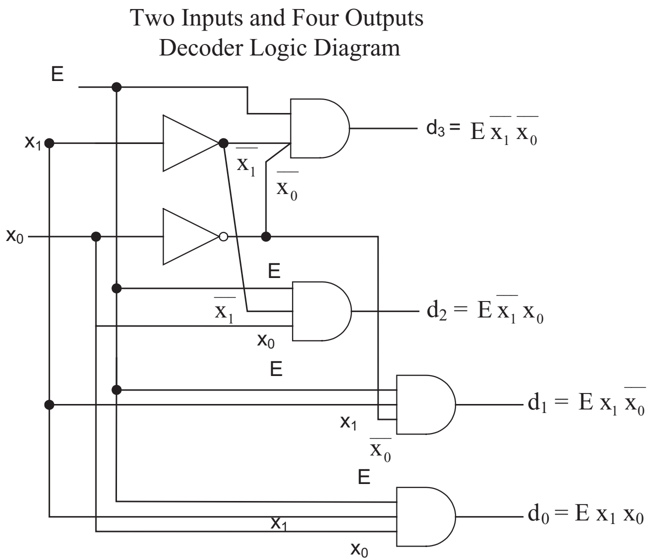


Figure 1.11 Two inputs and four outputs decoder block and logic diagrams.

Table 1.28 Truth Table for Two Inputs and Four Outputs Decoder

Inputs			Outputs			
E (Enable)	x_1	x_0	d_3	d_2	d_1	d_0
1	0	0	**1**	0	0	0
1	0	1	0	**1**	0	0
1	1	0	0	0	**1**	0
1	1	1	0	0	0	**1**

$d_3 = E\overline{x_1}\,\overline{x_0}$; $d_2 = E\overline{x_1}x_0$; $d_1 = Ex_1\overline{x_0}$; $d_0 = Ex_1x_0$.

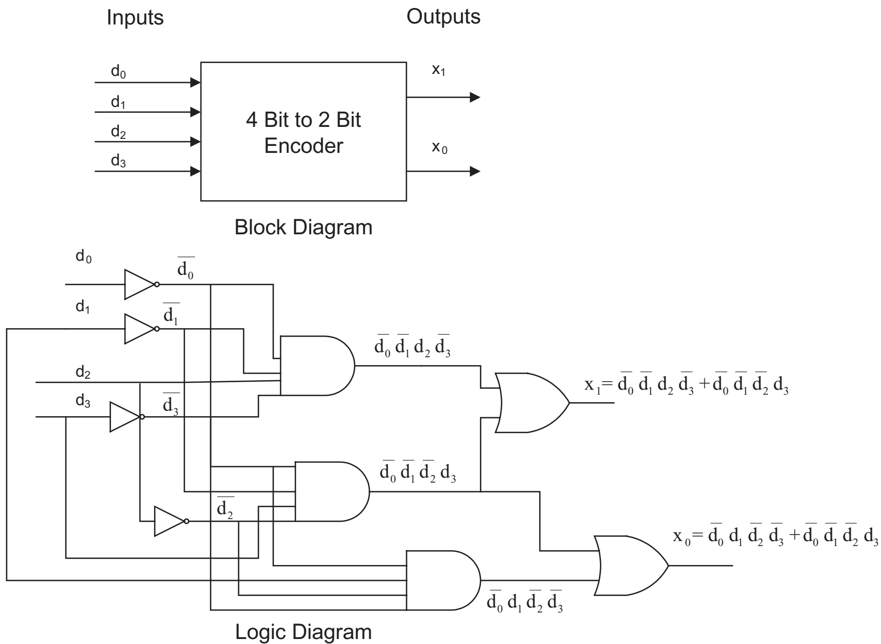


Figure 1.12 The 4-bit to 2-bit encoder block and logic diagrams.

relationships in the truth table (Table 1.29) are used to design the logic circuit in Figure 1.12. The outputs x_1 and x_0 are generated as the sum of the products of inputs where there are **1**s in the x_1 and x_0 columns as signified by the bolded quantities.

An application of the encoder is data compression in which we could shrink 4 bits of input to 2 bits of output in a database application that deals with large quantities of data. For example, representing $d_0d_1d_2d_3 = 0100$ as $x_1x_0 = 01$.

Table 1.29 Truth Table for 4-Bit to 2-Bit Decoder

Inputs				Outputs	
d_0	d_1	d_2	d_3	x_1	x_0
1	0	0	0	0	0
0	1	0	0	0	**1**
0	0	1	0	**1**	0
0	0	0	1	**1**	**1**

$x_1 = \overline{d_0}\,\overline{d_1}d_2\overline{d_3} + \overline{d_0}\,\overline{d_1}\,\overline{d_2}d_3$, $x_0 = \overline{d_0}d_1\overline{d_2}\,\overline{d_3} + \overline{d_0}\,\overline{d_1}\,\overline{d_2}d_3$.

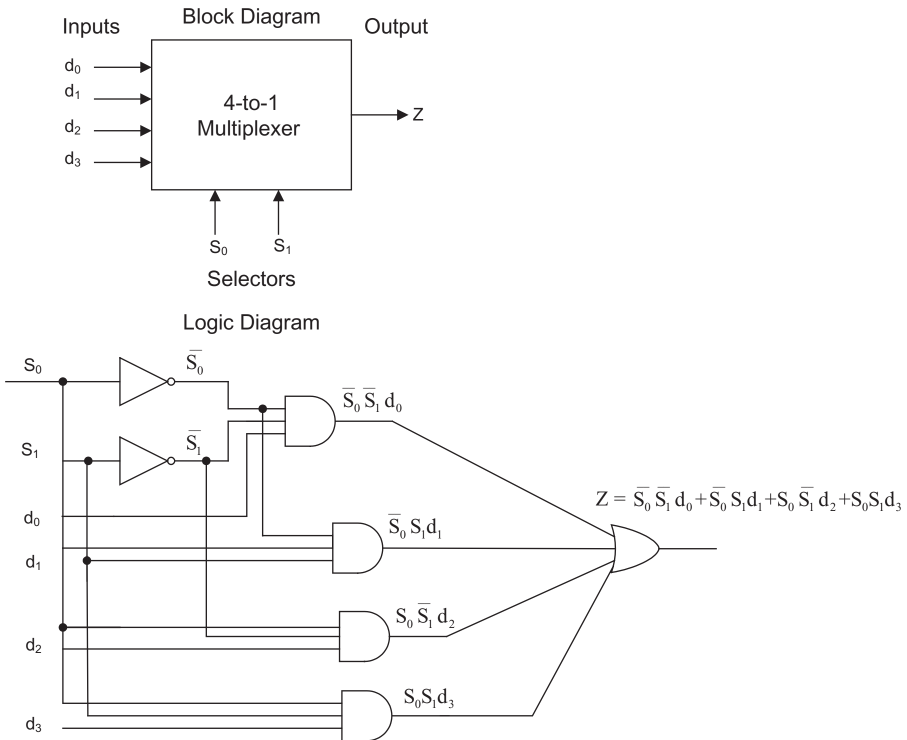


Figure 1.13 The 4-to-1 multiplexer block and logic diagrams.

Multiplexers

A multiplexer acts as a data selector, meaning that if the multiplexer has n select lines, one of 2^n inputs can be selected as the output. For example, in Figure 1.13, using selector lines S_0 and S_1, one of four inputs, d_0, d_1, d_2, d_3, can be selected at the output Z. The output equation for Z is derived from Table 1.30, noting that a given output is produced for given values of the selectors, for example, $Z = d_0$ when

Table 1.30 Truth Table for 4-to-1 Multiplexer

	Selector	Output
S0	S1	Z
0	0	d_0
0	1	d_1
1	0	d_2
1	1	d_3

$Z = \overline{S_0}\,\overline{S_1}d_0 + \overline{S_0}S_1d_1 + S_0\overline{S_1}d_2 + S_0S_1d_3.$

$\overline{S_0}\,\overline{S_1} = 11$. Multiplexers differ from decoders and encoders by virtue of select lines that cause inputs to be produced at the output. An application is to combine data received from the Internet on input lines d_0, d_1, d_2, and d_3 onto a single microprocessor memory line Z, if an Internet interrupt has occurred, that has a code represented by selector lines S_0S_1.

Demultiplexers

A demultiplexer causes an input x to be transferred to one of 2^n output lines, where n is the number of select inputs in Figure 1.14. Output equations for a demultiplexer with two select inputs and four outputs are shown in the truth table, Table 1.31. The demultiplexer does the reverse of the multiplexer; for example, it distributes Internet data, which have been multiplexed on input line x, to each of four microprocessor output ports d_0, d_1, d_3, and d_4.For example, Internet data will be distributed to output port d_0 when $\overline{S_0}\ \overline{S_1} = 11$ in Table 1.31.

SEQUENTIAL CIRCUITS

A *clocked synchronous sequential circuit* uses flip-flops to store data, and its outputs depend on both the *previous* and *current* values of inputs [HAR07]. These circuits are called state machines, wherein states are stored in flip-flops, and state changes are triggered by CPs. In an *asynchronous sequential circuit*, the completion of an operation starts the next operation (i.e., a clock is not needed).

Flip-Flops and Latches

A flip-flop is a *clocked synchronous sequential circuit* with a 1-bit memory. The output of the flip-flop can be changed by the rising or falling edge of a CP. A clock prevents the flip-flop from changing state when spurious inputs occur. Instability can arise if inputs change during the CP. This problem is avoided by holding data stable for specified periods of time before and after the CP. The former period is called *setup time* and the latter is called *hold time*.

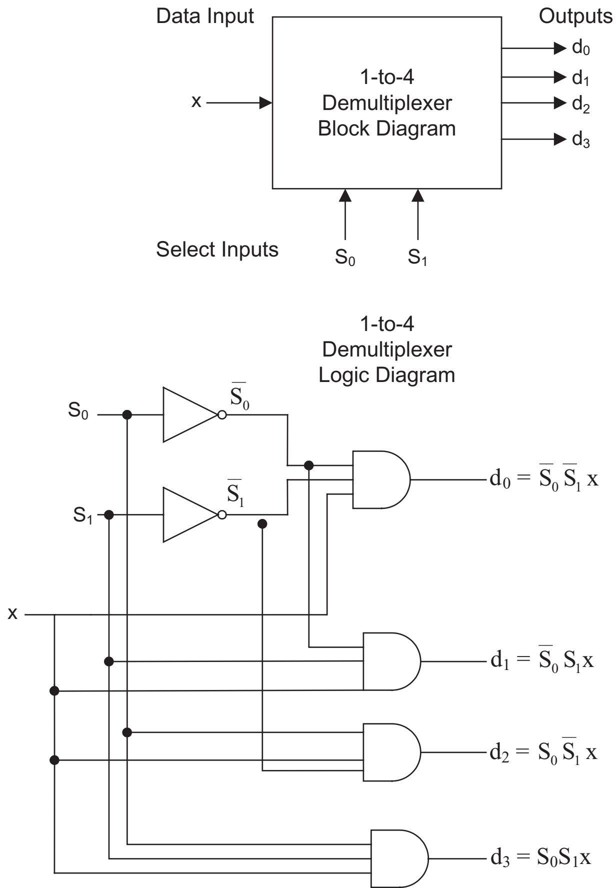


Figure 1.14 The 1-to-4 demultiplexer block and logic diagrams.

