The AMULET2e Cache System

J.D. Garside, S. Temple, R. Mehra

Department of Computer Science, The University of Manchester,
Oxford Road, Manchester, M13 9PL, U.K.
jgarside@cs.man.ac.uk, stemple@cs.man.ac.uk, rmehra@cs.man.ac.uk

Abstract

AMULET2e is an asynchronous microprocessor system based on the AMULET2 processor core. In addition to the processor it incorporates a number of distinct subsystems, the most notable of which is an asynchronous on-chip cache. This includes several novel features which exploit the asynchronous design style to increase throughput and reduce power consumption.

These features are evident at a number of levels in the design. For example, the cache is micropipelined to increase its throughput, at the architectural level there is an arbitration free non-blocking line fetch mechanism while at the circuit design level there is a power-saving RAM sense amplifier control circuit.

A significant property of the cache system is its ability to cycle in a data dependent way which allows the system to approach average case performance.

1. Introduction

The performance of modern RISC microprocessors is typically limited by their memory system and it is usual to improve this by providing a memory hierarchy which includes some form of high-speed cache. For an asynchronous microprocessor it is logical that the cache should be asynchronous too. It should also be borne in mind that current commercial memories present a synchronous interface and the asynchronous microprocessor must be able to interface to these devices at some level.

AMULET1 [5], [11] demonstrated that a commercial microprocessor can be implemented in a completely self-timed fashion. Interfacing AMULET1 to conventional memories was difficult however, and its performance would be greatly enhanced by providing an on-chip cache. Thus, the AMULET2 project entailed not only enhancements to the AMULET1 core but also the design of an asynchronous on-chip cache and a memory interface which would connect directly to commodity memory devices. The resulting chip is intended as an embedded microprocessor and is designated AMULET2e (figure 1).

Although a basic asynchronous memory system is not inherently difficult to design there are some interesting problems to solve. More importantly a number of features of the self-timed design provide advantages over a rigorously cycled, clocked system. The design which has evolved has also provided the opportunity to include a number of power saving mechanisms.

The resulting design is quite simple in operation but provides a number of advanced features. It includes asynchronous pipelined operation, a data dependent cycle time, and a non-blocking, arbitration free cache line fetch mechanism. The microarchitecture is based on a bundled data micropipelined [13] system with four-phase control circuits [3]. It has been implemented using retargetable 0.5µm design rules with three layer metal and characterised for 3.3V operation.

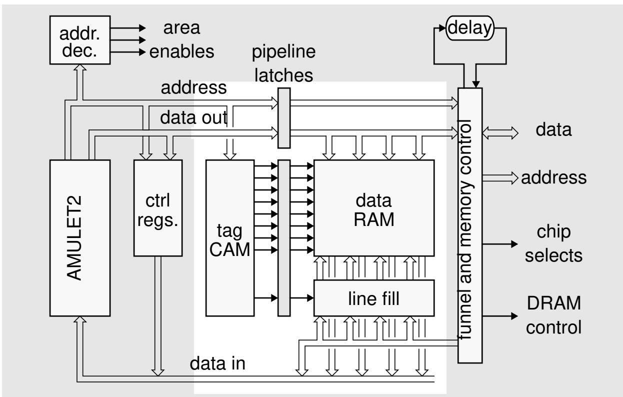

Figure 1: AMULET2e

2. Basic cache architecture

The AMULET2e cache is built from a number of independent 1 kilobyte blocks (figure 2) selected according to the memory address. This allowed the final decision about

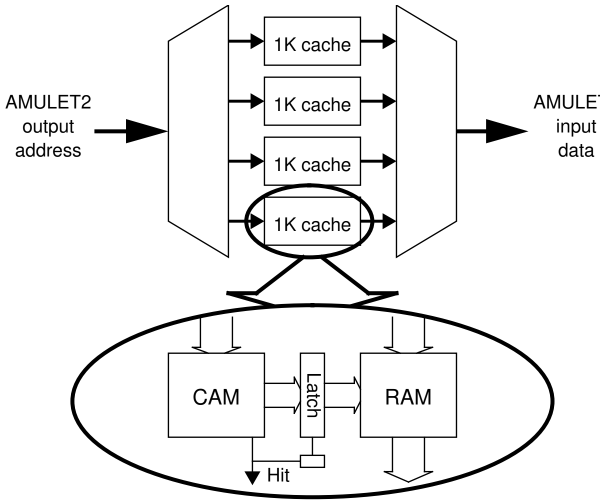

Figure 2: AMULET2e cache block structure

the overall cache size to be deferred until the chip was being assembled; it also allows the cache to be adapted and reused easily in future designs. Within each cache block (set) the architecture is fully associative with 64 cache lines each of four words. In AMULET2e four blocks are employed; it therefore has a 4 Kbyte cache which is 64 way associative.

A “write-through” [6] scheme is employed, where data is always sent to the external memory as well as updating the cached copy if present. A “write around” policy [6] is adopted, i.e. no cache allocation is performed for write operations which miss the cache.

Cache lines are selected for replacement using a pseudo-random number generator. This gives a ‘graceful’ degradation in performance as the size of the working set increases, making performance measurements less subject to pathological fluctuations. It is also easier to implement than a least recently used (LRU) algorithm.

Fundamentally this is the same as the ARM3 and ARM6 series cache architecture. For a relatively small cache this has been demonstrated to give significantly better hit rates than a cache of lower associativity [4]. This choice also allows a fairer comparison of other factors with existing synchronous ARM systems.

