Run-time Parameterised Circuits for the Xilinx XC6200

Rob Payne
rep@dcs.ed.ac.uk

Department of Computer Science, the University of Edinburgh

Abstract. Current design tools support parameterisation of circuits, but the parameters are fixed at compile-time. In contrast, the circuits discussed in this paper fix their parameters at run-time. Run-time parameterised circuits can potentially out-perform custom VLSI hardware by optimising the FPGA circuit for a specific instance of a problem rather than for a general class of problem. This paper discusses the design of run-time parameterised circuits, and presents a study of run-time parameterised circuits for finite field operations on the Xilinx XC6200. The paper includes a comparison with implementation on a self-timed version of the XC6200 architecture, which illustrates the potential benefits of self-timing for dynamically reconfigurable systems.

1 Introduction

Parameterisation is a common part of many FPGA design tools. For instance, many graphical design tools allow the definition of a bit-sliced component of arbitrary width, such as a N-bit wide adder. More comprehensive parameterisation can often be achieved through use of a Hardware Design Language, such as VHDL.

However, the parameterisation of these designs is fixed at compile time. Often it would be useful for an application to specify the parameterisation of a hardware accelerator at run-time rather than compile-time. For example, a constant multiplier is quicker and more compact than a general multiplier circuit. If a large number of data values are to be multiplied by a constant value, then it is beneficial to configure an instance of a parameterised circuit at run-time, rather than using a general-purpose multiplier circuit. Indeed, run-time parameterised circuits could potentially out-perform a dedicated VLSI implementation, since a dedicated VLSI solution is only optimised for solving a class of problems, whilst a run-time parameterised circuit is optimised to solve a particular instance of a problem. Similar ideas are being explored by other researchers in the context of partial evaluation in functional HDLs [1, 2]

The generation of run-time parameterised circuits poses several new challenges for designers. The key issue is that the generation of the configuration must be done quickly, otherwise the speed-up of using a run-time parameterised circuit is lost in the time taken to generate the FPGA configuration. As a result, neither timing-consuming place and route algorithms, or standard delay analysis algorithms can be used in generating the run-time parameterised circuits.

As standard place and route algorithms are too complex for run-time parameterisation, the configuration has to be constructed using a regular pattern of placement and routing. However, fixed (non-parameterised) components of a run-time parameterised circuit can be placed and routed using standard design tools at compile-time. The second problem of insufficient time for delay analysis at run-time requires that the delays in a class of parameterised circuit have to be analysed at design-time. Given the parameters of a circuit it should be possible to produce a relationship to the delay of the final circuit. Again, a regular structure to the circuit configuration assists in deriving this relationship.

An alternative solution to the problem of delay analysis would be to move away from using synchronous circuits, where the global clock period has to be determined. Circuits using self-timed (asynchronous) blocks to build the run-time parameterised circuit would not require delay analysis, as there is no global clock constraint to be met.

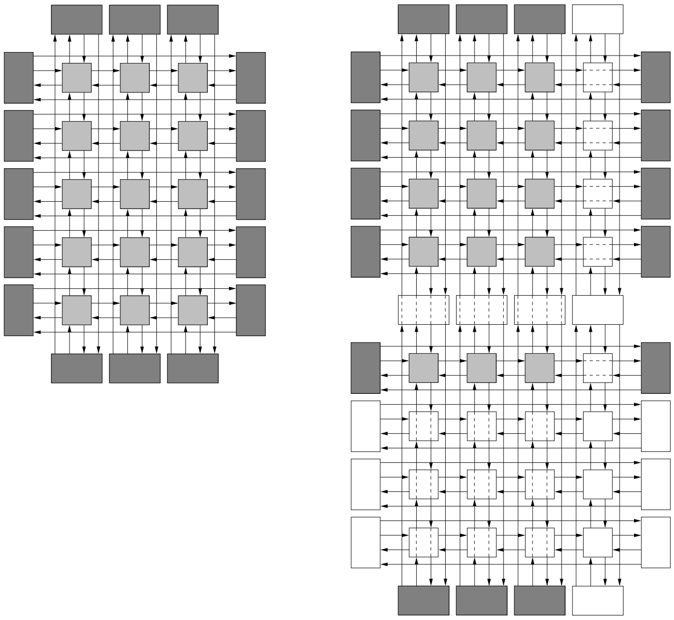
The rest of the paper illustrates techniques for run-time parameterisation using the example of circuits for finite field operations on the Xilinx XC6200 FPGA [3]. Such circuits have application in a number of areas such as Reed-Solomon error correcting. The circuits presented can be used to implement the dynamically reconfigurable communications system proposed by Klindworth [4], where the power of the error-correcting code is varied according to the current noise conditions.

