AMULET1: A Micropipelined ARM

S.B. Furber, P. Day, J.D. Garside, N.C. Paver and J.V. Woods

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

Abstract

A fully asynchronous implementation of the ARM microprocessor has been developed in order to investigate the potential of asynchronous logic for low-power applications. The work demonstrates the feasibility of complex asynchronous design and shows that the cost and performance characteristics are similar to clocked designs.

AMULET1 is the first attempt at applying asynchronous techniques to a design of this complexity and as such there is much room for improvement. This paper introduces the design approach and organisation of the chip; it then covers the lessons learned from the first design and points towards future strategies for its enhancement and the likely benefits which will accrue from mature asynchronous technology.

1: Introduction

The growth in the market for portable computing equipment and the problems of thermal dissipation of high-end microprocessors are both factors which indicate that power consumption is an increasingly important engineering concern in the design of computers. One source of power inefficiency in current design practice is the use of free-running high frequency clocks which cause all parts of the system to burn power whether or not they are doing useful work. Asynchronous design avoids this inefficiency since functional units use power only when doing useful work. Therefore asynchronous approaches have been proposed as offering a route to more power-efficient computing.

The AMULET project was established at Manchester University in late 1990 to investigate the potential power savings of asynchronous design. The goal was the development of a fully asynchronous implementation of the ARM microprocessor [1], primarily to demonstrate the feasibility of applying asynchronous design techniques on a commercially interesting scale. The design was completed early in 1993 after 5 man-years of design effort and was submitted for fabrication. The work shows that designs on this scale are practical and produce results which are similar in design effort, chip area and performance to conventional clocked designs.

Though the first design is fully functional, it must be viewed as an initial attempt at an asynchronous implementation of a commercial microprocessor architecture rather than the last word on the matter. Experience gained during the development and subsequent evaluation of the design points to considerable scope for future enhancement.

2: Micropipelines

The asynchronous design style used in AMULET1 is based on Sutherland's Micropipelines [2] which employ a 2-phase bundled data interface for sending data between functional units. The communication protocol is illustrated by the timing diagram in figure 1. A valid data value is placed on a conventional bus by the sender which then indicates the availability of the data by causing a transition on a *Request* wire. The receiver senses this transition, accepts the data and then causes a transition on the *Acknowledge* wire, completing the transfer. The sender may then issue another data value in a similar manner. Note that only the order of these events is significant; the delays between

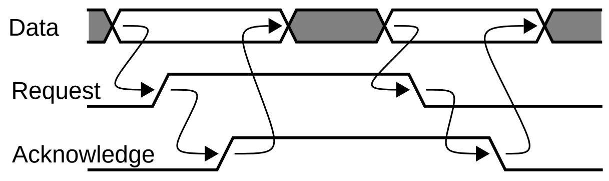

Figure 1: Micropipeline handshake protocol

them are arbitrary (though long delays will, of course, reduce performance). Also note that the Request and Acknowledge wires use 2-phase signalling; rising and falling edges are both significant and have the same meaning.

3: AMULET1

The AMULET1 organisation has been described elsewhere [3] so only a summary will be presented here. The processor to memory interface follows the Micropipeline convention, with one (output) bundle to send address, control and write data to the memory and a second (input) bundle to return read data from the memory. The memory system may have an arbitrary pipeline depth and delay, but must return read values in the requested order.

Internally the processor is constructed from several function units (figure 2) which operate independently and concurrently, exchanging information through bundled data interfaces. The role of each of these units is described below.

3.1: Address interface

The address interface is responsible for issuing read and write requests to memory. It issues instruction prefetch requests autonomously and accepts data transfer and branch target addresses from the execution unit as required. Branch target addresses are immediately issued to memory and also change the prefetching stream to continue from the target location; data transfer addresses temporarily interrupt the prefetching stream which resumes once the data address has been issued.

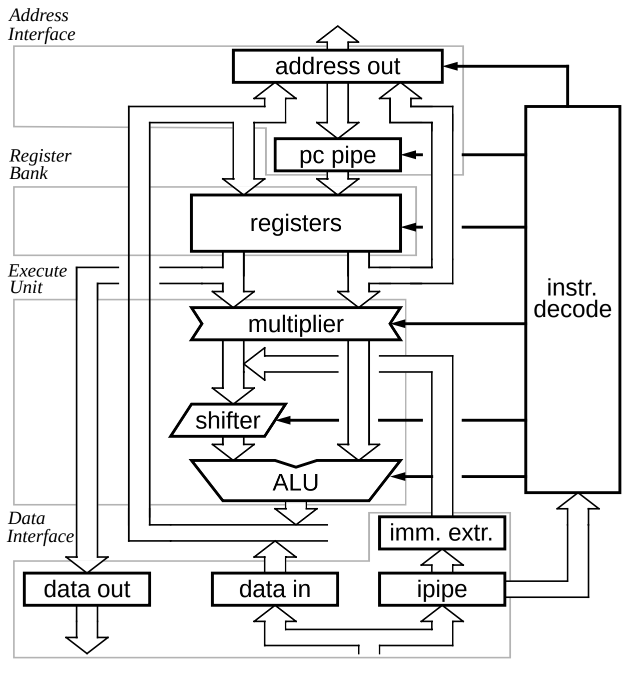

Figure 2: AMULET1 organisation

The ARM architecture makes the program counter readily accessible to the programmer as register 15 in the register bank. PC values are therefore copied from the address interface to the register bank through a PC pipeline which buffers the values until the associated instruction arrives from memory.

