The Design of a Fast Asynchronous Microprocessor

Sam S. Appleton, Shannon V. Morton, Michael J. Liebelt †

Abstract

The recent explosion of interest in asynchronous design approaches has seen a number of very complex systems, such as microprocessors, implemented in high-performance technologies. In this paper, we describe the architecture and performance of an asynchronous microprocessor implemented using a two-phase communication strategy. The processor incorporates some features never seen in asynchronous microprocessors previously, such as caches *and* external synchronism*.*

I Introduction

Asynchronous design methodologies are experiencing a period of sustained interest and intensive research, primarily spurred by the view that the problems associated with synchronous systems design are becoming too complex. Rapid technological developments in the VLSI field are resulting in chips of growing area and reduced feature size. These developments make possible the design of more complex systems that operate at higher speeds, a by-product of which is higher power dissipation. These technology trends make the design task of distributing the global clock, which controls a synchronous system, an arduous task. In addition, the system designer must ensure that each block meets the timing constraints of the global clock in order for the design to function. Clock skewing, clock drivers, signal routing, and timing verification are becoming overriding concerns of the synchronous circuit designer.

The asynchronous designer mitigates these problems by removing the global clock, and replacing it with a local communication protocol between the blocks of the system, using some form of request-acknowledge protocol. This can result in a number of benefits. In general, the overall power dissipation of a chip is significantly reduced, as there is no continous transitioning of signals in parts of the system which are not performing computations. Each component of the system may operate at a speed independent every other component, allowing for the possiblity of average-case rather than worst-case operation, as well as allowing the performance of the system to be improved *incrementally*, rather than in a *step-wise* manner. Furthermore, each component may also operate in a *data-dependent* manner, completing its operation in a time heavily dependent on the input data. However, asynchronous approaches are not lacking in their own problems. Perhaps the two greatest perceived problems are the issues related to design and performance. The design of asynchronous systems is commonly percieved as extremely difficult, not to mention hazard-prone. Much work has been done to address this concern [1, 2, 3, 4], although asynchonous systems have some way to go before they are truly easy to design. The performance issue is a critical one. As this class of systems uses *local communication* in place of a global synchronising signal, logic must be used locally to implement this local signalling, resulting in some *overhead* for communication in each stage (even though the area and power consumed are usually minimal in relation to the datapath). Various techniques for minimisation of this overhead have been pursued, but it seems likely that some penalty will always exist for using a local communication protocol.

The positive factors are currently pushing the transfer of asynchronous techniques into the design mainstream, where limited classes of asynchronous techniques are beginning to be used in commercial products [5, 6] . However, the design and performance issues of asynchronous systems, as well as the critical issue of testability, are hindering wider acceptance of these techniques.

It is in this context that the Event Controlled Systems (ECS) methodology, a specification system we employed for the design of control and data paths, was developed and applied to the design of a prototype microprocessor, *ECSTAC*. This paper discusses the significant and novel aspects of ECS and the microprocessor ECSTAC, along with some expected performance figures based on simulation results.

Event Controlled Systems (ECS) is based on the 'two-phase' or *transition signalling* approach to local communication. The ECS approach is geared towards the practical implementation of high-performance VLSI systems – it allows for simple expression and design of control and datapath structures within the one framework. The development of large-scale systems, such as microprocessors, was thus seen as critical in estabilishing ECS as a viable and usable methodology.

†Authors are with the Dept. of Electrical & Electronic Engineering, University of Adelaide, Australia. E-mail sam@eleceng.adelaide.edu.au.

II Overview of the ECS approach

The ECS design style is characterized by two main criteria - control systems generally operate under fundamental mode (FM) as opposed to speed-independence (SI) or delay-insensitivity[7], and the representation of these systems is given in a new *temporal* domain. Fundamental Mode circuits assume that the delays embodied in gates and wires can be *bounded*, and thus assumptions about the timing characteristics of a particular network can be known. Speed Independent circuits assume that the delays embodied in gates cannot be bounded, and delays in wires are negligible compared to gate delays. ECS operates in FM because it generally results in much faster and smaller circuits than can be achieved in SI [8, 10] by slightly relaxing the I/O requirements. The *temporal domain* allows two-phase asynchronous gates to have a representation similar to boolean algebra, resulting in a simple and intuitive framework for design.