The paper is structured as follows. Section 2 discusses techniques for regular placement and routing for the Xilinx XC6200, by identifying regular subsets of the XC6200 architecture. These techniques are illustrated in the next two sections; Section 3 discusses the design of a constant multiplier over $GF(2^k)$, which illustrates a simple level of parameterisation, and Section 4 describes a circuit for division by a fixed polynomial over $GF(2^k)$, which illustrates a circuit with more complex parameterisation. Section 5 discusses how the circuits can overcome the problems of delay analysis, and describes the benefits of using a self-timed FPGA architecture for the construction of run-time parameterised circuits. Finally, Section 6 summarises the paper and looks at possible future work.

2 Run-time Mapping Architecture Subset

As discussed in the introduction, a key issue in creating run-time parameterised circuits is that the circuits must have a regular structure to allow rapid parameterisation. Irregular routing resources, i.e. those only useable by certain cells cannot be used. As a consequence, before designing the run-time parameterised circuit, a regular subset of the logic and routing resources of the FPGA being mapped to must be identified. For many FPGA architectures, several such regular subsets can be defined depending on the basic building block of the architecture used for the construction of the run-time parameterised circuit. This section identifies regular subsets of the XC6200 for run-time routing and placement.

The Xilinx XC6200 is constructed of a hierarchy of cells that are grouped into 4×4, 16×16 and 64×64 blocks of cells that each have their own routing resources called *flyovers* (*fast lanes* in later versions of the datasheets [3]). Run-time parameterised circuits can be defined using any of these block sizes as the basic building block of the circuit. A problem with using any of these units as the basic building blocks, is that it does not allow the use of the higher level routing resources. For example, a run-time parameterised circuit defined in terms of the basic cells of the architecture, cannot use flyover or `Magic` routing resources, as these are not a regular routing resources with respect to single cells.



(a) Abstract 3×5 Block (b) Actual 4×4 Blocks

Fig. 1. Converting from an Abstract $N \times M$ Block to 4×4 Blocks

To allow the limited use of higher level routing resources, the concept of an abstract $N \times M$ block of cells is introduced. The abstract block just fits the run-time parameterised circuit that is to be instantiated, and except for its dimensions, it is similar to a 4×4 block of cells, consisting of a nearest neighbour array of cells with boundary multiplexors driving flyover signals that cross the block. Fig.1(a) illustrates an abstract 3×5 block. Such blocks can quickly be mapped to the 4×4 blocks in the architecture, by directly mapping cells to those in an array of 4×4 blocks large enough to contain the circuit, as illustrated in Fig.1(b). Unused cells and boundary multiplexors (unshaded in the figure) are configured so that signals pass through them unaffected and continue in the same direction. Unused cells can be rapidly configured using the

XC6200 wildcard configuration interface.

The technique of defining a circuit using an abstract block can be generalised to any of the basic building blocks. For example, a circuit defined in terms of 4×4 cell blocks can utilise level 16 flyovers by mapping an abstract $N \times M$ array of 4×4 blocks to the 16×16 blocks of the architecture.

3 $GF(2^k)$ Constant Multiplier

The section describes the design of a run-time parameterised circuit for multiplying by a constant in $GF(2^k)$. The circuit illustrates the construction of a simple run-time parameterised circuit using the technique of mapping from an abstract $N \times M$ block of cells. The circuit has two basic parameters, the size of the base field, k, and the constant in $GF(2^k)$ to be multiplied by. Operations in $GF(2^k)$ can be define as operations over polynomials in $GF(2)$ modulo an irreducible polynomial. Thus, multiplication of A by a constant B in $GF(2^k)$ can be represented by the multiplication of an input polynomial $A(x)$ by a constant polynomial $B(x)$ over $GF(2^k)$, as shown in Equation 1.

