Memory Faults in Asynchronous Microprocessors

D.W. Lloyd, J.D. Garside, D.A. Gilbert
Department of Computer Science, The University of Manchester,
Oxford Road, Manchester, M13 9PL, U.K.
lloydd@cs.man.ac.uk, jgarside@cs.man.ac.uk, dg@treblig.org

Abstract

Although a large number of asynchronous microprocessors have now been designed, relatively few have attempted to handle memory faults. Memory faults create problems for the design of any pipelined system which are exacerbated by the non-deterministic nature of an asynchronous processor.

This paper describes these problems as encountered in the design of asynchronous ARM processors and discusses their specific solutions in the AMULET3 processor. Different mechanisms were used, as expedient, to maintain coherency for the various state-holding elements within the processor; these include register renaming and history buffering in addition to resource locking.

1: Introduction

Programs are normally executed in a sequential order, and the programmer's view of the system reflects this. However, behind the scenes, the quest for higher performance has led to a disruption of this simple model; many modern, high-performance microprocessors execute code in a manner which is anything but simple or sequential. This is largely possible because the code's behaviour is still predictable and speculation on instructions is relatively successful although, occasionally, events happen which are unpredictable and fall outside this scope.

Such events are usually classified as 'exceptions', a term which can include everything from operating system (OS) service calls to memory faults. Handling the exception requires the processor to suspend the running program, run an exception handling routine and, usually, restore the original state to continue the program.

Exception handling is essential in a practical microprocessor. However exceptions are rare events compared with 'conventional' instructions. It is therefore of primary importance that the ability to process an exception does not detract from the processor's overall performance.

The precise exception model [1] views an exception as occurring at a point exactly between two instructions such that all preceding instructions have completed and none of the subsequent instructions has started. To implement precise exceptions requires the processor to save its state immediately before the execution of one particular instruction and after all prior instructions have completed. This is straightforward in a processor which fetches and executes only a single instruction at a time [2]; however, many modern microprocessors process multiple instructions concurrently either by pipelining, superscalar issue, or other techniques. In these systems the global state of the processor (at any given point) does not represent a precise instruction state since more than one instruction may be being executed.

Most causes of exceptions are detectable early in the execution lifetime of an instruction; for example OS calls and emulator traps are performed as dedicated instructions. Other exceptions – notably interrupts – may be inserted almost arbitrarily into the instruction stream and processed at leisure. However, a few exceptions are detectable only after the processor has committed to seemingly innocuous instructions; examples of this last category are arithmetic errors (such as division by zero) and memory faults.

In synchronous processors the clock provides a convenient mechanism for predicting the state of all stages in a pipeline. The consequent predictability simplifies the determination of which stages were executing instructions prior to the exception and which were executing instructions subsequently. This allows the separation of the state which must be stored for recovery and the state which must be discarded.

In asynchronous systems there is no clock and so this predictability is lost; other mechanisms must be provided to generate a precise state. Previous work undertaken on the FRED asynchronous processor [3] reduced the complexity of the problem by providing only a 'functionally precise' exception mechanism: the saved state is not precise but contains sufficient information to allow the operating system to recover. Whilst this approach is acceptable in the design of a new processor architecture, it cannot be used for the reimplementation of an existing architecture such as the ARM [4].

This paper describes the approaches used to process memory faults in the AMULET3 asynchronous microprocessor. The focus is upon memory faults, however, the techniques can easily be applied to solve other problems such as bus errors.

Previous asynchronous solutions required the processor to stall whilst uncertainties were resolved. The solutions, developed for AMULET3, alleviate this requirement allowing speculation on memory accesses whilst maintaining compatibility with the ARM's exception model. The result is a significant improvement in performance. Three different mechanisms are used: a reorder buffer, history buffering, and stalling. Each was employed according to a trade-off between speed, complexity and frequency of invocation. Although these techniques are familiar in synchronous design it is believed that AMULET3 is the first asynchronous microprocessor to exploit the first two.

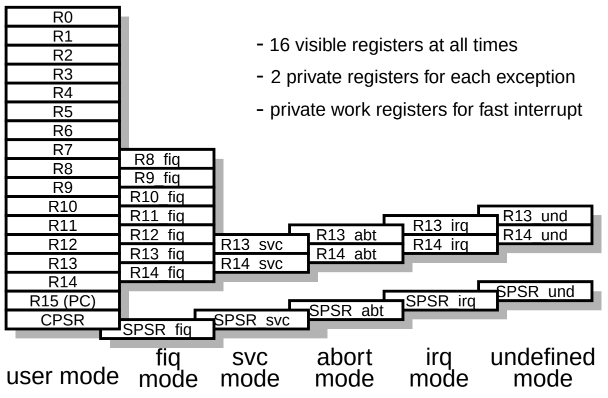

Figure 1: ARM register organization

2: The ARM architecture

The programmer's model of the ARM registers is shown in figure 1. This is organised as:

- General Purpose Registers. These contain the current values of data items used by the program. To simplify exception handlers each mode is provided with its own link and stack pointer registers (banked copies of R14, and R13, respectively) to enable it to create its own private stack.
- Current Program Status Register (CPSR). This register contains the condition code flags, interrupt disable bits and processor operating mode.
- Saved Program Status Registers (SPSR). These are copies of the CPSR which are updated by the processor whenever a relevant exception occurs.

