

# Product-Term Based Synthesizable Embedded Programmable Logic Cores

Andy Yan and Steven J.E. Wilton  
Department of Electrical and Computer Engineering  
University of British Columbia, Vancouver, Canada  
{ayan, stevew}@ece.ubc.ca

## Abstract

*As integrated circuits become increasingly complex, the ability to make post-fabrication changes will become more important and attractive. This ability can be realized using programmable logic cores. Currently, such cores are available from vendors in the form of a "hard" layout. Previous work has suggested an alternative approach: vendors supply a synthesizable version of their programmable logic core and the integrated circuit designer synthesizes the programmable logic fabric using standard cells. This paper presents a new family of architectures for these synthesizable cores; unlike previous architectures which were based on lookup-tables, the new family of architectures is based on a collection of product-term arrays. Compared to lookup-table based architectures, the new architectures result in density improvements of 35% and speed improvements of 72% on standard benchmark circuits.*

## 1. Introduction

Recent years have seen an impressive improvement in the achievable density of integrated circuits. In order to utilize this excess capacity, while maintaining reasonable design costs, the System-on-a-Chip (SoC) design methodology has emerged. In this methodology, pre-designed and pre-verified blocks, often called cores, are obtained from internal sources or third-parties, and combined onto a single chip. Although this technique partially alleviates some of the complexity, the design and test of a correctly-functioning integrated circuit is still a difficult task.

One way to ease this task is to use embedded programmable logic cores. A programmable logic core is a flexible logic fabric that can be customized to implement any digital circuit after fabrication [1-5]. Before fabrication, the designer embeds a programmable fabric (consisting of many uncommitted gates and programmable interconnects between the gates) onto an SoC. After fabrication, the designer can then program these gates and the connections between them. This technique is attractive for a number of reasons. In some cases, it may be possible that some design details can be left until late in the design

cycle. In a communications application, for example, the development of a chip can proceed while standards are being finalized. Once the standards are set, they can be incorporated into the programmable portion of the chip. A second reason that the use of programmable logic cores is attractive is when products are upgraded or as standards change, it may be possible to incorporate these changes using the programmable part of the chip, without fabricating an entirely new device. Finally, the use of a programmable logic core may make it possible to fabricate a single chip for an entire family of devices. The characteristics that differentiate each member of the family can be implemented using the programmable logic. This would amortize the cost of developing the ASIC over several products. Several integrated circuits containing programmable logic cores have been described [6-8].

Despite these compelling advantages, the use of programmable logic cores has not become mainstream. In fact, many companies that develop these cores have either changed focus or gone out of business. There are a number of reasons for this. One reason is that designers often find it difficult to identify a subsystem that can be implemented in programmable logic. A second reason is that embedding a core with an unknown function makes timing, power distribution, and verification difficult. Lastly, embedded programmable logic cores must address physical connection and placement issues. This is difficult when the regions of fixed and programmable logic are tightly coupled together and or when there are a number of small programmable pieces distributed over the entire SoC. This extra design complexity limits the use of programmable logic cores to only the very best VLSI designers.

In [9], an alternative technique is described which addresses the last two concerns. In this technique, core vendors supply a synthesizable version of their programmable logic core (a "soft" core) and the integrated circuit designer synthesizes the programmable logic fabric using standard cells. A "soft core" is one in which the designer obtains a description of the behaviour of the core, written in a hardware description language. Note that this is distinct from the behaviour of the circuit to be implemented in the core, which is determined after

fabrication. Here, we are referring to the behaviour of the programmable logic core itself.

Since the designer receives only a description of the behaviour of the core, he or she must use synthesis tools to map the behaviour to gates. These synthesis tools can be the same ones that are used to synthesize the fixed (ASIC) portions of the chip. The primary advantage of the new method is that existing ASIC tools can be used to implement the chip. No modifications to the tools are required, and the flow follows a standard integrated circuit design flow that designers are familiar with. This will significantly reduce the design time of chips containing these cores. A second advantage is that this technique allows small blocks of programmable logic to be positioned very close to the fixed logic that connects to the programmable logic. The use of a “hard core”, however, requires that all the programmable logic be grouped into a small number of relatively large blocks. A third advantage is that the new technique allows users to customize the programmable logic core to support his or her needs precisely. This is because the description of the behaviour of the programmable logic core is a text file which can be edited and understood by the user. Finally, it is easy to migrate the circuit to new technologies; new programmable logic cores from the core vendors are not required.

The primary disadvantage of the proposed technique is that the area, power, and speed overhead will be significantly increased, compared to implementing programmable logic using a hard core. Thus, for large amounts of circuitry, this technique would not be suitable. It only makes sense if the amount of programmable logic required is small. An envisaged application might be the next state logic in a state machine.

