A Synthesizable Datapath-Oriented Embedded FPGA Fabric

Steve J.E. Wilton[2], C.H. Ho[1], Philip H.W. Leong[1,3], Wayne Luk[1], Brad Quinton[2]

[1]Department of Computing
Imperial College London
London, England
{cho,wl}@doc.ic.ac.uk

[2]Dept. of Electrical
and Computer Engineering
University of British Columbia
Vancouver, B.C., Canada
{stevew,bradq}@ece.ubc.ca

[3]Dept. of Computer
Science and Engineering
Chinese University of Hong Kong
Hong Kong
phwl@cse.cuhk.edu.hk

ABSTRACT

We present an architecture for a synthesizable datapath-oriented Field Programmable Gate Array (FPGA) core which can be used to provide post-fabrication flexibility to a System-on-Chip (SoC). Our architecture is optimized for bus-based operations that are common in signal processing and computation intensive applications. It employs a directional routing architecture, which allows it to be synthesized using standard ASIC design tools and flows. We also describe a proof-of-concept layout of our core. It is shown that the proposed architecture is significantly more area efficient than the best previously reported synthesizable programmable logic core.

Categories and Subject Descriptors

B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Gate Arrays*

General Terms

Design

Keywords

Field Programmable Gate Array, Datapath, Synthesis, Integrated Circuit, System-on-Chip, Embedded Block

1. INTRODUCTION

Recent years have seen an impressive improvement in the achievable density of integrated circuits. This improvement has led to an increase in the cost and difficulty of designing and testing a correctly-functioning chip. Stand-alone FPGAs (Field Programmable Gate Arrays) provide one way of reducing the design cost; however, many designs are not suitable for FPGAs because of their speed, density or power requirements. For these types of designs, a fixed-function chip, often designed using standard cells and the System-on-Chip (SoC) methodology [12], may be the only option.

Configurability can be provided by embedding one or more programmable logic cores into the fixed-function chip. In such a chip, most of the design is implemented using fixed-function ASIC (Application Specific Integrated Circuit) gates, while programmable logic is used sparingly in those parts of the chip that are likely to change. These changes may be due to errors in the design or specification, future upgrades, or to allow for the customization of an integrated circuit for multiple customers. Embedded programmable logic can also provide a mechanism to add debug capability [11].

A programmable logic fabric can either be *hard* or *soft*. An ASIC designer using a hard fabric would obtain a layout and embed it directly into the integrated circuit. One challenge with this approach is that design tools that allow seamless integration of fixed and programmable logic are still not mature. Timing analysis, power distribution, and verification are difficult when the function to be implemented in the core is not known.

An alternative technique is recently described which addresses this concern by shifting the burden from the ASIC designer to mature standard-cell synthesis tools [14, 15]. In this technique, an ASIC designer would obtain a synthesizable version of their programmable logic fabric (a *soft* core) written in a hardware description language, and would synthesize it along with the rest of the ASIC. The primary advantage of this technique is that the task of integrating such cores is far easier than the task of integrating hard cores. The synthesis tools can be the same ones that are used to synthesize the fixed (ASIC) portions of the chip. No modifications to the tools are required, and the flow follows a standard integrated circuit design flow that designers are familiar with.

Despite these advantages, there are significant area, speed, and power penalties when using these cores. Previous architectures [14, 15] suffer a 6.4 times overhead compared to a hard programmable logic core. This limits their application to small circuits such as state machines.

In this paper we present a new architecture that is between 6 times and 426 times more area efficient than the best previously reported synthesizable programmable logic core. Moreover, we show that the new architecture has a density similar to that of a standard full-custom fine-grained FPGA. The density improvement is obtained by using a datapath-style architecture, optimized for performing computations

FPGA'07, February 18–20, 2007, Monterey, California, USA.

