AMULET3 Revealed

J. D. Garside, S. B. Furber and S-H Chung
Department of Computer Science, The University of Manchester,
Oxford Road, Manchester M13 9PL, UK.
{jgarside, sfurber, chungsh}@cs.man.ac.uk

Abstract

AMULET3 is the third fully asynchronous implementation of the ARM architecture designed at the University of Manchester. It implements the most recent version of the ARM architecture (v4T), including the Thumb instruction set. Significant architectural changes from its predecessors help achieve higher performance without sacrificing the advantages of asynchronous design and some new power-saving features have been incorporated.

This paper outlines the AMULET3 microprocessor core, highlighting where this design differs from its predecessors. Most notable among the changes are the use of a Harvard architecture to increase memory bandwidth and the inclusion of a reorder buffer to handle data forwarding and memory faults.

1: Introduction

AMULET3 (figure 1) is the third asynchronous implementation of the ARM architecture [1] to be produced at the University of Manchester. Its predecessors, AMULET1 [2] and AMULET2 [3], were intended to demonstrate that asynchronous circuits of this complexity are feasible and practicable; AMULET3 has been designed to be a commercially competitive macrocell. It is therefore required to deliver a performance similar to that of the contemporary synchronous ARM, the ARM9TDMI [4], and to implement the most recent version (v4T [5]) of the instruction set architecture including the 16-bit Thumb instruction set [6]. AMULET3 is being implemented in the same generic 0.35 µm 3 metal layer process as the ARM9TDMI. This implies a performance target of well over 100 MIPS (measured with Dhrystone 2.1), compared to the 40 MIPS delivered by AMULET2e on a 0.5 µm process.

Achieving this performance has necessitated a considerably different microarchitecture from the earlier AMULET processors. AMULET3 has been influenced by aspects of several recent microprocessors, most notably the StrongARM [7] developed by Digital and now owned by Intel.

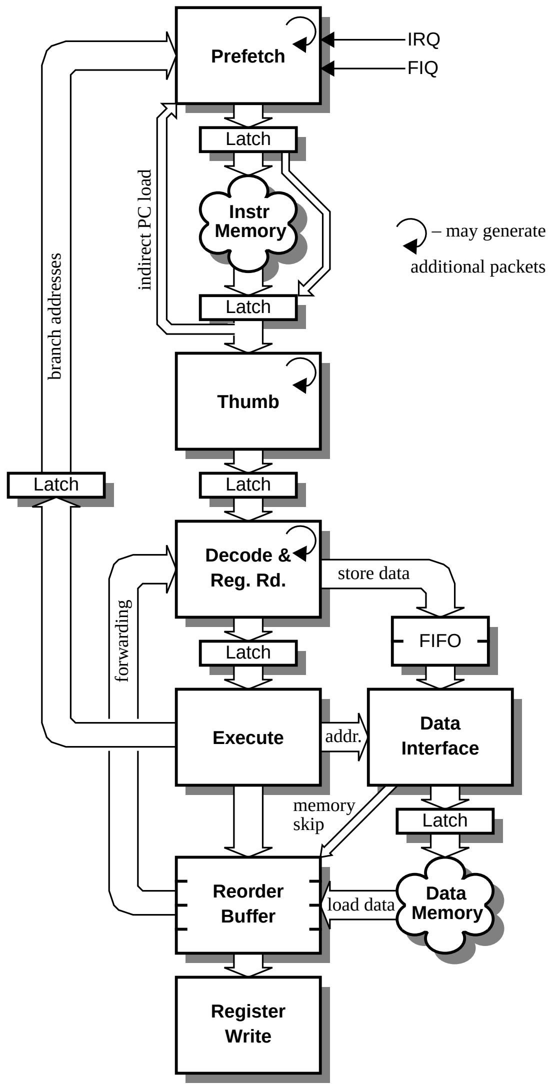

Figure 1: AMULET3 internal overview

Some of these changes – such as the addition of a third register read port – are not innovative, although they have a considerable bearing on the processor's performance; oth-

