Performance-Driven Syntax-Directed Synthesis of Asynchronous Processors

Luis A. Plana Doug Edwards Sam Taylor
Advanced Processor Technologies Group
School of Computer Science
The University of Manchester
{lplana, dedwards, smtaylor}@cs.manchester.ac.uk

Abstract

The development of robust and efficient synthesis tools is important if asynchronous design is to gain more widespread acceptance. Syntax-directed translation is a powerful synthesis paradigm that compiles transparently a system specification written in a high-level language into a network of pre-designed handshaking modules. The transparency is provided by a one-to-one mapping from language constructs to the module networks that implement them. This gives the designer flexibility, at the language level, to optimise the resulting circuit in terms of performance, area or power.

This paper introduces new techniques that exploit this flexibility to improve the performance of synthesised asynchronous systems. The results of a series of transistor level simulations show that these techniques, combined with optimised handshake module implementations, can produce up to a ten-fold improvement in the performance of a 32-bit, ARM-compatible, asynchronous processor used in an experimental smartcard SoC, without introducing any changes to the original processor architecture.

1 Introduction

Most modern embedded systems are synthesised using CAD tools. Although a large proportion of these systems are synchronous, interest in asynchronous circuits and tools is continually growing for their low EMI, robust on-chip interconnect and their potential to deal effectively with process variation.

Several asynchronous synthesis systems have been recently introduced. Some target the synthesis of asynchronous controllers, *e.g.*, Petrify [1] and Minimalist [2]. Others target both control and datapath but may require user intervention or guidance during the synthesis process, *e.g.*, TAST [3] and the CSP-like CHP system [4]. Tangram [5, 6] and Balsa [7] are fully-automated systems that have successfully synthesised large-scale circuits using syntax-directed compilation. This paper focuses on this synthesis approach and examines the opportunities to optimise the performance of the generated circuits.

2 Syntax-Directed Synthesis

Syntax-Directed translation is a powerful synthesis technique. The first stage of the synthesis process involves compiling descriptions written in a high-level language into a network of handshaking modules. This approach gives a 'transparent' compilation, i.e., there is a one-to-one mapping from a language construct to the network of modules that implements it. This direct mapping gives the designer flexibility at the language level to impact the resulting circuit: incremental changes at the language level result in predictable changes in the implementation. Clearly, the source code specification may have a large impact on the performance, power consumption and area of the resulting circuit.

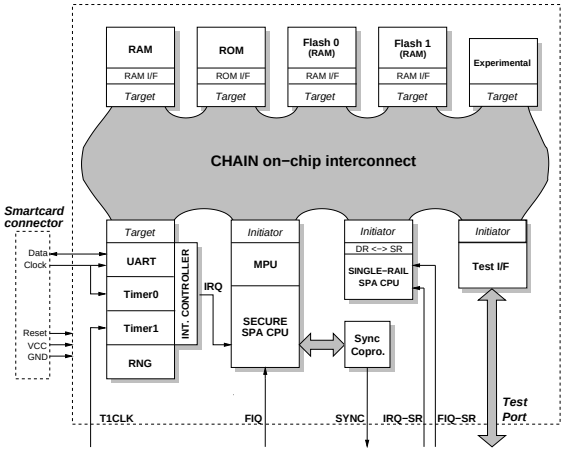

Figure 1. Smartcard System Architecture.

In many cases, the synthesis system can evaluate the gen-

erated module network to provide the user with an early estimate of the performance and area of the resulting circuit. In such cases, an experienced designer can optimise the resulting circuit in terms of performance, area or power by choosing the right specification.

Syntax-directed translation has been used successfully in the synthesis of an 80C51 microcontroller [8], the G3Card smartcard System-on-Chip [9], and an asynchronous MIPS microprocessor [10]. The G3Card SoC, shown in Figure 1, is a good example of an asynchronous embedded system. A prototype was fabricated in a 0.18μm process and was fully functional on first silicon. It contains two different, full-featured implementations of an ARM-compatible, asynchronous processor, a Memory Protection Unit, an asynchronous interface to standard synchronous RAM, a synchronisation coprocessor, and several peripherals, all synthesised using the Balsa synthesis system.

