XPLA Architecture White Paper

Mark Aaldering
Philips Semiconductors
Programmable Products Group Albuquerque, NM USA

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 Philips 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 pre-cursor 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, it's 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 Tpd and Tsu 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 pin-outs. 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 pin-outs. 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 65000 fuses. More significantly, these fully populated crosspoint switches were relatively slow - accounting for an 8 to 15ns 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 2 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.

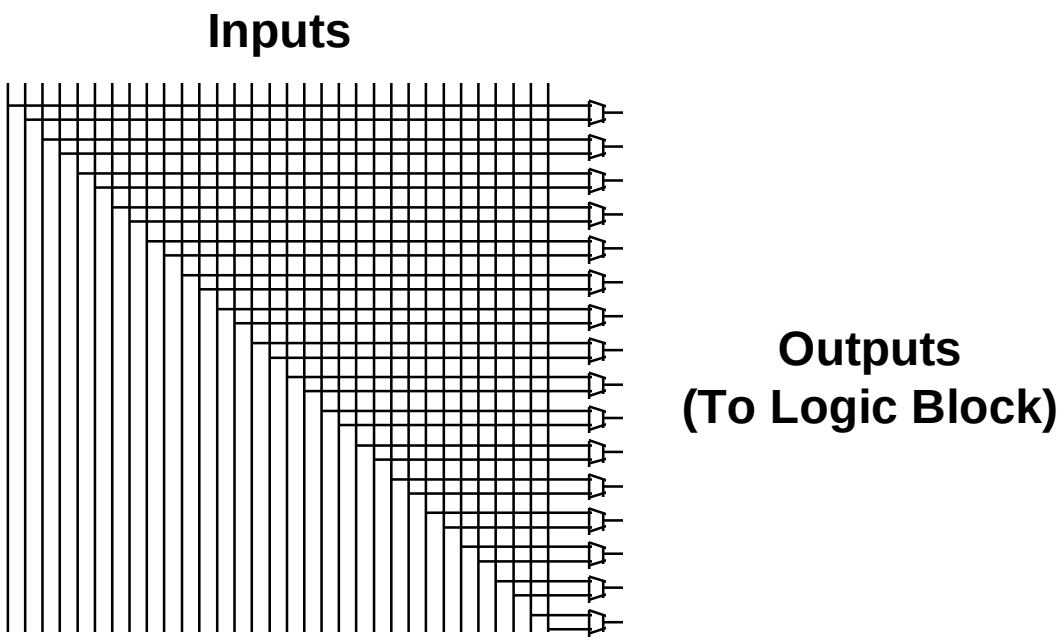

Figure 1

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 with has routed signals ACK and /IRQ connected to pins A an 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.

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 /IRQ signal. In order to add this feature, the engineer must re-layout his PCB. If this occurs on the Friday before Comdex, the engineer will not be happy about this change.

