Chapter 2. Circuit Implementation Alternatives and Technologies

Overview of chip design styles

Circuit designers have a range of methods for implementing digital logic, with widely differing capabilities and costs. In the following sections we will discuss many of these different alternatives. This will serve both to explain what FPGAs are, and also to show how FPGAs enable new opportunities that were not possible with other technologies. Note that throughout this chapter, the pictures are meant merely to illustrate the concepts discussed. Many (less important) details are omitted, and the quantities of resources are often reduced to aid visibility.

The different methods of implementing digital logic can have significantly different capacity, performance, and costs (both in time and money), as well as differing levels of automation of the design process. Full custom designs offer the highest performance and capacity possible. However, they also take the longest time to be fabricated, have little or no automation in the design process, and cost the most money, in both per-chip and NRE (non-recurring engineering) costs. NRE's are initial costs to set up the production process, and thus are paid once for the first chip produced. Other design styles sacrifice some of the performance and capacity of full-custom design in order to lower costs, speed up the design process, and reduce the fabrication time. Many such technologies will be covered in the following sections.

Full custom

In full custom design, the designer specifies all the details of how an integrated circuit will be fabricated. This gives the designer the maximum flexibility, yielding the highest possible performance and gate capacity. Thus, for some applications, this is the only possible method of implementing the desired functionality, since other technologies may not accommodate the application's capacity or performance demands. However, this power and flexibility does come at a cost. Since the circuit will be custom fabricated, sharing no features with other designs, the fabrication will be costly, and will take a significant amount of time. Other technologies will share some features from design to design, in general yielding lower costs due to economies of scale, and faster turnaround due to prefabrication of shared features.

The designer of a full custom implementation specifies exactly where the different features of the circuit will be by specifying where the various materials will be placed on the chip. That is, the designer decides where diffusion, polysilicon, and metal will be applied to create the desired functionality. There are some design rules that restrict the allowed circuit designs. However, these rules are local constraints on circuit geometries required to avoid structures that cannot be fabricated. For example, wires must be a certain width, and must be spaced apart by a certain distance, in order to avoid breaks or short circuits in signal lines. However, the designer still has a considerable amount of flexibility in deciding how to design the circuit.

Because of the high degree of flexibility in full custom circuits, the process of designing such a circuit is quite complex and time-consuming. Since the designer must decide where each circuit feature will be placed, there are a huge number of decisions to be made. Thus, it takes a large amount of time and effort to develop a full custom design, time beyond what is required simply to fabricate the circuit. Since each fabrication of the circuit is quite costly, the designer will also expend a significant amount of time to make sure the design is correct before fabrication. While software can be used to simplify some of this process, most designers of full custom circuits are using this implementation technology specifically to achieve the highest quality implementation possible, and thus avoid inefficiency due to software designed circuits. Thus, software support for full-custom design is in general restricted to validation of the circuit, and actual design of only non-critical subcircuits, places where some inefficiency can be tolerated.

The process of fabricating the design is done in multiple steps, building up the layers necessary to implement the circuit (Figure 1). Early stages make wells inside which the transistors will be placed. Next, diffusion and polysilicon layers will be added to build the actual transistors. Finally, metal layers will be created to route signals between the different transistors. Note that many processes have multiple metal layers, so steps **h**-**j** will be repeated for each metal layer.

Each of these layers is created by a photolithographic process (Figure 2) that patterns the layer of material. Photoresist, an acid resistant substance that changes properties when exposed to a light source, is applied on top of the layer material to be patterned. The photoresist is then exposed to a light source shone through a mask, so that the mask's image selectively alters the photoresist. Unaltered portions of the photoresist are removed, and the underlying layer is etched by an acid. Since the areas still coated with photoresist will not be affected by the acid, the layer is selectively etched in the pattern described by the mask.

Because all layers of the design of a full custom circuit are specified by the designer, there is no sharing of masks or other layer processing between different designs. Thus, full custom designs are in general the most expensive and time-consuming, since there are little economies of scale. Many other design styles will standardize some or all processing steps, speeding up the processing, and lowering costs.

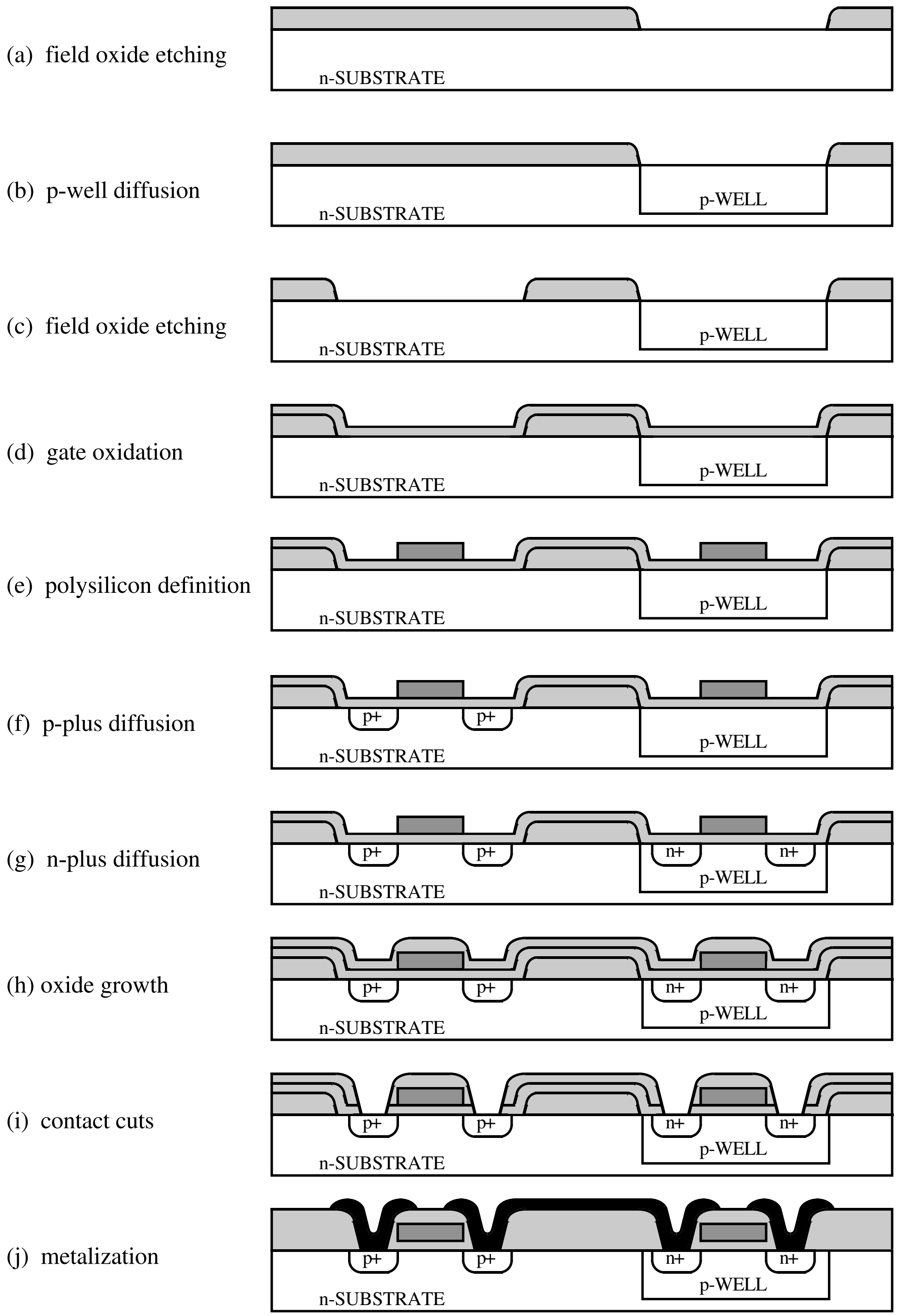


Figure 1. Steps in the fabrication of a full custom design (p-well CMOS) [Weste85].

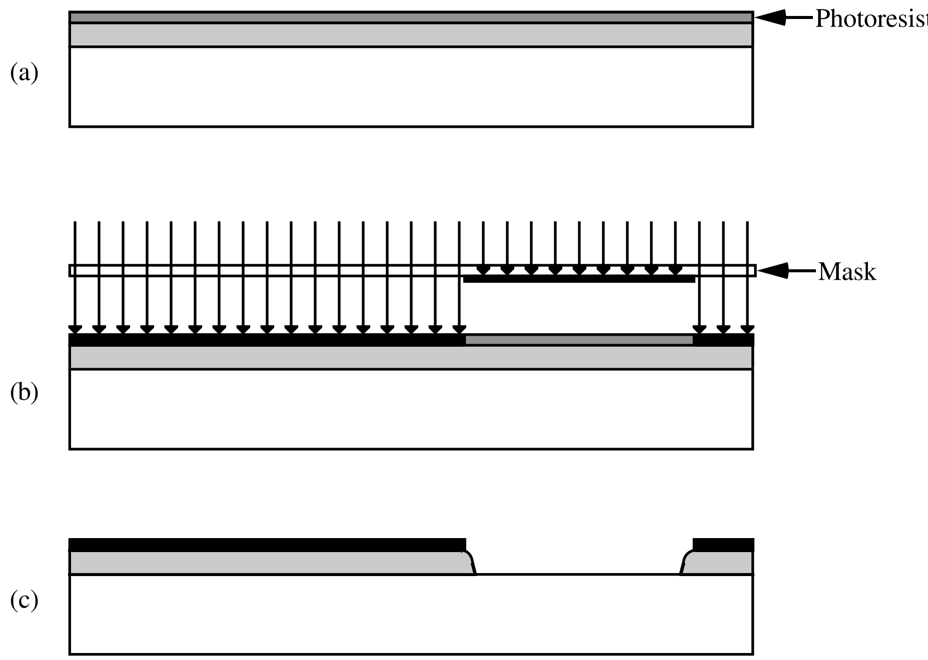


Figure 2. Steps in patterning a layer [Weste85]. (a) Photoresist covers the layer to be patterned. (b) A light source is shown through a mask, selectively polymerizing the photoresist. (c) The unpolymerized photoresist is removed, and the wafer is exposed to an acid, which etches only the areas not covered by photoresist.

Standard cells

Creating a complete full custom design takes a lot of time and effort, since all the details of the chip creation must be specified. However, in many cases the complete flexibility of the full custom design process are not needed for a given circuit. In these cases, it is advantageous to restrict some of the design flexibility to speed up the design process. This is exactly the approach taken in standard cell design. A standard cell design restricts the allowed circuit geometries and dictates a specific layout topology. By doing so, software tools can be designed to efficiently automate the physical design process, greatly speeding up the design process. However, since the layout is more restrictive, this approach delivers lower capacity and performance than a full custom design.

In a standard cell design, all gates are predefined in a library. This library consists of a set of standard cells that implement all the different gate structures necessary to map a circuit, as well as possibly some larger, more complex gate structures included for better mapping efficiency. All of these cells have the same height, the same placement of power and ground lines (which run horizontally across the cell), and have input and output terminals at the top and/or bottom. In this way cells can be abutted horizontally, creating a row of cells with uniform height, and with power and ground lines connecting together. Cells can be of arbitrary widths, so that cells with widely differing functionality (from a simple inverter to a multiple gate cell) can all be created efficiently. A standard cell chip is laid out in interleaved routing and logic rows (Figure 3). The logic rows are all of uniform height, while the routing rows are as high as necessary to accommodate the required routing. While horizontal routing is handled in these routing rows (or "channels"), vertical routing is either accomplished in higher metal layers, or by the inclusion of route-through cells which connect signals between adjacent routing channels.

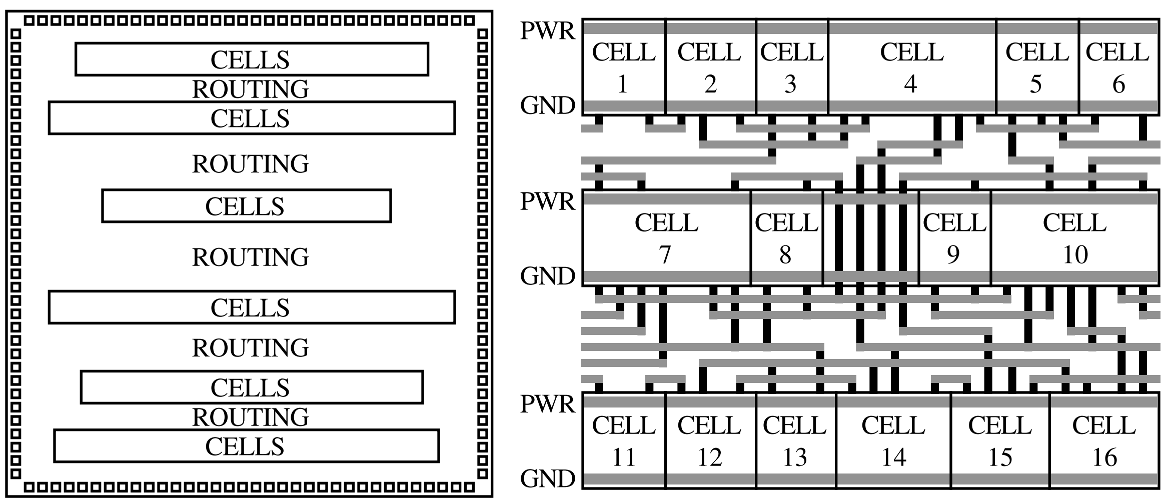


Figure 3. A standard cell chip layout (left), and a detail of a portion of the array (right), including a route-through between **CELL 8** and **CELL 9**.

With the layout of the gates handled by a predefined library, the process of creating a standard cell layout involves translating the circuit to be implemented into a collection of standard cell functions (technology mapping), placing these cells to achieve the required performance with the minimum routing channel widths (placement), and connecting the terminals of the standard cells together in the proper pattern (routing). Because of the regularity of the standard cell layout, each of these problems can be handled automatically via software. Thus, a logic designer need only specify the desired circuit functionality, and the software will automatically create a layout. This layout will then be fabricated at a silicon foundry exactly the same way as a full custom design.

