A Cache Line Fill Circuit for a Micropipelined, Asynchronous Microprocessor

R. Mehra, J.D. Garside
rmehra@cs.man.ac.uk, jdg@cs.man.ac.uk

Department of Computer Science, The University,
Oxford Road, Manchester, M13 9PL, U.K.

Abstract

In microprocessor architectures featuring on-chip cache the majority of memory read operations are satisfied without external access. There is, however, a significant penalty associated with cache misses which require off-chip accesses when the processor is stalled for some or all of the cache line refill time.

This paper investigates the magnitude of the penalties associated with different cache line fetch schemes and demonstrates the desirability of an independent, parallel line fetch mechanism. Such a mechanism for an asynchronous microprocessor is then described. This resolves some potentially complex interactions deterministically and automatically provides a non-blocking mechanism similar to those in the most sophisticated synchronous systems.

1: Introduction

Although any properly designed cache will intercept a large proportion (greater than ninety percent [12]) of memory read requests and satisfy them at high speed, it is inevitable that there will be occasional cache misses. Such misses usually instigate a cache *line fetch* where a number of words are read from the main memory and copied into the cache. This requirement for external memory cycles can impose a significant penalty on overall performance as a complete line fetch can take an order of magnitude longer than an internal memory cycle.

When evaluating the performance of a CPU and cache sub-system it is usual to consider the overall execution time of a chosen application program. Assuming that the cache and CPU cycle at the same speed (t_{hit}) when the cache hits and the CPU is only stalled when a cache miss occurs, the execution time for a program with $N_{memreqs}$ memory requests can then be expressed by equation 1.1

$$t_{total} = N_{memreqs} \cdot (hr \cdot t_{hit} + (1 - hr) \cdot t_{miss}) \quad \text{EQ 1.1}$$

To reduce the total execution time (t_{total}) the hit ratio (hr) can be increased or the miss penalty (t_{miss}) reduced. Increasing the size of the cache or changing its structure (e.g. making it more associative) can improve the hit rate. Having a faster external memory (e.g. adding further levels of cache) speeds up the line fetch; alternatively a wider external bus means that fewer transactions are required to fetch a complete line.

Such direct methods for reducing the miss penalty are not always available or desirable for reasons such as chip-area, pin-count or power consumption. A complementary approach is to hide the time taken for line fetch operations.

This paper examines the magnitudes of the various line fetch penalties and the methods available for reducing them. It goes on to present a micropipeline [13] circuit which not only manages the line fetch problem correctly but provides several advanced features at very low hardware cost.

2: Line Fetch Strategies

Broadly speaking there are two approaches to reducing the performance penalty due to the line fetch operation. The first method is to ensure that the line fetch starts prematurely and completes before any words from the line are needed (prefetching). Failing this the penalty imposed in waiting for a line fetch may be minimised by reducing any stall times.

2.1: Reducing the miss penalty

When a cache miss occurs, the simplest procedure is to stall the processor and fetch the entire missing line starting at the lowest address. When the line fetch is complete the requested word is sent to the CPU and processing continues. This is easy to implement and is commonly used in mid-range microprocessors (e.g. MIPS-X [5], Intel 486 DX2 [14]). It is clearly sub-

TIME

The order of CPU address requests: ... Y1 Y2 X2 X3 A B C X0 X1 ...

