AMULET2e: An Asynchronous Embedded Controller

S. B. Furber, J. D. Garside, S. Temple, J. Liu,
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 demonstrated the feasibility of building an asynchronous implementation of the ARM microprocessor. Although functional, this first asynchronous ARM microprocessor did not fully exploit the potential of the asynchronous design style to deliver improved performance and power consumption.

This paper describes AMULET2e, an embedded system chip incorporating an enhanced asynchronous ARM core (AMULET2), a 4 Kbyte pipelined cache, a flexible memory interface and assorted programmable control functions. AMULET2e silicon demonstrates competitive performance and power-efficiency, ease of design, and innovative features that exploit its asynchronous operation to advantage in power-sensitive applications.

1. Introduction

While asynchronous design is enjoying increasing attention from the academic community and initial stirrings of interest from industry, its progress towards realising its full commercial potential continues to be impeded by a shortage of large-scale demonstrations of merit. For the last five years the AMULET group at the University of Manchester has spent most of its energies addressing this shortfall and gaining experience of asynchronous engineering ‘in the large’.

The first milestone in this work was AMULET1 [1], an asynchronous implementation of the ARM [2] 32-bit RISC microprocessor which used a two-phase bundled data design style based closely on Sutherland’s Micropipelines [3]. AMULET1 was broadly comparable with, but not superior to, clocked ARM processors built on the same technology, fulfilling its primary role of demonstrating the feasibility of designing complex asynchronous circuits with the resources and tools available to the group. It also taught us a great deal about practical asynchronous design both from the things that we got right and from the things we got wrong.

The second milestone in this work has now been reached. AMULET2e is an asynchronous embedded controller incorporating AMULET2 (a significantly enhanced version of AMULET1), 4 Kbytes of RAM which can be configured to operate as a cache, a flexible memory interface which makes the system designer’s job look quite conventional, a counter-timer for real-time reference and various configuration and control registers. First silicon arrived on October 1 1996 having passed functional tests at the foundry (VLSI Technology, Inc.) without difficulty and within a few hours a sample was communicating with the standard ARM development tools and running compiled C programs. The parts are highly functional, perform exactly as predicted by our simulation tools, and have the sort of performance and flexibility that will attract applications developers to look again at asynchronous technology.

In the next section we review AMULET1, looking particularly at the lessons we learnt from it which influenced the design of AMULET2e. Then the AMULET2 processor core is described in section 3. In section 4 we present the organisation of AMULET2e and in section 5 we present a simple system designed around the part. In section 6 we give a summary and analysis of the test results and we draw conclusions in section 7.

2. AMULET1

The AMULET1 organisation has been described elsewhere [1,4,5,6,7] so only a summary is presented here. The processor to memory interface follows the Micropipeline convention, with one (output) bundle to send address, control and write data to the memory and a second (input) bundle to return read data from the memory. The memory system may have an arbitrary pipeline depth and delay, but must return read values in the requested order.

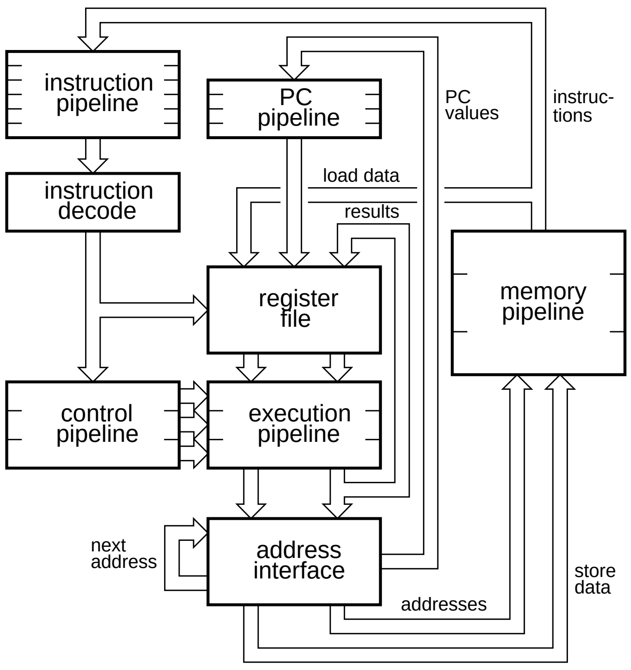

Figure 1: AMULET1 internal organization.

Internally the processor may be viewed as comprising several pipeline units (Figure 1) which operate independently and concurrently, exchanging information through bundled data interfaces. The role of each of these units is described briefly below.

Address interface. The address interface is responsible for issuing read and write requests to memory. It issues instruction prefetch requests autonomously and accepts data transfer and branch target addresses from the execution unit as required. Branch target addresses are immediately issued to memory and also change the prefetching stream to continue from the target location; data transfer addresses temporarily interrupt the prefetching stream which resumes once the data address has been issued.

The ARM architecture makes the program counter readily accessible to the programmer as register 15 in the register bank. PC values are therefore copied from the address interface to the register bank through a PC pipeline which buffers the values until the associated instruction arrives from memory.

