A Result Forwarding Mechanism for Asynchronous Pipelined Systems

D.A. Gilbert, J.D. Garside
Department of Computer Science, The University of Manchester,
Oxford Road, Manchester, M13 9PL, U.K.
{gilbertd, jgarside}@cs.man.ac.uk

Abstract

Modern, fast microprocessors are deeply pipelined to enhance their performance. Thus they cannot afford to wait for each instruction to complete before starting the next. When inter-instruction dependencies are encountered it is essential that data are forwarded *from their point of production to where they are needed as rapidly as possible. This has been a problem in asynchronous processors because of the lack of synchronisation between the units producing and consuming the data. This paper presents a solution to this problem. The mechanism described allows the depth of speculative execution to be increased, improving memory efficiency by hiding the load latency yet still allowing precise exceptions.*

1. Introduction

Inter-instruction dependencies can cause severe degradation to the performance of a microprocessor. Most of these dependencies arise from an instruction needing an operand produced by a closely preceding instruction although the ordering of result production is also important. In the simplest model instructions are delayed until a dependency is resolved, resulting in a loss of overall performance. Result forwarding is a method used to alleviate the penalty caused by these dependencies.

It is possible to reduce the number of dependencies by reordering instructions to separate dependent instructions as far as is possible. Further improvements can be made if instructions are issued *out of order* [8] although this requires considerable hardware to implement. However instruction reordering does not usually solve the dependency problem completely and it is still beneficial to be able to reuse results quickly and – in a pipelined system – directly at the point of their consumption. This is the process of result *forwarding* [7] and is typically used to eliminate the need to write a register in one cycle and wait for the next cycle before it can be read again. Forwarding can be used beneficially with or without out of order execution.

Forwarding can be especially useful if particular instructions take a long time to complete. For an integer processor the most numerous such operations are memory transfers which typically require at least one extra cycle (and possibly more). This imposes a delay in the *completion* (or “retirement”) of the memory operation which can delay following instructions even though they operate entirely within the processor. It is possible to allow subsequent, faster instructions to continue and partially overtake memory operations. However it can be important to ensure that the instructions complete in the same order as they were placed in the code to ensure that the state of the system can be recovered if the memory operation causes an exception rather than completing as expected. In this case the memory transfer and any following (speculative) results must be discarded so that the system state is preserved and the aborted operation can later be restarted.

Forwarding can allow a number of instructions with dependencies (issued in order) to be evaluated whilst a load instruction is outstanding. A simple model is illustrated in figure 1, where the result of the addition, destined for R3, may be sped on to the subsequent instruction. Note that in this figure the writing of the register results has been kept in order.

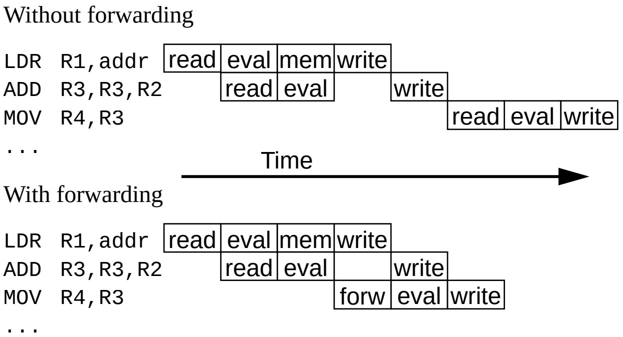

Figure 1: Performance benefits from simple forwarding

Result forwarding requires some synchronisation between the pipeline stage producing the data and the stage requiring them. This problem is easily solved using the global clock in synchronous systems; it is harder to achieve in an asynchronous environment because of the absence of any inter-unit timing coherency.

This paper presents an implementation of a *reorder buffer* [10], [8] which solves the problem of both result ordering and result forwarding in an asynchronous environment. Dependency hazards are avoided without recourse to register locking and – in consequence – the resulting latency of the circuit's implementation is expected to be reduced significantly. The resultant structure will form a key part of the AMULET3; a third generation asynchronous implementation of the ARM processor [1].

2. Asynchronous register forwarding

2.1: Earlier asynchronous dependency avoidance techniques

In AMULET1 [5] operations are issued in order and dependencies resolved by waiting for register updates using register locking [9]. Although functional this is slow – producing frequent dependency stalls – and the locking mechanism increases the register read cycle time. The long memory latency is alleviated by causing the memory system to produce a fault/no fault response early in the memory cycle and proceeding accordingly; a memory operation is then allowed to complete out of order as it is known that it *will* complete. The penalty imposed by this is low in a primitive system but increases with the complexity of the Memory Management Unit (MMU); large penalties would result if, for example, the access caused 'table-walking' in the MMU or if the cycle completion depended on external factors such as waiting for a bus timeout. Loaded data is returned asynchronously, out of order with the purely internal operation stream and is multiplexed into the register write stream using an arbiter. This mechanism, and its associated non-determinacy, was an added source of complication and required careful design to avoid the possibility of complex deadlocks. There is also a potential hazard when writing to the same register from different sources (a write-after-write hazard) which is only avoided by timing assumptions.