2.1 The Balsa Synthesis System

Balsa is a synthesis system that generates purely asynchronous macromodular circuits, called Handshake Circuits. Proposed originally for use with the Tangram language (upon which Balsa is heavily based), handshake circuits offer an attractive paradigm for circuit synthesis. Complex descriptions written in the source language are translated into a circuit consisting of instances of handshake components composed in a macromodular style.

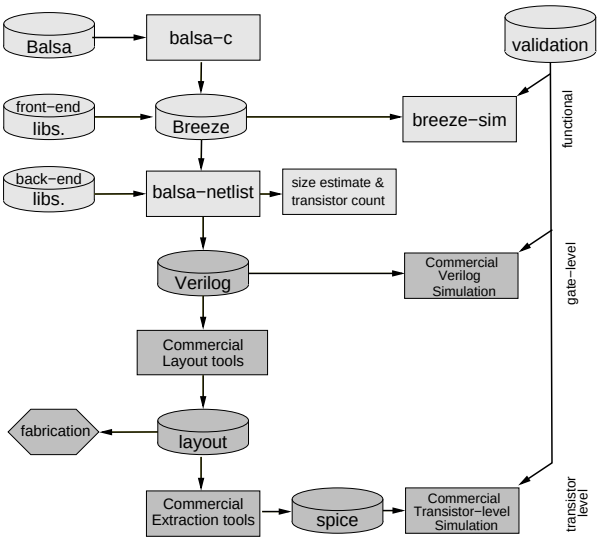

Figure 2. Balsa design flow.

Handshake components are selected from a relatively small set and are straightforward to implement. They are interconnected through channels. Each channel connects an *active* port on one component to a *passive* port on another. The sense of the port (active or passive) indicates the direction of the handshake. An active port initiates a handshake (sends the request) and the passive port acknowledges requests. Channels can carry data and this can flow in either the same direction as the handshake (a *push* channel) or in the opposite direction (a *pull* channel). Channels that carry no data are known as *sync* channels or frequently as *activation* channels as they are used to start the operation of many components when connected to an activation port.

The generation of the Handshake Circuit is the first step in the Balsa synthesis flow, shown in Figure 2. The Balsa compiler generates an intermediate netlist, in *Breeze* format, which can be used for functional validation and early performance estimates. The Balsa netlister generates a structural verilog netlist based on the target library cell and the selected asynchronous style and data encoding. The verilog netlist can be processed with commercial layout and extraction tools for further validation and fabrication.

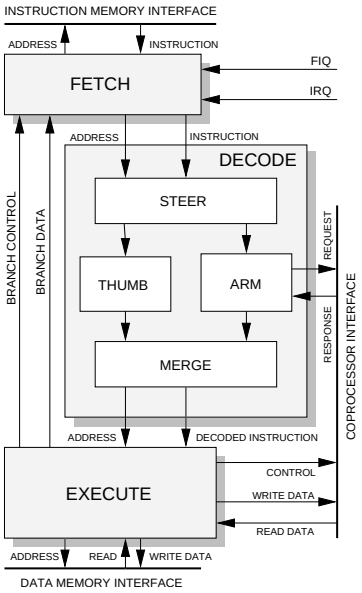

Figure 3. SPA Processor.

The G3Card SoC synthesised with Balsa was based on the SPA processor, a fully synthesised, 32 bit, 100% ARM compatible processor core. SPA was implemented as a simple, ARM7 style, 3-stage pipeline, depicted in Figure 3. Both *dual-rail* (1-of-2 data encoding, quasi-delay insensitive (Q^nDI) timing assumptions) and *bundled data* (single-rail data encoding, data-bundling timing assumptions) were implemented from the same Balsa specification.

The main goal of the dual-rail processor implementation was to defeat power analysis, therefore, a simple, power-balanced circuit was targeted. Performance was not a significant requirement for the smartcard, however, the syn-

thesised SPA was significantly slower than expected. The following section explores the reasons for the reduced performance and introduces new handshake component designs and performance-oriented techniques that result in improved performance without changes to the original architecture of the processor.

