# AMULET3: A High-Performance Self-Timed ARM Microprocessor

S. B. Furber, J. D. Garside,
Department of Computer Science,
The University of Manchester,
Oxford Road, Manchester M13 9PL, UK.
sfurber, jgarside@cs.man.ac.uk

D. A. Gilbert,
Cogency Technology, Inc.,
Bruntwood Hall, Schools Hill,
Cheadle, Cheshire SK8 1HX, UK.
dg@cogency.co.uk

## Abstract

*AMULET3 is a fully asynchronous implementation of ARM architecture v4T and was designed at the University of Manchester between 1996 and 1998. It is the third generation asynchronous ARM, and is aimed at a significantly higher performance level than its predecessors. Achieving this higher performance has required significant enhancements to the internal micro-architecture, such as the introduction of a reorder buffer to support efficient forwarding while retaining exact exception handling.*

*In this paper we present an overview of the AMULET3 core, highlighting the issues which arise when striving for high performance and instruction set compatibility in an asynchronous design framework.*

## 1: Introduction

Recent industrial electronic design practice has been based predominantly around the use of clocked circuits. Asynchronous techniques have been little used due to the perceived difficulty of developing designs that can be relied upon to work correctly. However, two factors have contributed to renewed interest in asynchronous techniques:

- as semiconductor process technologies advance clocked design is becoming increasingly difficult, particularly with respect to clock skew management, power-efficiency and electromagnetic compatibility (EMC);
- asynchronous design techniques have been progressively advancing within adademia and certain industrial research laboratories, and over the last decade new styles have evolved which avoid most of the difficulties inherent in the older approaches [1,2].

Since asynchronous designs are devoid of problems with clock skew and have been shown to have very attractive EMC and power-efficiency properties [3], there is an opportunity for these advantages to be translated for commercial advantage and for asynchronous design to resume its role in industrial design practice.

A number of research groups both in academia and in industry are now producing self-timed processors targetted at a variety of applications. Some recent examples include: TITAC2 [4], the CalTech MIPS [5], the Philips 80C51 [6], ASPRO-216 [7], the Cogency DSP [3] and the LSI Logic/ DTU TinyRISC [8].

The AMULET series of microprocessors has been developed to demonstrate the feasibility, desirability and practicability of employing asynchronous techniques in embedded applications. The milestones so far include:

- AMULET1 [9,10], developed during 1991-93, showed that complex asynchronous design is possible;
- AMULET2e [11] (1994-96) showed that the advantages of asynchronous design can be realised in practice.

AMULET3 (1996-98) is being developed to establish the commercial viability of asynchronous design. Like its predecessors, AMULET3 is a full-functionality ARM-compatible [12] microprocessor with support for interrupts and memory faults. AMULET1 and AMULET2 implemented the ARM6 architecture (ARM architecture version 3). AMULET3 supports the current ARM architecture, version 4T, including the 16-bit Thumb instruction set [13], a compressed representation of the 32-bit ARM instruction set which improves code density and power-efficiency.

The objective of the AMULET3 project is to produce an asynchronous implementation of ARM architecture v4T which is competitive in terms of power-efficiency and performance with the latest clocked ARM core, the ARM9TDMI. This implies a performance target of well over 100 MIPS (measured with Dhrystone 2.1) on a 0.35 μm process, compared to the 40 MIPS delivered by AMULET2e on a 0.5 μm process. Increasing the performance by a factor of three or more required a radical change to the core organisation.