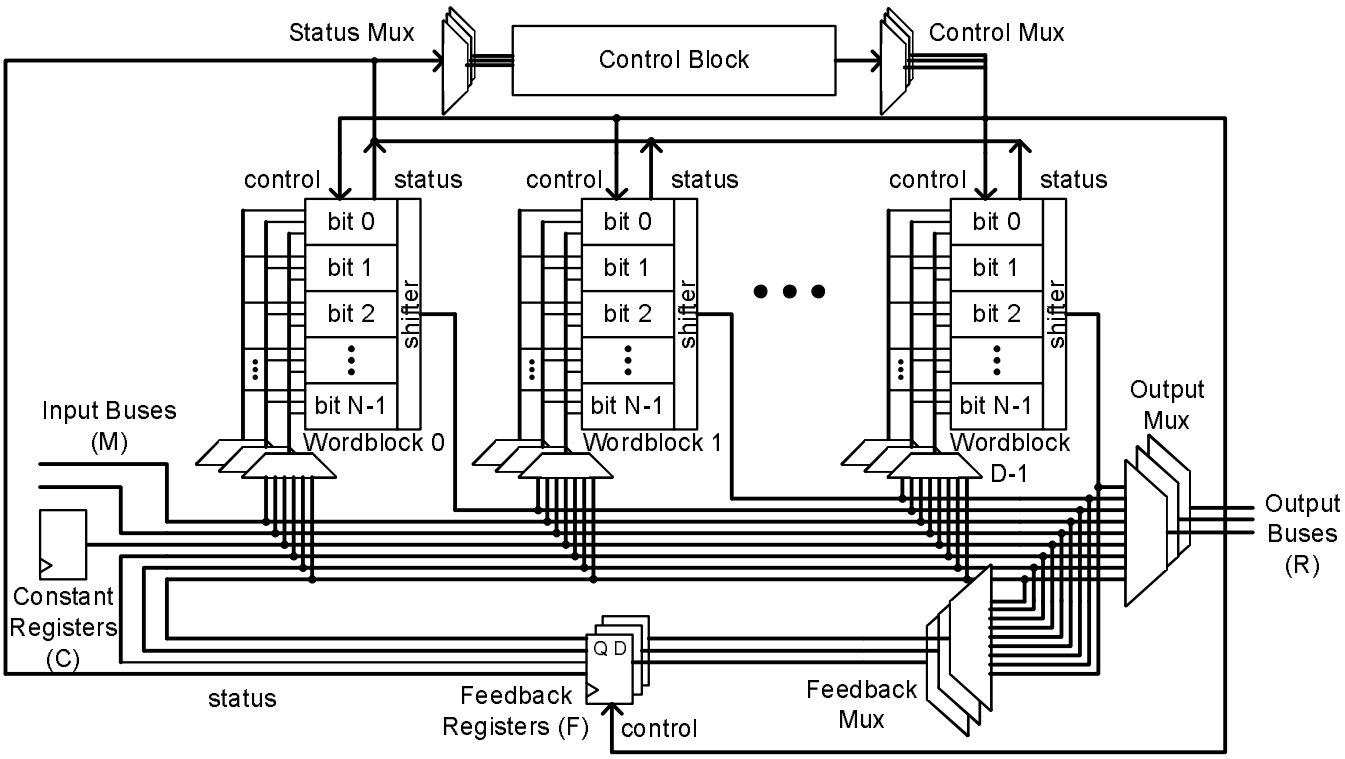


Figure 1: Fabric Architecture (configuration elements not shown).

such as those found in signal processing and arithmetic applications. Although such cores would be less flexible than their bit-level counterparts, this is less of a concern in an embedded FPGA core than in a stand-alone FPGA since the context in which the core will be used is known when the chip is designed. As an example, a programmable logic core embedded into the datapath of a signal processing ASIC will certainly be used to implement multiply/accumulate-type functions, rather than a more general logic circuit. This allows us to create a core that is optimized for datapath operations without worrying about how well it can implement random logic functions. In addition, if buses are used to connect the programmable logic core and the fixed function circuitry (as would be expected in a datapath-oriented circuit), the specific pins on which these buses are mapped, as well as the width of the bus, are known at the time the fabric is instantiated, and will not change over the lifetime of the ASIC. This allows us to further optimize our fabric.

The proposed architecture is similar to both the RaPiD architecture [3] and datapath-oriented FPGAs [2, 5, 8, 16, 17]. However, the fact that we want to implement our fabric using existing synthesis tools motivates us to change the architecture significantly. Section 2 describes the motivations, and presents our architecture. Section 3 gives an example of how an application can be mapped to our architecture. Section 4 measures the efficiency of our architecture as a function of various architectural parameters, and Section 5 compares our architecture to the best previous synthesizable programmable logic core, as well as to an ASIC. Section 6 describes our proof-of-concept implementation. Section 7 compares our approach to that taken in stand-alone datapath-oriented FPGAs and coarse-grained architectures. Finally, Section 8 presents concluding remarks and opportunities for future work.

2. ARCHITECTURE

In this section, we first outline the requirements of an architecture for a synthesizable FPGA core, and then describe our architecture in detail. We actually describe a *family* of architectures, where each member of the family is differentiated by various parameters. An SoC designer would select an architecture from this family based on the amount of programmable logic required, as well as the number and nature of the connections to the programmable logic.

2.1 Requirements of a Synthesizable Architecture

The proposed design methodology requires that the programmable logic fabric be *synthesizable*. By this, we mean the fabric can be synthesized and implemented using existing synthesis and ASIC design tools with no modifications to the tools or the CAD flow.

For a fabric to be synthesizable in this way, it must not contain combinational loops. Standard synthesis tools, timing analysis tools, and power estimation tools are optimized for circuits without combinational loops. Although circuits with such loops can be synthesized, this usually requires the designer to manually "break" the loops by identifying some false paths. This requires considerably more understanding about the internals of the core that a typical ASIC designer would have. Note that a standard unconfigured FPGA contains many combinational loops. A designer will rarely con-

figure the FPGA to implement combinational loops, but before configuration, such loops exist.

On the other hand, our methodology provides a unique opportunity for optimization. When designing a hard layout for an FPGA, layout effort is reduced by dividing the design into tiles, where each tile is identical. In our case, the tiles are synthesized and laid out automatically by CAD tools; thus, it is no longer critical that each tile is identical.

One important aspect of our work is that we are focusing on *small* user circuits. Large circuits would typically be implemented using a hard-programmable logic core. An example circuit might be a small debug controller, as will be described later in this paper.

2.2 Our Architecture

Figure 1 shows our architecture. The fabric contains D identical *wordblocks*, each containing N identical *bitblocks*. Unlike a fine-grained FPGA, the bitblocks within a wordblock are all controlled by the same set of control bits. This means all bitblocks within a wordblock perform the same function. We will consider the implication of this feature on density in Section 4.

As shown in Figure 2, each bitblock contains two lookup-tables, several multiplexers, and a flip-flop. A single wordblock can implement an N bit adder/subtractor, an N-bit wide three-input multiplexer, any other three-input logic function, or some five-input functions. Two control inputs k_1 and k_2 (from the Control Block, to be described below) allow for efficient implementation of multiplexers and other datapath functions that require a control input. The same two control lines are driven to all bitblocks in a wordblock. The select lines of the three multiplexers in Figure 2 as well as the function lines of the two lookup-tables are driven by configuration bits. In total, 35 configuration bits are required per bitblock; as described above, these bits are shared between all bitblocks in a wordblock. The wordblock also contains a programmable shifter, which can pass data through unchanged, or shift the word one bit to the right (signed or unsigned shift) or one bit to the left; the state of the shift block is controlled by two configuration bits.