3 Performance-Driven Synthesis

The Handshake Components and techniques introduced in this section target optimised performance of Balsa-synthesised circuits. Reduced power consumption is a secondary consideration in cases where the choices have no significant impact on performance. Area is not considered a significant factor, although the Results Section shows that significant reductions in area are also achieved.

3.1 Improved Handshake Components

A drawback of syntax-directed synthesis is the overhead imposed on the circuits by the control-driven approach to the translation. A Handshake Circuit can be considered as a large monolithic tree of control components that direct the movement of data through datapath components. Data and control are frequently synchronised and, often, control is slower than data thus reducing the performance of the circuit as data is stalled while control catches up.

Previous attempts at improving syntax-directed synthesised systems have focused in the optimisation of the control tree by resynthesis and peephole transformations [11, 12]. The goal of the techniques is to identify sections of the control structure that can be clustered together and resynthesised. The behaviour of the control tree is not changed, it is implemented in a more efficient way. Although these techniques tend to optimise circuit area, up to 21% improvement in performance have been reported.

The basic control commands supported by Balsa are operations such as reading and writing from channels, arithmetic and logic operations, and channel synchronisation. Complex commands are formed by composing commands in parallel or sequentially. These are implemented using the Concur and Sequence components. Due to the asynchronous operation of the circuits, a FalseVariable component is used in every data channel input to detect the arrival of new data and trigger the control unit. These handshake components have a large impact on control and have been redesigned.

The design of the new Handshake Components is not based on the resynthesis approach used previously. The new components change the behaviour of the control structure to improve its performance while maintaining the same overall operation of the circuit.

The key factors identified as the main contributors to inefficient control components are: (i) the *return-to-zero* phase present in the four-phase handshake protocol, (ii) unnecessary synchronisation between data and control in channel inputs, and (iii) power- and time-balanced dual-rail adder and other datapath components.

The impact of return-to-zero phases is reduced by the overlapping of two or more phases so they operate concurrently rather than sequentially. In addition, some return-to-zero phases can be overlapped with processing phases where this does not cause a hazard.

The new input control components trigger the control units without waiting for the actual arrival of data as, in asynchronous systems, valid data identifies itself. This eliminates unnecessary synchronisation and gives control a head start.

The dual-rail Balsa back-end was optimised to synthesise secure systems, i.e., systems that can resist power and timing analyses. The adder and other datapath components were designed for data-independent speed and power consumption. This can only be achieved by making every case the same as the worst case. A performance-driven approach must abandon the security goal and optimise the performance of the datapath components. In particular, the dual-rail adder has been redesigned for average-case processing time and quick, fixed-delay return-to-zero time.

Details of the Handshake Components redesign can be found elsewhere [13].

3.2 Efficient Pipeline Control

Almost all modern embedded processors are pipelined, therefore, asynchronous synthesis tools must generate efficient pipeline control logic. Most asynchronous pipelines follow the micropipeline [14] model, a simple and elegant way to implement elastic asynchronous pipelines.

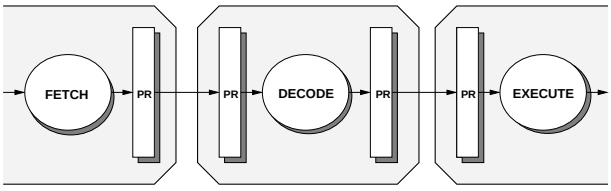

Figure 4. SPA half-occupancy pipeline.

Balsa has no special language constructs or handshake components to specify or implement pipelines. Pipeline stages are usually specified in Balsa procedures and pipeline registers are implemented using conventional variables. Balsa does not allow concurrent reads and writes to the same variable which means that, when a variable is used

as a pipeline register, the stages at either side of the variable cannot normally process data concurrently.

