A Synthesizable Datapath-Oriented Embedded FPGA Fabric for Silicon Debug Applications

STEVEN J.E. WILTON
University of British Columbia
CHUN HOK HO
Imperial College London
BRADLEY QUINTON
University of British Columbia
PHILIP H.W. LEONG
Chinese University of Hong Kong
and
WAYNE LUK
Imperial College London

We present an architecture for a synthesizable datapath-oriented FPGA core that can be used to provide post-fabrication flexibility to an SoC. Our architecture is optimized for bus-based operations and employs a directional routing architecture, which allows it to be synthesized using standard ASIC design tools and flows. The primary motivation for this architecture is to provide an efficient mechanism to support on-chip debugging. The fabric can also be used to implement other datapath-oriented circuits such as those needed in signal processing and computation-intensive applications. We evaluate our architecture using a set of benchmark circuits and compare it to previous fabrics in terms of area, speed, and power.

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 Corporation, the NSERC of Canada, and the Research Grants Council of Hong Kong (grant CUHK413707).
Authors' addresses: S.J.E. Wilton and B. Quinton, University of British Columbia, 2332 Main Mall, Vancouver, BC, Canada V6T 1Z4; C.H. Ho and W. Luk, Imperial College London, 434 Huxley Building, South Kensington Campus, London SW7 2AZ, United Kingdom; P.H.W. Leong, Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China, Contact email: stevew@ece.ubc.ca.

Categories and Subject Descriptors: B.7.1 [**Integrated Circuits**]: Types and Design Styles—*gate arrays*

General Terms: Design, Verification

Additional Key Words and Phrases: Field programmable gate array, silicon debug, integrated circuit, system-on-chip

ACM Reference Format:
Wilton, S. J. E., Ho, C. H., Quinton, B., Leong, P. H. W., and Wayne, L. 2008. A synthesizable datapath-oriented embedded FPGA fabric for silicon debug applications. ACM Trans. Reconfig. Techn. Syst. 1, 1, Article 7 (March 2008), 25 pages. DOI: 10.1145/1331897.1331903. http://doi.acm.org/10.1145/1331897.1331903.

1. INTRODUCTION

Advances in integrated circuit (IC) technology have made possible the integration of a large number of functional blocks on a single chip. One of the challenges in creating such large chips is ensuring that the design is functionally correct. No matter how careful the designer is, some design errors will not be detected by simulation, and will not become apparent until the fabricated chip is tested. Debugging an integrated circuit after fabrication is especially challenging, since there is limited controllability and observability through the external pins. Tight re-spin schedules make efficient debugging even more critical. As a result, post-fabrication debug represents a major time and cost investment for integrated circuit manufacturers.

Recently, designers of large IC's have started to incorporate extra logic that is intended to be used exclusively for post-fabrication debugging. Although some design-for-testability structures, such as scan chains, can be used to enhance observability and controllability, these structures may not be suitable for debugging. Unlike post-fabrication testing, post-fabrication debugging is not usually an automated process. The tedious loading and unloading of long scan-chains for functional debug is not feasible for chips with high-speed internally generated clocks, and in any case will limit the ability to quickly exercise the integrated circuit to identify the causes of errors.

Another possible approach is to monitor signals and record traces for key signals in a small RAM block. These signal traces can then be uploaded to a host PC or analyzed with an on-chip microprocessor. This scheme does not support some complex debugging operations. As an example, consider a packet processing integrated circuit. Packets typically contain sequence numbers, and a receiver can use these sequence numbers to determine whether a packet arrives out of order. A common debugging operation might be to monitor these sequence numbers to precisely determine when an error occurs. Storing traces on-chip for post-analysis may not be an option if the error packet does not occur until after millions of packets have been processed. In addition, some debugging operations may involve monitoring a number of buses to determine when a specific data item arrives on any of them. If there are a large number of buses, and the run time is large, this may overwhelm any reasonable amount of on-chip memory. Thus, a method of analyzing packets on-the-fly is required.

One way of providing this ability is to embed a small amount of programmable logic onto the chip [Quinton and Wilton 2005; Abramovici et al. 2006]. After fabrication, an engineer can implement small test circuits using this programmable logic to help find the source of incorrect behaviour. The programmable logic core would be connected to key signals throughout the integrated circuit using a programmable flexible network as described in Quinton and Wilton [2005]. As an example, if it is suspected that the system crashes when a certain pattern appears on a system bus, a small test circuit that monitors the bus and collects key data when the pattern occurs could be implemented. More complex test data generators such as a uniform or Gaussian random number generators and complex error monitoring circuits that pre-process or compress data before storage are also possible. Normally, this embedded programmable logic would be disabled after debugging; however, it would also be possible to use such a core to "patch" design errors [Sarangi et al. 2006, 2007; Wagner et al. 2006] or enhance post-fabrication testability [Abramovici et al. 2002].

A key part of this embedded debug infrastructure is the programmable logic fabric. Although it would be possible to create a fabric based on commercial stand-alone Field-Programmable Gate Array (FPGA) architectures, this may not be desirable for three reasons. First, commercial architectures are optimized for large applications. Since we intend to implement only small test circuits, it is likely that such architectures will provide far more routing resources than are required, leading to increased area overhead. Area overhead is especially important in our application since the embedded debug fabric is pure overhead that will not be used when the chip is finally operational. Second, commercial FPGA architectures are optimized to work for a wide variety of circuits in different applications. Since much debugging is performed by monitoring buses, we would expect our applications to be primarily datapath-oriented. In addition, because the embedded fabric is a fixed part of an integrated circuit, the context in which it will be used (the buses that will be monitored, etc) will not change over time. As we will show in Section 6, we can significantly reduce the area required by our architecture by taking advantage of this. The third reason commercial FPGA fabrics may not work well is that they are not easily synthesizable by common commercial synthesis tools. Most System-on-Chip (SoC) designs are implemented by synthesizing hardware description language (HDL) specifications into standard cells. The use of embedded cores will be much more palatable to SoC designers if their integration can be made as seamless as possible [Wilton et al. 2005]. We will further discuss the concept of synthesizable fabrics in Section 2.

