data, while the bulk of the data resides in slower, lower-cost memory devices such as DRAM. (In this context, "active data" refers to data or instructions that a program is currently using, or which the operating system believes the program will want to use in the immediate future.)

In fact, the concepts behind virtual hardware are actually quite easy to understand. Each large macro-function in a device is usually formed by the combination of a number of smaller micro-functions such as counters, shift registers, and multiplexers. Two things become apparent when a group of macro-functions are divided into their respective micro-functions. First, functionality overlaps, and an element such as a counter may be used several times in different places. Second, there is a substantial amount of *functional latency*, which means that, at any given time, only a portion of the micro-functions are active. To put this another way, relatively few micro-functions are in use during any given clock cycle. Thus, the ability to dynamically reconfigure individual portions of a virtual hardware device means that a relatively small amount of logic can be used to implement a number of different macro-functions.

By tracking the occurrence and usage of each micro-function, then consolidating functionality and eliminating redundancy, virtual hardware devices can perform far more complex tasks than they would appear to have logic gates available. For example, in a complex function requiring 100,000 equivalent gates, only 10,000 gates may be active at any one time. Thus, by storing, or caching, the functions implemented by the extra 90,000 gates, a small, inexpensive 10,000-gate device can be used to replace a larger, more expensive 100,000-gate component (Figure 21-5).

In fact, it is even possible to "compile" new design variations in real-time, which may be thought of as dynamically creating subroutines in hardware! Hence the phrase *adaptive hardware* referred to above.

Adaptive Computing Machines (ACMs)

Ah, how I love the smell of freshly minted silicon chips in the morning. So I really wish I'd been present when the folks at QuickSilver Technology (www.qstech.com) unveiled their first *adaptive computing machine (ACM)* test chip in April 2002. This little rapscallion demonstrated a breakthrough performance three times that of a dedicated ASIC, which is no small beans, let me tell you! (Also check out picoChip at www.picochip.com.)

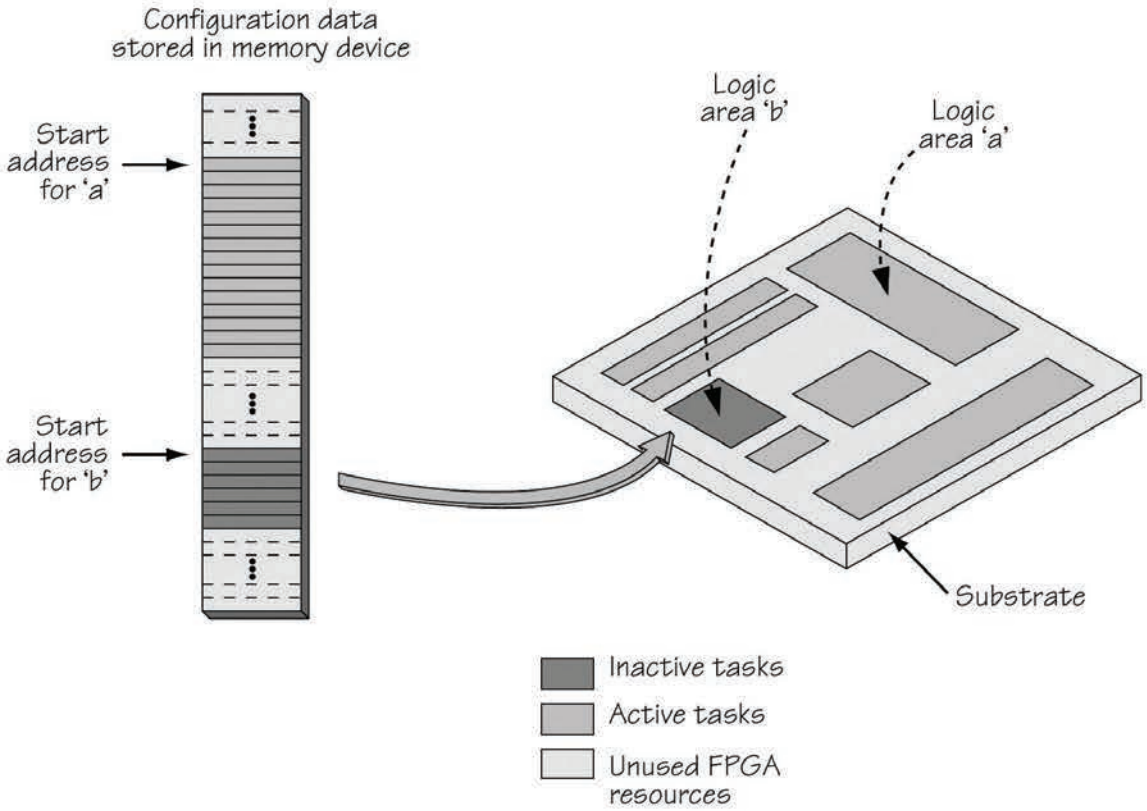


Figure 21-5. Virtual Hardware

Let's start with the fact that a product like a *software defined radio (SDR)* wireless handset typically contains a mixture of ASIC, DSP, and RISC microprocessor material. In order to reduce size and power, these will probably be implemented as a number of cores on a system-on-chip (SoC) device.

The RISC material appears in the form of a general-purpose microprocessor providing high-level, non-compute-intensive control functions. The DSP material is used because it's way faster than the general-purpose micro, and its software can be developed way faster than an ASIC can be designed. The ASIC material is reserved for the most compute-intensive, mission-critical functions only, because ASICs have two big problems: they take a lot of time and money to develop, and any algorithms they contains are effectively "frozen in silicon." This latter point is a bit of a downer in the wireless world where standards can change faster than you can say *"I love this standard . . . Good grief, it just changed again!"*

Spatial and temporal segmentation (SATS)

The ACM is a new type of digital IC whose architecture is designed to allow dynamic software algorithms to be directly mapped into dynamic hardware resources on-the-fly, resulting in the most efficient use of hardware in terms of cost, size (silicon real estate), speed, and power consumption. One of the most important aspects of the ACM is its ability to have its architecture change on demand tens or hundreds of thousands of times per second, while consuming very little power. This distinguishes the ACM from any other IC implementation technology, because it can perform *spatial and temporal segmentation (SATS)*.

SATS is the process of rapidly adapting the ACM's dynamic hardware resources to perform the various portions of an algorithm in different "segments" of time and in different locations on the ACM. As a simple example, consider that some operations on a wireless phone are modal, which means they only need to be performed some of the time. The four main modes are *acquisition* (search), *idle*, *traffic* (receive), and *traffic* (transmit) (Figure 21-6).

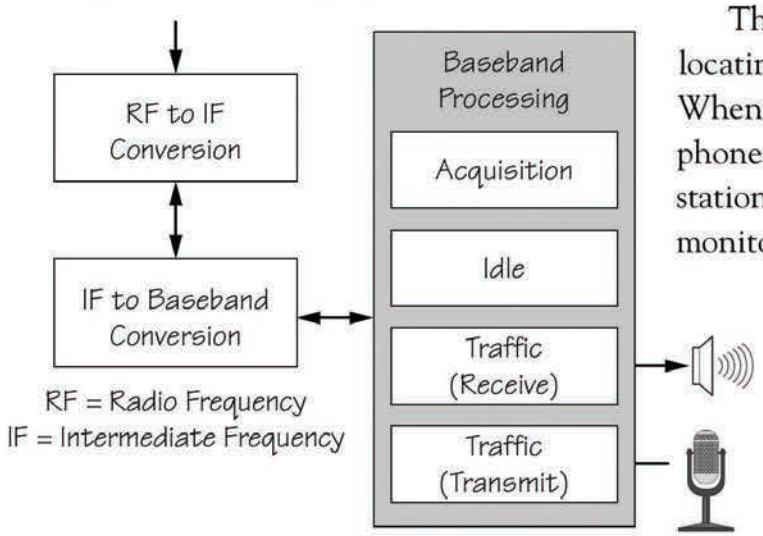


Figure 21-6. Some of the operations on a wireless phone are modal

The *acquisition* mode refers to locating the nearest base station. When in *idle* mode, the wireless phone keeps track of the base station it's hooked up to and monitors the paging channel, looking for a signal that says "*Wake up because a call is being initiated.*" The *traffic* mode has two variations: *receiving* and *transmitting* (although you may think the widgets inside a phone are performing both tasks simultaneously, the reality is that, inside the digital chip, the processors are often sequencing back and forth between the transmit and receive functions).

In the case of a wireless phone based on conventional IC technologies, each of these baseband processing functions requires its own DSP or ASIC (or some

area on a common ASIC). This means that even when a function isn't being used, it still occupies silicon real estate and consumes power. In turn, this affects decisions made by the design engineers. For example, when in *acquisition* mode, you can increase the number of "searcher" functions to speed things up, but each function requires ASIC material real estate. This causes engineers to limit the number of search functions they use (hence the fact that it can take your cell phone a seemingly endless time to locate the nearest base station when you first power it on).

By comparison, a next-generation ACM-based wireless phone requires only a single ACM that can be adapted on-the-fly to perform each baseband function as required. For example, when in *acquisition* mode, the ACM would be adapted to perform only acquisition; during *transmit* mode the ACM would be adapted to the transmit functions . . . and so forth. This means that at any point in time, only the function that is required would be resident in the ACM and consuming resources. More importantly, it also means that when the phone is in *acquisition* mode, for example, all of the ACM's resources can be configured as searchers, which means that you can use many more searchers than you could afford to support in ASIC material, which in turn means that the ACM can out-perform the ASIC for this type of real-world function.

Three-Dimensional Molded Interconnect

For many years, designers have wished for the ability to create robust, three-dimensional (3D) circuit boards for use with products such as handheld cellular telephones, radios, and calculators. In addition to providing the interconnect, these 3D circuit boards would also act as the product's package. (Alternatively, depending on your point of view, we could say that designers want to be able to create a 3D product package that also acts as the interconnect). In addition to reducing the product's size and weight, there can also be significant benefits in terms of cost and manufacturability.

A number of processes to create 3D circuit boards have emerged over the years, but none have gained any significant level of commercial acceptance. One hopeful contender involved creating a standard two-dimensional (2D) circuit board and then molding it into a 3D shape. This process achieved some acceptance for simple product packages, such as those for power supplies, with limited aesthetic requirements. However, the process is not suitable for

products that require aesthetic appeal and ergonomic shapes involving complex surfaces.

Another technique involved the prefabrication of tracks on the inside surface of a mold. The process of injecting plastic into the mold caused the tracks to become an integral part of the packaging. But this process had its own problems—not the least of which was creating the interconnects on the 3D surface of the mold in the first place.

One development in 3D photo-imaging technology in the early 1990s reawakened interest in the injection molding technique. The process commences with the injection molding of a plastic material capable of withstanding high enough temperatures to undergo reflow soldering or vapor-phase soldering processes.[11] In addition to the physical shape of the product, it is also possible to mold in features such as holes, ribs, recesses, standoffs, and chamfered edges. These features offer substantial savings compared to their equivalent drilling, routing, and grinding operations, and also reduce part counts and labor.

A 3D photo-tool, or *mask*, is formed by molding a PVC-based material to conform exactly to the contours of the plastic part. Next, a computer-controlled laser is used to draw an image of the desired circuit onto the photo-tool (Figure 21-7).

The remainder of the process is very similar to that for a standard circuit board, and can therefore leverage off existing technology. The surface of the plastic is covered by a special resist, the photo-tool is inserted, the whole assembly is exposed to ultraviolet light, the degraded resist is removed, and copper tracks are grown using an additive process.

One of the major problems with the process at this time is the lack of appropriate computer-aided design tools. Existing tools are geared to standard 2-D circuit boards and do not have the ability to transform 2D designs into a 3D space. Sadly, there is not yet a large enough market potential to cause this position to change. However, this technology is still in its infant stage and, with sufficient market interest, the requisite electromechanical toolsets could certainly become available.

---

[11] These soldering processes were introduced in Chapter 18.

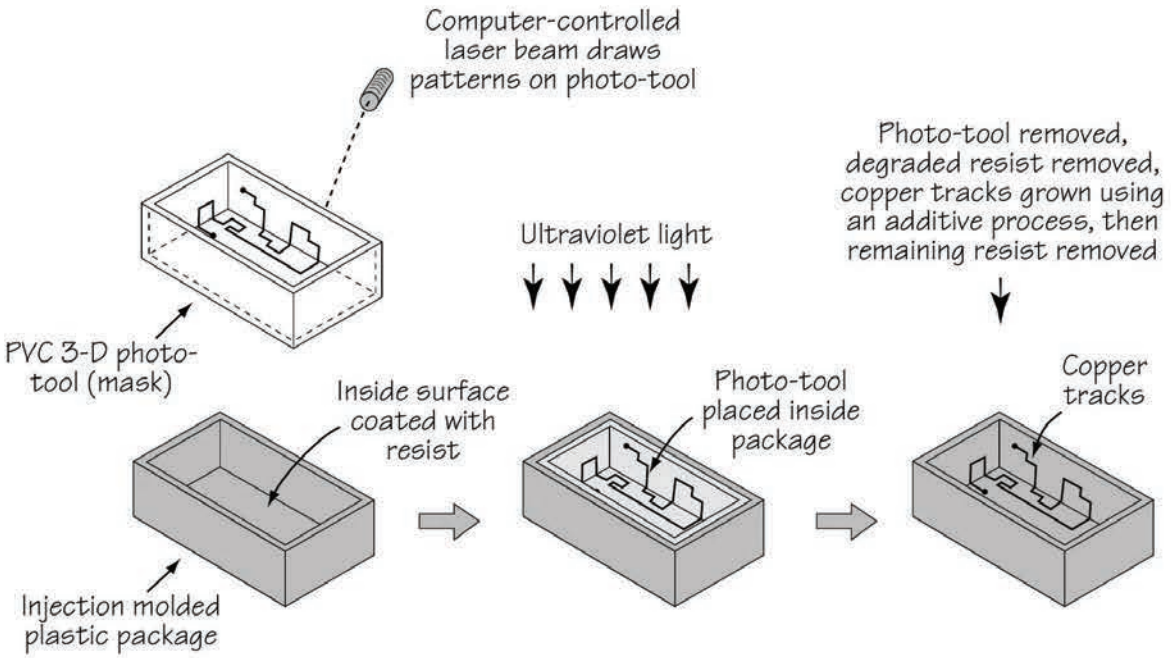


Figure 21-7. Three-dimensional molded interconnect

Optical Interconnect

Electronics systems exhibit ever-increasing requirements to process ever-increasing quantities of data at ever-increasing speeds. Interconnection technologies based on conducting wires are fast becoming the bottleneck that limits the performance of electronic systems.

To relieve this communications bottleneck, a wide variety of opto-electronic interconnection techniques are undergoing evaluation. In addition to the extremely fast propagation of data,[12] optical interconnects offer greater signal isolation, reduced sensitivity to electromagnetic interference, and a far higher bandwidth than do conducting wires.

[12] Light travels at 299,792,458 meters per second in a vacuum. Thus, a beam of light would take approximately 2.6 seconds to make a round trip from the earth to the moon and back again!

Fiber-Optic Interconnect

The fibers used in fiber-optic systems are constructed from two different forms of glass (or other materials) with different refractive indices. These fibers, which are finer than a human hair, can be bent into weird and wonderful shapes without breaking. When light is injected into one end of the fiber, it repeatedly bounces off the interface between the two glasses, undergoing almost total internal reflection with minimal loss, until it reemerges at the other end (Figure 21-8).

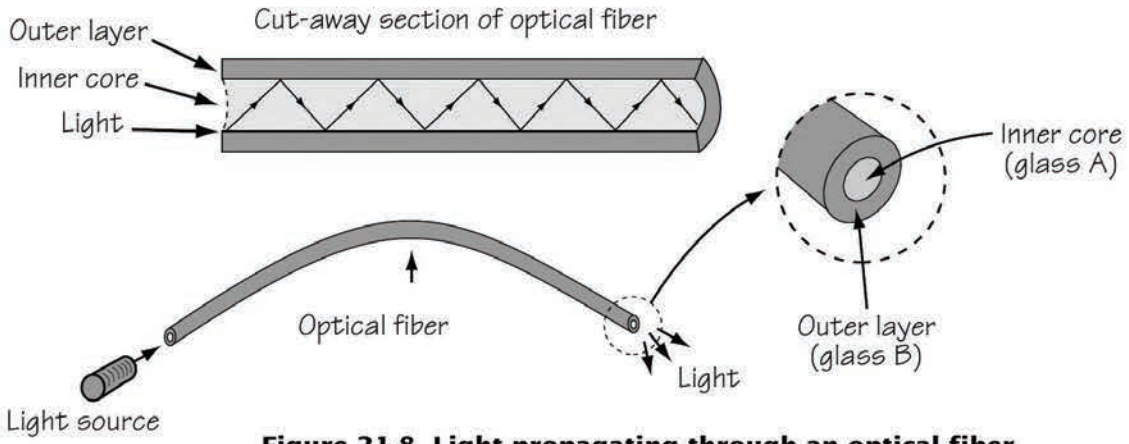


Figure 21-8. Light propagating through an optical fiber

Experimental systems using fiber-optic interconnect have been evaluated at all levels of a system—for example, to link bare die in a multichip module (Figure 21-9).

The transmitting device employs a surface-emitting laser-diode, which is constructed along with the transistors and other components on the integrated circuit's substrate. The receiving device uses a photo-transistor to convert the incoming light back into an electrical signal.

Each die can support numerous transmitters and receivers, which can be located anywhere on the surface of their substrates. However, there are several problems with this implementation, including the difficulty of attaching multiple optical fibers, the difficulties associated with repair and rework (replacing a defective die), and the physical space occupied by the fibers. Although the individual fibers are extremely thin, multichip modules may require many thousands of connections. Additionally, in this form, each optical fiber can be used only to connect an individual transmitter to an individual receiver.

At the board level, a variation of discrete wired technology[13] has been developed, in which optical fibers are ultrasonically bonded into the surface of a board. When combined with *chip-on-board (COB)* techniques, this process offers significant potential for the future. In fact, due to the way in which the optical fibers are connected to the die, it is possible for an individual transmitter to be connected to multiple receivers. Unfortunately, the techniques used to achieve this may not be disclosed here, because the author has been sworn to secrecy using arcane oaths and strange handshakes, and an Englishman's word is his bond!

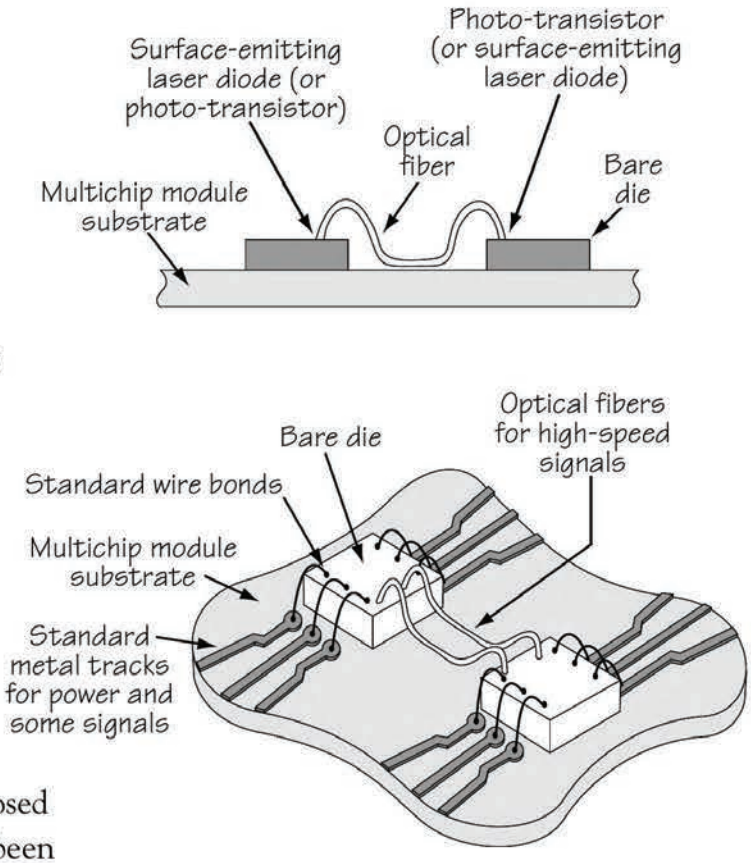


Figure 21-9. Fiber-optic interconnect: linking bare die in a multichip module

Last but not least, optical fibers may be used to provide intra-board connections, which may be referred to as *optical backplanes*. Once again, discrete wire technology, modified to accommodate optical fibers, may be employed to create the backplanes. Alternatively, the daughter boards may be mounted in a rack without an actual backplane, and groups of optical fibers may be connected into special couplers (Figure 21-10).

Each optical fiber from a transmitter is connected into a coupler, which amplifies the optical signal and can re-transmit it to multiple receivers. This form of backplane offers great latitude in regard to the proximity of the boards. In fact, boards connected in this way can be separated by as much as tens of meters.

---

[13] Discrete wired technology was introduced in Chapter 18.

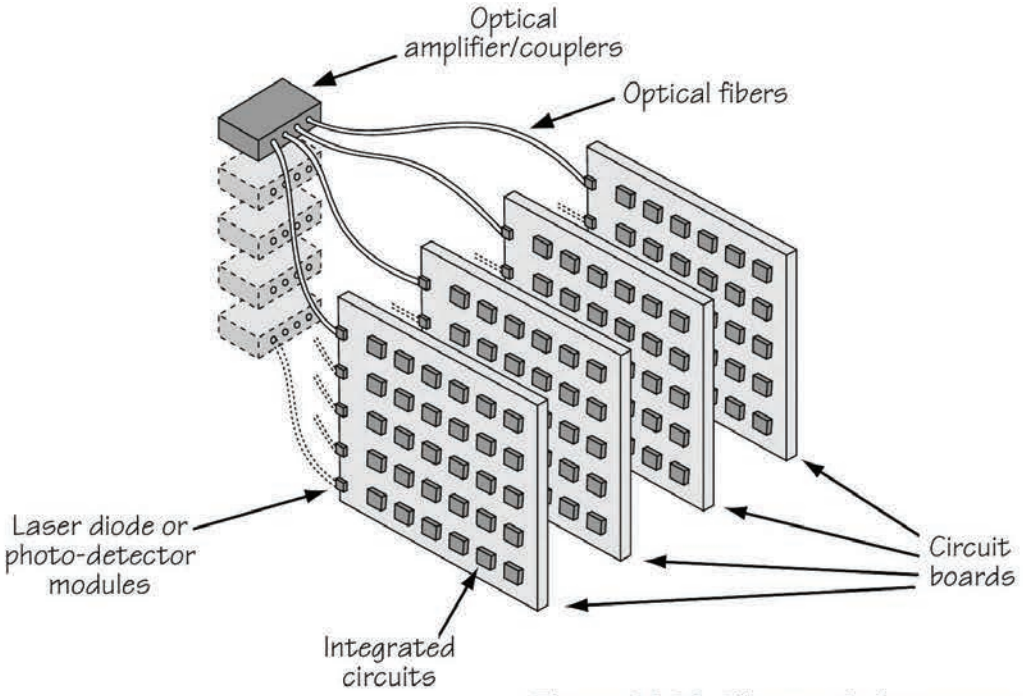


Figure 21-10. Fiber-optic interconnect: optical backplanes

In 1834, the Scottish scientist John Scott Russell was observing a barge being pulled along a canal by a pair of horses. When the barge stopped, he noticed that the bow wave continued forward without appearing to deteriorate in any way. Most of us wouldn't have paid this any attention, but Russell jumped on a horse and followed the wave for miles.

What Russell had observed was a special form of wave called a *soliton* that, due to its contour, can retain its shape and speed. One example of a soliton is the *Severn bore*: a wave caused by unusual tidal conditions (where the Severn is a river that separates England and Wales). Thus a bore (wave) can travel miles up the river without significant diminishment or deformation.

The reason this is of interest is that light waves become distorted and attenuated as they travel through optical fibers—similar to the way in which electrical pulses deteriorate as they propagate through conducting wires. Scientists and engineers are now experimenting with light solitons—pulses that can travel long distances through optical fibers without distortion.

Using light solitons, data can retain its integrity while being transmitted over 20,000 kilometers through optical fibers at a rate of 10 gigabits per second, which is four times faster than is possible using conventional techniques.

Free-Space Interconnect

In the case of the *free-space* technique, a laser-diode transmitter communicates directly with a photo-transistor receiver without employing an optical fiber. Consider a free-space technique used to link bare die mounted on the substrate of a multichip module (Figure 21-11).

In this case, the transmitters are constructed as side-emitting laser diodes along the upper edges of the die; similarly with the photo-transistors on the receivers. Each die may contain multiple transmitters and receivers. The free-space technique removes some of the problems associated with its fiber-optic equivalent: there are no fibers to attach and replacing a defective die is easier.

However, as for optical fibers, each transmitter can still be used only to connect to an individual receiver. Additionally, the free-space technique has its own unique problems: the alignment of the devices is critical, and there are thermal tracking issues. When a laser-diode is turned on, it rapidly cycles from ambient temperature to several hundred degrees Celsius. The heat generated by an individual laser-diode does not greatly affect the die because each diode is so small. However, the cumulative effect of hundreds of such diodes does affect the die, causing it to expand, thereby disturbing the alignment of the transmitter-receiver pairs.

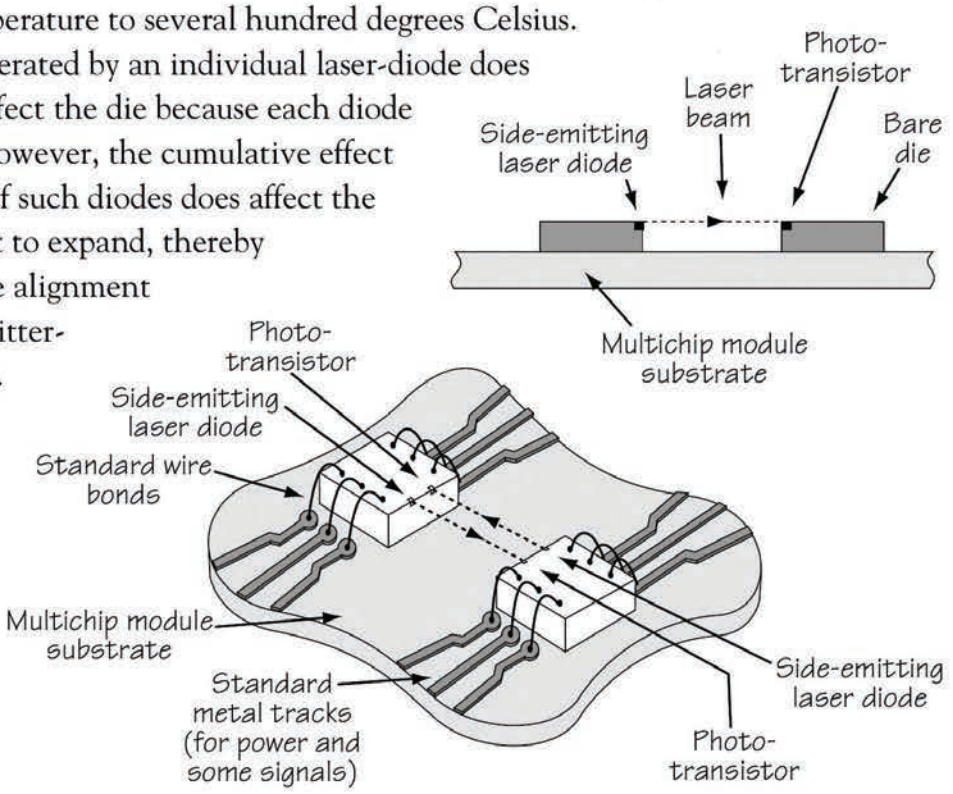


Figure 21-11. Free-space interconnect: linking bare die in a multichip module

Guided-Wave Interconnect

Another form of optical interconnect that is receiving significant interest is that of *guided-wave*, whereby optical waveguides are fabricated directly on the substrate of a multichip module. These waveguides can be created using variations on standard opto-lithographic thin-film processes. One such process involves the creation of silica waveguides (Figure 21-12).[14]

Using a *flipped-chip* mounting technique, the surface-emitting laser-diodes and photo-transistors on the component side of the die are pointed down towards the substrate. One major advantage of this technique is that the waveguides can be constructed with splitters, thereby allowing a number of transmitters to drive a number of receivers. On the down side, it is very difficult

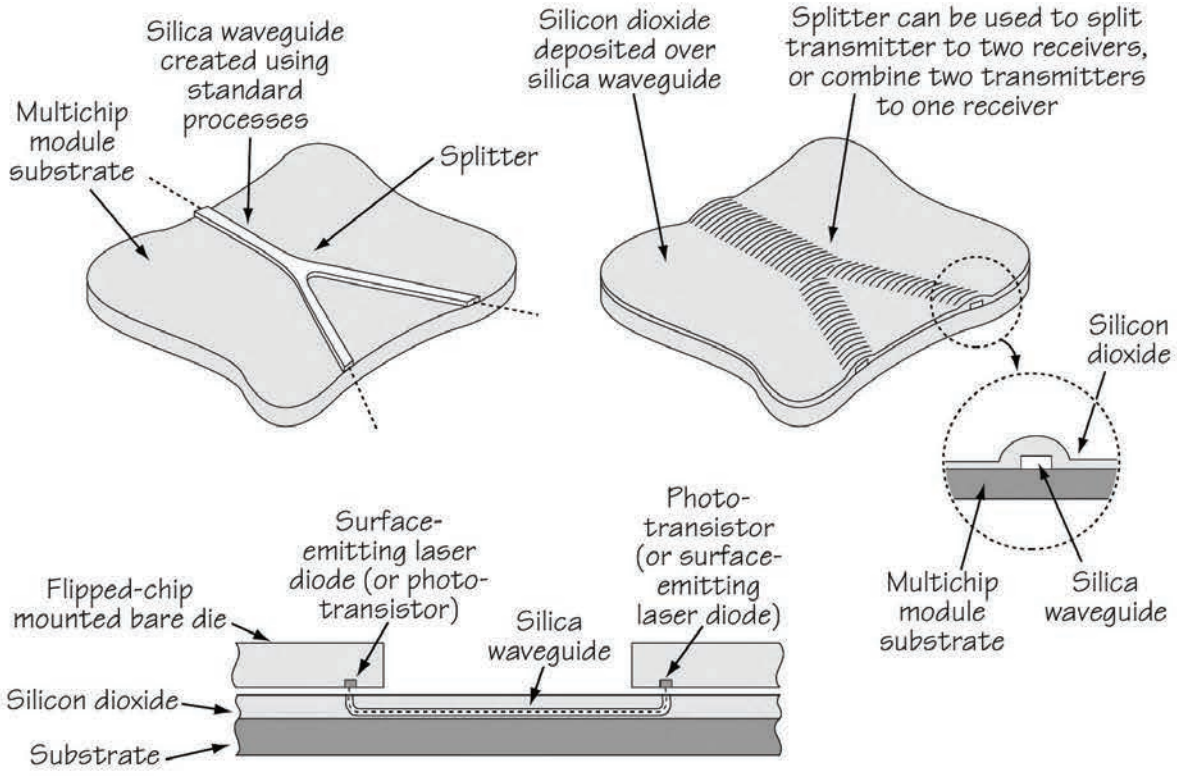


Figure 21-12. Guided wave interconnect: silica waveguides

[14] Thanks for the information on silica waveguides go to Dr. Terry Young of the GEC-Marconi Research Center, Chelmsford, Essex, England, with apologies for all the extremely technical details that were omitted here.

to route one waveguide over another, because the crossover point tends to act like a splitter and allows light from the waveguides to "leak" into each other.

An alternative form of waveguide technology, *photo-imagable polyimide interconnect*, which was announced towards the tail-end of 1993, is of particular interest in the case of multichip modules of type MCM-D. As you may recall from Chapter 20, MCM-D devices are ceramic, glass, or metal substrates that are covered with a layer of dielectric material such as polyimide. The dielectric coat is used to modify the substrate's capacitive characteristics, and tracks are created on the surface of the dielectric using thin-film processes.

When exposed to light passed through an appropriate mask, a layer of photo-imagable polyimide can be imprinted with patterns in a similar way to exposing a photograph. After being developed, the polyimide contains low-loss optical waveguides bounded by relatively opaque reflective surfaces (Figure 21-13).

Apart from its inherent simplicity, one of the beauties of this technique is that both the exposed and unexposed areas of polyimide have almost identical dielectric constants. Thus, in addition to leveraging off existing technology, the polyimide waveguides have relatively little impact on any thin-film metalization tracking layers that may be laid over them.

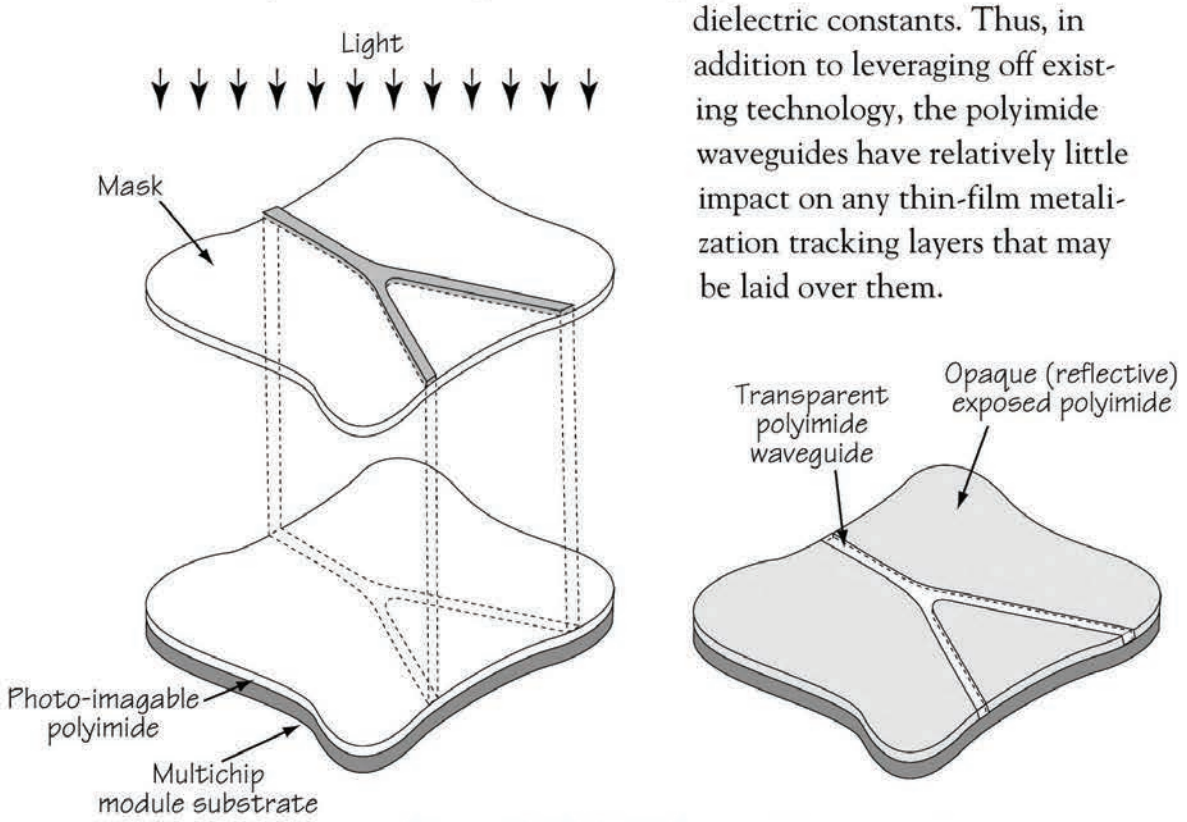


Figure 21-13. Guided wave interconnect: photo-imagable polyimide waveguides

This technique is currently finding its major audience in designers of multichip modules, but it is also being investigated as a technique for multilayer circuit boards. In the future, circuit boards could be fabricated as a mixture of traditional copper interconnect and very high speed optical interconnect.