Table 1.31 Truth Table for 1-to-4 Demultiplexer

Select inputs		Data input	Data output			
S_0	S_1		d_0	d_1	d_2	d_3
0	0	x	$\overline{S_0}\,\overline{S_1}x$	0	0	0
0	1	x	0	$\overline{S_0}S_1x$	0	0
1	0	x	0	0	$S_0\overline{S_1}x$	0
1	1	x	0	0	0	S_0S_1x

$d_0 = \overline{S_0}\,\overline{S_1}x$; $d_1 = \overline{S_0}S_1x$; $d_2 = S_0\overline{S_1}x$; $d_3 = S_0S_1x$.

Table 1.32 SR Latch Truth Table Using NOR Gates

S	Q(t) (present state)	R	$\overline{Q}(t)$ (present state)	Q(t + 1) (Gate #1) (next state)	$\overline{Q}(t+1)$ (next state)
0	0	0	1	0 (no change)	1(no change)
1	0	1	1	0 (illegal)	0 (illegal)
0	1	1 (reset)	0	0 (change state)	1(change state)
1 (set)	0	0	1	1(change state)	0(change state)

Flip-flops use storage circuits called latches. The term "latch" refers to the ability to receive and hold data (set) until the latch is reset. The most common latch is the SR (set–reset). An application of a latch is to set and hold an interrupt flag when an input device needs attention by the microprocessor. A flip-flop is a latch with clock input (CLK). Flip-flops implement changes in circuit states that are triggered by a CP. For example, when the CP and the input line cause the flip-flop to assume the set state, a computer program would execute a branch operation; when the CP and the input line cause the flip-flop to assume the reset state, a computer program would return to the main line of the program. An interesting question is how a latch or flip-flop manages to be in the initial state. The answer is that the latch or flip-flop will be in the initial state determined by the initial state settings wired into the flip-flop.

SR Latch

The logic rules of the SR latch are the following:

NOR Gate output = 1, if *all* inputs = 0; output = 0, if *any* input = 1.

These rules are applied in the truth table shown in Table 1.32 and the logic diagram in Figure 1.15. Notice in Table 1.32 and Figure 1.15 that there are illegal next states in the case of S = 1 and R = 1 because it is not possible to simultaneously set and reset the latch.

Reset–Set (RS) Flip-Flop

The RS flip-flop is a clocked SR latch. This flip-flop is important because all other flip-flops are derived from it. Figure 1.16 shows the implementation of this flip-flop using NAND gates and the truth table, Table 1.33, shows the gate relationships for present state at time t and next state at time (t + 1), including simultaneous set and reset that should be avoided. In Figure 1.16, notice that there is feedback from Gate 3 to Gate 4 of Q(t + 1) and from Gate 4 to Gate 3 of $\overline{Q}(t+1)$.

The design in Figure 1.16 is obtained by employing the equations below, which in turn are obtained from Table 1.33 and the K-map in Table 1.34. The components of the equations are annotated on Figure 1.16. The K-map is constructed by noting whether the next state output Q(t + 1) is a 1. If it is, the corresponding present state

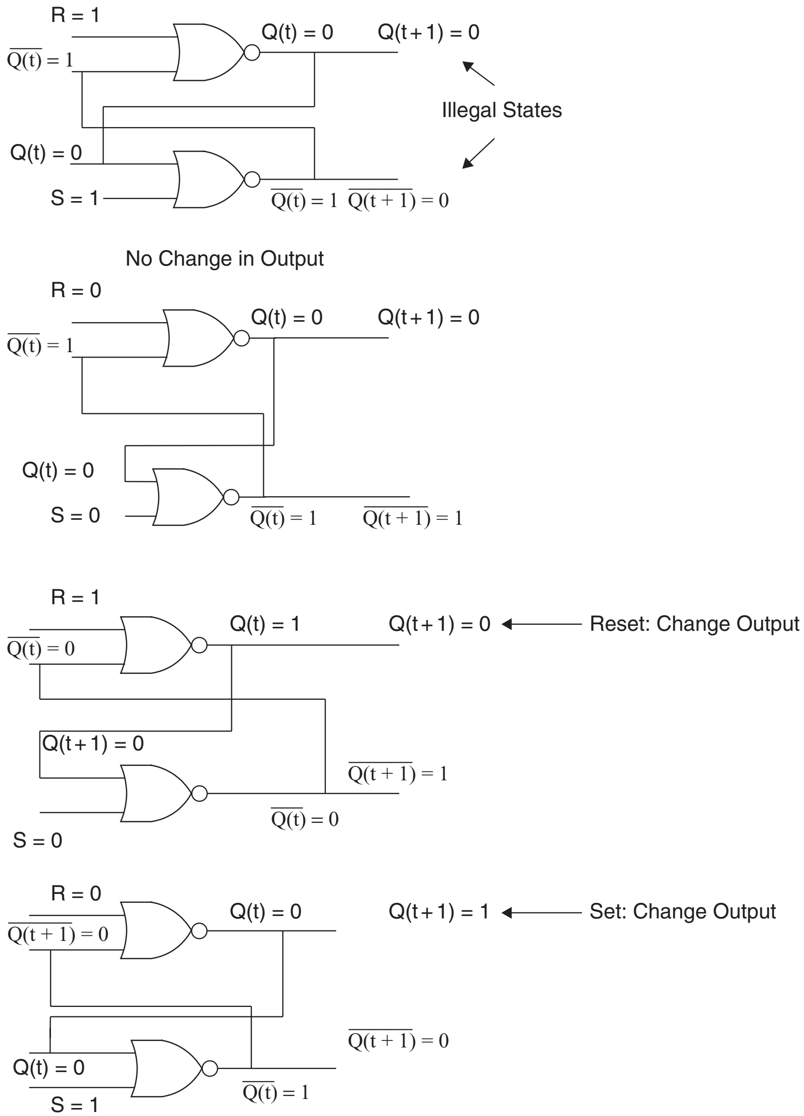


Figure 1.15 SR latch logic diagram.

output Q(t) is inserted into the K-map. The corresponding next state and present state outputs are bolded in Table 1.33. You can see that Table 1.33 contains eight entries, corresponding to whether the Present State Q(t) (Gate #3) is **0** or **1**; however, Figure 1.16 shows five cases, sufficient to demonstrate the logic of the RS flip-flop.

Based on Table 1.33, the K-map is constructed in Table 1.34. Then the K-map is used to formulate the equations for the flip-flop:

Table 1.33 RS Flip-Flop Truth Table

S (Gate #1)	R (Gate #2)	Present state Q(t) (Gate #3)	Next state Q(t + 1) (Gate #3)	Present state $\overline{Q}(t)$ (Gate #4)	Next state $\overline{Q}(t+1)$ (Gate #4)
0	0	0	0 (hold)	**1**	**1**
0	0	**1**	**1** (hold)	0	0
0	1 (reset)	0	0	**1**	**1**
0	1 (reset)	1	0	**0**	**1**
1 (set)	0	**0**	**1**	1	0
1 (set)	0	**1**	**1**	0	0
1(illegal)	1(illegal)	**0**	**1**	1	0
1(illegal)	1(illegal)	**1**	**1**	0	0

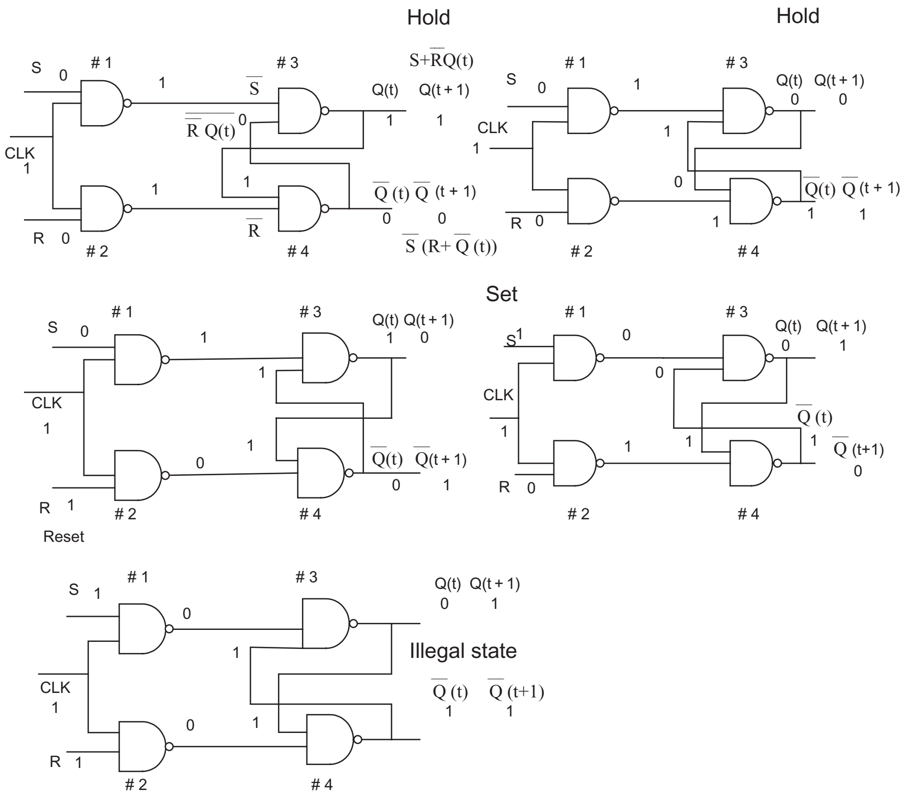


Figure 1.16 RS flip-flop.

Table 1.34a K-Map

S	R	Present State Q(t) (Gate #3)	
		0	1
0	0		1
0	1		
1	1	1	1
1	0	1	1

S $\quad \overline{R}Q(t)$

Table 1.34b K-Map

S	R	Present State $\overline{Q}(t)$ (Gate #4)	
		0	1
0	0		1
0	1	1	1
1	1		
1	0		

$\overline{S}R$ $\quad \overline{S}\,\overline{Q}(t)$

Table 1.34a: (Gate #3): $Q(t+1) = S + \overline{R}\ Q(t)$

Table 1.34b: (Gate #4): $\overline{Q(t+1)} = \overline{S}(R + \overline{Q}(t))$

Problem: What are the illegal states of the RS flip-flop?

Answer: The states S = 1 (set) and R = 1 (reset) are not allowed in an RS flip-flop because set and reset cannot exist simultaneously (indeterminate state).

