Modelling SAMIPS: A Synthesisable Asynchronous MIPS Processor

Q. Zhang & G. Theodoropoulos
School of Computer Science, The University of Birmingham
Birmingham B15 2TT, UK
email: {qyz,gkt}@cs.bham.ac.uk

Abstract

The last fifteen years have witnessed a resurgence of interest in asynchronous digital design techniques as they promise to liberate VLSI systems from clock skew problems, offer the potential for low power and high performance and encourage a modular design philosophy which makes incremental technological migration a much easier task. This activity has revealed a need for modelling and simulation techniques suitable for the asynchronous design style. The concurrent process algebra Communication Sequential Processes (CSP) is increasingly advocated as particularly suitable for this purpose. This paper discusses the modelling of SAMIPS, a synthesisable asynchronous MIPS processor core, in Balsa, a CSP-based, asynchronous hardware description language and synthesis tool.

1 Introduction

A digital system is typically designed as a collection of subsystems, each performing a different computation and communicating with its peers to exchange information. Before a communication transaction takes place, the subsystems involved need to synchronise, namely to wait for a common control state to be reached, which guarantees the validity of data exchanged.

The predominant synchronization technique in hardware design today is the utilisation of a global clock whose transitions define the points in time when communication transactions can take place. This synchronous approach however has reached a critical point, with clock distribution becoming an increasingly costly and complicated issue.

Thus, the last decade has witnessed an explosion of interest in asynchronous design techniques, which do not rely on global clocks but achieve synchronization by means of localized synchronization protocols between the communicating subsystems. These protocols are typically in the form of local request and acknowledge signals, which provide information regarding the validity of data signals. An example of such a protocol is the *two-phase bundled data* handshake synchronisation protocol illustrated in figure 1.

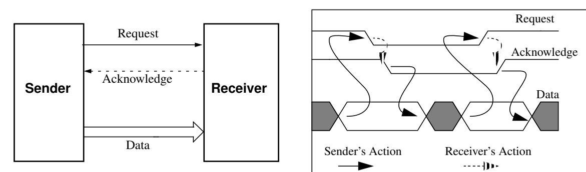


Figure 1. The Two-Phase Bundled Data Interface Protocol

Other potential advantages of asynchronous logic are low power consumption, high performance and support for a modular design philosophy which makes incremental technological migration a much easier task.

Several asynchronous design techniques have been developed [24] and are progressively finding their place in the mainstream VLSI design, not least in the development of GALS (Globally Asynchronous Locally Synchronous) systems [13]. A number of asynchronous processors have been developed including NSR and Fred at the University of Utah, STRiP at Stanford University Sun's Counterflow pipeline processor FAM and TITAC at Tokyo University and Institute of Technology respectively, Hades at the University of Hertfordshire, Sharp's Data-Driven Media Processor, CalTech's processors and Lutonium and the series of asynchronous implementations of the ARM RISC processor (AMULET1, AMULET2e, AMULET3 and SPA) developed by the AMULET group at the University of Manchester.

SAMIPS is an asynchronous implementation of the MIPS processor currently under development at the University of Birmingham, UK. This work forms part of a larger project, which aims to develop an integrated framework for formal verification and distributed simulation of Asynchronous Hardware, utilising Balsa, a CSP-oriented modelling and synthesis tool developed at the University of Manchester [1]. The project is jointly undertaken by the Modelling and Analysis of Systems group at the University of Birmingham and the AMULET group at the University of Manchester [20].

In previous papers we have presented the design of SAMIPS and the techniques that have been developed to deal with data and control hazards [36]. This paper focuses on modelling and simulation issues, describing how SAMIPS has been modelled in Balsa and presenting some initial simulation results.

2 Modelling and Simulation of Asynchronous Hardware

The need to deal with the ever increasing size and complexity of computer system designs and quality assurance, reliability and relentless time-to-market pressures have assigned a crucial role to modelling and simulation in computer and telecommunication industries [30]. Modelling and simulation are essential tools for measuring the performance as well as validating the timing behaviour and functional correctness of alternative architectural designs. Several simulation modelling languages and tools for synchronous logic design have been developed and have underpinned the development of ever more complex synchronous VLSI circuits.

In the case of asynchronous systems, the role of simulation is even more crucial as their concurrent, non-deterministic behaviour makes any attempt to reason about their correctness and performance a very complicated task. This complexity renders modelling and simulation essential tools in the endeavour to gain an insight and understanding of the behaviour of asynchronous systems [26, 27, 28, 29]. However, although synchronous languages and tools can and indeed have been used for asynchronous hardware too, their application in that context is proving awkward and inefficient. Fundamentally, conventional, sequential, synchronous hardware description languages are not suitable for describing concurrent non-deterministic asynchronous behaviour. Thus, the recent interest in asynchronous design has fuelled an intense research activity aiming to develop techniques appropriate for modelling and simulating asynchronous systems.

