

MODULE 1:**Introduction and Methodology:**

Digital Systems and Embedded Systems, Real-World Circuits, Models, Design Methodology (1.1, 1.3 to 1.5 of Text).

Combinational Basics: Combinational Components and Circuits, Verification of Combinational Circuits. (2.3 and 2.4 of Text)

Sequential Basics: Sequential Datapaths and Control Clocked Synchronous Timing Methodology (4.3 upto 4.3.1, 4.4 upto 4.4.1 of Text).

1.1 DIGITAL SYSTEMS AND EMBEDDED SYSTEMS

Digital refers to electronic circuits that represent information in a special way, using just two voltage levels. The main rationale for doing this is to increase the reliability and accuracy of the circuits. We can think of the two voltage levels as representing truth values, leading us to use rules of logic to analyze digital circuits. This gives us a strong mathematical foundation on which to build. The word *design* refers to the systematic process of working out how to construct circuits that meet given requirements while satisfying constraints on cost, performance, power consumption, size, weight and other properties.

Digital circuits were preceded by mechanical systems, electromechanical systems, and analog electronic systems. Most of these systems were used for numeric computations in business and military applications, for example, in ledger calculations and in computing ballistics tables. However, they suffered from numerous disadvantages, including inaccuracy, low speed, and high maintenance.

Early digital circuits, built in the mid-twentieth century, were constructed with relays. The contacts of a relay are either open, blocking current flow, or closed, allowing current to flow. Current controlled in this manner by one or more relays could then be used to switch other relays. However, even though relay-based systems were more reliable than their predecessors, they still suffered from reliability and performance problems. The advent of digital circuits based on vacuum tubes and, subsequently, transistors led to major improvements in reliability and performance. However, it was the invention of the *integrated circuit* (IC), in which multiple transistors were fabricated and connected together, that really enabled the “digital revolution.” As manufacturing technology has developed, the size of transistors and the interconnecting wires has shrunk. This, along with other factors, has led to ICs, containing billions of transistors and performing complex functions, becoming common place now.

The key is *abstraction*. By abstraction, we mean identifying aspects that are important to a task at hand, and hiding details of other aspects. The other aspects can't be ignored arbitrarily. Rather, we make assumptions and follow disciplines that allow us to ignore those details while we focus on the aspects of interest. As an example, the *digital abstraction* involves only allowing two voltage levels in a circuit, with transistors being either turned “on” (that is, fully conducting) or turned “off” (that is, not conducting). One of the assumptions we make in supporting this abstraction is that transistors switch on and off

virtually instantaneously. One of the design disciplines we follow is to regulate switching to occur within well-defined intervals of time, called “clock periods.”

The benefit of the digital abstraction is that it allows us to apply much simpler analysis and design procedures, and thus to build much more complex systems.

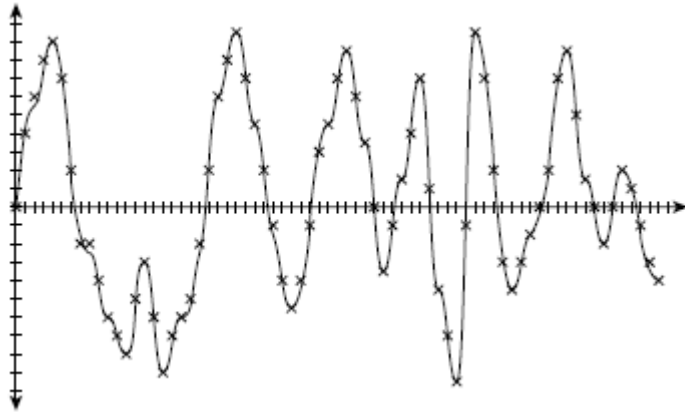


FIGURE 1.1 A pressure waveform of a sound, continuously varying over time, and the discrete representation of the waveform in a digital system.

Digital systems are electronic circuits that represent information in discrete form. An example of the kind of information that we might represent is an audio sound. In the real world, a sound is a continuously varying pressure waveform, which we might represent mathematically using a continuous function of time. However, representing that function with any significant precision as a continuously varying electrical signal in a circuit is difficult and costly, due to electrical noise and variation in circuit parameters. A digital system, on the other hand, represents the signal as a stream of discrete values sampled at discrete points in time, as shown in Figure 1.1.

Each sample represents an approximation to the pressure value at a given instant. The approximations are drawn from a discrete set of values, for example, the set $\{-10.0, -9.9, -9.8, \dots, -0.1, 0.0, 0.1, \dots, 9.9, 10.0\}$. By limiting the set of values that can be represented, we can encode each value with a unique combination of digital values, each of which is either a low or high voltage. Furthermore, by sampling the signal at regular intervals, say, every 50_s, the rate and times at which samples arrive and must be processed is predictable. Discrete representations of information and discrete sequencing in time are fundamental abstractions.

An embedded computer consists of a *processor core*, together with memory components for storing the program and data for the program to run on the processor core, and other components for transferring data between the processor core and the rest of the system. The programs running on processor cores form the *embedded software* of a system. The way in which embedded software is written bears both similarities and differences with software development for general purpose computers.

REAL-WORLD CIRCUITS

In order to analyze and design circuits as we have discussed, we are making a number of assumptions that underlie the digital abstraction. We have assumed that a circuit behaves in an ideal manner, allowing us to think in terms of 1s and 0s, without being concerned about the circuit's electrical behavior and physical implementation. Real-world circuits, however, are made of transistors and wires forming part of a physical device or package. The electrical properties of the circuit elements, together with the physical properties of the device or package, impose a number of constraints on circuit design. In this section, we will briefly describe the physical structure of circuit elements and examine some of the most important properties and constraints.

INTEGRATED CIRCUITS

Modern digital circuits are manufactured on the surface of a small flat piece of pure crystalline silicon, hence the common term "silicon chip." Such circuits are called integrated circuits, since numerous components are integrated together on the chip, instead of being separate components. Transistors are formed by depositing layers of semiconducting and insulating material in rectangular and polygonal shapes on the chip surface. Wires are formed by depositing metal (typically copper) on top of the transistors, separated by insulating layers.

The physical properties of the IC determine many important operating characteristics, including speed of switching between low and high voltages. Among the most significant physical properties is the minimum size of each element, the so-called *minimum feature size*. Early chips had minimum feature sizes of tens of microns (1 micron = 10^{-6} m). Improvements in manufacturing technology has led to a steady reduction in feature size, from $10\text{ }\mu\text{m}$ in the early 1970s, through $1\text{ }\mu\text{m}$ in the mid 1980s, with today's ICs having feature sizes of 90nm or 65nm. As well as affecting circuit performance, feature size helps determine the number of transistors that can fit on an IC, and hence the overall circuit complexity. Gordon Moore, one of the pioneers of the digital electronics industry, noted the trend in increasing transistor count, and published an article on the topic in 1965. His projection of a continuing trend continues to this day, and is now known as Moore's Law. It states that the number of transistors that can be put on an IC for minimum component cost doubles every 18 months. At the time of publication of Moore's article, it was around 50 transistors; today, a complex IC has well over a billion transistors.

One of the first families of digital logic ICs to gain widespread use was the "transistor-transistor logic" (TTL) family. Components in this family use bipolar junction transistors connected to form logic gates. The electrical properties of these devices led to widely adopted design standards that still influence current logic design practice. In more recent times, TTL components have been largely supplanted by components using "complementary metal-oxide semiconductor" (CMOS) circuits, which are based on field-effect transistors (FETs). The term "complementary" means that both n-channel and p-channel

MOSFETs are used. Figure 1.2 shows how such transistors are used in a CMOS circuit for an inverter. When the input voltage is low, the n-channel transistor at the bottom is turned off and the p-channel transistor at the top is turned on, pulling the output high. Conversely, when the input voltage is high, the p-channel transistor is turned off and the n-channel transistor is turned on, pulling the output low. Circuits for other logic gates operate similarly, turning combinations of transistors on or off to pull the output low or high, depending on the voltages at the inputs.

LOGIC LEVELS

The first assumption we have made is that all signals take on appropriate “low” and “high” voltages, also called *logic levels*, representing our chosen discrete values 0 and 1. As a consequence, there are now a number of “standards” for logic levels. One of the contributing factors to the early success of the TTL family was its adoption of uniform logic levels for all components in the family. These *TTL logic levels* still form the basis for standard logic levels in modern circuits. Suppose we nominate a particular voltage, 1.4V, as our *threshold voltage*. This means that any voltage lower than 1.4V is treated as a “low” voltage, and any voltage higher than 1.4V is treated as a “high” voltage.

In our circuits in preceding figures, we use the ground terminal, 0V, as our low voltage source. For our high voltage source, we used the positive power supply. Provided the supply voltage is above 1.4V, it should be satisfactory. (5V and 3.3V are common power supply voltages for digital systems, with 1.8V and 1.1V also common within ICs.) If components, such as the gates in Figure 1.5, distinguish between low and high voltages based on the 1.4V threshold, the circuit should operate correctly. In the real world, however, this approach would lead to problems. Manufacturing variations make it impossible to ensure that the threshold voltage is exactly the same for all components. So one gate may drive only slightly higher than 1.4V for a high logic level, and a receiving gate with a threshold a little more above 1.4V would interpret the signal as a low logic level. This is shown in Figure below

FIGURE 1.12 Problems due to variation in threshold voltage. The receiver would sense the signal as remaining low.

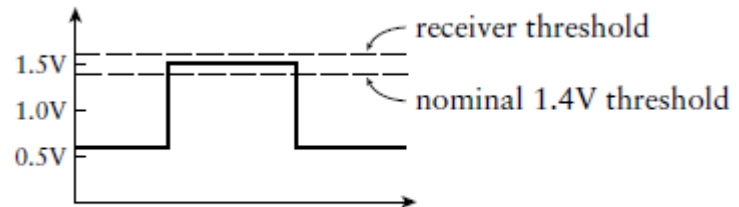
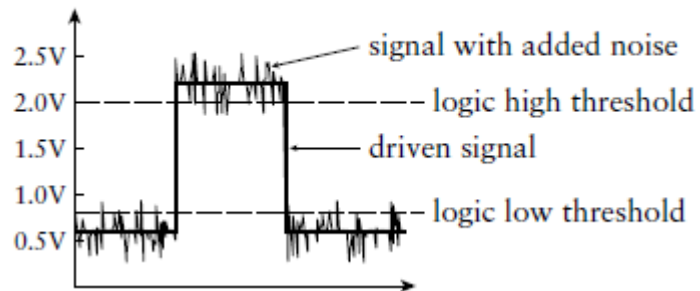
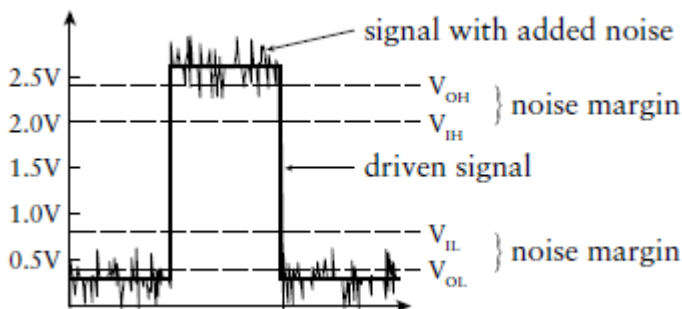


FIGURE 1.13 Problems due to noise on wires.



As a way of avoiding this problem, we separate the single threshold voltage into two thresholds. We require that a logic high be greater than 2.0V and a logic low be less than 0.8V. The range in between these levels is not interpreted as a valid logic level. We assume that a signal transitions through this range instantaneously, and we leave the behaviour of a component with an invalid input level unspecified. However, the signal, being transmitted on an electrical wire, might be subject to external interference and parasitic effects, which would appear as voltage noise. The addition of the noise voltage could cause the signal voltage to enter the illegal range, as shown in Figure 1.13, leading to unspecified behavior.

The final solution is to require components driving digital signals to drive a voltage lower than 0.4V for a “low” logic level and greater than 2.4V for a “high” logic level. That way, there is a *noise margin* for up to 0.4V of noise to be induced on a signal without affecting its interpretation as a valid logic level. This is shown in Figure 1.14. The symbols for the voltage thresholds are VOL: output low voltage—a component must drive a signal with a voltage below this threshold for a logic low VOH: output high voltage—a component must drive a signal with a voltage above this threshold for a logic high



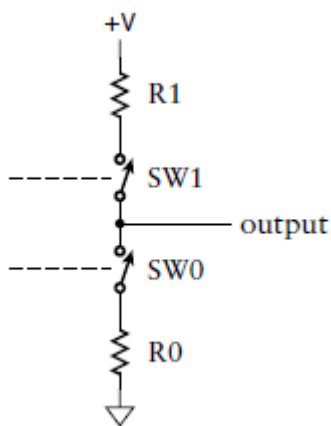
VIL: input low voltage—a component receiving a signal with a voltage below this threshold will interpret it as a logic low

VIH: input high voltage—a component receiving a signal with a voltage above this threshold will interpret it as a logic high

The behavior of a component receiving a signal in the region between V_{IL} and V_{IH} is unspecified. Depending on the voltage and other factors, such as temperature and previous circuit operation, the component may interpret the signal as a logic low or a logic high, or it may exhibit some other unusual behavior. Provided we ensure that our circuits don't violate the assumptions about voltages for logic levels, we can use the digital abstraction.

STATIC LOAD LEVELS

A second assumption we have made is that the current loads on components are reasonable. For example, in Figure 1.3, the gate output is acting as a source of current to illuminate the lamp.



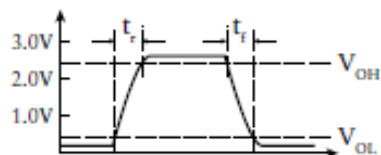
An idealized component should be able to source or sink as much current at the output as its load requires without affecting the logic levels. In reality, component outputs have some internal resistance that limits the current they can source or sink. An idealized view of the internal circuit of a CMOS component's output stage is shown in Figure. The output can be pulled high by closing switch SW1 or pulled low by closing switch SW0. When one switch is closed, the other is open, and *vice versa*. Each switch has a series resistance. (Each switch and its associated resistance is, in practice, a transistor with its on-state series resistance.) When SW1 is closed, current is sourced from the positive supply and flows through R1 to the load connected to the output. If too much current flows, the voltage drop across R1 causes the output voltage to fall below V_{OH} . Similarly, when SW0 is closed, the output acts as a current sink from the load, with the current flowing through R0 to the ground terminal. If too much current flows in this direction, the voltage drop across R0 causes the output voltage to rise above V_{OL} .

The amount of current that flows in each case depends on the output resistances, which are determined by component internal design and manufacture, and the number and characteristics of loads connected to the output. The current due to the loads connected to an output is referred to as the *static load* on the output. The term *static* indicates that we are only considering load when signal values are not changing. Each input draws a small amount of current when the input voltage is low and sources a small amount of current when the input is high. The amounts, again, are determined by component internal

design and manufacture. So, as designers using such components and seeking to ensure that we don't overload outputs, we must ensure that we don't connect too many inputs to a given output. We use the term *fanout* to refer to the number of inputs driven by a given output. Manufacturers usually publish current drive and load characteristics of components in data sheets. As a design discipline when designing digital circuits, we should use that information to ensure that we limit the fanout of outputs to meet the static loading constraints.

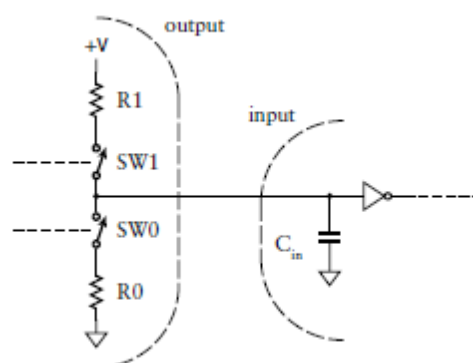
CAPACITIVE LOAD AND PROPAGATION DELAY

The signals change between logic levels instantaneously. In practice, level changes are not instantaneous, but take an amount of time that depends on several factors that we shall explore. The time taken for the signal voltage to rise from a low level to a high level is called the *rise time*, denoted by t_r , and the time for the signal voltage to fall from a high level to a low level is called the *fall time*, denoted by t_f . These are illustrated in Figure 1.16.



One factor that causes signal changes to occur over a nonzero time interval is the fact that the switches in the output stage of a digital component, illustrated in Figure, do not open or close instantaneously.

Rather, their resistance changes between near zero and a very large value over some time interval. However, a more significant factor, especially in CMOS circuits, is the fact that logic gates have a significant amount of capacitance at each input.



Thus, if we connect the output of one component to the input of another, the input capacitance must be charged and discharged through the output stage's switch resistances in order to change the logic level on the connecting signal.

If we connect a given output to more than one input, the capacitive loads of the inputs are connected in parallel. The total capacitive load is thus the sum of the individual capacitive loads. The effect is to make transitions on the connecting signal correspondingly slower. For CMOS components, this effect is much

more significant than the static load of component inputs. Since we usually want circuits to operate as fast as possible, we are constrained to minimize the fanout of outputs to reduce capacitive loading.

A similar argument regarding time taken to switch transistors on and off and to charge and discharge capacitance also applies within a digital component. Without going into the details of a component's circuit, we can summarize the argument by saying that, due to the switching time of the internal transistors, it takes some time for a change of logic level at an input to cause a corresponding change at the output. We call that time the *propagation delay*, denoted by t_{pd} , of the component. Since the time for the output to change depends on the capacitive load, component data sheets that specify propagation delay usually note the capacitive load applied in the test circuit for which the propagation delay was measured, as well as the input capacitance.

WIRE DELAY

A change in the value of a signal at the output of a component is seen instantaneously at the input of other connected components. In other words, we've assumed that wires are perfect conductors that propagate signals with no delay. For very short wires, that is, wires of a few centimeters on a circuit board or up to a millimeter or so within an IC, this assumption is reasonable, depending on the speed of operation of the circuit. For longer wires, however, we need to take care when designing high-speed circuits. Problems can arise from the fact that such wires have parasitic capacitance and inductance that are not negligible and that delay propagation of signal changes.

Such wires should be treated as transmission lines and designed carefully to avoid unwanted effects of reflection of wavefronts from stubs and terminations. A detailed treatment of design techniques in these cases is beyond the scope of this book. However, we need to be aware that relatively long wires add to the overall propagation delay of a circuit. Later, we will describe the use of computer-based tools that can help us to understand the effects of wire delays and to design our circuits appropriately.

SEQUENTIAL TIMING

A flip-flop stores the value of its data input at the moment the clock input rises from 0 to 1. Moreover, we assumed that the stored value is reflected on the output instantaneously. It should not be surprising now that these assumptions are an abstraction of realistic sequential circuit behavior, and that we must observe design disciplines to support the abstraction. Real flip-flops require that the value to be stored be present on the data input for an interval, called the *setup time*, before the rising clock edge. Also, the value must not change during that interval and for an interval, called the *hold time*, after the clock edge.

Finally, the stored value does not appear at the output instantaneously, but is delayed by an interval called the *clock-to-output delay*. In this timing diagram, we have drawn the rising and falling edges as

sloped, rather than vertical, to suggest that the transitions are not instantaneous. We have also drawn both 0 and 1 values for the data input and output, suggesting that it is not the specific values that are relevant, but the times at which values change, shown by the coincident rising and falling values. The diagram illustrates the constraint that changes on the data input must not occur within a time window around the clock rising edge, and that the data output cannot be assumed correct until after some delay after the clock edge. In most sequential digital circuits, the output of one flip-flop is either connected directly to the data input of another, or passes through some combinational logic whose output is connected to the data input of another flip-flop. In order for the circuit to operate correctly, a data output resulting from one clock edge must arrive at the second flip-flop ahead of a setup interval before the next clock edge.

This gives rise to a constraint that we can interpret in two ways. One view is that the delays in the circuit between flip-flops are fixed and place an upper bound on the clock cycle time, and hence on the overall speed at which the circuit operates. The other view is that the clock cycle time is fixed and places an upper bound on the permissible delays in the circuit between flip-flops. According to this view, we must ensure that we design the circuit to meet that constraint.

POWER

Many modern applications of digital circuits require consideration of the power consumed and dissipated. Power consumption arises through current being drawn from a constant-voltage power supply. All gates and other digital electronic components in a circuit draw current to operate the transistors in their internal circuitry, as well as to switch input and output transistors on and off. While the current drawn for each gate is very small, the total current drawn by millions of them in a complete system can be many amperes. When the power supply consists of batteries, for example, in portable appliances such as phones and notebook computers, reducing power consumption allows extended operating time.

The electrical power consumed by the current passing through resistance causes the circuit components to heat up. The heat serves no useful purpose and must be exhausted from the circuit components. Designers of the physical packaging of ICs and complete electronic systems determine the rate at which thermal energy can be transferred to the surroundings. As circuit designers, we must ensure that we do not cause more power dissipation than can be handled by the thermal design, otherwise the circuit will overheat and fail. Puffs of blue smoke are the usual sign of this happening!

There are two main sources of power consumption in modern digital components. The first of these arises from the fact that transistors, when turned off, are not perfect insulators. There are relatively small

leakage currents between the two terminals, as well as from the terminals to ground. These currents cause *static power* consumption. The second source of power consumption arises from the charging and discharging of load capacitance when outputs switch between logic levels. This is called *dynamic power* consumption. To a first approximation, the static power consumption occurs continuously, independent of circuit operation, whereas dynamic power consumption depends on how frequently signals switch between logic levels.

As designers, we have control over both of these forms of power consumption. We can control static power consumption of the circuit by choosing components with low static power consumption, and, in some cases, by arranging for parts of circuits that are not needed for periods of time to be turned off. We can control dynamic power consumption by reducing the number and frequency of signal transitions that must occur during circuit operation. This is becoming an increasingly important part of the design activity, and computer-based tools for power analysis are gaining increased use.

AREA AND PACKAGING

In most applications of digital electronics, cost of the final manufactured product is an important factor, particularly when the product is to be sold in a competitive market. There are numerous factors that influence cost, many of them based on business decisions rather than engineering design decisions. However, one factor over which designers have control and that strongly affects the final product cost is circuit area.

As mentioned earlier, digital circuits are generally implemented as integrated circuits, in which transistors and wires are chemically formed on the surface of a wafer of crystalline silicon. The more transistors and wires in our circuit, the more surface area it consumes. The manufacturing process for ICs is based on wafers of a fixed size, up to 300mm in diameter, with a fixed cost of manufacture. Multiple ICs are manufactured on a single wafer in a series of steps. The wafer is then broken into individual ICs, which are encapsulated in packages that can be soldered onto the circuit board of a complete system. Thus, the larger the individual IC, the fewer there are per wafer, and so the greater their cost.

Unfortunately, the manufacturing process is not perfect, so defects occur, distributed across the surface of the wafer. Those ICs that have a defect that cause them not to function correctly are discarded. Since the cost of manufacturing a wafer is fixed, the functional ICs must bear the cost of those that are nonfunctional, increasing the final product cost of the IC. The larger an individual IC, the greater the proportion that have defects. So the final cost of an IC is disproportionately dependent on area. Since

each IC is packaged individually, the cost of the package is a direct cost of the final product. The package serves two purposes.

One is to provide connection pins, allowing the wires of the IC to be connected to external wires in the larger digital system, as well as providing for power supply and ground connections. An IC with more external connections requires more pins and, thus, a more costly package. Therefore, the pin count of the IC is a factor that constrains our designs. The other purpose served by the IC package is to transfer heat from the IC to the surroundings so that the IC does not overheat. If the IC generates more thermal power than the package can dissipate, additional cooling devices, such as heat sinks, fans or heat pipes, are required, adding to cost. Thus, thermal concerns arising from packaging also constrain our designs.

Problem: Develop a Verilog model that expresses the logical structure of the gate circuit in Figure below. Assume that the sensor signals and the switch signal are inputs to the model, and that the buzzer signal is the output from the model.

Solution: Suppose a factory has two vats, only one of which is used at a time. The liquid in the vat in use needs to be at the right temperature, between 25°C and 30°C. Each vat has two temperature sensors indicating whether the temperature is above 25°C and above 30°C, respectively. The vats also have lowlevel sensors. The supervisor needs to be woken up by a buzzer when the temperature is too high or too low or the vat level is too low. He has a switch to select which vat is in use. Design a circuit of gates to activate the buzzer as required. solution For the selected vat, the condition for activating the buzzer is “temperature not above 25°C or temperature above 30°C, or level low.” This can be implemented with a gate circuit for each vat. The switch can be used to control the select input of a multiplexer to choose between the circuit outputs for the two vats. The output of the multiplexer then activates the buzzer. The complete circuit is shown in Figure

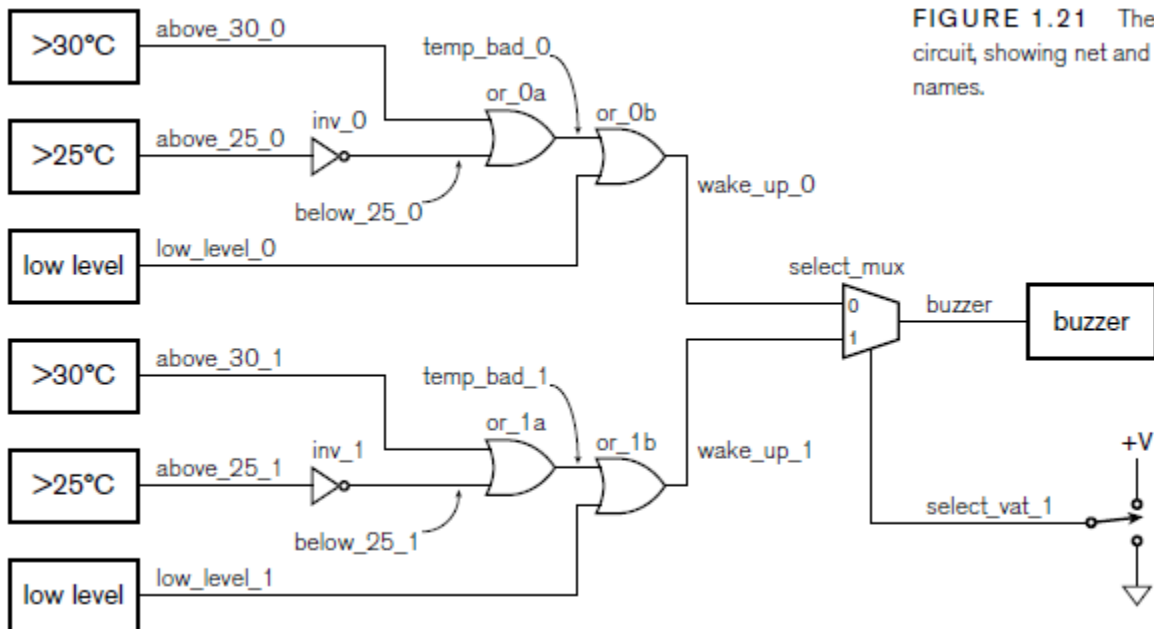


FIGURE 1.21 The vat buzzer circuit, showing net and component names.

Solution :

```

module vat_buzzer_struct ()
output buzzer,
input above_25_0, above_30_0, low_level_0,
input above_25_1, above_30_1, low_level_1,
input select_vat_1 );
wire below_25_0, temp_bad_0, wake_up_0;
wire below_25_1, temp_bad_1, wake_up_1;
// components for vat 0
not inv_0 (below_25_0, above_25_0);
or or_0a (temp_bad_0, above_30_0, below_25_0);
or or_0b (wake_up_0, temp_bad_0, low_level_0);
// components for vat 1
not inv_1 (below_25_1, above_25_1);
or or_1a (temp_bad_1, above_30_1, below_25_1);
or or_1b (wake_up_1, temp_bad_1, low_level_1);
mux2 select_mux (buzzer, select_vat_1, wake_up_0, wake_up_1);
endmodule

```

Develop a Verilog model that expresses the function performed by the gate circuit for the above

```
module vat_buzzer_behavior ( output buzzer,  
  
input above_25_0, above_30_0, low_level_0,  
  
input above_25_1, above_30_1, low_level_1,  
  
input select_vat_1 );  
  
assign buzzer = select_vat_1 ? low_level_1 | (above_30_1 | ~above_25_1): low_level_0 |  
                (above_30_0 | ~above_25_0);  
  
endmodule
```

DESIGN METHODOLOGY

Designing a digital system of any significant complexity is a large undertaking, requiring a systematic approach. This is especially important when many people are collaborating on a design, as is usually the case. Depending on the complexity of the product, design teams can range in size from a handful of engineers for a relatively simple product, to several hundred people for a complex IC or packaged system. We use the term design methodology to refer to the systematic process of design, verification and preparation for manufacture of a product. A design methodology specifies the tasks undertaken, the information required and produced by each task, the relationships between the tasks, including dependencies and sequencing, and the CAD tools used.

A mature design methodology will also be reflective, specifying measurements that will be made of the design process, such as adherence to schedule and budget, and numbers of design errors detected and missed. Accumulated data from previous projects can be used to improve the design methodology for subsequent projects. The benefit of a good methodology is that it makes the design process more reliable and predictable, thus reducing risk and cost. Even a small design project benefits from a design methodology, though perhaps on a reduced scale.

The starting point is a collection of requirements and constraints. These are usually generated externally to the design team, for example, by the marketing group of a company or by a customer for whom a product development is undertaken. They usually include function requirements (what the product is to do), performance requirements (how fast it is to do it), and constraints on power consumption, cost and packaging.

The design methodology specifies three tasks—design, synthesis and physical implementation—each of which is followed by a verification task. (The design and functional verification tasks are outlined to

indicate that they are actually a bit more involved than is shown on the diagram. We will come back to this shortly.) If verification fails at any stage, we must revisit a previous task to correct the error. Ideally, we would like to revisit only the immediately preceding task and make a minor correction. However, if the error is severe enough, we may need to backtrack further to make more significant changes. Hence, when performing a given design task, it is worth keeping in mind the constraints applying in subsequent tasks, so as to avoid introducing errors that will be detected later.