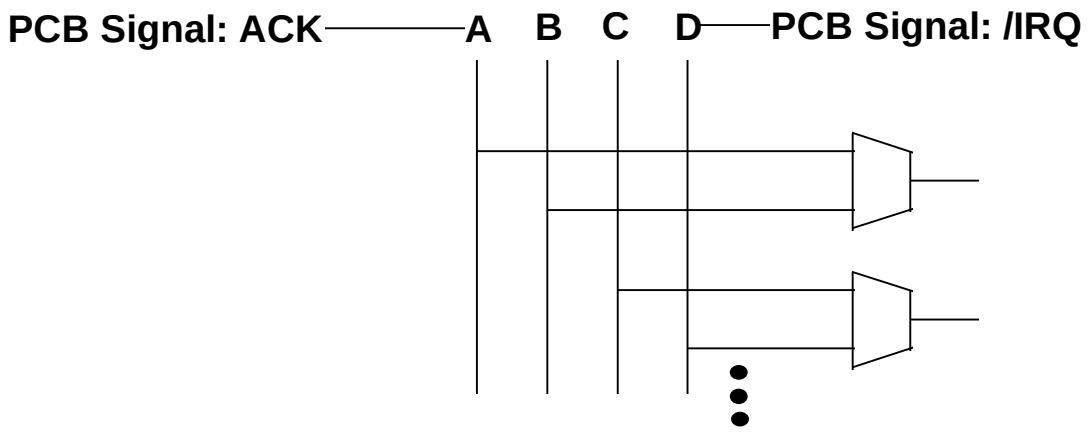

Figure 2

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

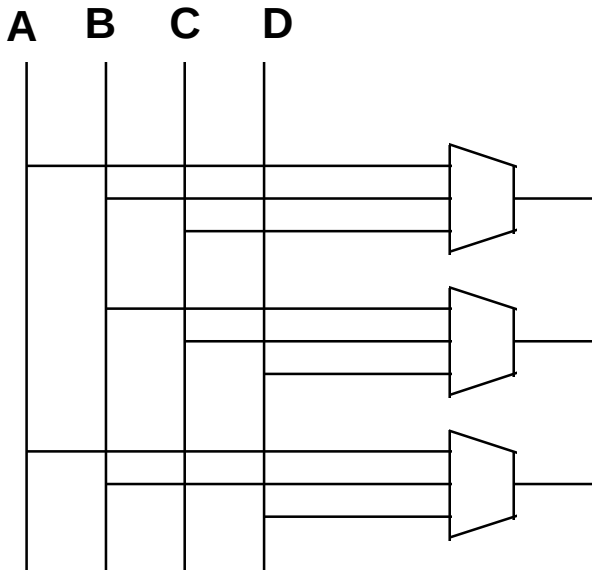

Figure 3

If in fact the muxes were 4 bits wide, this mux based interconnect would in fact logically emulate a crosspoint switch, but would require more E^2 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. Philips XPLA interconnect deploys a sufficiently large number of input muxes, of sufficient width, to guarantee routability under worst case conditions. At Philips, this interconnect architecture was subjected to over 16 Million

iterations of worst case fixed pin-out 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 pin-out. If only 35 of the 36 logic block inputs are used, 100% of the 16 million fixed signal routings completed successfully. We believe that this solution allows designers total freedom to make design iterations without the fear of having to re-layout his 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.

