Rapid Prototyping of Self-timed Circuits

S.W. Moore and P. Robinson
University of Cambridge, Computer Laboratory,
New Museums Site, Pembroke Street,
Cambridge CB2 3QG, U.K.
{Simon.Moore,Peter.Robinson}@cl.cam.ac.uk

Abstract

Self-timed circuits relieve the designer of problems like clock distribution, but introduce new constraints in the form of isochronic forks and equipotential regions. This paper shows how the combination of floor- and geometry-planning tools can be used to address these new problems. As a result, prototype self-timed circuits can be developed on conventional, clocked FPGAs without sacrificing performance. We also present a solution to the problem of designing arbiters on FPGAs.

1. Introduction

FPGAs tailored for prototyping self-timed circuits have been proposed but will not be readily available until self-timed techniques become widespread. This paper explains how self-timed circuits may be mapped onto commercial (clocked) FPGAs.

The table based function generators provided on FPGAs allow glitch free combinatorial functions to be implemented. Many popular self-timed circuit cells [12, 2] can be easily mapped onto these functions generators [3, 7, 8, 9].

The programmable interconnect exacerbates wire delays. Consequently, more care has to be taken when making delay assumptions. Section 2 explains the constrains imposed by wire delay properties. Sections 3 & 4 explain how we met these constraints and Section 5 presents a case study.

Arbiters (mutual exclusion elements) are not provided on clocked FPGAs though they are essential for many self-timed systems. Section 6 presents an FPGA arbiter design with testing and analysis in Sections 7 & 8.

2. Self-timed Properties

Two important circuit properties that are essential when designing self-timed circuits: *equipotential regions* and *isochronic forks*. Equipotential regions are small circuits where wire delays are sufficiently short that one may treat a signal as identical at all points on a wire [11]. Basic circuit elements like RS flip-flops rely on equipotential regions since the feedback paths must be "instantaneous" if the circuit is to function correctly.

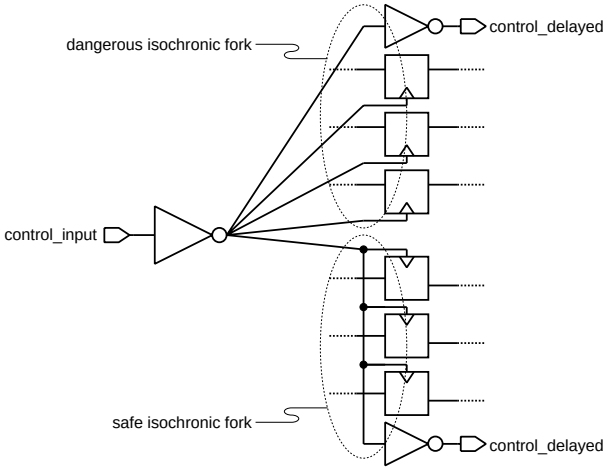

Figure 1. Dangerous and safe isochronic forks

An isochronic fork is a forking wire with constraints on the arrival times at the ends of the wires radiating from the fork [1]. Isochronic properties are not just governed by wire delays; the loads being driven have a marked effect on performance [1]. This makes general isochronic forks difficult to use, particularly if the wires are long. However, distribution of a signal to many gates can be achieved safely by using a single wire which is routed to each gate in turn, thereby avoiding long forks (see Figure 1).

Typically commercial place and route tools do not understand isochronic forks, except for clock distribution which is usually treated separately. The next two sections of this

paper present floor planning and geometry planning techniques which may be used to enforce equipotential and isochronic fork properties.

3. Floor Planning

Modern floor planning tools allow logical hierarchy (pre-layout) to be imposed on the physical hierarchy (post-layout). For example, to ensure that the carry propagation path in an adder design is short, the floor plan could constrain the adder, and all of its associated subcomponents, so that it resides in one small area.

Detailed floor planning may be used to enforce equipotential regions. Such circuits must be small to satisfy the equipotential properties, and tend to be numerous. Consequently it would be convenient if such regions could be specified as part of a hardware description language (HDL). However, commercial floor planning tools are usually separate from the HDL and synthesis process.

Floor planning is less appropriate for enforcing isochronic fork properties. In practice short isochronic forks can be thought of as equipotential regions. However, long isochronic forks need to be enforced via routing constraints rather than area/floor planning constraints.