In the next section we introduce the context in which AMULET3 will be used. In section 3 we describe the organisation of the AMULET3 core, and section 4 covers the issues of maintaining compatibility with the existing

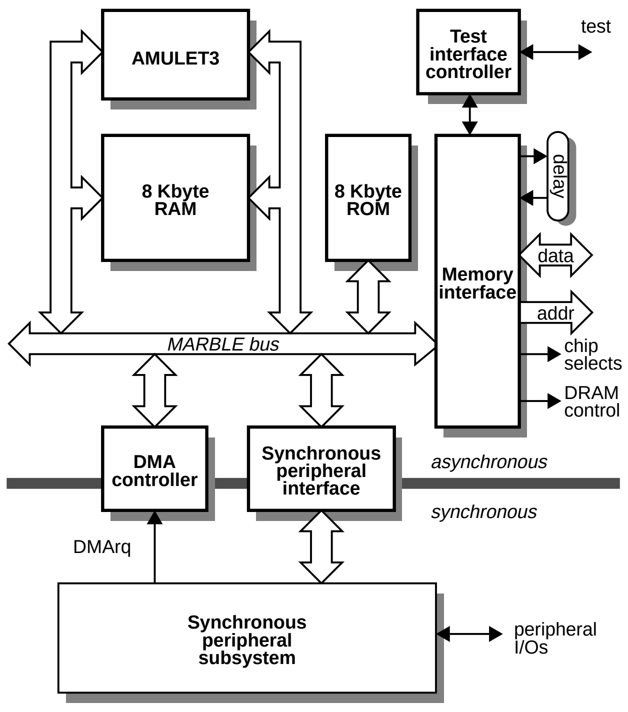

**Figure 1. AMULET3 asynchronous subsystem.**

ARM instruction set. In section 5 the dual-ported memory structure is described, and section 6 concludes the paper.

## 2: The AMULET3 subsystem

AMULET3 is being developed in the context of a microcontroller application which requires that it be interfaced to a range of synchronous peripheral controllers. To reap the benefits of asynchronous operation the core must have access to some memory that operates asynchronously, and there are significant EMC benefits in having off-chip memory also operate asynchronously.

The organisation of the asynchronous subsystem is illustrated in figure 1. The AMULET3 core is connected directly to a dual-ported RAM (discussed further in section 5) and then to the MARBLE on-chip bus [14]. MARBLE is similar in concept to ARM's AMBA bus [12], the major difference being that it does not use a clock signal.

System components other than the local RAM are accessed via the MARBLE bus. These include on-chip ROM, a DMA controller, a bridge to the synchronous bus where the application-specific peripheral controllers reside, and an interface to external memory. The external memory interface presents a conventional set of signals for off-chip devices. It is similar to the AMULET2e interface, being highly configurable and using a reference delay to time external accesses [11].

Overall about half of the device will operate asynchronously, but as this includes all of the highest speed components and most of the off-chip drivers, the EMC and power-efficiency advantages of asynchronous operation should not be too severely compromised by the presence of the clocked peripheral controllers.

## 3: AMULET3 core organisation

The ARM instruction set is register-oriented, and the register file organisation is central to the operation of the processor. The AMULET3 register file follows the example of the StrongARM (and subsequently ARM9TDMI) in having three register read ports rather than the two used in previous ARM (and AMULET) processors.

A simple examination of the ARM instruction set offers little justification for this; the frequency of instructions that require three source operands is too low to justify the cost of a third port. However, this is not the whole story. Without a third read port instructions which require three source operands must be decoded out and controlled separately so that they can access their operands in two cycles. This adds complexity to, and slows down, the decode logic. Adding a third register read port removes the need to handle these instructions separately; this simplifies the decode and control logic in away which more than offsets the added cost of the third port.

This means that almost all instructions are single cycle on AMULET3, the exceptions being multiplications with 64-bit results which may require *four* operands and produce two 32-bit results; these require two cycles. It is worth mentioning that the multiple register load/store operations (LDM/STM) are executed as single cycle operations by the execution path; whilst the data interface and part of the decoder must cycle repeatedly the ALU is used only once to calculate the required address offset and the instruction prefetch is likely to fill up and stall. This is a simple consequence of the asynchronous nature of the processor which contributes to its low-power operation.

### 3.1: The reorder buffer

The performance of a pipelined processor depends critically on maintaining the smooth flow of the pipeline. This flow is disturbed whenever pipeline stalls or flushes occur.