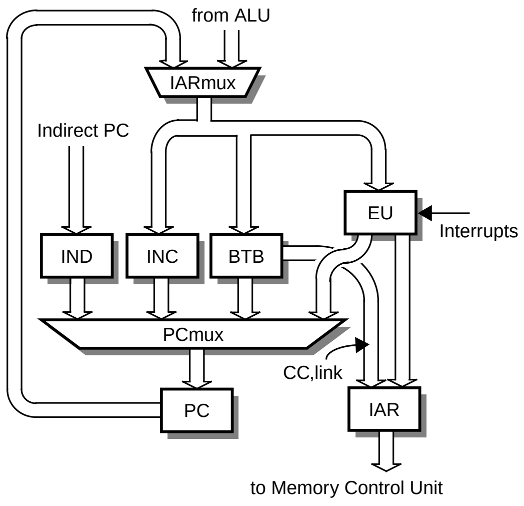

Figure 2: Prefetch unit organization

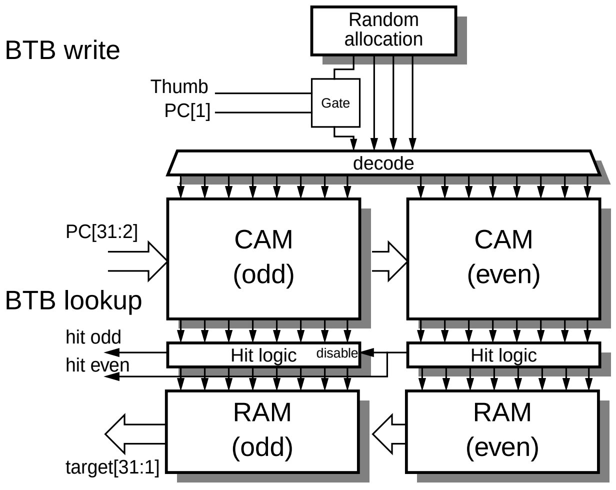

Figure 3: BTB CAM organization

ers – such as the reorder buffer – are asynchronous implementations of techniques widely used in synchronous processors, and some we believe to be wholly new.

As in the previous AMULET processors the architectural design is based on an asynchronous Micropipeline [8] structure using four-phase [9] control signals. Figure 1 shows the basic pipeline stages although it excludes details of the external memories and their interfaces (which may also be pipelined to an arbitrary depth). Each of these stages will be described briefly with most attention being given to the more novel mechanisms.

2: Prefetch unit

The prefetch unit (figure 2) is responsible for generating addresses for the instruction memory which are sent via the Instruction Address Register ('IAR' in figure 2). The prefetch unit has a highly parallel organization, speculatively computing the outcome of all scenarios in parallel and then selecting the appropriate course of action in the final multiplexer. Although such speculation causes some unnecessary activity (and therefore wastes power) it is necessary here to meet the required throughput.

Usually the output addresses form an ascending sequence and are provided by a simple loop containing an incrementer ('INC'). When a branch occurs this loop may be interrupted asynchronously (via an arbiter) and loaded with a new address from the ALU. However in AMULET3 several other functions are performed here also.

2.1: Branch prediction

Branches disturb program flow and incur a considerable penalty in deeply pipelined systems. Sophisticated branch prediction mechanisms are now in use on state-of-the-art processors, but even a relatively simple branch predictor can significantly reduce pipeline disruption.

AMULET3 uses the same branch prediction as AMULET2 [10], namely a Branch Target Buffer ('BTB') which predicts a previously-taken invariant branch as 'always taken' until it is displaced from the BTB by a new entry. However there are two significant differences between the BTB in AMULET2 and that in AMULET3.

The AMULET2 BTB records an address containing a branch instruction together with its target address. However, if a branch instruction address subsequently hits in the BTB the instruction is still fetched from memory and executed as it may be conditional and it may require a return address saving (if it is a BL – Branch-and-Link – instruction, used for procedure entry).

The information which AMULET2 gets from memory when it fetches a predicted branch amounts to only five bits (four condition bits and the 'L' bit). In AMULET3 these five bits are stored in the BTB so that the instruction does not have to be fetched in repeat encounters and the instruction memory may be bypassed. As branches account for 10%-15% of ARM instructions [11], and the majority of these are cached in the BTB [10], this reduces the number of instruction fetches, yielding both a considerable power saving and a potential speed advantage (exploited automatically by the asynchronous pipeline).

The second BTB difference from AMULET2 is due to the presence of the Thumb decoder. ARM instructions are fetched as 32-bit words. When running Thumb code a choice must be made whether to fetch the 16-bit Thumb instructions individually or in pairs. As the speed and power consumption of a memory cycle is almost independent of the transfer size the decision was made to fetch Thumb instructions in pairs. However, as either or both of these instructions may be cached branches, the BTB must be able to cope with zero, one or two simultaneous hits.

This is achieved by splitting the BTB Content Addressable Memory (CAM) into two sections (see figure 3). In Thumb mode each section works with one half word of the instruction pair; any potential conflicts are resolved by taking the 'even' half word (the Thumb instruction at the lower address) prediction because this will always be first in the instruction sequence.

When running ARM code the two sections are merged. This allows the BTB to cache a mixture of ARM and Thumb branches simultaneously without compromising the number of usable entries in either case

2.2: Halting and interrupts

Most current CMOS technology dissipates very little power when not switching. This has been exploited in AMULET2e by causing the system to halt when no useful work can be performed, with demonstrable power-efficiency benefits [3].

