Virtual Embedded Blocks: A Methodology for Evaluating Embedded Elements in FPGAs

C.H. Ho, P.H.W. Leong and W. Luk
Department of Computing
Imperial College London, UK
{cho, phwl, wl}@doc.ic.ac.uk

S.J.E. Wilton
Department of Electrical and Computer Engineering
University of British Columbia, Canada
stevew@ece.ubc.ca

S. Lopez-Buedo
Escuela Politecnica Superior,
Universidad Autonoma de Madrid, Spain
sergio.lopez-buedo@uam.es

Abstract

Embedded elements, such as block multipliers, are increasingly used in advanced field programmable gate array (FPGA) devices to improve efficiency in speed, area and power consumption. A methodology is described for assessing the impact of such embedded elements on efficiency. The methodology involves creating dummy elements, called Virtual Embedded blocks (VEBs), in the FPGA to model the size, position and delay of the embedded elements. The standard design flow offered by FPGA and CAD vendors can be used for mapping, placement, routing and retiming of designs with VEBs. The speed and resource utilisation of the resulting designs can then be inferred using the FPGA vendor's timing analysis tools. We illustrate the application of this methodology to the evaluation of various schemes of involving embedded elements that support floating-point computations.

1. Introduction

As field programmable gate array (FPGA) technology matures, a trend is to include coarse grained embedded elements such as memories, multipliers, digital signal processing (DSP) blocks, microprocessors and networking components as a means of improving performance and reducing area and power consumption. This trend of including embedded elements with dedicated functional units is becoming a central issue in FPGA research [1].

Introducing embedded elements is a way to support domain-specific customisation of the FPGA fabric, and it involves a trade-off between flexibility and specialisation. For example, while an embedded floating-point unit may have the best possible speed and power consumption performance, the silicon area is wasted if floating-point is not required in the application. In contrast, lookup table (LUT) based programmable logic has the highest flexibility, but an equivalent design occupies much larger area and has significantly reduced speed compared with an embedded very large scale integration (VLSI) implementation.

In this work, we present a device and vendor independent methodology for rapid assessment of the effects of adding embedded elements to an existing FPGA architecture. The key element of our methodology is to adopt virtual embedded blocks (VEBs), created from the FPGA's logic resources, to model the placement and delay of the embedded block to be included in the FPGA fabric. Using this method, the benefits of incorporating embedded elements in improving application performance and reducing area usage can be quickly evaluated, even if an actual implementation of the element is not available.

To summarise, the main contributions of this paper are:

- A methodology which allows existing high and low-level tools to be used to study the effects of embedded elements in FPGAs.
- An illustration of this methodology based on a modifiable compiler and commercial tools to model embedded elements using VEBs.
- An assessment of the accuracy of this framework by modelling existing embedded elements in FPGAs over various applications.
- An exploration of technology trends involving embedded elements based on systematic variation of VEB parameters in applications.

The remainder of the paper is organised as follows. Section 2 covers background material and related work. Section 3 describes the generic aspects of our methodology involving VEBs as a means of modelling FPGA resources, while section 4 explains how this methodology can be supported using vendor-specific tools such as those from Xilinx and Synplicity. Section 5 introduces the benchmarks that we use. Section 6 discusses our experimental results, and section 7 draws conclusions.

2. Background

There has been research on the effects of coarse grained components in an FPGA fabric to form a heterogeneous device. However much of the reported work concerns major architectural changes to the logic and routing blocks, rather than introducing embedded blocks to existing devices.

Leijten-Nowak and van Meerbergen [2] proposed mixed-level granularity logic blocks and compared their benefits with a standard island-style FPGA using the Versatile Place and Route tool (VPR) [3]. Ye, Rose and Lewis [4] studied the effects of coarse grained logic cells and routing resources for datapath circuits, also using VPR.

Beck revised VPR to explore the effects of introducing hard macros [5], while Beauchamp et. al. augmented VPR to assess the impact of embedding floating-point units in FPGAs [6]. However, we are not aware of studies concerning the effect of adding embedded blocks to existing commercial FPGA devices, nor of methodologies to facilitate such studies.

From the above description, it is clear that the VPR tool has often been used in research concerning FPGA architectures. VPR is open source software and has the associated advantage that arbitrary modifications can be made to it. Moreover, it is specifically designed for architectural exploration.

Compared with VPR, an approach that can use vendor's tools and devices offers the following benefits:

- Since our methodology involves advanced commercial tools targeting a real FPGA device with the latest technology, it enables a tighter integration between the VEBs and the associated reconfigurable fabric than VPR. This capability means that the modelled architecture can match very closely with that of existing devices. In contrast, the island-style FPGA architectural model in VPR is a relatively crude approximation to existing commercial FPGAs. Moreover, it is difficult to model logic and routing delays, particularly for the latest FPGAs, due to the proprietary nature of commercial FPGA architectures.

- Commercial quality synthesis tools such as Synplicity's Synplify can be used. In particular, advanced optimising features such as retiming are not available in standard VPR based design flows. This feature is used extensively in our study.

- The accuracy of our methodology can be evaluated by comparing real embedded blocks (such as embedded multipliers) and their corresponding VEB models. We shall illustrate this evaluation in section 6.

