

AMULET1: An Asynchronous ARM Microprocessor

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

Abstract—An asynchronous implementation of the ARM microprocessor has been developed using an approach based on Sutherland's Micropipelines [1]. The design allows considerable internal asynchronous concurrency. This paper presents the rationale for the work, the organization of the chip, and the characteristics of the prototype silicon. The design displays unusual properties such as nondeterministic (but bounded) prefetch depth beyond a branch instruction, a data dependent throughput, and employs a novel register locking mechanism. This work demonstrates the feasibility of building complex asynchronous systems and gives an indication of the costs and benefits of the Micropipeline approach.

Index Terms—Processor architectures, single data stream architectures; control structures and microprogramming, control design styles; integrated circuits, types, and design styles



1 INTRODUCTION

ASYNCHRONOUS logic design techniques have been largely ignored by mainstream designers for many years. There is now a resurgence of interest, however, as synchronous design techniques approach some fundamental limits.

Today's VLSI chips incorporate very large numbers of transistors and it is becoming increasingly difficult to maintain global clock synchrony over a large chip area. This problem has long existed in board-level design, but is only now becoming a problem at chip level. As clock rates rise, and the chip area over which the clock must be distributed expands, clock skew becomes ever more difficult to control. Remarkable engineering techniques have been employed to contain the problem [2], but only at the cost of considerable silicon area and high peak supply currents.

Designers of synchronous systems must always consider worst-case circuit conditions (with respect to both operand values and environmental conditions) to ensure that operations will always complete within the clock period. Different parts of the system will bias the designer towards different clock rates and the ultimate choice of clock frequency will be a compromise. With the industry trend toward component reuse, the best clock frequency choice is increasingly subject to conflicting constraints.

Power consumption in large VLSI chips is also becoming a significant issue for two reasons. First, portable equipment such as lap-top computers and mobile telephones demand low power consumption to optimize battery life. Second, high performance desktop processors now dissipate enough power to make chip cooling a problem in an office environment.

Asynchronous operation offers the possibility of removing the need for a global clock. Functional units generate timing pulses locally, only when required, and dissipate power only when there is useful work to be done. There is thus the potential to make the design process easier by removing a global delay management problem, to reduce power consumption by activating functional units only when necessary, and to maximize performance by cycling components at the appropriate rate for the current data values and environmental conditions. Standardized interfaces also make component reuse straightforward.

With such potential advantages in mind, the AMULET1 project was established to investigate how these factors interrelate in practice. The goal was to develop a fully asynchronous circuit of sufficient complexity to be commercially relevant. The ARM microprocessor [3] was chosen as a suitable architectural target and this has been re-implemented successfully in a fully asynchronous style. This work establishes the feasibility of designing complex circuits without a clock and demonstrates that one of the major obstacles to the commercial exploitation of asynchronous design techniques has been overcome.

Previous work had demonstrated the feasibility of building a relatively simple microprocessor using asynchronous techniques [4] and a microprocessor architecture employing *matched-delay* circuits based on the *data-driven* computation model has been described [5]. AMULET1 takes this a stage further by re-engineering an architecture proven in commercial use and containing features such as interrupts and exact exceptions; these present a significant challenge to processor designers.

2 ASYNCHRONOUS BACKGROUND

Several approaches to asynchronous design are currently being researched [6], each with its own particular advantages. Purist asynchronous digital circuit design uses a dual-rail encoding system where a Boolean value is transmitted by

• J.V. Woods, S.B. Furber, J.D. Garside, and S. Temple are with the Department of Computer Science, The University, Oxford Road, Manchester, M13 9PL, U.K. E-mail: jvwoods@cs.man.ac.uk

• P. Day and N.C. Paver are with Cogency Technology UK, Bruntwood Hall, Bruntwood Park, Cheadle, SK8 1HX, U.K.

Manuscript received Mar. 1, 1995; revised Jan. 31, 1996.

For information on obtaining reprints of this article, please send e-mail to: transcom@computer.org, and reference IEEECS Log Number C96329.

asserting a signal on one of a pair of wires, thus conveying both value and timing information. Such a design is usually designated “delay-insensitive” as correct operation is independent of both circuit and interconnection delays. It is, however, costly in terms of both interconnection and logic element complexity. Despite this the dual-rail style is often preferred by investigators into circuit compilation [7], [8] because of its immunity to delays introduced by automatic circuit layout tools.

In contrast, an engineering approach may accept the need for local delay management and associate an asynchronous control signal with a bundle of data wires to indicate the validity of the data. The management of “clock skew” is then restricted to the timing of this data only.