Internally each cache block is an independent unit; one block is selected on each memory cycle according to an appropriate set of address bits. The remaining sets remain quiescent. Within each cache block is a self-timed micropipeline. This has two stages: the first stage being a CAM look-up and the second a RAM access which is conditional on a cache hit (figure 2). In cases such as a cache miss the RAM is not activated directly and the CAM stage is able to invoke the common external memory interface. This process is described later.

Constructing the memory from relatively small blocks is beneficial in that it reduces internal capacitance and so increases the speed of the block whilst also decreasing its power consumption. The price is that of a small added decoder stage with its resulting impact on the access time.

The cache is unified in that instruction and data references are mixed before leaving the processor core. Studies of such caches on ARM processors yields a read hit rate of around 95% [10].

3. Self-timed memory accesses

With the high degree of associativity used in the AMULET2e cache it is sensible to use a special purpose CAM for the cache “tags”. This means that there are two self-timed blocks within the system (CAM & RAM).

It would be impractical to build a large regular structure such as a memory which included delay insensitive self-timing. However an accurate timing model must be produced which is capable of adapting to the operating conditions of the circuit as well as manufacturing process fluctuations which may yield devices varying by up to a factor of four in speed. This also increases the process portability of the resulting design. As in other parts of the AMULET devices, timing has been done by including extra, dummy bits within the custom logic of the datapath. These have the same layout as the real data lines and are as analogous as possible in every sense.

3.1. CAM timing

The Content Addressable Memory (CAM) presents a particular problem in that it requires two separate, serial self-timed processes. This is necessary because the memory array contains distributed high fan-in gates which must be implemented using dynamic CMOS logic. The first of these gates determines whether any bit in a given CAM line does not match the presented address. If this is the case then that CAM line has “missed”. When operating it is normal for either all or (hopefully) all but one CAM line to miss.

It is also necessary to know whether any of the CAM lines did hit to determine whether or not to proceed with a RAM cycle. Because this is also performed by a distributed dynamic gate – in this case with a fan-in of 64 – this is only possible once the status of the individual hit lines is resolved. It is also necessary to know when this second process is complete so that the CAM cycle can be terminated.

The CAM array (figure 3) therefore includes two timing signals. The first is a dummy CAM line. This is an analogue of the normal CAM lines except that it is guaranteed to “miss” with a single bit mismatch. The first of these conditions ensures that the dummy line will produce an output

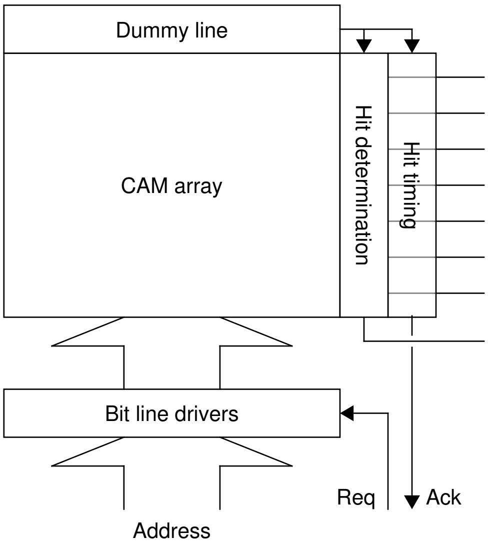

Figure 3: CAM timing mechanism

transition, the second that this models the worst case timing for any of the true CAM lines. At this point the status of the individual hit lines is known, but not the overall hit state.

The output of the first timing signal activates the remaining circuitry. This comprises two distributed gates, the first being used to determine the overall hit state, the second being a timing model for this. The timing output includes a small extra delay which guarantees that when it makes its transition the hit signal will already have done so if the cache has hit.

In some cases a third process is also invoked. This is when the cache has missed and a cache line is to be replaced. In this case the CAM is also written to.

CAM writes only occur following a CAM read. The (previously failing) address is already present and there is no need to charge/discharge large, capacitive wires, so write cycles are quite rapid. Rather than expending time and power in an attempt to detect write completion it is easier to produce a pulse with a separate timing circuit to control the write. This is longer than the worst case write time, but is still more efficient than attempting to monitor the write operation directly. As CAM writes only occur when a cache line is allocated or reallocated and are therefore infrequent it is not important to optimise this process.

3.2. RAM timing

The RAM self-timing is rather simpler than that of the CAM in that only a single process – either a read or a write – is invoked in any given cycle. The write process is almost identical to that in the CAM and is modelled in a similar