ECS splits the *signals* of a system into two *absolutely distinct* types – *event lines* are control signals using a transition-signalling approach, and *data lines* are normal boolean signals, whether they be used for control signalling or data transfer. Elements are constructed from event controlled units (ECUs) which receive an input trigger event and (generally) some associated data, and after processing produce the corresponding outputs. This demands that all associated input data is valid before the input event occurs (which triggers the operation of the ECU on the input data). This requirement is embodied in *Association Principle* of ECS.

The *temporal transformation*, which transforms signals in the *voltage* to the *temporal* domain, is a unique concept. The transformation for data signals is one to one.

$$\begin{array}{rcl} \triangle_v in & \Leftrightarrow & \triangle_t in \\ \bigtriangledown_v in & \Leftrightarrow & \bigtriangledown_t in \end{array}$$

where $\triangle$ and $\bigtriangledown$ respectively represent positive and negative transitions of the boolean signal, the subscripts v and t represent those transitions in either the voltage or temporal domain, and $\Leftrightarrow$ merely represents equivalence of the two statements through the transformation. However, for *event lines*, the signal becomes *true* (or *one*) in the temporal domain whenever a transition occurs in the *voltage* domain, i.e.

$$\triangle_v \partial in + \bigtriangledown_v \partial in \quad \Leftrightarrow \quad \triangle_t \partial in$$

Note that event lines are prefixed with the symbol ∂ to distinguish them from data lines.

This transformation is applied as the actual *direction* of the transition is unimportant for transition-signalling systems - we only need know that a transition *occured*. This then frees up the negative transition ($true \rightarrow false$) of the event line in the temporal domain for redefinition. In ECS, we say that no further input events to an ECU may occur until the *corresponding* output event (which always exists) has occured[1] – this is embodied in the *Exclusion Principle* of ECS. For example,

$$\bigtriangledown_t \partial in > \triangle_t \partial out$$

where '$>$' is a timing operator called *after*, indicating the truth of the effect given the truth of the cause (i.e. these equations are of the form *effect* $>$ *cause*, indicating that effect follows cause in time) [11]. In the above example, the input event ∂in is set *false* in the temporal domain only when the output event, ∂out, has become true.

An event controlled system (ECS) is defined as being the top level of hierarchy of a combination of ECUs. These ECUs are specified *completely* by a set of timing equations (TEs), like those shown above, which describe the event and data flow in the component. A TE essentially specifies that "Condition y occurs after condition(s) x", and is comprised of a number of operators in addition to the *after* operator described above[13].

Fig. 1 shows a simple ECS gate, termed a *last* gate (otherwise known as a Muller-C or simply C-element). A transition on either input (∂ina or ∂inb) in the voltage domain creates an upwards ($\triangle$) transition in the temporal domain. The TE defines ∂out *occuring* (i.e. becoming true) when both inputs are true in the temporal domain (the '.' symbol is the simple boolean operator AND). An upwards transition on ∂out sets the inputs *false* in the temporal domain (by the above definition), and as a consequence the output event is only true *instantaneously* in the temporal domain. To return from the temporal domain to the voltage domain, we simply apply the temporal transformation to get ∂out. Intuitively, this gate is an AND function for transition systems, which is reflected in the notation. All ECS gates have this simple & intutive notation, making it simpler to understand the function of a given system.

III Overview of ECSTAC Architecture

Testing the utility of the ECS approach involves making some large system and ensuring that it operates correctly, such as a prototype microprocessor. The processor prototype has a simple architecture modelled on RISC systems. ECSTAC has an 8-bit datapath and a 24-bit address path – a larger address bus was used to ensure that a full array of test programs

[1] We make these principles *design requirements* for using ECS, and it is not difficult to adhere to them[13].