ARM processors employ the same basic mechanism for handling all exceptions. When an exception arises the ARM processor:

- completes the execution of all preceding instructions
- saves the return address in R14 of the new mode
- saves the CPSR in the SPSR of the new mode
- enters the appropriate privileged mode
- branches to the appropriate exception handler

On completion of the exception handler the state and mode are restored, and execution of the original program is resumed. The CPSR is restored from the SPSR at the same time.

The processor state represents only part of the system state which also includes the state of the memory and of any coprocessors (ARM has provision for up to sixteen coprocessors which can be included to perform extensions to the original instruction set). These, too, must be preserved when an exception such as a page fault occurs to ensure that it can be processed successfully. The coprocessor state is not discussed here, but the memory state is considered in a later section.

The ARM architecture provides support for precise exceptions resulting from both internal and external events [4]. This support comprises two levels of interrupt, a software interrupt (SWI), an undefined instruction trap and (separate) page fault mechanisms for instruction fetch and data access; the ARM does not generate exceptions on arithmetic instructions.

The memory system alerts the processor to a memory exception by raising the dedicated external abort pin. To simplify the task for the exception mechanism, early ARM processors required the abort response half-way through the memory access cycle. This is called *early* abort timing. However, this requirement severely constrained the operation of the memory system and so recent processors allow the memory system to flag the abort response at the end of the memory access cycle. This is called *late* abort timing.

The ARM model differentiates memory faults which occur when accessing instructions, called *prefetch aborts*, from memory faults which occur when accessing data, called *data aborts*. Although they share the 'abort' mode of operation different exception processing routines are called for each form of exception. More significantly, a prefetch abort occurs as the processor fetches an instruction to execute *later.* Only when (and if) the instruction reaches the execute stage of the pipeline is the prefetch abort exception taken. A data abort occurs *during* the execution of a load or store instruction which may already have started to change the processor's state.

In both cases the processor needs to determine the memory address which faulted in order to restart the aborted instruction. In the ARM, data access memory addresses are generated by adding or subtracting an immediate or register-based offset to or from a base-register. The restart operation is complex as the value in the base register may be auto-indexed or overwritten. An implementation can elect to supply either the original base register value or the auto-indexed value to the exception handler, but it must provide a value which has not been overwritten. Early ARM processors (up to and including the ARM7) adopt the *base updated* approach, and supply the handler with a base register with any modifications to the value it contained due to auto-indexing allowed to complete. This simplifies the hardware implementation and allows faster exception entry, but requires additional software in the exception handler to restore the original value. Later ARM processors adopt the *base restored* approach, where the exception handler is presented with a preserved and unmodified base register value.

The ARM architecture specifies an integrated system for dealing with exceptions which conforms to the precise model. Detailed high level requirements are given, but designers are given a large degree of flexibility in how to implement these.

3: Existing mechanisms for state management

To support precise exceptions it is necessary to provide mechanisms to manage the system state. These mechanisms typically operate by preventing speculative state changes or providing temporary storage for either the original state so that it can be restored or the speculative state so that it can be discarded [5].

Locking mechanisms are designed to prevent speculative state changes, a variety have been proposed whose common aim is to delay the execution of instructions until their status has been determined.

The ‘history buffer’ is a mechanism which stores the original values held in registers before speculative changes are made. As each instruction is issued an entry in the history buffer is allocated. If an exception occurs the entries in the history buffer are used to restore the original state.

A ‘reorder buffer’ provides a temporary store for speculative results. Results from the reorder buffer are used to update the register bank in instruction issue order, only when the instructions are known to have completed successfully. When an exception occurs all outstanding instructions are allowed to complete and the results in the reorder buffer prior to the failed instruction are written to the register bank, while those after the exception are discarded. The reorder buffer causes a delayed write back of results which increases the latency for result availability. To counteract this problem forwarding is allowed from the reorder buffer enabling instructions to use these speculative results.

While the storage and recovery of results is a major part of exception handling there are other issues which prove straightforward for synchronous designers and more challenging in the asynchronous world. The key simplification found in synchronous design is the global coordination for exception recovery provided by the clock. This is not possible in an asynchronous design as the designer cannot be sure that the recovery mechanisms will be examining a global signal at the required time. Traditionally this has restricted asynchronous designers to implement precise exceptions using only the most primitive mechanisms, such as locking techniques.

4: AMULET1 and AMULET2

The AMULET1 [6] and AMULET2 [7] processors represent the first and second generation, respectively, of asynchronous ARMs. They are object code compatible with the ARM6, (although no support is provided for coprocessor instructions). Internally the processors are constructed from functional units which operate independently and concurrently. They are architecturally similar and, for convenience, only AMULET2 will be discussed. Before describing the exception mechanisms it is necessary to review some features of these earlier implementations.