Holographic Interconnect

Unfortunately, all of the optical interconnection technologies introduced above have their own unique problems and limitations. The technique that shows most potential is that of guided waves, but even this has the problem of routing waveguides over each other. Additionally, both waveguides and optical fibers share a common shortcoming, the fact that the light tends to bounce around an awful lot. In fact, the light bounces around so much that it actually travels approximately four to six times the straight-line distance between the transmitter and the receiver.

Quite apart from these concerns, it is doubtful if any of the above techniques will be capable of dealing with the sheer number of interconnection paths that will be required. A new contender that is rising to the challenge is *holographic interconnect*. Using the term "holographic" in this context may at first seem a little strange, because holography is traditionally considered to be a method of obtaining three-dimensional images known as holograms (from the Greek: *holos*, meaning "whole" and *gram*, meaning "message"). However, the term holographic is appropriate, because this form of interconnect is actually based on a three-dimensional image.

In the case of holographic interconnect for a multichip module, the process commences with an extremely thin slice of quartz, into which sophisticated patterns are cut using a laser. The quartz slice is then mounted approximately 1 mm above the die, and a *face-surface mirror*[15] is mounted approximately 2 cm above the quartz (Figure 21-14).

When a surface-emitting laser diode on one of the die is turned on, it transmits a laser beam straight up into one of the patterns in the quartz. This pattern causes the laser beam to be deflected so that it bounces backwards and forwards between the face-surface mirror and the quartz. When the laser hits

[15] The "face-surface" appellation refers to the fact that the reflective coating is located on the face of the mirror, and not behind it as would be the case in a mirror you look into when you shave or apply your makeup (or both, depending on how liberated you are).

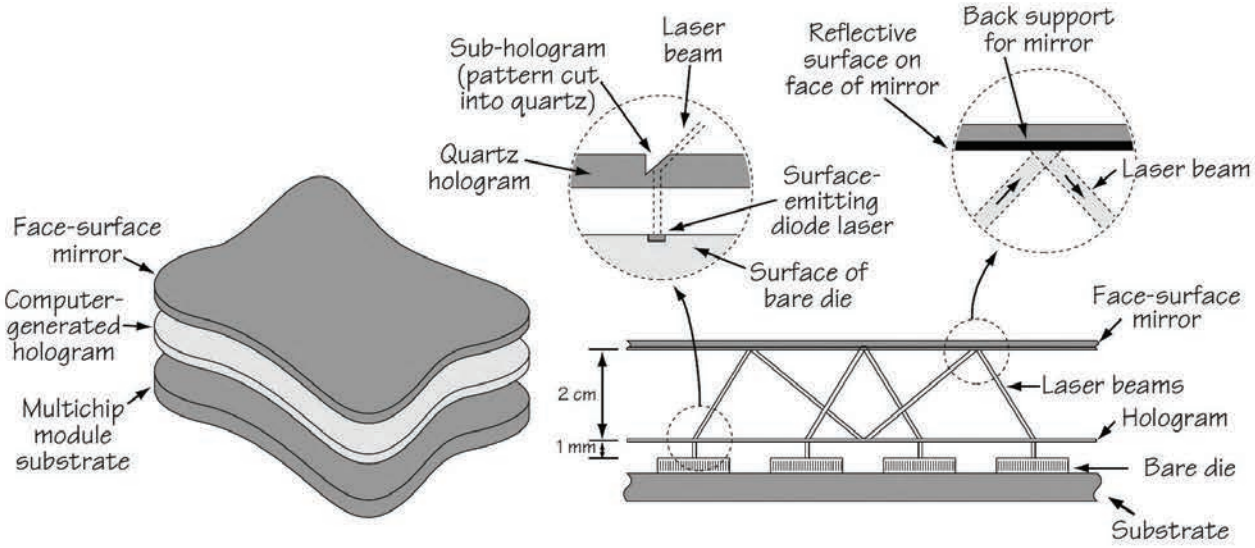


Figure 21-14. Holographic interconnect: multichip module

the point on the quartz directly above a photo-transistor on the receiving die, another pattern cut into the quartz causes it to be deflected back through the quartz and down into the receiver.

Note that the patterns cut into the quartz in the illustration above are gross simplifications. Additionally, there are typically two patterns cut into the quartz over the receiver, one on each side of the slice. The pattern on the upper side of the slice captures the laser beam and deflects it into the slice, and the pattern on the lower side then captures it again and redirects it out of the slice down onto the receiver. These patterns also perform the same function as a lens, focusing the beam precisely onto the receiver.

The angles of the patterns cut into the quartz are precisely calculated, as are the number of reflections between the face-surface mirror and the quartz, thereby ensuring that each laser beam exactly hits the target for which it is intended and no other. All of these calculations are performed by a computer, and the laser used to cut the three-dimensional patterns into the quartz is also controlled by a computer. Thus, the slice of quartz is referred to a *computer-generated hologram (CGH)* and the individual patterns above each transmitter and receiver are known as *sub-holograms*.

In fact, the laser beam actually undergoes relatively few reflections between the face-surface mirror and the quartz: far fewer than an equivalent beam transmitted through a waveguide or an optical fiber. Thus, the time taken for a

signal to propagate from a transmitter to a receiver—known as the *time-of-flight*—is actually close to that for free-space interconnect. But the clever part is still to come. The sub-hologram above a laser diode can be created in such a way that it splits the laser beam into sub-beams, each of which can target a different receiver (Figure 21-15).

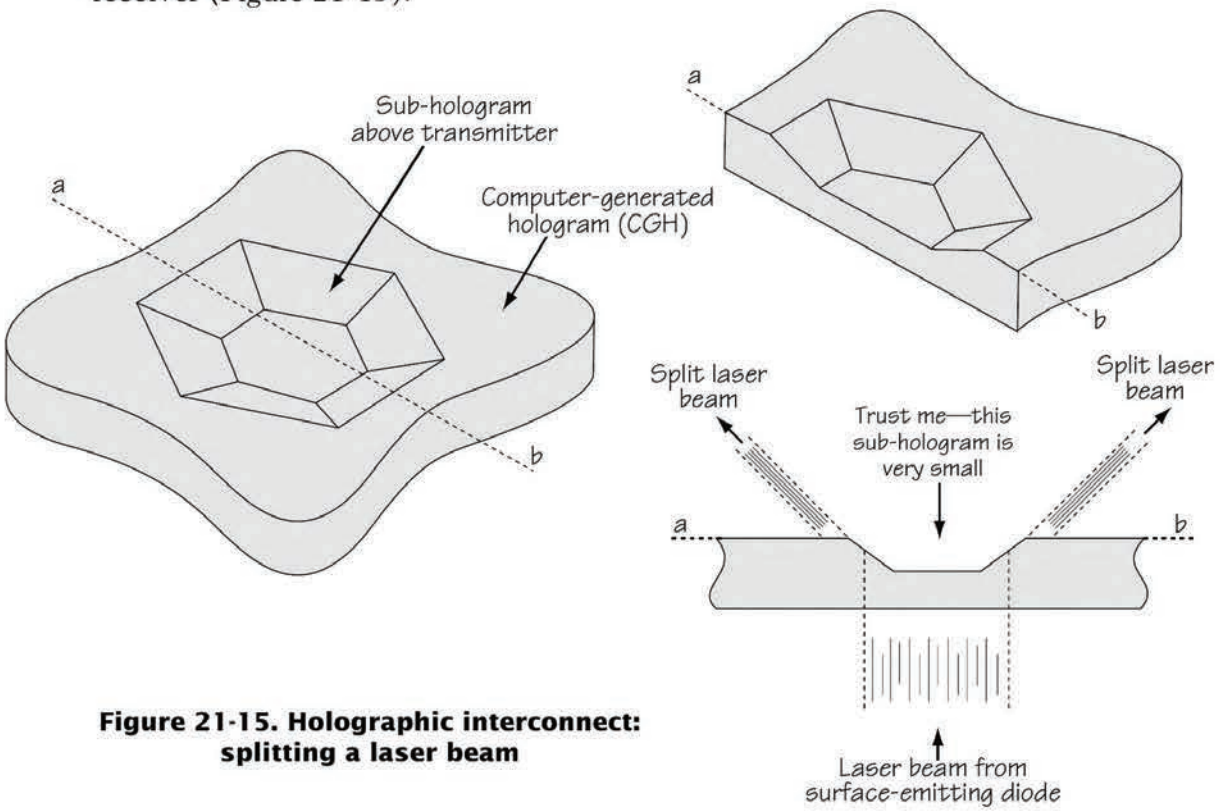


Figure 21-15. Holographic interconnect: splitting a laser beam

For the sake of clarity, Figure 21-15 shows a hexagonal sub-hologram with equal angles between the vertices, and equal angles of incidence for each face. However, the angles between the vertices can vary, and each face can have a different angle of incidence. Furthermore, the sub-holograms above receivers can be correspondingly complex, and capable of receiving signals from multiple transmitters.

Finally, in addition to multichip module applications, holographic interconnect is also undergoing evaluation as a possible backplane technology for the interconnection of circuit boards. These backplanes—known as *holo-backplanes*—offer an additional, amazingly powerful capability. It is technically feasible to create multiple holograms in the backplane's equivalent of the quartz

slice. Thus, by moving the holographic slice a fraction of a millimeter in a direction parallel to the face-surface mirror, the interconnection pattern could be reconfigured and a completely new set of board-to-board interconnects could be established. The mind boggles!

Optical Memories

It has been estimated that the total sum of human knowledge is doubling approximately every ten years. Coupled with this, the amount of information that is being generated, stored, and accessed is increasing at an exponential rate. This is driving the demand for fast, cheap memories that can store gigabits (a billion bits), terabits (a thousand billion bits), or even petabits (a million billion bits) of data.

One medium with the potential to cope with this level of data density is *optical storage*. Among many other techniques, evaluations are being performed on extremely thin layers of glass-based materials,[16] which are doped with organic dyes or rare-earth elements. Using a technique known as *photochemical hole-burning (PHB)*, a laser in the visible waveband is directed at a microscopic point on the surface of the glass (Figure 21-16).

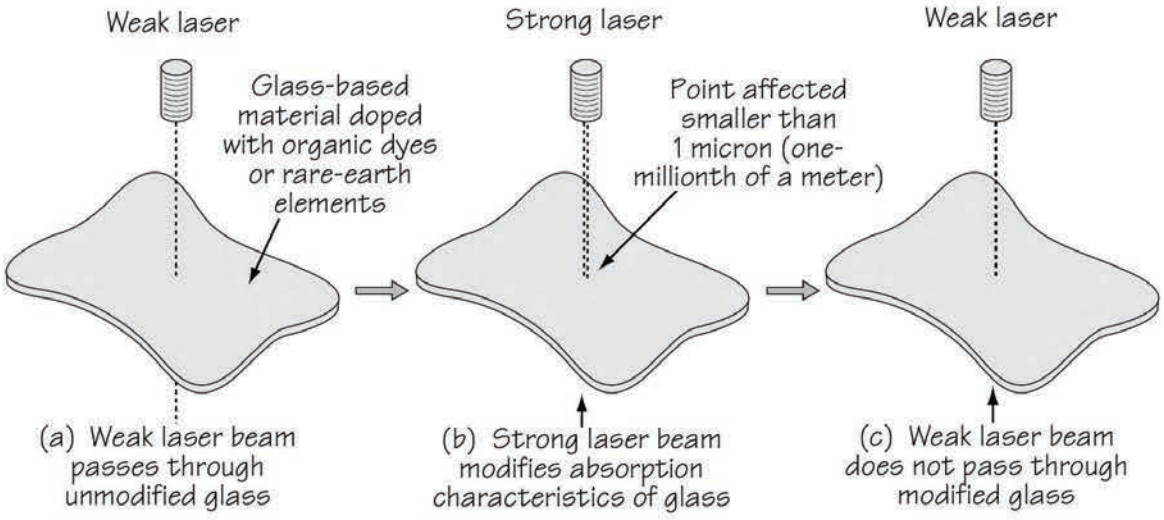


Figure 21-16. Optical memories: photochemical hole-burning (PHB)

[16] One such material, boric-acid glass, is also widely used in heat-resistant kitchenware!

If the laser is weak, its light will pass through the glass without affecting it and reappear at the other side. If the laser is stronger (but not intense enough to physically damage the glass), electrons in the glass will be excited by the light. The electrons can be excited such that they change the absorption characteristics of that area of the glass and leave a band, or hole, in the absorption spectrum. To put this another way, if the weak laser beam is redirected at the same point on the glass surface, its light would now be absorbed and would not reappear at the other side of the glass.

Thus, depending on whether or not the light from the weaker beam passes through the glass, it can be determined whether or not that point has been exposed to the strong laser. This means that each point can be used to represent a binary 0 or 1. Because the point affected by the laser is so small, this process can be replicated millions upon millions of times across the surface of the glass.

If the points occur at one micron (one millionth of a meter) intervals, then it is possible to store 100 megabits per square centimeter, but this still does not come close to the terabit storage that will be required. However, it turns out that each point can be "multiplexed" and used to store many bits of information. A small change in the wavelength of the laser can be used to create another hole in a different part of the spectrum. In fact, 100x multiplexing has been achieved, where each point on the glass was used to store 100 bits of data at different wavelengths. Using 100x multiplexing offers a data density of 10 gigabits per square centimeter, and even higher levels of multiplexing may be achieved in the future!

Protein Switches and Memories

Another area receiving a lot of interest is that of switches and memories based on proteins. We should perhaps commence by pointing out that this concept doesn't imply anything gross like liquidizing hamsters to extract their proteins! Out of all the elements nature has to play with, only carbon, hydrogen, oxygen, nitrogen, and sulfur are used to any great extent in living tissues, along with the occasional smattering of phosphorous, minuscule portions of a few choice metals, and elements like calcium for bones. The basic building blocks of living tissues are twenty or so relatively simple molecules called amino acids. For example, consider three of the amino acids called threonine, alanine, and serine (Figure 21-17).

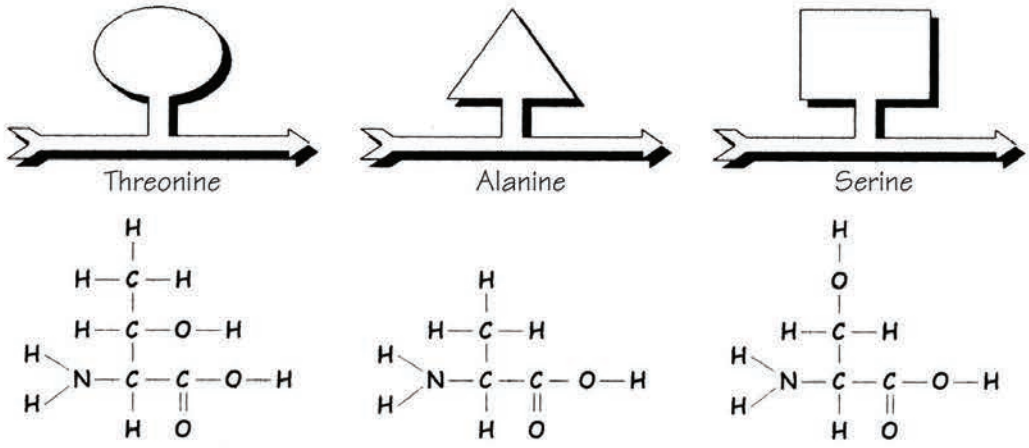


Figure 21-17. Threonine, alanine, and serine are three of the twenty or so biological building blocks called amino acids

These blocks can join together to form chains, where the links between the blocks are referred to as *peptide bonds*, which are formed by discarding a water molecule (H_2O) from adjacent COOH and NH_2 groups (Figure 21-18).

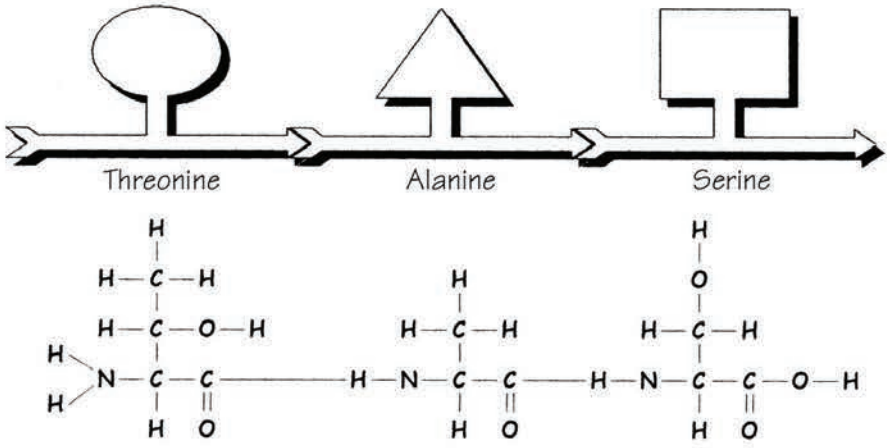


Figure 21-18: Amino acids can link together using peptide bonds to form long polypeptide chains

Proteins consist of hundreds or thousands of such chains of amino acids. Note that the distribution of electrons in each amino acid varies depending on the size of that acid's constituent atoms, leaving areas that are slightly more positively or negatively charged (similar to a water molecule, as is discussed later in this chapter). The linear chain shown in Figure 21-18 is known as the *primary structure* of the protein, but this chain subsequently coils up into a

spring-like helix, whose shape is maintained by the attractions between the positively and negatively charged areas in the chain. This helix is referred to as the protein's *secondary structure*, but there's more, because the entire helix subsequently "folds" up into an extremely complex three-dimensional structure, whose shape is once again determined by the interactions between the positively and negatively charged areas on the helix. Although this may seem to be arbitrarily random, this resulting *tertiary structure* represents the lowest possible energy level for the protein, so proteins of the same type always fold up into identical (and stable) configurations.

Organic molecules have a number of useful properties, not the least of which is that their structures are intrinsically "self healing" and reject contamination. Also, in addition to being extremely small, many organic molecules have excellent electronic properties.

In the case of certain proteins, it's possible to coerce an electron to move to one end of the protein or the other, where it will remain until it's coerced back again (note that the term "end" is somewhat nebulous in this context). Thus, a protein of this type can essentially be used to store and represent a *logic 0* or a *logic 1* value based on the location of this electron.[17] Similarly, it's possible for some protein structures to be persuaded to act in the role of switches.

In the case of traditional semiconductor-based transistors, even when one considers structures measured in fractions of a millionth of a meter, each transistor consists of millions upon millions of atoms. By comparison, protein-based switches and registers can be constructed using a few thousand atoms, which means that they are thousands of times smaller, thousands of times faster, and consume a fraction of the power of their semiconductor counterparts.

Unlike metallic conductors, some proteins transfer energy by moving electron excitations from place to place rather than relocating entire electrons. This can result in switching speeds that are orders of magnitude faster than their semiconductor equivalents.

Some proteins react to electric fields, while others respond to light. For example, there is a lot of interest in the protein *Rhodopsin*, which is used

[17] In the case of some proteins, rather than physically moving an electron from one "end" to the other, it's possible to simply transfer an *excitation* from one electron to another. This requires far less power and occurs much faster that moving the electron itself, but it's a little too esoteric a concept to explore in detail here.

by certain photosynthetic bacteria to convert light into energy. The bacteria that contain Rhodopsin are the ones that cause ponds to turn red, and their saltwater cousins are responsible for the purple tint that is sometimes seen in San Francisco Bay.

In certain cases, light from a laser can be used to cause such optically responsive proteins to switch from one state to another (which they do by changing color) and back again. Additionally, some varieties of proteins are only responsive to the influence of two discrete frequencies. This feature is extremely attractive, because it offers the possibility of three-dimensional optical protein memories.

Experiments have been performed in which 3D cubes have been formed as ordered arrays of such bi-frequency proteins suspended in transparent polymers. If the protein were affected by a single laser, then firing a beam into the cube would result in a line of proteins changing state. But in the case of bi-frequency proteins, two lasers mounted at 90° to each other can be used to address individual points in the 3-D space (Figure 21-19).

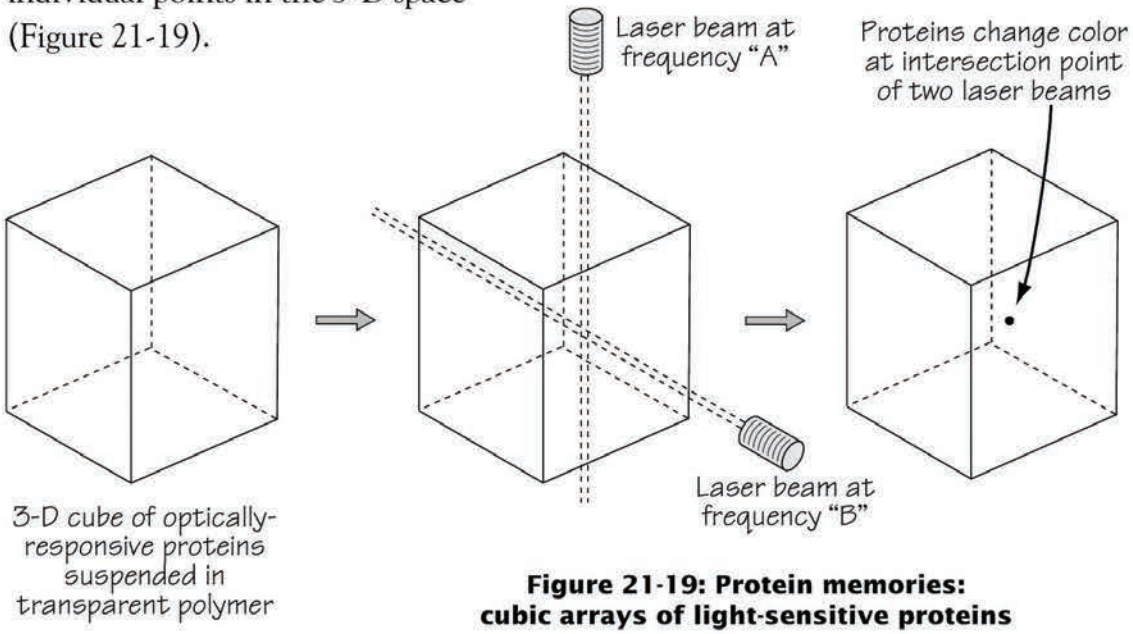


Figure 21-19: Protein memories: cubic arrays of light-sensitive proteins

By only slightly enhancing the technology available today, it may be possible to store as much as 30 gigabits in a 1 cm × 1 cm × 1 cm cube of such material, where even one gigabit would be equivalent to more than 100 of today's 256 megabit RAM devices!

Electromagnetic Transistor Fabrication

For some time it has been known that the application of strong electromagnetic fields to special compound semiconductors can create structures that behave like transistors. The original technique was to coat the surface of a semiconductor substrate with a layer of dopant material, and then to bring an extremely strong, concentrated electromagnetic field in close proximity.

The theory behind this original technique was that the intense field caused the electromigration of the dopant into the substrate. However, much to everyone's surprise, it was later found that this process remained effective even without the presence of the dopant!

Strange as it may seem, nobody actually understands the mechanism that causes this phenomenon. Physicists currently suspect that the strong electromagnetic fields cause microscopic native defects in the crystals to migrate through the crystal lattice and cluster together.

Heterojunction Transistors

If there is one truism in electronics, it is that "faster is better," and a large proportion of research and development funds are invested in increasing the speed of electronic devices.

Ultimately there are only two ways to increase the speed of semiconductor devices. The first is to reduce the size of the structures on the semiconductor, thereby obtaining smaller transistors that are closer together. The second is to use alternative materials that inherently switch faster. However, although there are a variety of semiconductors, such as *gallium arsenide (GaAs)*, that offer advantages over silicon for one reason or another, silicon is cheap, readily available, and relatively easy to work with. Additionally, the electronics industry has billions of dollars invested in silicon-based processes.

For these reasons, speed improvements have traditionally been achieved by making transistors smaller. However, some pundits believe that we are reaching the end of this route using conventional technologies. At one time, the limiting factors appeared to be simple process limitations: the quality of the resist, the ability to manufacture accurate masks, and the features that could be achieved with the wavelength of ultraviolet light. Around 1990, when structures with dimensions of 1.0 microns first became available, it was believed that structures of 0.5 microns would be the effective limit that could be achieved with opto-

lithographic processes, and that the next stage would be a move to X-ray lithography. However, there have been constant improvements in the techniques associated with mask fabrication, optical systems and lenses, servo motors and positioning systems, and advances in chemical engineering such as chemically-amplified resists.[18] The combination of all these factors means that it is now possible to achieve structures smaller than 0.1 microns by continuing to refine existing processes.

However, there are other considerations. The speed of a transistor is strongly related to its size, which affects the distance electrons have to travel. Thus, to enable transistors to switch faster, technologists have concentrated on reducing size, a strategy which is commonly referred to as *scaling*. However, while scaling reduces the size of structures, it is necessary to maintain certain levels of dopants to achieve the desired effect. This means that, as the size of the structures is reduced, it is necessary to increase the concentration of dopant atoms. Increasing the concentration beyond a certain level causes leakage, resulting in the transistor being permanently ON and therefore useless. Thus, technologists are increasingly considering alternative materials and structures.

An interface between two regions of semiconductor having the same basic composition but opposing types of doping is called a *homojunction*. By comparison, the interface between two regions of dissimilar semiconductor materials is called a *heterojunction*. Homojunctions dominate current processes because they are easier to fabricate. However, the interface of a heterojunction has naturally occurring electric fields, which can be used to accelerate electrons, and transistors created using heterojunctions can switch much faster than their homojunction counterparts of the same size.

One form of heterojunction that is attracting a lot of interest is found at the interface between silicon and germanium. Silicon and germanium are in the same family of elements and have similar crystalline structures. In theory this should make it easy to combine them, but it's a little more difficult in practice. A process currently being evaluated is to create a standard silicon wafer with

[18] In the case of a chemically amplified resist, the application of a relatively small quantity of ultraviolet light stimulates the formation of chemicals in the resist, which accelerates the degrading process. This reduces the amount of ultraviolet light required to degrade the resist and allows the creation of finer features with improved accuracy.

doped regions, and then to grow extremely thin layers of a silicon-germanium alloy where required.

The two most popular methods of depositing these layers are *chemical vapor deposition (CVD)* and *molecular beam epitaxy (MBE)*. In the case of chemical vapor deposition, a gas containing the required molecules is converted into a plasma[19] by heating it to extremely high temperatures using microwaves. The plasma carries atoms to the surface of the wafer where they are attracted to the crystalline structure of the substrate. This underlying structure acts as a template. The new atoms continue to develop the structure to build up a layer on the substrate's surface (Figure 21-20).

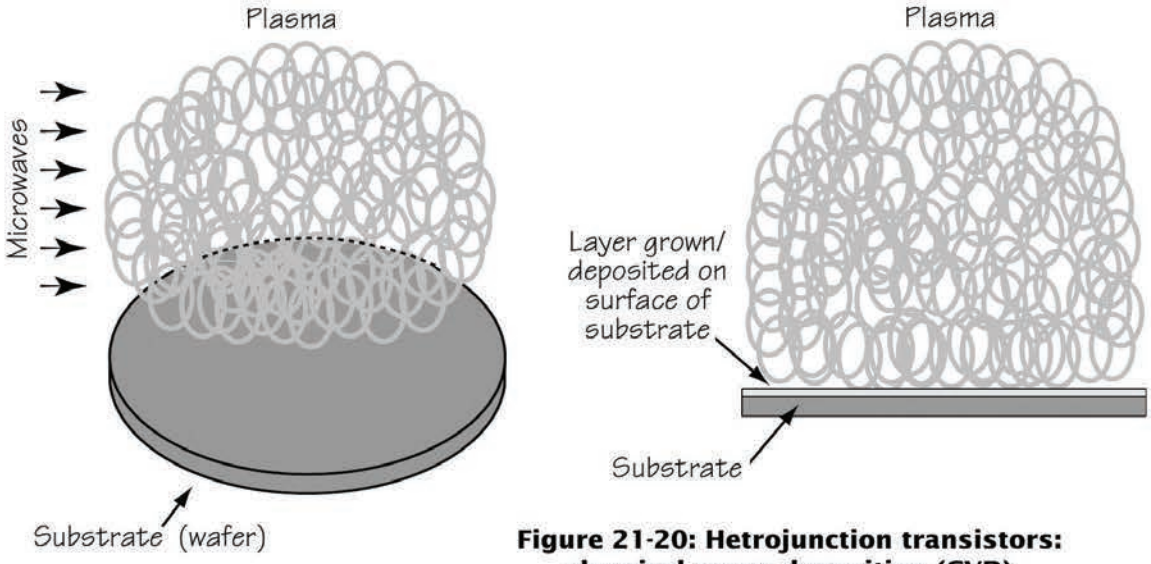


Figure 21-20: Hetrojunction transistors: chemical vapor deposition (CVD)

By comparison, in the case of molecular beam epitaxy, the wafer is placed in a high vacuum, and a guided beam of ionized molecules is fired at it, effectively allowing molecular-thin layers to be "painted" onto the substrate where required.[20]

---

[19] A gaseous state in which the atoms or molecules are dissociated to form ions.

[20] Molecular beam epitaxy is similar to *electron beam epitaxy (EBE)*, in which the wafer is first coated with a layer of dopant material before being placed in a high vacuum. A guided beam of electrons is then fired at the wafer causing the dopant to be driven into it.

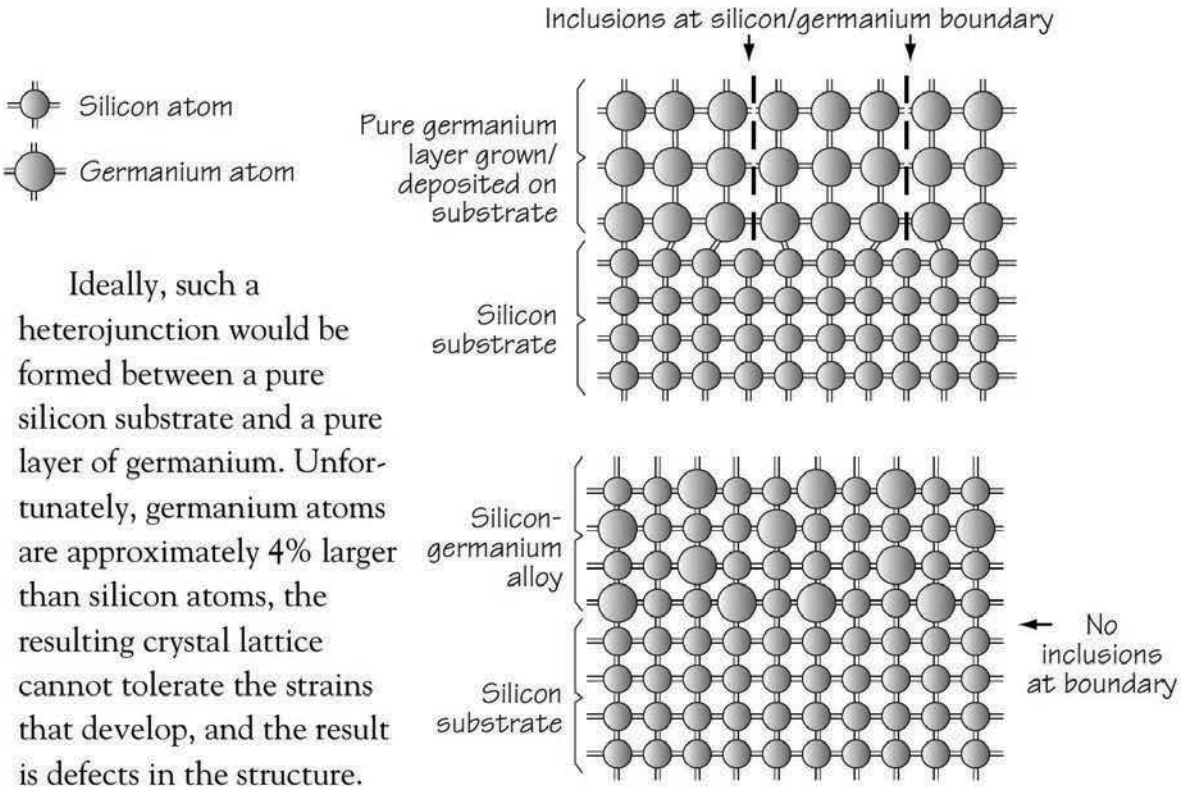


Figure 21-21: Hetrojunction transistors: depositing a layer of pure germanium versus a layer of silicon-germanium alloy

Ideally, such a heterojunction would be formed between a pure silicon substrate and a pure layer of germanium. Unfortunately, germanium atoms are approximately 4% larger than silicon atoms, the resulting crystal lattice cannot tolerate the strains that develop, and the result is defects in the structure. In fact, millions of minute inclusions occur in every square millimeter, which prevents the chip from working. Hence, the solution of growing a layer of silicon-germanium alloy, which relieves the stresses in the crystalline structure, thereby preventing the formation of inclusions (Figure 21-21).

Heterojunctions offer the potential to create transistors that switch as fast, or faster, than those formed using gallium arsenide, but use significantly less power. Additionally, they have the advantage of being able to be produced on existing fabrication lines, thereby preserving the investment and leveraging current expertise in silicon-based manufacturing processes.

Buckyballs and Nanotubes

Prior to the mid-1980s, the only major forms of pure carbon known to us were graphite and diamond. In 1985, however, a third form consisting of spheres formed from 60 carbon atoms (C_{60}) was discovered. Officially known

as Buckminsterfullerine[21]—named after the American architect R. Buckminister Fuller who designed geodesic domes with the same fundamental symmetry—these spheres are more commonly known as "buckyballs." In 2000, scientists with the U.S. Department of Energy's Lawrence Berkeley National Laboratory (Berkeley Lab) and the University of California at Berkeley reported that they had managed to fashion a transistor from a single buckyball.

Sometime later, scientists discovered another structure called the *nanotube*, which is like taking a thin sheet of carbon and rolling it into a tube. Nanotubes can be formed with walls that are only one atom thick. The resulting tube has a diameter of 1 nano (one thousandth of one millionth of a meter).

Nanotubes are an almost ideal material. They are stronger than steel, have excellent thermal stability, and they are also tremendous conductors of heat and electricity. For example, a conductor formed from millions of nanotubes arranged to form a 1cm cross-section could conduct more than 1 billion amps.

In addition to acting as wires, nanotubes can be persuaded to act as transistors—this means that we now have the potential to replace silicon transistors with molecular-sized equivalents at a level where standard semiconductors cease to function. We're only just starting to experiment with nanotube-based transistors, but they can theoretically run at clock speeds of one terahertz or more (one thousand times faster than today's processors).

In the longer-term future, it may be that some integrated circuits will use nanotubes to form both the transistors and the wires linking them together. In the shorter-term, we can expect to see interesting mixtures of nanotubes with conventional technologies. For example, in the summer of 2002, IBM announced an experimental transistor called a *carbon nanotube FET (CNFET)*. This device is based on a field-effect transistor featuring a carbon nanotube measuring only 1.4 nanometers in diameter acting as the channel (the rest of the transistor is formed using conventional silicon and metal processing technologies). It will be a number of years before these devices become commercially available, but they are anticipated to significantly outperform their silicon-only counterparts.[22]

---

[21] Science magazine voted Buckminsterfullerine the "*Molecule of the Year*" in 1991.