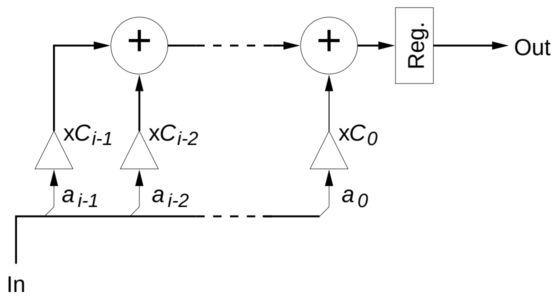
$$A(x).B(x) = \sum_{i=0}^{k-1} a_i x^i.B(x) \mod H(x) \tag{1}$$

In Equation 1, let $C_i(x)$ equal the term $x^i.B(x) \bmod H(x)$. The terms $C_i(x)$ are constant polynomials over $GF(2)$, since $B(x)$, x^i and $H(x)$ are constant. Thus, the multiplication can be expressed as a sum of the products $a_i.C_i(x)$. Creating products of a polynomial with a term in $GF(2)$ is trivial, since only the multiples one and zero are possible.

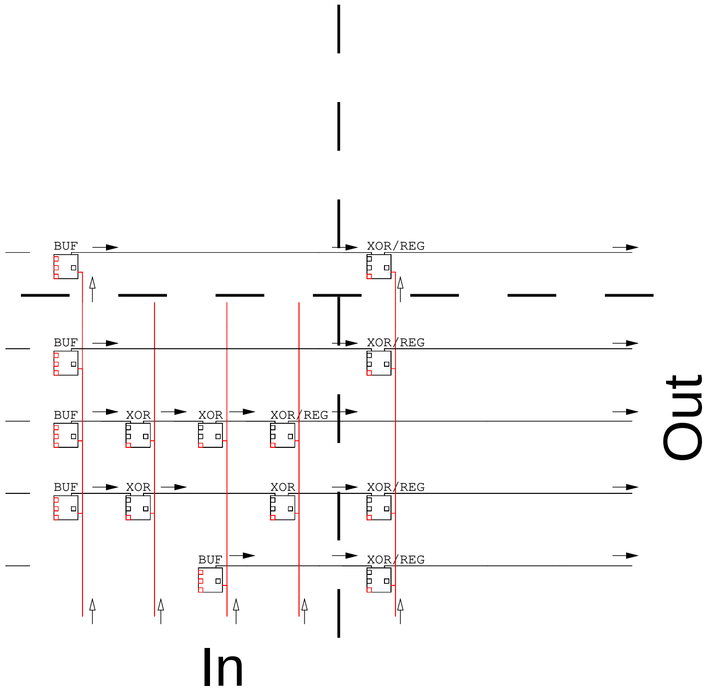
Fig.2(a) illustrates the pattern of data flow in the circuit. The input polynomial is split into its component terms a_i, which are multiplied by $C_i(x)$ to give $a_i.C_i(x)$. These terms are then summed to give the output. An example of the circuit layout as generated by the Xilinx tools is shown in Fig.2(b), for $GF(2^5)$. In the figure, the input terms are distributed to the cells using the level-4 flyovers. Additional information to assist in interpretation of the figure has been superimposed on the output of the Xilinx tools. The breakdown into 4×4 blocks is shown by the thick dotted lines. The direction of signal flow is indicated by small arrows next to the signals. Arrows with filled heads are used for local routing, whilst arrows with unfilled heads are used for level-4 flyovers.

The example configuration for $GF(2^5)$ was chosen to illustrate how circuits defined in terms of cells may not map well to 4×4 blocks. In Fig.2(b), the 4×4 block boundaries are marked by the thick dashed lines. Only the bottom left 4×4 block is completely used, the others are mostly full of unused cells. Little can be done to solve this problem without losing the regular structure of the circuit. Other $GF(2^k)$ fields give better mappings; if k is a multiple of 4, then the circuit maps exactly to 4×4 blocks.

The example also illustrates the potential savings from using run-time parameterised circuits. The constant $GF(2^k)$ multiplier is much smaller than a general $GF(2^k)$ multiplier, approximately by a factor k, and is also approximately twice as fast (See [5] for details).



(a) Data Flow



(b) Example Circuit for $GF(2^5)$

Fig. 2. Constant Multiplier in $GF(2^k)$

4 Fixed Polynomial Division

This section discusses the construction of run-time parameterised circuit for division of a polynomial over $GF(2^k)$ by a constant polynomial over $GF(2^k)$. The circuit can be used for the generation of Reed-Solomon error correcting codes. By varying the length of the generator polynomial and the choice of field $GF(2^k)$ representation, the error correcting power of the code can be varied.

