Sim-async: an Architectural Simulator for Asynchronous Processor Modeling using Distribution Functions

J. M. Colmenar[1], O. Garnica[2], J. Lanchares[2], J. I. Hidalgo[2], G. Miñana[2], S. Lopez[2]

[1] C. E. S. Felipe II, Complutense U. of Madrid
jmcolmenar@cesfelipesegundo.com
[2] Dept. of Computer Arch. and System Engineering, Complutense U. of Madrid
{ogarnica, julandan, hidalgo}@dacya.ucm.es,
guamiro@fdi.ucm.es, slopezal@dacya.ucm.es

Abstract. In this paper we present *sim-async*, an architectural simulator able to model a 64-bit asynchronous superscalar microarchitecture. The aim of this tool is to serve the designers on the study of different architectural proposals for asynchronous processors. *Sim-async* models the data-dependant timing of the processor modules by using distribution functions that describe the probability of a given delay to be spent on a computation. This idea of characterizing the timing of the modules at the architectural level of abstraction using distribution functions is introduced for the first time with this work. In addition, *sim-async* models the delays of all the relevant hardware involved in the asynchronous communication between stages.
To tackle the development of *sim-async* we have modified the source code of SimpleScalar by substituting the simulator's core with our own execution engine, which provides the functionality of a parameterizable microarchitecture adapted to the Alpha ISA. The correctness of *sim-async* was checked by comparing the outputs of the SPEC2000 benchmarks with SimpleScalar executions, and the asynchronous behavior was successfully tested in relation to a synchronous configuration of *sim-async*.

1 Introduction

Due to the current integration level and clock frequencies in microprocessor architectures, synchronization with a single clock source and negligible skew is an extremely difficult task. Fully asynchronous designs built using self-timed circuits replace the clock signal by local synchronization protocols. Then, these systems have no problems associated with the clock signal, and the global circuit performance corresponds to the performance of the average case because a new computation starts immediately after the previous has finished [1].

In the field of fully asynchronous systems, designers usually develop general purpose processors (like those presented in [2,3,4,5]) using high-level description languages like Occam, Tangram, Balsa or VHDL++. In addition, some works

like [6,7] have proposed simulators of asynchronous processors, but they are slightly parameterizables and they do not model the asynchronous behavior at the architectural level of design. Albeit, these simulators are not able to run standard benchmarks.

As occurs in the synchronous paradigm, asynchronous systems designers need infrastructures for computer system modeling that abstract the implementation of hardware models. These infrastructures must be capable of model the data-dependant delay of a fully asynchronous system at the architectural level of abstraction, and also they have to be able to run complete applications. The main example of such a configurable, flexible and wide-spread toolset in the synchronous world is SimpleScalar [8]. SimpleScalar allows to modify cache, branch predictor or any other architectural parameter, and is able to run standard benchmarks in order to get comparable measures for any kind of data related to performance and also to custom statistics. Up to our best knowledge, such flexible infrastructures for simulation and architectural modeling of high-performance fully asynchronous processors have not been reported in literature.

Once argued the necessity of a modeling infrastructure, one of the key questions is how the tool will model the data-dependant computation delays of the modules that form an asynchronous processor. Since asynchronous circuits take distinct amounts of time when computing different values, it is possible to collect a large set of delays for a given circuit by running low-level simulations using a representative number of inputs. From that set of delays one may obtain the distribution function which characterizes the behavior of the circuit. *Sim-async* applies this idea inside out, that is, the simulator uses distribution functions (included as parameters) to dynamically select the delay for each computation of each one of the modules of the processor. This solution is introduced in this paper as a novelty related to the architectural asynchronous processor simulation.

Therefore, in this paper we present *sim-async*, an architectural simulator for asynchronous superscalar processor modeling. *Sim-async* is able to model, at the architectural level of abstraction, the data-dependant behavior of the modules of the processor by using distribution functions. In addition, *sim-async* is able to execute any test program compiled for the Alpha ISA, as SimpleScalar does.