Of course, there are various VPR experiments that are not supported by the proposed approach. In particular, since we are dealing with a real FPGA and the associated tools, we cannot change the FPGA fabric such as: the architecture of the lookup table in the FPGA cells, the number of cells, or the amount of routing resources.

3. Methodology: Generic Aspects

In this section, the methodology is first described as a generic approach which can be applied to any FPGA and the associated design tools. The next section will cover the actual vendor-specific design flow used in this study.

We shall first provide an overview of our methodology that supports rapid generation of various benchmark applications to target reconfigurable architectures with VEB models. A modifiable compiler, called *fly* [7], is used so that different wordlength and back-end operator instances can easily be produced from a single algorithmic description. This allows both fixed- and floating-point implementations to be generated from the same description. We apply this methodology to a set of benchmark circuits generated in this fashion.

To measure the accuracy of this approach, block multipliers are modelled using VEBs and compared with FPGAs having this feature. A study of the benefits of double-precision floating-point embedded blocks is also made. Using this approach, the speedup of an application as a function of the speed of the embedded block can be easily quantified, and these studies are made for some of the benchmarks. Power consumption is not considered in this study.

In the descriptions that follow, we use the term logic cell (LC) for the smallest logic unit in the FPGA (usually a lookup table plus a register) and configurable logic block (CLB) for an array of LCs that are interconnected via the connection and switch blocks in the FPGA.

The basic strategy employed is to use the logic resources of a real FPGA to match the expected position, area and delay of an application specific integrated circuit (ASIC) implementation of an embedded block (EB). This could be achieved using appropriate vendor's tools or generic tools such as VPR [3]. In order to estimate its performance, the EB is modelled using logic cell resources in VEBs. Our model of an FPGA with EBs is called a virtual FPGA as illustrated in figure 1.

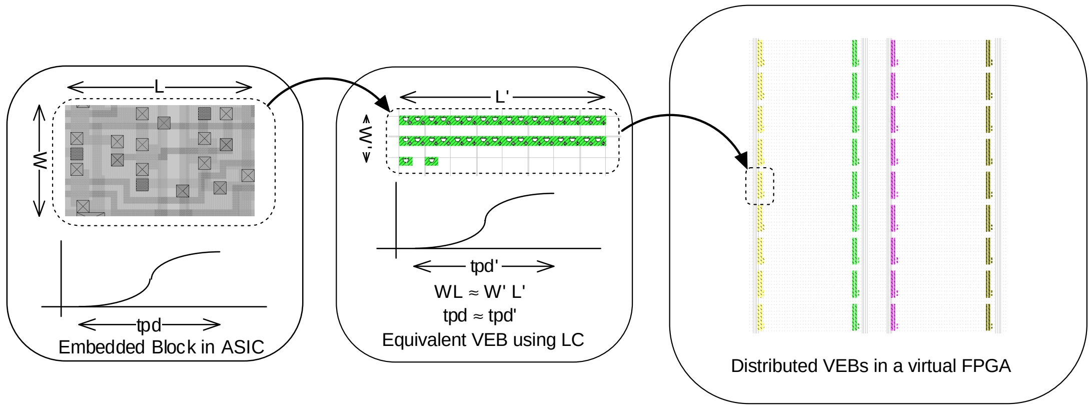

Figure 1. Modelling embedded elements in FPGAs using Virtual Embedded Blocks.

To employ this methodology, an area and delay model for the EB is required. The model should provide a high level estimate of the area and delay of the block, extracted from simulations of an existing design or come from measurements of an actual ASIC. The area model is translated into equivalent logic cell resources in the virtual FPGA. In order to make this translation, an estimate of the area of a logic cell in the FPGA is required. All area measures are normalised by dividing the actual area by the square of the feature size, making the area estimates independent of feature size. The VEB utilisation can then be computed as the normalised area of the EB divided by the normalised area of a logic cell. This value is in units of equivalent logic cells and the mapping encourages thinking about EBs in terms of FPGA resources. Table 1 shows a number of logic cell area estimates. The area estimate of the embedded blocks studied are given in section 6. We assume that there are sufficient ports to allow interconnection of the EB to the routing fabric. This may not be the case in some designs, particularly those with small EBs.

In order to accurately model delay, both the logic and wiring delay of the virtual FPGA must match that of the FPGA. The logic delay can be matched by introducing delays in the VEB which are similar to those of the EB. In the case of very small EB/VEBs, it may not be possible to accurately match the number of ports, area or logic delay and some inaccuracies will result. A complex EB might have many paths, each with different delays. It is possible to either assume that all delays are equal to the longest one (i.e. the critical path), or generate different delays for important paths. In the latter case, shorter delays can be obtained by taking intermediate points along the longest delay path.

Modelling wiring delays is more problematic, since the placement of the virtual FPGA must be similar to that of an FPGA with EBs so that their routing is similar. This requires that:

- The absolute location of VEBs match the intended locations of REBs in the FPGA with EBs.
- The design tools be able to assign instantiations of VEBs in the netlist to physical VEBs while minimising routing delays.

The first requirement is addressed by locating VEBs at predefined absolute locations that match the floorplan of the FPGA with EBs. The assignment of physical VEBs is currently made by manually specifying its placement. Automated methods will be the subject of a later study.