Other embedded fabrics have been described. Both datapath fabrics [Cherepacha and Lewis 1996; Hauck et al. 2004; Leijten-Nowak and van Meerbergen 2003; Ye et al. 2003; Ye and Rose 2005] and coarse-grained architectures [Marshall et al. 1999; Cronquist et al. 1999; Goldstein et al. 2000; Singh et al. 2000] may provide better density than commercial FPGAs, but still suffer in that they are not easily synthesizable. Also like commercial FPGAs, these architectures have been optimized for large stand-alone applications. Synthesizable programmable logic fabrics have been described in Wilton et al. [2005],

and Yan and Wilton [2006]. These architectures are not datapath oriented, and hence suffer from a significant density overhead.

In this article we describe a novel architecture for an embedded FPGA fabric that has been optimized for small datapath applications such as debug circuits. Such an architecture can also be used to implement other arithmetic-intensive circuits. We compare this architecture to both a fine-grained synthesizable architecture, and an Application-Specific Integrated Circuit (ASIC) implementation of the debug circuitry. Unlike the fine-grained architecture, our fabric contains support for word-level operations and routing, and contains embedded multipliers. Unlike the ASIC implementation, our fabric is flexible enough to implement a wide variety of embedded applications. We show that the new architecture (including embedded multipliers) has a density similar to that of a standard full-custom fine-grained FPGA (without embedded multipliers).

This article is organized as follows. Section 2 describes the environment in which our embedded core will be used, and describes the requirements of our architecture. The architecture itself is then described in Section 3. Section 4 then gives an example of how an application can be mapped to our architecture. Section 5 reports the efficiency of our architecture as a function of various architectural parameters, and Section 6 compares our architecture to a previous synthesizable programmable logic core, as well as to an ASIC implementation. Power and delay numbers are given in Section 7. Section 8 shows how our fabric can be integrated into an ASIC. Section 9 compares our approach to that taken in stand-alone datapath-oriented FPGAs and coarse-grained architectures. Finally, Section 10 presents concluding remarks and opportunities for future work.

An early version of this article appears in Wilton et al. [2007]. In this article, we extend this work by including post-configuration delay and power measurements for our datapath fabric, new benchmark circuits dedicated to on-chip debugging, and discussion of SoC integration.

2. FRAMEWORK AND ARCHITECTURAL REQUIREMENTS

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. These hard fabrics could either be based on commercial FPGA designs, or generated automatically using a layout or architecture generator [Padalia et al. 2003; Compton and Hauck 2007; Holland and Hauck 2007].

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 has been recently described which addresses this concern by shifting the burden from the ASIC designer to mature standard-cell synthesis tools [Wilton et al. 2005; Yan and Wilton 2006]. 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.

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 than a typical ASIC designer would have. Note that a standard unconfigured FPGA contains many combinational loops. A designer will rarely configure the FPGA to implement combinational loops, but before configuration, such loops exist. Thus, the first requirement of our architecture is that it does not contain any combinational loops.

The second requirement of our architecture is that it is as small as possible. The area devoted to on-chip debug will not be used during the normal operation of the chip (of course, the fabric could be removed from production versions of a high-volume chip). Existing synthesizable fabrics suffer a 6.4 times area overhead, compared to a hard programmable logic core [Wilton et al. 2005]. As will be shown in the next section, we address this by taking advantage of the datapath nature of the anticipated debug circuits. In addition, we take advantage of the fact that the context in which the core will be used is known when the SoC is designed. As an example, if buses are connected to the core, the specific pins on which these buses are mapped, as well as the width of each bus, are known when the fabric is instantiated, and will not change over the lifetime of the chip.

The third requirement is that the fabric should be as fast as possible. Ideally, we would like to run our integrated circuit "at speed" during debugging. The nature of programmable logic means we may not be able to achieve this, but we would like to be as close as possible to this goal. Power consumption is a secondary concern, since the fabric will likely only be used "in the lab" during debugging. If our fabric is to be used to implement other arithmetic-oriented applications in a production version of an integrated circuit, then power consumption may become important.

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.

3. ARCHITECTURE

In this section, we describe a family of architectures for our embedded programmable logic core. Each member of the family is differentiated by various parameters. An SoC designer would select an architecture from this family

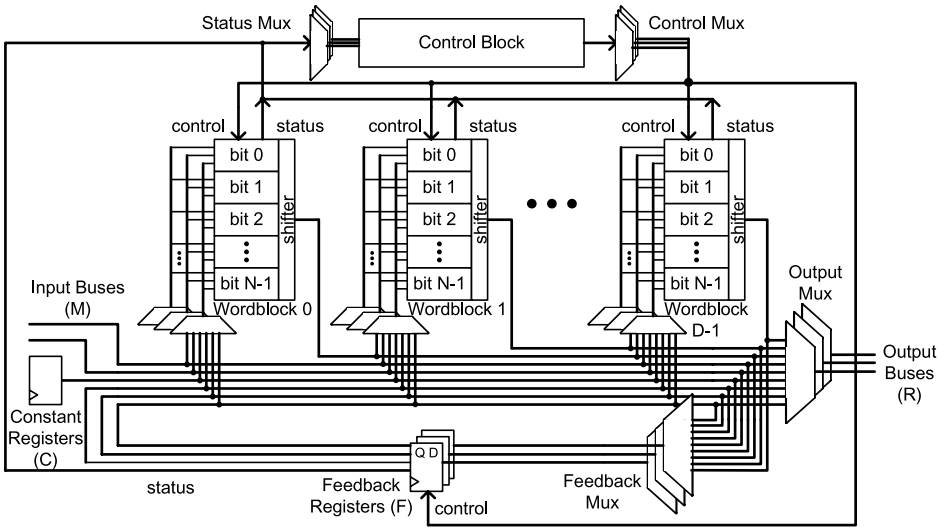


Fig. 1. Fabric Architecture (configuration elements not shown).

based on the amount of programmable logic required, as well as the number and nature of the connections to the programmable logic.

