WP101 (v1.2) October 9, 2000

XPLA1 Family Architecture

Author: Mark Aaldering

Summary

In designing with CPLDs, designers want it all—devices that offer high speed, high density, and the flexibility to make changes to their design at any stage of the design process—especially last minute changes to the logic. A particular devices' ability to meet all of these critical needs efficiently is often constrained by the basic architecture of the CPLD. The Xilinx XPLA (eXtended Programmable Logic Array) architecture is the result of extensive research into the effect of architecture on these three critical system needs—speed, density, and design flexibility—and delivers a fourth generation solution that is superior to previous architectures.

The XPLA Architecture

The basic components of CPLD architecture that affect the devices' speed, density, and design flexibility can be broken into four distinct areas. These four areas are the basic interconnect methodology, the size of the logic blocks, the methods used to allocate logic to the Macrocells, and the timing model of the device.

Interconnect

Early in the evolution of PAL Architectures, designers at MMI developed a device called the 'MegaPAL". This device was an early precursor to today's CPLDs. The basic concept was that if a small 16 series PAL was good, a much larger device would be even better. They did this by simply growing the size of the programmable AND fixed OR PAL array to accommodate the larger number of inputs and macrocell outputs. Unfortunately, the speed obtainable through a conventional PAL array decays near exponentially as additional inputs and outputs are added—the MegaPAL was also MegaSlow—and never became commercially successful. From this failure came the seeds to successfully making PLD architecture devices larger, by adding a simple interconnect array that joins many smaller PLD-like blocks (logic blocks) onto a single chip.

In CPLDs, this interconnect resource acts like a crosspoint switch to route signals from the Inputs, I/Os, and Macrocell feedbacks to a number of logic blocks. As a small, simple switching mechanism, its design can avoid the capacitive loading that caused the large, unified MegaPAL array to suffer in performance. In addition, the logic blocks themselves are kept relatively small, and as a result their programmable logic arrays are fast.

The XPLA Architecture's interconnect, called the ZIA (Zero-power Interconnect Array) is conceptually similar to other CPLDs interconnect. The ZIA offers a fixed, deterministic delay for routing signals from any macrocell or pin to the logic blocks. This delay is so small (on the order of 1/5th of a nanosecond), that the timing is not specified as a separate specification but is included in the T_{PD} and T_{SU} specifications. In addition, the ZIA has been designed to consume zero static power in operation.

The single most critical parameter of interconnect operation, however, is rarely discussed by CPLD vendors. This metric is in fact the degree to which the interconnect fulfills its ability to route signals under worst case conditions like a true crosspoint switch would. Many users have been burned by architectures that are able to do an initial routing, when the software "floats" the pins, but fails to route signals into the logic blocks when minor design changes occur late in the design after the printed circuit board has been laid out. The interconnect is the first area that impacts the ability to support fixed pinouts. The ideal performance of an interconnect is to fully

emulate a crosspoint switch, where every input to the array can be connected to every output of the array under fixed pinouts. Some first generation devices used full crosspoint switch arrays, and as a result offered 100% routability, at a significant price. In building a crosspoint switch, these devices required a fuse at every intersection of the input and output line in the array. For a 128 macrocell device, this would translate into more than 65,000 fuses. More significantly, these fully populated crosspoint switches were relatively slow—accounting for an 8 ns to 15 ns delay by themselves.

The next step in interconnect evolution was the use of muxes to emulate crosspoint switches, a technique that all contemporary devices deploy. In Figure 1, a set of 16 muxes that are two bits wide form an interconnect that has 32 inputs and 16 outputs. The use of muxes has two immediate effects. The first is that the delay through the interconnect is typically equivalent to a single mux delay which is typically well under 1 ns.