In both approaches, a choice must be made between level or transition encoding of signals. A level-sensitive design would represent values in the “conventional” way, typically a logic 1 as a high voltage level and a logic 0 as a low voltage. Transition signaling protocols use a change in signal level to convey information; either a data value when used in conjunction with dual-rail encoding, or as a timing event on a single line. Fig. 1 shows the *Request* and *Acknowledge* interchange required for the asynchronous transfer of information using both the level-sensitive, or “four-phase,” and the transition-sensitive, or “two-phase,” signaling protocols. It can be seen that the two-phase protocol does not have the requirement for the signal to return to the “nonactive” state; thus there is the possibility of exploiting the fewer transitions required to enhance performance and to reduce power dissipation. There is a cost here, however, as transition-signaling logic modules are generally more complex than their level-sensitive equivalents.

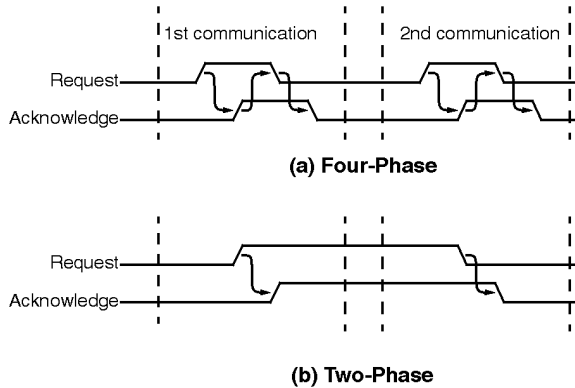


Fig. 1. Four-phase and two-phase signaling protocols.

2.1 Micropipelines

An asynchronous approach based on a two-phase bundled data convention was described by Sutherland in his Turing Award “Micropipelines” lecture [1]. Fig. 2 shows the bundled data interface where the *Sender* unit places valid data on the data bundle prior to producing a transition, or “event,” on the *Request* wire; the *Receiver* takes the data and then generates an event on the *Acknowledge* wire to indicate to the *Sender* that the information has been taken.

The signal protocols for this micropipeline transfer are illustrated in Fig. 3 where the equivalence of positive and

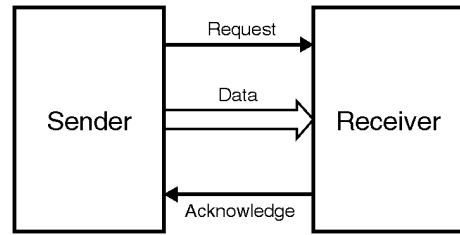


Fig. 2. Two-phase bundled data interface.

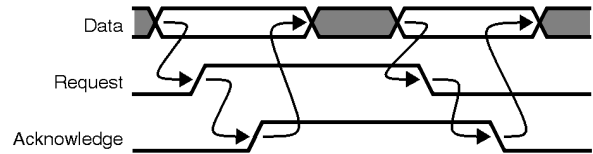


Fig. 3. Micropipeline handshake protocol.

negative going transitions as events can be seen. It is the responsibility of the *Sender* to ensure that all bits of the data bundle are valid prior to sending the *Request* event; this is the local delay management. Similarly, the *Receiver* must have accepted the data bundle prior to sending its *Acknowledge* which will permit the *Sender* to remove the old data and place a new value on the data lines.

Local delay management must also ensure that interconnection delays are such that data arrives at the *Receiver* sufficiently in advance of the *Request* to conform with set-up requirements.

2.2 The Asynchronous Logic Family

The basic OR and AND functions for transition signaling differ from those for level sensitive logic as they must respond to transitions rather than logic levels. The family of event driven modules used is shown in Fig. 4.

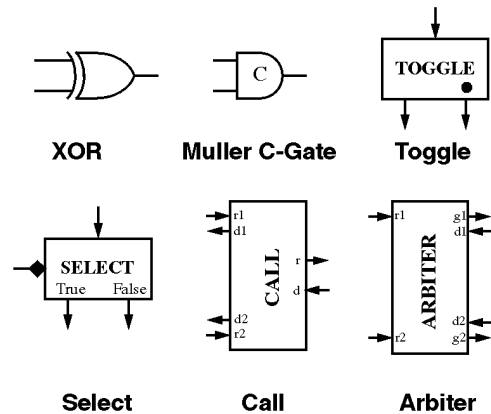


Fig. 4. Event driven logic family.

The OR function for events is easily provided by the Exclusive-OR circuit of the level-sensitive family as any input transition, or “event,” results in an output transition. The AND function requires a Muller C-gate [9] which outputs a transition only when all inputs have experienced transitions

changing their input levels from those which caused the existing output state. Other logic functions peculiar to transition logic include the following: A “Toggle” circuit which steers input events alternately to each of two outputs, a “Select” block which steers an input event to one of two outputs determined by the state of a Boolean predicate on a control input, a “Call” module which accepts events requesting access to a shared resource and directs an eventual response to the appropriate requesting source. Finally, an “Arbiter” module is also necessary in any asynchronous design where there can be multiple uncoordinated requests for a resource.

The asynchronous design style differs from the synchronous with respect to arbitration because the problem of allowing sufficient delay for an arbiter to resolve metastability with acceptable probability does not occur. Provided that the resolution of metastability can be detected, it can be used to permit the continuation of processing, eliminating the possibility of arbitration failure [10].