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 impact of this feature on density in Section 5.

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/subtracter, 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 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

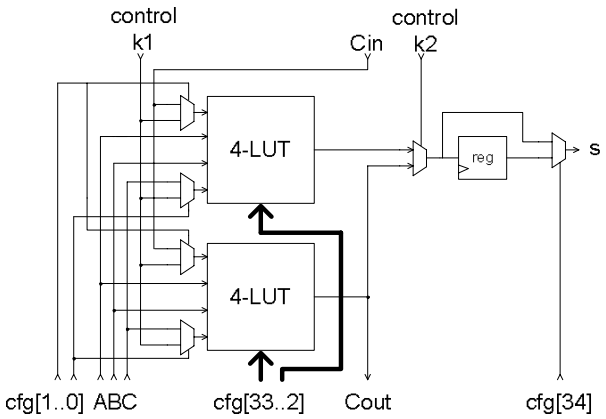


Fig. 2. Bitblock (status flags not shown).

of one bus and part of another bus, although 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.

Multipliers are an important part of some target applications. Therefore, 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 Yan and Wilton [2006]. The fabric contains P product-term blocks, each with 9 inputs, 10 product terms, and 3 outputs (this was shown to work well in Yan and Wilton [2006]). 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

Table I. Architectural Parameters.

D	Number of wordblocks (including 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

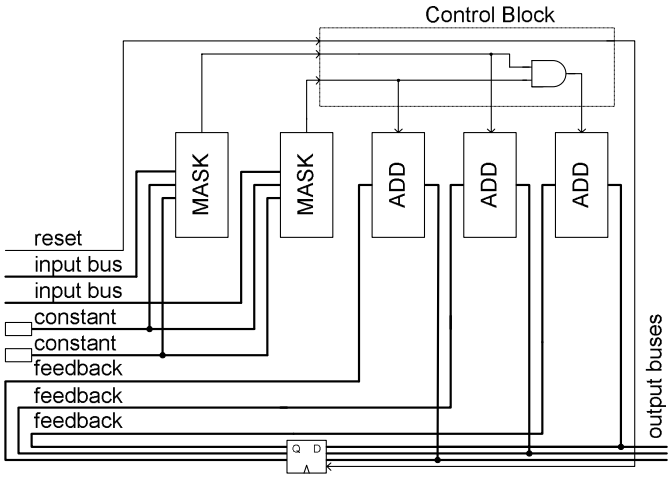


Fig. 3. Example mapping.

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 I.

4. 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; 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 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. The mask value is represented by two constants. Each bit in the constant corresponds to one bit in the incoming data stream. As shown in Table II, the two bits together determine whether the corresponding bit in the data stream must be a '1', '0', or is a "don't care" bit.

Table II. Meaning of Mask Bits in Example

Bit i from Constant value 1	Bit i from Constant Value 2	Meaning
0	0	Data bit i must be 0
0	1	Data bit i must be 1
1	0	Data bit i can be 0 or 1

The left-most wordblock in Figure 3 combines the incoming data word on the first input bus with the two constant values to determine whether the incoming data word is a match. To do this, each of the bitblocks within the wordblock performs the following function:

$$\text{not}\,(a_i + \overline{b_i \oplus d_i})$$

where a_i is bit i of the first constant, b_i is bit i of the second constant, and d_i is bit i of the incoming data word. If the result of this function is 0, a match in bit i has occured. If *all* bits produce a result of 0, then a match has occured. As described in Section 3, each wordblock has a "zero" flag output that is asserted when the result from all wordblocks are 0; this flag is sent to the control block to indicate a match has occured. The second wordblock in Figure 3 performs the same function, but uses the incoming data word on the second input bus.

The control block then uses these two match flags to determine which counters to increment. If the first flag is set, the first counter is incremented (implemented using the third wordblock in Figure 3). The second counter is incremented when the second flag is set, and the final counter is incremented when both flags are set. 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.

5. PARAMETER OPTIMIZATION

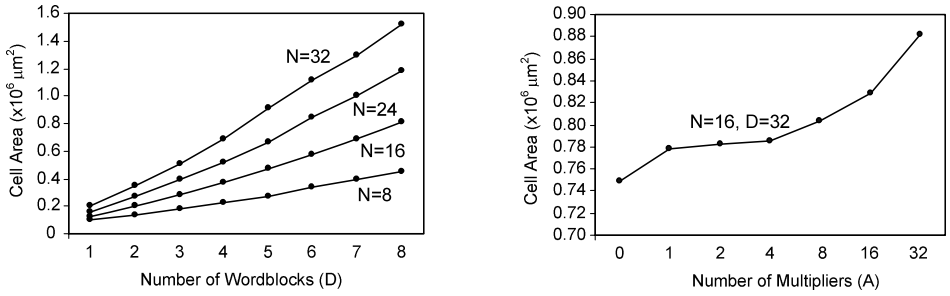
In this section, we first determine the impact of the parameters in Table I on the area of the fabric. Delay and power will be considered in Section 7.

Table III 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 are synthesized using Synopsys Design Compiler, and the cell area predicted by the same tool is reported. All area values are given to three significant digits. 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 [Wilton et al. 2005]. A 130-*nm* process is assumed.