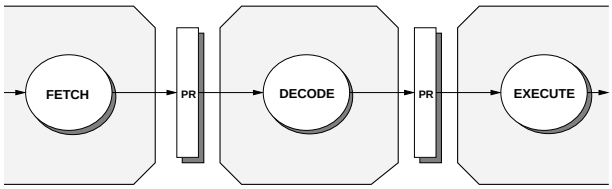

Figure 5. New full-occupancy pipeline.

Figure 4 shows how the SPA pipeline was implemented. As indicated earlier, the pipeline registers are variables inside each stage and both input and output registers are used. This structure essentially implements a half-occupancy pipeline: a pipeline with twice as many stages, with alternating *processing* and *empty* stages. Without the empty stages, adjacent processing stages would not be able to operate concurrently, severely limiting the throughput of the pipeline.

The use of the new Handshake Components introduced earlier results in a more efficient pipeline implementation. Pipeline registers are not general-purpose variables: they are always written by a stage and then read by the following one. This write/read access pattern allows the use of a single variable as a true pipeline register. The resulting pipeline structure is shown in Figure 5. In this case, the registers are specified outside the processing stages and these contain only combinational logic.

```
procedure pipeReg
(
  parameter DataType : type;
  input  in  : DataType;
  output out : DataType
) is
    variable reg : DataType
begin
    loop
        in -> reg
      ;
        out <- reg
    end -- loop
end -- procedure pipeReg
```

Figure 6. Pipeline register specification.

Figure 6 shows how the new pipeline register is specified in Balsa. The specification is parameterised in the type of data that the register holds.

Figure 7 shows the handshake circuit for a Balsa-synthesised pipeline register. The data is stored in a latch (Variable component) and a Sequence component is used to

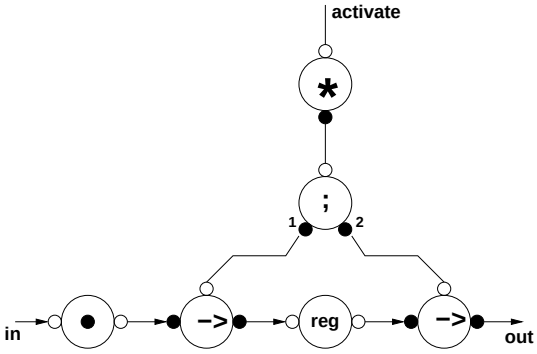

Figure 7. Synthesised pipeline register.

sequence the writing to and reading from the Variable. A Loop component repeatedly activates the Sequence component. A PassivatorPush component synchronises the input signal with the sequencing control to allow new data in only when the register is ready to accept it. The pipeline register implemented using the new Handshake Components turns out to be very simple and performs quite well compared to highly optimised controllers [15].

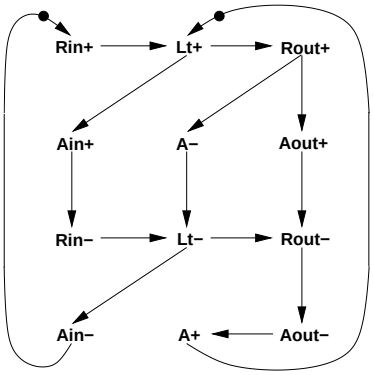

Figure 8. Pipeline control behaviour.

Figure 8 shows the behaviour of the pipeline controller. R and A represent the request and acknowledge signals used to implement the in and out channels, and Lt represents the latch enable signal.

The behaviour depicted in the figure shows that the Balsa-synthesised pipeline controller implements an efficient, fully-decoupled, request-activated protocol, similar to those presented in [15]. Figure 9 shows the implementation of the pipeline register controller as generated by the Balsa back-end. The $activate$ signal is used to initialise the register.

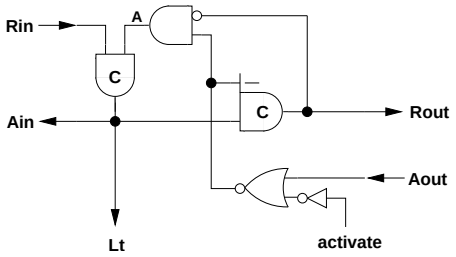

Figure 9. Pipeline control implementation.