The rest of the paper is organized as follows: Section 2 is devoted to describe the simulated processor microarchitecture and the functionality of its stages. In Section 3 we define the synchronization domains and detail the delays that model them, the implementation of those delays as input parameters of the simulator, and the communication protocol between the domains. In Section 4 we show the validation of the simulator by running the SPEC2000 benchmarks under both asynchronous and synchronous configurations. Finally, in Section 5 we explain the conclusions and the future work.

2 Description of the Processor Microarchitecture

Sim-async models the microarchitecture of a 64-bit fully asynchronous superscalar processor with out-of-order and speculative execution of instructions, and

this section is devoted to its introduction. The processor consists on five stages: *fetch*, *issue*, *exec*, *write-back* and *commit*. In Figure 1 we show the schema of the microarchitecture[3], where we have illustrated the Exec Unit with higher detail[4].

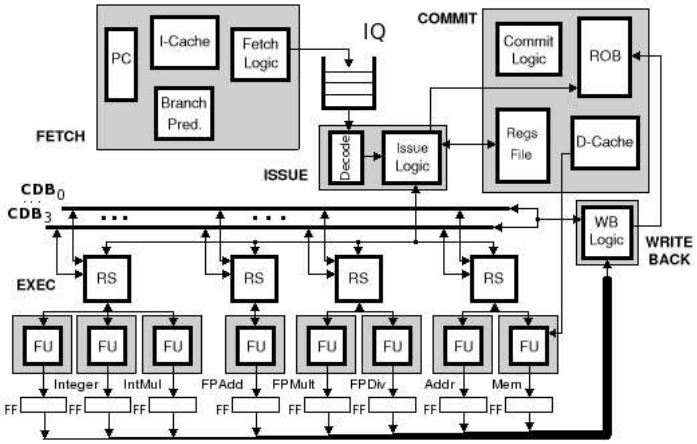


Fig. 1. Schema of the modeled microarchitecture. The logic involved in the communication between modules is not included within this schema.

The implementation of the asynchronous processor is identical to the synchronous one, but substituting the clock network by a set of components that allows the communication of results between modules. For the sake of clarity, we briefly describe the functionality of each stage in this section.

Fetch. A parameterizable number of instructions is read from the I-cache taking into account the branch prediction. The instructions are moved to the instruction queue (IQ), where they wait for the issue stage. If one of the instructions in the middle of the fetch group is a taken conditional branch or an unconditional branch, then the subsequent instructions in the fetch group will be discarded.

Issue. As it is well-known, the design of the issue stage is crucial to obtain high performance on a superscalar processor. We have chosen the implementation called *instruction shelving* with reorder buffer (ROB) [9] for the issue stage because it decouples the instruction issue and the dependency checking. With shelving, the only fact that will provoke the block of the issue of instructions

[3] The twelve shadowed areas of the figure represent the different synchronization domains we have defined in the processor, but we postpone its explanation until the following section.

[4] This level of details of the execution unit will be useful in the following sections.

is the lack of free entries in the reservation stations (RS, or *shelving buffers*) or ROB, not the data dependencies, which are more frequently to appear.

This stage decodes and in-order issues a parameterizable number of instructions from the IQ to their corresponding RS and to the ROB. The issue is performed in-order because preserving sequential consistency for out-of-order issue requires a much higher effort than in-order issue does. In addition, due to the rarely blocking of issue with shelving, implementing out-of-order issue would only have a marginal benefit [9].

Execution. The RS preserve data dependencies maintaining the tags of the instructions which will generate the pending operands, and hold values waiting for the execution into the functional units (FU). As shown in Figure 1, the microarchitecture is provided with four RS. The dispatch logic decides which one of the ready instructions from the RS is issued to its corresponding FU taking into account that as older the ready instruction as sooner it is issued.