3 EVENT DRIVEN PIPELINES

The design of the storage element to be used in the registers of a digital system is of particular importance in view of the large number of elements involved in a typical design. An event driven storage element differs from a conventional design in that it must respond identically to rising and falling transitions.

Sutherland’s approach employs a “capture-pass” latch as a data storage element; an implementation is illustrated in Fig. 5. This comprises two simple latch circuits side-by-side which are activated alternately. The switches connecting *Din* to the latches change over in response to an event on the *Capture* line while the switch connecting *Dout* to the latches changes over in response to *Pass* events. The capture-pass latch is transparent until an event occurs on its *Capture* line; this causes the latch to hold any data which was at its input line, *Din*, at that time. The *Capture-Done* event signals that the capture operation has completed and *Dout* now represents the captured data. Any change on *Din* will now have no effect on the *Dout* value. An event on *Pass* signals that the latch contents have been consumed and that the latch may return to its transparent state, ready for the next data value and input event. The *Pass-Done* event signals the completion of the pass operation.

Capture-pass latch structures can be interconnected to form micropipelines using Muller C-gates to ensure correct operation of the bundled data protocol; Fig. 6 shows a basic

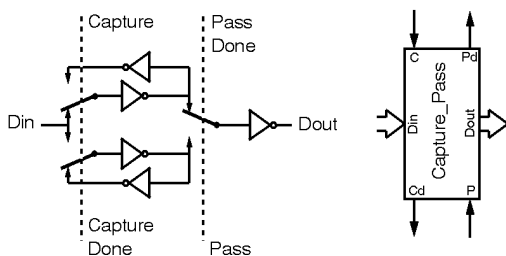


Fig. 5. Capture-pass data storage element.

micropipeline structure; here the Muller C-gates have one input primed, at reset, to permit initial switching in response to a single transition on the other input; a “bubble” indicates the primed input. On initialization, all C-gate outputs are initialized to zero; the inversion is thus necessary to “prime” the C-gate for firing on the first event received on *Rin*. This will cause the first stage to capture the data presented at *Din*. On completion of the capture, an acknowledge is returned to the sender of *Din* as an event on *Ain*; the sender can then respond with new data on *Din*. This acknowledge event is also propagated down the pipeline, via a delay, to the C-gate controlling the second capture-pass stage. The delay element ensures that data is valid at the second stage prior to the issue of a capture event. This delay clearly must embrace the delay of any combinatorial logic placed between stages and represents the local delay management.

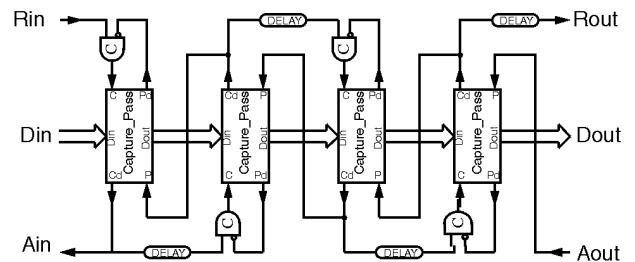


Fig. 6. Sutherland micropipeline structure.

The second stage C-gate will also be primed after initialization, so a capture event will be generated and thus the data will be latched. On *Capture-Done*, the second stage latch will forward an event to the next stage and also send an event back to the preceding stage to signal that the input data may be removed. This event resets the first stage latch to its transparent state and, on *Pass-Done*, primes the input C-gate ready for the next input event. This process continues down the pipeline until an output request is generated at *Rout*.

Further data can be input to the pipeline which will progressively fill if data is not removed quickly enough and eventually stall at the input C-gate of the first stage. New input data at *Din*, with a corresponding event on *Rin*, cannot then progress until the first stage capture-pass latch becomes transparent, this being similarly stalled on the second stage, and so on. An acknowledge event from the receiver on *Aout* signals that the data on *Dout* has been consumed, thus enabling the pipeline data to progress a stage further and generate a new *Rout* event with its corresponding *Dout*.

3.1 Practical Micropipeline Registers

Processor pipelines make heavy use of registers and hence there is a multiplied cost for every extra transistor used in a latch design. Fig. 7 shows a CMOS implementation of a capture-pass latch, requiring 18 transistors, juxtaposed with a conventional pass-transistor transparent latch requiring only six transistors. The ARM microprocessor requires a 32-bit wide data path and a compact, high density, layout was

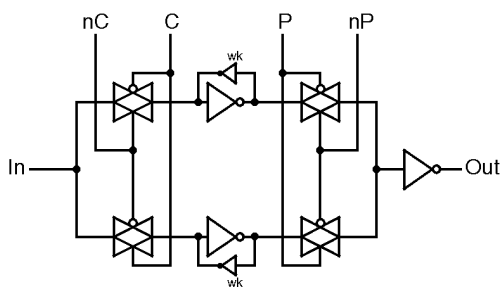


Fig. 7. Capture-pass and pass-transistor latch structures.

regarded as vital; in view of this the capture-pass latch was deemed too costly in terms of area and transistor count, so the more conventional transparent latch design was chosen instead.