3.3 True Asynchronous Operation

The pipeline structures shown in Figures 4 and 5 operate in a *pseudo-synchronous* fashion, i.e., the transfers of all data items from one stage to the next occur simultaneously as in synchronous systems. Although there is no global clock, data advances through the pipeline in *lockstep*, using local handshake channels. This *regular* operation is easy to understand and evaluate but can reduce the overall performance of the synthesised processor.

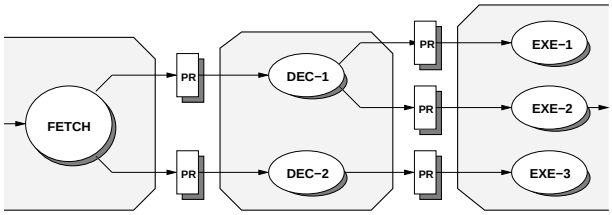

Figure 10. True asynchronous pipeline.

True asynchronous pipelines, on the other hand, make no attempt to have a lockstep operation, as shown in Figure 10. Each unit within a pipeline stage is allowed to progress at its own pace, handshaking individually with units in the previous and following stages. This means, for example, that different data items sent by units in the Decoder can arrive in the Execute stage at completely unrelated times. Consequently, different units in the Execute stage can operate on data items that correspond to different instructions, giving some of them a head start. Given the elastic nature of asynchronous pipelines, true asynchronous operation will result in improved performance in most cases.

3.4 Speculative Operation

Speculative operation is an important tool in the design of modern embedded processors. Significant performance improvements can be obtained if the results of speculative operations are useful most of the time. The ARM ISA establishes that all instructions are conditional, i.e., they can be executed or skipped depending on the condition codes. Program execution statistics show that most instructions are executed, opening the possibility of speculatively starting the instruction and throwing away the results only if the condition code test fails.

Speculative operation is not always straightforward to implement in synthesised asynchronous systems. In these systems, handshake channels, and not individual signals, are used to communicate data. A system is likely to deadlock if a channel is prevented from completing a handshake cycle. For this reason, SPA has no speculative operation: it evaluates the condition code of an incoming instruction and starts execution only if the condition passes.

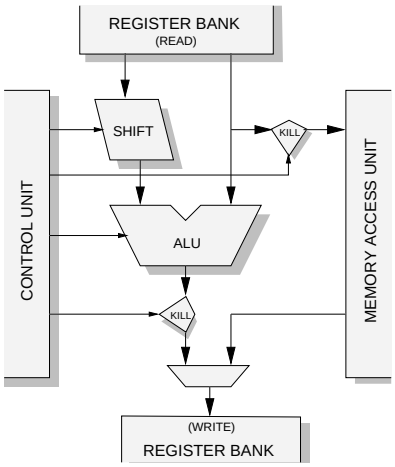

Figure 11. Speculative operation control.

A performance-oriented implementation can incorporate speculative operation in the execute unit: the evaluation of the condition code can be carried out concurrently with the execution of the instruction. If the condition fails the instruction is discarded at a checkpoint without any result being written back. The key issues are the location of the checkpoints and the need to allow all handshake channels to complete their cycles, as shown in Figure 11. In this example, data-processing operations are started speculatively and, if the condition test fails, are discarded before the write-back of the results. On the other hand, data memory operations are not started speculatively as the performance and power penalties for the discarded instruction would be extremely high. Clearly, this strategy will result in a performance improvement only if the percentage of executed instructions is high.

3.5 Optimal Combinational Logic

Pipeline stages usually require combinational logic, either for datapath operations or for control. The evaluation

of the condition code referred to in the previous section is a good example. Although Balsa supports a rich set of functions that can be used to implement this combinational logic, the use of handshake channels to communicate the data between several pre-designed Handshake Components may, in some cases, result in reduced performance.