4.1: PC tracking

The ARM architecture requires that instructions must have access to their correct PC value prior to execution in case it is needed as an operand (R15). This requirement is implemented in the AMULET2 processor by matching each instruction as it is returned from memory with its associated PC value (see figure 2). The PC values are stored in a FIFO which tracks the instruction flow. This also enables

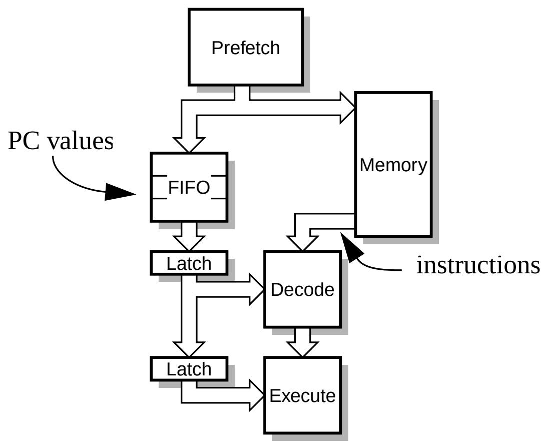

Figure 2: PC and instruction synchronization

the processor to provide the return address from the exception handler simply by collecting the PC value associated with the aborted instruction in the execution stage.

4.2: Branches and instruction ‘colouring’

In a synchronous processor the depth of instruction prefetch can be controlled, or at least measured, quite easily by counting clock cycles. Knowing the depth of prefetch makes discarding erroneously prefetched instructions straightforward. In an asynchronous architecture, because the prefetch and execution units are de-synchronised, controlling or measuring the depth of prefetch is not easy, and so determining which instructions to discard in the advent of a branch proves a complex task.

One approach is to specify the depth of instruction prefetch by means of a fixed set of ‘tokens’ (e.g. as used in ASPRO-216 [8]). A penalty is that this exact number of instructions must either be discarded or executed as delay slots following a branch instruction. An alternative that has used in all the AMULET processors to date, is to tag each instruction fetch with an identifying ‘colour’ [6]. A branch causes the execution stage to interrupt the prefetch unit (via an arbiter and therefore at a non-deterministic position in the instruction stream) and causes it to start fetching from a new address in a new colour. Instructions in the branch ‘shadow’, i.e. fetched speculatively beyond the branch, will be in the old colour; they are detected and discarded by the execution stage.

In AMULET2 only two colours (represented by a single colour bit) are required because a second branch cannot be taken until the new instruction stream arrives.

4.3: Prefetch aborts

Prefetch aborts are detected by the processor before instruction issue when the aborted instruction reaches the primary decoder. This enables the exception mechanism to operate during the decode stage as it does for software interrupts (SWIs), undefined instructions and external interrupts.

When a prefetch abort occurs the memory returns an (invalid) data word marked by an abort flag. On entry to the decoder the prefetch abort flag causes the op-code to be ignored and an exception entry substituted. This ensures a clean entry to the exception handler; following instructions – which also may be potential aborts – lie in the shadow of the first and so are discarded by the colour mechanism in the same manner as instructions following a branch operation.

4.4: Data aborts

The implementation of an exception mechanism for data aborts is more complex as the offending instruction is already in the execution unit, actively engaged in changing the processor’s state when the exception occurs.

AMULET2 adopts the simplest available strategy; no speculation. For every memory access the processor checks with the memory management unit (MMU) to ensure that it will complete successfully before the next instruction is started. Conceptually this brings the MMU into the execution pipeline, although it is still a physically separate unit; figure 3 illustrates the arrangement. The cycle time must be stretched to accommodate this, resulting in a slower ‘execution’ cycle.

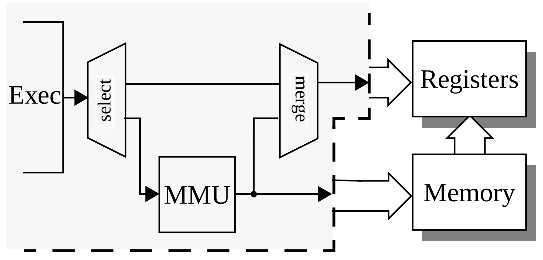

Figure 3: Incorporation of MMU into the execute stage

The memory timing is divided into two sequential stages: exception detection and data access. During the exception detection stage the MMU is presented with the required operation. It must respond with a fault/no-fault result indicating whether or not that operation caused an exception. If no exception occurred the execute unit proceeds to the data access stage where the transfer of data is performed. If the response is a data abort then the processor needs to save sufficient information to restart the instruction and then it enters the exception handler.

Entry to the exception handler causes a change in instruction stream; this is handled identically to any other branch operation, changing the ‘colour’ of the instruction fetches and discarding any erroneously fetched instructions.