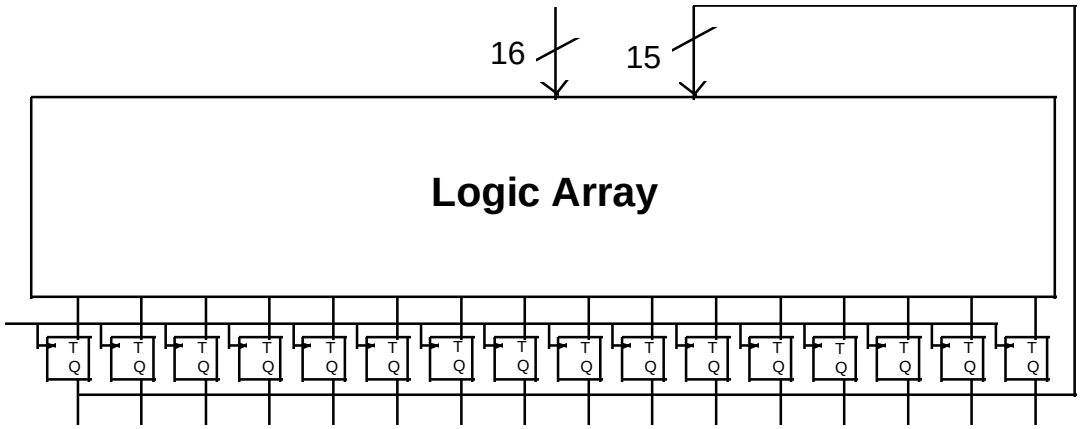

Figure 4

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 8. 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 8 product terms, another two that have 10 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 8 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 in advance that additional logic may be needed in the 11th hour of the design, many designers tried to implement designs that did not use all of the product terms available on an output. Having a few 'spare' product terms allowed small changes to be made without grief. Unfortunately small changes sometimes follow Murphy's law and become large design changes, and the amount of logic needed goes beyond what is available on the output. Once again, because the logic is allocated on a fixed rather than a dynamic basis, the designer is placed into a difficult situation. Thus having logic that has a fixed allocation scheme can result in a design that cannot easily undergo changes and refit into a fixed pinout. Therefore logic allocation methods are the second area where refitting fixed pinout designs are affected in PLDs of any size. The second issue with the 22V10 logic allocation method is the granularity with which product terms are distributed. Since the minimum number of product terms on any output is 8, what happens on logic that requires only one product term? Since the logic allocation is fixed, the other 7 product terms are wasted. PLD Synthesis experts we consulted with have stated that 70% to 80% of all designs deploy fewer than 5 product terms per output. In an architecture that has no fewer than 8 product terms per output, it is clear that there is considerable loss of logic in unused terms. Every engineer that has used this device knows that there are outputs that use only a few product terms. But at the same time most engineers have run into sticky problems that need more than the 16 product terms available in this device. Thus the fixed variable logic distribution present in the 22V10 attempts to minimize the loses that occur due to the granularity of the logic while still offering sufficient logic on specific outputs to perform complex logic.

Early CPLDs attempted to solve the problems of logic allocation by providing both a fixed set of product terms and a dynamic 'pool' of logic that could be used by all of the macrocells. These devices offered 3 dedicated product terms per output, citing statistics by the manufacturer that "80-90% of all logic implemented in typical designs could be achieved with 3 terms". To augment these dedicated product terms, they added an array of foldback NAND gates (also referred to as 'shared expander product terms'). The use of foldback NAND gates was previously pioneered (and patented) by Philips in a line of devices call Programmable Macro Logic, that could implement extremely wide gating functions (up to a 128 wide AND). In use in these CPLDs, the foldback NAND 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, and increase the register count. The basic problem with foldback NANDs (aside from patent violation issues...) is that the logic that uses them first must make a pass through the array into the foldback NAND, then be fed back into the array, then be AND'ed into the dedicated product terms. Thus two full passes through the array must be taken to implement the desired logic. Where the dedicated product terms could implement logic in 25ns, using the foldback NANDs would stretch this to 40ns. In addition to being relatively slow, foldback NAND structures were much more difficult to synthesize general purpose logic to. As a result much of their use was based on hard-coded macros to perform very specific functions.

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 when needed.

