AMULET2e

S. B. Furber, J. D. Garside, S. Temple,
Department of Computer Science, The University of Manchester,
Oxford Road, Manchester M13 9PL, UK.

P. Day, N. C. Paver,
Cogency Technology Inc., Bruntwood Hall, Schools Hill,
Cheadle, Cheshire SK8 1HX, UK.

Abstract. AMULET1, developed in the OMI-MAP project, showed that complex asynchronous design is feasible. AMULET2e, developed in the OMI-DE project, is the next step in establishing self-timed processing as a commercially viable technology. It incorporates a new core, AMULET2, which is significantly faster than AMULET1, together with on-chip RAM which can be configured as a fixed block of RAM or as a cache. A flexible memory interface supports a 'conventional' external memory and peripheral system, freeing the board-level designer from the asynchronous timing concerns which have made some previous self-timed chips (including AMULET1) difficult to use as components.

1. Introduction

The AMULET processor cores are fully asynchronous implementations of the ARM architecture [1], which means that they are self-timed, operating without an externally supplied clock. At the time of writing they are unique; no other commercial microprocessor architecture has been implemented in this style.

1.1. Motivation for self-timed design

Although some early computers operated with asynchronous control, for the last three decades nearly all commercial design has been based upon the use of a central clock. Synchronous design is conceptually simpler than asynchronous design, and great progress has been made in the performance and cost-effectiveness of electronic systems using straightforward clocked design. It is perhaps necessary, therefore, to justify abandoning the design methodology which has supported these great advances.

The principal motivation for re-examining asynchronous design is its potential for power-efficiency. The clock in a synchronous circuit runs continuously, causing transitions in the (typically CMOS) circuit that dissipate electrical power. The clock frequency must be set so that the processor can cope with the peak work-load and, although the clock rate can be adjusted under software control to accommodate varying demands, this can only be done relatively crudely at a coarse granularity. Therefore much of the time the clock is running faster than is necessary, resulting in wasted power. An asynchronous design, on the other hand, only causes transitions in the circuit in response to a request to carry out work; it can switch instantaneously between zero power dissipation and maximum performance. Since many embedded applications have rapidly varying work-loads, an asynchronous processor appears to offer the potential of significant power savings.

Additional reasons for looking at asynchronous design include its lower emission of electromagnetic radiation due to less coherent internal activity, and its potential to deliver typical rather than worst-case performance since its timing adjusts to actual conditions whereas a clocked circuit must be toleranced for worst-case conditions.

Asynchronous designs are inherently modular and display the properties of data encapsulation and clean interfaces that are so favoured by object-oriented programmers. This makes them well-suited to the construction of very large systems; within the next decade a single chip will encompass a very large system as transistor counts grow towards a billion. Concurrent systems are also most readily expressed in terms that map naturally onto asynchronous rather than clocked hardware, and this translates into efficient synthesis routes that are more straightforward than those available for clocked systems [2].

1.2. Industrial applicability

There is considerable resistance amongst industrial designers to the use of asynchronous design techniques, in part because of obsolete perspectives that view asynchronous design as unreliable (this objection has now largely been overcome by new, more rigorous techniques) and in part because of genuine difficulties with production testability which are only now beginning to be addressed by the asynchronous research community.

However, clocked design is becoming ever more difficult as manufacturing process technologies shrink. Wiring delays increasingly dominate logic delays on high performance chips, causing the global clock wiring to compromise the local logic performance. Clock generators consume an increasing share of the silicon resource and the power budget, and increasing clock frequencies cause worsening radio emissions and power consumption.

The choice facing manufacturers of portable digital equipment will therefore be either to sacrifice performance or to abandon fully synchronous design. The pressure will be most apparent in physically small systems that include both radio receivers and high-performance digital electronics, such as digital mobile telephones and pagers. PDAs, mobile email terminals and portable multimedia terminals will also soon benefit from asynchronous design. In the longer term the modularity, composability and synthesis potential of asynchronous circuits will make them attractive to a wide range of applications, when dealing with a global clock signal on a 1000,000,000 transistor chip will become simply too difficult to manage.

The AMULET processors are intended to demonstrate the feasibility and benefits of asynchronous techniques, thereby hastening commercial application of the technology.

2. Asynchronous design

Asynchronous design is a complex discipline with many different facets and many different approaches [3,4]. Among the fundamental concepts is the idea of asynchronous communication, or how the flow of data is controlled in the absence of a reference clock.