As shown in the table, 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

Table III. Area Breakdown

Module		Area in μm^2	Percentage
Datapath	wordblocks	86,300	23.8 %
	multipliers	45,200	12.5 %
	config. bits	24,300	6.70 %
	feedback regs	2,320	0.600 %
	routing muxes	86,300	33.2 %
	total datapath	120,000	76.7 %
status multiplexer		18,500	5.10%
control multiplexer		14,600	4.00%
control block		51,400	14.2%
Total		363,000	100%



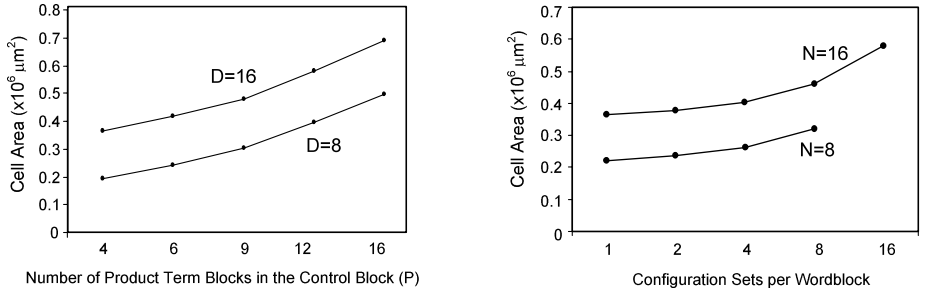
(a) Impact of D and N (b) Impact of A

Fig. 4. Parameter sweeps, where M=3, R=2, F=3, C=2, A=4, P=4 unless otherwise specified.

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 the case 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).



(a) Impact of size of Control Block (b) Impact of Wordblock granularity

Fig. 5. Parameter sweeps, where M=3, R=2, F=3, C=2, A=4, P=4 unless otherwise specified.

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 has a linear effect on area. Increasing M from 1 to 8 increases the area by 15%, increasing R from 1 to 8 increases the area by 7.8%, increasing F from 0 to 6 increases the area by 25%, and increasing C from 0 to 8 increases 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 are shared among all bitblocks in a wordblock. To investigate the impact of this feature on density, we vary 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, with all other parameters the same as before. As the graph shows, more flexible architectures with more configuration sets per wordblock require more area 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.

6. AREA RESULTS

In this section, we use benchmark circuits to compare our architecture to a fine-grained synthesizable programmable logic core [Yan and Wilton 2006] and to an ASIC implementation. 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.

6.1 Benchmark Circuits

To evaluate our architecture, we use a collection of datapath circuits. Although the primary motivation for our architecture is to implement debug circuits, the fabric can actually be used to implement any small datapath-oriented circuit. Thus, in order to fully exercise the fabric, we have created a suite of benchmark circuits representative of the types of circuits that would be implemented in our fabric. These circuits typically contain a single datapath controlled by a small controller. We focused on these single datapath circuits since circuits with multiple intersecting datapaths are likely too large to implemented using a synthesizable core.

We used ten benchmark circuits. Three of these are example debug applications, and the remainder are circuits that are similar in size and structure to the type of circuits that would be implemented in our core. The first debug circuit, *debug1* is the circuit described in Section 4. The second debug circuit, *seqchk* is a sequence number checking circuit. Many packet based inter-chip communication schemes (such as PCI Express) use sequence numbers to ensure that packets arrive in order and are not lost. The circuit monitors incoming data words, identifies the start of a packet (using a pre-determined mask), parses through the packet (using a counter) to find the sequence number, and compares it with the previous sequence number. Any out-of order sequence number, which would indicate a lost packet on a direct point-to-point link, increments a counter.

The third debug circuit, *fletcher*, can be used to detect checksum mismatches. In many communication applications, when a circuit detects a checksum mismatch, it will enter an error state and ask for re-transmission of the data. Determining that this is happening in a chip can often be an important step in the debugging process; it can explain low performance/throughput, it can alert the debugger that a different state of the circuit is being stimulated, and can potentially point to overall system problems. In its simplest form the checksum is calculated by simply adding the bytes in the data stream. However, this allows rearranged words or extra zero bytes to pass undetected. The Fletcher algorithm contains an additional accumulator to help detect these error conditions [Fletcher 1982; Nakassis 1988]. The benchmark circuit monitors an incoming bus, and uses the Fletcher algorithm to compute the checksum of the incoming stream.

Of the remaining benchmarks, three, *bfly*, *dscg* and *fir4* are used in Ho et al. [2006]. 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 *dotv3*, *momul*, and *median* circuits were constructed for this work. The *dotv3* benchmark computes the dot product of two input vectors. The *egcd* circuit implements an extended binary greatest common divisor algorithm [Menezes et al 1996]. The *momul* benchmark is a Montgomery Multiplier [Menezes et al 1996]. Finally, the *median* circuit is a median filter that

Table IV. Parameters Used for Each Benchmark Circuit

	Fabric Parameters							
Benchmark	D	N	M	R	C	F	A	P
debug1	5	16	2	3	2	3	0	1
seqchk	5	16	1	1	3	3	0	2
fletcher	8	16	1	2	2	3	0	2
bfly	8	8	6	1	0	5	4	0
dotv3	5	8	6	1	0	2	3	0
dscg	8	8	3	2	0	2	4	1
egcd	27	8	2	4	1	9	0	15
fir4	11	8	1	1	4	0	0	0
median	8	16	1	1	0	4	0	2
momul	13	8	7	2	0	6	1	8

accepts streaming data and returns the median (actually second-largest) of the last four entries.

All benchmarks assume 8 bit operands, except *median*, *debug1*, *fletcher*, and *seqchk* 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.

6.2 Optimized Parameters

We first compare our architecture to a previous synthesizable architecture [Yan and Wilton 2006] and to a nonprogrammable 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 [Chen et al. 2001]. 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. The parameters used to construct the datapath for each benchmark circuit is shown in Table IV. Each fabric is then synthesized using Synopsys Design Compiler, and the cell area predicted by the same tool is reported. Again, a 130-nm CMOS process was assumed. The results are shown in Column 2 of Table V (note that these results are slightly different than those from Wilton et al. [2007] since here we assume all inputs and outputs are registered). All results are shown to three significant digits.