Register file. All the user accessible state is held in the register bank, which employs a novel locking mechanism [8] to allow multiple pending writes from the execution pipeline and from external memory. The locking mechanism ensures the correct behaviour of instruction streams with data dependencies between successive instructions and enables register read and write processes to proceed asynchronously without arbitration and without risk of metastability in the control and data circuits.

Execution pipeline. Arithmetic processing is carried out in the execution pipeline. This incorporates a '3-bits at a time' carry-save multiplier, a barrel shifter and rotator and an ALU. The ALU has a data dependent propagation delay which detects the longest carry chain in an addition [9]. This allows a relatively simple ALU to give better average performance on a typical mix of operand values than the more complex ALU in the clocked ARM6, since there is no need to coerce the worst case addition into a fixed clock period.

Instruction decoder. The instruction decoder accepts instructions from the instruction pipeline and generates the necessary control signals to pass to the register file and to the execution pipeline (via the control pipeline, where some further decoding takes place). The major decode function of the instruction decoder is implemented by a large PLA, but there are other complex control functions (such as splitting a single ARM instruction into several execution pipeline operations) that lead to considerable complexity in this area.

2.1. AMULET1 lessons

AMULET1 was a major design project which had to be completed with limited resource and within a limited time. It clearly solves all the problems which must be solved to implement a functional asynchronous microprocessor, but the solutions are not all equally good. In some areas we consider that we have found solutions which are elegant and efficient:

- the (patented) register locking mechanism [8] has still not been bettered. Although it is desirable to avoid stalls by bypassing when possible (which AMULET1 made no attempt at, though AMULET2 does), the totally dependable consistency offered by this mechanism has stood up well through the developments that followed;
- the instruction prefetching with its 'colour' management of non-determinism [1] has also scaled well. The non-determinism is a potential source of difficulty for test vector generation, but otherwise it solves a tricky problem in a straightforward and efficient manner;
- the overall organisation based on interacting micropipelines has proved reasonably straightforward to design and optimise.

Against these positive lessons, there were a number of experiences with AMULET1 that we wished to avoid repeating:

- although micropipelines worked well on chip, they proved very troublesome at board level. AMULET1 is a basic processor core with a two-phase micropipeline interface at the pins, and debugging the logic which handled these two-phase signals took a long time - it took almost a month from receiving the first silicon before we knew that the chips were basically functional;

- two-phase design is conceptually straightforward, but our CMOS implementations of two-phase control elements were somewhat inefficient. All event registers had two- to four-phase conversions inside them, and all dynamic logic structures needed four-phase conversions also. Even where four-phase control is not required, steering two-phase signals (which are edges) requires circuits with state and XOR gates, since CMOS is fundamentally a level-sensitive technology;
- building deep pipelines in a micropipeline circuit is too easy. The AMULET1 execution pipeline is deeper than is useful, and performance is lost as a result. It is actually quite hard to balance asynchronous pipelines!

3. AMULET2

These lessons formed the starting point for AMULET2. A four-phase bundled data design style was adopted [10], a little more care taken over the pipeline depths, and a lot more attention was paid to the system interface at the chip pins. In addition, several architectural features were added to improve the performance and power-efficiency of the device. These are described below.

3.1. Pipeline reorganisation

As was mentioned above a retrospective analysis of the AMULET1 design revealed that the depth of pipelining is too great. This is partly due to FIFO buffers being conceptually easy to use within the Micropipeline design style, and as a result too many were added. There are many stages that contribute little (or nothing) towards performance but still cost silicon area, transistors and power dissipation, and some stages actually decrease performance!

This analysis identified the main execution pipeline as a candidate for pipeline simplification.

The ARM architecture specifies that the shifter can be used to shift one of the operands in many of the instruction classes. However, in practice most shift operations are performed on immediate values and this can be done before the immediate value is passed to the execution pipeline, in parallel with reading the register bank. This, coupled with the fact that a high percentage of instructions do not take advantage of the shift operation at all, means that there are performance and power gains to be achieved by bypassing the shifter when it is not in use (which is most of the time).

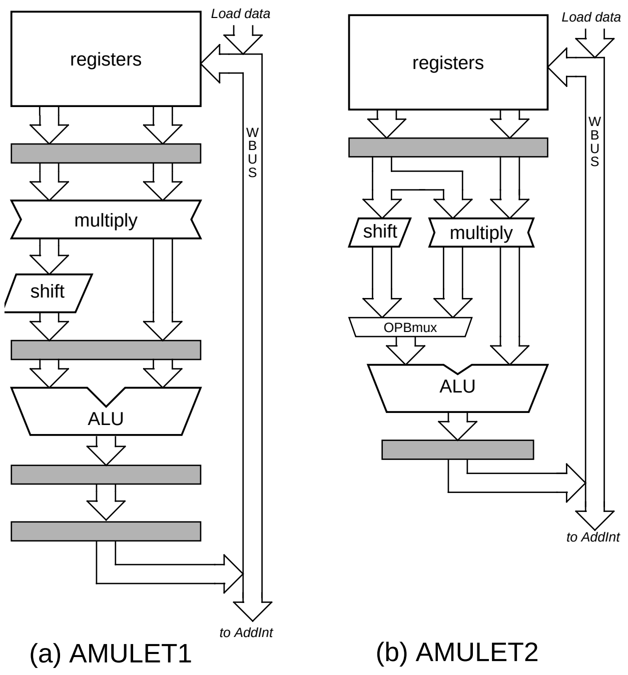

Figure 2: AMULET1&2 execution pipelines

Figure 2 shows a comparison of the AMULET1 and AMULET2 execution pipeline structures with the AMULET2 pipeline showing the shifter bypass route through the multiplier. (The multiplier contains an internal bypass mechanism, so if neither the shifter or multiplier are to be activated the internal multiplier bypass path is used.) The shifting of immediate values is performed elsewhere in order to exploit this pipeline organisation fully.