Each wordblock receives up to three inputs from either the M primary bus inputs, the F feedback paths, the C constant registers, or any of the outputs of wordblocks to the left. The control lines for the input selection multiplexers are driven by configuration bits. Note that buses are switched as a unit; this improves density, since one set of configuration bits can be shared among all bits. However, it also reduces flexibility, since it is not possible to select part of one bus and part of another bus (this functionality can be implemented within a wordblock by careful use of a "mask" in one of the C constant registers). The R output buses of the architecture can be selected from the same set of $M + F + C$ buses or from the output of any of the D wordblocks. The same signals (except the C constants) can be fed back, through a flip-flop, to all wordblocks; this provides a mechanism to connect wordblock outputs to the inputs of wordblocks to the left and also supports an efficient way to delay signals by one clock cycle without using a wordblock.

Wordblocks can efficiently implement combinational functions including adders and multiplexers, and can perform masking operations in conjunction with one or more of the constant registers. However, they cannot efficiently implement multipliers. Since multipliers are an important part

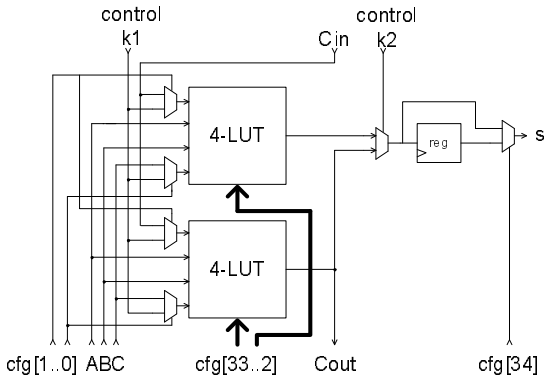


Figure 2: Bitblock (status flags not shown).

of our target applications, selected wordblocks in the fabric are replaced with embedded multipliers. Each embedded multiplier has two N-bit inputs which are selected from the $M + C + F + i$ (where i is the number of wordblocks to the left of the multiplier) buses using routing multiplexers. The multiplier produces two output buses, one for the high order result and one for the low order result. These outputs can be selected by all subsequent routing multiplexers including the output and feedback multiplexers. We denote the number of multipliers as A, and assume each multiplier displaces one wordblock (so, the number of wordblocks is $D - A$).

Although our architecture is aimed at datapath-oriented applications, a small amount of control logic is sometimes needed to control the datapath. Such logic can be implemented in the Control Block. This block contains fine-grained product-term based programmable logic resources, and is similar to the architecture described in [15]. The fabric contains P product-term blocks, each with 9 inputs, 10 product terms, and 3 outputs (this was shown to work well in [15]). The control block also contains registers to support state machines. Inputs to the Control Block are selected from a number of status signals generated throughout the datapath. Each wordblock generates a carry-out, an overflow, an MSB, an LSB, and a zero flag; each feedback path generates the same flags, with the exception of the carry-out. This large number of status bits are multiplexed into a small number of inputs using the Status Multiplexer, which is controlled by configuration bits. The exact number of these status bits that can be provided to the Control Block depends on the size of the Control Block. Similarly, the Control Block generates a number of outputs. These outputs can be provided to various control lines in the fabric using the Control Multiplexer; for each control line in the fabric, any of the Control Block outputs or the constants '0' or '1' can be selected.

The parameters used to describe the architecture are summarized in Table 1.

3. EXAMPLE MAPPING

To demonstrate how this architecture can be used to implement a circuit, we focus on a single example. The example is a common debugging operation [11]; the circuit monitors two buses, and counts the number of times a certain mask (composed of 1's, 0's and "don't care" bits) matches

D	Number of Wordblocks (incl. multipliers)
N	Bit Width
M	Number of Input Buses
R	Number of Output Buses
F	Number of Feedback Paths
C	Number of Constant Registers
A	Number of Multipliers
P	Number of Product-Term Blocks

Table 1: Architectural Parameters.

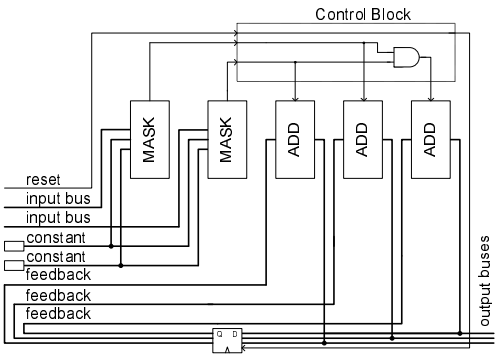


Figure 3: Example Mapping.

each bus, as well as the number of times both buses match the mask at the same time.

Figure 3 illustrates how the application can be implemented. Two constant registers are used to hold the mask value (two registers are required so that "don't care" bits can be specified). One wordblock combines these two mask values and the first input bus to produce a 0 if the bit matches (or is a "don't care") or a 1 otherwise. A second wordblock performs the same function on the second bus. Both wordblocks provide their zero flag (indicating a match has occurred) to the Control Block; the Control Block provides this signal to the carry-in signals of two adders (each implemented in a wordblock). The Control Block also provides the AND of the two zero flags to a third adder (implemented in another wordblock). Each of the three accumulated counts are stored in the feedback registers; these counts are fed back to the input signals of the adders. The reset control lines for the feedback registers are also controlled by the Control Block. Finally, the three adder outputs are connected to the outputs of the fabric.