In this paper, we present a new family of architectures for a synthesizable embedded programmable logic core (PLC). Compared to the architecture in [9], the new architecture is significantly more area and speed efficient. Unlike the architectures in [9], which are based on lookup-tables (LUT's), our new family of architectures is based on a collection of product-term array blocks. It is well known that product-term array blocks can result in density and speed improvements for small size circuits [10]; in this paper, we show that the small combinational circuits envisaged for these synthesizable cores are very suitable for product-term based architectures. In addition, this paper shows that the nature of synthesizers and synthesized circuits places unique demands on a product-term based architecture.

This paper is organized as follows. Section 2 describes the new architecture family, and shows how it differs from standard commercial product-term based architectures. Section 3 then describes the experimental results aimed at optimizing several architectural parameters. Finally, Section 4 compares the new architecture to the LUT-based architecture from [9].

## 2. Synthesizable Product-Term Based Architecture Family

In this section, we describe our family of product-term based architectures. Each member of the family is composed of one or more product term-based blocks (PTB's) connected using a novel interconnect architecture. The number of PTB's, as well as the number of input and output pins, vary between members of the family.

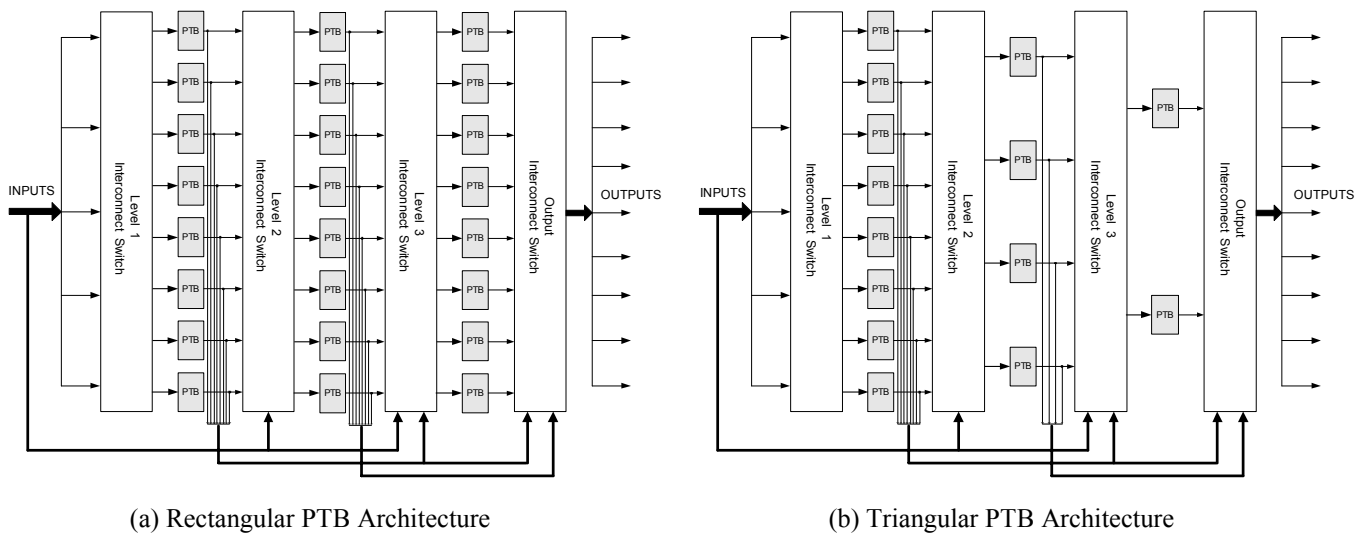


Figure 1: Product-Term Block Architectures

### Basic Product Term-Based Block Architecture

Product term-based blocks (PTB's) are essentially circuits that can implement any Boolean function in a sum-of-product form. A PTB consists of two planes – the AND plane and the OR plane. The AND plane is a product term generator; it is used to create products terms that can be fed into the OR plane. The OR plane is used to “sum” the product terms to create the desired Boolean function. Clearly, the size of a PTB can be scaled by altering the number of primary inputs, the number of product terms (the number of AND gates in the AND array) and the number of outputs (which is often equal to the number of OR gates in the OR array). In this paper, we will refer to the size of a PTB using the tuple  $(i,p,o)$  where  $i$  is the number of inputs,  $p$  is the number of product terms (p-terms), and  $o$  is the number of outputs. Unlike commercial PTB based architectures, our PTB's do not contain registers on their outputs. As in [9], we are targeting small combinational circuits (such as the next state logic in a state machine). Registers can be attached to the periphery of the core if desired. We focus on PLA-type logic cores (which we call PTB's) instead of PAL-type logic cores because their flexibility allows for more efficient logic implementation and because of available PLA-based technology mapping algorithms.