Since standard cells and full custom designs require exactly the same fabrication processes, the fabrication time and costs are similar. The big differences between the two styles are the time it takes to ready a design for fabrication, and the resulting size and performance of the design. In a full custom design, the designer can optimize all aspects of the layout, and achieve the highest performance and capacity circuit. However, it takes a significant amount of time to create such a design. A standard cell design can be created much more quickly, but the capacity and performance will suffer. As a result, full custom design is used primarily where the capacity and performance penalties of standard cells cannot be tolerated, while other circuits will use standard cells.

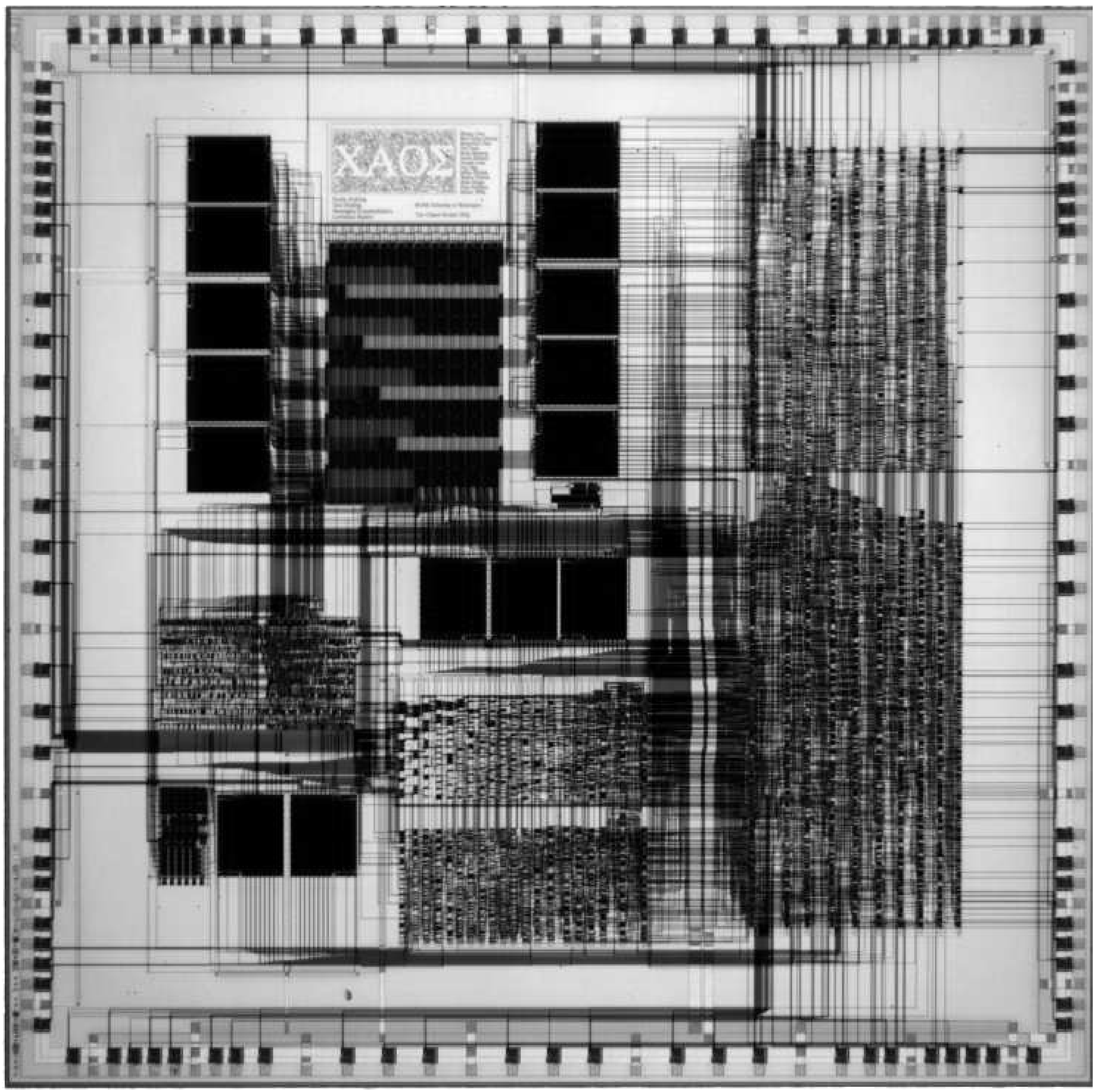


Figure 4. The Chaos router chip [Bolding93], which includes full-custom (the dark black regions) and standard cell (the regions of scattered black lines) elements.

The decision of what style to use can occur not only at the chip level, with an entire design created either as a full custom or standard cell layout, but also at the logic level, with the two styles combined within a single chip. For example, a multiprocessor routing chip may require the highest performance possible in the datapath, while the control circuitry can be designed somewhat less optimally. Thus, the design will include a full-custom datapath, while the control logic will be generated with standard cells (Figure 4). This tradeoff works especially well in this case, since the datapath will have a large amount of regularity, and will thus be somewhat simpler to design by hand, while the potentially complex and irregular control circuitry can be automatically created. This gives much of the benefits of both full custom and standard cells in a single design.

Another possibility in a standard cell is to include megacells or cores. These are complex, predesigned circuit elements that can be inserted in a design. For example, there are microcontrollers, DSPs, multipliers and other mathematical functions, standard bus interfaces, and many other functions available as megacells or cores [Arnold94]. These elements can be included in a standard cell design, providing complex, compact, high performance functionality without the need to custom design these parts. They will be significantly larger than standard cells, so some provision for including these elements is required. However, current software can in general automatically include these elements in a standard cell design.

Mask-programmable gate arrays

While standard cells simplify the design process over full custom designs, they do nothing about the time and cost involved in the actual fabrication of the chips. Although standard cell designs restrict the topologies and logic structures available to the designer, a standard cell design is still fabricated from scratch. Mask-programmable gate arrays (MPGAs) take the logic standardization of standard cells a step further. They speed up the fabrication process and lower costs, while retaining the ability of standard cells to harness automatic mapping software.

In an MPGA [Hollis87] much of the fabrication of the chip is carried out ahead of time. An MPGA has a standardized logic structure, with the layout of all transistors in the system fixed and predefined. That is, a given type of MPGA has a specific logic structure dictated to its users. For example, the MPGA might consist of simply a sea of individual transistors covering the chip. Others may have sets of four, six, or more transistors partially interconnected in a pattern optimized for mapping certain types of gates. To customize the MPGA in order to implement a given functionality, the designers are able to interconnect these logic elements in any manner they desire. Thus, one design can connect four transistors to power, ground, and each other in the proper manner to implement a 2-input NAND gate, while another design uses these transistors as a portion of a complex latch, or whatever else is desired.

Because the transistor layouts are fixed for a given MPGA, the foundry can stockpile partially fabricated chips, with all but the metal layers already created. A designer then specifies the interconnection pattern for the routing layers, and the fabrication of the chips is completed accordingly. In this way, most of the processing steps are already completed ahead of time, with the masks shared across all the designs that use a given MPGA. This decreases both processing time, since much of the processing has already been completed, as well as amortizing the cost, since much of the processing is shared across numerous designs. The designers still have a great deal of design flexibility, since they can specify an arbitrary interconnection pattern for the logic elements in the system, as long as there is enough routing area to accommodate the routing demands.

There are some limitations to MPGAs. Since the layout of the logic in the MPGA is predetermined, it will not be optimal for a given design. Some designs may require more or less of a given resource than the MPGA is designed for, leading to inefficiencies. Also, since the locations are predetermined, the routing may be more complex than in either a standard cell or full custom design. This is especially true in MPGAs that predefine the routing channels on the chip, since these routing channels will inevitably be either too large or too small for a given design. Thus, either routing resources are wasted because the channels are too large, or logic resources are sacrificed since the routing necessary to use them is not available. Some MPGAs are designed as Sea-of-Gates. In a Sea-of-Gates design, the entire chip area has logic fabricated on it. To handle routing, some of these logic functions are ignored, and arbitrary routing is performed over the top of the logic functions. In this way, the restrictions on routing channel size are removed, yielding greater flexibility in the design. However, even with a Sea-of-Gates design, there are still numerous inefficiencies due to fixed logic placement and resource mixes, yielding lower performance and logic capacities than either standard cell or full custom designs.

Programmable logic devices

Mask-programmable gate arrays reduce some of the fabrication time by prefabricating the logic on the chips. However, customizing an MPGA still requires processing at a silicon foundry, with the associated time lag and mask costs. Programmable logic devices (PLDs) in general avoid these costs by prefabricating the entire chip ahead of time. In order to customize the design, the PLD has configuration points scattered throughout the system. These configuration points are controlled by static memory cells, antifuses, or other devices that can be programmed after the chip has been fabricated. Thus, to customize the chip for a given design, the user programs these elements, which causes the chip to implement the desired functionality. The amount and locations of these configuration points will be optimized for a given PLD. Large amounts of configurability yield a much more flexible part, while less configurability yield higher capacity and performance (since the configuration points and their programming elements consume chip area and add delay to the signals connected to them).

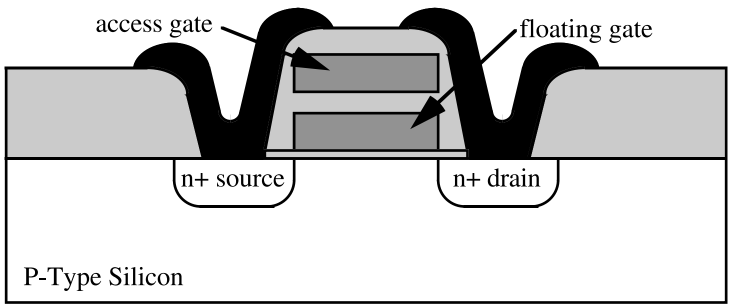


Figure 5. Floating gate structure for EPROM/EEPROM. The floating gate is completely isolated. An unprogrammed transistor, with no charge on the floating gate, operates the same as a normal n-transistor, with the access gate as the transistor's gate. To program the transistor, a high voltage on the access gate plus a lower voltage on the drain accelerates electrons from the source fast enough to travel across the gate oxide insulator to the floating gate. This negative charge then prevents the access gate from closing the source-drain connection during normal operation. To erase, EPROM uses UV light to accelerate electrons off the floating gate, while EEPROM removes electrons by a technique similar to programming, but with the opposite polarity on the access gate ([Altera93] pp. 467-471, [Wakerly94] pp. 399-400).

A simple example of this type of device is a memory. A memory is a device with N address signals, M outputs, and $M*2^N$ configuration points. The configuration points can take on the value 0 or 1. The address lines select one of 2^N sets of M configuration bits, and the outputs are set to the value of the selected M bits. Thus, the memory can be used to implement any M functions of N inputs, since each combination of these N inputs can produce a unique output value on each of the M outputs. There are a significant number of different configuration technologies ([Brown92a] pp. 2-3). Mask-programmable Read-Only Memories (ROM) configure their logic like MPGAs, with the configuration fixed during manufacture at a silicon foundry. Field-programmable versions also exist, with the programming accomplished without custom fabrication. Thus, users can easily customize field-programmable chips at their workbenches. Some Programmable Read-Only Memories (PROMs) contain fuses that can be blown by selectively applying a high voltage, permanently configuring the PROM. Erasable PROMs (EPROM) allow the configuration to be changed by exposure to a UV light source, while Electrically Erasable PROMs (EEPROM) allow the chip to be reprogrammed via a completely electrical process (Figure 5). Finally, Random-Access Memories (RAM) allow the configuration to be changed on the fly during circuit operation, allowing read-write access to the chip, while ROMs are read-only. Thus, configuration technology enables a large variety of methods for setting the memory, from custom fabrication to one-time-programmable fuses to the complete read-write flexibility of a RAM. Other PLDs (as well as FPGAs, discussed later) will offer most or all of these same programming technologies to configure their logic.

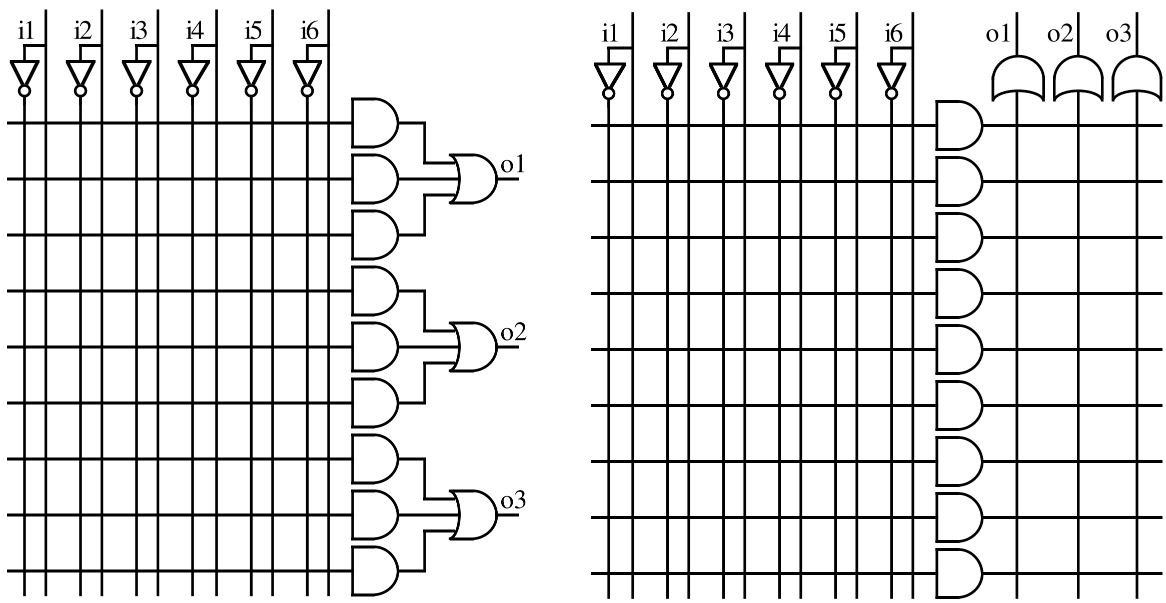


Figure 6. PAL (left) and PLA (right) structures [Biehl93].