To restart the aborted instruction the processor needs to obtain the memory address which faulted (contained in the base register) and the return address from the exception. The return address is obtained by piping the PC to the execution unit for any instruction which *may* cause a data abort (i.e. any data transfer). The piped PC value is normally discarded when the MMU signals that the cycle is to proceed, but it can be salvaged in the case of an abort. To simplify the design the ‘base updated’ approach is used for base register preservation, with the consequence that the memory address has to be recalculated by the abort handler. A final complication is that a load multiple (LDM) instruction may still overwrite the base register before aborting on a later cycle. To allow recovery in such cases the base address for

LDMs is preserved in a special register until after the last cycle of the operation, when it is discarded or restored in an additional final operation.

Whilst this scheme is successful it has several drawbacks. Firstly it is slow because the execution stage always waits for an external response for every address issued. Furthermore (unless an extreme penalty is paid) the memory system supports only early aborts: the page fault can be signalled only at the start of the memory cycle, disallowing bus errors and timeouts. Finally the mechanism for recovery from aborted LDMs is complex, requiring extra cycles and thus, again, sacrificing performance even when no abort occurs.

5: AMULET3

AMULET3 is the third generation asynchronous ARM processor and is aimed at a significantly higher performance than its predecessors. Achieving this performance required a redesign of the data abort exception mechanism to overcome the inherent limitations found in the design for AMULET2. In AMULET3 performance is increased by allowing each memory cycle to complete before signalling its abort response, removing the MMU from the execute stage (and the critical path), and, most importantly, allowing speculation on memory accesses. The reorganization of the execute stage is shown in figure 4. This arrangement allows AMULET3 to support complex memory management hardware without penalty and permits it to support late aborts: a page fault can be signalled at the end of the memory cycle, allowing bus errors and timeouts.

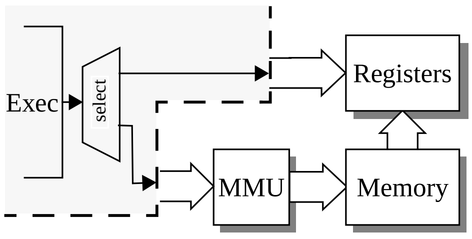

Figure 4: Removal of MMU from execute stage

The rationale for this new approach is that, because data aborts occur only infrequently, it is more productive to speculate on the memory cycle completing sucessfully and only consider data aborts if and when they occur, even if a performance penalty is paid in this circumstance. This requires a mechanism which allows instructions subsequent to a memory access to 'complete', but permits the recovery of the system state, at the point of a memory access, if that memory access is later found to have aborted. To achieve an efficient implementation for the AMULET3 abort exception mechanism it was found that several different techniques for preserving state were required. The following sections detail the selection and operation of the various state preserving mechanisms.

5.1: Reorder buffer

An asynchronous reorder buffer [9, 10] protects the registers' state in AMULET3. The reorder buffer allows internal operations to be speculated upon whilst memory cycles proceed. It provides a temporary store for results from speculatively executed instructions, allowing these stored results to be forwarded to following speculative instructions, and controls the updating of the register bank state.

The position of the reorder buffer within AMULET3 is shown in figure 5. As prefetched instructions pass through the decoder, storage locations in the reorder buffer, called 'slots', are assigned for their results. When an instruction has been issued it must therefore continue until its result(s) reach the reorder buffer. Note that, in addition to arithmetic results and loaded data, a slot is provided for the token returned by a store operation; this is necessary both to keep the loads and stores in order and to provide a space in the buffer for the return address in case a store operation aborts.

Results arrive, at the reorder buffer, both from the data interface as it returns values from memory and from arithmetic operations performed in the execute unit. These two streams are un-synchronized and so the results may arrive out-of-order.

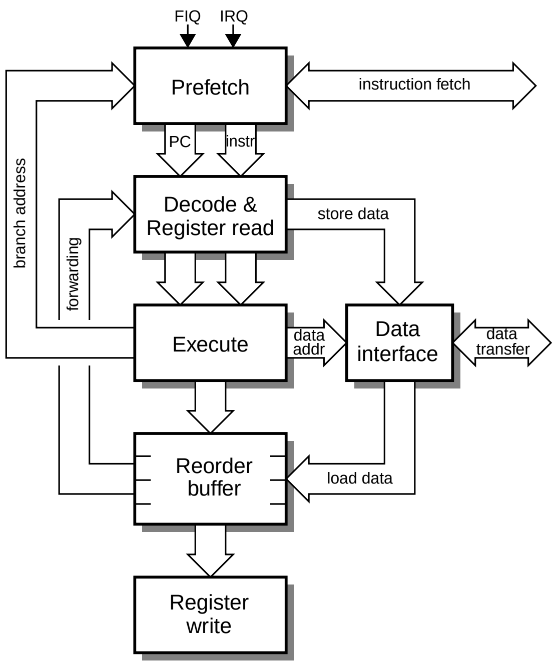

Figure 5: AMULET3 core organisation.