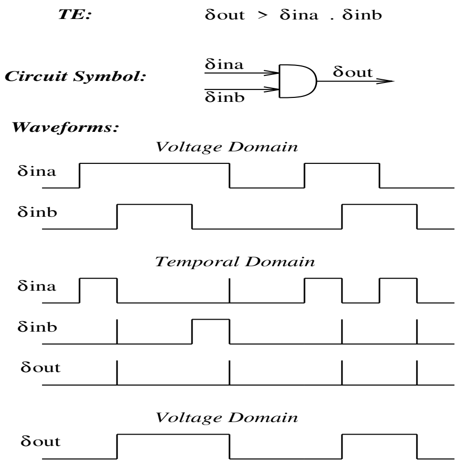

Figure 1: The *Last* gate in ECS, symbol & example waveforms in the *voltage* and *temporal* domains

could be loaded into the memory space of the processor, although the processor does not implement virtual memory (as it is just a prototype). A block diagram of the functional blocks that make up ECSTAC is shown in Figure 2 – the details of some of these blocks will be discussed in this paper.

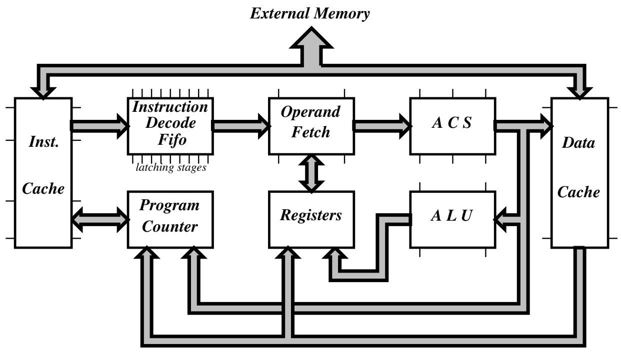

Figure 2: Architecture of the ECSTAC microprocessor

The instruction set architecture of ECSTAC is a simple RISC-style set. There are 46 instruction types, many of which have more than one mode. Instructions are grouped into one of four different *groups* of instructions – memory access (LOAD and STORE), computation, branch, and a miscellaneous group including program control, stack operations and test commands. The details of the instruction set and addressing modes will not be detailed here[9], but it should be mentioned that the length of instructions is *variable* to accomodate all the required instructions and modes within an 8-bit datapath. Note that ECSTAC does not implement exceptions or interrupts – these aspects were considered too complex to manage in a prototype such as ECSTAC, although they will be addressed in the next version of the processor.

ECSTAC uses a split-cache (Harvard) architecture to minimise the arbitration frequency for cache requests[9, 15]. The Instruction Cache (ICache) continuosly fetches bytes (for the eight bit datapath) from the PC address and places the data into the Instruction Decode (ID) stage. This stage is heavily pipelined, and essentially acts as an asynchronous *fifo*, performing instruction decode and instruction buffering in the one unit. The output of the ID stage is fed into the *O*perand *F*etch stage (OF) which controls the number of register fetches required (the OF stage only *fetches* operands) and the orginisation of the accessed fields, which are placed onto a minimal set of output buses.

The ACS stage (*A*dder, *C*omparator, & *S*tack Pointer stage) performs address offset calculations (which are 24 bits long - this would be too awkward to do in the ALU's 8-bit adder). The additions are done in two stages comprising 12 bit Manchester Carry Adders in each. The ACS stage interacts with the ALU, the Data Cache (which handles all LD/ST requests and stack operations for the pipeline), and also writes the Program Counter (PC) on a taken branch.

The ALU is a single pipeline stage, incorporating a dedicated shifter, logical unit, and 8-bit MCA. The completion time of the ALU depends on the operation requested – shifts and logical operations are much faster than adds or subtracts in the ALU.

The register bank is a conventional two read and one write port block with 16 registers, incorporating additional gates to do *register scoreboarding*, the chosen method for handling RAW data hazards[14]. This will be discussed further in Section V.