I-Nets [19], Signal Graphs [22], Petri Nets (e.g. the Petrify tool [7]), Signal Transition Graphs (STGs) [5] (e.g. the Versify [32], CASCADE [2], SIS [23] and the IMEC Laboratory [34] tools), State Transition Diagrams (e.g. the MEAT tools [8]), Asynchronous Finite State Machines (e.g. the 3D [35] and MINIMALIST [11] tools), ST-V (the Self-Time Verilog developed by Cogency Technology Inc. [6]) and CCS (by Birtwistle et. al. at the University of Leeds [17]) are some of these tools and formalisms that have been employed in asynchronous logic design.

The computation and communication semantics of Communicating Sequential Processes (CSP, the concurrent process algebra developed by Tony Hoare for the specification of parallel systems [14]), in particular, has been extensively advocated as particularly suitable for describing the behaviour of asynchronous systems.

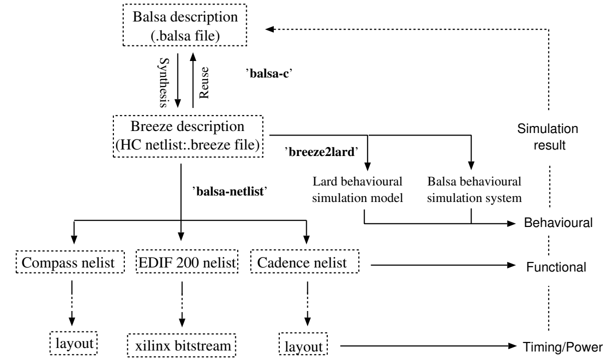


Figure 2. Balsa System

Several asynchronous modelling approaches and systems have been developed which use CSP-based notations, including Theodoropoulos's [25, 26, 27, 28, 29], Martin's [18], Hulgaard's [15] and Brunvand's [4] work, *trace theory* [9], *Delay-Insensitive algebra* [16], *Tangram*[31], *SHILPA* [12], *LARD* [10] and Balsa [1].

3 Balsa

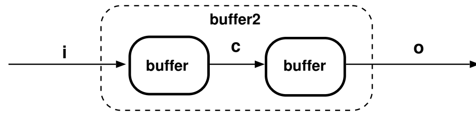
Balsa (figure 2) is both an asynchronous hardware synthesis framework and the language for describing such systems. It has been demonstrated by synthesising the DMA controller of Amulet3i as well as SPA, an AMULET core for smartcard applications. Balsa uses CSP-based constructs to express Register Transfer Level design descriptions in terms of channel communications and fine grain concurrent and sequential process decomposition.

Descriptions of designs (*.balsa* file) are then translated (***balsa-c***) into implementations in a syntax directed-fashion with language constructs being mapped into networks of parameterised instances of "handshake components" (*.breeze* file) each of which has a concrete gate level implementation [3].

A number of tools are available to process the breeze handshake files. ***balsa-netlist*** automatically generates CAD native netlist files, which can then be fed into the commercial CAD tools that further synthesize the netlist to the fabricable layout. Three commercial CAD systems are currently supported: Compass Design Automation tools from Avant, Xilinx Alliance FPGA design tools and Cadence Design Framework II.

Balsa uses one-to-one mapping between the language constructs in the specification and the intermediate handshake circuits that are produced. The transparency of compilation makes it relatively easy for an experience user to trace the incremental changes made at the language down to the circuit implementation level.

Three levels of simulation are supported. Balsa has a simple behavioural simulator while memory related behavioural simulation relies on the LARD toolkit.




```
import [balsa.types.basic]         (1)
type word is 16 bits               (2)

procedure buffer(input I: word;    (3)
                 output O: word)   (4)
is variable x: word                (5)
    loop                  -- infinite iteration
          I -> x          -- Input Communication
          ;               -- Sequential operator
          O <- x          -- Output Communication
    end
end

procedure buffer2(input i: word; output o: word)
is channel c:word
begin
     buffer(i, c) ||      -- "||" parallel operator
     buffer(c, o)
end

(1) "import" statement includes the files with type
    declarations or library procedures
(2) "type" statement is the definition of user
    defined data type.
(3) Inside "procedure" is the definition of a
    component (a piece of circuits) which is either
    the whole system or only a small part called by
    other procedures
(4) "input/output" ports define external channels
(5) Definition of local registers and internal
    channels
```

Figure 3. Two-place buffer

breeze2lard tool helps to translate the .breeze file to a LARD simulation model. The native simulators of the supported commercial CAD tools carry out the other two low levels simulation (figure 2). A research project is currently incorporating into Balsa formal verification and distributed simulation capabilities [20].