To safeguard the state of the register bank it is not updated until all preceding operations have completed correctly; the results are written back to the register bank *in issue order*. Register update is controlled by the writeback process which ensures that only valid results are written back, invalid ones being discarded. A result may be invalidated if its instruction fails its condition code test, or falls in the shadow of a branch or a data abort. The validity status is passed with the result to the reorder buffer by each unit. If a data abort *does* occur any contents of the reorder buffer subsequent to this point are abandoned and the processor registers' state may be recovered from the register bank.

5.2: Extending the colour mechanism

The AMULET2 processor required only a single colour bit to manage its instruction streams. This is not sufficient for a processor which can support speculation on memory accesses. The shortcomings can be demonstrated by considering the execution of the following code fragment:

```
LDR  R1, [bad_address]
BNE  somewhere
ADD  R2, R3, R4
```

Internal operations tend to be completed faster than external memory accesses, so the branch operation could be performed before the memory system can return the abort response. Consequently the entry to the abort exception handler may occur before the instructions from the branch target arrive. In these circumstances there will be three streams of instructions:

- the original stream containing the load and branch
- the stream from the branch target
- the stream from the exception handler

Both the original stream and that fetched from the branch target need to be discarded as they lie in the abort shadow. As there are three instruction streams at least three colours are needed. The simplest solution is to use two independent colour bits: the 'branch colour' bit used in AMULET2 is retained, and a new 'abort colour' bit is added. The abort bit toggles on the occurrence of a data abort and the branch bit toggles, as before, on the occurrence of branches and exceptions other than data aborts. Only when both colour bits match at the execute stage is the instruction performed.

This is adequate to protect the registers' state but it is insufficient to protect the state of the memory itself. Consider the following code fragment:

```
LDR  R1, [bad address]
STR  R2, [good address]
```

If the memory is pipelined it is possible that the seemingly valid store operation is issued before the processor realises that its predecessor has aborted. It is important that the store operation does not change the system's state. This cannot be directly enforced by the processor because both operations could have been dispatched to memory before the first fault is discovered. The memory system must therefore be responsible for managing its own state and must be able to distinguish operations occurring before, during, and after the abort shadow. This is accomplished by passing the abort colour to the MMU (figure 6).

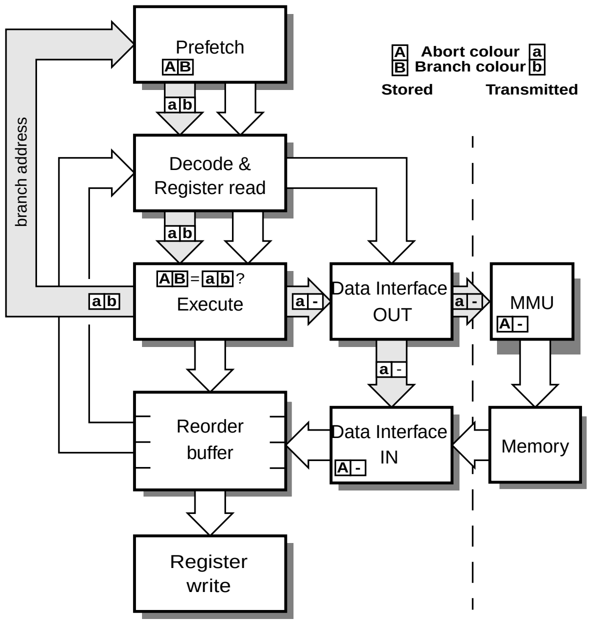

Figure 6: The flow of colour information through the core

The MMU initiates the abort and so is able to recognize the start of the abort shadow. It reacts by recording the current abort colour and suppressing any state changes (mainly store operations) until the new instruction sequence arrives. Execution resumes when operations arrive with a mismatched colour, which indicates that the instruction stream has changed. This algorithm is summarised in figure 7.

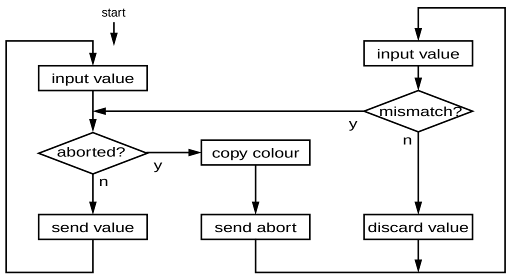

Figure 7: Abort shadow algorithm

Note that, because only the abort colour bit is used, an indeterminate number of branches may be accommodated in the interim; this is important because not all instructions communicate with the memory subsystem.

A similar mechanism is used within the processor to ensure that only one abort is taken at a time; values returned from memory are checked and marked as invalid by the data interface if they are within the abort shadow.