Once the tasks and associated verification activities have been performed, the product can be manufactured, and each unit tested to ensure that it is functional. The design task involves understanding the requirements and constraints and developing a specification of a digital circuit that meets the requirements and constraints. The information produced by this task is a collection of models that describe the design. The methodology then specifies that we verify the function of the design, using techniques such as simulation and formal verification. In preparation for the verification task, we should prepare a *verification plan* that identifies what input and output cases should be verified, and what CAD tools should be used.

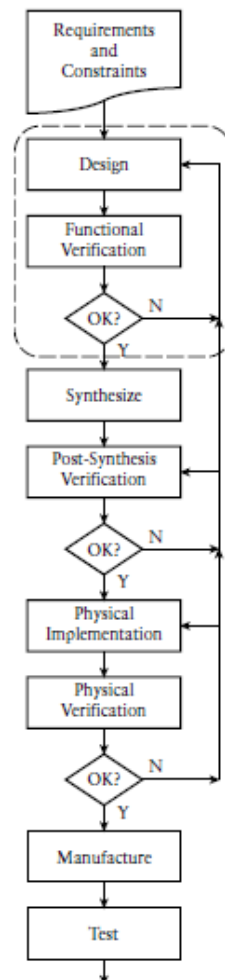


Fig: Simple Design Methodology flow

Hierarchical composition in a design also makes functional verification more tractable. We can first verify each of the most primitive subcircuits as independent units. Next, we can verify a subsystem that uses the subcircuits by treating the subsystem as a collection of black boxes. In particular, we can use more abstract models of the black box subcircuits instead of detailed models that describe their internal composition. This approach means that the verification tools have less work to do, allowing them to verify more input/output cases for the subsystem. We can repeat this process until we have verified the entire system.

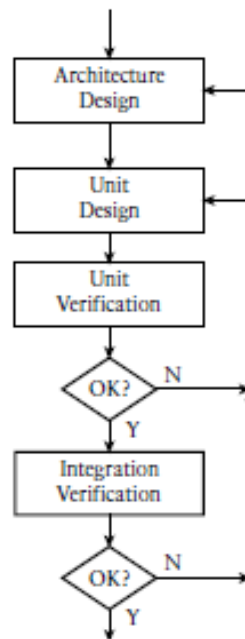


Fig: Hierarchical Design and Methodologies

EMBEDDED SYSTEMS DESIGN

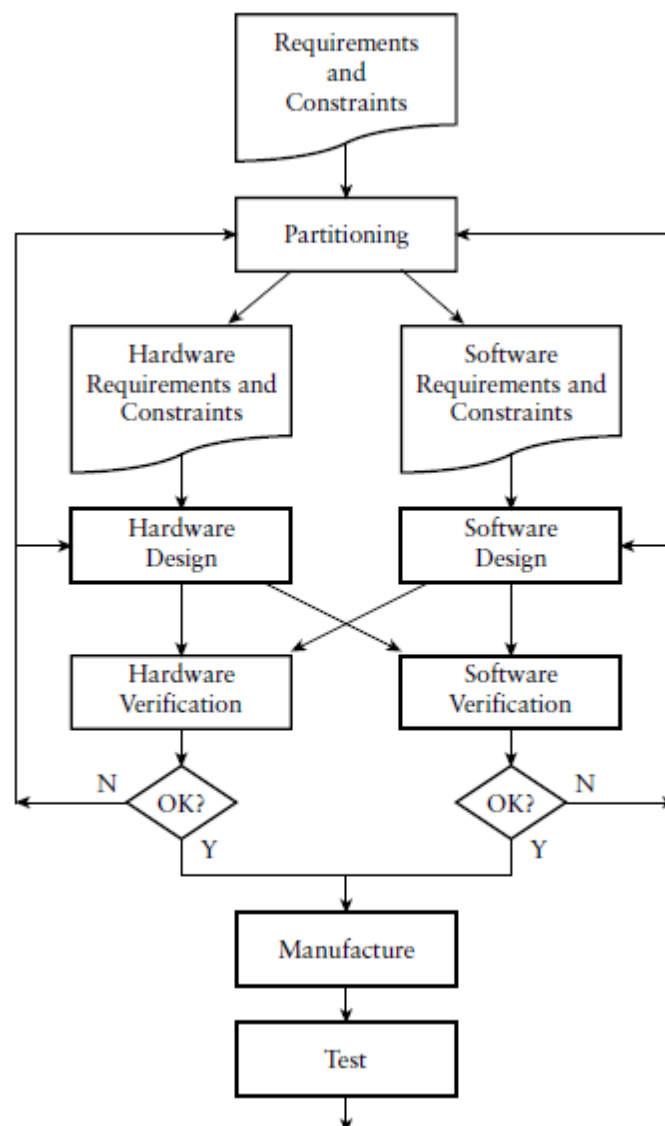
Each embedded computer comprises a processor core, memory and interfaces with other parts of the system. Since the computers must be programmed to implement part of the system's functionality, we must augment our design methodology to include embedded software design. Recall that the initial inputs to the design methodology are the functional requirements and constraints for the system. As part of our architectural design considerations, we can choose which aspects of functionality can be implemented by embedded software on a processor core, and which parts can be implemented as digital subcircuits, that is, by hardware. Designing the hardware and software for a system together is called *hardware/software codesign*.

Deciding which parts to put in hardware and which in software is called *partitioning*. There are numerous trade-offs to consider. Functionality that involves testing many conditions and taking alternative actions can be hard to implement in hardware, but is relatively straightforward in software. On the other hand, functionality that involves performing rapid computations on large amounts of data or data that arrives at

a high rate may need a very high performance (and hence costly and power hungry) processor core, and so may more readily be performed by customized hardware.

A further consideration is that embedded software may be stored in memory circuits that may be reprogrammed after the system is manufactured or deployed. Thus, the software may be upgraded to correct design errors or add functionality without revising the hardware design or replacing deployed systems.

Once functionality has been partitioned between hardware and software, development of the two can proceed concurrently, as shown in Figure below. For those aspects of the embedded software that depend on hardware, the abstract behavioral models from the hardware design task can be used to verify the software design. This can be done using an instruction-set simulator for the processor core working in tandem with a simulator for the hardware model. A similar approach can be used to verify parts of the hardware that interface directly with a processor core.



Test programs can be run using the processor simulator running in tandem with the hardware simulator. The benefits of developing hardware and software concurrently include avoiding the extra time involved in developing one after the other, and early detection of errors in the interplay of software and hardware.

2.3 COMBINATIONAL COMPONENTS AND CIRCUITS

2.3.1 DECODE RS AND ENCODERS

information can be binary coded. In many designs, we need to derive a number of control signals from a binary coded signal, with one control signal corresponding to each valid code word. When the encoded signal takes on a given code value, the corresponding control signal is activated. We call a circuit that derives the control signals in this way a *decoder*. For an n -bit code, if every code word is valid, the decoder will have $2n$ outputs. Decoders are an important building block in memory designs.

We can derive the Boolean equation for each output of a decoder by looking at the corresponding code word. To illustrate, suppose we have an encoded 4-bit input signal (a_3, a_2, a_1, a_0) , and we need to determine the Boolean equation for the output corresponding to the code word 1011.

Each is the logical AND of the input bits, either directly (for bits that are 1 in the corresponding code word) or negated (for bits that are 0 in the corresponding code word).

Problem: Develop a Verilog model for a decoder for use in the inkjet printer . The decoder has three input bits representing the choice of color cartridge and six output bits, one to select each cartridge.

Solution: A module with assignment statements representing the Boolean equations for the outputs is

```
module ink_jet_decoder ( output black, cyan, magenta, yellow, light_cyan, light_magenta, input color2,
color1, color0 );
assign black = ~color2 & ~color1 & color0;
assign cyan = ~color2 & color1 & ~color0;
assign magenta = ~color2 & color1 & color0;
assign yellow = color2 & ~color1 & ~color0;
assign light_cyan = color2 & ~color1 & color0;
assign light_magenta = color2 & color1 & ~color0;
endmodule
```

If an invalid code occurs on the input bits, none of the outputs is activated. This can be considered a “fail safe” design.

The inverse of a decoder is called an *encoder*. It has, as inputs, a number of single-bit signals, and as outputs, a collection of signals representing the bits of an encoded value. We will assume for the moment that at most one of the inputs is 1 at any time, and the others are all 0. The code word at the output corresponds to the particular input that is 1.

We can derive the Boolean equation for each bit of the output by identifying those inputs for which the output bit is 1. The output bit is then the logical OR of those inputs. However, we need to take account of the possibility that none of the inputs is 1, since that would cause our encoder to output a code word of all 0 bits. If that code word is invalid, we can use it to imply that no inputs are 1, essentially extending the code.

Alternatively, if the all-0s code word is valid and corresponds to one of the inputs being 1, we need to have a separate output that indicates when any of the inputs is 1. When this output is 0, we ignore the code word produced by the encoder.

Problem: Design an encoder for use in a domestic burglar alarm that has sensors for each of eight zones. Each sensor signal is 1 when an intrusion is detected in that zone, and 0 otherwise. The encoder has three bits of output, encoding the zone as follows:

Zone 1: 000 Zone 2: 001 Zone 3: 010 Zone 4: 011

Zone 5: 100 Zone 6: 101 Zone 7: 110 Zone 8: 111

Solution Since all code words are used, we need a separate output to indicate when there is a valid code-word output. The module definition is

```

module alarm_eqn ( output [2:0] intruder_zone, output valid, input [1:8] zone );
assign intruder_zone[2] = zone[5] | zone[6] | zone[7] | zone[8];
assign intruder_zone[1] = zone[3] | zone[4] | zone[7] | zone[8];
assign intruder_zone[0] = zone[2] | zone[4] | zone[6] | zone[8];
assign valid = zone[1] | zone[2] | zone[3] | zone[4] | zone[5] | zone[6] | zone[7] | zone[8];
endmodule

```

The left-most bit of the output code is 1 when any of the zone 5 through zone 8 inputs is 1, so the equation for that output is the logical OR of those zone inputs. The equations for the other two output code bits are derived similarly. The valid output is the logical OR of all of the zone inputs.

Problem: Revise the encoder for the burglar alarm to be a priority encoder, with zone 1 having highest priority, down to zone 8 having lowest priority.

Solution: The port list is unchanged, since we need the same inputs and outputs for the encoder. The truth table for the priority encoder is shown below

zone								intruder_zone			
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(2)	(1)	(0)	valid
1	-	-	-	-	-	-	-	0	0	0	1
0	1	-	-	-	-	-	-	0	0	1	1
0	0	1	-	-	-	-	-	0	1	0	1
0	0	0	1	-	-	-	-	0	1	1	1
0	0	0	0	1	-	-	-	1	0	0	1
0	0	0	0	0	1	-	-	1	0	1	1
0	0	0	0	0	0	1	-	1	1	0	1
0	0	0	0	0	0	0	1	1	1	1	1
0	0	0	0	0	0	0	0	-	-	-	0

```
module alarm_priority ( output [2:0] intruder_zone, output valid, input [1:8] zone );
```

```
wire [1:8] winner;
```

```
assign winner[1] = zone[1];
```

```
assign winner[2] = zone[2] & ~zone[1];
```

```
assign winner[3] = zone[3] & ~(zone[2] | zone[1]);
```

```
assign winner[4] = zone[4] & ~(zone[3] | zone[2] | zone[1]);
```

```
assign winner[5] = zone[5] & ~( zone[4] | zone[3] | zone[2] | zone[1]);
```

```
assign winner[6] = zone[6] & ~( zone[5] | zone[4] | zone[3] | zone[2] | zone[1]);
```

```
assign winner[7] = zone[7] & ~( zone[6] | zone[5] | zone[4] | zone[3] | zone[2] | zone[1]);
```

```
assign winner[8] = zone[8] & ~( zone[7] | zone[6] | zone[5] | zone[4] | zone[3] | zone[2] | zone[1]);
```

```
assign intruder_zone[2] = winner[5] | winner[6] | winner[7] | winner[8];
```

```
assign intruder_zone[1] = winner[3] | winner[4] | winner[7] | winner[8];
```

```
assign intruder_zone[0] = winner[2] | winner[4] | winner[6] | winner[8];
```

```
assign valid = zone[1] | zone[2] | zone[3] | zone[4] | zone[5] | zone[6] | zone[7] | zone[8];
```

```
endmodule
```

Another way of expressing this in Verilog is shown in the following module:

```

module alarm_priority_1 ( output [2:0] intruder_zone, output valid, input [1:8] zone );

assign intruder_zone = zone[1] ? 3'b000 :
zone[2] ? 3'b001 :
zone[3] ? 3'b010 :
zone[4] ? 3'b011 :
zone[5] ? 3'b100 :
zone[6] ? 3'b101 :
zone[7] ? 3'b110 :
zone[8] ? 3'b111 :
3'b000;

assign valid = zone[1] | zone[2] | zone[3] | zone[4] | zone[5] | zone[6] | zone[7] | zone[8];

endmodule

```

Problem: Develop a Verilog model for a 4-to-1 multiplexer.

Solution: The module definition is

```

module multiplexer_4_to_1 ( output reg z, input [3:0] a, input sel );

always @ *

case (sel)

2'b00: z _ a[0];

2'b01: z _ a[1];

2'b10: z _ a[2];

2'b11: z _ a[3];

endcase

endmodule

```

MODULE 2- MEMORIES

Syllabus: Concepts, Memory Types, Error Detection and Correction

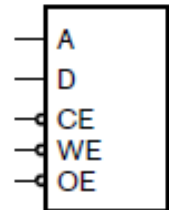
MEMORY TYPES

1) ASYNCHRONOUS STATIC RAM

- One of the simplest forms of memory is asynchronous static RAM. It is called *asynchronous* because it does not depend on a clock for its timing.
- The term *static* means that the stored data remains indefinitely as long as power is applied to the memory component. Static RAM is *volatile*, that is, it requires power to maintain the stored data, and loses data if power is removed.
- Asynchronous SRAM internally uses 1-bit storage cells that are similar to the D-latch. Within the memory component, the address is decoded to select a particular group of cells that comprise one location.
 - For a write operation, the selected latch cells are enabled and the input data is stored.
 - For a read operation, the address activates a multiplexer that routes the outputs of the selected latch cells to the data outputs of the memory component.
- Asynchronous SRAMs are usually available as packaged integrated circuits, hence they have bidirectional tristate data input/output pins.

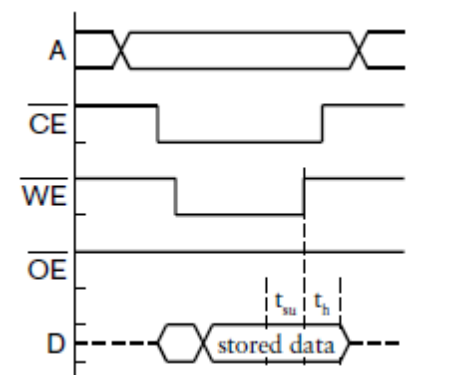
- Figure aside shows a symbol for a typical asynchronous SRAM.

- The chip-enable input (\overline{CE}) is used to enable or disable the memory chip. This signal is driven from a select control signal, for example, from an address decoder in a composite memory.
- The write-enable input (\overline{WE}) controls whether the memory performs a write or read operation if it is enabled.
- The output-enable input (\overline{OE}) controls the tristate data drivers during a read operation. When (\overline{OE}) is low during a read, the drivers are enabled and can drive the read data onto the data pins. When (\overline{OE}) is high, the drivers are in the high-impedance state.



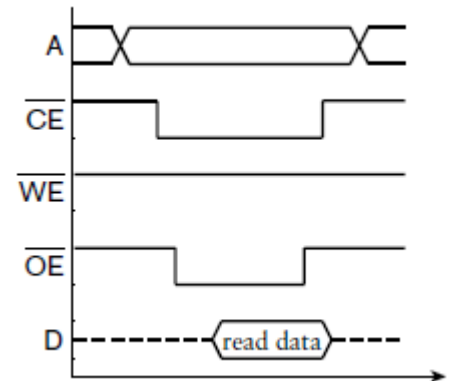
- The sequence of signals to perform a write operation is shown in Figure aside.

- The control section that sequences the datapath containing the memory must ensure that the address is stable before commencing the write operation and is held stable during the entire operation.
- The control section selects the particular memory chip by driving \overline{CE} low, activates the write operation by driving \overline{WE} low, and ensures that the chip's tristate drivers are disabled by driving \overline{OE} high.
- The final data to be stored must be stable on the data signals a setup time before the rising edge of the \overline{WE} signal or the \overline{CE} signal, whichever occurs first. The data and the address must also remain stable for a hold time after \overline{WE} or the \overline{CE} signal goes high.



➤ The sequence of signals for a read operation is similar, and is shown in Figure aside.

- The difference is that the \overline{WE} signal is held high, and the \overline{OE} signal is driven low to enable the memory chip's tristate drivers.
- The read operation is a combinational operation, involving decoding the address and multiplexing the selected latch-cell's value onto the data outputs. Changing the address simply causes a different cell's value to appear on the outputs after a propagation delay.



➤ *Access time*, is the delay from the start of a read operation to having valid data at the outputs.

➤ *Write cycle time* and the *read cycle time*, are the times taken to complete write and read operations, respectively.

➤ It is difficult to use asynchronous SRAM clocked synchronous systems. The need to set up and hold address and data values before and after activation of the control signals and to keep the values stable during the entire cycle means that we must either perform operations over multiple clock cycles, or use delay elements to ensure correct timing within a clock cycle. The former approach reduces performance, and the latter approach violates assumptions inherent in the clocked synchronous methodology, and so complicates timing design and analysis. For these reasons, asynchronous SRAMs are usually used only in systems with low performance requirements, where their low cost is a benefit.

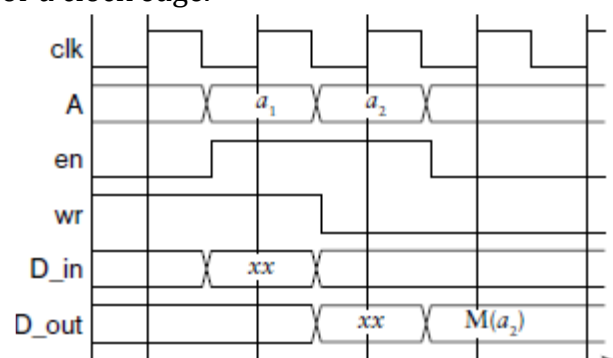
2) SYNCHRONOUS STATIC RAM

➤ *Synchronous SRAMs*, known as SSRAMs are similar as those of asynchronous SRAMs but includes clocked registers for storing the address, input data and control signal values.

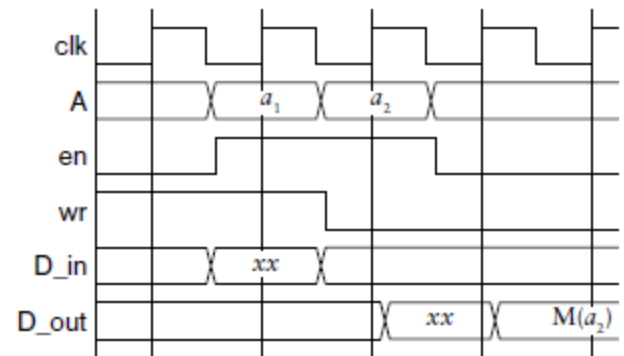
➤ The simplest kind of SSRAM is often called a *flow-through* SSRAM. It includes registers on the inputs, but not on the data outputs. The term flow-through refers to the fact that data read from the memory cells flows through directly to the data outputs. Having registers on the inputs allows us to generate the address, data and control signal values according to our clocked synchronous design methodology, ensuring that they are stable in time for a clock edge.

➤ Figure illustrates the timing for a flow-through SSRAM.

- During the first clock cycle, we set up the address (a_1), control signals and input data (xx) in preparation for a write operation.
- These values are stored in the input registers on the next clock edge, causing the SSRAM to start the write operation.
- The data is stored and flows through to the output during the second clock cycle. While that happens, we set up the address (a_2) and control signals in preparation for a read operation.
- Again, these values are stored on the next clock edge, and during the third cycle the SSRAM performs the read operation. The data, denoted by $M(a_2)$, flows through from the memory to the output. In the third cycle, the enable signal is set to 0. This prevents the input registers from being updated on the next clock edge, so the previously read data is maintained at the output.



- Another form of SSRAM is called a *pipelined* SSRAM. It includes a register on the data output, as well as registers on the inputs. A pipelined SSRAM is useful in higher-speed systems where the access time of the memory is very significant. If there is no time to perform combinational operations on the read data before the next clock edge, it needs to be stored in an output register and used in the subsequent clock cycle.
- The timing for a pipelined SSRAM is illustrated in Figure
 - Timing for the inputs is the same as that for a flow-through SSRAM. The difference is that the data output does not reflect the result of a read or write operation until one clock cycle later, immediately after the clock edge marking the beginning of that cycle.



Modeling Pipelined SSRAM in Verilog

```

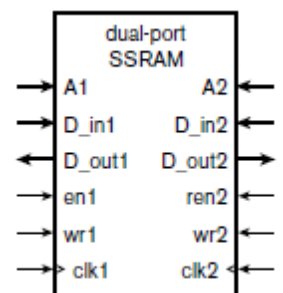
reg    pipelined_en;
reg [15:0] pipelined_d_out;
...

always @(posedge clk) begin
    if (pipelined_en) d_out <= pipelined_d_out;
    pipelined_en <= en;
    if (en)
        if (wr) begin
            data_RAM[a] <= d_in; pipelined_d_out <= d_in;
        end
        else
            pipelined_d_out <= data_RAM[a];
    end
end

```

3) MULTIPORT MEMORIES

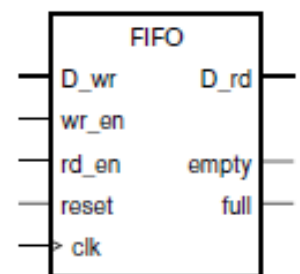
- A single-port memory can perform only one access (a write or a read operation) at a time. In contrast, a *multiport* memory has multiple address inputs, with corresponding data inputs and outputs. It can perform as many operations concurrently as there are address inputs.
- The most common form of multiport memory is a *dual-port* memory, illustrated in Figure which can perform two operations concurrently.
- A multiport memory typically consumes more circuit area than a single-port memory with the same number of bits of storage, since it has separate address decoders and data multiplexers for each access port.
- The cost of the extra circuit area is necessary in some applications, such as high performance graphics processing and high-speed network connections.
- Having separate access ports for the subsystems has a problem that is the case of both subsystems accessing the same memory location at the same time.



- If both accesses are reads, they can proceed.
 - If one or both is a write, the effect depends on the characteristics of the particular dual-port memory. In an asynchronous dual-port memory, a write operation performed concurrently with a read of the same location will result in the written data being reflected on the read port after some delay.
 - Two write operations performed concurrently to the same location result in an unpredictable value being stored.
 - In the case of a synchronous dual-port memory, the effect of concurrent write operations depends on when the operations are performed internally by the memory.
- Some multiport memories, provide additional circuits that compare the addresses on the access ports and indicate when problem arises. They may also provide circuits to arbitrate between conflicting accesses, ensuring that one access proceeds only after the other has completed. If we use multiport memory components or circuit blocks that do not provide such features then our application may result in conflicting accesses.
- An alternative is to ensure that the subsystems accessing the memory through separate ports always access separate locations, for example, by ensuring that they always operate on different blocks of data stored in different parts of the memory.

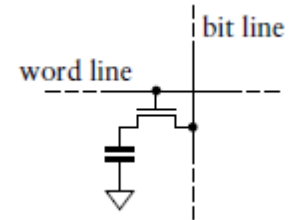
FIFO

- One specialized form of dual-port memory is a *first-in first-out* memory, or FIFO. It is used to queue data arriving from a source to be processed in order of arrival by another subsystem.
- The data that is first in to the FIFO is the first that comes out; hence, the name. The most common way of building a FIFO is to use a dual-port memory as a *circular buffer* for the data storage, with one port accepting data from the source and the other port reading data to provide to the processing subsystem.
- Each port has an address counter to keep track of where data is written or read. Data written to the FIFO is stored in successive free locations.
- When the write-address counter reaches the last location, it wraps to location 0. As data is read, the read-address counter is advanced to the next available location, also wrapping to 0 when the last location is reached.
- If the write address wraps around and catches up with the read address, the FIFO is full and can accept no more data.
- If the read address catches up with the write address, the FIFO is empty and can provide no more data.
- Symbol for a FIFO with empty and full status outputs is shown in figure.
- Thus, a FIFO can store a variable amount of data, depending on the rates of writing and reading data. The size of memory needed in a FIFO depends on the maximum amount by which reading of data lags writing.
- One important use for FIFOs is to pass data between subsystems operating with different clock frequencies, that is, between different *clock domains*.
- A FIFO allows us to smooth out the flow of data between the domains. Data arriving is written into the FIFO synchronously with the sending domain's clock, and the receiving domain reads data synchronously with its clock.



4) DYNAMIC RAM

- *Dynamic RAM (DRAM)* is another form of volatile memory that uses a different form of storage cell for storing data. A storage cell for a dynamic RAM uses a single capacitor and a single transistor, illustrated in Figure. The DRAM cells are thus much smaller than SRAM cells, so we can fit many more of them on a chip, making the cost per bit of storage lower.
- However, the access times of DRAMs are longer than those of SRAMs, and the complexity of access and control is greater. Thus, there is a trade-off of cost, performance and complexity against memory capacity. DRAMs are most commonly used as the main memory in computer systems, since they satisfy the need for high capacity with relatively low cost.
- A DRAM represents a stored 1 or 0 bit in a cell by the presence or absence of charge on the capacitor. When the transistor is turned off, the capacitor is isolated from the bit line, thus storing the charge on the capacitor.
- To write to the cell, the DRAM control circuit pulls the bit line high or low and turns on the transistor, thus charging or discharging the capacitor.
- To read from the cell, the DRAM control circuit precharges the bit line to an intermediate level, then turns on the transistor. As the charges on the capacitor and the bit line equalize, the voltage on the bit line either increases slightly or decreases slightly, depending on whether the storage capacitor was charged or discharged.
- A sensor detects and amplifies the change, thus determining whether the cell stored a 1 or a 0. Unfortunately, this process destroys the stored value in the cell, so the control circuit must then restore the value by pulling the bit line high or low, as appropriate, before turning off the transistor. The time taken to complete the restoration is added to the access time, making the overall read cycle significantly longer than that for an SRAM.
- Another property of a DRAM cell is that, while the transistor is turned off, charge leaks from the capacitor. This is the meaning of the term “dynamic” applied to DRAMs. To compensate, the control circuit must read and restore the value in each cell in the DRAM before the charge decays too much. This process is called **refreshing** the DRAM.
- DRAM manufacturers typically specify a period of 64ms between refreshes for each cell. The cells in a DRAM are typically organized into several rectangular arrays, called banks, and the DRAM control circuit is organized to refresh one row of each bank at a time. Since the DRAM cannot perform a normal write or read operation while it is refreshing a row, the refresh operations must be interleaved between writes and reads. Depending on the application, it may be possible to refresh all rows in a burst once every 64ms.
- Historically, timing of DRAM control signals used to be asynchronous, and management of refreshing was performed by control circuits external to the DRAM chips. More recently, manufacturers changed to synchronous DRAMs (SDRAMs) that use registers on inputs to sample address, data and control signals on clock edges.
- Since applications with very high data transfer rate requirements may be limited by the relatively slow access times of DRAMs, manufacturers have more recently incorporated further features to improve performance. These include the ability to access a burst of data from successive locations without having to provide the address for each, other than the first, and the ability to transfer on both rising and falling clock edges (double-data rate, or DDR, and its successors, DDR2 and DDR3). These features are mainly motivated by the need to provide high-speed bursts of data in computer systems,



5) READ - ONLY MEMORIES

- A *read-only memory*, or ROM, can only read the stored data. This is useful in cases where the data is constant, so there is no need to update it. The data is incorporated into the ROM during its manufacture, or is programmed into the ROM subsequently.

Combinational ROMs

- A simple ROM is a combinational circuit that maps from an input address to a constant data value. We could specify the ROM contents in tabular form, with a row for each address and an entry showing the data value for that address. ROM circuit structures are generally much denser than arbitrary gate-based circuits, since each ROM cell needs at most one transistor.

Programmable ROMs

- ROM in which the contents are manufactured into the memory is suitable for applications in which we are sure that the contents will not need to change over the lifetime of the product.
- In some applications, it is preferable to revise the ROM contents from time to time, or to use a form of ROM with lower costs for low-volume production. A *programmable ROM* (PROM) meets these requirements. It is manufactured as a separately packaged chip with no content stored in its memory cells. The memory contents are programmed into the cells after manufacture, either using a special programming device before the chip is assembled into a system, or using special programming circuits when the chip is in the final system.
- There are a number of forms of PROMs. Early PROMs used fusible links to program the memory cells. Once a link was fused, it could not be replaced, so programming could only be done once. These devices are now largely obsolete. They were replaced by PROMs that could be erased, either with ultraviolet light (so called EPROMs), or electrically using a higher-than-normal power-supply voltage (so-called electrically erasable PROMs, or EEPROMs).

Flash Memories

- Flash memory, is a form of electrically erasable programmable ROM. It is organized so that blocks of storage can be erased at once, followed by programming of individual memory locations. A flash memory typically allows only a limited number of erasure and programming operations, typically hundreds of thousands, before the device “wears out.” Thus, flash memories are not a suitable replacement for RAMs.
- There are two kinds of flash memories, NOR and NAND flash, referring to the organization of the transistors that make up the memory cells. Both kinds are organized as blocks (commonly of 16, 64, 128, or 256 Kbytes) that must be erased in whole before being written.
- In a NOR flash memory, locations can be written (once per erasure) and read (an arbitrary number of times) in random order.
- In a NAND flash memory, locations are written and read one page at a time, a page being typically 2 Kbytes. Read access to a given location would require reading the page containing the location, followed by selection of the required data, taking several microseconds. If all of the locations in a page are required, sequential reading is much faster, comparable in time to SRAM. Erasing a block and writing a page of data are significantly slower than SRAM access times.
 - The advantage of NAND flash memory is that the density of storage cells is greater than that of NOR flash. Thus, NAND flash chips are better suited to applications in which large amounts of data must be stored.
 - One of the largest applications of NAND flash memories is in memory cards for consumer devices such as digital cameras. They are also used in USB memory sticks for general purpose computers.