Halting an asynchronous pipeline at any point soon causes the whole system to halt. Because there is no free running clock this reduces the number of transitions – and hence the power consumption – to near zero. In a synchronous system the clock oscillator could be stopped, but this is quite a complex procedure. The asynchronous system also recovers quickly (as there is no clock to restart). AMULET2e and AMULET3 exploit this by decoding a branch back to itself as a 'Halt' instruction and use this to stall the pipeline; this is fully compatible with much existing ARM code. The halt state is exited by the assertion of an enabled interrupt.

As alluded to above, the stall can occur anywhere within the pipeline. AMULET2, for example, stalls in the execution stage. AMULET3 adopts a somewhat cleaner model by stalling at the prefetch stage. This means that the processor restarts with an empty pipeline which provides the fastest possible response, any 'rubbish' being cleared out at halt entry.

Scrutiny of figure 1 reveals an interesting consequence of this operation – the interrupt signals are fed into the prefetch unit rather than the instruction decoder. This rather unusual feature provides both a clean interrupt model and a low interrupt latency. When an (enabled) interrupt is asserted it is arbitrated into the prefetch cycle and treated much like a predicted 'BL'. The interrupt 'hijacks' an instruction address, bypasses the memory, and proceeds down the execution path to save the return address. The PC is loaded with the address of the service routine (which is a constant, generated in the Exception Unit, 'EU' in figure 2) in parallel and the prefetching of this code begins immediately.

A consequence of this approach is that the prefetch unit must store an up-to-date copy of the interrupt enable status. One danger is that this may be out of date because an operation already prefetched may change it. Another, related, problem is that the hijacked address may be in the 'shadow' of a branch and the interrupt may try to save an incorrect return address.

Both these problems are solved by treating control instructions (such as enabling/disabling interrupts) as branches, and branches as potential control instructions. This is not particularly onerous because almost all instructions which can alter these flags (e.g. software interrupts, return from interrupts, etc.) also cause flow changes anyway. If an interrupt has occurred in a branch shadow it will be discarded in the same way as any other erroneously prefetched instruction. Concurrently, the branch will reach the prefetch unit, re-enable interrupts, and immediately cause the interrupt entry mechanism to repeat, this time saving the branch target as the return address.

2.3: Indirect branches

ARM programs often load the Program Counter (PC) directly from memory as part of a subroutine return (and, less frequently, as a result of a jump table lookup). Typically a subroutine return restores the PC together with a set of working registers using a load multiple (LDM) instruction. The load ordering is such that the lowest numbered register is loaded first, and thus the PC (R15) is loaded last. This delays the start of instruction fetching from the return address and compromises performance.

AMULET3 incorporates an optimization which exploits the separate instruction memory port (see figure 1). The execution unit passes the load address of the PC value back to the prefetch unit via the branch address path in parallel with initiating the other register transfers in the data interface. This 'branch' terminates prefetching from the redundant instruction stream and prompts a single read cycle which fetches the new PC. This is then returned to the prefetch unit (via 'IND' in figure 2) where instruction fetching resumes. With a typical subroutine return much of this should happen whilst the data transfers are proceeding and so the new instructions should be available before the LDM has completed.

Note that this feature imposes a significant constraint on the memory designer: the instruction and data memories must be coherent because a PC value is stored via the data port and then read via the instruction port. The first AMULET3-based system has a unified memory which is dual-ported to give independent instruction and data ports. Coherence is therefore not an issue here.