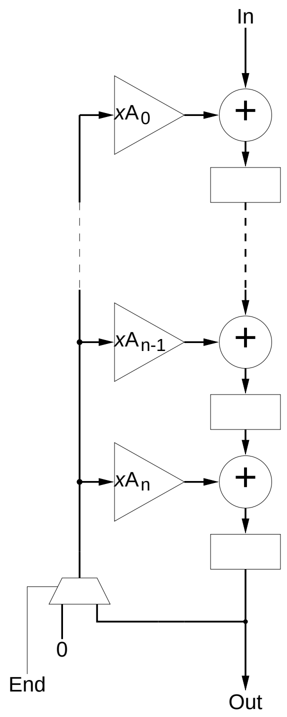


Fig. 3. Data Flow for Fixed Polynomial Division

The circuit illustrates the construction of a complex run-time parameterised circuit with a large number of parameters. The circuit's parameters are the size of the field $GF(2^k)$, the length of the input and fixed divisor polynomials, and the coefficients of the fixed divisor polynomial. In contrast to the last example, the circuit is built using 4×4 blocks of cells as the basic building blocks for the circuit, rather than individual cells.

Fig.3 illustrates the data flow for the fixed polynomial division circuit. It is similar to the linear feedback shift registers used for division by a fixed polynomial over $GF(2)$ in the generation of CRCs (Cyclic Redundancy Codes). The terms of the dividend are shifted down the shift register on the right. On each shift, a multiple of the divisor is subtracted from the value in the shift register. Adders are shown in Fig.3, since addition and subtraction are identical over $GF(2^k)$. In CRC circuits, generating the multiples is simple, since the only multiples possible in $GF(2)$ are zero and one, but in $GF(2^k)$, constant multiplier circuits must be used for each term in the divisor polynomial. When the whole dividend has been shifted in, the `End` flag is set, which zeros the feedback path and allows the remainder to be shifted out.

Fig.4 illustrates the configuration for a small divisor polynomial of length two with terms in $GF(2^4)$ as generated by the Xilinx Tools. The layout is similar to the layout of the data flow diagram in Fig.3. The 4×4 blocks which are used to construct the circuit are marked with the function that they perform. The `Adder` blocks integrate the addition and shifting stages in the data flow. The `Scale` blocks generate the multiples of the constant divisor using the constant $GF(2^k)$ multiplier circuits discussed in Section 3. The `Tjunc` block is used to route signals to the output and the feedback path. The `Fifo` block routes the output value to the edge of the circuit. The `Corner` block zeros the feedback when the remainder is shifted out (i.e. it generates the `End` signal in Fig.3), and is parameterised according to the length of the divisor polynomial. The `Counter` block generates the count used by the `Corner` block, to determine when to zero the feedback and is parameterised according to the length of the input polynomial.

5 Timing Analysis and Self-Timing

So far, the paper has described how the configuration of a run-time parameterised circuit can be generated rapidly using regular routing and placement. However, for synchronous circuits, once the configuration has been generated, some method is required to derive the delay of the circuit, to ensure that the circuit will meet the global clock constraint, or that the clock speed can be changed to match the delay of the circuit.

Since there is insufficient time to perform a delay analysis at run-time, a worst case delay bound for the circuit must be derived from the value of the parameters for the circuit. Thus during design, the class of run-time parameterised circuits, must be analysed to derive an equation for its worst case delay given its parameters. Circuits with a simple regular construction simplify the process of deriving this relationship. For example, it is clear from the structure of the $GF(2^k)$ constant multiplier of Fig.2(b) that the worst case delay of the class of circuit is the path from the bottom left corner to the top-right output via up to k cells configured as `XOR` and `BUF` gates.

Run-time parameterised circuits constructed from a hierarchy of parameterised parts, such as the fixed divisor circuit, could have their worst case delay calculated knowing the worst case delays of the components. If each parameterised component of the circuit is a pipeline stage, then the worst case delay is simply the largest delay of each stage. If each component is not a pipeline stage, then the components have to be grouped into pipeline stages first.