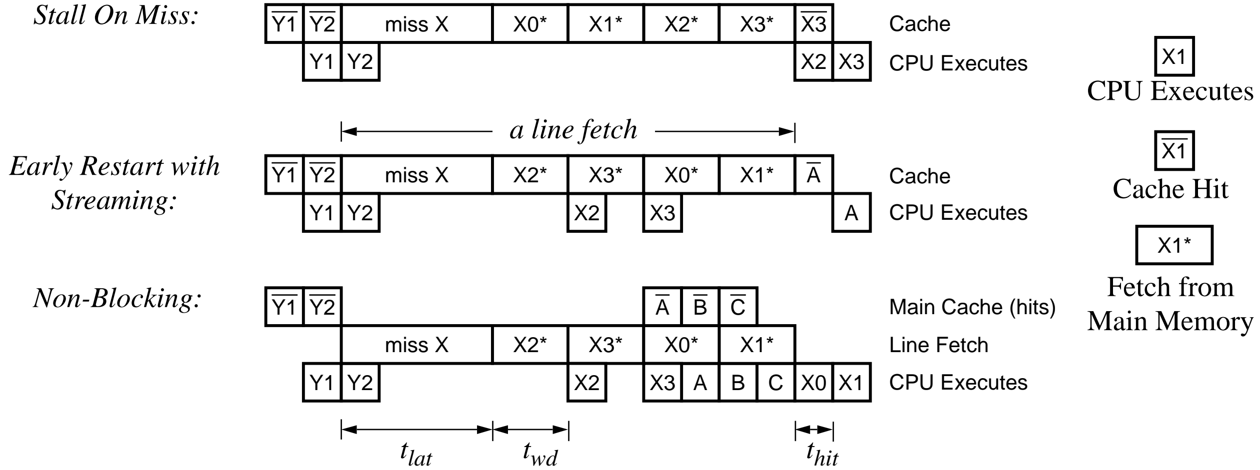

Figure 1: CPU and Cache Activity for Different Line Fetch Strategies

optimal as it requires the time for a number of memory cycles (typically four or eight [9]) to be added to the processing time for each cache miss. In more detail it means that t_{miss} is:

$$t_{miss} = t_{lat} + nw \cdot t_{wd} \qquad \text{EQ 2.1}$$

Where t_{lat} is the first word memory latency, *nw* is the number of memory transfers need to fill a line (i.e. the number of words in a line) and t_{wd} is the time taken to transfer one word.

There are many ways of alleviating this penalty. An obvious improvement is to allow processing to continue as soon as the required word is available (*early-restart* [11]). This introduces some parallelism in that the CPU may continue whilst the line fetch completes. A further optimization is to fetch the required word first – wrapping the line fetch addresses appropriately – (*desired word first/wrapping-fetch* [4]) thus reducing t_{miss} to a minimum ($t_{lat} + t_{wd}$). Both the early-restart and desired word first optimisations are employed in high performance architectures (e.g. RS/6000–560 [15]).

When employing early-restart a problem can occur when the processor attempts its subsequent memory access, in that the cache may still be fetching the remainder of the previous missed line. The next cycle may be any of the following:

1. A write operation.
2. A read operation which may be satisfied from the pre-existing cache.
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 will complete successfully (i.e. no exception occurs) it can be treated as *fire and forget*. The data can be written into the cache (or write-buffer [4]) thus hiding any further memory access penalty. For this reason write operations will be neglected for the remainder of this discussion.

It is clear that case (2) has no explicit dependency on the previously fetched cache line and such operations may – in principle – proceed independently of the line fetch process, although cache access conflicts between these processes must be resolved. Simpler mechanisms resolve any conflicts by stalling the processor until the line fetch is complete; more sophisticated systems employ *hit under miss* [6] which allows the line fetch to proceed in the background whilst the processor continues in parallel.

It is apparent that case (3) requires its own cache miss processing and line fetch. If there is contention for resources it must either abandon the current line fetch or wait until it has completed before proceeding.

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.

Options such as *early-restart*, *streaming*, *hit under miss* etc. can be combined to produce many different line fetch strategies. Three possible strategies are depicted graphically in Figure 1 and summarised below;

- *Stall On Miss*. A miss stalls the processor for the whole duration of the line fetch.
- *Early Restart with Streaming*. Desired word is fetched first and the CPU restarted early. Fetched data is streamed to the CPU as long as it requests it. If the CPU requests anything else it is stalled until the fetch completes.
- *Non-Blocking*. The previous strategy with the addition of hit under miss support. Requested data is forwarded to the processor as soon as possible in all cases thus giving the maximum degree of parallelism.

3: Simulation Results

Simulation was performed to determine the benefits of the various strategies. An ARM [1] instruction level simulator [2], was augmented with code modelling different cache architectures. A selection of programs was run (single tasked) and statistics were recorded.

The results of cache read requests were first classified into three categories; misses, hits in the main array and hits in the last cache line to be fetched. Figure 2 shows the proportions of these when a representative program (MPEG decoder) executes on one example cache architecture.

The total height of a bar gives the classically measured hit rate with the remainder representing read misses. The hit rate is divided into two parts; hits in the main array –which constitute the majority – and the smaller proportion of hits from the last line fetched.