The adoption of a conventional transparent latch as the basic register component yields an efficient 32-bit wide datapath implementation but introduces the need to control level-sensitive latches in an event driven environment. Thus there is the need to convert from the two-phase protocol to four-phase level sensitive signals to operate the latches and then to convert back to two-phase again to interface with the next pipeline stage.

Fig. 8 shows a practical circuit suitable for the control of a micropipeline register comprising transparent latches. The two-phase transition protocol is preserved with R_{in} and A_{in} performing the latch input handshake protocol; control signals En and nEn drive the enable lines of the latch registers. After initialization, the latch will be transparent with all event lines low. An event on R_{in} will therefore propagate through the primed C-gate, exclusive-OR, and drive buffer circuits, closing the transparent latches. To sense that these latches have closed fully, the En and nEn lines are connected to a C-gate. When both latch control signals have changed state, an event is propagated through the C-gate and Toggle, generating an output request R_{out} and an input acknowledge A_{in} . The event on R_{out} indicates that the latch output data is valid and the input data can be removed. A subsequent input request on R_{in} will be stalled by the input C-gate.

An output acknowledge on A_{out} signals that the latch data has been consumed. This event propagates through the exclusive-OR and drive buffer circuits to re-open the latches. The C-gate again detects that both latch control signals have changed state and the resulting event is steered by the Toggle to the input C-gate thus priming it for the next input request (which may, or may not, already have arrived).

Delay management to ensure correct operation of the micropipeline stage incorporates the sensing of changes on both phases of the latch enable signals to establish that

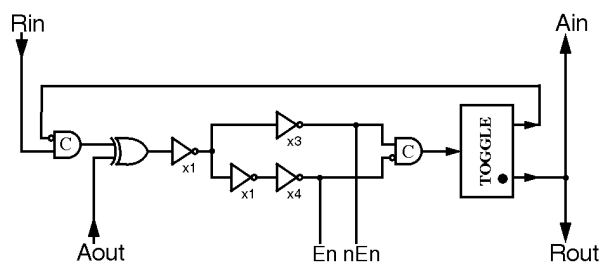


Fig. 8. Asynchronous ARM micropipeline latch control circuit.

latches are all in transparent or hold mode. Where the propagation of the enable signals across the width of the register may be significant, sensing should be at the most remote point. Where there is a processing delay prior to data being entered into a register, this must be matched by a delay in the signal path from R_{in} to the enable drive buffers. The arrangement of Fig. 8 has a large safety margin with regard to the timing of the data bundle relative to R_{out} due to the delay through both C-gate and Toggle to the event on R_{out} . Where the R_{in} to R_{out} delay imposes undue latency on the data, it is possible to derive the R_{out} signal directly from the input C-gate; great care is then necessary in the circuit layout of the stage to ensure that the data bundling constraints are not violated.

4 ARCHITECTURE CHOICE

The objective of the AMULET project was to investigate the viability of applying an asynchronous approach to the design of a digital system of significant complexity, and to evaluate the impact of asynchronous operation on performance and power-efficiency. To make such an evaluation credible, a comparison with a conventionally designed system was desirable. To this end the ARM architecture from Advanced RISC Machines Limited was selected as the target for re-implementation.

The ARM [3] was originally developed by Acorn Computers Limited in the UK in 1983-1985 and was the first commercial RISC. It drew its inspiration principally from the original Berkeley [11] and Stanford [12] work. It is a synchronous, 32-bit design using a load/store architecture and a register oriented instruction set. It has three pipeline stages comprising instruction fetch, decode, and execution. It is characterized by being both small and simple (the original silicon used only 25,000 transistors) and is therefore inexpensive and has low power requirements. The ARM is now a world standard architecture for a range of low-cost and low-power applications.

The choice of the ARM as the basis for AMULET1 was influenced by the relative simplicity of this early RISC processor, the familiarity of one of the authors with its design, and the active support of ARM Limited, a company spun out from Acorn to develop and exploit the architecture. It presents a difficult target in terms of performance and power consumption since the ARM architecture has been developed over many years to optimize the performance/power metric. For example, the current ARM6 macrocell delivers over 100MIPS/Watt.

The ARM has 16 registers programmer-visible at any one time from a total of 31. Register 15 is the program counter, register 14 holds the return address on subroutine call, other registers are general purpose. Exceptions are handled by entering privileged modes which switch in mode-specific registers in place of user registers 13 and 14; the former usually pointing to a private stack in main memory and the latter containing the exception return address. The fast-interrupt-mode also has some additional private work registers. Register organization is shown in Fig. 9.