Delay (D) Flip-Flop

The D or delay flip-flop, shown in Figure 1.17, uses NAND gates. It is widely used in computers for transferring data. Several of these flip-flops can be used to design a CPU register, where each flip-flop is used to store 1 bit [RAF05]. This flip-flop delays the input appearing at the output by one CP. The D input goes directly into the S input and the complement of the D input goes to the R input. The D input is sampled during the occurrence of the CP. If D is 1, the flip-flop is switched to the set state (unless it was already set). If D is 0, the flip-flop switches to the clear state. If CP = 1, the output Q(t + 1) of the upper flip-flop is fed to the input of the lower flip-flop in Figure 1.17. On the other hand, if CP = 0, Q(t) of the upper flip-flop is fed to the input of the lower flip-flop.

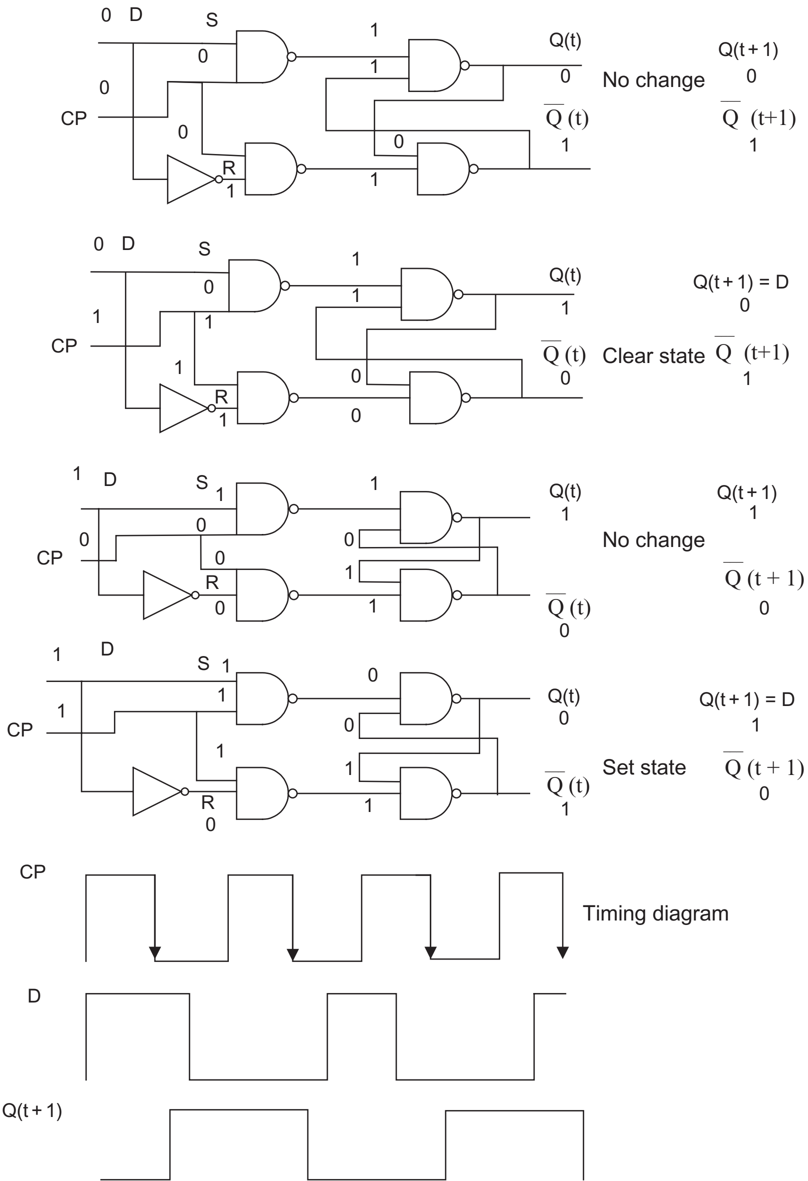


Figure 1.17 D flip-flop.

Problem: Given the above rules for the behavior of the D flip-flop, develop its truth table.

Solution: These relationships are embodied in Table 1.35.

A D flip-flop circuit can also be triggered by the negative-going edge of the CP, as opposed to being activated by pulse duration. The timing diagram for such a circuit

Table 1.35 D Flip-Flop Truth Table

D	CP	Present state Q(t)	Next state Q(t + 1) = D when CP = 1	Present state $\overline{Q}(t)$	Next state $\overline{Q}(t+1) = \overline{D}$ when CP = 1
0	0	0	0 (no change)	1	1 (no change)
0	1	1	0 (clear)	0	1
1	0	1	1 (no change)	0	0 (no change)
1	1	0	1 (set)	1	0

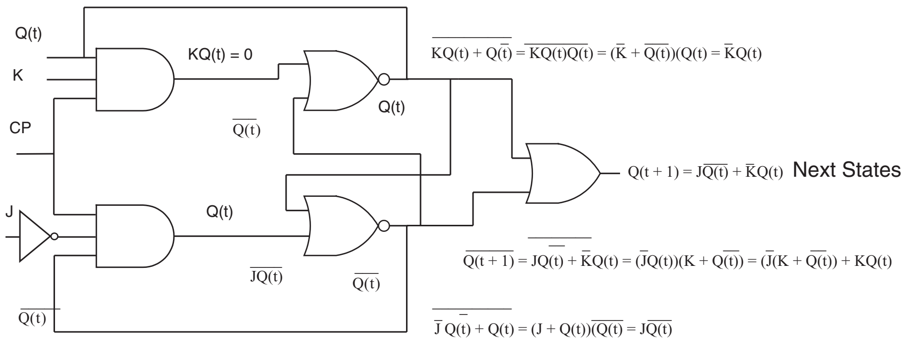


Figure 1.18 JK flip-flop circuit.

is shown in Figure 1.17. As the timing diagram shows, the D input is reflected in the Q(t + 1) (next state) output on the negative edge of the CP. Q(t + 1) follows the D input regardless of the present state Q(t), *if CP = 1*. If CP = 0, there is no change in the output. This property can be applied, for example, to transferring data from an input device (D) to microprocessor memory port Q(t + 1), according to the data transfer rules of Figure 1.17.

JK Flip-flop

A JK flip-flop is a refinement of the RS flip-flop by defining and allowing the illegal state of the RS flip-flop. In Figure 1.16, inputs J and K behave like inputs S and R to set and clear the flip-flop (note that in a JK flip-flop, the letter J is for set and the letter K is for clear). When logic 1 inputs are applied to both J and K simultaneously, the flip-flop switches to its complement state (e.g., if Q = 0, it switches to Q = 1 in Figure 1.18).

Note that because of the feedback connection in the JK flip-flop, a CP signal that remains a 1 (while J = K = 1) after the outputs have been complemented once will cause repeated and continuous transitions of the outputs. To avoid this, the CPs must have a time duration less than the propagation delay through the flip-flop.

Table 1.36 shows how the state of output Q at t + 1 changes as a function of the original state of Q(t) and the set input J and the clear input K. The K-map for

Table 1.36 JK Flip-Flop Truth Table

J	K	CP	Q(t) present state	Q(t + 1) next state	$\overline{Q}(t)$ present state	$\overline{Q}(t+1)$ next state
0	0	1	0	0	1	1
0	1 (clear)	1	0	0	1	1
1(set)	0	1	0	**1**	1	0
1	1	1	0	**1**	1	0
0	0	1	1	**1**	0	0
0	1(clear)	1	1	0	0	1
1(set)	0	1	1	**1**	0	0
1	1	1	1	0	0	1

Table 1.37 K-Map for JK Flip-Flop

J	K	Q(t) Present State	Q(t) Present State
		0	1
0	0		1
0	1		
1	1	1	
1	0	1	1

$J\overline{Q}(t)$ $\overline{K}\,Q(t)$

JK flip-flop in Table 1.37 is derived from the truth table in Table 1.36 by plugging **1**s in the map wherever there is a Q(t + 1) = 1 in the Table 1.36 (bolded). For example, when J = 0, K = 0, Q(t) = 1, and Q t + 1) = 1 in Table 1.36, a 1 is placed in the Q(t) = 1 column in Table 1.37.

Problem: Based on the K-map, what are the next state equations for Q(t + 1) and $\overline{Q}(t+1)$?

Answer: Referring to Table 1.37, the next state Q(t + 1) is governed by the following equation:

$$Q(t+1) = J\,\overline{Q}(t) + \overline{K}\,Q(t).$$

Using this equation for Q(t + 1), the equation for $\overline{Q}(t+1)$ can be computed as follows:

$$\overline{Q(t+1)} = \overline{J\,Q\overline{(t)} + K}\;Q(t) = (\overline{J} + Q(t))(K + Q\overline{(t)}) = (\overline{J}(K + \overline{Q(t)}) + K\,Q(t).$$

These equations are annotated on Figure 1.18.

Table 1.38 T Flip-Flop Truth Table

T	CP	Q(t)	$Q(t+1) = T\,\overline{Q(t)} + \overline{T}\,Q(t)$	Q(t)	$\overline{Q}(t+1) = T\,Q(t) + \overline{T}\,\overline{Q(t)}$
0	1	0	0 (no change)	1	1 (no change)
1	1	0	1 (toggle)	1	0 (toggle)
0	1	1	1 (no change)	0	0 (no change)
1	1	1	0 (toggle)	0	1(toggle)

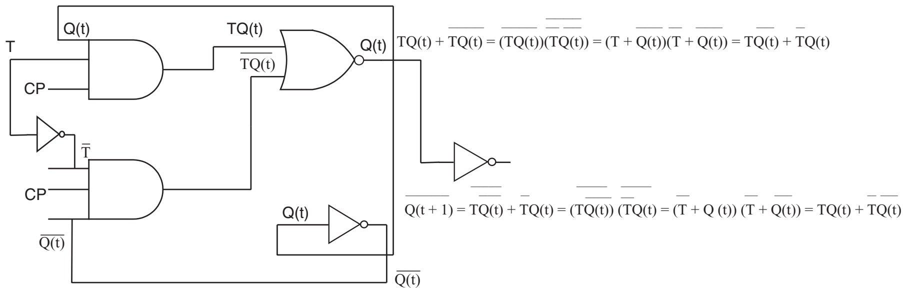


Figure 1.19 T flip-flop circuit diagram.

T Flip-Flop

The T flip-flop is a single input version of the JK flip-flop [RAF05]. It is typically used in the design of binary counters (covered later in the section "Design of Binary Counters," where complementation of the output is required. For example, in Table 1.38 when T = 1, the input Q(t) is toggled, producing its complement in output Q(t + 1). By examining the gate operations in Figure 1.19, at the Q output, we see that:

$$Q(t+1) = \overline{TQ(t) + \overline{T}\,\overline{Q(t)}} = (\overline{TQ(t)})(\overline{\overline{T}\,\overline{Q(t)}}) = (\overline{T} + \overline{Q(t)})(T + Q(t)) = T\,\overline{Q(t)} + \overline{T}\,Q(t).$$

Furthermore, the equation for $\overline{Q}(t+1)$ is derived as follows:

$$\begin{aligned}\overline{Q(t+1)} &= \overline{T\,\overline{Q(t)} + \overline{T}\,Q(t)} = (\overline{T\,\overline{Q(t)}})(\overline{\overline{T}\,Q(t)}) \\ &= (\overline{T} + Q(t))(T + \overline{Q(t)}) = T\,Q(t) + \overline{T}\,\overline{Q(t)}.\end{aligned}$$

Note that in Figure 1.19 feedback from the flip-flop outputs to the inputs is used to obtain the desired outputs at time t + 1.