Now that the shifter and multiplier are bypassed for most operations, there is little justification for providing a separate pipeline stage for them. This also facilitates the ALU register forwarding scheme (described in the next section) where the last result value only has to propagate backwards across a single pipeline stage.

The final difference between the two execution pipeline organisations is that the final result latch has been replaced with latches inside the register bank and address interface. This reduces the amount of time that the write bus (a shared resource) is busy for a given writeback operation and reduces arbitration clashes between the ALU and data interface, thus improving performance.

3.2. Register forwarding

The register management scheme in AMULET1 provides an effective mechanism for ensuring register coherency. However, the locking mechanism employed causes the pipeline to stall when any register dependency is detected; processing only continues when the value that is

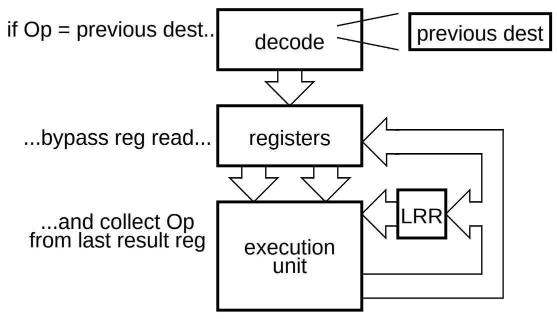

Figure 3: LRR control algorithm

required has returned to the register bank.

Conventional synchronous processors overcome this problem by the use of register forwarding. To implement register forwarding in an asynchronous organisation would be costly because complex synchronisation points would be required to compare register addresses and exchange values. AMULET2 therefore uses the concept of last result registers (sometimes known as reservation stations) to achieve similar results.

The are two schemes employed in AMULET2: the ALU last result register (LRR) is used when the result calculated by the ALU is required as an operand by the next instruction; the last loaded value (LLV) register is used when the operand being loaded from memory is required by one of the following instructions. The control mechanism and validity of the data is different for the two types.

Both mechanisms can be disabled independently if required, allowing their effectiveness to be measured (and allowing them to be turned off completely in the event that their design is incorrect and they cause the silicon to fail!).

ALU last result register. The ALU last result mechanism divides into two distinct parts; the last result register itself and the control to indicate when to use the value in the LRR.

The LRR is simply a transparent latch in anti-phase to the ALU result latch. Whenever the ALU is activated the previous value is automatically available on the LRR.

The control (see Figure 3) keeps a record of the destination address of the previous instruction and compares the operand addresses of the current instruction. If the comparison matches and the LRR value is valid (not all instructions produce results that can be used), then the instruction does not read its operand from the register bank but retrieves the required values from the LRR via a set of multiplexers (see Figure 4).

As the LRR is anti-phase with the ALU output latch its value is only valid for the next instruction. As soon as the next instruction passes through the ALU the LRR is automatically updated with the new last result value.

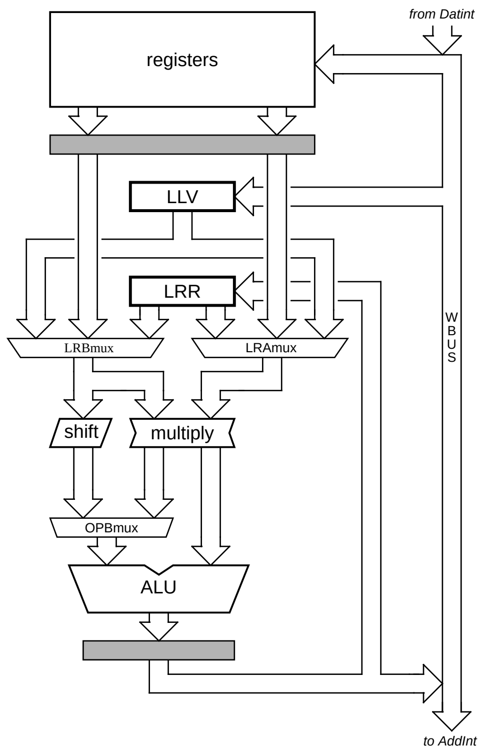

Figure 4: Register forwarding organisation

Last Loaded Value. The last load value (LLV) mechanism also divides into data storing and control partitions. This time the LLV register is updated with a load data value directly from the processor write bus every time a data value returns from memory to the register bank. The value in the LLV is therefore valid from when the data arrives from memory until the next update of the LLV (another load returning data from memory) or an ALU operation renders the cached register value in the LLV invalid by writing to the same destination register. The value in the LLV can therefore be valid for a number of consecutive instructions (c.f. the ALU last result where the validity of the data is from one instruction to the next only).