Figure 3 illustrates a simple Balsa modelling example of a two-place buffer.

4 SAMIPS Architecture

For our purposes, we are using the base MIPS application architecture as described in [21] and as exemplified by the R3000 processor.

The synchronous 32-bit microprocessor MIPS R3000 consists of two tightly-coupled processors, namely a full 32-bit RISC CPU and a system control co-processor (CP0), which deals with exception handling, memory management and memory address translation. The processor core (CPU) includes 32 32-bit general-purpose registers, two 32-bit registers for multiplication and division results, one program counter (PC), and a control logic unit. The datapath includes an ALU, a Shifter, a Multiplier/Divider, an Address Adder, and a PC incrementer.

The MIPS datapath is built around a five-stage pipeline consisting of: Instruction Fetch (IF), Decode/Register File Read (ID), Execution or Address Calculation (EX), Memory Access (MEM) and Register Write-back (WB). It is a Harvard architecture utilizing two memory ports: one for instruction fetch and one for data access.

SAMIPS, the asynchronous design of MIPS, adheres to the five-stage pipeline datapath, as depicted in figure 4 [36]. As dictated by the asynchronous design style, the required control signals are generated in ID stage during instruction decoding and then bundled together with the data and passed through the whole pipeline.

Asynchronous pipeline architectures introduce much more complicated data and control hazard problems which calls for innovative asynchronous solutions.

A data hazard will occur when an instruction depends on the result of a previous instruction still in pipeline. For SAMIPS, an innovative asynchronous forwarding mechanism [36] has been developed, which is based on history information. Since the sequence of instructions after they enter into pipeline is preserved, a data hazard can be detected when performing a register read at ID stage, if some history information is recorded inside the *RegBank* (*DHdetection* in figure 4).

Control hazards arise when an instruction such as a branch or a jump, or the occurrence of an unpredictable event such as an exception, changes the flow of control. In a pipeline architecture, if a control hazard occurs, the prefetched instructions following a hazard must be removed from the pipeline before instructions from the new stream (e.g. the branch target address or the exception vector address) are executed. SAMIPS extended the “colouring” mechanism first used in AMULET1 [33]. Instructions as well as the state of the processor at any particular moment are “coloured”. An instruction is executed only if its colour matches that of the processor. The colour of the processor changes every time a control hazard occurs, causing the rejection of subsequent instructions until the new instruction stream arrives.

5 Modelling SAMIPS in Balsa

BALSAMIPS is the Balsa model of SAMIPS. It is structured as a hierarchy of concurrent Balsa processes and consists of approx. 900 lines of Balsa code. It (meta-)executes MIPS machine language instructions produced by a standard MIPS cross-compiler.

Figure 5 shows the top level process structure graph of BALSAMIPS and the Balsa description. It consists of five concurrent Balsa processes, one for each of the pipeline stages.

Some of these processes are themselves structured as a network of lower-level concurrent Balsa processes, while others may not have any sub-processes due to their simplicity (e.g. MEM and WB). The following sections describe each of these five processes.

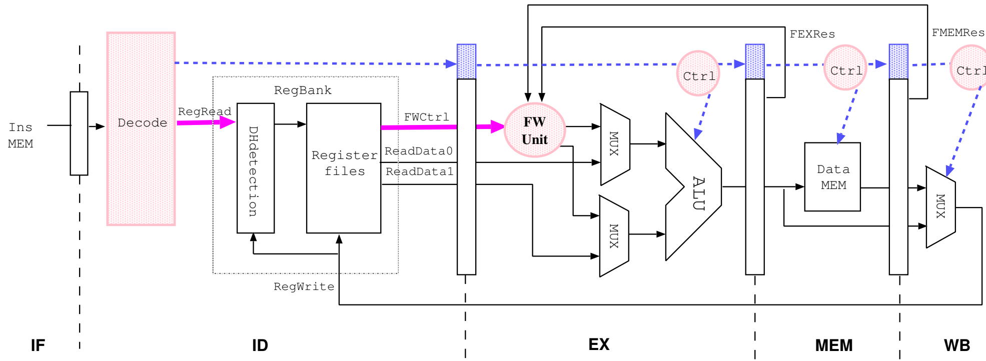
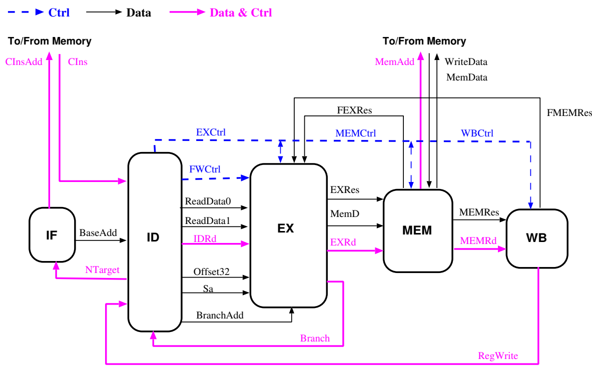