Problem: Based on the above equations, develop the T flip-flop truth table.

Solution: The truth table is shown in Table 1.38.

Triggering of Flip-Flops

There are situations where it is useful to have the output change only at the rising or falling edge of the CP, rather than *during* the CP. This stabilizes the circuit because

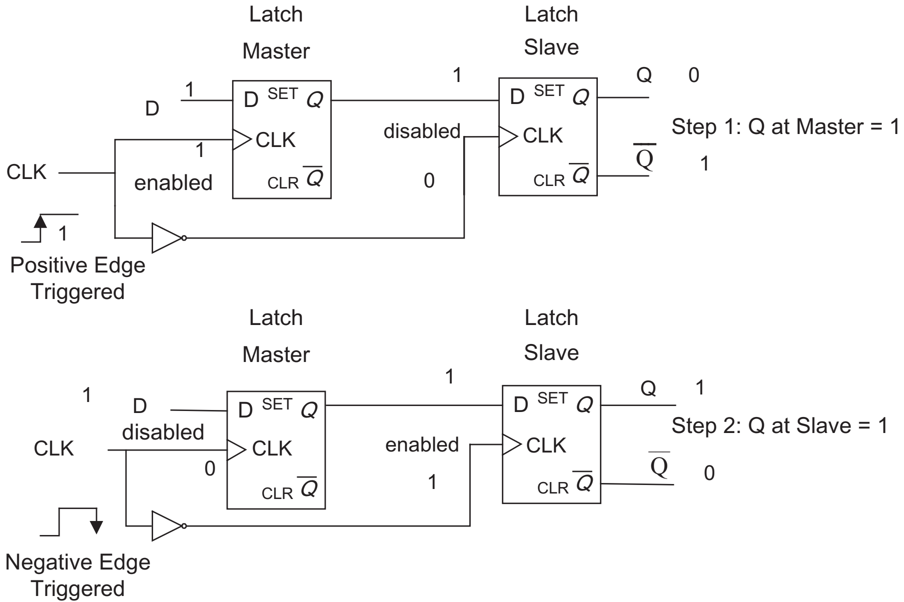


Figure 1.20 Edge-triggered flip-flop.

all changes are synchronized to the rising or falling edge of the CP. For example, when an input interrupt occurs, it should be held by the microprocessor until it can be serviced *during* the CP and only released on the *falling edge* of the CP. An edge-triggered flip-flop achieves this by combining a pair of latches in series. Figure 1.20 shows an edge-triggered D flip-flop where two D latches are connected in series, one directly, and one through an inverter. The first latch is called the master latch. When CLK is a 1 at Step 1, with a positive edge trigger, the master latch is enabled but the second latch, called the slave latch, is disabled with a negative edge trigger, so that a 1 is produced at the Q output of the master latch and a 0 is produced at the output of the slave latch. A 1 is produced at the master latch output because when CLK = 1, the Q output follows the D input. Contrariwise, when CLK is a 0 at Step 2, with a negative edge trigger, the master latch is disabled but the slave latch is enabled with a positive edge trigger (a negative edge is made positive with an inverter) so that a 1 is produced at the Q output of the slave latch by the Q output at the slave latch following the D input. In Step 2 it is assumed that Q still equals 1 in the master latch from Step 1. The Q output of the master latch does not change when CLK = 0, so that a 1 is transferred from the master latch to the slave latch.

Analysis of Asynchronous Sequential Circuits

As you have seen, edge-triggered flip-flops change state at the edge of a synchronizing CP. Many circuits require the initialization of flip-flops to a known state

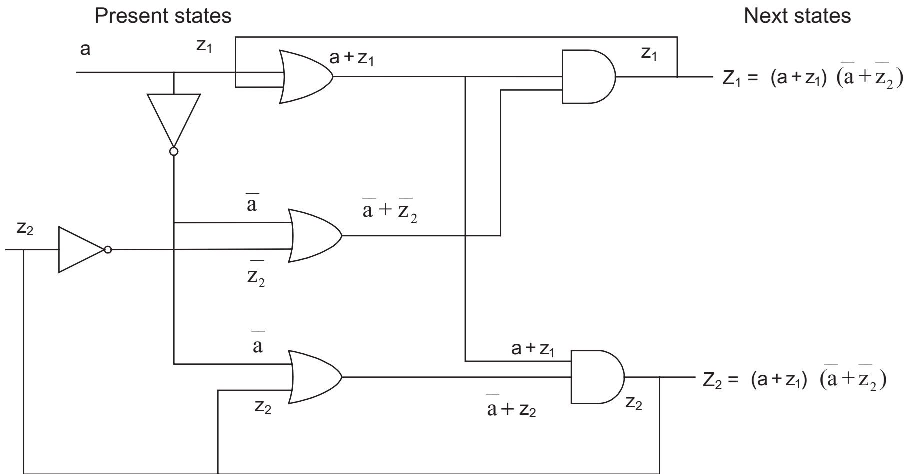


Figure 1.21 Analysis of asynchronous sequential circuit.

independent of the clock signal. Sequential circuits that change states whenever a change in input values occurs, independent of the clock, are referred to as *asynchronous sequential circuits*. Synchronous sequential circuits, latches, and flip-flops, on the other hand, change state only at the edge of the CP. For asynchronous sequential circuits, inputs are used to either set or clear the circuit *without* using the clock. Figure 1.21 is an example of an asynchronous sequential circuit. The next state equations for Z_1 and Z_2—as a function of present states a, z_1, and z_2—provide the logic for the outputs of the circuit in Figure 1.21. Feedback from outputs to inputs in Figure 1.21 produces the desired next states. The output equation

$$Z_1 = (a + z_1)(\bar{a} + \overline{z_2}) = a\,\bar{a} + \bar{a}\,z_1 + a\,\overline{z_2} + z_1\,\overline{z_2} = \bar{a}\,z_1 + a\,\overline{z_2} + z_1\,\overline{z_2}$$

can be reduced because the term $a\,\bar{a} = 0$, and the last term $z_1\,\overline{z_2}$ does not change the value of the equation, as demonstrated by the K-map in Table 1.40 that is used to minimize this equation, producing $Z_1 = \bar{a}\,z_1 + a\,\overline{z_2}$. Thus, the resultant terms $\bar{a}\,z_1$ and $a\,\overline{z_2}$ are identified in the K-map. The validity of this transformation is shown in the truth table for Z_1, Table 1.39. The K-map in Table 1.40 is produced by recording 1s in the map corresponding to **1**s (bolded) that appear for Z_1 in the truth table. This example demonstrates the fact that K-maps can accomplish Boolean expression reduction that is not possible with algebraic manipulation.

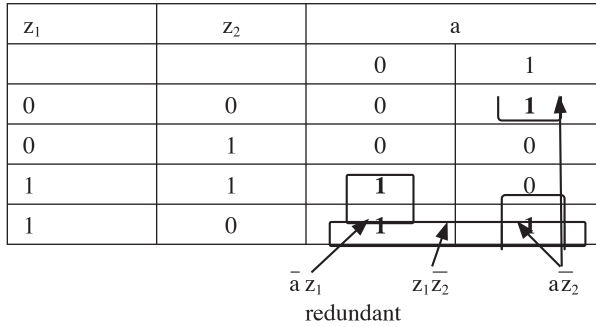
Problem: Reduce output equation Z_2 by developing the truth table and corresponding K-map.

Solution: The output equation $Z_2 = (a + z_1)(\bar{a} + z_2) = a\,\bar{a} + \bar{a}\,z_1 + a\,z_2 + z_1\,z_2 = \bar{a}\,z_1 + a\,z_2$ can be reduced, as shown above, because the first term $a\,\bar{a} = 0$ and the last term does not change the value of the equation, as demonstrated by the K-map in Table 1.41 that is used to minimize this equation, producing $Z_2 = \bar{a}\,z_1 + a\,z_2$,where it is shown that the term $z_1 z_2$ is redundant. Thus, the resultant terms $\bar{a}\,z_1$ and $a\,z_2$ are identified in the K-map. The validity of this

Table 1.39 Truth Table for $Z_1 = (a+z_1)(\bar{a}+\overline{z_2})\bar{a}\, z_1 + a\, \overline{z_2}$

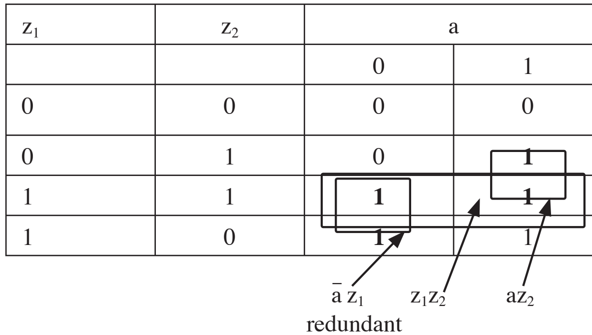
a	z_1	z_2	Decimal code = az_1z_2	$(a + z_1)$	$(\bar{a}+\overline{z_2})$	$Z_1 = (a+z_1)(\bar{a}+\overline{z_2})$	$\bar{a}\, z_1$	$a\, \overline{z_2}$	$Z_1 = \bar{a}\, z_1 + a\, \overline{z_2}$
0	0	0	0	0	1	0	0	0	0
0	0	1	1	0	1	0	0	0	0
0	1	0	**2**	1	1	**1**	1	0	**1**
0	1	1	**3**	1	1	**1**	1	0	**1**
1	0	0	**4**	1	1	**1**	0	1	**1**
1	0	1	5	1	0	0	0	0	0
1	1	0	**6**	1	1	**1**	0	1	**1**
1	1	1	7	1	0	0	0	0	0

Table 1.40 K-Map for $Z_1 = (a+z_1)(\bar{a}+\overline{z_2}) = \bar{a}\, z_1 + a\, \overline{z_2}$



z_1	z_2	a	
		0	1
0	0	0	**1**
0	1	0	0
1	1	**1**	0
1	0	**1**	**1**

Table 1.41 K-Map for $Z_2 = (a+z_1)(\bar{a}+z_2) = \bar{a}\, z_1 + a\, z_2$



z_1	z_2	a	
		0	1
0	0	0	0
0	1	0	**1**
1	1	**1**	**1**
1	0	**1**	1

transformation is shown in the truth table for Z_2 (Table 1.42). The K-map is produced by recording 1s in the map corresponding to **1**s (bolded) that appear for Z_2 in the truth table.