Write-back. Once the computation of each FU is finished, the result is held on its output flip-flop, triggered by a capture signal, till the write-back stage was completed. In this stage, the selection logic chooses the results to be distributed to the RS and the ROB through the number of instances (parameterizable) of the common data bus (CDB), also sending the tag of the instructions which generated each result. The wake-up logic of each RS compares the incoming tags with the tags of the pending instructions performing the update of values wherever a tag matches.

Commit. Each instruction at the ROB holds the result to be written to the register file or to the memory and the destination register or memory address. A parameterizable number of instructions is retired from the ROB maintaining program order, and branch prediction is checked each time this stage executes. Precise interruptions are also checked and the pipeline is flushed when a mispredicted branch is processed.

3 Modeling the Asynchrony

A synchronization domain consists in all the flip-flops triggered by the same signal and the combinational logic within their fan-in. In this paper we have defined twelve synchronization domains (see shadowed areas in Figure 1), where the communication between them is performed using a four-phase handshake protocol. The following subsections are devoted to present the temporal modeling of an individual domain and the assumed communication protocol.

3.1 Temporal Modeling of a Synchronization Domain

We have followed a mixed approach in the asynchronous paradigm to describe the temporal behavior of the domains. We have used the computation completion

mechanism described in [10] to detect the end of the computation, and we have employed a bounded delay approach to model the behavior of the control logic.

In the asynchronous systems the delay spent on computing a data, detecting the computation completion and communicating the result to the receiver module through the synchronization protocol takes a different and unpredictable value for each input data. That delay comes from the combination of several other delays that appear during the operation of a module. Let's examine these delays, indicated with dotted lines in the scheme of Figure 2 (a).

The *computation delay*, t_c , is the delay spent by the module on computing the input data and generating the results. It is a variable delay because asynchronous circuits present a data-dependant behavior. In our simulator, each stage and FU of the microarchitecture receives its own t_c as a distribution function. Then, whenever the module makes a computation, the simulator randomly selects a computation delay taking into account the shape of the distribution. Thus, the actual delay for computing these data is not obtained, but the data-dependant behavior of the module is maintained.

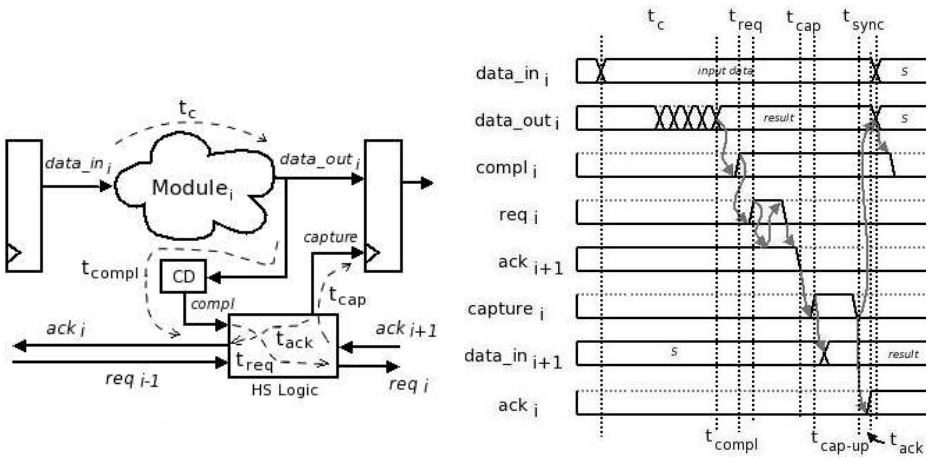
The *completion detection delay*, t_{compl} , corresponds to the time spent by the completion detection logic (CD) on detecting a valid output and asserting the *compl* signal. This delay is included as a constant input parameter on *sim-async*.

We use a delay insensitive codification and a completion detection logic due to the variability of t_c . Therefore, as Martin showed in [11], the modules alternate a neutral or synchronization value (S) which does not mean any Boolean value, and the encoding of a valid output. The generation of that synchronization value takes t_{sync} time units (t.u.) and, after that, the module is ready to receive new incoming data. This delay is also an input parameter of *sim-async*. The logic that orders the generation of the synchronization value is omitted in the mentioned figure for the sake of clarity.