This figure shows how both the total hit rate and proportional contribution of the last line fetched to the hit rate increase as the size of the cache line increases. It should be noted that the proportional contribution of hits from the last line fetched to the total hit rate is significantly higher than if each cache line had equal

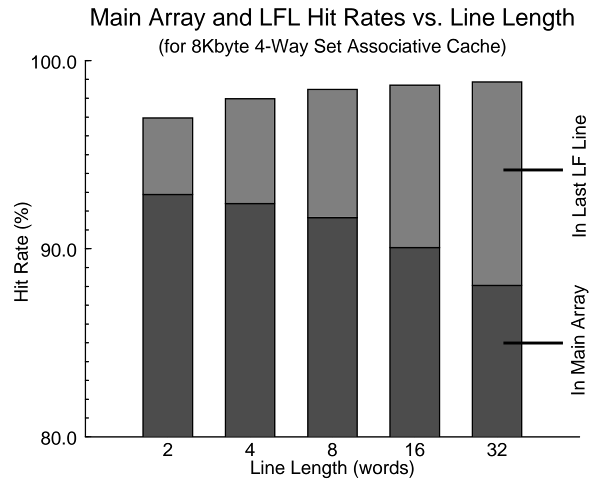

Figure 2: Hit Rate Breakdown for MPEG Decoder

importance. In an 8Kbyte, 4-way set associative cache with a 2 word line, for example, one cache line represents only 0.39% of cache storage; this is much smaller than the observed contribution of 4% from the last line fetched.

The “last line fetched” is an ephemeral location, rapidly replaced by the next line fetch. Further experiments where conducted to examine the degree of temporal locality exhibited during the line fetch. For each line fetch operation it was determined how many words contained within the line were used before a request for other data was issued. Since line fetching is performed on a demand basis, all line fetches result in at least one word being used but they may use more before moving off the line.

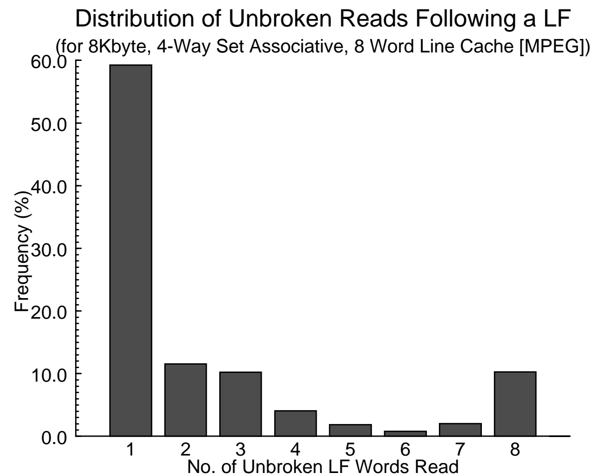

Figure 3: Histogram of Unbroken LFL Reads Following a Line Fetch

Figure 3 shows a typical distribution for the MPEG decoder. It can be seen that for about 60% of line fetch operations only the word that caused the miss is used – indicating that streaming would fail to give any performance improvement in these cases. A non-blocking cache would however, allow the following request to be serviced whilst allowing the line fetch to proceed in parallel.

4: Synchronous Implementations

As mentioned in §2.1 *Stall On Miss* is the most commonly used line fetch strategy since it is simple to implement. To improve performance some synchronous designs use an *Early Restart* based strategy (RS6000-560 [15], ARM [1]), in particular the ARM series of cached microprocessors use a variant which will be described below.

In the ARM a cache miss stalls the processor and causes a line fetch to be started from the address of the lowest word in the line. At the end of each memory read cycle the processor's request is compared[1] with the address of the fetched data entering to the cache. If they match it is sent to the CPU; if they do not, the CPU remains stalled for this cycle. Both the CPU and memory are clocked synchronously at the memory clock speed during this process. Thus streaming occurs as long as the processor continues to request data from sequential addresses.

The demand for greater performance has led to *decoupled architectures* becoming more common (PA-8000, MIPS R10000, HaL R1, P6), many of which utilise *non-blocking* caches in order to retain memory bandwidth during cache miss processing (HaL [7], P6 [6]).

A decoupled architecture is one where the CPU is fed instructions from a prefetch buffer large enough to hide some or all of the latency from a line fetch. This requires that the prefetch buffer can subsequently be refilled when the line fetch is complete so the cache must be able to supply instructions at a greater rate than the CPU requires.

Synchronous implementations to-date are expensive from a hardware perspective; utilising multi-ported, split caches with wide cache to CPU buses. The implementation complexity of these synchronous non-blocking caches is inappropriate for a small scale asynchronous microprocessor.