AMULET2 is basically similar to AMULET1, although some improvements were made, including removal of the write-after-write hazard by a more sophisticated register locking mechanism. It also incorporates limited forwarding measures by employing a "last result" register at the output of the ALU [6]. If the instruction decoder detects that an operation reuses the result from the immediately preceding instruction the register read is bypassed; this provides some performance gain but is limited and only useful for unconditional results. A similar mechanism is employed in the Hades processor [3] although this benefits from last result registers in each of several functional units. AMULET2 also supports a load forwarding register which gives a similar benefit for loaded data.

A radically different mechanism is used in the counterflow pipeline architecture [11]. This sends results 'backwards' up the pipeline so that close register dependencies rapidly meet their operands coming the other way. Designed solely for high performance this mechanism relies on numerous arbiters and many, wide buses. The size of this structure renders it unsuitable for small, low power processor implementations such as AMULET3.

Attempts have also been made to exploit the asynchronous, pipelined environment to avoid register storage and *only* forward results. An example is the SCALP architecture [4]. However this relies on a purpose designed instruction set and is thus not appropriate for implementing a currently existing commercial architecture.

2.2. Asynchronous FIFOs

To date, most asynchronous processors have been engineered using Micropipeline [12] style FIFOs which pass data from latch to latch (fig. 2). This mechanism is convenient as it only requires local connectivity and its control logic is simple. The disadvantages are that the latency increases with the length of the FIFO, power is consumed in shuffling values along the pipeline, and that the location of the data – once entered – is unknown until they exit. This precludes any attempt at reading data in the pipeline except at the single exit port.

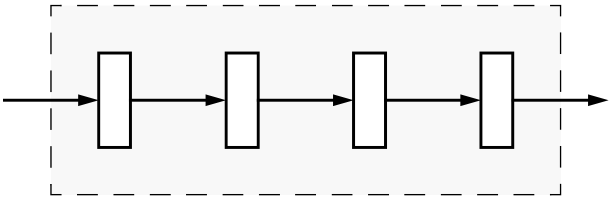

Figure 2: Simple FIFO structure

An alternative FIFO design distributes the input data to a set of latches, enabling each in turn, and recovering the data by multiplexing in a similar manner (fig. 3). This is the

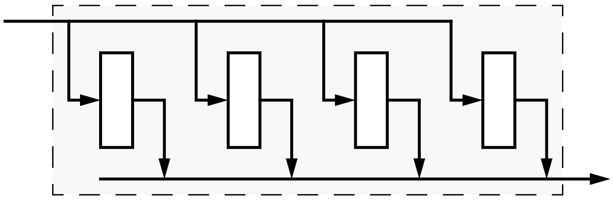

Figure 3: "Queue" FIFO structure

familiar “circular buffer”, referred to hereinafter simply as a **queue** for convenience. This suffers from increased control and wiring complexity, but it has the potential advantages of a lower latency from input to output and reduced power consumption because the individual latches make fewer transitions. For these reasons such buffers have been proposed for asynchronous systems before [14], [2].

The queue also has the attractive property that data remain unaltered *after* being copied out until they are overwritten by a new input. This lifetime is a fixed number of input operations (equal to the length of the queue) and is independent of the copy out process. Once a location is allocated it is known that either now or in the near future it will contain some particular data. This allows forwarding via a request which may need to wait for a result to arrive but can never be too late to gather its data. This is a single-bounded problem and is easily soluble in an asynchronous environment.

A queue acting as a simple FIFO can be described in terms of a pair of processes. The first (“head”) process waits for new data to arrive, allocates space and copies the data into that space. The second (“tail”) process seeks data in the queue and copies them out to the subsequent unit. These processes are unsynchronised and work independently, subject to the constraint that a queue of size “N” must contain between 0 and N data packets. In this way several entries may be outstanding at any time, the only proviso being that the head must not “lap” the tail.

2.3. Result Forwarding

To facilitate forwarding from an asynchronous queue the simple model described above must be elaborated. The “head” process is split into its two components, so that space is allocated for a result before processing commences, some time before the result arrives. In a pipelined system this allows several results to be outstanding at any time. The location of the queue in a simple processor structure is shown in figure 4.

If evaluation times vary, computations can be parallelised because the queue can act as a reorder buffer. This requires multiple ports into the queue (as shown in figure 5) which may operate asynchronously because the allocation process has guaranteed that write conflicts cannot occur. The “tail” process is unaltered and works on an ordered, serial stream.

Forwarding is required when an operation undergoing processing requires a recent result which *may* not have been returned to the general register bank. The queue position of this result is already known and the operation can examine this location; it may be able to continue at once or it may have to wait until the result is inserted before continuing.

The only complication to this mechanism is that not all allocated queue destinations receive valid data. After allocation and issue an instruction may be invalidated due to an abort, a condition code failure, the discovery that it was incorrectly fetched following a branch, etc. Thus it is sometimes necessary to mark data in the queue as invalid. As these should not be forwarded it is important to have a “default” value.