While memories are able to implement logic functions, they are in general too inefficient for most applications. The problem is that most logic functions do not need the complete flexibility of the full memory structure, and thus using a memory to implement these functions is quite inefficient (the memory must double in size with each additional input). A better alternative is the Programmable Array Logic (PAL), a device optimized for implementing two-level sum-of-products logic structures (Figure 6 left). The device consists of a set of arbitrary product terms (the AND gates) leading to OR gates that produce the chip's outputs. As shown in Figure 7 , these product terms are logic structures that can be programmed to implement an AND of any or all of its inputs. That is, the output is the NOR of all inputs whose programmable connection is enabled, while the function is insensitive to inputs whose programmable connections are disabled. By inverting the signals coming into this function, the NOR function becomes an AND. Thus, the product term can become the AND of A and $\overline{B}$ by making connections **P2** and **P3**, and disabling all other connections, while connections **P2**, **P4**, and **P6** would make the AND of A, B, and C. Since these product terms feed into an OR gate, multiple product terms can be ORed together. By programming two product terms leading to the same OR gate with the functions just discussed, the output would be $(A\overline{B}) + (ABC) = A(\overline{B} + C)$. While the implementation just discussed is less efficient than a similar function in a small memory, the efficiency of the memory implementation degrades significantly as the number of inputs increase. However, a PAL is usually not able to implement all possible functions because there are usually not enough product terms leading to the OR gates. For example, to compute an XOR function of 3 inputs requires 4 product terms (ABC, $\overline{A}\overline{B}C$, $\overline{A}B\overline{C}$, $A\overline{B}\overline{C}$), while the PAL structure shown contains only 3 product terms per output. Actual PALs can have many more inputs, outputs, and product terms per output, as well as latches on the outputs with feedback paths to the product term array (i.e., the output of the latch can be included in the product terms in the same way that inputs are handled). Generic Array Logics (GALs) are similar to PALs, except that they are flexible enough to be configured to implement several different types of PALs. They also can include optional inversions on the output. This is important, since it is often much easier to implement the complement of a function in a sum-of-products form and then invert it than it is to implement the uncomplemented sum-of-products form directly. For example, the function $\overline{(ABCD)}$ requires fifteen product terms to implement in sum-of-products form, while the complement requires only one product term.

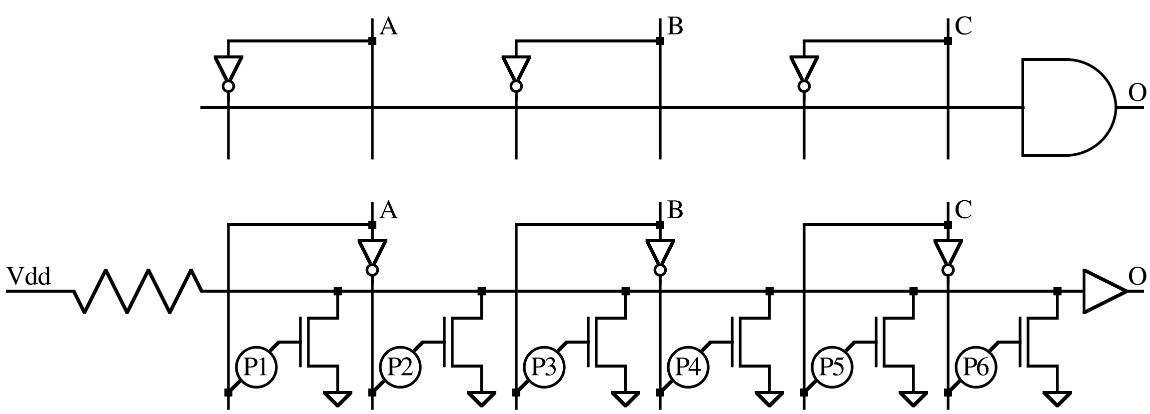


Figure 7. One portion of the PAL structure (top), and the corresponding implementation (bottom) [Wakerly94]. The circles labeled **P** are programmable connections. Note that the inverters are swapped between the two versions. In both circuits, the output value **O** is the AND of any or all vertical lines, depending on which programmable connections are enabled.

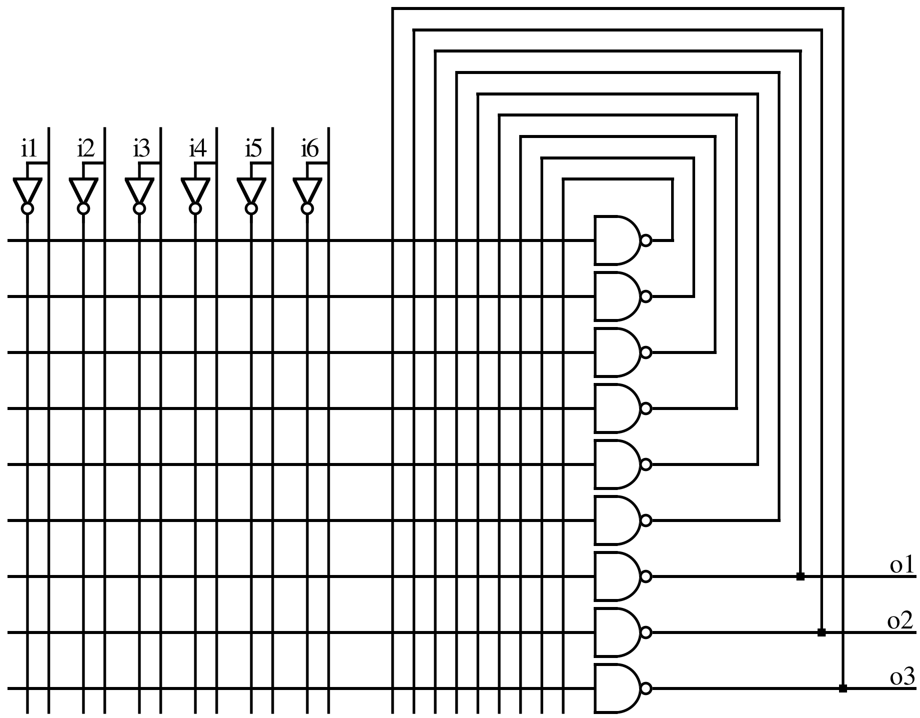


Figure 8. The PML structure [Biehl93].

One limitation of a PAL is that product terms cannot be shared between outputs. Thus, if two outputs both require the product term $\overline{A}B\overline{C}$, they would each need to generate the function with their own product terms. A different form of PLD, called a Programmable Logic Array (PLA), allows product terms to be shared between output functions (Figure 6 right). In a PLA, the AND array of product terms (the AND plane) leads to a similar OR array (the OR plane). These OR functions are implemented the same way the AND functions are, except that instead of inverting the inputs, the outputs are inverted (since the base structure computes a NOR, instead of inverting the inputs, which produces an AND, inverting the output produces an OR). Thus, the OR gates compute an OR of any or all of the input product terms. Thus, the XNOR and majority functions of three inputs can be implemented with five product terms, where a PAL structure would require eight (the product terms are $A1 = (\overline{A}BC)$, $A2 = (A\overline{B}C)$, $A3 = (AB\overline{C})$, $A4 = (\overline{A}\overline{B}\overline{C})$, $A5 = (ABC)$, and the output functions are $XNOR = (A1 + A2 + A3 + A4)$, $MAJ = (A2 + A3 + A4 + A5)$). While PLAs have more flexibility than PALs since the connections between the AND and OR gates are programmable, this flexibility results in lower performance. Primarily the performance degradation is due to the fact that in a PLA a signal must travel through two programmable connections (one in the AND plane, one in the OR plane), while in a PAL the signal goes through only one programmable connection. These connections can add significant delay to the system. Historically, PLAs were introduced before PALs, but PALs are today's most commonly used PLD [Wakerly94].

An even more general structure than the PLA is Programmable Macro Logic (PML). As shown in Figure 8, instead of separate AND and OR structures in the PLA, the PML has a single array of NAND gates, with the outputs feeding back into the array. Thus, the output of a NAND gate can be used as an input to another NAND gate in the same way as a chip input. As we know from basic logic design, a two-level NAND-NAND structure is identical to a two-level AND-OR structure, which means that the PML can implement the same structures as a PAL or PLA. However, by combining the AND and OR functionality into a single array, the user can trade off the number of AND and OR terms on a per-mapping basis, as well as choose to implement some outputs in 1, 2, 3, or more levels of logic, simply by using the proper feedback paths. However, just as in a PLA, each level must pass through a programmable connection, so multi-level paths will be significantly slower than one or two-level paths. A similar product, the ERASIC [Jenkins94], uses a single array of NOR gates instead of the PML's NAND gate structure.

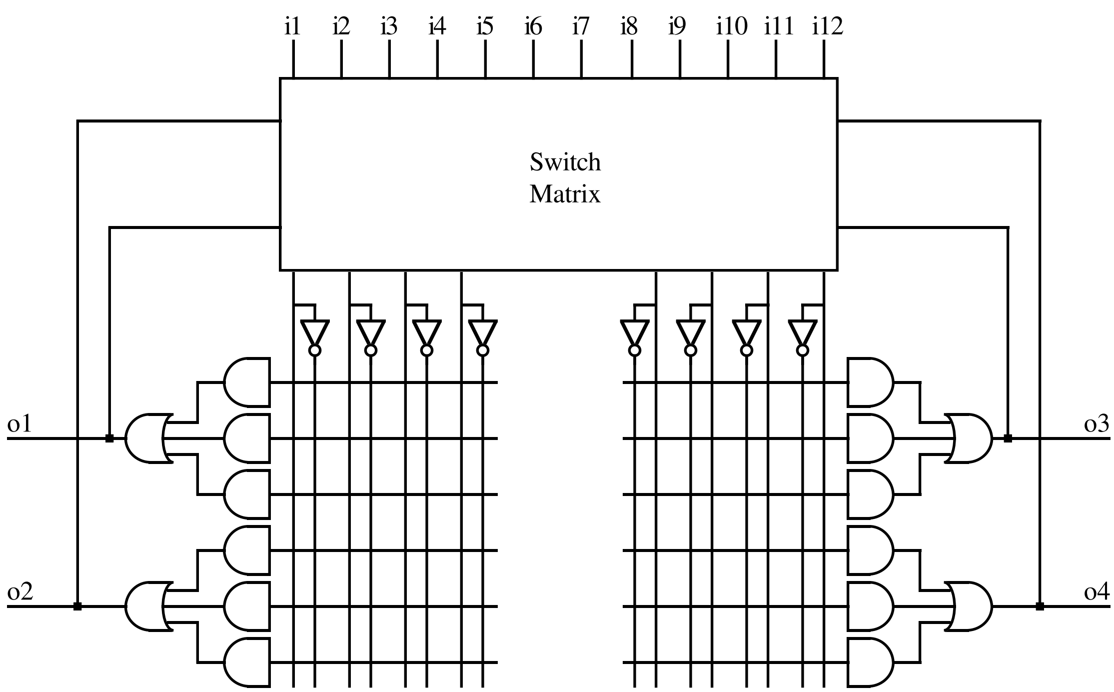


Figure 9. CPLD structure. The switch matrix provides configurable connectivity between the inputs, logic arrays, and any feedbacks from outputs [Biehl93].

While the previous structures are fine for small PLDs, they may not scale to larger arrays. Primarily, as the PLD gets larger, the number of connections to a given product term grows larger, slowing down the signal propagation. To deal with this, Complex Programmable Logic Devices (CPLDs) break the system up into several smaller product term arrays, connected together with a switch matrix (Figure 9). Thus, the CPLD can be viewed as a collection of smaller PLDs and an interconnection network. In this way, the CPLD can have much larger capacity than a single PLD, while keeping propagation delays low.

Because of the regularity and simplicity of PLDs, it is fairly easy to map to a PLD, a process that has been automated by software. These tools can take logic equations and automatically create the configuration files for the PLD. CPLDs may complicate the process by having switch matrices with restricted connectivity, as well as the requirement of partitioning the logic equations into the separate logic arrays, but the process is still much simpler than for many other technologies.

Field-programmable gate arrays

Programmable logic devices concentrate primarily on two-level, sum-of-products implementations of logic functions. They have simple routing structures with predictable delays. Since they are completely prefabricated, they are ready to use in seconds, avoiding long delays for chip fabrication. Field-Programmable Gate Arrays (FPGAs) are also completely prefabricated, but instead of two-level logic they are optimized for multi-level circuits. This allows them to handle much more complex circuits on a single chip, but it sacrifices the predictable delays of PLDs. Note that FPGAs are often considered another form of PLD, often under the heading Complex Programmable Logic Device (CPLD). In this thesis, we will use PLD to exclusively refer to the product-term oriented devices discussed in the previous section.

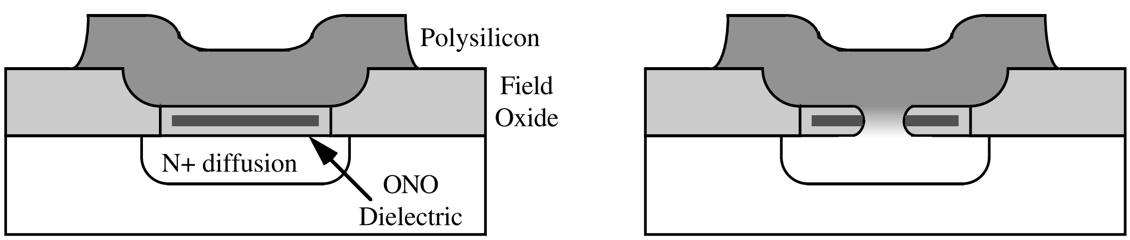


Figure 10. Actel's Programmable Low Impedance Circuit Element (PLICE). As shown at left, an unblown antifuse has an oxide-nitride-oxide (ONO) dielectric preventing current from flowing between diffusion and polysilicon. The antifuse can be blown by applying a 16 Volt pulse across the dielectric. This melts the dielectric, allowing a conducting channel to be formed (right). Current is then free to flow between the diffusion and the polysilicon [Actel94, Greene93].