The control of the LLV is considerably more complex that the ALU last result due to two key ‘features’:

- The validity of the LLV must be managed more explicitly rather than being taken care of automatically (as was the ALU last result);
- The LLV cannot be used until the value has returned from memory and there may be multiple outstanding load operations. In the ALU last result the value is available immediately in the ALU if the control indicates forwarding is possible.

The first ‘feature’ can be addressed by additional decode logic to detect instructions which would invalidate the LLV prematurely (e.g. an ALU operation with the same destination register as the value stored in the LLV).

The second ‘feature’ is more complex. An instruction wishing to use the LLV value must somehow synchronize with the returning value. Unfortunately the returning memory data does not know that it should be forwarded to the LLV as the load was dispatched before the following instruction detected a data fowarding opportunity. Therefore explicit synchronisation is not practical. A technique based upon the lock FIFO [8] principle can be used to solve the problem. Every load which is issued places a token in a 1-bit FIFO and every data value which returns is copied into the LLV register (overwriting any previous value) and a token is removed from the FIFO. An instruction which wishes to use the LLV value must wait until the FIFO is empty before taking the value and proceeding. The empty FIFO state confirms that the value in the LLV register is truly the ‘last’ value and not some preceding value.

Forwarding issues. Both forwarding mechanisms bypass a register bank read operation (otherwise they would stall there and deliver no benefit), so an instruction that uses forwarding must be sure that the value will, indeed, become available. A feature of the ARM instruction set [2] is that every instruction has a conditional guard on its execution; if the condition test fails it will produce no result. Therefore only the results of instructions with the guard ‘ALWAYS’ (that is, instructions that are unconditionally executed) are guaranteed to be available and only these results may be used for forwarding. In all other cases the instruction must fall back on the register locking mechanism to ensure it gets the correct operand values.

Since there is always a fall-back mechanism, the forwarding logic only serves to improve performance. There is little performance benefit from engaging forwarding in infrequently used instructions, so the decoder can take a conservative approach and only attempt to use the forwarding mechanism for frequently used instruction classes.

3.3. The branch target cache

AMULET1 prefetches instructions sequentially from the current 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 attempts to reduce this inefficiency by remembering where branches were previously taken and guessing that control will subsequently follow the same path. The organization of the Branch Target Cache (BTC) is shown in Figure 5; it is similar to the ‘Jump Trace Buffer’ used on the MU5 mainframe computer [11] developed at the University of Manchester between 1969 and 1974 (which also operated with asynchronous control).

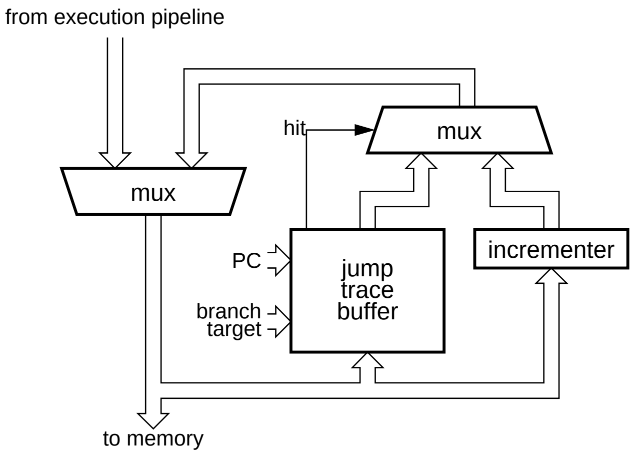

Figure 5: Branch target cache organisation

The BTC caches the program counters and targets of 20 recently taken branch instructions, and whenever it spots an instruction fetch from an address that it has stored it modifies the predicted control flow from sequential to the previous branch. When this prediction is correct, exactly the right instruction sequence is fetched. When it is wrong, the correction mechanism (an ‘unbranch’) has the same cost as an unpredicted taken branch.

Note that, except when it is being updated, the operation of the BTC is entirely local to one pipeline stage in the address incrementer loop. It is this aspect of the organisation which is well-suited to asynchronous implementation and led us to choose this approach over the many other branch prediction schemes used in clocked processors.

Although not shown in Figure 5, the flow of data required to update the cache is almost as convenient. The cache is updated whenever an unpredicted branch is taken. When this happens, the execution stage calculates the branch target by adding an offset to the PC and then passes the result along with the PC to the address interface. These are exactly the values required to update the BTC.

A good way to think about the BTC is to view the incrementer in AMULET1 as a first-order next instruction address predictor and the BTC as a second-order correction unit. They occupy exactly the same pipeline slot in the address interface and work in parallel, with the BTC having priority whenever it recognises the input address. When the prediction is correct the instruction flow is smooth; when it is wrong recovery is necessary. In AMULET1 the prediction is wrong whenever a branch is executed and recovery is performed by executing the branch. In AMULET2 the prediction is wrong when an unpredicted branch is executed (recovered by executing the branch) or when a branch is predicted in error (which is recovered by executing an ‘unbranch’).