4. Methodology: Vendor Specific Aspects

This section illustrates how a VEB can be used to model a real embedded multiplier block in Virtex II device as a case study. All of the results described in this work are obtained using the Synplicity Synplify Pro 8.0 synthesis tool, the Xilinx ISE 7.1i design tools, and the Xilinx Virtex II XC2V6000-6-FF1152 FPGA device.

4.1. VEB Delay and Area model

While the ports for the VEB must be the same as those of the real embedded block, the VEB logic delay is emulated using a dummy circuit in the VEB implementation. Although many methods are possible, in this study, delays are inserted using adder carry chains for the following reasons:

- Adder carry chains are common to most FPGA platforms, enhancing the portability of the proposed methodology.
- The adder carry chain can be specified as a behavioural description, hence a platform independent delay block can be constructed.

Device	LCs/CLB L	Area/CLB A (μm^2)	Feature Size f (μm)	Normalised LC area ($N = A/Lf^2$)
Apex 20K400E [8]	10	63161	0.18	195,000
Virtex E [8]	4	35462	0.18	267,000
Virtex II 3000 [9]	8	$71,429 \times 0.7$	0.12	434,000
Virtex II 1000 [10]	8	$72,782 \times 0.7$	0.12	442,000

Table 1. Estimates of logic cell area including configuration bit, buffer and interconnect overheads. The Virtex II value of A is based on the estimate that 70% of the total die area is used for logic cells, the other area being for pads, block memories, multipliers etc.

Delay name	Description	delay (ns)
T_{opcy}	F to COUT	0.665
T_{byp}	CIN to COUT	0.084
T_{ciny}	CIN to Y via XOR	0.940
T_{mult}	Embedded Multiplier	4.66
T_{multck}	Registered embedded multiplier	3.000
T_{dyck}	Register setup and hold time	0.293

Table 2. Delay parameters for Virtex II-6 devices.

- It is relatively easy to adjust an adder's carry chain delay by changing its length. This feature is used to model different embedded blocks.

The combinatorial logic delay of an adder carry chain can be modelled by $t_{pd} = T_{opcy} + \frac{N-4}{2} \times T_{byp} + T_{ciny}$, where N is the length of the adder carry chain, T_{opcy} is the combinatorial delay from the input to the COUT output, T_{byp} is the combinatorial delay from CIN to COUT, and T_{ciny} is the combinatorial delay from CIN to the Y output via an XOR gate. If the output is latched, the setup and hold time of a register (T_{dyck}) should be added to this value. Typical values for these parameters in the Virtex II adder carry chain and multiplier block are extracted from vendor's timing analysis tool and given in table 2.

As an example, to model a registered multiplier block with delay of 3 ns, $N = 30$ gives a logic delay (including setup and hold time) of 2.99 ns. In the Xilinx device, the carry chains run along the columns. One issue to note is that the carry chain only runs in a single direction in the device and breaking the carry chain introduces a long wiring delay. In our current approach, a certain amount of trail-and-error is required to achieve a given delay.

For the area model, the normalised LC area for the Virtex-II 1000 in table 1 is used in this study.

4.2. Integration of VEB into toolchain

In order to produce a VEB, it is first synthesised from a hardware description language (HDL) description. Features in the synthesis tool for regular design flows such as automatic I/O block insertion, pipelining and retiming are disabled. The resulting netlist is placed and routed using the vendor's toolchain. Area constraints must be specified to force the placement of the VEB in a rectangular block. The "trim unconnected logic" option is disabled to ensure that the VEB is not optimised away. After place and route, another constraint file which contains the actual placement information for each LC in the VEB is generated. The placement information and the netlist of the VEB is compiled to create a relationally placed macro (RPM).

To employ the VEB in an application, its HDL description is modified to instantiate the corresponding VEB block. Since the VEB is considered as a black box during synthesis, timing information must also be specified to allow the synthesis tool to take timing of the block into account during optimisation. This makes optimisations such as retiming possible.

During place and route, the VEBs are placed in a regular locations on the FPGA, modelling the expected locations of the EBs. This is achieved using placement constraints. The design is then placed and routed in the usual fashion. The delays introduced in the VEB model the logic delay and its placement means that realistic routing is required. The vendor's tools are used to obtain resource utilisation delay information about the circuit.

5. Benchmark Circuits

To evaluate the effect of including VEBs in real applications, a set of datapath-intensive benchmark applications are used. Note that only fixed-point versions of the unsigned multiplier and BGM benchmarks are available since, for the former, it is not possible to combine floating-point multipliers in the same way and, for the latter, the design is only available in fixed-point form.

Benchmark	Pipelined	# adders	# multipliers
dscg	Y	2	4
ode	N	3	2
mm3	N	2	3
fir4	Y	3	4
bfly	Y	4	4
mul34*	Y	-	4
mul68*	Y	-	16
mul136*	Y	-	64
bgm*	Y	-	46

Table 3. Number of operators used in the benchmarking circuits as well as their circuit type. Benchmarks with a * superscript are fixed-point only.

All other benchmarks are generated using a rapid prototyping approach with a modifiable compiler called *fly* [7]. This tool is used to produce both fixed-point and floating-point benchmarking circuit from a single description, facilitating the generation and debugging of benchmarks with different arithmetic systems (fixed and floating-point) and wordlengths (18-bit and 64-bit). In addition, *fly*'s distributed control scheme [11, 7] can be easily modified to cope with operators having different latencies.