A programmable logic core can be implemented using a single PTB. The behaviour of the single PTB can be written in a hardware description language, synthesized, and combined with the rest of the integrated circuit. This works well for small cores, but as the number of inputs,

product terms, and outputs grow, the size of the synthesized fabric will grow. The number of programming bits (which would be implemented using flip-flops in the synthesized fabric) is proportional to the sum of the product of the number of inputs and product terms, and the product of the number of product terms and outputs. For large cores, this becomes unwieldy. Because of this, large product-term based devices usually contain a collection of smaller PTB's connected using a very flexible interconnect switch matrix. For example, Altera and Xilinx Complex Programmable Logic Devices (CPLD's) contain a number of PTB's that surround a central global interconnect switch matrix [12-13].

### Interconnect Architecture

Unfortunately, the single global interconnect switch matrix found in commercial devices is not appropriate for a synthesizable architecture. This is because it allows PTB outputs to be connected to any other PTB input (or a large fraction of them). This can create combinational loops in the unprogrammed fabric (for example, if the output of a PTB is connected to the input of the same PTB). These combinational loops are normally not a problem, since it is up to the user to configure the fabric in such a way that these combinational loops do not occur. In our case, however, we wish to synthesize the fabric itself using standard synthesis tools. Standard synthesis tools have problems synthesizing circuits with combinational loops. Thus, we need a fabric without combinational loops.

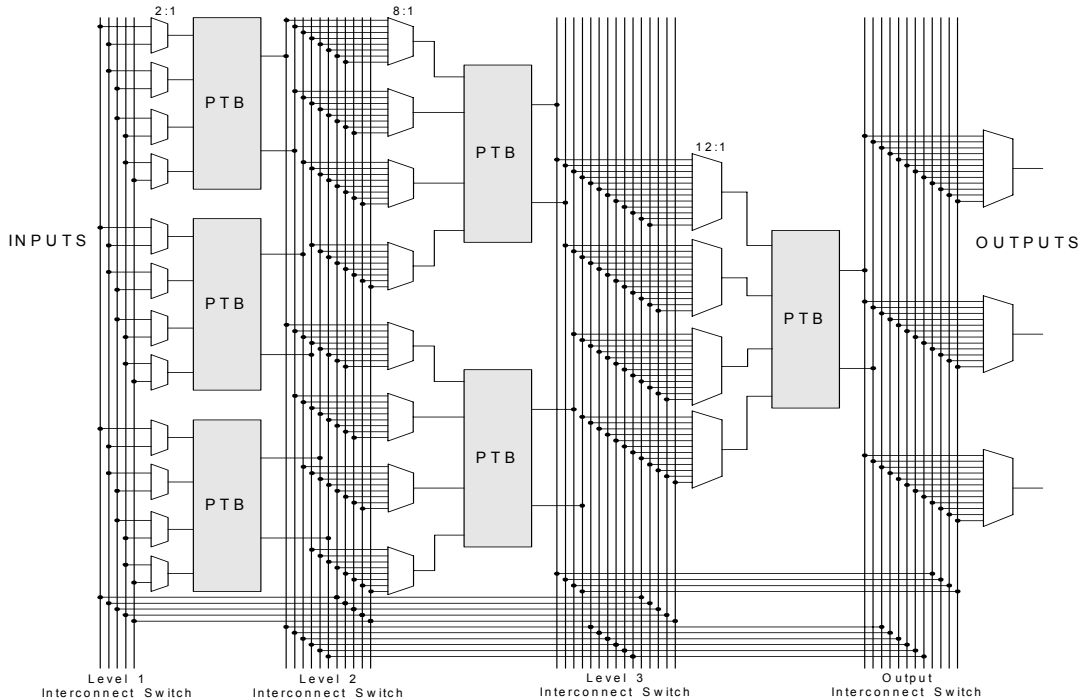


Figure 2: Detailed view of Routing Fabric

Figure 1 shows our novel routing architecture that can be used to connect PTB's. We have considered two interconnect strategies. In the first strategy, in Figure 1(a), the PTB's are arranged in a rectangular grid, while in the second strategy, in Figure 1(b), the PTB's are arranged in a triangular shape. In both cases, each core contains PTB's arranged in several levels. The outputs of PTB's in one level can be connected to the inputs in all subsequent levels (levels to the right) but can not drive any PTB's in the same level, or preceding levels (levels to the left). This results in a directional architecture, and eliminates the possibility of combinational loops, as described above.