On the other hand, Balsa supports a Case statement that can evaluate any expression using variable and channel values. To improve the results of the synthesis, Balsa Case commands are synthesised using espresso [16], a powerful logic minimiser, which produces efficient and-gate/or-gate PLA-style structures. The Balsa netlister uses the espresso results to synthesise a single Handshake Component that implements the required behaviour. C-element/or-gate structures are used in dual-rail implementations while and-gate/or-gate structures with added handshake signals are used in bundled data ones. The use of optimised Case statements results, in most cases, in significantly improved performance of the synthesised logic.

4 Results

This section shows the results of pre-layout, transistor-level simulations of several implementations of the SPA processor using a $0.18\mu m$ technology, standard cell library. The simulations show that the new components and techniques result in significant performance improvements.

The impact of the new components was evaluated using the original SPA specification. Table 1 contains the results of the execution of the Dhrystone benchmark program for different implementations of the SPA processor. The performance of the original SPA is set as the reference.

The table shows that eliminating the security elements of the dual-rail SPA –by introducing the new QDI adder and logic– improves its performance by 20% and reduces the size, in number of transistors, by 10% (see Table 3 below). Although this is an important improvement, it is not the most significant one.

The table also shows that the new implementations, which combine the use of all the new Handshake Components, obtain impressive 95% (bundled data) and 147% (dual-rail) performance improvements over the original SPA processors, with no significant increase in area. This clearly demonstrates the advantage of exploring and implementing alternative control behaviours over more efficient resynthesis of the same ones.

The performance-driven techniques described above were applied in the synthesis of nanoSpa, a new Balsa specification of the original SPA architecture. NanoSpa is organised as a 3-stage, Harvard-style pipeline, as shown in Figure 3, but has a few differences with respect to SPA: (i) the decoder stage lacks the Thumb module and the co-processor interface, (ii) the execute stage incorporates all the functional units present in SPA except for the multiplier and CLZ units –as it does not implement these instructions, and (iii) nanoSpa implements user and supervisor operating modes only, lacking the other ARM modes. Although not easy to evaluate, these differences should not have a large impact on the relative performances of the two implementations.

Table 2 shows the results of the execution of the Dhrystone benchmark program for the original SPA and different implementations of nanoSpa. The table includes results for bundled data and dual-rail versions. The performance of the original SPA is set as the reference.

The table shows that efficient pipeline control, true asynchronous operation, speculation and optimal combinational logic provide outstanding results: the basic nanoSpa cores, with the original Handshake Components, are a remarkable 2.85 (bundled data) and 3.65 (dual-rail) times faster that the original SPA implementations.

Table 2 also shows that the combination of the performance-driven specification with the use of the new Handshake Components results in very significant performance improvements, reaching 6.17 (bundled data) and 9.79 (dual-rail) times the performance of the original SPA.

Finally, Table 3 shows the transistor counts for the different processor implementations. It is clear from the table that the new Handshake Components, while significantly improving the performance of nanoSpa, have little impact on the size of the circuit. The large differences in transistor counts with respect to the original SPA indicate that there is enough room to incorporate the additional functionality without a large impact on the performance. Preliminary estimates based on work in progress suggest that a full-featured nanoSpa will require close to 70% of the number of transistors in SPA.

5 Conclusions

The work presented in this paper confirms that syntax-directed compilation is a powerful synthesis approach and, combined with an efficient set of Handshake Components, can automatically generate efficient asynchronous systems for complex, real world applications.

Extensive simulation results show that the introduction of new Handshake Components, used to implement parallel, sequential and input control, can double the performance of existing designs without the need to modify the source descriptions. Additionally, the introduction of new performance-oriented techniques used to implement efficient pipeline control, true asynchronous behaviour and speculative operation can triple the performance of existing designs. The combination of all new components and techniques has been shown to generate a new implementation of an existing 32-bit, ARM-compatible processor with

Processor	Bundled Data		Dual-Rail	
	Dhrystone MIPS	Relative Performance	Dhrystone MIPS	Relative Performance
SPA	9.57	1.00	6.18	1.00
SPA1 (SPA + new QDI adder and logic)	n/a		7.39	1.20
SPA2 (SPA1 + new control HCs)	18.68	1.95	15.26	2.47

Table 1. Impact of new handshake components.