In this section, the benchmark circuits employed this study are first described. This is followed by a description of the IEEE 754 compliant double precision library used to implement the operators for the benchmarks.

The floating-point benchmarks are generated by implementing the applications using double precision floating-point with round-to-nearest-even rounding mode and exception signals being ignored. The applications further assume the input data comes from an off-chip memory. Table 3 summarises the resource usage of the benchmarking circuits and indicates whether the circuit is fully pipelined (accepts an input and produces an output every cycle). If a pipelined implementation is not possible due to dependency or circuit size constraints, an iterative implementation is made. Note that for an non-pipelined implementation, pipelined operators require multiple cycles and hence affect the total number of cycles required to complete the benchmark.

5.1. Benchmark Descriptions

5.1.1. Digital Sine-Cosine Generator

The digital sine-cosine generator (dscg) [12] has a number of applications, such as the computation of the discrete Fourier transform and in certain digital communication systems, such as in future Hiperlan systems for high performance wireless communications. Let $s1_n$ and $s2_n$ denote the two outputs of a digital sine-cosine generator, the outputs at the next sample can be computed using the following formula:

$$\begin{bmatrix} s1_{n+1} \\ s2_{n+1} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \cos(\theta)+1 \\ \cos(\theta)-1 & \cos(\theta) \end{bmatrix} \begin{bmatrix} s1_n \\ s2_n \end{bmatrix} \qquad (1)$$

5.1.2. Ordinary Differential Equation

Many scientific problems involve the solution of ordinary differential equations (ODEs). An ODE solver (ode) is implemented as part of the floating-point benchmarks. The benchmark circuit solves the ODE [13]:

$$\frac{dy}{dt} = \frac{(t-y)}{2} \text{ over } t \in [0,3] \text{ with } y(0) = 1 \qquad (2)$$

using the Euler method. The trajectory of y is given by the difference equation $y_{k+1} = y_k + h\frac{(t_k-y_k)}{2}$ and $t_{k+1} = t_k + h$, where h is the step size. The ODE solver takes h as an input parameter and returns the value of y. Due to dependencies, this circuit cannot be fully pipelined and is hence implemented in an iterative fashion. Pipelined operators are used by waiting for the output so low latency has an advantage for this benchmark.

5.1.3. Matrix Multiplication

Matrix multiplication is used frequently in many signal processing and scientific applications. A 3×3 matrix multiplication application benchmark (mm3) is developed. The implementation sequentially computes 9 vector dot-products and each dot-product is computed sequentially with 3 multiplies and 2 additions.

5.1.4. FIR Filter

Digital filtering is another common application and we have implemented a 4-tap finite impulse response filter (fir4), which implements the equation $y_i = \sum_{j=0}^{4} k_j x_{i-j}$ where x_i is the input of the filter, k_i is the filter window and y_i is the output.

5.1.5. Butterfly

The fast Fourier transform (FFT) is another important signal processing primitive. The FFT is composed from butterfly operations which compute $z = y + x \times w$, where x and y are the inputs from previous stage and w is a twiddle factor. All values are complex numbers, therefore each multiplication involves 4 multipliers and 2 adders (bfly). A state machine is implemented to control the dataflow of the circuits. Figure 2 illustrates the datapath of a single butterfly which is used as the benchmark circuit.

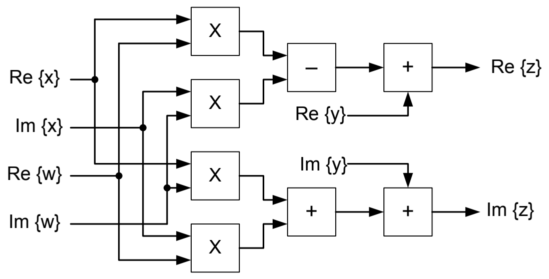

Figure 2. One butterfly stage in an FFT.

5.1.6. Unsigned Multiplier

Three parallel unsigned multipliers of size 34-bit (mul34), 68-bit (mul68) and 136-bit (mul136) are implemented by using 18×18 fixed-point multipliers as building blocks, and summing their outputs appropriately. They use 4, 16 and 34 18×18 embedded multipliers respectively. These are used to test the modelling of embedded multipliers and only used for fixed-point benchmarks.

5.1.7. BGM

The datapath of a design to compute Monte Carlo simulations of interest rate model derivatives priced under the Brace, Gątarek and Musiela (BGM) framework is used as the final test circuit (bgm) [14, 15]. Denote $F(t,t_n,t_{n+1})$ as the forward interest rate observed at time t for a period starting at t_n and ending at t_{n+1}. Suppose the time line is segmented by the reset dates $(T_1,T_2,...,T_N)$ (called the standard reset dates) of actively trading caps on which the BGM model is calibrated. In the BGM framework, the forward rates $\{F(t,T_n,T_{n+1})\}$ are assumed to evolve according to a log-normal distribution. Writing $F_n(t)$ as the shorthand for $F(t,T_n,T_{n+1})$, the evolution follows the stochastic differential equation (SDE) with d stochastic factors:

$$\frac{dF_n(t)}{F_n(t)} = \vec{\mu_n}(t)dt + \vec{\sigma}_n(t)\cdot d\vec{W}(t) \qquad n{=}1\ldots N. \tag{3}$$