Figure 4. SAMIPS Pipeline Structure




```
import [IF]  -- include predefined processes
....
--define input/output channels to memory
--Instruction Address carries "colour" bit
procedure CPU (output CInsAdd : InsAdd;
               ...
               input CIns : CIns) is
-- define the local channel
   channel Branch,JumAdd,BaseAdd1: InsAdd
   ...
   channel FWCtrl:FWCtrl
begin
   IF (NTarget, JumAdd, BaseAdd, CInsAdd) ||
   ID (BaseAdd, ..., FWCtrl) ||
   EX (..., FWCtrl,...) ||
   MEM (...) ||
   WB (...)
end
```

Figure 5. BALSAMIPS Top Level Process

5.1 IF stage

IF is the first stage of the pipeline. This is essentially the instruction-prefetching unit of the processor. In this stage, the physical address, which is either the current program counter plus 4 or a new target address from the datapath if a control hazard occurs, is calculated and then sent to the main memory (through an arbiter). Figure 6 shows how the physical address for the next instruction is generated and sent back to the *PC*. A notable feature of the IF stage is that all the internal/external channels except the *BaseAdd* (the current PC value used as a base address for the new branch/jump target calculation) carry "colour" information to implement the "colouring" algorithm mentioned in the previous section. The modelling of the arbitration unit is also illustrated in figure 6, where the "arbitrate" Balsa structure is used to model nondeterministic choice.

5.2 ID stage

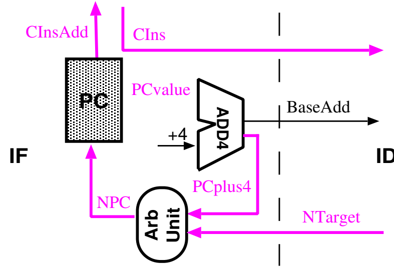
The main instruction decoder *DeCode* of the system resides in this stage. *DeCode* checks the first 6 bits of the instruction (*Opcode*) and the last 5 bits (*funct*, if the instruction is R-type). Based on that, operands or operand addresses are extracted and control information is also generated. Register file reading and writing is carried out in this stage as well. One of the innovative features of the *RegBank* is the implementation of an asynchronous forwarding mechanism mentioned in section 4. New target addresses for Branch or Jump addresses are calculated in *JumpAddALU*.

5.2.1 DeCode

The *DeCode* unit has one input channel and four output channels. The information carried by the *sa* (5-bit shift amount for shift operations) and *offset32* (signextended 32-bit from 16-bit immediate operand) channels are extracted from instructions directly. The rest three are all combined channels generated by *DeCode*:

- *EXCtrl*. Since all the control information is generated by DeCode and then propagated down the pipeline (figure 7), *EXCtrl* carries all the control information required by the next three stages, namely *EX*, *M* and *WB* and the "colour" of the current instruction. It is then left to local circuits of each following stage to pick up the relative control information.

- *RegRead*. This is the register address channel. In synchronous MIPS, only register file reading addresses are




```
arbitrate PCplus4 then
   if Arb_Col = PCplus4.colour then
      NPC <- PCplus4 end
| Ntarget then
   if Arb_Col = Ntarget.colour then
      Arb_Col:= not Arb_Col;
      --new "colour" is attached
      NPC <- {Arb_Col, Ntarget.address}
   end
end
```

Figure 6. IF Stage and ArbUnit Balsa Model

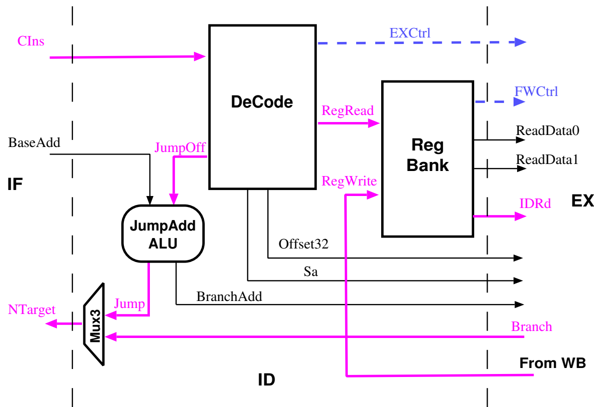


Figure 7. ID Stage

sent to *RegBank* in the ID stage. The register write address is passed through the entire pipeline and eventually sent back to *RegBank* in the WB stage. To implement our proposed asynchronous forwarding mechanism, the write address is also required in the ID stage, to enable *RegBank* to keep a record of all the instructions passing through which are going to perform a register write later.