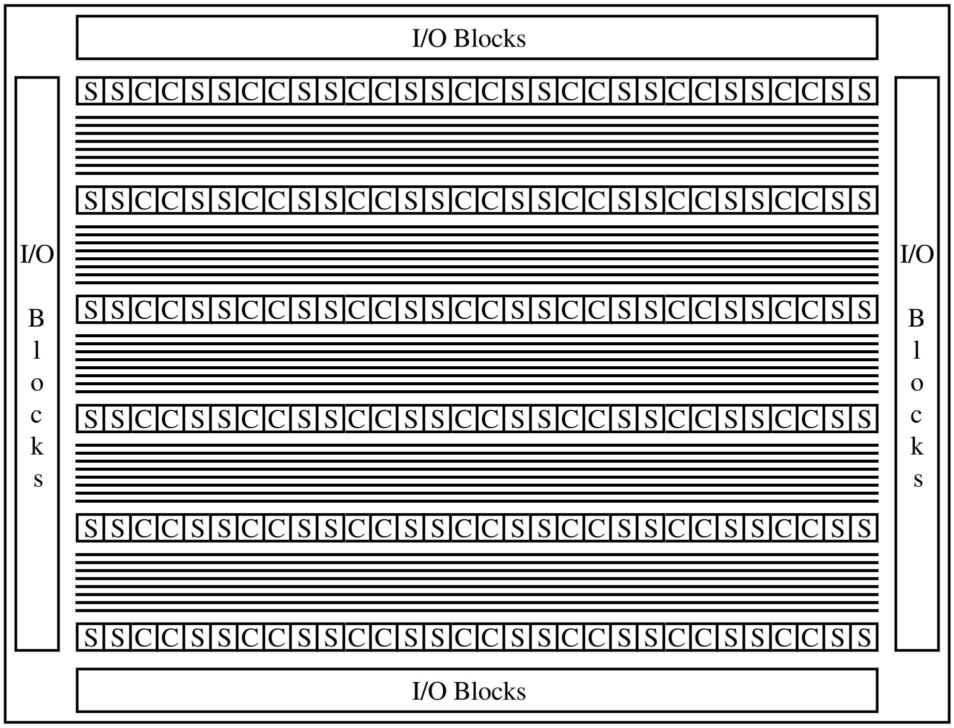


Figure 11. The Actel FPGA structure [Brown92a, Actel94]. Logic blocks (C-Modules and S-Modules) are surrounded by horizontal routing channels like an MPGA.

Just as in PLDs, FPGAs are completely prefabricated, and contain special features for customization. These configuration points are normally either SRAM cells or antifuses. Antifuses are one-time programmable devices (Figure 10), which when "blown" create a connection, while when "unblown" no current can flow between their terminals (thus, it is an "anti"-fuse, since its behavior is opposite to a standard fuse). Because the configuration of an antifuse is permanent, antifuse-based FPGAs are one-time programmable, while SRAM-based FPGAs are reprogrammable, even in the target system. Since SRAMs are volatile, an SRAM-based FPGA must be reprogrammed every time the system is powered up, usually from a ROM included in the circuit to hold configuration files. Note that FPGAs often have on-chip control circuitry to automatically load this configuration data. SRAM cells are larger than antifuses, meaning that SRAM-based FPGAs will have less configuration points than an antifuse based FPGA. However, SRAM-based FPGAs have numerous advantages. Since they are reprogrammable, their configurations can be changed for bug fixes or upgrades. Thus they provide an ideal prototyping medium. Also, these devices can be used in situations where they can expect to have numerous different configurations, such as multi-mode systems and reconfigurable computing machines. More details on such applications are included in Chapter 3. There are many different types of FPGAs, with many different structures. Instead of discussing all of them here, which would be quite involved, this chapter will present a few typical and well-known FPGAs. Details on many others can be found elsewhere [Brown92a, Rose93, Chan94, Jenkins94, Trimberger94, Oldfield95].

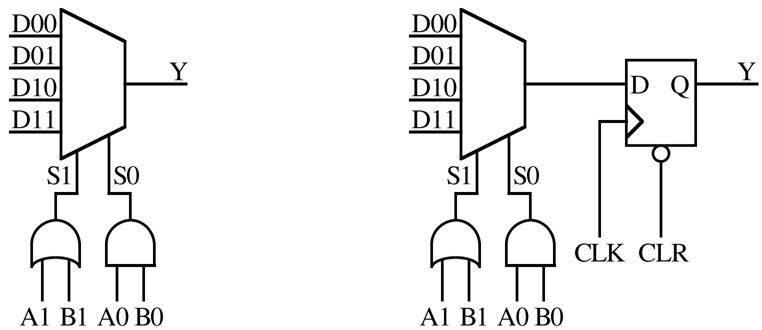


Figure 12. The Actel ACT 3 C-Module (left) and S-Module (right) [Actel94].

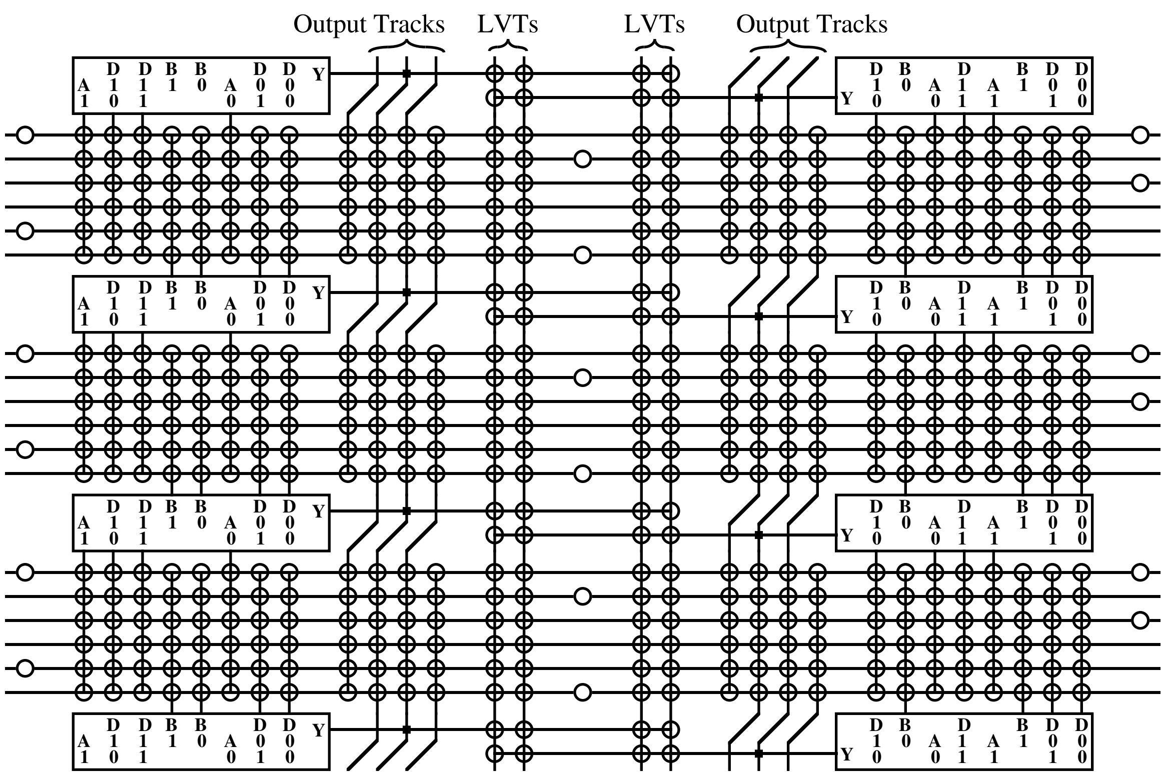


Figure 13. Actel routing structure [Actel94]. Circles are antifuses. Signal lines passing straight through an antifuse are always connected, while signals meeting at right angles in an antifuse or only touching the outside are only connected when the antifuse is blown.

One of the best known antifuse-based FPGAs is the Actel ACT series (we will discuss the third generation FPGA, the ACT 3, in this section). In many ways, the Actel FPGA is essentially a small MPGA that is ready to use instantly. The routing structure of the FPGA is very similar to an MPGA, with rows of logic cells between horizontal routing channels (Figure 11). The cells come in two types: C-Modules, for purely combinational functions, and S-Modules, for sequential logic (Figure 12). Both cells have a logic function based on a 4:1 multiplexer, though the control inputs are fed by either an AND or an OR gate. By properly assigning signals to inputs, this logic block can implement any 2-input function, and many other functions of up to eight inputs. To implement an arbitrary 2-input function, the two inputs are routed to A0 and A1, with B1=0 & B0=1, and D00-D11 are assigned either 0 or 1 to implement the desired function. To implement a function such as a 4-input AND (I1*I2*I3*I4), A1=I1, B1=0, A0=I2, A1=I3, D11=I4, D00=0, D01=0, D10=0. Many other functions are possible. In the S-Module, the logic block feeds into a flip-flop for implementing stateholding elements.

Unlike an MPGA, an FPGA is completely prefabricated, so configuration hardware must be imbedded in the system. In the Actel FPGA, this is accomplished with antifuses scattered throughout the routing topology (Figure 13). Horizontal routing channels run between the rows, and are broken into multiple segments. At the ends of these segments are antifuses, allowing the segments to either be used separately for two different signals (in which case the antifuse is left unblown), or combined together into a single longer route. Inputs to the logic cells come from signals in the channel directly above or below the cell (each input can take signals from only one of these tracks, with half coming from the upper and half from the lower). Which signal is chosen depends on which antifuse is blown. Antifuses are provided for connecting the input to any of the signals in the track, including dedicated power and ground lines for tying inputs to a constant value. The output of a cell is connected to a vertical wire (an output track) traversing the two channels above and two channels below the cell. There are also vertical wires (LVTs) traversing most of the chip. Some of these are segmented, though segments can be combined together in a similar fashion to that used for horizontal tracks. Antifuses are provided for connecting an output to any of the LVTs, and for connecting either the LVTs or the output tracks to any signal in the horizontal channels. Overall, the architecture is optimized primarily for horizontal routing, with approximately twice as many wires running horizontally as vertically. Thus, routes will primarily use the dedicated output track to reach the nearest four channels, and then route horizontally to all the signal destinations. Signals that must move further than the dedicated output will reach will have to use the LVTs, but a good logic placement will avoid this as much as possible.

Antifuses have a significant impact on both the architecture and usage of these systems. Since antifuses are much smaller than other programming technologies, an antifuse-based FPGA has many more configuration sites than an SRAM based FPGA. Thus, the number of choices in the routing structure is much higher than in other FPGAs. On the other hand, since the chips are not reprogrammable, they cannot be changed later for bug corrections, upgrades, nor used in reconfigurable systems. This means that an antifuse-based FPGA is essentially a slow, small MPGA which does not require custom fabrication. They are slower and less dense since in an MPGA the user has more control over the fabrication of the metal layers of the chip, while an antifuse-based FPGA must make the routing general enough to handle a wide range of applications. However, since they do not require custom fabrication in order to configure the chip, they are less expensive (at least at low volumes) than MPGAs, and are ready to use instantaneously.

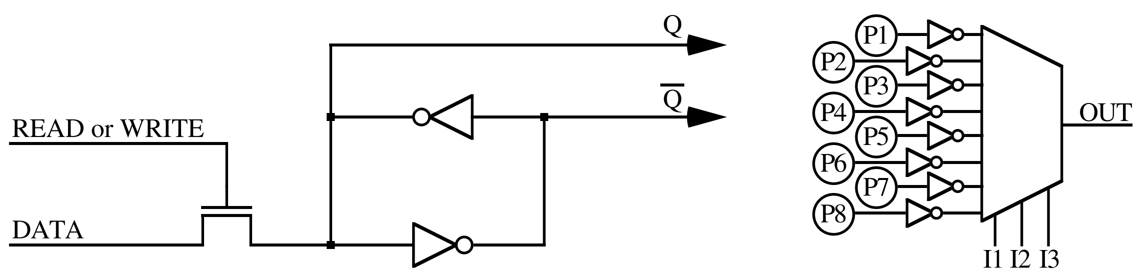


Figure 14. Programming bit for SRAM-based FPGAs (left) [Xilinx94], and a 3-input LUT (right).

In SRAM-based FPGAs memory cells are scattered throughout the FPGA instead of antifuses. As shown in Figure 14 left, a pair of cross-coupled inverters will sustain whatever value is programmed onto them. A single n-transistor gate is provided for either writing a value (the ratio of sizes between the transistor and the upper inverter is set to allow values sent through the n-transistor to overpower the inverter), or reading a value back out. The readback feature is used during debugging to determine the current state of the system. The actual control of the FPGA is handled by the Q and $\overline{Q}$ outputs. One simple application of an SRAM bit is to have the Q terminal connected to the gate of an n-transistor. If a 1 is assigned to the programming bit, the transistor is closed, and values can pass between the source and drain. If a 0 is assigned, the transistor is opened, and values cannot pass. Thus, this construct operates similarly to an antifuse, though it requires much more area. These programming bits can also be fed into decoders, allowing a few bits to control multiple features where only one feature can be active at a time. One of the most useful SRAM-based structures is the lookup table (LUT). By connecting 2^N programming bits to a multiplexer (Figure 14 right), any N-input function can be implemented. Although it can require a large amount of programming bits for large N, LUTs of up to 5 inputs can provide a flexible, powerful function implementation medium. Note that inverters may be necessary between the programming bits and the multiplexer to avoid the programming bit's value from being overwritten during operation.