For comparison, we also show the area that would be required to implement the same circuit using the fine-grained synthesizable fabric from Yan and Wilton [2006] in Column 3. These measurements were obtained using the architectures and tools described in Yan and Wilton [2006]. We were unable to compare our architecture to the architecture described in Wilton et al. [2005], since that architecture only supports combinational circuits, and most of our

Table V. Area Results when the Fabric is Optimized for Each Benchmark Circuit

Benchmark	Datapath (ours) (μm^2)	Fined-Grain [Yan and Wilton 2006] (μm^2)	ASIC (μm^2)	Fine-Grain/ Datapath	Datapath/ ASIC
debug1	87,300	1,300,000	3,640	14.9	24.0
seqchk	92,500	1,200,000	3,600	13.0	25.7
fletcher	133,000	2,580,000	4,660	20.0	28.5
bfly	68,200	132,000,000	17,800	1,940	3.83
dotv3	34,100	65,500,000	8,350	1,920	4.08
dscg	72,200	116,000,000	11,600	1,610	6.22
egcd	1,230,000	22,800,000	9,880	18.5	124
fir4	76,200	131,000,000	12,100	1,720	6.30
median	142,000	10,700,000	5,270	75.4	26.9
momul	294,000	11,400,000	7,100	38.8	41.4

benchmarks are sequential. Column 4 shows the area required by the benchmark circuit if synthesized directly in standard cells, in which case there is no programmability.

Column 5 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 1611 and 1940. In other words, our architecture is 1611 times to 1940 times more area-efficient than the fine-grained fabric. The remaining circuits show more modest ratios between 13.0 and 75.4.

These results are dramatic. First consider those benchmarks with ratios between 13.0 and 75.4. 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 these has to do with the inefficiencies of the fine-grained architecture when implementing very large circuits. The architecture in Yan and Wilton [2006] was optimized for somewhat smaller circuits (between 10 and 300 equivalent 4-input lookup tables). As the fine-grained architecture is scaled to implement larger circuits, the size of the routing multiplexers grows. Each multiplexer has an input for every primary input and every output in the previous levels within the fabric. In Yan and Wilton [2006], depopulating these multiplexers was not considered, since the circuits were small enough that the multiplexer area did not become unwieldy. In addition, the number of these multiplexers is proportional to the amount of logic in the fabric, since there is one multiplexer per product-term block input. This means that the overall size of the fabric grows quadratically with circuit size.

This quadratic increase in size suggests that the previous architecture is not efficient at implementing these sorts of large circuits. In addition, the architecture in Yan and Wilton [2006] was optimized for control circuits rather than datapath circuits. Thus, the comparison to the fine-grained architecture must be made with caution. However, even if the fine-grained architecture was optimized for our benchmark circuits, we would still expect that our architecture would be significantly smaller than the fine-grained architecture.

The above explanation does not cover 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). However, the fine-grained architecture does not contain these embedded blocks, so the multipliers must be implemented using the normal logic resources. This is aggravated by the fact that product-term based architectures, such as Yan and Wilton [2006] are notoriously bad at implementing XOR functions, which are common in multipliers.

Column 6 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 3.8 and 6.3. For circuits without a significant number of embedded multipliers, this number is between 24 and 124. 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 [Kuon and Rose 2007]. 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.

6.3 Derived Parameters

When gathering the results in Section 6.2 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 repeat the experiments in Section 6.2. 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 are known. Unlike the previous experiments, however, we calculate 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. 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. The parameters used for each benchmark circuit are shown in Table VI. In all cases, we compute $C = \lceil \frac{D}{4} \rceil$, $F = \lceil \frac{D}{2} \rceil$, $A = \lceil \frac{D}{4} \rceil$, and $P = \lceil \frac{D}{3} \rceil$.

Table VI. Parameters Used for Each Benchmark Circuit when Low-Level Parameters are Computed

	Fabric Parameters				Computed			
Benchmark	D	N	M	R	C	F	A	P
debug1	7	16	2	3	2	4	2	3
seqchk	9	16	1	1	3	5	3	3
fletcher	11	16	1	1	3	6	3	4
bfly	16	8	6	1	4	8	4	6
dotv3	9	8	6	1	3	5	3	3
dscg	16	8	3	2	4	8	4	6
egcd	70	8	2	4	18	35	18	24
fir4	16	8	1	1	4	8	4	6
median	11	16	1	1	3	6	3	4
momul	24	8	7	2	6	12	6	8

Table VII. Area Results when Low-Level Parameters are Computed

Benchmark	Datapath (ours) (μm^2)	Fine-Grain [Yan and Wilton 2006] (μm^2)	ASIC (μm^2)	Fine-Grain/ Datapath	Datapath/ ASIC
debug1	178,000	1,300,000	3,640	7.30	48.9
seqchk	220,000	1,200,000	3,600	5.45	61.1
fletcher	196,000	2,580,000	4,660	13.2	42.1
bfly	335,000	132,000,000	17,800	394	18.8
dotv3	226,000	65,500,000	8,350	290	27.1
dscg	325,000	116,000,000	11,600	357	28.0
egcd	3,190,000	22,800,000	9,880	7.15	323
fir4	307,000	131,000,000	12,100	427	25.4
median	272,000	10,700,000	5,270	39.3	51.6
momul	542,000	11,400,000	7,100	21.0	76.3

Although these may not be the optimum ratios, we do not have enough benchmark circuits to determine optimum ratios for each parameter. These ratios are 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.

Table VII shows the results, using the same columns as in Table V. Again, all results are shown to three significant digits. The size of the fine-grained fabric and the ASIC implementation are copied into Table VII for convenience. 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 Yan and Wilton [2006] to the area required to implement the same benchmark in our fabric now ranges from 7.1 to 427, 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 18.8 to 323.

Table VIII. Delay Estimates of Paths within Fabric

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)	14.9ns
Delay through 24 wordblocks and 8 multipliers	178ns

7. DELAY AND POWER RESULTS

The maximum clock frequency at which the fabric can run depends on the configuration implemented in the fabric. We first consider the delay of various paths within the fabric, and then consider the delay and power of the fabric when implementing our benchmark circuits.

7.1 Path Delays

Table VIII 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). 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.

7.2 Mapping Results

The delay and power dissipation of our architecture depend on the circuit implemented in the fabric. To estimate the delay and power overhead of our architecture, we mapped each of our benchmark circuits to a datapath constructed using the derived parameters from Table VI. For each mapping, we determined appropriate values for all configuration bits, and used Synopsys Design Compiler to estimate the critical path and dynamic power dissipated by the fabric with these configuration bits set properly. Again, a 130nm technology was assumed.