[22] As an aside, constructing the masks used to create integrated circuits with 0.06 micron (60 nano) geometries and smaller (especially those featuring esoteric technologies like CNFETs) looks set to be to be a major problem. One company that may have the answer with their nanotechnology-based innovations is Nano Innovation Systems (www.nanoinnovationsys.com).

Diamond Substrates

As was noted in the previous section, there is a constant drive towards smaller, more densely packed transistors switching at higher speeds. Unfortunately, packing the little devils closer together and cracking the whip to make them work faster substantially increases the amount of heat that they generate. Similarly, the increasing utilization of optical interconnect relies on the use of laser diodes, but today's most efficient laser diodes only convert 30% to 40% of the incoming electrical power into an optical output, while the rest emerges in the form of heat. Although each laser diode is relatively small (perhaps as small as only 500 atoms in diameter), their heating effect becomes highly significant when tens of thousands of them are performing their version of Star Wars.

And so we come to *diamond*, which derives its name from the Greek *adamas*, meaning "invincible." Diamond is famous as the hardest substance known, but it also has a number of other interesting characteristics: it is a better conductor of heat at room temperatures than any other material,[23] in its pure form it is a good electrical insulator, it is one of the most transparent materials available, and it is extremely strong and non-corrosive. For all of these reasons, diamond would form an ideal substrate material for multichip modules.[24]

In addition to multichip modules, diamond has potential for a variety of other electronics applications. Because diamond is in the same family of elements as silicon and germanium, it can function as a semiconductor and could be used as a substrate for integrated circuits. In fact, in many ways, diamond would be far superior to silicon: it is stronger, it is capable of withstanding high temperatures, and it is relatively immune to the effects of radiation (the bane of components intended for nuclear and space applications). Additionally, due to diamond's high thermal conductivity, each die would act as its own heat sink and would rapidly conduct heat away. It is believed that diamond-based devices could switch up to 50 times faster than silicon and operate at temperatures over 500°C.

---

[23] Diamond can conduct five times as much heat as copper, which is the second most thermally conductive material known.

[24] Actually, other exotic substrates are also of interest to electronic engineers, including sapphire, which is of particular use in microwave applications.

Chemical Vapor Deposition

Unfortunately, today's integrated circuit manufacturing processes are geared around fabricating large numbers of chips on wafers that can be up to 300 mm in diameter. By comparison, a natural diamond 10 mm in diameter would be considered to be really, REALLY large. It simply wouldn't be cost-effective to take one of these beauties, slice it up, and make diamond-based chips one at a time. Furthermore, if you were to be the proud owner of a large natural diamond, the last thing that would come to mind would be to chop it up into thin slices for electronics applications!

However, there are a number of methods for depositing or growing diamond crystals, one of the most successful being chemical vapor deposition (CVD), which was introduced in the earlier discussions on heterojunction transistors. With this CVD process, microwaves are used to heat mixtures of hydrogen and hydrocarbons into a plasma, out of which diamond films nucleate and form on suitable substrates. Although the plasma chemistry underlying this phenomena is not fully understood, polycrystalline diamond films can be nucleated on a wide variety of materials, including metals such as titanium, molybdenum, and tungsten, ceramics, and other hard materials such as quartz, silicon, and sapphire.

Chemical Vapor Infiltration

CVD processes work by growing layers of diamond directly onto a substrate. A similar, more recent technique—known as *chemical vapor infiltration (CVI)*[25]—commences by placing diamond powder in a mold. Additionally, thin posts, or columns, can be preformed in the mold, and the diamond powder can be deposited around them. When exposed to the same plasma as used in the CVD technique, the diamond powder coalesces into a polycrystalline mass. After the CVI process has been performed, the posts can be dissolved, leaving holes through the diamond for use in creating vias. CVI processes can produce diamond layers twice the thickness of those obtained using CVD techniques at a fraction of the cost.

---

[25] Thanks go to Crystallume, Menlo Park, CA, USA, for the information on their CVD and CVI processes.

Ubiquitous Laser Beams

An alternative, relatively new technique for creating diamond films involves heating carbon with laser beams in a vacuum. Focusing the lasers on a very small area generates extremely high temperatures, which rip atoms away from the carbon and also strip away some of their electrons. The resulting ions fly off and stick to a substrate placed in close proximity. Because the lasers are tightly focused, the high temperatures they generate are localized on the carbon, permitting the substrate to remain close to room temperature. Thus, this process can be used to create diamond films on almost any substrate, including semiconductors, metals, and plastics.

The number of electrons stripped from the carbon atoms varies, allowing their ions to reform in *nanophase* diamond structures that have never been seen before. Nanophase materials are a new form of matter that was only discovered relatively recently, in which small clusters of atoms form the building blocks of a larger structure. These structures differ from those of naturally occurring crystals, in which individual atoms arrange themselves into a lattice. In fact, it is believed that it may be possible to create more than thirty previously unknown forms of diamond using these techniques.

The Maverick Inventor

Last but not least, in the late 1980s, a maverick inventor called Ernest Nagy[26] invented a simple, cheap, and elegant technique for creating thin diamond films. Nagy's process involves treating a soft pad with diamond powder, spinning the pad at approximately 30,000 revolutions per minute, and maintaining the pad in close contact with a substrate. Although the physics underlying the process is not fully understood, diamond is transferred from the pad to form a smooth and continuous film on the substrate. The diamond appears to undergo some kind of phase transformation, changing from a cubic arrangement into a hexagonal form with an unusual structure. Interestingly enough, Nagy's technique appears to work with almost any material on almost any substrate!

---

[26] Nagy, whose full name is Ernest Nagy de Nagybaczon, was born in 1942 in Hungary. He left as a refugee in the 1956 uprising and now lives in England.

The requirement for Single-Crystal Diamond

All of the techniques described above result in films that come respectfully close, if not equal, to the properties of natural diamond in such terms as heat conduction. Thus, these films are highly attractive for use as substrates in multichip modules. However, the unusual diamond structures that are created fall short of the perfection required for them to be used as a substrate suitable for the fabrication of transistors.

Substrates for integrated circuits require the single, large crystalline structures found only in natural diamond. Unfortunately, there are currently no known materials onto which a single-crystal diamond layer will grow, with the exception of single crystal diamond itself (which sort of defeats the point of doing it in the first place). The only answer appears to be to modify the surface of the substrate onto which the diamond layer is grown, and many observers believe that this technology may be developed in the near future. If it does prove possible to create consistent, single-crystal diamond films, then, in addition to being *"a girl's best friend,"* diamonds would also become *"an electronic engineer's biggest buddy."*

Chip-On-Chip (COC)

The intra-chip connections linking bare die on a multichip module are a source of fairly significant delays. One obvious solution is to mount the die as closely together as possible, thereby reducing the lengths of the tracks and the delays associated with them. However, each die can have only eight others mounted in close proximity on a 2D substrate. The solution is to proceed into three dimensions. Each die is very thin and, if they are mounted on top of each other, it is possible to have over a hundred die forming a 3D cube (Figure 21-22).

One problem with this *chip-on-chip (COC)* technique is the amount of heat that is generated, which drastically affects the inner layers forming the cube. This problem could be eased by constructing the die out of diamond as discussed above. Another problem with traditional techniques is that any tracks linking the die must be connected down the outer surfaces of the cube. The result is that the chip-on-chip technique has typically been restricted to applications utilizing identical die with regular structures. For example, the most common application to date has been large memory devices constructed by stacking SRAM or DRAM die on top of each other.

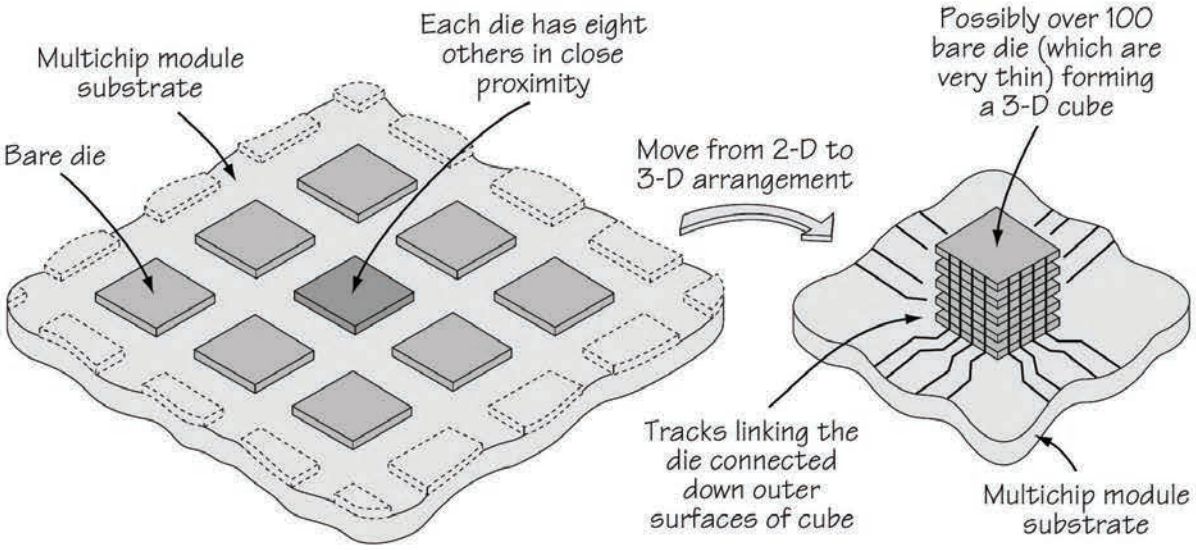


Figure 21-22: Chip-on-chip: die organized as 3D cubes

A relatively new technique which may serve to alleviate the problem of chip-on-chip interconnect is a process for creating vias through silicon substrates. Experiments have been performed in which aluminum "blobs" are placed on the surface of a silicon substrate and, by means of a gradient furnace, the aluminum migrates through the silicon providing vias from one side to the other (Figure 21-23).

Another technique more in keeping with the times is to create the vias by punching the aluminum through the silicon by means of a laser. These developments pave the way for double-sided silicon substrates with chips and interconnect on both sides. Additionally, they offer strong potential for interconnecting the die used in chip-on-chip structures.

Conductive Adhesives

Many electronics fabrication processes are exhibiting a trend towards mechanical simplicity with underlying sophistication in materials technology. A good example of this trend is illustrated by conductive, or *anisotropic*, adhesives, which contain minute particles of conductive material.

These adhesives find particular application with the flipped-chip techniques used to mount bare die on the substrates of hybrids, multichip modules, or circuit boards. The adhesive is screen printed onto the substrate at the site

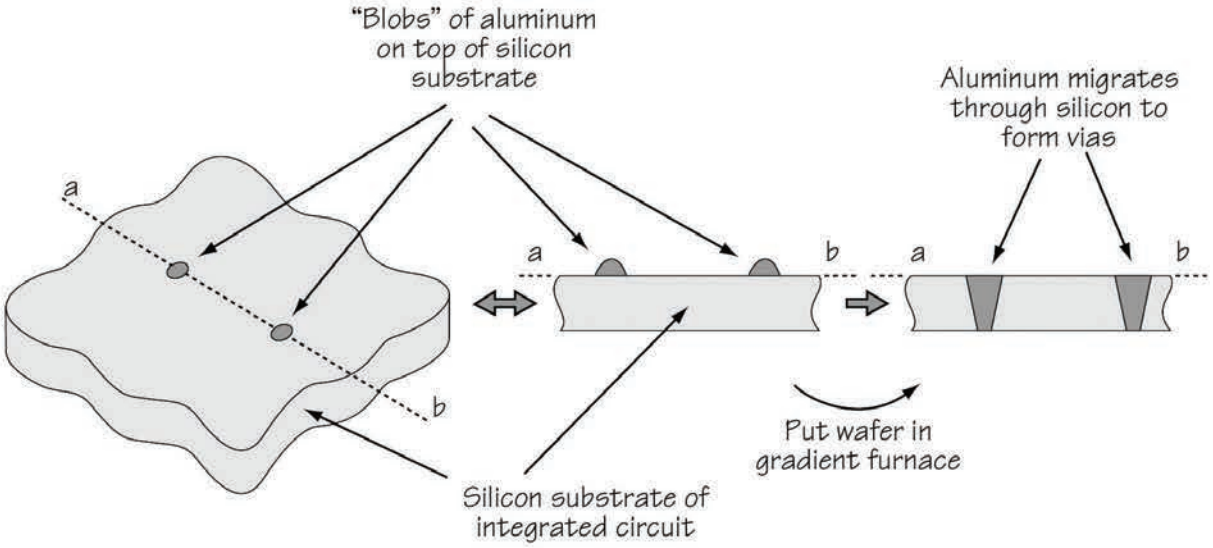


Figure 21-23: Chip-on-chip: creating vias through silicon chips

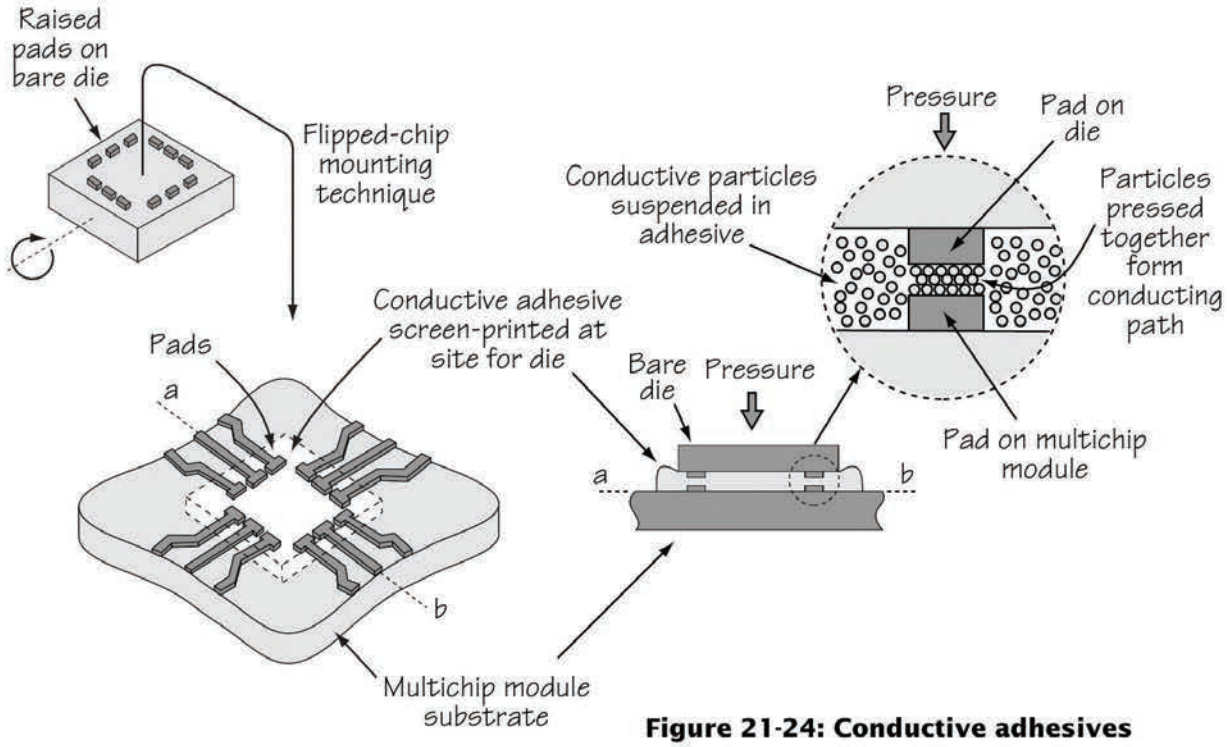


Figure 21-24: Conductive adhesives

where the die is to be located, the die is pressed into the adhesive, and the adhesive is cured using a combination of temperature and pressure (Figure 21-24).

The beauty of this scheme is that the masks used to screen print the adhesive do not need to be too complex, and the application of the adhesive does not need to be excessively precise, because it can be spread across all of the component pads. The conducting particles are only brought in contact with each other at the sites where the raised pads on the die meet their corresponding pads on the substrate, thereby forming good electrical connections.

The original conductive adhesives were based on particles such as silver. But, in addition to being expensive, metals like silver can cause electron migration problems at the points where they meet the silicon substrates. Modern equivalents are based on organic metallic particles, thereby reducing these problems.

In addition to being simpler and requiring fewer process steps than traditional methods, the conductive adhesive technique removes the need for solder, whose lead content is beginning to raise environmental concerns.

Superconductors

One of the "Holy Grails" of the electronics industry is to have access to conductors with zero resistance to the flow of electrons, and for such conductors, known as *superconductors*, to operate at room temperatures. As a concept, superconductivity is relatively easy to understand: consider two sloping ramps into which a number of pegs are driven. In the case of the first ramp, the pegs are arranged randomly across the surface, while in the second the pegs are arranged in orderly lines. Now consider what happens when balls are released at the top of each surface (Figure 21-25).

In the case of the randomly arranged pegs, the ball's progress is repeatedly interrupted, while in the case of the pegs arranged in orderly lines, the ball slips through *"like water off a duck's back."* Although analogies are always suspect (and this one doubly so), the ramps may be considered to represent conducting materials, the gravity accelerating the balls takes on the role of voltage differentials applied across the ends of the conductors, the balls play the part of electrons, and the pegs portray atoms.

The atoms in materials vibrate due to the thermal energy contained in the material: the higher the temperature, the more the atoms vibrate. An ordinary

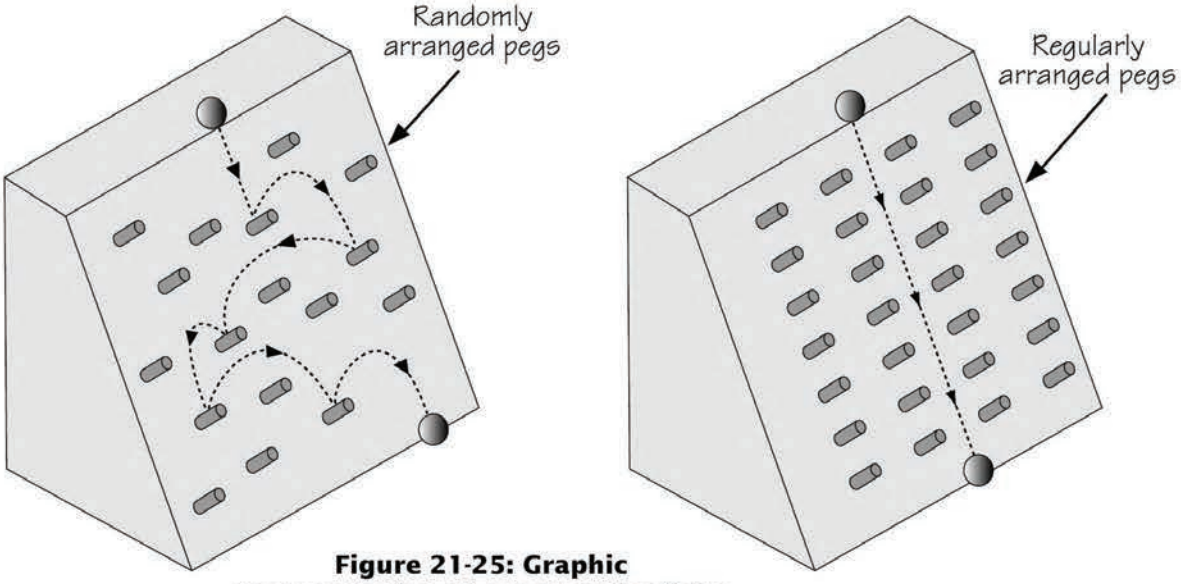


Figure 21-25: Graphic representation of superconductivity

conductor's electrical resistance is caused by these atomic vibrations, which obstruct the movement of the electrons forming the current. Using the Kelvin,[27] or absolute, scale of temperature, 0 K (corresponding to –273°C) is the coldest possible temperature and is known as absolute zero. If an ordinary conductor were cooled to a temperature of absolute zero, atomic vibrations would cease, electrons could flow without obstruction, and electrical resistance would fall to zero. A temperature of absolute zero cannot be achieved in practice, but some materials exhibit superconducting characteristics at higher temperatures.[28]

In 1911, the Dutch physicist Heike Kamerlingh Onnes discovered superconductivity in mercury at a temperature of approximately 4K (–269°C). Many other superconducting metals and alloys were subsequently discovered but, until 1986, the highest temperature at which superconducting properties were achieved was around 23K (-250°C) with the niobium-germanium alloy (Nb_3Ge).

---

27 Invented by the British mathematician and physicist William Thomas, first Baron of Kelvin.

28 If the author were an expert in superconductivity, this is the point where he might be tempted to start muttering about *"Correlated electron movements in conducting planes separated by insulating layers of mesoscopic thickness, under which conditions the wave properties of electrons assert themselves and electrons behave like waves rather than particles."* But he's not, so he won't.

In 1986, Georg Bednorz and Alex Müller discovered a metal oxide that exhibited superconductivity at the relatively high temperature of 30K (–243°C). This led to the discovery of ceramic oxides that superconduct at even higher temperatures. In 1988, an oxide of thallium, calcium, barium and copper ($Tl^2Ca^2Ba_2Cu_3O_{10}$) displayed superconductivity at 125K (–148°C), and, in 1993, a family based on copper oxide and mercury attained superconductivity at 160K (–113°C). These "high-temperature" superconductors are all the more noteworthy because ceramics are usually extremely good insulators.

Like ceramics, most organic compounds are strong insulators; however, some organic materials known as *organic synthetic metals* do display both conductivity and superconductivity. In the early 1990s, one such compound was shown to superconduct at approximately 33K (–240°C). Although this is well below the temperatures achieved for ceramic oxides, organic superconductors are considered to have great potential for the future.

New superconducting materials are being discovered on a regular basis, and the search is on for room temperature superconductors, which, if discovered, are expected to revolutionize electronics as we know it. In fact, in the summer of 2002 while penning the final words in this chapter, the author was made aware of a new class of polymer materials called *ultraconductors*, which exhibit superconducting properties at ambient (room) temperature. These materials are claimed to conduct electricity at least 100,000 times better than gold, silver, or copper! Unfortunately, the author has been bound to secrecy and may not reveal any more at this time.

Nanotechnology

Nanotechnology is an elusive term that is used by different research and development teams to refer to whatever it is that they're working on at the time. However, regardless of their particular area of interest, nanotechnology always refers to something extremely small; for example, motors and pumps the size of a pinhead, which are created using similar processes to those used to fabricate integrated circuits. In fact, around the beginning of 1994, one such team unveiled a miniature model car which was smaller than a grain of short-grain rice. This model contained a micro-miniature electric motor, battery, and gear train, and was capable of traversing a fair-sized room (though presumably not on a shag-pile carpet).

In 1959, the legendary American physicist Richard Feynman gave a visionary talk, in which he described the possibility by which sub-microscopic computers could perhaps be constructed. Feynman's ideas have subsequently been extended to become one of the more extreme branches of nanotechnology featuring micro-miniature products that assemble themselves! The theory is based on the way in which biological systems operate. Specifically, the way in which *enzymes*[29] act as *biological catalysts*[30] to assemble large, complex molecules from smaller molecular building blocks.

Back to the Water Molecule

Before commencing this discussion, it is necessary to return to the humble water molecule.[31] As you may recall, water molecules are formed from two hydrogen atoms and one oxygen atom, all of which share electrons between themselves. However, the electrons are not distributed equally, because the oxygen atom is a bigger, more robust fellow which grabs more than its fair share (Figure 21-26).

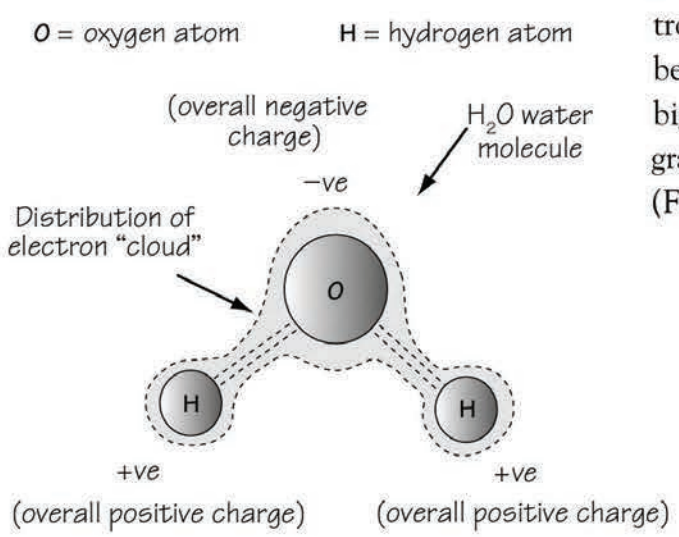


Figure 21-26: Distribution of electrons in a water molecule

The angle formed between the two hydrogen atoms is 105°. This is because, of the six electrons that the oxygen atom owns, two are shared with the hydrogen atoms and four remain the exclusive property of the oxygen. These four huddle together on one side of the oxygen atom and put

---

[29] Enzymes are complex proteins that are produced by living cells and catalyze biochemical reactions at body temperatures.

[30] A catalyst is a substance that initiates a chemical reaction under different conditions (such as lower temperatures) than would otherwise be possible. The catalyst itself remains unchanged at the end of the reaction.

[31] Water molecules were introduced in Chapter 2.

"pressure" on the bond angle. The bond angle settles on 105° because this is the point where the pressure from the four electrons is balanced by the natural repulsion of the two positively charged hydrogen atoms (similar charges repel each other).

The end result is that the oxygen atom has an overall negative charge, while the two hydrogen atoms are left feeling somewhat on the positive side. This unequal distribution of charge means that the hydrogen atoms are attracted to anything with a negative bias-for example, the oxygen atom of another water molecule. Although the strength of the resulting bond, known as a *hydrogen bond*, is weaker than the bond between the hydrogen atom and its "parent" oxygen atom, it is still quite respectable.

When water is cooled until it freezes, its resulting crystalline structure is based on these hydrogen bonds. Even in its liquid state, the promiscuous, randomly wandering water molecules are constantly forming hydrogen bonds with each other. These bonds persist for a short time until another water molecule clumsily barges into them and knocks them apart. From this perspective, a glass of water actually contains billions of tiny ice crystals that are constantly forming and being broken apart again.

Imagine a Soup

However, we digress. Larger molecules can form similar electrostatic bonds with each other. Imagine a "soup" consisting of large quantities of many different types of molecules, two of which, M_a and M_b, may be combined to form larger molecules of type M_{ab} (Figure 21-27).

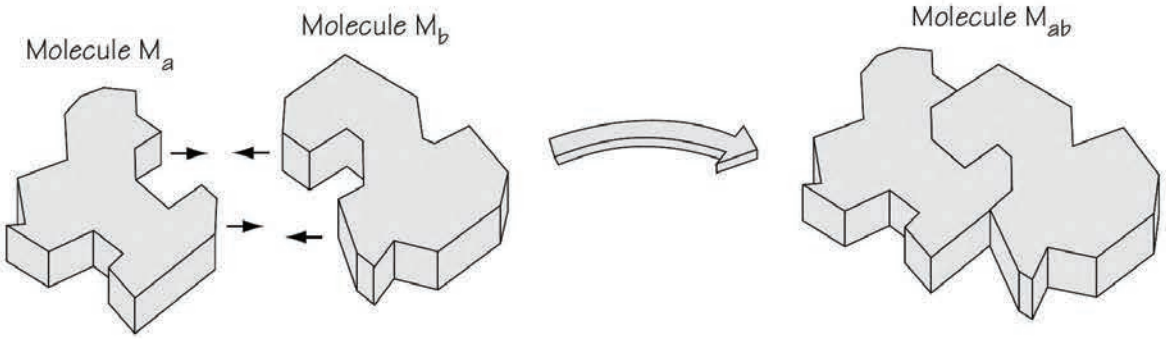


Figure 21-27: Combining molecules M_a and M_b to form M_{ab}

This is similar in concept to two pieces of a jigsaw, which will only fit together if they are in the correct orientation to each other. Similarly, M_a and M_b will only bond to form M_{ab} if they are formally presented to each other in precisely the right orientation. However, the surfaces of the molecules are extremely complex three-dimensional shapes, and achieving the correct orientation is a tricky affair. Once the molecules have been brought together their resulting bonds are surprisingly strong, but the chances of the two molecules randomly achieving exactly the correct orientation to form the bonds are extremely small.

It is at this point of the story that enzymes reenter the plot. There are numerous enzymes, each dedicated to the task of "matchmaking" for two of their favorite molecules. The surface of an enzyme is also an extremely complex three-dimensional shape, but it is much larger than its target molecules and has a better chance of gathering them up. The enzyme floats around until it bumps into a molecule of type M_a to which it bonds. The enzyme then continues on its trek until it locates a molecule of type M_b. When the enzyme bonds to molecule M_b, it orientates it in exactly the right way to complete the puzzle with molecule M_a (Figure 21-28).

The bonds between M_a and M_b are far stronger than their bonds to the enzyme. In fact, as soon as these bonds are formed, the enzyme is actually repelled by the two little lovebirds and promptly thrusts M_{ab} away. However, the enzyme immediately forgets its pique, and commences to search for two more molecules (some enzymes can catalyze their reactions at the rate of half a million molecules per minute).

The saga continues, because another, larger enzyme may see its task in life as bringing M_{ab} together with yet another molecule M_{cd}. And so it continues, onwards and upwards, until the final result, whatever that may be, is achieved.

As our ability to create "designer molecules" increases, it becomes increasingly probable that we will one day be able to create "designer enzymes." This would enable us to mass-produce "designer proteins" that could act as alternatives to semiconductors (see also the *Protein Switches and Memories* topic earlier in this chapter). As one of the first steps along this path, a process could be developed to manufacture various proteins that could then be bonded to a substrate or formed into three-dimensional blocks for optical memory

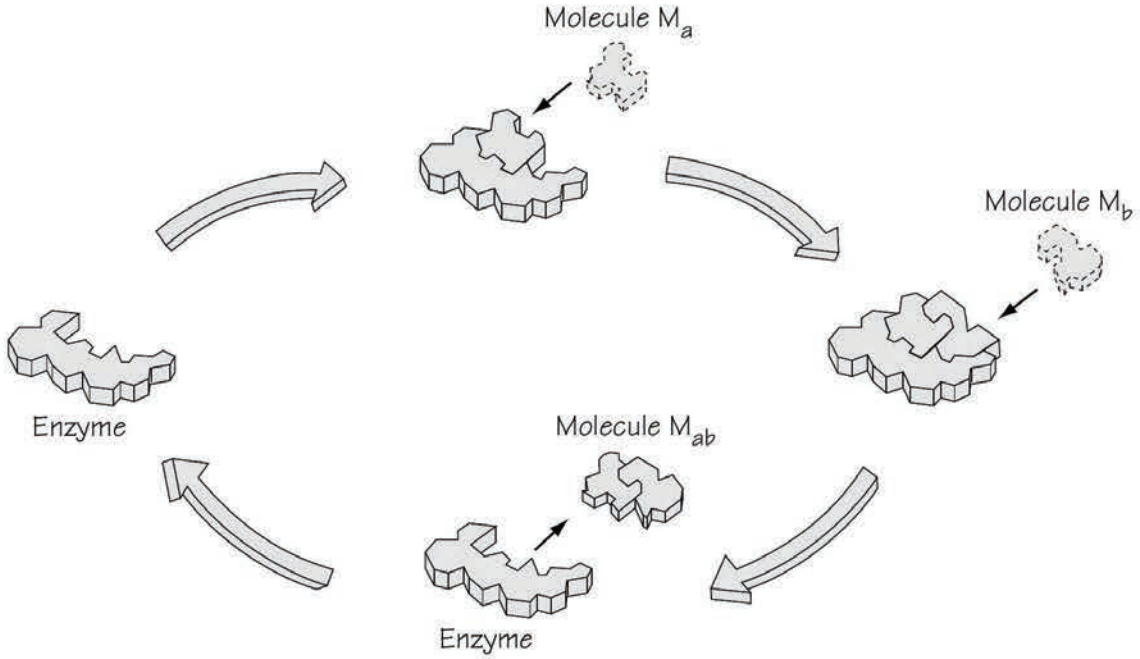


Figure 21-28: Using an enzyme to form molecule M_{ab}

applications. At a more sophisticated level, it may be possible for such a process to directly create the requisite combinations of proteins as self-replicating structures across the face of a substrate.

However, the possibilities extend far beyond the mass-production of proteins. It is conceivable that similar techniques could be used to assemble non-organic structures such as microscopic electromechanical artifacts. All that would be required (he said casually) would be for the individual components to be shaped in such a way that naturally occurring electrostatic fields would cause them to form bonds when they were brought together with their soul mates. In fact, this is one step along the path towards molecular-sized robots known as *nanobots*. Taken to extremes, the discipline of electronics in the future may not involve the extreme temperatures, pressures, and noxious chemicals that are in vogue today. Instead, electronics may simply involve "cookbook" style recipes; for example, the notes accompanying an electronics course in 2050 AD may well read as follows:

Intermediate Electronics (Ages 12 to 14)
Supercomputers 101

Instructions for creating a micro-miniature massively parallel supercomputer.*

a) Obtain a large barrel.

b) In your barrel, mix two parts water and one part each of chemicals A, B, C . . .

c) Add a pinch of nanobot-mix (which you previously created in Nanobots 101).

d) Stir briskly for one hour with a large wooden spoon.

Congratulations, you will find your new supercomputers in the sediment at the bottom of the barrel. Please keep one teaspoon of these supercomputers for your next lesson.

*These instructions were reproduced from *Bebop to the Boolean Boogie*, 50th edition, 2050, the most popular electronics book in the history of the universe!

Of course, some of this is a little far-fetched (with the hopeful exception of the references to *Bebop to the Boolean Boogie*). However, for what it's worth, the author would bet his ex-wife's life savings that this type of technology will occur one day, and also that it will be here sooner than you think!

Again, the Mind Boggles

In reality, we've only touched on a very few of the myriad ideas that are out there running wild and free. For example, one area that is currently seen as a growth industry is that of *MicroElectroMechanical Systems (MEMS)* and their optical counterparts OMEMS. These refer to devices that contain both electrical, mechanical, and—in the case of OMEMS—optical elements and are physically very small (sometimes measuring only a few millionths of a meter in size). MEMS can be used to create microscopic sensors to monitor the surrounding environment and actuators that can modify the environment with great precision, while OMEMS find use in communications systems.

As an example, in early 2001, biophysicists at the Hungarian Academy of Sciences devised a way to create microscopic machines that are constructed and operated by light. In one case they created gears that are each less than 1/5,000 of an inch (5 μm) in diameter. The rotors spin when illuminated by a low-power laser as photons hit their flanges. Such devices could pump materials across miniature chemical arrays.

There are several theories as to how life developed on Earth, including "seeding" by meteors from outer space[32] or random combinations of atoms forming simple molecules, which in turn formed rudimentary proteins, then . . . and so it goes.

Maybe a combination of both of these theories is true. However, to test the *"random combination"* theory, scientists decided to replicate the conditions on Earth billions of years ago before life developed, with the intention of duplicating the happy event. So, they took a large glass jar, added some water, filled the rest with the gases that were around at the time, and applied electrical discharges to the mixture to simulate the effects of lightning strikes. Much to their surprise . . . nothing happened.