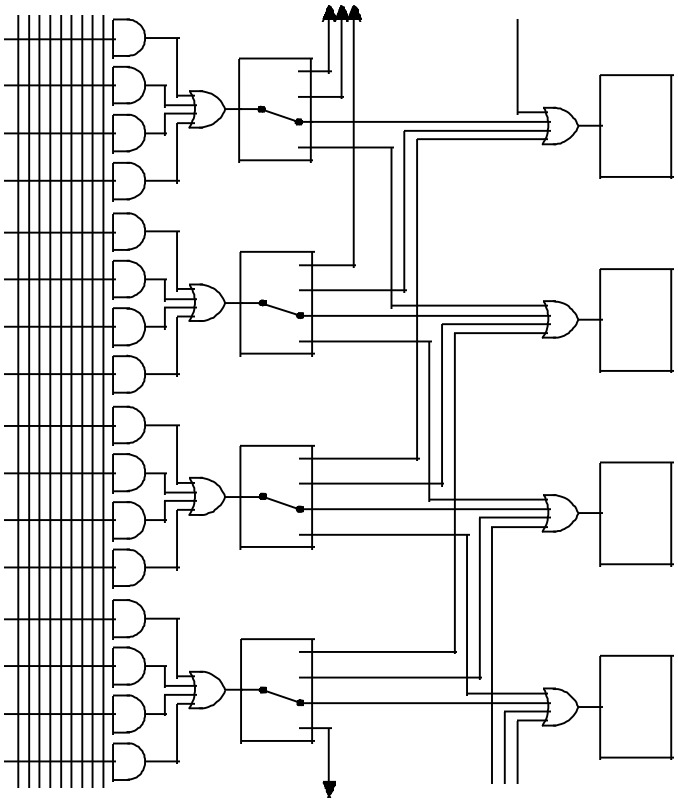

Figure 5

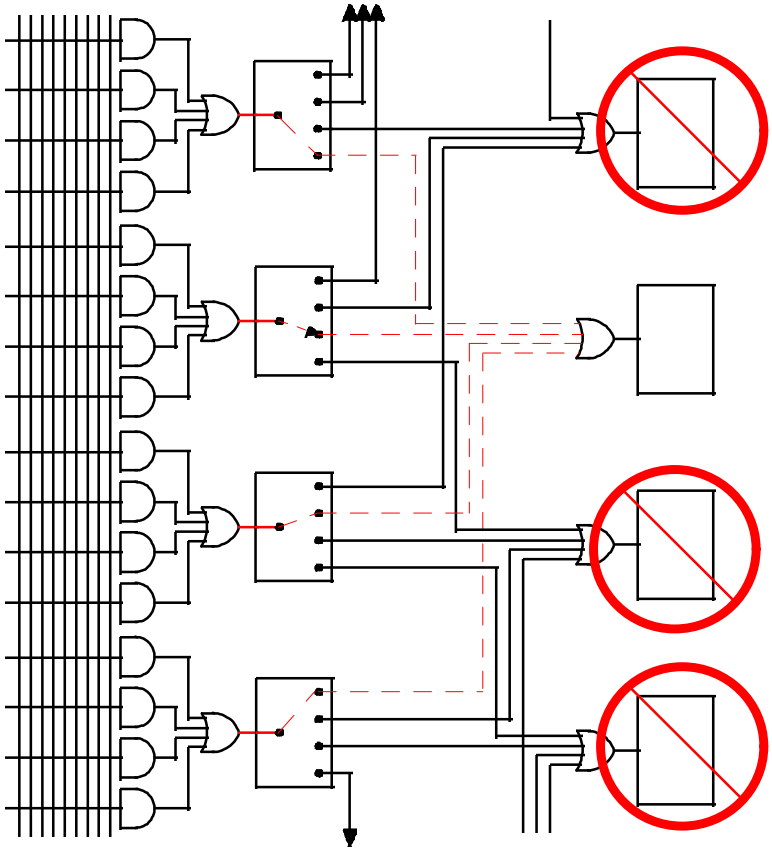

Figure 6

At first inspection in 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 whenever product terms are steered to a macrocell that needs more logic they 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 has 4 product terms routed to it in the native state. As shown, macrocell #2 has a complex equation associated with it that requires 16 product terms. To accomplish this, 3 neighbor macrocells switch their cluster of product terms to macrocell #2. At first glance, it is tempting to state that the logic steered away from the donor macrocells could be replaced with a cluster from some other macrocell. While this is true, what this ends up being is an elaborate shell game, as this next donor is left without logic. The net impact of macrocell #2 needing additional product terms is the loss of 3 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. The second issue with product term steering is the same problem encountered with the 22V10 - the issue of logic allocation granularity.Since the steering mechanism is deployed on groups of 4 product terms, some logic waste will occur when logic is needed that is not a multiple of 4. As an example, when an output requires 5 product terms as shown in Figure 7, only a single extra term is needed from the neighboring macrocell. Instead all four product terms are steered away, wasting 3 product terms that are not needed.