The state transition table, depicting the state changes in transitioning from input variables a, z_1, and z_2 to output variables Z_1 and Z_2, is shown in Table 1.43. This

Table 1.42 Truth Table for $Z_2 = (a + z_1)(\bar{a} + z_2)$

a	z_1	z_2	Decimal code = az_1z_2	$(a + z_1)$	$(\bar{a} + z_2)$	$Z_2 = Z_2 = (a + z_1)(\bar{a} + z_2)$	$\bar{a}\,z_1$	az_2	$Z_2 = \bar{a}\,z_1 + a\,z_2$
0	0	0	0	0	1	0	0	0	0
0	0	1	1	0	1	0	0	0	0
0	1	0	**2**	1	1	**1**	1	0	**1**
0	1	1	**3**	1	1	**1**	1	0	**1**
1	0	0	4	1	0	0	0	0	0
1	0	1	**5**	1	1	**1**	0	1	**1**
1	1	0	6	1	0	0	0	0	0
1	1	1	7	1	1	**1**	0	1	**1**

Table 1.43 State Transition Table for Asynchronous Sequential Circuit

Present state		Next state			
		a = 0		a = 1	
z_1	z_2	$Z_1 = \bar{a}\,z_1 + a\,\overline{z_2}$	$Z_2 = \bar{a}\,z_1 + a\,z_2$	$Z_1 = \bar{a}\,z_1 + a\,\overline{z_2}$	$Z_2 = \bar{a}\,z_1 + a\,z_2$
0	0	0	0	1	0
0	1	0	0	0	1
1	0	1	1	1	0
1	1	1	1	0	1

table is constructed by noting the values of Z_1 corresponding to a = 0 and a = 1 and values of Z_2 corresponding to a = 0 and a = 1 in Tables 1.39 and 1.42, respectively, and recording the relationships in Table 1.43. Table 1.43 is used to indicate transitions from microprocessor state $Z_1 = 1$ to state $Z_2 = 1$ and vice versa. Consider the following application: when a = 1, $z_1 = 0$, and $z_2 = 0$ (decimal code 4), Z_1 is in the next state = 1 processing transactions. However, when a = 1, $z_1 = 0$, and $z_2 = 1$ (decimal code 5), the microprocessor transitions to the next state $Z_2 = 1$ to receive additional transaction input.

Another application of the asynchronous sequential circuit is the occurrence of asynchronous inputs to a microprocessor that arrive from the Internet, not on schedule (not governed by CP), but unscheduled (i.e., asynchronously). For example, let a, z_1, and z_2 be the binary bits of a decimal transaction code, arriving from the Internet, in a database application, where one type of transaction is processed by a microprocessor at its input Z_1 and the second type at its input Z_2. Suppose the allowable decimal codes at Z_1 are **2**, **3**, **4**, and **6** in Table 1.39 (bolded), and the allowable codes at Z_2 are **2**, **3**, **5**, and **7** in Table 1.42 (bolded). Then, Tables 1.39 and 1.42 provide the required transaction processing logic for Z_1 and Z_2, respectively.

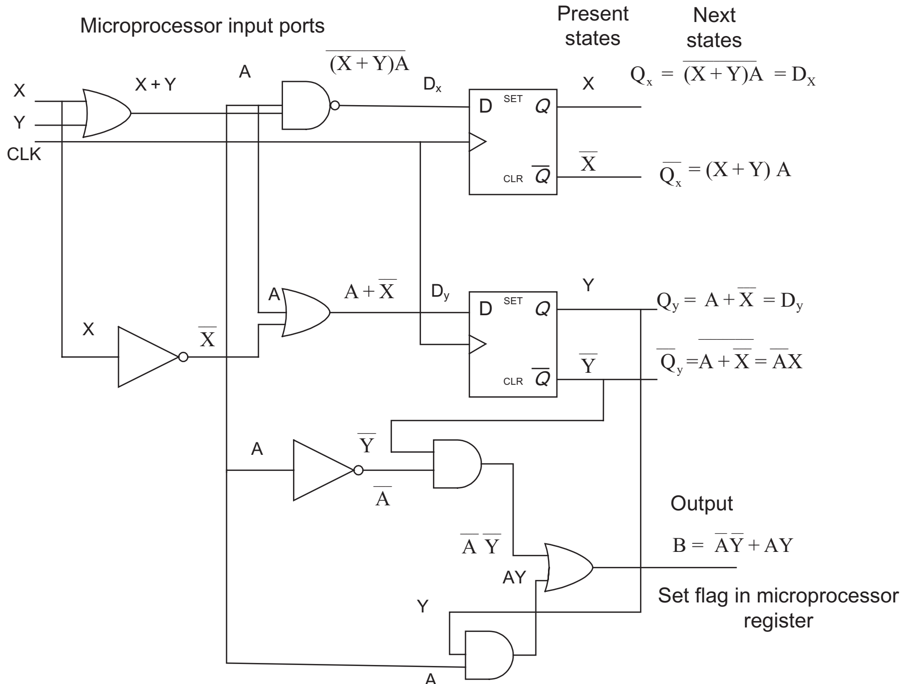


Figure 1.22 D flip-flops in asynchronous sequential circuit.

Relationship among Inputs, Flip-Flops, and Output States

Figure 1.22 shows an example of analyzing the inputs, D flip-flops, and output states of an asynchronous sequential circuit. The diagram shows the equations for the next states Q_x and Q_y, as a function of the present states D_x and D_y, recalling that for D flip-flops, output Q follows input D.

The equations below produce the values shown in the state transition table, Table 1.44, which shows the relationships among components.

$$Q_x = \overline{(X+Y)A} = D_x,$$

$$\overline{Q_x} = (X+Y)A,$$

$$Q_y = A+\overline{X} = D_y,$$

$$Q_y = \overline{A+\overline{X}} = \overline{A}X,$$

$$B = \overline{A}\,\overline{Y} + AY.$$

An application is the processing of transaction code bits occurring at microprocessor input ports X, Y, and A. An output B = 1 is produced by setting a flag B in a microprocessor register when correct transaction codes are received. For example, if decimal interrupt code 1, 3, 4, 6, or 7, corresponding to X, Y, A = 001, 011, 100,

Table 1.44 State Transition Table for the Analysis of Asynchronous Sequential Circuit

Inputs						Next state				Flip-flop inputs		Output
X	$\bar{X}$	Y	$\bar{Y}$	A	$\bar{A}$	Qx = Dx	$\bar{Q}_x$	Qy = Dy	$\bar{Q}_y$	$D_x = Q_x = \overline{(X + Y)A}$	$D_y = Q_y = A + \bar{X}$	$B = \bar{A}\bar{Y} + AY$
0	1	0	1	0	1	1	0	1	0	1	1	0
0	1	**0**	1	**1**	0	1	0	1	0	1	1	**1**
0	1	1	0	0	1	1	0	1	0	1	1	0
0	1	**1**	0	**1**	0	0	1	1	0	0	1	**1**
1	0	0	1	**0**	1	1	0	0	1	1	0	**1**
1	0	0	1	1	0	0	1	1	0	0	1	0
1	0	**1**	0	**0**	1	1	0	0	1	1	0	**1**
1	0	**1**	0	**1**	0	0	1	1	0	0	1	**1**

110, or 111 in Table 1.44, respectively, is received, the flag would be set. The microprocessor queries this flag to determine when to process transactions. The bolded terms in Table 1.44 indicate when the flag B is set.

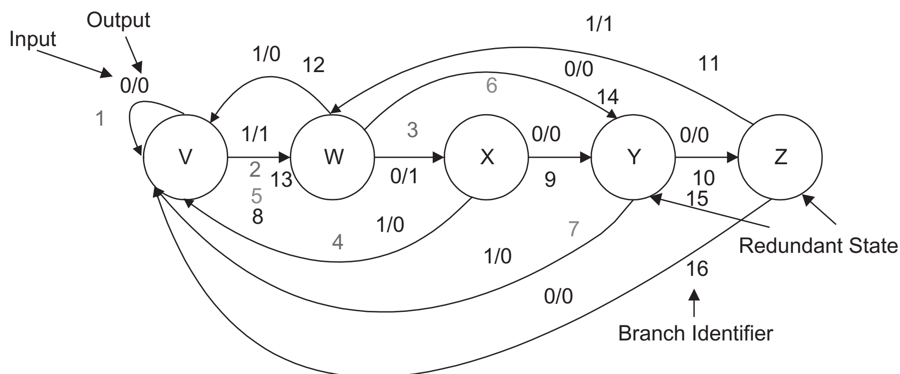
TYPES OF SYNCHRONOUS SEQUENTIAL CIRCUITS

Mealy and Moore Machines

In the Mealy machine, the output states depend on the inputs and the present states of the flip-flops [RAF05]. In the Moore machine, output states depend only on the present states of the flip-flops. For example, a Mealy machine would be used to control the execution sequence of a microprocessor that uses *both* data inputs and the current state of the program (i.e., program address) to decide which instruction to execute next (e.g., doing database management using input data from the Internet). On the other hand, the Moore machine would be used to control microprocessor program execution when *only* the current state of the program is relevant (e.g., doing a matrix multiplication on data stored in memory). Thus, the Mealy machine is the more versatile of the two.

Minimization of States

Figure 1.23 shows a state diagram for a synchronous sequential circuit, which is classified as a Mealy machine because outputs depend on *both* present states and inputs, where two of the paths are highlighted in red and green. It may be possible to minimize the number of states in these circuits by developing the state sequence diagram, based on Figure 1.23, to see whether there are any redundant states. If there are, the reduction in states is reflected in the revised state sequence table. Using Figure 1.23 and the original state sequence table, Table 1.45, state Z is identified as being redundant because the next state for both states V and Z is W, and the state changes have the same inputs and outputs (1, 1). Therefore, state Z is noted as



States: V,W,X,Y,Z
Path Sequences: (1,2,3,4) (5,6,7) (8,3,9,10,11,12) (13,14,15,16)
Input Sequence: (0 1 0 1) (1 0 1) (1 0 0 0 1 1) (1 0 0 0)
Output Sequence:(0 1 1 0) (1 0 0) (1 1 0 0 1 0) (1 0 0 0)

Figure 1.23 State diagram for minimization of states.

Table 1.45 Original State Sequence Table

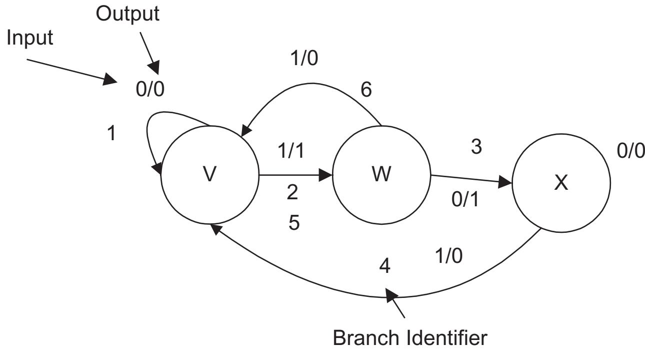
Originating state	V	V	W	X	**Y**	W	V	W	**Z**	States Y and Z are redundant
Branch	1	2	3	4	**6**	12	10	5	**12**	
Input	0	1	0	1	**1**	1	1	0	**0**	
Next state	V	W	X	V	**V**	V	W	Y	**V**	
Output	0	1	1	0	**0**	0	1	0	**0**	

Table 1.46 Revised State Sequence Table (Eliminating Redundant States)

	Next state		Output	
Present state	Input = 0	Input = 1	Input = 0	Input = 1
V	V	W	0	1
W	X	V	1	0
X		V		0

redundant in Figure 1.23 and Table 1.45. Another state indicated as redundant in Figure 1.23 and Table 1.45 is Y because both Y and W have the next state V, with same state change inputs and outputs (1, 0). State Y is also noted as redundant in Figure 1.23 and Table 1.45. Therefore, states Z and Y do not appear in the revised state sequence table, Table 1.46.