3.2: Register bank

All the user accessible state is held in the register bank, which employs a novel locking mechanism [4] to allow multiple pending writes from the execution pipeline and from external memory. The locking mechanism ensures the correct behaviour of instruction streams with data dependencies between successive instructions and enables register read and write processes to proceed asynchronously without arbitration and without risk of metastability in the control and data circuits.

3.3: Execution unit

Arithmetic processing is carried out in the execution pipeline. This incorporates a ‘3-bits at a time’ carry-save multiplier, a barrel shifter and rotator and an ALU. The ALU has a data dependent propagation delay which detects the longest carry chain in an addition [5]. This allows a relatively simple ALU to give better average performance on a typical mix of operand values than the more complex ALU in the clocked ARM6, since there is no need to coerce the worst case addition into a fixed clock period.

3.4: Data interface

The data interface is responsible for receiving data from memory and for steering it into the instruction pipeline, register bank or address interface. Instructions are stored in a pipeline awaiting execution, and immediate values are extracted at the top of the pipeline for use as operands as necessary. Loaded data values are aligned and byte-extracted as required by the instruction.

Data to be written to memory is also passed through this unit where it is synchronised with address and control information in the address interface before all these signals are passed to memory as a single bundle.

3.5: Overall pipeline structure

The overall pipeline structure of the processor is shown in figure 3. The shaded boxes are the pipeline registers in the main decode and execute paths, and indicate the depth of pipelining employed in the design.

Throughout the design the control and datapath pipeline stages are matched so that, at least in principle, all the logic units can be kept busy. Several pipeline FIFO structures are