Develop a Verilog model of a dual-port, 4K_ 16-bit flow through SSRAM. One port allows data to be written and read, while the other port only allows data to be read.

In the following module definition, the clk input is common to both memory ports. The inputs and outputs with names ending in "1" are the connections for the read/write memory port, and the inputs and outputs with names ending in "2" are the connection for the read-only memory port.

```

module dual_port_SSRAM ( output reg [15:0] d_out1,
                        input      [15:0] d_in1,
                        input      [11:0] a1,
                        input      en1, wr1,
                        output reg [15:0] d_out2,
                        input      [11:0] a2,
                        input      en2,
                        input      clk );

reg [15:0] SSRAM [0:4095]; // Declare 4K x 16memory
always @(posedge clk) // read/write port
    if (en1)
        if (wr1)
            begin
                data_RAM[a1] <= d_in1; // write operation
                d_out1 <= d_in1; // flow through SSRAM
            end
        else
            d_out1 <= SSRAM[a1]; // read operation

always @(posedge clk) // read-only port
    if (en2)
        d_out2 <= SSRAM[a2]; // read operation
endmodule

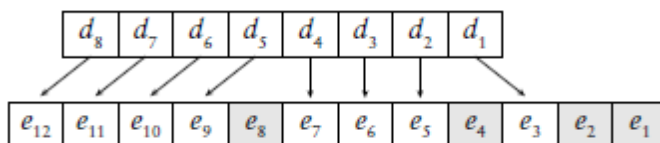
```

ERROR DETECTION AND CORRECTION

- Bit errors can occur in memories from a number of causes. Some errors are *transient*, also called *soft errors*, and involve a bit flip in a memory cell without a permanent effect on the cell's capacity to store data.
- In DRAMs, **soft errors** are typically caused by high-energy neutrons generated by collision of cosmic rays with atoms in the earth's atmosphere. The neutrons collide with silicon atoms in the DRAM chip, leaving a stream of charge that can disrupt the storage or reading of charge in a DRAM cell.
- The frequency of soft-error occurrence, the *soft-error rate*, depends on the way in which DRAMs are manufactured and the location in which they operate. Hence, soft-error rates are highly variable between systems. Soft errors can also occur in DRAMs and other memories from electrical interference, the effects of poor physical circuit design and other causes.
- Errors that persist in a memory circuit are called **hard errors**. They can result from manufacturing defects or from electrical "wear" after prolonged use. A memory cell or chip affected by a hard error is no longer able to store data. A read operation would always yield a 0 or a 1 value, regardless of the bit value that was previously written.
- Memories are more susceptible to bit errors than logic circuits using flip-flops and registers for storage, due to the storage density and the longevity of data in memories, hence different forms of error detection should be employed.

Error detection:

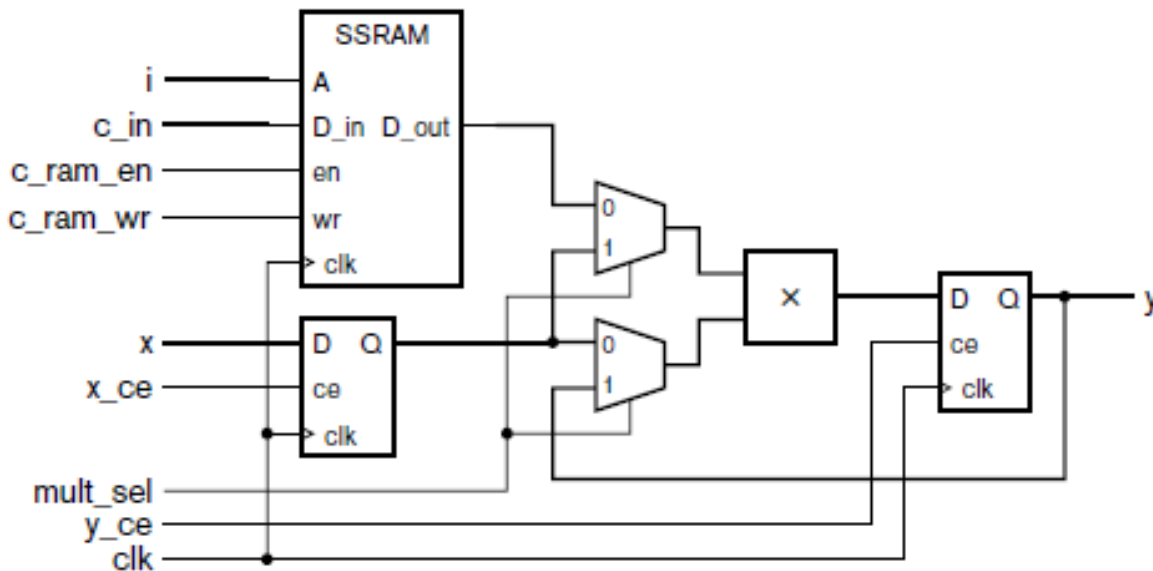
- A common approach is to use parity. In the case of memories, use of parity involves adding an extra bit cell to each memory location. When we write to a location, we compute the parity bit and store it in the extra cell. When we read a location, we check that the data, together with the parity bit, have the correct parity.
- The problem with using parity to check for errors, is that it only allows us to detect a single bit flip in a stored code word. It does not allow us to identify which bit flipped, nor does it allow us to detect an even number of bit flips. If we could identify the particular bit that flipped, we could correct the error by flipping the bit back to its original value, and then continue operating as normal.
- In order to be able to identify which bit flipped, we need to consider the invalid code words that result from flipping each bit of each valid code word. Provided all of those invalid code words are distinct, we can use the value of the invalid code word to identify the flipped bit.
- One scheme for doing this is to use a form of *error correcting code* (ECC) known as a *Hamming code*. We will start with a single-error correcting Hamming code, that is, a code that allows us to correct a single bit flip within a code word. If our code word has N bits, we need $\log_2 N + 1$ additional check bits for the ECC. For example, if we have 8 data bits, we need 4 check bits, giving a total of 12 bits. The check bits are computed from the values of the data bits during a write operation, and the entire ECC word is written to the memory location.
- The ECC bits whose indices are powers of 2 are used as check bits, and the remaining ECC bits are the data bits, in order, as shown in Figure.



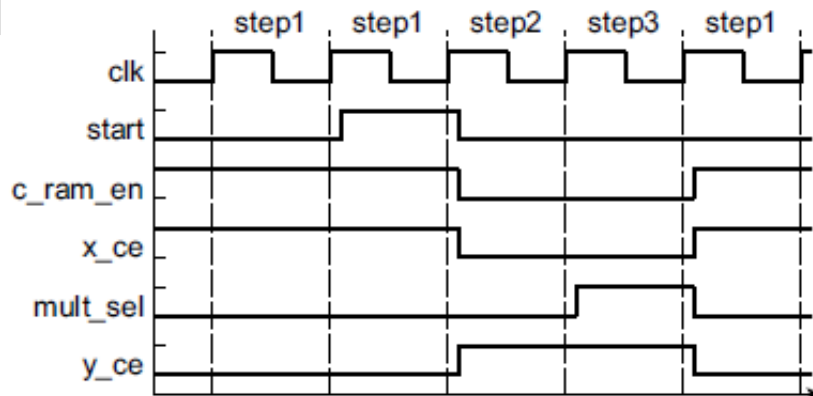
REFER CLASS NOTES FOR PROBLEMS ON MEMORY EXPANDING AND ERROR DETECTION & CORRECTION.

Problem : Design a circuit that computes the function $y = ci * x^2$, where x is a binary-coded input value and ci is a coefficient stored in a flow-through SSRAM. x , ci and y are all signed fixed-point values with 8 prebinary-point and 12 post-binary-point bits. The index i is also an input to the circuit, encoded as a 12-bit unsigned integer. Values for x and i arrive at the input during the cycle when a control input, $start$, is 1. The circuit should minimize area by using a single multiplier to multiply ci by x and then by x again.

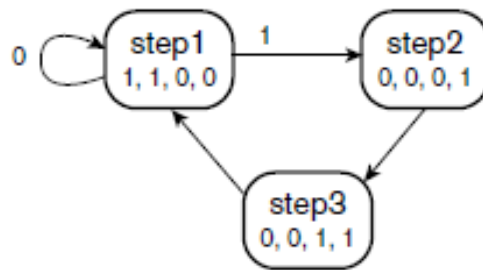
Solution: A datapath for the circuit is shown in Figure. The 4K x 20-bit flow-through SSRAM stores the coefficients. A computation starts with the index value, i , being stored in the SSRAM address register, and the data input, x , being stored in the register shown below the SSRAM. On the second clock cycle, the SSRAM performs a read operation. The coefficient read from the SSRAM and the stored x value are multiplied, and the result is stored in the output register. On the third cycle, the multiplexer select inputs are changed so that the value in the output register is further multiplied by the stored x value, with the result again being stored in the output register.



For the control section, we need to develop a finite state machine that sequences the control signals. It is helpful to draw a timing diagram showing progress of the computation in the datapath and when each of the control signals needs to be activated. The timing diagram is shown in Figure below, and includes state names for each clock cycle.



An FSM transition diagram for the control section is shown in Figure below.



The FSM is a Moore machine, with the outputs shown in each state in the order `c_ram_en`, `x_ce`, `mult_sel` and `y_ce`. In the `step1` state, we maintain `c_ram_en` and `x_ce` at 1 in order to capture input values. When `start` changes to 1, we change `c_ram_en` and `x_ce` to 0 and transition to the `step2` state to start computation. The `y_ce` control signal is set to 1 to allow the product of the coefficient read from the SSRAM and the `x` value to be stored in the `y` output register. In the next cycle, the FSM transitions to the `step3` state, changing the `mult_sel` control signal to multiply the intermediate result by the `x` value again and storing the final result in the `y` output register. The FSM then transitions back to the `step1` state on the next cycle.

Verilog Code:

```

module scaled_square ( output reg signed [7:-12] y,
                      input signed [7:-12] c_in, x,
                      input [11:0] i,
                      input start,
                      input clk, reset );

wire c_ram_wr;
reg c_ram_en, x_ce, mult_sel, y_ce;
reg signed [7:-12] c_out, x_out;
reg signed [7:-12] c_RAM [0:4095];
reg signed [7:-12] operand1, operand2;
parameter [1:0] step1 = 2'b00, step2 = 2'b01, step3 = 2'b10;
reg [1:0] current_state, next_state;

assign c_ram_wr = 1'b0;

always @(posedge clk) // c RAM – flow through
if (c_ram_en)
if (c_ram_wr)
begin
c_RAM[i] <= c_in;
c_out <= c_in;
end
else
c_out <= c_RAM[i];
always @(posedge clk) // y register
if (y_ce)
begin
if (!mult_sel) begin
operand1 = c_out;
operand2 = x_out;
end
end
  
```

```
else
    begin
        operand1 = x_out;
        operand2 = y;
    end
y <= operand1 * operand2;
end

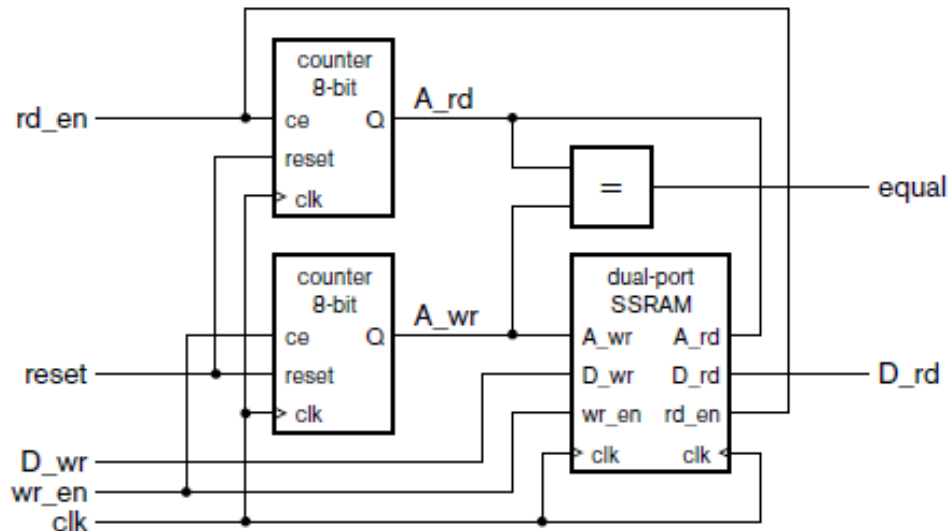
always @(posedge clk) // State register
if (reset)
current_state <= step1;
else
current_state <= next_state;

always @* // Next-state logic
case (current_state)
step 1 : if (!rdy)    next_state = step1;
        else        next_state = step2;
step 2 : next_state = step3;
step 3 : next_state = step1;
endcase

always @* begin // Output logic
begin
c_ram_en = 1'b0; x_ce = 1'b0; mult_set = 1'b0; y_ce = 1'b0;
case (current_state)
step 1 : begin
        c_ram_en = 1'b1;
        x_ce = 1'b1;
        end
step 2 : y_ce = 1'b1;
step 3 : begin
        mult_set = 1'b1;
        y_ce = 1'b1;
        end
endcase
end
endmodule
```

Problem: Design a FIFO to store up to 256 data items of 16 bits each, using a 256 x 16-bit dual-port SSRAM for the data storage. The FIFO should provide status outputs, , to indicate when the FIFO is empty and full. Assume that the FIFO will not be read when it is empty, nor be written to when it is full, and that the write and read ports share a common clock.

Solution: The datapath for the FIFO, shown in Figure, uses 8-bit counters for the write and read addresses. The write address refers to the next free location in the memory, provided the FIFO is not full. The read address refers to the next location to be read, provided the FIFO is not empty.



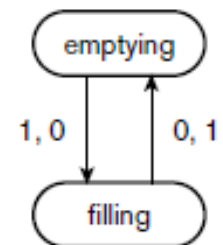
Both counters are cleared to 0 when the reset signal is active. The FIFO being empty is indicated by the two address counters having the same value. The FIFO is full when the write counter wraps around and catches up with the read counter, in which case the counters have same value again.

A write operation without a concurrent read means the FIFO is filling. If the write address becomes equal to the read address as a consequence of the FIFO filling, the FIFO is full.

A read operation without a concurrent write means the FIFO is emptying. If the write address becomes equal to the read address as a consequence of the FIFO emptying, the FIFO is empty.

If a write and a read operation occur concurrently, the amount of data in the FIFO remains unchanged, so the filling or emptying state remains unchanged.

Control section FSM, as shown in Figure in which the transitions are labeled with the values of the wr_en and rd_en control signals, respectively. The FSM starts in the emptying state. The empty status output is 1 if the current state is emptying and the equal signal is 1, and the full status output is 1 if the current state is filling and the equal signal is 1. Note that this control sequence relies on the assumption of a common clock between the two FIFO ports, since the FSM must have a single clock to operate.



Verilog Code:

```
module FIFO ( input clk, reset,
             input wr_en, rd_en,
             input [15:0] D.rd,
             output reg[15:0] D.rd,
             output empty,full);
```

```
reg [15:0] FIFO_RAM [0:255]; // size of total locations and memory locations.
```



```

reg [7:0] A_rd, A_r;
wire equal;
parameter emptying = 1'b0, filling = 1'b1;
reg current_state, next_state;

always@ (posedge clk) //Read counter
if (reset) A_rd<=0;
else if (rd_en) A_rd<= A_rd+1;

always@(posedge clk) //write counter
if(reset) A_wr<=0;
else if(wr_en) A_wr<=A_wr+1;

assign equal = A_rd == A_wr;
always@(posedge clk) //write port
if (wr_en) FIFO_RAM [A_wr] <=D_wr;

always @(posedge clk) // read port
if (rd_en) D_rd <= FIFO_RAM [A_rd];

always @(posedge clk)
if (reset) current_state <=emptying;
else current_state <=next_state;

always @(posedge clk)
case (current_state)
emptying : if (wr_en & ~ rd_en) next_state <= filling;
else next_state <= emptying;

filling : if(~ wr_en & rd_en) next_state <= emptying;
else next_state <= filling;
end case

assign empty = (current_state == emptying) & equal;
assign full = (current_state == filling) & equal;
endmodule

```

Module 3: Implementation fabrics

SYLLABUS: Integrated Circuits, Programmable Logic Devices, Packaging and Circuit boards, interconnection and Signal integrity

INTEGRATED CIRCUITS

Early digital systems were constructed using discrete switching components, such as relays, vacuum tubes, and transistors. However, the manufacture of a complete circuit on the surface of a silicon wafer resulted in cost reduction. Invention of the integrated circuit is credited to Jack Kilby at Texas Instruments in 1958. The techniques were refined by several developers, and the market for ICs grew rapidly during the 1960s.

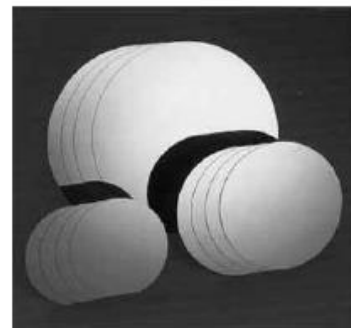
The history of development of digital IC technology is to be known for two reasons.

- First, when we deal with *legacy systems*, that is, systems designed some time ago but that are still in operation and need maintenance. Where obsolete parts are unavailable, we need to design replacement circuits to keep the system operating. Hence, we need to understand the operation of legacy components and the constraints under which they operate.
- Second, circuit technology is continually evolving. It's not sufficient just to learn how to design using current components, since they will be obsolete at some stage in the future. Instead, we need to understand technology evolution and trends, so that we can "future proof" our designs. Understanding history is important for projecting into the future.

INTEGRATED CIRCUIT MANUFACTURE

- Implementation fabrics for digital systems are based on integrated circuits (ICs), which are manufactured on the surface of a wafer of pure crystalline silicon using a sequence of photographic and chemical process steps.
- A number of identical rectangular ICs are manufactured together, and then broken apart for individual packaging.
- In preparation for chip manufacture, a cylindrical ingot of silicon is formed (Figure) and then sawn into wafers less than a millimeter thick and finely polished (Figure).

FIGURE 6.1 An ingot of crystalline silicon (left), and sawn wafers (right).



- Early wafers were 50mm in diameter, but, since then, improvements in manufacturing processes have yielded successively larger wafer sizes. Now, chips can be manufactured on 300mm diameter wafers. This allows more chips to be manufactured at once, and reduces the waste at the edges.
- The process of manufacturing a circuit on the wafer surface involves a number of steps that change the properties of certain areas of the surface silicon, or add a surface layer of some material in certain areas. There are several kinds of processing steps that can be applied to

selected areas of the wafer, including Ion implantation: exposing the surface to plasma of impurity ions that diffuse into the silicon, thus altering its electrical properties in controlled ways.

- **Etching:** Chemically eroding (removing) an underlying film of material that has been deposited onto the surface. Films include insulating materials, such as silicon dioxide; semiconducting materials, such as polycrystalline silicon (also known as polysilicon); and conducting materials, such as aluminum and copper.
- The key to selecting which areas are affected is *photolithography*, which means using a photographic process to draw on the surface (see Figure showing selective etching of a film).

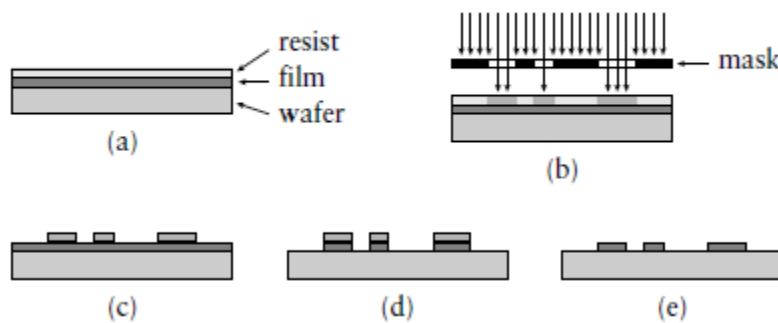


FIGURE 6.2 Steps in photolithographic etching: (a) the wafer and film coated with resist; (b) exposure through a photo-mask; (c) the resist developed; (d) etching of the underlying film; (e) the remaining resist stripped.

- The surface is coated with a thin layer of *photoresist*, a chemical whose resistance to chemical reaction is changed by exposure to light. The surface is then exposed to light through a mask that has opaque and transparent areas in the pattern of features to be drawn. The resist is then developed, dissolving either the exposed areas or the unexposed areas, depending on the kind of resist. The areas that are now uncoated can be processed, and then the remaining resist is stripped off.
- Once the circuits on a wafer have been manufactured, they must be tested to determine which ones work and which fail due to defects. Small defects can be introduced into an individual IC by stray particles obscuring light during photolithography, by impurities occurring in chemical process steps, or by particles impinging on wafers during handling in the manufacturing process.
- IC foundries are meticulous in their cleanliness, using chemicals of high purity and operating in special clean rooms. Nevertheless, stray particles and impurities cannot be completely avoided. A defect can prevent an IC on a wafer from working.
- The *yield* is the proportion of manufactured ICs that work. Since a whole wafer-lot of ICs is manufactured together, the cost of the discarded defective ICs must be amortized over those that work. Larger ICs have an increased chance of being defective, so it is important for designers to constrain IC area to reduce cost. After testing the ICs on a wafer, the wafer is broken into individual chips, which are then packaged.
- Most of the improvement in IC technology is attributable to advances in photolithography, including use of higher-resolution masks and shorter wavelengths of light. Reducing the feature size has a number of benefits. It results in smaller chips for a given function, thus reducing cost. It allows more circuitry to be placed on a chip of a given size, thus increasing functionality. It also reduces circuit delays, allowing higher operating speed.

SSI AND MSI LOGIC FAMILIES

- While many early ICs were developed for specific applications, in 1961 Texas Instruments introduced a family of logic components that designers could use as building blocks for larger circuits.

- Three years later, they introduced the 5400 and 7400 families of TTL (transistor-transistor logic) ICs that became the basis of logic design for many years. The 5400 family components were manufactured for high-reliability military applications, requiring operation over large temperature ranges,
- The 7400 family components were for commercial and industrial applications. The components in the 7400 family are numbered according to the logic functions they provide. For example, a 7400 component provides four NAND gates, these components integrate a relatively small number of circuit elements, and they are referred to as *small-scale integrated (SSI)* components.
- As manufacturing techniques improved, larger circuits could be integrated, leading to *medium-scale integrated (MSI)* components. Examples include the 7490 4-bit counter, and the 7494 4-bit shift register.
- The 7400 family, manufacturers developed alternative versions of the components with different internal circuitry and correspondingly different electrical characteristics. One variation reduced the power consumed by components, at the cost of reduced switching speed. Components in this family are identified by inclusion of the letter “L” in the part number, for example, 74L00 and 74L74. Another variation used Schottky diodes within the internal circuits to reduce switching delays, albeit at the expense of increased power consumption. These components include the letter “S” in the part number, for example, 74S00 and 74S74. One of the most popular variations, the 74LS00 family, combined the lower-power circuits with Schottky diodes to yield a good compromise between power and speed.
- Later variations included the 74F00 “fast” family components and the 74ALS00 “advanced low-power Schottky” family.
- One of the problems with TTL circuits is that they use bipolar transistors, which have relatively high power consumption even when not switching. The CMOS, which uses field-effect transistors, was originally developed around the same time as TTL. One of the earliest CMOS logic families was the 4000 family, which provided SSI and MSI functions, but with much lower power consumption. They could also operate over a much larger power supply range (3V-15V), compared to TTL’s nominal 5V, but were much slower and had logic levels that were incompatible with TTL components.
- Later, in the 1980s, some manufacturers introduced a new family of CMOS logic components, the 74HC00 family, that were compatible with TTL components. They provided the same functionality, but with lower power consumption and comparable speed. Subsequent variations, such as the 74AHC00 family, offered improved speed and electrical characteristics.
- One important characteristic of CMOS circuits is that the power consumption and speed are dependent on the power-supply voltage. By reducing the voltage, as well as by making the internal circuit features smaller, speed is increased and power consumption is reduced.
- As a result of these evolutionary steps, we now have numerous logic families from different manufacturers, each family has different trade-offs in power consumption, speed and logic-level thresholds.
- During the 1970s, IC technology developed to the level of *large-scale integration (LSI)*, at which it became feasible to provide a small computer, a *microprocessor*, on a single IC. Embedded systems using microprocessors became more cost-effective in many applications than systems constructed from SSI and MSI components.

APPLICATION- SPECIFIC INTEGRATED CIRCUITS (ASICs)

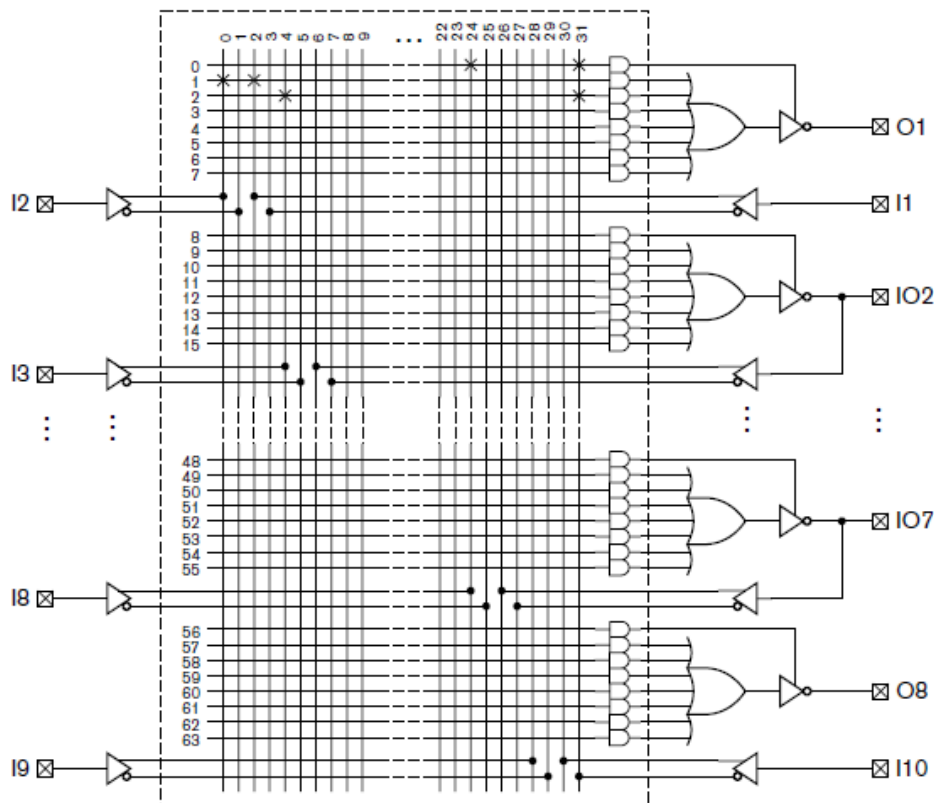
- The development of IC technology beyond the LSI level led to *very large scale integrated (VLSI)* circuits.
- The term *application-specific integrated circuit*, or *ASIC*, refers to an IC manufactured for a particular application. It is designed to meet a particular set of requirements, and so contains circuits customized for those requirements.
- It may be designed for a particular end product provided by one manufacturer, for example, a portable music player, a toy, an automobile, a piece of military equipment, or an industrial machine.
- Alternatively, it may be designed for use in a range of products provided by manufacturers in a particular market segment. These kinds of ASIC are sometimes called *application-specific standard products*, or *ASSPs*, since they are treated as a standard part within the market segment, but are not of use outside that segment. Examples include ICs for cell phones, which are used by a number of competing cell-phone manufacturers, but which are not of use in, say, and automobile control circuits.
- One of the main reasons an ASIC is developed for a product is that, being customized for that application, it has lower cost per IC than a programmable component such as an FPGA. However, in order to achieve that level of customization, we need to invest much more design and verification effort. We must amortize the *non-recurring engineering (NRE)* cost over the entire product units sold.
- Hence, it only makes sense to use an ASIC if our product sales volume is sufficiently large. The amortized NRE cost per unit should be less than the cost difference between an ASIC and a programmable part.
- There are two main design and manufacturing techniques for ASICs, differing in the degree of customization for the application.
 - ✓ First, *fully custom* integrated circuits involve detailed design of all of the transistors and connections in an ASIC. This allows the most effective use of the hardware resources on an IC and yields higher performance, but has high NRE cost and requires advanced VLSI design expertise within the design team. As a consequence, fully custom ASICs are usually only designed for high-volume products, such as CPUs and ICs for consumer appliances.
 - ✓ Second, *standard cell* ASICs involves selection of basic cells, such as gates and flip-flops, from a library to form the circuit. The cells have been previously designed by an IC manufacturer or an ASIC vendor, and are used by the synthesis tool during the design process to implement the design. The value of this approach is that the NRE cost for each ASIC design is significantly reduced, since the cost of designing the cell library is amortized over a number of ASIC designs. The compromise is that the ASIC may not be as dense or have the performance of a fully custom ASIC.