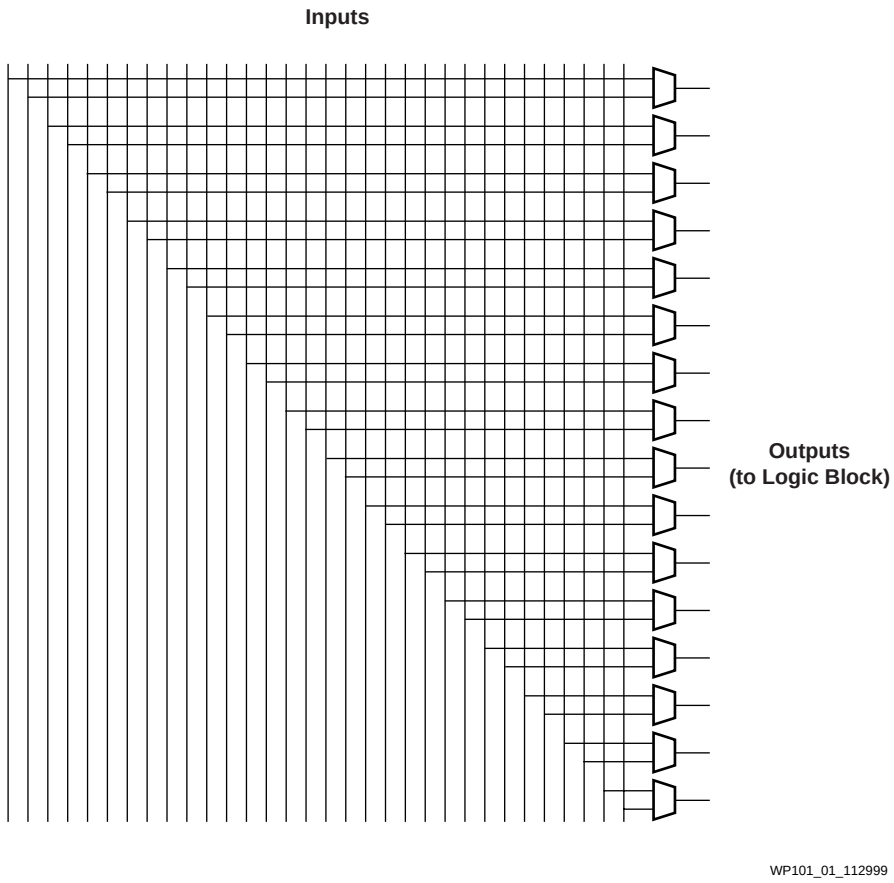

Figure 1: **Mux Interconnects**

The second obvious effect is the reduction of the number of fuses required to implement the interconnect. A 128 Macrocell device no longer requires a fuse at the intersection of every input and every output. As result, the number of fuses can be reduced from approximately 65,000 to less than 2,000. Unfortunately if the architecture of this interconnect is not well engineered, signal blocking can occur that results in devices that will not route as iterative changes are made to a design that has had the pinout fixed (sometimes referred to a refitting). As an example, consider a design as shown in Figure 2 that has routed signals ACK and $\overline{\text{IRQ}}$ connected to pins A and D early in the design before the PCB has been layed out. The design software will place these signals onto Muxes 1 and 2 so that they will both route into the logic block. Later in the design, the engineer is requested by marketing to add some power-down modes so that the end product can be sold as a "green" appliance.

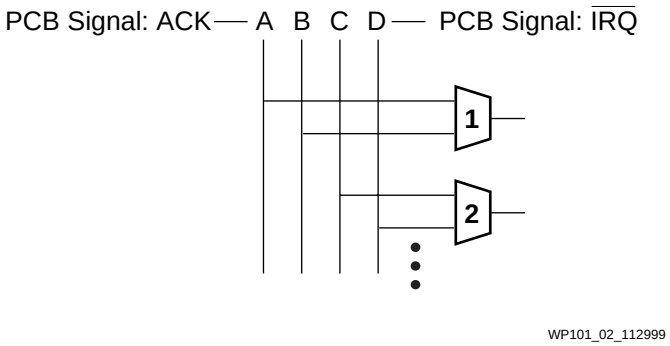

Figure 2: **Example Design Layout**

Now that the PCB has been completed, this signal can only be tied to pin C. Looking at our interconnect, we can see that pin C is blocked from entering the logic block, as the mux that would have allowed the sleep signal into the logic block is already in use by the $\overline{IRQ}$ signal. In order to add this feature, the engineer must relayout his PCB. If this occurs on the Friday before Comdex, the engineer will not be happy about this change.

Suppose instead that the Muxes were both wider and that there were a larger number of them. Looking at Figure 3, observe that each input signal now propagates to more muxes that are now three bits wide. As a result, each input now has more opportunities to enter the logic block—on average 2.25 instead of just 1.

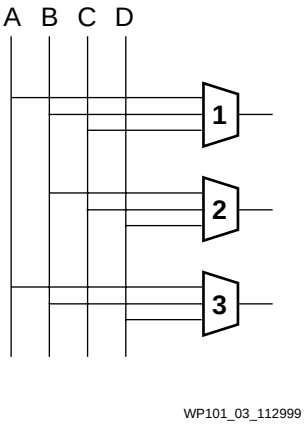

Figure 3: **Adding More Muxes**

If in fact the muxes were four bits wide, this mux based interconnect would in fact logically emulate a crosspoint switch, but would require more E2 fuses and be slower due to the wider mux width. This is the trade off that faces architects that design modern CPLD interconnects. By extensively simulating the width of the muxes, and the number deployed, mux based interconnect can be designed such that the probability of signal blocking is statistically very low. Xilinx XPLA interconnect deploys a sufficiently large number of input muxes, of sufficient width, to guarantee routability under worst case conditions. At Xilinx, this interconnect architecture was subjected to over 16 million iterations of worst case fixed pinout routing by our software design team. This resulted in worst case signal routing of 99.997% when every signal is in use and all signals have a fixed pinout. If only 35 of the 36 logic block inputs are used, 100% of the 16 million fixed signal routings completed successfully. This solution allows designers total freedom to make design iterations without the fear of having to re-layout the PCB.

Logic Block Size

The next area that merits consideration is the number of inputs to the logic block. If this number is too large, the size of the logic array inside the logic block will grow to a size where the speed of the array is compromised (just like the MegaPAL). If the number of inputs is too small, the complexity of the logic that can be implemented in a logic block (and therefore within a single clock cycle) will suffer. As an example, consider a common design example—a 16-bit loadable counter, as shown in Figure 4.

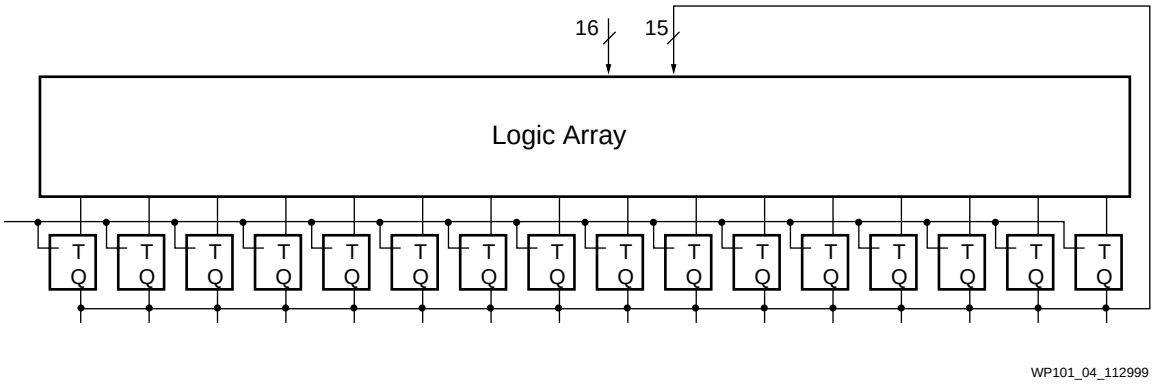

Figure 4: **A 16-bit Loadable Counter Design Example**

The counter requires 15 bits of feedback to operate correctly. In addition, to support the ability to load the counter with an external value, another 16 data bits are needed. Finally a minimum of a single control signal is required to enable the load function. Thus, a 16-bit loadable counter requires a minimum of 32 signals that must be able to enter the Logic Block. In architectures that feature 16 Macrocells, but have fewer than 32 inputs, this trivial counter design will not fit. This scenario also impacts complex state machines which inherently have extensive feedback and input signals. The XPLA Logic Blocks feature 36 inputs, which allow complex state machines and 32-bit decoders to easily fit along with any associated control signals.