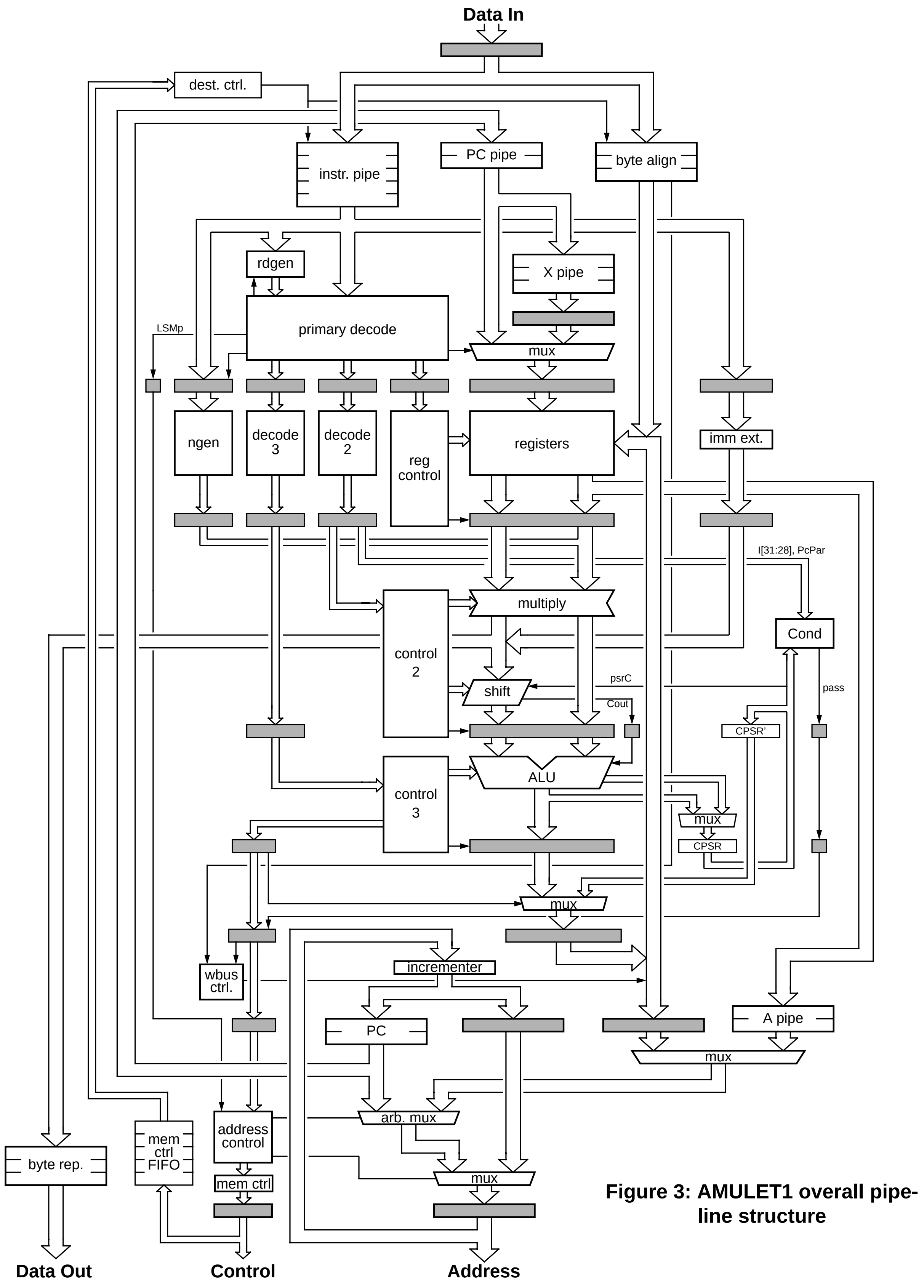

Figure 3: AMULET1 overall pipeline structure

employed as elastic buffer memories to decouple the peaks in the supply and demand characteristics of the logic units at either end; in many cases the depths of these FIFOs may be chosen arbitrarily in an attempt to optimise performance (or power consumption), though in some cases there are functionality issues to consider as well. An example is that the instruction and PC pipeline depths may be varied arbitrarily except that the Instruction pipeline must be at least 3 stages longer than the PC pipeline otherwise a deadlock can occur.

4: Asynchronous operation

The asynchronous implementation of the ARM instruction set is described below with reference to figures which show the movement of data around the AMULET1 bus structure.

These figures are very similar to the corresponding figures for the synchronous ARM [1]. Note, however, that whereas an instruction has sole occupancy of the datapath in the synchronous ARM, here there are several pipeline stages in the execution phase and several instructions may occupy different stages.

4.1: Register to register instructions

The basic operation of the AMULET1 datapath during a register to register instruction is illustrated in figure 4. The

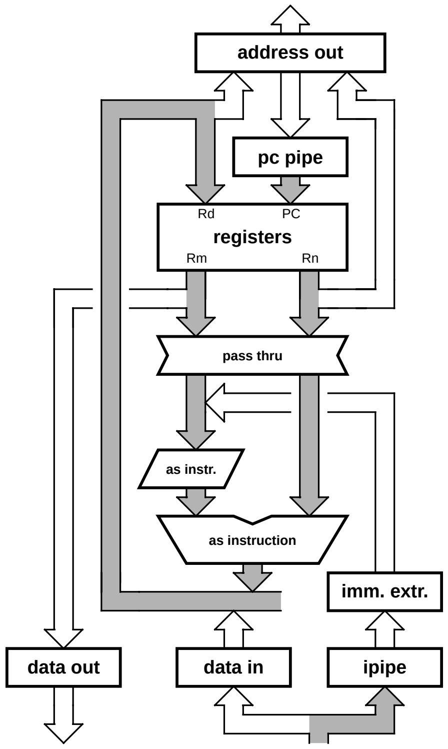

Figure 4: Datapath activity during a register to register instruction

two register operands are read from the register bank and passed through the multiplier. One operand is shifted as required by the instruction, then the two operands are combined in the ALU and the result returned to the register bank.

If one (or both) of the source operand addresses is register 15 the register file should return the PC value for the current instruction with an offset of 8 bytes. The offset is an artifact of the synchronous pipeline implementation on earlier ARM chips, but it is visible at the instruction set level and so must be emulated here. The PC resides in the address interface where it is autonomously incremented and issued for instruction prefetching, so PC values are copied into a PC pipeline as they are issued for use in the register file as R15.

Each instruction must present the correct PC value as R15, so as an instruction enters decode it is paired with the next value from the PC pipeline. The high level view of this operation is shown in figure 5.

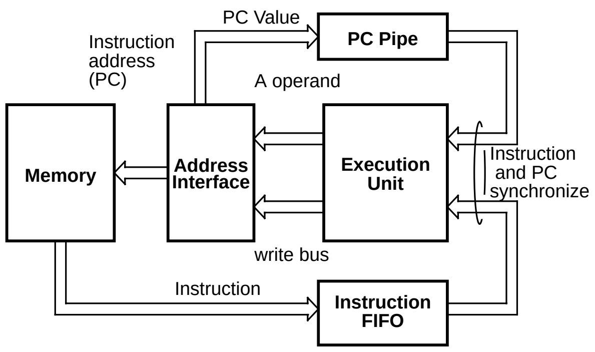

Figure 5: PC and instruction synchronization

4.2: Branch instructions

Where the instruction modifies the program counter the new value must be sent to the address interface. Figure 6 shows the movement of data during a branch instruction. The branch target address is computed in the ALU from the current PC value (which is passed from the PC pipeline via the register bank to the ALU) and an immediate offset contained in the instruction.