The results in this section are for the datapath portion of the fabric only. Measuring the delay paths through the control block is difficult. Determining the state of every programming bit in the datapath portion of the architecture is not difficult since there are only a small number of programming bits. However, since in the fine-grained control fabric, there are so many more programming bits, manually determining and setting the state of each bit would be infeasible.

Table IX shows the results. Column 2 shows the critical path delay of each circuit implemented on our architecture, Column 3 shows the same quantity

Table IX. Datapath Delay and Power Estimates for Configured Fabric

Benchmark	Datapath (ns)	ASIC (ns)	Ratio	Datapath (mW)	ASIC (mW)	Ratio
debug1	14.6	2.02	7.23	2.7	0.13	21
seqchk	15.4	2.27	6.78	4.0	0.12	33
fletcher	16.5	8.37	1.97	5.8	0.23	25
bfly	11.1	2.81	3.95	3.0	1.19	2.5
dotv3	9.94	3.75	2.65	1.7	0.52	3.3
dscg	7.64	4.72	1.62	2.6	0.70	3.7
egcd	14.3	6.65	2.15	26	0.40	65
fir4	10.5	4.21	2.49	2.3	0.62	3.7
median	16.5	2.33	7.08	4.6	0.44	10
momul	7.53	5.34	1.41	4.2	0.41	10

for each circuit implemented as an ASIC, and Column 4 shows the ratio between these two estimates. This ratio, which is the delay overhead imposed by reconfigurability, varies from 1.4 to 7.2. The larger ratios correspond to circuits that do not use the embedded multipliers. In our architecture, the embedded multipliers are implemented using ASIC circuitry, thus we would expect that circuits that make heavy use of the multipliers run closer to the speed of the corresponding ASIC implementation. As with the area results, these delay ratios are of the same order of magnitude as the ratio of the delay of a standard FPGA implementation to that of an ASIC implementation [Kuon and Rose 2007]. This means that the delay overhead due to configurability in our architecture is similar to the delay overhead inherent in a hand-designed stand-alone FPGA. Unlike our results, however, [Kuon and Rose 2007] found that the ratio did not depend strongly on the number of embedded multipliers used. This is likely because, in a standard FPGA, the delay of a net is primarily due to the routing connections between logic blocks, while in our fabric, the delay depends more on the gates and connections within each wordblock and multiplier.

The final three columns in Table IX show power measurements for our architecture and an ASIC. The ratios vary from 2.5 to 65. In general, the circuits that do not use embedded multipliers show a larger ratio, as expected. The ratio for *egcd* is significantly larger than the others. As shown in Table IV, this circuit requires more control logic than the other circuits. Because we are fixing the ratio of control resources (P) to wordblocks (D), a fabric large enough to implement the control part of the circuit has many more wordblocks than are needed. In our architecture, these unused wordblocks consume power (this suggests that we should "turn off" unused wordblocks, however we do not consider this in this article). In Kuon and Rose [2007], it is reported that a standard FPGA dissipated 14 times more power than an ASIC; once again, this is in-line with our results.

8. INTEGRATION OF FABRIC INTO AN SOC

Although the primary contribution of this article is the architecture of the embedded fabric, it is instructive to consider how this fabric could be embedded into an SoC. In this section, we focus on three aspects of this integration:

selecting the size of the fabric, connecting the fabric to the SoC, and matching the speed of the programmable logic core to that of the SoC.

8.1 Fabric Size

Although the fabric is flexible enough to implement many different debug circuits after fabrication, the size of the fabric itself must be determined before fabrication. This is difficult, since it is impossible to predict exactly what sorts of debugging circuits will be required.

However, there are several considerations that can guide a designer when choosing a fabric size. The first consideration is the nature of the integrated circuit itself. In a communication circuit, for example, the packet size and packet structure will not change. Based on these quantities, it may be possible to determine a "reasonable" amount of logic that would support operations that count through packets and perform operations on individual words. If a particular encoding scheme is used in the fixed part of the chip, it may be valuable to include enough programmable logic to be able to decode data encoded using this scheme. Although a designer can never know exactly what debug circuits would be required, his or her experience may help determine how much debugging logic is reasonable.

A second consideration is the amount of silicon area available for debug. Chip area is often partitioned early in the design process, and the amount of area devoted to debug logic is set based on detailed analysis of yield versus ease of debugging. In this case, the strategy would simply be to fill the allocated chip area with programmable logic.

Even if the amount of programmable logic is not sufficient to implement a desired debug function, the fabric may still be useful. During the debug process, the fabric would likely be used in conjunction with an on-chip processor or off-chip equipment. A resourceful engineer may find ways of partitioning the debugging functionality across these resources, given the fixed, predetermined amount of programmable logic.

8.2 Access Network

Our methodology assumes one or a small number of embedded programmable logic cores. Thus, it is necessary to collect key signals from across the chip and connect them to the programmable logic core. This is important; the architecture will only be useful if many signals can be routed to the core simultaneously.

We propose that the core be connected to the rest of the integrated circuit using several access networks, as described in Quinton and Wilton [2005]. The network in Quinton and Wilton [2005] contains two levels; the first level consists of a hyper-concentrator and would normally be local to one core in the SoC. The second level combines the signals from each hyper-concentrator and provides them to the programmable logic. The network itself is a concentrator with n inputs and m outputs, and has the property that any subset of size $\leq m$ of the n input signals can be connected to the outputs, but without regard for the order of the output signals. In Quinton and Wilton [2005], it is shown that

this technique can support up to 8000 observable signals in a 20 million gate chip with approximately 2% area overhead.

For our architecture, we extend this technique to two concentrators: one for the bus signals (a bi-directional network can be used to connect both input and output buses) and one network for the fine-grained bit signals that are inputs and outputs of the control block. The bus-based concentrator would route entire buses, as is done inside our architecture. This would reduce the overhead of the network even further.