4. Geometry Planning

Some place and route tools allow components in a block to be placed relative to each other. For example, the place and route tool supplied by Xilinx™ for their 4000 series FPGAs allows latches to be grouped into sets [14]. Latches within a set may be placed relative to each other. For example, the arbiter design presented in Section 6 uses four D-type latches which need to be placed next to one another to ensure minimum wiring delays (see Figure 3c).

Geometry requirements could be specified separately from the original HDL and applied after synthesis. However, these properties are a fundamental part of the functional correctness of the design so it seems more logical to add the information to the HDL description. Whilst such information could be added via a new HDL or by extending an existing HDL, this would require changes to all the other tools using the HDL. An alternative is to add information to instance names, vis:

```
reg aa_RLOC_R0C1;
```

(see Figure 4 for a more complete example). This specifies a register (in Verilog) which has additional relative location information supplied by the extension "`RLOC_R0C1`". The "`RLOC_`" indicates that what follows is geometry information. "`R0C1`" indicates that this register should be in row 0, column 1 on some virtual grid in relation to a collection of registers. In the case studies presented later, this geometry grid gives the relative positions within a set of programmable logic blocks.

Typically synthesis tools preserve instance names of registers, albeit with prefixed hierarchical information. Thus, geometry information for each hierarchical block may be derived by post processing the synthesised netlist. This proves to be quite convenient for the designer and does not require modifications to (commercial) synthesis tools.

Once placement has been constrained, routing is more predictable. For signals requiring a short path, the routing priority of the interconnect may be raised. This could be achieved by annotating signal instances with, for example, "`CRITICAL`". However, synthesis tools often rename signals and so the annotation would be lost. An alternative is to use a hard macro function (e.g. `CRITICAL()`) which explicitly instructs the synthesis tool to add the priority information to the synthesised netlist.

Asymmetric isochronic fork properties can be assured by careful geometry planning and marking of critical wires. Also, many FPGAs offer low latency long wires which can be useful for distributing asymmetric isochronic forks with many short branches.

5. Case Study — Loadable Counter

To demonstrate geometry- and floor-planning, this section presents a loadable counter design which will only function correctly if routing constraints are met. Philips suggested the loadable counter problem as an example for the 1996 ACiD Workshop in Groningen [13] which they required to implement the `for` n `do` statement in the Tangram language [2]. The requirement is for a counter which can receive a bundled data integer, n, and will then perform n four-phase handshakes on its output side. One solution is to use a systolic counter [4, 13] which is quasi-delay insensitive (QDI — see Glossary) apart from parallel loading the bundled data value n. However, the solution is somewhat large.

We have designed a loadable counter which, whilst not QDI, is more compact (see Figure 2). Although it is not truly QDI, it does use a form of completion detection to "time" the main activity. However, a few conservative delay assumptions are made in the control path which means that the design is not QDI. This we feel is acceptable since the resulting design is particularly efficient on current FPGAs because they support D-latches which are prevalent in this design.

This version produces $n+1$ output handshakes. A loadable ripple counter is used to hold the complement of n. To load a value, data is presented on the `data`x wires, and when they are stable and `loadack` is low, `load` may be