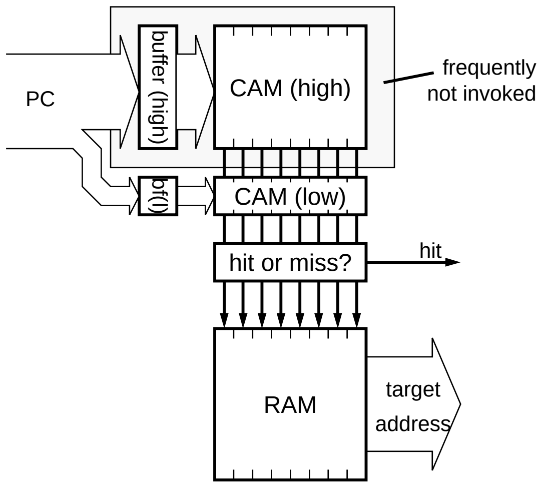

Figure 6: BTC internal structure

BTC implementation. Since the BTC performs a look-up on every instruction address issue it is important that it consumes minimal power. Its basic structure is that of an associative memory, and for performance reasons the associative store is built from content-addressable memory (CAM) which tends to be power-hungry. To reduce the power consumption of the CAM it is segmented into two sections (see Figure 6): the larger section takes all of the address apart from a few low-order bits; the smaller section deals with these low-order bits. Since most instruction fetches run sequentially, the high-order bits change rarely and the high section of the CAM need not be activated (provided that its last output was stored). Therefore only the small section of the CAM is active on every cycle. This segmentation of the CAM saves around 70% of the power consumption of the CAM, and it also reduces the average look-up time, improving performance.

The RAM part of the BTC is only activated when there is a hit in the CAM, so its contribution to the overall power consumption is small.

Despite careful design the BTC will still consume some power. However it should also reduce the total number of instruction fetches, improving performance and saving power in the cache and/or external memory, so with careful design it should save system power overall.

3.4. 'Halt'

Most ARM programs, when they run out of useful work to do, enter an idle loop implemented as 'B .' where an instruction continuously loops back to itself until an interrupt occurs. Since this wastes power, AMULET2 detects this instruction and a mechanism (patent applied for) stalls a control loop in the execution pipeline. The stall rapidly propagates throughout the system, halting all activity. An interrupt causes an immediate resumption of processing at maximum performance.

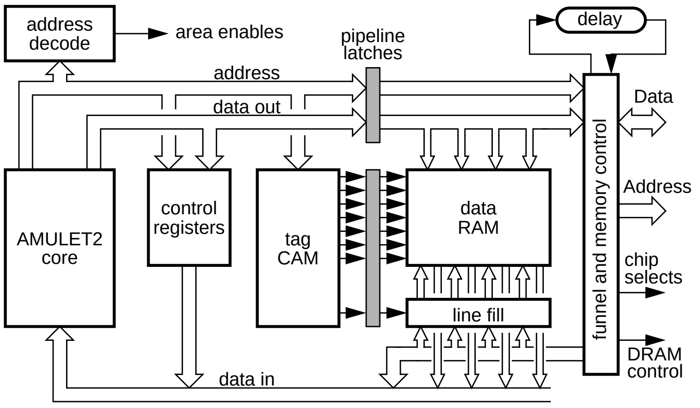

Figure 7: AMULET2e internal organisation

4. AMULET2e

AMULET2e is an asynchronous embedded system controller incorporating an AMULET2 core as described above along with a cache/RAM, a flexible memory interface and various control functions (including a timer/counter which will typically be driven from a 32 kHz crystal oscillator). Its organisation is illustrated in Figure 7.

4.1. The AMULET2e cache

The cache [12] is 4 Kbytes of RAM divided into four 1 Kbyte blocks, each block having an associated 64-entry tag CAM. When configured to operate as a cache (it may, alternatively, operate as a memory mapped RAM) the tag and data accesses are pipelined. The cache is 64-way associative with a quad-word line. Refill is addressed-word first [13], and the processor may continue accessing other cache locations while the refill completes. Refill data is held in the line-fetch latch until the next cache miss, so an additional CAM entry identifies subsequent hits on the line-fetch latch.

The CAM and RAM are self-timed for asynchronous operation using dummy matched paths, and the organisation incorporates a number of power-saving features:

- sequential accesses within a line bypass the CAM lookup (this also improves performance);
- the RAM sense amps do not turn on until the differential data is almost ready, and turn off as soon as the value has been sensed;
- only the addressed 1 Kbyte block is activated in any access. The other blocks remain inactive and consume no power.

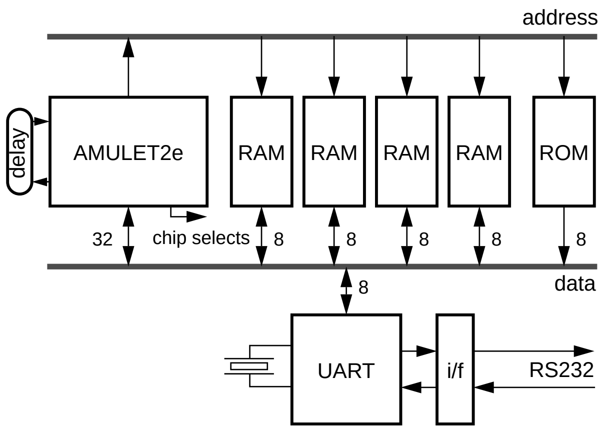

Figure 8: AMULET2e test card organization.