The register bank is always read regardless of any forwarding operation. This provides a value if there is no register to forward, either because the register is invalid or because that register is no longer in the queue. As the register bank updates are asynchronous the value read from the register bank may be the correct value, a previous value or even changing when read. However in all cases where there may be doubt a forwarded value will replace it.

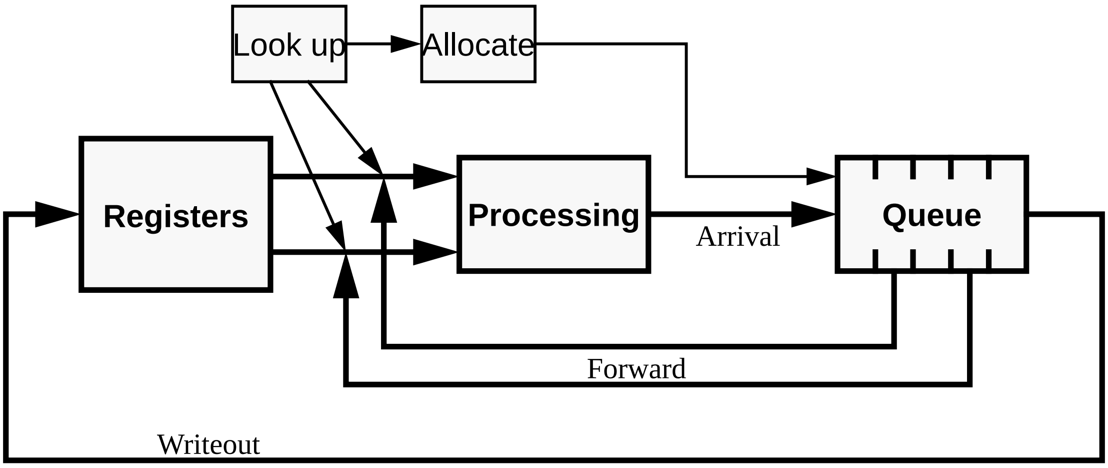

Figure 4: Queue position in processor cycle

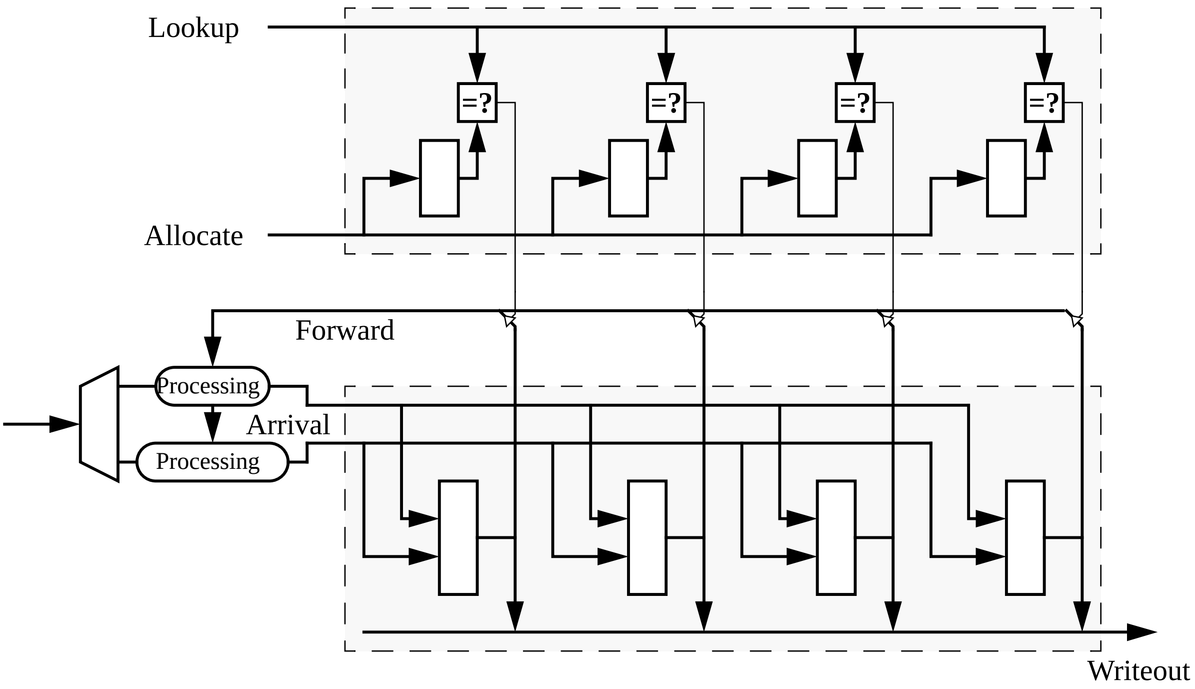

Figure 5: Details of queue showing various processes

2.4. Process model

There are now five distinct processes associated with queue operation (fig. 5):

Look up: When an instruction is decoded its source registers are examined to see if they are present (or should soon be present) in the queue. This is performed by a small CAM (Content Addressable Memory) structure which holds a destination register number for each queue entry. The resulting bit mask identifies zero or more possible data sources. By increasing the number of CAM comparisons it is possible to forward several values simultaneously.