Figure 1.24 shows the result of eliminating redundant states in the state diagram. It is important to note that it may not be possible to eliminate “redundant states”



States: V, W, X

Path Sequences: (1, 2, 3, 4) (5, 6)

Input Sequence: (0 1 0 1) (1 1)

Output Sequence: (0 1 1 0) (1 0)

Figure 1.24 Reduced state diagram.

because these states could be associated with important functions. For example, redundant states could be associated with two microprocessors—one the primary, currently executing, and the other, a backup, redundant microprocessor, designed to take over if the primary fails. However, in general, digital circuitry can be simplified by eliminating redundant states.

Design of Synchronous Sequential Circuits

To design synchronous sequential circuits, or any circuit for that matter, start with your objective. For example, suppose you want a microprocessor to produce an output Z dependent on input A (e.g., input data A has arrived from the Internet, and the microprocessor produces output Z); the present state of your computer program is represented by X (e.g., ready to read input data A) and the present state of the input buffer A is represented by Y (e.g., input buffer A empty). You need to identify the transition to the next computer program state, X+, (e.g., fill buffer with input A data) and Y+ (e.g., input A buffer full). Thus, referring to the state diagram in Figure 1.25, if an input occurs on microprocessor line $A = 1$, and the present program state are $X = 1$, $Y = 1$, representing instruction ready to execute and input buffer A empty, respectively, output is produced on microprocessor line $Z = 1$, and the program transitions to next state $X+ = 0$ (fill buffer) and $Y+ = 0$, (input buffer A full). The state diagram in Figure 1.25 is an example of a Mealy machine circuit specification because outputs depend on both inputs and states of the circuit.

To design your circuit, identify the states, inputs that cause state transitions, and outputs produced by inputs and state transitions, as in the above example. Then, note

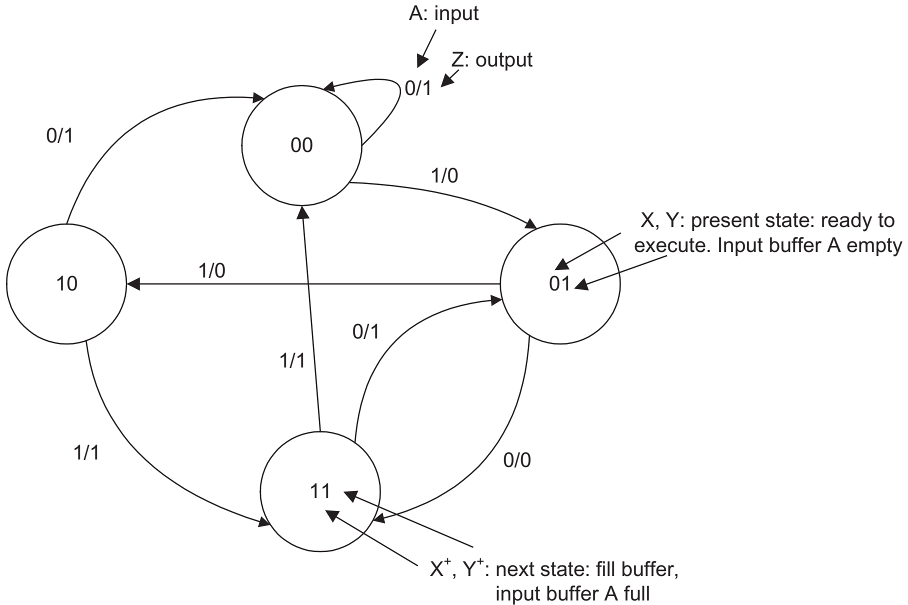


Figure 1.25 State diagram for design of sequential circuit.

Table 1.47 State Table for Sequential Circuit

Present states		Input	Next states		Flip-flop inputs		Output
X	Y	A	X^+	Y^+	$D_x = X^+ = \overline{X}Y + X\overline{Y}A$	$D_y = Y^+ = \overline{Y}A + Y\overline{A}$	$Z = \overline{Y}\overline{A} + X$
0	0	0	0	0	0	0	**1**
0	0	1	0	1	0	**1**	0
0	1	0	1	1	**1**	**1**	0
0	1	1	1	0	**1**	0	0
1	0	0	0	0	0	0	**1**
1	0	1	1	1	**1**	**1**	**1**
1	1	0	0	1	0	**1**	**1**
1	1	1	0	0	0	0	**1**

in Figure 1.25 and in Table 1.47 the present states X and Y and input A that generate next state output $X^+ = 1$. For example X = 0, Y = 1, and A = 1 (or $\overline{X}YA$) produce $X^+ = 1$. Next, for example, use the D flip-flop, noting that the output corresponding to next state X^+ is designated as D_x and its formulation is the following:

$$D_x = \overline{X}YA + \overline{X}Y\overline{A} + X\overline{Y}A = \overline{X}Y(A + \overline{A}) = \overline{X}Y + X\overline{Y}A.$$

Similarly, produce the next state Y^+ formulation in terms of a D flip-flop output, as follows:

$$D_y = \overline{X}\,\overline{Y}A + X\overline{Y}A + \overline{X}Y\overline{A} + XY\overline{A} = \overline{Y}A(\overline{X} + X) + Y\overline{A}(\overline{X} + X) = (\overline{Y}A + Y\overline{A}).$$

Also, develop the equation for output Z by noting in Figure 1.25 the present states X and Y and input A that generate Z = 1 output, producing the following equation:

$$Z = \overline{X}\,\overline{Y}\,\overline{A} + X\overline{Y}\,\overline{A} + XYA + XY\overline{A} = \overline{Y}\,\overline{A}(\overline{X} + X) + XY(A + \overline{A}) = (\overline{Y}\,\overline{A} + XY).$$

Then, using these equations, develop the state table in Table 1.47. Next, formulate the K-maps in Tables 1.48–1.50. Note that to construct the K-maps, 1s are placed in the cells of the maps wherever 1s appear for D_x, D_y, and Z in the state table. Recall that for D flip-flops, inputs are equal to the next states of the circuit. Last, based on the flip-flop and output equations, design the circuit in Figure 1.26.

Message Processing Design

Synchronous sequential circuits are highly adaptable to message processing systems, as shown in Figure 1.27. As shown in the figure, a message processing system

Table 1.48 K-Map for $D_x = \overline{X}YA + \overline{X}Y\overline{A} + X\overline{Y}A = \overline{X}Y(A + \overline{A}) = \overline{X}Y + X\overline{Y}A$

	YA			
X	00	01	11	10
0			1	1
1		1		

$X\overline{Y}A$ $\overline{X}Y$

Table 1.49 K-Map for $D_y = \overline{X}\,\overline{Y}A + X\overline{Y}A + \overline{X}Y\overline{A} + XY\overline{A} = \overline{Y}A(\overline{X} + X) + Y\overline{A}(\overline{X} + X) = (\overline{Y}A + Y\overline{A})$

	YA			
X	00	01	11	10
0		1		1
1		1		1

$\overline{Y}A$ $Y\overline{A}$

Table 1.50 K-Map for $Z = \overline{X}\,\overline{Y}\,\overline{A} + X\overline{Y}\,\overline{A} + XYA + XY\overline{A} = \overline{Y}\,\overline{A}(\overline{X} + X) + XY(A + \overline{A}) = (\overline{Y}\,\overline{A} + XY)$

	YA			
X	00	01	11	10
0	1			
1	1		1	1

$\overline{Y}\,\overline{A}$ XY

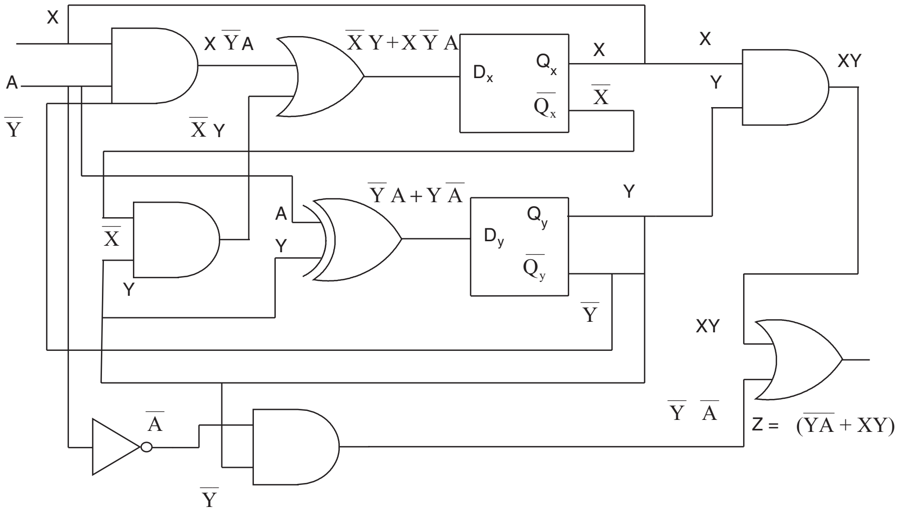


Figure 1.26 Logic diagram for synchronous digital circuit.

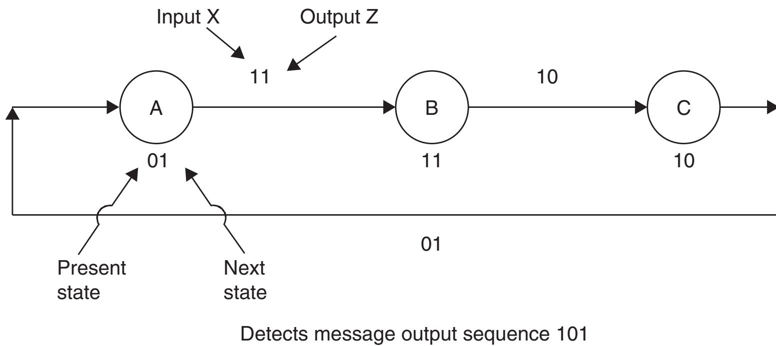


Figure 1.27 Message processing state diagram.

involves a sequence of inputs X with the objective of the circuit detecting a bit pattern, such as 101. The circuit accomplishes this objective by changing state according to the bit pattern received. When the desired bit pattern is recognized, the sequence 101 is generated at the output. An application is the detection of computer program operation codes by a microprocessor. For example, if the operation code for the add instruction is the decimal 5 (binary 101), the output 101 would be generated in Figure 1.27 designating that the add instruction should be executed.