Figure 2 shows a detailed view of our proposed directional routing architecture. In place of a central single interconnect matrix, we use multiplexors at each input of the PTB block. The multiplexors allow the PTB inputs to connect to the outputs of all PTB's in previous levels, as well as the primary inputs. We are able to reduce the size of the multiplexors, by taking advantage of the fact that all input connections to a PTB block are equivalent. For a circuit with  $n$  primary inputs, to be mapped into a  $(i,p,o)$  PTB block, the size of a multiplexor at level  $l$  is computed as:

$$Mux\ Size_l = n + \sum_{s=1}^{s=l-1} C_s \cdot o - (i-1)$$

where  $C_s$  denotes the number of PLA blocks at level  $s$ . The size of a routing multiplexor is calculated by summing the number of primary inputs with the number of signal outputs from the PTB's in the preceding levels, and subtracting by the number of inputs in a PTB block.

Allowing a "full-connect" fabric may not seem to be very area efficient, especially since the multiplexors grow in

size with the depth of the core. Because of this, [9] suggested a sparsely populated routing fabric used in their LUT-based architecture. However, in our case, the number of PTB blocks and the depth of the fabric are comparatively small, meaning the size of the multiplexors do not become unwieldy. An advantage of a fully connected architecture is that the placement and routing tasks become trivial.

#### Architectural Parameters

There are two classes of parameters we use to describe a specific design within our architectural family: high-level parameters and low-level parameters. Consider a VLSI designer who wishes to employ one of our cores. The designer would have a rough idea of how much logic should fit in the core (and hence, the size of the core in terms of number of LUT's or PTB's) as well as the number of inputs and outputs of the core. The designer would use these quantities, which we refer to as high-level parameters, to choose a specific core from a library or a core generator. The high-level parameters are summarized in Table 1.

Low-level parameters, on the other hand, would not normally be specified by the VLSI designer. These parameters describe the details of the core layout (size of each product-term block, details of the interconnect between the block, etc.). The designer of the core library itself (as opposed to the VLSI designer who uses the library) would like to use optimum values for these parameters in the design of each core in the library. The low-level parameters for both rectangular and triangular cores are listed in Table 2; Section 3 will seek optimum values for these parameters.

Parameter	Symbol
Number of Primary Input Pins	$PI$
Number of Primary Output Pins	$PO$
Number of PTB's in the first level	$y$

Table 1: High-Level Architectural Parameters that are used by the VLSI Designer to identify a core in a library

Parameter	Symbol	Range in Section 3
Inputs per PTB	$i$	12
P-terms per PTB	$p$	6-21
Outputs per PTB	$o$	3
Ratio of levels to number of PTB's in first level	$r$	0.2-1.0

(a) Parameters for rectangular cores

Parameter	Symbol	Range in Section 3
Inputs per PTB	$i$	12
P-terms per PTB	$p$	6-21
Outputs per PTB	$o$	3
Ratio of PTB's in neighbouring levels	$\alpha$	0.2-0.8

(b) Parameters for triangular cores

Table 2: Low-Level Parameters that describe each architecture in the library

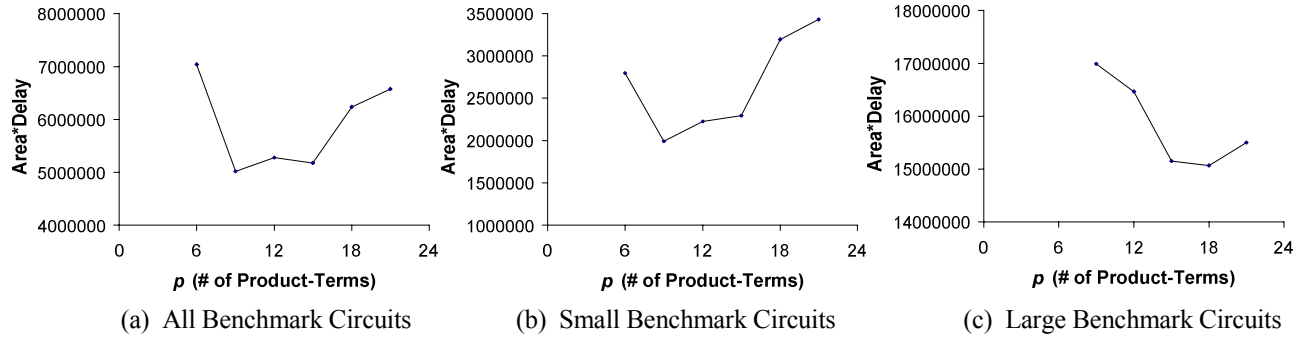


Figure 3: Area\*Delay as a function of the number of product terms per PTB

### 3. Low-Level Parameter Optimization

In this section, we seek to find optimum values of the low-level parameters in Table 2. We do not attempt to find optimum values for the high-level parameters in Table 1, since these are parameters that would usually be determined by the VLSI designer depending on the applications expected to be mapped to the programmable logic core. Although we have not investigated all possible combinations of values for the parameters in Table 2, we have varied three of the key parameters ( $p$ ,  $r$ , and  $\alpha$  to be explained in the following section), and have identified their impact on the area and delay of the resulting programmable logic core. In addition, this section will answer the question of whether a rectangular or triangular core is more area and delay efficient.