Allocation: Once the address mask is obtained the instruction's own destination address(es) can be written into the CAM, the writing position being allocated cyclically. This decouples the CAM from the data. Although the instruction may then be targeted at a queue location which it also needs to read it cannot corrupt its own input because the input must precede the write.

Forward: Concurrently with allocation, source registers previously discovered by the lookup process may be forwarded. The forwarding process examines each candidate, starting at the most recent, waits until the data are present and then checks their validity. Valid data are forwarded, otherwise the process proceeds to the next most recent possibility. If all the possibilities are exhausted the forwarding is abandoned and the default value from the register bank is used. Multiple read processes may be used to forward different values to different places simultaneously.

Arrival: Results arriving at the queue carry their own queue address. Although there may be multiple write ports the addresses are guaranteed to be non-conflicting by the allocation process and, when the data arrive, the previous data is known to have been both copied into the register bank and forwarded as required. They may therefore be overwritten and the queue location implicitly marked as “full” for the write process to detect. The result may also be tagged as invalid at this time if the instruction has been abandoned.

Writeout: The write process copies results back into the register file. It examines queue locations in a cyclic fashion and waits until the next candidate is “full”. It then copies the data to the register bank (providing that the location is not tagged “invalid” in which case this operation is skipped) and marks the location as “empty” before proceeding to the next entry. This is a relatively simple, free running process. In operation it should normally be fast enough to maintain the queue in a nearly empty state, stalling only when waiting for slow memory cycles.

3. ARM requirements

Although general in application, the asynchronous queue was devised for AMULET3 which imposes some specific demands and constraints. ARM instructions may produce zero, one or two results per cycle. (The LDM instruction may produce up to 17 results, but is subdivided into a number of cycles.) The instructions can be divided into four categories according to the number and source of their results:

0	No results (e.g. CMP)
1	Internal result (e.g. ADD)
2	External result (e.g. LDR)
3	Two results (e.g. stack "pop")

The last category writes back a modified register internally whilst loading a (usually different) register from memory. These operations are initiated in parallel but may complete at arbitrarily different times. However all register destinations can be supplied down one of these two possible streams, one for internal calculations the other (slower) for data from memory. This implies the need for two independent write ports on the result queue in AMULET3.

A complication in the ARM instruction set is that *any* instruction can be conditional. Waiting for the conditional dependencies to be determined would result in inefficient use of the pipeline, so queue spaces are allocated to conditional and unconditional instructions alike. If an instruction fails its condition code test and is thus abandoned it must still carry a token through to the queue to indicate that the instruction has been processed.

Invalid queue slots introduce wastage into the queue. In practice roughly 25% of ARM instructions are conditional. The majority of these are branches which are not allocated queue slots (whilst the Programme Counter (PC) is addressable as a general purpose register it is not implemented as such). About 10% of instructions are conditional operations requiring queue locations allocating and, as around 50% of conditional operations are executed, a queue slot will be marked invalid due to a condition code failure about 5% of the time.

Unpredicted branches account for about 5% of instructions. The invalidation mechanism is used to discard results from instructions in the shadow of a branch (i.e. fetched erroneously after a branch instruction). In AMULET3 only the instruction immediately following the branch need be allocated a queue slot and its result is marked invalid.

The final cause of queue invalidation – data aborts – are comparatively rare and may be neglected here. Thus about 10% of queue locations are expected to be lost by invalidation, which gives a respectable 90% utilisation.

In AMULET3 it is known whether a value is valid or invalid at the bottom of the ALU stage, with the sole exception of data aborts. An optimization here allows abandoned load instructions (whether prefetched in error or failing their condition check) to invalidate their queue places without passing through the memory system and incurring a significant, unnecessary delay. This operation is asynchronous to the others and requires a third write port on the queue. This extra port need not carry data values, merely an input to set the invalid bit in the relevant entry; its cost is therefore low.

A final form of "lost" queue entry is produced when a value is stored to the memory. The store operation is allocated a queue entry to prevent subsequent instructions from overtaking it. The store – if it does not abort – returns a null value which "fills" this location and allows subsequent state changes to proceed. Fortunately it is possible to economise on queue entries by allocating a single "place holder" to an (indivisible) multiple data store (STM) which is "filled" by the completion of the last memory operation.

Measurements on benchmark programmes suggest that the number of stores vary according to the application, but account for around 8%-12% of queue slots allocated.

3.1. Data Aborts

Data aborts are *precise* exceptions and must freeze the processor state so that the faulting instruction can be rerun in the future. This is usually due to a page fault – hopefully a rare, if expensive, occurrence – and so a simple mechanism may be employed to recover from this.

The current intention is to mark the aborting instruction in the queue and send an 'interrupt' to the prefetch engine causing it to jump to the relevant exception handler. As the processor begins to recover the queue drains into the register bank until the aborting instruction reaches the bottom; this and subsequent queue entries may then be discarded until the new instruction stream arrives.