In this equation, dF_n is the change in the forward rate, F_n, in the time interval dt. The drift coefficient, $\vec{\mu_n}$, is given by

$$\vec{\mu_n}(t) = \vec{\sigma}_n(t)\cdot\sum_{i=m(t)}^{n}\frac{\tau_i F_i(t)\vec{\sigma}_i(t)}{1+\tau_i F_i(t)} \tag{4}$$

where $m(t)$ is the index for the next reset date at time t and $t \le t_{m(t)}$, $\tau_i = T_{i+1} - T_i$ and σ_n is the d-dimensional volatility vector. In the stochastic term (the second term on the right hand side of Equation 3), $d\vec{W}$ is the differential of a d-dimensional uncorrelated Brownian motion $\vec{W}$, and each component can be written as $dW_k(t) = \epsilon_k\sqrt{dt}$ where ϵ_k is a Gaussian random number drawn from a standardised normal distribution, i.e. $\epsilon \sim \phi(0,1.0)$.

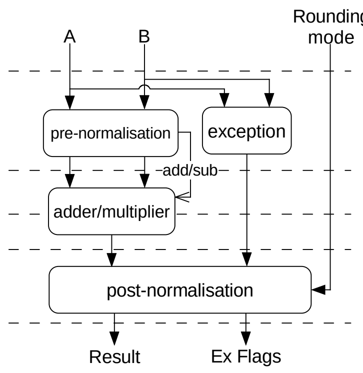

Figure 3. Simplified datapath for the floating-point adder/multiplier.

5.2. Floating-Point Library

A library of floating-point operators is developed in the Verilog HDL, based on a heavily modified open source floating-point library [16]. Extra pipeline registers are added to the original design to improve performance. Different floating-point operators are extracted as a single top-level entity, where the original design is a single floating-point unit (FPU) entity. Moreover, the library is modified to support arbitrary size of exponent and significand, and both single and double-precision libraries are verified using a common floating-point testbench generation scheme. This library is fully-compliant with the IEEE754 [17] standard, supporting all 4 rounding modes, subnormal numbers and exceptions.

5.2.1. Floating-Point Adder

A simplified block diagram of the floating-point adder and multiplier is shown in figure 3. The dotted-lines indicate the location of pipeline registers. In the pre-normalisation stage, the inputs are registered and the exponents compared. Inputs are swapped if necessary. The significands are shifted right for alignment and a mode which indicates the operation to be performed (either addition or subtraction) is checked. The most expensive circuits in the adder are two barrel shifters used in the pre and post-normalisation blocks.

Special inputs such as subnormals, infinity and not a number (NaN) are handled in the exception handling block. Flags (subnormal, zero, infinity, NaN) are set according to the combination of inputs. This circuit only requires several comparators. The addition block takes the output from the pre-normalisation block, in which the data has been properly aligned and adds or subtracts the numbers according to the operation mode.

The post-normalisation block is the most complicated part of the adder. A priority encoder takes the addition block output and determines the number of leading zeros. The exponent is then adjusted and the significand left-shifted. Different rounding schemes are enforced according to the input to produce final result. Exception flags such as inexact number, overflow, underflow are generated based on the final result.

5.2.2. Floating-Point Multiplier

In the pre-normalisation stage, the intermediate exponent is determined by adding the input exponents. Hidden bits of the significands are recovered and attached to the significands based on the exponent values.

The multiplication block takes the output from the pre-normalisation block (including hidden bits) and the significands are multiplied. The result is then sent to the post-normalisation block. The multiplier circuit consumes most of the resources in this block.

The exception and post-normalisation blocks are similar to those in the floating-point adder.

5.2.3. Floating-Point Verification

To verify that the floating-point operators are compliant with the IEEE 754 standard, an open-source program called *TestFloat-2a* [18] is employed. Tests are made up of simple pattern tests intermixed with weighted random inputs for the floating-point operators. The "level 1" test in *TestFloat-2a* covers all 4 rounding modes, and all boundary cases of given arithmetic, including underflows, overflows, invalid operations, subnormal inputs, zeros (positive and negative), infinities (positive and negative), and NaNs. Using this test and ModelSim 5.7d, our library is simulated using more than half a million test cases and no errors are found.

6. Results

6.1. Verification of the VEB Approach

In order to verify the results obtained using our methodology, we develop a VEB for an embedded 18×18 multiplier (EM). As such multipliers are found in Virtex II devices, it is possible to compare the routing and logic delays of benchmark circuits from the VEB approach with those given by the actual EMs.

To estimate the normalised area of an EM in Virtex II, we assume that they occupy a total of 2% of the die area which, in turn, is reported to be 93 mm^2 [10]. This translates to a normalised LC area of approximately 2,751,000, which is 6 LCs. The timing information is extracted from the data sheet of the device; the relevant parameters are shown in table 2.