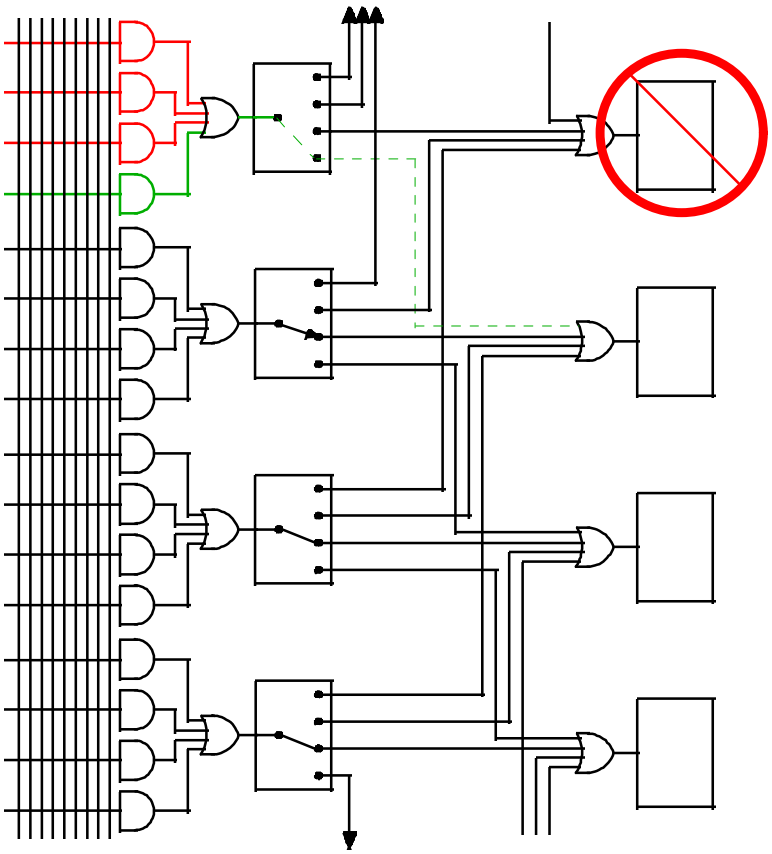

Figure 7

The final issue in product term steering is the ability to refit designs. As an example, let’s look at a design that is in the final stages of completion that has utilized all of the macrocells in the block, and being uncharacteristically well behaved uses just 3 product terms per output. Should marketing suddenly require an extra feature, the design tweak will inevitably require that one of these outputs have more than 4 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 that are hard wired to pins or employs an additional 'output routing pool'. As a result, designers that use these devices often learn through painful experience that with these architectures it is safest to leave a number of macrocells unused if refitting 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 after soldering to the board as not being able to refit when attempting to do a field update can be quite expensive.

Philips, when it was operating under the Signetics label for its PLD division, developed the first commercially available device with a programmable logic array (PLA) array in it's 82S100. This device offered a fully programmable AND array that delivered its' product terms to a fully programmable OR array. This Philips patented combination of programmable AND with a programmable OR array eliminates 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 that featured a programmable AND with a fixed OR (PAL). As a result, the XPLA logic allocation method uses a patent-pending structure that combines a PAL array and a PLA array - hence the term eXtended Programmable Logic Array.

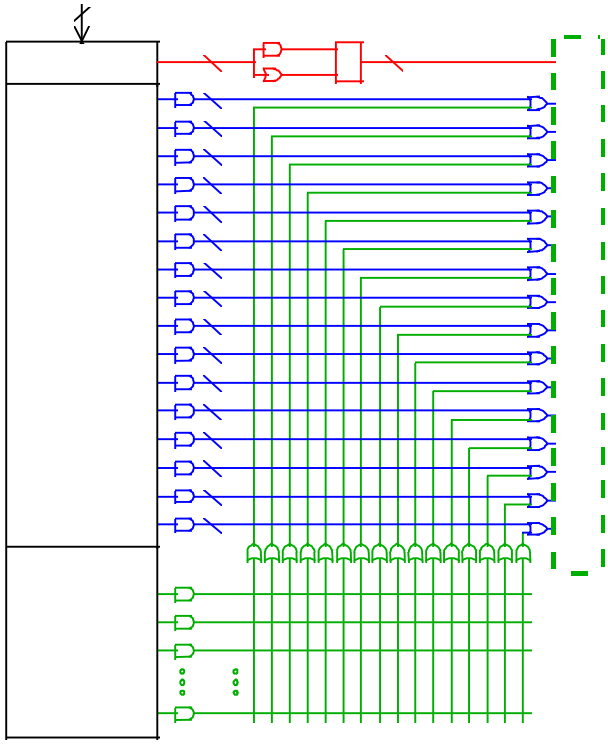

Figure 8

As detailed in Figure 8, the PAL array has 5 dedicated product terms for every output macrocell. These PAL terms are never steered, stolen, or folded back. Speeds from any pin to any pin through this PAL array section on 32 macrocell devices are estimated to be 6nS. 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 that requires 6 product terms uses its 5 dedicated terms, and one additional term from the PLA array to reach the required logic

needed. It is important to contrast that this logic was used without stealing logic from neighbor macrocells, and the additional logic was allocated exactly as needed - just one additional term, not four with three wasted. In the XPLA architecture, the cost of using this additional logic is a fixed 2nS delay. For the example above, the Tpd from any pin to the pin that is now using logic in the PAL + PLA is 8nS total.