- *JumpOffset*. This channel is used to send the extracted Branch/Jump offset to *JumpAddALU* for the new target address calculation. To distinguish between different types of Branch and Jump, a 2-bit control field is attached.

RegNo	*Flag*	
	Clean	*Index*
0	1	/
1	0	2
...	...	...
31	/	/

Figure 8. Data Hazard Detection Table

```
if (Flag[RegRead.RegRNo0].clean = true) then
  ReadData0 <- R[RegRead.RegRNo0]
else
  if (Flag[RegRead.RegRNo0].Index+1 as 2 bits)
    = CurIndex then
         FWcase0 := FEXR
  else   if (Flag[RegRead.RegRNo0].Index+2
         as 2 bits)= CurIndex then
               FWcase0 := FMEMR
         else  FWcase0 := WBR   end
  end
end ||
......;
if (FWcase0 = WBR or FWcase1 = WBR) then
  RegWrite -> RegWrite_R;
  if RegWrite_R.WriteCtrl=w then
     R[RegWrite_R.Rd.RegNo]:=RegWrite_R.RegData
  end ||
  Flag[RegWrite_R.Rd.RegNo].clean:= true ||
  if FWcase0 = WBR then
     ReadData0 <- RegWrite_R.RegData
  end ||
  if FWcase1 = WBR then
     ReadData1 <- RegWrite_R.RegData
  end
end;
.....
```

Figure 9. The Balsa Model of RegBank

5.2.2 RegBank

The Register Bank contains thirty two 32-bit general purpose registers and has two input channels (*RegRead* and *RegWrite*) and two output channels (*ReadData0* and *ReadData1*) for register read and write respectively.

It also has a module to implement the asynchronous forwarding mechanism (*DHdetection*), which, in the BALSAMIPS model utilises a table (DHDT, Data Hazard Detection Table), as illustrated in figure 8. DHDT contains three bits of information for each register referred to as the register *Flag*. *Flag* comprises one bit (*Clean*) indicating whether the register is pending to be written and two bits (*Index*) essentially indicating which instruction will rewrite the register (or, in other words, which stage will forward the result). At most, four instructions can be at any time in the pipeline from ID to WB, and therefore two bits are enough for the Index field. The algorithm makes use of another two bits to point to the current instruction in the register bank, called *CurIndex*. *CurIndex* is incremented by one (modular 4) every time a new instruction enters the register bank.

The Balsa model of the *DHDetection* module is illustrated in figure 9. Three 2-bit internal registers (Balsa vari-

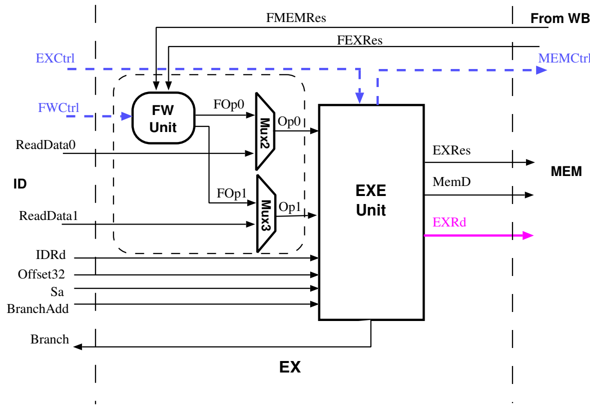


Figure 10. EX Stage

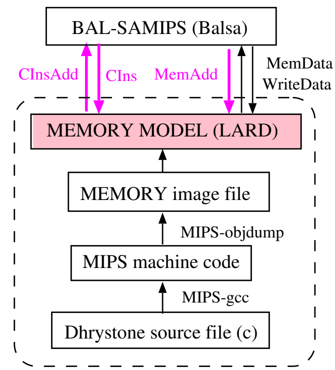


Figure 11. BALSAMIPS Simulation Environment

ables), namely *FRAQue* (Forwarded Result Acknowledgement Queue), *Fwcase0* and *Fwcase1* are used. Whenever a new *RegRead* signal is received, the corresponding entry of DHDT is checked and compared with *CurIndex* before doing register file reading. The Result Forwarding Algorithm is then carried out. When the register file read is complete, *FRAQue* and *FWCase0/1* are bundled together on *FWCtrl* channel and are sent to *FWunit* in the EX stage; DHDT and *FEAQue* also need to be updated if the current instruction is going to modify the register file.