The AMULET designs both use forms of the *Request-Acknowledge* handshake to control the flow of data. The sequence of actions comprising the communication of data from the *Sender* to the *Receiver* is as follows:

1. The Sender places a valid data value onto a bus.
2. The Sender then issues a *Request* ‘event’.
3. The Receiver accepts the data when it is ready to do so.
4. The Receiver issues an *Acknowledge* ‘event’ to the Sender.
5. The Sender may then remove the data from the bus and begin the next communication when it is ready to do so.

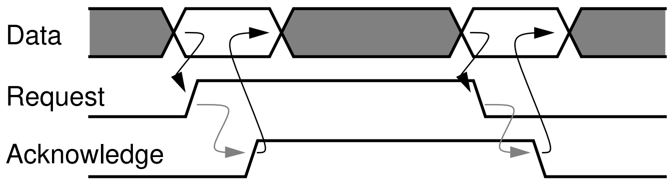

Figure 1. Transition-signalling communication protocol.

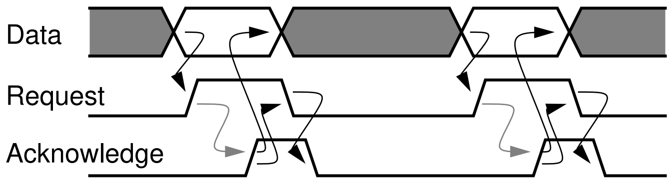

Figure 2. Level-signalling communication protocol.

The data is passed along the bus using a conventional binary encoding, but the nature of an 'event' is open to different interpretations. There are two commonly-used ways of signalling the Request and Acknowledge events:

- Transition signalling [5]

  AMULET1 [4,6-10] uses transition encoding where a change in level (either high to low or low to high) signals an event; this is illustrated in Figure 1.

- Level signalling

  AMULET2 uses level encoding where a rising edge signals an event and a return-to-zero phase must occur before the next event can be signalled; this is illustrated in Figure 2.

Transition signalling was used on AMULET1 since it is conceptually cleaner; every transition has a role and its timing is therefore determined by the circuit's function. It also uses the minimum number of transitions, and should therefore be power-efficient. However the CMOS circuits used to implement transition control are relatively slow and inefficient. Therefore AMULET2 uses level signalling which employs circuits which are faster and more power-efficient (despite using twice the number of transitions) but leave somewhat arbitrary decisions to be taken about the timing of the recovery (return-to-zero) protocol phases [11].

2.1. Self-timed pipelines

An asynchronous pipelined processing unit can be constructed using self-timing techniques to allow for the processing delay in each stage and one of the above protocols to send the result to the next stage.

When the circuit is correctly designed, variable processing delays and arbitrary external delays can be accommodated. All that matters is the local sequencing of events; long delays will degrade performance but they will not affect functionality.

Unlike a clocked pipeline, where the whole pipeline must always be clocked at a rate determined by the slowest stage under worst-case environmental (voltage and temperature) conditions and assuming worst-case data, an asynchronous pipeline will operate at a variable rate determined by the current conditions. It is possible to allow rare worst-case conditions