PROGRAMMABLE LOGIC DEVICES

The components in SSI and MSI families and ASICs all have fixed functions, determined by the logic circuit for each component. *Programmable logic devices* (PLDs), on the other hand, can be programmed after manufacture to have different functions. There are various types of PLDs namely:

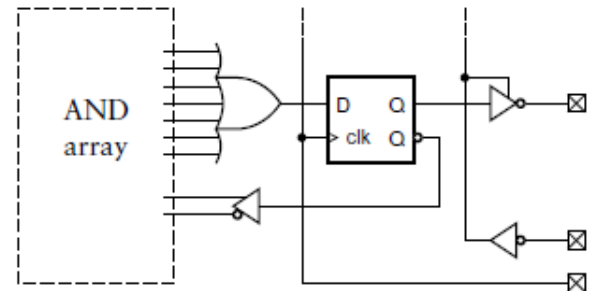
1) PROGRAMMABLE ARRAY LOGIC

- One of the first successful families of PLDs was introduced in the late 1970s by Monolithic Memories, Inc., and called *programmable array logic* (PAL) components.
- A simple representative component in the family is the PAL16L8, whose circuit is shown in Figure. The component has 10 pins that are inputs, 2 pins that are outputs, and 6 pins that are both inputs and outputs. This gives a total of 16 inputs and 8 outputs (hence the name “16L8”). The symbol at each input in Figure represents a gate that is a combination of a buffer and an inverter.
- Thus, the vertical signals carry all of the input signals and their negations. The area in the dashed box is the *programmable AND array* of the PAL. Each horizontal signal in the array represents a p-term of the inputs, suggested by the AND-gate symbol at the end of the line. In the unprogrammed state, there is a wire called a *fusible link*, or *fuse*, at each intersection of a vertical and horizontal signal wire, connecting those signal wires.
- The PAL component can be programmed by blowing some of the fuses to break their connections, and leaving other fuses intact. This is done by a special programming instrument before the component is inserted into the final system.
- On the diagram of Figure an ‘X’ is drawn at the intersection of a vertical and a horizontal signal to represent an intact fuse. An intersection without an X means that the intersecting signals are not connected. So, the horizontal signal numbered 0 has connections to the vertical signals numbered 24 and 31, which are the signals I8 and I10.



- Some of the p-terms are connected to the enable control signals for the inverting tristate output drivers. Others are connected to the 7-input OR gates. So, for each output, we can form the AND-OR-INVERT function of inputs, with up to 7 p-terms involved. In the circuit shown in Figure, output $O1$ implements the function $I1.I2 + I3.\overline{I10}$ with the output enabled by the condition $I8.\overline{I10}$.

- A PAL component such as the PAL16L8 can be programmed to implement a variety of combinational functions. Other PAL components also include registers, allowing us to implement simple sequential circuits. As an example, the output circuit of a PAL16R8 component is shown in Figure. The feedback from the register output to the programmable AND array is useful for implementation of FSMs.



- A synthesis tool is used to transform the equations into a gate-level circuit, which can be verified using the same test bench that we used to verify the Boolean-equation model.
- Finally, a physical design CAD tool is used to transform the gate-level model into a *fuse map*, that is, a file used by the programming instrument to determine which fuses to blow.

- Generic Array Logic (GAL) components provide **output logic macro cells (OLMCs)** that replace the combinations of OR gates, registers and tristate drivers in PAL output circuits.

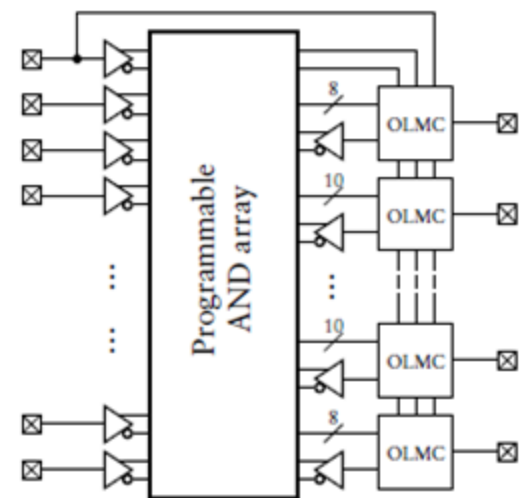
- Each OLMC includes circuit elements together with programmable multiplexers.

- The internal circuit organization of the GAL22V10 component is shown in the figure.

- The OLMC has p-term inputs from an AND array with the same organization as that of a PAL component. The output of the OR gate connects to a D flip-flop that has clock, asynchronous reset and synchronous preset signals in common with other OLMCs in the component.

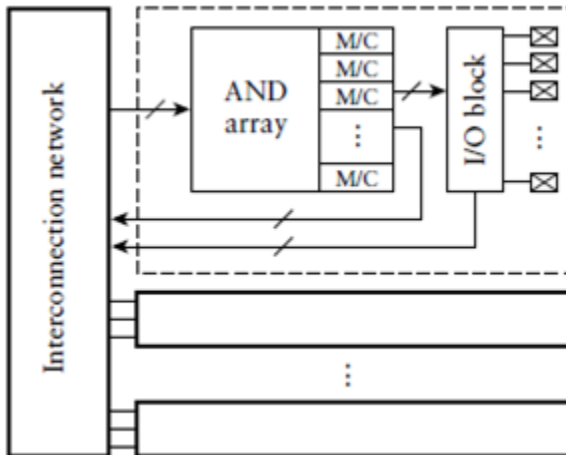
- The select inputs of the multiplexers are set by programming the component. The four-input multiplexer allows selection of registered or combinational output, either inverting or non-inverting.

- The two-input multiplexer allows either registered or combinational feedback, or, if the output driver is in the high-impedance state, direct input from the component pin.



COMPLEX PLDS

- A further evolution of PLDs, tracking advances in integrated circuit technology, led to the development of *complex programmable logic devices (CPLDs)*. A CPLD incorporates multiple PAL structures, all interconnected by a programmable network of wires, as shown in Figure.

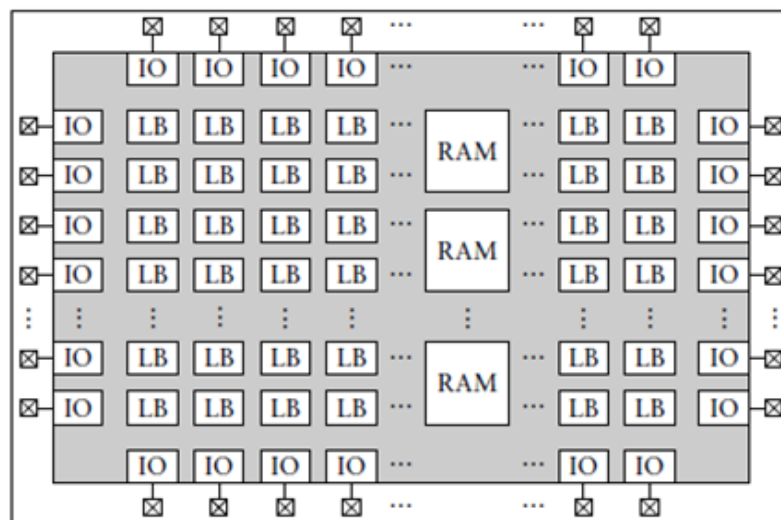


- Each of the PAL structures consists of an AND array and a number of embedded macrocells (M/Cs in the figure).
- The macrocells contain OR gates, multiplexers and flip-flops, allowing choice among combinational or registered connections to other elements within the component, with or without logical negation,
- The external pins are connected to an I/O block, which allows selection among macrocell outputs to drive each pin. The network interconnecting the PAL structures allows each PAL to use feedback connections from other PALs as well as inputs from external pins.
- CPLDs use SRAM cells to store configuration bits that control connections in the AND-OR arrays and the select inputs of multiplexers. Configuration data is stored in nonvolatile flash RAM within the CPLD chip, and is transferred into the SRAM when power is applied.
- Separate pins are provided on the chip for writing to the flash RAM, even while the chip is connected in the final system. Thus, designs using CPLDs can be upgraded by reprogramming the configuration information.
- Manufacturers provide a range of CPLDs, varying in the number of internal PAL structures and input/output pins. A large CPLD may contain tens of thousands of gates and hundreds of flip-flops, allowing for implementation of quite complex circuits.
- CAD tools are used to synthesize a design from an HDL model and to map the design to the resources provided by a CPLD.

FIELD-PROGRAMMABLE GATE ARRAYS(FPGAs)

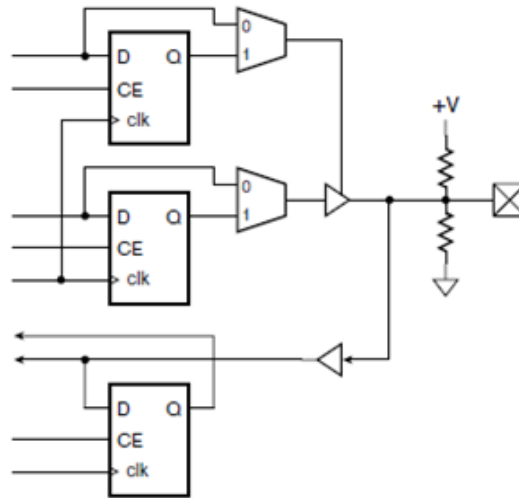
- For large designs, mapping the circuit onto CPLD resources becomes very difficult and results in inefficient use of the resources provided by the chip.
- Therefore, manufacturers designed another programmable circuit structure, based on smaller programmable cells to implement logic and storage functions, combined with an interconnection network whose connections could be programmed.
- They named such structures *field-programmable gate arrays* (FPGAs), since they could be thought of as arrays of gates whose interconnection could be programmed **“in the field,”** away from the factory where the chips were made.
- FPGAs include
 - ✓ An array of logic blocks that can be programmed to implement simple combinational or sequential logic functions;
 - ✓ Input/output (I/O) blocks that can be programmed to be registered or nonregistered, as well as implementing various specifications for voltage levels, loading and timing;
 - ✓ Embedded RAM blocks to store information.
 - ✓ A programmable interconnection network.
- The more recent FPGAs also include special circuits for clock generation and distribution. The specific organization, as well as the names used for the blocks, varies between manufacturers and FPGA families.

FIGURE The internal organization of an FPGA consisting of logic blocks (LB), input/output blocks (IO), embedded RAM blocks (RAM) and programmable interconnections (shown in gray).



- In many FPGA components, the basic elements within logic blocks are small 1-bit-wide asynchronous RAMs called *lookup tables* (LUTs). The LUT address inputs are connected to the inputs of the logic block.
- The I/O block of an FPGA is typically organized as shown in Figure; the select inputs of the multiplexers are programmed to control whether the output is registered or combinational.
 - The top flip-flop and multiplexer control the high-impedance state of the tristate driver that drives the pin as an output.
 - The middle flip-flop and multiplexer drive the output value.
 - The pull-up and pull-down resistors are programmable, allowing them to be connected and their resistance to be selected. The reason for making all of these characteristics programmable is to allow the FPGA to be used in a wide range of systems that use different Signaling standards between chips, and to accommodate the different drivers and loads to which different pins of an FPGA may be connected.

FIGURE Typical organization of an FPGA I/O block.



- The RAM blocks in an FPGA provide for storage of information to be processed by the FPGA circuitry. They can be used to store such chunks between processing steps. Also, when an embedded processor is implemented within an FPGA, RAM blocks provide a place to store the processor's instructions and the data upon which it operates.
- Typical modern FPGAs provide synchronous static RAM (SSRAM) blocks that can be programmed to be flow-through or pipelined, and that have two access ports that can be programmed to be read-only or read-write. The RAM blocks are each relatively small in capacity, but can be interconnected to form larger memories.
- The connections can be programmed so that a given input or output of a block can be connected (or not) to a wire that passes the block. The interconnections between logic blocks consist of a mix of short and long wires, and possibly wires of intermediate length, depending on the FPGA.
- Short wires connect nearby logic blocks, whereas long wires connect distant logic blocks or connect to a number of logic blocks distributed across the FPGA. It is the job of the place and route software to ensure that parts of the design are implemented in logic blocks in such a way that the interconnection resources can be programmed to "wire up" the design.
- There are two forms of FPGA that differ in the way they are configured. The first form uses RAM cells to store the configuration information. The main advantage of this approach is that an FPGA can be programmed after the chip has been assembled into a system, without the need for any separate handling during manufacture. Furthermore, the system can be upgraded after delivery by storing new configuration information, rather than having to replace chips or other hardware.
- If the configuration is stored using volatile SRAM cells, it needs to be loaded each time power is applied to the system. Hence, the configuration needs to be stored in a separate nonvolatile memory, and additional circuits need to be included in the system to manage loading the configuration.
- The second main form of FPGA uses *antifuses* to configure the device. An antifuse is a conductive connection that is formed during programming, as opposed to being blown.

Platform FPGAs

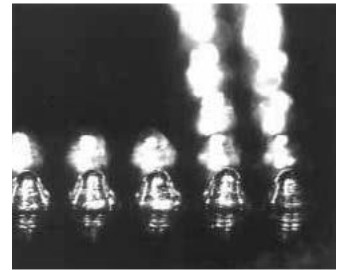
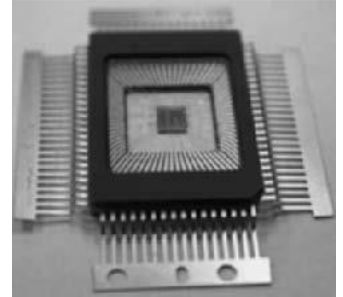
- FPGAs have become denser and faster, it has become feasible to use them for applications requiring significant computational performance, such as audio and video processing and information encryption and decryption.
- In order to improve their usability for these kinds of applications, manufacturers have added specialized circuitry to the larger recent FPGAs, including processor cores, computer network transmitter/receivers and arithmetic circuits.
- Such FPGAs are often called *platform FPGAs*, meaning that the chip serves as a complete platform upon which a complex application can be implemented.
- Embedded software can run on the processor cores with instructions and data stored in block RAMs. The network connections can be used to communicate with other computers and devices, and the programmable logic and specialized arithmetic circuits can be used for high-performance data transformations required by the application.
- A minimal amount of circuitry is required externally to the FPGA, thus reducing the overall cost of the system.

Structured ASICs

- Recently, manufacturers have developed a new kind of IC, called *structured ASICs*, that is midway between PLDs and standard-cell ASICs.
- A structured ASIC is an array of basic logic elements, like an FPGA.
- It is not programmable and does not have the programmable interconnect. The logic elements are very simple, and consist of a collection of transistors that can be formed into logic gates and flip-flops.
- An FPGA is customized by loading a configuration program, a structured ASIC is customized by designing the top one or more layers of metal interconnection for the chip.
- Since the underlying logic elements and lower interconnection layers are fixed, the design effort and NRE cost for customization are much lower than those for a standard-cell ASIC. Further, since the structured ASIC is not programmable, just customized by a design and manufacturing process, the performance is potentially very close to that of a standard cell ASIC.

PACKAGING AND CIRCUIT BOARDS

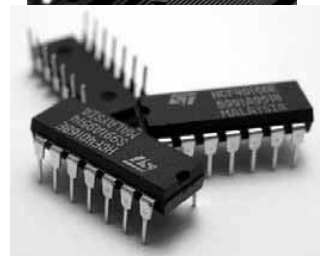
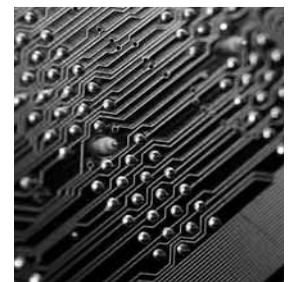
- A single bare IC does not form a complete digital system. It needs to be packaged so that it can be connected to other ICs and components, including input and output displays for interacting with a user and connectors for cables for interacting with other systems.
- An IC is bonded into a package that serves several purposes. It protects the IC from moisture and airborne contaminants, it provides electrical connections, and it removes heat.
- There are numerous different kinds of IC package, each with different physical, electrical and thermal properties.
- Within a package, the IC is fixed to the bottom of a cavity. Fine gold wires are connected from bonding pads on the edge of the IC to points on the package's lead frame (see Figure), which is the metal framework leading to the external package pins. The cavity is then sealed to protect the IC and the wires.
- As IC technology has developed, the maximum number of pins has increased and also the operating speeds. For a high pin-count, high performance IC, using bond wires introduces mechanical problems and delays and degrades signals.
- Recent packages for these ICs have adopted flip-chip technology. The connection pads on the IC are covered in conductive material forming bumps (Figure). The IC is then flipped over and affixed to the substrate of the package, with the bumps in direct contact with substrate connection points. The connection points lead to the external pins of the package.
- The packaged ICs and other components in a system are assembled together on a *printed circuit board* (PCB). This consists of layers of fiberglass or other insulating material separating layers of metal wiring. The metal is deposited in a layer on a fiberglass sheet, and then etched using a photolithographic process, similar to that used in manufacturing ICs. Several layers are sandwiched together. Small holes are drilled through the layers and coated with metal to form connections, called *vias*, between the layers. The completed PCB contains all the circuit wiring needed for the product.



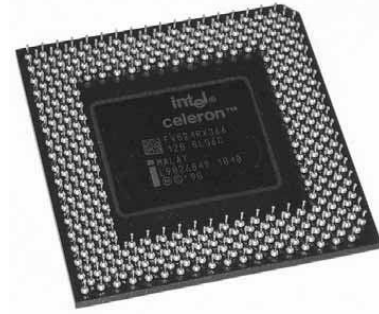
TYPES OF PCB and IC Packages

1) One form of PCB, is a **through-hole PCB** (Figure), it includes metal-coated holes into which IC package pins are inserted. A metal alloy with a low melting point is melted into the holes to form electrical connections between the pins and the PCB wiring.

- Products using this form of manufacture need ICs in *insertion-type* packages, such as:
 - ❖ **Dual in-line packages (DIPs):** have two rows of pins with 0.1-inch spacing. These were among the first IC packages to be introduced, being used for SSI and MSI components, They are limited in the number of pins they can provide, with a 48-pin DIP being about the largest practical size.

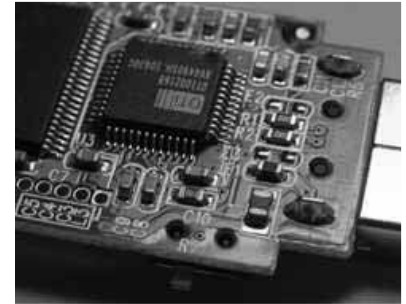


- ❖ **Pin-Grid Array (PGA):** ICs requiring more pins can be packaged in a pin-grid array (PGA) package, having up to 400 or more pins. They are mainly used for ICs such as computer CPUs that are to be mounted in sockets so that they can be removed.
- One of the advantages of through-hole PCBs is that they can be manually assembled, since the component sizes are manageable. This is good for low-volume products, since the cost of setting up a manufacturing run is less than that for automated assembly.

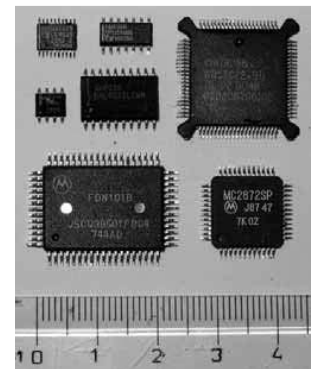


2) The second form of PCB is a *surface-mount* PCB (Figure),

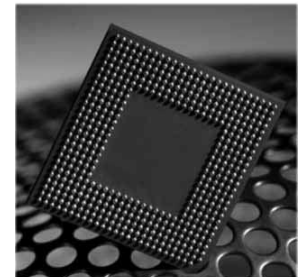
- It is so-called because components are mounted on the surface rather than being inserted in holes. This has the advantage of reduced manufacturing cost (for higher-volume products), finer feature sizes and increased circuit density.
- Surface mounting IC packages have pins or connection points that come into contact with a metal pad on the PCB. Solder paste is applied between each pin and pad and melted, forming the connection.
- There are numerous different surface mounting packages, such as:



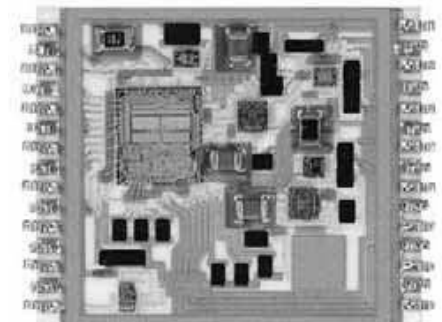
- ❖ **Quad flat-pack (QFP):** In this the packages have pins along all four sides, and are suitable for ICs with up to 200 or so pins. The spacing between pins varies from 1 mm for the packages with fewer pins, down to 0.65mm for the higher pin-count packages. Fine pitch QFP packages allow increased pin count, up to nearly 400 pins, by reducing the pin spacing to 0.4mm. The packages are not suitable for manual handling and assembly.



- ❖ **Ball-Grid Array (BGA):** The package in use for high pin-count ICs is the ball-grid array (BGA) package. Depending on the package size and the pin spacing, BGA packages can accommodate ICs with up to 1800 pins. Higher pin-count BGA packages are also being developed.



- ❖ In recent times, high-density packaging techniques have been developed for use in products where space is constrained. **multichip modules (MCMs)** attach the bare chips to a ceramic substrate (see Figure). Interconnection wires and passive components (resistors and capacitors) are also printed or soldered onto the substrate. The complete module is then encapsulated with external connections made through package pins to a PCB. Even denser packaging can be achieved by building in three dimensions, rather than laying them out on a two-dimensional surface.

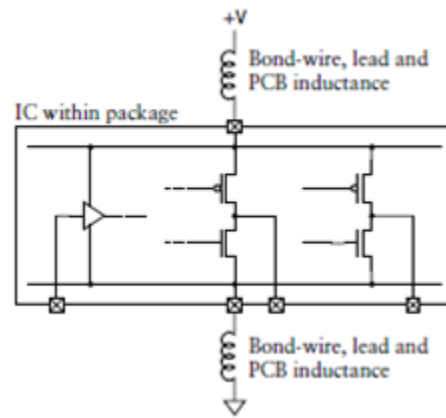


- ❖ For example, *chip stacking* involves placing two or more chips in a vertical stack. Connections can be made between adjacent chips by metal contacts, and between chips and the containing package by bond wires. Several flash memory manufacturers are using these techniques to provide high-capacity storage in very small packages. As demand for high-performance mobile devices increases, we can expect to see continued development of these high density packaging techniques.

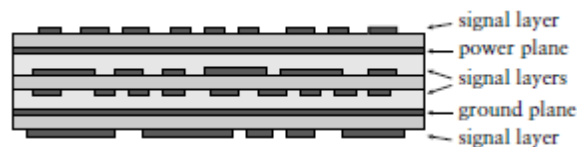
INTERCONNECTION AND SIGNAL INTEGRITY

- A signal change must propagate from the source driver, through the bond wire, package lead frame and pin of the source IC, along the PCB trace, through the pin, lead frame and bond wire of the destination IC, and into the receiver.
- Along this path, there are several influences that can cause distortion of the signal and introduce noise. The term *signal integrity* refers to the degree to which these effects are minimized.
- A change in a signal value causes a change in the current flowing through the PCB trace. This causes a change in the electric and magnetic fields around the trace. Propagation of those fields determines the speed of propagation of the signal change along the trace.
- In PCB materials, the maximum propagation speed is approximately half the speed of light in a vacuum. Since the speed of light is $3 \times 10^8 \text{ms}^{-1}$, we get 150mm per nanosecond as a speed for signal propagation along a PCB trace.
- For low speed designs and small PCBs, this element of total path delay is insignificant. However, for high-speed designs, particularly for signals on critical timing paths, it is significant.
- Two cases in point are the routing of clock signals and parallel bus signals. If a clock signal is routed through paths of different lengths to different ICs, we may introduce clock skew,
- Similarly, if different signals within a parallel bus are routed along paths of different lengths, changes in elements of the bus may not arrive concurrently, and may be incorrectly sampled at the destination's receiver.
- In these cases, it may be necessary to tune the timing of the system by adding to the length of some PCB traces to match propagation delays. CAD tools used for PCB layout offer features to help designers perform such tuning semi automatically.
- A major signal integrity issue in PCB design is *ground bounce*, which arises when one or more output drivers switch logic levels. During switching, both of the transistors in the driver's output stage are momentarily on, and transient current flows from the power supply to ground.
- Ideally, the power supply can source the transient current without distortion. In reality, however, there is inductance in both the power and the ground connections, as shown in Figure.

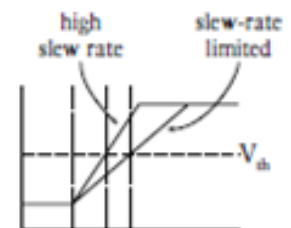
FIGURE Inductance in the bond-wires, package leads and PCB connections for power and ground.



- The inductance causes voltage spikes in the power supply and ground on the IC. This can cause voltage spikes on other output drivers, causing false transitions in the receivers to which they are connected. It can also cause transient shifting of the threshold voltage of receivers on the IC, causing false transitions at those receivers.
- In order to reduce the effects of ground bounce, a number of important measures can be taken:
- First, we can place *bypass capacitors* between power and ground around a PCB. These capacitors hold a reserve of charge that can quickly supply the needs of switching drivers. A common rule of thumb is to place a capacitor close to each IC package. Values of $0.01\mu\text{F}$ to $0.1\mu\text{F}$ are common.
- Second, we can use **separate PCB layers** for the ground and power supply (Figure). This gives a low-inductance path for the power supply current and its ground return. It also has other benefits, mentioned below.



- Third, we can **limit the rate of voltage change (the *slew rate*)** and limit the drive current of the output drivers. These actions limit the rate of change of current, and so limit the inductive effect of the change.
- Components such as modern FPGAs have programmable output drivers that allow selection of slew rate and drive current limits.
- Reducing the slew rate means that a signal takes longer to change from one logic level to the other, as illustrated in Figure.



- Hence, limiting slew rate may increase propagation delay through circuits, consequently requiring a reduction in clock rate. This is a case where a trade-off between speed of operation and noise immunity may be required.
- Finally, we can use differential signaling, to make the system more immune to noise induced by ground bounce.
- Another signal integrity issue for high-slew rate signals is noise due to transmission-line effects. When the time for a transition between logic levels is similar to or shorter than the propagation delay along a signal path, the transition is affected by reflections at the driving and receiving ends of the path and the signal may suffer from partial transitions, overshoot, undershoot and ringing (Figure).

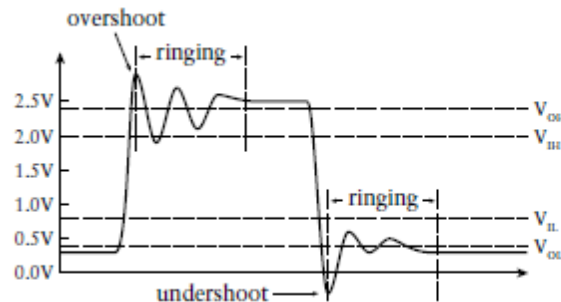


FIGURE 6.27 Overshoot, undershoot and ringing transmission-line effects.

- The main design techniques for managing transmission-line effects involve appropriate layout and proper termination of PCB traces. By running a trace of specific dimensions at a controlled distance between two ground or power planes in the PCB, we create a **stripline** transmission line with a controlled characteristic impedance. Where the transmission line effects are less critical, we can run a trace over just one plane, creating a **microstrip** transmission line.
- Transitions between logic levels on a signal cause electromagnetic fields to propagate around the PCB trace. Some of the field energy is radiated out from the system, and may impinge on other electronic systems, where it induces noise. This form of unwanted coupling is called *electromagnetic interference* (EMI). There are government and other regulations that limit the amount of EMI that a system may emit in various environments, since excessive EMI can be annoying (for example, if it interferes with your TV reception) or a safety hazard (for example, if it interferes with your aircraft navigation).

DIFFERENTIAL SIGNALING

- The use of *differential signaling* is based on reducing system's susceptibility to interference.
- Rather than transmitting a bit of information as a single signal S , we transmit both the positive signal S_P and its negation S_N .
- At the receiving end, we sense the voltage difference between the two signals. If $S_P - S_N$ is a positive voltage, then S is received as the value 1; if $S_P - S_N$ is a negative voltage, then S is received as 0. This arrangement is illustrated in Figure.

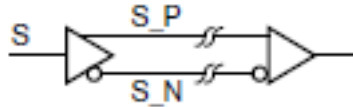
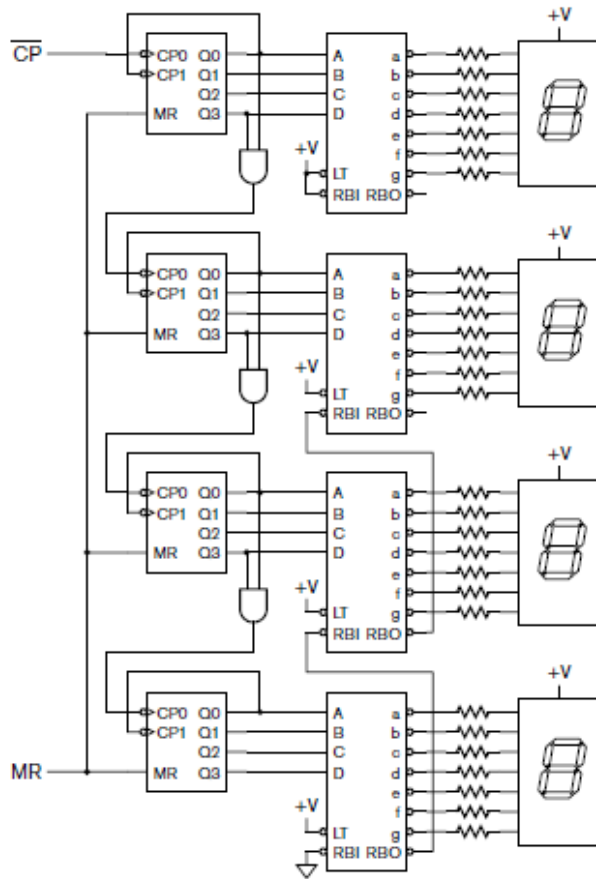


FIGURE 6.28 A differential driver and receiver.

- The assumption behind the differential signaling approach is that noise is induced equally on the wires for both S_P and S_N . Such common-mode noise is cancelled out when we sense the voltage difference. To show this, suppose a noise voltage V_N is induced equally on the two wires.
- At the receiver, we sense the voltage $(S_P + V_N) - (S_P - V_N) = S_P + V_N - S_P + V_N = S_P - S_N$
- For the assumption of common-mode noise induction to hold, differential signals must be routed along parallel paths on a PCB. While this might suggest a problem with crosstalk between the two traces, the fact that the signals are inverses of each other means that they both change at the same time, and crosstalk effects cancel out.
- As well as rejecting common-mode noise, differential signaling also has the advantage that reduced voltage swings are needed for a given noise margin. Even though each of S_P and S_N switches between V_{OL} and V_{OH} , the differential swing at the receiver is between $V_{OL} - V_{OH}$ and $V_{OH} - V_{OL}$, that is, twice the swing of each individual signal.
- Reducing the voltage swing has multiple follow-on effects, including reduced switching current, reduced ground bounce, reduced EMI, and reduced crosstalk with other signals. Thus, use of differential signals can be very beneficial in high-speed designs.