IV ECSTAC Memory Structure and External Interface

As ECSTAC is pipelined, there are several places in the pipeline where access to the external memory system will be required, shown in Fig. 3.

The ICache provides instruction data for the pipeline, and will need to fill from external memory when it *misses*. The Data Cache services data requests coming from the ACS stage, and it too will require access to external memory when it *misses*, or when STOREs are processed. The ICache and DCache are autonomous units, and requests for access to external memory may come at any time, leading to the well-known problem of *arbitration*[3, 5]. A block diagram of an arbiter in ECS is shown in Fig. 4. Requests and associated data arrive from the ICache or DCache at arbitrary times. These requests are fed into *input controllers* which sequence the signals applied to the *arbiter module* and latches in the system. The arbiter module decides which cache has access to the external memory

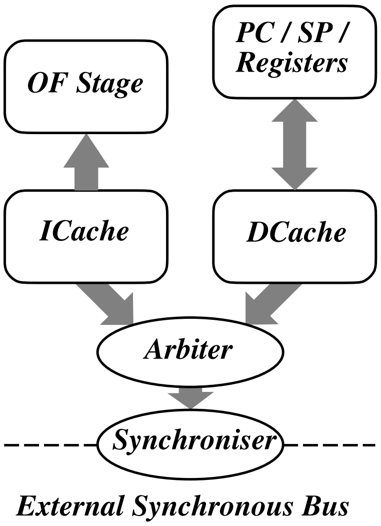

Figure 3: Memory System Overview

at any one time, and is implemented using a well-studied technique[3] that hides metastable operation. The output of the arbiter module is used to drive output control & data lines via a *Join & MUX Module*.

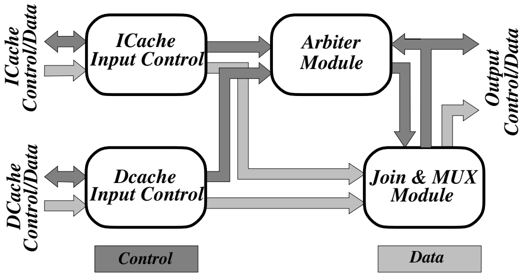

Figure 4: Arbiter Design Outline in ECSTAC

Memory system requests leaving the arbiter are still asynchronous in nature. However, since the external system is synchronous, the requests coming from the arbiter must be *synchronised* to the external clock signal, leading to another well-known problem, *metastability*[16, 17]. A representation of the synchroniser circuit used in ECSTAC is shown in Fig. 5. Memory Requests arriving at the synchroniser enter a control circuit, and are ultimately synchronised by a normal D-Latch (indicated by the '*!*' in Fig. 5). The input to the flip-flop, *Req*, is asynchronous with respect to *Clock*. Thus the output of the flip-flop, *Sync'ed Req*, can enter a metastable state for a potentially unbounded period. This is obviously a critical issue for designing high-reliability asynchronous systems, and much research is being done into reducing (and perhaps eliminating) synchronisation errors in ECS.

The ICache is a pipelined functional unit – HSPICE simulations of the blocks of the ICache indicate that a considerable performance benefit can be obtained by pipelining[9, 15]. A simple simulation of the pipeline, which modelled the asynchronous nature of the system, was used to determine the optimum parameters for the ICache and DCache. The same performance enhancement (pipelining) is not applied to the DCache. A simple diagram of the ICache pipeline is shown in Fig. 6.