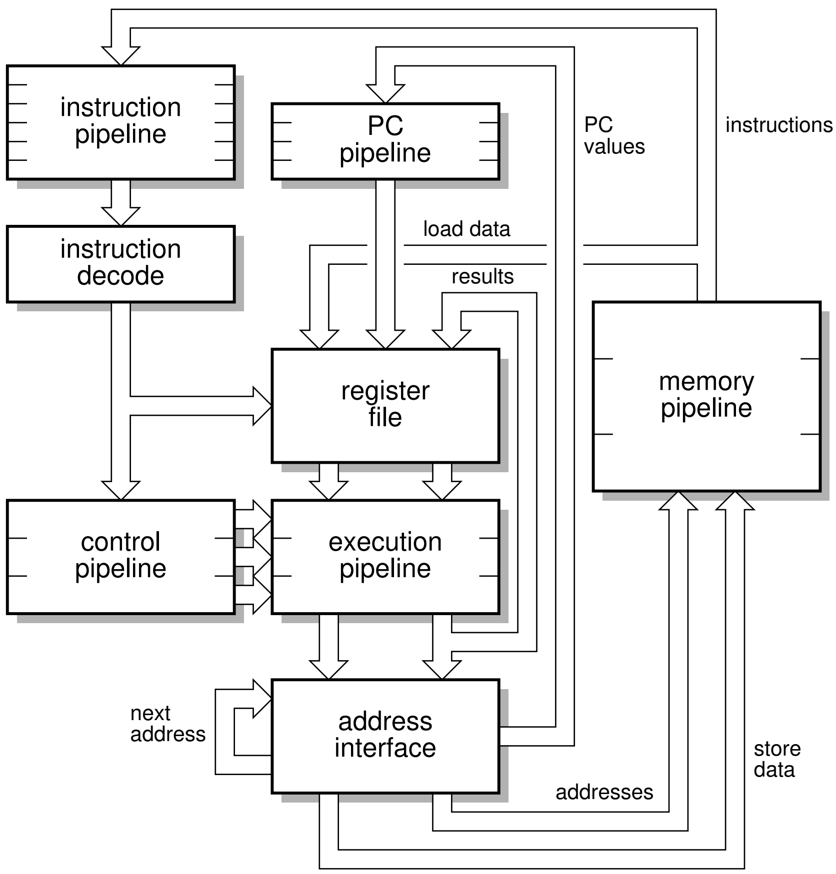

Figure 3. AMULET internal organization.

to cause a processing unit to take a little longer. There will be some performance loss when these conditions do arise, but so long as they are rare enough the impact on overall performance will be small. For example, a self-timed adder can terminate when the carry propagation has completed, which incurs a data-dependent delay. Similarly a cache can exploit address sequentiality to give a faster access time.

Performance can also be traded-off against power-efficiency by adjusting the supply voltage (possibly dynamically [12]) without having to adjust a clock frequency.

3. Self-timed ARM cores

Both AMULET processor cores have the same high-level organization which is illustrated in Figure 3. The design is based upon a set of interacting asynchronous pipelines, all operating in at their own speed. These pipelines might appear to introduce unacceptably long latencies into the processor but, unlike a synchronous pipeline, an asynchronous pipeline can have a very low latency since data can propagate very rapidly through empty stages.

The operation of the processor begins with the address interface issuing instruction fetch requests to the memory. The address interface has an autonomous address incrementer which enables it to prefetch instructions as far ahead as the capacities of the various pipeline buffers allow. The AMULET2 address interface also incorporates a *Jump Trace Buffer* which attempts to predict branches from past behaviour.

Instructions from the memory flow into the instruction pipeline then into the instruction decoder. They access their operands in the register file, process them, and the results are then passed back to the register file or used as addresses to access data in memory.

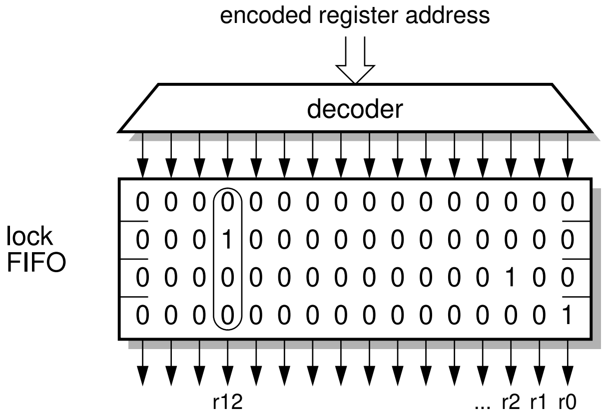

Figure 4. AMULET register lock FIFO organization.

3.1. Address non-determinism

Instructions that need to generate an out-of-sequence memory address, such as data transfer instructions and unpredicted branches, calculate the new address in the execution pipeline and then send it to the address interface. Since it arrives with arbitrary timing relative to the internal incrementing loop within the address interface, the point of insertion of the new address into the address stream is not predictable. This means that the processor's depth of prefetch beyond a branch instruction is fundamentally non-deterministic.

3.2. Register coherency

If the execution pipeline is to work efficiently, the register file must be able to issue the operands for an instruction before the result of the previous instruction has returned. However, in some cases an operand may depend on a preceding result (this is a *read-after-write* hazard), in which case the register file cannot issue the operand until the result has returned unless there is a forwarding mechanism to supply the correct value further down the pipeline.