The modeling of the handshake protocol is divided on two delays: *request delay,* t_{req} , which is the time spent from the assertion of the *compl* signal to the assertion of the request signal, req_i ; and *capture delay,* t_{cap} , which is the time spent from the falling edge of the acknowledge signal from the receiver module, ack_{i+1} , and the assertion of the *capture* signal. The time spent during the handshake is an uncertain delay that can be accurately obtained only by simulation because it mainly depends on the occupation or availability of the structures of the receiver module at each moment. Both t_{req} and t_{cap} are included as constant input parameters on *sim-async*.

Once the protocol is completed, the *capture* signal is asserted as a pulse. This assertion does not violate any timing assumptions because we consider t_{compl} to be longer than the setup delay of the destination register. In addition, the width of the pulse of *capture*, denoted as t_{cap-up} , must be higher than the hold delay of the register triggered by the *capture* signal because the generation of the synchronization value is ordered by the falling edge of that pulse. The t_{cap-up} delay is included as another constant input parameter of our simulator.

The delay spent from the fall of the *capture* signal and the assertion of the ack_i signal is denoted as t_{ack} , also included as an input parameter of *sim-async*.



(a) (b)

Fig. 2. (a) General scheme of a synchronization domain. Dotted lines are referred to delays. (b) Chronogram showing the delays of the logic involved on one computation of the module i and the communication of results to the next domain.

3.2 Communication Protocol

The communication between domains is performed through channels implementing a four-phase handshake protocol like the one described in [12]. Figure 2 (b), shows the chronogram of an example of communication between the domain i and its neighbor. We next explain this communication using the delays defined in the previous subsection.

The moment when the module starts to compute is the instant in which the $data\_in_i$ signal propagates the input data. Then, the module processes these data and, after a data-dependant delay, t_c , the result is propagated through the $data\_out_i$ signal. The *compl* signal is asserted after t_{compl} t.u. and then the handshake logic activates the req_i signal in order to start the communication protocol. The receiver module is ready to process new data because ack_{i+1} is asserted. At that point, the handshake logic deasserts the request signal and waits for the fall of ack_{i+1} . The receiver module unsets the acknowledge signal and the communication protocol ends. After that, the handshake logic generates a pulse in the *capture* signal. On the raising edge of *capture* the destination register latches the results of the module and, on the falling edge of *capture*, the logic of the module return to the synchronization value before the next computation. In addition, the falling edge of *capture* also provokes the assertion of ack_i , which indicates that the module i is ready to receive new input data.

4 Experimental Results

In order to validate *sym-async* we have run the SPEC2000 benchmarks on different timing configurations of the simulator. Then, we have compared the results of these executions with those obtained from the original SimpleScalar (*sim-safe* flavor) under the same cache and branch predictor configuration. The tests were run parameterizing *fetch, issue, write-back* and *commit* stages to process up to four instructions each time they execute. Table 1 (a) shows the architectural configuration of the microarchitecture.

Table 1. (a) Architectural configuration of the microarchitecture in the simulations. (a) Worst case delay of stages and FU operations in the asynchronous simulations.

Branch Predictor:	2-level PAg
Level 1	1024 entr, his 10
Level 2	1024 entr
BTB	4096 sets, 2-way
Instructions queue (IQ) size	100 entries
Integer RS queue size	6 entries
FP Addition RS queue size	3 entries
FP Mul, FP Div/Sqrt RS queue size	2 entries
Memory RS queue size	5 entries
Integer / FP Register File	32 / 32
ROB size	100 entries

(a)