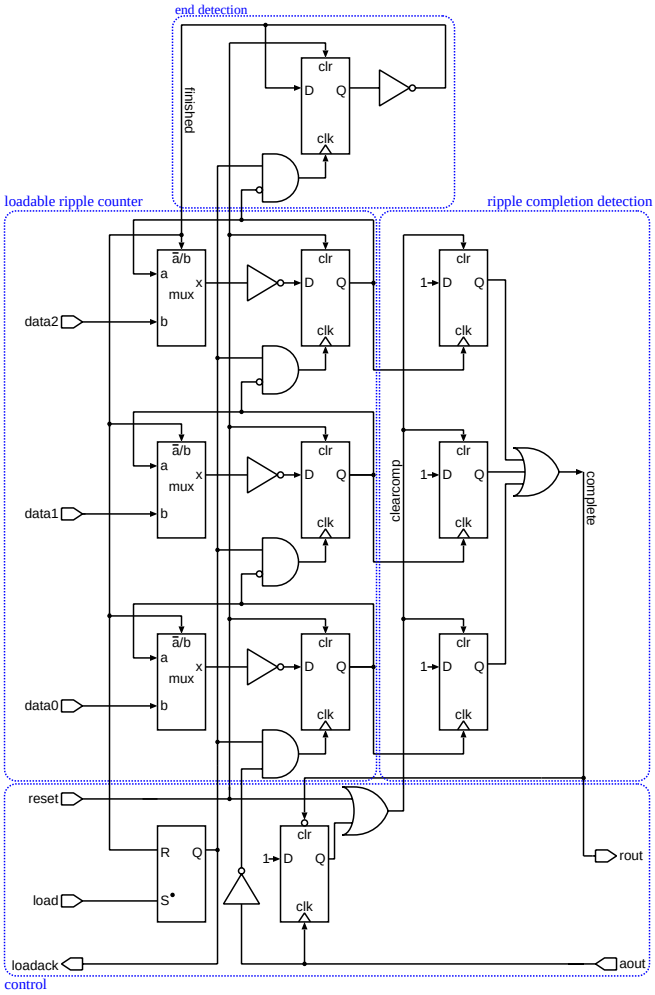

Figure 2. Loadable counter

raised. The `load` signal sets an RS flip-flop (the S input has priority over R) which in turn cause the data to be latched once the acknowledge on the output side is low (which should always be the case by the time `loadack`=`0`).

The value in the ripple counter is incremented when the acknowledgement on the output side falls (`aout`−). `aout`− causes the first flip-flop to toggle, and if it changes from a 1 to a 0 then it will cause the next flip-flop to toggle, and so on. A ripple has completed when one of the flip-flops toggles from a 0 to a 1, since this will not propagate the ripple any further. Completion can, therefore, be detected using edge triggered D-latches. Completion causes the output request to raise (`rout`+). The environment may then respond with an acknowledgement (`aout`+) which resets the completion detection circuit, thus causing `rout`−. Once all of the flip-flops in the ripple counter reach zero, completion will not be detected because all of the flip-flops will make a negative transition. However the roll over from all 1's to 0's triggers the end detection flip-flop which raises `finished` to clear `loadack`. A new value may then be loaded.

This design requires geometry planning to ensure that internal delays are minimised. In particular the propagation of the `load` signal must clear the `finished` signal before the environment has time to react to the `loadack` signal and lower `load`. This timing requirement could be removed by adding a matched delay to the `loadack` signal. There is a similar critical path when clearing the ripple completion detection since the clear signal must be removed before the flip-flops will detect another edge on their clock inputs. This timing requirement can be reduced to the output environment not reacting too quickly (around 2 gate delays) and lowering `aout` when `rout`− occurs. A matched delay could similarly be added to `rout` to guarantee this timing constraint is met, but the output environment may well be slow enough to make this unnecessary.

6 FPGA Arbiter Design

Arbiters (mutual exclusion elements) are also rarely present in standard cell libraries though they are essential for many self-timed systems. Typically an arbiter is constructed from an RS flip-flop (or $\overline{\mathrm{R}}\,\overline{\mathrm{S}}$ flip-flops — Figure 3) to resolve which request (`ra` or `rb`) arrived first. If both requests arrive simultaneously then the flip-flop will go metastable but will eventually resolve to respond to one request. To filter out the metastable state, Seitz [10] proposed that two inverters powered from the RS flip-flop be used (see Figure 3a for a CMOS variant). This frequently used solution can be constructed easily from transistors but not from standard cells. The Philips team [6] proposed the use of four input NOR gates to act as filters since they have a higher threshold voltage. However, this technique increases the load on the outputs of the RS flip-flop which will make metastability resolution slower. Furthermore, although the solution is suitable for ASICs, it is inappropriate for FPGAs because threshold levels cannot be controlled in this way.

Designing an arbiter on an FPGA is even more difficult. In order to minimise metastability effects the built in flip-flops must be used — typically D-latches — since they will have the highest gain and lowest feedback times. Conversely, flip-flops constructed from combinatorial blocks have longer feedback paths and often have lower gain so metastability resolution is poor. Another problem is filtering out the metastable case. One solution is to use a clocked circuit to sample the request signals using D-latches, perform a simple function to determine which acknowledge should be raised, and then latch the output function (see Figure 3c). Whilst this does introduce a clock into the circuit it is a simple solution which will fail only if the metastable state is not resolved between clock edges. In practice (see Section 7) this design is reliable provided the slew rate of the request signals is reasonably fast (less than one gate delay). However, the latency through this arbiter is long which is less desirable.