On the AMULET processors register coherency is achieved through a novel form of register locking, based on a register 'lock FIFO' (first-in-first-out queue) [13]. The destination register numbers are stored, in decoded form, in a FIFO, until the associated result is returned from the execution or memory pipeline to the register bank.

The organization of the lock FIFO is illustrated in Figure 4. Each stage of the FIFO holds a '1' in the position corresponding to the destination register. In this figure the FIFO controls 16 registers (in the AMULET chips the FIFO has a column for each physical register) and is shown in a state where the first result to arrive will be written into r0, the second into r2, the third into r12 and the fourth FIFO stage is empty. If a subsequent instruction requests r12 as a source operand, an inspection of the FIFO column corresponding to r12 (outlined in the figure) reveals whether or not r12 is valid. A '1' anywhere in the column signifies a pending write to r12, so its current value is obsolete. The read waits until the '1' is cleared, then it can proceed.

The 'inspection' is implemented in hardware by a logical 'OR' function across the column for each register. This may appear hazardous since the data in the FIFO may move down the FIFO while the 'OR' output is being used, but it is not. Data moves in an asynchronous FIFO by being duplicated from one stage into the next, only then is it removed from the first stage. Therefore a propagating '1' will appear alternately in one or two positions and will never disappear completely, and the 'OR' output will be stable even though the data is moving.

3.3. AMULET2 register forwarding

In order to reduce the performance loss due to register dependency stalls, AMULET2 incorporates forwarding mechanisms to handle common cases. The register bypass mechanism used in clocked processor pipelines is inapplicable to asynchronous pipelines, so novel techniques are required. The two techniques introduced on AMULET2 are:

- A *last result* register.

  The instruction decoder keeps a record of the destination of the result from the execution pipeline and if the immediately following instruction uses this register as a source operand the register read is bypassed and the value is collected from the last result register.

- A *last loaded data* register.

  The instruction decoder keeps a record of the destination of the last data item loaded from memory, and whenever this register is used as a source operand the register read phase is bypassed and the value is picked up directly from the last loaded data register. A mechanism similar to the lock FIFO serves as a guard on the register to ensure that the correct value is collected.

Both these mechanisms rely on the required result being available; where there is some uncertainty (for example when the result is produced by an instruction which is conditionally executed) the instruction decoder can fall back on the locking mechanism, exploiting the ability of the asynchronous organization to cope with variable delays in the supply of the operands.

3.4. AMULET2 jump trace buffer

AMULET1 prefetches instructions sequentially from the current program counter (PC) value and all deviations from sequential execution must be issued as corrections from the execution pipeline to the address interface. Every time the PC has to be corrected performance is lost and energy is wasted in prefetching instructions that are then discarded.

AMULET2 reduces this inefficiency by remembering where branches were previously taken and guessing that control will subsequently follow the same path. The organization of the jump trace buffer is shown in Figure 5; it is similar to that used on the MU5 mainframe computer [9] developed at the University of Manchester between 1969 and 1974 (which also operated with asynchronous control).

The buffer caches the program counters and targets of recently taken branch instructions, and whenever it spots an instruction fetch from an address that it has recorded it modifies the predicted control flow from sequential to the previous branch target. If this prediction turns out to be correct, exactly the right instruction sequence is fetched from memory; if it turns out to be wrong and the branch should not have been taken, the branch is executed as an ‘unbranch’ instruction to return to the previous sequential flow.

The effectiveness of the jump trace buffer depends on the statistics of typical branch behaviour. Typical figures are shown in Table 1.

In the absence of the jump trace buffer, the default sequential fetch pattern is equivalent to predicting that all branches are not taken. This is correct for one third of all branches, and incorrect for the remaining two-thirds. A jump trace buffer with 20 entries reverses these proportions, correctly predicting around two-thirds of all branches and mispredicting or failing to predict around one third.