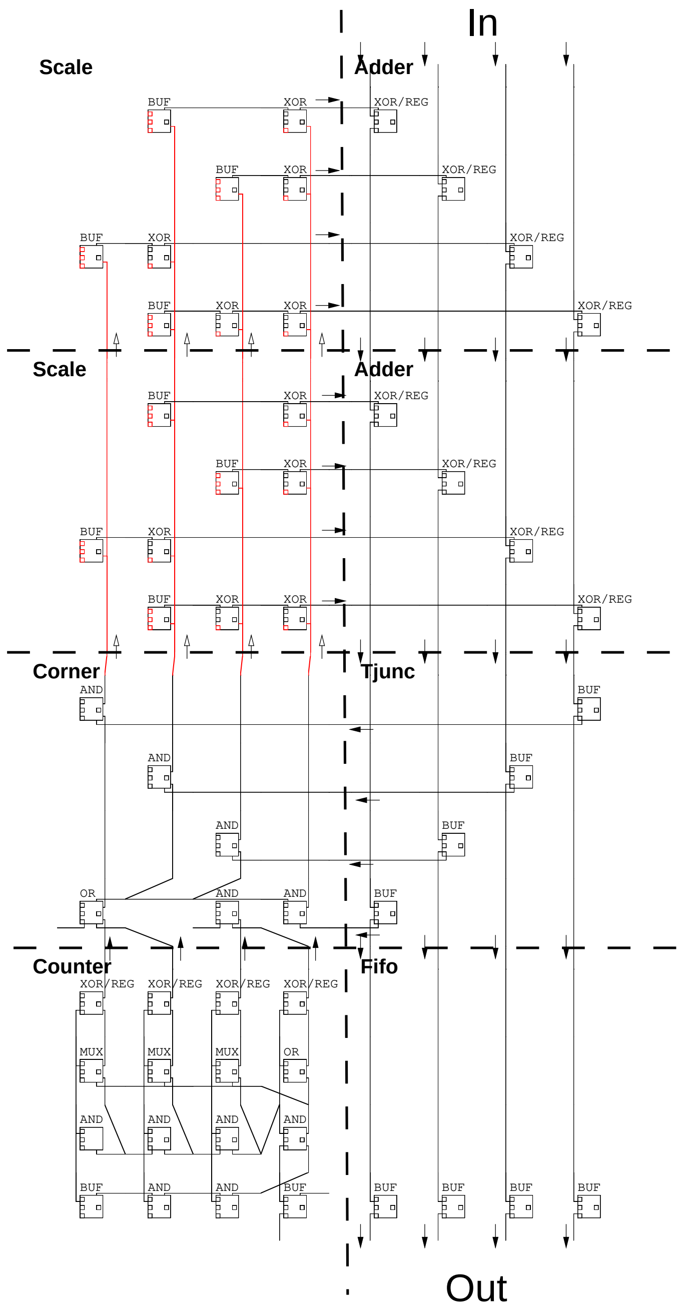


Fig. 4. Fixed Division by a Polynomial of Length Two over $GF(2^4)$

An alternative approach is to avoid delay analysis by using self-timed circuits. In this case, the circuit only has to meet the local timing constraints of the self-timed protocol used rather the global clocking constraints of the synchronous protocol. This means that circuits can be rapidly constructed from self-timed parts with no need for delay analysis of the whole circuit. Other benefits of the self-timed approach, is that their is that parts of the circuit may be separated from other parts, potentially on separate chips, and the self-timed protocols will accommodate the longer routing delays. Also, self-timed circuits can be designed to exploit data-dependent delays.

The circuits described here have been simulated on a self-timed version of the Xilinx XC6200 architecture [5] based on the STACC model[6] for self-timed FPGA architectures. Self-timed FPGA architectures in general are discussed in [7]. The STACC model involves replacing the global clock of an FPGA with an array of *timing cells* that generate local timing control. The model is general enough to apply to a wide range of FPGA architectures. An important feature of the STACC model is that it preserves the structure of an FPGA's logic array, so that designs can be readily transferred between synchronous and self-timed architectures.

The run-time parameterised circuits were simulated on two versions of the self-timed XC6200, which used different delay techniques. The first model used a reconfigurable fixed delay in each timing cell to provide the local delay. This delay technique, still required the local delay of each block to be derived from its parameters to set the reconfigurable delay element in each timing cell, but did not need any further analysis, for circuits composed from these parameterised circuits.