The branch target address is passed to the address interface where it replaces the old PC value. The address interface then begins to fetch the new instruction stream by issuing sequential addresses starting from the branch target.

As the branch target address arrives asynchronously to the operations within the incrementing loop in the address interface, it breaks into the loop at a non-deterministic point. A consequence of this is that AMULET1 displays a non-deterministic depth of prefetching beyond a branch instruction.

A branch on ARM6 takes three clock cycles to complete; this is because pipeline flushing is handled directly by the instruction and the extra two cycles match precisely the depth of prefetching. On AMULET1 the branch is performed by a single pipelined operation with a separate mechanism to control flushing. In practice there is no performance advantage in reducing the branch to a single cycle since the flushing overhead remains the same, but the non-deterministic amount of flushing required makes the simple mechanism employed on ARM6 impractical here.

4.3: Load and store instructions

One place where AMULET1 uses fewer datapath operations than ARM6 to good effect is in single register load and store instructions. ARM6 has a memory interface which allows the transfer of one word per clock cycle, and since load and store instructions require two words to be transferred across this interface (one word of data and one instruction word) these instructions require a minimum of two clock cycles to complete their execution phase. In fact load instructions take three cycles to complete since data arrives from memory too late to be written directly back into the target register, and a third cycle is required to complete the transfer.

Since two (or three) cycles are required for these instructions, the ARM6 datapath is available for useful work during these cycles. The ARM instruction set exploits these cycles to allow more powerful addressing modes to be supported than are normally found on RISC processors. (Other RISCs generally use a Harvard architecture to avoid the need for extra clock cycles for loads and stores.)

AMULET1 can perform (most) load and store operations in a single datapath operation. Autonomous instruction prefetching results in a buffered supply of instructions, so interrupting the prefetch operation for a data transfer should not result in the processor becoming starved of instructions. The register locking mechanism manages any delay in the memory response to the transfer request, so provided the compiler can schedule independent instructions after the load the processor can continue doing useful work.

Figure 7 illustrates the datapath activity for a pre-indexed load instruction. Here the transfer address is computed by adding or subtracting a register offset to or from the base register. The result is used both as the memory address and to overwrite the original base value, giving the auto-indexing behaviour required by the instruction semantics. When the requested data returns from memory it passes through the data-in block onto the ALU result bus and to the destination register. This can happen after an arbitrary delay and is independent of the execution pipeline

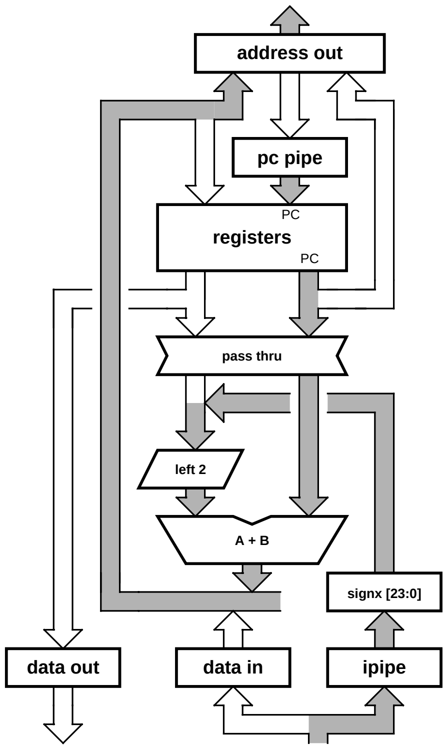

Figure 6: Datapath activity during a branch

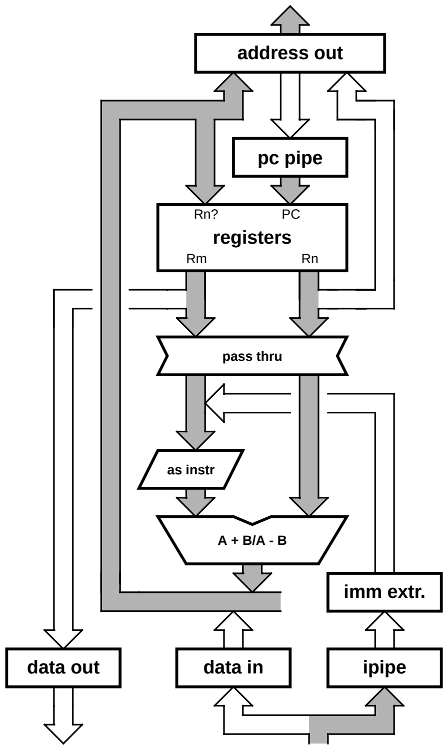

Figure 7: Datapath activity during a pre-indexed load instruction

control logic. Access to the ALU result bus is controlled non-deterministically by an arbiter; steering the result to the correct register is controlled by the register bank lock FIFO. (Where the destination is R15, which in ARM is an alias for the PC, the memory control pipeline steers the data directly to the address interface rather than to the register bank.)