Obviously this was something of a downer. While they were all scratching their heads, a tentative hand was raised in the air, and a high-pitched voice belonging to the tea boy said "Excuuussse me, but what about adding rocks?" After they had slapped him round the back of the head for his insolence, they thought "What the heck, let's give it a whirl." So, they added a mixture of rocks, zapped everything with electric discharges again, and, lo and behold, they found amino acids and simple proteins.

It turned out that when certain shales are split, their rough surfaces form exactly the right templates to bond atoms and molecules together to create the rudimentary building blocks of life!

Now, this does not either prove or disprove the existence of God. However, there is almost certainly something somewhere, because everything fits together so perfectly. At the very least, if there is a God, then it certainly serves to confirm the fact that "*He/She works in mysterious ways, His/Her wonders to perform.*"

[32] Scientists have found amino acids (the basic building blocks of proteins, and therefore life itself) in a number of meteors.

Small as they are, MEMS and OMEMS are still huge on the molecular scale. In early 2001, the University of Illinois Beckman Institute for Advanced Science and Technology reported research with organic molecules like tiny mechanical switches. These molecular switches are only attached to the substrate by a single atom and they can spin as fast as 100 trillion times a second, which provides the potential for switching arrays running at 100 terahertz!

And just to illustrate the range of things that are being investigated, biomedical engineers at the Georgia Institute of Technology and Emory University are working on linking electronics to living neurons to create incredibly sophisticated neural networks.

Summary

The potpourri of technologies introduced here have been offered for your delectation and delight. Some of these concepts may appear to be a little on the wild side, and you certainly should not believe everything that you read or hear. On the other hand, you should also be careful not to close your mind, even to seemingly wild and wacky ideas, in case something sneaks up behind you and bites you on the ####.[33]

As the *Prize* said in *Where is Earth* by Robert Sheckley: *"Be admiring but avoid the fulsome, take exception to what you don't like, but don't be stubbornly critical; in short, exercise moderation except where a more extreme attitude is clearly called for."*

And so, with our lower lips quivering and little tears rolling down our cheeks, we come to the close of this, the penultimate chapter. But turn that frown upside down into a smile, because Chapter 22 (on the CD accompanying this book) is going to have you sitting on the edge of your seat, and there are still oodles of yummy appendices containing enough mouth-watering information to form a book in their own right. As that great British Prime Minister Winston Spencer Churchill (1874–1965) would have said: *"Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning."*[34]

---

[33] Arsek no questions!

[34] Speech at the Lord Mayor's Day Luncheon, London, England (November 10, 1942).

Appendix



Assertion-Level Logic

The purpose of a circuit diagram is to convey the maximum amount of information in the most efficient fashion. One aspect of this is assigning meaningful names to wires; for example, naming a wire system_reset conveys substantially more information than calling it big_boy. Another consideration is that the signals carried by wires may be *active-high* or *active-low* (see sidebar).

Consider a portion of a circuit containing a tri-state buffer with an active-low ~enable input (Figure A-1).

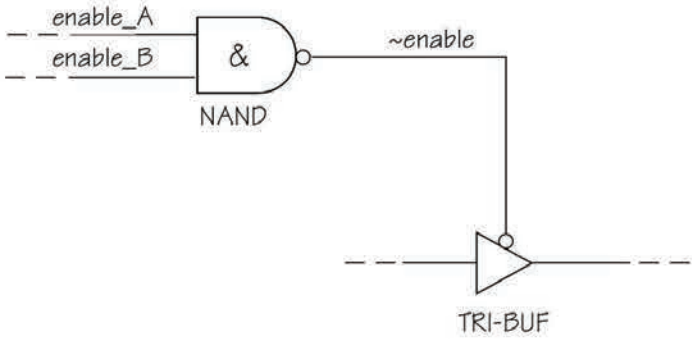


Figure A-1. Naming active-low signals

Beware — here be dragons!

The definitions of *active-high* and *active-low* are subject to confusion. Some academics (and even text books) define an active-low signal as one whose asserted (TRUE or logic 1) state is at a lower voltage level than its unasserted (FALSE or logic 0) state. However, this definition serves only to cause confusion when combined with negative logic implementations (see also Appendix B).

Engineers in the trenches generally take the position that an active-high state is one whose active state is considered to be TRUE or logic 1, while an active-low state is one whose active state is considered to be FALSE or logic 0. These definitions allow all forms of logic—including the assertion-level logic introduced in this chapter—to be represented without any confusion, regardless of whether positive or negative logic implementations are employed. These manly-man definitions are the ones used throughout this book. However, when using the terms active-high and active-low in discussions with other folks, you are strongly advised to make sure that you all understand them to mean the same thing before you find yourself up to your ears in alligators.

Instead of a tilde, some designers prefer to use an exclamation mark '!' nicknamed a *shriek* to indicate active-low signals—for example, !enable. Alternatively, designers may prefix or postfix active-low signal names with a special letter; for example, Nenable or enableB. Yet another technique is to draw a horizontal line, or bar, over the name, but this is not recommended for a number of reasons. First, signal names with bars over them are difficult to replicate in textual form on a computer. Second, bars are often used to indicate negations in Boolean equations. In this latter case it would be impossible to determine whether a bar formed an integral part of a signal's name or if it was intended to indicate that the signal should be negated. Tilde characters may also be used to indicate complementary outputs; for example, the q and ~q outputs from a latch or a flip-flop.

To convey as much information as possible, it is preferable to indicate the nature of an active-low signal in its name. One method of achieving this is to prefix the name with a tilde character '~'; hence, ~enable (see sidebar). When both the enable_A and enable_B signals are set to their active-high states, the ~enable signal is driven to its active-low state and the tri-state buffer is enabled.

It is important to note that the active-low nature of ~enable is not decided by the NAND. The only thing that determines whether a signal is active-high or active-low is the way in which it is used by any target gates, such as the tri-state buffer in this example. Thus, active-low signals can be generated by ANDs and ORs as easily as NANDs and NORs.

The problem with the standard symbols for BUF, NOT, AND, NAND, OR, and NOR is that they are tailored to reflect operations based on active-high signals being presented to their inputs. To address this problem, special *assertion-level* logic symbols can be used to more precisely indicate the function of gates with active-low inputs. Consider how we might represent a NOT gate used to invert an ~enable signal into its enable counterpart (Figure A-2).



Figure A-2. Standard versus assertion-level NOT symbols

Both symbols indicate an inversion, but the assertion-level symbol is more intuitive because it reflects the fact that an active-low input is being transformed into an active-high output. In both cases the bobbles on the symbols indicate the act of inversion. One way to visualize this is that the symbol for a NOT has been pushed into a symbol for a BUF until only its bobble remains visible (Figure A-3).

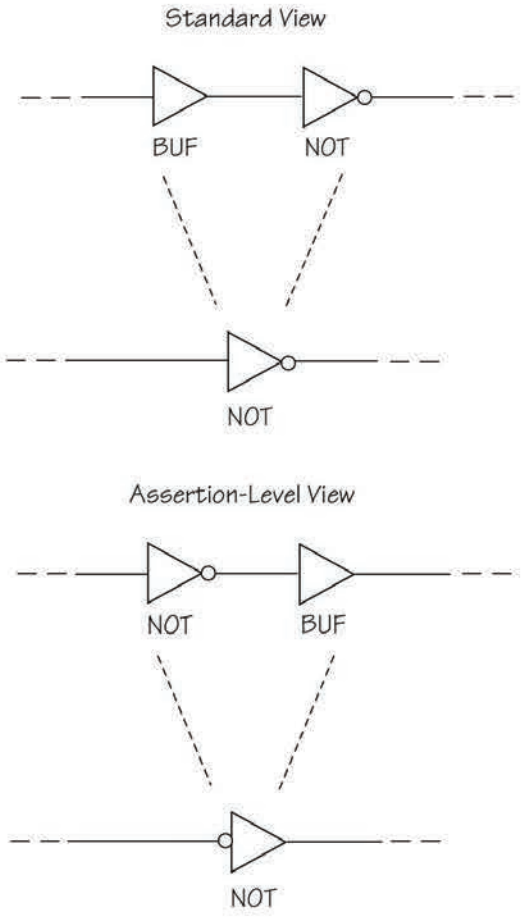


Figure A-3. One way to visualize standard versus assertion-level NOT symbols

In the real world, both standard and assertion-level symbols are implemented using identical logic gates. Assertion-level logic does not affect the final implementation, but simply offers an alternative way of viewing things. Visualizing bobbles as representing inverters is a useful technique for handling more complex functions. Consider a variation on our original circuit in which the *~enable_A* and *~enable_B* signals are active-low and the tri-state buffer has an active-high *enable* input (Figure A-4).

The purpose of this circuit is to set the *enable* signal to its active-high state if either of the *~enable_A* or *~enable_B* signals are in their active-low states.

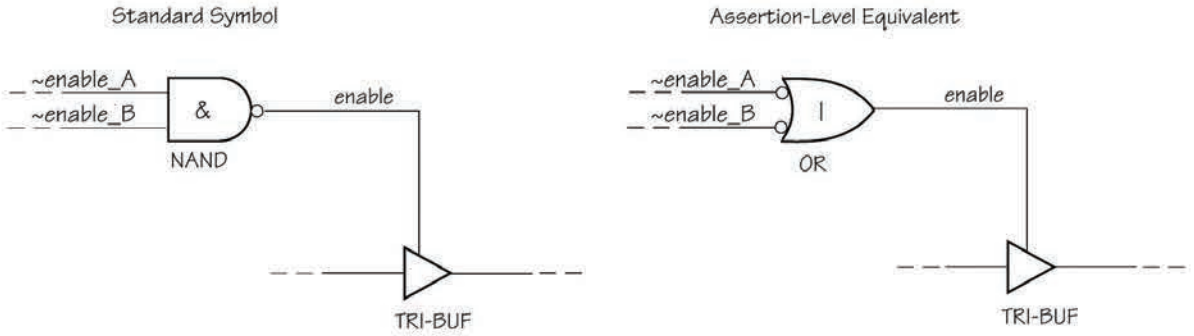


Figure A-4. Standard NAND versus assertion-level OR symbols

Remember that both of these circuit representations are functionally identical (you can easily prove this by drawing out their truth tables). Once again, however, the assertion-level representation is the more intuitive, especially for someone who is unfamiliar with the function of the circuit. This is because the assertion-level symbol unambiguously indicates that *enable* will be set to logic 1 if either of the *~enable_A* or *~enable_B* signals are presented with logic *0*s.

Any standard primitive gate symbol can be transformed into its assertion-level equivalent by inverting all of its inputs and outputs, and exchanging any & (AND) operators for | (OR) operators (and vice versa). In fact, a quick review of Chapter 9 reveals that these steps are identical to those used in a DeMorgan Transformation. Thus, assertion-level symbols may also be referred to as DeMorgan equivalent symbols. The most commonly used assertion-level symbols are those for BUF, NOT, AND, NAND, OR, and NOR (Figure A-5).

Note that the assertion-level symbols for XOR and XNOR are identical to their standard counterparts. The reason for this is that if you take an XOR and invert its inputs and outputs, you end up with an XOR again; similarly for an XNOR. A little experimentation with their truth tables will quickly reveal the reason why.

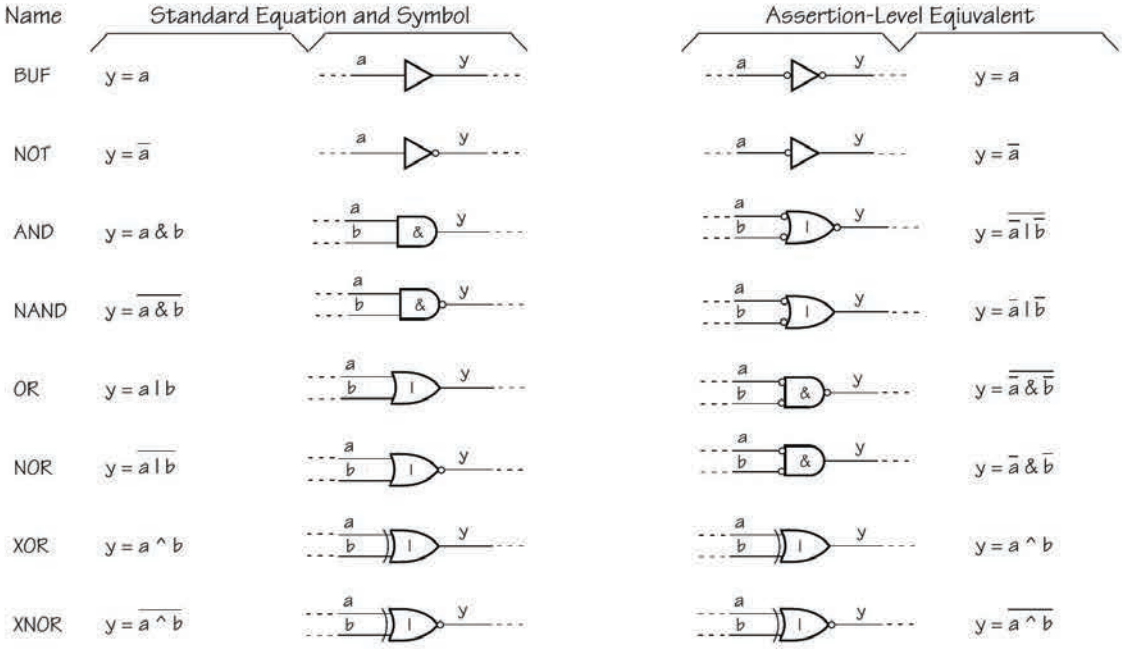


Figure A-5. Commonly used assertion-level symbols

Appendix

B

Positive Logic versus Negative Logic

The terms *positive logic* and *negative logic* refer to two conventions that dictate the relationship between logical values and the physical voltages used to represent them. Unfortunately, although the core concepts are relatively simple, fully comprehending all of the implications associated with these conventions requires an exercise in lateral thinking sufficient to make even a strong man break down and weep!

Before plunging into the fray, it is important to understand that logic 0 and logic 1 are always equivalent to FALSE and TRUE, respectively.[1] The reason these terms are used interchangeably is that digital functions can be considered to represent either *logical* or *arithmetic* operations (Figure B-1).

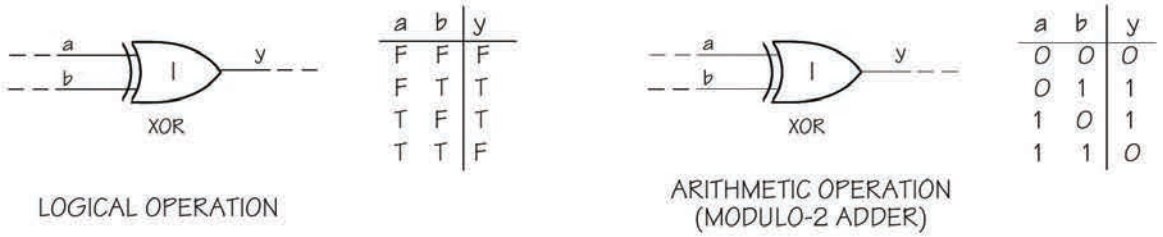


Figure B-1. Logical versus arithmetic views of a digital function

Having said this, it is generally preferable to employ a single consistent format to cover both cases, and it is easier to view logical operations in terms of 0s and 1s than it is to view arithmetic operations in terms of Fs and Ts. The key point to remember as we go forward is that, by definition, logic 0 and logic 1 are logical concepts that have no direct relationship to any physical values.

---

[1] Unless you're really taking a walk on the wild side, in which case all bets are off.

Physical to Abstract Mapping (NMOS Logic)

Let's gird up our loins and meander our way through the morass one step at a time . . . The process of relating logical values to physical voltages begins by defining the frames of reference to be used. One absolute frame of reference is provided by truth tables, which are always associated with specific functions (Figure B-2).

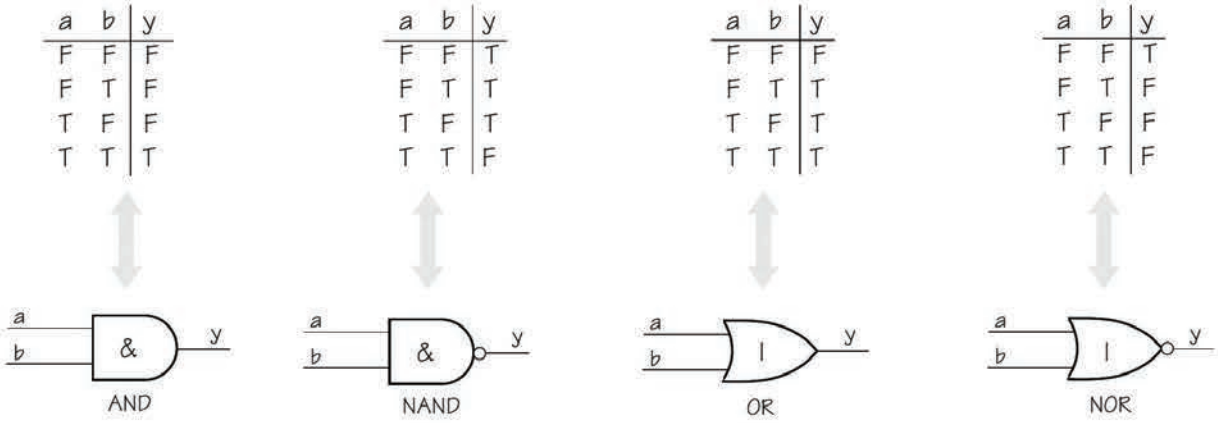


Figure B-2. Absolute relationships between truth tables and functions

Another absolute frame of reference is found in the physical world, where specific voltage levels applied to the inputs of a digital function cause corresponding voltage responses on the outputs. These relationships can also be represented in truth table form. Consider a logic gate constructed using only NMOS transistors (Figure B-3).

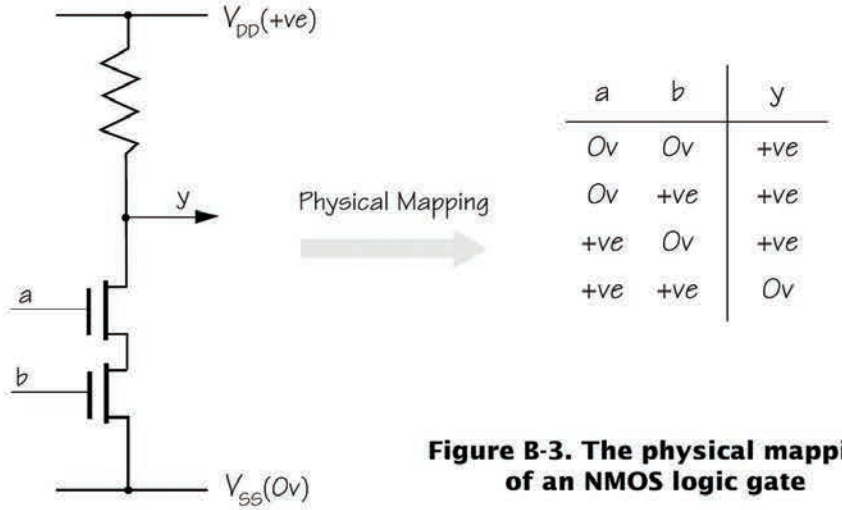


Figure B-3. The physical mapping of an NMOS logic gate

With NMOS transistors connected as shown in Figure B-3, an input connected to the more negative V_{SS} turns that transistor OFF, and an input connected to the more positive V_{DD} turns that transistor ON. The final step is to define the mapping between the physical and abstract worlds; either 0v is mapped to FALSE and +ve is mapped to TRUE, or vice versa (Figure B-4).

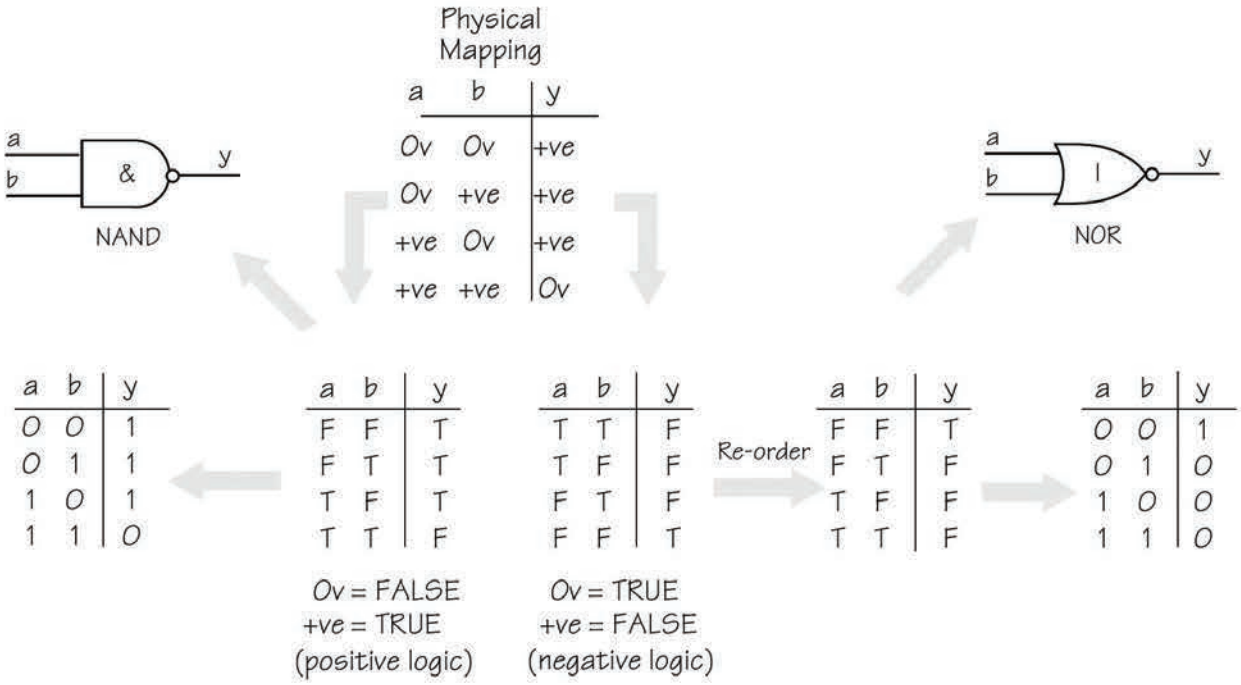


Figure B-4. The physical to abstract mapping of an NMOS logic gate

Using the positive logic convention, the more positive potential is considered to represent TRUE[2] and the more negative potential is considered to represent FALSE. By comparison, using the negative logic convention, the more negative potential is considered to represent TRUE[3] and the more positive potential is considered to represent FALSE. Thus, this circuit may be considered to be performing either a NAND function in positive logic or a NOR function in negative logic. (Are we having fun yet?)

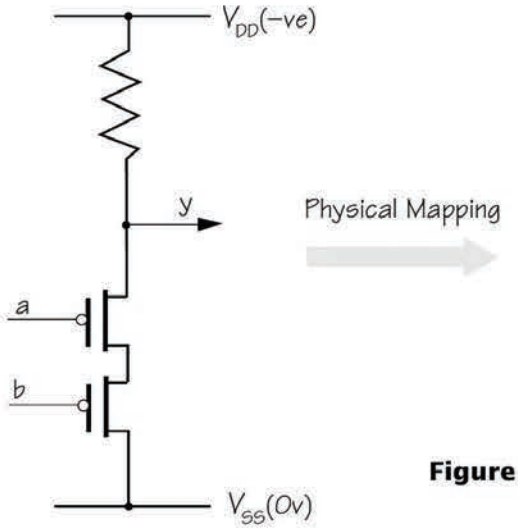
---

[2] Hence, positive logic is also known as *positive-true*.

[3] Hence, negative logic is also known as *negative-true*.

Physical to Abstract Mapping (PMOS Logic)

From the previous example it would appear that positive logic is the more intuitive as it is easy to relate logic 0 to 0v (no volts) and logic 1 to +ve (presence of volts). On this basis one may wonder why negative logic was ever invented. The answer to this is, as are so many things, rooted in history. When the MOSFET technology was originally developed, PMOS transistors were easier to manufacture and were more reliable than their NMOS counterparts. Thus, the majority of early MOSFET-based logic gates were constructed from combinations of PMOS transistors and resistors. Consider a logic gate constructed using only PMOS transistors (Figure B-5).



a	b	y
0v	0v	–ve
0v	–ve	–ve
–ve	0v	–ve
–ve	–ve	0v

Figure B-5. The physical mapping of a PMOS logic gate

Circuits constructed using the original PMOS transistors typically made use of a negative power supply (that is, a power supply with a 0V terminal and a negative (–ve) terminal). With PMOS transistors connected as shown above, an input connected to the more positive V_{SS} turns that transistor OFF, and an input connected to the more negative potential V_{DD} turns that transistor ON. Once again, the final step is to define the mapping between the physical and abstract worlds; either 0v is mapped to FALSE and –ve is mapped to TRUE, or vice versa (Figure B-6).

Thus, this circuit may be considered to be performing either a NOR function in positive logic or a NAND function in negative logic. In this case, negative logic is the more intuitive as it is easy to relate logic 0 to 0v (no volts)

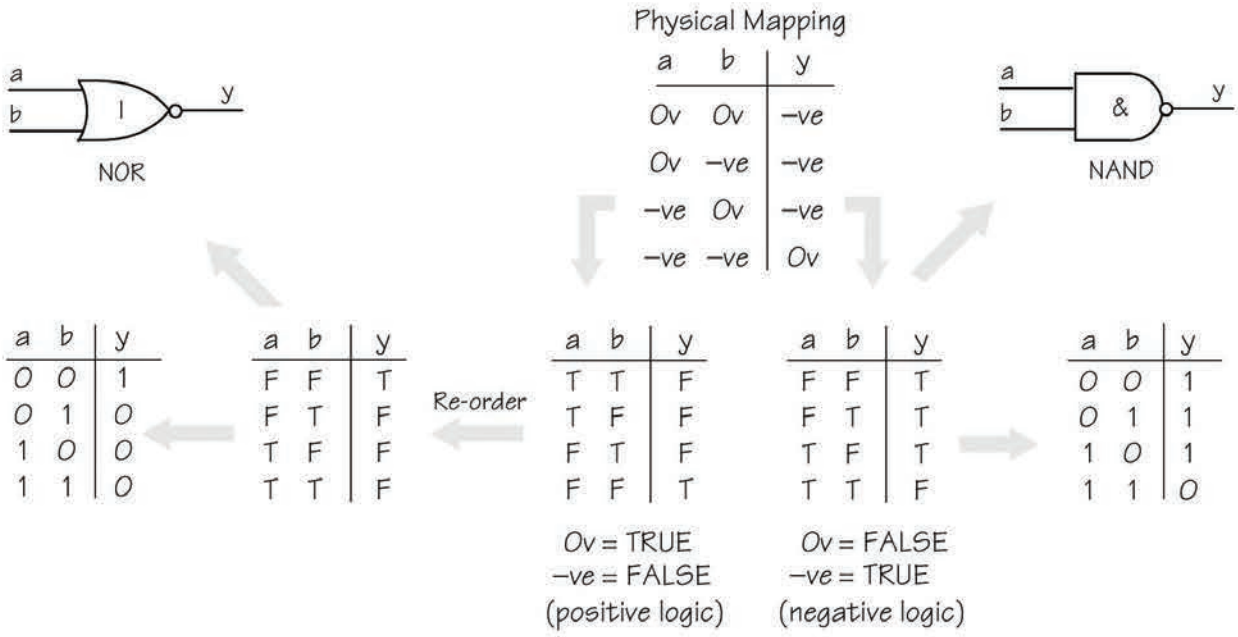


Figure B-6. The physical to abstract mapping of a PMOS logic gate

and logic 1 to –ve (presence of volts). Additionally, the physical structure of the PMOS gate is identical to that of the NMOS gate; if the NMOS gate is represented in positive logic and the PMOS gate is represented in negative logic, then both representations equate to NAND functions, which is, if nothing else, aesthetically pleasing.

Physical to Intermediate to Abstract Mapping

The discussions above offer the most consistent way to view the positive and negative logic conventions, and the casual reader is advised to stop at this point and proceed to something less demanding on the brain.

You're still here? Well, don't say I didn't warn you! Occasionally, an additional level of indirection is included in the progression from the physical to the abstract worlds. One reason for this is that supply and switching voltage values are relative rather than absolute quantities. In the examples above, the NMOS gate is shown as being powered by V_{SS} = 0v and V_{DD} = +ve, while the PMOS gate is shown as being powered by V_{SS} = 0v and V_{DD} = –ve.

However, within certain physical constraints, the actual values of V_{DD} and V_{SS} are largely immaterial. The only real requirements are for V_{DD} to be more

positive than V_{SS} for NMOS gates, and for V_{DD} to be more negative than V_{SS} for PMOS gates. Assuming that both technologies require a 5-volt differential between V_{DD} and V_{SS}, the following shows a small selection of the many possible voltage combinations that would suffice (Figure B-7).

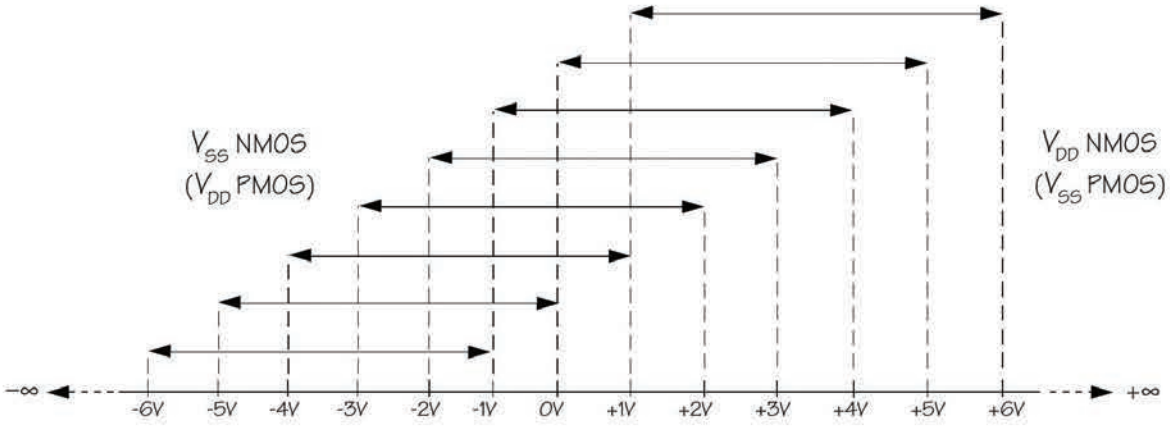


Figure B-7. Alternative values for V_{DD} and V_{SS}

Now consider an NMOS gate using V_{DD} = +2 volts and V_{SS} = –3 volts. Although it is preferable to map from the physical world directly to FALSE and TRUE values, it is also possible to map to intermediate HIGH and LOW values. The problem arises in the definitions of HIGH and LOW; some would regard HIGH as being the more positive potential and LOW as being the more negative, while others would adopt absolute values as measured from 0 volts as being the criteria. In this latter case, –3 volts would be considered to be HIGH based on the fact that it has a greater absolute value with respect to 0 volts than +2 volts, which would therefore be considered to be LOW.

This means that, for an NMOS gate supplied by V_{DD} = +2v and V_{SS} = –3v, the progression of mapping from *physical* through *intermediate* to *abstract* representations could be as shown in Figure B-8.

It is easy to see how this progression can become extremely confusing. To further illustrate the possible complexities involved, consider a design containing a combination of NMOS and PMOS logic gates.[4] Employing three voltage

[4] Note that this is not the same as connecting NMOS and PMOS transistors together in a complementary manner so as to form CMOS gates (this was discussed in Chapter 6).

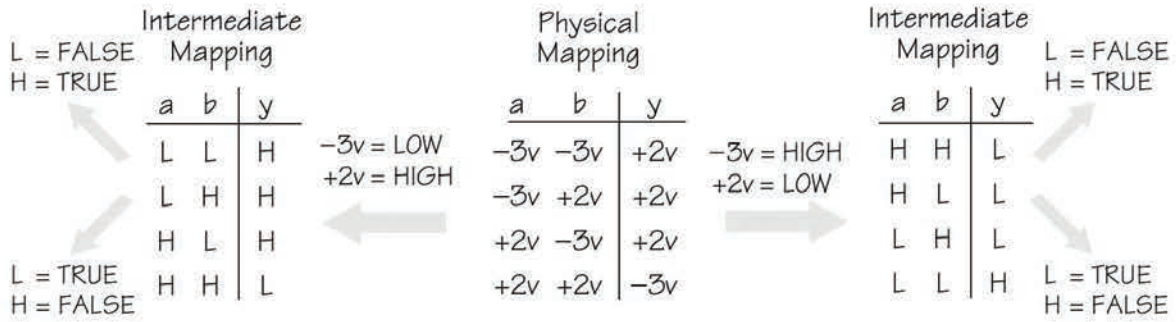


Figure B-8. Physical to intermediate to abstract mapping for an NMOS logic gate

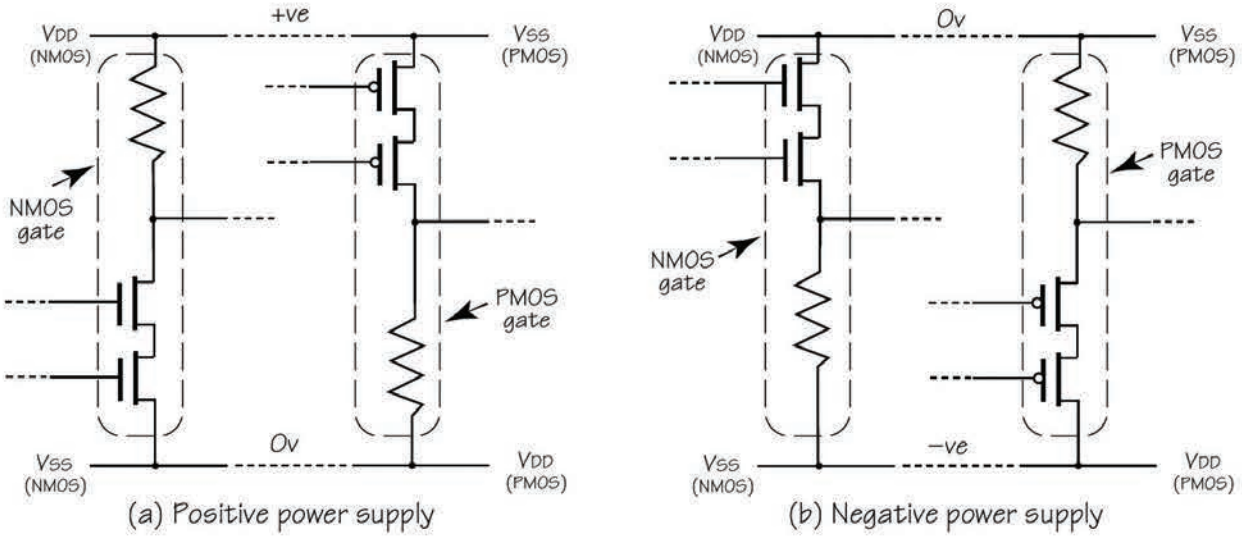


Figure B-9. NMOS and PMOS logic gates using common voltage levels

levels, 0v, +ve and –ve, would be less than satisfactory, because the gates would not be able to communicate with each other. However, there are solutions requiring only two voltage levels (Figure B-9).