When a new *RegWrite* signal is received, the new value is written to the corresponding *RegData[i]* and *Flag.Clean[i]* is reset. A problem may occur if two consecutive instructions modify the same register. In this case, the first *RegWrite* signal will mark the register 'clean', which is of course wrong since the register is still to be modified by the next instruction. This problem can be solved if *RegWrite* carries with it a 2-bit snapshot of the value of the *CurIndex* (attached in *IDRd* initially) as it was when the instruction passed through *RegBank*. This value is then compared with *Flag.Index[i]* inside DHDT. The mismatch means that there is at least one another instruction waiting to rewrite this register in the pipeline.

5.3 EX stage

This is where the instruction is executed. Inside *EXEunit* reside the ALU, a shifter and a functional unit for Branch decision-making. The multiplication and division operations of MIPS have not been implemented yet in the SAMIPS. *EXEunit* is also responsible for checking the "colour" and discard invalid instructions. The EX stage also hosts the *FWunit* module which implements the remaining part of the asynchronous forwarding mechanism.

5.3.1 FWunit

FWunit is the final destination of the forwarded results (*FEXRes* and *FMEMRes*). Inside *FWunit* these two forwarded results are acknowledged and sent to *Mux1/Mux2* if necessary, based on the content of *FWCtrl*.

5.4 MEM stage

MIPS is a Harvard architecture with two ports for the Instruction and Data Memory respectively. The MEM stage interacts with the Data Memory via two main external channels: *MemAdd* (data address) and *MemData* (the data). The result from EX stage is forwarded back to the EX stage via *FEXRes*. *ExRd* is acknowledged and stored in a local register and it is then passed to the next stage on channel *MemRd*.

5.5 WB stage

WB is the last stage of the MIPS pipeline. The function of this stage is very simple compared to the previous four: it generates the *RegWrite* signal, if the current instruction requires a write operation to *RegBank*, and forwards the result from MEM back to EX on channel *FMEMRes*.

6 Simulation Environment

The environment used to simulate SAMIPS is illustrated in figure 11. We are currently utilising LARD to do behavioural simulation of SAMIPS. The memory model used is written in LARD. The memory is modelled as an integer array and is assumed to be 2GB, the first 256B of which is used to store data. The contents of the memory are loaded during initialization as 32-bit quantities in the hexadecimal format from an image file. Whenever an address signal arrives, either from the Instruction Memory port (*CInsAdd*, *CIns*) or the Data Memory port (*MemAdd*, *MemData*, *WriteData*), the model outputs the result with a delay of 100 time units.

LARD simulation is able to detect deadlocks by terminating the simulator when a deadlock occurs and utilising a

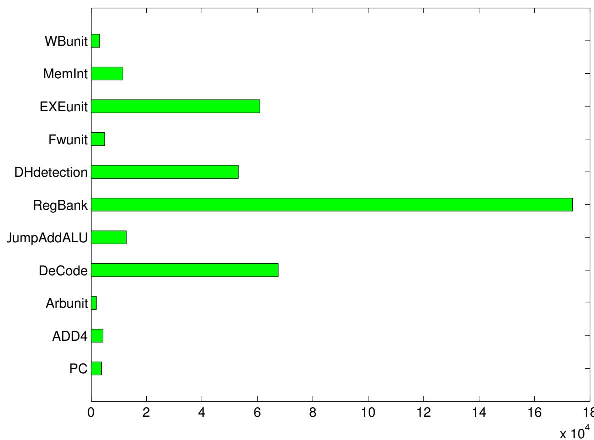


Figure 12. Cost Estimation

Channel	AVERAGE	MAX	MIN
FEXRes	1.91	20	1
FMEMRes	283.21	415	227
RegRead	235.07	353	141
RegWrite	252.32	383	195
Branch	6	6	6
Jump	/	/	/

Figure 13. Channel Activity

channel activity viewer for post-mortem simulation analysis.

7 Results

At the time of writing, a Balsa model of the current SAMIPS has been developed and is currently being validated.

Figure 12 illustrates the area costs obtained from the Balsa breeze-cost utility. The costs provided by breeze-cost are only guideline figures and depend on particular back-end implementation. The units presented in the figure are linear microns of standard cells based on an old 1um library with a cell pitch of 37.5um and a typical density of about 2/3 cells, 1/3 routing.