The first step in the design process is to specify the state transitions, as shown in Figure 1.27, where the desired detected bit pattern is shown. State transitions are identified that will serial process the incoming bit stream, looking for the desired pattern in Figure 1.27. Additional steps involve designing the state transition table in Table 1.51 to represent the logic of Figure 1.27 in a tabular form and selecting a flip-flop type to implement state transitions. In this case, the T flip-flop is selected because its output toggles with each CP. If T = 1, the flip-flop causes complementation of the present state. This is the logic required to detect the input sequence 101 in Figure 1.28.

Table 1.51 State Transition Table

		Input X = 1	Input X = 1	Input X = 1
Present state	Present T flip-flop binary state Q	Next T flip-flop state Q	Next T flip-flop binary state Q	Output Z
A	0	B	1	1
B	0	C	1	0
C	0	A	1	1

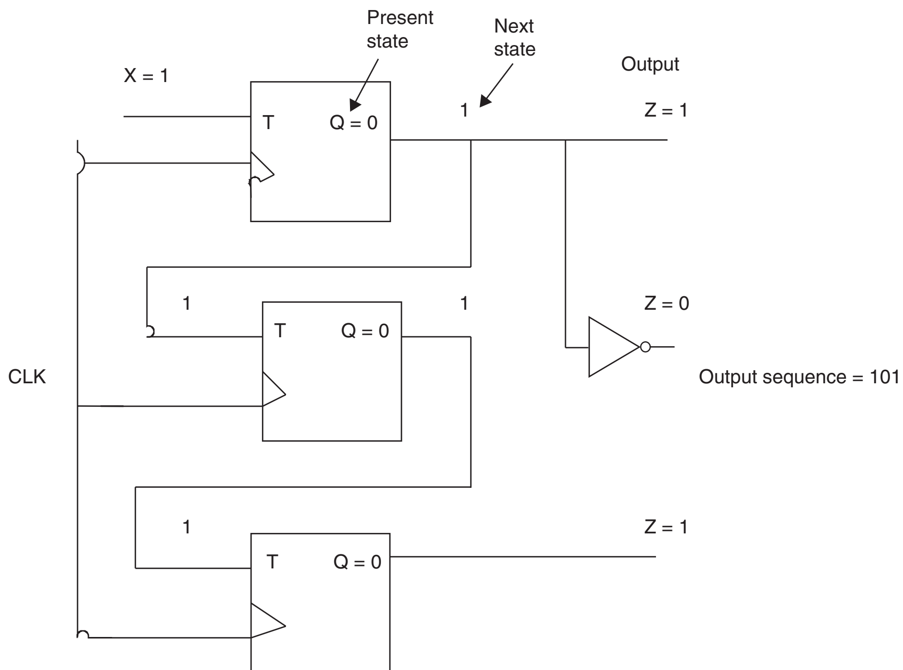


Figure 1.28 Message processing circuit.

Design of Binary Counters

Two-Bit Counter

The binary counter is an example of a synchronous sequential circuit designed to count a sequence of binary digits. For example, if the counter can count two binary digits at a time, it would be able to process the following sequence of digits: 00, 01, 10, and 11. Thus, the counter can count 2^n binary numbers, using flip-flops (e.g., T flip-flops), where n is the number of binary bits in the count. Figure 1.29 shows the state transition diagram for a 2-bit binary counter that implements the binary sequence count rules (e.g., if the sequence is 00, it is recognized by the next state 01). After Figure 1.29 has been constructed, the state table (Table 1.52) for flip-flops 1 and 2 is developed followed by the state table (Table 1.53) for flip-flops 3 and 4. The outputs b_0 and b_1 follow the logic rule: $T\overline{Q}(t) + \overline{T}Q(t)$ in Figure 1.29. Note

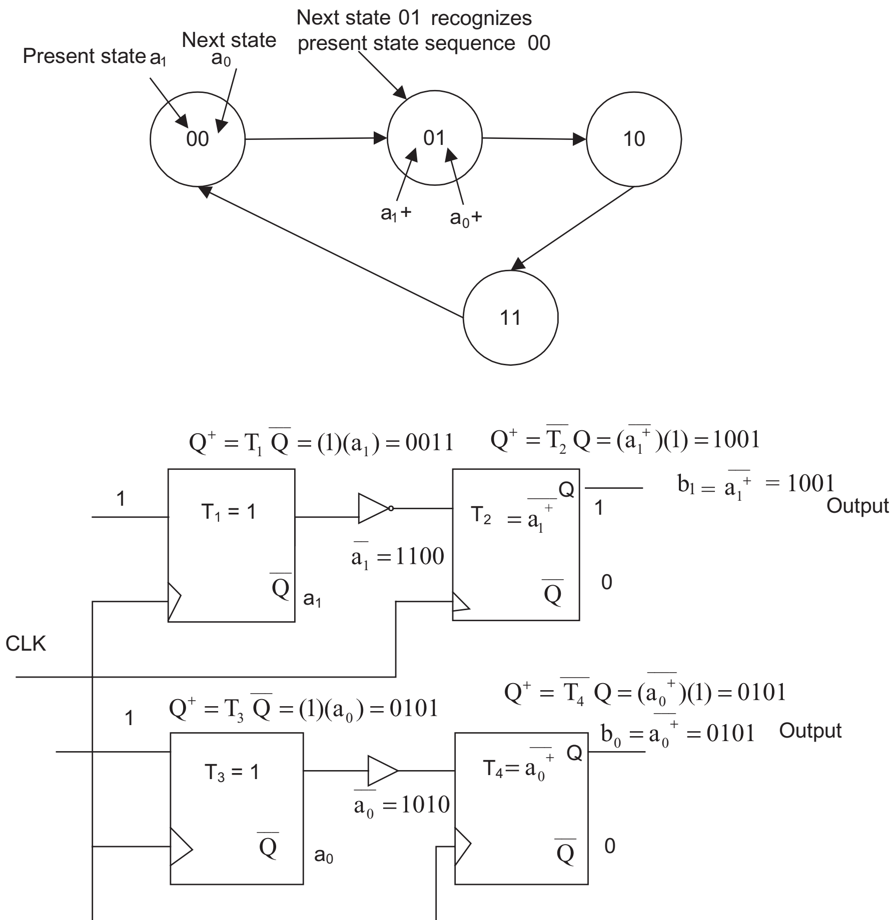


Figure 1.29 Binary counter state transition diagram and circuit.

Table 1.52 Binary Sequence Counter State Table

Present state		Next state flip-flop 1	Next state flip-flop 2	Output
a_1	a_1^+	$Q^+ = T_1\overline{Q} = (1)(a_1)$	$Q^+ = \overline{T_2}Q = (\overline{a_1^+})(1)$	$b_1 = \overline{a_1^+}$
0	0	0	1	1
0	1	0	0	0
1	1	1	0	0
1	0	1	1	1

Table 1.53 Binary Sequence Counter State Table

Present state		Next state flip-flop 3	Next state flip-flop 4	Output
a_0	a_0^+	$Q^+ = T_3\overline{Q} = (1)(a_0)$	$Q^+ = \overline{T_4}Q = (\overline{a_0^+})(1)$	$b_0 = \overline{a_0^+}$
0	0	0	0	0
0	1	0	1	1
1	1	1	0	0
1	0	1	1	1

that an inverter is inserted between the flip-flops in Figure 1.29 in order to achieve the correct state transitions.

Three-Bit Counter

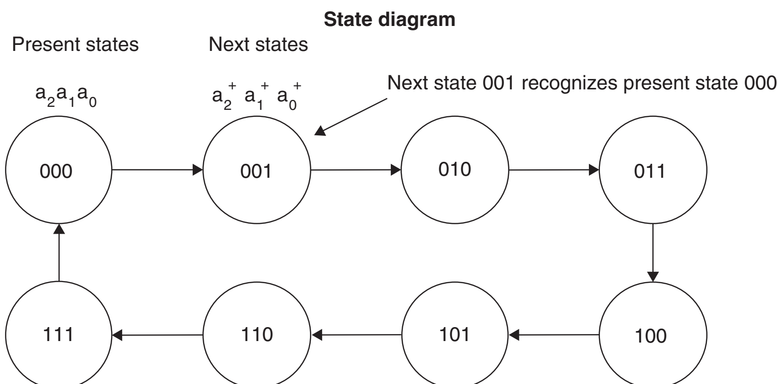
A 3-bit counter design proceeds by first constructing the state diagram in Figure 1.30, with present and next states annotated. Next, using JK flip-flops, show the 3-bit counter excitation table (Table 1.54), noting flip-flop states and flip-flop inputs. The salient state conditions can be summarized as follows: when $Q = 0$ and $J = 0$, no change in state; when $J = 1$, set the flip-flop; when $K = 1$, clear the flip-flop; and when $Q = 1$ and $K = 0$, no state change. The reader may wonder how the present states are obtained in Figure 1.30. The answer is that present states correspond to the present states of the flip-flops that, in turn, correspond to the condition where there is no CP (e.g., $a_2a_1a_0 = 000$).

To demonstrate the validity of the JK flip-flop transformations in Figure 1.30, recall the fundamental property of the JK flip-flop: Q^+ (next state) $= J\,\overline{Q(t)} + \overline{K}\,Q(t)$. For example, in the state transition $a_2a_1a_0 = 000\ a_2^+a_1^+a_0^+ = 001$, applying Q^+(next state) yields:

$$a_2^+ = J_2\overline{Q_2(t)} + \overline{K_2}Q_2(t) = a_1a_0\overline{a_2} + \overline{a_1a_0}a_2.$$

Thus,

$$a_2^+ = a_1a_0\overline{a_2} + \overline{a_1a_0}a_2 = 001 + \overline{00}0 = 0,$$
$$a_1^+ = J_1\overline{Q_1(t)} + \overline{K_1}Q_1(t) = a_0\overline{a_1} + \overline{a_0}a_1.$$



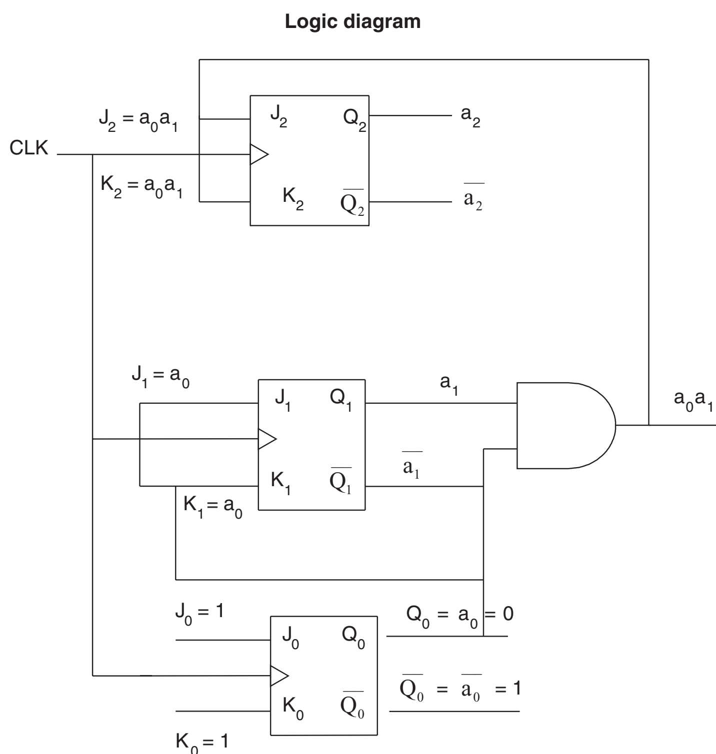


Figure 1.30 Three-bit counter state diagram and logic diagram.