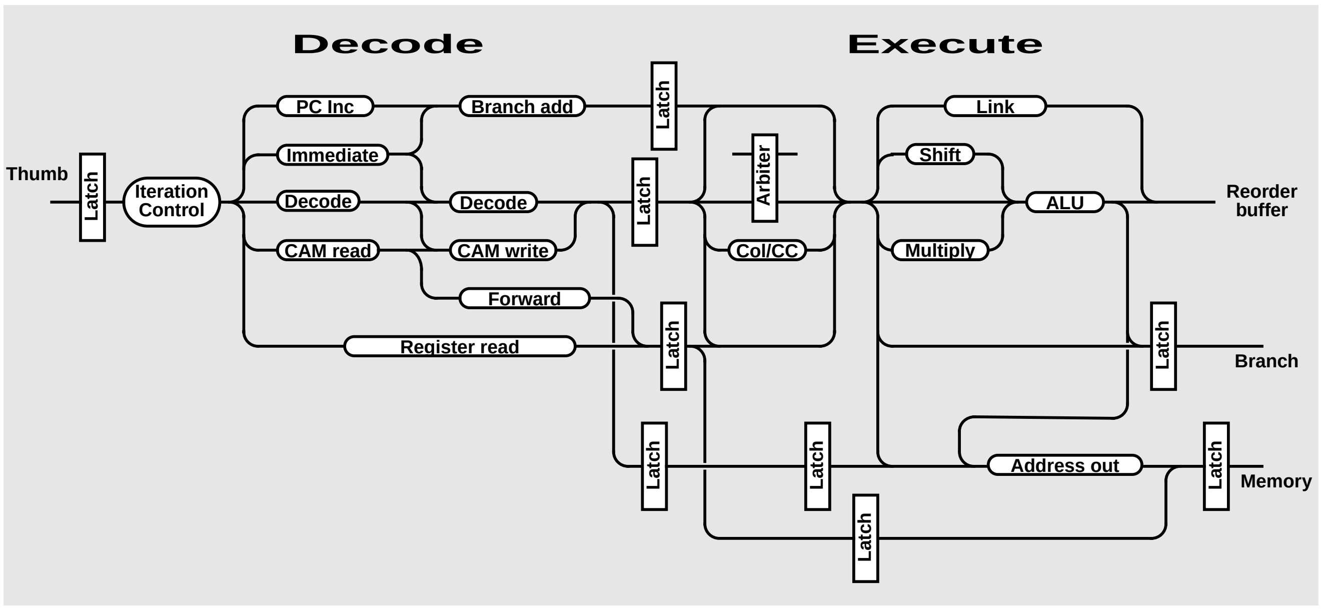

Figure 4: Data flows in the decode/execution pipeline

3: Thumb decoder

The Thumb instruction set [6] is intended to allow higher code density, which is especially important in many embedded systems. It is not a complete instruction set architecture but a compression to 16 bits of many of the more frequently used ARM instructions. With a few minor additions to the execution stage Thumb instructions can be expanded into 32-bit ARM instructions.

The Thumb decode stage in AMULET3 receives 32-bit instruction packets which may contain a single 32-bit ARM instruction, two 16-bit Thumb instructions or an exception case such as a predicted branch or an ‘interrupt service instruction’ as described in sections 2.1 and 2.2. In most cases this packet undergoes little processing here and is passed rapidly forwards to the main instruction decoder. However if it is marked as containing Thumb instructions this stage will perform two (slower) cycles, expanding each Thumb instruction for the main decoder in turn. Because of the asynchronous nature of the pipeline it accommodates this ‘one-in, two-out’ behaviour automatically.

This can be contrasted with the approach used in the ARM7TDMI and ARM9TDMI which fetch Thumb code one 16-bit instruction at a time, thus smoothing the pipeline flow but doubling the number of external cycles performed.

4: Instruction decoder

The instruction decoder represents much of the complexity of the system. Figure 4 illustrates the main data flows through the instruction decoder and subsequent pipeline stages. As can be seen there is considerable parallelism (although not all paths are invoked for every instruction). Note that the latch ‘stage’ between the decoder and the execution unit – one of the widest paths in the processor – comprises several disparate elements which may operate at (slightly) different times. This helps to reduce penalties due to forwarding (register operands may arrive slightly later than the decoded instructions without penalty) and is also an attempt to reduce emitted electro-magnetic noise by skewing the latch enable times. (Only one register read/forward path is shown in figure 4; the processor contains three).

Despite its complexity relatively little of the instruction decoder design is novel or unusual. The two features which may be of interest are the register bank/reorder buffer – used here for register forwarding – and the mechanism for supporting the multi-cycle load (LDM) and store (STM) instructions.

4.1: Register bank

The register bank in AMULET2 has two read ports and a single write port, although the latter is shared by two incoming data streams (from the ALU and the memory) using an arbitrating multiplexer. This port structure is a reasonable match for the ARM6 instruction set, although several instructions require more than two register operands and therefore need two (or more) decode cycles. It uses the register ports efficiently, but imposes an overhead in the control logic.

AMULET3’s register bank has three independent read ports. This increases the physical size of the unit but, because these may be invoked or bypassed for any given cycle, only the required registers are read so little extra power is consumed and artificial register dependencies are avoided. This means that almost all instructions can be executed in a single cycle, reducing the control overhead in this, and other, stages.

The register bank is tied closely to the reorder buffer (see section 7.1). When an instruction arrives its register operands are both read directly and checked (in a CAM) to see if they should be forwarded from outstanding results. Any forwarding is asynchronous, and will override any value from the register bank. Naturally, results which are invalid – for example from instructions which fail their condition code test – are not forwarded.