The second delay methodology used was Current Sensing Completion Detection (CSCD) [8]. CSCD generates a completion signal for each block by monitoring the current drawn by a CMOS circuit. The circuit has completed evaluating when this current becomes negligible. CSCD can exploit data-dependent delays for each block. For example, if the same data is presented in succession then the block will complete immediately. The CSCD run-time parameterised circuits needed no delay analysis, since the delay is generated by the architecture.

The performance of run-time parameterised circuits on different architecture are presented in Table 1. The figures quoted are the delay between results being produced by the circuits. For the synchronous versions this could involve multiple clock cycles. The self-timed figures are averages as the self-timed circuits' delays are data dependent.

The self-timed fixed delay architecture is slower on smaller examples due to additional pipelining overheads (see [5]) but gains on larger examples (lower in the table) due to being able to exploit some data-dependent delays between self-timed regions. The self-timed CSCD architecture consistently out-performs the synchronous architecture by exploiting data-dependent delays within the local self-timed regions. The figures quoted for the CSCD architecture are conservative as they are based on worst case delay figure for the cells from Xilinx. Actual CSCD implementations, can exploit the actual delays of individual cells.

Table 1. Performance Figures

Circuit	**Sync.**	**Self-Timed Fixed Delay**		**Self-Timed CSCD**	
	/ns	*/ns*	*diff%*	*/ns*	*diff%*
General $GF(2^k)$ Multiplier	50.5	60.2	+19.2%	45.9	-9.1%
Fixed Polynomial Division	57.5	73.3	+27.5%	54.4	-5.5%
Fixed Polynomial Evaluation	162.0	213.6	+31.9%	137.0	-15.4%
General Poly. Evaluation	314.0	387.6	+23.4%	267.8	-14.5%
$GF(2^k)$ Division	330.0	323.7	-1.9%	272.1	-18.5%
Polynomial Division	98.5	94.7	-3.9%	85.1	-13.6%

6 Conclusions

This paper has described the generation of run-time parameterised circuits for the Xilinx XC6200. Such circuits can potentially out-perform a custom VLSI solution as each circuit is optimised to solve a particular problem rather than a general class of problem. Run-time parameterisation requires techniques that allow the rapid generation of configurations at run-time. The examples illustrated that for rapid placement and routing, circuits must be constructed with a regular structure. Also, for synchronous circuits, a bound on the delays in the circuit must be determined, without sufficient time for a full delay analysis. Self-timing simplifies the problem, as only local timing constraints have to be met. Results also highlighted how the self-timed architectures, especially the CSCD version, can exploit data-dependent delays.

The future development of run-time parameterised circuits will require the development of specialised design tools. Such tools will have to analyse a class of circuits rather than a specific instance of a circuit. Unlike current design tools, the output of the tool will not be an FPGA configuration, instead, the design tools will produce a software program that can generate an instance of the parameterised circuit quickly at run-time.

Acknowledgements: Thanks to Gordon Brebner and Iain Lindsay for their advice and guidance.

References

1. S. Singh, J. Hogg, and D. McAuley. Expressing Dynamic Reconfiguration by Partial Evaluation. In *FCCM96: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1996.
2. W. Luk, N. Shirazi, and P. Y. K. Cheung. Compilation Tools for Run-Time Reconfigurable Designs. In *FCCM97: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1997.
3. Xilinx. *XC6200 Datasheet*, 1996.

4. A. Klindworth. FPLD-Implementation of Computations over Finite Fields $GF(2^m)$ with Applications to Error Control Coding. In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications*, volume 975 of *Lecture Notes in Computer Science*, pages 261–271. Springer-Verlag, 1995.
5. R. E. Payne. *Self-Timed Field Programmable Gate Array Architectures*. PhD thesis, University of Edinburgh, 1997.
6. R. E. Payne. Self-Timed FPGA Systems. In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications*, volume 975 of *Lecture Notes in Computer Science*, pages 21–35. Springer-Verlag, 1995.
7. R. E. Payne. Asynchronous FPGA Architectures. *IEE Proceedings on Computers and Digital Techniques - Special Issue on Asynchronous Processors*, 143(5):282–286, September 1996.
8. M. E. Dean, D. L. Dill, and M. Horowitz. Self-Timed Logic Using Current-Sensing Completion Detection (CSCD). In *Proc. International Conf. Computer Design (ICCD)*, pages 187–191. IEEE Computer Society Press, October 1991.

This article was processed using the LaTeX macro package with LLNCS style