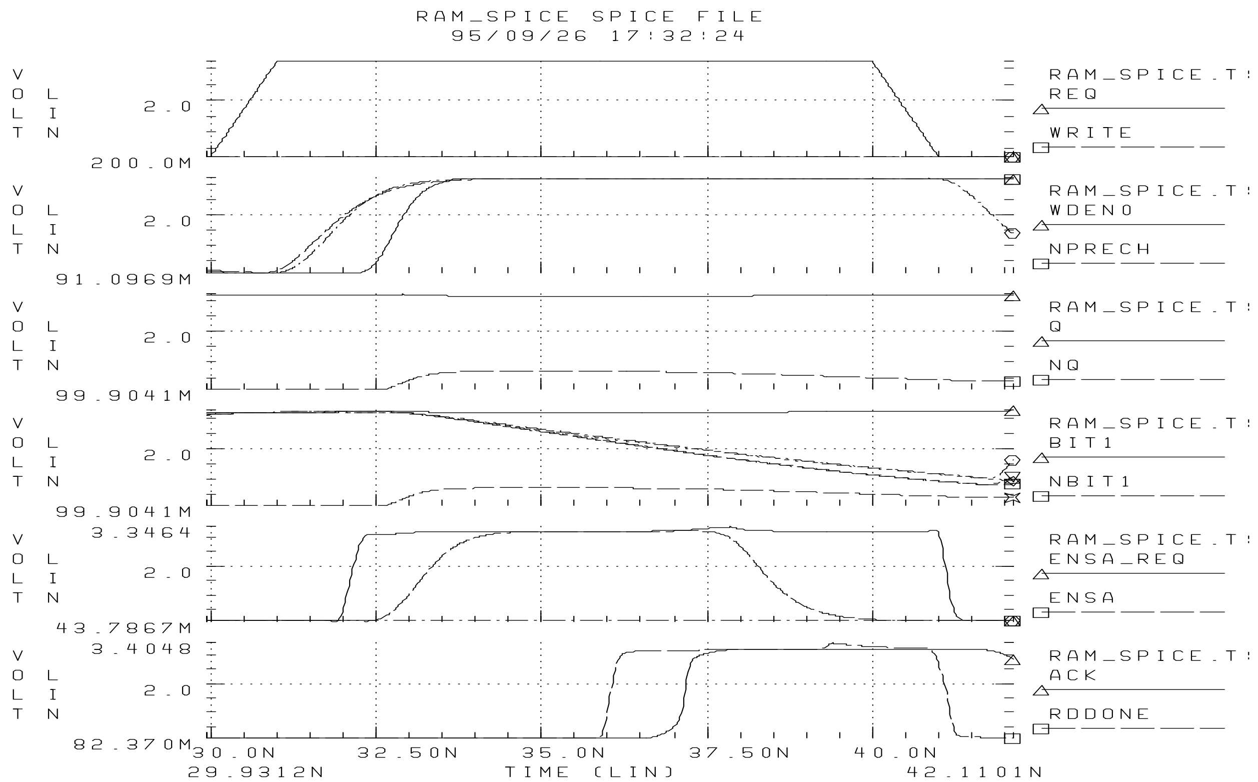

Figure 4: RAM read timing

way. However, due to the nature of RAM design, the read process is inherently quite slow and deserves further attention.

The RAM array is one cache line or 128 bits wide. The addition of a 129^{th} (dummy) bit is therefore a small extra imposition. This bit is modelled closely on the other RAM bits, but includes a small extra delay in order to guarantee functionality.

Figure 4 illustrates a RAM read cycle simulated using SPICE. The first panel shows the external request being applied and later removed to prepare for the next cycle. Below this are its internal reflections on the $\overline{\text{precharge}}$ and the selected word line. The stored bit (third panel) discharges the bit line (fourth panel) and the dummy timing bit is also discharged in parallel.

The sense amplifier timing (panel 5) is described below; the last panel shows the internal read completion signal and the final acknowledgement that the RAM read is complete. The time from request to acknowledge is around 7ns for this cycle.

4. Self timed sense amplifiers

The primary design consideration in memory circuits is to pack the maximum number of bits into the smallest possible space. To achieve this the individual memory cells are constructed from small transistors which must drive large, capacitive loads. This, in turn, means that reading the memory can be a lengthy process as it takes the memory cell a considerable time to drive its output bus as shown in figure 4.

As slow access times are unacceptable in high performance circuits it is desirable to produce an early response from the memory. To do this circuits known as sense amplifiers are used to detect small changes in voltage and to produce full-rail CMOS levels before the RAM cell itself could do so. This is clearly seen in figure 4, where the output has been determined before the bit line has been discharged to more than a third of its initial voltage. A further advantage in this case is that, because the bit lines do not need to be fully discharged before a decision is made, power can be saved on cycling these numerous and highly capacitive wires.

Unfortunately the sense amplifiers must be analogue circuits, drawing power continuously while they are enabled, unlike digital CMOS which draws significant current only when switching. This is a considerable penalty which may be alleviated by activating the sense amplifiers only when they are needed.

In the AMULET2e cache the sense amplifier control is self-timed. Activation of the sense amplifiers is deliberately delayed because there will be an initial period during which the memory state is not determinable (figure 4, signal “ensa”). This delay is imposed by a chain of gates. If this delay is too long the sense amplifier enable will be delayed and the memory access time increased; if too short extra power will be wasted. The simulated delay has therefore been measured (SPICE) and set as being slightly shorter than optimal to prevent impact on the access time.

In order to determine when the output of the sense amplifiers is valid an identical sense amplifier is attached to the dummy bit. This always makes a transition (called “RdDone”) which signals that the read data has been resolved. When this has occurred the data output is latched (within the sense amplifier) the output’s presence is acknowledged and, simultaneously, the sense amplifiers are disabled. Thus the sense amplifiers are on for (just over) the minimum possible time. The (digital) output of the sense amplifier may be held indefinitely without further power consumption, necessary because the time taken for the processor to accept the proffered data is not determinable.

5. Exploiting sequentiality

A major feature of the AMULET2e cache is its ability to exploit the generally sequential nature of memory references. For the ARM architecture – which AMULET emulates – typically 85% of memory references are instruction fetches [8] and the majority (75%-80%) of these are sequential. This means that a high proportion (>50%) of addresses generated refer to the same cache line as their predecessor.

This is easy to exploit since AMULET2 generates sequentiality information in its address interface. If a cycle is known to be in the same cache line as its predecessor then its hit status and, in the case of a hit, the cache address of the data is already known. In this case the associative look-up in the CAM need not be performed and the previously used RAM line can remain selected, thus avoiding a slow and power consuming precharge-discharge cycle in each of these blocks. It also means that “sequential” cache cycles can be significantly faster than their “random” counterparts.

In sequential cycles the action of the timing circuitry is the same as in “random” cycles, except that the initial delay in enabling the sense amplifiers is omitted. As the data will be ready to be read immediately, the sense amplifiers are switched on as soon as possible and rapidly switch off again (see figure 5).