4.2. The memory interface

Perhaps the most forceful lesson from AMULET1 was the need to make the memory interface easier to use. AMULET2e presents a relatively conventional interface to the system designer, with a bidirectional data bus and an address bus and a range of chip select lines. DRAM strobes are also generated on-chip.

Since it is unreasonable to require external memory and peripheral components to provide completion signals (at least until asynchronous design has wider commercial support), some mechanism must be provided to ensure that the timing requirements of these components are met. In a clocked circuit, the period of a crystal oscillator provides a very reliable reference for this purpose, but we wish to avoid the power overhead of running an oscillator at memory speeds. The solution adopted on AMULET2e is to provide a 'reference delay' and to program all external accesses in multiples of this delay. Since on-chip delays are subject to process variation, the reference delay is off chip. It may be a simple RC delay, an integrated delay line or a silicon delay line.

External memory may be 8-, 16- or 32-bits wide. The ARM 32-bit address space is divided into eight regions, and each region can be configured independently for bus width and access timing. This allows the system to boot from a single slow 8-bit ROM, for example.

Full details of the memory configuration registers are available in the AMULET2e datasheet [14].

5. AMULET2e system design

As an example of AMULET2e system design, Figure 8 shows how the AMULET2e equivalent of the ARM 'PIE' (Platform Independent Evaluation) card is designed.

An 8-bit ROM holds the 'Demon' debug monitor code, which is unchanged from the PIE ROM apart from a few instructions used to configure the various memory regions. One location in the ROM is used by the AMULET2e hardware to configure the region occupied by the ROM itself so that the system can boot-strap.

Four SRAM chips provide the main memory. The system can operate with one or two RAMs, but four gives the best performance.

The remaining components are the UART and RS232 line driver chips used to communicate with the host machine.

At board level, then, the chip is conventional and building a system straightforward. The flexible memory interface results in a very low chip count.

6. AMULET2e test results

First silicon was delivered from VLSI Technology, Inc., on 1 October 1996. The chips had been packaged and tested, passing the test program with little trouble. The test, while not giving the level of coverage that would be required for commercial production, was sufficiently extensive to give considerable confidence that the parts were functional. For example, at one point the program loads a RAM test routine into the on-chip memory which the AMULET2 core than executes at full speed, without external intervention from the tester, before returning a signature result which confirms that the memory has passed the test.

The parts were functionally tested in a card as described in the previous section. The first result was that the objective of simplifying the system design interface was highly successful. AMULET2e was running code within a few hours of its arrival, unlike AMULET1 which took a month to bring into life. The device also appears to be very robust.

Only one hardware fault has been identified so far. The device fails by deadlock if the BTC and aborts are enabled at the same time under certain interrupt conditions. Since most embedded applications make no use of aborts this problem is easily avoided. We don't yet know the exact diagnosis of the problem, but it is under investigation.

6.1. Performance

The fastest mode of operation is to run a program from internal RAM. TimeMill simulations predicted 68 kDhry–stones (2.1) in this case, and our first measurement was 69 kDhrystones. This constitutes remarkably accurate modelling on the part of the simulator, upon which we depended totally for the final verification of the design. When all the performance features were turned on this increased to over 74 kDhrystones (42 MIPS based on the Dhrystone 2.1 benchmark). These measurements are at 3.3 V, the nominal operating voltage of the device. Run-

	ARM710	AMULET2e	ARM810
Process	0.6µm 2LM	0.5µm 3LM	0.5µm 3LM
Area mm^2	32	41	76
Transistors	570,295	454,000	836,022
Cache	8 K 4-way	4 K 64-way	8 K 64-way
MIPS	23	40	86
Conditions	3v3 25 MHz	3v3 20 deg.C	3v3 72 MHz
Power mW	120	150	500
MIPS/W	192	250	172

Table 1: AMULET2e characteristics

ning at this peak rate it consumes just under 150 mW (excluding I/O power, but there is very little I/O activity). On similar process technologies the ARM710 delivers 23 MIPS at 120 mW and the ARM810 86 MIPS at 500 mW (see table 1), so the AMULET2e performance falls between these two with slightly better power-efficiency (though the ARM figures do include I/O power).

Running from external 80 ns static memory with the cache enabled the processor delivers 45 kDhrystones (26 MIPS), and with the cache disabled 17 kDhrystones (10 MIPS).

Multiplier speed. AMULET2 includes a 4-bits per cycle multiplier [15] which uses data-dependent early termination. The multiplier cycle time (measured by varying the operand values) is around 6.5 ns, demonstrating the merits of allowing different functions to operate at different speeds. (The main execution pipeline cycles in about 25 ns.) A multiplier constrained to operate at the same cycle rate as the execution pipeline, such as a multiplier in a clocked processor, would require four times as many Booth's encoders and carry-save adders to deliver the same performance. This illustrates the fact that, under certain circumstances, asynchronous design can save hardware cost.

Low voltage operation. The processor appears to tolerate supply voltage variations well. The whole board operates between 2.5 V and 4 V, but failures of other components stop the board operating below 2.5 V. In tests which do not involve the other components the chip appears fully functional down to 2 V, at which point the I/Os stop working. The core appears to continue operating down to 1.1 V, though this can only be surmised by taking it down from and back up to 2 V, since below 2 V the I/O failures prevent external observations from being made.