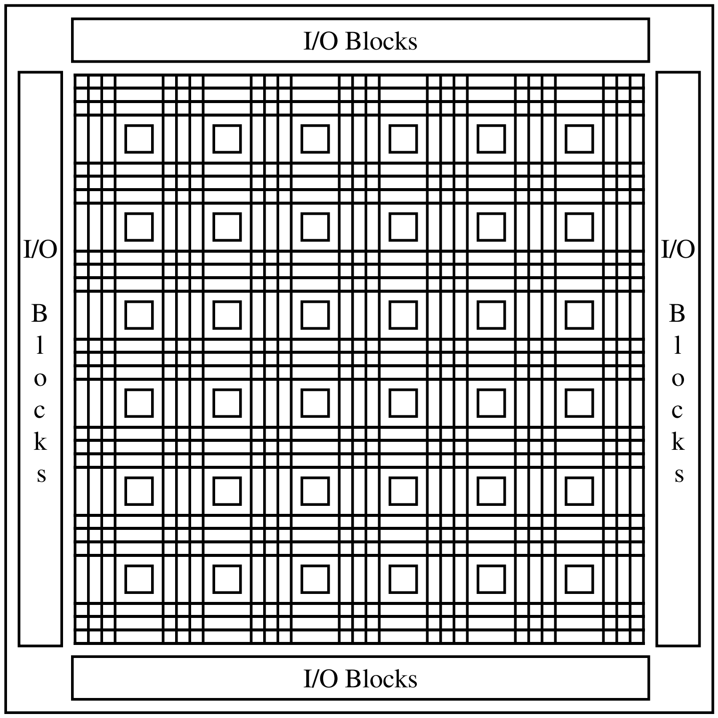


Figure 15. The Xilinx 4000 series FPGA structure [Xilinx94]. Logic blocks are surrounded by horizontal and vertical routing channels.

One of the best known of all FPGAs is the Xilinx Logic Cell Arrays (LCAs) [Trimberger93, Xilinx94]. In this section we will describe their third generation FPGA, the Xilinx 4000 series. As opposed to Actel's channel architecture, the Xilinx array is an "Island-style" FPGA [Trimberger94]. As shown in Figure 15, the logic cells are embedded in a general routing structure that permits arbitrary point-to-point communication. There is no horizontal bias in the routing (as is found in the Actel FPGAs), and the only requirement for good routing is that the source and destinations be relatively close together. Details of the routing structure are shown in Figure 16. Each of the inputs of the cell (**F1**-**F4**, **G1**-**G4**, **C1**-**C4**, **K**) comes from one of a set of tracks adjacent to the cells. The outputs are similar (**X**, **XQ**, **Y**, **YQ**), except they have the choice of both horizontal and vertical tracks. The routing structure is made up of three lengths of lines. Single-length lines travel the height of a single cell, where they then enter a switch matrix (Figure 17 right). The switch matrix allows this signal to travel out vertically and/or horizontally from the switch matrix. Thus, multiple single-length lines can be cascaded together to travel longer distances. Double-length lines are similar, except that they travel the height of two cells before entering a switch matrix (notice that only half the double-length lines enter the switch matrix, and there is a twist in the middle of the line). Thus, double-length lines are useful for longer-distance routing, traversing two cell heights without the extra delay and the wasted configuration sites of an intermediate switch matrix. Finally, longlines are lines that go half the chip height, and do not enter the switch matrix. In this way, very long-distance routes can be accommodated efficiently. With this rich sea of routing resources, the Xilinx 4000 series is able to handle fairly arbitrary routing demands, though mappings emphasizing local communication will still be handled more efficiently.

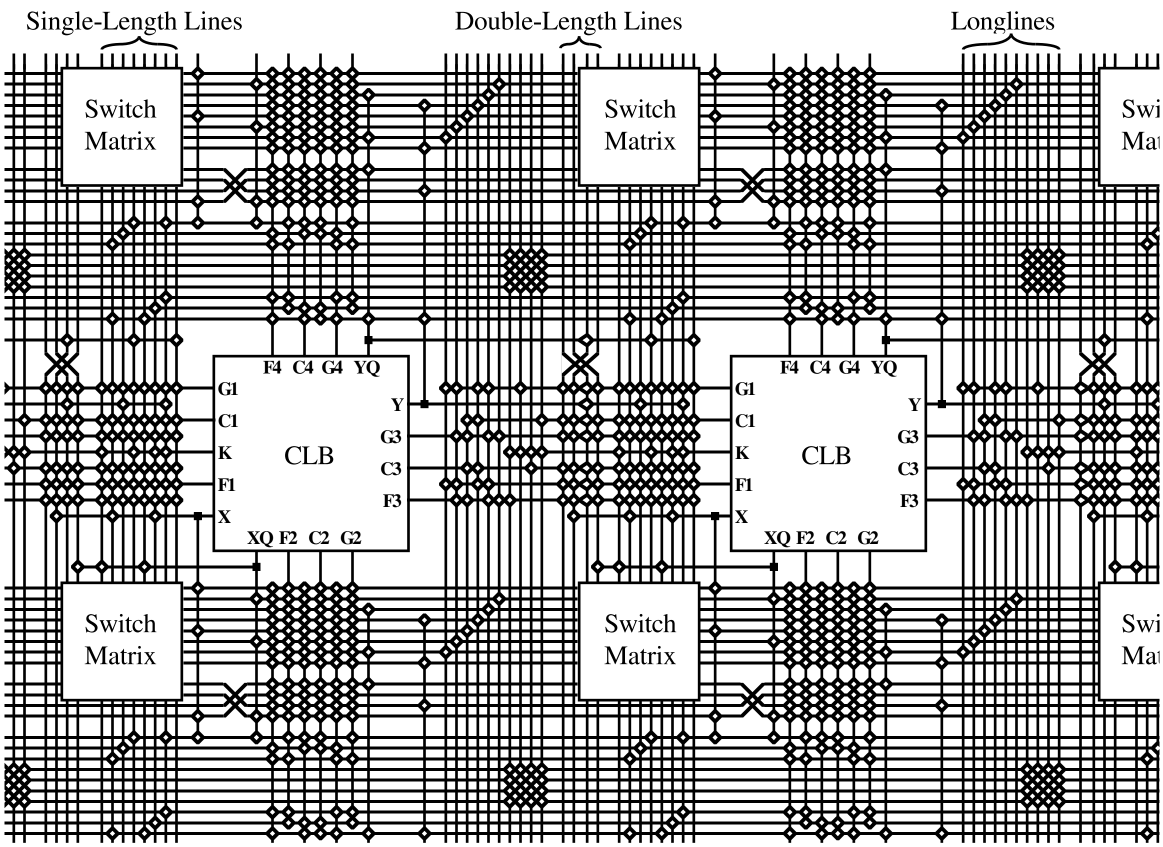
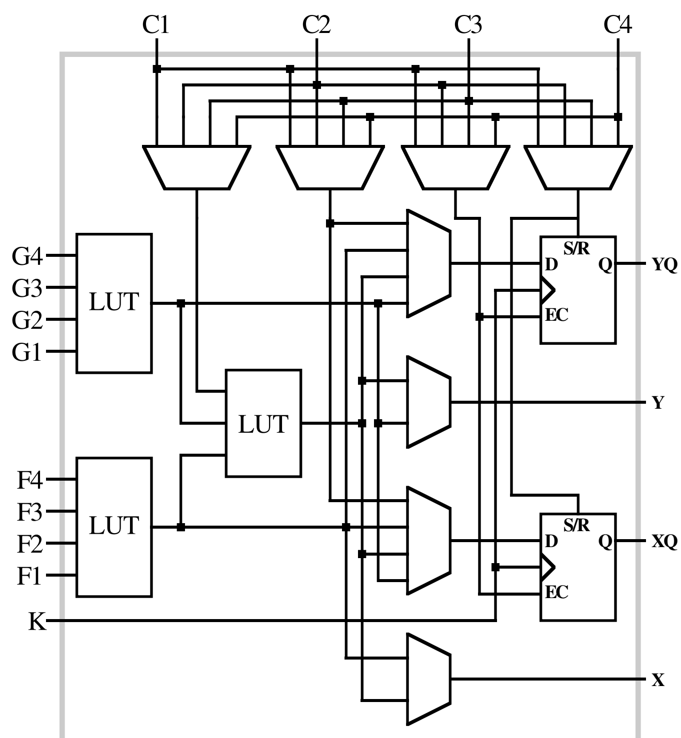


Figure 16. Details of the Xilinx 4000 series routing structure [Xilinx94]. The CLBs are surrounded by vertical and horizontal routing channels containing Single-Length Lines, Double-Length Lines, and Longlines. Empty diamonds represent programmable connections between perpendicular signal lines (signal lines touching on opposite sides of the diamonds are always connected).

As shown in Figure 17 left, the Xilinx 4000 series logic cell is made up of three lookup-tables (LUTs), two programmable flip-flops, and multiple programmable multiplexers. The LUTs allow arbitrary combinational functions of its inputs to be created. Thus, the structure shown can perform any function of five inputs (using all three LUTs, with the **F** & **G** inputs identical), any two functions of four inputs (the two 4-input LUTs used independently), or some functions of up to nine inputs (using all three LUTs, with the **F** & **G** inputs different). SRAM controlled multiplexers then can route these signals out the **X** and **Y** outputs, as well as to the two flip-flops. The inputs at top (**C1**-**C4**) provide the third input to the 3-input LUT, enable and set or reset signals to the flip-flops, and a direct connection to the flip-flop inputs. This structure yields a very powerful method of implementing arbitrary, complex digital logic. Note that there are several additional features of the Xilinx FPGA not shown in these figures, including support for embedded memories and carry chains.



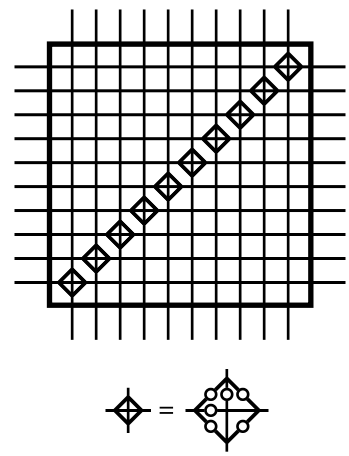


Figure 17. Details of the Xilinx CLB (left) and switchbox (top right) [Xilinx94]. The multiplexers, LUTs, and latches in the CLB are configured by SRAM bits. Diamonds in the switchbox represent six individual connections (bottom right), allowing any permutation of connections between the four signals incident to the diamond.

While many SRAM-based FPGAs are designed like the Xilinx architecture, with a routing structure optimized for arbitrary, long-distance communications, several other FPGAs concentrate instead on local communication. The “Cellular” style FPGAs [Trimberger94] feature fast, local communication resources, at the expense of more global, long-distance signals. As shown in Figure 18, the CLi FPGA [Jenkins94] has an array of cells, with a small amount of routing resources running horizontally and vertically between the cells. There is one local communication bus on each side of the cell. It runs the height of eight cells, at which point it enters a repeater. Express buses are similar to local buses, except that there are no connections between the express buses and the cells. The repeaters allow access to the express buses. These repeaters can be programmed to connect together any of the two local buses and two express buses connected to it. Thus, a small amount of global communication can be accomplished on the local and express buses, with the local buses allowing shorter-distance communications and connections to the cells, while express buses allow longer-distance connections between local buses.

While the local and global buses allow some of the routing flexibility of the Xilinx FPGA's arbitrary routing structure, there are much fewer buses in the CLi architecture than are present in the Xilinx FPGA. The CLi FPGA instead features a significant amount of local communication resources. As shown in Figure 19, each cell receives two signals from each of its four neighbors. It then sends the same two outputs (**A** and **B**) to all of its neighbors. That is, the cell one to the north will send signals **AN** and **BN**, and the cell one to the south will send **AS** and **BS**, while both will receive the same signals **A** and **B**. The input signals become the inputs to the logic cell (Figure 20).

Instead of Xilinx's LUTs, which require a large amount of programming bits per cell, the CLi logic block is much simpler. It has multiplexers controlled by SRAM bits which select one each of the **A** and **B** outputs of the neighboring cells. These are then fed into AND and XOR gates within the cell, as well as a flip-flop. Although the possible functions are complex, notice that there is a path leading to the **B** output that produces the NAND of the selected **A** and **B** inputs, and sending it out the **B** output. This path is enabled by setting the two 2:1 multiplexers to their constant input, and setting **B**'s output multiplexer to the 3rd input from the top. Thus, the cell is functionally complete. Also, with the **XOR** path leading to output **A**, the cell can efficiently implement a half-adder. The cell can perform a pure routing function by connecting one of the **A** inputs to the **A** output, and any of the **B** inputs to the **B** output, or vice-versa. This routing function is created by setting the two 2:1 multiplexers to their constant inputs, and setting **A**'s and **B**'s output multiplexer to either of their top two inputs. There are also provisions for bringing in or sending out a signal on one or more of the neighboring local buses (**NS1**, **NS2**, **EW1**, **EW2**). Note that since there is only a single wire connecting the bus terminals, there can only be a single signal sent or received from the local buses. If more than one of the buses is connected to the cell, they will be coupled together. Thus, the cell can take a signal running horizontally on an **EW** local bus, and send it vertically on a **NS** local bus, without using up the cell's logic functionality. However, by bringing a signal in from the local buses, the cell can implement two 3-input functions.