Although the depth of prefetching beyond a branch is non-deterministic, a prefetch depth of around 3 instructions is observed on AMULET1. The fetched instructions are used

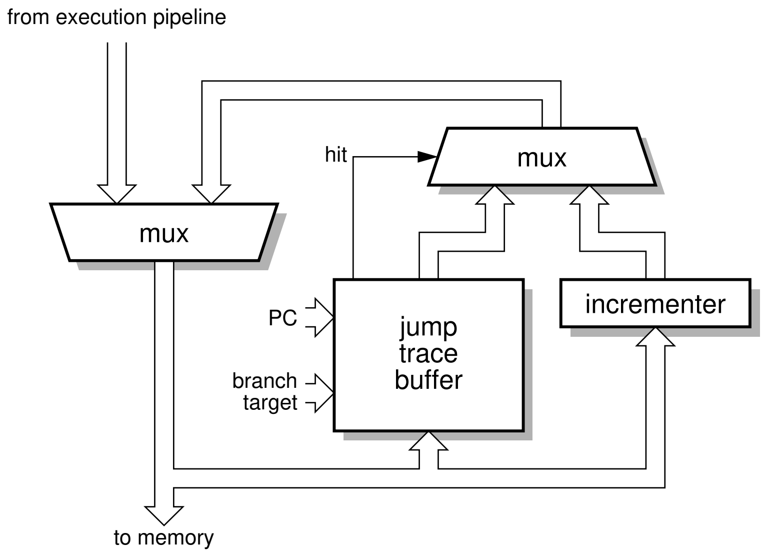

Figure 5. The AMULET2 jump trace buffer.

Table 1. AMULET2 branch prediction statistics.

Prediction algorithm	**Correct**	**Incorrect**	**Redundant fetches**
Sequential	33%	67%	2 per branch (ave.)
Trace buffer	67%	33%	1 per branch (ave.)

when the branch is predicted correctly, but are discarded when the branch is mispredicted or not predicted. The jump trace buffer therefore reduces the average number of redundant fetches per branch from two to one. Since branches occur around once every five instructions in typical code, the jump trace buffer may be expected to reduce the instruction fetch bandwidth by around 20% and the total memory bandwidth by 10% to 15%. Where the system performance is limited by the available memory bandwidth this saving translates directly into improved performance, and it always represents a power saving due to the elimination of redundant activity (especially in the memory system).

3.5. AMULET2 'halt' instruction

An important aspect of minimizing power wastage is to ensure that when there is nothing useful to do, the processor ceases all activity. AMULET2 detects when the software has entered an idle loop and stops execution at that point. An interrupt request cause the processor to resume execution immediately, switching from zero power consumption to maximum throughput instantly without any software overhead.

4. AMULET2e

AMULET2e is an AMULET2 processor core combined with 4 Kbytes of memory, which can be configured either as a cache or a fixed RAM area, and a flexible memory interface (the *funnel*) which allows 8-, 16- or 32-bit external devices to be connected directly, including memories built from DRAM. The internal organization of AMULET2e is illustrated in Figure 6.

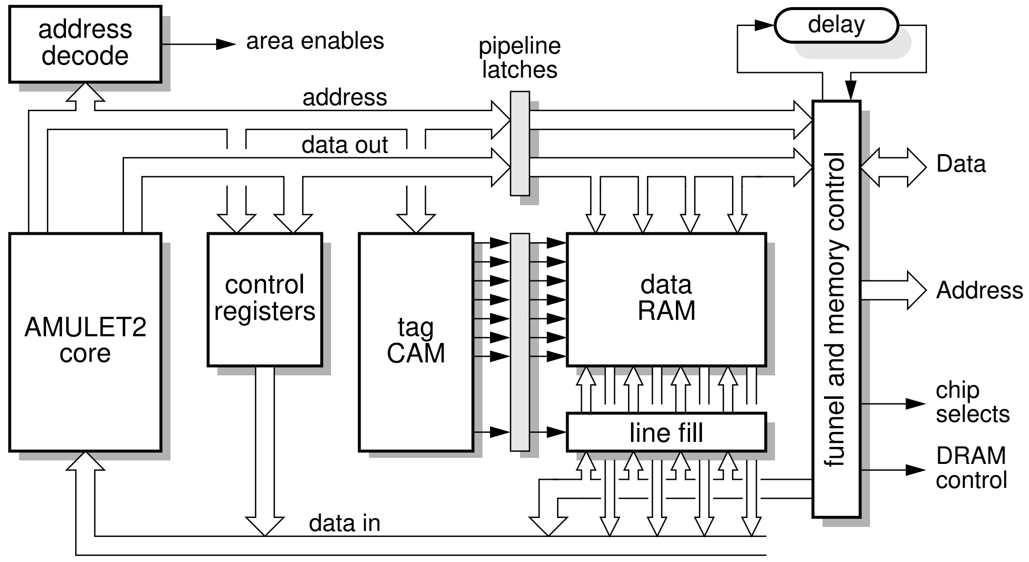

Figure 6. AMULET2e internal organization.