Logic Block Allocation Method

In the earliest simple PLDs, the logic array allocated a fixed number of product terms to each output—typically eight. In later versions, the number of product terms was still fixed, but in order to accommodate the need for varying amounts of logic in different macrocells the number of product terms was varied by output. The classic example of this approach is the workhorse of simple PLDs, the 22V10. This device has two output macrocells that have eight product terms, another two that have ten product terms, and then pairs with 12, 14, and 16 product terms, respectively. This approach allows the designer to place logic that requires a large amount of logic on an output that has a higher number of product terms—say 16. This approach is a distinct improvement over the earlier approach of eight fixed product terms per output, but still has some problems.

These problems are the fixed nature of the product terms and the granularity of the logic distribution. The difficulty with the product terms on an output having a fixed number is that the logic required may exceed the amount available. The worst case scenario is that the logic becomes so complex that the number of product terms require exceeds the maximum available on the device—16 product terms. When this occurs early enough in the design, different synthesis options, taking multiple passes through the array, or splitting of the logic may resolve the problem. If this occurs after the PCB has been layed out the designer may be facing the difficult necessity of re-laying out his board to accommodate additional devices.

Knowing that additional logic might be needed in the "11th hour" of a design cycle, many designers would try to use fewer than the maximum number of product terms available on a given output. Retaining a few "spare" product terms allowed small changes to be made without undue grief. Unfortunately, as every designer knows, small changes have a way of becoming

large changes, and the amount of logic required to implement them can quickly evolve beyond what is still available on the output. Once again, because the logic is allocated on a fixed rather than a dynamic basis, the designer is thrust into a difficult situation where the design cannot easily be changed. In PLDs of any size, therefore, the *logic allocation method* comprises a second area that critically impacts the refitting of fixed-pinout designs.

The 22V10 logic allocation method results in inflexible product term granularity that can under-utilize logic in a device. Since the minimum number of product terms on any output is eight, what happens when the logic requires only one product term? Since the allocation is fixed, the other seven product terms are wasted. Recent studies of synthesis practices show that 70% to 80% of all designs deploy fewer than five product terms per output. In an architecture that has no fewer than eight product terms per output, this represents a considerable loss of logic in unused terms. Everyone who has designed with a 22V10 knows that some outputs use only a few product terms. At the same time, however, most engineers will likewise find that a few complex logic situations require more than the 16 product terms available in this device. Thus, the fixed variable logic distribution present in the 22V10 attempts to minimize the losses that occur due to the granularity of the logic, while still offering sufficient logic on specific outputs to perform complex logical tasks.

Early CPLDs attempted to solve these logic allocation problems by providing both a fixed set of product terms and a dynamic "pool" of logic that could be used by all the macrocells. Citing statistics by the manufacturer that "80-90% of all logic implemented in typical designs could be achieved with three terms," these devices offered three dedicated product terms per output. To augment these dedicated product terms, they added an array of *foldback NAND gates*, also referred to as "shared expander product terms." In use, the foldback NAND method implemented logic that would be fed back into the array for use by any or all of the dedicated product terms to expand the logic on an output. As a side benefit, pairs of foldback NANDs could be configured as a "soft" register, thereby increasing the register count. However, in addition to adding time (as the signal must be fed back into the array), foldback NAND structures are much more difficult to work with in terms of synthesizing general purpose logic into them.

The next trend to appear was the use of *dynamic product term switching* schemes. Alternatively called "product term steering" or "parallel expanders," the basic premise is that the groups of product terms (often called clusters) are sent to a switch box that can route them from one macrocell to another as needed.

At first inspection (Figure 5), this seems like a rational method of increasing the logic on outputs where it is needed. What is often overlooked, however, is that when product terms are steered to a macrocell requiring more logic, the terms have to be taken away from some *other* macrocell. When this occurs, the donor macrocell is either stranded without any associated logic, or is left with so little logic that it is unlikely to be useful. In the example in Figure 6, each macrocell in its native state has four product terms routed to it. As shown, macrocell #2 has a complex equation associated with it that requires 16 product terms. To accomplish this, three neighbor macrocells switch their cluster of product terms to macrocell #2. At first glance, it is tempting to assume that the logic steered away from the donor macrocells could be replaced with a cluster from some other macrocell. While this is true, this ends up being an elaborate shell game, as this next donor is then left without logic. The net impact of macrocell #2 needing additional product terms is the loss of three macrocells to serve the logic needs of one output—a significant penalty. This is why some refer to this scheme as product term "stealing," as the donor macrocells are likely to encounter logic starvation. Recent architectures have lessened this starvation issue by allowing single product terms to be steered to adjacent macrocells, although there can be invariant timing changes associated with this steering.