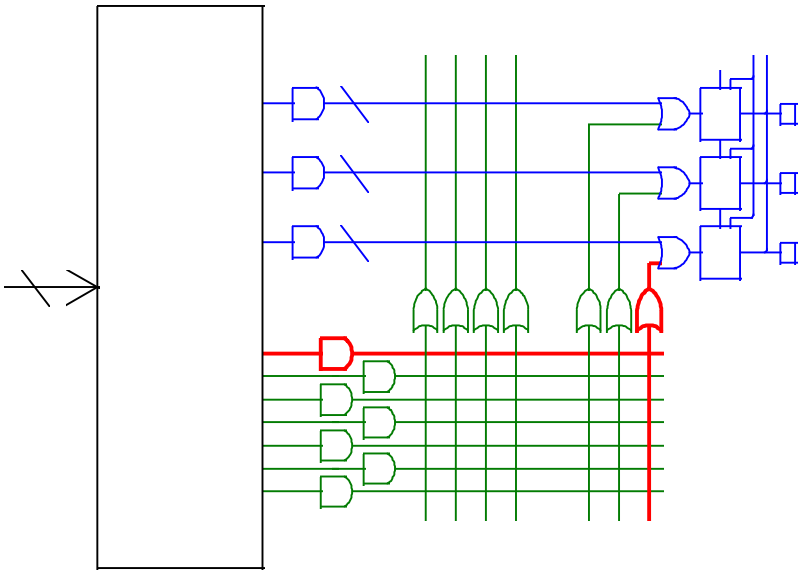

Figure 9

In figure 10, we see the case where 8 product terms are now needed on this macrocell. In this case, three product terms are used from the PLA in addition to the 5 dedicated PAL terms.

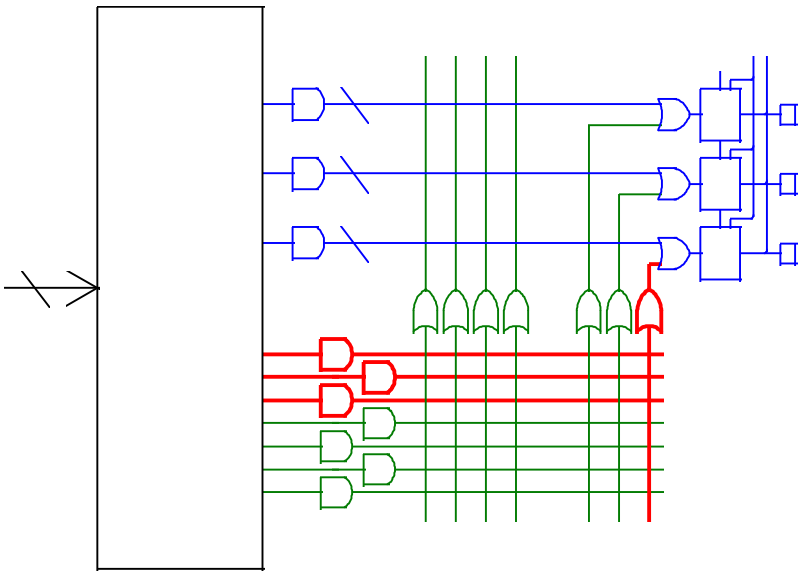

Figure 10

Once again, logic is utilized in the exact amount required, exactly where needed without sacrificing other macrocells. And the Tpd from pin to pin is still 8ns for this example. In fact, as many as all 32 PLA terms can be used on any output in conjunction with the 5 dedicated product terms on the output to deliver a total of 37 terms - and the Tpd is still 8nS pin to pin. Thus the XPLA mechanism delivers logic where needed in precisely the quantity needed to minimize waste 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 we see a case in which multiple macrocells share 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 that do not have this fast sharing capability would replicate the logic needed on terms at each macrocell to implement this design. As a final consideration in the area of logic allocation, lets consider the problem of refitting designs with fixed pinouts.