4.1. *Timing reference*

The absence of a reference clock in an asynchronous system makes timing memory accesses an issue that requires careful consideration. The solution incorporated into AMULET2e uses a single external reference delay connected directly to the chip and configuration registers, loaded at start-up, which specify the organization and timing properties of each memory region. The reference delay will normally reflect the external SRAM access time, so the RAM will be configured to take one reference delay. The ROM, which is typically much slower, may be configured to take several reference delays. (Note that the reference delay is only used for off-chip timing; all on-chip delays are self-timed.)

4.2. *AMULET2e cache*

The write-through cache [14] comprises four 1 Kbyte blocks, each of which is a fully associative random replacement store with a quad-word line and block size. Two address lines select which of the four blocks is used for a particular access.

A pipeline register between the CAM and the RAM sections allows a following access to begin its CAM lookup while the previous access completes within the RAM; this exploits the ability of the AMULET2 core to issue multiple memory requests before the data is returned from the first. Sequential accesses are detected and bypass the CAM lookup, thereby saving power and improving performance.

Cache line fetches are non-blocking [15], accessing the addressed item first and then allowing the processor to continue while the rest of the line is fetched. The line fetch automaton continues loading the line fetch buffer while the processor accesses the cache. There is an additional CAM entry that identifies references to the data which is stored in the line fetch buffer. Indeed, this data remains in the line fetch buffer where it can be accessed on equal terms to data in the cache until the next cache miss, whereupon the whole buffer is copied into the cache while the new data is loaded from external memory into the line fetch buffer. This allows the main cache to be accessed even while the line fetch mechanism is operating.

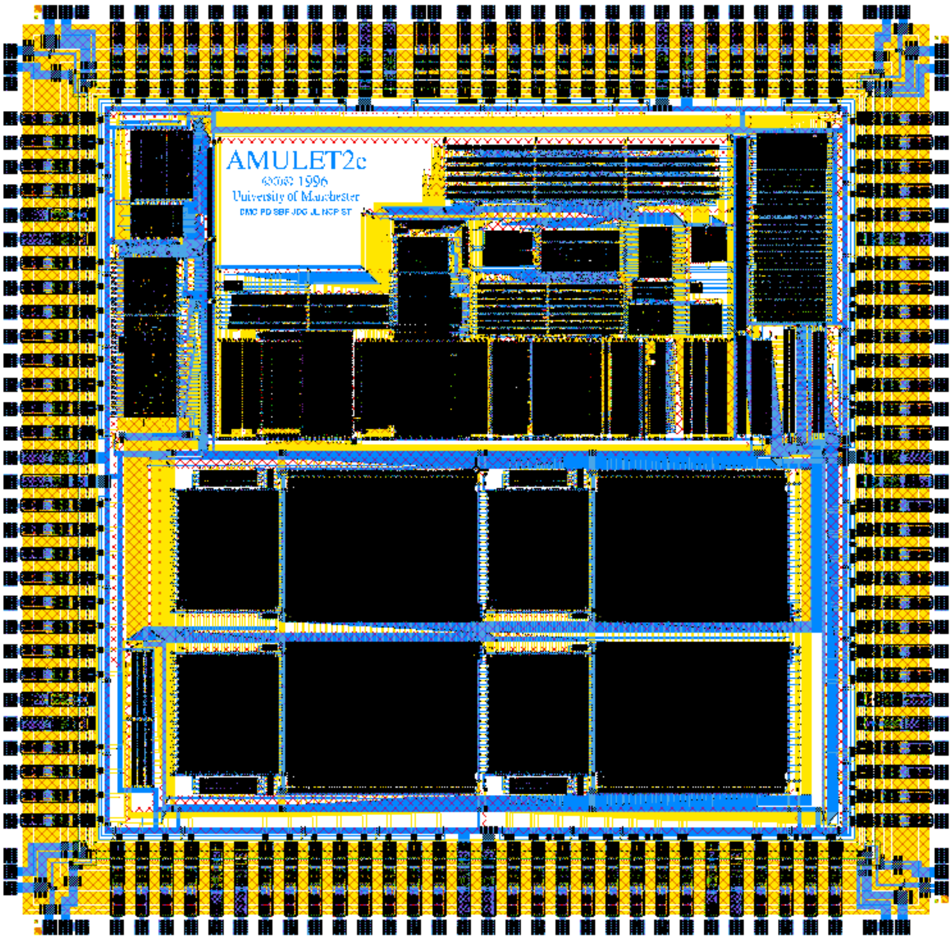

Figure 7. AMULET2e die plot.