1. There is a match if either i) the word that has arrived is the word that caused the original miss or ii) if processor has requested a sequential address (a non-sequential request causes the CPU to stall until the line fetch has completed)

4.1: An Asynchronous Implementation

The cache being designed is targeted for the next version of the AMULET asynchronous ARM [3]; AMULET2e. This includes an improved processor core, a cache and other peripherals on a single chip. To save space there will be a single, unified instruction and data cache. Data and instructions share the same memory interface with data requests being arbitrated into the prefetched instruction stream. In this situation a non-blocking strategy should be particularly effective because it allows the independent prefetch and data streams to merge with minimum interference.

If the asynchronous fetch strategy is to allow for concurrent cache and line fetch activity, then there must be methods by which the two processes can be synchronised when necessary (e.g. when reading data that has just been fetched). In synchronous system this is done by synchronising on clock edges; comparing the state of two systems when they are known to be stable. In an asynchronous system no clock exists thus an arbiter is usually employed. Arbiters, however, are potentially slow circuits and thus it is desirable to avoid them in a design.

Figure 4 illustrates the positioning of a pipelined cache between the micropipelined processor and memory. The cache is divided into three stages; the Tag and Data stores and the Memory Interface.

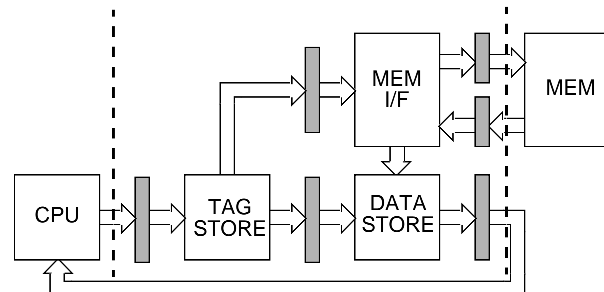

Figure 4: A Micropipelined Cache

Cache hit determination is done in the tag store with all the cached data being held in the data store. The memory interface is responsible for all transactions that involve main memory and in particular, it issues the multiple data read requests that constitute a line fetch operation.

The key to this design is the division of the data store into two parts; the main (conventional) cache

RAM array holding the majority of the cached data and a set of *line fetch latches*. The line fetch latches comprise a data latch for each word in a line and are used to store data as it arrives from memory. This data is held here rather than in the cache body until the next line fetch is initiated.

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 latches 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.

The Hit and LFL Hit control signals, shown in figure 5, 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 determined by comparing the input address with an additional tag which indicates the range of addresses currently being held in the line fetch latches. Note that a cache miss, after instigating a line fetch, looks for its data in the line fetch latches.

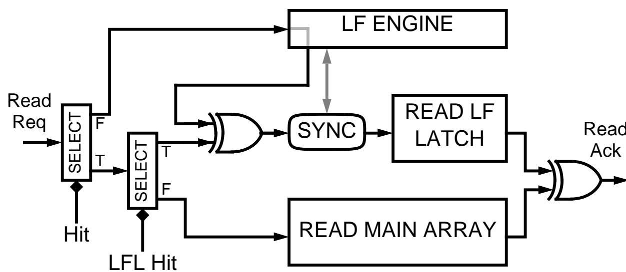

Figure 5: Control Circuit Request Steering.

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 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, 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. The cache is thus non-blocking.

The key point in this mechanism is the line fetch synchronisation block which must delay any line fetch reads until the correct data is present. Details of this block are shown in Figure 6.

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 data that has been fetched from memory. The synchronisation latches are transparent latches which stall latch read requests until the data that they require has arrived. A request to read a line fetch latch (Request In) is decoded and steered onto the correct individual word request line (LF Read Req0-3).

When a line fetch starts all the *line fetch latches* are empty and all the *synchronisation latches* are opaque. Opaque synchronisation latches block the progress of latch read requests until they are made transparent. This occurs when the line fetch engine receives data from memory (on the LF DATA bus), and asserts a LF Latch Control signal. This causes the appropriate line fetch latch to latch the fetched data and makes the corresponding synchronisation latch transparent allowing any stalled LF read req to proceed.