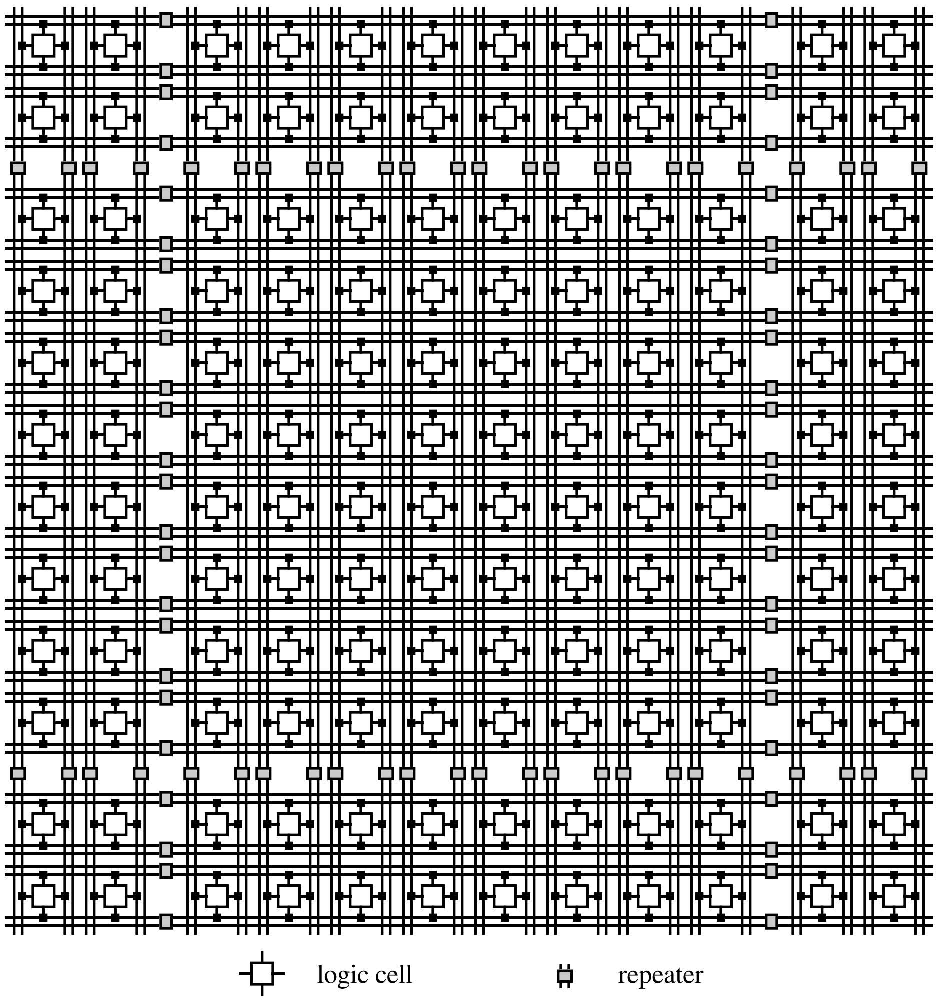


Figure 18. The CLi6000 routing architecture [Jenkins94]. One 8x8 tile, plus a single set of surrounding rows and columns, is shown. The full array has many of these tiles abutted horizontally and vertically.

The major differences between the Island style architecture of the Xilinx 4000 series and the Cellular style of the CLi FPGA is in their routing structure and cell granularity. The Xilinx 4000 series is optimized for complex, irregular random logic. They feature a powerful routing structure optimized for arbitrary global routing, and large logic cells capable of providing arbitrary 4-input and 5-input functions. This provides a very flexible architecture, though one that requires a large amount of programming bits per cell (and thus cells that take up a large amount of space on the chip). In contrast, the CLi architecture is optimized for highly local, pipelined circuits such as systolic arrays and bit-serial arithmetic. Thus, they emphasize local communication at the expense of global routing, and have simple cells. Because of the very simple logic cells there will be many more CLi cells on a chip than is found in the Xilinx FPGA, yielding a greater logic capacity for those circuits that match the FPGA's structure. Because of the restricted routing, the CLi FPGA is much harder to automatically map to than the Xilinx 4000 series, though the simplicity of the CLi architecture make it easier for a human designer to hand-map to the CLi's structure. Thus, in general cellular architectures tend to appeal to designers with appropriate circuit structures who are willing to spend the effort to hand-map their circuits to the FPGA, while the Xilinx 4000 series is more appropriate for handling random-logic tasks and automatically-mapped circuits.

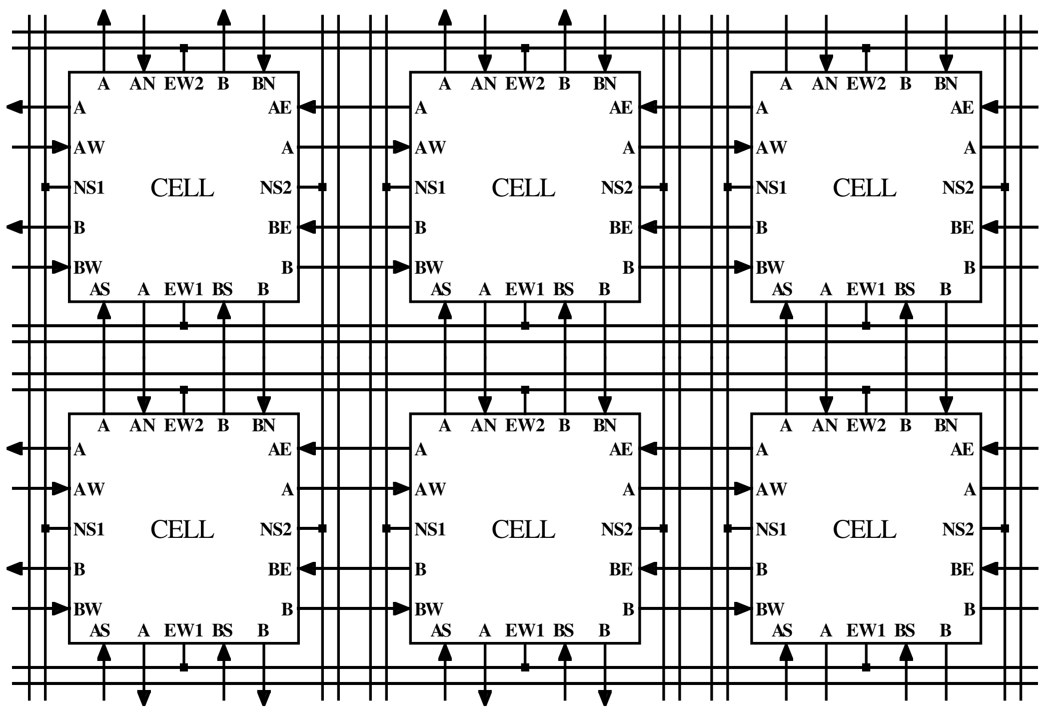


Figure 19. Details of the CLi routing architecture [Jenkins94].

Compared with technologies such as full-custom, standard cells, and MPGAs, FPGAs will in general be slower and less dense. In the case of SRAM-based FPGAs, this is due to the configuration points, which take up a significant amount of space, and add extra capacitance and resistance (and thus delay) to the signal lines. Thus, the programming bits add an unavoidable overhead to the circuit, which can be reduced by limiting the configurability of the FPGA, but never totally eliminated. Also, since the metal layers in an FPGA are prefabricated, while the other technologies custom fabricate the metal layers for a given circuit, the FPGA will have less optimized routing. This again results in slower and larger circuits. However, even given these downsides, FPGAs have the advantage that they are completely prefabricated. This means that they are ready to use instantly, while mask-programmed technologies can require weeks to be customized. Also, since there is no custom fabrication involved in an FPGA, the fabrication costs can be amortized over all the users of the architecture, removing the significant NRE's of other technologies. However, per-chip costs will in general be higher, making the technology better suited for low volume applications. Also, since SRAM-based FPGAs are reprogrammable, they are ideal for prototyping, since the chips are reusable after bug fixes or upgrades, where mask-programmed and antifuse versions would have to be discarded.

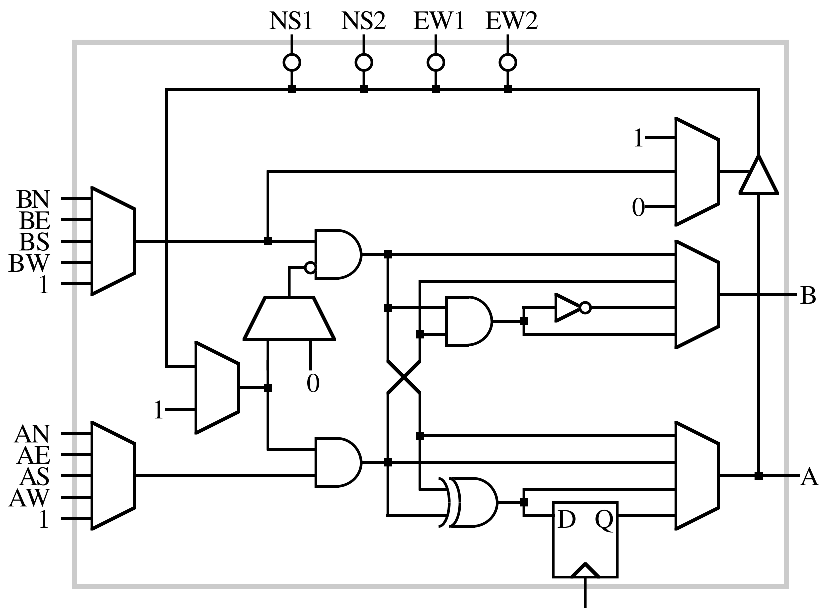


Figure 20. The CLi logic cell [Jenkins94].

Although, as we just discussed, FPGAs have an unavoidable overhead due to configurability, making them slower and larger than mask-programmed and custom technologies developed in the same processes, SRAM-based FPGAs may give the designer access to better fabrication technology than other approaches. Specifically, as a fabrication process is being readied for production use, the fabricator will produce some early designs to debug the technology. In general, the first chips off the line are test structures, and then SRAMs. Increasingly, an SRAM-based FPGA is the next chip to be fabricated on these lines. For the fabricator, SRAM-based FPGAs have the advantages that they can quickly be designed for a given technology, since they consist of a single tile replicated across the chip surface, yet are highly testable because of their programmability, and have many of the same features of full-custom designs. Thus, an FPGA can quickly be created in a technology, and used for fault location and diagnosis to isolate errors in the fabrication process. For the FPGA manufacturer (which often do not have their own fabrication lines, and thus use outside fabricators), this gives them the earliest access to a new process technology, increasing the performance and capacity of their products. Antifuse-based FPGAs do not have this same advantage. Creating the antifuses requires extra fabrication steps. This complicates the fabrication, relegating antifuse manufacture to older, lower quality fabrication processes. While antifuses are much denser than SRAM bits, which makes antifuse-based FPGAs denser than SRAM-based FPGAs created with the same process, an SRAM-based FPGA's access to new fabrication technologies may offset this cost.

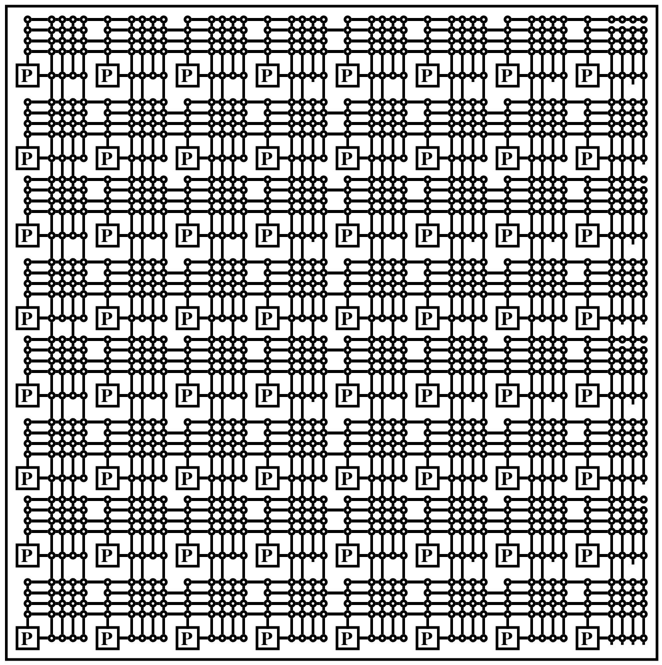


Figure 21. The Aptix FPIC architecture [Aptix93a]. The boxed P indicates an I/O pin.

A technology similar to SRAM-based FPGAs is Field-Programmable Interconnect Components (FPIC) [Aptix93a] and Devices (FPID) [I-Cube94] (we will use FPIC from now on to refer to both FPIC & FPID devices). Like an SRAM-based FPGA, an FPIC is a completely prefabricated device with an SRAM-configured routing structure (Figure 21). Unlike an FPGA, an FPIC has no logic capacity. Thus, the only use for an FPIC is as a device to arbitrarily interconnect its I/O pins. While this is not generally useful for production systems, since a fixed interconnection pattern can be achieved by the printed circuit board that holds the circuit, it can be quite useful in prototyping and reconfigurable computing (these applications are discussed in Chapter 3). In each of these cases, the connections between the chips in the system may need to be reconfigurable, or this connection pattern may change over time. In a reconfigurable computer, many different mappings will be loaded onto the system, and each of them may desire a different interconnection pattern. In prototyping, the connections between chips may need to be changed over time for bug fixes and functionality upgrades. In either case, by routing all of the I/O pins of the logic-bearing chips to FPICs, the interconnection pattern can easily be changed over time. Thus, fixed routing patterns can be avoided, hopefully increasing the performance and capacity of the prototyping or reconfigurable computing machine.

There is some question about the economic viability of FPICs. The problem is that they must provide some advantage over an FPGA with the same I/O capacity, since in general an FPGA can perform the same role as the FPIC. One possibility is providing significantly more I/O pins in an FPIC than are available in an FPGA. This can be a major advantage, since it takes a significant amount of smaller I/O chips to match the functionality of a single high-I/O chip (i.e., a chip with N I/Os requires three chips with 2/3 the I/Os to match the flexibility). However, because the packaging technology necessary for such high I/O chips is somewhat exotic, FPICs can be expensive. Another possibility is to provide higher performance or smaller chip size with the same I/O capacity. Since there is no logic on the chip, the space and capacitance due to the logic can be removed. However, even with these possible advantages, FPICs face the significant disadvantage that they are restricted to a limited application domain. Specifically, while FPGAs can be used for prototyping, reconfigurable computing, small volume products, fast time-to-market systems, and multi-mode systems, FPICs are restricted to the interconnection portion of prototyping and reconfigurable computing solutions. Thus, FPICs may never become commodity parts, greatly increasing their unit cost.

Discrete components

In the previous sections we have discussed custom fabricated and field-programmable implementation technologies for digital logic. The final alternative is to use standardized, non-programmable chips. Specifically, simple logic structures such as inverters, 2-input to 4-input logic gates, and latches can be prefabricated on individual small chips. By connecting these chips in the proper manner, the desired functionality can be implemented. This interconnection is done at the board level, either by prototyping methods such as wire-wrap and breadboarding, or by fabricating a custom circuit board to hold the parts. Performing the routing customization at the board level instead of the chip level tends to be cheaper and requires less lead time, since board fabrication is much simpler than chip fabrication. Also, since the chips are simple and standardized, they can be prefabricated, and will be quite cheap because of their small size and their economies of scale. These discrete components have been used in many different circuits, amortizing many of the costs. However, since each of the chips will hold only a small amount of logic (perhaps four 2-input functions), a large circuit will require a large amount of chips. This slows down the circuit, complicates the circuit board design, and greatly increases the circuit board size. Thus, while this approach may be cheap for small circuits, it quickly becomes unattractive for large systems.