Processor	Bundled Data		Dual-Rail	
	Dhrystone MIPS	Relative Performance	Dhrystone MIPS	Relative Performance
SPA	9.57	1.00	6.18	1.00
nanoSpa	27.28	2.85	22.57	3.65
nanoSpa1 (nanoSpa + new QDI adder and logic)	n/a		25.27	4.09
nanoSpa2 (nanoSpa1 + new control HCs)	59.04	6.17	60.53	9.79

Table 2. Impact of performance-driven specification and new handshake components.

up to ten times the performance of the original one.

References

[1] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, March 1997.

[2] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, and L. A. Plana. MINIMALIST: An environment for the synthesis and verification of burst-mode asynchronous machines. In *Proc. International Workshop on Logic Synthesis*, June 1998.

[3] TIMA Laboratory, Concurrent Integrated Systems Group. TAST: Tool for asynchronous circuit synthesis. *http://tima.imag.fr/cis/Tast/tast.html*, 2002.

[4] A. J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C.A.R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64, Addison-Wesley, Reading MA, 1990.

[5] Kees van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*. Cambridge University Press, 1993.

[6] A. Peeters and K. van Berkel. Single-rail handshake circuits. In *Proc. Working Conf. on Asynchronous Design Methodologies*, pages 53–62, May 1995.

[7] D. Edwards and A. Bardsley. Balsa: An asynchronous hardware synthesis language. *Computing Journal*, 45(1):12–18, 2002.

[8] Hans van Gageldonk, Daniel Baumann, Kees van Berkel, Daniel Gloor, Ad Peeters, and Gerhard Stegmann. An asynchronous low-power 80c51 microcontroller. In *Proc. International Symposium on Asynchronous Circuits and Systems*, pages 96–107, 1998.

[9] L. A. Plana, P. A. Riocreux, W.J. Bainbridge, A. Bardsley, J. D. Garside, and S. Temple. SPA – a synthesisable Amulet core for smartcard applications. In *Proc. International Symposium on Asynchronous Circuits and Systems*, pages 201–210. IEEE Computer Society Press, April 2002.

[10] Q.Y. Zhang and G. Theodoropoulos. Towards an asynchronous MIPS processor. In *Cryptographic Hardware and Embedded Systems (CHES 2003)*, volume 2779 of *Lecture Notes in Computer Science*, pages 137–150. Springer-Verlag, 2003.

[11] Tilman Kolks, Steven Vercauteren, and Bill Lin. Control resynthesis for control-dominated asynchronous designs. In *Proc. International Symposium on Asynchronous Circuits and Systems*, pages 233–243. IEEE Computer Society Press, March 1996.

	Bundled Data		Dual-Rail	
Processor	Transistors	Relative Size	Transistors	Relative Size
SPA	280,682	1.00	705,389	1.00
SPA1 (SPA + new QDI adder and logic)	n/a		640,013	0.91
SPA2 (SPA1 + new control HCs)	300,314	1.07	656,154	0.93
nanoSpa	110,765	0.39	349,003	0.49
nanoSpa1 (nanoSpa + new QDI adder and logic)	n/a		320,325	0.45
nanoSpa2 (nanoSpa1 + new control HCs)	149,341	0.53	325,169	0.46

Table 3. Transistor counts for the different processor implementations.

[12] T. Chelcea and S. M. Nowick. Resynthesis and peephole transformations for the optimization of large-scale asynchronous systems. In *Proc. ACM/IEEE Design Automation Conference*, 2002.

[13] Luis A. Plana, Sam Taylor, and Doug Edwards. Attacking control overhead to improve synthesised asynchronous circuit performance. In *Proc. International Conf. Computer Design (ICCD)*, pages 703–710. IEEE Computer Society Press, October 2005.

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

[15] J. Liu. *Arithmetic and control components for an asynchronous microprocessor*. PhD thesis, Department of Computer Science, University of Manchester, 1997.

[16] R.L. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Trans. on Computer-Aided Design*, 6(5):727–750, September 1987.