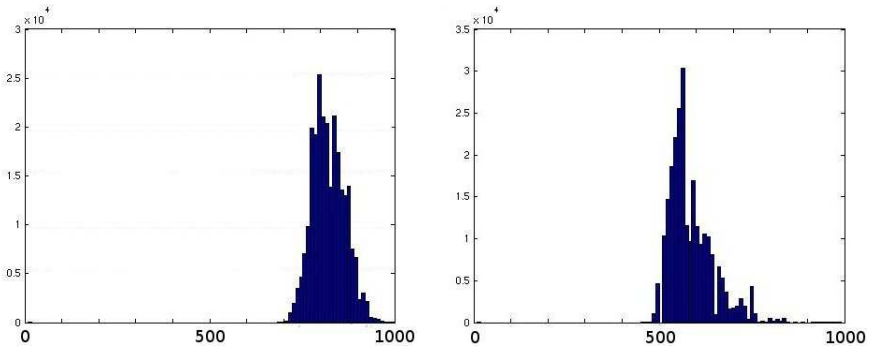
Stage / FU Operation	T. U.
Fetch, Issue, Int/Logic, WB, Commit	1000
IntMul	7000
MemLoad, FPAdd, FPMul	4000
FPDiv/Sqrt	30000

(b)

The first timing configuration tested was the fully asynchronous one. In this asynchronous configuration we used two distribution functions to characterize the computation delays of the modules: slow case (SC) and medium case (MC) functions. These functions were selected from the set of back-annotated gate-level simulations of related asynchronous circuits, and were normalized to the same upper bound (the worst case) of 1000 t.u..

The slow case (SC) function, shown in Figure 3 (a), whose average delay is near the worst delay, represents a slow behavior because the most of the data take a high delay. We had not made any assumptions about the implementation of the functional units, so they were individually characterized through the SC function. However, we considered the use of long-latency non-pipelined FU for FP operations and integer multiplications, so the normalization of the function was conveniently corrected to a higher upper bound for these slow non-pipelined FU, according with the Table 1 (b).

The medium case (MC) function, presented in Figure 3 (b), describes an asynchronous behavior where the average delay is close to the half of the worst delay. We have use this function to characterize the rest of the stages: *fetch*, *issue*, *write-back* and *commit*.



(a) (b)

Fig. 3. (a) Slow case (SC) and (b) Medium case (MC) distribution functions.

It is important to remark that the aim of these functions is not to be actual patterns of the modules of the modeled processor. We present these functions as typical examples of asynchronous circuit behaviors obtained from previous low-level simulations.

In order to establish the delays of the control logic, we have considered the work of Cheng in [10]. In that paper Cheng implemented a circuit for completion detection and synchronization (reset completion-detection) of data lines using a four-phase handshake protocol and dual-rail codification. He obtained an average delay of 0.28 ns for the completion detection circuit and 0.71 ns for the synchronization (reset). Considering that digital IC performance has tracked Moore's Law and improved by 30% annually, the delays of that circuit using current technology could be about 16 ps and 41 ps respectively.

Supposing that the critical path of the modules of the processor will be under 1.25 ns (that means a maximum frequency of 800 MHz in a synchronous version), which we have normalized to 1000 t.u., the normalized values for t_{compl} and t_{sync} taking into account the scaling are 12.8 t.u. and 32 t.u. respectively, which correspond to average delays. In our simulations we have conservatively doubled that delays to 26 t.u. and 64 t.u. for all the modules. The rest of the control logic delays, t_{req} , t_{cap} and t_{cap-up} were fixed to 5, 5 and 10 t.u. respectively, and t_{ack} was considered equal to t_{sync} .

We have checked that the outputs obtained by *sim-async* running the SPEC2000 benchmarks under the asynchronous configuration are identical to those generated by SimpleScalar for all the benchmarks. That is, *bzip* generates the same compressed file, *gcc* returns the same compilation statistics, and so on. In addition, we have compared the number of instructions committed on both simulators for the execution of those benchmarks and they only differ in a negligible range between 0.21% and -0.012% (attributed to the slightly different implementation

of the system calls), as shown in Table 2 (a). Therefore, *sim-async* performs correct simulations and successfully executes the Alpha ISA.

Table 2. (a) Number of instructions committed for several SPEC2000 benchmarks on SimpleScalar (*sim-safe*) and for *sim-async* under the asynchronous configuration. (b) Average differences between the instructions executed and the use of modules of *sim-async* on synchronous and asynchronous configurations running the SPEC2000.