Post-indexed addressing modes require the base register to be used unmodified as the memory address, then an offset is added to the base to prepare it for the next transfer. Again this is achieved in a single datapath operation as shown in figure 8. Note here how the base register is copied directly from the register bank to the address interface via a dedicated bus. This bus is absent on ARM6 (where it could not be exploited because of the memory interface restrictions) but is needed here to make single cycle operation of post-indexed instructions possible.

The load and store addressing modes allow immediate offsets to be used instead of the register offsets illustrated in figures 7 and 8; these are implemented in a similar way to the branch offset flow shown in figure 6. The buses are configured so that when an immediate offset is used the second register read port can be used to access data for storing, as shown in figure 9. The majority of store instructions in typical programs use immediate offsets, but when a register offset is required it is no longer possible to complete the instruction in a single datapath operation, since three register operands are required. In this case the address computation is performed in one operation exactly as for a load instruction, then the data is accessed in a subsequent operation. Since the store data-out route is less complex than the address route, the data will be available to rendezvous with the address without delaying the issue of the memory request. The only cost of the second operation is the delay to the following instruction. Although stores with register offsets are rare in typical code, they must be supported for ARM6 compatibility.

4.4: Load and store multiple register instructions

In addition to the single register load and store instructions, the ARM architecture also supports instructions which can load or store any subset (or all) of the registers in current use. These instructions are used for procedure entry and return, context switching, and high-bandwidth memory block copies. Although they occur less frequently in typical code than the single register transfer instructions, they are responsible for a similar total number of registers loaded and stored, so similar attention must be paid to their efficient implementation.

Load and store multiple register instructions support auto-indexing addressing modes which are similar to the single register transfer instructions, and use similar tech-

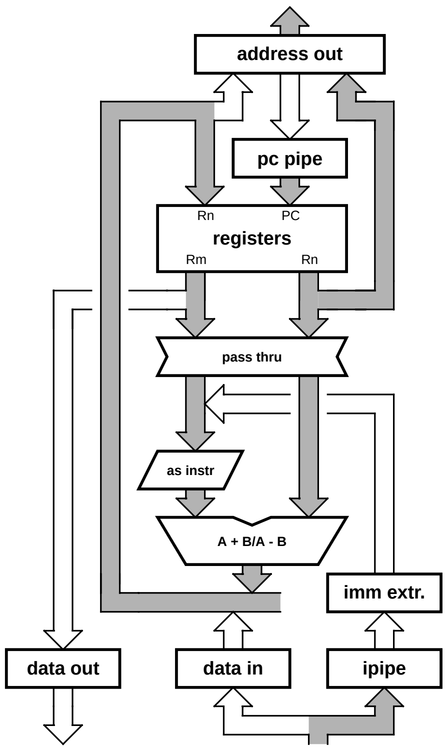

Figure 8: Datapath activity during a post-indexed load instruction

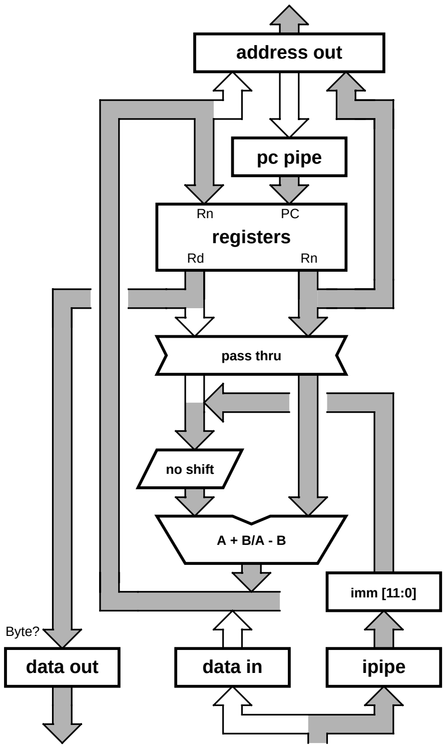

Figure 9: Datapath activity during a post-indexed store instruction with immediate offset

niques to compute the first transfer address. Dedicated hardware is used to identify the registers to be transferred, and the PC incrementing loop in the address interface generates the sequential memory addresses used for the transfers.

The major difficulty in the implementation of these instructions is the need to provide an exact exception and restart mechanism when memory faults arise (for example when a virtual memory system has paged the target memory area out onto disk). The problem is particularly severe in the case of load multiple, since this instruction can modify most of the processor state before the fault is detected.

The implementation of load multiple on AMULET1 uses four phases:

- The address of the first transfer (which is always the lowest address) is computed and the data transfer is initiated. Subsequent addresses are computed autonomously in the address interface.
- The base write-back value is computed and written back to the base register, but a copy is kept in the execution pipeline in case a fault arises.
- The target registers are marked to receive their data values. Note that the base register may appear in the transfer list and be corrupted.
- The base register is read again to ensure that any transfer to it has completed. If a fault has arisen, the write-back value is written back again to prepare for a restart.