Microcontrollers and DSPs

By prefabricating all of the components, a discrete component methodology reduces costs and avoids delays associated with custom fabrication. In the discrete component approach, all the chips are quite simple, having very restrictive functionality. However, premade components do not necessarily need to be simple. For example, microprocessors are completely prefabricated, but are some of the most complex chips currently made. Premade components can be found for many standardized, generally useful applications. Two types of premade components that merit special consideration are microcontrollers and digital signal processors (DSPs). These are complex, prefabricated parts that are widely used for many applications.

$$Out = \sum_{i=1}^{k} \left(C_i \times X_i \right)$$

Equation 1. FIR filter equation.

A microcontroller is essentially a simple microprocessor optimized for embedded control situations. The microcontroller has much the same internal structure as a microprocessor, with a datapath capable of performing complex operations and control flow dictated by a series of instructions. The major difference between a microcontroller and a microprocessor is that the microcontroller tends to be smaller, slower, and have a smaller address space, but it has more interfaces for external communications. Many microcontrollers are only 8-bit or 16-bit processors, with memories on the order of only a few kilobytes. However, because they are simple, they can be very cheap. This makes them ideal for control applications, moderating the operation of embedded systems such as automobile engines, microwave ovens, and many other applications. Since they have good external connectivity, and since the performance demands of these applications are in general relatively low, a microcontroller becomes a very cheap alternative to either a large number of discrete components, or expensive custom ASICs. The operation of the microcontroller is completely dictated by a program stored either in an internal or external PROM, making it field-programmable much in the same way as PLDs and FPGAs. However, a microcontroller is capable of much more complex control flow than an equivalent PLD or FPGA, albeit at a much slower speed, making it valuable for many applications. Also, since many applications will fit into the on-chip ROM, a microcontroller provides a complete, highly integrated processing element for many applications. Replacing a microcontroller with a standard microprocessor would require several more chips, since standard microprocessors require separate chips to implement the ROM and the external interfaces.

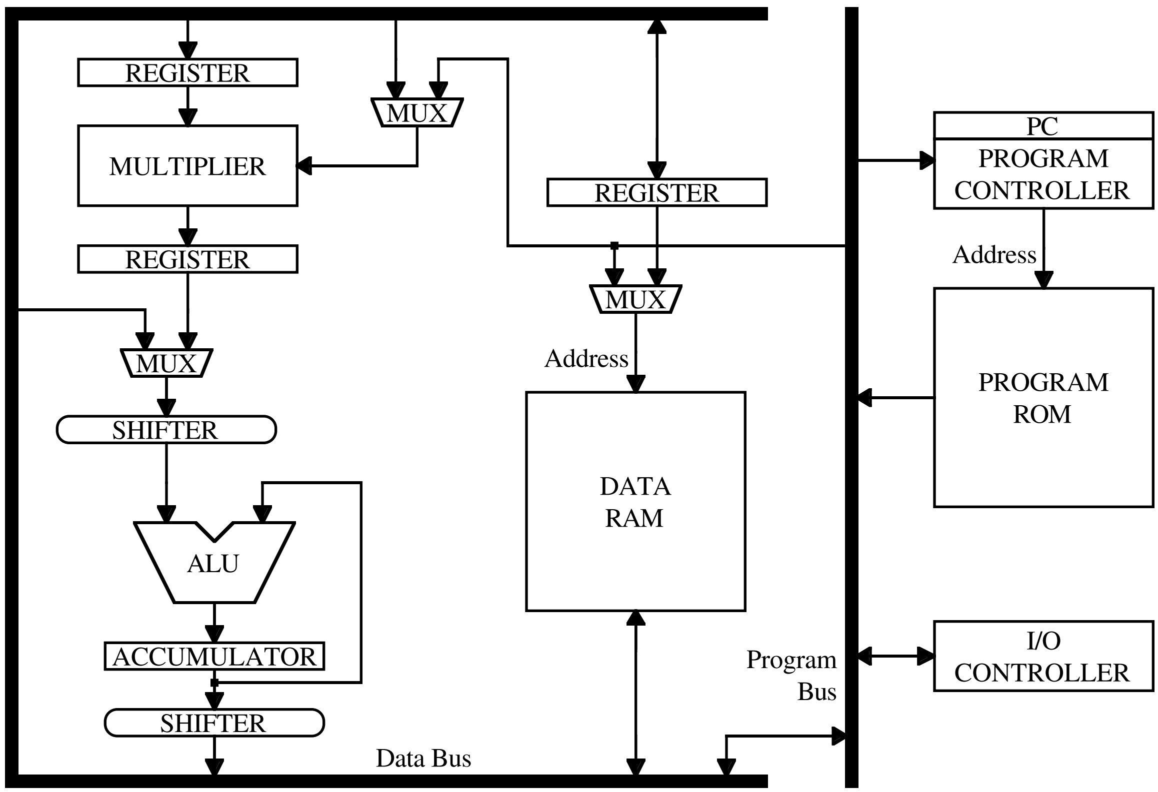


Figure 22. An example DSP architecture [Lee88]. The datapath is at left, which is optimized for high-speed multiply-accumulate computations. Shifters are provided to scale values to avoid overflows. At center is the data RAM addressing hardware, and at right is the program fetch hardware and the I/O controller.

Digital signal processors (DSPs) [Lee88, Lee89] are also quite similar to microprocessors, but instead of using slow, simple processors to reduce costs like a microcontroller, a DSP is highly optimized to provide the highest possible performance for a restricted application domain. A DSP is designed to perform multiply-accumulate operations (A := A + B*C) as fast as possible. An example of a DSP architecture is shown in Figure 22. As can be seen, the DSP includes instruction and data addressing hardware like a microprocessor, but the datapath is optimized for multiply-accumulate operations instead of the more general operations found in a standard processor. Because of the restricted application domain, the processor can be optimized to achieve high performance operations, at least for those operations that fit the DSP's computation model. This yields a programmable processor capable of handling many signal processing tasks. For example, a finite impulse response (FIR) filter computes the function shown in

Equation 1, where X_i is the input value received i steps previous. This computation fits the multiply-accumulate model quite well, and current DSPs can compute a k-input FIR filter in one step per input.

Design style summary

In the previous sections we have discussed many different methods for implementing digital logic. For traditional logic implementation tasks the choice of implementation technology is usually most constrained by capacity, performance, and price. Full-custom designs provide the highest capacity and highest performance, but at a significant price. Other implementations tend to be cheaper (at least in lower volumes), but have lower capacity and performance. In order, from highest capacity and performance to lowest, are Standard Cell, MPGA, FPGA, PLD, and discrete component implementations. The same ordering is true in price, at least at low volumes, with Standard Cell and MPGA implementations costing the most, and FPGA, PLD, and discrete component implementations costing the least. Thus, when choosing an implementation technology, the designer will use the lowest price device that can meet their capacity and performance demands. FPGA and MPGA designs are normally preferred over full-custom implementations, but they may not be powerful enough to handle the designer’s needs. Note that as the volume of parts produced increases, the cost of the various technologies changes. Specifically, while FPGAs and PLDs are much cheaper than other technologies at low volumes, since they have no up-front costs for masks and other fabrication costs, their unit costs are much higher than other technologies. Full custom designs have the highest startup costs, because none of their fabrication is shared with other products, and the design process is quite complex. However, as the quantity of parts produced grows, this fixed cost is amortized over a larger number of chips. This makes the per-chip costs dominate, and a full-custom design becomes the cheapest implementation.

In some cases, the most important feature of an implementation technology is not cost, capacity, or performance, but is instead some other issue. For example, time-to-market can often be a primary factor. If a product must wait for the design and fabrication of a full-custom design, it will take many months or even years for the product to reach the market. However, if an FPGA or PLD implementation is chosen, the logic might be ready in a matter of days. Note that a hybrid approach is also possible. A product can be quickly implemented in FPGAs, and brought to market quickly. While the FPGA-based version is shipped, an MPGA or full-custom implementation can be readied. Once this design is debugged and fabricated, it will replace the FPGAs in the product being shipped. Thus, the lower per-unit costs of an MPGA or full-custom design decrease costs for later products, while the FPGA implementation allows the product to be brought to market as soon as possible.

There are many other situations where the unique abilities of FPGAs play a critical role. These are detailed in Chapter 3. As we will show, the reprogrammability of FPGAs, coupled with their ability to contain significant amounts of logic, open up many opportunities not possible in other technologies.

Overview of single FPGA mapping software

While some circuits are designed by hand, in many cases automatic mapping software is critical to logic development. This is particularly true for technologies such as FPGAs, where in general the complete mapping process is carried out by mapping software. In the following sections we will discuss some of the most important steps in mapping to LUT (lookup-table) based FPGAs. Note that the process is similar, but not identical, for other types of FPGAs. First is technology mapping, which restructures the input netlist into the logic blocks of the FPGA. Next, placement decides which specific logic blocks inside the FPGA will contain the logic functions created by technology mapping. Finally, routing determines what routing resources inside the FPGA will be used to carry the signal from where they are generated to where they are used. A more detailed treatment of all of these tasks can be found elsewhere [Venkateswaran94].

Technology mapping

The user of an FPGA provides as input a circuit specified as some interconnection of basic logic gates and functions. These functions may havemore or less inputs than the LUT in the FPGAs that will implement them. When the logic gate has too many inputs, it needs to be split into smaller functions that can fit inside the LUTs in the FPGAs. If the gates have too few inputs, several interconnected gates could be combined to fit into a single LUT, decreasing the amount of LUTs needed to handle the mapping. By reducing the number of LUTs, more logic can be fit in the same sized FPGA, or a smaller FPGA could be used. The process of restructuring the logic to best fit the logic blocks in an FPGA is called technology mapping.

There are many different methods and approaches to the technology mapping of circuits for FPGA implementation [Brown92a, Vincentelli93]. An example of this process is shown in Figure 23. The circuit at left is restructured into four 5-input LUTs, designated by the gray loops at right. Some of the logic function, such as gate **7** at left, have more inputs than the LUT can handle. Thus, the gate will be decomposed into two gates (**7** and **8** at right). Then, the gates are grouped together into LUTs, while trying to minimize the total number of LUTs required. Note that this grouping can be complex. For example, even though gates **1** and **3** each have three inputs, and thus should not fit into a single LUT with gate **2**, since input **C** is shared between the two gates a single 5-input LUT can handle all three gates. Finding this reconvergent fanout can be difficult. Also, this grouping process can cause the logic to be restructured. For example, gate **5** at left is duplicated, creating gates **5** and **6** at right. Although this seems like it would increase the hardware cost, replicating the logic can actually reduce the logic cost. In the example shown, if gate **5** was not replicated, it could not be grouped with either of its fanouts, since by grouping a gate with its fanout the gate's output is no longer available to other functions. However, by duplicating the gate, the duplicates can each be grouped with one of the fanouts, reducing the total LUT count.

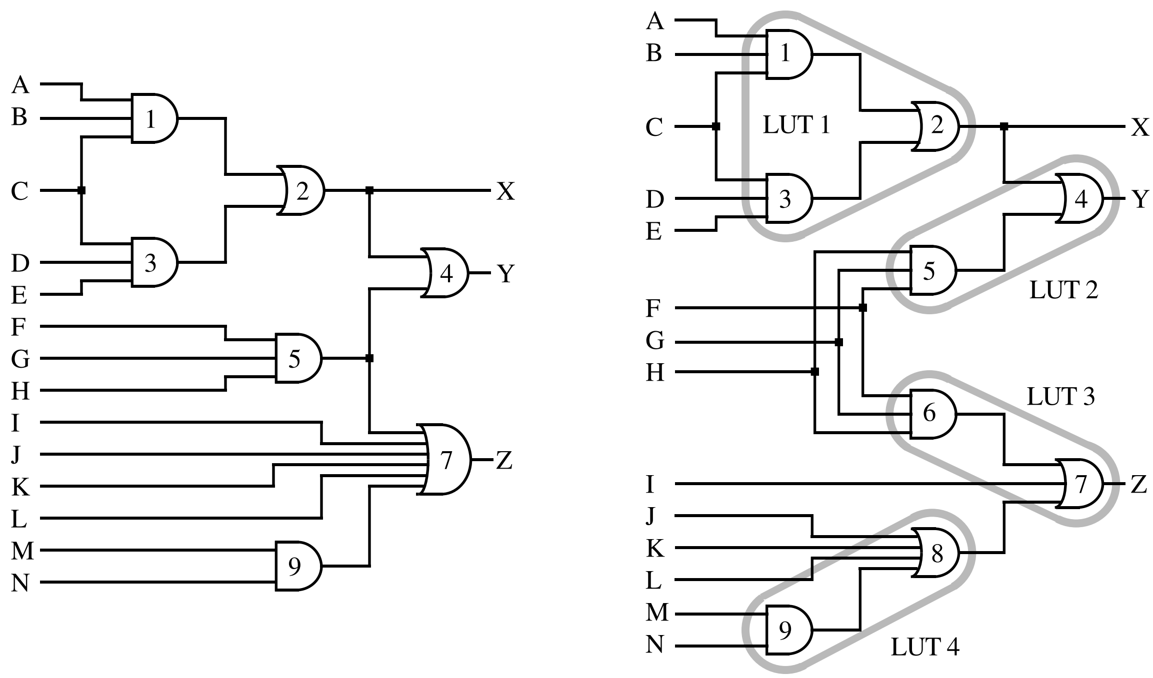


Figure 23. Example of 5-input LUT technology mapping. The input circuit (left) is restructured and grouped together into 5-input functions (right). The gray loops at right indicate individual LUTs. The numbers on the gates are for identification.

Many approaches and optimizations are possible for technology mapping. Instead of just mapping for LUT count, some algorithms optimize for performance or routeability (i.e., how easy it is to route the logic generated by technology mapping). Also, real FPGAs usually have logic blocks that are more complex than a single n-input LUT, and thus require more complex mapping algorithms. Numerous approaches to these and other mapping issues are presented in the literature.