Problem: Use the following components to design a 4-digit decimal counter with a 7-segment LED display: two 74LS390 dual decade counters, four 74LS47 BCD to 7-segment decoders, four 7-segment displays, plus any additional gates required.



The 74LS390 component contains two counters. Internally, the counter consists of a single-bit counter clocked on the falling edge of CP0, and a 3-bit divide-by-five counter clocked on the falling edge of CP1. A decade (divide-by-ten) counter can be formed by using the single bit counter for the least significant bit and connecting the Q0 output externally to the CP1 input. When Q0 changes from 1 to 0, it causes the more significant bits to count up. The MR input to the counter is a master reset input. When 1, it forces the counter outputs to 0000.

We can cascade the 74LS390 decade counters together, using the outputs of each decade to generate a clock for the next decade. The outputs of a given decade changing from 1001 (the binary code for 9) to 0000 should cause the next decade to count up. The only time this occurs is when Q3 and Q0 of the given decade both change to 0, so we can use an AND gate to generate the clock for the next decade,

The ripple-blank input (RBI) and ripple-blank output (RBO) are used to turn off any leading zero digits in the displayed value. When the RBI input to a decoder is low and the BCD value is 0000, all of the segments are turned off and RBO is driven low. We tie the RBI input of the most significant digit low, and chain the RBO of all digits to the RBI of the next digit (except the least significant digit, which we always want to display something).

MODULE 4- I/O- INTERFACING

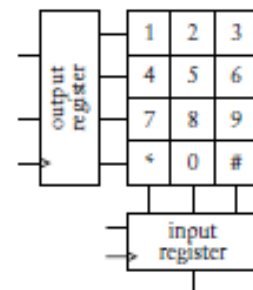
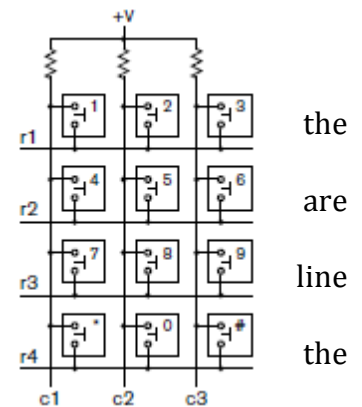
Syllabus: I/O devices, I/O controllers, Parallel Buses, Serial Transmission, I/O Software

1) INPUT DEVICES

Many digital systems include mechanically operated switches of various forms as input devices. These include push-button and toggle switches operated by human users, and microswitches operated by physical movement of mechanical or other objects. The various input devices are as follows:

1.1) Keypads and Keyboards

- Push-button switches are also used in keypads, for example, in phones, security system consoles, automatic teller machines, and other applications.
- The key switches are arranged into a matrix, as shown in Figure, and matrix is scanned for closed contacts.
- When all of the key switches are open, all column lines (c1 through c3) pulled high by the resistors.
- When a key switch is closed, one column line is connected to one row (r1 through r4).
- The matrix is scanned by driving one row line low at a time, leaving rest of the row lines pulled high, and seeing if any of the column lines become low.
- For example, if the 8 key is pressed, c2 is pulled low when r3 is driven low. If more than one key in a given row is pressed at the same time, all of the corresponding column lines will be pulled low when the row line is driven low.
- Thus, we are able to determine the same information about which keys are pressed as we would had we used individual connections for each key switch.
- Thus to drive the row lines low a counter, together with circuitry that stores the count value and the column-line values for access by the embedded software is used.
- However, that would require synchronizing the processor with changes in count value so that the software could read the values at the appropriate times.
- A simpler approach is to provide a register into which the processor can write values to be driven on the row lines and another register for the processor to read the values of the column lines. This is shown in Figure aside.
- The embedded software running on the processor needs to scan the matrix repeatedly. When it detects a key closure, it must check that the same key is still closed some time (say, 10ms) later.
- Similarly, when it detects a key release, it must check that the same key remains released some time later.
- The scan must be repeated sufficiently often to debounce key presses without introducing a delay in response to key presses.



1.2) Knobs and Position Encoders

- Rotating knobs have been used in the user interfaces of electronic equipment to allow the user to provide information of a continuous nature. A common example is the volume control knob on audio equipment, or the brightness control on a light dimmer.
- In analog electronic circuits, the knob usually controls a variable resistor or potentiometer.

- With the introduction of digital systems, knobs were replaced by switches in many applications. For example, the volume control on audio equipment was replaced with two buttons, one to increase the volume and another to decrease the volume.
- One form of digital knob input uses a shaft encoder. This form has the advantage that the absolute position of the knob is provided as an input to the system. However, a simpler form of input device uses an *incremental encoder* to determine direction and speed of rotation.
- An incremental encoder can also be used for a rotational position input in applications other than user interfaces, provided absolute positioning is not required. It can also be used for rotational speed input.
- An incremental encoder operates by generating two square-wave signals that are 90° out of phase, as shown at the top of Figure aside.
- The signals can be generated either using electromechanical contacts, or using an optical encoder disk with LEDs and photo-sensitive transistors, as shown in the middle and at the bottom of Figure.
- As the shaft rotates counterclockwise, the A output signal leads the B output signal by 90°. For clockwise rotation, A lags B by 90°. The frequency of changes between low and high on each signal indicates the speed of rotation of the shaft. A simple approach to using a knob attached to an incremental encoder involves detecting rising edges on one of the signals.
- Suppose we assume the knob is at a given position when the system starts operation. For example, we might assume a knob used as the volume control for a stereo is at the same setting as when the stereo was last used. When we detect a rising edge on the A signal, we examine the state of the B signal. If B is low, the knob has been turned counterclockwise, so we decrement the stored value representing the knob's position. If, on the other hand, B is high, the knob has been turned clockwise, so we increment the stored value representing the knob's position. Using an incremental encoder instead of an absolute encoder in this application makes sense, since the volume might also be changed by a remote control. It is a change in the knob's position that determines the volume, not the absolute position of the knob.

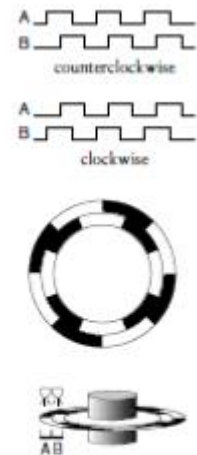


FIGURE Operation of an incremental encoder: quadrature signals output from the encoder (top); an optical encoder disk (middle); and the disk and optical sensors attached to a shaft (bottom).

1.3) Analog Inputs

Sensors for continuous physical quantities vary greatly, but they all rely on some physical effect that produces an electrical signal that depends on the physical quantity of interest. In most sensors, the signal level is small and needs to be amplified before being converted to digital form. Some sensors and the effects they rely on include:

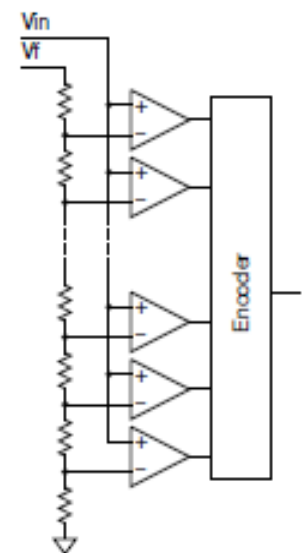
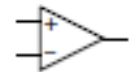
- **Microphones.** These are among the most common sensors in our everyday lives, and are included in digital systems such as telephones, voice recorders and cameras. A microphone has a diaphragm that is displaced by sound pressure waves. In an electret microphone, for example, the diaphragm forms one plate of a capacitor. The other plate is fixed and has a permanent charge embedded on it during manufacture. The movement of the plate's together and apart in response to sound pressure creates a detectable voltage across the plates that vary with the sound pressure. The voltage is then amplified to form the analog input signal.
- **Accelerometers** are used for measuring acceleration and deceleration. A common form of accelerometer used in automobile air bag controllers, for example, has a microscopic cantilevered beam manufactured on a silicon chip. The beam and the surface over which it is suspended form the two plates of a capacitor. As the chip accelerates (or, more important, in the air bag

application, decelerates), the beam bends closer to or farther from the surface. The corresponding change in capacitance is used to derive an analog signal.

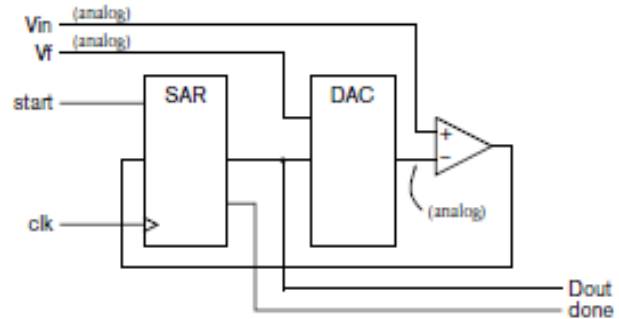
- **Fluid flow sensors.** There are numerous forms of sensor that rely on different effects to sense flow. One form uses temperature dependent resistors. Two matched resistors are self-heated using an electric current. One of the resistors is placed into the fluid stream which cools it by an amount dependent on the flow rate. Since the resistance depends on the temperature, the difference in resistance between the two resistors depends on the flow rate. The resistance difference is detected to derive an analog input signal. Other forms of flow-rate sensor use rotating vanes, pressure sensing in venture restrictions, and doppler shift of ultrasonic echoes from impurities. Different forms of sensor are appropriate for different applications.
- **Gas detection sensors.** There are numerous forms of sensors that use different effects and are appropriate for different applications. As an example, a photo-ionizing detector uses ultraviolet light to ionize a sample of atmosphere. Gas ions are attracted to plates that are held at a potential difference. A circuit path is provided for charge to flow between the plates. The current in the path depends on the concentration of the gas in the atmospheric sample. The current is sensed and amplified to form the analog input signal.

1.4) Analog-to-Digital Converters

- The analog input signals from sensors need to be converted into digital form so that they can be processed by digital circuits and embedded software. The basic element of an analog-to-digital converter (ADC) is a comparator, shown in Figure aside, which simply senses whether an input voltage (the +ve terminal) is above or below a reference voltage (the -ve terminal) and outputs a 1 or 0 accordingly.
- The simplest form of ADC is a *flash ADC*, illustrated in Figure below.
- A converter with n output bits consists of a bank of $2^n - 1$ comparators that compare the input voltage with reference voltages derived from a voltage divider.
- For a given input voltage $V_{in} = k V_f$, where V_f is the full scale voltage and k is a fraction between 0.0 and 1.0, a proportion k of the comparators have their reference voltage above V_{in} and so output 1, and the remaining comparators have their reference voltage lower than V_{in} and so output 0. The comparator outputs drive the encoder circuit that generates the fixed-point binary code for k .
- Flash ADCs have the advantage that they convert an input voltage to digital form very quickly. High speed flash ADCs can perform tens or hundreds of millions of samples per second, and so are suitable for converting high bandwidth signals such as those from high-definition video cameras, radio receivers, radars, and so on.
- Their disadvantage is that they need large numbers of comparators.
- Hence, they are only practical for ADCs that encode the converted data using a relatively small number of bits. Common flash ADCs generate 8 bits of output data. We say they have a *resolution* of 8 bits, corresponding to the precision of the fixed-point format with which they represent the converted signal.



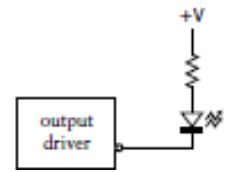
- For signals that change more slowly, we can use a **successive approximation ADC**, shown in Figure below. It uses a digital-to-analog converter (DAC) internally to make successively closer approximations to the input signal over several clock periods.
- To illustrate how the ADC works, consider a converter that produces an 8-bit output. When start input is activated, the successive approximation register (SAR) is initialized to the binary value 01111111.
- This value is provided to the DAC, which produces the first approximation, just less than half of the full-scale voltage. The comparator compares this approximation with the input voltage. If the input voltage is higher, the comparator output is 1, indicating that a better approximation would be above the DAC output.
- If the input voltage is lower, the comparator output is 0, indicating that a better approximation would be below the DAC output.
- The comparator output is stored as the most significant bit in the SAR, and remaining bits are shifted down one place. This gives the next approximation, d7011111, which is either one-quarter or three-quarters of the full-scale voltage, depending on d7.
- During the next clock period, this next approximation is converted by the DAC and compared with the input voltage to yield the next most significant bit of the result and a refined approximation, d7d6011111.
- The process repeats over successive clock cycles, refining the approximation by one bit each cycle. When all bits of the result are determined, the SAR activates the done output, indicating that the complete result can be read.
- The advantage of a successive approximation ADC over a flash ADC is that it requires significantly fewer analog components: just one comparator and a DAC.
- These components can be made to high precision, giving a high-precision ADC.
- 12-bit successive approximation ADCs, for example, are commonly available. The disadvantage, however, is that more time is required to convert a value. If the input signal changes by more than the precision of the ADC while the ADC is making successive approximation, we need to *sample and hold* the input. This requires a circuit that charges a capacitor to match the input voltage during a brief sampling interval, and then maintain the voltage on the capacitor while it is being converted.
- Another disadvantage of the successive approximation ADC is the amount of digital circuitry required to implement the SAR. However, that function could be implemented on an embedded processor, requiring just an output register to drive the DAC and an input bit from the comparator. The sequencing of successive approximations would then form part of the embedded software.



2) OUTPUT DEVICES

2.1) LEDs

- Among the most common output devices are indicator lights that display on/off or true/false information. For example, an indicator might show whether a mode or operation is active, whether the system is busy, or whether an error condition has occurred. The simplest form of indicator is a single light-emitting diode (LED) as shown.
- When the output from the driver is a low voltage, current flows through the LED, causing it to turn on. The resistor limits the current so as not to overload the output driver or the LED.
- We choose the resistance value to determine the current, and hence the brightness of the LED. When the output from the driver is a high voltage, the voltage drop across the LED is less than its threshold voltage, so no current flows; hence, the LED is turned off.
- Alternatively, the LED and resistor can be connected to ground, allowing a high output voltage to turn on the LED and a low output voltage to turn it off.
- However, output circuits designed to drive TTL logic levels are better able to sink current in the low state than to source current in the high state. Thus, it is more common to connect an LED as shown in Figure.

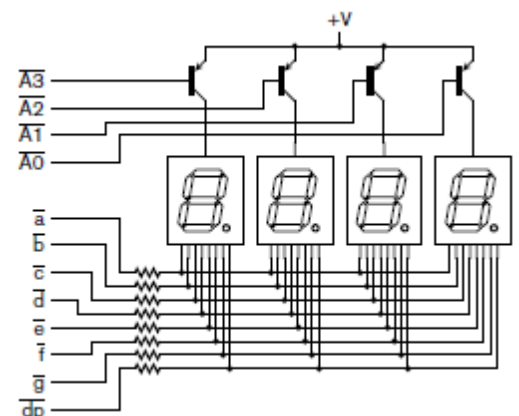
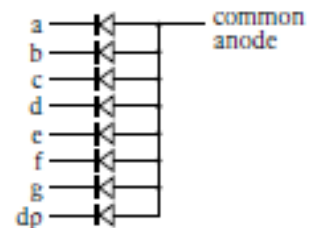


Problem: Determine the resistance for an LED pull-up resistor connected to a 3.3V power supply. The LED has a forward-biased voltage drop of 1.9V, and is sufficiently bright with a current of 2mA.

Solution Assuming the output driver low voltage is close to 0V, the voltage drop across the resistor must be $3.3V - 1.9V = 1.4V$. Using Ohm's Law with a current of 2mA means the resistance must be $1.4/0.002 = 700\Omega$. The closest standard value is 680 Ω .

2.2) Displays

- 7-segment displays consists of LEDs connected together to display digits. There are two types of displays namely common anode and common cathode.
- The connections for the LEDs in each digit, in this case, with common anodes, are shown in Figure. In addition to the seven LEDs for the segments, there is an LED for a decimal point (dp).
- The output connections for four digits are shown in Figure below. Each of the outputs $\overline{A0}$ through $\overline{A3}$, when pulled low, turns on the transistor that enables a digit.
- These external transistors are needed since IC outputs cannot source enough current to drive up to eight LEDs directly. To display four digits, we pull each of $\overline{A0}$ through $\overline{A3}$ low in turn. When $\overline{A0}$ is low, enabling the least significant digit, we drive the segment lines, \overline{a} through \overline{g} \overline{dp} , low or high as required for the segment pattern for that digit. When $\overline{A1}$ is low, we drive the segment lines for the next digit, and so on. After driving the most significant digit, we cycle back



to the least significant digit. If we cycle through the digits fast enough, our eyes' persistence of vision smooth's out any flickering due to each digit only being active 25% of the time.

- The advantage of this scanned scheme is that we only need one signal for each digit plus one for each segment of a digit. For example, to drive four digits, we need 12 signals, compared with the 32 signals we would need had we driven segments individually.
- Depending on our application, we might use a counter or a shift register to drive the digit enable outputs and an 8-bit-wide multiplexer to select the values to drive onto the segment outputs. Often, however, the display is controlled by an embedded processor. In that case, we can simply provide output registers for the digit and segment outputs and let the embedded software manage the sequencing of output values.
- As an alternative to using LEDs for displays, some systems use liquid crystal displays (LCDs). Each segment of an LCD consists of liquid crystal material between two optical polarizing filters. The liquid crystal also polarizes light, and, depending on the angle of polarization, can allow light to pass or be blocked by the filters.
- The liquid crystal is forced to twist or untwist, thus changing its axis of polarization, by application of a voltage to electrodes in front of and behind the segment. By varying the voltage, we can make the segment appear transparent or opaque.
- Thus, LCDs require ambient light to be visible. In low light conditions, a back light is needed, which is one of their main disadvantages.
- The other disadvantages include their mechanical fragility and the smaller range of temperatures over which they can operate.
- They have several advantages over LEDs, including readability in bright ambient light conditions, very low power consumption, and the fact that custom display shapes can readily be manufactured.
- Seven-segment displays are useful for applications that must display a small amount of numeric information. However, more complex applications often need to display alphanumeric or graphical information, and so may use LCD display panels.
- These can range from small panels that can display a few characters of text, to larger panels that can display text or images up to 320 x 240 dots, called *pixels* (short for picture elements).

2.3) Electromechanical Actuators and Valves

- One of the simplest forms of actuator used to cause mechanical effects is a *solenoid*, shown in Figure. With no current flowing through the coil, the spring holds the steel armature out from the coil. When current flows, the coil acts as an electromagnet and draws the armature in against the spring.
- In a digital system, we can control the current in a small solenoid with a transistor driven by a digital output signal, as shown in Figure below.
- The diode is required to absorb the voltage spikes that arise when the current through the inductive load is turned off. The direct mechanical effect of activating a solenoid is a small linear movement of the armature. We can translate this into a variety of other effects by attaching rods and levers to the armature, allowing us to control the operation of mechanical systems.

- There are two important classes of devices based on solenoids, the first being solenoid valves. We can attach the armature of a solenoid to a valve mechanism, allowing the solenoid to open and close the valve, thus regulating the flow of a fluid or gas. This gives us a means of controlling chemical processes and other fluid or gas based processes.
- Importantly, a hydraulic solenoid valve (controlling flow of hydraulic fluid) or a pneumatic solenoid valve (controlling flow of compressed air) can be used to indirectly control hydraulic or pneumatic machinery. Such machines can operate with much greater force and power than electrical machines. So solenoid valves are important components in the interface between the disparate low-power digital electronic domain and the high-power mechanical domain.
- The second class of device based on solenoids is relays. In these devices, the armature is attached to a set of electrical contacts. This allows us to open or close an external circuit under digital control. The reasons for using a relay are twofold. First, the external circuit can operate with voltages and currents that exceed those of the digital domain. For example, a home automation system might use a relay to activate mains power to a mains powered appliance. Second, a relay provides electrical isolation between the controlling and the controlled circuit. This can be useful if the controlled circuit operates with a different ground potential, or is subject to significant induced noise.

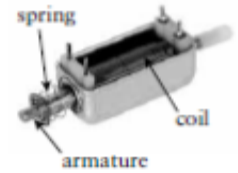


FIGURE A solenoid actuator.

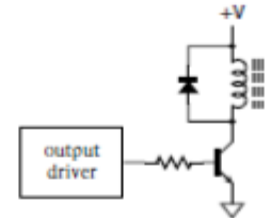
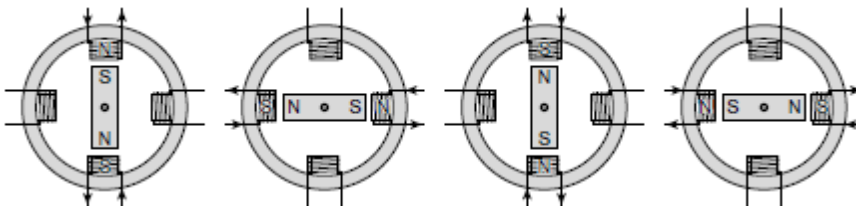


FIGURE Solenoid controlled by a digital output.

2.4) Motors

- Solenoids allow us to control a mechanical effect with two states, many applications require mechanical movement over a range of positions and at varying speeds. For these applications, we can use electric motors of various kinds, including stepper motors and servo motors.
- Both can be used to drive shafts to controlled positions or speeds. The rotational position or motion can be converted to linear position or motion using gears, screws, and similar mechanical components.
- A stepper motor is the simpler of the two kinds of motors that we can control with a digital system. Its operation is shown in simplified form in Figure below.

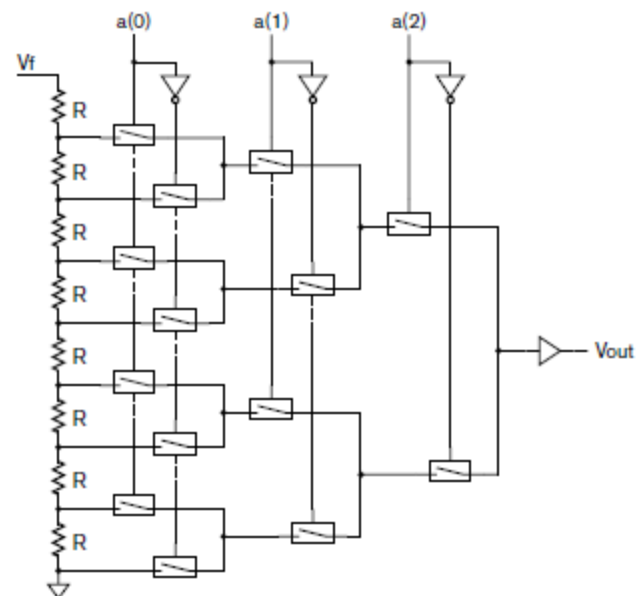


- The motor consists of a permanent magnet rotor mounted on the shaft. Surrounding the rotor is a stator with a number of coils that can be energized to form electromagnetic poles. The figure shows that, as coils are energized in sequence, the rotor is attracted to successive angular positions, stepping around through one rotation. The magnetic attraction holds the rotor in position, provided there is not too much opposing torque from the load connected to the motor shaft. The order and rate in which the coils are energized determines the direction and speed of rotation.

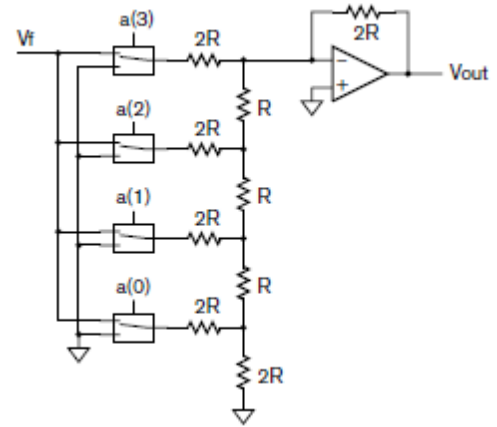
- Practical stepper motors have more poles around the stator, allowing the motor to step with finer angular resolution. They also have varying arrangements of coil connections, allowing finer control over stepping. In practical applications, current through the coils is switched in either direction using transistors controlled by digital circuit outputs. The fact that the motor is activated by the on/off switching of current makes stepper motors ideal for digital control.
- A servo-motor, unlike a stepper motor, provides continuous rotation. The motor itself can be a simple DC motor, in which the applied voltage determines the motor's speed, and the polarity of the applied voltage determines the direction of rotation.
- The "servo" function of the motor involves the use of feedback to control the position or speed of the motor. If we are interested in controlling position, we can attach a position sensor to the motor shaft. We then use a servo controller circuit that compares the actual and desired positions, yielding a drive voltage for the motor that depends on the difference between the positions. If we are interested in controlling the speed, we can attach a tachometer (a speed sensor) to the shaft, and again use a comparator to compare actual and desired speed to yield the motor drive voltage. In both cases, we can implement the servo controller as a digital circuit or using an embedded processor.
- We need a digital-to-analog converter to generate the drive voltage for the motor. Realistic servo control involves fairly complex computations to compensate for the non-ideal characteristics of the motor and any gearbox and other mechanical components, as well as dealing with the effects of the mechanical load on the system.

2.5) Digital-to-Analog Converters

- Digital-to-analog converters (DACs) are the complement of analog-to-digital converters. A DAC takes a binary-encoded number and generates a voltage proportional to the number. We can use the voltage to control analog output devices, such as the servo motors, loudspeakers, and so on.
- One of the simplest forms of DAC is an *R-string DAC*, shown in Figure. Like the flash ADC, it contains a voltage divider formed with precision resistors.
- The binary-encoded digital input is used to drive a multiplexer formed from analog switches, selecting the voltage corresponding to the encoded number. The selected voltage is buffered using a unity gain analog amplifier to drive the final output voltage.
- This form of DAC works well for a small number of input bits, since it is possible to match the resistances to achieve good linearity.
- However, for a larger number of input bits, we require an exponentially larger number of resistors and switches. This scheme becomes impractical for DACs with more than eight to ten input bits.



- An alternative scheme is based on summing of currents in resistor networks. One way of doing this is shown in Figure aside called as an $R/2R$ ladder DAC.
- Each of the switches connected to the input bits connects the $2R$ resistance to the reference voltage V_f if the input is 1, or to ground if the input is 0.
- The currents sourced into the input node of the op-amp when the switches are in the 1 position are binary weighted. Those switches in the 0 position source no current. The superposition of the sourced currents means that the total current is proportional to the binary coded input. The op-amp voltage is thus also proportional to the binary coded input, in order to maintain the virtual ground at the op-amp input.



3) I/O CONTROLLERS

Given transducers, analog-to-digital converters and digital-to-analog converters, we can construct digital systems that include circuits to process the converted input information in digital form to yield output information. However, for an embedded computer to make use of the information, we need to include components that allow the embedded software to read input information and to write output information.

For dealing with input, we can provide an *input register* whose content can be loaded from the digital input data and that can be read in the same way that the processor reads a memory location. For dealing with output, we can provide an *output register* that can be written by the processor in the same way that it writes to a memory location.

The output signals of the register provide the digital information to be used by the output transducer. Many embedded processors refer to input and output registers as *ports*.

In practice, both input and output registers are parts of input and output controllers that govern other aspects of dealing with transducers under software control.

3.1) SIMPLE I/O CONTROLLERS

The simplest form of controller consists just of an input register that captures the data from an input device, or just an output register to provide data to a device. Usually, there are several I/O registers, so we need to select which register to read from or write to.

This is similar to selecting which memory location to access, and is solved in the same way, namely by providing each register with an address. When the embedded processor needs to access an input or output register, it provides the address of the required register. We decode the address to select the register, and only enable reading or writing of that register.

Some processors use memory mapped I/O; that is, they just use certain memory addresses to refer to I/O registers and use the same load and store instructions for accessing both memory location and I/O registers. We can use address decoding circuits connected to the processor to identify whether memory or I/O registers are being accessed, and enable the memory chips or the appropriate register as required. While a simple I/O controller just has registers for input and output of data, more involved I/O controllers also have registers to allow the embedded processor to manage operation of

the controller. Such registers might include *control registers*, to which a processor writes parameters governing the way transducers operate, and *status registers*, from which the processor reads the state of the controller.

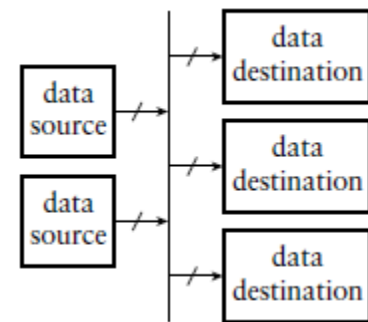
We often require such registers for controllers whose operation is sequential, since we need to synchronize controller operation with execution of the embedded software. As a consequence, we may have a combination of readable and writable registers used to control an input-only device or an output-only device.

3.2) AUTONOMOUS I/O CONTROLLERS

- The simple I/O controllers in the previous section either involve no sequencing of operations or just simple sequencing in response to accesses by a processor. More complex I/O controllers, on the other hand, operate autonomously to control the operation of an input or output device. For example, a servo-motor controller, given the desired position in an output register, might independently compute the difference between desired and actual position, compensate for mechanical lead and lag, and drive the motor accordingly. Interaction with the processor might only occur through the processor updating the desired position in the output register and monitoring the position difference by reading an input register. In some cases, if an autonomous controller detects an event of interest to the embedded software, for example, an error condition, the controller must notify the processor.
- One reason for providing autonomy in the controller is that it allows the processor to perform other tasks concurrently. This increases the overall performance of the system, though at the cost of the additional circuitry required for the controller. Another reason is to ensure that control operations are performed fast enough for the device. If the device needs to transfer data at high rates, or needs control operations to be performed without delay, a small embedded processor may not be able to keep up. Making the I/O controller more capable may be a better trade-off than increasing the performance or responsiveness of the processor.
- The use of an autonomous controller may be appropriate for a device that must transfer input or output data at high rates. Often, such data must be written to memory (in the case of input data) or read from memory (in the case of output data). If the data transfer were done by a program copying data between memory and controller registers, that activity would consume much of the processor's time. An alternative, commonly adopted in high-speed autonomous controllers, is to use *direct memory access* (DMA), in which the controller reads data from memory or writes data to memory without intervention by the processor. The processor provides the starting memory address to the controller (by writing the address to a control register), and the controller then performs the data transfer autonomously.