8.3 Speed Mismatch

Even though we use a datapath-oriented architecture, the maximum speed of the programmable logic will be slower than that of the fixed function ASIC. During debugging, this may mean slowing down the system clock somewhat. After debugging, during normal operation, the clock can run at ASICs speeds, since the embedded fabric would no longer be needed.

In some cases, it may not be desirable to slow down the clock. For example, it is possible that a certain bug may not show itself until the chip is run at full speed. For those sorts of applications, we propose using the interface buffer from Quinton and Wilton [2005]. The interface buffer is a serial-to-parallel converter that can be used to split a single incoming bus into multiple buses within the programmable logic core. Since our architecture is parameterized in the number of input buses, it is straightforward to have wordblocks operating in parallel, doing the same operations on different data words.

9. 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 [Cronquist et al. 1999]. This section compares our architecture to several previous studies, as well as to architectures that have been previously proposed for debugging.

9.1 Alternative Debugging Architectures

Several other embedded debugging architectures have been proposed. Abramovici et al. [2006] have described their reconfigurable design-for-debug infrastructure for SoCs. Like our proposal, this infrastructure is targeted at general-purpose digital logic in a SoC design. Their architecture, however, is based on a distributed heterogeneous reconfigurable fabric. Distributing debug circuitry across the chip has the advantage that the debug logic is likely to be positioned closer to the source of the monitored signals, and eliminates the need for a signal collection network as described in Section 8.2. However, distributed circuitry makes it more difficult to combine the debugging resources to implement larger debugging functions. A centralized scheme like ours can likely support more complex debugging operations, and perhaps can better amortize the cost of the debugging circuitry across different parts of the SoC. Another difference between our architecture and that in Abramovici et al.

[2006] is that ours does not require the identification of specific trigger signals when the debugging circuity is instantiated and connected to the SoC.

Sarangi et al. [2007] have described a proposal for using programmable hardware to help patch design errors in processors. As part of their patching process they make use of programmable logic to detect specific conditions in the processor. In some cases, after the detection of these specific problem conditions, they can make use of existing processor features, such as pipeline flushes, cache refills, or instruction editing to correct the error; in other cases they can cause an exception to be serviced by the operating system or hypervisor. The primary motivation of their proposal is the in-field correction of processor design errors, and not post-silicon debug, however it is clear that their proposal could also be used for post-silicon debug. Because their architecture has been designed with one application in mind, it may not be general enough for implementing debugging circuits useful in other types of integrated circuits.

A similar proposal is described in Wagner et al. [2006]. The focus in that work is on providing a configurable state machine that matches error states in a processor and takes corrective action when these error states occur during system operation. Although this was not designed specifically for post-silicon debugging, it may be helpful in uncovering some types of design errors in a processor. Again, however, it is not as flexible as our architecture in which more general debug circuits can be implemented.

Our previous work [Quinton and Wilton 2005] describes another design-for-debug proposal. In that work, we employ a fine-grained programmable logic core based on a standard FPGA architecture. This previous work leads to perhaps the most general implementation of programmable debug circuitry, but it suffers from the overhead implicit in a fine-grained architecture. In addition, this previous core was not synthesizable, which is a key attribute of the architecture described in this article.

9.2 Fine-Grained Synthesizable Fabric

Although previous fine-grained synthesizable fabrics were not designed with debugging in mind, they could be used for this purpose. We have compared our architecture to a previous synthesizable architecture in Section 6.2 using a set of benchmark circuits. The architecture proposed in [Yan and Wilton 2006] 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 is 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 contribute to large area consumption.

For some other circuits which do not have a large number of multipliers, the area ratio is between 13 and 75. We observe that the architecture in Yan and Wilton [2006] is not efficient when implementing large circuits. The architecture in Yan and Wilton [2006] contains many routing multiplexers. Both the

size of these multiplexers and the number of multiplexers grow linearly with the size of fabric. When the fabric is scaled sufficiently large to implement the given benchmark circuits, these multiplexers become unwieldy and cause the area to grow significantly.

9.3 Datapath-Oriented FPGAs

Several previous studies have considered datapath-oriented FPGAs [Cherepacha and Lewis 1996; Hauck et al. 2004; Leijten-Nowak and van Meerbergen 2003; Ye and Rose 2005; Ye et al. 2003]. In these architectures, configuration bits are shared among multiple lookup-tables and multiple routing switches. Again, these could also be used for debugging.

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. The absence of combinational loops is a key requirement of a synthesizable architecture. Although these architectures can be synthesized (as in Leijten-Nowak and van Meerbergen [2003]), 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 Ye and Rose [2005], 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.

9.4 Coarse-Grained Fabrics

Coarse-grained architectures, in which lookup-tables are replaced by ALUs, have also been described in Cronquist et al. [1999], Goldstein et al. [2000], Marshall et al. [1999], and Singh et al. [2000]. Of these, the RaPiD architecture [Singh et al. 2000] 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 the number of each type of functional unit 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.

10. 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 primary application of such a core is to enable efficient on-chip debugging, but it can also be used to implement small datapath circuits. The proposed architecture features sharing configuration bits, carry chains, directional routing architecture and embedded multipliers. Compared to a previous synthesizable embedded programmable logic core, our architecture is between 7 times and 427 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. We have shown that the delay and power overhead of our architecture is also similar to that of standard FPGAs. A proof-of-concept layout of the core is described in Wilton et al. [2007].

There are two important limitations of these comparisons. First, the fine-grained archtecture was optimized for smaller control-type circuits, and thus is inefficient at implementing larger datapath circuits. If the fine-grained architecture was optimized for our benchmark circuits, the difference between the fine-grained and datapath architecture would be reduced significantly. Second, the fine-grained architecture does not contain embedded multipliers, while the datapath architecture does. If multipliers were added to the fine-grained architecture, the fine-grained architecture would perform better on those circuits that contain multiplication (four of our ten benchmark circuits).

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.