Many pipeline stalls are due to dependencies between consecutive instructions, where an instruction cannot proceed until its predecessors have yielded their operands. AMULET1 employs a register locking mechanism [15] to handle dependencies; this ensures correct operation but does nothing to minimise stalls. AMULET2 introduced forwarding paths which remove stalls in some common cases, but is complex to control. In AMULET3 a completely different approach has been adopted, based upon the use of a reorder buffer [16] as shown in figure 2.

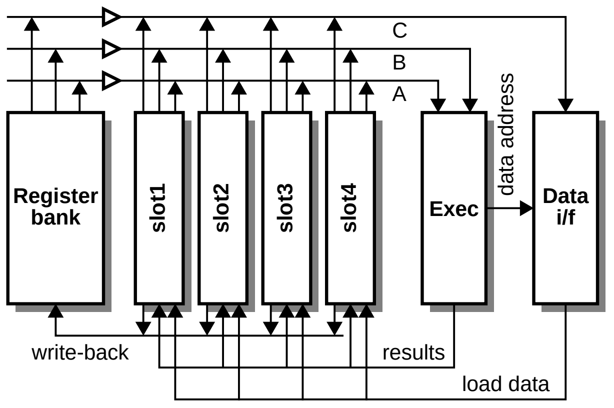

**Figure 2. Reorder buffer organisation.**

The reorder buffer is not a new idea – it has been used before in synchronous processors. However, as with many organisational features of clocked processors, transferring the idea into an asynchronous framework requires a careful rethink of the principles of operation since many of the timing assumptions upon which the synchronous use is based are invalid in a system which does not employ global synchronisation.

The basic operation of the AMULET3 reorder buffer is straightforward. Results arrive from one of two (or, arguably, three) streams: internally computed results are passed directly from the Execution unit; values loaded from memory are passed from the Data Interface. Since these two streams are unsynchronised, they generate results out of order. Any result which has arrived can be forwarded to enable a subsequent instruction to be issued speculatively.

Results are written back into the register file in order, in the course of which they are checked for validity. An attempted load from memory may have caused a memory fault, in which case an exception is generated, subsequent results are discarded, and the processor state upon exception entry is exactly that required for resumption once the cause of the exception has been remedied.

So far the use of the reorder buffer is conventional. The difficulty that must be addressed is to determine exactly when values in the reorder buffer can be used for forwarding since the write-back process operates completely asynchronously to the issue mechanism. The insight here is to observe that writing a value back to the register file is a copy process that does *not* invalidate the entry in the reorder buffer. The entry can continue to be forwarded until its location is reallocated to a subsequent result. Since entry allocation is carried out in the Decode stage where fowarding is also controlled, there is no synchronisation required. In effect, forwarding takes place in complete ignorance of whether or not the value being forwarded has already been transferred back into the register file and the two operations may even be concurrent.

This last case could mean that the register is being changed whilst it is being read, resulting in an indeterminate output. However, whenever this is the case the register value will always be replaced by a forwarded value. To guarantee correct operation it is only necessary to ensure that the indeterminate values (which may not even have valid logic levels) cannot cause excess power consumption.

An added complication to the reorder buffer inherent in the ARM instruction set is that any instruction may be conditional, thus instructions may be issued which expect to produce a value but subsequently do not. Unlike the case of memory faults these instructions must interleave with valid operations, marking only their own results as invalid. The forwarding mechanism must then cope with cases where, for example, several consecutive ARM instructions conditionally modify a source register for a subsequent instruction. Here the reorder buffer must be searched, in the correct order, for the first valid result for this register. At each entry the search may have to wait for the result to arrive before its validity can be checked. The typical case is very simple; there will be zero or one entries to examine. The worst case will involve all the entries being examined in sequence. However, the asynchronous control structures easily tolerate the variability in search time and, provided it occurs infrequently, the occasional worst-case search will have little impact on the processor's overall throughput.

## 3.2: Branch prediction

Pipeline flushes happen as a result of branch instructions. AMULET3 will minimise the impact of these by using a branch prediction mechanism which is an enhanced version of the mechanism used on AMULET2e.