Figure 5 displays certain implementational artifacts. For example a local precharge between the cycles is visible. This is present to avoid potential charge sharing problems as the different output words are multiplexed; it causes a fairly insignificant ripple in the bit line discharge which is, nevertheless, modelled by the timing signal.

Sequential access not only allows faster memory cycles but it saves power too. Firstly the analogue sense amplifiers are enabled for a shorter time; secondly the precharge cycle of the RAM bit lines is omitted when it is known that the following cycle is going to be sequential and in the same cache line.

This feature could be applied to any cache, synchronous or asynchronous. However in a synchronous cache the time of any cycle is effectively limited to an integer multiple of the clock period, and without very fast clocks it is difficult to effectively exploit the speed advantage of this property. Asynchronous circuits adapt quite naturally to this situation however and can provide average case performance.

5.1. Average case performance

A frequently used argument in favour of asynchronous systems is that they are capable of being optimised for an "average" case performance rather than the worst case which synchronous systems must be designed for. Unfortunately, due to complex and often data dependent timing interactions between subsystems this "ideal" cannot always be achieved.

If an average case throughput is to be achieved in a given system over a given period of time it must be free of external timing constraints over this period; thus it must never be starved of input data or blocked because of a backlog of output data. In many systems this is difficult to guarantee.

The cache described above exhibits the requisite properties needed for average case performance. It has a variable, data-dependent cycle time; however the coherence of the input data stream ensures that a slow, non-sequential cycle is frequently followed by one or more faster, sequential cycles. This means that the period needed to demonstrate an "average" performance is of the order of a few memory cycles. The cache input is supplied by the proces-

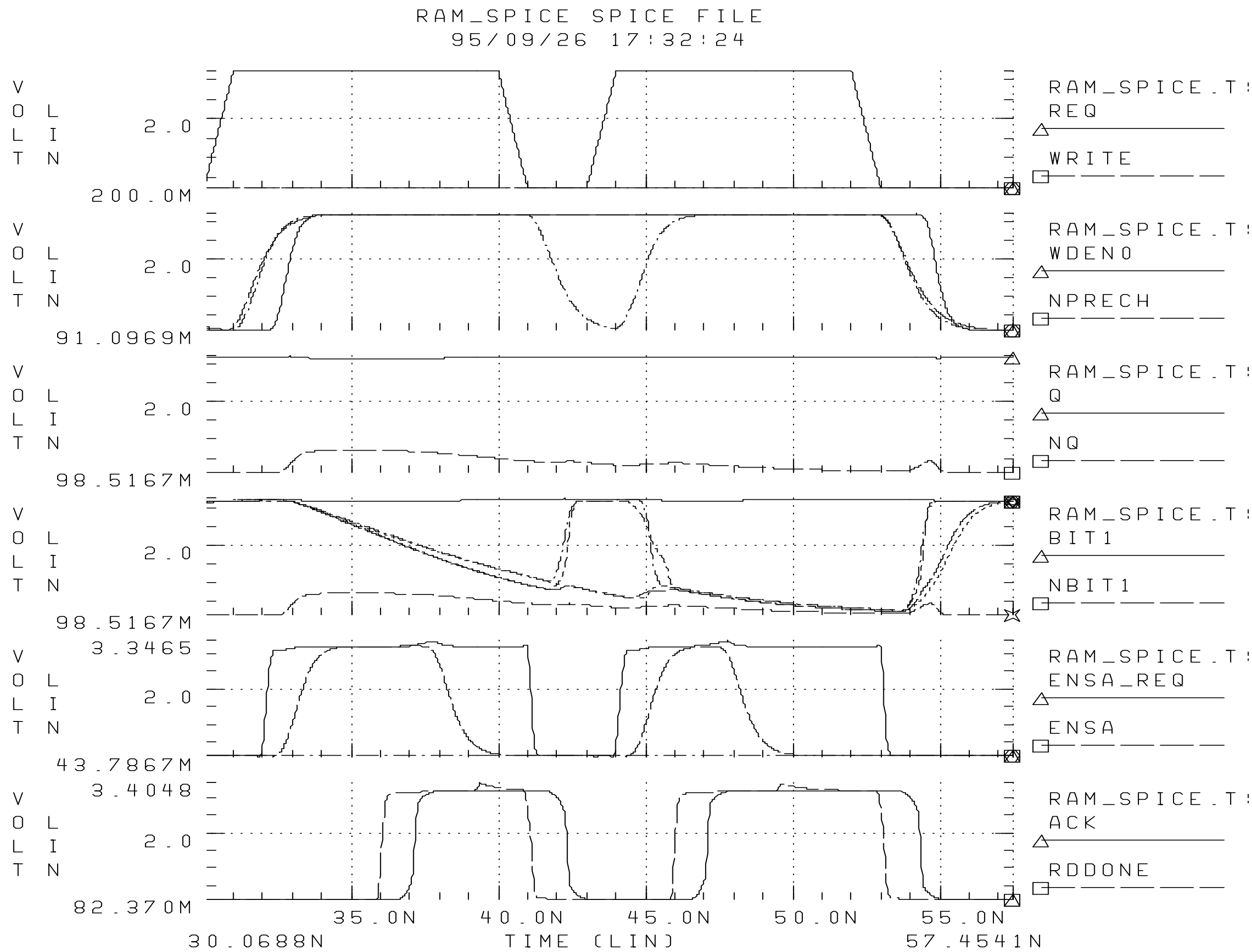

Figure 5: Sequential RAM read cycles

sor memory interface which is a simple circuit capable of cycling somewhat more rapidly than most parts of the processor, thus input starvation should be relatively rare. The output of the cache is the processor's data interface where the majority of returned data are routed to the prefetch buffer. This is a fast, elastic buffer which is made sufficiently large to prevent it filling up very often.