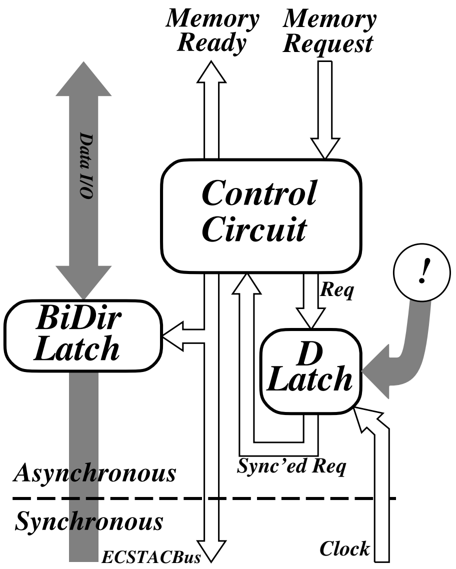

Figure 5: Asynchronous ↔ Synchronous Interface

A special *deadlock* condition, shown in Fig 7, must be avoided to ensure ECSTAC operates for all possible instruction streams. In this situation, deadlock has occured because the ICache cannot push new data into the second ICache stage, and the DCache cannot complete its operation because the memory interface is locked by the ICache[15]. This *unique* condition in the ECSTAC pipeline can simply be avoided by requiring the ICache second stage to be *free* before beginning miss processing.

V ECSTAC pipeline

There are several aspects of the ECSTAC pipeline that deserve special attention – flow control & fast-forwarding, handling of data hazards, and instruction ordering schemes.

V-A Flow Control

The OF stage only latches *decoded* information from the ID stage on the first byte of every instruction – any subsequent bytes provide register locations and function codes, and the decoded information from ID is ignored. Once all the data for a particular instruction has been received and the operands fetched, it is *issued* by sending it to the ACS stage.

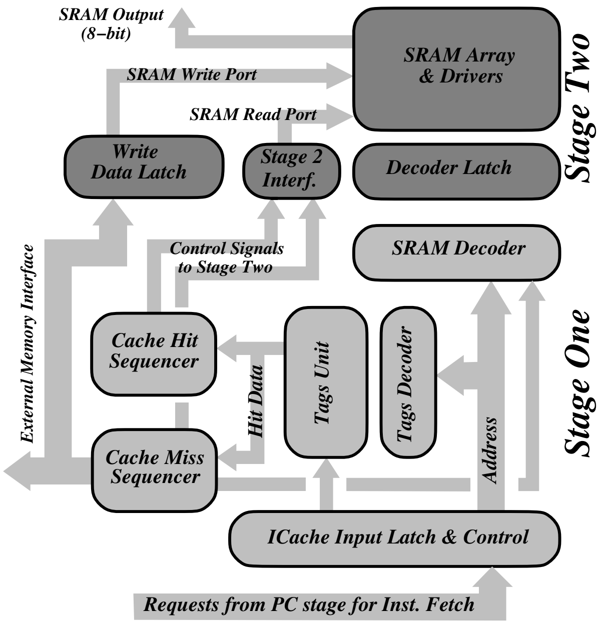

Figure 6: ICache Pipeline

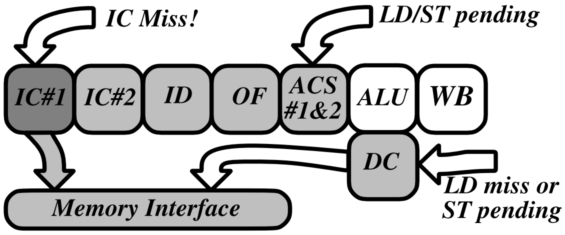

Figure 7: Deadlocked Pipeline Condition

The ACS stage is perhaps the most interesting pipeline stage of ECSTAC, since it controls the evaulation and PC modification for branches, the control of the stack pointer, and the generation of memory addresses for the DCache.

If the logic of the ACS stage is not required or the input instruction is a NoOp, the precharged logic of the two ACS stages is completely bypassed, and the output event for the ACS is sent. This results in significant power and time savings in the ACS, as it appears almost transparent to certain instructions. The second stage of the ACS can also be bypassed if the intermediate carry of the first stage is zero (or the ACS is not required), resulting in further savings. The *bypassing* structure of the ACS is shown in Fig. 8.