Although AMULET2e predicts branch operations, the branch instructions are still fetched in order to determine their conditional status and whether they save a return address (for procedure entry). This means that a 32-bit instruction is fetched to obtain five bits of information; this is clearly inefficient. In AMULET3, therefore, these bits are stored in the prefetch unit along with the branch target. When a branch is predicted the instruction is already present within the processor and therefore the memory cycle can be avoided. As branches account for 10%-15% of instructions [17], and the majority of these are cached [18], this represents a considerable power saving. Furthermore, this 'prefetch' cycle will be considerably faster than a memory access and this will be exploited automatically by the asynchronous pipeline.

## 3.3: Halting and Interrupts

Experience with AMULET2e has demonstrated the benefits of a halt mechanism within an asynchronous system, especially with regard to saving power [11]. To summarise,

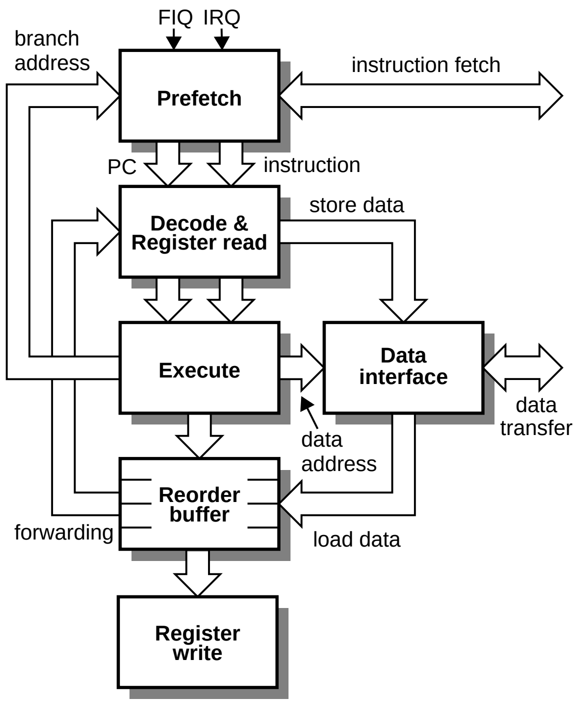

**Figure 3. AMULET3 core organisation.**

stalling the pipeline at any point will cause the whole pipeline to stop extremely rapidly and power consumption to drop to near zero (there being no clock signals causing further transitions); recovery to full-speed operation is equally rapid. AMULET2e and AMULET3 decode a branch to itself as a ‘Halt’ instruction and use this to stall the pipeline.

Whereas AMULET2e stalls in the execution stage, AMULET3 adopts a somewhat cleaner model to halting by stalling at the prefetch stage. This means that the processor restarts with an empty pipeline which provides the fastest possible response, any “rubbish” being cleared out at halt entry. Halt abrogation is caused by the arrival of an enabled interrupt.

A reference to figure 3 shows an interesting consequence of this operation – the interrupt signals are fed into the prefetch unit rather than the instruction decoder. This rather unusual architectural feature provides both a clean interrupt model and a low interrupt latency. The assertion of an interrupt signal is detected and arbitrated into the prefetch stream in the same way as a new branch; because the interrupt code begins at a predefined position, the prefetch unit can then switch its instruction fetch stream in a way similar to it predicting a branch. The interrupt thus sends a marker to save the return address (which, like a predicted branch, can bypass the memory part of the prefetch) and follows this immediately with the interrupt service code.

### 3.4: Indirect branches

ARM programs often load the program counter (PC) directly from memory as part of subroutine returns (and, less frequently, jump table lookups). When the PC is loaded from the stack during a subroutine return, it is loaded after any working registers which are restored as part of the same load multiple instruction. This ordering – necessary to allow the processor to recover from a memory fault which may arise during the load – delays the resumption of instruction fetching, losing performance.

AMULET3 incorporates an optimisation which exploits the separate instruction memory port (see figure 3). The execution unit passes the address of the PC value back to the Prefetch unit early in the transfer via the branch path, allowing the Prefetch unit to fetch the PC value and then start fetching from the return address. The Data Interface must still perform the PC fetch later to check for a possible memory fault.