Using a transparent latch for synchronisation is not hazardous in this case because although a synchronisa-

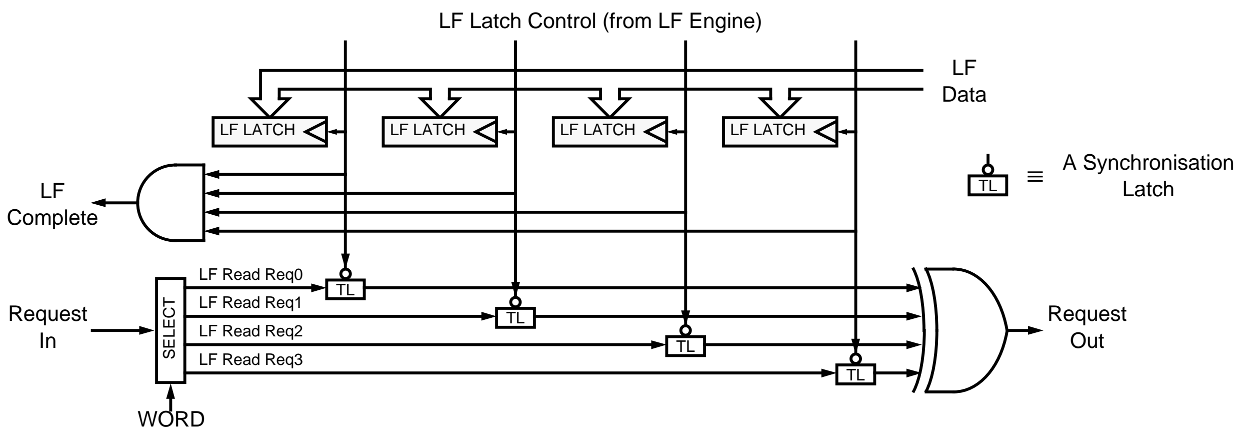

Figure 6: LF Read Synchronisation Circuit

tion latch may be *opened* before, after or at the same time as a request arrives it is never *closed* when a request may arrive and so there is no possibility of metastability.

The only further synchronisation required is between successive line fetch operations because they each require use of the line fetch engine and line fetch latches. Thus when a new read miss occurs it must wait for the preceding fetch to complete. Once the previous line fetch has completed, the contents of the line fetch latches must be copied into the main cache array before allowing the new fetch to begin.

In this scheme only the autonomous line fetch process is aware that a cache line fetch has completed. This is safe because it can stall a request for another line fetch arbitrarily. This mechanism also leaves the last line fetched in the line fetch latches until the next request arrives.

The final task of the line fetch engine is to copy the last fetched 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 latches are closed and the contents of the line fetch latches are copied – in parallel – into the RAM array as a standard cache line replacement. This is a rapid operation and may be hidden under the start-up time of the new line fetch.

5: Performance

A number of cache line fetch mechanisms have been described and combinations of these allow for a large number of different designs. The interaction between the different read requests can extend over several memory cycles and it is difficult to draw a clear picture of this.

Ultimately the desire is to maximise parallelism to increase performance. To determine the success of the proposed line fetch mechanism it has therefore been simulated using realistic estimated delays for units such as cache and main memory speeds. The speed of main memory was chosen to reflect the types of DRAM memory commonly available – a total latency of 70ns incurred when starting a line fetch with words arriving 40ns apart thereafter. The speed at which the final asynchronous system would cycle was not well defined and so both 10ns and 20ns cycle times were simulated.

The base system consists of an 8Kbyte, 4-way set associative, unified cache in which external (buffered) writes do not block line fetch activity. Figure 7 shows how the total execution time for the MPEG decoder varies as the line length is increased. The performance of both the stalling (hollow symbols) and non-blocking mechanisms is shown (solid symbols) relative to the execution time of the fastest configuration.

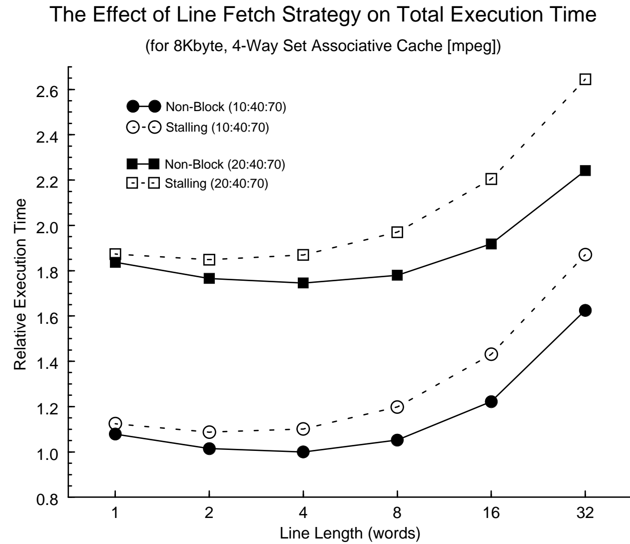

Figure 7: Overall Execution Time for Stalling and Non-Blocking Line Fetch Strategies