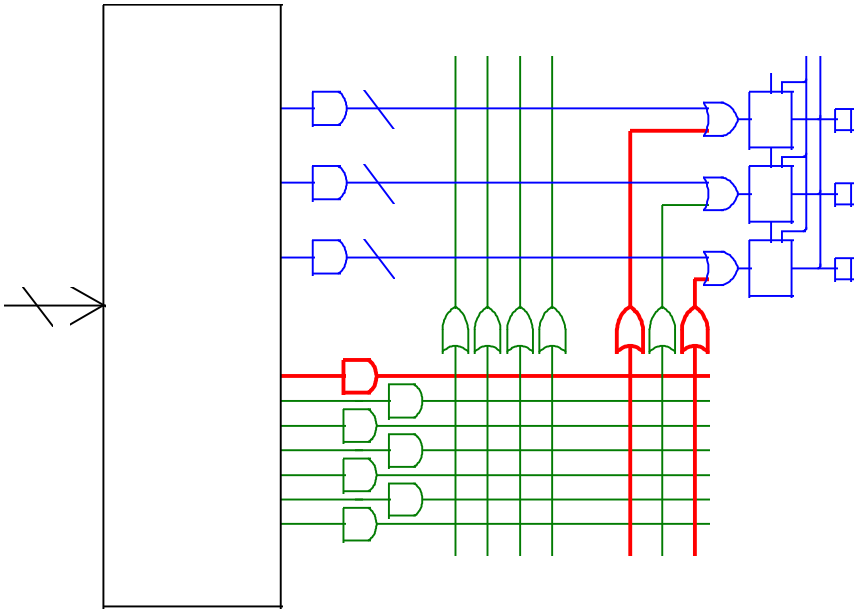

Figure 11

With the 22V10, the fixed product term distribution leads to problems in refitting where late design changes require more logic on an output than available. This is true even if there are extra product terms available on other macrocells, because these terms cannot be dynamically re-allocated to where they are needed. In the XPLA architecture, as extra logic is needed it is dynamically added from the PLA array to any macrocell that needs it. Therefore, the only limitation in refitting designs is quite simple - there must be 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 - refitting will always be successful as long as there are enough gates available to fit the logic. In product term steering, if all of the product term clusters or macrocells are being used, then a change that requires more logic on an output than is previously remaining in the cluster (which in the best case would be only 3 product terms) will not refit as there is no donor logic left that will accommodate the existing PCB layout. With the XPLA architecture, 100% of the macrocells and associated PAL terms can be used because additional logic can once again be allocated from the PLA as needed. Therefore, unlike steering, as many as 32 terms are always available for use for additional logic utilization on any of the outputs. As stated previously, the only limitation to refitting a design is the total capacity of the device.

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 are the propagation delay from pin to pin (Tpd) for combinatorial applications, and the set-up (Tsu) & clock to output time (Tco) for registered applications. Internal logic feedback times are typically faster that external Tsu, so for most designs, these three specifications provide designers with all the information they need to know to determine whether the device will 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 design before they began using the device.

As FPGAs became available, designers were attracted by the large number of available gates these devices offered. They soon found however that the timing of the finished device was no where near the flip-flop toggle rates touted by manufacturers. In fact, the designs' performance not only seemed difficult to predict, but would change from one design iteration to the next. The fundamental problems that made the timing of these devices difficult to deal with 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. The second issue is that the logic elements themselves typically have a small number of inputs - often less than 8 - and therefore designs that needed wide gating like complex state machines required multiple passes through the logic elements. Each additional pass through logic elements held the potential of cutting performance in half. If a last minute design change required an additional pass, the logic might fit, but now the performance could be far lower than needed.