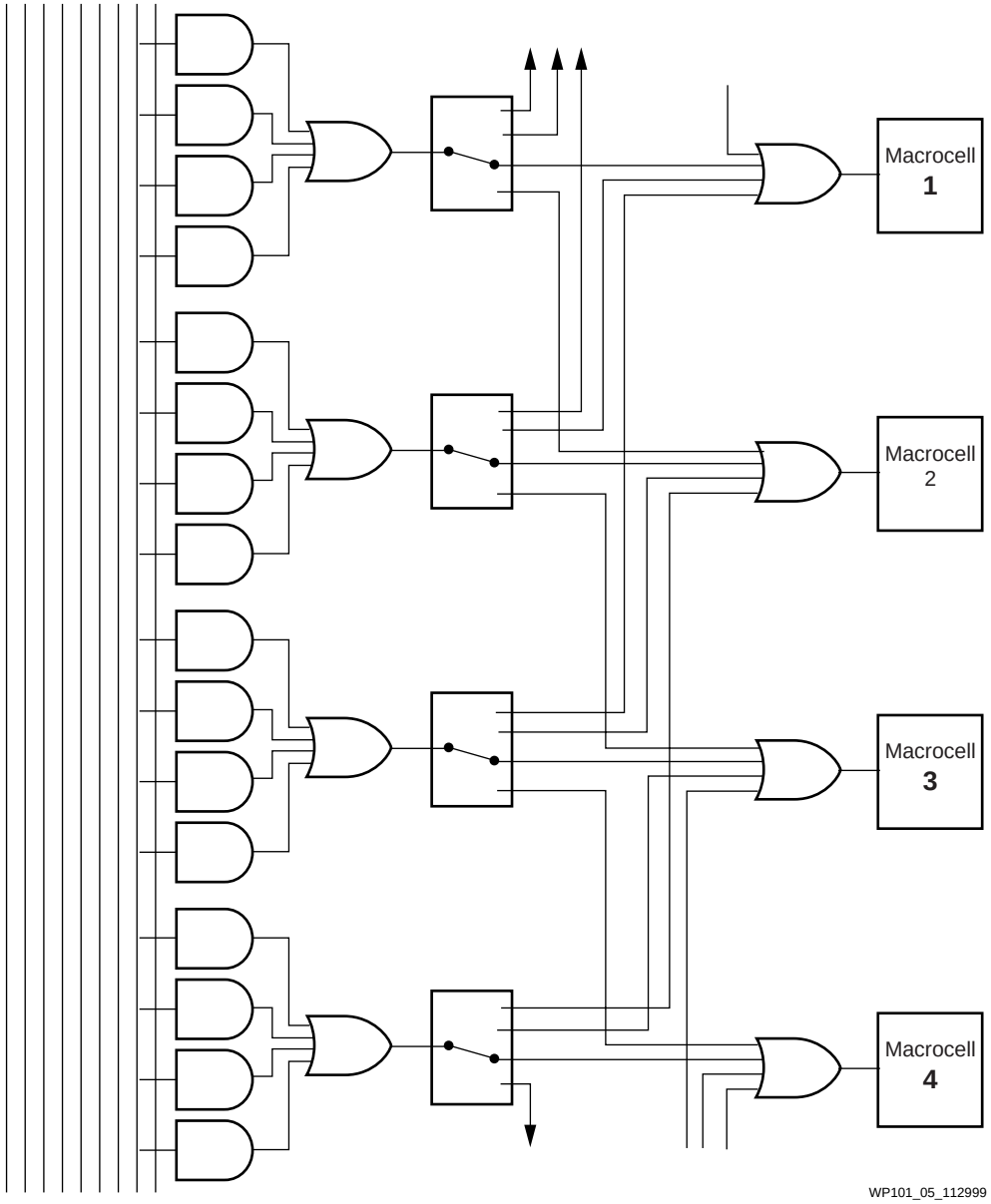

Figure 5: **Dynamic Product Term Switching Schemes**

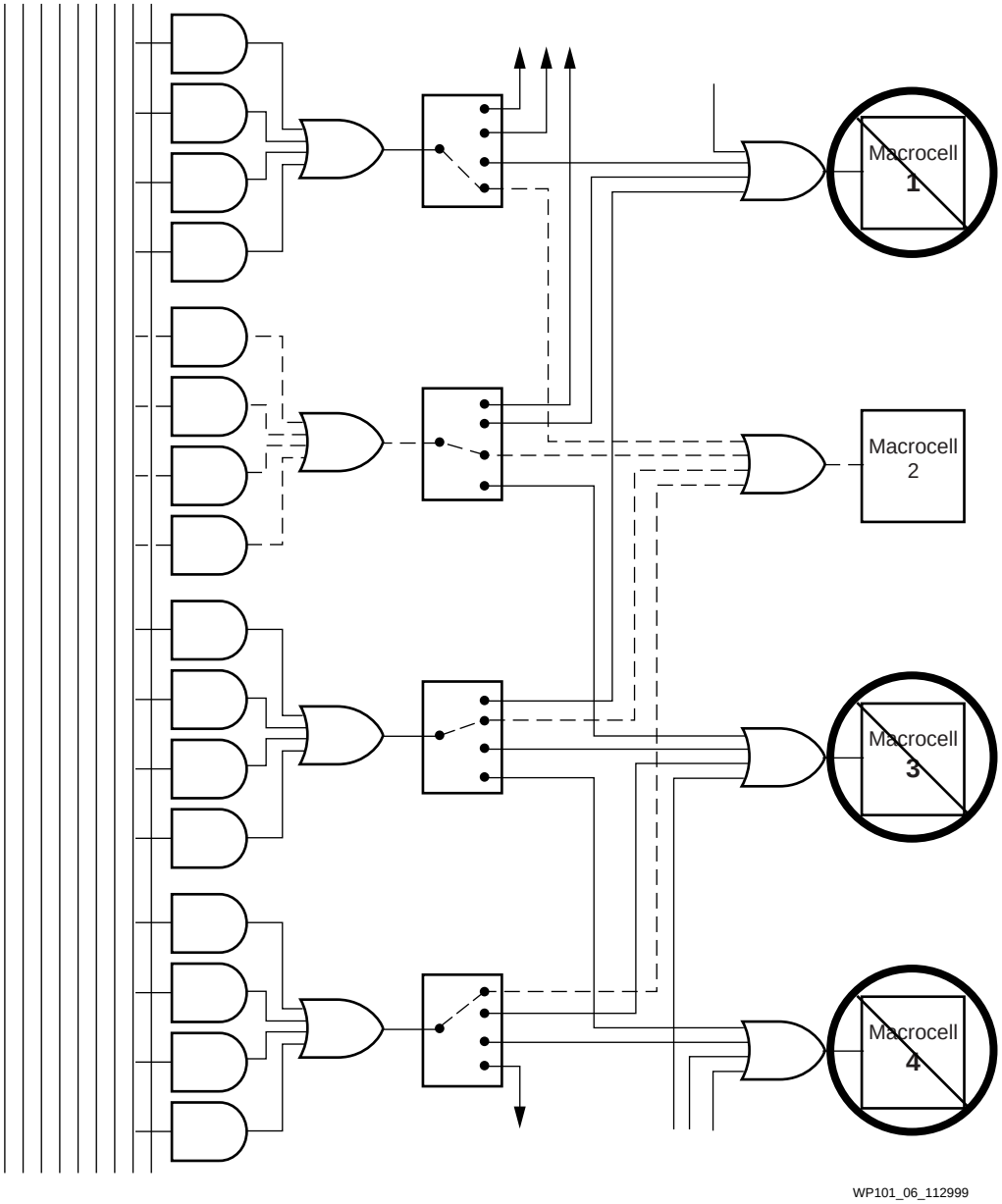

Figure 6: **Example of Product Term "Stealing"**

The final issue in product term steering is the ability to refit designs. As an example, one can examine a design in the final stages of completion that has utilized all the macrocells in the block, and, being uncharacteristically well behaved, uses just three product terms per output (Figure 7). Should marketing suddenly require an extra feature, the design tweak will inevitably require one of these outputs to have more than four product terms. Since we are using all of our macrocells, stealing logic from one of them will result in the design no longer fitting, as the "starved" macrocell can no longer route logic to the output pin it was serving on the PCB. This is true regardless of whether the device has macrocells hard-wired to pins, or employs an additional "output routing pool." As a result, designers who use these devices often learn through painful experience that, with these architectures, it is safest to leave a number of macrocells unused if the ability to refit