4. PARAMETER OPTIMIZATION

In this section, we first determine the impact of the parameters in Table 1 on the area and delay of the fabric.

Table 2 shows a breakdown of the area of a fabric with N=16, D=16, M=3, R=2, F=3, C=2, A=4, and P=4. The various components were synthesized using Synopsys Design Compiler, and the cell area predicted by Synopsys was reported. Configuration circuits, clock circuits, and all other essential parts of the core were included in the synthesizable model. Although it would be more accurate to perform place and route on the Synopsys-generated netlist and measure the chip area directly, previous results have shown that the Synopsys area results have a good correlation to the final chip area results [14]. A 130nm process was assumed.

Module		Area in μm^2	Percentage
Datapath	Wordblocks	86, 251	23.8 %
	Multipliers	45, 236	12.5 %
	Config. Bits	24, 323	6.7 %
	Feedback Regs	2, 322	0.6 %
	Routing Muxes	86, 251	33.2 %
	Total Datapath	120, 460	76.7 %
Status Multiplexer		18, 520	5.1%
Control Multiplexer		14, 603	4.0%
Control Block		51, 418	14.2%
Total		363, 136	100.0%

Table 2: Area Breakdown.

As one can see, most of the area is used to implement the datapath portion of the fabric. Within the datapath, the largest component of the area is due to the routing multiplexers. The four multipliers and 12 wordblocks also consume a significant amount of area. The configuration bits within the datapath consumes 6.7% of the entire fabric.

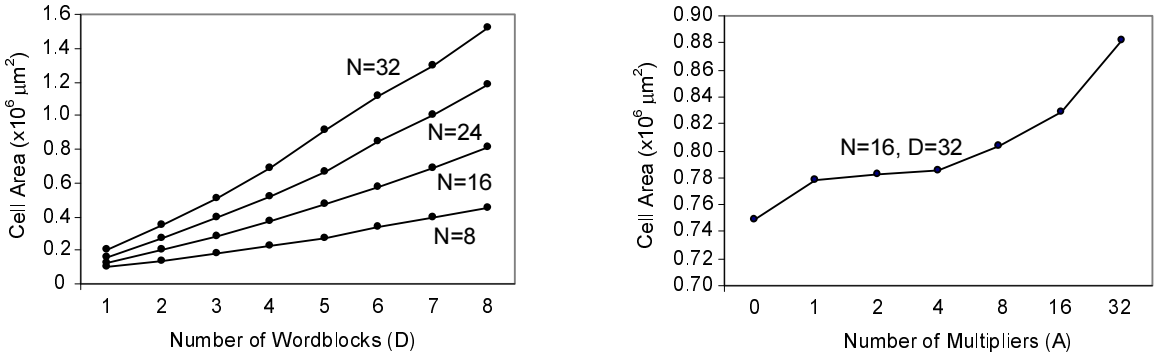
Figure 4(a) shows the impact of N and D on area. In this experiment, M=3, R=2, F=3, C=2, A=4, and P=4. As the graph shows, the area is roughly proportional to both D and N; increasing D increases the number of wordblocks and corresponding routing multiplexers, while increasing N increases the sizes of these blocks.

The impact on area of the number of multipliers, A, is shown in Figure 4(b). All other parameters are as before, with N=16 and D=32. Intuitively, as A increases, the area goes up. This is despite the fact that the area of the 32-bit multiplier is roughly the same as the area of a 32-bit wordblock (including the associated routing multiplexers and configuration bits). The reason that the area goes up as A increases is that the multiplier produces two bus outputs (a wordblock produces one). This increases the size of the routing multiplexers in all downstream wordblocks, as well as the output multiplexers and feedback multiplexers. The graph shows that the increase from $A = 0$ to $A = 1$ is larger than the increase from $A = 1$ to $A = 2$. This is because if there is only one multiplier, it is placed in the left-most slot. This increases the size of all subsequent routing multiplexers. When a second multiplier is added, it is placed in the middle of the fabric, so only half of the routing multiplexers are increased (those to the right of the new multiplier).

Figure 5(a) shows the impact of P on the area of the fabric. As one can see, the number of product-term blocks in the control block has a significant effect on the size of the overall architecture.

We also measured the impact of M, R, C, and F. Each of these parameters had a linear effect on area. Increasing M from 1 to 8 increased the area by 15%, increasing R from 1 to 8 increased the area by 7.8%, increasing F from 0 to 6 increased the area by 25%, and increasing C from 0 to 8 increased the area by 17%. Parameter R (the number of output buses) has the smallest effect on area, since an increase in R does not imply an increase in the size of any of the routing multiplexers. For all other parameters, as the parameter is increased, additional buses are created; these buses are supplied to all routing multiplexers, making them larger. Parameter F has the largest impact since each feedback register is associated with three status bits and one control bit.

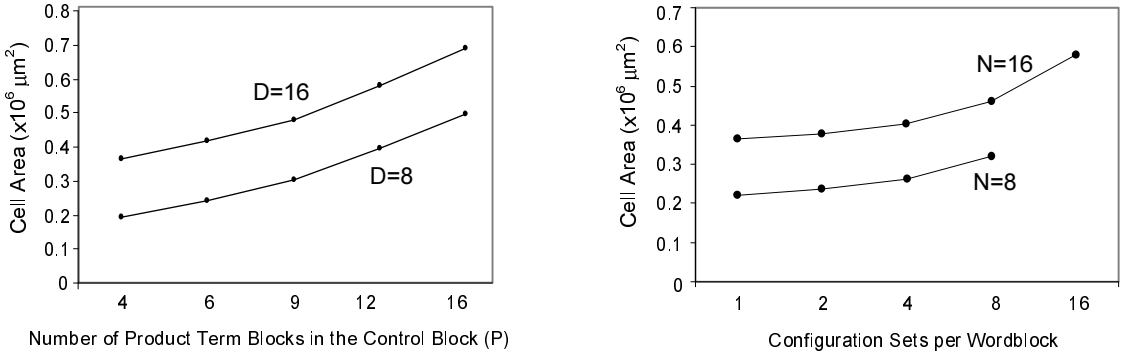
In our architecture, the same set of 35 configuration bits



(a) Impact of D and N

(b) Impact of A

Figure 4: Parameter Sweeps, where M=3, R=2, F=3, C=2, A=4, P=4 unless otherwise specified.