The benchmark circuits are implemented both using the EMs and the VEB multiplier. Table 4 summarises the resource utilisation and critical path delay for both implementations. Let us first compare the critical path delay, which is usually the parameter of most interest to a designer since it determines the maximum clock frequency at which the circuit can be operated. As one can see from the table, the difference between the two approaches is at most 11%. For most of the circuits, the critical path would involve the multiplier. In those cases where it is not, the longest delay through the multiplier is very close to the critical path of the circuit.

For the bgm benchmark, table 4 shows that a speedup of 1.2 is gained by retiming. In designs where the stages are not as well balanced, as is often the case when a VEB is introduced, more dramatic speedups are often observed. The retiming feature is absent from most VPR based design flows [3].

Table 5 shows the breakdown of the critical path into logic and routing delays for the EM implementation. The corresponding path in the VEB implementation is identified and shown in the same table. The sum of the logic and routing delay for the EM should be equal to the corresponding value in table 4, but due to clock skew it is slightly different. The logic delays between the two implementations are very similar. The routing delays differ greatly because the EM and VEB implementations often have different placement, but since the nets are not on the critical path in the VEB implementation, they do not affect the maximum operating frequency of the circuit. It would be possible to also match the routing delays by locking placement of all of the LCs in the design rather than just the VEB, if closer matching of the routing delays is desired.

For the bgm circuit with retiming enabled, there is no corresponding path between the EM and VEB implementation because the registers are moved during this optimisation. The critical path of the VEB implementation is shown in this case and the difference column left blank.

6.2. Faster Embedded Multipliers

The VEB approach can be used to (a) obtain a single performance estimate for introducing embedded blocks, (b) analyse performance/area trade-offs, and (c) determine the EM speed required to meet a given system performance. To illustrate this point, we measure the bgm performance over a range of VEB delays. Retiming is used in such experiments since, for pipelined designs, improving the performance of one pipeline stage can create slack in another stage, moving the bottleneck to a different stage of the pipeline. A similar situation occurs in multicycle designs.

The results are shown in figure 4. An EM performance

Benchmark	Size (slices)	# of EMs	EM delay (ns)	VEB delay (ns)	Difference (ns)	Difference (%)
dscg	177	4	4.599	4.981	0.382	8%
fir4	193	4	4.616	4.704	0.088	2%
ode	204	2	4.402	4.539	0.137	3%
mm3	469	3	4.859	4.815	0.044	1%
bfly	629	4	5.668	5.224	0.444	8%
mul34	141	4	11.191	11.287	0.096	1%
mul68	604	16	12.553	14.099	1.546	11%
mul136	2426	64	14.632	13.248	1.384	10%
bgm	2315	46	14.055	13.866	0.189	1%
bgm*	2205	46	11.594	11.602	0.008	0%

Table 4. Summary of resource utilisation and critical path delay for embedded multiplier (MULT18X18) and VEB implementations. A * indicates that retiming is enabled during synthesis.

Benchmark	EM delay		Equivalent VEB path delay		Difference			
	logic (ns)	routing (ns)	logic (ns)	routing (ns)	logic (ns)	logic (%)	routing (ns)	routing (%)
dscg	3.449	1.15	3.445	1.536	0.004	0.116%	0.386	25%
fir4	3.449	1.167	3.445	0.815	0.004	0.116%	0.352	43%
ode	3.449	0.911	3.445	0.672	0.004	0.116%	0.239	36%
mm3	3.449	1.366	3.445	1.067	0.004	0.116%	0.299	28%
bfly	3.449	2.062	3.445	1.411	0.004	0.116%	0.651	46%
mul34	8.818	2.345	8.99	2.202	0.172	1.913%	0.143	6%
mul68	8.682	3.687	8.99	4.96	0.308	3.426%	1.273	26%
mul136	8.682	5.95	8.99	4.258	0.308	3.426%	1.692	40%
bgm	10.119	3.901	10.019	1.916	0.1	0.998%	1.985	104%
bgm*	8.439	3.155	7.631*	3.971*	n/a	n/a	n/a	n/a

Table 5. Breakdown of critical path delay for embedded multiplier and VEB implementations. A * indicates that retiming is enabled during synthesis.

of 1 is the same as the performance of the Xilinx EM, and a normalised system performance of 1 corresponds to the execution time of the bgm benchmark. From this figure, one can determine the maximum speedup that can be achieved in this application via faster EMs to be approximately 1.4, which can be obtained by speeding up the block multiplier in Virtex II devices by 2.2 times.

As an example of estimating system performance of a design fabricated in a different process technology, consider a 16×16 bit combinational multiplier operating at 1 GHz with an area of 0.474 mm^2 at 1.3 V in 90 nm technology [19]. Assuming velocity saturated general scaling of transistor lengths from 90 nm to 0.13 μm ($1/S = 0.13/0.09$), the delay would scale by $1/S$, i.e. from 1 ns to 1.44 ns [20]. The scaled area of the implementation would be 132 LCs. Such an implementation is thus 1.44 times faster but uses 3.6 times more area than the Xilinx EM, and improves bgm performance by 15%.

Operator	size (LCs)	# of EMs	Latency	Delay (ns)
FP Adder	3554	0	5	7.465
FP Multiplier	4300	9	5	13.197
FPU (VEB)	570	0	1	7.151

Table 6. FPGA implementation results for floating-point operators, where FPU(VEB) indicates the equivalent ASIC implementation of FPU using VEB approach. The FPU(VEB) is 6 times smaller than the floating-point adder and has only one clock cycle latency.