As a final note it is only possible to preserve the state if all effects of the faulting instruction are discarded. A load with an address writeback (e.g. a stack "POP") has two destinations; correct abort response is achieved by ensuring that the external memory destination is allocated 'before' the internal writeback destination. Thus the base register is not written back to the register bank until the memory cycle has completed successfully; the base register can, of course, still be forwarded for use in subsequent instructions.

3.2. Operational example

To illustrate the foregoing, figure 6 shows the state of the queue in operation on a (contrived) ARM code fragment. In this example the writeout process has stalled waiting for the memory to return a result, but execution has continued, including forwarding the R4 result. The STR (STore Register) instruction is currently being evaluated and has been allocated a place in the queue to delay following instructions from completing in case it aborts. The AND that is just being issued will be able to forward R0 immediately (although the first location examined is invalid) but must wait for R3 to be loaded before continuing. This instruction will overwrite one of its operands, but

```
   MOV   R0,#0
   CMP   R0,#1
   MOVEQ R0,#99
   MOV   R2,#2
   LDR   R3,slow_memory
   ADD   R4,R0,R1
   MOV   R5,R4
   STR   R2,some_address
⇒  AND   R7,R0,R3
   ...
```

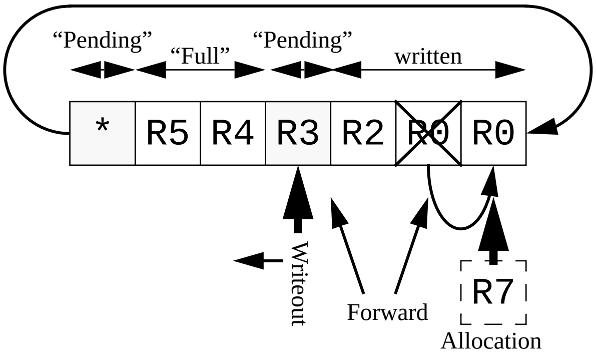

Figure 6: Queue during operation

only after it has been read.

Assuming it does not abort, when R3 arrives the AND instruction will forward its value and continue, and the writeout process can start to empty the queue further. If the load aborts the writeout process will initiate the abort response and discard the queue entries for R3, R4 etc. until the new instruction stream arrives. In this circumstance the AND instruction would detect invalid forwarding information (from R3), assume the default value from the register bank and continue until it had filled its allocated queue slot (where it too would be discarded).

Note in figure 6 that there is not a 1:1 correspondence between instructions and queue locations; the latter are allocated according to need (see below). Therefore the CMP instruction has no destination whilst some ARM instructions require more than one destination register.

3.3. Operating Constraints

In AMULET3 the queue does not operate in isolation, and this fact is used to guarantee correct operation. The major external constraint ensures that the queue never fills beyond its capacity, therefore overwriting earlier results before they can be written back to the register bank. This requires some interaction between the 'write' process and the 'allocation' process which ensures that there is a maximum to the number of results outstanding at any time, which is the size of the queue. If this maximum is reached then the allocation is suspended until a queue entry is freed. With a queue of sufficient entries it is unlikely that this will happen.

The current proposal is simply to pass a token from the write process to the allocation process every time a location is freed. Each token represents a queue slot. Tokens are buffered in a FIFO so that a pool of free slots is normally maintained, which dries out if the write process is stalled for a significant period.

The queue can never be "over emptied" since the write process is suspended if the queue becomes empty.

4. Context in AMULET3

In AMULET3 the register forwarding queue lies between the ALU output and the register bank's write port. The current design has two parallel input channels, one leading directly from the ALU, the other delayed by passing through the data memory system. At the time of writing a third port, to be used for coprocessor result transfers, is also under consideration.

It is expected that a single port to the register file is sufficient to meet bandwidth requirements; this port will become stalled when an outstanding memory operation reaches the bottom of the queue, but should have a higher throughput than other parts of the processor so that it is subsequently able to catch up. This is an area where asynchronous operation is an advantage. Not being locked to a global clock even slightly faster cycles are sufficient to begin emptying the queue once it is freed.

In a similar model to DEC's StrongARM [13], AMULET3 will have three independent read ports on the register bank and so three forwarding ports will be provided, one for each register field. Although most instructions require fewer than three operands the cost of the extra hardware is deemed economic considering the control simplification this yields.

5. Queue Size & Performance

It is difficult to predict the ideal size of the queue, not least because code can be reordered at the compiler level to optimise for a given implementation. Estimates must therefore be influenced by coarse measurements and "reasonable" assumptions.