We have successfully executed Dhrystone (version 2.1) and we are conducting a preliminary performance evaluation of SAMIPS. Figure 13 illustrates the results of a preliminary timing analysis of activity on six chosen main channels when executing Dhrystone (1 loop). The figures show the average, min and max times (LARD time units) that a channel was active during the simulation, namely the period between the channel sending its request and getting the acknowledgement back.

8 Summary and Future Work

This paper has presented the Balsa model of SAMIPS, an asynchronous design of MIPS under development at the University of Birmingham. The process structure and the functionality of each of the Balsa processes has been described and some preliminary simulation results have been presented. Future work will complete the SAMIPS design implementing exceptions and interrupts; perform a detailed evaluation of the processor through simulation; and finally synthesise the processor. Parallel work is developing a distributed simulation kernel for Balsa and is incorporating formal verification capabilities.

Acknowledgements

We would like to thank the members of the AMULET group, and in particular Andrew Bardsley and Doug Edwards for their invaluable advice and help. Also Nick Shrine for his help with the MIPS cross-compiler and Wang Xu for his comments on early drafts of this paper. The research is funded by EPSRC and the School of Computer Science, University of Birmingham (ORS)[20].

References

[1] The Balsa Asynchronous Synthesis System, `http://www.cs.man.ac.uk/apt/projects/Balsa/index.html`

[2] Beister, J., Eckstein, G., Wollowski, R., “CASCADE: A Tool Kernel Supporting a Comprehensive Design Method for Asynchronous Controllers”, Proceedings of ICATPN 2000, Aarhus, DK

[3] Berkel, K. van, “Handshake circuits - an Asynchronous Architecture for VLSI Programming”, Cambridge International Series on Parallel Computers, Cambridge University Press, Cambridge, 1993

[4] Brunvand, E., Starkey, M., “An Integrated Environment for the Design and Simulation of Self Timed Systems”, Proceedings of VLSI’91, 1991, pp. 4a.2.1-4a.3.1.

[5] Chu, T.A., “Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications”, Ph.D. thesis, Massachusetts Institute of Technology, MA, US, June 1987

[6] Cogency Technology Inc., `http://www.cogency.com`

[7] Cortadella, J., Kishinevsky, M., C.E., “Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers”, IEICE Transactions on Informations and Systems, pp. 315-325, 1997

[8] Davis, A., Nowick, S. M., “Synthesizing Asynchronous Circuits: Practice and Experience”, *Asynchronous Digital Circuit Design*, 1995, pp. 104-150.

[9] Ebergen, J. C., “A Formal Approach to Designing Delay-Insensitive Circuits”, *Distributed Computing*, 5 (3), 1991, pp. 107-119.

[10] Endecott, P.B., Furber, S.B., “Modelling and Simulation of Asynchronous Systems using the LARD Hardware Description Language”, Proceedings of the 12th European Simulation Multiconference, Manchester, 1998, Society for Computer Simulation International, pp. 39-43.

[11] Fuhrer, R. M., Jha, N. K., C.E., “MINIMALIST: An Environment for the Synthesis, Verification and Testability of Burst-Mode Asynchronous Machines”, Tech. Report CUCS-020-99, Department of Computer Science, Columbia University, NY, US, July 1999

[12] Gopalakrishnan, G., Akella, V., “Specification, Simulation, and Synthesis of Self-Timed Circuits”, Proceedings of the 26th Hawaii International Conference on System Sciences, 1993, pp. 399-408.

[13] Hassoun, S., Marculescu, D., “Towards GALS Design Methodologies”, Proceedsings of FMGALS’2003, Italy, Sep. 2003

[14] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice Hall International, 1985.

[15] Hulgaard, H., Burns, S. M., “Bounded Delay Timing Analysis of a Class of CSP Programs with Choice”, Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems, 1994.

[16] Josephs, M. B., Udding, J. T., “Delay-Insensitive Circuits: An Algebraic Approach to their Design”, LTNC, Vol. 458, 1990, pp. 342-366.

[17] Liu, Y., Aldwinckle, J., C.E., “Designing Parallel Specifications in CCS”, Proceedings of the Canadian Conference on Electrical and Computer Engineering, Vancouver, 1993.

[18] Martin, A.J., “Synthesis of Asynchronous VLSI Circuits”, J.Staunstrup, editor, Formal Methods for VLSI Design (North Holland, 1990).

[19] C. E. Molnar, T-P. Fang, “Synthesis of Reliable Speed-Independent Circuit Modules: I. General Method for Specification of Module-Environment Interaction and Derivation of a Circuit Realisation, Tech. Report 297, Computer Systems Laboratory, Institute for Biomedical Computing, Washington University, St. Louis, 1983.