SPEC	SimpleScalar	Async *Sim-async*	Diff (%)
ammp	45812883	45810845	-0.004
apsi	197579651	197612776	0.017
bzip	1819780172	1819780267	0.000
crafty	94419973	94420229	0.000
galgel	139306245	139310055	0.003
gap	82873902	82874407	0.001
gcc	2016139124	2016204817	0.003
gzip	601857009	601857104	0.000
lucas	19239488	19242782	0.017
mesa	1608605448	1608410610	-0.012
parser	268979662	269006191	0.010
perlbmk	205853718	205914747	0.030
sixtrack	11699655	11724227	0.210
swim	23557475	23562358	0.021
vortex	453666	454534	0.191

(a)

Async vs. Synch	% Avg Diff
# Insn Exec	0.132
Use of Fetch	-42.076
Use of Issue	-61.993
Use of Int	-66.062
Use of IntMul	-99.894
Use of FPAdd	-95.665
Use of FPMul	-97.733
Use of FPDiv	-99.921
Use of Addr	-79.314
Use of Mem	-81.116
Use of WB	-52.324
Use of Commit	-73.852

(b)

With the aim of test that *sim-async* not only executes the Alpha ISA correctly, but it also correctly models the asynchronous behavior, we have made the comparison between the former simulations and those resulting from *sim-async* parameterized in order to model a synchronous processor. This is possible because the synchronous behavior is a particular case of the asynchronous one. That is, in a synchronous processor all the modules spend the same time on computing a data (the worst case of the slowest stage) and the communication protocol spends a delay of zero t.u. due to the clock signal.

Then, we set the parameters t_{compl} , t_{req} , t_{cap} , t_{cap-up} , t_{sync} and t_{ack} to zero t.u., and t_c was fixed to a distribution where all the delays were 1000 t. u. long, the worst case of the asynchronous simulations, but considering the slowest FU (IntMul, FPAdd and FPMul/Div) as fully-pipelined units. The *capture* signal (see Section 3) is only asserted if the receiving module is ready to accept new input data.

The synchronous simulations were run under the same architectural configuration described for the asynchronous simulations, and we obtained identical outputs and also identical number of committed instructions. In addition, we took some statistics in order to measure the asynchronous behavior. As shown

in Table 2 (b), the number of instructions executed (including those speculative) is, on average, 0.132 % higher in the asynchronous configuration. This occurs because the average delays of the asynchronous stages are shorter than the synchronous worst case. Then, the asynchronous microarchitecture is able to advance on the execution of instructions faster than the synchronous one.

Albeit, the number of executions of the asynchronous modules is reduced in relation to the synchronous simulations. The average reduction ranges from the 42.076 % of the fetch stage to the 99.921% of the FPDiv functional unit, which remains idle almost all the time (see Table 2 (b)). This behavior corresponds to the one expected for an asynchronous circuit because the modules only compute when useful work has to be performed.

As an additional statistic, the speedup reached by the asynchronous configuration in relation to the synchronous one is, on average, 1.135 for the SPEC2000.

Thus, this comparison between both asynchronous and synchronous simulations verifies the correct modeling of the asynchronous behavior that *sim-async* performs by using distribution functions to characterize the computation delay of the modules of the microarchitecture.

5 Conclusions and Future Work

In this paper we have presented *sim-async*, an architectural simulator able to correctly model the behavior of a 64-bit asynchronous superscalar microarchitecture at the architectural level of abstraction. To tackle this goal, we have modified the source code of SimpleScalar by substituting the simulator's core with our own execution engine which provides the functionality of a parameterizable superscalar architecture adapted to the Alpha ISA.

In order to provide flexibility, we have defined twelve synchronization domains, and the delays involved on their computation, including them as parameters of *sim-async*. Albeit, due to the necessity of modeling a data-dependant behavior of the modules which form the simulated microarchitecture, we have introduced the idea of modeling the data-dependant computation delay of the modules by using distribution functions.