Note that this optimisation requires that the separate instruction and data ports connect to coherent memories as the PC was saved through the data port but is loaded through the instruction port (see section 5).

## 4: ARM v4T compatibility

Previous AMULET processors achieved full code-compatibility with the ARM6 processor, but this instruction set architecture has been largely superseded by more recent developments in the ARM family [19]. Particularly in the market for portable embedded controllers, the 16-bit Thumb instruction set [13] has been widely accepted because of its beneficial effects on code density and hence power-efficiency. In systems with 8- and 16-bit external memories, Thumb code can also improve performance.

### 4.1: Thumb code

The Thumb instruction set does not define a complete architecture; Thumb systems still depend on the underlying processor supporting the full ARM instruction set for infrequent operations such as interrupt handling, and time-critical routines can run as standard ARM code (usually from 32-bit on-chip memory). Thumb can properly be seen as a compressed representation of a carefully selected subset of the ARM instruction set, and a processor which supports it may do so by means of decompression hardware between the instruction fetch and decode stages.

The ARM7TDMI supports Thumb in exactly this way. It employs a 3-stage pipeline which has enough slack in the decode stage to decompress Thumb code without incurring any time penalty. The more recent ARM9TDMI has a tighter 5-stage pipeline where decompression would require an additional pipeline stage, so instead the designers chose to decode Thumb instructions directly rather than decompressing them into ARM instructions and then using the standard ARM decode logic.

The AMULET3 strategy for Thumb code is intermediate between these two extremes. The elastic nature of the asynchronous pipeline opens up some options which are not available when a rigid clocked pipeline is used. The Thumb decode logic is split into three aspects:

- the time-critical register operand selectors are decoded directly;
- the immediate field extraction logic works directly on the Thumb binary codes;
- the remaining decode work is done via decompression and the ARM decode logic.

Where the depth of the Thumb decode logic is in danger of compromising the cycle time of the decode pipeline stage, parts of it can be pushed upstream into earlier buffer stages. Here they will affect the latency but not the throughput of the pipeline.

## 4.2: Program counter tracking

The ARM architecture defines the program counter (PC) to appear in register 15 in the register file. Here it can be read and written much like any other register, though the architecture definition places some restrictions on its use.

The high visibility of the PC is a considerable programming convenience. Position-independent code is easy to generate, and literals can be picked up from pools near the code using PC-relative loads (though this practice is discouraged when a Harvard ARM core is used at it results in inefficient use of the caches). One particularly powerful use of the PC is illustrated by the standard procedure prologue and epilogue, where on entry a single store multiple instruction pushes temporary work register values and the return address onto the stack, and on exit a single load multiple instruction restores the work registers and causes a return by loading the PC directly from the stack.

The value that appears in register 15 is not, however, the address of the current instruction. The 3-stage ARM pipeline increments the PC twice between fetching an instruction and executing it, so r15 yields the value PC+8 (where here PC *is* the address of the current instruction). The ‘+8’ corresponds to the double increment where each increment is the size of one ARM instruction, namely 4 bytes.

The ARM7TDMI pipeline behaves in exactly the same way when executing Thumb code, but Thumb instructions are only 2 bytes long, so r15 yields PC+4 in Thumb code.

In AMULET3, instruction fetching proceeds autonomously, independently from the register read process. Therefore each instruction is associated with its fetch address (see figure 3) when it is sent to the Decode stage. The fetch address can then be modified (by adding 8 in ARM code or 4 in Thumb code) to generate the appropriate r15 value. This means that a consequence of the earlier, synchronous ARM implementations imposes a noticable cost on AMULET3.