The register forwarding process can wait for an arbitrary time before delivering data (or deciding that the operation will not complete). The instruction will stall until all its data fields are filled. This mechanism therefore replaces the register locking mechanism [12] used in AMULET1 and AMULET2; an added advantage is that forwarding from the reorder buffer is faster than passing values through the register bank and more flexible than the limited forwarding provided by AMULET2 [3], thus leading to increased performance.

The reorder buffer also provides several independent register write ports; these are discussed in section 7.1.

4.2: Multiple register loads and stores

A ‘normal’ (i.e. single register) ARM load instruction generates a single memory transfer following an address calculation. Thus there is a need to invoke both the execution unit (to calculate the address) and the data interface.

A multi-register transfer performs all its data transfers from contiguous addresses. This means that the execution unit need only be invoked on the first cycle (figure 5), subsequent addresses being generated in the data interface. This saves cycling the execution unit unnecessarily (saving power) and can exploit efficiently the resulting sequential address sequence (e.g. for page mode transfers). An added bonus is that, because the relatively high memory latency causes an instruction backlog, a small amount of buffering on this pipeline can allow subsequent (non-dependent) instructions to begin flowing again even before the last memory transfer cycles commence (see also section 2.3).

All ARM instructions are conditional, so an LDM may fail its condition code check and be discarded. It is clearly more efficient to discard a multi-cycle instruction as a single unit rather than as many cycles, but – unfortunately – the condition information is not known at the instruction decode stage. To alleviate this problem an additional handshake is used. The instruction decoder despatches the first cycle of the operation to both the execution and data interface units and then pauses. Whilst the address is being sent between these units a separate pipeline sends a ‘go/no go’ signal back to the decoder which can then continue or proceed to the next instruction. This imposes a small time penalty but the cycle is lengthened by the data interface unit timing in any case.

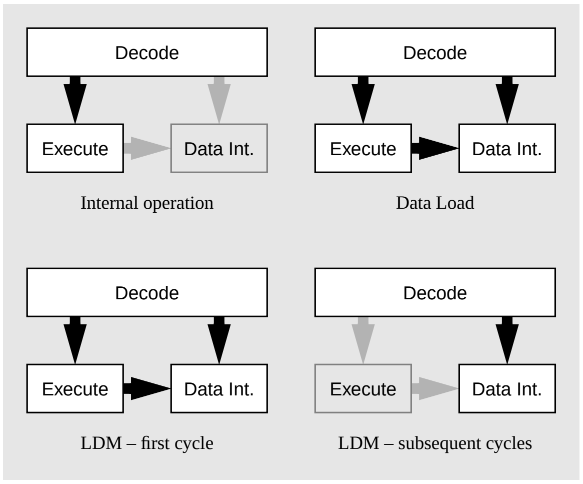

Figure 5: Sub-instruction routeing

5: Execution unit

The execution unit is responsible for computing results, making decisions and despatching values to their destinations. To support the ARM instruction set it contains a barrel shifter and iterative multiplier in series with the ALU. The shifter and multiplier are used in only a few instructions so they are normally bypassed (see figure 4); the execution cycle is stretched when they are required, again exploiting the asynchronous nature of the processor.

Execution begins by deciding whether the instruction is to be completed; this is a result of its condition code test (all ARM instructions are conditional) and a ‘colour’ test to see if the instruction is in a branch shadow. In parallel with this the required operands are collected and an arbiter is sampled (the arbiter allows page faults from the data memory to be accepted and processed).

When the initial phase is complete the required operations may be performed. For example a simple ADD instruction which has passed its condition check will bypass the shifter and multiplier, be evaluated in the ALU and be written into the reorder buffer; if it failed its condition check the ALU is not activated and an invalid ‘result’ is passed to the reorder buffer instead.

The execution unit may also pass an address to the data memory. This may be an unadulterated register value or a computed result, corresponding to the ARM’s post- and