[20] An Integrated Framework for Distributed Simulation and Formal Verification of Asynchronous Hardware, EPSRC Project No. GR/S1109/01 & GRS11084/01 http://www.cs.bham.ac.uk/research/parlard/

[21] Patterson, D.A., Hennessy, J.L., *Computer Organization & Design*, second edition, Morgan Kaufmam, 1997

[22] Rosenblum, L.Y., Yakovlev, A.V., “Signal graphs: from self-timed to timed ones” Proceedings of International Workshop on Timed Petri Nets, Torino, Italy, pp. 199-207, July 1985

[23] Sentovich, E. M., Singh, K. J., C.E., “SIS: A System for Sequential Circuit Synthesis”, Tech. Report UCB/ERL. M92/41(May 1992), Department of Electrical Engineering and Computer Science, University of California, Berkely, CA, USA

[24] Spars, J., Furber, S., *Principles of Asynchronous Circuit Design - A Systems Perspective* Kluwer Academic Publishers, Hardcover ISBN 0-7923-7613-7, 2001

[25] Theodoropoulos G., Woods, J. V., Occam: An Asynchronous Hardware Description Language?”, Proceedings of the 23rd IEEE Euromicro Conference on New Frontiers of Information Technology, Budapest, Hungary, September 1997.

[26] Theodoropoulos, G., “Modelling and Distributed Simulation of Asynchronous Hardware”, Simulation Practice and Theory Journal, Elsevier. Volume 7, Issue 8 , 15 March 2000, Pages 741-767

[27] Theodoropoulos, G., “Simulating Asynchronous Hardware on Multiprocessor Platforms”, Concurrency-Practice and Experience Journal, John Wiley and Sons Ltd Publishers. Volume 13, Issue 10, Date: 25 August 2001, Pages: 869-904

[28] Theodoropoulos, G. “Distributed Simulation of Asynchronous Hardware: The Program Driven Synchronisation Protocol”, Journal of Parallel and Distributed Computing, Special Issue on ”Parallel and Distributed Discrete Event Simulation–An Emerging Technology”, Academic Press. Volume 62, Issue 4, April 2002, Pages 622-655

[29] Theodoropoulos, G., “Modelling an Asynchronous Microprocessor”, Transactions of the Society for Computer Simulation International. To appear.

[30] Theodoropoulos, G., “Modelling and Simulation of Computer Systems”, Editorial, International Journal of Simulation Systems, Science & Technology Special Issue on: Modelling and Simulation of Computer Systems, Volume 4, Numbers 3-4, September 2003, pp. 1-3

[31] Van Berkel, C. H., Kessels, J., C.E., “The VLSI-Programming Language Tangram and its Translation into Handshake Circuits”, Proceedings of EDAC, 1991, pp. 384-389.

[32] VERSIFY Release 2.0, http://www.ac.upc.es/vlsi/versify/

[33] Woods, J.V., Day,P., C. E., “AMULET1: An Asynchronous ARM Microprocessor”, IEEE Transactions on Computers 46 (4)(1997) pp.385-398.

[34] Ykman-Couvreur, C., Lin, B., De Man, H., “ASSASSIN: a synthesis system for asynchronous control circuits”, Tech. report IMEC Laboratory, Sep. 1994.

[35] Yun, K.Y., Dill, D. L., Nowick, S.M., “Synthesis of 3D Asynchronous State Machines”, Proceedings of ICCD’92: VLSI in Computers and Processors, Cambridge, MA, US, pp.346-350, October 1992

[36] Zhang, Q., Theodoropoulos, G., “Towards an Asynchronous MIPS R3000 Processor”, Proceedings of ACSAC’2003, Janpan, Sep. 2003

Author Biographies

QINAYI ZHANG is a PhD student studying in the School of Computer Science at the University of Birmingham, UK. She received a BSc in Computer Science and Technology from the Hohai University, Nanjing, China, in 2001 and an Advanved MSc in Computer Science from the School of Computer Science at the University of Birmingham, in 2002. Her research focuses on the design and synthesis of asynchronous microprocessors.

GEORGIOS THEODOROPOULOS holds a Diploma in Computer Engineering from the University of Patras, Greece, and an MSc and a PhD in Computer Science from the University of Manchester, U.K. He is currently a Lecturer in the School of Computer Science, University of Birmingham, U.K. where he has established and leads the “Systems, Models and Simulation” (SysMoS) research group. His research is mainly in parallel systems, computer architectures and networks, Grid and cluster computing and modelling and distributed simulation. Dr Theodoropoulos is a co-founder and member of the Management Board of MeSC, the Midlands e-Science Center of Excellence in Modelling and Analysis of Large Complex Systems. http://www.cs.bham.ac.uk/~gkt.