designs is important. Therefore, to be safe, expensive devices are often not fully utilized. This is especially true in the case of devices that are in-circuit programmed on the board, as being unable to refit when attempting a field update can be quite expensive.

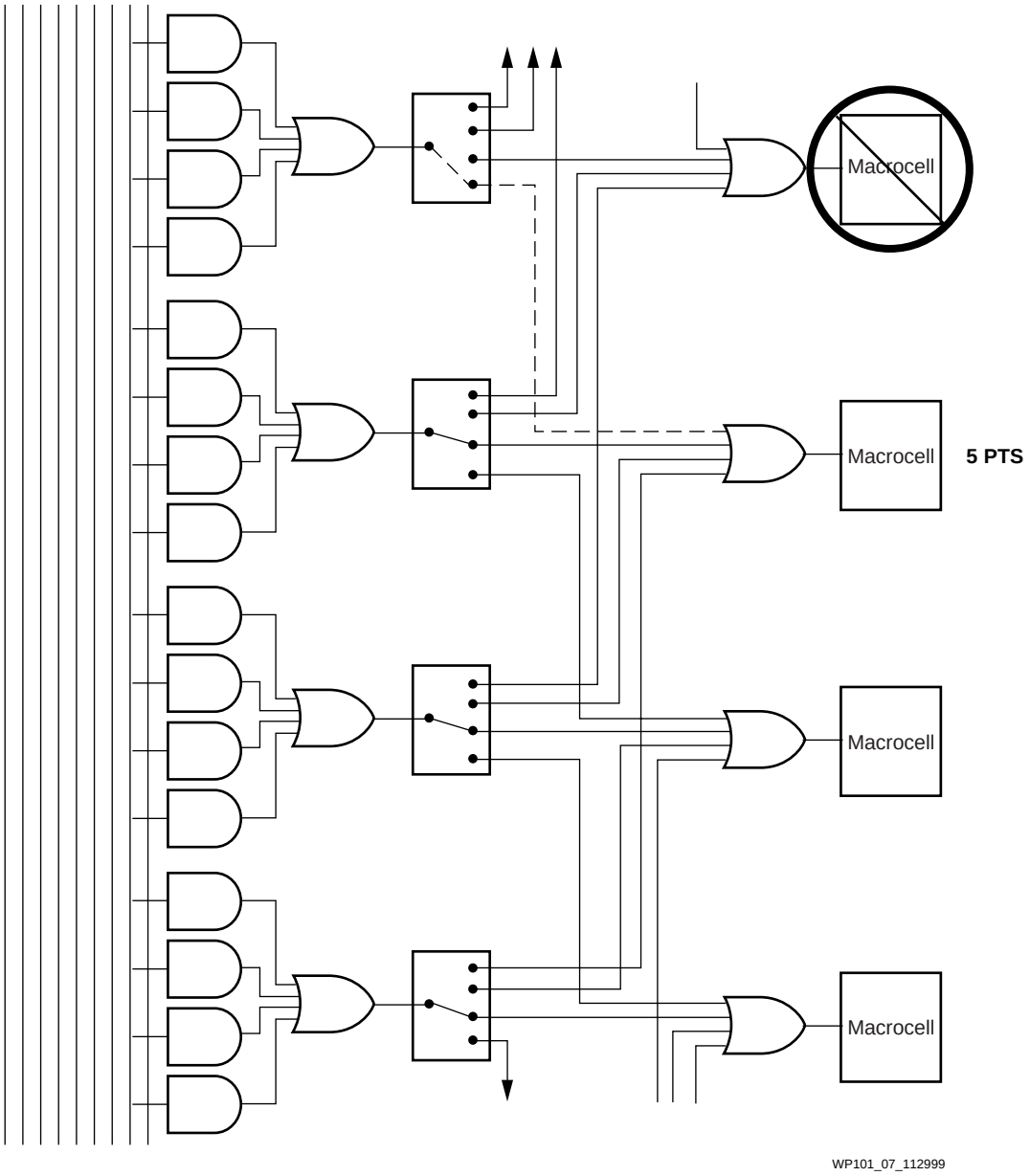

Figure 7: **Five Product Term Requirement**

Philips Semiconductors, when it was operating its PLD division under the Signetics label, developed the first commercially available device with a programmable logic array (PLA) array in its 82S100. This device offered a fully programmable AND array that delivered its product terms to a fully programmable OR array, a combination which eliminated the logic allocation issues associated with foldback NANDs and product term steering mechanisms. Traditionally, however, having two fully programmable arrays resulted in devices that were slower than devices featuring a programmable AND with a fixed OR (PAL). As a result, the Original XPLA logic allocation method uses a patented structure that combines a PAL array and a PLA array—hence the term eXtended Programmable Logic Array. This patent is now owned by Xilinx.