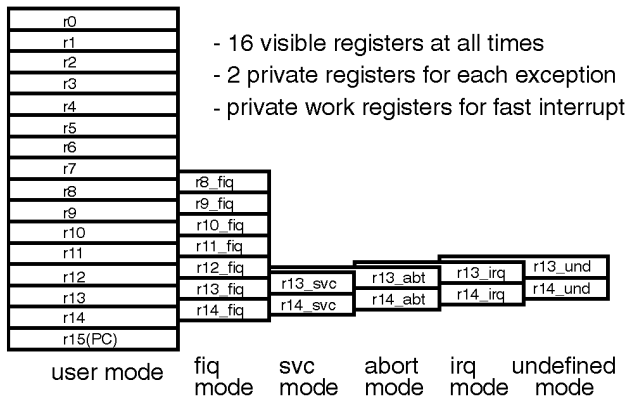


Fig. 9. ARM register organization.

In application code, the processor normally operates in user-mode and switches to one of five privileged modes when an exception arises. An exception may be software induced, by an undefined instruction for example, or may be caused by external hardware. In the latter case, two interrupts of differing priorities are provided, as is the ability to signal a memory fault to allow various memory protection and translation schemes to be supported. Memory faults are implemented as exact exceptions so that the faulting access may be retried after operating system intervention to make the memory accessible.

In addition to the general registers described above, there are six Program Status Registers (Fig. 10). The Current Program Status Register (CPSR) is visible in all modes and contains the current processor mode, interrupt enable status and condition code flag bits. There are also five Saved Program Status Registers (SPSR) corresponding to each of the exception modes. These are used to save the

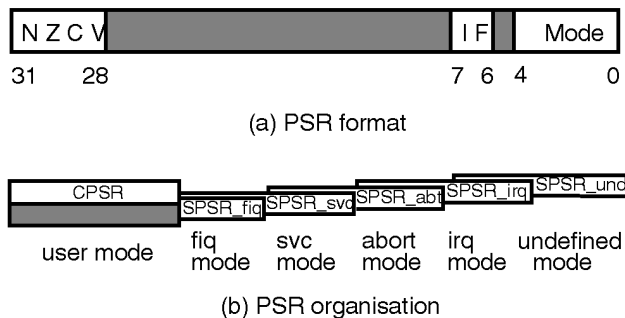


Fig. 10. ARM program status registers.

Cond	00	Opcd	S	Rn	Rd	Operand 2			Data op	
Cond	000000	AS	Rd	Rn	Rs	1001	Rm		Multiply	
Cond	00010	E 00	Rn	Rd	0000	1001	Rm		Swap	
Cond	01	FUSWL	Rn	Rd	Offset				Load/Store	
Cond	011	XXXXXXXXXXXXXXXXXXXXXXXXXXXX							XXXXXXXX	Undefined
Cond	100	FUSWL	Rn	register list						Block xfer
Cond	101	L	offset							Branch
Cond	110	FUNWL	Rn	CRd	CP#	offset			Coproc L/S	
Cond	1110	CPop	CRn	CRd	CP#	CP 0	CRm		Coproc op	
Cond	1110	CP	L	CRn	Rd	CP#	CP 1	CRm	Coproc reg	
Cond	1111	ignored by processor								SWI

Fig. 11. ARM instruction set.

CPSR when an exception occurs and to restore it when the exception handler returns.

The ARM instruction set [13] is summarized in Fig. 11. In common with other RISC processors, ARM separates instructions which perform data processing functions from those which move data between memory and registers. Two unusual features of the instruction set are the conditional execution of all instructions and the presence of block transfer instructions. Conditional execution reduces the number of branches required, allowing, for example, some if-then-else clauses to be compiled without a branch instruction. The conditional execution is based on the state of the condition code flags set by arithmetic and logical operations. The block transfer instructions can move any subset of the currently visible registers to and from memory; their use improves the efficiency of procedure entry and exit and increases the rate of data block moves.

Referring to Fig. 11, which summarizes the instruction set, Data Operations perform ALU functions on one or two register operands and place results in destination registers. A barrel shifter in the data path allows one of the operands to be shifted by an arbitrary amount prior to an ALU operation. The Multiply instruction performs 32 by 32 bit multiplication of two registers and places the least significant 32-bit result in a register. Load/Store instructions transfer single words between registers and memory with a wide variety of addressing schemes, including optional auto increment/decrement of index registers. The Block Transfer instructions move any subset of the 16 registers to/from memory with a more limited set of addressing modes but still providing auto increment/decrement of the index register. The Swap instruction performs an indivisible read/write cycle on a single memory location, swapping register contents with the memory. The Load/Store and Swap instructions can specify their operands as words or bytes.

The Branch instruction performs a PC relative branch within a 64 Mbyte range and can optionally save a return address in R14. The Software Interrupt (SWI) provides a means to enter operating system code and contains a 24-bit field which may be used to request specific services. The ARM also supports instruction set expansion either by software, using the Undefined instruction formats, or by hardware coprocessors. Three classes of coprocessor instruction are defined to communicate with up to 16 external coprocessors.