5.3: Invalidating results

To maintain the register state following a data abort the processor must discard speculative results stored in the reorder buffer and those which may arrive subsequently. This could be done by passing the abort colour through the reorder buffer to the register writeback process and discarding results by means of the method outlined in section 5.2, but this has a number of significant drawbacks. Firstly, it would be necessary to perform an additional test (a colour match) on every result before it can be used to update the register bank, slowing down the writeback process. Secondly, the reorder buffer must be extended to allow storage of the abort colour information. Finally, the processor has to wait for the aborting result to be written to its reorder buffer slot, and then for the writeback process to reach this slot, before abort exception entry is initiated. This results in a high latency for abort entry.

A more convenient method exploits the existing "invalid" flag used for tagging operations which fail their condition test. Loaded values following the abort will be invalidated by the data interface (described in section 5.2). Internally generated results will also arrive marked as invalid *after* the arrival of a data abort 'interrupts' the execution unit and changes the colour. Thus only results from instructions which were present *before* the abort arrived may have the incorrect validity.

The reorder buffer is allowed to drain naturally until writeback reaches the aborted value, after which the remaining results in the reorder buffer are invalidated by broadcasting a global invalidation signal. The global invalidation is realised as a simple clear of a set of latches and was chosen for its implementation simplicity. However it also presents a danger.

In an asynchronous system the forwarding process may attempt to read the value of the validity flag simultaneously with the clear operation. This is not a logical problem since any forwarded data will be discarded later anyway, but could cause implementational problems by introducing metastability in later latches, possibly impairing the correct operation of the control circuitry.

To avoid this hazard, two validity flags are stored, one for each of the two possible abort colours (a flag is provided for current stream colour and a separate flag for the future exception handler colour). Both of these are written to in parallel and both must be valid in order for writeback to return a value to the register bank, but only one is needed to forward data. The global invalidation process only affects the flag *not* read by the current (potentially forwarding) process, hence the validity flag used by the forwarding process is not changed and potential problems with metastability are avoided.

The global invalidation mechanism locks the instruction decoder preventing entry by prefetched instructions from the exception handler until the invalidation process is complete and the reorder buffer is stable. This solves two problems. Firstly, it prevents speculative results invalidated by the abort from being forwarded to instructions from the exception handler. Secondly, it ensures that the valid results produced by the execution of instructions from the exception handler are not accidentally invalidated by the global invalidation.

5.4: PC recovery

To allow the program to restart a return address from the data abort handler is required. AMULET3, in similar fashion to the earlier AMULET processors, matches each instruction with its PC value (see section 4.1), however, for loads and stores the PC value must now be retained until the memory transfer is complete. When a data transfer instruction begins execution the associated PC value is copied into a history buffer (referred to as the exception pipeline or Xpipe) as shown in figure 8.

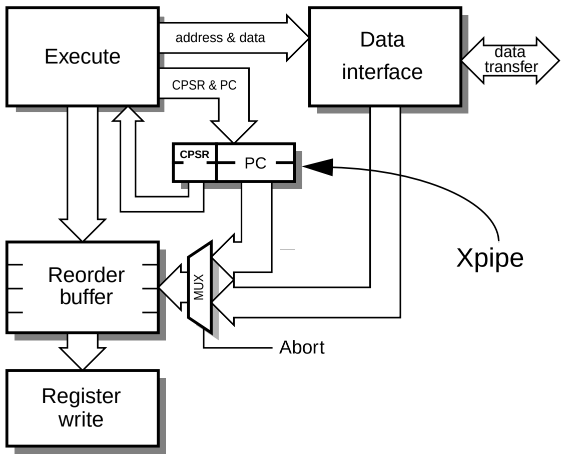

Figure 8: PC & CPSR history buffer

When a transfer completes successfully the Xpipe entry is discarded. If, however, an abort occurs the PC is copied into the reorder buffer, 'hijacking' the slot originally assigned for the results from the memory access. As this slot will have been previously assigned to another register for its results, the 'hijacking' is invisible to the forwarding process which will still be allowed to read this value as if the load had been successful; this allows any forwarding to proceed and, although the wrong value will be used, this presents no problem because it will be discarded later.

Note that the multiple data transfer (LDM/STM) instructions use multiple slots but only require a single PC value. Although, in theory, it would be possible for STM to allocate only a single reorder buffer slot this complicates the implementation to little advantage.

As can be seen from figure 8 the PC is accompanied in the history buffer by the CPSR; this is described and justified in section 5.6.

5.5: Base register restoration

AMULET3 provides the *base restored* value to the exception handler. In the ARM programmer's model some instructions, such as single register load from memory with writeback, produce more than one result. In this case two results are produced: the loaded data from memory and the updated base register value. Generally, the updated base register value will be available before the data is returned from memory and so is traditionally written to the register bank first.

The reorder buffer allocation implies a specific ordering to the writeback. Swapping the order of writeback enables AMULET3 to return the updated base-register value *after* the data loaded from memory; forwarding removes any dependency stalls from this process. If a data abort occurs the reorder buffer will thus prevent the updated base register value from being written back, removing the need for restoration work which would have previously been required to enable the instruction to be re-run.