As detailed in Figure 8, the PAL array has five dedicated product terms for every output macrocell. These PAL terms are never steered, stolen, or folded back. Through this PAL array section on 32-macrocell devices, speed from any pin to any pin is 6 ns. As additional logic is needed on any output, the free pool of 32 product terms in the PLA section can be tapped via

the fully programmable OR array. As shown in the close-up detail of Figure 9, the lower macrocell requiring six product terms uses its five dedicated terms, and one additional term from the PLA array to reach the required logic needed. Significantly, this logic allocation was achieved without stealing logic from neighboring macrocells, and the additional logic was allocated exactly as needed—just one additional term, not four with three wasted. In the Original XPLA architecture, the cost of using this additional logic is a fixed 2 ns delay. For the example above, the T_{PD} from any pin to the pin now using logic in the PAL + PLA is 8 ns total.

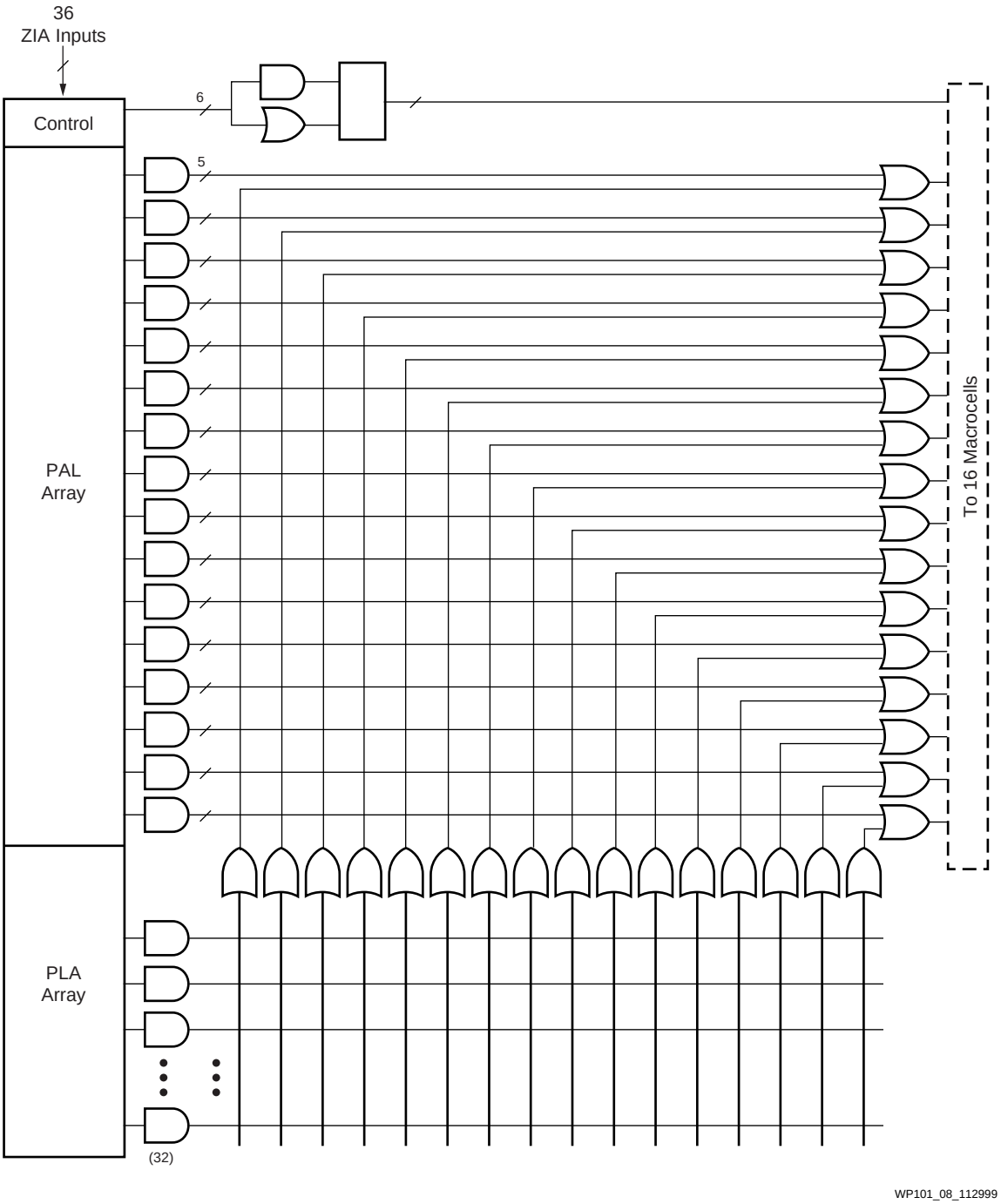

Figure 8: **PAL Array with Five Dedicated Product Terms**

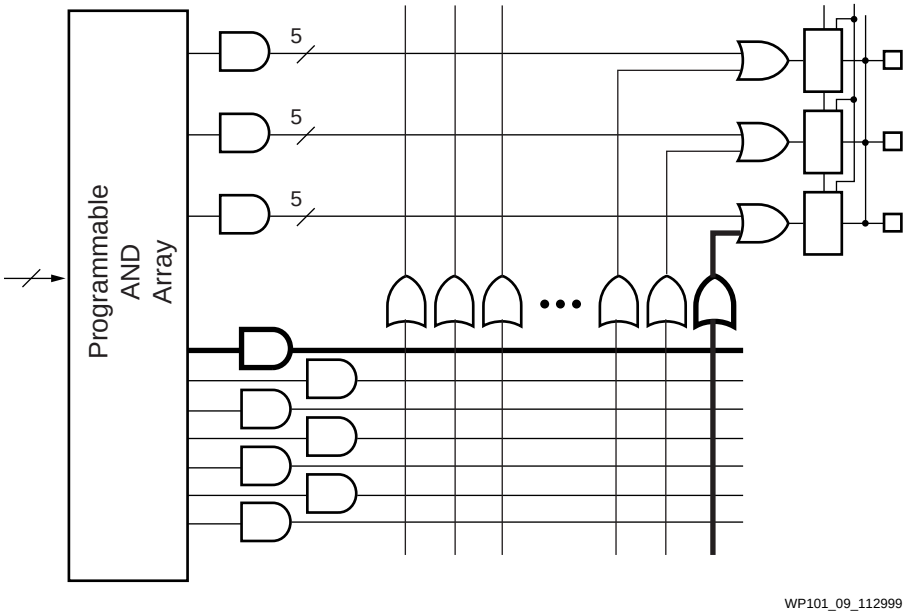

Figure 9: **Close-up Detail**

In Figure 8, eight product terms are now needed on this macrocell. In this case, three product terms are used from the PLA in addition to the five dedicated PAL terms. Once again, logic is utilized in the exact amount required, exactly where needed without sacrificing other macrocells. And the T_{PD} from pin-to-pin is still 7.5 ns for this example. In fact, as many as all 32 PLA terms can be used on any output in conjunction with the five dedicated product terms on the output to deliver a total of 37 terms—and the T_{PD} is still 7.5 ns pin-to-pin. Thus, the Original XPLA mechanism minimizes waste by delivering logic where needed in precisely the quantity needed, at very high speed. The additional benefit of the programmable OR array is that the PLA product terms can be shared across multiple outputs. Consider Figure 11, where multiple macrocells share a common set of logic.