#### Number of Product Terms per PTB:

We used combinational MCNC benchmark circuits as the basis of our experiments. Their sizes range from 10 to 300 equivalent 4-LUT's with primary inputs and outputs ranging from 4 to 200 and 1 to 70, respectively. To limit our optimization space we initially fixed PTB input,  $i$  to 12 and output,  $o$  to 3; these values are in line with results from previous work [10,14]. We choose to optimize the number of product terms first, instead of the number of inputs or outputs in a PTB as during experimentation product terms account for a large amount of total PTB area and was found to greatly influence the utilization efficiency of the PTB blocks [14]. After obtaining the optimal number of PTB product terms, we focused on finding the optimal values of PTB inputs and outputs. Experimentally, we confirmed that a PTB with 12 inputs and 3 outputs is a good choice.

To find the optimal number of PTB product terms,  $p$ , we did an experimental parameter sweep ranging from 6 to 21 terms. We mapped each benchmark circuit to PTB's using PLAmapping [11], and chose the minimum triangular core size (with  $\alpha=0.5$ ) into which the circuit would fit. A behavioural description of each core was generated, and synthesized using Synopsys Design Compiler with the Virtual Silicon 0.18 $\mu$ m library.

For each value of  $p$ , we measured the area and delay after synthesis, and plotted the geometric average of the area\*delay product in Figure 3(a). The best value for  $p$  is 9. However, we have observed small circuits tend to prefer a smaller  $p$ , while larger circuits tend to prefer a larger  $p$ . We repeated our experiments, but partitioned the benchmark circuits into two sets – one set for small circuits (less than 50 equivalent 4-LUT's) and the other set for larger circuits. The results are shown in Figures 3(b) and 3(c). We see small circuits prefer  $p=9$ , with  $p=12$  or 15 resulting in a relative difference of 12-13%. Breaking down the graph into area and delay components (not shown),  $p=12$  or 15 results in a 23% degradation in area compared to 9 product terms. For larger circuits, we see  $p=18$  or 15 provides the best area and delay results. When  $p=9$ , we get a 11% increase in area-delay product. As a result, we propose that cores aimed at small circuits use PTB's with  $p=9$ , and cores aimed at larger circuits use PTB's with  $p=18$ . This combination results in a 5% improvement in area-delay product than would be obtained by just setting  $p$  to 9 for all cores.

#### Value of $r$ for rectangular cores

To find the optimum value of  $r$  (the ratio of the number of levels to the number of PTB's in each level) for rectangular cores, we used a similar procedure. The same benchmark circuits were used, and PTB's of either (12,9,3) or (12,18,3) were used (depending on the core size, as described above). Figure 4 shows the impact of this parameter on area, circuit depth, and area\*depth averaged over our benchmark circuits (again, geometric average is used). We have used depth instead of estimated delay in these circuits; since we are using a fixed-size PTB, these quantities are well correlated. As the graph shows, as  $r$  increases, the number of levels in the core increases, leading to a longer delay. On the other hand, the area decreases as  $r$  increases. A shallow core (small  $r$ ) places more constraints on the placement of logic, since more stringent precedence relationships must be obeyed (recall, each PTB can drive PTB's in subsequent levels only). Overall, a value of  $r=0.4$  gives the best area\*depth result.