A local clock may be provided for the FPGA arbiter (see Figure 3d). The local clock is enabled when any of the inputs, outputs or internal nodes is active (high). This frees the arbiter from a global clock at the expense of substantially more circuitry (an extra 3 CLBs).

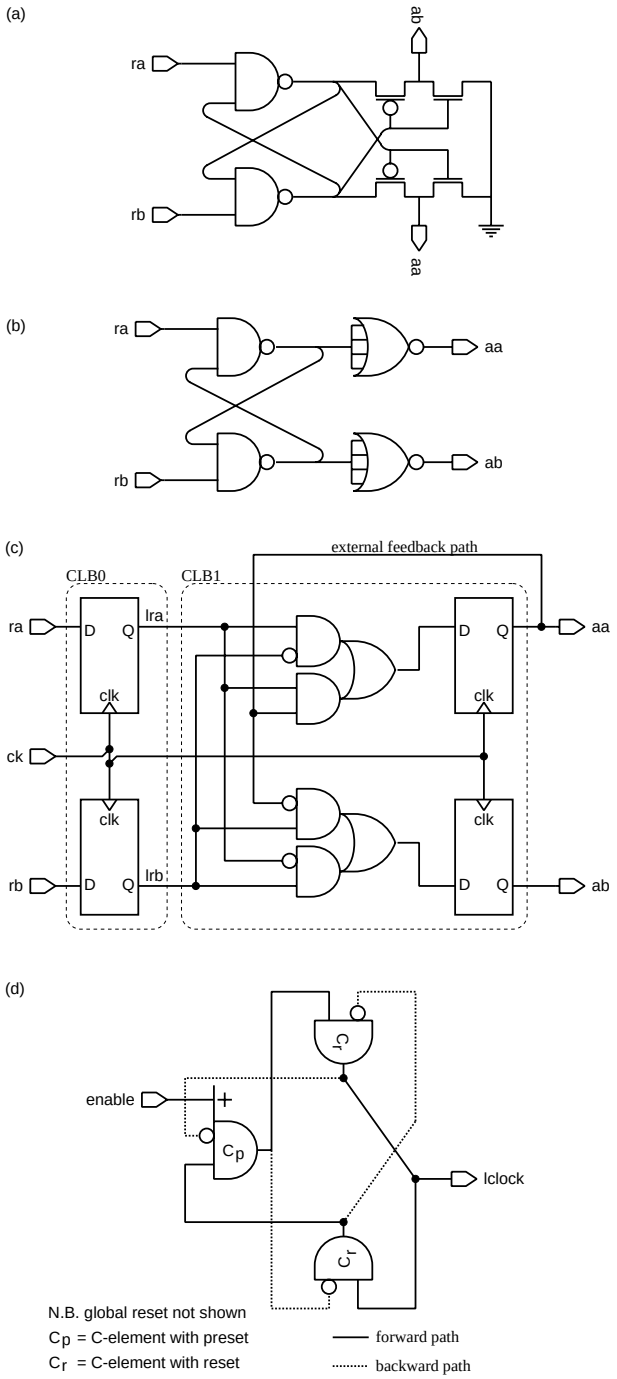

Figure 3. Arbiter designs: (a) Seitz's transistor level design, (b) van Berkel's gate level CMOS design and (c) an FPGA design (d) local clock for FPGA arbiter

7 Arbiter Tester

Testing arbiters to ensure that they resolve metastable conditions cleanly is difficult because the outputs cannot be easily viewed without passing the signal through an output pad which is likely to modify any metastable response. Instead we decided to see whether the erroneous signals from arbiters resulted in side effects in the receiving control circuits.

We began by designing a four-phase *call* module (sometimes called a *mixer*) which acts as a hardware subroutine

```
module XI_ARBITER(aa_RLOC_R0C1,ab_RLOC_R0C1,ra,rb,reset,ck);

  input  reset;                    // reset
  input  ra,rb;                    // request signals
  input  ck;                       // sampling clock
  output aa_RLOC_R0C1,ab_RLOC_R0C1; // acknowledge signals

  reg lra_RLOC_R0C0,lrb_RLOC_R0C0;
  reg aa_RLOC_R0C1,ab_RLOC_R0C1;
  initial begin
    lra_RLOC_R0C0=0;
    lrb_RLOC_R0C0=0;
    aa_RLOC_R0C1=0;
    ab_RLOC_R0C1=0;
  end

  always @(posedge ck)
    begin
      lra_RLOC_R0C0 <= ra;
      lrb_RLOC_R0C0 <= rb;
      aa_RLOC_R0C1  <= (lra_RLOC_R0C0 & ~lrb_RLOC_R0C0) |
                       (lra_RLOC_R0C0 &  aa_RLOC_R0C1);
      ab_RLOC_R0C1  <= (lrb_RLOC_R0C0 & ~lra_RLOC_R0C0) |
                       (lrb_RLOC_R0C0 & ~aa_RLOC_R0C1);
    end

  // ensure feedback via a short path
  CRITICAL(aa_RLOC_R0C1);
endmodule
```