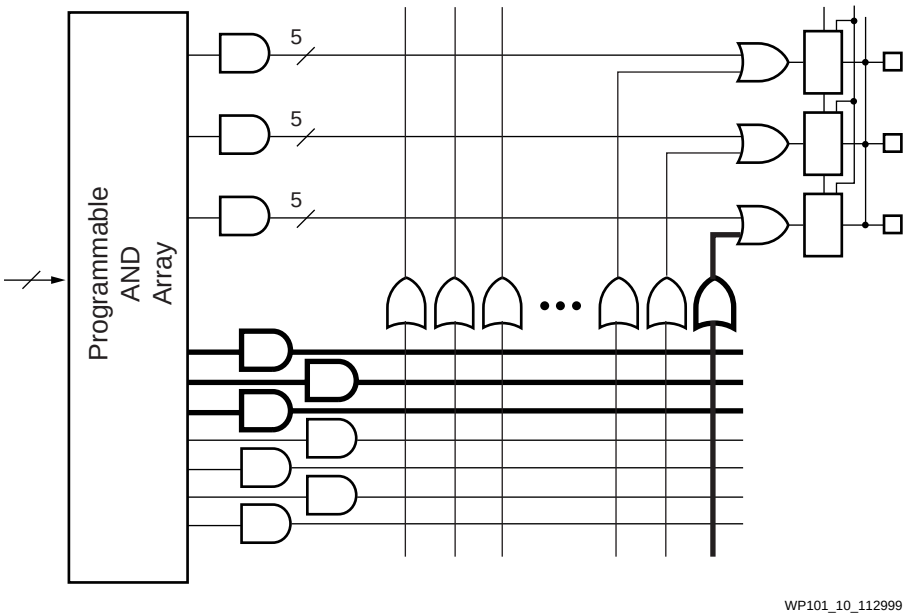

Figure 10: **Macrocell Needing Eight Product Terms**

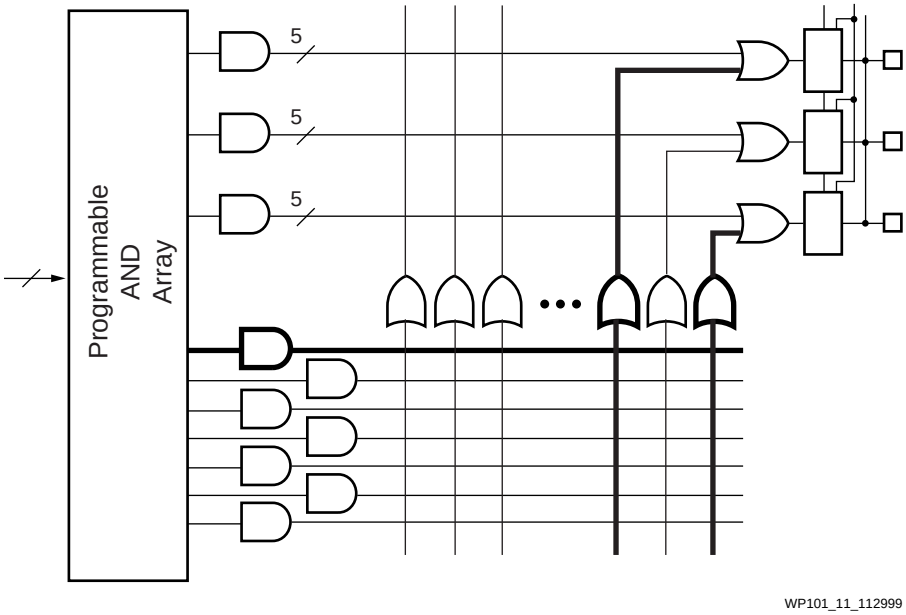

Figure 11: **Mutiple Macrocells Sharing a Common Set of Logic**

Via the OR array, this logic is made available to both macrocells from a single product term. This product sharing capability of the PLA increases the effective density of the device. Architectures without this fast sharing capability would have to implement this design by replicating the logic needed on terms at each macrocell.

As a final consideration in the area of logic allocation, consider the problem of refitting designs with fixed pinouts. With the 22V10, the fixed product term distribution can lead to problems in refitting where late design changes require more logic on an output than is available. This is true even if there are extra product terms available on other macrocells, because these terms cannot be dynamically reallocated to where they are needed. In the Original XPLA architecture, extra logic is dynamically added from the PLA array to any macrocell that needs it. Therefore, the only limitation in refitting designs is that there is sufficient remaining logic in the PLA array to implement the desired change.

Another way to state this is that the only limitation to refitting designs is the total capacity of the device—that is, refitting will always be successful as long as there are enough gates available to fit the logic. In product term steering, by contrast, a change that requires more logic on an output than remains in the cluster (which at best would be only three product terms) will not refit, as there is no donor logic left that will accommodate the existing PCB layout. With the Original XPLA architecture, 100% of the macrocells and associated PAL terms can be used because additional logic can be allocated from the PLA as needed. Therefore, unlike steering, as many as 32 terms are always available for additional logic utilization on any of the outputs.