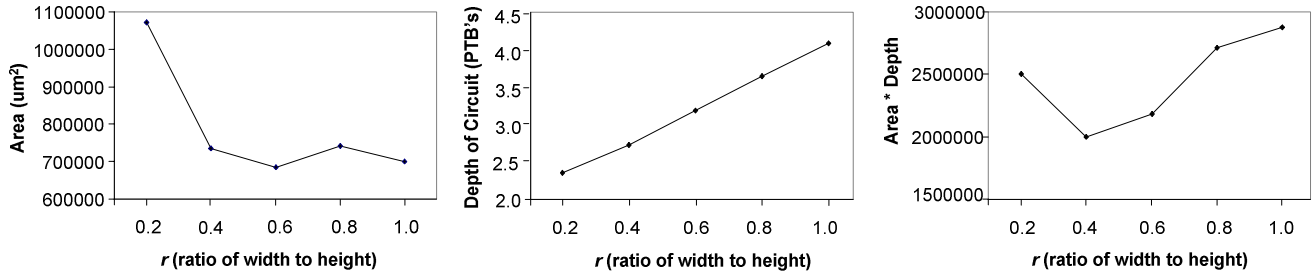


Figure 4: Area, Depth, Area\*Depth results for a rectangular core

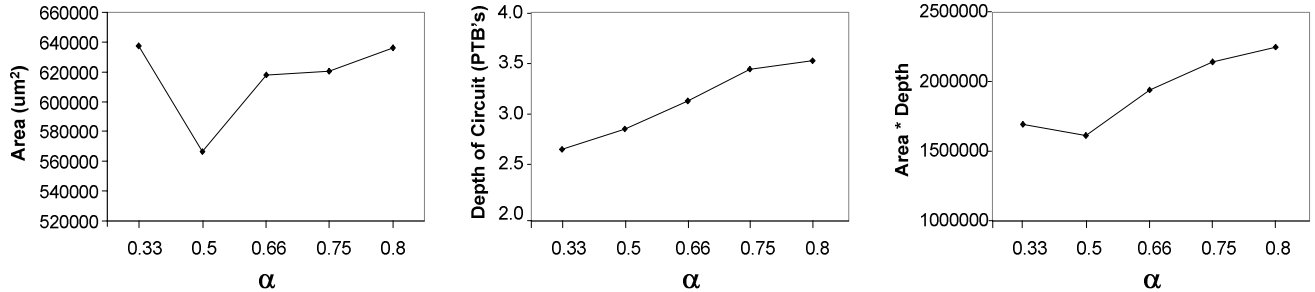


Figure 5: Area, Depth, Area\*Depth results for a triangular core

#### Value of $\alpha$ for triangular cores

We next investigate the effect  $\alpha$  has on triangular cores.  $\alpha$  is defined as the amount of drop-off in the number of PTB's in neighbouring levels. Given a triangular core that has  $y$  PTB's in the first level, there are  $n_i = \alpha^{i-1}y$  (rounded to nearest integer) PTB's in the  $i^{\text{th}}$  level. The number of levels in a triangular core depends on  $\alpha$  indirectly. All triangular cores have a minimum of 2 levels. For  $y > 3$ , there are  $x$  levels, where  $x$  is the smallest value for which  $n_x$  is less than or equal to 3.

Figure 5 shows the impact of  $\alpha$  on area, depth, and area\*depth of our benchmark circuits. As the graphs show, a value of  $\alpha=0.5$  is a good choice. We see that both delay and area increases for  $\alpha$  larger than 0.5. A large  $\alpha$  implies a larger depth and more PTB's, thereby increasing both delay and area. When  $\alpha$  is less than 0.5, mapping depth decreases, but area increases. This is because there are not enough levels to map the benchmark circuits for small  $\alpha$ , so thus the number of PTB's in the first level must increase to provide the required mapping depth.

#### Comparison of triangular and rectangular cores:

Comparing the results from Figures 4 and 5, we can see that the best triangular core leads to 23% smaller area but a 5.2% larger depth (and hence delay) than a rectangular core, on average. The best area\*delay product is 19% lower in a triangular core than in a rectangular core. Thus, we use triangular cores with  $\alpha=0.5$  in the remainder of this paper.

## 4. Comparison to LUT-based Architectures

In this section, we compare the area and delay efficiency of our synthesizable core with that of the LUT-based architecture in [9]. Based on the results of Section 3, we used a triangular core with  $\alpha=0.5$  and PTB input  $i=12$ , product terms  $p=9$  or 18, and output  $o=3$ .

The results are shown in Table 3. Overall, the PTB-based architecture is 35% smaller and 72% faster than the LUT-based architecture. In the architecture of [9], most of the area and delay was due to large routing multiplexors. In our architecture, since we have larger, and hence fewer, product-term blocks, the routing fabric is simpler and faster. This result is explained in Figure 6.

Figure 6 shows area and delay as a function of the number of logic blocks (LUT's or PTB's) in the first level. The results show that the area of the LUT-based architecture increases at a faster rate than the area of the PTB-based architecture. This is because there are more logic blocks in the LUT architecture than PTB's in the product-term architecture for a given circuit. Fewer logic blocks means fewer routing multiplexors are required, and thus, area increases more slowly. We see a similar trend with delay; because the product-term architecture uses larger blocks, fewer logic levels are required compared to the LUT-based architecture.