6. The Line Fetch Mechanism

In addition to efficiently returning data items which it has cached, the cache system should also attempt to perform effectively when a cache miss occurs. In AMULET2e, a write cycle which misses causes no action in the cache. A read however causes a line fetch to begin and four words are fetched from the (slower) external memory system. Although infrequent this can still impose a major performance penalty, and measures must be adopted to alleviate this delay.

The simplest mechanism to adopt whilst filling a cache line is to stall the processor until the fetch is complete. When the line fetch finishes the requested word is sent to the CPU and processing allowed to continue. This solution is commonly used in mid-range microprocessors (e.g. MIPS-X [7], Intel 486 DX2 [14]). It also has the advantage that the various processes are kept in strict sequence, which can simplify an asynchronous implementation. It is clearly sub-optimal however, as it would require (in this case) the time for four memory cycles to be added to the processing time for each cache miss.

Ideally what is required is that the line fetch starts by fetching the required word, forwards this to the processor and then continues autonomously to complete the line fill. The processor is then free to continue in parallel.

This is known as early-restart [12] and is a technique employed in high performance architectures (e.g. RS/6000-560 [15]).

The parallelism introduced by this mechanism can cause problems when the processor requests its next memory cycle. To illustrate this the cycle following the miss can be classified as follows:

1) A write operation.
2) A read operation which may be satisfied from the existing cache contents.
3) A read operation which is a cache miss and requires another line fetch.
4) A read operation on a value which is in the last line to be requested.

Once it has been determined that a write operation is not going to abort it can safely be buffered and dealt with at leisure. Many modern cache architectures contain a write-buffer [6]; in an elastically pipelined system this is almost automatic. In general this allows the write operation to be delayed without impact on the processor's performance. Read operations are more interesting in that they must return results to the processor, preferably with as little delay as possible.

Providing that the line fetch process and the processor do not conflict over cache accesses it is clear that case (2) may – in principle – proceed without impediment. This process – known as hit under miss [9] – is sometimes employed in current synchronous designs although many systems resolve potential conflicts by stalling the processor until the line fetch is complete.

It is apparent that case (3) requires its own cache miss processing and line fetch. If a line fetch is still in progress there will be contention for the external bus. In this case either the current line fetch must be abandoned or the subsequent request stalled until it is complete.

In situation (4) various options are available. One method, known as streaming, is to allow the read request to be synchronised with the incoming line fetch data. When the required value is available (the request might have to wait) it is forwarded to the processor. Another, simpler, option is to stall the request, allow the line fetch to complete and then send the required data (which is now known to be present) to the processor.

6.1. An Asynchronous Line Fetch Implementation

To avoid stalling the processor for extended periods during a line fetch a mechanism for parallelising the line fetch was sought. If the asynchronous fetch strategy is to allow for concurrent cache and line fetch activity, then there must be a method by which the two processes can be synchronised when necessary (e.g. when reading data that has just been fetched). In synchronous system this can be done by counting and synchronising on clock edges, comparing the state of the subsystems when they are known to be stable. Typically in an asynchronous system an arbiter is employed. Unfortunately arbiters are potentially slow circuits which may introduce non-determinism into the system and thus it is desirable to avoid them.

The solution was to divide the cache block into two parts. The larger part is a conventional, associative cache comprising CAM and RAM arrays. Added to this is another single entry CAM/RAM pair which has been designated the Line Fetch Latch (LFL). The function of the LFL is to cache the last line which was *requested* from memory. The CAM entry is conventional in that it is written to when deciding to fetch a line from memory; the "RAM" entry is unconventional as it is filled gradually at the external memory's speed and thus may or may not be full at any given time.

When a read request arrives at the tag store it is classified as a hit in the main cache array, a hit in the line fetch

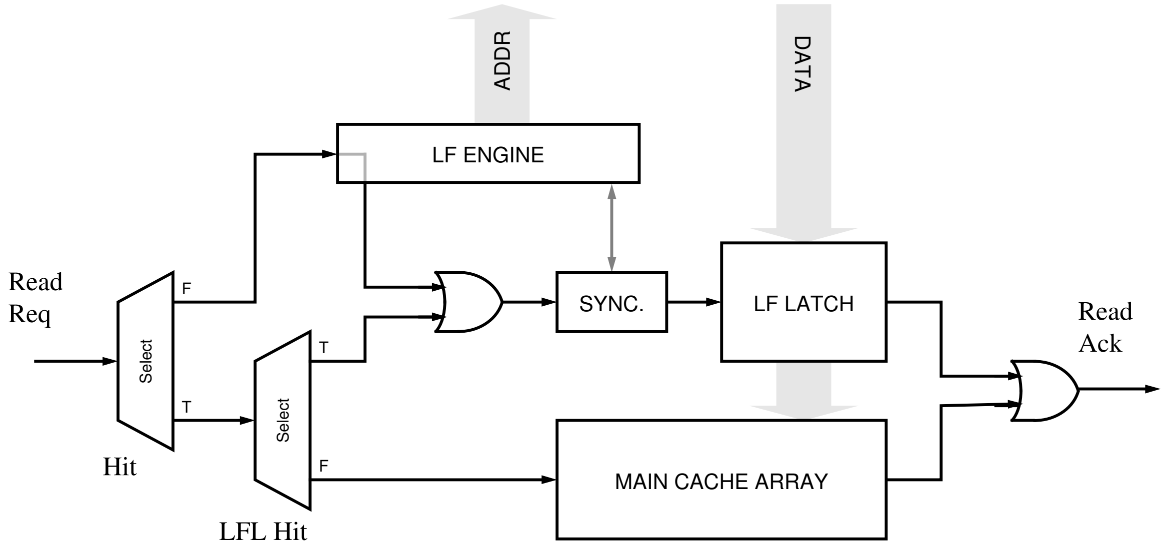

Figure 6: Control Circuit Request Steering.

latch or a miss. In the first two cases the data is read from the appropriate source and then sent to the CPU; the last must instigate a new line fetch and is thus more complex. The flow of these requests is shown in figure 6.