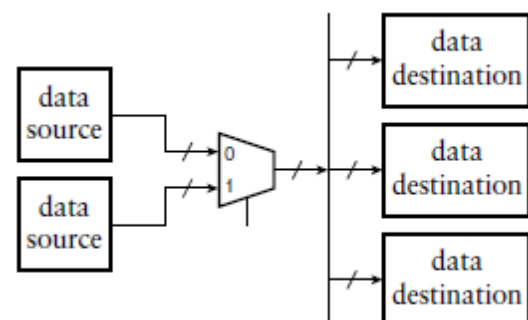
4) PARALLEL BUSES

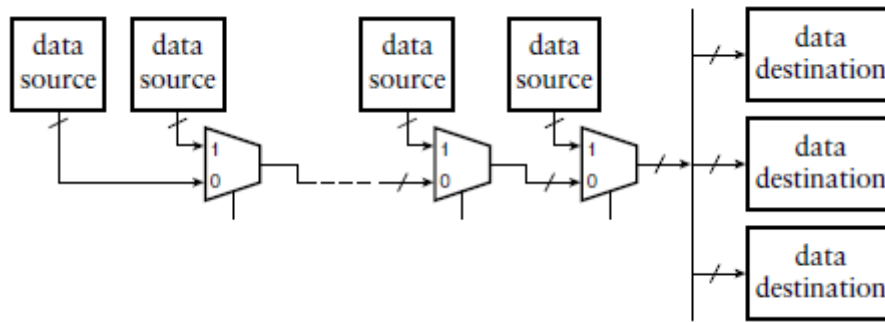
- Digital circuits consist of various interconnected components. Each component performs some operation or stores data. The interconnections are used to move data between the components. When the data is binary coded, several signals are connected in parallel, one per bit of the encoding.
- Many of the interconnections are simple point-to-point connections, with one component as the source of data and a single separate component as the destination. In other cases, connections fan out from a single source to multiple destinations, allowing each of the destination components to receive data from the source.
- In some systems, especially embedded systems containing processor cores, parallel connections carry encoded data from multiple sources to several alternate destinations. Such connection structures, shown conceptually in Figure, are called *buses*.
- A bus is a collection of signals carrying the data, and control remains in a separate control section that sequences operation of the data sources and destinations.
- In more elaborate buses, data sources and destinations are autonomous, each with its own control section. In such cases, the control sections must communicate to synchronize the transfer of data.
- The Figure shows the general idea of bus connection structures, but it is not realizable directly as shown. Since the bus signals are shared between the data sources, only one of them should provide data at once. Most of the circuit components that we have considered so far always drive either a low or a high logic level at their outputs. If one data source drives a low level while another drives a high level, the resulting conflict would cause large currents to flow between the two components, possibly damaging them.
- There are several solutions to this problem, as following:



4.1) MULTIPLEXED BUSES

- One solution is to use a multiplexer to select among the data sources, as shown in Figure aside. The multiplexer selects the value to drive the bus signals based on a control signal generated by a control section. If the bus has n data sources, an n -input multiplexer is required for each bit of the encoded data transmitted over the bus.
- Depending on the number of sources and the arrangement of the components and signals on the integrated circuit chip, the multiplexer may be implemented as a single n -input multiplexer, or it may be subdivided into sections distributed around the chip. One extreme form of subdivision of bus multiplexers is the fully distributed structure shown in Figure below.

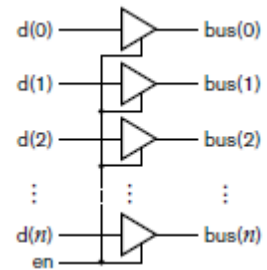
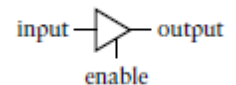




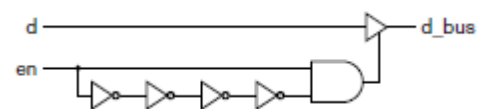
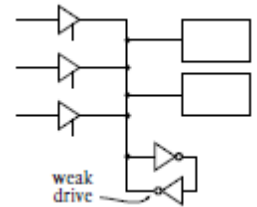
The data signals are connected in a chain going past all of the sources and then routed to the destinations. Each multiplexer either connects its associated data source to the chain (when the multiplexer's select input is 1) or forwards data from a preceding source (when the select input is 0). The advantage of this form of distributed multiplexer is the reduction in wiring complexity.

4.2) TRISTATE BUSES

- A solution to avoiding contention on a bus is to use *tristate* bus drivers. The outputs of a tristate driver can be turned off by placing it in a *high-impedance*, or *hi-Z*, state. The symbol for a tristate driver is shown in Figure aside.
- When the enable input is 1, the driver behaves like an ordinary output, driving either a low or a high logic level on the output. When the enable input is 0, the driver enters the high-impedance state by turning its output-stage transistors off.
- We can implement a bus with multiple data sources by using tristate drivers on the outputs of each data source. We use one driver for each bit of encoded data provided by the source, and connect the enable inputs of the drivers for a given source together, as shown in Figure 2.
- That way, a source either drives a data value onto the bus, or has all bits in the high impedance state. The control section selects a particular source to provide data by setting the enable input of that source's drivers to 1, and all other enable inputs to 0.
- One of the main advantages of tristate buses is the reduction in wiring that they afford. For each bit of the encoded data on the bus, one signal wire is connected between all of the data sources and destinations.
- However, there are some issues to consider. First, since bus wires connect all of the sources and destinations, they are generally long and heavily loaded with the capacitance of the drivers and inputs. As a consequence, the wire delay may be large, making high-speed data transfer difficult.
- Moreover, the large capacitance means we need more powerful output-stage circuits, increasing the area and power consumption of the chip.
- A second issue is difficulty in designing the control that selects among data sources. The control section must ensure that one source's drivers are disabled before any other source's drivers are enabled.
- When we design the control section, we need to take into account the timing involved in disabling and enabling drivers. This is shown in Figure 3.



- When the enable input of a driver changes to 0, there is a delay, toff, before the driver disconnects from the bus. Similarly, when the enable input changes to 1, there is a delay, ton, before the driver delivers a valid low or high logic level on the bus. In the intervening time, the bus *floats*, indicated on the timing diagram by a dashed line midway between the low and high logic levels.
- Since there is no output driving a low or high logic level on the bus signals, each signal drifts to an unspecified voltage. Letting the bus float to an unspecified logic level can cause switching problems in some designs. The bus signal might float to a voltage around the switching threshold of the bus destination inputs. Small amounts of noise voltage induced onto the bus wire can cause the inputs to switch state frequently, causing spurious data changes within the data destination and consuming power unnecessarily.
- We can avoid floating logic levels on the bus signals by attaching a *weak keeper* to the signal, as shown in Figure aside.
- The keeper consists of two inverters providing positive feedback to the bus signal. When the bus is forced to a low or high logic level by a bus driver, the positive feedback keeps it at that level, even if the forcing driver is disabled. The transistors in the output circuit of the inverter driving the bus are small, with relatively high on-state resistance, and so cannot source or sink much current. They are easily overridden by the output stages of the bus drivers.
- When we need to change from one data source to another, it might seem reasonable to disable one driver at the same time as enabling the next driver. However, this can cause driver contention. If the toff delay of the disabled driver is at the maximum end of its range and the ton delay of the enabled driver is at the minimum end, there will be a period of overlap where some bits of the enabled driver may be driving opposite logic levels to those of the disabled driver. The overlap will be short-lived and is unlikely to destroy the circuit. However, it does contribute extra power consumption and heat dissipation and ultimately will reduce the operating life of the circuit.
- The overlap effect can be exacerbated by clock skew in the control section. If the flip-flop that generates the enabling signal receives its clock earlier than the flip-flop that generates the disabling signal, there will be an increased chance of overlap, even if the on and off delays of the tristate drivers are near their nominal values. Given these considerations, the safest approach when designing control for tristate buses is to include a margin of dead time between different data sources driving the bus. A conservative approach is to defer enabling the next driver until the clock cycle after that in which the previous driver is disabled.
- A more aggressive approach is to delay the rising edges of the enable signals, for example, using the circuit of Figure aside, to avoid overlap between drivers. As many pairs of inverters are included as give the required delay. However, this approach requires very careful attention to timing analysis to ensure that it works effectively across the expected range of operating conditions.
- A third issue relating to design of tristate buses is the support provided by CAD tools. Not all physical design tools provide the kinds of timing and static loading analyses needed to design tristate buses effectively. Similarly, tools that automatically incorporate circuit structures to enable testing of circuits after their manufacture don't always deal with tristate buses correctly. If the tools we use don't support tristate buses, we must resort to manual methods to complete and verify our design.



Modeling Tristate Drivers in Verilog

There are two aspects to modeling tristate drivers: representing the high impedance state, and representing the enabling and disabling of drivers.

- Nets and variables can also take on the value Z for representing the high-impedance state. In a Verilog model for a circuit, we can assign Z to an output to represent disabling the output. Subsequently, assigning 0 or 1 to the output represents enabling it again.
- Z value can be written using either an uppercase or lowercase letter. Thus, 1'bZ and 1'bz are the same.
- Second, we can only write literal Z values as part of a binary, octal or hexadecimal number, such as 1'bZ, 3'oZ and 4'hZ. In an octal number, a Z represents three high-impedance bits, and in a hexadecimal number, a Z represents four high-impedance bits.
- Third, Verilog allows us to use the keyword **tri** instead of wire for a net connected to the output of a tristate driver. Thus, we might write the following declaration in a module:

```
tri d_out;
```

or the following port declaration: **module** m (**output tri** a, ...);

- Apart from the use of the different keyword, a tri net behaves exactly the same as a wire net. The tri keyword simply provides documentation of our design intent. Note that there is no corresponding keyword for a variable that is assigned a Z value; we continue to use the reg keyword for that purpose.
- For multibit buses, we can use vectors whose elements include Z values. While we can assign 0, 1 and Z values individually to elements of vectors, we usually assign either a vector containing just 0 and 1 values to represent an enabled driver or a vector of all Z values to represent a disabled driver. Verilog's implicit resizing rules for vector values involve extending with Z elements if the leftmost bit of the value to be extended is Z. So we can write 8'bz to get an 8-element vector of Z values.
- Each assignment within the module represents one of the 8-bit sections of the component. The condition in the assignment determines whether the 8-bit tristate driver is enabled or disabled. The driver is disabled by assigning a vector value consisting of all Z elements.
- When we have multiple data sources for a tristate bus, our Verilog model includes multiple assignment statements that assign values to the bus. Verilog must *resolve* the values contributed by the separate assignments to determine the final value for the bus. If one assignment contributes 0 or 1 to a bus and all of the others contribute Z, the 0 or 1 value overrides the others and becomes the bus value.
- This corresponds to the normal case of one driver being enabled and the rest disabled. If one assignment contributes 0 and another contributes 1, we have a conflict. Verilog then uses the special value X, called *unknown*, as the final bus value, since it is unknown whether a real circuit would produce a low, high or invalid logic level on the bus.
- Depending on how the Verilog model of a data destination receiving an X value is written, it might propagate the unknown value to its outputs, or produce arbitrary 0 or 1 values. Ideally, it would include a verification test statement that would detect unknown input values. If all assignments to a bus contribute Z, the final signal value is Z. This corresponds to the bus floating.
- Again, since this does not represent a valid logic level, a Verilog model of a data destination receiving a Z input should propagate an X output and detect the error condition.

- Assignment of X to an output is a notational device used in simulation to propagate error conditions in cases where we cannot determine a valid output value.
- We can write Verilog statements that test whether a bus has the value Z or X, but it only makes sense to do so in testbench models. If we need to test for Z or X values in a testbench model, we should use the == operator in Verilog, known as the *logical equality* operator, represents a hardware equivalence operation.
- If either operand is Z or X, the result is X, since it is unknown whether the values in a real circuit are equivalent or not. Similarly, the != operator, *logical inequality*, represents a hardware unequivalence operation, and returns X if either operand is Z or X. Thus, for example, the expressions 1'b0 == 1'bX and 1'bZ != 1'b1 both yield X.
- If we want to test for Z and X values, we must use the === and !== operators, known as the *case equality* and *case inequality* operators, respectively. These perform an exact comparison, including X and Z values. Thus, 1'b0 === 1'bX yields 0 (false), and 1'bZ !== 1'b1 yields 1 (true). Note that, like the Z value, we can use an uppercase or lowercase letter, and we can only write literal X values in binary, octal, or hexadecimal numbers.

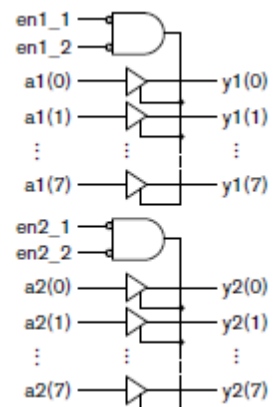
Problem: Write a Verilog statement to model a tristate driver for an output net d_out . The driver is controlled by a net d_en , and when enabled, drives the value of an input d_in onto the output net.

Solution : We can use an assignment statement, as follows: $assign\ d_out = d_en ? d_in : 1'bZ;$

Problem: The SN74x16541 component manufactured by Texas Instruments is a dual 8-bit bus buffer/driver in a package for use in a printed circuit board system. The internal circuit of the component is shown in Figure. Develop a Verilog model of the component.

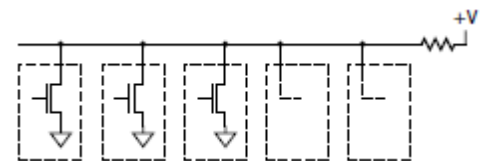
Solution: We can use vector ports for each of the 8-bit inputs and outputs, and single-bit ports for the enable inputs. The module definition is:

```
module sn74x16541 ( output tri [7:0] y1, y2,
input [7:0] a1, a2,
input en1_1, en1_2, en2_1, en2_2 );
assign y1 = (~en1_1 & ~en1_2) ? a1 : 8'bz;
assign y2 = (~en2_1 & ~en2_2) ? a2 : 8'bz;
endmodule
```



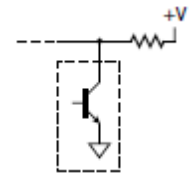
4.3) OPEN-DRAIN BUSES

- A third solution to avoid bus contention is to use open-drain drivers, as shown in Figure. Each driver connects the drain terminal of a transistor to the bus signal. When any of the transistors is turned on, it pulls the bus signal to a low logic level. When all of the transistors are turned off, the termination resistor pulls the bus signal up to a high logic level.
- If multiple drivers try to drive a low logic level, their transistors simply share the current load. If there is a conflict, with one or more drivers trying to drive a low level and others letting the bus be pulled up, the low-level drivers win. Sometimes, this kind of bus is called a wired- AND bus, since



the bus signal is only 1 if all of the drivers output 1. If any driver outputs 0, the bus signal goes to 0. The AND function arises from the wiring together of the transistor drains.

- We can also use this form of bus with drivers that use bipolar transistors instead of MOSFET transistors. In that case, we connect the collector terminal of a transistor to the bus signal, as shown in Figure. Such a driver is called an open-collector driver.



- Given the need for a pull-up resistor on each bus signal, open-drain or open-collector buses are usually found outside integrated circuits. For example, they may be used for a bus that connects a number of integrated circuits together, or for the signals in a backplane bus that connects a number of printed circuit boards together.
- Implementing pull-up resistors within an integrated circuit takes up significant area and consumes power. Hence, we usually use multiplexed or tristate buses within an integrated circuit chip. If we need the AND function that would be formed by open-drain connection, we can implement it with active gates.

Modeling Open-Drain and Open-Collector Connections in Verilog

- We can model open-drain and open-collector drivers using a different kind of net, declared with the keyword `wand` (short for wired-AND). For example:

```
wand bus_sig;
```

- We assign 0 to a wand net to represent a driver whose output transistor is turned on, pulling the net low. We assign 1 to the net to represent a driver whose output transistor is turned off. When a wand net is resolved, any 0 values override all other values. However, if all of the drivers are turned off, contributing 1 values, the final value of the net is 1.

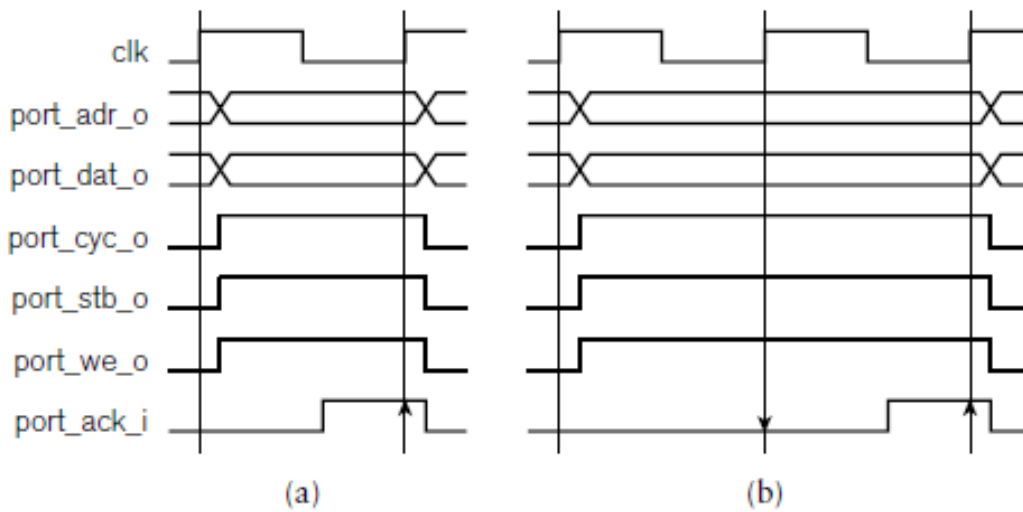
4.4) BUS PROTOCOLS

- In most design projects, subsystems are often designed by different team members. Some subsystems may also be procured from external providers, or be implemented using off-the-shelf components.
- If the subsystems are to be interconnected using buses, it would be preferable for them to use the same bus signals with the same timing requirements; otherwise, interface glue logic is required.
- In order to facilitate connection of separately designed components, a number of common *bus protocols* have been specified. Some of the specifications are embodied in industry and international standards, whereas others are simply specifications agreed upon or promoted by component vendors.
- The specification of a bus protocol includes a list of the signals that interconnect compliant components, and a description of the sequences and timing of values on the signals to implement various bus operations.
- Bus specifications and protocols vary, depending on their intended use. Some, intended for connecting separate chips on a circuit board or separate boards in a system, use tristate drivers for signals that have multiple data sources.
- Examples include the PCI bus used to connect add-on cards to personal computer systems, and the VXI bus used to connect measurement instruments to controlling computers. Others are intended for connecting subsystems within an IC. They have separate input and output signals, allowing for connection using multiplexers or switching circuits. Examples include the AMBA buses specified

by ARM, the Core-Connect buses specified by IBM, and the Wishbone bus specified by the Open Cores Organization. Buses also vary in the number of parallel signals for transferring addresses and data, and in the speed of operation.

- The Wishbone I/O bus signals for the Gumnut are described as follows:
 - ✓ port_cyc_o: a “cycle” control signal that indicates that a sequence of I/O port operations is in progress.
 - ✓ port_stb_o: a “strobe” control signal that indicates an I/O port operation is in progress.
 - ✓ port_we_o: a “write enable” control signal that indicates the operation is an I/O port write.
 - ✓ port_ack_i: a status signal that indicates that the I/O port acknowledges completion of the operation.
 - ✓ port_adr_o: the 8-bit I/O port address.
 - ✓ port_dat_o: The 8-bit data written to the addressed I/O port by an out instruction.
 - ✓ port_dat_i: the 8-bit data read from the addressed I/O port by an inp instruction.

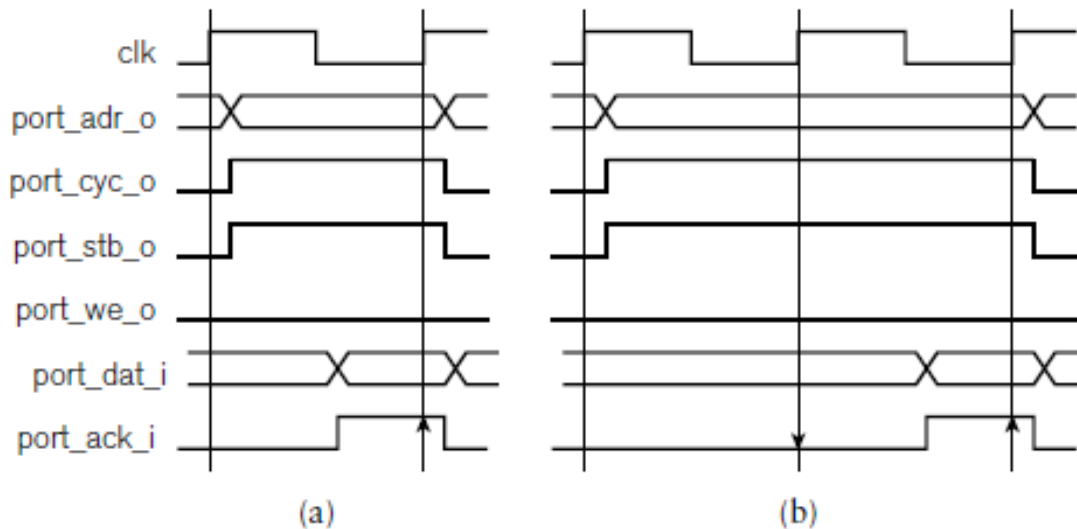
When the Gumnut core executes an out instruction, it performs a port write operation. The timing of the operation is shown in Figure below.



- Transitions are synchronized by the system clock. The Gumnut starts a write operation by driving the port_adr_o signals with the address computed by the out instruction and the port_dat_o signals with the data from the source register of the out instruction.
- It sets the port_cyc_o, port_stb_o and port_we_o control signals to 1 to indicate commencement of the write operation.
- The system in which the Gumnut is embedded decodes the port address to select an I/O controller and to enable the addressed output register to store the data. If the addressed controller is able to update the register within the first clock cycle, it sets the port_ack_i signal to 1 in that cycle, as shown in Figure (a). On the next rising clock edge, the Gumnut sees port_ack_i at 1 and completes the operation by driving port_cyc_o, port_stb_o and port_we_o back to 0.
- If, on the other hand, the addressed controller is slow and is not able to update the output register within the cycle, it leaves port_ack_i at 0, as shown in Figure (b). The Gumnut sees port_ack_i at 0 on the rising clock edge, and extends the operation for a further cycle. The controller can keep port_ack_i at 0 for as long as it needs to update the register. Eventually, when it is ready, it drives

port_ack_i to 1 to complete the operation. This form of synchronization, involving strobe and acknowledgment signals, is often called *handshaking*.

The Gumnut performs a port read operation when it executes an inp instruction. The timing for the operation, shown in Figure below, is similar to that for a port write.

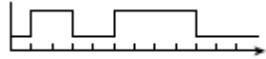
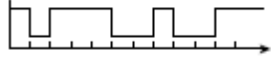


- The Gumnut starts the port read operation by driving the port_adr_o signals with the computed address, driving the port_cyc_o and port_stb_o signals to 1, and leaving port_we_o at 0. Again, the system decodes the address to select an I/O controller and enable the addressed input register onto the port_dat_i signals.
- The controller drives the port_ack_i signal to 1 as soon as it has supplied the data, either during the first cycle, as in Figure (a), or in a subsequent cycle, as in Figure 8.31(b). On seeing port_ack_i at 1, the Gumnut transfers the data from the port_dat_i signals to the destination register identified in the inp instruction. It then completes the port read operation by driving port_cyc_o and port_stb_o back to 0. At first sight, it might appear that the port_cyc_o and port_stb_o signals are duplicates of each other.

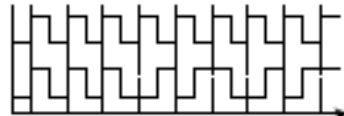
5) SERIAL TRANSMISSION

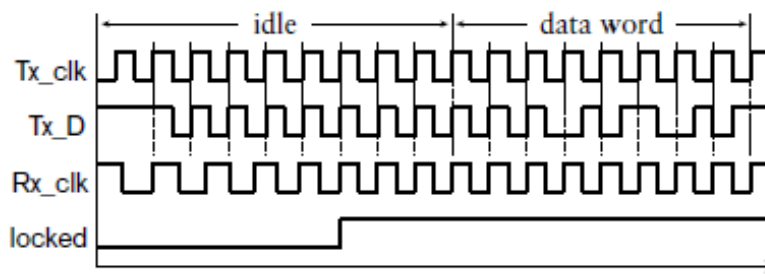
- Transfer of binary-encoded data using *parallel transmission*, requires one signal wire per bit. For wide encodings, the wiring takes up significant circuit area, and makes layout and routing of the circuit more complex. For connections that extend between chips, parallel transmission requires more pad drivers and receivers, more pins, and more PCB traces. These all add cost to the system. Moreover, there are secondary effects, such as increased delay due to the extra space required for the connections, problems with crosstalk between wires routed in parallel, and problems with skew between signals. Dealing with these problems adds cost and complexity to the system.
- Thus an alternative scheme for transferring binary-encoded data is the use *serial transmission*, since bits are transmitted one bit at a time in series over a single signal wire.

5.1) SERIAL TRANSMISSION TECHNIQUES

- In order to transform data between parallel and serial form, we can use shift registers. At the transmitting end, we load the parallel data into a shift register and use the output bit at one end of the register to drive the signal. We shift the content of the register one place at a time to drive successive bits of data onto the signal. At the receiving end, as each bit value arrives on the signal, we shift it into a shift register. When all the bits have arrived, the complete data code word is available in parallel form in the shift register.
- The terms *serializer/deserializer*, or *serdes*, are used for shift registers used in this way. The advantage of serial transmission is that we only need one signal wire to transfer the data. Thus, we reduce the circuit area and cost for the connection. Moreover, if necessary, we can afford to optimize the signal path so that bits can be transferred at a very high rate. Some serial transmission standards in use today allow for rates exceeding 10 gigabits per second.
- One important issue that we need to address when transferring data serially is the order in which we transmit the bits. In principle, the order is arbitrary, so long as the transmitter and receiver agree. Otherwise, the receiver will end up with the bits in reverse order.
- Serial transmission in a system is often governed by a standard that specifies the order. Another important issue is synchronization of the transmitter and the receiver. If we just drive the signal with the data bit values, there is no indication of when the time for one bit ends and the time for the next bit starts. This form of serial transmission is called *non-return to zero (NRZ)*, and is illustrated in Figure , which shows the logic levels on a signal for NRZ serial transmission of the value 11001111, with the most significant bit being transmitted first. We assume in this case that the value on the signal when no bit is being transmitted is 0.
 
- In the figure, we have drawn a timescale showing the interval in which each bit occurs. However, that information is implicit, rather than being explicitly transmitted to the receiver along with the data. If the receiver, for some reason, assumed intervals twice as long for each bit, it would receive the value 10110000.
- To avoid this problem, we need to synchronize the transmitter and receiver, so that the receiver samples each bit value on the signal at some time during the interval when the transmitter drives the signal with the bit value.
- There are three basic ways in which we can synchronize the transmitter and receiver.
 - ✓ The first is by transmitting a clock on a separate signal wire.
 - ✓ The second is by signaling the start of a serial code word and relying on the receiver to keep track of the individual bit intervals. A common way of doing this originated with teletypes, which were computer terminals consisting of a keyboard and a printer connected to a remote computer using serial transmission.
 - ✓ A refined version of such serial transmission is still used to connect some devices to serial communications ports on modern PCs.
- In this second scheme, the signal is held at a high logic level when there is no data to transmit. When data is ready to be transmitted, transmission proceeds as shown in Figure, again with the most significant bit transmitted first.
 
 The signal is brought to a low logic level for one bit time to indicate the start of transmission. This is the start bit. After that, the bits of data are transmitted, each for

one bit time. We might also transmit a parity bit after the data bits, in case the signal wire is subject to induced noise. This would allow us to detect some errors that might occur during transmission. Finally, we drive the signal high for one further bit time to indicate the end of transmission of the data. This is the stop bit.

- We can then transmit the next piece of data, starting with a start bit, or leave the signal high if there is no data ready to transmit. At the receiving end, the receiver monitors the logic level on the signal. While it remains at a high logic level, the receiver is idle. When the receiver detects a low logic level of the start bit, it prepares to receive the data. It waits until the middle of the first bit time and shifts the value on the signal into the receiving shift register. It then waits for further successive bit times, shifting each bit into the shift register. The complete data is available after the last bit is received. The receiver uses the stop-bit time to return to the idle state.
- Note that the transmitter and the receiver must agree on the duration of the bit times on the signal. Usually, this is fixed in advance, either during manufacture or by programming. The transmitter and receiver typically have independent clocks, each several times faster than the serial bit rate. The sender uses its clock to transmit the data, and the receiver uses its clock to determine when to sense the data, synchronized by occurrence of the start bit.
- Historically, computer component manufacturers provided a component called a *universal asynchronous receiver/transmitter*, or *UART*, for serial communications ports. The software on the computer could program the bit rate and other parameters. UARTs are still useful in some applications for connecting remote devices to digital systems via serial communications links. For example, an instrumentation system with remote sensors that transmit data at relatively low bit rates can use serial transmission managed by UARTs.
- The third scheme for synchronizing a serial transmitter and receiver involves combining a clock with the data on the same signal wire. This avoids the need for tight clock synchronization, since there is an indication of when each bit arrives. As an example of such a scheme, is the *Manchester encoding*. As with NRZ transmission, Manchester encoding transmits each bit of data in a given interval. However, rather than representing each bit using one or other logic level, it represents a 0 with a transition from low to high in the middle of the bit interval, and a 1 with a transition from high to low. (We could equally well choose the opposite assignment of transmissions, so long as transmitter and receiver agree.) At the beginning of the bit interval, a transition may be necessary to set the signal to the right logic level for the transition in the middle of the interval. Manchester encoding of the value 11100100 is shown in Figure, with the most significant bit transmitted first and with bit intervals defined by the transmitter's clock.
 