From the results in Table 3, we observe that larger circuits tend to result in higher area improvements. This is because the LUT-based architecture in [9] contains large

Circuit	Area ( $\mu\text{m}^2$ )			Delay (ns)		
	LUT architecture	Product-term architecture	Improvement	LUT architecture	Product-term architecture	Improvement
cm138a	79 450	74 766	5.9 %	20.9	5.7	73.0 %
cm42a	79 670	74 327	6.7 %	20.9	5.7	73.0 %
cm163a	80 365	91 757	-14.2 %	20.9	9.5	54.9 %
cm85a	109 995	81 097	26.3 %	26.3	7.9	69.8 %
rd53	107 263	117 784	-9.8 %	26.3	8.0	69.6 %
cm150a	216 144	97 668	54.8 %	34.4	11.4	66.9 %
mux	302 770	97 668	67.7 %	41.5	11.4	72.5 %
cmb	163 693	98 794	39.6 %	32.8	11.4	65.2 %
x2	114 248	81 092	29.0 %	26.3	6.5	75.2 %
cm162a	80 023	88 244	-10.3 %	20.9	10.5	50.0 %
pml	122 989	115 882	5.8 %	26.3	10.0	62.1 %
decod	85 597	118 772	-38.8 %	20.9	8.0	61.8 %
cc	537 513	210 651	60.8 %	46.3	11.7	74.8 %
il	179 248	163 153	9.0 %	32.8	11.5	65.1 %
misex1	163 588	78 751	51.9 %	32.8	7.4	77.3 %
pcl	120 500	121 407	-0.8 %	26.3	9.0	65.8 %
sct	180 436	230 886	-28.0 %	32.8	10.0	69.5 %
cu	228 576	112 613	50.7 %	34.4	9.8	71.4 %
sqrt8	300 204	121 122	59.7 %	41.5	8.6	79.2 %
sqrt8ml	377 658	187 347	50.4 %	45.7	9.3	79.6 %
lal	340 068	385 927	-13.5 %	41.5	18.0	56.5 %
squar5	218 648	74 766	65.8 %	34.4	5.7	83.6 %
ldd	331 367	170 068	48.7 %	41.5	9.5	77.1 %
pcler8	336 547	384 407	-14.2 %	41.5	20.6	50.4 %
c8	1 124 829	398 006	64.6 %	66.5	17.1	74.4 %
count	421 721	577 258	-36.9 %	45.7	19.8	56.8 %
comp	496 800	399 112	19.7 %	46.3	19.8	57.6 %
b9	817 867	470 760	42.4 %	62.6	18.5	70.5 %
unreg	339 868	182 660	46.3 %	41.5	12.2	70.5 %
frgl	1 062 808	386 761	63.6 %	66.5	17.0	74.4 %
misex2	420 493	220 726	47.5 %	45.7	11.6	74.6 %
f51m	749 837	232 024	69.1 %	62.6	9.8	84.3 %
cht	3 139 111	690 038	78.0 %	112.2	22.4	80.1 %
b12	902 504	162 551	82.0 %	67.7	12.8	81.1 %
5xp1	756 187	191 364	74.7 %	62.6	7.3	88.3 %
inc	388 765	149 277	61.6 %	45.7	6.7	85.3 %
vg2	507 168	613 361	-20.9 %	46.3	21.4	53.8 %
ttt2	1 300 072	618 105	52.5 %	77.4	21.5	72.2 %
rd73	732 367	534 130	27.1 %	62.6	14.9	76.2 %
term1	2 210 396	931 829	57.8 %	100.4	18.9	81.1 %
9symml	3 266 584	349 458	89.3 %	117.3	10.3	91.2 %
apex7	3 460 155	1 135 306	67.2 %	117.3	19.3	83.6 %
bw	1 556 027	540 700	65.3 %	85.3	16.8	80.3 %
clip	2 662 875	537 040	79.8 %	107.1	16.7	84.4 %
rd84	1 923 606	1 226 707	36.2 %	94.2	21.6	77.1 %
alu2	2 394 192	1 345 000	43.8 %	102.0	21.6	78.8 %
Average	771 539	331 981	35.0 %	52.3	12.9	72.2 %
Geomean	411 950	228 084		45.8	11.9	

Table 4: Comparison to a LUT-based architecture

routing multiplexors; the sizes of these multiplexors grow as the core increases. Although our multiplexors also grow as the core size increases, our multiplexors form a far smaller portion of the overall fabric than the corresponding multiplexors in the LUT-based architecture.