When R15 is in the transfer list it is always transferred last, so any fault will arise before the PC is corrupted. Therefore the PC and the base register are always preserved when a fault arises, which is sufficient to ensure that the instruction is restartable.

5: AMULET1 characteristics

The principal characteristics of AMULET1 are summarised in table 1, along with the corresponding characteristics of ARM6, a functionally similar synchronous design.

Table 1: Characteristics of AMULET1 and ARM6

	AMULET1	ARM6
Process	1μm DLM CMOS	1μm DLM CMOS
Cell area	5.5mm x 4.1mm	4.1mm x 2.7mm
Transistors	58,374	33,494
Performance	9K dhrystones	14K dhrystones @ 10MHz
Power	83mW	75mW @ 10MHz

These characteristics show that the asynchronous design uses twice the number of transistors and twice the silicon area of the clocked chip on an equivalent process, and delivers rather lower performance at a similar power consumption.

The performance measure in table 1 is based upon dhrystone code compiled for the ARM6; subsequent work has shown that this can be improved significantly by the inclusion of optimisations which take account of the inter-instruction dependencies of the AMULET1 design. With such optimisations the performances of the two chips are then very similar.

Note also that for both chips these performance figures are based on worst case slow-slow simulations. Production ARM6 chips have been qualified at significantly higher clock rates.

The design effort which was required to define the organisation of AMULET1 and then to convert this organisation into a silicon layout was around 5 man years. Again, this is comparable with the design resource required to produce an equivalent clocked design

6: Retrospective analysis

The primary motivation for this investigation into complex asynchronous circuits was the need for design styles which deliver lower power consumption than is achieved through current practice.

If the measure of power efficiency is the delivered MIPS/Watt at peak performance, then AMULET1 has little to offer; indeed, on this measure it performs less well than an ARM6 (though note that ARM6 is a world-leading design on this measure, so it sets a high target to beat). However, before writing-off asynchronous logic for low-power applications, the following points should be noted:

- AMULET1 is a first attempt at an asynchronous design of this complexity. As such the primary goal at this stage was to prove feasibility, not low power. There is considerable scope for improving the power efficiency.
- Power consumption at peak performance may not be the most important measure. Most processors are idle much of the time and asynchronous logic has the potential to drop instantly to zero power under idle conditions. Clocked systems can only save power on a coarse granularity by stepping the clock rate down.

The organisation of AMULET1 employs a relatively deep pipeline, which accounts for much of the increase in transistor count and die size relative to ARM6. The benefit of the deeper pipeline should be an increase in throughput, but in practice this has not been borne out. This can be attributed to a number of factors including:

- The primary instruction decoder cannot issue instructions at the rate required to keep the pipeline busy.
- Data dependencies between consecutive instructions in typical compiled code cause frequent interlocks which stall the pipeline.

The second of these is partly a function of the compiler. The ARM6 compiler makes no attempt to separate dependent instructions because the ARM6 shows no benefit from so doing. Preliminary improvements in compiler optimization strategies have yielded significant performance improvements in AMULET1.

In retrospect the pipeline structure on AMULET1 is too deep. Micropipelines make building FIFO buffers easy and it is therefore tempting to use them too liberally; the AMULET1 design falls into that trap. Many of the pipeline stages are contributing nothing to the chip's performance, or in some cases actually diminish it, whereas they all dissipate power.

The asynchronous control structures used on AMULET1 are not a major source of inefficiency. It has been estimated that they represent a small area overhead (10% to 20%) compared with an equivalent clocked organisation and can support similar operating speeds. The asynchronous design style does, however, preclude the use of some architectural enhancements which contribute to the high performance of clocked processors with deep pipelines - the prime example of this being register forwarding (which will be discussed further below).

In some areas the asynchronous approach shows a particularly strong win. The ALU, for example, can exploit a simple structure with operand dependent delay which gives better throughput on a typical mix of operand values than a much more complex ALU for a synchronous system, optimised for rare worst case operand values [5]. The flexibility of the asynchronous approach allows a design to be optimised for typical conditions whilst supporting rare but necessary functions with simple logic; there is no need here to commit large amounts of silicon to ensuring that infrequently used operations can complete within a fixed clock cycle.

7: Future enhancements

The AMULET1 design was completed early in 1993, since which considerable effort has been put into analysing the behaviour of the design to identify sources of performance loss with a view to the design of a more efficient successor. This work has produced a number of enhancements of the design, as detailed below.

7.1: Faster decode logic

The major bottle-neck in the design is the primary instruction decoder. The logic task here is complex as the ARM instruction set exposes some features of the synchronous pipeline, and these must be unwound here and emulated on a very different pipeline.