The “Hit” and “LFL Hit” control signals are generated by tag comparisons. “Hit” indicates that the required data is cached and “LFL Hit” specifically indicates that it is in the line fetch latches rather than the main array. “LFL Hit” is the output of the special LFL tag line. Note that a cache miss, after instigating a line fetch, looks for its data in the line fetch latches.

Assuming that the previous line fetch has completed, a read miss first instigates a new line fetch and then issues a request for the required word within the line fetch latches. It must then synchronise with the incoming data word before carrying it back to the processor. Any following read operations that require data from this line are directed solely to the line fetch latches where their data may already be present; if not they too must wait until the data arrive.

A read hit in the main array requires no synchronisation. It is routed to the main array and can proceed unhindered. This route completely bypasses the line fetch process and so may be serviced independently and at any time. There is no conflict for access to the RAM array because the line fetch process does not attempt to write into this area. The cache is thus non-blocking [16].

The key point in this mechanism is the line fetch synchronisation block (figure 7) which must delay any line fetch reads until the correct data are present. For each word in the cache line there is both a line fetch latch and a synchronisation latch. The line fetch latches are edge triggered latches which hold words that have been fetched from memory. The synchronisation latches are flip-flops which are set when a particular word is valid. These stall read requests until the required data have arrived. A request to read a line fetch latch (“LF_req”) is decoded and steered onto the correct individual word request line ($LF\_req_{0\text{-}3}$) where it synchronises with the appropriate flip-flop.

When a line fetch starts all the line fetch latches are emptied and all the synchronisation flip-flops are cleared. This occurs when the line fetch process is instigated and before the processor can attempt to read the LFL.

When the LFL receives data from memory the appropriate flip-flop is set allowing a pending request (if any) to proceed. This mechanism is non-hazardous because the process can only delay a request; no requests are sent until all the flip-flops are cleared. Synchronisation occurs in a glitch free fashion and there is no possibility of metastability.

The only further synchronisation required is between successive line fetch operations. The autonomous line fetch process is aware that a cache line fetch has completed and it can stall a request for a subsequent line fetch arbitrarily. Thus when a new read miss occurs it must wait for the preceding fetch to complete in a conventional, serial fashion.

This mechanism leaves the last line fetched in the line fetch latches indefinitely. The final task of the line fetch engine is to copy this line into the main RAM array before accepting data from the next fetch. It does this when it is both ready (LF Complete is asserted) and a new read miss has occurred, in which instance it has control of the complete subsystem. At this time the synchronisation flip-flops are cleared and the contents of the line fetch latches are copied – in parallel – into the RAM array as a standard