5 AMULET1—THE ASYNCHRONOUS ARM

The AMULET1 processor [14], [15], [16] is divided into four major functional units; their relationship and the pipelining applied to these will be described. The major units, shown in Fig. 12, comprise the Address Interface, Register Bank, Execute Unit, and Data Interface; these operate concurrently and asynchronously, synchronizing with each other only to exchange data. The data interface and the execute pipe are relatively simple pipelines from the control viewpoint; the register bank incorporates a novel coherency mechanism [17] to handle read after write dependencies and the address interface autonomously issues instruction prefetches to a nondeterministic, but bounded, depth.

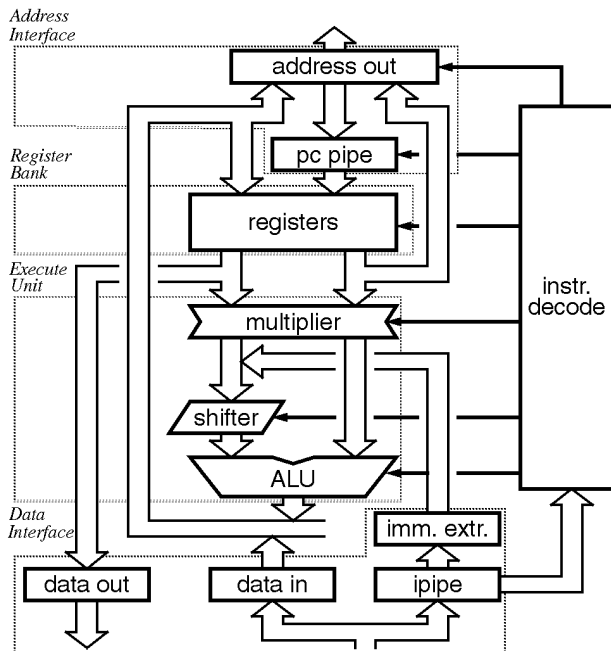


Fig. 12. AMULET1 organization.

Each of the functional units has its own internal pipeline structure; the pipeline depth of AMULET1 is thus considerably greater than that of the synchronous ARM. The following sections describe these functional units in greater detail with particular emphasis on features introduced because of the asynchronous nature of the design.

5.1 The Address and Data Interfaces

The address interface includes a PC incrementing loop for the autonomous prefetching of instruction data; this is shown in Fig. 13. The automatic prefetch requests instruction words until there is a risk of deadlock because accommodation in the pipeline can no longer be guaranteed. At any time during this process, a memory address from the ALU may arrive and an arbitrating multiplexer is required to break into the autonomous prefetch loop to route this operand address to the memory system. Because of the asynchronous nature of the requests the precise point at which the prefetch is interrupted is non-deterministic, resulting in an unknown depth of prefetching beyond a

branch. The interruption to the PC-increment process may be transitory, in the case of a data access, or permanent, in the case of a branch. In the former case the PC value is preserved in the PC register and prefetching continues once the data address has been accepted by the memory; in the latter case the old PC value is discarded and the new value takes over the prefetch loop.

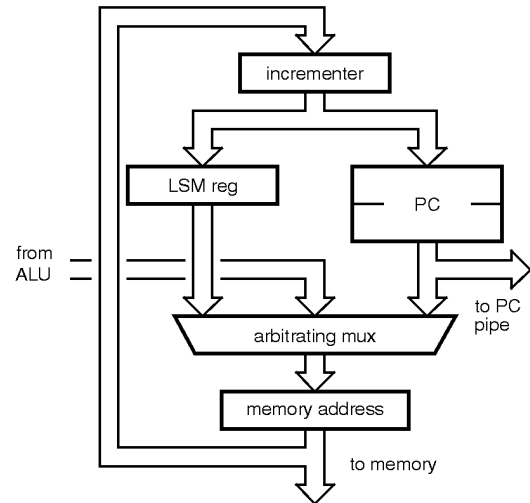


Fig. 13. The address interface.

An "LSM" (load/store multiple) register is shown alongside the PC register in Fig. 13. This is included so that the address incrementer may be used to enhance the performance of load/store multiple operations without intervention from the datapath. Prefetching is suspended during this process but resumes once the transfers are complete.

The data interface receives data from the memory system and routes it to the instruction pipeline, register bank, or address interface as appropriate. Instructions are buffered in the IPIPE of Fig. 12 to await decoding; immediate values are extracted at the top of the pipeline for use as operands as required. Loaded data values are aligned and byte extracted as specified by the instruction. Data to be written to memory is also passed through this unit where it is synchronized with the address and control information from the address interface prior to passing to the memory as a single bundle.

The feature of the ARM instruction set which permits the specification of the Program Counter (PC) as an operand poses a particular problem in a pipelined implementation where the depth of instruction prefetch is nondeterministic.