(a) Impact of size of Control Block

(b) Impact of Wordblock granularity

Figure 5: Parameter Sweeps, where M=3, R=2, F=3, C=2, A=4, P=4 unless otherwise specified.

are shared among all bitblocks in a wordblock. To investigate the implication of this feature on density, we varied the number of configuration bit sets per wordblock from 1 (the baseline architecture) to N, in which every bitblock is controlled by a separate set of 35 configuration bits. The impact on area is shown in Figure 5(b) for two values of N (all other parameters are the same as before). As the graph shows, the more flexible architecture, the more area is required (because of the extra configuration bits). For $N = 16$, an architecture in which each bitblock has its own configuration set is 60% larger than an architecture in which all bitblocks within a wordblock share a configuration set.

The maximum clock frequency at which the fabric can run depends on the configuration implemented in the fabric. Table 3 shows post-synthesis, pre-place and route delay estimates for various paths within the fabric. The delay through the wordblock is the delay from the output of the register in one wordblock to the input of the register in the next wordblock. This quantity is independent of N, and depends very slightly on M, C, and F, as well as the position of the wordblock in the array (since these parameters determine the size of the routing multiplexer used to select inputs for the second wordblock). On the other hand, the delay of the multiplier goes up as N increases. Measurements of the maximum carry chain delay within one wordblock are also given in the table (from the carry-in of the least significant bit to the carry-out of the most significant bit). The last entry in the table shows the delay of a combinational path that passes through all wordblocks in a fabric with D=32 and A=8; clearly, most applications would not configure the fabric to have such a long critical path.

Delay through one wordblock	3.25ns
Delay through one multiplier (8 bits)	5.39ns
Delay through one multiplier (16 bits)	8.50ns
Delay through carry chain (8 bits)	8.71ns
Delay through carry chain (16 bits)	14.93ns
Delay through 24 wordblocks and 8 multipliers	178ns

Table 3: Delay Estimates.

5. MAPPING RESULTS

In this section, we use benchmark circuits to compare our architecture to a fine-grained synthesizable programmable logic core [15]. We first describe our benchmark circuits. We then present mapping results, first assuming that the architecture is tailored for each benchmark, and then assuming the more realistic case in which the fabric is not tuned for each benchmark.

5.1 Benchmark Circuits

As described earlier, we are focusing on user circuits. An example is the debug controller described in Section 3. Such circuits typically contain a single datapath controlled by a small controller; circuits with multiple intersecting datapaths are likely too large to be implemented using a synthesizable core, and thus, we do not consider such circuits in this section.

We used eight benchmark circuits. Three of the benchmarks, *bfly*, *dscg* and *fir4* were used in [6]. The *bfly* benchmark performs the computation $z = y + x * w$ where the inputs and output are complex numbers; this is commonly used within a Fast Fourier Transform computation. The *dscg* circuit is the datapath of a digital sine-cosine generator. The *fir4* circuit is a 4-tap finite impulse response filter. The other four circuits were constructed specifically for this work: The *dotv3* benchmark computes the dot vector product of two inputs. The *egcd* circuit implements an extended binary greatest common divisor algorithm [10]. The *momul* benchmark is a Montgomery Multiplier [10]. The *median* circuit is a median filter that accepts streaming data and returns the median (actually second-largest) of the last four entries. Finally, the *debug1* benchmark is the debugging circuit considered in Section 3. All benchmarks assume 8 bit operands, except *median* and *debug1* which assume 16 bit operands. We have specifically chosen these circuits since they are small, and support the type of application we would expect to implement on a synthesizable programmable logic core. Large user circuits would be typically implemented using a hard programmable logic core.

5.2 Optimized Parameters

We first compare our architecture to the best previous synthesizable architecture [15] and to a non-programmable ASIC implementation of each circuit. This will give an upper-bound of the efficiency of our architecture if tuned properly.

To map each benchmark to our architecture, the benchmark was first split into datapath and control sections. The datapath portion of the circuit was mapped (by hand) to wordblocks, and appropriate values of D, N, M, R, D, A, F, and C were chosen. The control section was mapped to product-term blocks, using PLAmap [1]. Using the number of product-term blocks required by PLAmap to implement the circuit, as well as the datapath parameters described above, a custom-built tool was used to generate an appropriately-sized fabric. This fabric was then synthesized using Synopsys Design Compiler, and the cell area predicted by Synopsys was reported. Again, a $130nm$ CMOS process was assumed. The results are shown in Column 10 of Table 4.

For comparison, we also show the area that would be required to implement the same circuit using the fine-grained synthesizable fabric from [15] in Column 11. These measurements were obtained using the architectures and tools described in [15]. We were unable to compare our architecture to the architecture described in [14], since that architecture only supports combinational circuits, and most of our benchmarks are sequential. However [15] shows that their architecture is significantly more dense than that in [14], even for combinational circuits. Column 12 shows the area required by the benchmark circuit if synthesized directly in standard cells (in which case there is no programmability).