6.3. Embedded Floating-Point Unit

An FPGA implementation of a double-precision FPU is made by synthesising the floating-point library in section 5.2 targeting Virtex II technology. The size and performance of the adder and multiplier in this FPU are shown in table 6.

The area and delay model of a VEB floating-point unit (FPU) is made based on area and speed estimates of the Blue Gene ASIC [21, 22]. This is a state-of-the art FPU fabricated in a similar technology ($0.13\mu m$) to the Xilinx Virtex II. It operates at a clock frequency of 700 MHz, with an area estimated to be 4.26 mm^2 [21] which translates to 570 LCs. The area estimate is very conservative, since this FPU is much more sophisticated than the one used for the FPGA implementation.

Since the Blue Gene 700 MHz FPU design has a much smaller logic delay than the routing delay of the FPGA, a better implementation can be obtained by reducing both its latency and clock frequency by a factor of 5. Thus the VEB FPU considered has a clock frequency of 140 MHz with a one cycle latency. This essentially trades off clock frequency for reduced latency.

The performance of the Virtex II FPGA is compared to a virtual FPGA with embedded FPUs using the floating-point benchmarks. A summary of the results is given in table 7. As one can see, augmenting the FPGA with embedded FPUs leads to a mean improvement in area and delay by factors of 3.7 and 4.4 respectively. In contrast, a recent investigation of embedding double-precision FPUs in FPGAs based on VPR with a different set of benchmarks results in estimates of average area savings of 55.0% and average increase of 40.7% in clock rate over existing architectures [6]. We attribute the differences to: different benchmarks being used; CAD tools; FPU delay and latency; FPGA model; and our use of retiming optimisations during synthesis.

Note that, for instance in the case of the ode benchmark, one can potentially support 3.8 times more dedicated FPUs in the same area as FPUs from programmable resources, meaning that more instances of the design can operate in parallel. Hence in the limit, the system throughput can be improved by up to 40 times if we include both improvement in speed and in parallelism due to area reduction.

Dedicated FPUs are wasted resources for designs that do not make use of them; however, each FPU occupies approximately the same area as 72 CLBs, which translates to 0.9% of the chip area of an XC2V6000 device.

6.4. Impact of Embedded Block Performance

Experiments are conducted, similar to those in section 6.2, to assess the impact of embedded block performance on system performance. Specifically, we study the speedup of the bfly benchmark as a function of the FPU performance (figure 5), normalised to the speed of the Blue Gene model described in section 6.3. It can be seen that a modest improvement in FPU speed can lead to a large improvement in the bfly benchmark: for instance improving the FPU performance by 30% improves bfly performance by 40%. Beyond a factor of 1.4, the speedup of the benchmark increases rather more slowly. This type of information can be used to determine the best option for ASIC implementations of EBs in which the synthesis tools offer a wide range of possible area/delay trade-offs.

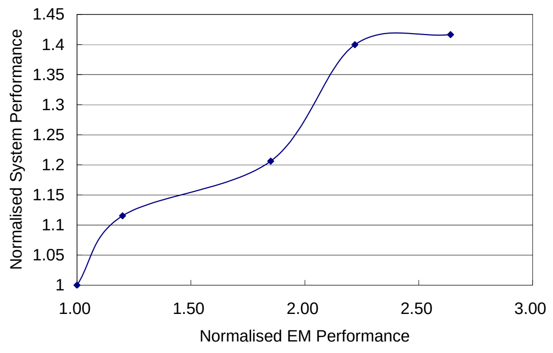

Figure 4. Performance of fixed-point bgm benchmark for different VEB delays, with retiming.

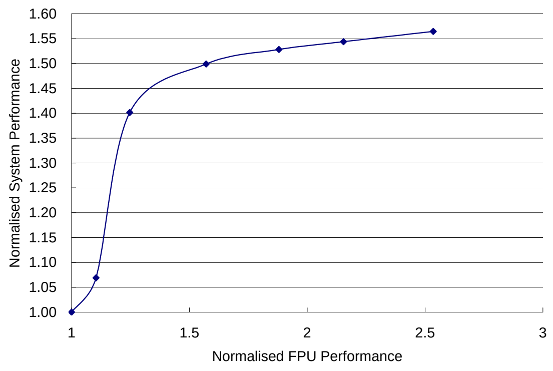

Figure 5. Performance of floating-point bfly benchmark for different FPU delays, with retiming.