4.3. AMULET2e systems

AMULET2e has been designed to make building small systems as straightforward as possible. As an example, Figure 8 shows a simple development board incorporating AMULET2e. The only components, apart from AMULET2e itself, are four byte-wide SRAM chips (though the system could equally well have been designed to operate with just one, at lower performance), one byte-wide ROM chip, a UART and an RS232 line interface. The UART uses a crystal oscillator to control its bit rate and to provide a real-time clock, but all the system timing functions are controlled by AMULET2e using the single reference delay.

The ROM contains the debug monitor code used on the clocked ARM development boards [1] and the host computer, at the other end of the RS232 serial line, runs the standard ARM development tools. This system demonstrates that using an asynchronous processor need be no more difficult than using a conventional clocked processor provided that the memory interface has been carefully thought out.

5. Results

At the time of writing AMULET2e has just been shipped for fabrication and therefore parts are not yet available for testing. The device is designed on a 0.5 micron CMOS process with 3 metal layers and uses 454,000 transistors (93,000 of which are in the processor core) on a die which is 6.4 mm square. It will be packaged in a 128-pin flat-pack and operate at 3.3 V.

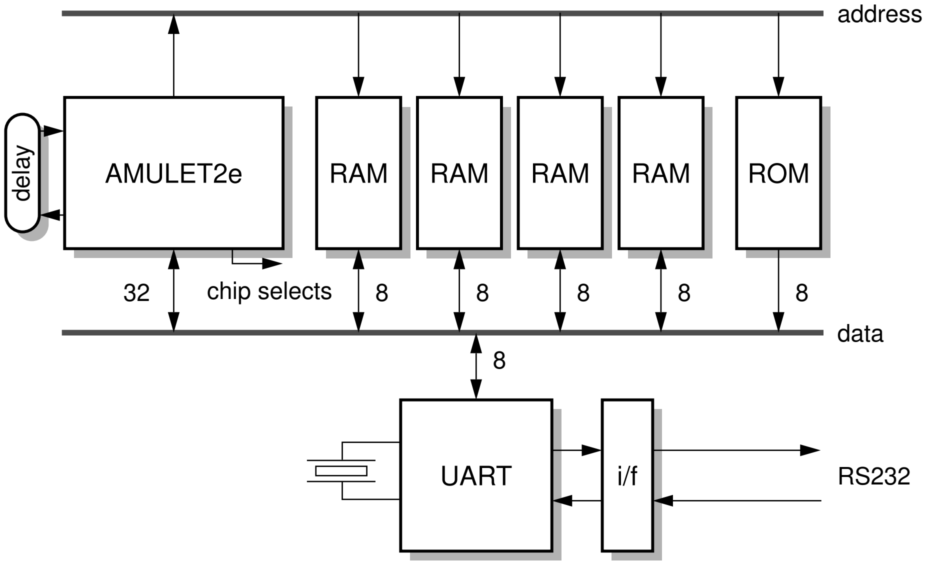

Figure 8. AMULET2e development card organization.

Timing-accurate simulation has been carried out on the transistor netlist using TimeMill from EPIC Design Technology, Inc. This shows a performance of 38 dhrystone MIPS running code produced by the standard ARM C compiler, which is faster than the ARM710 but only about half the performance of the recently announced ARM810 on the same process technology. Back-end optimization to reorder instructions should deliver some improvement in performance since AMULET2 has code order dependencies which the ARM compiler does not take into account.

The motivation for the work is primarily to demonstrate the low-power potential of asynchronous technology and secondarily to investigate its EMC advantages. Analysis of these factors must await the availability of silicon. However at this stage it is clear that the performance, design and manufacturing costs are broadly in line with those of a clocked device.

6. Conclusions

The AMULET chips are presently research prototypes and are not about to replace synchronous ARM cores in commercial production. However, there is a world-wide resurgence of interest in the potential of asynchronous design techniques to save power and to offer a more modular approach to the design of computing hardware.