Note that the *Program Counter* value corresponding to the instruction issued from *OF* is passed to the ACS stage – the ACS stage requires the current PC value to determine the correct address for relative branches, and to save return addresses for CALL instructions, and thus every stage prior to the ACS must hold PC

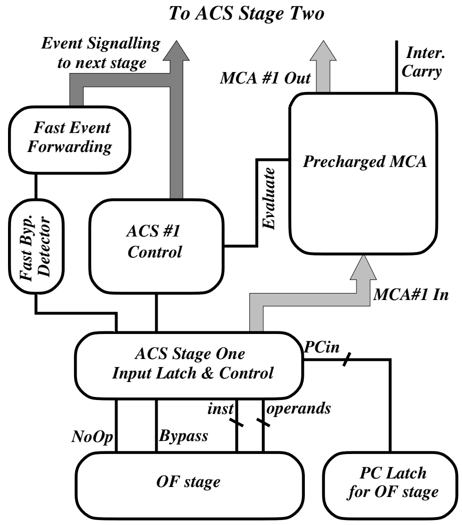

Figure 8: ACS bypassing mechanisms

values and pass them down the pipeline.

The first stage of the ACS determines whether a branch is followed based on data supplied by the OF stage. Untaken branches are simply converted to NoOps and passed down the pipeline. Taken branches cause the IFetch process to be halted instantly (some degree of arbitration exists here), and all instructions in the pipeline previous to the ACS are converted to NoOps, concurrently to the final determination of the new PC value. This new value is loaded into the PC when available and the IFetch process restarted – instructions fetched prior to branch evaluation quickly pass though the pipeline as NoOps.

V-B Register Tagging & Instruction Ordering

Data hazards in ECSTAC are handled using an approach based on the *scoreboarding* technique, which also allows out-of-order completion. The scheme used is shown in Fig. 9. Another approach was developed[9], but was not used since it did not allow *out-of-order* completion.

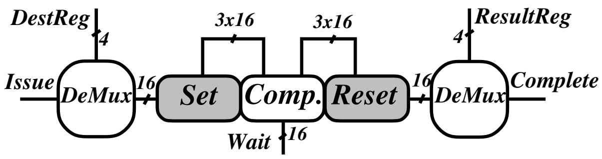

Figure 9: Register Tagging Scheme

For each register, there are two 3-bit counters with a comparator connected to their outputs. This results in the 16-bit bus *Wait* of the comparator outputs. When the corresponding *Wait* values are high, the source registers for the instruction in OF (waiting be issued) can be fetched, otherwise OF must wait until the *Wait* values go high.

When the instruction is ready to issue, *Issue* goes high, incrementing the value of the *Set* counter corresponding to the *DestReg* (Destination Register) of the current instruction. This will also result in the value of *Wait* for *DestReg* going low. When the result of this instruction arrives in the *WB* stage, *Complete* goes high, incrementing the *Reset* counter, setting *Wait* low. This scheme handles RAW (*R*ead-*A*fter-*W*rite) hazards as an instruction arriving in OF that has a RAW conflict with a previously-issued instruction will observe the relevant *Wait* signal being *low* until the result is available. Counters are required to deal with WAW (*W*rite-*A*fter-*W*rite) hazards – the OF stage will allow two or more instructions with the same *DestReg* to issue without delay. This causes the value of the *Set* counter to be incremented more than once, and thus *Wait* will only return to the *high* state when all instructions writing to the relevant *DestReg* have completed.

Both the ALU and the DCache require access to the WB stage to write their respective results. One solution to this access problem would be to maintain instruction ordering, since then only one of these units could access the WB stage at any given time. However, the completion time for a DCache operation may be considerable if the DCache misses, and maintaining instruction ordering would thus *stall* the pipeline until the DCache had finished. If *out-of-order* completion is allowed, then the pipeline can continue processing until it requires the result of the DCache operation (which will then result in a *stall*). A diagram of this resultant interface for this scheme is shown in Fig. 10. The special block WAW HD/S (Hazard-Detector/Staller) stalls the input signals to the offending block if a potential *erroneously reordered* write to an identical register is about to occur.