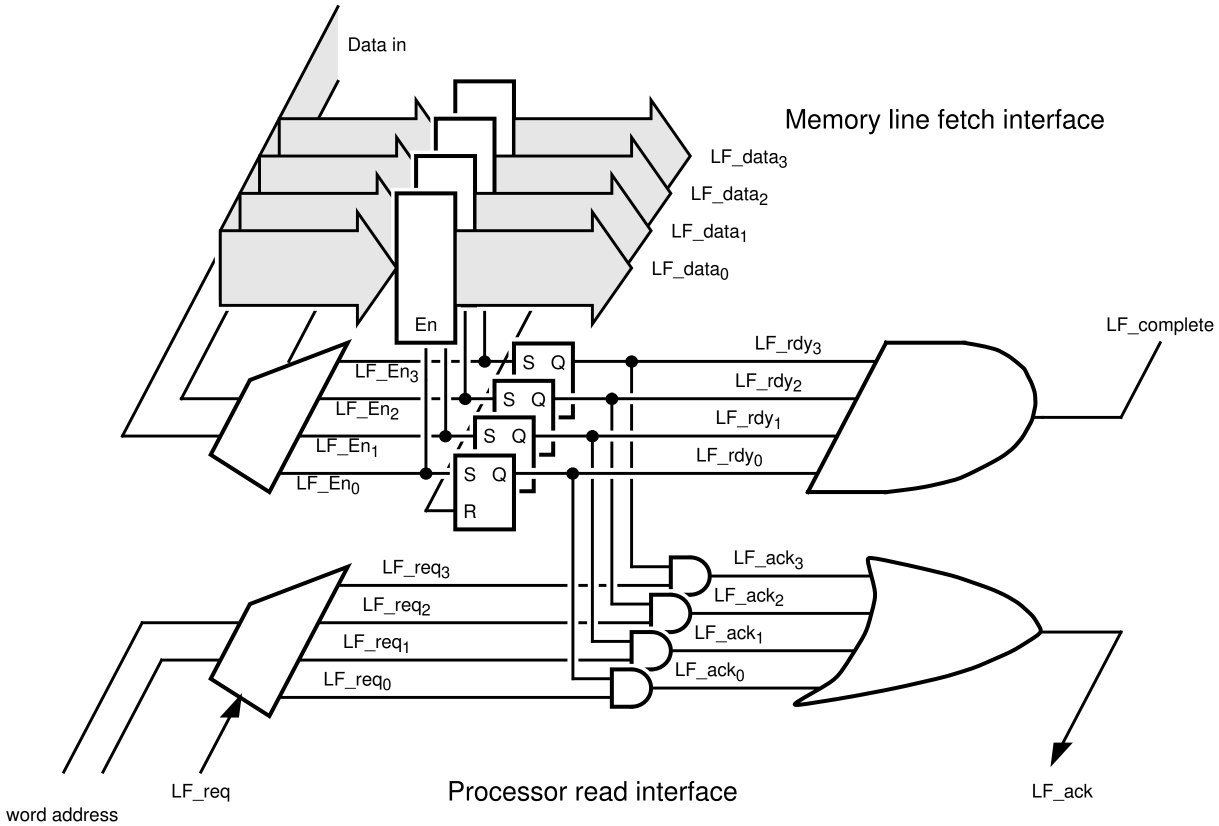

Figure 7: Line fetch latch read synchronisation

cache line replacement. This is a rapid operation and is hidden under the start-up time of the new line fetch.

6.2. Performance Evaluation

An ARM [1] instruction level simulator [2], was augmented with code which modelled the cache architectures. A selection of programs was run (single tasked) and statistics were recorded. This revealed two important points.

1) Given a cache comprising 256 lines and with a >90% hit rate it might be estimated that the chance of an access to a specified cache line would be about 0.4%. However if the specified line is the last line fetched from memory this chance is significantly higher at around 4%.

2) Perhaps surprisingly, after a cache miss has started a line fetch the subsequent number of sequential accesses is often quite small. In about half the line fetches performed only a single word was required before an access to some other cache line is requested.

The former is explicable by code fetches where sequential data is required. When a new section of programme is run many line fetches will be performed and most of the fetched words will be required. The latter appears to be attributable to data references where the processor immediately finds its next op-code already cached.

The performance gain attributable to the non-blocking line fetch mechanism described over a naive, stalling line fetch process varies according to the programme being evaluated. However the reduction in overall run time was significant and averaged approximately 15% for the cache employed in AMULET2e.

7. External Memory Interface

AMULET2e was designed to provide a simple interface to commodity memory and peripheral devices. It provides a conventional “synchronous” bus interface to which such devices may be directly attached. Support for static RAMs and peripherals which behave like static RAM is provided as is an interface to dynamic RAMs. The external data bus is 32 bits wide but it can be configured to perform multiple cycles for 16 or 8 bit wide memories.

Although AMULET2e is internally self-timed, the external memories to which it is interfaced are not. In the absence of a clock it is necessary to generate timing models for these devices. To cater for a wide range of access times for these devices, a single delay is defined by external components. Multiples of this basic delay are then used when a particular memory device is accessed. Internal control registers are used to determine the timing characteristics of various parts of the AMULET2e memory map.

8. Performance

The layout of a single 1Kbyte cache block is shown in figure 8. This block measures about 1mm x 2mm and clearly shows the CAM (left hand side) and the RAM pitch-matched together; the pipeline latch is visible in the centre. The address enters the CAM at the bottom and the dummy CAM line lies across the top of this block. The hit determination and its timing logic are on the right of the CAM block.

The RAM has two interfaces; the processor data buses lie at the bottom, together with the sense amplifiers, write drives and tri-state bus buffers. The line fetch latches are at the top of the array, together with their more numerous write drivers. The LFLs are also passed through the array to enable the processor to read them. The dummy line for timing is visible on the extreme right hand side.

At the time of writing the circuit has not been fabricated so a measured performance is not available. However detailed simulation on many of the elements has been performed and the times extracted. Times quoted are for “typical” silicon at room temperature characterised on a 0.5µm 3-layer metal, CMOS process, run at 3.3V. Some of the timings are approximate because the simulation of all the ‘high-level’ blocks was not finalised at the time of writing.

- CAM access time (hit) 8ns
- CAM precharge time 3ns
- CAM cycle time (hit) 11ns
- CAM cycle time (miss/write) 16-20ns
- RAM access time (random access) 7-8ns
- RAM access time (sequential access) 4ns
- RAM precharge time 4ns
- RAM cycle time (random access) 11-12ns
- RAM cycle time (sequential access) 8ns
- Estimated approximate overall cycle time (including estimate of control logic) 12-15ns

Unfortunately tools were unavailable to extract precise power consumption figures from the cache. However some qualitative estimates of the power saving due to sequential accesses have been attempted by extracting the capacitances of the major signals within the cache. These suggest that a sequential RAM access will consume significantly less than half the power of a random access; most of the power dissipating signals (such as the bit line precharge) are omitted. In addition – during a sequential cycle – the CAM is bypassed entirely and thus dissipates no power.