REFERENCES

ABRAMOVICI, M., BRADLEY, P., DWARAKANATH, K., LEVIN, P., MEMMI, G., AND MILLER, D. 2006. A reconfigurable design-for-debug infrastructure for SoCs. In *Proceedings of the ACM IEEE Design Automation Conference*. ACM, New York, 7–12.

ABRAMOVICI, M., STROUD, C., AND EMMERT, M. 2002. Using embedded FPGAs for SoC yield improvement. In *Proceedings of the Design Automation Conference*. 713–724.

CHEN, D., CONG, J., ERCEGOVAC, M., AND HUANG, Z. 2001. Performance-driven mapping for CPLD architectures. In *Proceedings of the ACM International Symposium on Field-Programmable Gate Arrays*. ACM, New York, 39–47.

CHEREPACHA, D. AND LEWIS, D. 1996. DP-FPGA: An FPGA architecture optimized for datapaths. In *Proceedings of the International Conference on VLSI Design*. 329–343.

COMPTON, K. AND HAUCK, S. 2007. Automatic design of area-efficient configurable ASIC cores. *IEEE Trans. Comput. 56,* 5 (May), 662–672.

CRONQUIST, D., FRANKLIN, P., FISHER, C., FIGUEROA, M., AND EBELING, C. 1999. Architecture design of reconfigurable pipelined datapaths. In *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*. 23–40.

FLETCHER, J. 1982. An arithmetic checksum for serial transmissions. *IEEE Trans. Commun. COM-30,* 1 (Jan), 247–252.

GOLDSTEIN, S., SCHMIT, H., BUDIU, M., CADAMBI, S., MOE, M., AND TAYLOR, R. 2000. Piperench: A reconfigurable architecture and compiler. *IEEE Comput. 33,* 4 (Apr), 70–77.

HAUCK, S., FRY, T., HOSLER, M., AND KAO, J. 2004. The Chimera Reconfigurable functional unit. *IEEE Trans. VLSI 12,* 2 (Feb.), 206–217.

HO, C., LEONG, P., LUK, W., WILTON, S., AND LOPEZ-BUEDO, S. 2006. Virtual embedded blocks: A methodology for evaluating embedded elements in FPGAs. In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines*. 35–44.

HOLLAND, M. AND HAUCK, S. 2007. Automatic creation of domain-specific reconfigurable CPLD for SoC. *IEEE Trans. Comput.-Aid. Des. Integrat. Circ. Syst. 26,* 2 (Feb), 291–295.

KUON, I. AND ROSE, J. 2007. Measuring the gap between FPGAs and ASICs. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. 26,* 2 (Feb), 203–215.

LEIJTEN-NOWAK, K. AND VAN MEERBERGEN, J. L. 2003. An FPGA architecture with enhanced datapath functionality. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 195–204.

MARSHALL, A., STANSFIELD, T., KOSTARNOV, I., VUILLEMIN, J., AND HUTCHINGS, B. 1999. A reconfigurable arithmetic array for multimedia applications. In *Proceedings of the ACM International Symposium on Field-Programmable Gate Arrays*. ACM, New York, 135–143.

MENEZES, A., VAN OORSCHOT, P., AND VANSTONE, S. 1996. *Handbook of Applied Cryptography*. CRC Press, 602–606.

NAKASSIS, T. 1988. Fletcher's error detection algorithm: How to implement it efficiently and how to avoid the most common pitfalls. *ACM Comput. Commun. Rev. 18,* 5 (Oct), 86–94.

PADALIA, K., FUNG, R., BOURGEAULT, M., EGIER, A., AND ROSE, J. 2003. Automatic transistor and physical design of FPGA tiles from an architecture specification. In *Proceedings of the ACM International Conference on FPGAs*. ACM, New York, 164–172.

QUINTON, B. AND WILTON, S. 2005. Post-silicon debug using programmable logic cores. In *Proceedings of the International Conference on Field-Programmable Technology*. 241–247.

SARANGI, S., NARAYANASAMY, S., CARNEAL, B., TIWARI, A., CALDER, B., AND TORRELLAS, J. 2007. Patching processor design errors with programmable hardware. *IEEE Micro 27,* 1 (Jan-Feb), 12–25.

SARANGI, S., TIWARI, A., AND TORRELLAS, J. 2006. Pheonix: Detecting and recovering from permanent processor design bugs with programmable hardware. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. ACM, New York, 26–37.

SINGH, H., LEE, M., LU, G., KURDAHI, F., BAGHERZADEH, N., AND CHAVES, E. 2000. Morphosys: An integrated reconfigurable system for data-parallel and compute intensive applications. *IEEE Trans. Comput. 49,* 5 (April), 465–481.

WAGNER, I., BERTACCO, V., AND AUSTIN, T. 2006. Shielding against design flaws with field repairable control logic. In *Proceedings of the Design Automation Conference*. 344–347.

WILTON, S., HO, C., LEONG, P. H., LUK, W., AND QUINTON, B. 2007. A synthesizable datapath-oriented embedded fpga fabric. In *Proceedings of the ACM International Symposium on Field-Programmable Gate Arrays*. ACM, New York, 33–41.

WILTON, S., KAFAFI, N., WU, J., BOZMAN, K., AKEN'OVA, V., AND SALEH, R. 2005. Design considerations for soft embedded programmable logic cores. *IEEE J. Solid-State Circ. 40,* 2 (Feb.), 485–497.

YAN, A. AND WILTON, S. 2006. Product-term based synthesizable embedded programmable logic cores. *IEEE Trans. VLSI 14,* 5 (May), 474–488.

YE, A. AND ROSE, J. 2005. Using bus-based connections to improve field-programmable gate array density for implementing datapath circuits. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 3–13.

YE, A., ROSE, J., AND LEWIS, D. 2003. Architecture of datapath-oriented coarse-grain logic and routing for FPGAs. In *Proceedings of the IEEE Custom Integrated Circuits Conference.* IEEE Computer Society Press, Los Alamitos, CA, 61–64.

Received May 2007; revised September 2007; accepted December 2007