Column 13 shows the ratio of the area required to implement each benchmark using the fine-grained fabric to the area required to implement the same benchmark in our architecture. As the table shows, there are two categories of circuits. Circuits *bfly*, *dotv3*, *dscg* and *fir4* all show ratios of between 1610 and 1940. In other words, our architecture is 1610 times to 1940 times more area-efficient than the fine-grained fabric. The remaining circuits show more modest ratios between 14 and 75.

These results are dramatic. First consider those benchmarks with ratios between 14 and 75. Given that, for each circuit, we are creating a fabric in which configuration bits are shared between either 8 or 16 bits, we would expect to see a ratio of no larger than 8 or 16. The reason our ratios are larger than this has to do with the inefficiencies of the fine-grained architecture when implementing very large circuits. The architecture in [15] contains many routing multiplexers; the size of these multiplexers *and* the number of these multiplexers both grow linearly with the size of the fabric. For the small circuits for which the previous architecture was designed, these multiplexers are small. However, when the fabric is scaled large enough to implement our benchmark circuits, these multiplexers become unwieldy, causing the area to grow significantly.

This does not explain the four benchmarks that have ratios greater than 1600. These benchmarks all contain a significant number of multipliers. In our architecture, these multipliers are implemented as a hard embedded block (as in many commercial stand-alone FPGAs). On the other hand, the fine-grained architecture does not contain these embedded blocks, meaning the multipliers must be implemented using the normal logic resources. This is aggravated by the fact that product-term based architectures, such as [15] are notoriously bad at implementing XOR functions, which are common in multipliers.

Column 14 shows the ratio of the area required to implement each benchmark circuit in our fabric to the area required to implement the same benchmark circuit using fixed ASIC cells (with no programmability). This measure is the overhead resulting from configurability using our architecture. As the table shows, for the circuits with a significant number of embedded multipliers, this ratio is between 5 and 8. For circuits without a significant number of embedded multipliers, this number is between 25 and 117. It is interesting that these larger numbers are of the same order of magnitude as the ratio of an FPGA implementation to an ASIC implementation [7]. In other words, the overhead due to configurability in our architecture is similar to the overhead inherent in a hand-designed stand-alone FPGA. This is a surprising result; it shows that synthesizable cores *can* provide the density that designers currently accept from non-synthesized programmable logic devices.

Bench-mark	Fabric Parameters								Datapath (ours)	Fined-Grain [15]	ASIC	Fine-Grain/ Datapath	Datapath/ ASIC
	D	N	M	R	C	F	A	P					
bfly	8	8	6	1	0	5	4	0	68,190	132,339,335	9,300	1940	7.33
dotv3	5	8	6	1	0	2	3	0	34,119	65,534,780	6,575	1921	5.19
dscg	8	8	3	2	0	2	4	1	72,178	116,271,968	9,473	1611	7.62
fir4	11	8	1	1	4	0	0	0	76,213	130,971,120	9843	1718	7.74
egcd	27	8	2	4	1	9	0	27	1,225,231	22,776,474	10,420	18.6	117
momul	13	8	7	2	0	6	2	8	294,135	11,448,589	7,097	38.9	41
median	8	16	1	1	0	4	0	2	142,172	10,733,962	4,420	75.5	32
debug1	5	16	2	3	2	3	0	1	87,265	1,302,928	3,484	14.9	25

Table 4: Area results when the fabric is optimized for each benchmark circuit.

5.3 Derived Parameters

When gathering the results in Section 5.2 we chose all fabric parameters independently for each circuit. This unfairly biases the results in our favour. One of the drawbacks of partitioning the fabric between control and datapath is that different user circuits require different amounts of control and datapath; since we do not know what will be implemented in the fabric when the ASIC is designed, choosing the amount of each type of fabric is difficult. If the partition is not chosen carefully, either control resources or datapath resources will be wasted. This is not a problem with fine-grained architectures, since the fine-grained fabric can be used to build either control or datapath structures. In this section, we address this issue by fixing this parameter (as well as other parameters) as a function of the fabric size.

We repeated the experiments in Section 5.2. We choose values of D, N, M, and R independently for each benchmark circuit. This is reasonable; when including a fabric in an ASIC, the bit-width, the number of input and output buses, and the desired fabric size is known. Unlike the previous experiments, however, we calculated the remaining parameters as a function of D. If the resulting architecture has more constant registers, feedback paths, multipliers, or product term blocks than are needed by the benchmark circuit, then the extra resources are wasted. On the other hand, if the fabric does not contain enough of any of these resources, the fabric size (D) is increased until the benchmark circuit can be implemented.

Table 5 shows the results, using the same columns as in Table 4. The size of the fine-grained fabric and the ASIC implementation are copied into Table 5 for convenience. In all cases, we compute $C = \lceil \frac{D}{4} \rceil$, $F = \lceil \frac{D}{2} \rceil$, $A = \lceil \frac{D}{4} \rceil$, and $P = \lceil \frac{D}{3} \rceil$. Although these may not be the optimum ratios, we do not have enough benchmark circuits to determine optimum ratios for each parameter. These ratios were selected because they appear "reasonable" based on our experience (for example, since each product term block has three outputs, setting $P = \lceil \frac{D}{3} \rceil$ means that, on average, one select line per wordblock can be generated). If additional experiments were conducted, and the optimum ratios found, they would tend to improve the results in this section.

As the results in Table 5 show, in general, the area required to implement each benchmark circuit on our fabric has increased, due to the benchmark circuits not exactly matching the generated architecture. The ratio of the area required to implement each circuit in the fine-grained architecture of [15] to the area required to implement the same benchmark in our fabric now ranges from 10.9 to 426, while the ratio of the area required to implement each circuit in our fabric to the area required to implement the same circuit in an ASIC ranges from 31.2 to 363.