As was noted above, the only real requirements are for V_{DD} to be more positive than V_{SS} for NMOS gates, and for V_{DD} to be more negative than V_{SS} for PMOS gates. Both these requirements can be satisfied using only two voltage levels as long as the labels associated with those levels are redefined as necessary. The two examples shown in Figure B-9 are actually different views of the same

thing; any two voltage levels with the appropriate relationship to each other would suffice. The application of the positive or negative logic systems in a mixed scenario of this type would prove to be an interesting exercise.

Fortunately, the majority of modern technologies favor positive logic representations and, unless otherwise stated, the positive logic convention is typically assumed. However, it is ultimately the designers' responsibility to define and/or ascertain which convention is being employed in any particular design.

Appendix

C

Reed-Müller Logic

Some digital functions can be difficult to optimize if they are represented in the conventional sum-of-products or product-of-sums forms, which are based on ANDs, ORs, NANDs, NORs, and NOTs. In certain cases it may be more appropriate to implement a function in a form known as Reed-Müller logic, which is based on XORs and XNORs. One indication as to whether a function is suitable for the Reed-Müller form of implementation is if that function's Karnaugh Map displays a checkerboard pattern of 0s and 1s. Consider a familiar two-input function (Figure C-1).

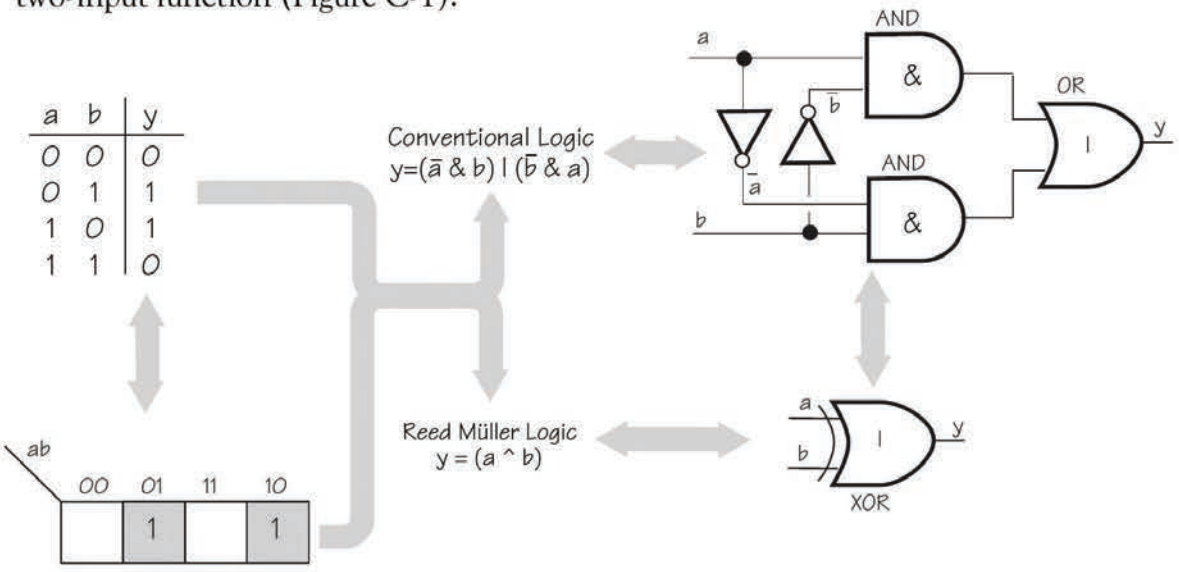


Figure C-1. Two-input function suitable for a Reed-Müller implementation

Since the above truth table is easily recognizable as being that for an XOR function, it comes as no great surprise to find that implementing it as a single XOR gate is preferable to an implementation based on multiple AND, OR and NOT gates. A similar checkerboard pattern may apply to a three-input function (Figure C-2).

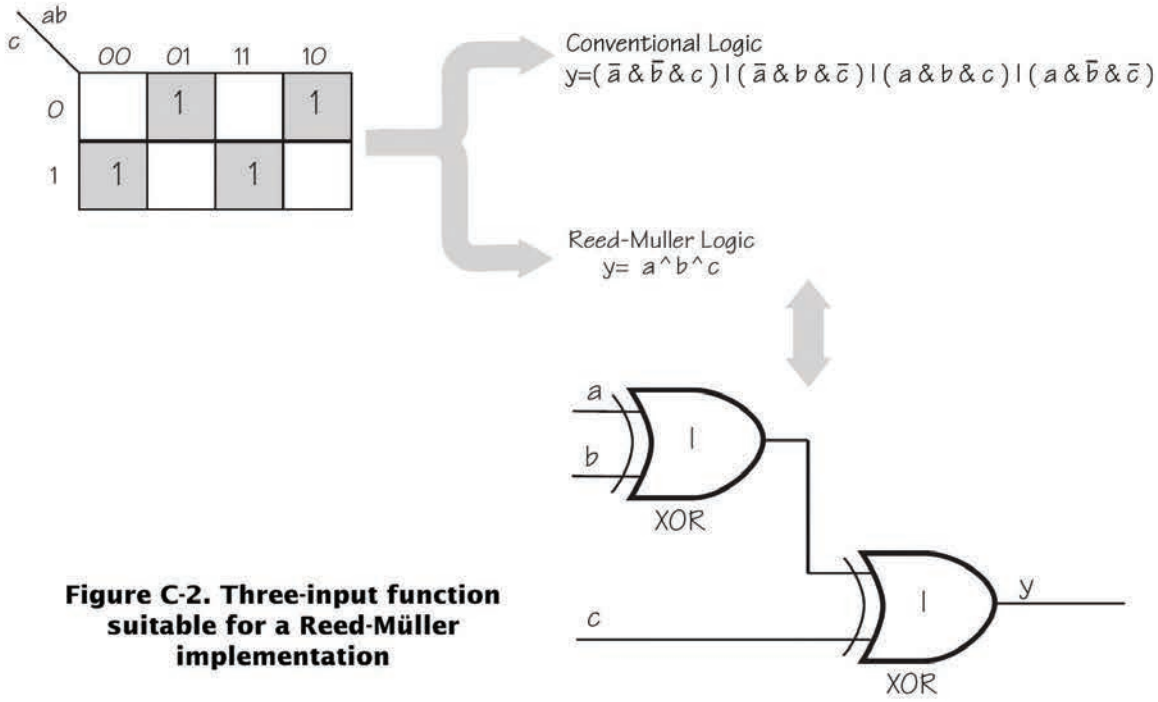


Figure C-2. Three-input function suitable for a Reed-Müller implementation

As XORs are both commutative[1] and associative,[2] it doesn't matter which combinations of inputs are applied to the individual gates. The checkerboard pattern for a four-input function continues the theme (Figure C-3).

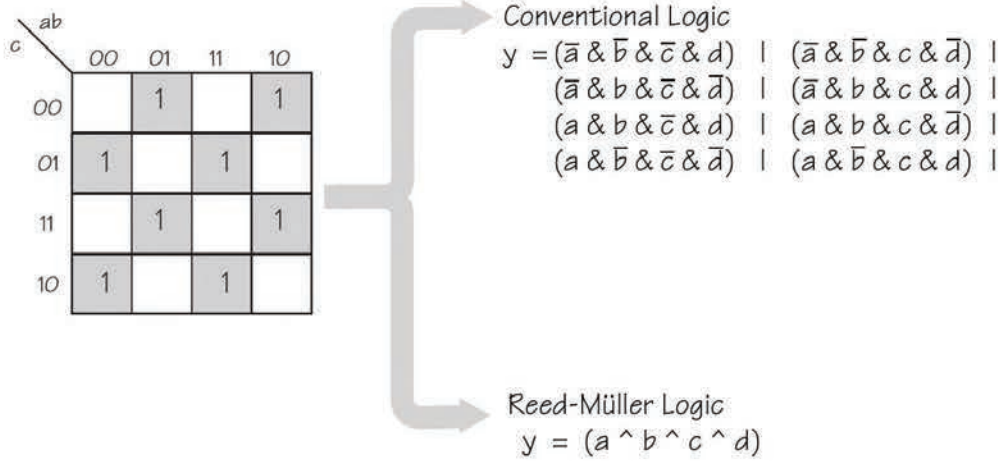


Figure C-3. Four-input function suitable for a Reed-Müller implementation

[1] For example, (a ^ b) ≡ (b ^ a). (Remember that the "≡" symbol means "is equivalent to.")
[2] For example, (a ^ b) ^ c ≡ a ^ (b ^ c).

Larger checkerboard patterns involving groups of 0s and 1s also indicate functions suitable for a Reed-Müller implementation (Figure C-4).

Once you have recognized a checkerboard pattern, there is a quick "rule of thumb" for determining the variables to be used in the Reed-Müller implementation. Select any group of 0s or 1s and identify the significant and redundant variables,[3] and then simply XOR the significant variables together.

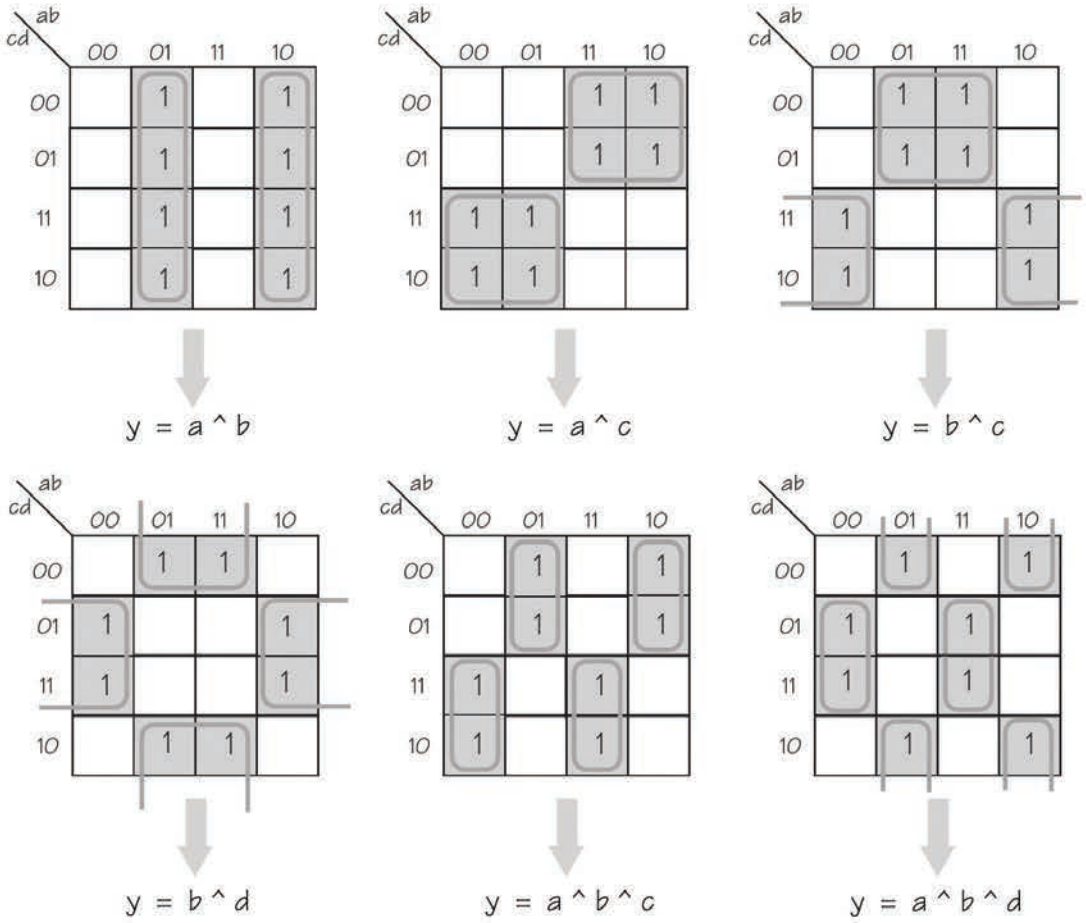


Figure C-4. Additional functions suitable for Reed-Müller implementations

[3] The significant variables are those whose values are the same for all of the boxes forming the group, while the redundant variables are those whose values vary between boxes.

As all of the checkerboard patterns above include a logic *0* in the box in the upper left corner,[4] the resulting Reed-Müller implementations can be realized using only XORs. However, any pair of XORs may be replaced with XNORs, the only requirement being that there is an even number of XNORs.

Alternatively, if the checkerboard pattern includes a logic 1 in the box in the upper left corner, the Reed-Müller implementation must contain an odd number of XNORs. Once again, it does not matter which combinations of inputs are applied to the individual XORs and XNORs (Figure C-5).

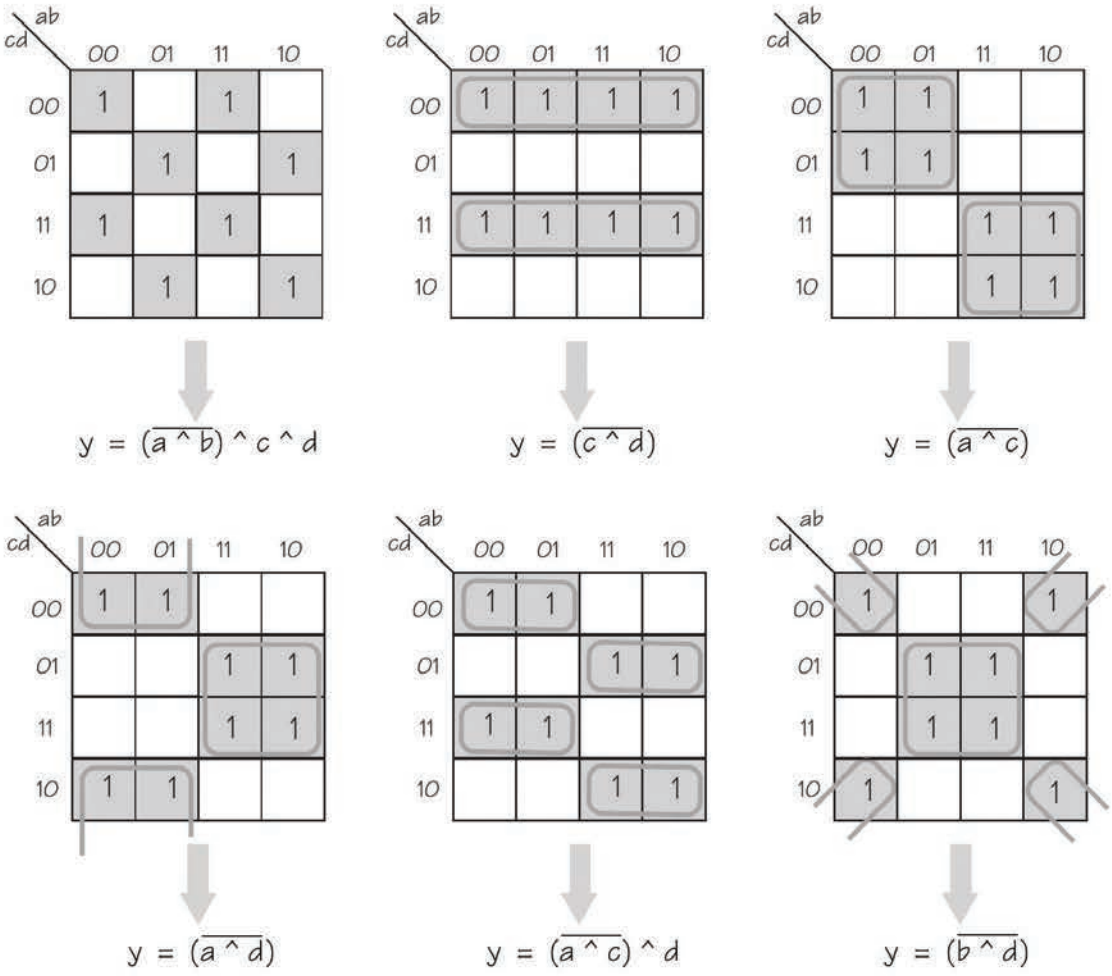


Figure C-5. Reed-Müller implementations requiring XNOR gates

[4] Corresponding to all of the inputs being logic *0*.

Although these examples provide a very limited introduction to the concept of Reed-Müller logic, checkerboard Karnaugh Map patterns are easy to recognize and appear surprisingly often. Reed-Müller implementations are often appropriate for circuits performing arithmetic or encoding functions (see also Appendix D).

Appendix D

Gray Codes

When moving between states in a standard binary sequence, multiple bits may change from 0 to 1 or vice versa; for example, three bits change value when moving from 0011_2 to 0100_2. In the physical world there is no way to ensure that all the bits will transition at exactly the same time, so a system may actually pass through a sequence of intermediate states. Thus, an intended state change of 0011_2 to 0100_2 could actually result in the sequence 0011_2 to 0111_2 to 0100_2.

One way to avoid this problem is to use a *gray code*, in which only a single bit changes when moving between states (Figure D-1).

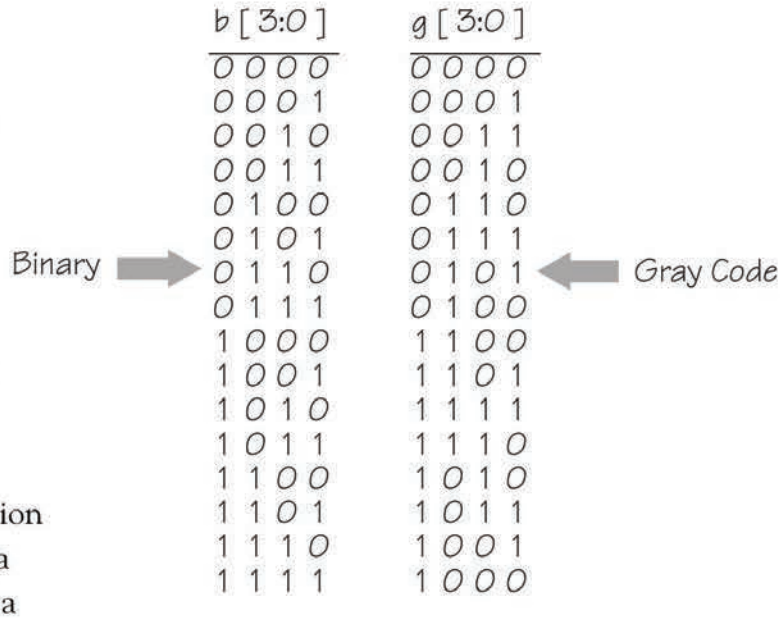


Figure D-1. Binary versus gray code

Gray codes are of use for a variety of applications, such as the ordering of the input variables on Karnaugh Maps.[1] Another application is encoding the angle of a mechanical shaft, where a disc attached to the shaft is patterned with areas of conducting material (Figure D-2).

The conducting areas are arranged in concentric circles, where each circle represents a binary digit. A set of electrical contacts, one for each of the circles, is used to detect the logic values represented by the presence or absence of the conducting areas. A digital controller can use this information to determine the

[1] This was discussed in detail in Chapter 10.

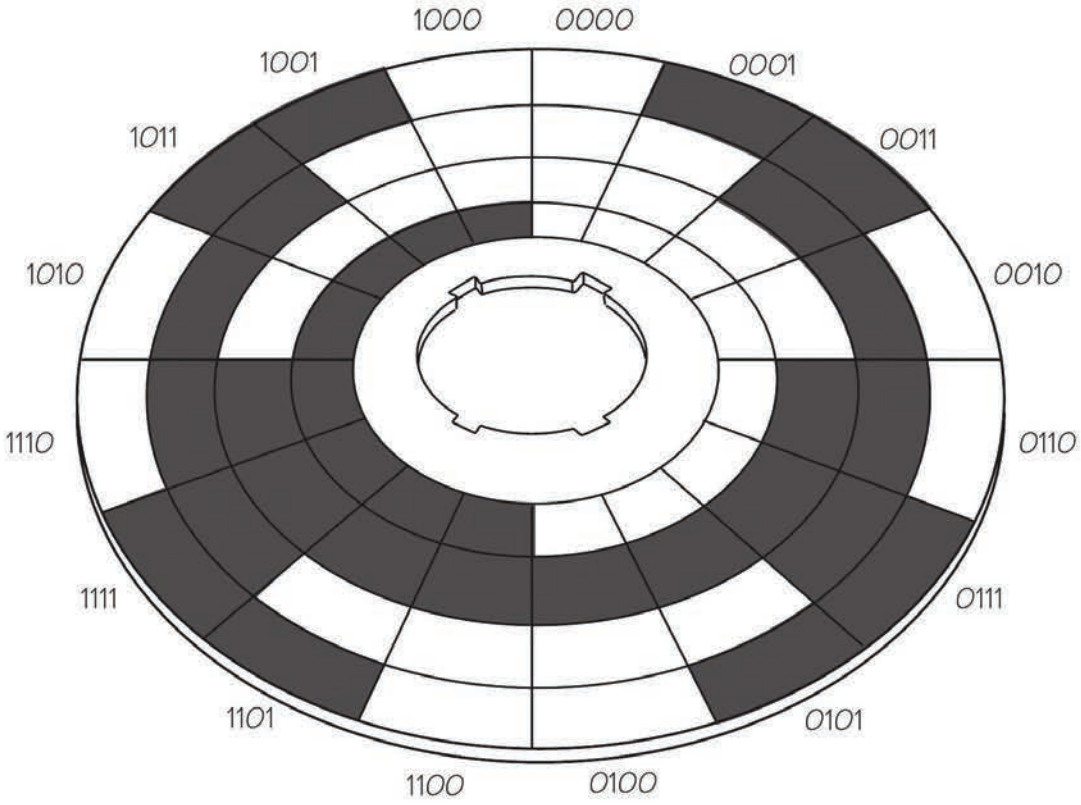


Figure D-2. Gray code used for shaft-angle encoding

angle of the shaft. The precision of the measurement depends on the number of bits (circles). A 4-bit value representing 16 unique states allows the shaft's angle to be resolved to 22.5 degrees, while a 10-bit value representing 1,024 unique states allows the shaft's angle to be resolved to 0.35 degrees. Using a gray code sequence to define the conducting and non-conducting areas ensures that no intermediate values are generated as the shaft rotates.

Commencing with a state of all zeros, a gray code can be generated by always changing the least significant bit that results in a new state. An alternative method which may be easier to remember and use is as follows:

a) Commence with the simplest gray code possible; that is, for a single bit.

b) Create a mirror image of the existing gray code below the original values.

c) Prefix the original values with 0s and the mirrored values with 1s.

d) Repeat steps b) and c) until the desired width is achieved.

An example of this mirroring process used to generate a 4-bit gray code is shown in Figure D-3.

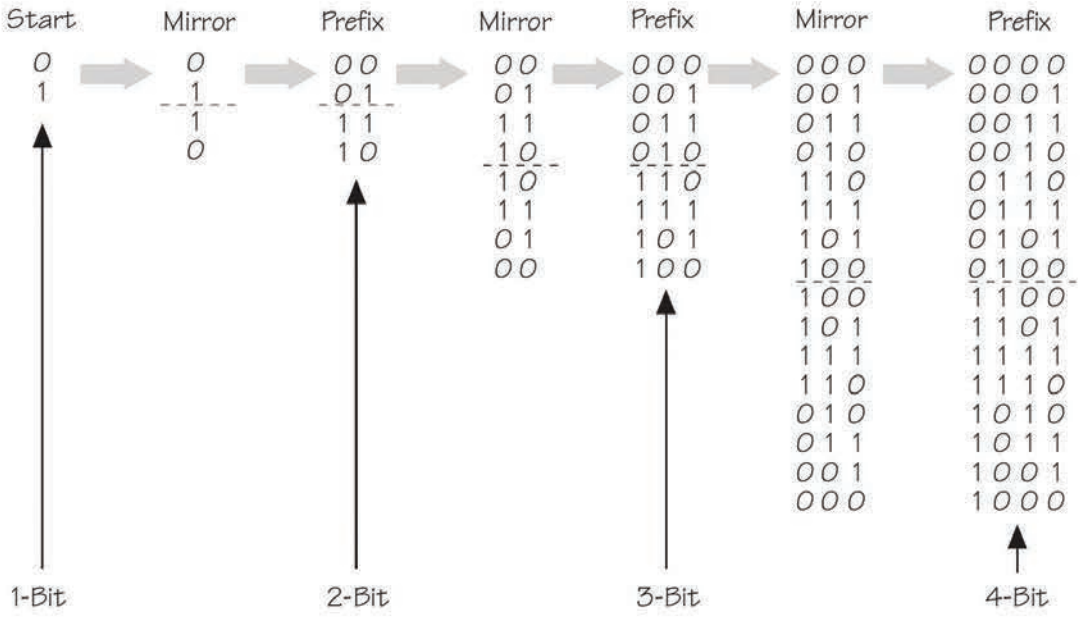


Figure D-3. Generating a 4-bit gray code by means of the mirroring process

It is often required to convert a binary sequence into a gray code or vice versa. Such converters are easy to create and are primarily of interest here due to their affinity to the Reed-Müller implementations that were introduced in Appendix C. Consider a binary-to-gray converter (Figure D-4). The checkerboard patterns of 0s and 1s in the Karnaugh Maps immediately indicate the potential for Reed-Müller implementations.

Similar checkerboard patterns are also seen in the case of a gray-to-binary converter (Figure D-5).

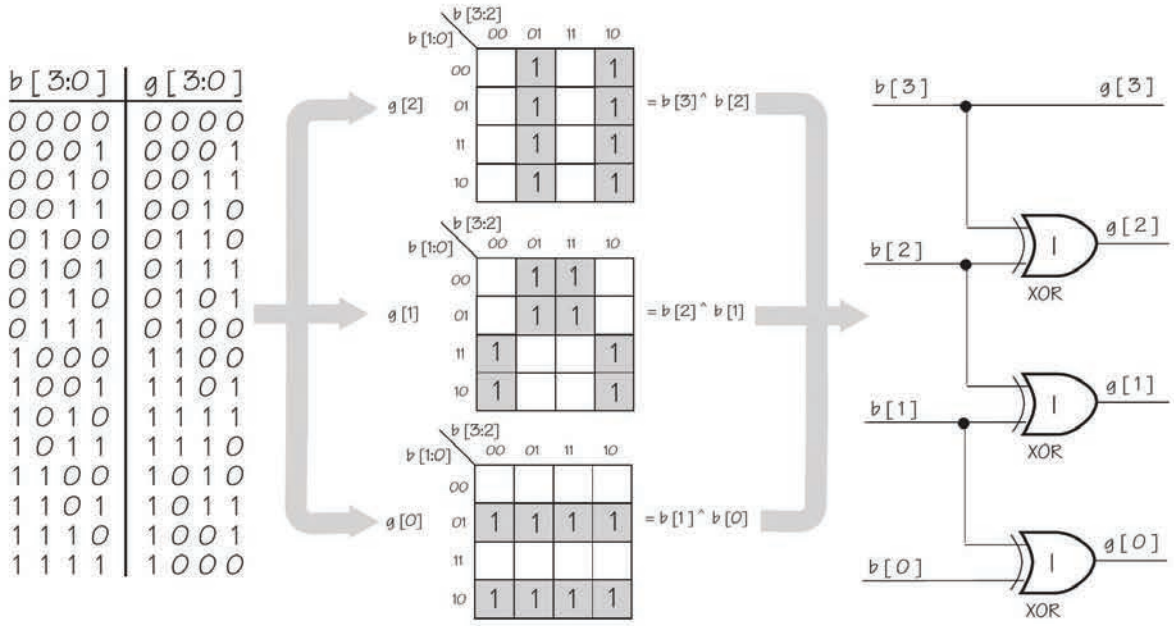


Figure D-4. Binary-to-gray converter

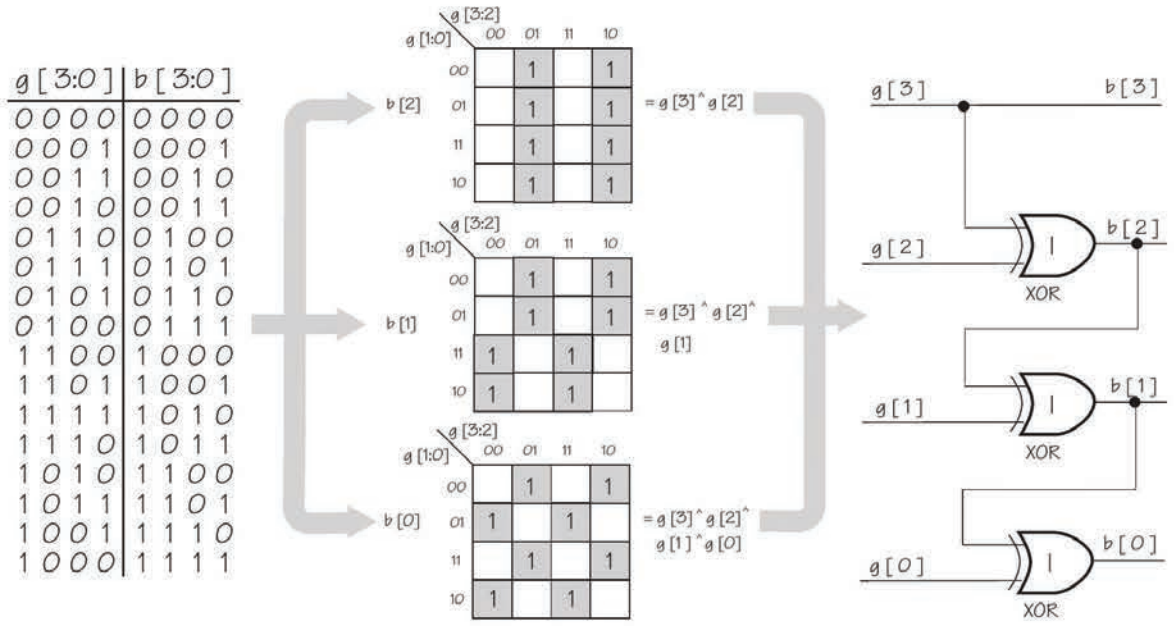


Figure D-5. Gray-to-binary converter

Appendix



A Reed-Müller Extraction Utility

In this appendix we discover a thought-provoking little mind game to delight the secret computer programmer that lurks within all of us.[1,2,3] After reading the previous two appendices, you should be a *"master of the mystic arts"* when it comes to Reed-Müller logic and gray codes. This is rather fortunate because, without this knowledge, what you are poised on the brink of reading would be likely to make your brains leak out of your ears. The simple program discussed below contains the never-before-seen-in-public, soon-to-be-famous, mind-bogglingly-convoluted, *Max's Algorithm* for automatically extracting Reed-Müller expressions.

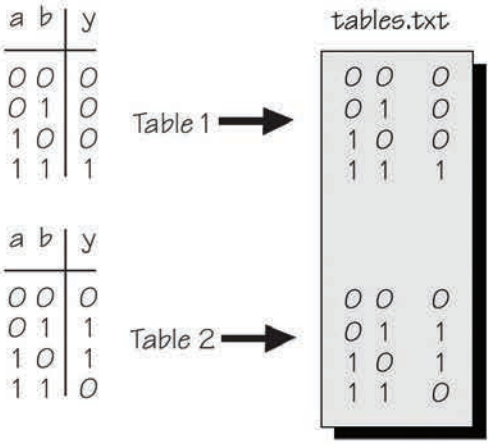


Figure E-1: Example truth tables for Reed-Müller evaluation

Before examining the program which, by some strange quirk of fate, is called *max_prog*, it would be useful to know what it actually does. First, you create a text file representation of one or more truth tables that you would like to examine and, second, you run the program using this file as its input. In return, the program determines which truth tables are suitable for Reed-Müller implementations and returns the appropriate expressions. For example, consider the truth tables shown in Figure E-1.

---

[1] The program is written in the C programming language and is intended to be compatible with any ANSI-standard C compiler.

[2] In certain cases, good programming practices have been thrown out of the window for the sake of brevity. For example, functions are generally not declared before they are called, and no error checking is performed on the input data.

[3] If you'd like to know more about writing programs in C, you can't do much better than reading "*The C Programming Language*" (Second Edition) by Brian W. Kernighan and Dennis M. Ritchie. For those of you who wish to voyage a little further, "*Reusable Data Structures For C*" by Roger Sessions is well worth perusing.

As you will see, the part of the program that reads the input file is rather simple, hence the file's somewhat minimalist look. When you've created the input file, you run the program from the operating system's command line as follows:

```
max_prog < tables.txt                          {Calls the program 'max_prog' and      }
                                               {feeds it with the text file 'tables.txt'. }

Truth table 1: No Reed-Muller solution         {These are the results returned by the }
Truth table 2: Reed-Muller solution is y = (a ^ b)  {program for the tables shown above.  }
```

As truth tables 1 and 2 in Figure E-1 are obviously those for an AND and an XOR, respectively, the results returned by the program should come as no surprise. Note that this version of the program assumes that the inputs to the truth tables are named a, b, c,... etc., and that the output is named y. In fact, this version of the program can handle truth tables with up to ten inputs (but only one output), and the number of inputs could be easily extended if necessary. Now, onto the program itself, the first part of which simply declares anything that the program is going to use.[4]

```
/* Max's algorithm to detect and evaluate functions suitable for Reed-Muller evaluation.  */
/* Accepts truth tables with up to ten inputs and returns any Reed-Muller equations.      */
/* Beware! Contains minimal parsing and no error detection — you have been warned!        */

#include <stdio.h>              /* Include the standard files supplied with C.           */
#include <ctype.h>
#include <string.h>

#define XSIZE 1024              /* Set the maximum number of elements necessary          */
                                /* to handle truth tables with two to ten inputs.        */

int bin_to_gray[XSIZE];         /* This will be the binary to gray-code lookup table.    */
int gray_to_bin[XSIZE];         /* This will be the gray-code to binary lookup table.    */

char workspace[XSIZE];          /* This will be the working area. Each element           */
                                /* corresponds to a box in a Karnaugh Map.               */
```

[4] If you do decide to play with this program, it is necessary to enter all the program segments in the same order that they are presented here.

```
struct xy {
  int x, y;
}  xy_table[11] = {                  /* When presented with the number of inputs      */
     { 0, 0},  /*  0 inputs */      /* associated with a truth table, this structure will */
     { 2, 1},  /*  1 inputs */      /* return the most appropriate dimensions for an  */
     { 2, 2},  /*  2 inputs */      /* equivalent Karnaugh Map (as nearly rectangular */
     { 4, 2},  /*  3 inputs */      /* as possible).                                  */
     { 4, 4},  /*  4 inputs */      /*                                                */
     { 8, 4},  /*  5 inputs */      /* The 'x' and 'y' values returned by the structure */
     { 8, 8},  /*  6 inputs */      /* represent the number of columns (across) the   */
     {16, 8},  /*  7 inputs */      /* map and the number of rows (down) the map      */
     {16,16},  /*  8 inputs */      /* respectively.                                  */
     {32,16},  /*  9 inputs */
     {32,32}   /* 10 inputs */
};
```

We will return to discuss the lookup tables and the working area in a little while, but first let us move briskly on to the function that controls everything:

```
int main()
{
   int     num_tt = 0,      /* The number of the truth table currently being processed.  */
           num_in,          /* The number of inputs in the current truth table.          */
           mask;            /* A binary mask which is used to track the significant inputs. */

   load_lookup_tables();                              /* Loads the lookup tables.        */

   while (num_in = get_truth_tables()) {              /* Load one truth table at a time. */
      num_tt++;                                       /* Keep track of the No. of the table. */
      if (test_reed_muller(num_in)) {                 /* If this table has a R-M solution . . . */
         mask = extract_reed_muller(num_in);          /* Extract the expression.         */
         output_reed_muller(mask,num_tt,num_in);      /* Output the equation.            */
      }
      else                                            /* If no R-M solution then say so. */
         printf("Truth table %i: No Reed-Muller solution\n",num_tt);
   }                                                  /* Repeat for all the tables until none */
   return 0;                                          /* remain, then return 0 and exit. */
}
```

The variable num_tt, which keeps track of the number of the truth table being processed, is initialized with zero, and then the function load_lookup_tables()

is called to initialize the lookup tables (what a surprise). The program then enters a loop as follows . . .