Whilst adequate for single register loads this mechanism is insufficient to preserve the base register if an abort occurs during a multiple register load (LDM) instruction. Such an instruction can load any set of the currently available registers (permitting the base register to be overwritten), and may abort on any of the memory cycles.

In principle the reorder buffer could be used as a temporary store for these values but this would require the reorder buffer to accommodate up to sixteen values which is impractical for this unlikely circumstance.

An alternative, low-cost solution is to use another history buffer. This is used to save the original value of the base register, allowing it to be restored (and so the state to be rolled-back) on the occurrence of an exception. A copy of the unmodified base register value together with a tag identifying the register is transferred by the execute unit into a special FIFO called the *base-restore pipe* (BR-pipe) whenever the processor is required to perform a LDM instruction (see figure 9). If the LDM completes successfully, the corresponding values in the pipe can be discarded.

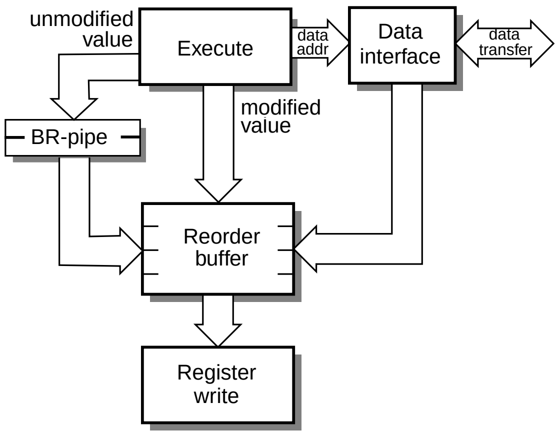

Figure 9: Base restore pipe.

In the event of an abort, the original value can be recovered from the BR-pipe and written back to the source register.

The writeback process requires the insertion of a writeback cycle, from the reorder buffer to the register bank which is not accommodated by the reorder buffer. This can be performed without hazard, because it will be the last *valid* operation before the exception handler is entered, as long as it is completed before the registers are read in the exception handler. The asynchronous nature of the processor allows the writeback cycle to be 'stretched' in this case. The value written back will be the unmodified base register value, which is consistent with the base restored model.

5.6: CPSR preservation

The mechanisms described thus far preserve or recover the memory state and the processor's register state in the event of a data abort. There is some remaining state in the ARM architecture which is not stored in the register bank but which must be preserved.

This remaining state divides into the Current Program Status Register (CPSR) and the bank of Saved Program Status Registers (SPSRs). The CPSR contains the flag register, current operating mode (user, supervisor, etc.), and interrupt enable flags. As it contains the processor's flags it is altered frequently. It is usually changed just before it is used; for example:

```
CMP  R1, #29        ; CPSR update
BGT  somewhere_else ; condition test
```

Using a reorder buffer to resolve such dependencies for these few bits is expensive and, potentially, sacrifices performance. Instead the CPSR is changed immediately for each instruction and a separate mechanism (figure 8) is used to restore the correct state in the rare event of a data abort.

Every memory reference dispatched records the state of the CPSR (this can be compressed into 10 bits) which is placed, with the PC of the operation, in the X-pipe. When the memory reference completes successfully this is discarded, but if an abort occurs the CPSR may be restored, whilst the PC is sent to the reorder buffer to form the link value.

5.7: SPSR preservation

The Saved Program Status Registers (SPSRs) are copies of the CPSR which are used to preserve status information when a different mode is entered (such as on interrupt entry). There are five SPSRs – each of 10 bits – which change very rarely, although in theory any or all could be written to just after a data transfer is dispatched. In practice they are generally changed on entry to a new operating mode. For example:

```
LDR  R1, [bad address]
SWI  something ; software interrupt
```

From user mode the SWI (software interrupt) instruction could switch into supervisor mode and copy the user mode CPSR into its SPSR before the memory reports the abort. The exception mechanism for the subsequent data abort must ensure that both the user mode CPSR and the supervisor mode SPSR are correctly restored.

Two possible methods of history buffering could be applied: the first is to buffer all the SPSR bits for every instruction; the second is to buffer each former SPSR value as it is overwritten. The first method is very expensive in hardware, the second is exceedingly complex to recover from.

Instead it was decided that it is adequate to delay any alterations to the SPSRs until it is known that no potential aborts are outstanding. This has almost no performance impact [9] and is cheap in circuit terms. It is implemented using an asynchronous semaphore circuit based on the locking mechanism designed for the register bank on AMULET1 [11]. This is a dataless FIFO which is incremented to signal that a data transfer operation has started and decremented when it is complete; a Boolean output indicates whether the FIFO is empty or not (figure 10). SPSR changes are delayed until this semaphore is empty. Despite the asynchronous operation of the increment, decrement and read processes the only potential danger could occur if the semaphore was incremented into a ‘not-empty’ state whilst being sampled. In practice, this cannot happen because no ARM instruction can cause both a memory transfer *and* modify the SPSR. Thus in the above code fragment the SWI operation could despatch its target address but would not change the operating mode and allow further instructions to flow until and unless the LDR completed successfully.