The improvements under investigation include increasing the level of concurrency in the decode logic, and switching from two-phase to four-phase (return to zero) asynchronous control within the decoder.

7.2: Result forwarding

Conventional register forwarding depends on performing a comparison between the destination register address of one instruction and the source register addresses of a successor instruction. These addresses are associated with instructions at different pipeline stages, but in a synchronous design these stages are controlled by the same clock so the comparison can be performed directly.

In an asynchronous pipeline comparisons between addresses in different stages cannot be compared without explicit synchronisation between the two stages, which is a potential source of inefficiency (since synchronisation is always achieved by slowing the faster stage down). Therefore AMULET1 does not employ forwarding logic but passes all the results through the register bank where a locking mechanism is used to avoid hazards.

Because typical code contains frequent data dependencies between successive instructions, the locking mechanism will often hold the processor up (particularly when the decode bottleneck is removed). Therefore an equivalent to register forwarding is required. This has been solved within the asynchronous environment by retaining the result register address of one instruction for use during the decode of its successor. If the successor requires this register as a source operand the decoder will bypass the register read phase and collect the operand from a 'last result' register which will be associated with the ALU and act like a local register cache controlled by the instruction decoder.

A similar mechanism can be used to forward data items loaded from memory directly to the execution unit when required.

7.3: Register lock latency reduction

When the result forwarding mechanism is not applicable the register bank locking mechanism will manage data dependencies as on AMULET1 [4]. A simple enhancement to the existing mechanism has been developed which reduces the write to read delay significantly; this is illustrated in figure 10.

On AMULET1 the write word line is driven directly from the lock FIFO output. This must remain stable until the write has completed and the write word line has been disabled; then the lock FIFO entry can be removed and a read operation which is held on the lock can proceed.

In figure 10 the write enable latch has been added to the circuit. This latch is now responsible for holding the write word lines stable until the write has completed, so the lock FIFO entry can be cleared as soon as the register contents are updated and the read can proceed concurrently with the disabling of the write word line. This significantly reduces the write to read delay when a read stalls on a lock.

It also illustrates the modularity of the asynchronous design style; this sort of change can be incorporated internally in a block such as the register bank and, provided the interfaces to adjacent blocks are preserved, the enhanced register bank can be inserted into the design without any changes elsewhere in the system. It is rarely so straightforward to introduce a single-point enhancement in a synchronous system.

7.4: Pipeline simplification

The pipeline structure employed on AMULET1 is now believed to be deeper than is optimal. A deep pipeline can support high throughput (when instructions can be issued fast enough to keep it full) but has poor latency properties which are particularly noticeable when a change in control flow occurs which, in typical code, is on average once every five instructions.

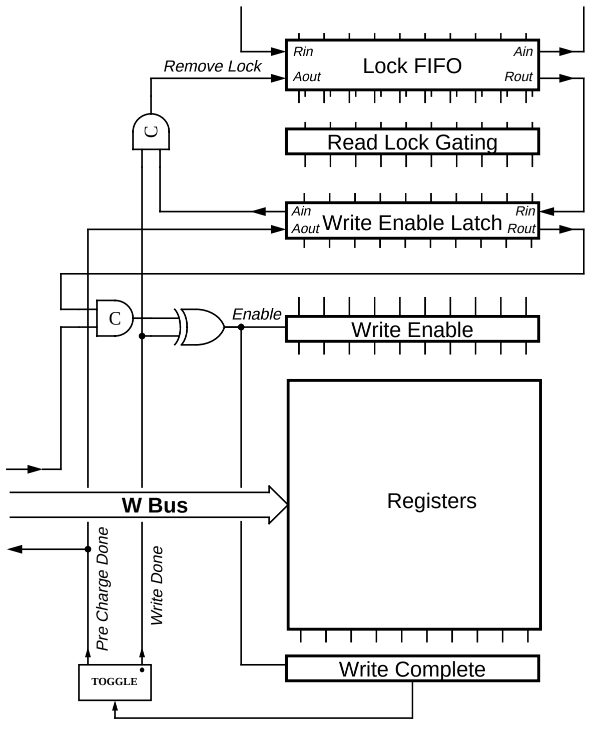

Figure 10: Improved latency register read logic

The main pipeline will work better if it is made wider and shallower. We intend to remove the separate pipeline stage for the multiplier and shifter, and bypass these completely when they are not required (which already happens with the multiplier, but not the shifter). Instructions which require the shifter will take longer in the execute stage, but as these are relatively infrequent the reduced pipeline latency for other instructions will show an overall advantage. There will also be a power saving whenever the shifter is bypassed, which will frequently be possible with the addition of dedicated logic to perform alignments of immediate (literal) operands.

Subsidiary control pipelines are in many places longer than is useful, and these will be reduced. The maximum depth of prefetching will also be reduced by reducing the depth of the instruction and PC pipelines, saving power and improving performance.