Timing Model

With simple PLDs like the 22V10, determining whether the device would meet the required timing requirements was fairly simple. The critical specifications for this device were the pin-to-pin propagation delay (T_{PD}) for combinatorial applications, and the setup (T_{SU}) and clock-to-output (T_{CO}) time for registered applications. Internal logic feedback times were typically faster that external T_{SU}, so for most applications, these three specifications provided designers with all the information they needed to determine whether the device would be fast enough. As a result of the simplicity of this timing model, designers could also make reasonably accurate estimations of the performance of their designs before they began using the device.

As FPGAs became available, designers were at once attracted by the large number of gates these devices offered. However, they found that the timing of the finished device was sometimes slower than expected according to the specified flip-flop toggle rates. In fact,

performance not only seemed difficult to predict, it actually changed from one design iteration to the next. The fundamental problems that made device timing difficult were twofold: First, the timing from logic element to logic element varied with relative placement. Since the placement would change with design iterations, so would the timing. Second, the logic elements themselves typically had a small number of inputs—often less than eight—and designs that needed wide gating, like complex state machines, required multiple passes through logic elements. Each additional pass through logic elements could potentially cut performance in half. If a last-minute design change required an additional logic pass, the logic might fit, but the device performance might fall far below requirements.

With the first CPLDs, the timing situation was greatly improved relative to FPGAs. CPLDs offered a fixed, constant delay through the interconnect, regardless of where the logic had to be routed, so they escaped the FPGA placement-dependent timing problems. Unfortunately, they were still not as simple to use as the venerable 22V10. These first devices had quoted T_{PD} of 25 ns, which sounded fairly good at the time—but buried in the "fine print" was the fact that T_{PD} was measured from a dedicated input pin that bypassed the interconnect array. Since there were only four dedicated input pins on devices with 44 pins and up, this was equivalent to an automobile manufacturer quoting 200 "peak" miles per gallon when coasting downhill with a tailwind. As noted above, the interconnect array in these devices added 8 to 15 ns to the T_{PD}. Inside the logic block, the logic using the dedicated product terms offered the fastest performance. When additional logic was needed, however, foldback NANDs were used that added as much as 15 ns to the delay path.

The second generation of these devices offered faster interconnect performance by implementing a MUX-based interconnect, but still retained dedicated inputs faster than signals that propagated through the interconnect. In the logic block, the device retained the use of foldback NAND "shared expanders," but added product term steering that could incrementally steal logic from neighboring macrocells. This product term steering added 0.8 ns of delay for every cluster of product terms stolen, in addition to starving the donor macrocells. With these changes, the timing was now fragmented in many different options—fast, dedicated product terms, stolen "parallel expander" product terms that added delay in multiple increments as used, and foldback NANDs that were available to all outputs but were still quite slow. All these options presented the designer with "pay as you go" features that cost nanoseconds as the features were used. While the discussion here has focused on this one architecture, it is common to find devices whose timing diagrams seem complex enough to be the flow diagrams for controlling a nuclear reactor. Relative to the simplicity of the 22V10, it becomes quite difficult with all these different options to predict performance early in the design cycle.

The Original XPLA architecture timing model delivers a model that is almost identical to that of the 22V10. As seen in Figure 12, the Original XPLA timing model has only two variations from the 22V10 timing model. For combinatorial logic, there is a T_{PD} through the dedicated PAL product terms (T_{PD_PAL}), and a second T_{PD} specification for logic that uses both PAL and PLA product terms (T_{PD_PLA}). In registered applications, there is a setup time associated with logic that uses only the PAL terms (T_{SU_PAL}), and a setup time for logic that again uses both the PAL and PLA terms (T_{SU_PLA}). There is a single registered clock-to-output specification (T_{CO}). Thus, the only variation from the simplicity of the 22V10 timing model is the additional 2-ns path through the PLA array for implementing additional logic on an output. It is important to note that the PLA timing remains constant regardless of the number of PLA terms that are used—from one to 32 on an output—or the number that are shared by multiple outputs.

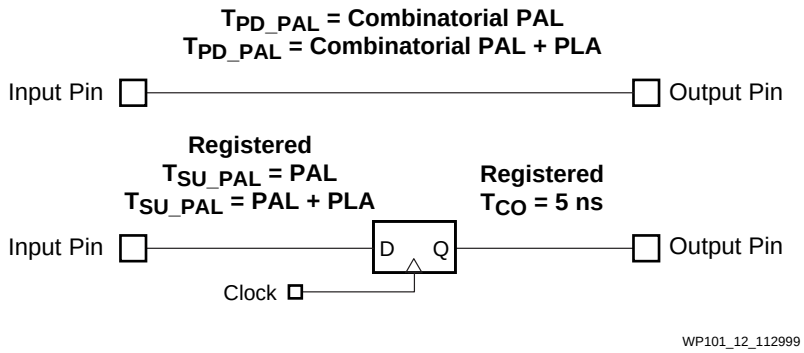

Figure 12: **XPLA Timing Model**

Conclusion

The Original XPLA architecture brings an interconnect methodology and logic allocation method that guarantees the ability to refit designs using 100% of the pins, macrocells, and logic in the device. The logic blocks offer sufficient width (36 inputs per block) to allow the development of highly complex state machines, an area where CPLDs typically excel. The unique combination of PAL and PLA arrays allows logic to be allocated on an as-needed, where-needed basis, without causing macrocell starvation. Simultaneously, this logic is allocated at a granularity of one product term across all macrocells, and can be shared resulting in the highest level of efficiency possible. Finally, the timing model for the device is simple and deterministic, allowing designers the ability to accurately predict design performance with tools no more complex that a pencil and paper napkin. These capabilities provide designers with high-speed, high-density devices which offer the ultimate in flexibility for making last minute design changes.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
09/01/99	1.0	Initial release.
04/07/00	1.1	Updated and reformatted into Xilinx format.
10/09/00	1.2	Added Discontinuation Notice.