6. PROOF-OF-CONCEPT LAYOUT

As a proof-of-concept, we performed place and route on the datapath portion of our fabric with D=12, N=8, M=7, R=2, F=6, A=0, and C=0. The Verilog description of the fabric was synthesized with Synopsys Design Compiler, targeting the STMicroelectronics 90nm, 7-layer metal process using the STMicroelectronics CORE90GPSVT standard cell library. The netlist was flattened into a single level of hierarchy before layout. The pre-layout netlist contained a total gate area of 300098 μm^2. The cell placement, cell sizing and repeater insertion was performed by Cadence SoC Encounter. Detailed wire routing was performed using Cadence NanoRoute and was completed with no violations. The total gate area after place and route was 336402 μm^2. The placement region set to approximately 625 $\mu m \times 625$ μm, resulting in a gate density of 86.1%.

7. COMPARISON TO PREVIOUS WORK

Our architecture inherits ideas from previous work on fine-grained synthesizable fabric, datapath-oriented FPGAs and coarse-grained reconfigurable architectures, such as RaPiD. This section compares our architecture to several previous studies.

7.1 Fine-Grained Synthesizable Fabric

We have compared our architecture to the best synthesizable architecture in Section 5.2 using a set of benchmarking circuits. The architecture proposed in [15] is fine-grained and the configurability is provided by programmable logic arrays (PLA). For the circuits which contain significant number of multipliers, our architecture is 1610 times to 1940 times more area-efficient than the fine-grained fabric. This is because the multiplier in our architecture are implemented as a hard embedded block while the fine-grained architecture does not contain these blocks. It means the multipliers must be implemented using normal logic resources which contributes large area consumption.

For some other circuits which do not have large number of multipliers, the area ratio is between 14 and 75. We observe that the architecture in [15] is not efficient when implementing large circuits. The architecture in [15] contains many routing multiplexers. Both the size of these multiplexers and the number of multiplexers grow linearly with the

Benchmark	Fabric Parameters				Computed				Datapath (ours)	Fine-Grain [15]	ASIC	Fine-Grain/ Datapath	Datapath/ ASIC
	D	N	M	R	C	F	A	P					
bfly	16	8	6	1	4	8	4	6	332,091	132,339,335	9,300	399	35.7
dotv3	9	8	6	1	3	5	3	3	225,518	65,534,780	6,575	291	34.3
dscg	16	8	3	2	4	8	4	6	325,029	116,271,968	9,473	358	34.3
fir4	16	8	1	1	4	8	4	6	307,154	130,971,120	9843	426	31.2
egcd	70	8	2	4	18	35	18	24	3,778,611	22,776,474	10,420	6.02	363
momul	22	8	7	2	6	11	6	8	486,316	11,448,589	7,097	23.5	68.5
median	9	16	1	1	3	5	3	3	194,654	10,733,963	4,420	55.1	44
debug1	6	16	2	3	2	3	2	2	119,286	1,302,928	3,484	10.9	34

Table 5: Area results when low-level parameters are computed.

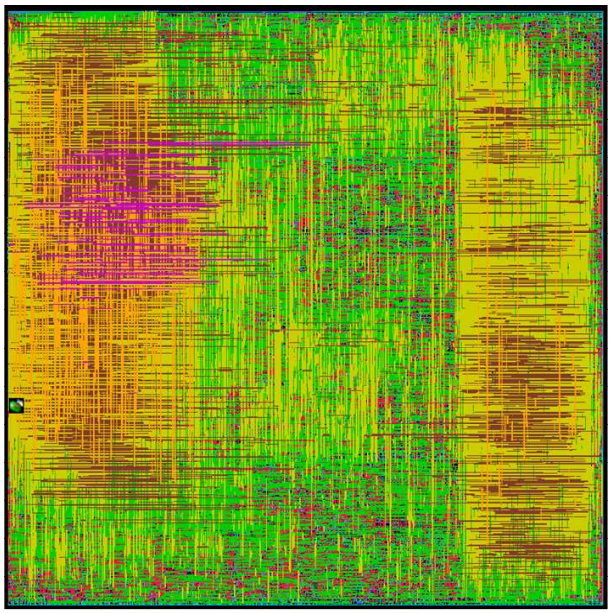


Figure 6: Proof-of-Concept Layout.

size of fabric. When the fabric is scaled large enough to implement the given benchmarking circuit, these multiplexers become unwieldy and it causes the area to grow significantly.

7.2 Datapath-Oriented FPGAs

Several previous studies have considered datapath-oriented FPGAs [2, 5, 8, 16, 17]. In these architectures, configuration bits are shared among multiple lookup-tables and multiple routing switches.

In these previous works, it is assumed that the FPGA is to be laid out by hand or using a custom layout tool, and thus, no attempt is made to remove combinational loops in the unprogrammed fabric. This is a key requirement of a synthesizable architecture. Although these architectures can be synthesized (as in [8]), the combinational loops will require designers to "break" these loops by declaring false paths; this increases the difficulty of including these fabrics in a large SoC.

A second difference between these datapath FPGAs and our architecture is that these previous architectures have been optimized assuming that the bus width of the target application and the pin assignments of the buses are not known when the fabric is designed. This limits the amount of optimization possible; for example, in [16], it is found that the number of blocks sharing a set of configuration bits should be no more than four. In our context, the bus width and pin assignments are determined when the ASIC is designed, and will not change over the lifetime of the chip. This allows us to share a set of configuration bits across all datapath bits in a word.