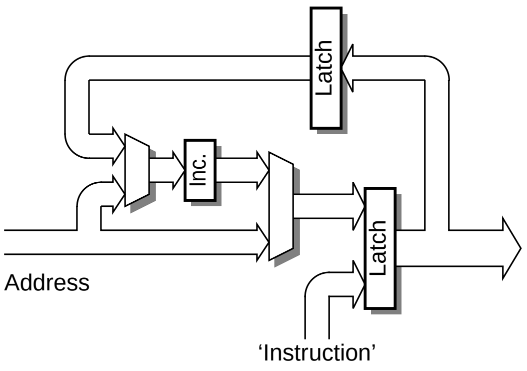

Figure 6: Data address interface

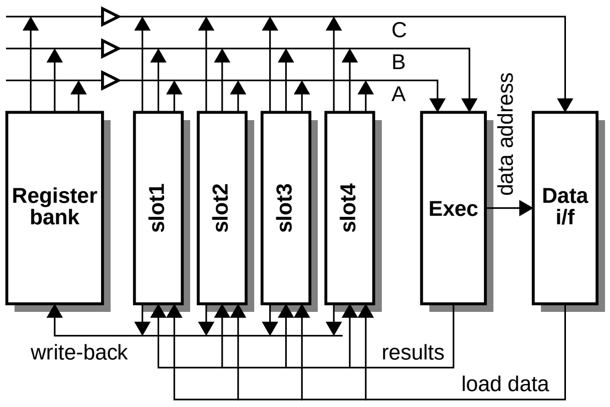

Figure 7: Reorder buffer organisation

pre-indexing addressing modes; in either case a computation can take place and the adjusted address written back into the reorder buffer concurrently with the data access.

Finally a branch operation may be required. Because the ARM can branch in numerous ways it is necessary to allow the ALU result to reach the branch output latch. However the majority of branches are caused by explicit branch instructions which – to reduce branch latency – are evaluated earlier in the decode stage and thus are available as soon as the condition check is complete. Note that the latch on the branch output (figures 1 & 4) is essential; the branch target may not be accepted immediately by the prefetch unit and instructions in the branch shadow must keep flowing (and be discarded) to ensure that the pipeline does not deadlock.

6: Data interface

The data interface (figure 6) is responsible for despatching memory transfers and, as noted in section 4.2, operates semi-autonomously. As it is located on a separate pipeline from the execution unit it receives its own 'instructions' irrespective of whether the instruction should be executed or not. Notice of whether an instruction is to be abandoned (for example due to a condition code failure) or continued is received from the execution unit along with the address. Reorder buffer space has already been allocated for this operation so it cannot simply be thrown away, but it is clearly inefficient to pass this packet through the whole memory system to no avail. Instead a decision can be made to route the packet directly back to the reorder buffer, bypassing the data memory stage. This is possible because the reorder buffer accepts inputs in any order (naturally!) and is inexpensive because this channel carries no data. Any potential dependencies on 'failed' memory operations can then be resolved much earlier than would otherwise be possible and – of course – there is a power saving in preventing unwanted memory operations.

7: Reorder buffer & register writeback

7.1: The reorder buffer

The reorder buffer was already referred to in section 4.1. A reorder buffer [13] implements a form of register renaming. Data packets carry their own destination position within the buffer and are thus sorted into order; the reorder buffer is emptied, in order, to the register bank, thus ensuring that any permanent state changes happen in instruction issue order. The advantage is that a result may be forwarded from this buffer even if it is uncertain that the instruction which created it should have been permitted to complete.

Reorder buffers have been used in synchronous systems for some time. However it is believed that this is the first asynchronous implementation (figure 7) [14, 15]. The difficulty arises because several asynchronous processes are interacting here:

- There are three processes which can write to the buffer (ALU, memory and memory skip - when a load or store fails its condition check). As these cannot write to the same location at the same time it is adequate to ensure that they merely have their own write ports. The 'write port' in the case of a memory skip is very primitive, as no data is carried;
- there is a single writeback process (section 7.2) which reads the values out, in order, to the register file;
- the forwarding processes need access to data in the reorder buffer at arbitrary times. The key point here is to observe that the register writeback process does not destroy or invalidate the contents of any reorder buffer location – only the arrival of a new result does that. It is therefore possible to forward data before, during or after their transfer to the register file.

The reorder buffer also allows speculation beyond memory aborts (e.g. caused by page faults). The processor is able to continue speculative execution (subject to register dependencies) without committing to a definitive state change. If an abort does occur the contents of the reorder buffer can be discarded and the register bank state preserved at a coherent point in the program execution.

A final complication of the reorder buffer is due to the conditional nature of all ARM instructions. When an instruction is issued it is allocated a reorder buffer slot. If it is not executed because it fails its condition test it must still fill this slot (i.e. mark it as invalid). This means that the reorder buffer must be searched for each register read and the latest *valid* data read, if any are present. This is implemented using a simple, asynchronous chained search and imposes a data-dependent delay; of course, this presents no problems to the rest of the system.

7.2: Register writeback

Although it incorporates a number of features connected with recovery from page faults, the register writeback stage is basically a serial process which either copies entries from the reorder buffer into the register bank or, if they are marked as invalid, discards them. This process is independent from the forwarding process and is thus not time-critical, although it is important that it is not so slow that the reorder buffer becomes full and thus stalls operation.