## 5. Conclusions

In this paper, we have presented a product-term based synthesizable programmable logic device, and compared it to the lookup-table based device in [9]. Overall, we found that our new architecture is 35% smaller and 72% faster, primarily due to a dramatic reduction in the amount of circuitry needed to route signals. We also investigated the effects of various architectural parameters on the efficiency of our core.

Better synthesis results could be obtained by “tweaking” the standard-cell library to include cells specifically optimized to implement our programmable logic fabric. We have not considered this in this paper, since our goal was to create architectures that can be implemented using the standard synthesis tools, cell libraries, and design flows that integrated circuit designers are already familiar with. Nonetheless, if this design technique was to become mainstream, specially-designed standard cells could be created.

## Acknowledgements

This work was supported by Micronet, Altera, and the National Sciences and Engineering Research Council of Canada. The authors wish to thank Deming Chen, Zhijun Huang and Jason Cong for providing their technology mapping program, PLAmap, and Jason Anderson for supplying TEMPLA.

## References

[1] S.J.E. Wilton, R. Saleh, “Programmable Logic IP Cores in SoC Design: Opportunities and Challenges”, Proceedings of the 2001 Custom Integrated Circuits Conference, pp. 63-66.

- [2] Actel Corp., “VariCore Embedded Programmable Gate Array Core (EPGA) 0.18um Family”, Datasheet, December 2001.
- [3] Leopard Logic Inc, “HyperBlox FP Embedded FPGA Cores”, Product Brief, 2002.
- [4] M2000 Inc., “M2000 FLEXEOS Configurable IP Core”, <http://www.m2000.fr>.
- [5] eASIC, “eASIC 0.13um Core”, <http://www.easic.com/products/easiccore013.html>
- [6] M. Borgatti, F. Lertora, B. Foret, L. Cali, “A Reconfigurable System featuring Dynamically Extensible Embedded Microprocessor, FPGA, and Customisable I/O”, IEEE Journal of Solid-State Circuits, vol. 38, no. 3, March 2003, pp. 521-529.
- [7] T. Vaida, “PLC Advanced Technology Demonstrator TestChipB”, Proceedings of the 2001 Custom Integrated Circuits Conference, May 2001, pp. 67-70.
- [8] J.C.H. Wu, V. Aken’Ova, S.J.E. Wilton, R. Saleh, “SoC Implementation Issues for Synthesizable Embedded Programmable Logic Cores”, in the Proceedings of the 2003 Custom Integrated Circuits Conference.
- [9] N. Kafafi, K. Bozman, S.J.E. Wilton, “Architectures and Algorithms for Synthesizable Embedded Programmable Logic Cores”, Proceedings of the ACM International Symposium on Field-Programmable Gate Arrays, Feb 2003.
- [10] A. Kaviani, S. Brown, “The Hybrid Field Programmable Architecture”, IEEE Design and Test, April-June 1999.
- [11] D. Chen, J. Cong, M. Ercegovic, Z. Huang, “Performance-Driven Mapping for CPLD Architectures”, in the ACM International Symposium on Field-Programmable Gate Arrays, Feb 2001, pp. 39-47.
- [12] Altera Corp, “Max 7000 Programmable Logic Device Family Datasheet”, v.6.6, 2003.
- [13] Xilinx Corp., “XPLA3 CPLD, Xilinx Preliminary Product Specification DS012”, v.1.6, 2003.
- [14] J.L. Kouloheris, A.E. Gamal, “FPGA Performance vs. Cell Granularity”, Proceedings of the 1991 Custom Integrated Circuits Conference.

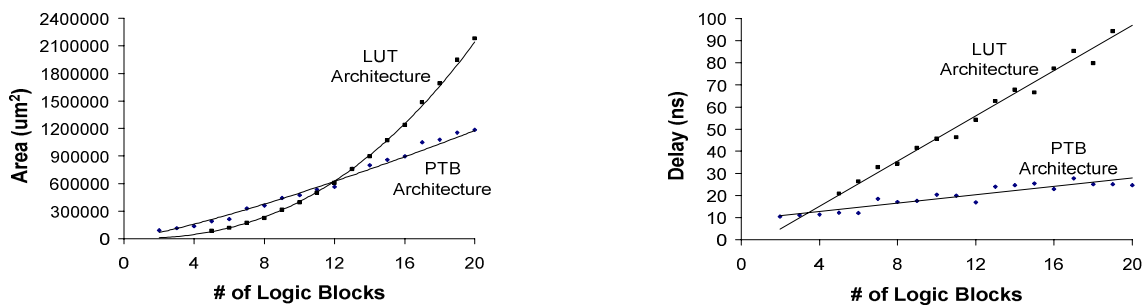


Figure 6: Area, Depth results for PTB and LUT-based architecture