With the first CPLDs, the situation for timing was greatly improved relative to FPGAs. They offered a fixed, constant delay through the interconnect regardless of where the logic had to be routed. Thus they escaped the FPGA problem of timing that depended on logic placement. Unfortunately, they were still not as simple to use as the venerable 22V10. These first devices had quoted Tpd's of 25nS, which sounded fairly good at the time. But buried in the fine print was the point that this Tpd was from a dedicated input pin that bypassed the interconnect array. Since there were only 4 dedicated input pins on devices that had 44 pins and up, this was equivalent to an automobile manufacturer quoting 200 'peak' miles per gallon when coasting downhill with a tail wind. As noted above, the interconnect array in these devices added from 8 to 15ns to the Tpd. Inside the logic block, the logic that used the dedicated product terms offered the fastest performance. When additional logic was needed, foldback NANDs were used that added as much as 15nS to the delay path.

The second generation of these devices offered faster interconnect perfor mance by transitioning to MUX based interconnect, but still retained dedicated inputs that are faster than signals that propagate 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 0.8nS of delay for every cluster of product terms that were stolen, in addition to starving the donor macrocells. With these changes, the timing was now fragmented in many different options - fast dedicated product terms, 'parallel expander' stolen 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 of these options presented the user with 'pay as you go' features that cost nanoseconds as you use them. While we have focused on one architecture, it is common to find devices whose timing diagrams seem complex enough to be the flow diagrams for controlling a nuclear reactor. With all of these different options, relative to the simplicity of the 22V10, it becomes quite difficult to predict the performance of design early in the cycle.

The XPLA architectures' timing model delivers a model that is almost identical to that of the 22V10's. As seen in Figure 12, the XPLA timing model has only two variations from the 22V10 timing model. For combinatorial logic, there is a Tpd through the dedicated PAL product terms (Tpd_pal), and a second Tpd specification for logic that uses both PAL and PLA product terms (Tpd_pla). In registered applications, there is a setup time associated with logic that uses only the PAL terms (Tsu_pal), and a setup time for logic that again uses both the PAL and PLA terms (Tsu_pla). There is a single registered clock to output specification (Tco). Thus the only variation from the simplicity of the 22V10 timing model is the additional 2nS path through the PLA array for implementing additional logic on an output.

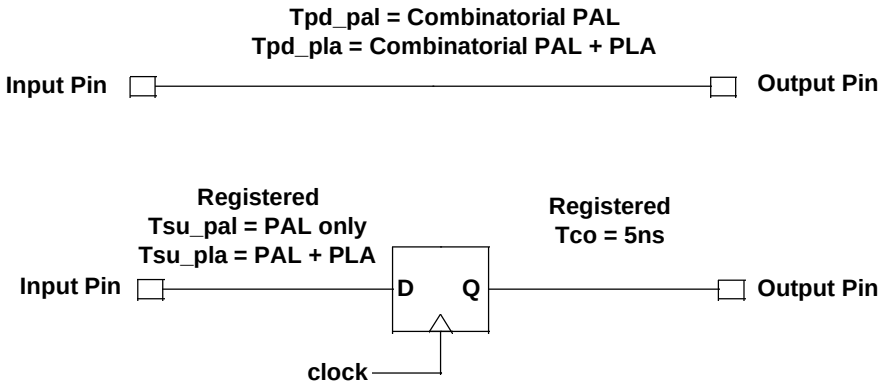

Figure 12

It is important to note that the PLA timing remains constant regardless of the number of PLA terms that are used - from 1 to 32 on an output - or the number that are shared by multiple outputs.

Conclusion

The XPLA architecture brings an interconnect methodology and logic allocation method that guarantees the ability to refit designs that use 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. Philips has long been a pioneer in developing and patenting structures used in logic arrays such as the PLA and foldback NAND structures. The unique XPLA 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 the highest level of efficiency possible. Finally, the timing model for the device is simple and deterministic, allowing designer the ability to accurately predict design performance with tools no more complex that a common pencil and paper napkin. The combination of these capabilities provides designers with devices with high speed, high density, and the ultimate in flexibility to make last minute changes to their designs.