The performance and power-efficiency scale with voltage according to the standard formulae[2].

EMC. Further work is required in the important area of EMC (Electro-Magnetic Compatibility) measurement. This is pending the availability of suitable facilities.

Memory:	Internal	Cache	External
Idle power	76 mW	162 mW	66 mW
Idle power, 'Halt' on	0.1 mW	0.1 mW	0.1 mW
BTC MIPS	+6%	+3%	+7%
BTC power-efficiency	-5%	-3%	-4.5%
Fwd MIPS	+2%	+0.5%	0%
Fwd power-efficiency	0%	0%	0%
All features MIPS	+8%	+4%	+7%

Table 2: Architecture feature results

6.2. Benefits of architectural features

The various architectural features described in Section 3 were justified on the grounds of their contribution to performance, power-efficiency, or both. It is interesting, therefore, to see what effect they have on the prototype silicon. This is particularly straightforward as they can all be enabled and disabled under software control. A summary of the measurements taken on the test card is given in table 2. All the measurements were taken running the Dhrystone 2.1 synthetic benchmark program, with the program either resident in the internal memory, cached from the external memory or executed directly from the external memory. The external memory is 80 ns 32-bit static RAM and the power figures are for the AMULET2e core logic including the cache/RAM but excluding the I/O pads.

The last line in table 2 shows the performance benefits of turning all the architectural features on together.

Last result and last loaded value registers. The mechanisms used on AMULET2 to perform forwarding (grouped together under 'Fwd' in table 2) were the outcome of considerable development effort and their contribution to the performance of the device cannot be described as anything other than disappointing. The LRR contributes about twice the benefit of the LLV, but even taken together the net result is meagre. This clearly demands some explanation.

The AMULET2 forwarding mechanisms exist to reduce the time an instruction waits for a stalled register read to be resolved. If an instruction never stalls because the register bank is fully updated by the time it attempts the read, there is no scope for forwarding. This is the case when some other pipeline stage limits the instruction issue rate to below that required by the execution stage to complete register write-back. For example, external memory in the test card is slow relative to the processor cycle time so when it is used as the source of instructions there is no performance gain from enabling the forwarding paths. As the instruction source gets faster, gains appear. The cache demonstrates a (very) small gain from forwarding, and the internal RAM (which is faster than the cache) a little more.

Our conclusion is therefore that the forwarding mechanisms on AMULET2 are capable of delivering more than the rest of the AMULET2e system can expose. We know that the AMULET2 execution pipe can cycle in 20 ns but the rest of the chip can only sustain an average of 25 ns. With a faster decode stage, address interface and memory we would expect considerably more benefit from the forwarding paths in the existing execution pipeline.

The next question which must be answered is to explain why this result was not evident during simulation. The answer lies in the critical dependency of the effectiveness of the forwarding mechanisms on the detailed timing of all parts of the chip. Early high-level simulations which were used to define the architecture were based upon approximate timing estimates. Accurate timings only became available during post-layout simulation using 'TimeMill', by which time the architecture was frozen.

A conclusion is that it is hard to optimise an asynchronous architecture early in the design process. This could be turned into a case against asynchronous design. However, there is an alternative view. The reason forwarding has so little benefit on AMULET2e is that the execution pipeline is running ahead at its own speed, making more progress (relative to other parts of the design) than was anticipated. A clocked pipeline would be held back by the clock, thus ensuring that forwarding had the expected benefits, but at the cost of preventing part of the system from going as fast as it can. Is this really better?

Branch target cache. The BTC caches 20 branch targets, a number which was chosen on the basis of extensive simulation of a range of applications. Unfortunately, it became clear during those simulations that Dhrystone, the benchmark program used for most our other measurements, has very unusual branch characteristics - it is basically a single large loop with over 20 branches in it. This would suggest a BTC of at least 24 entries is needed to optimise Dhrystone performance. We resisted the temptation to optimise the architecture for this synthetic benchmark. Whereas the BTC gives a performance improvement of around 10% on typical programs, it delivers only 7% on Dhrystone. The power-efficiency of the core drops by 5% when the BTC is turned on, though the overall system power-efficiency rises by 4% when the code is being executed from external memory due to the reduction in wasted instruction fetches.

The BTC therefore performs largely as expected and makes a useful contribution to system performance and power-efficiency.

Halt. When AMULET2e enters an idle loop without the 'Halt' function enabled it consume between 66 mW and 162 mW depending on how fast the memory system allows the processor to access instructions. With the 'Halt' function enabled the power drops to under 0.1 mW if the 32 KHz oscillator is running (and 3 μW if it isn't). The 'Halt' feature therefore delivers a three to four orders of magnitude power saving during idle periods, automatically and in a way which works with much existing code (including the 'Demon' ROM code used in these tests). In many systems, when the processor halts the external system power will also drop to very low levels. The power consumption of the test card is dominated by a single LED when the processor is halted.

A clocked system can approach this idling efficiency, but only with considerable effort. The clock must be gated off to all parts of the system that consume significant dynamic power, but in a way that leaves interrupts enabled, and an interrupt must gate the clock back on. Often power management software is used to detect idle periods and to step the power consumption down through a progressive series of stages, reducing clock frequencies and gating out particular parts of the system. Power management software consumes power itself when running. For very low consumption the oscillators and phase-locked loops must also be turned off. Stopped oscillators and PLLs take considerable time to stabilise when they are turned back on, compromising response time when an interrupt occurs. Overall, power optimisation in a clocked system is a complex matter involving many trade-offs.