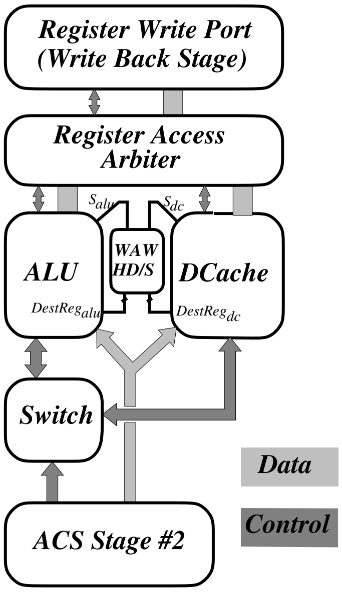

Figure 10: ALU/DCache/WB Interface

VI ECSTAC Test & Performance

ECSTAC makes some concessions to testability issues in its design. As the caches effectively insulate the processor pipeline from the external environment, the ISA includes instructions to *disable* the caches so that the effects of instructions on the internal state of the machine can be more closely observed. In addition, many of the computation blocks in ECSTAC have their timing characteristics modelled by a simple delay block. Synchronous systems permit direct manipulation of clock frequency for test, allowing the clock to be slowed or even stopped for observation of intermediate processing steps. The delay elements used in ECSTAC are powered by a seperate power bus so that their delay may be adjusted from its nominal value (up to a 70% increase in delay may be achieved using this technique). However, these simple measures are not a comprehensive strategy for dealing with testability in ECS, although test issues are being dealt with using other approaches[19]. It is unclear whether conventional synchronous techniques such as Scan Path and BIST are ideally suited to the asynchronous domain. Testability will be a major focus of future ECS research.

ECSTAC is being implemented in a 0.8μm DLM CMOS process, and the layout task is almost complete at the time of writing. The major functional blocks of ECSTAC have been simulated using HSPICE, and some statistical details are shown in Table 1. As can be seen, the integration level of ECSTAC is considerable.

VII Conclusion

We have presented the design of the prototype microprocessor, ECSTAC, and the asynchronous design methodology employed. ECS is a simple & intuitive way to develop two-phase asynchronous systems, and can be used as the stepping stone for more advanced techniques like automatic synthesis, testing, and verification. ECSTAC uses a simple RISC-style architecture, and incorporates *caches* and *external synchronism* - two features not seen before in asynchronous processors. Based on our simulation results to date, we expect ECSTAC to have a throughput of approximately 40 MIPS. The mismatch between data bus width and (variable) instruction width causes some complex design problems which impact on performance - a dramatically higher throughput will be achieved when a full 32-bit processor is implemented (which was not an option for the prototype development). ECS's orientation towards practical and low overhead implementations in VLSI (using FM assumptions) has led to many interesting design techniques which could not be fully explored here. ECSTAC is

Block	Size(mm)	Area (mm^2)	Transistors	Cycle Time$_{ave}$ (ns)
ID	0.67 $\times$ 0.85	0.57	4722	8
OF	0.46 $\times$ 0.63	0.29	2058	14
ACS	0.75 $\times$ 0.53	0.40	4518	17
ALU	0.40 $\times$ 0.31	0.12	1307	9
ICache	3.39 $\times$ 1.93	6	120K	$\sim$10 /stage
DCache	3.59 $\times$ 1.79	4	70K	$\sim$30 /stage
ECSTAC	4.20 $\times$ 4.00	17	$\sim$200K	?

Table 1: **ECSTAC Block Statistics**

expected to be completed and fabricated towards the end of 1995, after which detailed performance figures will be generated.

In the future, ECS will be extended to incorporate features such as verification, design automation, and an integrated testability strategy to enhance its utility. Concurrently and assisting in this development will be the implementation of the next generation ECS processor, a 32-bit machine with a more advanced architecture than ECSTAC. Testability issues are seen as critical in this development, which will be a major theme in extending the ECS methodology.