The ratio of CPU/cache hit cycle time to memory latency and transfer rate are shown in brackets, thus the upper pair of curves relates to the CPU/cache configuration with a slower 20ns cycle time and the lower has a 10ns cycle time. It can be seen that the non-blocking strategy consistently provides a speed advantage over the stalling mechanism; the speed-up ranges from 2.0% to 15.2%. These figures for speed-up are somewhat greater than those presented by Przybylski [10] [11]. This is explained by the nature of this processor and cache combination which is small as modern caches go, and its unified nature means that data references are arbitrated into the instruction stream.

The graph confirms the line length that gives the lowest total execution time for the cache with a stalling fetch policy is two words [8]. The preferred length for the non-blocking caches lines, however, is four words.

When line is increased to eight words the non-blocking cache performance is degraded by 5.2% in 10ns system and by 1.9% in the 20ns system. Even with this degradation in overall performance the non-blocking caches with eight-word lines still perform

better than the optimal stalling cache configuration. In general the non-blocking mechanism is more amiable to an increasing line length since longer lines allow greater time for concurrent cache and CPU activity.

6: Conclusions

A cache's performance is ultimately limited by its miss penalty which is largely dominated by the length of the cache line. The time taken to fetch a cache line completely can impede the processor's operation.

To alleviate the miss penalty it is desirable to have parallel CPU and cache activity during line fetch operations, particularly in systems with small, unified caches where occasional data references can break up the largely sequential instruction stream. In this situation concurrent CPU and cache activity can improve the performance of an optimally configured cache and processor system by up to 10%.

A non-blocking cache can also be employed in situations where an increased line length is desired, for example to increase the overall cache size without increasing the tags store. In this situation the non-blocking cache performs better than the best blocking configuration.

Traditional synchronous processors provide some parallelism by streaming which, in an asynchronous framework, required some complex synchronisation problems to be solved. In designing an arbiter free micropipelined circuit to provide this synchronisation it was found that a non-blocking cache, which further parallelises CPU and cache activity at no extra cost, was the simplest solution.

The design presented is modular. For example few constraints placed on the line fetch engine. This may fetch a line starting at any address (lowest, requested etc.) at the designer's discretion and words within the line may be fetched in any order. This allows design decisions to be made here which will not affect the function of other parts of the circuit although they may alter overall performance.

Advanced features such as cache hints may also be included. These start a line fetch in advance of the need for the data and cannot be allowed to stall the cache. Such software controlled cache prefetch directives have been shown to be highly effective at increasing the performance of code. Again this enhancement does not change the basic design of the circuit.

Finally it may be possible to abort a line fetch if it seems unnecessary. Again this would be a local function of the line fetch engine but in this instance arbitration would be required to allow a new line fetch request to interrupt one already in progress. Although this process has not yet been investigated it does not seem unlikely that a further performance benefit could accrue.

7: References

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

[2] "ARM Tools 200", ARM Ltd., Cambridge, England, January1995.

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

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

[5] 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.

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

[7] Linley Gwennap "HaL Reveals Multichip SPARC Processor", Microprocessor Report, Vol. 9 Number 3, March 6 1995.

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

[9] Przybylski, S., Horowitz, M., Hennessy, J., "Performance Tradeoffs in Cache Design", Proceedings of 15th Annual International Symposium On Computer Architecture, Honolulu, May 1988, pp290–298.

[10] Przybylski, S., "The Performance Impact of Block Sizes and Fetch Strategies", ACM SIGARCH: Computer Architecture News, Vol. 18 Number 2, pp160–169, June 1990.

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

[12] Smith, A.J, "Cache Memories", ACM Computing Surveys, Vol. 14 Number 3, September 1982, pp 473–530.

[13] Sutherland I.E., "Micropipelines", Communications of the ACM, Vol. 32 Number 6, January, 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.