Placement

Placement takes the logic functions formed by technology mapping and assigns them to specific logic blocks in the FPGA. This process can have a large impact on the capacity and performance of the FPGA. Specifically, routing between distant points in an FPGA requires a significant amount of routing resources.

Thus, this path will be much slower, and use up many valuable resources. Because of this, the primary goal of placement is to minimize the length of signal wires in the FPGA. To do this, logic blocks that communicate with one another are placed as close together as possible. For example, Figure 24 shows a placement of the logic functions created by technology mapping in Figure 23. Since function **2** takes the output of function **1** as an input, and shares inputs **F**, **G**, and **H** with function **3**, function **2** is placed between function **1** and function **3**. Also, **F**, **G**, and **H** are assigned to I/O blocks close to function **2**.

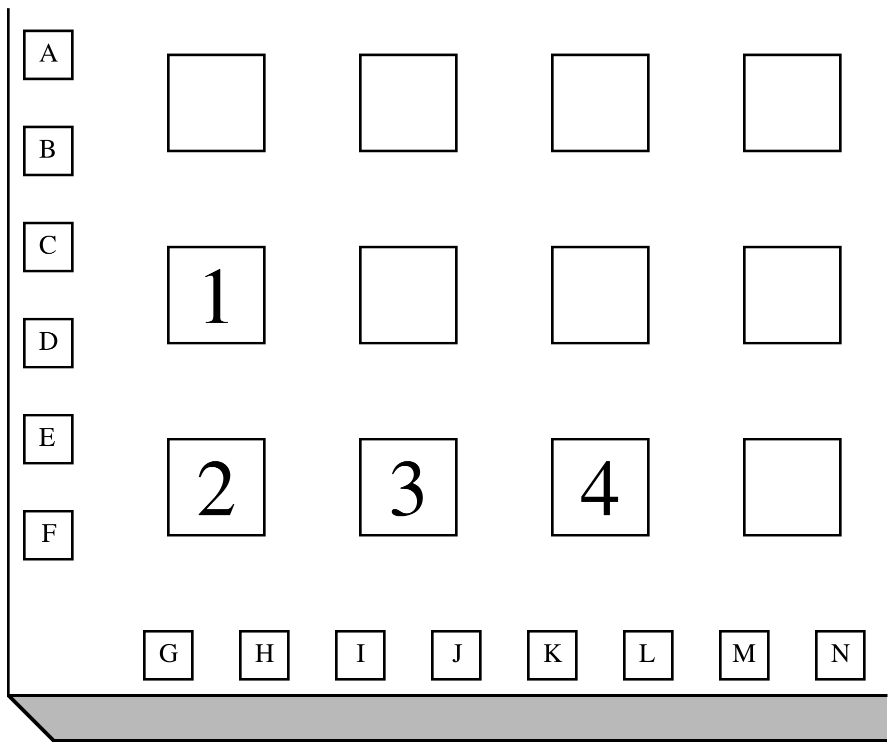


Figure 24. Placement of the circuit from Figure 23. The numbers are the number of the LUT (from the technology mapping) assigned to that logic block, while the letters are the assignment of input signals to I/O blocks.

Placement is a complex balancing act. Logic circuits tend to have a significant amount of connectivity, with many different functions communicating together. Trying to find the best 2D layout of these elements can be quite difficult, since many functions may want to be placed together to minimize communications, while only a small fraction will fit within a given region of the FPGA. Thus, the placement tool must decide which functions are most important to place together, not just to minimize the distances of communications between these functions, but to minimize the total communication in the system.

The most common technique for performing placement for FPGAs (as well as other technologies) is simulated annealing [Shahookar91]. In order to use simulated annealing to solve an optimization problem, the programmer must generate a cost function and a move function. A cost function looks at a state of the system and assigns a value to the desirability of that state, with a lower value indicating a better result. For placement, a state of the system would be an assignment of logic functions to logic blocks, and I/O connections to I/O blocks. A cost function could be the total wirelength necessary to route in this configuration (this would need to be an estimate, since exact numbers are time-consuming to calculate, and simulated annealing requires the cost metric to be quickly computed). Thus, states that have the smallest cost would require the least amount of routing, and thus would be better placements. More complex cost metrics, which take into account issues such as critical paths, are also possible. A move function is simply a method of transforming the current state of the system into a new state. Through repeated applications, this function should be capable of transforming any state of the system to any other. For placement, a move function could be to randomly pick two logic blocks in the FPGA and swap their contents.

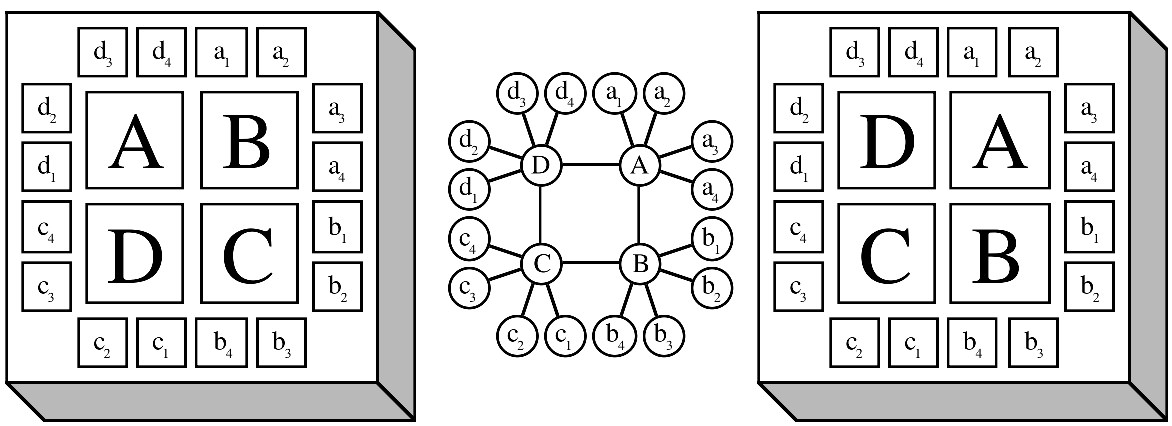


Figure 25. An example of a local minima in placement. The circuit at center, placed as shown at left, is in a local minima. No swap of logic or I/O functions will reduce the total wirelength. However, the placement at right is significantly better.

One way to perform placement once a cost and move function are defined is to first pick a random starting point. The algorithm then repeatedly applies the move function to the current state of the system, generating a new state. If this state has a lower cost than the current state, it is accepted, and replaces the current state. Otherwise the current state is retained. Thus, this algorithm will greedily accept good moves, and move into a local minimum in the cost function’s state space. The problem is that most cost functions have a huge number of local minima, many of which are much worse than the optimal placement (Figure

25). Specifically, from a given state there may be no way to swap two logic functions and reduce the cost (thus, we are in a local minima), though two or more pairs of swaps can greatly improve the placement.

Simulated annealing avoids the problem of getting caught in local minima. Like the greedy algorithm, simulated annealing takes the current state, uses the move function to generate a new state, and compares the cost metric in both states. If the move is a good move (that is, the cost function is lower for the new state than the old state) the new state replaces the current state. However, instead of rejecting all bad moves (move that increase the cost function), simulated annealing accepts some bad moves as well. In this way, the algorithm can get out of a local minima by accepting one or two bad moves. Subsequent good moves will then improve the results again, hopefully finding better results than the previous local minima.

The method of how to determine what bad moves to accept is critical. The probability of accepting a bad move is usually $\exp(-\Delta C / T)$, where ΔC is the difference between the current and the new state's cost functions, and T is the temperature, a parameter that allows more or less bad moves to be accepted. Whenever the algorithm finds a bad move, it calculates the different between the current and new state. The algorithm then randomly determines whether to accept this bad move, and it is more likely to accept moves causing small increases in the cost function than big increases in the cost function (that is, the worse the move, the less like it is to be accepted). Also, over time it gets pickier, accepting less and less bad moves. This is done by lowering the temperature parameter T. At the beginning of the annealing the algorithm accepts many bad moves, and randomly wanders around the search space. Since it always accepts good moves, it tends to stay in the portion of the search space where better states are found, but the large amount of bad moves accepted keep it from getting stuck in any one place. As time goes on, T is decreased, and the algorithm accepts less and less bad moves. As this happens, it gets harder for the algorithm to wander away from the areas in the cost function where the better states are found. Thus, it is stuck in one region of the state space based on very coarse-grain measures of goodness, and it begins to stay in parts of this region where better states are found. The algorithm continues to accept less and less bad moves, until eventually it accepts only good moves. At this point, the algorithm is zeroing in on a local minima, though this minima tends to be much better than the average local minima in the search space, since the algorithm has slowly gravitated to areas in the search space where better states are found. In this way, simulated annealing can find much better results than greedy approaches in complex search spaces.

By applying simulated annealing to a placement problem, the complex relationships between the functions in the mapping can be considered. The algorithm will slowly optimize the state, coming up with a good final placement. Note that simulated annealing can be quite time-consuming. This is because the algorithm must be allowed to accept many good and bad moves at each acceptance level, so that the algorithm can explore much of the search space. Thus, multiple hour annealing runs are not exceptional.

Routing

Routing for FPGAs is the process of deciding exactly which routing resources will be used to carry signals from where they are generated (the source) to where they are used (the destinations). Unlike many other technologies, FPGAs have prefabricated routing resources. Thus, instead of trying to limit the size of routing channels (the goal in standard cell routing), an FPGA router must work within the framework of the architecture's resources. Thus, the router must consider the congestion of signals in a channel, making sure than no more routes are made through a region than there are resources to support them. Otherwise, if too many resources are required the routing fails, while in other technologies the region could just be enlarged.

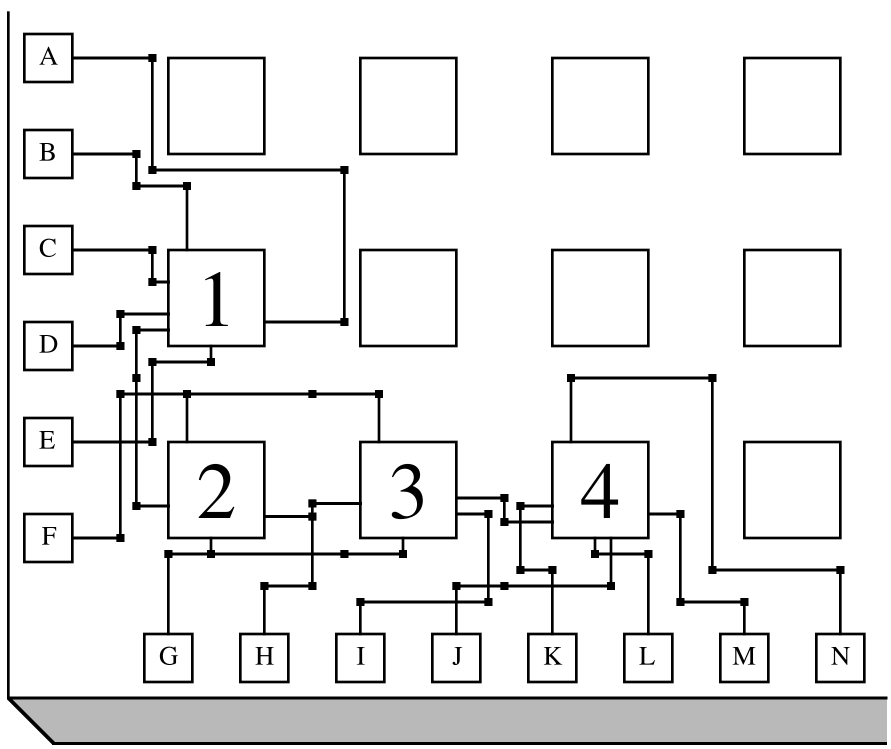


Figure 26. Routing of the placement from Figure 24. The small black squares are the configuration points used for this mapping.

An example of the routing of the placement from Figure 24 is shown in Figure 26. The router must decide which inputs and outputs of the logic blocks to connect to, which channels to route through, and how to connect through the switchboxes in the architecture. It is allowed to choose which terminal on the logic block to connect to because the logic is implemented in LUTs, and all inputs to a LUT are equivalent (assuming the programming of the LUT is modified accordingly). In deciding which channels and wires to use, and how to connect through the switchboxes, the router must ensure that there are enough resources to carry the signal in the chosen routing regions, as well as leaving enough resources to route the other signals in the system. One algorithm for performing this routing is presented in [Brown92b]. Here, the algorithm is divided into a global and a detailed router. The global router picks which routing regions the signal will move through. Thus, it will select the routing channels used to carry a signal, as well as the switchboxes it will move through. In this process, it takes care not to assign more signals to a channel than there are wires. However, the global router does not decide which specific wires to use to carry the signal. The detailed router handles this problem, making sure that it finds a connected series of wires, in the channels and switchboxes chosen by the global router, that connects from the source to all the destinations. Both algorithms worry both about congestion-avoidance, making sure that all signals can be successfully routed, as well as minimizing wirelength and capacitance on the path, attempting to optimize the performance of the circuit. By running both algorithms together, a complete routing solution can be created.

Software summary

Tools are available that automatically map a structural circuit into FPGA logic. This involves several steps. First, technology mapping restructures the logic to fit the logic blocks of the FPGA, attempting to minimize the number of blocks required. Then, placement assigns these functions to locations in the FPGA, attempting to minimize the routing required. Finally, routing determines which specific resources in the FPGA will carry the signals from source to destinations, simultaneously attempting to route the largest number of signals, and achieve the fastest implementation. Once routing is done, there is a complete description of how the circuit should be mapped to the FPGA. A simple translation takes this description and creates a programming file that, when downloaded to the FPGA, implements the desired functionality.

As discussed, mapping to a single FPGA is a highly automated process. The user is only required to specify the desired functionality, and the software automatically creates a realization. This thesis concerns systems of FPGAs used to implement circuits. While many of the issues are different for the multi-chip case than for the single chip implementations discussed here, single-chip mapping tools are still important. Specifically, several software steps, applied in series, reduce a multi-FPGA mapping into several single-FPGA mappings. Then, single-chip mapping tools are applied to the system's chips. In Chapter 9 we discuss the algorithms necessary to convert a multi-FPGA mapping into several single-FPGA mappings.