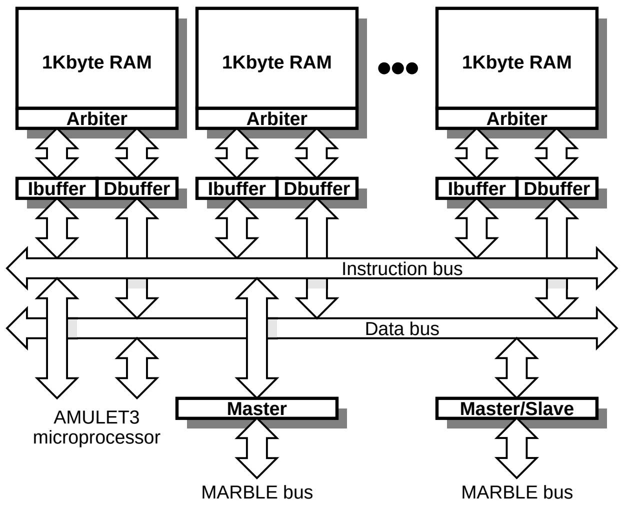

**Figure 4. AMULET3 memory organisation.**

There are several other places in the organisation where adjustments must be made to the PC value to retain compatibility with the behaviour of the 3-stage pipeline clocked ARM. Examples are:

- the procedure return address calculation (PC+4, or PC+2 in Thumb mode) which is not calculated by the ALU in AMULET3, and
- the address which is stored on a memory abort (PC+8, Thumb mode independent).

## 5: AMULET3 memory organisation

The AMULET3 processor core has separate address and data buses for instruction and data memory accesses. This would normally require separate instruction and data memories; RISC systems frequently employ a ‘modified Harvard’ architecture where there are separate instruction and data caches with a unified main memory.

The AMULET3 controller employs direct-mapped RAM rather than cache memory as this is more cost-effective and has more deterministic behaviour for real-time applications. It also avoids separate instruction and data memories (and the associated difficulties of keeping them coherent) through the use of a dual-ported memory structure (see figure 4). Dual-porting the memory at the individual bit level would be too costly, so instead the memory is divided into eight 1 Kbyte blocks, each of which has two ports which are arbitrated internally. When concurrent data and instruction accesses are to different RAM blocks, each can proceed unimpeded by the other; when they happen to conflict on the same block, one access will suffer a delay while it waits for the other to complete.

Conflicts (and average memory access times) are further reduced by including separate quad-word instruction and data buffers in each RAM block. Each access to a block first checks to see whether the required data is in the buffer. Only if it is not must the RAM be interrogated, with a risk of conflict. Simulations suggest that about 60% of instruction fetches may be satisfied from within these buffers and many short, time-critical loops will run entirely from here.

These buffers, in effect, form simple 128 byte first-level caches in front of the RAM blocks. This is a particularly apt analogy when it is observed that the avoidance of the RAM array results in a faster read cycle, an occurrence which is accepted automatically by the asynchronous pipeline.

## 6: Conclusions

AMULET3 is a third generation asynchronous implementation of the ARM instruction set architecture. It is intended for use as a core macrocell to demonstrate the commercial viability of asynchronous design. Particular benefits are apparent where low-power consumption and EMC are important, such as portable telecommunications, palmtop computers and contactless smartcards.

In order to achieve the performance expected of current embedded cores, the internal organisation of AMULET3 has been significantly enhanced over previous AMULET designs. Stalls due to data dependencies are minimised through the use of a reorder buffer which, while not a new idea in itself, has required new insights in order to adapt it to an asynchronous design framework. Circuit-level simulation of those parts which are near completion suggest that the performance should at least equal the ARM9TDMI. Preliminary layout indicates a core area of around 4.0 mm$^2$ (ARM9TDMI is 4.8 mm$^2$).

Backwards instruction set compatibility is a source of complexity in any new processor design. The ARM instruction set has evolved over the 15 years of its existence into a dense encoding which, while well-suited to the power-efficient applications for which it is intended, makes any new implementation a difficult task. The exposure of the original 3-stage pipeline in the instruction set definition is also a source of difficulty for any implementation which uses a different pipeline organisation. AMULET3 shows that an asynchronous design framework does not prevent these difficulties from being overcome.

## 7: Acknowledgments

The development of AMULET3 has been support primarily within the EU-funded OMI-DE2 and OMI-ATOM projects, and authors are grateful to the European Commission for their continuing support for this work. ARM Limited coordinated these projects; their support, and that of the other project partners, is also acknowledged.