Figure 4. Verilog code for a Xilinx™ arbiter

call allowing multiple accesses to a shared resource (see Figure 5). An arbiter is required because the `call` module can cope with simultaneous requests. A request signal is sent from a client to the subroutine, and after the subroutine acknowledges, the acknowledge is routed back to the appropriate client.

Two call modules were then coupled to an event ring so that events could be inserted or extracted (see Figure 6). If an arbitration fails then either an event will fail to be inserted or extracted, or an extra event will be inserted or extracted. Thus, sequences of inserts followed by extracts can be performed to test the circuit. Because the circuit is self-timed, the tester does not need to be fast. In fact it is advantageous to allow periods between sequences of inserts and extracts to allow events to spin freely around the ring, thereby ensuring that the self-timed circuit does not become phase locked with the tester which might otherwise avoid chances for metastability. Changes in environment temperature will also add randomness to the timing properties which helps to stress the circuit.

This test procedure was performed on a number of different arbiter designs on Xilinx™ 4000 series FPGAs. Designs based on RS flip-flops failed due to poor metastability characteristics. The only design which failed to produce an error after 4 billion insert and extract sequences was the D latch based design presented in Figure 3c. However, this circuit does fail if the slew rate of the request signals was long. In particular, we were using the parallel port on a laptop computer to provide test sequences. This had a long rise time which, despite buffering on the Xilinx™ chip, caused

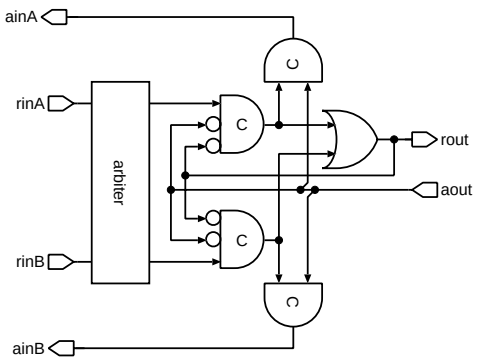

Figure 5. 4-phase call element with arbitration

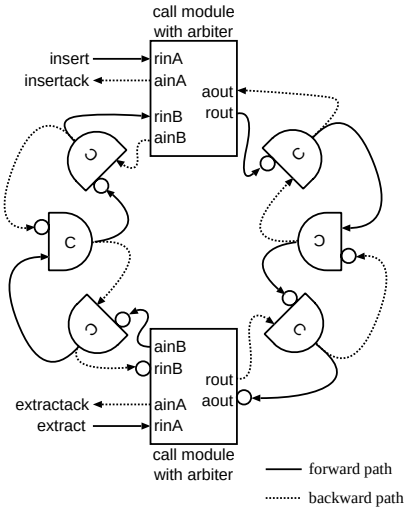

Figure 6. Arbiter test circuit

errors. Adding a TTL buffer was sufficient to clean up the signal.

8 Analysis of FPGA Arbiter Reliability

A statistical failure rate of the circuit in Figure 3c can be determined on the basis of the characteristics given by Xilinx [14]. A generally accepted equation for the mean time between failures (MTBF) for a latch is [5]:

$$MTBF = \frac{e^{k_2 . t}}{f_1 . f_2 . k_1}$$

where:

k_1 the metastability-catching setup time window of the latch

k_2 the gain-bandwidth product in the feedback path of the master latch of the latch

f_1 is the frequency of the asynchronous data input

f_2 is the latch clock frequency

t is the time allowed for metastability to be resolved

k_2 can be measured experimentally and for Xilinx™ XC4005E-3 parts is quoted as $19.4ns^{-1}$. k_1 is also around $0.1ns$ for D-latches in this device and f_2 is often around $100MHz$. In normal asynchronous operation f_1, the input frequency to the arbiter, would be no more than $f_2/4$ — the time for a cycle $ra+ \rightarrow aa+ \rightarrow ra- \rightarrow aa-$.