7.5: Improved latch circuits

The micropipeline style of asynchronous design requires the use of self-timed latches to control each of the pipeline stages. The latches used on AMULET1 are conventional transmission gate latches with weak feedback inverters to give static operation. These latches require complementary control wires. The latch control wires are a significant source of capacitive load (and hence power dissipation) during pipeline operation, so the need for two control wires for each latch is undesirable for low-power designs. The control problem is also made more difficult since both wires must be monitored to ensure that latch operations have completed.

Investigations into alternative latch technologies have been carried out to examine the applicability of single-phase latches [6] which require only one control wire. These have been used to considerable effect on high-performance designs, most notably the DEC Alpha [7]. For low-power applications some modification is required to give the latches fully static behaviour, but this is possible and appears to give significant benefits both in speed of operation and energy per cycle compared with the AMULET1 latch technology.

Four-phase control has been mentioned above in the context of instruction decode logic; it may also offer benefits for the control of simple pipelines.

7.6: Architectural extensions

The ultimate goal of the work beyond AMULET1 is to produce an asynchronous integrated CPU which incorporates an MMU and cache memory. This chip will operate asynchronously for all internal operations, but will interface to a conventional external memory system through a clocked interface.

In order to support the MMU and cache the processor will require a coprocessor interface. This is the major feature of the ARM6 which is missing on AMULET1. The ARM6 coprocessor interface is synchronous and inappropriate for asynchronous use, so a new interface will be developed. The MMU and cache will be based on existing ARM designs for architectural compatibility, but will be enhanced for asynchronous applications by pipelining where appropriate.

Taken branches have a high latency cost on AMULET1, so various history and trace techniques are currently under investigation to establish their applicability to asynchronous pipelined processors. It is not yet clear whether such techniques will be cost-effective on a successor design.

8: Conclusions

The AMULET1 design demonstrates the feasibility of developing complex asynchronous circuits. The design encompasses many of the more complex features of modern high-performance RISC processors, such as exact exception handling, instruction set compatibility and pipelined operation, and shows that asynchronous techniques are applicable to commercial scale digital design tasks.

AMULET1 is within a factor 2 of the ARM6 with respect to important parameters such as performance, power consumption and die size. As such it demonstrates that asynchronous designs are comparable with clocked designs, but AMULET1 does not, at this stage, demonstrate any significant advantage over its clocked predecessors.

It should be borne in mind, however, that ARM6 is a highly evolved fourth generation synchronous processor, whereas AMULET1 is the first of its line. Furthermore, many features of the ARM instruction set have been designed with reference to its synchronous implementation; these have been sources of difficulty in the asynchronous implementation and bias the comparison unfavourably against AMULET1.

Work is still going on to enhance the design and substantial improvements are possible in many areas. It is hoped that in the near future we will be able to demonstrate real advantages for asynchronous methodologies based on the techniques outlined in this paper.

9: Acknowledgements

The AMULET1 design work described in this paper was carried out as part of ESPRIT project 5386, OMI-MAP (the Open Microprocessor systems Initiative - Microprocessor Architecture Project). Subsequent work has been supported as part of ESPRIT project 6909, OMI/DE-ARM (the Open Microprocessor systems Initiative - Deeply Embedded ARM Applications project). The authors are grateful for this support from the CEC.

The authors are also grateful for material support in various forms from Advanced RISC Machines Limited, Acorn Computers Limited, Compass Design Automation Limited, VLSI Technology Limited and GEC Plessey Semiconductors Limited. The encouragement and support of the OMI-MAP and OMI/DE-ARM consortia are also acknowledged.

10: References

[1] Furber, S. B., “VLSI RISC Architecture and Organization”, Marcel Dekker, New York, 1989.

[2] Sutherland, I.E., “Micropipelines”, Communications of the ACM. Vol. 32, No. 6, January 1989, pp. 720-738.

[3] Furber, S.B., Day, P., Garside, J.D., Paver, N.C. and Woods, J.V., “A Micropipelined ARM”, Proceedings of the IFIP TC 10/WG 10.5 International Conference on Very Large Scale Integration (VLSI’93), Grenoble, France, September 1993. Ed. Yanagawa, T. and Ivey, P. A. Pub. North Holland.

[4] 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.

[5] Garside, J.D. “A CMOS VLSI Implementation of an Asynchronous ALU”. IFIP Working Conference on Asynchronous Design Methodologies, April 1993. Ed. Furber, S. B. and Edwards, M. D. Pub. North Holland.

[6] Yuan, J., and Svensson, C., “High-Speed CMOS Circuit Techniques”, IEEE Journal of Solid-State Circuits, Vol. 24, No. 1, February 1989, pp. 62-70.

[7] Dobberpuhl, D. W. et al., “A 200-MHz 64-b Dual-Issue CMOS Microprocessor”, IEEE Journal of Solid-State Circuits, Vol. 27, No. 11, November 1992, pp. 1555-1565.