In practice the simplicity of this pipeline stage means that it is significantly faster than stages such as the instruction decoder or data memory. Thus it is capable of maintaining the reorder buffer in a near empty state unless it is waiting for data from external memory, and it is capable of catching up after a stall. In practice it cycles in 60%-70% of the core cycle time; this would be inconvenient in a synchronous system but is easily accommodated in AMULET3.

8: ARM compatibility

From the outset the objective of the AMULET projects has been to reimplement a successful synchronous architecture using an asynchronous design style. It is an interesting lesson to observe which features have been relatively straightforward to implement and which cause problems; such issues could be considered in future instruction set architectures. Some features which have caused particular problems are outlined below.

The ARM contains a few **multi-cycle instructions**, notably the multiple register memory transfer operations (LDM/STM). These are responsible for the introduction of extra cycles into the pipeline and present difficulties involving pipeline stalls in both synchronous and asynchronous implementations. In some ways the introduction of these extra sub-instructions is easier in the asynchronous implementation because the pipeline stall is compensated for automatically; it does impose an added complication in the control circuit however. This is exacerbated by the decision to synchronise with the execution stage when decoding such instructions in order to ensure that if such an instruction fails its condition test it is abandoned after the first cycle. Despite this the benefits of LDM/STM are clear.

More questionable are the **‘long’ multiplication instructions**, which require up to four 32-bit source registers and deliver a 64-bit result (i.e. two register destinations). Unless extra register ports are added purely for these (rarely used) operations they must be decoded specially and sequenced in two cycles. The need for such steering logic delays the control signals and slows the whole decoder down; here little benefit is gained.

Another area where compatibility issues impose on the asynchronous design is in **program counter (PC) tracking**. The ARM architecture defines that the PC appears in the register file (as R15). It may be written to much like any other register (with a few restrictions). However when it is read the value is assumed to have been stepped on by two instructions (8 bytes) and so the value is PC+8. (This is a backward compatibility issue, derived from the first implementations.) When executing Thumb code a similar argument applies; however in this case the instructions are only two bytes long, so the value read from R15 is PC+4. These values are also used in other situations, such as when calculating the targets of relative branches and the address of aborting instructions.

AMULET3 must maintain full code compatibility with existing code. To this end it appends the PC value to *all* fetched instructions. This allows the correct value to be obtained at the instruction decoder despite the prefetch unit being desynchronised with this stage. The expected value is obtained by incrementing the PC appropriately on demand (if it is not required the incrementer is not activated). Several PC incrementers thus litter the datapath to provide the correct value (after a value dependent, but typically small, delay). Here an architecture which specified the exact PC value would be both cleaner and simpler.

The asynchronous design framework does make the implementation of several instruction set architecture features more straightforward than it is in a rigid clocked framework. For example:

The elastic pipeline structure allows the inclusion of the **Thumb instruction decompression logic** with very little difficulty. A clocked pipeline requires that a time slot is included in the pipeline schedule for this logic, and this time is wasted when running 32-bit ARM code. Where this cost is too high, as on the ARM9TDMI, a separate full instruction decoder must be included to support Thumb code, thereby negating some of the benefits of the compression scheme. On AMULET3 the Thumb decode stage automatically collapses to occupy very little time when the processor is running ARM code.

The ARM instruction set includes high functionality **shift and add** instructions. These require a barrel shifter to be placed in series with the ALU in the critical 'execute' stage of the processor, compromising the cycle time. The dynamic frequency of instructions which use the full power of this feature is low, especially in compiled code. In an asynchronous implementation it is easy to switch the shifter in and out of the path as required, so the overhead is only incurred when the feature is being used. This is much harder to handle in a clocked design and has led to some complex compromises. In the ARM8 core, for example, the hardware can perform a generalized shift in series with a logic operation but only a limited subset of shifts in series with an arithmetic operation. When the instruction calls for a shift which is not in this subset in series with an arithmetic operation the execution unit must double cycle [1].

9: AMULET3 performance

At time of writing the processor layout is incomplete and therefore definitive performance figures are not available. However a considerable amount of the processor is laid out and it is therefore possible to give some figures based on simulations of these parts. These include the instruction decoder, register bank, reorder buffer and execution stage – including the shifter, multiplier and ALU.

The following numbers were obtained by simulating extracted layout using EPIC Timemill. The simulation rules are based on a VLSI Technology, Inc. 0.35µm three-layer metal process; simulation conditions were V_{dd}=3.3V, 20°C with 'typical' silicon. Note that these figures apply to the execution stages of the processor and assume that the instruction supply is not limited by the throughput of the prefetch unit or the instruction memory.