The function get_truth_tables() loads the outputs from the first truth table into the working area. This function also returns the number of inputs for the truth table and assigns the value to the variable num_in. When get_truth_tables() returns a value of zero for the number of inputs, then all of the truth tables in the input file have been processed and the program terminates. Otherwise, the first action in the loop is to increment num_tt, which keeps track of the number of the truth table that is being analyzed.

Once a truth table has been loaded, the function test_reed_muller() examines it to determine if it has a Reed-Müller solution. If there is such a solution, then the function extract_reed_muller() is used to determine which of the truth table's inputs are significant. The information on the significant inputs is presented in the form of a bit-mask which is assigned to the variable mask. Finally, the function output_reed_muller() displays the Reed-Müller equation for all the world to see. Good grief, if only everything in life could be so simple. Now, before plunging into the inner workings of the program, there are two utility functions that are called from time to time:

```
/* power_two: Generates a power of two.                                    */
/*          : Accepts an integer 'exponent'.                               */
/*          : Returns an integer equal to 2 raised to the power of 'exponent'. */
int power_two(int exponent)
{
  return 1 << exponent;      /* Returns 1 shifted to the left by the number of bits defined */
}                            /* by 'exponent' (which equals 2 to the power of 'exponent'). */
/* get_binary_value : Extracts a binary value presented as a string of 0 and 1 characters. */
/*                  : Accepts a pointer to the string of characters, '*p'.  */
/*                  : Returns the integer represented by the 0 and 1 characters. */
int get_binary_value(char *p)
{
  int value=0;                    /* Declare and initialize a temporary variable. */

  for (; *p ; p++)                /* Keep on processing the character pointed to by */
    value = (value<<1) + (*p - '0'); /* '*p' and incrementing '*p' until it points to a */
  return value;                   /* null character, then return the contents of 'value'. */
}
```

The get_binary_value() function accepts a pointer to a string of characters such as "1001", which have been read from the input file and which represent a set of binary input values associated with a line from one of the truth tables. The function converts this string into the numerical value that it represents; for example, "1001" = 9. The function is also used to process a single "0" or "1" character representing an output value associated with a line in a truth table, and to convert it into its numerical equivalent (0 or 1).

The first interesting action performed by the main controlling routine is to call the function load_lookup_tables(), and it doesn't take a rocket scientist to guess what this one does:

```
/* load_lookup_tables : Loads the 'bin_to_gray' lookup table with gray codes, and the      */
/*                    : 'gray_to_bin' lookup table with binary codes.                      */
/*                    : Returns a dummy parameter */
int load_lookup_tables()
{
  int bin, gray;                        /* Declare two temporary variables.              */

  for (bin = 0; bin < XSIZE; bin++) {   /* Increment 'bin' through a binary sequence . . .  */
    gray = bin ^ (bin >> 1);            /* . . . 'gray' equals the gray-code equivalent of 'bin' */
    bin_to_gray[bin] = gray;            /* . . . load gray code into 'bin_to_gray' lookup table. */
    gray_to_bin[gray] = bin;            /* . . . load binary value into 'gray_to_bin' lookup.  */
  }

  return 0;
}
```

The load_lookup_tables() function revolves around a loop, in which the variable bin is incremented from zero to the maximum number of elements in the tables. For each value of bin, its equivalent gray code is generated and assigned to the variable gray. Finally, the element in the bin_to_gray lookup table pointed to by bin is loaded with the value in gray, and the element in the gray_to_bin lookup table pointed to by gray is loaded with the value in bin. Note that the statement "gray = bin ^ (bin >>1)" is the software equivalent of the binary-to-gray converter illustrated in Figure D-4 in Appendix D.

Throughout the rest of this program, binary-to-gray and gray-to-binary conversions can be easily performed by accessing the appropriate value in the bin_to_gray and gray_to_bin lookup tables respectively. For example, assuming that an integer variable called index has already been assigned the equivalent of the binary value 1100_2:

```
temp = bin_to_gray[index];    {Would result in 'temp' containing the binary value 1010₂  }
temp = gray_to_bin[index];    {Would result in 'temp' containing the binary value 1000₂  }
```

You can check these results using Figure D-1 in Appendix D. After everything has been declared and the lookup tables have been initialized, the program is ready to rock-and-roll. The first function of any significance is *get_truth_tables()*, which loads the truth tables, one at a time, from the input file:

```
/* get_truth_tables : Loads one truth table at a time from the input file into the working  */
/*                  : area, 'workspace'. Be very careful—this function contains minimal   */
/*                  : parsing and no error detection. As a wise man once said—"You can    */
/*                  : make it fool proof . . . but you can't make it damn-fool proof."    */
/*                  : Returns the number of inputs in the table being processed.          */
int get_truth_tables()
{
   #define XLINE 132                  /* Set the max No. of characters allowed on a line.  */
   char *p;                           /* Declare a character pointer variable.             */
   char *strtok();                    /* Declare a function (from the standard C library). */
   char line[XLINE];                  /* Reserve space to load a line from the input file. */
   int num_in;                        /* Keeps track of the number of truth table inputs.  */
   int inputs, output;                /* Binary equivalents of the inputs and output       */
   int in_table = 0;                  /* Keeps track of whether we're in a truth table     */

   while(gets(line)) {                /* Use the standard function 'gets()' to read a line.*/
      if (!(p=strtok(line," \t"))) {  /* If std function 'strtok()' says this line empty . . .*/
         if (in_table)                /* . . . if we were already processing a table then  */
            return num_in;            /* we've finished, so return number of inputs.       */
         else                         /* . . . but if we weren't already processing a table,*/
            continue;                 /* then skip the rest of the "while" loop, go back   */
      }                               /* to the start of the loop, and get the next line.  */

      in_table = 1;                   /* Have got a non-blank line, so must be in a table. */
      num_in = strlen(p);             /* Use std function 'strlen()' to find No. of inputs.*/
      inputs = get_binary_value(p);   /* Get the integer value represented by the inputs.  */
      p = strtok((char *)0," \t");    /* Tell 'strtok()' to get next token (the output).   */
      output = get_binary_value(p);   /* Get the integer value represented by the output.  */
      workspace[gray_to_bin[inputs]] = output; /* Load the output data into the workspace. */
   }                                  /* Go back to the beginning of the loop.             */

   return in_table ? num_in : 0;      /* Exit loop at end-of-file. If in a table then return */

}                                     /* the number of inputs, otherwise return zero.      */
```

Functions that read-in and write-out data are generally rather boring, and this one is no exception. On the other hand, even though it is "mean and lean," *get_truth_tables()* permits a reasonable amount of latitude, and it's worth taking the time to note its main features and limitations:

FEATURES	LIMITATIONS
Allows any number of blank lines, including lines containing spaces and tabs, at the beginning and end of the file, and also between truth tables.	No blank lines are allowed in the middle of a truth table, but there must be at least one blank line separating one truth table from another.
Allows multiple spaces and tabs at the beginning and end of each truth table line, and also between the inputs and outputs.	All the input values must be grouped together without any spaces or tabs, but there must be at least one space or tab separating the inputs from the output.
The lines of a truth table can be specified in any order	All the lines in a truth table must be specified.
This version of the program supports truth tables with two to ten inputs.	This version of the program only supports truth tables with one output.

Actually, considering the size of this function, or its lack thereof, you couldn't really ask for too much more (what do you want for nothing, your money back?). The function processes each line of a truth table and loads the output value associated with that line into the working area *workspace[]*. When the function detects a blank line indicating the end of the truth table, it terminates and returns the number of inputs associated with this table to the main controlling loop.

The statement of most interest is "workspace[gray_to_bin[address]] = data;", which loads the output value into the working area. This statement uses the gray-code to binary lookup table to transpose the address representing the truth table's input values into an address used by the computer to point to the working area, and to load the output value from the truth table into the element in the working area pointed to by this transposed address. The organization of the working area is the key to the way in which the rest of the program works.

The problem is that it is not easy to check for Reed-Müller implementations if the data is represented in the form of a truth table. The most efficient method (as far as the author knows) is to visualize and process the data in the form of a Karnaugh Map. However, for a variety of reasons, it is easier for the program to handle the data as a one-dimensional array. Additionally, any solution must easily accommodate truth tables (or Karnaugh Maps) of different sizes; that is, with different numbers of inputs.

The working area used by this program is a one-dimensional array of characters called *workspace[]* (Figure E-2). An array of characters is used because they are only 8-bits wide, and therefore occupy less of the computer's memory than would integers, which may each require 16-bits or 32-bits depending on the compiler being used:

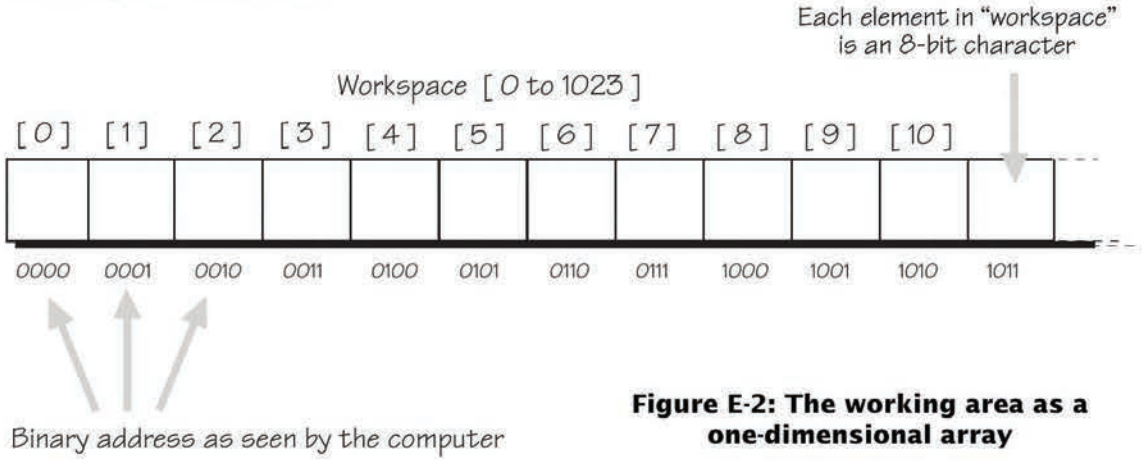


Figure E-2: The working area as a one-dimensional array

In fact, this program uses only the least-significant bit of each character's 8-bit field to store the contents (output values) of the truth table, and the other seven bits are therefore available for possible future enhancements. The next thing is to understand how the contents of the truth table are mapped into the working area. The easiest way to visualize this is to first view the truth table as a Karnaugh Map, and then to imagine grabbing the upper left-hand and upper right-hand corner boxes of map and pulling them to the left and right respectively, thereby expanding the map like a concertina (Figure E-3).

The next point to note is that there are number of alternative ways of viewing the Karnaugh Map associated with a particular truth table. For example,

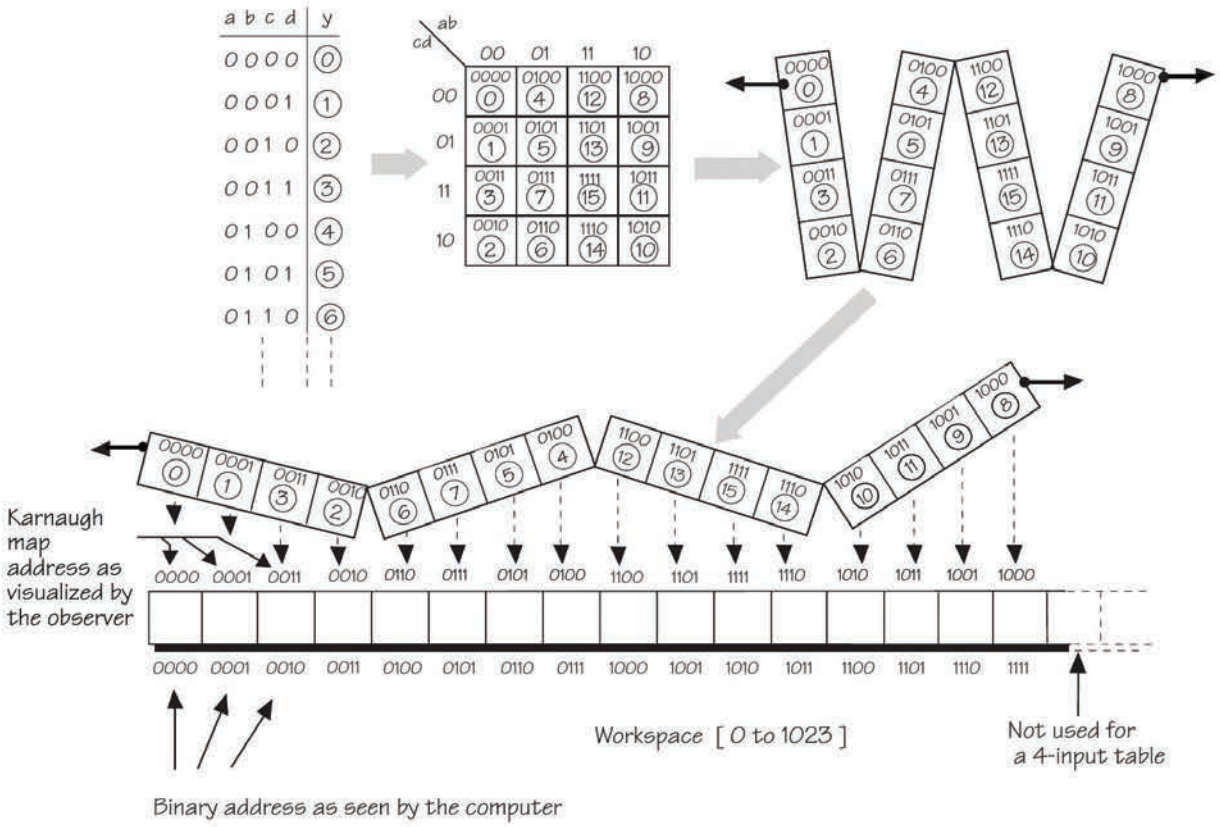


Figure E-3: Mapping a truth table into the working area

a table with four inputs may be represented as a Karnaugh Map organized with four columns and four rows, two columns and eight rows, eight columns and two rows, etc. Consider a 3-input truth table represented as 4 × 2 and 2 × 4 Karnaugh Maps (Figure E-4).

The beauty of the technique used by this program to map the data from a truth table into the working area is that, irrespective of the size of the truth table, and irrespective of how the truth table is visualized as a Karnaugh Map, the data is always mapped into the same locations in the working area (Figure E-5).

Thus, the statement "workspace[gray_to_bin[address]] = data;" in the `get_truth_tables()` function transposes the truth table (or Karnaugh Map) address, which is considered to be in the form of a gray code, into a binary

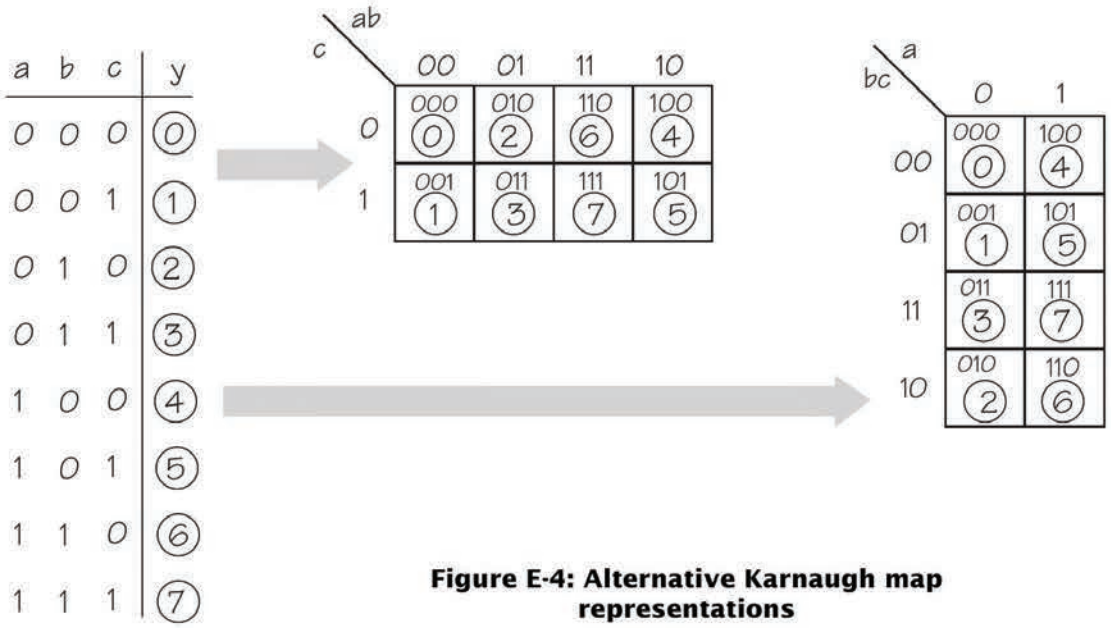


Figure E-4: Alternative Karnaugh map representations

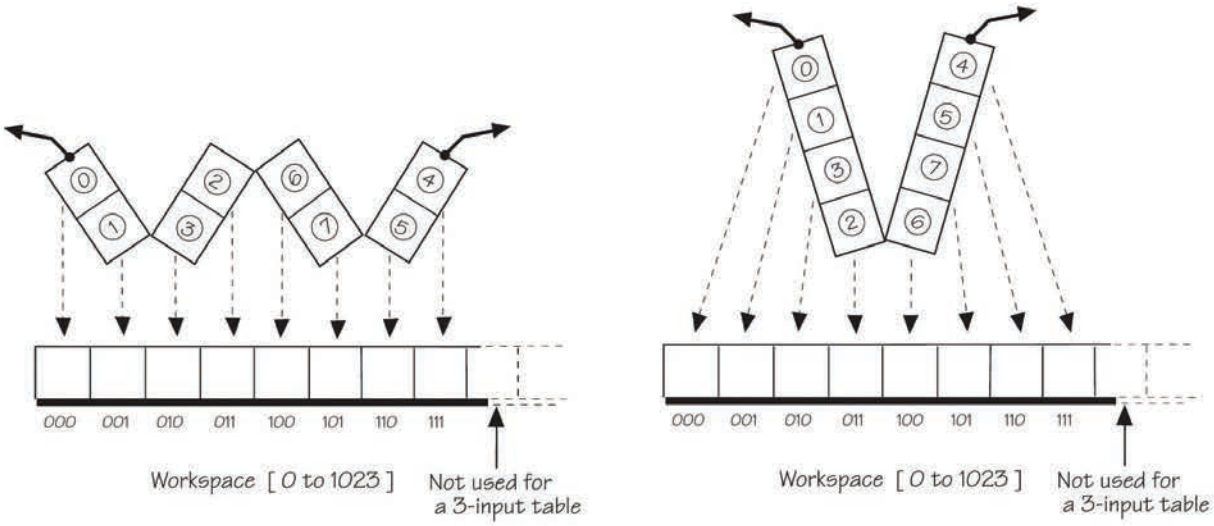


Figure E-5: Consistent mapping to the working area

equivalent that is used to point to the working area. The output value from the truth table is loaded into the working area at this transposed address.

After all of the data from a truth table has been loaded into the working area, the function *test_reed_muller*() is called to determine whether this particular truth table has a Reed-Müller solution:

```
/* test_reed_muller : Checks the contents of the truth table to see if this table has a      */
/*                  : Reed-Müller solution.                                                   */
/*                  : Accepts one parameter, 'num_in,' the number of inputs in this table.    */
/*                  : Returns 1 if this table has a Reed-Müller solution, and 0 if it doesn't. */
int test_reed_muller(int num_in)
{
  char *bottom, *top;                    /* Declare temporary character pointers.          */
  int start, end, look_for;              /* Declare local variables.                       */

  for (start = 4, end = power_two(num_in); start <= end; start <<=1) {   /* Main loop.   */
    bottom = &workspace[0];              /* Point 'bottom' at the LHS of 'workspace'.      */
    top = &workspace[start - 1];         /* Point 'top' at the RHS for this pass.          */
    look_for = *bottom++ ^ *top--;       /* Find out what we're looking for.               */
    do {                                 /* Enter a sub-loop.                              */
      if ((*bottom++ ^ *top--) != look_for)  /* If the rest aren't what we want . . .      */
        return 0;                        /* . . . Return 0 to indicate no solution.        */
    } while (bottom < top);              /* . . . Otherwise continue with the sub-loop.    */
  }
                                         /* If we get this far, then there is a solution,  */
  return 1;                              /* so return 1 to tell the world.                 */
}
```

The test_reed_muller() function performs a series of actions which are more easily described pictorially. Consider the case of a 2-input truth table (Figure E-6).

The main loop first initializes bottom and top to point at locations 0 and 3 in the working area respectively. The first action is to perform an exclusive-OR of the contents of the working area pointed to by bottom and top, and to store the result in the variable look_for. It doesn't actually matter whether the result stored in look_for is a 0 or a 1, because whatever it is, that's what we're looking for. The values of bottom and top are then incremented and decremented respectively, an exclusive-OR is performed on the contents of the working area that they are now pointing to, and the result is compared to the value stored in look_for.

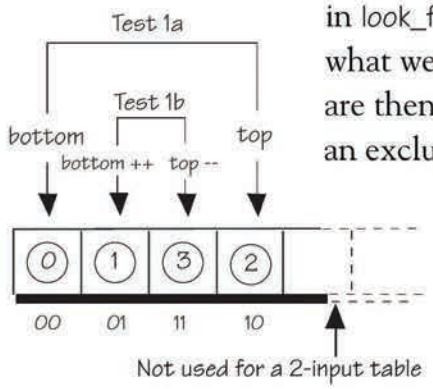


Figure E-6: Testing a 2-input truth table

If the result of this exclusive-OR is not the same as the value stored in look_for, then there is no Reed-Müller

solution, and test_reed_muller() returns a 0 and terminates. But if the result is the same as the value stored in look_for, then the function continues onward. In fact, in the case of a 2-input truth table, all of the tests have been performed, so the function would return a 1 indicating that there is a Reed-Müller solution. Now consider the case of a 3-input truth table (Figure E-7).

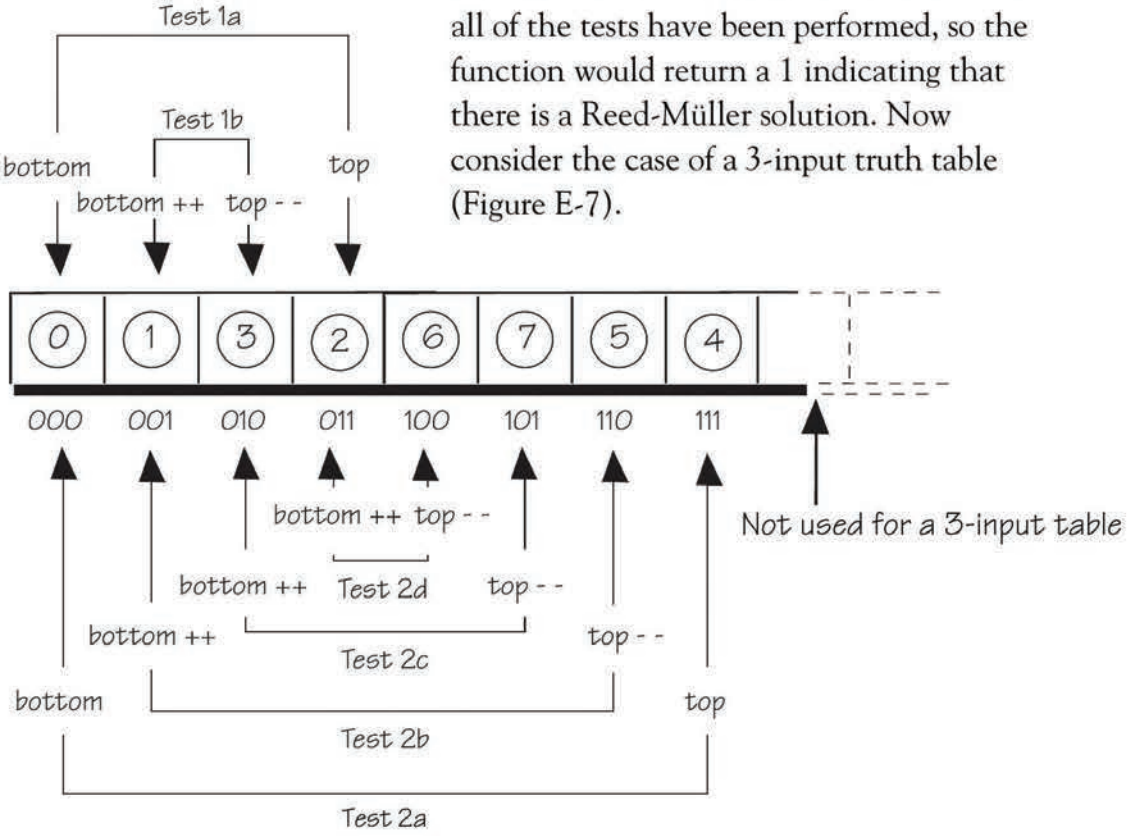


Figure E-7: Testing a 3-input truth table

The first series of tests are exactly the same for a 2-input truth table. Assuming that the table passes these tests, the main loop reinitializes bottom and top to point at locations 0 and 7 in the working area respectively. Once again, the first action in this iteration of the loop is to perform an exclusive-OR of the contents of the working area pointed to by bottom and top, and to store the result in look_for. And once again, it doesn't actually matter whether the result stored in look_for is a 0 or a 1, because whatever it is, that's what we're looking for this time around.

The values of bottom and top are then incremented and decremented respectively, an exclusive-OR is performed on the contents of the working area that they are now pointing to, and the result is compared to the value stored in

look_for. This continues until it is determined that there is no Reed-Müller solution, or until bottom and top meet in the middle, in which case there is a solution.

Similarly, in the case of a 4-input truth table, the first series of tests would be the same as for a 2-input table, and the second series of tests would be the same as for a 3-input table. Assuming that the table passes these tests, the main loop would commence a third series of tests by reinitializing bottom and top to point at locations 0 and 15 in the working area respectively. As usual, the first action in this third iteration of the loop would be to perform an exclusive-OR of the contents of the working area pointed to by bottom and top, and to store the result in look_for. The process would then continue as before: incrementing bottom, decrementing top, and testing their contents against the value in look_for until it is determined that there is no Reed-Müller solution, or until bottom and top meet in the middle, in which case there is a solution.

The problem is that, although these tests are simple, there can be a lot of them. On the positive side, if there is no Reed-Müller solution then this function usually finds out fairly quickly. But if there is a solution, then it can take a lot of tests to prove it. In fact, an n-input truth table will require $(2^n - 2)$ tests if it has a solution. This is not ideal, but there does not appear to be a better way . . . unless you can find one (more below). Assuming that there is a Reed-Müller solution, then the extract_reed_muller() function is used to determine the significant inputs:

```
/* extract_reed_muller  : Extracts the significant inputs for the Reed-Müller solution.      */
/*                      : Accepts one parameter, 'num_in', the No. of inputs in the table.  */
/*                      : Returns a bitmask, 'mask,' representing the significant inputs.   */
int extract_reed_muller(int num_in)
{
  int i, x, y, mask, index, look_for;     /* Declare all the local variables.              */
  mask = power_two(num_in) - 1;           /* Set 'mask' to contain 1s for each input.      */
  look_for = workspace[0];                /* Set 'look_for' to the value in [0,0] of the map. */
  x = xy_table[num_in].x;                 /* Get the most appropriate number of columns.   */
  y = xy_table[num_in].y;                 /* Get the most appropriate number of rows.      */

/* The following routines view the working area as a Karnaugh Map. When I wrote           */
/* them, only God and myself knew how they worked . . . now God only knows![5]            */
```

[5] This is an old programming joke, but it's also true far more times than you'd care to believe.

```
    for (i = 1; i < x; i++) {                /* This loop commences at box [0,0] in the map   */
      if (!(i & 1))                          /* (the upper-left corner) and searches "East."  */
        index = i * y;                       /* For every box on its journey it must generate */
      else                                   /* a value for 'index', which acts as a pointer  */
        index = ((i + 1) * y) - 1;           /* into the working area.                        */

      if (workspace[index] == look_for)      /* If the box contains the same as 'look_for' . . .  */
        mask &= ~(bin_to_gray[index]);       /* . . . Remove non-significant input from 'mask.'   */
      else                                   /* . . . Otherwise break out of this loop and move   */
        break;                               /* onto the next one.                                */
    }

    if (i < (x >> 1)) {                      /* Before beginning this loop, a test is performed */
      for (i = (x - 1); i > 0; i — ) {       /* to see if it's actually necessary. If the last  */
        if (!(i & 1))                        /* loop searched half way or all the way across    */
          index = i * y;                     /* the map, then there's no point in performing    */
        else                                 /* the tests in this loop.                         */
          index = ((i + 1) * y) - 1;
                                                 /* Otherwise, this loop searches "West"    */
        if (workspace[index] == look_for)        /* from box [0,0] in the map. Because of   */
          mask &= ~(bin_to_gray[index]);         /* the wrap-around effect associated with  */
        else                                     /* Karnaugh Maps, this means that the      */
          break;                                 /* search actually commences in the        */

      }                                          /* upper-right corner of the map, and      */
    }                                            /* then heads back towards box [0,0].      */

    for (i = 1; i < y; i++) {                /* This loop commences at box [0,0], and          */
      if (workspace[i] == look_for)          /* and searches "South," that is, towards the     */
          mask &= ~(bin_to_gray[i]);         /* bottom-left corner.                            */
      else
          break;
    }

    if (i < (y >> 1)) {                      /* Tests to see if this loop is actually necessary.  */
      for (i = (y - 1); i > 0; i — ) {       /* If it is, then search "North" from box [0,0].     */
        if (workspace[i] == look_for)        /* Because of the Karnaugh Map wrap-around           */
          mask &= ~(bin_to_gray[i]);         /* effect, the search actually commences in the      */
        else                                 /* lower-left corner of the map, and then heads      */
          break;                             /* back towards box [0,0].                           */
      }
    }

    return mask;                             /* Finally, return the significant inputs in the  */
}                                            /* form of the bitmask, 'mask.'                   */
```

The extract_reed_muller() function is the hardest to follow, but it is certainly a good exercise in reverse-polish-logic-lateral-thinking which will keep your brain in peak mental fitness. The easiest way to visualize the East-West-South-North searches is shown below. Consider the Karnaugh Map associated with a 4-input truth table (Figure E-8).

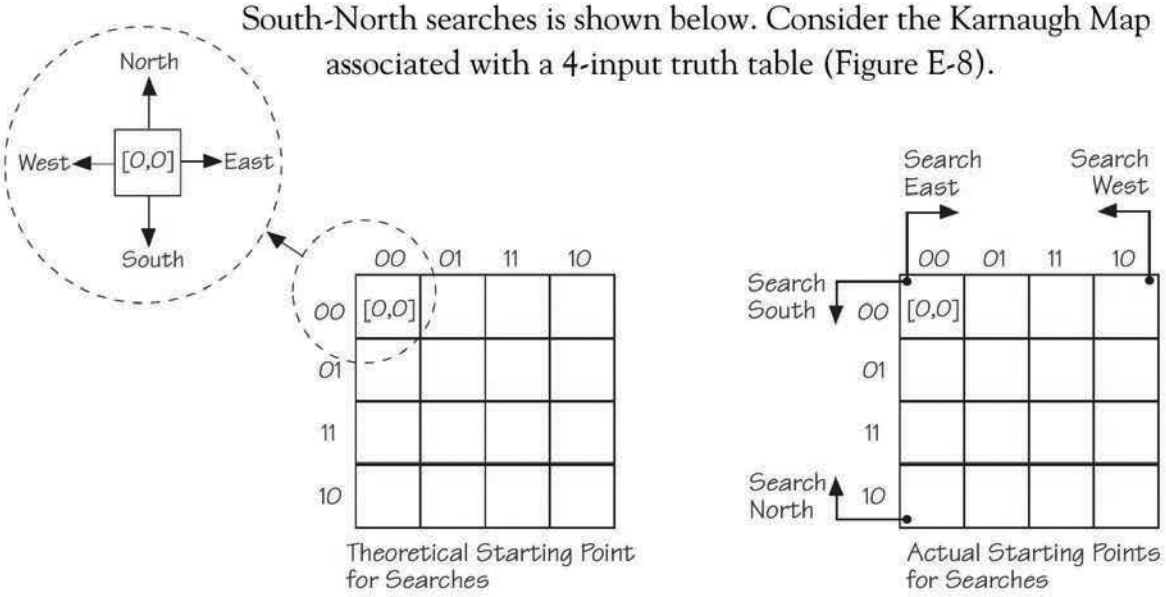


Figure E-8: Visualizing the searches to locate the significant inputs

In fact, the extract_reed_muller() function is reasonably efficient. One of the rules for a Reed-Müller solution is that there must be the same number of 1s stored in the Karnaugh Map boxes as there are 0s. This means that the largest possible block of 1s or 0s can occupy only half of the boxes, which in turn means that this function actually performs relatively few tests. In fact, for a 4-input truth table, this function will perform a maximum of only four tests (excluding the tests to see whether further tests are necessary). Now it only remains to output the equation for the Reed-Müller implementation:

```
/* output_reed_muller : Output the equation for the Reed-Muller implementation.          */
/*                    : Accepts three parameters: 'mask,' which contains the significant */
/*                    : inputs, 'num_tt,' which contains the No. of the truth table in the */
/*                    : input file, and 'num_in,' which contains the No. of table inputs. */
/*                    : Returns a dummy value.                                           */
int output_reed_muller(int mask, int num_tt, int num_in)
{
```

```
    int i;                              /* Declare a temporary integer variable.          */
    char c;                             /* Declare a temporary character variable.        */

    if (!workspace[0])                                          /* If box [0,0] holds     */
      printf("Truth table %i: Reed-Muller solution is y = (",num_tt); /* a zero, then only  */
    else                                                        /* need XORs, else        */
      printf("Truth table %i: Reed-Muller solution is y = !(",num_tt); /* add an inversion. */

    for (i = (power_two(num_in) >> 1), c = 'a'; !(mask & i); i >>= 1, c++);
    printf("%c ",c);
    for (i >>= 1, c++; i > 0; i >>= 1, c++)  /* Explaining this rather cunning routine     */
      if (mask & i)                          /* is left as an exercise for the reader . . .*/
        printf("^ %c ",c);                   /* . . . Ha!                                  */
    printf(")\n");

    return 0;                           /* Return a dummy value.                          */
}
```

How to Become Famous

There are a number of ways in which the program described above could be modified or extended. For example, the input function, *get_truth_tables()*, could be easily modified to accept truth tables with multiple outputs. Alternatively, it could be augmented to accept the textual equivalent of Karnaugh Maps. A more interesting exercise would be to modify the program to accept *don't care* values (represented using question marks, "?") on the inputs (Figure E-9).[6]

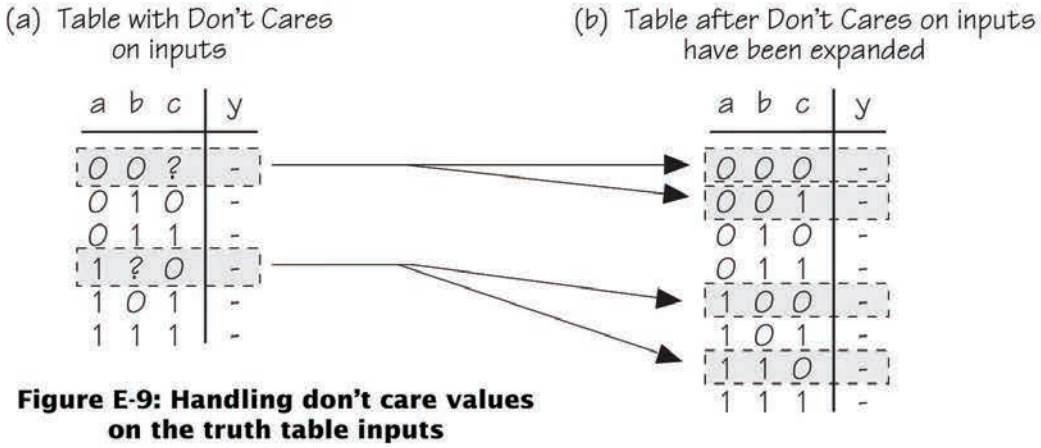


Figure E-9: Handling don't care values on the truth table inputs

[6] The concepts of *don't cares* on inputs and outputs were introduced in more detail in Chapter 10.