In the synchronous ARM, when the PC is read as a register, the value $PC + 8$ (bytes) results; this is an address two words ahead of the current instruction position. This peculiarity is an artifact of the original three-stage pipeline of the synchronous ARM design which must be reproduced in AMULET1 for compatibility. In AMULET1, it is necessary to provide the appropriate PC value at the point of execution for any instruction specifying PC as an operand, although the PC value has continued to increment under the control of the prefetch system. To this end, a FIFO buffer,

the PC-Pipe, is incorporated to hold PC values corresponding to instructions in the process of being fetched from memory or pending execution in the Instruction FIFO. This organization is illustrated in Fig. 14 which shows the synchronization of an instruction with its corresponding PC prior to execution.

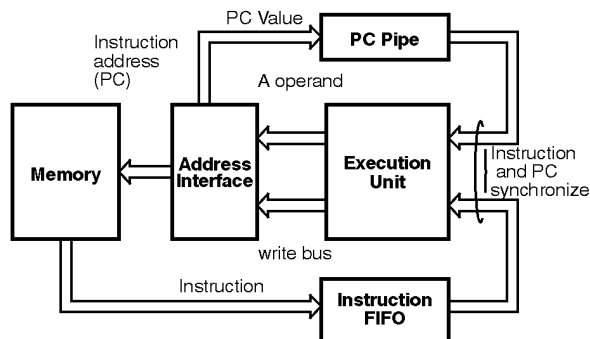


Fig. 14. PC and instruction synchronization.

5.2 Pipeline Disruption Handling

Pipeline flow will be disrupted by program-dictated events such as branches, by interrupts, and by exceptions such as memory faults. The challenge in the design of a pipelined processor is to reconcile performance requirements, which encourage designers to push speculative execution as far as possible, with recovery complexity constraints, which seek to limit state change until correctness is established so that minimum state needs be preserved on interruption to processing.

Branching events change the flow by injecting the branch target address into the address interface asynchronously to the operation of the PC incrementing loop; this is done while prefetching from the interrupted instruction stream is still pending. Instructions prefetched beyond the branch instruction must be discarded. This process is complicated by the nondeterministic depth of prefetch beyond the branch instruction so that the number of instructions to be discarded is uncertain. This problem is addressed by an instruction stream “coloring” approach whereby every instruction fetch is allocated a color corresponding to the current operating color of the processor. This color is changed whenever the execution of the current instruction stream terminates, all new instruction fetches bear this new color. Thus if the “fetch color” of an instruction presented for execution does not match the current operating color then the instruction is discarded. As only one branch instruction may be outstanding at any time, this coloring process may be implemented by use of a single “color” bit which accompanies each instruction fetch; every branch instruction will toggle the fetch and operating colors and a simple color check will suffice to validate an instruction for execution.

A check that the color of the instruction under execution conforms to the current color of the processor is made at the ALU result stage so it may be abandoned prior to writing to its result destination if the color match fails. In addition, the color matching is also checked at the decode stage so that once a processor color change has been established pre-

fetched instructions entering the decode stage can be abandoned without passing through the pipeline to the ALU stage. This continues until instructions from the new target stream, which match the processor in color, are encountered with the consequential saving of power consumed by useless pipeline activity.

Instruction flow can also be changed by error conditions. For example, an attempt to access a nonexistent memory page will cause a fault. A failing memory access on instruction fetch causes a trap to be entered but it is important that this is not caused erroneously by the prefetch of an instruction which is *not* eventually required. To handle this requirement, a failing instruction access will complete the transaction returning void data accompanied by a Boolean flag marking it as invalid. A trap resulting from a failed memory access for instruction data is therefore initiated only when the void instruction enters the decode stage and it can be established from its color match that this instruction was genuinely required. If, however, the color match fails, the instruction is discarded and no trap results from this instruction access.

In a virtual memory environment a memory fault due to an access to a data page not currently within the upper levels of a memory hierarchy is a normal, if infrequent, event. It is a requirement that program execution should resume seamlessly once the absent page has been moved to an accessible memory level by operating system action initiated by the exception. The processor handles all memory faults as *exact exceptions* and thus must inhibit any irreversible change of internal state until the success of an access is established; operations could be suspended until a load or store operation completes but this would degrade performance severely. The approach adopted is to hold an instruction subsequent to a load/store instruction at the ALU stage until it has been established that the load/store instruction will not cause a memory fault.

Thus, for an operand data access, an *abort/no-abort* response from the memory system is critical to the performance of the processor; a subsequent instruction cannot proceed to completion until a *no-abort* response is received. If a *no-abort* is signaled the processor operation can continue in parallel with the data transfer. If an *abort* is indicated, the operating *color* of the processor is changed immediately thus causing instructions behind the faulting one to be discarded.