Thus,

$$a_1^+ = a_0\overline{a_1} + \overline{a_0}a_1 = 01 + 10 = 0,$$
$$a_0^+ = J_0\overline{Q_0(t)} + \overline{K_0}Q_0(t) = 1\overline{a_0} + 0a_0 = 11 + 00 = 1.$$

Thus, the state transition $a_2a_1a_0 = 000$ $a_2^+a_1^+a_0^+ = 001$ is demonstrated.

Next, using Figure 1.30, formulate the truth table (Table 1.54), incorporating the state transitions from Figure 1.30 and the flip-flop inputs that generate these transitions. Next, the K-maps in Tables 1.55–1.60, by noting the flip-flop inputs that are bolded in Table 1.54, and resultant equations, are developed for the flip-flop inputs.

Table 1.54 Three-Bit Counter Truth Table

Present State			Next state			Flip-flop inputs					
a_2	a_1	a_0	a_2^+	a_1^+	a_0^+	$J_2 = a_1a_0$	$K_2 = a1_1a_0$	$J_1 = a_0$	$K_1 = a_0$	$J_0 = 1$	$K_0 = 1$
0	0	0	0	0	1	0	0	0	0	1	1
0	**0**	**1**	0	1	0	0	0	**1**	**1**	1	1
0	1	0	0	1	1	0	0	0	0	1	1
0	**1**	**1**	1	0	0	**1**	**1**	**1**	**1**	1	1
1	0	0	1	0	1	0	0	0	0	1	1
1	**0**	**1**	1	1	0	0	0	**1**	**1**	1	1
1	1	0	1	1	1	0	0	0	0	1	1
1	**1**	**1**	0	0	0	**1**	**1**	**1**	**1**	1	1

Table 1.55 K-Map for J_2

		a_1a_0			
		00	01	11	10
a_2	0			1	
	1			1	

$J_2 = a_1a_0$

Table 1.56 K-Map for K_2

		a_1a_0			
		00	01	11	10
a_2	0			1	
	1			1	

$K_2 = a_1a_0$

Table 1.57 K-Map for J_1

		a_1a_0			
		00	01	11	10
a_2	0		1	1	
	1		1	1	

$J_1 = a_0$

Table 1.58 K-Map for K_1

		a_1a_0			
		00	01	11	10
a_2	0	X	1	1	
	1	X	1	1	

$K_1 = a_0$

Table 1.59 K-Map for J_0

		a_1a_0			
		00	01	11	10
a_2	0	1	1	1	1
	1	1	1	1	1

$J_0 = 1$

Table 1.60 K-Map for K_0

		a_1a_0			
		00	01	11	10
a_2	0	1	1	1	1
	1	1	1	1	1

$K_0 = 1$

Shift Register Design

The design process starts by documenting the elements of the basic building block of the shift register—called the basic cell in Figure 1.31—comprised of the multiplexer and the D flip-flop. The D flip-flop is used because the flip-flop Q output follows the multiplexer basic cell D input, thus enabling the shift operation. The

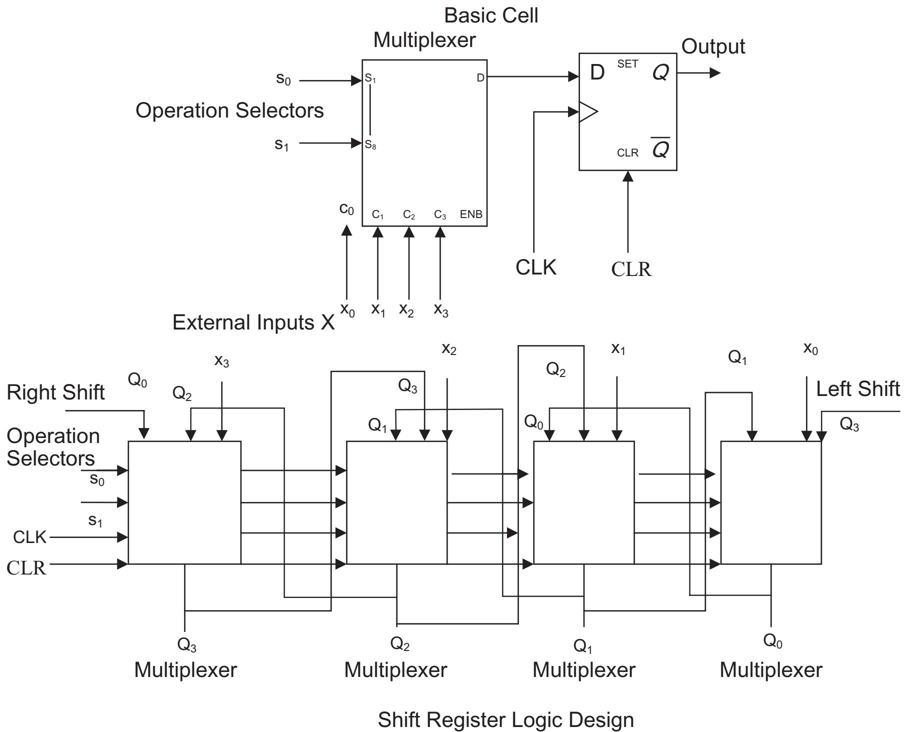


Figure 1.31 Basic cell and logic design of shift register.

basic cell is replicated in the shift register logic design, also shown in Figure 1.31. The shift register operates in Figure 1.31 by shifting the least significant bit, x_0, for a left shift, one flip-flop output to the left on each CP. For a right shift, the most significant bit, x_3, is shifted one flip-flop output to the right on each CP. These shifts are referred to as "end around" because for a right shift, the least significant bit, represented by Q_3 in Table 1.61, is shifted to the most significant bit position. Moreover, in a left shift, the most significant bit, represented by Q_0 in Table 1.61, is shifted to the least significant bit position. The type of shift is based on the values of the operation selectors in Table 1.61.

RAM DESIGN

There are two types of RAM: static and dynamic. Static RAM stores data in flip-flops. Dynamic RAM stores data in capacitors. Because capacitors gradually lose their charge, dynamic RAM must be refreshed periodically. A RAM circuit is shown in Figure 1.32 where 1 bit, 0 or 1, can either be read or written depending on whether a read or write operation is selected and whether a 1 or 0 appears at the input.

Table 1.61 Truth Table for Shift Register

Operation selectors		Clock input CLK	Clear input CLR	Operation	Input	Output
s_0	s_1					
0	0		1	Clear	$Q_0\ Q_1\ Q_2\ Q_3$	0000
0	1		0	No operation	$Q_0\ Q_1\ Q_2\ Q_3$	$Q_0\ Q_1\ Q_2\ Q_3$
1	0	↑	0	Shift right "end around"	$Q_0\ Q_1\ Q_2\ Q_3$	$Q_3\ Q_0\ Q_1\ Q_2$
1	1	↑	0	Shift left "end around"	$Q_0\ Q_1\ Q_2\ Q_3$	$Q_1\ Q_2\ Q_3\ Q_0$

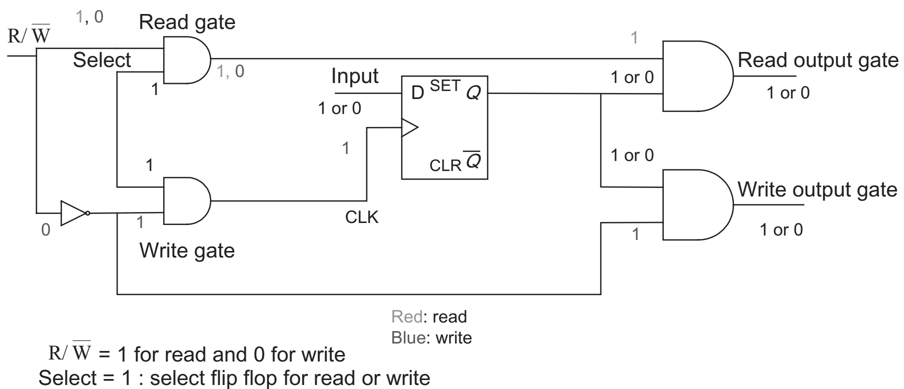


Figure 1.32 Random access memory (RAM) circuit.

HARDWARE DESCRIPTION LANGUAGE (HDL)

Given the complexity of some digital circuits, implementing them can be error prone. Therefore, as a design aid, aimed to increase design productivity and reduce errors, HDLs have been developed. In electronics, an HDL is any language from a class of computer languages for formal description of electronic circuits, and more specifically, digital logic. It can describe the circuit's operation, its design and organization, and tests to verify its operation by means of simulation.

Using the proper subset of virtually any HDL, a software program called a synthesizer can infer hardware logic operations from the language statements and produce equivalent hardware functions to implement the specified logic. Synthesizers use clock edges as the way to time a circuit.

HDLs are text-based expressions of the logical and timing characteristics of electronic systems. Like concurrent programming languages, HDL syntax and semantics includes notations for expressing concurrency. Languages whose only purpose is to express circuit connectivity between blocks are classified as computer-aided design languages.

The *automated* steps in using an HDL are the following:

Develop the logic diagram, using truth tables.

Generate the logic equations corresponding to the truth table relationships.

Minimize the logic equations, if necessary, using K-maps.

Use the simulator component of the HDL to verify the correct operation of the circuit logic, in particular, test timing constraints.

More details on HDL can be found in Salcic and Smailagic [SAL08].

SUMMARY

This chapter has provided the reader with numerous microprocessor design fundamentals and practical examples that lay the groundwork for the practicing engineer or student to design a complete microprocessor. In addition to elucidating principles, the chapter explained why circuits operate the way they do. Furthermore, there was a focus on design process to provide the reader with a road map to successful design. Last, many examples of digital logic were drawn from everyday experience to show the reader that the application of digital logic is not limited to designing microprocessors.

REFERENCES

[GIB80] G. A. Gibson and Y. Liu, *Microcomputers for Engineers and Scientists*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1980.

[GRE80] S. E. Greenfield, *The Architecture of Microcomputers*. Cambridge, MA: Winthrop Publishers, Inc., 1980.

[HAR07] D. M. Harris and S. L. Harris, *Digital Design and Computer Architecture*. New York: Elsevier, 2007.

[RAF05] M. Rafiquzzaman, *Fundamentals of Digital Logic and Microcomputer Design*. New York: Wiley-Interscience, 2005.

[SAL08] Z. Salcic and A. Smailagic, *Digital Systems Design and Prototyping: Using Field Programmable Logic and Hardware Description Languages*. Boston: Kluwer Academic Publishers, 2000.