The first measurement which can be made is the percentage of register values which will be forwarded from a queue of a specified length. These figures are shown in figure 7 for a few benchmark programmes. As the queue must have at least two locations (since up to two results can be produced by a single instruction cycle) it can be seen that

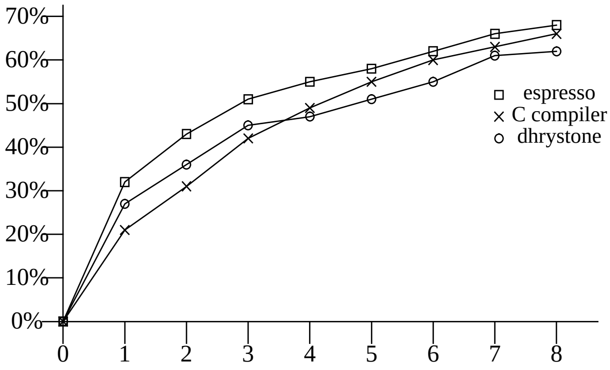

Figure 7: Percentage register reuse vs queue

at least ~30% of results will be forwarded from the queue.

The queue should be large enough to accommodate the results of any speculative instructions whilst a memory reference is outstanding. This size depends on the memory latency, a slower memory needing more non-dependent instructions to fill its “delay slots”. An upper limit is the number of instructions which follow a memory operation which do not depend upon it. This is relatively small for load operations – benchmark measurements suggest that around half of the data values loaded are required by the subsequent instruction – but in other cases some performance benefit may be derived from speculating further, and code reordering may improve this slightly. The full benefit is derived from all store operations, barring those which abort.

Most of these factors advocate a long queue which would never fill completely. However the need for a fast implementation suggests that the queue should be short. A compromise is therefore required to determine the shortest queue that rarely becomes full and so does not produce a significant number of stalls. Although studies are ongoing, a preliminary working size of four entries is being considered as a reasonable compromise; this should be sufficient to avoid limiting performance due to a lack of queue space except for certain rare cache misses.

6. Implementation

Each queue entry comprises the following fields:

- 7-bit CAM (4-bit register identifier
  & 3-bit operating mode)
- 32-bit data field
- Full
- Invalid
- Aborted
- Abort colour
- Forward colour

The CAM (Content Addressable Memory) serves a dual role. Its primary function is to allow rapid association of a required register address with the current queue contents. Because the CAM is very small it can be built of static gates, and is thus very fast. The CAM’s secondary role is to store the register destination address of a queue entry, which is used when the result value is copied out.

The full bit indicates that the queue entry contains data which is waiting to go back to the register bank. This is set by data arrival and cleared by writeout and so forms a four phase indicator of the state of the queue location for the writeout process. It is not stored explicitly as it forms part of the control circuitry.

The Invalid, Aborted and Abort Colour bits have the same timing characteristics as the data; they are used to control the writeout and forwarding processes. Invalid is set if the data should not be forwarded or returned to the register bank. Abort is set if a result from memory aborted; it is trapped by the writeout process and used to initiate exception entry. The abort colour is used to identify instructions following an aborted memory operation; it is changed every time an abort is initiated and subsequent operations with the former colour are then discarded.

The final bit maintained by the queue is a forward colour bit. This is used by the forwarding process to indicate whether a result value has reached the queue. A forwarding request carries with it an “expected” value of this colour sent by the lookup process. If the value in the queue matches the expected colour it is known that the result has arrived (although a validity check is still needed). If there is a mismatch the result is outstanding and the forwarding process must wait until the match is made. The local colour is a two-phase signal which is changed only when results arrive in the queue; it is independent of the writeout process.

6.1. Physical implementation

It is necessary for the forwarding mechanism to be as rapid as possible to ensure that (potentially frequent) stalls waiting for a result value are minimised. Typically the critical path will be from data arrival in the queue to forwarding it to a following operation. Because the queue is relatively small (four locations or thereabouts, rather than the 31 entry register bank) it can be constructed of physically large latches to provide a fast, high drive output and have a static read mechanism to avoid precharge delays. The control overhead is also relatively small so that the data should be available in about the time taken to drive the bus through the multiplexers

Figure 8 shows a draft of the forwarding mechanism. If a forwarding request is made it is delayed until the forwarding colour (held by the toggle flip-flop and changed by result arrival) matches the expected value; it is certain that

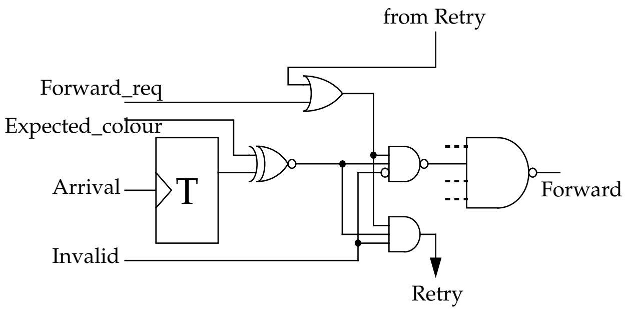

Figure 8: Forwarding validation scheme

the required result *will* arrive if it is not already present. When the request and the data are both present an output request is generated unless the result has been invalidated during processing; in the latter case a “retry” is invoked and further queue entries are examined. If all the queue entries are exhausted the forwarding attempt is abandoned.