7.3 Coarse-Grained Fabrics

Coarse-grained architectures, in which lookup-tables are replaced by ALUs, have also been described in [3, 4, 9, 13]. Of these, the RaPiD architecture [13] was specifically designed for use in an SoC. RaPid contains a linear array of dedicated functional units connected using dedicated buses. Control logic is implemented using a separate module that provides control signals to the functional units.

RaPiD is intended to support fairly large applications such

as image and signal processing, and may be best implemented as a hard programmable logic core. It would be possible to "scale down" RaPiD and use it as a synthesizable core. However, like the datapath FPGAs described in the previous section, the unprogrammed RaPiD fabric contains combinational loops. Our architecture eliminates these using a directional routing network.

Another difference between RaPiD and our architecture is that RaPiD (as well as many coarse-grained architectures) contains a heterogeneous mix of fixed-function datapath elements rather than configurable wordblocks. When creating a RaPiD fabric, one must choose how many of each type of functional unit is to be included in the fabric. However, once that decision is made, the *location* of each functional unit does not matter, since buses can be routed from any functional unit to any other functional unit. In our architecture, however, the routing network requires less area but is less flexible, so it is less likely that a pre-positioned set of fixed functional units could be connected to implement a target application. Thus, we provide a general-purpose wordblock that can be used to implement many functions. The only exceptions to this rule are the embedded multiplier blocks; we distribute these evenly across the fabric to maximize the likelihood that applications can be mapped successfully.

8. CONCLUSION

We have presented an architecture for a datapath-oriented synthesizable FPGA core which can be used to provide post-fabrication flexibility to an SoC. The proposed architecture features with sharing configuration bits, carry chains, directional routing architecture and embedded multipliers. Compared to the previous best synthesizable embedded programmable logic core, our architecture is between 6 times and 426 times more area efficient, depending on the number of embedded multipliers in the fabric. This opens the use of synthesizable embedded programmable logic cores to significantly larger applications, and provides a configuration overhead similar to that of standard hand-designed FPGAs. Current and future work includes automating the design and optimization of synthesizable embedded FPGA fabrics and the associated design mapping tools, and the support of complex hardwired elements such as floating-point operators in such fabrics.

9. ACKNOWLEDGMENTS

This work was performed at Imperial College London. The authors gratefully acknowledge the support of the UK EPSRC (grant EP/C549481/1 and grant EP/D060567/1), Altera Corp. and the NSERC of Canada.

10. REFERENCES

[1] D. Chen, J. Cong, M. Ercegovac, and Z. Huang. Performance-driven mapping for CPLD architectures. In *ACM Int. Symp. on Field-Programmable Gate Arrays*, pages 39–47, Feb. 2001.

[2] D. Cherepacha and D. Lewis. DP-FPGA: An FPGA architecture optimized for datapaths. In *Int. Conf. on VLSI Design*, pages 329–343, 1996.

[3] D. Cronquist, P. Franklin, C. Fisher, M. Figueroa, and C. Ebeling. Architecture design of reconfigurable pipelined datapaths. In *Twentieth Anniversary Conf. on Advanced Research in VLSI*, page 23, 1999.

[4] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor. Piperench: A reconfigurable architecture and compiler. *IEEE Computer*, 33(4):70–77, April 2000.

[5] S. Hauck, T. Fry, M. Hosler, and J. Kao. The Chimera Reconfigurable functional unit. *IEEE Trans. on VLSI*, 12(2):206–217, Feb. 2004.

[6] C. Ho, P. Leong, W. Luk, S. Wilton, and S. Lopez-Buedo. Virtual embedded blocks: A methodology for evaluating embedded elements in FPGAs. In *Int. Symp. on Field-Programmable Custom Computing Machines*, pages 35–44, Apr. 2006.

[7] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. In *Int. Symp. on Field-Programmable Gate Arrays*, pages 21–30, Feb. 2006.

[8] K. Leijten-Nowak and J. L. van Meerbergen. An FPGA architecture with enhanced datapath functionality. In *Int. Symp. on Field-Programmable Gate Arrays*, pages 195–204, Feb. 2003.

[9] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings. A reconfigurable arithmetic array for multimedia applications. In *ACM Int. Symp. on Field-Programmable Gate Arrays*, pages 135–143, Feb. 1999.

[10] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*, pages 602–606. CRC Press, 1996.

[11] B. Quinton and S. Wilton. Post-silicon debug using programmable logic cores. In *Int. Conf. on Field-Programmable Technology*, pages 241–247, Dec. 2005.

[12] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. Pande, C. Grecu, and A. Ivanov. System-on-chip: Reuse and integration. *Proceedings of the IEEE*, 94(6):1050–1069, June 2006.

[13] H. Singh, M. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves. Morphosys: An integrated reconfigurable system for data-parallel and compute intensive applications. *IEEE Trans. on Computers*, 49(5):465–481, April 2000.

[14] S. Wilton, N. Kafafi, J. Wu, K. Bozman, V. Aken'Ova, and R. Saleh. Design considerations for soft embedded programmable logic cores. *IEEE Journal of Solid-State Circuits*, 40(2):485–497, Feb. 2005.

[15] A. Yan and S. Wilton. Product-term based synthesizable embedded programmable logic cores. *IEEE Trans. on VLSI*, 14(5):474–488, May 2006.

[16] A. Ye and J. Rose. Using bus-based connections to improve field-programmable gate array density for implementing datapath circuits. In *Int. Symp. on Field-Programmable Gate Arrays*, pages 3–13, Feb. 2005.

[17] A. Ye, J. Rose, and D. Lewis. Architecture of datapath-oriented coarse-grain logic and routing for FPGAs. In *IEEE Custom Integrated Circuits Conf.*, pages 61–64, Sept. 2003.