For the circuit being analysed, t is the time between clock edges ($10ns$ at $100MHz$) minus the time to pass data from the outputs of the first D latch pair, over a local routing network through a function block (F or G) to the next pair of D-latches. Based on the manufacturer's worst case delay characteristics, one can determine that t is at least $3ns$.

This yields a worst case MTBF of around 240 years. This MTBF may be substantially improved upon (if required) by adding an extra pair of D-latches to the inputs so that the request signals are sampled twice, rather than once, before being used. This will, of course, add an extra 50% to the latency.

9. Conclusions

Geometry- and floor- planning tools have been been proposed to ensure that critical self-timed characteristics (equipotential regions and isochronic forks) are met. A case study demonstrate how these techniques can be applied to enable rapid production of designs on FPGAs without the need for laborious hand placement and routing.

Whilst floor-planning is often supported by CAD software, geometry planning is an innovation we have been working on. Both techniques are only possible with support from place and route tools. The effect of long programmable interconnect delays is similar to the significant wire delays exhibited by deep submicron CMOS technology. In order to manage these delays, CAD software will have to allow the designer to specify delay bounds on wires. Whilst the intention will no doubt be to assist designers of clocked circuits, we hope that it will also provide additional constraint mechanisms to allow automated place and route of self-timed designs for ASICs.

The other significant problem we have tackled is the design of a reliable arbiter for commercial (clocked) FPGAs. Experimental results give us confidence in this design and an analysis of the metastable characteristics adds further reassurance. This work, together with earlier work on self-timed cell sets, provides a practical toolkit for rapidly prototyping self-timed circuits on commercial (clocked) FPGAs.

Acknowledgements

This work was supported by the EPSRC under grant GR/J62708 with further support from Olivetti/Oracle Research Laboratory (ORL) in Cambridge. We would also like to thank Dr David Greaves for use and assistance with his Verilog compiler and simulator. Dr Myra Van Inwegen and Mr Daniel Gordon gave useful advice on Verilog techniques, and to Prof Ivan Sutherland and Mr Steve Wilcox contributed to many useful discussions.

References

[1] K. v. Berkel. Beware the isochronic fork. *Integration, the VLSI journal*, 13(2):103–128, June 1992.

[2] K. v. Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[3] E. Brunvand. Implementing self-timed systems with FPGAs. In W. R. Moore and W. Luk, editors, *FPGAs*, pages 312–323. Abingdon EE & CS Books, 1991.

[4] G. Goossens and M. B. Josephs. Proceedings of ACiD-WG/EXACT workshop on asynchronous controllers and interfaces. Technical Report EXACT/D.2/IMEC/m3/D1, IMEC, Leuven, Belgium, Sept. 1992.

[5] H. W. Johnson and M. Graham. *High-Speed Digital Design — A Handbook of Black Magin*. Prentice Hall, 1993.

[6] J. Kessels and P. Marston. Designing asynchronous standby circuits for a low-power pager. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 268–278. IEEE Computer Society Press, Apr. 1997.

[7] K. Maheswaran and V. Akella. Hazard-free implementation of the self-timed cell set for the Xilinx 4000 series FPGA. Technical report, U.C.Davis, 1994.

[8] K. Maheswaran and J. Lipsher. A cell set for self-timed design using Xilinx XC4000 series FPGA. Technical report, U.C.Davis, 1994.

[9] J. Oldfield and C. Kappler. Implementing self-timed systems: comparison of configurable arrays with full custom circuits. In *FPGAs: International Workshop on Programmable Logic and Applications*. Abingdon EE&CS Books, 1991.

[10] C. L. Seitz. Self-timed VLSI systems. In C. L. Seitz, editor, *Proceedings of the 1st Caltech Conference on Very Large Scale Integration*, pages 345–355, Pasadena, CA, Jan. 1979. Caltech C.S. Dept.

[11] C. L. Seitz. System timing. In C. A. Mead and L. A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[12] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[13] J. T. Udding. ACiD-WG workshop. Technical Report CSN 9602, University of Groningen, Dept. of Comp. Science, 1996.

[14] Xilinx. The programmable logic. Data Book published by Xilinx, 1998.