We would like to thank Dr. D.A. Pucknell for his pioneering contributions, Professor Wonchan Kim and Gyudong Kim of Seoul National University for helpful suggestions, and the financial support of the Australian Research Council and the Frank Perry Scholarship.

References

[1] Sutherland, I.E., "Micropipelines", Communications of the ACM, pp.720-738, June 1989.

[2] Brunvand, E.L., "Translating Concurrent Communicating Programs into Asynchronous Circuits", PhD thesis, Carnegie-Mellon University, September 1991.

[3] Martin, A.J., "Programming in VLSI : From Communicating Processes to Delay-Insensitive Circuits", Developments in Concurrency and Communication, Addison-Wesley, 1990.

[4] Chu, T.A., "Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications", PhD thesis, Dept. of EEE, MIT, June 1987.

[5] IEEE Design and Test of Computers, "Special Issues on Asynchronous Techniques", IEEE Computer Society, Fall 1994, Summer 1994, Spring 1995.

[6] G.Matsubara et. al., "30-ns 55-b Shared Radix-2 Division and Square Root Using a Self-Timed Circuit", Proceedings of the 12th Symposium on Computer Arithmetic, July 1995, Bath, England.

[7] Hauck, S., "Asynchronous Design Methodologies : An Overview", Technical Report 93-05-07, Dept. of Computer Science and Engineering, University of Washington, WA.

[8] Morton, S. V., "Independent investigations into FM & SI logic bypass techniques", Internal Report, The University of Adelaide, December 1994†.

[9] Morton, S.V, Appleton, S.S., & Liebelt, M.J., "ECSTAC : A Fast Asynchronous Microprocessor", 2nd Working Conference on Asynchronous Design Methodologies, SouthBank University, England, July 1995†.

[10] Morton, S. V., "Pipelining Principles ", *Internal report HPCA-ECS-93/1*, Dept. of E.E.E., The University of Adelaide, January 1993†.

[11] Morton, S. V., & Appleton, S. S., "Temporal Transition Graphs for Event Controlled Systems ", *Internal Report HPCA-ECS-93/12*, Dept. of E.E.E., The University of Adelaide, October 1993†.

[12] Morton, S. V., "Deadlock & Persistency Detection of Temporal Specifications ", *Technical Report HPCA-ECS-95/2*, Dept. of E.E.E., The University of Adelaide, January 1995†.

[13] Morton, S.V., & Appleton, S.S., "Event Controlled Systems Design Methodology using a Temporal Specification Approach v.2 ", *Technical Report HPCA-ECS-93/10*, Dept. of E.E.E., The University of Adelaide, September 1993†.

[14] Hennessy, John L., & Patterson, David A., "Computer Architecture : A Quantitative Approach", Morgan Kaufmann Publishers, 1990.

[15] Appleton, S. S., Morton, S. V., & Liebelt, M. J., "Instruction and Data Caches for the ECSTAC microprocessor ", *Internal Report HPCA-ECS-94/7*, Dept. of E.E.E., The University of Adelaide, Decmeber 1994†.

[16] Horstmann, J.U., Eichel, H.W.,& Coates, R.L., "Metastability Behaviour of CMOS ASIC Flip-Flops in Theory and Test", IEEE Journal of Solid-State Circuits, Vol. 24 No. 1, Febuary 1989.

[17] Marino, L.R., "General Theory of Metastable Operation", IEEE Transactions on Computers, Vol. C-30 No. 2, Febuary 1981.

[18] Morton, S. V., Appleton, S. S., & Liebelt, M. J., "Register Control, Tags, and Write Back Strategies for ECSTAC-P ", *Internal Report HPCA-ECS-94/4*, Dept. of E.E.E., The University of Adelaide, February 1994†.

[19] Khoche, A., & Brunvand, Erik, "Testing Micropipelines", International Symposium on Advanced Research in Asynchronous Circuits and Systems", Salt Lake City, Utah, November 1994.

†These reports can be obtained from http://www.eleceng.adelaide.edu.au/Groups/DIGITAL/ECS/ecs.html