This modification would be relatively simple, because it would only affect the input function. Note that handling a single don't care value on a line of inputs is trivial, but handling multiple don't cares is a little trickier. For those who enjoy a real challenge, modifying the program to cope with don't cares on the outputs should offer one! This is where the spare bits that are available in the working area may start to earn their keep.

Another interesting area to explore would be the *test_reed_muller()* function. As was noted during the discussions on this function, an n-input truth table will require $(2^n - 2)$ tests (assuming that the table has a Reed-Müller solution). It may be that you can discover a more efficient technique to determine whether the table has a solution.[7]

During the course of this discussion, the question may have arisen as to why the maximum number of truth table inputs is set as high as ten. After all, this would equate to a truth table with 1,024 lines, and there is little chance that anyone would take the time to type these in (and even less chance of their doing so without errors). However, one way in which the program could be modified would be for it to read in simple Boolean equations. For example, the input function could be modified to process the following text file:

```
INPUTS          = a, b, c, d;
OUTPUTS         = x, y;
INTERNAL_NODES  = j, k;

j  =    (a & b);
k  = a | (c ^ d);
x  = j & (c | d);
y  =    !(k & b);

END
```

The format shown here is fairly generic, but you could invent your own if you wish. The point is, if you start processing equations in this form, you can easily reach the ten-input limit.

---

[7] Beware when you alter this function, there are many traps for the unwary. A useful test is to apply your routine to the 4-input truth table equivalent of a 4 × 4 Karnaugh Map which contains only four 1s, one in each corner. If your routine decides that this table has a Reed-Müller solution, then it's wrong!

In fact, extending the program to accept Boolean equations is a large stride into the arena of *logic synthesis*, in which programs are used to optimize the logic used to implement a design. Many commercially available logic synthesis utilities use a method of representing the data internally known as *binary decision diagrams* (BDDs), in which equations are stored in decision trees. For example, consider the binary decision diagram for the simple expression "y = (a & b) | c" (Figure E-10).

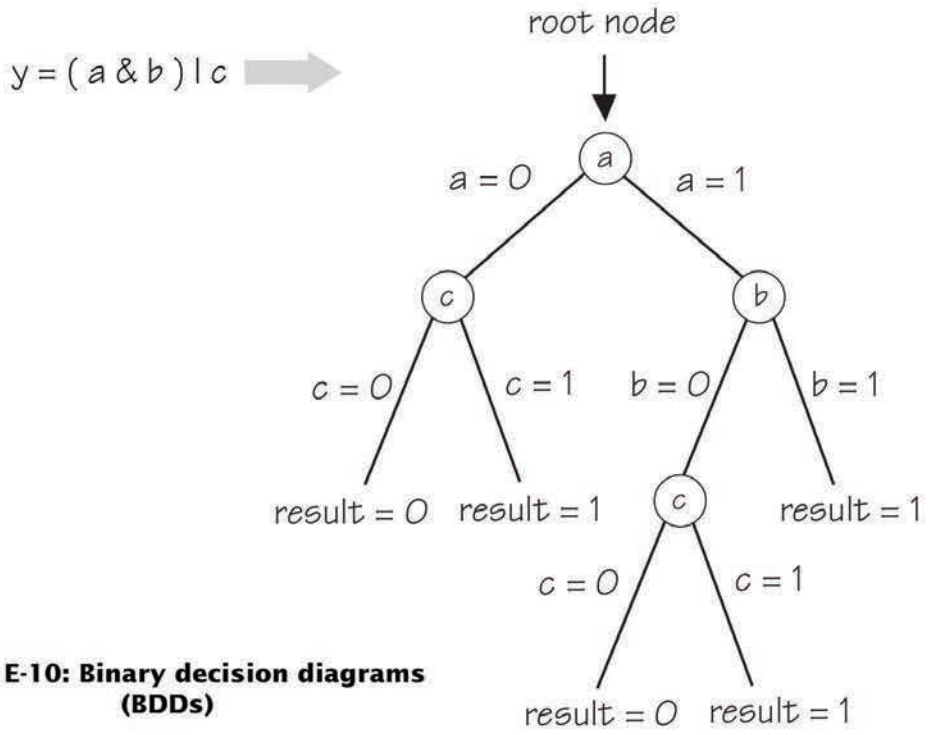


Figure E-10: Binary decision diagrams (BDDs)

The circles representing the inputs are known as *nodes*, and the 0 and 1 values associated with the lines linking two nodes indicate the possible state of the upper node. The 0 and 1 values at the bottom of the diagram indicate the results of the equation for each combination of inputs. The initial node is known as the *root*, the links between nodes may be referred to as *branches*, and the lowest links and their values are referred to as *leafs* or *leaf nodes*.[8]

[8] This terminology makes sense if the diagram is visualized as an inverted tree.

The binary decision diagram form of representation finds many uses. For example, as an alternative to actually evaluating a complex expression, it is possible to quickly determine the output from the expression by marching through the diagram using the values on the inputs as signposts. The reason why this is of interest, is that it may be possible to use these diagrams as inputs to *Max's Algorithm*. As the size of an expression increases, it becomes increasingly unlikely that the entire expression will have a Reed-Müller solution. However, it is more than likely that some portions of the expression may have such solutions. Under the direction of a controlling function, it would be possible to examine the decision diagram for Reed-Müller solutions working from the leaf nodes upwards.

You may feel free to use this program in your own projects. However, it would be nice if you could reference Max's Algorithm and its source, *Bebop to the Boolean Boogie*. Additionally, if you do adopt any of the suggestions above, or make modifications of your own, or use the algorithm for a real application, please contact the author (care of the publisher) and spread a little joy in the world. The most interesting derivations may well find themselves featured in the next edition of *Bebop*, thereby making you famous beyond your wildest dreams (and making your mother very proud).

Appendix F

Linear Feedback Shift Registers (LFSRs)

The *Ouroboros*—a symbol of a serpent or dragon devouring its own tail and thereby forming a circle—has been employed by a variety of ancient cultures around the world to depict eternity or renewal.[1] The equivalent to the Ouroboros in the world of electronics would be the *linear feedback shift register (LFSR)*,[2] in which the output from a standard shift register is cunningly manipulated and fed back into its input in such a way as to cause the function to endlessly cycle through a sequence of patterns.

Many-to-One Implementations

LFSRs are simple to construct and are useful for a wide variety of applications, but are often sadly neglected by designers. One of the more common forms of LFSR is formed from a simple shift register with feedback from two or more points, or *taps*, in the register chain (Figure F-1).

The taps in this example are at bit 0 and bit 2, and can be referenced as [0,2]. All of the register elements share a common *clock* input, which is omitted from the symbol for reasons of clarity. The data input to the LFSR is generated by

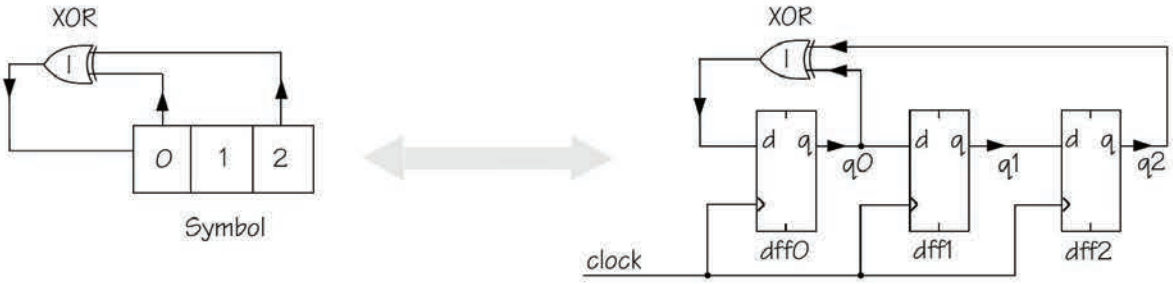


Figure F-1. LFSR with XOR feedback path

[1] Not to be confused with the *Amphisbaena*, a serpent in classical mythology having a head at each end and capable of moving in either direction.

[2] In conversation, LFSR is spelled out as "L-F-S-R".

XOR-ing or XNOR-ing the tap bits; the remaining bits function as a standard shift register. The sequence of values generated by an LFSR is determined by its feedback function (XOR versus XNOR) and tap selection (Figure F-2).

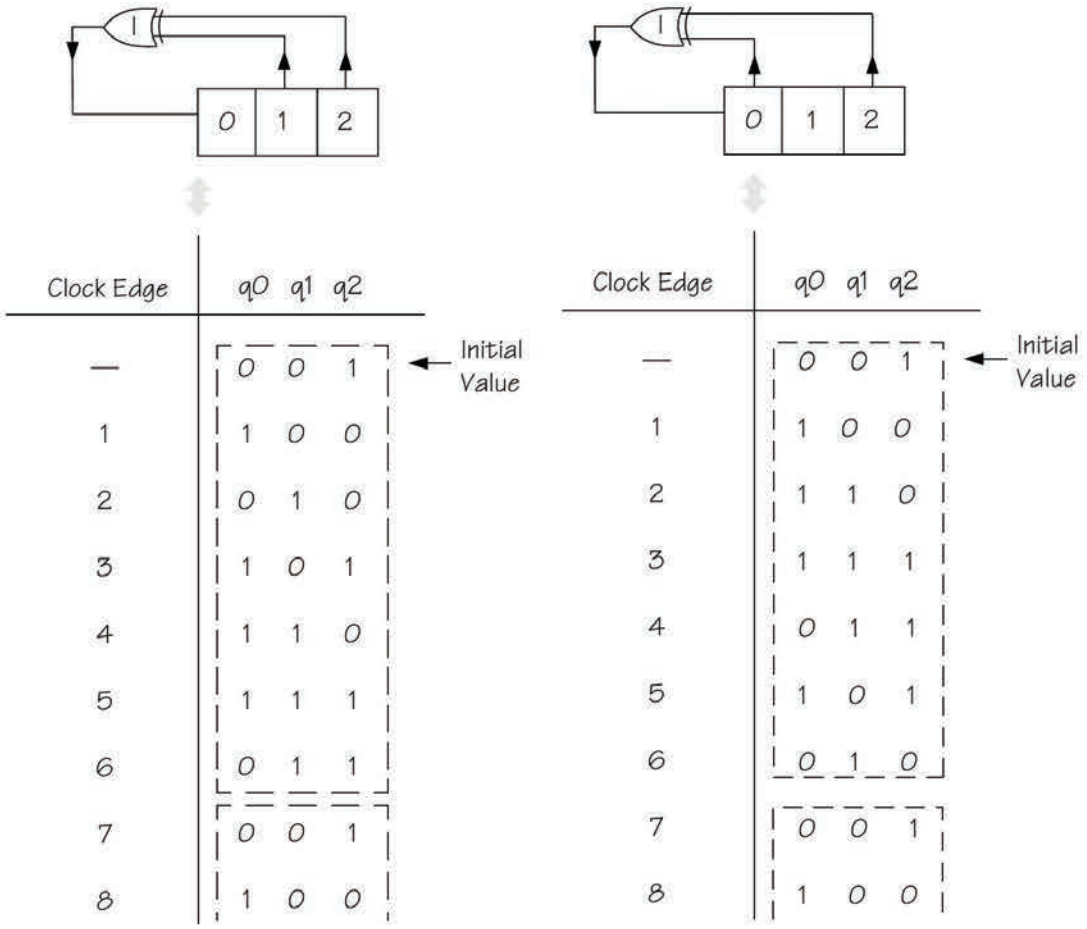


Figure F-2. Comparison of alternative tap selections

Both LFSRs start with the same initial value but, due to the different taps, their sequences rapidly diverge as clock pulses are applied. In some cases an LFSR will end up cycling round a loop comprising a limited number of values. However, both of the LFSRs shown in Figure F-2 are said to be of *maximal length* because they sequence through every possible value (excluding all of the bits being 0) before returning to their initial values.

A binary field with n bits can assume 2^n unique values, but a maximal-length LFSR with n register bits will only sequence through $(2^n - 1)$ values. This is because LFSRs with XOR feedback paths will not sequence through the value where all the bits are 0, while their XNOR equivalents will not sequence through the value where all the bits are 1 (Figure F-3).

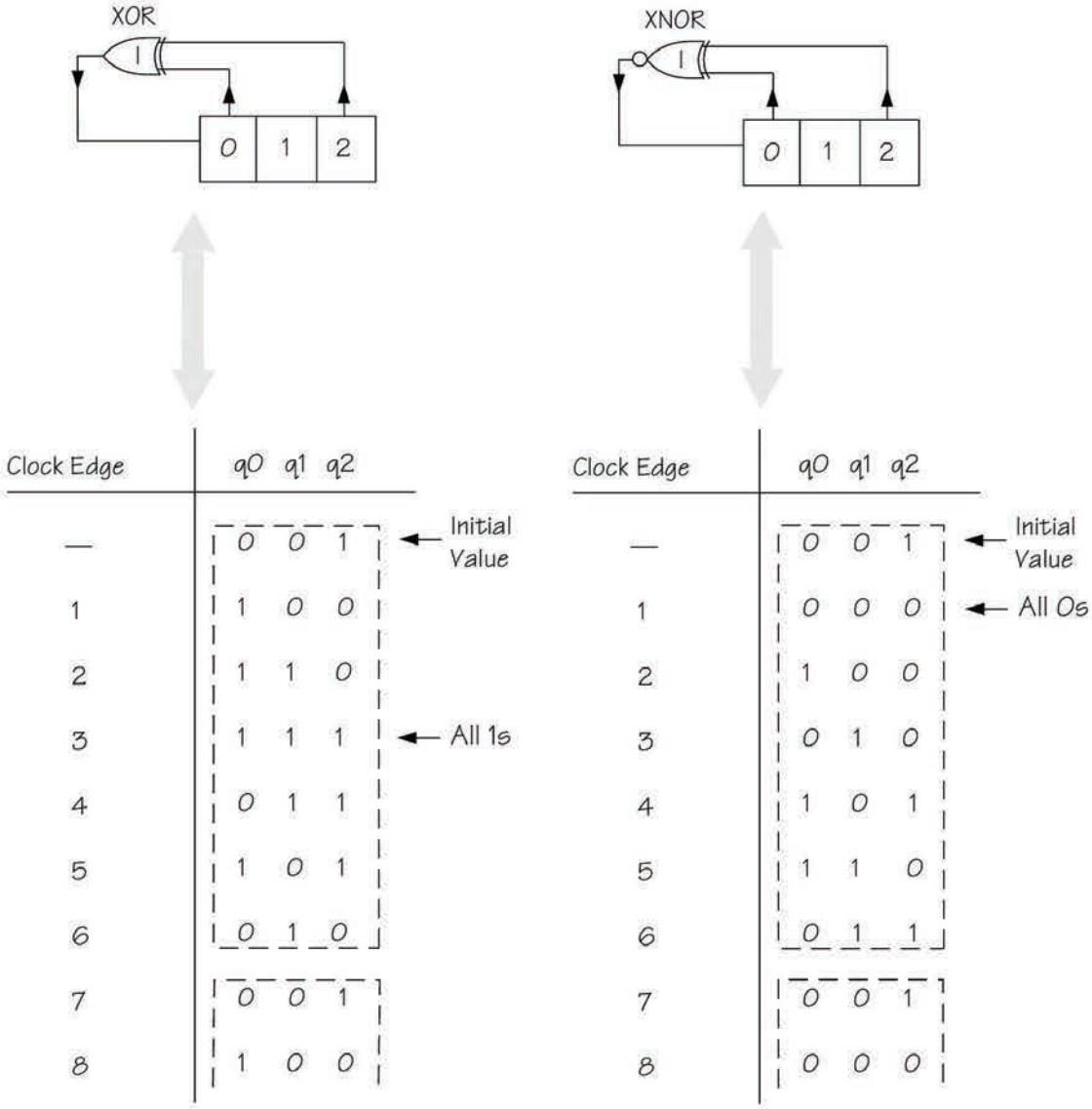


Figure F-3. Comparison of XOR versus XNOR feedback paths

More Taps Than You Know What to Do With

Each LFSR supports a number of tap combinations that will generate maximal-length sequences. The problem is weeding out the ones that do from the ones that don't, because badly chosen taps can result in the register entering a loop comprising only a limited number of states.

# of Bits	Length of Loop		Taps
2	3	*	[0,1]
3	7	*	[0,2]
4	15		[0,3]
5	31	*	[1,4]
6	63		[0,5]
7	127	*	[0,6]
8	255		[1,2,3,7]
9	511		[3,8]
10	1,023		[2,9]
11	2,047		[1,10]
12	4,095		[0,3,5,11]
13	8,191	*	[0,2,3,12]
14	16,383		[0,2,4,13]
15	32,767		[0,14]
16	65,535		[1,2,4,15]
17	131,071	*	[2,16]
18	262,143		[6,17]
19	524,287	*	[0,1,4,18]
20	1,048,575		[2,19]
21	2,097,151		[1,20]
22	4,194,303		[0,21]
23	8,388,607		[4,22]
24	16,777,215		[0,2,3,23]
25	33,554,431		[2,24]
26	67,108,863		[0,1,5,25]
27	134,217,727		[0,1,4,26]
28	268,435,455		[2,27]
29	536,870,911		[1,28]
30	1,073,741,823		[0,3,5,29]
31	2,147,483,647	*	[2,30]
32	4,294,967,295		[1,5,6,31]

Figure F-4. Taps for maximal length LFSRs with 2 to 32 bits

The author created a simple C program to determine the taps for maximal-length LFSRs with 2 to 32 bits. These values are presented for your delectation and delight in Figure F-4 (the * annotations indicate sequences whose length is a prime number).

The taps are identical for both XOR-based and XNOR-based LFSRs, although the resulting sequence will of course differ. As was previously noted, alternative tap combinations may also yield maximum-length LFSRs, although the resulting sequences will vary. For example, in the case of a 10-bit LFSR, there are two 2-tap combinations that result in a maximal-length sequence: [2,9] and [6,9]. There are also twenty 4-tap combinations, twenty-eight 6-tap combinations, and ten 8-tap combinations that satisfy the maximal-length criteria.

One-to-Many Implementations

Consider the case of an 8-bit LFSR, for which the minimum number of taps that will generate a maximal-length sequence is four. In the real world, XOR gates only have two inputs, so a four-input XOR function has to be created using three XOR gates arranged as two levels of logic. Even in those cases where an LFSR does support a minimum of two taps, you may actually wish to use a greater number of taps such as eight (which would result in three levels of XOR logic).

The problem is that increasing the levels of logic in the combinational feedback path can negatively impact the maximum clocking frequency of the function. One solution is to transpose the *many-to-one implementations* discussed above into their *one-to-many counterparts* (Figure F-5).

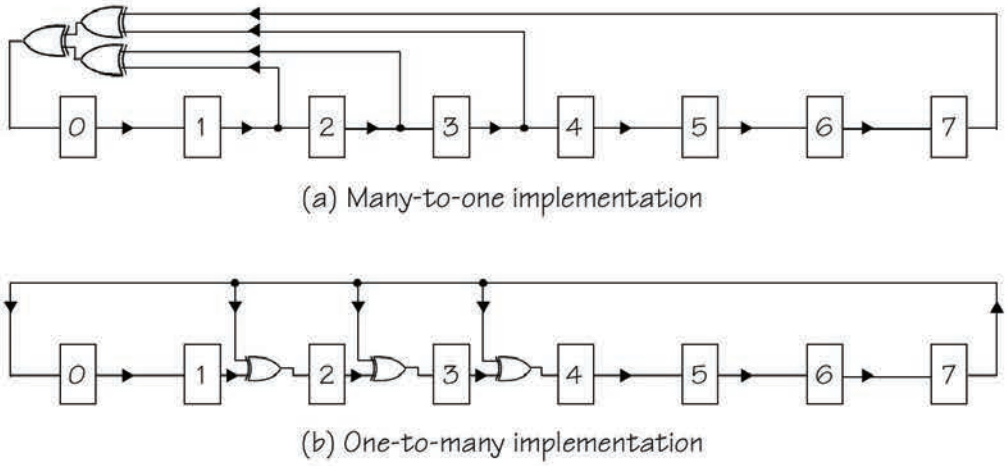


Figure F-5: Many-to-one versus one-to-many implementations

The traditional many-to-one implementation for the eight-bit LFSR has taps at [7,3,2,1]. To convert this into its one-to-many counterpart, the most-significant tap (which is always the most significant bit) is fed back directly into the least significant bit, and is also individually XORed with the other original taps (bits [3,2,1] in this example). Note that although both styles result in maximal-length LFSRs, the actual sequences of values will differ between them. But the main point is that using the one-to-many style means that there is never more than one level of combinational logic in the feedback path, irrespective of the number of taps being employed.

Seeding an LFSR

One quirk with an XOR-based LFSR is that, if it happens to find itself in the all 0s value, it will happily continue to shift all 0s indefinitely (similarly for an XNOR-based LFSR and the all 1s value). This is of particular concern when power is first applied to the circuit. Each register bit can randomly initialize containing either a logic 0 or a logic 1, and the LFSR can therefore "wake up" containing its "forbidden" value. For this reason, it is necessary to initialize an LFSR with a *seed* value.

One method for loading a seed value is to use registers with reset or set inputs. A single control signal can be connected to the reset inputs on some of the registers and the set inputs on others. When this control signal is placed in its active state, the LFSR will load with a hard-wired seed value. However, in certain applications it is desirable to be able to vary the seed value. One technique for achieving this is to include a multiplexer at the input to the LFSR (Figure F-6).

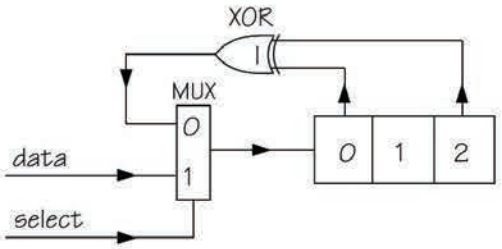


Figure F-6. Circuit for loading alternative seed values

When the multiplexer's data input is selected, the device functions as a standard shift register and any desired seed value may be loaded. After loading the seed value, the feedback path is selected and the device returns to its LFSR mode of operation.

FIFO Applications

The fact that an LFSR generates an unusual sequence of values is irrelevant in many applications. Consider a 4-bit × 16-word *first-in first-out (FIFO)* memory device (Figure F-7).

A brief summary of the FIFO's operation is as follows. The write and read pointers are essentially 4-bit registers whose outputs are processed by 4:16 decoders to select one of the sixteen words in the memory array. The reset input is used to initialize the device, primarily by clearing the write and read pointers such that they both point to the same memory word. The initialization also causes the empty output to be placed in its active state and the full output to be placed in its inactive state.

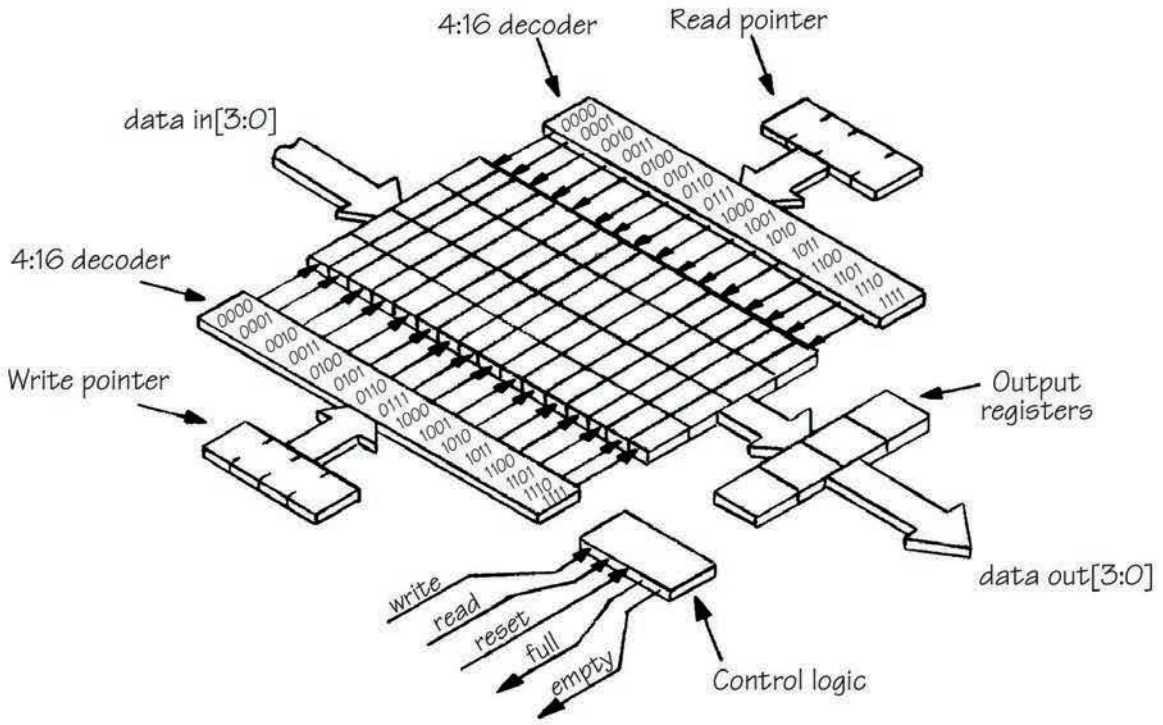


Figure F-7. First-in first-out (FIFO) memory

The write and read pointers chase each other around the memory array in an endless loop. An active edge on the *write* input causes any data on the *data in*[3:0] bus to be written into the word pointed to by the write pointer; the *empty* output is placed in its inactive state (because the device is no longer empty) and the write pointer is incremented to point to the next empty word.

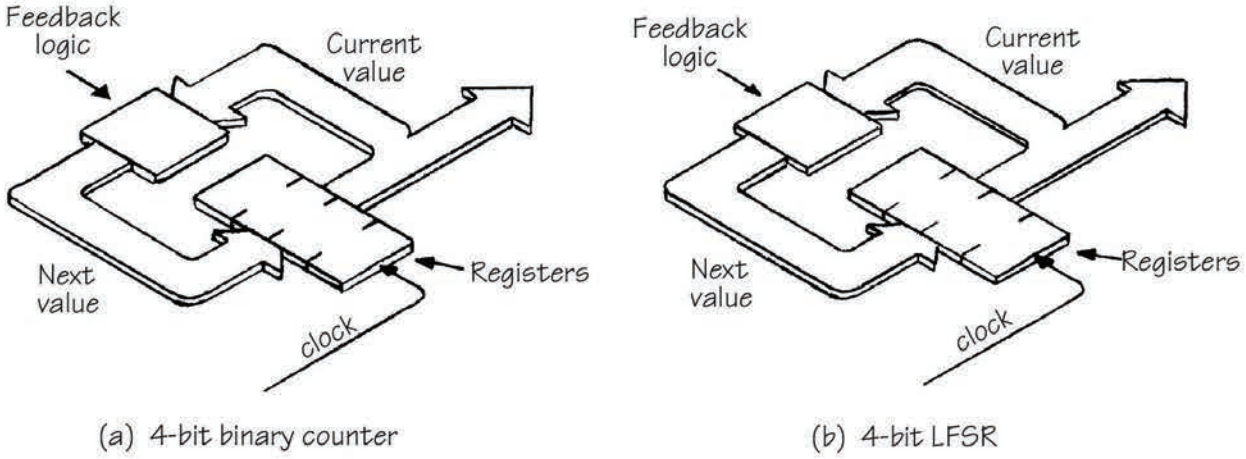
Data can be written into the FIFO until all the words in the array contain values. When the write pointer catches up to the read pointer, the *full* output is placed in its active state (indicating that the device is full) and no more data can be written into the device.

An active edge on the *read* input causes the data in the word pointed to by the read pointer to be copied into the output register; the *full* output is placed in its inactive state and the read pointer is incremented to point to the next word containing data.[3]

[3] These discussions assume *write-and-increment* and *read-and-increment* techniques, but some FIFOs employ an *increment-and-write* and *increment-and-read* approach.

Data can be read out of the FIFO until the array is empty. When the read pointer catches up to the write pointer, the *empty* output is placed in its active state, and no more data can be read out of the device.

The write and read pointers for a 16-word FIFO are often implemented using 4-bit binary counters.[4] However, a moment's reflection reveals that there is no intrinsic advantage to a binary sequence for this particular application, and the sequence generated by a 4-bit LFSR would serve equally well. In fact, the two functions operate in a very similar manner as is illustrated by their block diagrams (Figure F-8).



(a) 4-bit binary counter (b) 4-bit LFSR

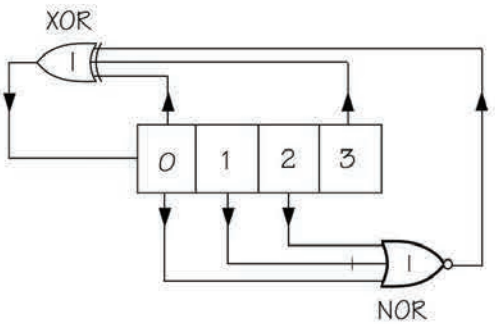
Figure F-8. Binary counter versus LFSR

However, the combinational logic for the 4-bit LFSR consists of a single, two-input XOR gate, while the combinational logic for the 4-bit binary counter requires a number of AND and OR gates. This means that the LFSR requires fewer tracks and is significantly more efficient in terms of silicon real estate. Additionally, the LFSR's feedback only passes through a single level of logic, while the binary counter's feedback passes through multiple levels of logic. This means that the new data value is available sooner for the LFSR, which can therefore be clocked at a higher frequency. These differentiations become even more pronounced for FIFOs with more words requiring pointers with more bits. Thus, LFSR's are the obvious choice for the discerning designer of FIFOs.

[4] The implementation of a 4-bit binary counter was discussed in Chapter 11.

Modifying LFSRs to Sequence 2n Values

The sole downside to the 4-bit LFSRs in the FIFO scenario above is that they will sequence through only 15 values ($2^4 - 1$), as compared to the binary counter's sequence of 16 values (2^4). Designers may not regard this to be a problem, especially in the case of larger FIFOs. However, if it is required for an LFSR to sequence through every possible value, there is a simple solution (Figure F-9).



Clock Edge	q0	q1	q2	q3	
—	0	0	0	1	
1	0	0	0	0	← All 0's
2	1	0	0	0	
3	1	1	0	0	
4	1	1	1	0	
5	1	1	1	1	
6	0	1	1	1	
7	1	0	1	1	
8	0	1	0	1	2^4=16 Values
9	1	0	1	0	
10	1	1	0	1	
11	0	1	1	0	
12	0	0	1	1	
13	1	0	0	1	
14	0	1	0	0	
15	0	0	1	0	
16	0	0	0	1	

Figure F-9. LFSR modified to sequence 2n values

For the value where all of the bits are 0 to appear, the preceding value must have comprised a logic 1 in the most-significant bit (MSB)[5] and logic 0s in the remaining bit positions. In an *unmodified* LFSR, the next clock would result in a logic 1 in the least-significant bit (LSB) and logic 0s in the remaining bit positions. However, in the *modified* LFSR, the output from the NOR is a logic 0 for every case but two: the value preceding the one where all the bits are 0 and the value where all the bits are 0. These two values force the NOR's output to a logic 1, which inverts the usual output from the XOR. This in turn causes the sequence to first enter the all-0s value and then resume its normal course. (In the case of LFSRs with XNOR feedback paths, the NOR can be replaced with an AND, which causes the sequence to cycle through the value where all of the bits are 1.)

[5] As is often the case with any form of shift register, the MSB in these examples is taken to be on the right-hand side of the register, and the LSB is taken to be on the left-hand side (this is opposite to the way we usually do things).

Accessing the Previous Value

In some applications it is required to make use of a register's previous value. For example, in certain FIFO implementations, the "full" condition is detected when the write pointer is pointing to the location preceding the location pointed to by the read pointer.[6] This implies that a comparator must be used to compare the *current* value in the write pointer with the *previous* value in the read pointer. Similarly, the "empty" condition may be detected when the read pointer is pointing to the location preceding the location pointed to by the write pointer. This implies that a second comparator must be used to compare the *current* value in the read pointer with the *previous* value in the write pointer.

In the case of binary counters (assuming that, for some reason, we decided to use them for a FIFO application), there are two techniques by which the previous value in the sequence may be accessed. The first requires the provision of an additional set of so-called *shadow registers*. Every time the counter is incremented, its current contents are first copied into the shadow registers. Alternatively, a block of combinational logic can be used to decode the previous value from the current value. Unfortunately, both of these techniques involve a substantial overhead in terms of additional logic. By comparison, LFSRs inherently remember their previous value. All that is required is the addition of a single register bit appended to the MSB (Figure F-10).

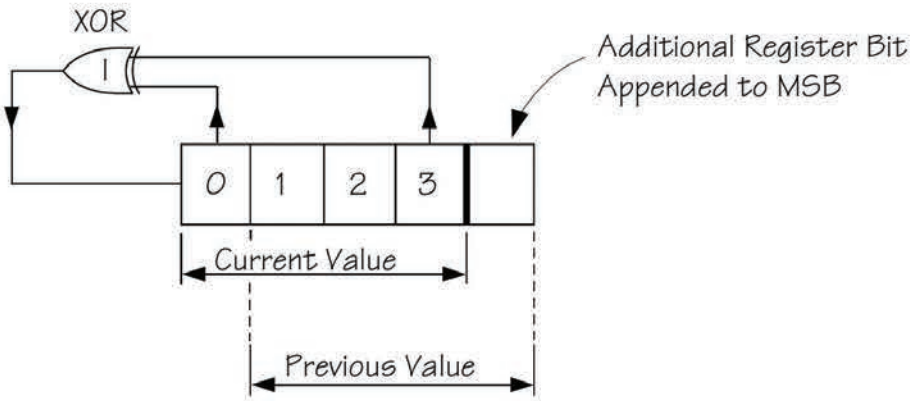


Figure F-10. Accessing an LFSR's previous value

[6] Try saying that quickly!

Encryption and Decryption Applications

The unusual sequence of values generated by an LFSR can be gainfully employed in the encryption (scrambling) and decryption (unscrambling) of data. A stream of data bits can be encrypted by XOR-ing them with the output from an LFSR (Figure F-11).