We have verified the correctness of *sim-async* by comparing the outputs of the SPEC2000 benchmarks run on the original SimpleScalar with those generated by *sim-async*. In addition, we have run simulations of *sim-async* where the delays were defining a synchronous microarchitecture. The number of instructions executed (including those speculative) was, on average, 0.132 % higher in the asynchronous configuration. This occurs because the average delays of the asynchronous stages are shorter than the synchronous worst case. In addition, the number of executions of the asynchronous modules suffered an important reduction in relation to the synchronous simulations. This behavior corresponds to the one expected for an asynchronous circuit because the modules only compute when useful work has to be performed. Then, the comparison between the asynchronous and the synchronous simulations shows that the modeling of the asynchronous behavior is correct. In addition, the asynchronous configuration of

the processor presented an average speedup of 1.132 in relation to its synchronous counterpart.

Currently we are working on two ways: on one hand, we are tuning *sim-async* with the aim of reducing its execution time, which is still high (about thirty six hours each set of benchmarks). On the other hand, we are working on the implementation of the asynchronous modules of the microarchitecture in order to reach higher performance.

Acknowledgments

This research has been supported by Spanish Government Grant number TIN2005-05619.

References

1. D. Kearney, "Theoretical Limits on the Data Dependent Performance on Asynchronous Circuits," *Proc. of Intl. Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 201–207, 1999.
2. A. J. Martin, A. Lines, R. Manohar, M. Nystroem, P. Penzes, R. Southworth, and U. Cummings, "The Design of an Asynchronous MIPS R3000 Microprocessor," in *Adv. Research in VLSI*, pp. 164–181, 1997.
3. D. K. Arvind and R. D. Mullins, "A Fully Asynchronous Superscalar Architecture," in *Proc. of the 1999 Intl. Conf. on Parallel Architectures and Compilation Techniques* (I. C. S. Press, ed.), pp. 17–22, 1999.
4. J. D. Garside, W. J. Bainbridge, A. Bardsley, D. M. Clark, D. A. Edwards, S. B. Furber, J. Liu, D. W. Lloyd, S. Mohammadi, J. S. Pepper, O. Petlin, S. Temple, and J. V. Woods, "AMULET3i - An Asynchronous System-on-Chip," in *Proc. of the 6th Intl. Symposium on Advanced Research in Asynchronous Circuits and Systems* (I. C. S. Press, ed.), pp. 162–175, April 2000.
5. Q. Zhang and G. Theodoropoulos, "Modelling SAMIPS: a Synthesisable Asynchronous MIPS Processor," in *Proc. of the 37th Annual Simulation Symposium*, pp. 205–212, April 2004.
6. C. Chien, M. A. Franklin, T. Pan, and P. Prabhu, "ARAS: Asynchronous RISC Architecture Simulator," *Proc. of the 2nd Working Conference on Asynchronous Design Methodologies (ASYNC'95)*, 1995.
7. V. Rebello, *On the Distribution of Control in Asynchronous Processor Architectures.* PhD thesis, 1997.
8. T. M. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer Journal*, vol. 35, 2, February 2002.
9. D. Sima, "Superscalar Instruction Issue," *IEEE Micro*, vol. 17, pp. 28–39, Sep.-Oct. 1997.
10. F. Cheng, "Practical Design and Performance Evaluation of Completion Detection Circuits," in *Proc. of the Intl. Conf. on Computer Design* (I. C. S. Press, ed.), pp. 354–359, 1998.
11. A. J. Martin, "Asynchronous Datapaths and the Design of an Asynchornous Adder," *Formal Methods in System Design*, vol. 1, pp. 119–137, July 1992.
12. T. H.-Y. Meng, R. W. Brodersen, and D. G. Messerschmitt, "Asynchronous Design for Programmable Digital Signal Processors," *IEEE Trans. on Signal Processing*, vol. 39(4), pp. 939–952, 1991.