In an application with a significant proportion of idle time, AMULET2e should display remarkable power-efficiency with very little effort on the part of the designer.

At first sight it might appear that having only two states (maximum performance or halted) is not as flexible as being able to step a clock down through lower frequencies. However lowering a clock frequency does not, of itself, improve power-efficiency. It takes the same energy to perform a given calculation in half the time and to halt for the remaining time as it does to perform the calculation at half the clock rate for the whole time. The only way a lower clock rate can improve power-efficiency is if there is a corresponding reduction in the supply voltage, which is rarely used in practice and can equally (and more easily, as there is no clock to adjust) be applied to an asynchronous system.

An additional benefit of the halted state of AMULET2e is that, by stopping all activity, it also removes all sources of electromagnetic interference. Although we fully expect tests to show that AMULET2e has good EMC properties when running under maximum load, these can be further improved by halting the processor when 'radio silence' is required. There is no high-speed clock oscillator continuing to generate interference, and maximum performance is available instantly after an interrupt. This technique has potential benefits in many modern time-division multiplexed digital radio communication systems, where in weak signal areas the processor can be shut down during the time slot used to receive each packet.

7. Conclusions

AMULET2e is a highly usable asynchronous embedded system chip. Its performance and power-efficiency are competitive with the industry-leading clocked ARM designs, and in an idle loop its power reduces below that achievable in a clocked design without stopping the clock (whereafter the clocked chip takes considerable time to resume full performance). Its EMC properties are unknown at present, but may demonstrate another meritorious aspect of asynchronous design.

AMULET2e incorporates several new architectural features and the means to evaluate them (the LRR, LLV, BTC and abort handling can all be turned on or off independently). As such it contributes to the growing pool of architectural knowledge which must expand considerably further before asynchronous designers can compete with synchronous designers on equal terms.

However, seeing is believing, and the reactions of the systems designers who have seen AMULET2e since prototypes first ran have been very favourable. Several prototype applications for the chip are presently under development. We sense that the barriers to the commercial exploitation of asynchronous design are beginning to fall.

8. Acknowledgements

The development of AMULET2e has been supported as part of ESPRIT project 6909, OMI/DE-ARM (the Open Microprocessor systems Initiative - Deeply Embedded ARM Applications project). The authors are grateful for this support from the CEC.

The authors are also grateful for support in various forms from Advanced RISC Machines Limited and for the fabrication service supplied by VLSI Technology, Inc. The chip was designed using EDA tools from Compass Design Automation, with accurate simulation using TimeMill from EPIC Design Technology, Inc., being vital to the final veri–fication of the design.

9. References

[1] Paver, N.C., “The Design and Implementation of an Asynchronous Microprocessor”, PhD Thesis, Dept. of Computer Science, Manchester University, England, June 1994.

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

[3] Sutherland, I.E., “Micropipelines”, *Communications of the ACM*. Vol. 32, No. 6, January 1989, pp. 720-738.

[4] Furber, S.B., Day, P., Garside, J.D., Paver, N.C. and Woods, J.V., “A Micropipelined ARM”, Proceedings of the 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. Pub. North Holland.

[5] Furber, S.B., Day, P., Garside, J.D., Paver, N.C. and Woods, J.V., “AMULET1: A Micropipelined ARM”, CompCon ‘94, San Francisco, March 1994.

[6] Furber, S.B., Day, P., Garside, J.D., Paver N.C. and Woods, J.V., “The Design and Evaluation of an Asynchronous Microprocessor”, *Proc. ICCD'94*, 217-220.

[7] Furber, S.B., “Computing without Clocks: Micropipelining the ARM Processor”, in Birtwistle, G. and Davis, A. (eds.) *Asynchronous Digital Circuit Design*, Springer, ISBN 3-540-19901-2, 211-262.

[8] Paver, N.C., Day, P., Furber, S.B., Garside, J.D. and Woods, J.V., “Register Locking in an Asynchronous Microprocessor”, *1992 IEEE International Conference on Computer Design: VLSI in Computers & Processors*. October 1992.

[9] Garside, J.D., “A CMOS VLSI Implementation of an Asynchronous ALU”. *IFIP Working Conference on Asynchronous Design Methodologies*, April 1993. Ed. Furber, S. B. and Edwards, M. D. Pub. North Holland.

[10] Furber, S.B. and Day, P., “Four-Phase Micropipeline Latch Control Circuits”, *IEEE Trans. on VLSI Systems* **4**, 247-253, 1994.

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

[12] Garside, J.D., Temple, S. and Mehra, R., “The AMULET2e Cache System”, *Proc. Async'96*, Aizu-Wakamatsu, 208-217.

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

[14] AMULET2e datasheet, Dept. of Computer Science, Manchester University, England. Available by anonymous FTP via http://www.cs.man.ac.uk/amulet/AMULET2e_uP.html

[15] Liu, J., MSc Thesis, Dept. of Computer Science, Manchester University, England, October 1995.