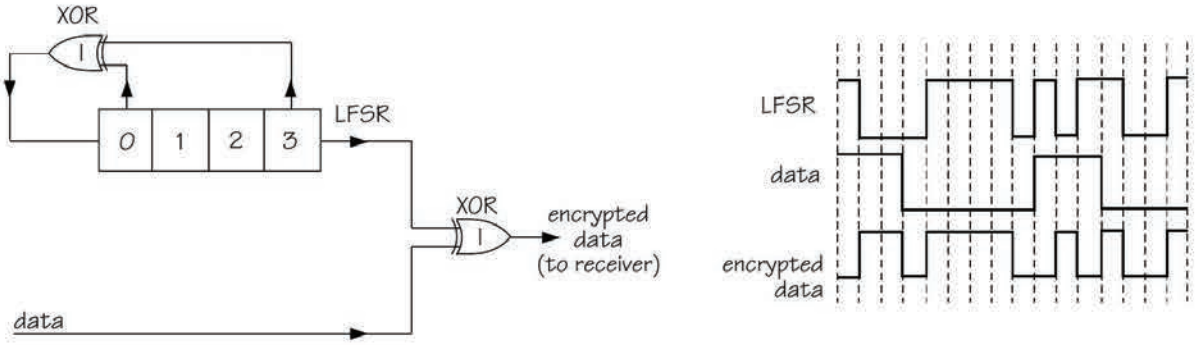


Figure F-11. Data encryption using an LFSR

The stream of encrypted data bits seen by a receiver can be decrypted by XOR-ing them with the output of an identical LFSR.[7]

Cyclic Redundancy Check Applications

A traditional application for LFSRs is in *cyclic redundancy check (CRC)* calculations, which can be used to detect errors in data communications. The stream of data bits being transmitted is used to modify the values fed back into an LFSR (Figure F-12).

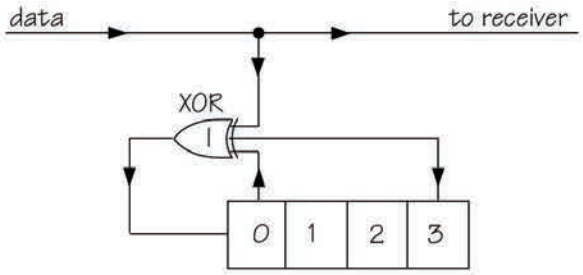


Figure F-12. Cyclic redundancy check (CRC) calculations

The final CRC value stored in the LFSR is known as a *checksum*, and is dependent on every bit in the data stream. After all of the data bits have been transmitted, the transmitter sends its checksum value to the receiver. The receiver contains an identical CRC

[7] This is obviously a very trivial form of encryption that's not very secure, but it's "cheap-and-cheerful" and may be useful in certain applications.

calculator and generates its own checksum value from the incoming data. Once all of the data bits have arrived, the receiver compares its internally generated checksum value with the checksum sent by the transmitter to determine whether any corruption occurred during the course of the transmission. This form of error detection is very efficient in terms of the small number of bits that have to be transmitted in addition to the data.

In the real world, a 4-bit CRC calculator would not be considered to provide sufficient confidence in the integrity of the transmitted data. This is due to the fact that a 4-bit LFSR can only represent 16 unique values, which means that there is a significant probability that multiple errors in the data stream could result in the two checksum values being identical. However, as the number of bits in a CRC calculator increases, the probability that multiple errors will cause identical checksum values approaches zero. For this reason, CRC calculators typically use a minimum of 16-bits providing 65,536 unique values.

There are a variety of standard communications protocols, each of which specifies the number of bits employed in their CRC calculations and the taps to be used. The taps are selected such that an error in a single data bit will cause the maximum possible disruption to the resulting checksum value. Thus, in addition to being referred to as *maximal-length*, these LFSRs may also be qualified as *maximal-displacement*.

In addition to checking data integrity in communications systems, CRCs find a wide variety of other uses: for example, the detection of computer viruses. For the purposes of this discussion, a computer virus may be defined as a self-replicating program released into a computer system for a number of purposes. These purposes range from the simply mischievous, such as displaying humorous or annoying messages, to the downright nefarious, such as corrupting data or destroying (or subverting) the operating system.

One mechanism by which a computer virus may both hide and propagate itself is to attach itself to an existing program. Whenever that program is executed, it first triggers the virus to replicate itself, yet a cursory check of the system shows only the expected files to be present. In order to combat this form of attack, a unique checksum can be generated for each program on the system, where the value of each checksum is based on the binary instructions forming the program with which it is associated. At some later date, an anti-virus program can be used to recalculate the checksum values for each program,

and to compare them to the original values. A difference in the two values associated with a program may indicate that a virus has attached itself to that program.[8]

Data Compression Applications

The CRC calculators discussed above can also be used in a data compression role. One such application is found in the circuit board test strategy known as *functional test*. The board is plugged into a functional tester by means of its edge connector. The tester applies a pattern of signals to the board's inputs, allows sufficient time for any effects to propagate around the board, and then compares the actual values seen on the outputs with a set of expected values stored in the system. This process is repeated for a series of input patterns, which may number in the tens or hundreds of thousands (Figure F-13).

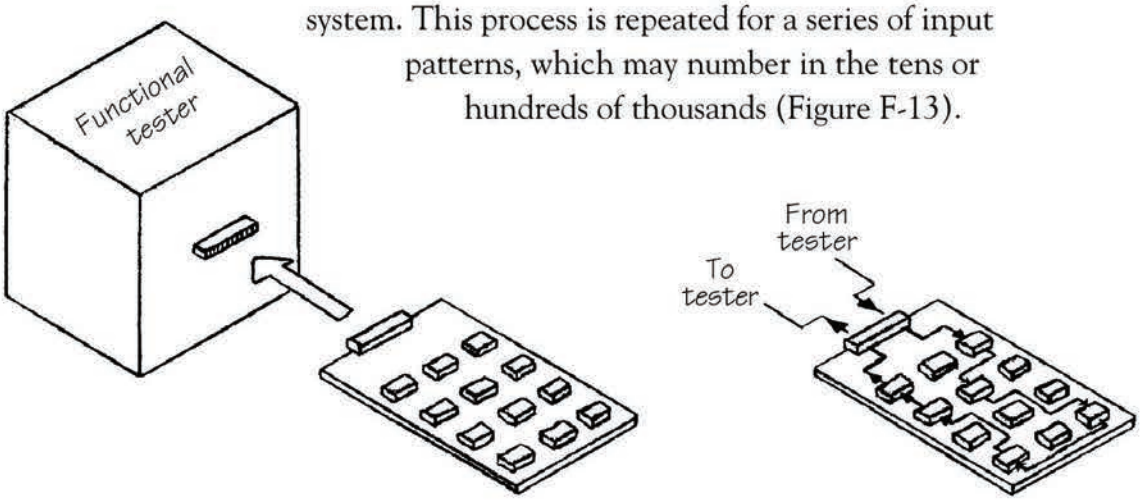


Figure F-13. Functional test

The illustration above is simplified for reasons of clarity. In practice, the edge connector may contain hundreds of pins while the board may contain thousands of components and tracks. If the board fails the preliminary tests, a more sophisticated form of analysis known as *guided probe* may be employed to identify the cause of the failure (Figure F-14).

[8] Unfortunately, the creators of computer viruses are quite sophisticated, and some viruses are armed with the ability to perform their own CRC calculations. When a virus of this type attaches itself to a program, it can pad itself with dummy binary values, which are selected so as to cause an anti-virus program to return an identical checksum value to the original.

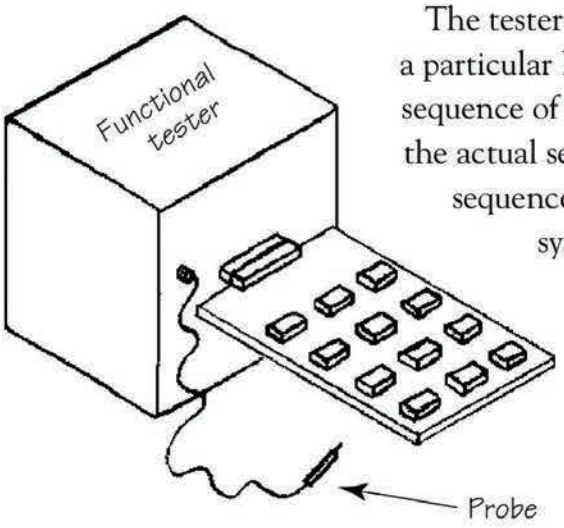


Figure F-14. Guided probe analysis

The tester instructs the operator to place the probe at a particular location on the board, and then the entire sequence of test patterns is rerun. The tester compares the actual sequence of values seen by the probe with a sequence of expected values that are stored in the system. This process (placing the probe and running the tests) is repeated until the tester has isolated the faulty component or track.

A major consideration when supporting a guided probe strategy is the amount of expected data that must be stored. Consider a test sequence comprising 10,000 patterns driving a board containing 10,000 tracks. If the data were not compressed, the system would have to store 10,000 bits of expected data per track, which amounts to 100,000,000 bits of data for the board. Additionally, for each application of the guided probe, the tester would have to compare the 10,000 data bits observed by the probe with the 10,000 bits of expected data stored in the system. Thus, using data in an uncompressed form is an expensive option in terms of storage and processing requirements.

One solution to these problems is to employ LFSR-based CRC calculators. The sequence of expected values for each track can be passed through a 16-bit CRC calculator implemented in software. Similarly, the sequence of actual values seen by the guided probe can be passed through an identical CRC calculator implemented in hardware. In this case, the calculated checksum values are also known as *signatures*, and a guided probe process based on this technique is known as *signature analysis*. Irrespective of the number of test patterns used, the system has to store only two bytes of data for each track. Additionally, for each application of the guided probe, the tester has to compare only the two bytes of data gathered by the probe with two bytes of expected data stored in the system. Thus, compressing the data results in storage requirements that are orders of magnitude smaller and comparison times that are orders of magnitude faster than the uncompressed data approach.

Built-In Self-Test Applications

One test strategy which may be employed in complex integrated circuits is that of *built-in self-test (BIST)*. Devices using BIST contain special test generation and result gathering circuits (Figure F-15).

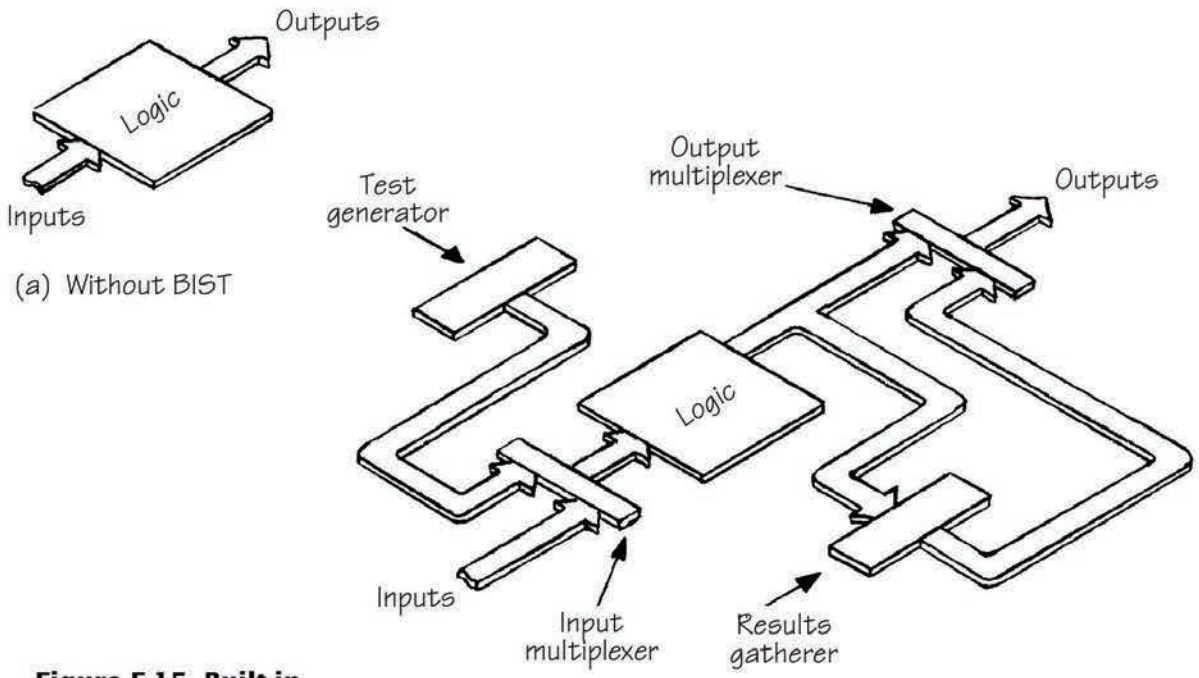


Figure F-15. Built-in self-test (BIST)

A multiplexer is used to select between the standard inputs and those from the *test generator*. A second multiplexer selects between the standard outputs and those from the *results gatherer*. Both the test generator and results gatherer can be implemented using LFSRs (Figure F-16).[9]

The LFSR forming the test generator is used to create a sequence of test patterns, while the LFSR forming the results gatherer is used to capture the results. The results-gathering LFSR features modifications that allow it to accept parallel data. (Note that the two LFSRs are not obliged to contain the same number of bits, because the number of inputs to the logic being tested may be different to the number of outputs from the logic.)

[9] The standard inputs and outputs along with their multiplexers have been omitted from this illustration for reasons of clarity.

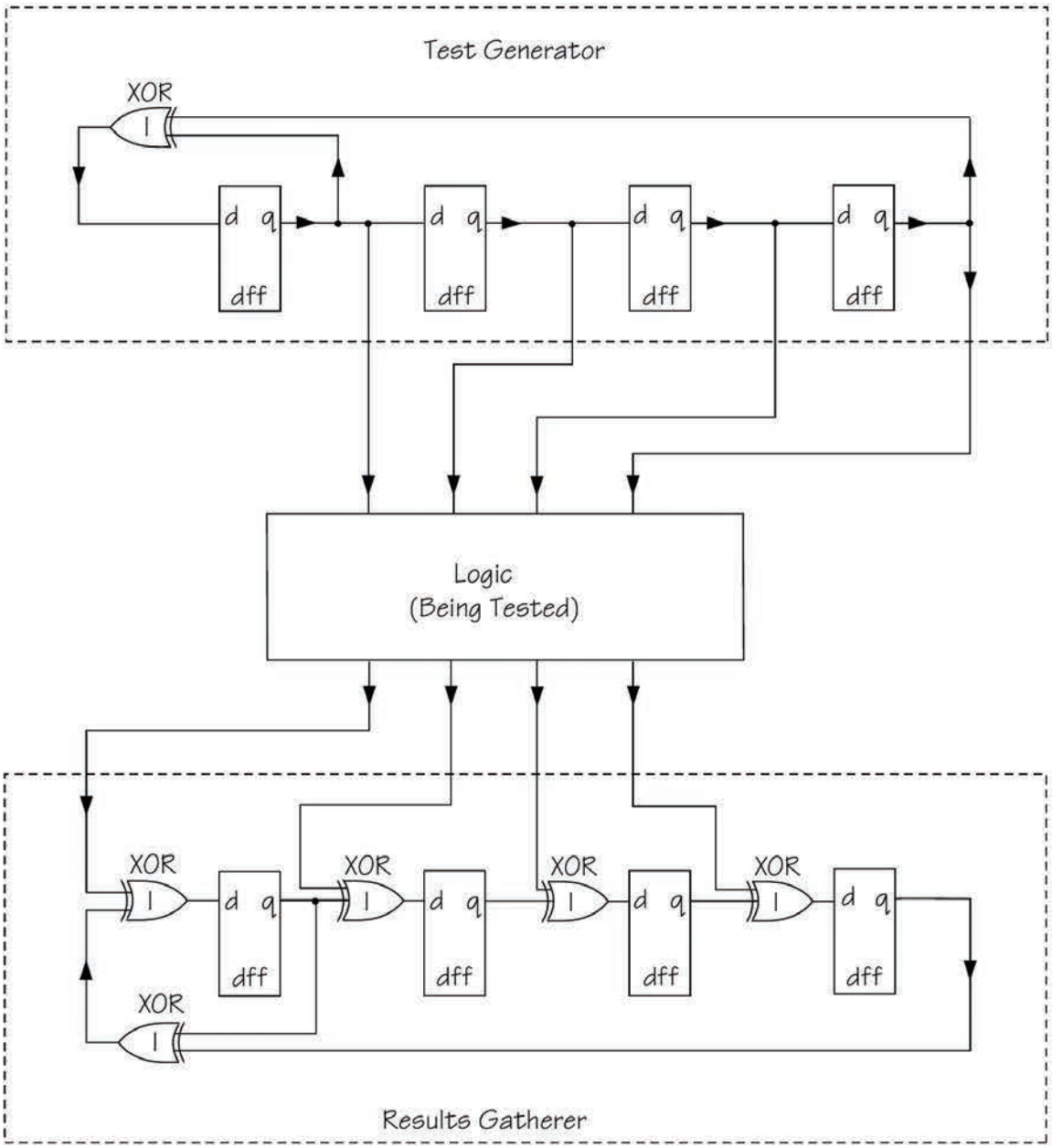


Figure F-16. BIST implemented using LFSRs

Pseudo-Random Number Applications

Many computer programs rely on an element of randomness. Computer games such as "Space Invaders" employ random events to increase the player's enjoyment. Graphics programs may exploit random numbers to generate intricate patterns. All forms of computer simulation may utilize random numbers to more accurately represent the real world. For example, digital simulations[10] may benefit from the portrayal of random stimulus such as external interrupts. Random stimulus can result in more realistic design verification, which can uncover problems that may not be revealed by more structured tests.

Random number generators can be constructed in both hardware and software. The majority of these generators are not truly random, but they give the appearance of being random and are therefore said to be *pseudo-random*. In fact, pseudo-random numbers have an advantage over truly random numbers, because the majority of computer applications typically require repeatability. For example, a designer repeating a digital simulation would expect to receive identical answers to those from the previous run. However, designers also need the ability to modify the seed value of the pseudo-random number generator so as to spawn different sequences of values as required.

There are a variety of methods available for generating pseudo-random numbers. A popular cheap-and-cheerful technique uses the remainder from a division operation as the next value in a pseudo-random sequence. For example, a C function which returns a pseudo-random number in the range 0 to 32767 could be written as follows:[11]

```
int rand ()
{
    next = next * 1103515245 + 12345;
    return (unsigned int) (next/65535) % 32768;
}
```

[10] Digital simulation is based on a program called a *logic simulator*, which is used to build a virtual representation of an electronic design in the computer's memory. The simulator then applies stimulus to the design's virtual inputs, simulates the effect of these signals as they propagate through the design, and checks the responses at the design's virtual outputs.

[11] The C Programming Language (Second Edition) by Brian W. Kernighan and Dennis M. Ritchie.

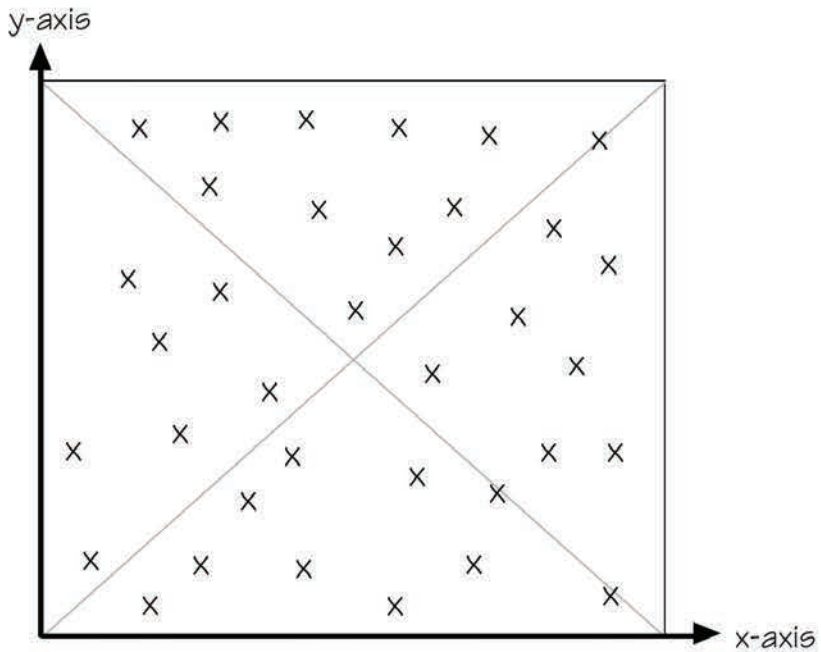
The variable next would be declared as global and initialized with some seed value. Also, an additional function would be provided to allow the programmer to load next with a new seed value if required. Every time this function is accessed it returns a new pseudo-random value. Unfortunately, pseudo-random implementations based on division do not always produce a "white spectrum". The series of generated values may be composed of a collection of sub-series, in which case the results will not be of maximal length. By comparison, the sequence of values generated by a software implementation of a maximal-length LFSR provides a reasonably good pseudo-random source, but is somewhat more expensive in terms of processing requirements.

A simple technique for checking the "randomness" of a pseudo-random number generator is as follows. Assume the existence of a function called rand() which returns a pseudo-random number as a 16-bit value. The most significant and least significant bytes may be considered to represent the y-axis and x-axis of a graph respectively. To test the randomness of the function, a simple C program segment may be created to plot a reasonable sample of random numbers:

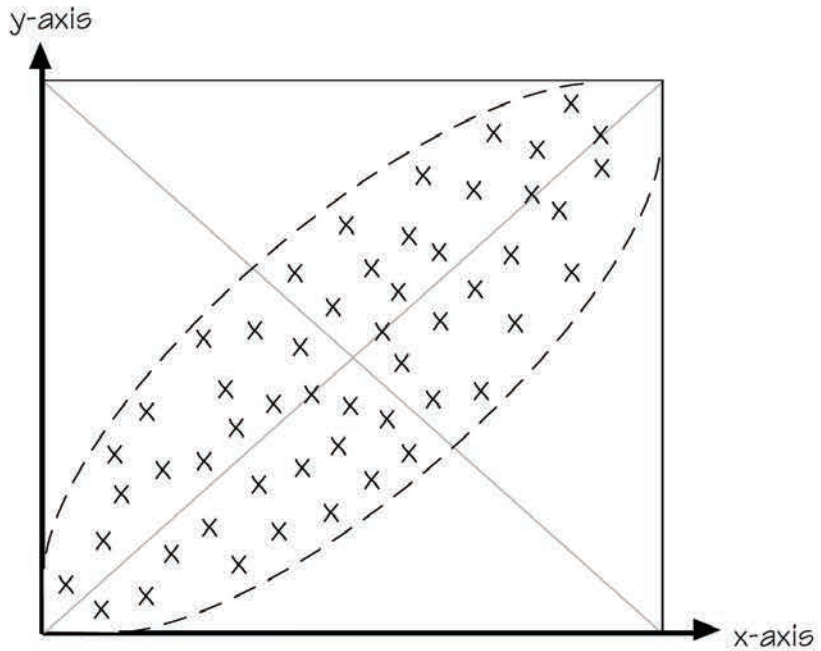
```
int x_axis, y_axis, i, my_number;          /* Declare the variables.              */
for (i = 1; i <= 1000; i++) {              /* Perform a loop 1,000 times.         */
   my_number = rand();                     /* Get a pseudo-random number.         */
   x_axis = my_number & 0x00FF;            /* Extract the least-significant byte. */
   y_axis = (my_number & 0xFF00) >> 8;     /* Extract the most-significant byte.  */
   plot(x_axis, y_axis);                   /* Plot the value.                     */
}
```

This example assumes the existence of a function plot() that will cause the point represented by the variables x_axis and y_axis to be displayed on a computer screen. (Note that the masking operation used when extracting the most significant byte is included for reasons of clarity, but is in fact redundant.)

If the resulting points are uniformly distributed across the plotting area, then the pseudo-random generator is acceptable. But if the points are grouped around the diagonal axis or form clusters, then the generator is less than optimal (Figure F-17).



(a) Good



(b) Not-so-good

Figure F-17. Checking for "randomness"

Last But Not Least

LFSRs are simple to construct and are useful for a wide variety of applications, but be warned that choosing the optimal polynomial (which ultimately boils down to selecting the tap points) for a particular application is a task that is usually reserved for a master of the mystic arts, not to mention that the maths can be hairy enough to make a grown man break down and cry (and don't even get the author started on the subject of *cyclotomic polynomials*,[12] which are key to the tap-selection process).

[12] Mainly because he hasn't got the faintest clue what a *cyclotomic polynomial* is!

Appendix

G

Pass-Transistor Logic

The term *pass-transistor logic* refers to techniques for connecting MOS transistors such that *data* signals pass between their *source* and *drain* terminals. These techniques minimize the number of transistors required to implement a function, but they are not recommended for the novice or the unwary because strange and unexpected effects can ensue. Pass-transistor logic is typically employed by designers of full-custom integrated circuits or ASIC cell libraries. The following examples introduce the concepts of pass-transistor logic, but they are not intended to indicate recommended design practices.

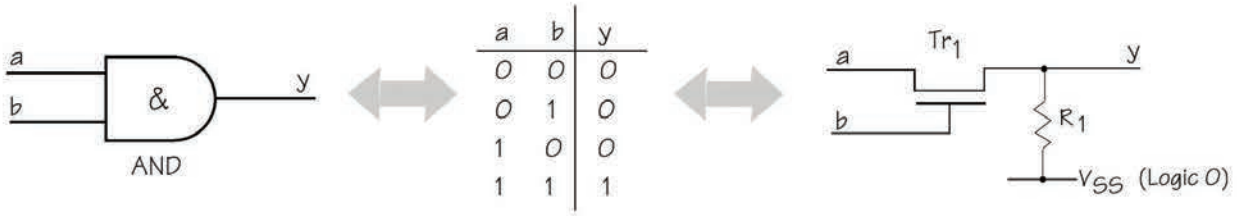


Figure G-1. Pass-transistor implementation of an AND gate

In the case of the AND (Figure G-1), the resistance of R_1 is assumed to be sufficiently high that its effect on output y is that of a very weak logic 0. When input *b* is set to logic 0, the NMOS transistor Tr_1 is turned OFF and y is "pulled down" to logic 0 by resistor R_1. When input *b* is set to logic 1, Tr_1 is turned ON and output y is connected to input *a*. In this case, a logic 0 on input *a* leaves y at logic 0, but a logic 1 on *a* will overdrive the effect of resistor R_1 and force y to a logic 1.

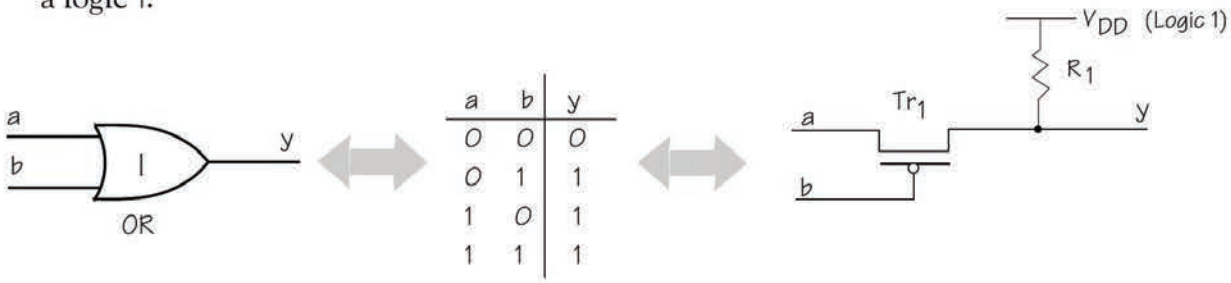


Figure G-2. Pass-transistor implementation of an OR gate

In the case of the OR (Figure G-2), a logic 1 applied to input *b* turns the PMOS transistor Tr_1 OFF, and output y is "pulled up" to logic 1 by resistor R_1. When input *b* is set to logic *0*, Tr_1 is turned ON and output y is connected to input *a*. In this case, a logic 1 on input a leaves y at logic 1, but a logic *0* on *a* will overdrive the effect of resistor R_1 and force y to a logic *0*.

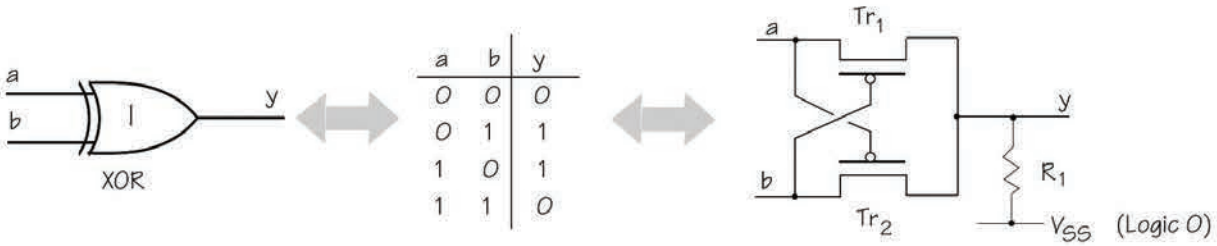


Figure G-3. Pass-transistor implementation of an XOR gate

In the case of the XOR (Figure G-3), logic 1s applied to inputs *a* and *b* turn Tr_1 and Tr_2 OFF, respectively, leaving output y to be "pulled down" to logic *0* by resistor R_1. When input *b* is set to logic *0*, Tr_1 is turned ON and output y is connected to input *a*. In this case, a logic *0* on input *a* leaves y at logic *0*, but a logic 1 on *a* will overdrive the effect of resistor R_1 and force y to a logic 1. Similarly, when input *a* is set to logic *0*, Tr_2 is turned ON and output y is connected to input *b*. In this case, a logic *0* on input *b* leaves y at logic *0*, but a logic 1 on *b* will overdrive the effect of resistor R_1 and force y to a logic 1 (phew!).

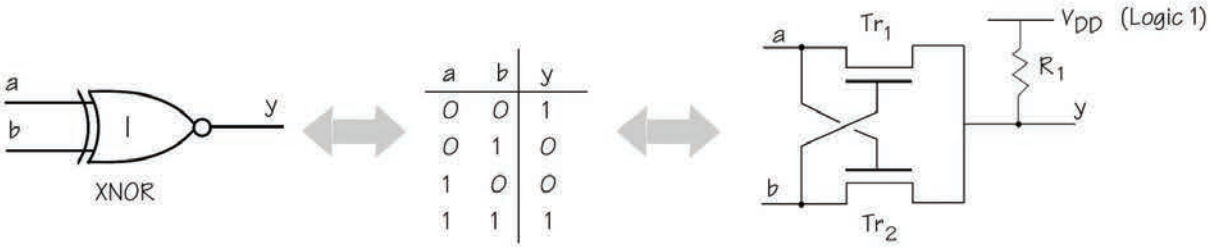
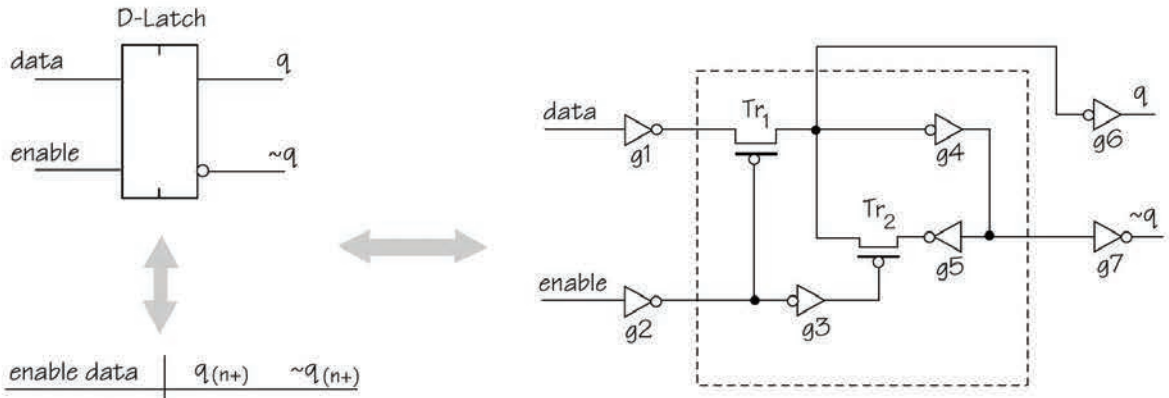


Figure G-4. Pass-transistor implementation of an XNOR gate

The case of the XNOR (Figure G-4) is simply the inverse of that for the XOR. Logic *0*s applied to inputs *a* and *b* turn transistors Tr_1 and Tr_2 OFF, leaving output y to be "pulled up" to logic 1 by resistor R_1. When input *b* is set to logic 1, Tr_1 is turned ON and output y is connected to input *a*. In this case, a logic 1 on

input *a* leaves y at logic 1, but a logic *0* on a will overdrive the effect of resistor R_1 and force y to a logic *0*. Similarly, when input *a* is set to logic 1, Tr_2 is turned ON and output y is connected to input *b*. In this case, a logic 1 on input *b* leaves y at logic 1, but a logic *0* on *b* will overdrive the effect of resistor R_1 and force y to a logic *0*.



enable	data	q(n+)	~q(n+)
0	?	q(n)	~q(n)
1	0	0	1
1	1	1	0

Figure G-5. Pass-transistor implementation of a D-type latch

Before examining the pass-transistor implementation of a D-type latch with an active-high enable shown in Figure G-5, we should briefly review our definition of these functions. First, when the *enable* input is in its active state (a logic 1), the value on the *data* input is passed through the device to appear on the *q* and *~q* outputs.[1] Second, if the *data* input changes while *enable* remains active, the outputs will respond to reflect the new value. Third, when *enable* is driven to its inactive state, the outputs remember their previous values and no longer respond to any changes on the *data* input.

Now onto the pass-transistor implementation.[2] When *enable* is in its active state, the output from *g2* turns transistor Tr_1 ON, and the output from *g3* turns transistor Tr_2 OFF. Thus, the value on the *data* input passes through *g1*, Tr_1, and *g6* to appear on the *q* output, and through *g1*, Tr_1, *g4*, and *g7* to appear at

[1] As usual, the *data* appears in inverted form on the *~q* (complementary) output.

[2] Note the use of assertion-level logic symbols in the circuit diagram. These symbols were introduced in Appendix A.

the ~q output.[3] Any changes on data will be reflected on q and ~q after the delays associated with the gates and transistors have been satisfied. When enable is driven to its inactive state, the output from g2 turns Tr_1 OFF and the output from g3 turns Tr_2 ON. Disabling Tr_1 blocks the path from the data input, while enabling Tr_2 completes the self-sustaining loop formed by g4 and g5. It is this loop that acts as the memory for the device.

Compare this pass-transistor implementation with its standard counterpart introduced in Chapter 11. The standard implementation required one NOT, two ANDs, and two NORs totaling twenty-two transistors. By comparison, the pass-transistor implementation requires two discrete transistors and seven NOTs totaling only sixteen transistors. In fact, the heart of the pass transistor implementation consisting of Tr_1, Tr_2, g3, g4, and g5 requires only eight transistors. The remaining gates, g1, g2, g6, and g7, are used only to buffer the latch's input and outputs from the outside world.[4]

[3] Note the use of the tilde '~' character to indicate the complementary output ~q. The use of tildes was discussed in Appendix A.

[4] NOTs are used in preference to BUFs because NOTs require fewer transistors and are faster (these considerations were introduced in Chapter 6).