The power savings which result from removing the global clock, leaving each subsystem to perform its own timing functions whenever it has useful work to perform, are clear in theory but there are few demonstrations that the benefits can be realized in practice with circuits of sufficient complexity to be commercially interesting. The AMULET research is aimed directly at adding to the body of convincing demonstrations of the merits of asynchronous technology. It is also clear that, should asynchronous technology gain acceptance as a low-power design style, the AMULET work places the ARM architecture in the vanguard of the asynchronous assault on the stronghold of the global clock.

The objective of AMULET2e is to demonstrate that a self-timed processing system can deliver competitive performance in a very flexible way, simplifying power-efficient design and minimising electromagnetic interference. Asynchronous designs are naturally miserly with power, since they are inactive until presented with work to do. The power benefits are expected to be particularly manifest in systems with highly variable work-loads, hence the emphasis on embedded applications. The electromagnetic interference benefits come from spreading the peak currents in time and frequency.

AMULET1 showed that asynchronous design is feasible; AMULET2e will demonstrate its advantages.

7. Acknowledgements

AMULET1 was developed using European Community funding within the Open Microprocessor systems Initiative - Microprocessor Architecture Project (OMI-MAP). AMULET2 was developed within the Open Microprocessor systems Initiative - Deeply Embedded ARM (OMI-DE/ARM) project.

The work also received support in various forms from Advanced RISC Machines Limited, GEC Plessey Semiconductors and VLSI Technology, Inc. Tools from Compass Design Automation were important to the success of both projects, and TimeMill from EPIC Design Technology, Inc. was vital to the accurate modelling of AMULET2.

The authors are grateful for this support from industry and the EC.

8. References

[1] S. B. Furber, *ARM System Architecture*, Addison Wesley Longman, 1996. ISBN 0-201-40352-8.

[2] K. van Berkel, *Handshake Circuits. An Asynchronous Architecture for VLSI*, Cambridge University Press, 1993.

[3] G. Gopalakrishnan and P. Jain, *Some Recent Asynchronous System Design Methodologies*, Dept. of C.S., Univ. of Utah, Tech. Report UU-CS-TR-90-016, October 1990.

[4] G. Birtwistle and A. Davis (eds.), *Asynchronous Digital Circuit Design*, Proc. 1993 VIIth Banff High Order Workshop, Springer, 1995. ISBN 3-540-19901-2.

[5] I. E. Sutherland, Micropipelines, *Communications of the ACM*, **32** (1989) 720-738.

[6] S. B. Furber, P. Day, J. D. Garside, N. C. Paver and J. V. Woods, A Micropipelined ARM, *Proc. IFIP TC 10/WG 10.5 International Conference on Very Large Scale Integration* (VLSI'93), Grenoble, France, September 1993. Ed. Yanagawa, T. and Ivey, P. A. North Holland.

[7] S. B. Furber, P. Day, J. D. Garside, N. C. Paver and J. V. Woods, AMULET1: A Micropipelined ARM, *Proc. IEEE Computer Conference*, March 1994.

[8] 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*, Boston, October 1994.

[9] D. Morris and N. N. Ibbett, *The MU5 Computer System*, Macmillan, New York, 1979.

[10] N. C. Paver, *The Design and Implementation of an Asynchronous Microprocessor*, PhD Thesis, University of Manchester, June 1994.

[11] S. B. Furber and P. Day, Four-Phase Micropipeline Latch Control Circuits, *IEEE Trans. on VLSI Systems*, June 1996.

[12] K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, F. Schalij and R. van de Wiel, A Single-Rail Re-implementation of a DCC Error Detector Using a Generic Standard-Cell Library, *Proc. Async'95*, London, UK, May 30-31 1995.

[13] N. C. Paver, P. Day, S. B. Furber, J. D. Garside and J. V. Woods, Register Locking in an Asynchronous Microprocessor, *Proc. ICCD '92*, Boston, October 1992, pp. 351-355.

[14] J. D. Garside, S. Temple and R. Mehra, The AMULET2e Cache System, *Proc. Async'96*, Aizu-Wakamatsu, Japan, March 18-21 1996.

[15] R. Mehra and J. D. Garside, A Cache Line Fill Circuit for a Micropipelined Asynchronous Microprocessor, *TCCA Newsletter*, IEEE Computer Society, October 1995.