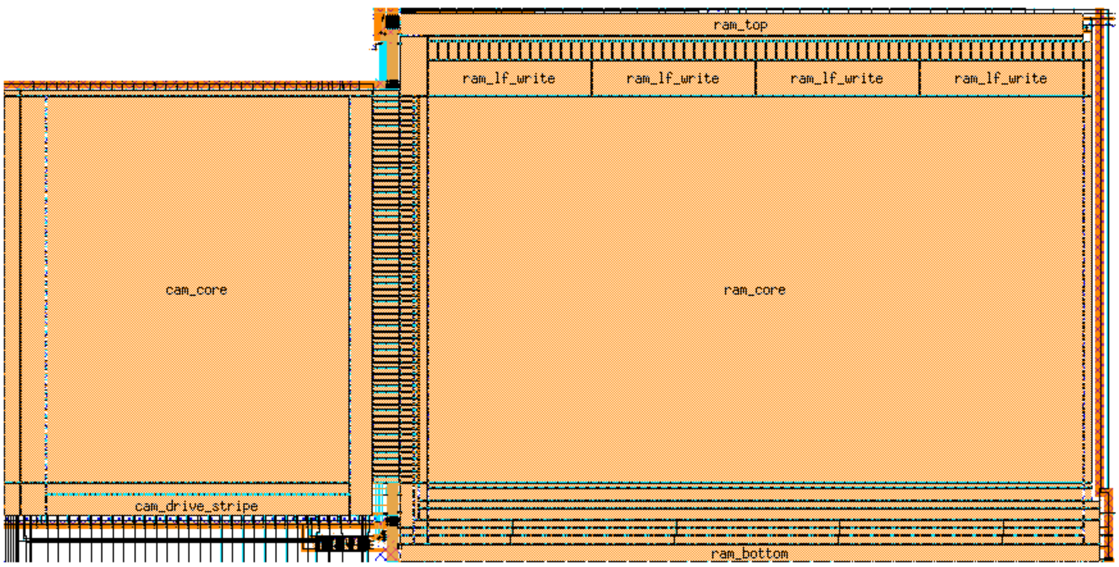

Figure 8: One kilobyte CAM/RAM cache block

9. Conclusions

We have demonstrated that a self-timed on-chip cache can be produced and readily interfaced to an asynchronous microprocessor. In some parts of the design the asynchronous style provides a distinct advantage over a conventional, synchronous approach. An example of this is the mechanism used to reduce power consumption by switching off the RAM sense amplifiers when they are known to have produced their output; this method has also been applied to ostensibly synchronous circuits in the past.

The self-timing, typically performed by using carefully matched timing signals within the full custom logic can create problems however. A good example of this is in the CAM where two serial self-timed paths must be used. This is an unfortunate source of excess power consumption as the "bundle" timed by the second of these signals comprises a single wire.

The exploitation of sequential cache accesses is a valuable mechanism for reducing power consumption. Avoiding the precharge not only saves significant power but also reduces the access time needed for subsequent cycles. This can be exploited in an asynchronous environment because even small differences in speed may be taken advantage of. This may be contrasted with a synchronous system where a speed difference must exceed a whole clock cycle before it becomes useful.

The cache therefore provides a good example of a system which provides "average case performance", an often used argument in favour of asynchronous systems. This is not just a consequence of a data dependent cycle time; it is also necessary that the environment provides suitable operating conditions. In this case cycles of different lengths are closely interleaved, there is an elastic buffer (the processor's instruction prefetch buffer) downstream and the address interface is sufficiently fast to provide a ready stream of addresses.

The cache line fill mechanism provides a great deal of flexibility in a simple manner. Despite servicing the microprocessor's requests and operating the memory interface as autonomous asynchronous processes, it allows the cache to be completely non-blocking. It is also completely deterministic and arbiter-free. Although the benefits may be less marked there is no reason why an analogous mechanism could not be employed in a synchronous system.

10. Acknowledgments

The work described in this paper was carried out 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 encouragement and support of the members of the OMI/DE-ARM consortium is also acknowledged.

The authors are grateful to Advanced RISC Machines Limited and VLSI Technology Limited for their material support. Particular thanks must go to the AMULET development team, especially Paul Day and Nigel Paver for the development of the AMULET2 processor core.

The VLSI designs described in this paper were developed using tools from Compass Design Automation Limited, and final verification used *TimeMill* from EPIC Design Technology, Inc. The quality and accuracy of these tools contributes significantly to the feasibility of the self-timed designs presented here.

11. References

[1] ARM Ltd., "ARM600 Datasheet", Cambridge, England, September 1991.

[2] ARM Ltd., "ARM Tools 200", Cambridge, England, January 1995.

[3] Day, P., Furber S.B. "Investigations into Micropipeline Latch Design Styles" IEEE Transactions on VLSI Systems, June 1996

[4] Furber, S.B., "VLSI RISC Architecture and Organization", Marcel Dekker Inc., New York, 1989.

[5] Furber, S.B., et al., "A Micropipelined ARM", In proceedings of VLSI '93, Grenoble, France, September 1993.

[6] Hennessy, J.L., Patterson, D.A. "Computer Architecture: A Quantitative Approach", Morgan Kaufmann, 1990.

[7] Horowitz, M. et al., "MIPS-X: 20-MIPS Peak, 32-bit Multiprocessor with On-Chip Cache", IEEE Journal of Solid-State Circuits, Vol. 22, No. 5, October 1987.

[8] Jaggar, D.V. "A Performance Study of the Acorn RISC Machine" M.Sc. Thesis, University of Canterbury, 1990

[9] Linley Gwennap "Intel's P6 Uses Decoupled SuperScalar Design", Microprocessor Report, Vol. 9 Number 2, February 16 1995.

[10] Mehra, R "Micropipeline Cache Design Strategies for an Asynchronous Microprocessor", M.Sc. Thesis, University of Manchester, October 1992.

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

[12] Przybylski, S., "Cache and Memory Hierarchy Design", Morgan Kaufmann, 1990.

[13] Sutherland, I.E., "Micropipelines", Communications of the ACM, Vol. 32, Number 6, June 1989, pp 720-738.

[14] Taufik, T. "Cache and Memory Design Considerations for the Intel486 DX2 Microprocessor", Intel Application Note AP-469, May 1992.

[15] Weiss, S., Smith, J.E., "POWER and PowerPC", Morgan Kaufmann, 1994.

[16] Öner, K., Dubois, M. "Effects of Main Memory Latencies on the Performance of Non-blocking Caches" Technical Report #CENG-92-14, University of Southern California, 1992.