7. Conclusion

We propose a methodology for estimating the effects of introducing embedded blocks to existing FPGA devices. The methodology is evaluated by modelling block multipliers in Xilinx Virtex II devices, and we find that prediction of critical paths to approximately 10% accuracy can be achieved. The methodology is then applied to predict the impact of embedded floating-point units, showing a possible reduction in area of 3.7 times and speedup of 4.4 times. Current and future work includes refining our VEB-based tools to support, for instance, better modelling of the interconnection between VEBs and the routing fabric; VEB-aware technology mapping and power consumption estimation; exploring ways of combining our methodology

	FPGA				VEB				Reduction Factor	
	EMs	throughput (# of cycle)	size (LC)	delay (ns)	FPUs	throughput (# of cycle)	size (LC)	delay (ns)	Area	Delay
dscg	36	1	19006	22.711	6	1	3420 + 940	8.807	4.4	2.6
fir4	36	1	20590	23.545	7	1	3990 + 996	9.539	4.1	2.5
ode	18	20	13984	17.756	5	4	2850 + 870	8.525	3.8	10.4
mm3	27	225	17236	19.320	5	45	2850 + 2390	8.587	3.3	11.3
bfly	36	1	25640	20.245	8	1	4560 + 3424	8.821	3.2	2.3
								Geometric Mean:	3.7	4.4

Table 7. FPGA implementation results for floating-point benchmark applications. The VEB size is given as the FPU area (in equivalent LC resources) plus the LC resources needed to implement the rest of the circuit.

with related tools such as VPR to provide a comprehensive framework for exploring and developing next-generation FPGA architectures; and extending the set of benchmarks for evaluating our approach.

References

[1] J. Rose, "Hard vs. soft: The central question of pre-fabricated silicon," in *34th International Symposium on Multiple-Valued Logic (ISMVL'04)*, May 2004, pp. 2–5.

[2] K. Leijten-Nowak and J. L. van Meerbergen, "An FPGA architecture with enhanced datapath functionality," in *Proc. FPGA '03*, ACM Press, 2003, pp. 195–204.

[3] V. Betz, J. Rose, and A. Marquardt, Eds., *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[4] A. Ye, J. Rose, and D. Lewis, "Architecture of datapath-oriented coarse-grain logic and routing for FPGAs," in *CICC '03: Proceedings of the IEEE Custom Integrated Circuits Conference*, 2003, pp. 61–64.

[5] L. Beck, *A Place-and-Route Tool for Heterogeneous FPGAs*. Distributed Mentor Project Report, Cornell University, 2004.

[6] M. Beauchamp, S. Hauck, K. Underwood, and K. Hemmert., "Embedded floating point units in FPGAs," in *Proc. FPGA '06*, ACM Press, 2006.

[7] C. Ho, P. Leong, K. H. Tsoi, R. Ludewig, P. Zipf, A. Ortiz, and M. Glesner, "Fly - a modifiable hardware compiler," in *Proc. FPL*. LNCS 2438, Springer, 2002, pp. 381–390.

[8] K. Padalia, R. Fung, M. Bourgeault, A. Egier, and J. Rose, "Automatic transistor and physical design of FPGA tiles from an architectural specification," in *Proc. FPGA '03*, ACM Press, 2003, pp. 164–172.

[9] Saab Ericsson Space AB European Space Agency Contract Report, *Application-like Radiation Test of XTMR and FTMR Mitigation Techniques for Xilinx Virtex-II FPGA*. https://escies.org/public/radiation/esa/database/-ESA_QCA0415S_C.pdf, 2005.

[10] C. Yui, G. Swift, and C. Carmichael, "Single event upset susceptibility testing of the Xilinx Virtex II FPGA," in *Military and Aerospace Applications of Programmable Logic Conference (MAPLD)*, 2002.

[11] I. Page and W. Luk, *Compiling Occam into FPGAs*. Abingdon EE&CS Books, 1991, pp. 271–283.

[12] S. K. Mitra, *Digital Signal Processing A Computer-Based Approach International Editions 1998*. McGraw-Hill, 1998, pp. 339–416.

[13] J. Mathews and K. Fink, *Numerical Methods Using MATLAB*, 3rd ed. Prentice Hall, 1999, pp. 433–441.

[14] J. Hull, *Options, futures and other derivatives*, 5th ed. Prentice-Hall, 2002.

[15] G. Zhang, P. Leong, C. H. Ho, K. H. Tsoi, C. Cheung, D.-U. Lee, R. Cheung, and W. Luk, "Reconfigurable acceleration for Monte Carlo based financial simulation," in *Proc. ICFPT*, 2005, pp. 215–222.

[16] R. Usselmann, *Floating Point Unit*. http://www.opencores.org/project.cgi/web/fpu/overview, 2005.

[17] N. Y. ANSI/IEEE, *IEEE Standard for Binary Floating-Point Arithmetic*, The Insittution of Electrical and Electronic Engineering, Inc, Tech. Rep., 1985, IEEE Std 754-1985.

[18] J. Hauser, *TestFloat Release 2a General Documentation*. http://www.jhauser.us/arithmeic/testfloat.txt, 1998.

[19] S. Hsu, S. Mathew, M. Anders, B. Zeydel, V. Oklobdzija, R. Krishnamurthy, and S. Borkar, "A 110 GOPS/W 16-bit multiplier and reconfigurable PLA loop in 90-nm CMOS," *IEEE Journal of Solid State Circuits*, pp. 256–264, 2006.

[20] J. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits A Design Perspective*. Prentice-Hall, 2002.

[21] A. Bright et. al., "Blue Gene/L compute chip: synthesis, timing, and physical design," *IBM J. Res & Dev.*, vol. 49, no. 2/3, pp. 277–287, March/May 2005.

[22] C. Wait, "IBM PowerPC 440 FPU with complex-arithmetic extensions," *IBM J. Res & Dev.*, vol. 49, no. 2/3, pp. 249–254, March/May 2005.