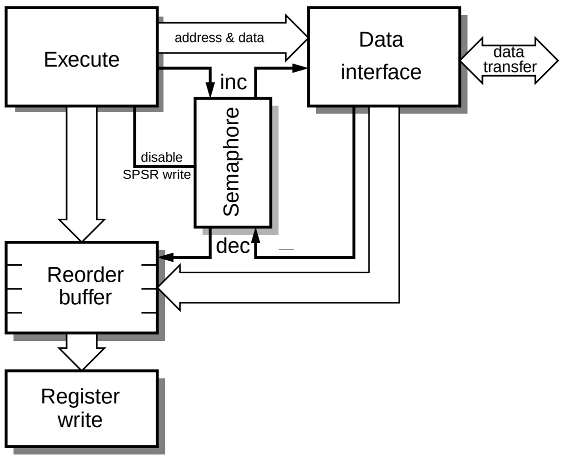

Figure 10: SPSR lock semaphore

6. Conclusions

Whilst inconvenient, exception support is essential in a commercial processor architecture. Most exceptions can be treated as instructions and thus are relatively simple to process. However late occurring exceptions such as arithmetic and memory faults need special treatment.

This paper has described the treatment of memory faults in AMULET3 as an example of how such late occurring exceptions may be handled to preserve or recover the system state in a high performance asynchronous architecture. The three mechanisms used are:

- a reorder buffer for the large number of general registers
- history buffering for state with many dependencies
- semaphore locking for infrequently changed state

The combination of these methods represents an advance on previous schemes, where only the last was used. The ability to mix and match these as appropriate has provided opportunities to significantly streamline the memory interface.

The methods described here are generic, with wider applications. One good example is the support of the ARM debug architecture, which allows the setting of breakpoints and watchpoints on the memory buses. Breakpoints cause a form of exception when a certain address or data pattern occurs during an instruction fetch; a watchpoint performs the same service during a data read or write. By requiring a comparison on any returned data the watchpoint cannot begin to signal the exception until both the read and the comparision are complete. The read will not complete until very late in the memory cycle and then the comparison will require additional time. The solution to this in ARM7 was to allow an extra half cycle for the return of this information; in AMULET3 the reorder buffer write can be started - allowing the data to be forwarded - whilst stretching the existing cycle if a comparison is required. The watchpoint exception can then be guaranteed to occur at precisely the correct point. Such a 'late abort' would be impossible with the mechanism used in AMULET2.

Techniques similar to the reorder buffer may be used in a future asynchronous cache/memory interface. In this application it is possible that write operations to a slow external memory could be queued and overtaken by more urgent data reads. The write buffer would need to ensure memory consistency in the event of page faults etc. and would need to forward values to any read requests attempting to pre-empt access to relevant locations. These ideas form the basis for ongoing studies.

Mechanisms of this type are already widely used in high-performance synchronous processors. This work has shown that, despite the lack of global synchronisation, these techniques can be adapted equally well to the asynchronous world.

7. Acknowledgements

The development of AMULET3 has been support primarily within the EU-funded OMI-DE2 and OMI-ATOM projects, and authors are grateful to the European Commission for their continuing support for this work. ARM Limited coordinated these projects; their support, and that of the other project partners, is also acknowledged.

Aspects of the work have benefited from support from the UK government through the EPSRC, particularly in the form of a PhD studentship and the tools development funded under ROPA grant GR/K61913.

The VLSI design work has leant heavily on CAD tools from Compass Design Automation (now part of Avant!) and EPIC Design Technology, Inc. (now part of Synopsis).

The authors would also like to thank other members of the AMULET research group at Manchester University.

8. References

[1] 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.

[2] van Gageldonk, H., "An Asynchronous Low-Power 80C51 Microcontroller", IPA Dissertation Series, 1998. ISBN 90-74445-42-X.

[3] Richardson, W.F. & Brunvand, E., "Precise Exception Handling for a Self-Timed Processor", 1995 IEEE International Conference on Computer Design: VLSI in Computers & Processors. October 1995.

[4] Jaggar, D., "Advanced RISC Machines Architecture Reference Manual". Prentice Hall, 1996. ISBN 0-13-736299-4.

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

[6] 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.

[7] 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

[8] Renaudin, M., Vivet, P. and Robin, F., "ASPRO-216: A Standard-Cell QDI 16-Bit RISC Asynchronous Microprocessor", Proc. Async'98 pp. 22-31, San Diego, April 1998.

[9] Gilbert, D.A., "Dependency and Exception Handling in an Asynchronous Microprocessor", PhD Thesis, University of Manchester, 1997.

[10] Gilbert, D.A., Garside, J.D., "A Result Forwarding Mechanism for Asynchronous Pipelined Systems", Proc. Async'97, Eindhoven, Netherlands, April 1997, pp. 2-11.

[11] 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.