The *arrival* and *writeout* logic (fig. 9) is also kept as simple as possible in order to facilitate rapid operation. An input request (Rin) is simply steered to the enable of the relevant queue location. This is noted by the C gate which acts as a latch, holding the “Full” indicator. The input can then be acknowledged and the input handshake completed.

The output is a freely running cycle which activates each location in turn by passing a token from location to location (Tin to Tout). A location attempts to drive Rout when it is both full and has received the token. This causes a handshake cycle on Rout/Aout, although this is averted if the location is invalid. The cycling of the Aout signal causes the token to shift from Tin to Tout ready for the next cycle and removes the Full signal.

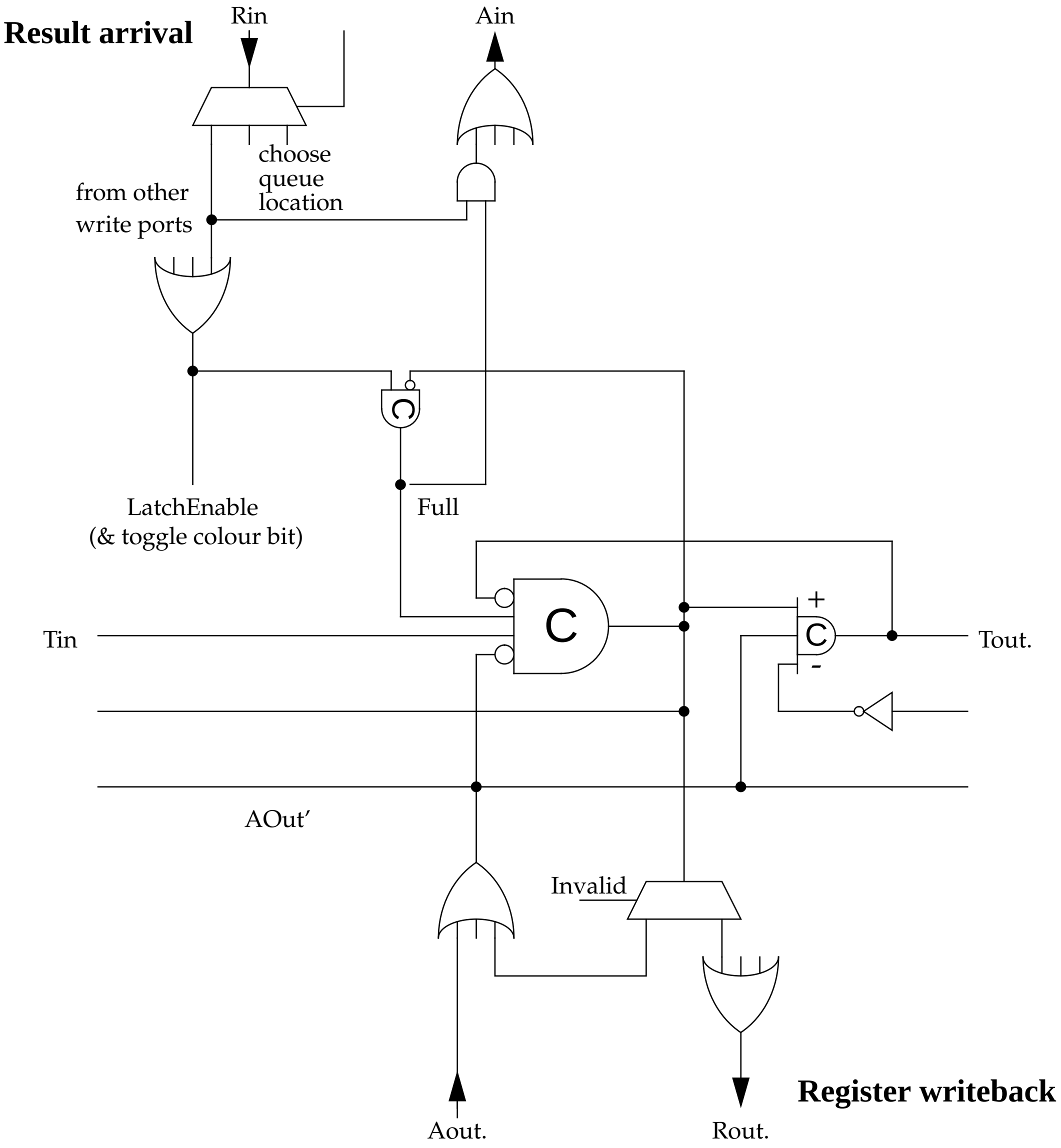

Figure 9: Queue write/read controller

The input and output processes are free running so that the queue may be in any state from empty to completely full at a given time. These broad constraints allow considerable freedom of operation for the other pipeline stages. The only external constraint necessary is that, as the queue fills, a result is not overwritten before it has been output; this is imposed by the allocation process described in section 2.4.