- Since Manchester encoding of data is synchronized with the transmitter's clock and that clock is combined with the data, the receiver must be able to recover the transmitted clock and data from the signal. It does so using a circuit called a *phase-locked loop* (PLL), which is an oscillator whose phase can be adjusted to line up with a reference clock signal. A system using Manchester encoding usually transmits a continuous sequence of encoded 1 bits before transmitting one or more data words. The encoding of such a sequence gives a signal that matches the transmitter's clock. The receiver's PLL locks onto the signal to give a clock that can be used to determine the bit intervals for the transmitted data. This is shown in Figure.



- The main advantage of Manchester encoding over NRZ transmission is that it contains sufficient transitions to allow clock synchronization without the need for separate signal wires. The disadvantage is that the bandwidth of the transmission is double that of NRZ transmission. However, for many applications, that is not an overriding disadvantage. Manchester encoding has been used in numerous serial transmission standards, including the original Ethernet standard. Other serial encoding schemes that are similar in concept but more involved are now becoming widely used.

5.2) SERIAL INTERFACE STANDARDS

Given the advantages of serial transmission over parallel transmission for applications where distance and cost are significant considerations, numerous standards have been developed. These standards cover two broad areas of serial interfaces: connection of I/O devices to computers, and connection of computers together to form a network. Since most digital systems contain embedded computers, they can include standard interfaces for connecting components. The benefits of doing so include avoiding the need to design the connection from scratch, and being able to use off-the-shelf devices that adhere to standards. As a consequence, we can reduce the cost of developing and building systems, as well reducing the risk of designs not meeting requirements. Some examples of serial interface standards for connecting I/O devices include:

RS-232: This standard was originally defined in the 1960s for connecting teletype computer terminals with modems, devices for serial communication with remote computers via phone lines. Subsequently, the standard was adopted for direct connection of terminals to computers. Since most computers included RS232 connection ports, RS232 connections were incorporated in I/O devices other than terminals as a convenient way to connect to computers.

Examples included user-interface devices such as mice, and various measurement devices. Serial transmission in RS232 interfaces uses NRZ encoding with start and stop bits for synchronization. Data is usually transmitted with the least significant bit first and most significant bit last.

I2C: The Inter-Integrated Circuit bus specification is defined by Philips Semiconductors, and is widely adopted. It specifies a serial bus protocol for low-bandwidth transmission between chips in a system (10kbit/sec to 3.4Mbit/sec, depending on the mode of operation). It requires two signals, one for NRZ-coded serial data and the other for a clock. The signals are driven by open-drain drivers, allowing any of the chips connected to the bus to take charge by driving the clock and data signals. The specification defines particular sequences of logic levels to be driven on the signals to arbitrate to see which device takes charge and to perform various bus operations. The advantage of the I2C bus is its simplicity and low implementation cost in applications that do not have high performance requirements. It is used in many off-the-shelf consumer and industrial control chips as the means for an embedded microcontroller to control operation of the chip. Philips Semiconductor has also

developed a related bus specification, I2S, or Inter-IC Sound, for serial transmission of digitally encoded audio signals between chips, for example, within a CD player.

USB: The Universal Serial Bus is specified by the USB Implementers Forum, Inc., a non profit consortium of companies founded by the original developers of the bus specification. USB has become commonplace for connecting I/O devices to computers. It uses differential signaling on a pair of wires, with a modified form of NRZ encoding. Different configurations support serial transfer at 1.5Mbit/sec, 12Mbit/sec or 480Mbit/sec. The USB specification defines a rich set of features for devices to communicate with host controllers. Since there is such a diversity of devices with USB interfaces, application-specific digital systems can benefit from inclusion of a USB host controller to enable connection of off-the shelf devices. USB interface designs for inclusion in ASIC and FPGA designs are available in component libraries from vendors.

FireWire: This is another high-speed bus defined by IEEE Standard 1394. Whereas USB was originally developed for lower bandwidth devices and subsequently revised to provide higher bandwidth, FireWire started out as a high-speed (400Mbit/sec) bus. There is also a revision of the standard defining transfer at rates up to 3.2Gbit/sec.

FireWire connections use two differential signalling pairs, one for data and the other for synchronization. As with USB, there is a rich set of bus operations that can be performed to transmit information among devices on the bus. FireWire assumes that any device connected to the bus can take charge of operation, whereas USB requires a single host controller. Thus, there are some differences in the operations provided by FireWire and USB, and some differences in the applications for which they are suitable. FireWire has been most successful in applications requiring high-speed transfer of bulk data, for example, digital video streams from cameras.

6) I / O SOFTWARE

I/O devices interact with the real physical world, and the embedded software needs to be able to respond to events when they occur, or to cause events at the right time. Dealing with *real time* behavior is one of the main differences between embedded software and programs for general purpose computers. Embedded software needs to be able to detect when events occur so that it can react. It also needs to be able to keep track of time so that it can perform actions at specific times or at regular intervals.

6.1) POLLING

- The simplest I/O synchronization mechanism is called *polling*. It involves the software repeatedly checking a status input from a controller to see if an event has occurred. If it has, the software performs the necessary task to react to the event.
- If there are multiple controllers, or multiple events to which the software must respond, the software checks each of the status inputs in turn, reacting to events as they occur, as part of a busy loop.
- Polling has the advantage that it is very simple to implement, and requires no additional circuitry beyond the input and output registers of the I/O controllers.
- However, it requires that the processor core be continually active, consuming power even when there is no event to react to.

- It also prevents the processor from reacting immediately to one event if it is busy dealing with another event. For these reasons, polling is usually only used in very simple control applications where there is no need for fast reaction times.

6.2) INTERRUPTS

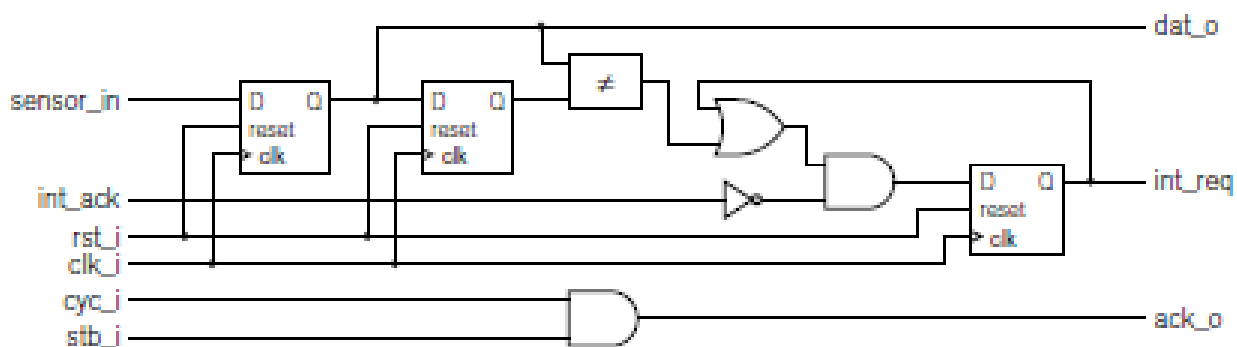
- The most common way to synchronize embedded software with I/O events is through use of *interrupts*. The processor executes some background tasks, and when an event occurs, the I/O controller that detects the event interrupts the processor. The processor then stops what it was doing, saves the program counter so that it can resume later, and starts executing an *interrupt handler*, or *interrupt service routine*, to respond to the event.
- When it has completed the handler, it restores the saved program counter and resumes the interrupted program. In some systems, if there is no background task to run, the processor may enter a low-power standby state from which it emerges in response to an interrupt. This has the benefit of avoiding power consumption due to busy-waiting, though it may add delay to the interrupt response time if the processor requires some time to resume full-power operation.
- Different processors provide different mechanisms for I/O controllers to request an interrupt. Some provide mechanisms, allowing different controllers to be assigned different priorities, so that a higher-priority event can interrupt service of a lower-priority event, but not *vice versa*. Some provide a way for the controller to select the interrupt handler to be executed by the processor. The aspects that are common to most systems are:
- First, the processor must have an input signal to which controllers can connect to request interrupts.
- Second, the processor must be able to prevent interruption while it is executing certain sequences of instructions, often called *critical regions*. Hence processors generally have instructions or means of *disabling interrupts* and *enabling interrupts*.
- Third, the processor must be able to save sufficient information about the program it was executing when interrupted so that it can resume the program on completion of the interrupt handler. This includes saving the program counter value. Since the processor responds to an interrupt after completing one instruction and before starting the next, the program counter contains the address of the next instruction in the program. That is the instruction to be resumed after the interrupt handler. The processor must provide a register or some other storage in which to save the program counter. If there is other state information in the processor that might be modified by the interrupt handler, such as condition code bits, they must also be saved and restored.
- Fourth, when the processor responds to an interrupt, it must disable further interrupts. Since response to an interrupt involves saving the interrupted program's state in registers, if the interrupt handler is itself interrupted, the saved state would be overwritten. Thus, the handler needs to prevent interruption, at least during the initial stages of responding to an interrupt.
- Some processors allow the storage containing the saved state information to be read by a program. That allows a handler to copy the saved state into memory. The handler can then re-enable interrupts, allowing the interrupt handler itself to be interrupted to deal with another event. We call this *nested interrupt* handling. The handler must disable interrupts again when it has

completed its operation so that it can restore the saved state before resuming the interrupted program.

- Fifth, the processor must be able to locate the first instruction of the interrupt handler. This involves the interrupting controller to provide a *vector*: either a value used to form the address of the handler, or an index into a table of addresses in memory.
- Finally, the processor needs an instruction for the interrupt handler to return to the interrupted program. Such a return from interrupt instruction restores the saved program counter and any other saved state.
- The requesting controller must keep the request signal active, otherwise the request may go unnoticed. Failure to respond to an event may be a critical error in some systems. Processors typically have a mechanism to *acknowledge* an interrupt request, that is, to indicate that the event has been noticed and that the interrupt handler has been activated. If there are multiple I/O controllers that can request interrupts, the processor needs to acknowledge each request individually, so that none are overlooked. Once a request has been acknowledged, the controller must deactivate the interrupt request signal. Otherwise, multiple responses might occur for the one event

PROBLEM: Design an input controller that has 8-bit binary-coded input from a sensor. The value can be read from an 8-bit input register. The controller should interrupt the embedded Gumnut core when the input value changes. The controller is the only interrupt source in the system.

Solution: The controller contains a register for the input value. Since we need to detect changes in the value, we also need a register for the previous value, that is, the value on the previous clock cycle. When the current and previous values change, we set an interrupt-request state bit. Since there is only one interrupt source, we can use the `int_ack` signal from the processor core to clear the state bit. The controller circuit is shown in Figure



```
Module sensor_controller ( input clk_i, rst_i,
                          input cyc_i, stb_i,
                          output ack_o,
                          output reg [7:0] dat_o,
                          output reg int_req,
                          input int_ack,
                          input [7:0] sensor_in );
```

```
reg [7:0] prev_data;
```

```
always @(posedge clk_i) // Data registers
if (rst_i)
    begin
        prev_data <= 8'b0;
        dat_o <= 8'b0;
    end
else
    begin
        prev_data <= dat_o;
        dat_o <= sensor_in;
    end
always @(posedge clk_i) // Interrupt state
if (rst_i)
    int_req <= 1'b0;
else
    case (int_req)
    1'b0: if (dat_o != prev_data)
        int_req <= 1'b1;
    1'b1: if (int_ack)
        int_req <= 1'b0;
    endcase
assign ack_o = cyc_i & stb_i;
endmodule
```

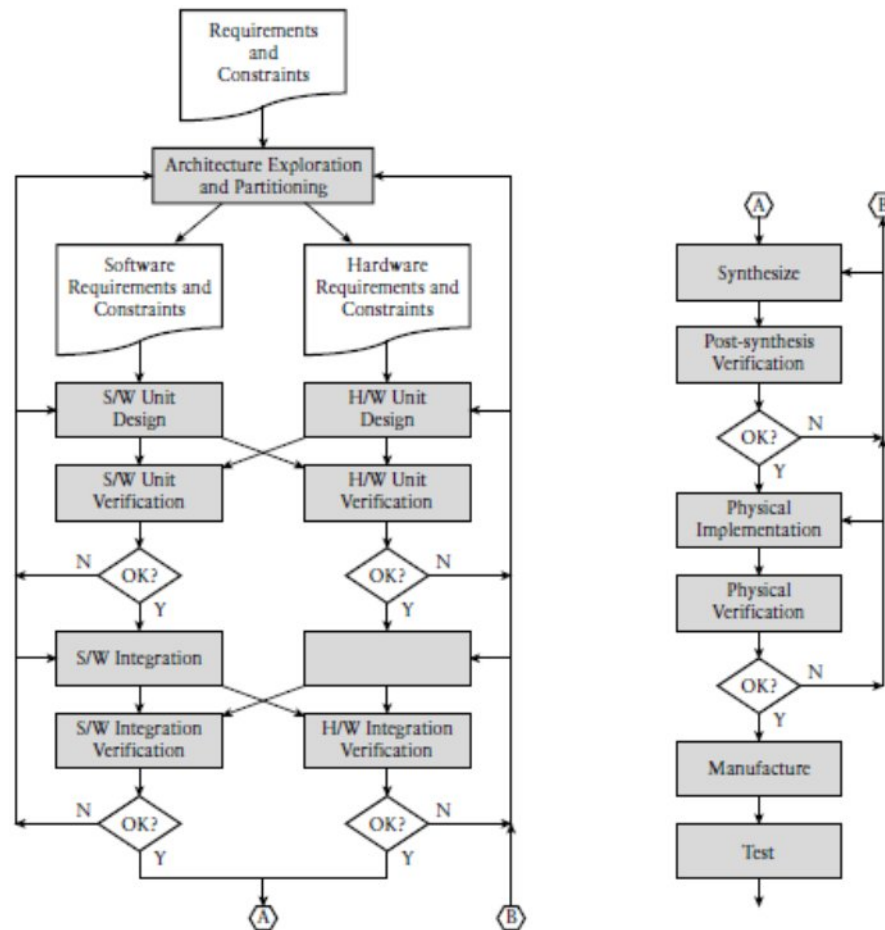
SVIT

Module 5: DESIGN METHODOLOGY

Syllabus: Design flow, Design optimization, Design for test, Nontechnical Issues

1) DESIGN FLOW

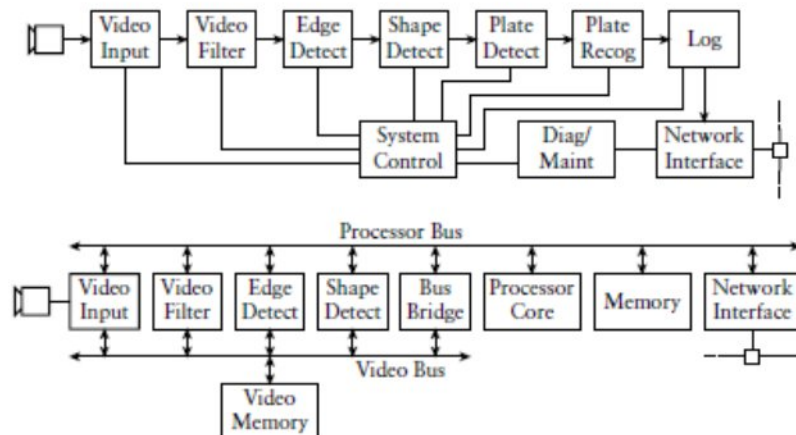
Figure below shows the elements of the design flow, including hierarchical hardware/software co design, integrated into a single diagram.



1.1 ARCHITECTURE EXPLORATION:

- A digital system is designed and manufactured to meet some functional requirements, subject to various constraints. The term architecture exploration, or design space exploration, refers to the task of abstract modeling and analysis of designs.
- One important aspect of architecture exploration is partitioning of operations among components of a system. Partitioning is the application of a divide-and-conquer problem-solving strategy. If our system requirements involve a number of processing steps, we can divide our system into a number of components, each of which performs one of the processing steps.
- The components interact with one another to complete the overall task of the system. When working at the abstract level of architecture exploration, the components need not be physical parts of the system. Instead, it is a logical partitioning, that is identifying parts of the system that will implement the various processing steps. This form of partitioning is also called functional decomposition.
- The physical partitions can include processor cores, accelerators, memories and I/O controllers

- As an example of system partitioning, consider a road transport monitoring system that checks whether goods trucks drive from one part of the country to another in too short an interval.
- Stations on freeways each have a video camera on a gantry over the road. The video images are analyzed to identify the license plate of each truck passing underneath, and the time and license number are logged. The information is transmitted to a central facility for recording and comparison with information from other stations.
- A functional decomposition of the monitoring station is shown at the top of Figure.



- It includes logical components for input of video from a camera, filtering to remove noise, edge-detection, shape detection, license plate detection, character recognition to identify the license number, logging, network interface, system control, and diagnostic and maintenance tasks.
- This logical structure can be mapped onto the physical structure shown at the bottom of Figure. In this case, the physical components comprise an embedded system with accelerators for video processing up to the shape-detection stage.
- License plate detection and recognition, logging, system control, and diagnostics and maintenance tasks are mapped onto software tasks running on the processor core.
- Architecture exploration and partitioning is often done by expert system designers. Decisions made in this early stage of the design flow have a major impact on the rest of the design. Unfortunately, it is very difficult to automate these tasks using EDA tools. Instead, system designers often rely on ad hoc system models, expressing algorithms in programming languages such as C or C++, and using spreadsheets and mathematical modeling tools to analyze system properties.
- The most valuable asset in this stage is the experience of the system designer. Lessons learned from previous projects can be brought to bear on new design projects.
- However architecture exploration design is done, whether by systematic or ad hoc means, the result is a high-level specification of the system.
- For each of the components in the system, the specification describes the function it is to perform, the connections to other components, and the constraints upon its implementation. The specification might be expressed in a language that can be executed or simulated, such as certain forms of the Unified Modeling Language (UML). The specification is used as the input to the next stage of the design flow.

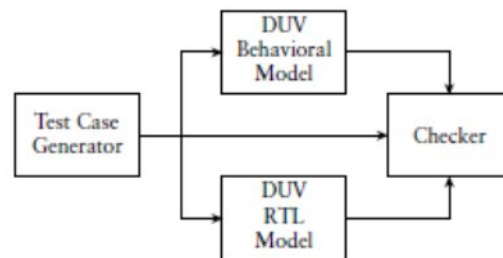
1.2 FUNCTIONAL DESIGN

- Architecture specification has decomposed the system into physical components, each of which must implement one or more logical partition.
- Architectural design is the top level of a top-down design process. We can decompose each component into subcomponents, which we then design and verify as units.
- A behavioral model of the component, expressing its functionality at an intermediate level of abstraction between system level and register transfer level (RTL). The behavioral model might include a description of the algorithm to be implemented by the component without detailed cycle-by-cycle timing, or it might just be a bus functional model.
- The purpose of the behavioral model is to allow function verification of the component before proceeding to detailed implementation.
- One approach to design is develop a new implementation by refining the higher-level model, An alternative, however, is to reuse a component from a previous system, from a library of components, or from a component vendor. The term intellectual property, or IP, refers to such reusable components, since they constitute a valuable intangible resource.
- The benefit of reuse is the saving in design time it affords. Moreover, if the IP has been specifically developed for reuse and has been thoroughly verified as a unit, then we can save effort in verification.
- Even if an IP block does not exactly meet the requirements for our system, it can be made to adapt with less effort than would be required for a fresh start.
- If the IP performs the required function, but does not have quite the right interface connections or timing, we might be able to embed it in a wrapper, circuitry that deals with the differences.
- Another alternative for implementing a component may be to use a core generator, which is an EDA tool that generates a model of a component based on parameters that describe its function. Core generators are available for common kinds of functions, such as memories, arithmetic units, bus interfaces, digital signal processing, and finite-state machines.

1.3 FUNCTIONAL VERIFICATION

- Successful verification of a system requires a verification plan that identifies what parts of the design will be verified, the functionality that will be verified, and how verification will be performed.
- The first question, what parts to verify, can be answered by appealing to the hierarchical decomposition of the system. Since the system is composed of subsystems, each subsystem must be correct for the entire system to be correct. Thus, verifying each subsystem can be considered to be a prerequisite for verifying the entire system.
- The second question, what functionality to verify, can be answered by appealing to the specification for each component. At the lower levels of the design hierarchy, the functionality of each component is relatively simple, and so the component can be verified fairly completely. At higher levels of the design hierarchy, the functionality of subsystems and the complete system gets much more complex. Thus, it is much harder to verify that a subsystem or the system meets functional requirements under all circumstances. Instead, we might focus on the interactions among components, for example, checking for adherence to protocols.
- The term coverage refers to the proportion of functionality that is verified.
 - Code coverage refers to the proportion of lines of code that have been executed at least once during simulation of the design. The benefit of using code coverage is that it is easy to measure, but it does not give a reliable indication that all of the required functionality has been implemented and implemented correctly.
 - Functional coverage, include the distinct operations that have been verified, the range of data values that have been applied, the proportion of states of registers and state machines that have been visited, and the sequences of operations and values that have been applied.

- The third question in the verification plan is how to verify. There are a number of techniques that can be applied.
 - Directed testing involves identifying particular testcases to apply to the DUV and checking the output for each test case. This approach is very effective for simpler components where there are only a small number of categories of stimulus. However, for more complex components, achieving significant function coverage is not feasible, and so we must complement directed testing with other techniques.
 - Constrained random testing involves a test case generator randomly generating input data, subject to constraints on the ranges of values allowed for the inputs. Specialized verification languages, such as Vera and e, include features for specifying constraints and random generation of data values to be used as stimulus to a DUV. Both directed and constrained random testing require checkers that ensure that the DUV produces the correct outputs for each applied test case.
- If, as part of our top-down design process, we have developed a behavioral model of a component, a checker can be used for the register-transfer level implementation. A comparison testbench, illustrated in Figure, that verifies that the implementation has the same functionality as the behavioral model. We use the same test-case generator to provide test cases to two instances of the design under verification: one an instance of the behavioral model, and the other an instance of the RTL implementation. The checker then compares the outputs of the two instances, making any necessary adjustments for timing differences.



- Formal verification, on the other hand, allows complete verification that a component meets a specification. The specification is embodied in one or more asserted properties, expressed in a property specification language, such as PSL which can be used as an adjunct to a hardware description language such as VHDL or Verilog.
- A formal verification tool performs state-space exploration to verify the asserted properties. It exhaustively examines all possible sequences of input values, determines the resulting values for signals in the design at all clock cycles, and checks that the asserted properties hold for all cycles. Where a property does not hold, the tool uses the sequence of input values to construct a counter-example leading to the failure. Properties can also be used to express assumptions about the inputs to a component, which help the formal verification tool limit the space of possible values that it has to explore.
- The strength of formal verification is that it provides a rigorous proof that the assertions hold. However, its completeness is only as good as the properties that are verified. If those properties do not cover all of the functional requirements, then a formal verification does not achieve complete functional coverage.

Hardware/Software Co-Verification

- In an embedded system, much of the system's functionality may be implemented in software that interacts with hardware. In order to verify functionality of the system, we need to verify the software and its interaction with the hardware. Hardware models of the processor and the instruction and data memories, could verify the software by simulating its execution on the hardware models. The instructions are loaded into the instruction memory and start the simulation. The operation of the processor, fetching and executing instructions and reading and writing I/O controller and accelerator registers would then be simulated. The problem with this approach is that it is very slow, since simulation of each processor instruction involves simulation of much of the detail of hardware operation.
- There are approaches to make hardware/software co-verification much faster. One approach recognizes that software and hardware development are usually done by different people. Allowing the software development team to start software verification as early in the design process as possible reduces the overall time to complete the system.
- The key to this is to divide the software into two layers: a lower layer that depends on the hardware, and an application layer that is insulated from the hardware by the lower layer. The lower layer, sometimes called the hardware abstraction layer (HAL), or the board support package (BSP) for processors that are components on a printed circuit board, contains driver code and interrupt service routines for I/O controllers, memory management code, and so on.
- It provides an abstract interface that can be called by the application layer. With this division of the embedded software, development and verification of the application layer code can proceed without waiting for the hardware design. Instead, a software verification tool can emulate the operations provided by the hardware abstraction layer.
- Once the hardware design team has developed models of the hardware that interacts with the embedded software, we can perform cosimulation of the hardware and software. This usually involves collaboration between an ISS and a simulator for the hardware model. The two simulators run concurrently, communicating when the processor performs bus read and write operations. Initially, at least, the hardware models need not be fully functional behavioral or RTL models.

1.4 SYNTHESIS

- Synthesis is the refinement of the functional design to a gate-level net list. For most designs, synthesis can be performed largely automatically using an RTL synthesis tool.
- RTL synthesis starts with models of the design refined to the register transfer level.
- Early synthesis tools performed relatively simple pattern recognition on the HDL source code to determine which hardware circuits were implied.
- Different synthesis tools accept different subsets of the input language, and hence it can be difficult to develop RTL models that are portable across a range of tools.
- A synthesis tool starts by analyzing the model, checking to make sure the code conforms to its style requirements.
- It also performs some design rule checks, such as checking for unconnected outputs, undriven inputs, and multiple drivers on nonresolved signals. The tool then infers hardware constructs for the model. This involves things like:
- Analyzing wire and variable declarations to determine the encoding and the number of bits required to represent the data.
- Analyzing expressions and assignments to identify combinational circuit elements, such as adders and multiplexers, and to identify the input, output and intermediate signal connections.
- Analyzing always blocks to identify the clock and control signals, and to select the appropriate kinds of flip-flops and registers to use. For each of these inferred hardware elements, the synthesis

tool determines an implementation using primitive circuit elements selected from a technology library.

- The process of translating the design into a circuit of library components is guided by synthesis constraints that we specify. Such constraints include bounds on clock periods and propagation delays.
- The synthesis tool uses the constraints to choose among alternative implementations. For example, two alternative circuit structures might implement the required functionality inferred from the RTL code: one with fewer gates connected in a deeper chain, and so with greater propagation delay; and the other with more gates connected less deeply, and so with less propagation delay. If our constraints specified minimal overall delay as the synthesis goal, the tool would choose the latter implementation. In making such timing-based choices, the tool uses a simple wire model to determine the wire delays, based on average wire lengths and loading, since at this stage the actual layout and wiring is yet to be done.
- Simulating the synthesized design and making sure it behaves in the same way as the RTL design is a good check that we have used the tools correctly.

1.5 PHYSICAL DESIGN

- The final stage in the design flow is physical design, in which we refine the gate-level design into an arrangement of circuit elements in an ASIC, or build the programming file that configures each element in an FPGA.
- Physical design for ASICs, in its basic form, consists of floor planning, placement, and routing.
 - The first step, floor planning, involves deciding where each of the blocks in the partitioned design is to be located on the chip. There are a number of factors that influence the floor plan.
 - Blocks that have a large number of connections between them should be placed near each other, since that reduces wire length and wiring congestion.
 - Similarly, blocks that are connected to external pins should be placed near the edge of the chip. The position of those blocks also determines the allocation and positioning of pins for external connections.
 - The blocks should be arranged to make the chip as close to square as possible, since that influences the size of the package that can be used. Square chips are easier to package than rectangular chips.
 - Floorplanning also involves the arrangement of power supply and ground pins and internal connections, and, importantly, the connection and distribution of clock signals across the chip.
 - Finally, floorplanning also involves provision of channels for laying out interconnections between blocks.
 - Devising a good floorplan for an ASIC can be quite challenging. EDA tools can assist by providing graphical tools to help us visualize floorplans and rearrange blocks, ensuring all the time that a floorplan is feasible, and by analyzing alternative floorplans to determine figures of merit. Having determined a floorplan for an ASIC, we then proceed to placement and routing.
 - Placement and Routing: This step involves positioning each cell in a synthesized design (placement) and finding a path for each connection (routing).
 - The main goals are to position all cells and route all connections while minimizing area and delay of critical signals. The result of placement and routing is a suite of files to send to the chip foundry for fabrication.
 - We can also generate detailed timing information, based on the actual positions of components and wires, and use this in a more accurate simulation model of the gate-

level design. This detailed timing simulation is a final check that our design meets its timing constraints. In contrast, physical design for FPGAs involves deciding how to implement the synthesized design using the programmable resources of a prefabricated chip.

2) DESIGN OPTIMIZATION

In most design projects, we need to perform optimization of the design, making trade-offs of one property against another. If during some stage of the design flow there is no feasible optimization, we need to revisit earlier stages to revise design choices previously made.

Thus, realistic design flows are not linear, starting with design concept and leading directly to final implementation. Instead, they are cyclic, with the design evolving as more “back-end” implementation detail informs design choices made in “front-end” stages.

The three main properties of a design that are constrained and that need to be optimized are: area, timing, and power. At each stage of the design flow, decisions can be made that affect these properties. Decisions early in the flow, starting with architecture exploration and partitioning, generally have the greatest impact.

If fine-tuning is insufficient, then we need to revisit earlier stages to make more substantial changes.

2. 1) AREA OPTIMIZATION

- The area of a circuit is a significant determinant of cost. The cost of fabricating circuits on a wafer is distributed among the chips on that wafer. Larger chips thus take on a larger share of the wafer fabrication cost. Chips are rectangular and wafers are circular, larger chips leave more wasted area near the edges of the wafer, so the proportion of wafer cost borne by each chip is not just the ratio of chip area to wafer area. Instead, the relationship is nonlinear.
- A further nonlinearity arises from the fact that larger area increases the chance of a defect occurring on any given chip and causing the chip not to function. Since non-functional chips must be discarded after the wafer is fabricated, the remaining functional chips must bear the cost of fabricating and testing the non-functional chips.
- This leads to final chip cost being approximately proportional to the square of the chip area.
- Next, the chip must be packaged, a larger chip requires a larger and more costly package than a smaller chip.
- Also, since the chip is larger, it has more transistors than a smaller chip, it consumes more power, and so the resulting heat must be dissipated. These effects also lead to package costs that are nonlinearly related to area.
- Therefore, we can only affect cost indirectly through managing the area of our design.
- One of the goals of floorplanning is to find an arrangement of blocks that minimizes area. We can also do some floorplanning, at a preliminary level, as part of the partitioning step of architecture exploration.
- At the partitioning step, we can also estimate the number of pins that will be required for the chip, since that will influence the floorplan. In particular, if the pin count is large, the area required for the pad ring may constrain the overall area of the chip, leading to consideration of alternative architectures with reduced pin counts.
- By making these decisions early in the project, we can avoid wasting time on a design that we subsequently discover cannot be made to fit well on a chip.
- In the functional design stage of the design flow, we can influence circuit area through our choice of components, whether explicitly instantiated or implied by RTL model code.