Aspects of the work have benefited from support from the UK government through the EPSRC, particularly in the form of a PhD studentship (DAG) and the tools development funded under ROPA grant GR/K61913.

The VLSI design work has leant heavily on CAD tools from Compass Design Automation (now part of Avant!) and EPIC Design Technology, Inc. (now part of Synopsis).

## 8: References

[1] I.E. Sutherland. Micropipelines. *Communications of the ACM*, 32 (6): 720-738, June 1989.
[2] K. van Berkel. *Handshake Circuits: An Asynchronous Architecture for VLSI Programming*, vol. 5, Intl. Series on Parallel Computation, Cambridge University Press, 1993.
[3] N.C. Paver, P. Day, C. Farnsworth, D.L. Jackson, W.A. Lien and J. Liu. A Low-Power, Low-Noise Configurable Self-Timed DSP. *Proc. Async'98*, April 1998.
[4] A. Takamura, M. Kuwako, M. Imai, T. Fujii, M. Ozaw, I. Fukasaku, Y. Ueno and T. Nanya. TITAC2: An Asynchronous 32-Bit Microprocessor Based on Scalable-Delay Insensitive Model. *Proc. ICCD'97*, 288-294, October 1997.
[5] A.J. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings and T.K. Lee. The Design of an Asynchronous MIPS R3000 Microprocessor. *Proc. 17th Conf. on Advanced Research in VLSI*, September 1997.
[6] H. van Gageldonk, D. Baumann, K. van Berkel, D. Gloor, A. Peeters and G. Stegmann. An Asynchronous Low-Power 80C51 Microcontroller. *Proc. Async'98*, April 1998.
[7] M. Renaudin, P. Vivet and F. Robin. ASPRO-216: A Standard-Cell QDI 16-Bit RISC Asynchronous Microprocessor. *Proc. Async'98*, April 1998.
[8] K.T. Christensen, P. Jensen, P. Korger and J. Sparsoe. The Design of an Asynchronous TinyRISC TR401 Microprocessor Core. *Proc. Async'98*, April 1998.
[9] S.B. Furber, P. Day, J.D. Garside, N.C. Paver and J.V. Woods. The Design and Evaluation of an Asynchronous Microprocessor. *Proc. ICCD'94*, 217-220, October 1994.
[10] N.C. Paver. *The Design and Implementation of an Asynchronous Microprocessor.* PhD Thesis, Manchester Univ., 1994.
[11] S.B. Furber, J.D. Garside, S. Temple, J. Liu, P. Day and N.C. Paver. AMULET2e: An Asynchronous Embedded Controller. *Proc. Async'97*, 290-299, April 1997.
[12] S.B. Furber. *ARM System Architecture*. Addison Wesley Longman, 1996. ISBN 0-201-40352-8.
[13] S. Segars, K. Clarke and L. Goudge. Embedded Control Problems, Thumb, and the ARM7TDMI. *IEEE Micro*, 15(5):22-30, October 1995.
[14] W.J. Bainbridge and S.B. Furber. Asynchronous Macrocell Interconnect using MARBLE. *Proc. Async'98*, April 1998.
[15] N.C. Paver, P. Day, S.B. Furber, J.D. Garside and J.V. Woods. Register Locking in an Asynchronous Microprocessor. *Proc. ICCD'92*, 351-355, October 1992.
[16] D.A. Gilbert. *Dependency and Exception Handling in an Asynchronous Microprocessor.* PhD Thesis, Manchester Univ., 1997.
[17] D. Jaggar. *A Performance Study of the Acorn RISC Machine*. M.Sc. Thesis, Univ. of Canterbury, 1990
[18] R. York. *Branch Prediction Strategies for Low Power Microprocessor Design*. MSc Thesis, Manchester Univ., 1994.
[19] D. Jaggar. *Advanced RISC Machines Architecture Reference Manual*. Prentice Hall, 1996. ISBN 0-13-736299-4.