7. Other queue applications

The queue mechanism described can be used in applications other than register forwarding. An analogous example is memory forwarding in a write buffer. When a store operation is initiated the processor may be left to proceed whilst the store completes in parallel. If the store is slow (such as in a write-through cache where stores proceed at external memory speed) a performance increase is available if the writes are queued and their cost hidden in parallel with subsequent instructions. If a subsequent instruction is a load it may require some data which may or may not have been transferred to memory. The simplest solution is to ensure that the write buffer has emptied before the load may proceed, but the cost of this may be unacceptable. It is more sensible to wait only if the load refers to a pending write although potential hazards must then be identified by introducing a tag to the write buffer. If this has been done it is simple to allow the last “N” writes to be forwarded from a queue, effectively removing the penalty of waiting altogether. This application is under consideration for the AMULET3 system.

8. Conclusions

An implementation of a reorder buffer which solves the twin problems of result forwarding and exception handling within an asynchronous pipelined system has been presented. It is expected that this will form a key component in the AMULET3 microprocessor. This unit, the *queue*, allows a high degree of flexibility in operation (such as out of order instruction completion) whilst avoiding all classes of dependency hazards; read after write is stalled until the relevant value appears, whereas the reordering function ensures write after write hazards are averted.

The dependency stalls which were handled by the register locking mechanism in AMULET1 and AMULET2 are now deferred to the queue forwarding mechanism. The cycle time of the decode/register read unit is thus reduced. Result forwarding also reduces the distance critical results have to travel to be reused, reducing the latency imposed by a dependency stall.

By allowing data transfer completion to be speculative and deferring abort handling the memory reference mechanism may be streamlined, again reducing the cycle time of a major pipeline stage. Non-dependent instructions may follow memory references much more closely than in the earlier AMULET processors.

The solution is a general one and is applicable to other situations; the example of forwarding from a memory write buffer has already been suggested above. The queue could also be used as a reorder buffer for result gathering and forwarding in – for example – a future superscalar and out-of-order issue asynchronous microprocessor. Subject to physical limits the queue can have an arbitrarily large number of entries, input ports, and forwarding ports. It is also possible to parallelise the writeout process to a great extent if it is necessary to increase its output bandwidth.

Rapid forwarding of results is an important performance enhancing feature in many high-performance synchronous microprocessors which has formerly proved difficult to adapt into an asynchronous environment. We believe that the mechanism presented here is another means of closing the performance gap between asynchronous and synchronous systems.

9. Acknowledgements

This work has been partly supported as part of ESPRIT project 20452, OMI/DE2 (the Open Microprocessor systems Initiative - Deeply Embedded project 2) and partly by the EPSRC (grant number 93315548). The authors are grateful for this support. The authors would also like to thank other members of the AMULET research group at Manchester University.

10. References

[1] ARM Ltd., *ARM Architecture Reference*, July 1995.

[2] van Berkel, K. *Handshake Circuits An asynchronous architecture for VLSI programming,*
International Series on Parallel Computation 5,
Cambridge University Press, 1993

[3] Elston, C.J., Christianson, D.B., Findlay, P.A., Steven, G.B., *Hades – Towards the design of an Asynchronous Superscalar Processor*, Proceedings 2nd Working Conference on Asynchronous Design Methodologies,
IEEE Comp. Soc. Press, May 1995.

[4] Endecott, P.B., *SCALP: A Superscalar Asynchronous Low-Power Processor*, PhD Thesis, Department of Computer Science, University of Manchester, 1996
http://www.cs.man.ac.uk/amulet/publications/thesis/endecott96_phd.html

[5] Furber S.B., Day P., Garside J.D., Paver N.C., Woods J.V., *A Micropipelined ARM*, Proceedings of VLSI ‘93, Grenoble, France, September 1993.

[6] Furber, S.B., Garside, J.D., Temple S. , Liu, J., Day, P., Paver, N.C., *AMULET2e: An Asynchronous Embedded Controller* Proceedings Async '97, IEEE Comp. Soc. Press, April 1997

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

[8] Johnson, Mike, *Superscalar Microprocessor Design*,
Prentice Hall Series in Innovative Technology. 1991.
ISBN 0-13-875634-1

[9] 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.

[10] Smith, James E. & Pleszkun, Andrew R., *Implementing Precise Interrupts in Pipelined Processors*,
IEEE Transactions on Computers, Vol. 37, No.5, May 1988, pp.562-573.

[11] Sproull, R.F., Sutherland, I.E., Molnar, C.E.,
Counterflow Pipeline Processor Architecture,
Sun Microsystems Laboratories,
April 1994 *http://www.sunlabs.com/technical-reports/1994/smli-tr-94-25.ps*

[12] Sutherland I.E., *Micropipelines*,
Communications of the ACM.
32(6): pp.720-738, January 1989.

[13] Turley, Jim, *StrongArm Punches Up ARM Performance*,
Microprocessor Report Vo. 9 No. 15 pp.16-19 – Nov. 13th 1995

[14] Yantchev, J.T., Huang, C.G., Josephs, M.B., Nedelchev, I.M.
Low-Latency Asynchronous FIFO Buffers,
Proceedings 2nd Working Conference on Asynchronous Design Methodologies,
IEEE Comp. Soc. Press, May 1995.