Ex: different forms of adders and multipliers have differing circuit complexity and hence circuit area, traded off against propagation delay or cycle count for performing an operation.

Also, choosing minimal bit-widths for data, helps to keep circuit area to a minimum, since components that process the data can then be of the minimal size, and the minimal amount of wiring is required between the components.

- In the synthesis stage, we can influence the circuit area by specifying constraints to the synthesis tool. We can direct the tool to use a synthesis strategy that minimizes area instead of delay, or to use additional effort to optimize the design instead of reducing turnaround. In the case of hierarchically structured designs, we can direct the tool to try to optimize across block boundaries, possibly combining components from different blocks in order to reduce area. In cases where a tool does not automatically infer use of special resources within an implementation fabric, such as RAMs or ROMs, we might provide hints that specific parts of a design be implemented using specific resources.
- Finally, in the physical design stage, we can influence circuit area through intervention in the floorplanning, placement and routing of the circuit. At this level, we can just fine tune. We cannot change the number or kind of components used or the amount or connectivity of the wiring between them. Thus decisions made earlier in the flow have more significant impact.

2.2) TIMING OPTIMIZATION

- The aim of timing optimization is to ensure that a design meets performance constraints. Performance and timing are essentially the inverses of each other.
- Performance is in terms of the number of operations completed per unit time. The inverse of this is the time taken to complete an operation. Our goal is to maximize the number of operations per second, or, conversely, to minimize the time per operation.
- In the architecture exploration stage of the design flow, we have the greatest impact on performance through application of parallelism.
- Increasing parallelism is in conflict with minimizing area and power, since the extra resources required to realize the parallelism take up area and consume power.
- Thus we aim to find a circuit for the block that performs the required operation in the least amount of time, consistent with our other constraints.
- We need to make estimates of the clock frequency that can be achieved. The clock period constrains the propagation delay on combinational paths through the register-transfer-level circuit. That includes paths from block inputs through combination logic to register inputs, and paths from register outputs through combinational logic to block outputs. If blocks are designed by separate designers, we must ensure that the combined path from a register output in one block to a register input in another block meets the timing constraint.
- One way to do this is to allocate a timing budget to each block, specifying maximum clock-to-output delays and input-to-clock setup times for each block. Any deviation from the budget must be specifically agreed between designers, documented, and carefully verified.
- In the functional design stage of the design flow, we can influence timing through our choice of components. This is a similar argument to that for influencing area in the functional design stage, except that the two objectives are in conflict with each other (less area more time and vice-versa).
- Directives and hints are used to a synthesis tool to optimize timing of the detailed design, and then analyze the resulting synthesized circuit to verify if the timing constraints are met. If they are not, we might revise the directives and hints and resynthesize. If we are unable to meet constraints through this iterative process, we need to revisit earlier stages of the design flow and make larger changes to the design at higher levels of abstraction.
- Analyzing the synthesized design is typically done with a static timing analysis tool. The tool uses timing estimates for each of the components in the technology library, together with simple

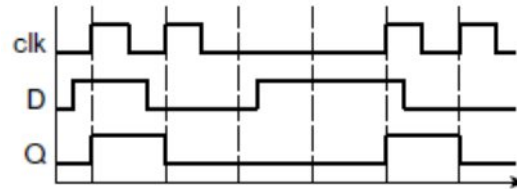
wire-load models. Since the design has not been placed and routed at this stage, the delays due to the lengths of wires can only be estimated.

- Moreover, the actual propagation delay of each library component and the load on each wire may vary as a result of mapping the design to technology-specific components. However, the estimates used are sufficient to guide optimization at this stage.
- The static timing analysis tool aggregates the delay through combinational circuits and wiring between successive registers, it thus identifies the critical path in the design and determines whether the clock period constraint is met.
- In the physical design stage, we can fine tune timing by choice of placement of components and wires. However, since this process is very computationally intensive, it is unlikely that we could do a better job manually than an EDA tool can do automatically.
- Once the physical design has been performed, it is possible to extract accurate delay values for components and wiring. We can then repeat the static timing analysis using these values to verify whether timing constraints have been met. If they have not, we need to revisit earlier stages of the design flow to improve the timing of the circuit.

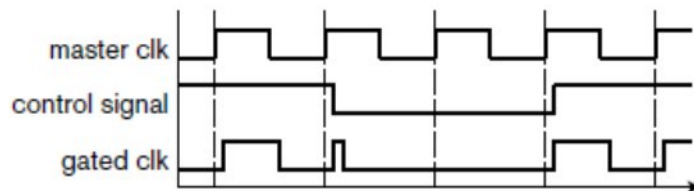
2.3) POWER OPTIMIZATION

- As digital systems have become more complex, power consumption has become a more significant constraint in their design. This is particularly the case for mobile battery-operated devices, such as cell phones, PDAs, and portable media players.
- The amount of power consumed by the circuit directly affects how long the device functions on a single battery charge, or, alternatively, how large a battery is required.
- Electrical power consumed by a circuit is turned into heat, which must be dissipated through the chip and system packaging. Dealing with additional heat dissipation adds cost to a system, so keeping power consumption to a minimum is part of keeping cost down.
- Larger circuits generally contain more transistors, each of which consumes power.
- However, there are other approaches we can consider that focus on power consumption. One such approach is to identify blocks of a system that remain idle for substantial periods during the system's operation, and to remove power from those blocks during idle periods. Some laptop computers take this approach, for example, by powering down a network card when the computer is not connected to a network cable.
- While powering down blocks of a system can reduce average power consumption significantly, it is not simple to implement. In particular, if other parts of the system to which the block is connected must continue operating, then the interface signals must be disabled to avoid spurious activation of the parts that remain active.
- Further, when power is restored to a block, it takes a significant number of clock cycles before the block can resume operation. This delay may affect performance, so the technique is only appropriate where the delay can be tolerated.
- In a clocked synchronous digital system, we have many flip-flops, all of which are controlled by a global clock signal. Each flip-flop contains several transistors that switch state on clock edges, even if the stored data does not change. These transitions consume power without affecting the computation performed by the circuit. If the performance requirements of a system are not constant, that is, if there are periods where high performance is required and other periods where lower performance is acceptable, we can reduce dynamic power consumption by reducing the clock frequency. This requires that the source of the clock signal be adjustable. Often, we would implement power management through clock frequency control within the real-time operating system of an embedded computer. The clock generator in such a system would need to be adjustable under program control.

- Another common way of reducing power in CMOS systems is clock gating. This involves turning off the clock to parts of a circuit whose stored values do not need to change.
- With clock gating, the components see no clock transitions when the clock is turned off, as shown in Figure. Here, the clock is gated off for two cycles. During that interval, the component consumes no dynamic power.



- Gating a clock is not as simple as inserting an AND gate in the clock signal. Given the delay in an AND gate, the resulting clock edges would be skewed from those of the ungated clock, making it difficult to meet timing constraints. Also, since the gating control signal is typically generated by a clocked control section, it can lead to glitches on the gated clock signal, as shown in Figure below.



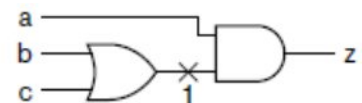
- The glitch may cause unreliable triggering of the components to which the gate clock is connected.
- Further, if the control signal has glitches, for example, due to differing delays in paths through combination logic that generates the signal, those glitches may be passed through to the gated clock. The solution to these problems is not to express the clock gating in the RTL model of the circuit. Rather, we should treat clock gating as a power optimization to be implemented by clock insertion tools during physical design. Several clock synthesis tools can perform such power optimization.

3) DESIGN FOR TEST

- Once the chips have been manufactured, they must be tested to ensure that they work correctly. There are several reasons why manufactured chips might not work, including problems arising during wafer manufacture and during packaging.
- Some problems cause whole batches of chips to fail, whereas others produce isolated failures. The aim of testing after manufacture is to identify the defective chips so that they can be discarded rather than supplied to customers.
- Ideally, the cause and location of faults can be recorded so that the manufacturing process can be adjusted to enhance yield.
- In many systems, faulty PCBs are not discarded. Instead, defective chips on the PCBs are replaced, and the repaired PCB retested and returned to service.
- For a simple circuit, testing involves applying test cases on the chip inputs and verifying that the chip produces the correct outputs.
- As the number of possible input values and input sequences increases, testing that the chip operates correctly in all cases becomes infeasible. The time available for testing is much less than for design verification, since thousands or millions of chips must be tested individually. Furthermore, testing requires use of test equipment to physically apply input values and measure output values. Such equipment is a costly resource, so its use must be minimized.
- We can reduce the time and cost involved in testing a system by including additional circuitry to improve the system's testability. Such circuitry includes elements that make internal nodes observable, or that perform testing automatically as a special mode of system operation.
- Design for test (DFT) refers to design techniques that seek to improve testability.

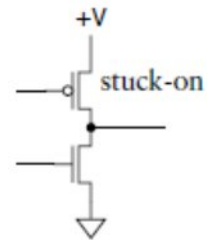
3.1) FAULT MODELS AND FAULT SIMULATION

- As part of the design flow, we need to develop a set of test vectors, or test patterns, that is, combinations of input values for the circuit that can be used to expose faults. The idea is that application of each test vector (or small sequence of test vectors) should cause the circuit to produce a given output. If the circuit produces a different output, the circuit is faulty.
- In order to work out how to expose faults, we need to consider the kinds of faults that can occur in a circuit.
- We use fault models that are abstractions of the effects produced by faults. We use a fault simulator that simulates the operation of the circuit with a given fault injected at a given location. The simulator applies test vectors until an incorrect output results, indicating that the fault has been detected.
- If no incorrect output is produced for all of the test vectors, the fault remains undetected by that set of vectors. The simulator repeats the simulation for other faults and other locations in the circuit. Once all of the faults have been simulated, the fault coverage of the test vectors, that is, the proportion of faults detected can be determined. Ideally, the fault coverage should be 100%, but for a large design, this may not be feasible.
- In choosing the test vectors, we can use an automatic test pattern generator (ATPG), an EDA tool that analyzes a circuit and seeks to create a minimal set of test vectors with as close to full coverage as possible.
- A simple and commonly used fault model is the stuck-at model, in which an input or output of a gate in a circuit can be stuck at 0 or stuck at 1, rather than being able to change between 0 and 1. Such a fault might be caused by a short circuit to the ground or power supply. This is illustrated in Figure aside in which an input to the AND gate is stuck at 1.
- For some input combinations to the circuit ($b = 1$ or $c = 1$), the value



at the stuck node would normally be the same as the stuck-at value; the circuit would produce the correct output, and we would not detect the fault.

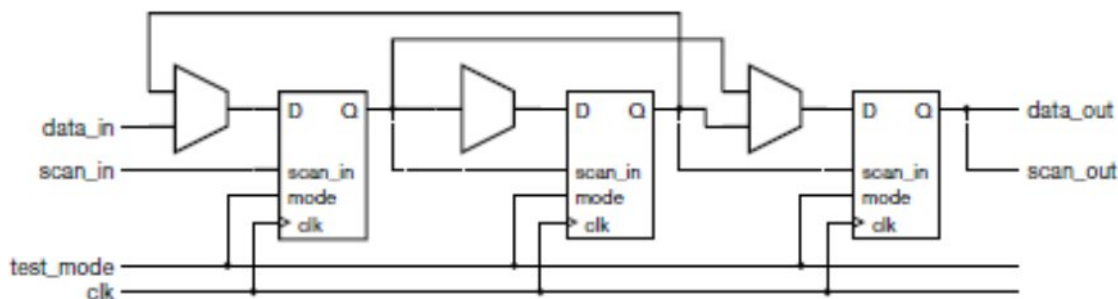
- For other input combinations ($b = 0$ and $c = 0$), the value at the stuck node would be the opposite of the stuck-at value. Whether we could detect the fault would then depend on the remaining logic between the stuck node and the circuit's output. In this circuit, if $a = 0$, the output value is independent of the value at the stuck node, so the fault is masked. However, if $a = 1$, the value of the stuck node is propagated to the output, allowing us to detect the fault.
- In general, detecting the fault involves applying a combination of input values that sensitizes the path from the fault to the output and that drives the stuck node to the opposite of its stuck-at value. A node in a circuit is observable if a fault at the node can be made to result in an incorrect output value. The node is controllable if there are input combinations that cause the node to take on a given value. Observability and controllability of nodes in a circuit determine the testability of the circuit.
- Other fault models consider the transistor-level circuits for gates, and involve transistors being stuck on or stuck off. This allows us to detect faults that are not adequately represented by the stuck-at fault model.
- For example, given the output driving circuit of a gate, shown in Figure aside, a fault might cause the upper transistor to be stuck on.
- When the gate should be driving a high logic level, its output is correct.
- However, when it should be driving a low logic level, both transistors are on. This creates a voltage divider, and the output logic level is an invalid level between the valid high and low levels.
- A testing approach used for such faults is to measure the steady-state current drawn from the power supply (I_{DDQ}) to detect the increase when both transistors are on.
- Further fault models include bridging faults, representing short-circuit connections between signal wires; delay faults, in which the propagation delay of a circuit is longer than normal; and faults in storage elements.



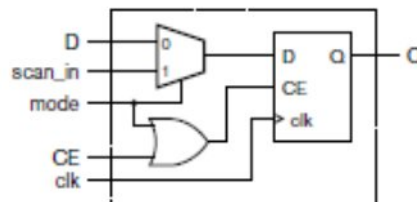
3.2) SCAN DESIGN AND BOUNDARY SCAN

SCAN DESIGN

- The fault models and fault detection techniques work well for combinational circuits, but are difficult to adapt to detect faults in registers and other storage elements.
- The problems are compounded when the registers are buried deep within a datapath, since they are significantly more difficult to control and observe.
- Scan design techniques address this problem by modifying the registers to allow them to be chained into a long shift register, called a scan chain, as shown in Figure below.



- Test vectors can be shifted into the registers in the chain, under control of the test mode input, thus making them controllable.
- Stored values can also be shifted out of the registers, thus making them observable.
- The chain of registers allows us to control and observe the combinational blocks between registers. Each combinational block can be tested separately. This process consists of shifting test values into the register chain until the test vector for each block reaches the input registers for that block.
- We then run the system in its normal operational mode for one clock cycle, clocking the output of each block into the block's output registers.
- We also need to apply test vectors to the external inputs of the system and observe the external outputs of the system, in order to test any combinational input and output circuits.
- Finally, we shift the result values out through the register chain. The test equipment controlling the process compares the output values with the expected results to detect any faults.
- This sequence is repeated until the entire test vectors have been applied to all of the combinational blocks, or until a fault is detected.
- Advantages of this form of design for test are the increased controllability and observability provided. This makes achieving high fault coverage feasible, especially for large circuits.
- The test generation problem can be reduced to by use of ATPG (Automatic Test Pattern Generation) to achieve 100% fault coverage.
- The modification of the registers to allow them to function as shift registers can also be automated. One approach is to design and synthesize the circuit normally, generating a gate-level circuit with flip-flops implementing the registers. Then as a part of physical design, a tool can substitute modified flip-flops that have a shift mode, as shown in Figure below.



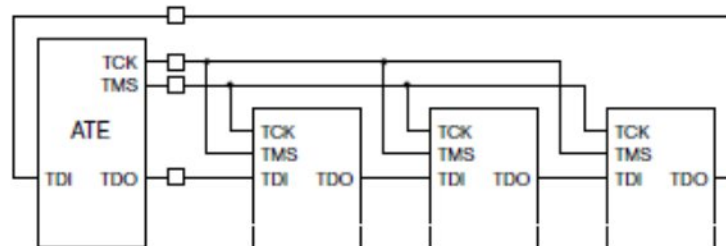
- The main disadvantage of scan design is the overhead, both in circuit area and delay. The modified flip-flops have additional circuitry, including an input multiplexer to select between the normal input and the output of the previous flip-flop in the scan chain.
- The area overhead for scan design has been estimated at between 2% and 10%. The input multiplexer imposes additional delay in the combinational path leading to the flip-flop input.
- Another disadvantage of scan design, when compared to some other DFT techniques, is that the scan chain is very long. Shifting test vectors in and result vectors out takes a large fraction of test time, so the system cannot be tested at full operational speed.
- The overhead time can be reduced by dividing the scan chain into segments that can be shifted in parallel. However, each chain requires separate input and output pins, so we must compromise between test-time overhead and test-pin-count overhead.

BOUNDARY SCAN

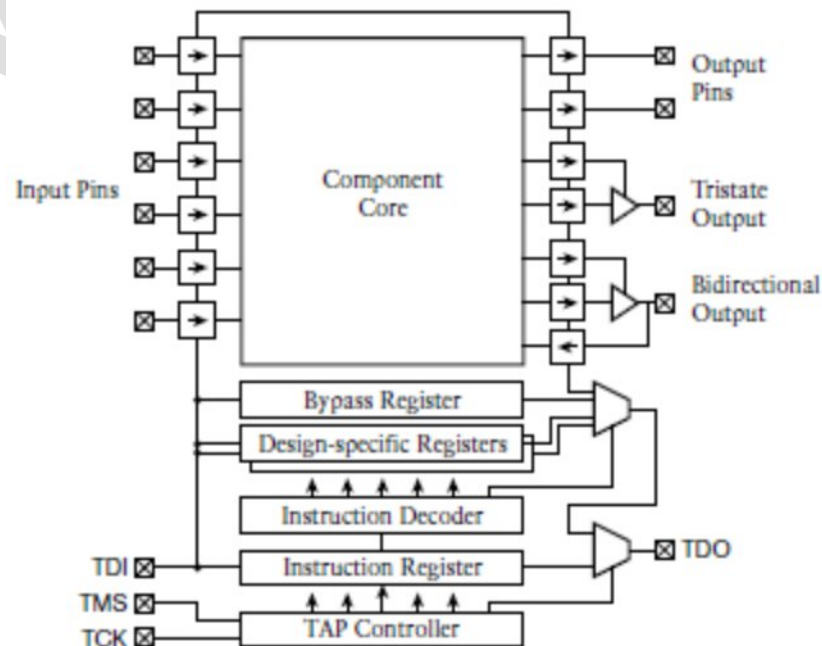
- The concept of scan design can be extended for use in testing the connections between chips on a PCB, leading to a technique called boundary scan. The idea is to include scan-chain flip-flops on the external pins of each chip.

- To test the PCB, the test equipment shifts a test vector into the scan chain. When the chain is loaded, the vector is driven onto the external outputs of the chips. The scan-chain flip-flops then sample the external inputs, and the sampled values are shifted out to the test equipment.
- The test equipment can then verify that all of the connections between the chips, including the chip bonding wires, package pins and PCB traces, are intact.
- Various test vectors can be used to detect different kinds of faults, including broken connections, shorts to power or ground planes, and bridges between connections.
- The success of boundary scan techniques led to the formation of the Joint Test Action Group (JTAG) in the 1980s for standardizing boundary scan components and protocols. Standardization has been managed for some time by the IEEE as IEEE Standard 1149.1.
- The JTAG standard specifies that each component have a test access port (TAP), consisting of the following connections:
 - ✓ Test Clock (TCK): provides the clock signal for the test logic.
 - ✓ Test Mode Select Input (TMS): controls test operation.
 - ✓ Test Data Input (TDI): serial input for test data and instructions.
 - ✓ Test Data Output (TDO): serial output for test data and instructions.

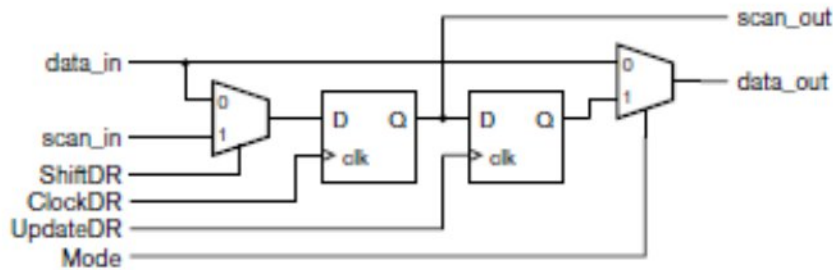
Figure below shows a typical connection of automatic test equipment (ATE) to the TAPs of components on a PCB.



- The test logic within each component is illustrated in the figure aside.
- The TAP controller governs operation of the test logic.
- There are a number of registers for test data and instructions, and a chain of boundary scan cells inserted between external pins and the component core.
- Input and output pins of the component each require just one cell.
- Tristate output pins require two cells: one to control and observe the data, and the other to control and observe the output enable.
- Bidirectional pins require three cells, as they are a combination of a tristate output and an input.
- The TAP Controller operates as a simple finite-state machine, changing between states depending on the value of the TMS input.



- A typical boundary scan cell for an input and output pin is as shown in Figure below.



- Depending on the control inputs to the cell, data can flow straight through, input data can be captured, output data can be driven, and test data can be shifted through.
- Thus the boundary scan technique is a means of testing connections between components on a board, the JTAG boundary scan cells have been designed to allow testing of the component core also. The cells can be configured to isolate the component core's inputs from the package input pins. Test data can be shifted into the cells at the inputs and then driven onto the core's inputs. The core's outputs can be sampled into the cells at the output pins and then shifted out to the ATE.
- Thus, the JTAG architecture solves two problems: in-circuit testing of components in a system, and in-circuit testing of the connections between the components. This flexibility has led to the widespread use of the standard, with EDA tools available for insertion of test logic into designs during various stages of the design flow. The JTAG standard has also been extended to support in-circuit programming of ROMs and configuration of PLDs, including FPGAs.

3. 3) Built In Self Test(BIST)

The Design for Test (DFT) approaches rely on developing test vectors during design of a system and applying the vectors to manufactured components to test them. Scan-design and boundary scan techniques improve testability of components; there is still significant time overhead in shifting test vectors in and results out of each component.

Furthermore, the components cannot be tested at full operating speed, since test vectors for each cycle of system operation must be shifted in over many clock cycles.

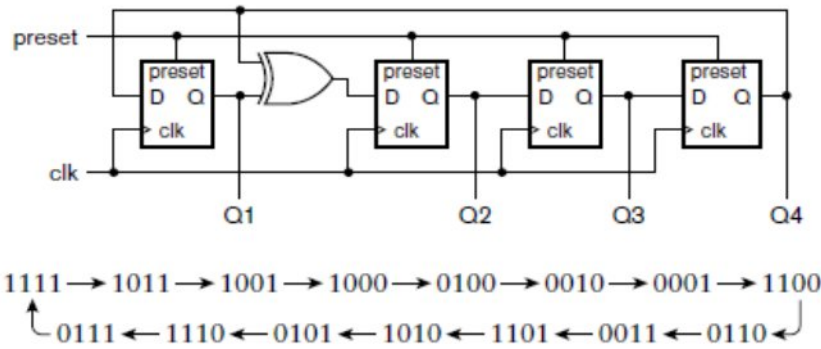
To overcome these problems we use built-in self test (BIST) techniques, which involve adding test circuits that generate test patterns and analyse output responses.

One of the advantages of BIST is that, being embedded in a system, it can generate test vectors at full system speed. This significantly reduces the time taken for test.

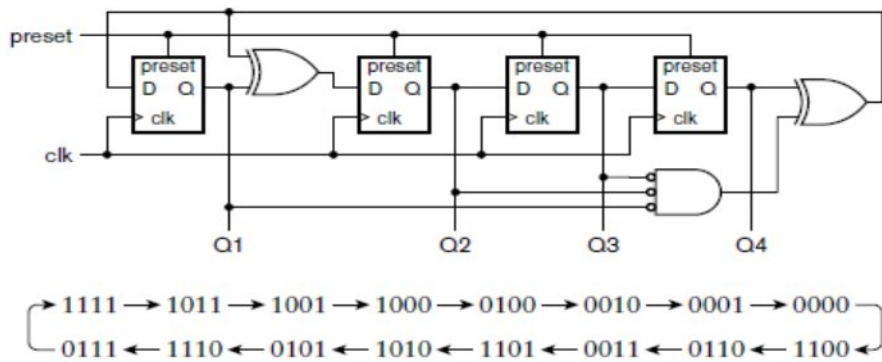
There are two main aspects to consider when designing a BIST implementation: how to generate the test patterns, and how to analyze the output response to determine whether it is correct.

The most common means of generating test patterns is a pseudorandom test pattern generator. Unlike true random sequences, pseudorandom sequences can be repeated from a given starting point, called the seed.

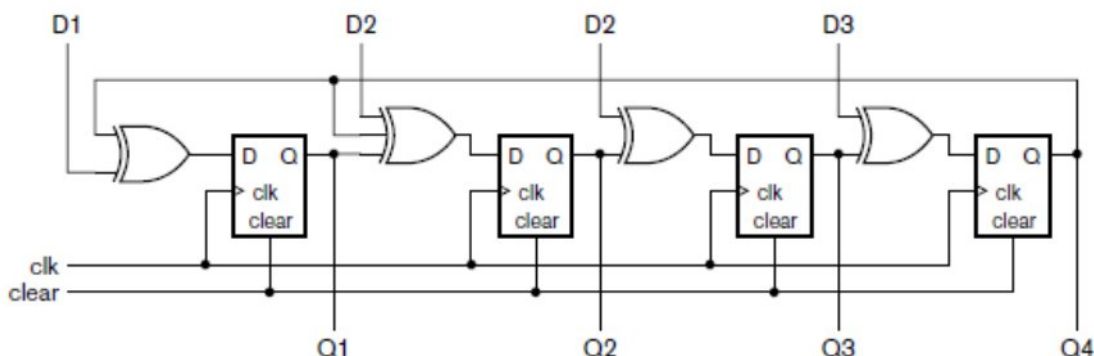
Pseudo-random sequences can be readily generated with a simple hardware structure called a linear-feedback shift register (LFSR) shown in figure which generates sequences of 4-bit values. The sequence is initiated by presetting the flip-flops, generating the test value 1111 as the seed. The sequence contains all possible 4-bit values except 0000.



In most applications, it is desirable to include that value also. Therefore, we modify the LFSR to form a complete feedback shift register (CFSR), as shown in Figure, which generates all possible values.



- Placement of the XOR gates within the LFSR is determined by the characteristic polynomial of the LFSR, referring to the mathematical theory underlying LFSR operation.
- Analyzing the output response of a circuit to the test patterns presents more of a problem. In most cases, it would be infeasible to store the correct output response for comparison with the circuit's output response, since the storage required could well be larger than the circuit under test. Instead, we need to devise a way of compacting the expected output response and the circuit's output response. Doing so requires less storage, and comparison hardware, though at the cost of circuitry to compact the circuit's outputs.
- There are several schemes for output response compaction, but the most commonly used is signature analysis.
- A signature register forms a summary, called a signature, of a sequence of output responses. Two sequences that differ slightly are likely to have different signatures. Figure shows an example of a multiple-input signature register (MISR), with four inputs from a circuit under test and a 4-bit signature.



Use of BIST using an LFSR for test-pattern generation and a signature register for response analysis requires us to perform a logic simulation of the circuit without faults. Since the sequence generated by the LFSR is determined by the seed, we can perform the simulation with that sequence of input values. We use the output values from the simulation to compute the expected signature, and save the signature for use during test.

NON TECHNICAL ISSUES

- Electronics products, like most products, go through life cycles. Product design is just one stage in the life cycle. It is preceded by market research and financial modeling. After design, manufacturing facilities and supply channels, and sales and distribution channels need to be established.
- Depending on the product, there may be a need for maintenance and repair, or for customer service. During the product's lifetime, it may be redesigned to meet changing needs, or may be reused in other products. Finally, the product becomes obsolete and is retired from production and support.
- There are various financial models that can be applied to estimate revenue from a product over its life cycle. Generally, revenue from a product typically peaks early in the product's life cycle, and tails off until obsolescence. The non-recurring engineering (NRE) costs of developing the product, along with other up-front costs, must be met from the revenue stream.
- If the product is aimed at a competitive market, entering the market early has a critical impact on revenue. Late entry allows competitors to gain market share, reducing the revenue available for the late product, and possibly making it unprofitable.
- Hence, time-to-market is an important nontechnical measure for a design project. For many consumer products, such as cell phones and media players, product life cycles are very short, so there is only a short window of opportunity to gain sufficient revenue for profitability. Time-to-market pressures for design of such products are very intense.
- In other industry segments, products have very long life cycles. Examples are military systems and telecommunications infrastructure. For such products, attributes such as reliability and maintainability are important. For example, considerations such as longevity of a vendor company may override technical considerations in choice of components for a system. Such long-lived products must typically be supported throughout their lifetime. Hence, the design phase will involve more than just the technical design of the circuit. It will also involve development of design documentation, and liaison with support service providers to develop support plans, procedures and documents.
- Another important factor to consider in design of digital systems is that the implementation technology continues to evolve rapidly. Each generation period, a new technology generation is likely to be available when the product reaches the manufacturing stage. Designing using the previous generation may well lead to a product with lower performance or capacity than competitors' products. When we start a design project, we must be aware of technology trends and make projections to determine the appropriate technology for the future manufacture of our product.
- For smaller systems, a small team of engineers can feasibly deal with product definition and specification, detailed design, verification, and manufacture. Even so, a systematic methodology reduces the risk of the product development project going off the rails. For larger systems, a larger development team is typically needed. Different team members bring expertise in different areas to the project. Indeed, larger teams are often structured with sub teams being responsible for different aspects of the design methodology, such as architectural definition, detailed design,

verification, test development, and liaison with the manufacturing facility. It is important for individual team members to understand the structure of the overall project and the context in which they are working. In particular, maintaining good communication and information flow within the project is critical. Good project management is essential to a successful outcome.

SVIT