The throughput of the processing stages has been measured by injecting a stream of identical instructions into the instruction input stage and measuring the achieved input request frequency. The pipeline stages under test are fairly well balanced, with the decode and execution stages taking about the same time for operations such as MOV, although the execution stage limits performance for 'longer' operations, especially multiplications. These tests were performed using ARM instructions, so the Thumb decoder stage is fast and merely acts as a prefetch buffer. The register writeback is performed in parallel with the forwarding operation and, as mentioned in section 7.2, is significantly faster than the other pipeline stages.

The simulated frequency of a sequence of MOV operations is about 125 MHz; ADD operations are somewhat slower (around 115 MHz) but this should be improved before the device is fabricated. Including a shift operation increases the cycle time by ~35%. Multiplication (32x32) is performed by a dedicated, iterative unit and increases the cycle time of the execution stage to about 30ns; early termination is performed but the performance gain is relatively slight.

Instruction dependencies are resolved through the forwarding mechanism; the test sequences of instructions with and without dependencies have shown no difference in cycle time so it appears that the performance penalty of result forwarding is negligible.

9.1: Analysis

To give a fair comparison the throughputs described above should be compared with the equivalent part of AMULET2. The decode/execute stages of AMULET2 were faster than the overall throughput of the AMULET2e chip might suggest (~50MHz rather than ~40MHz), as the bottleneck occurred in the instruction memory access. AMULET2e was fabricated on a 0.5µm process and process scaling would suggest that, on a 0.35µm process, it could achieve an instruction cycle rate around 70MHz. Design improvements therefore account for almost a factor of two in throughput; indeed as AMULET2 suffered some penalty due to operand dependencies – not reflected in the "50MHz" figure above – a factor of two is probably a reasonable claim.

The increased parallelism in the processor – especially the extra register port and the removal of extra cycles supporting multi-register load/store operations – means that the number of cycles per instruction (CPI) is also reduced. High level simulations suggest that AMULET2 achieved around 1.35 CPI, whilst AMULET3 achieves around 1.2 CPI, approximately a 10% improvement. There should also be a noticeable improvement due to the dual memory interface, but this has not yet been quantified as it is dependent on the memory/bus cycle times. These improvements are summarised in table 1.

Feature	Improvement
Architecture (CPI)	+10%
Architecture (memory)	+ not yet quantified
Cycle time	+80%
Process	+40%
Total (approx.):	+180% (+)

Table 1: AMULET3 performance gains over AMULET2

10: Conclusions

AMULET3 is the culmination of almost a decade of intensive work at the University of Manchester on the design of asynchronous implementations of the ARM instruction set. In the course of this work we have identified and designed many asynchronous mechanisms for solving the organizational problems of general-purpose processors – problems for which there have existed classical clocked solutions for many years. Some of these ideas have worked well, some have been disappointing. Now, at last, we believe we have an adequate library of tools and techniques (some developed by us, but many coming from others' work) to claim that an asynchronous ARM can be competitive with the clocked original.

In part, the story of the development of the AMULET processors has been the story of finding a good solution to the problem of data dependencies, especially in a context where exact exceptions are required on memory faults. AMULET1 used register locking to ensure correct functionality [2]; AMULET2 added limited forwarding paths [3]. AMULET3 is based around a completely new approach, using a reorder buffer which has turned out to be surprisingly simple, small and efficient in its implementation.

AMULET3 promises very similar functionality and performance to an ARM9TDMI, with unusual low-power properties and unique electromagnetic compatibility characteristics which are advantageous in appropriate application areas.

11: Acknowledgments

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).

12: References

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

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

[3] Furber, S.B., Garside, J.D., Temple, S., Liu, J., Day, P. and Paver, N.C., "AMULET2e: An Asynchronous Embedded Controller", Proc. Async'97, Eindhoven, April 1997, pp. 290-299.

[4] Segars, S., "The ARM9 Family - High Performance Microprocessors for Embedded Applications", Proc. ICCD'98, Austin, October 1998, pp. 230-235.

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

[6] Segars, S., Clarke and Goudge, "Embedded Control Problems, Thumb, and the ARM7TDMI", IEEE Micro, **15** (5), October 1995, pp. 22-30.

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

[8] Sutherland, I.E., "Micropipelines", Communications of the ACM, **32** (6), June 1989, pp 720-738.

[9] Furber, S.B. and Day, P., "Four-Phase Micropipeline Latch Control Circuits", IEEE Trans. on VLSI, 4 (2), June 1996, pp. 247-253.

[10] York, R. "Branch Prediction Strategies for Low Power Microprocessor Design" M.Sc. Thesis, University of Manchester 1994

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

[12] Paver, N.C., Day, P., Furber, S.B., Garside, J.D. and Woods, J.V., "Register Locking in an Asynchronous Microprocessor", Proc. ICCD'92, Boston, October 1992, pp. 351-355.

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

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

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