When a load/store instruction causes a memory fault, the corresponding PC value is required for the exception to be exact and to allow resumption of the instruction flow. While in the synchronous implementation the PC value can be calculated, in AMULET1 it is necessary to preserve the PC value of a load/store instruction passing the execution point until a response indicating the success of the access has been received from the memory system. To this end the instruction address at the end of the PC pipe at the time of execution is passed on to another FIFO register, the X-pipe as shown in Fig. 15; this will give a precise location of the faulting instruction should a memory fault occur and cause the instruction to be aborted. The memory system *no-abort* response which releases the instruction at the execution point also results in discarding the entry at the end of the X-pipe; an *abort* response however results in the copying of

the entry into the exception latch (X-latch). The presence of a value in the X-latch causes a data abort exception to be raised and the exception entry process uses the X-latch value to form the return address in this case.

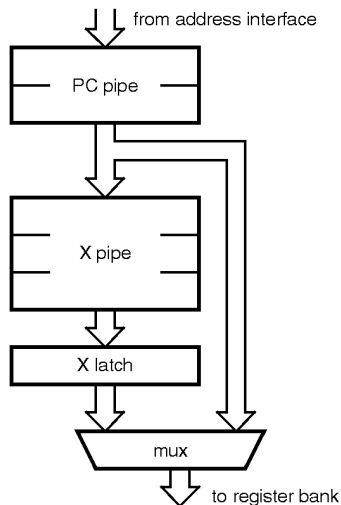


Fig. 15. The PC pipelines.

5.3 The Register Bank

A high level view of the register bank in relation to the execution unit of the processor is given in Fig. 16. The register bank is designed as a three port system to support the two-source plus one-destination register requirement of the ARM instruction set. The pipelined design of AMULET1 must handle multiple pending write operations to the register bank and the read locking of any register subject to alteration by an outstanding instruction; this must all be accomplished in an environment where read and write operations must interact asynchronously. All these requirements are met by the use of an asynchronous FIFO Lock Register [17], [18] to hold the addresses of registers with pending write operations. Addresses are queued in a fully decoded form so that each register of the FIFO contains, at most, a single "1"; this structure is shown in Fig. 17. A column of the FIFO thus contains all information pertaining to pending writes to a particular register and a logical OR of these column bits provides the register lock control. Such an OR function would not normally be permissible across an asynchronous structure but is possible here because the signaling protocol of the Sutherland Micropipeline ensures that data is not removed from a pipeline stage until it has been established in the succeeding stage; the lock output is thus glitch-free.

The lock information is used to defer the application of the decoded read word lines until the required register contents are valid as illustrated in Fig. 18. Multiple locks on a single register can be handled correctly and no arbiter is necessary to manage the asynchronous interaction between read and write operations; these proceed independently when there is no interdependency and the lock mechanism synchronizes them when a dependency is encountered.

It can be seen from Fig. 17 that two lock FIFOs are employed in the design. One is used to lock registers sub-

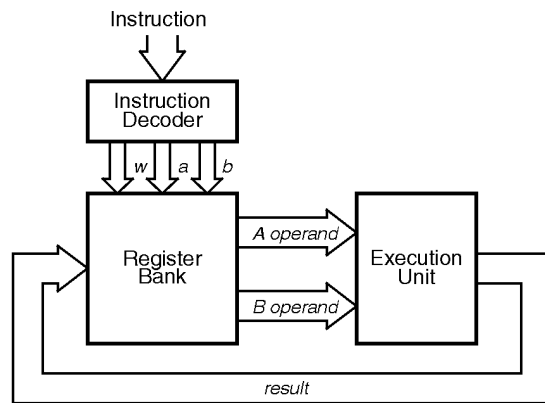


Fig. 16. Register bank overview.

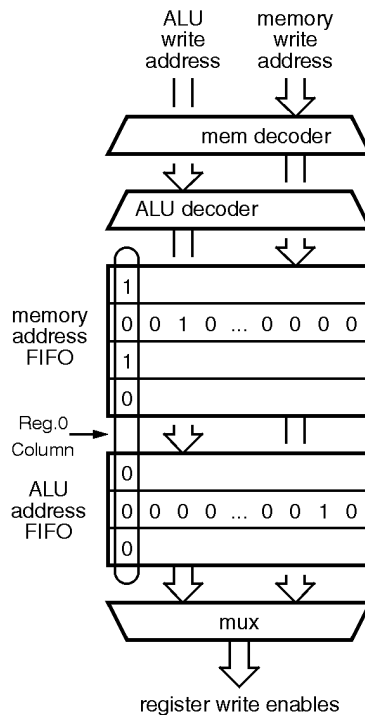


Fig. 17. Lock FIFO.

ject to alteration by data requested from the memory system while the other locks registers which are destinations of ALU based instructions; the latter may thus overtake responses from the potentially slower memory. Any contention between write requests from the two data sources is arbitrated prior to their presentation to the register bank.

The overall organization of the register bank is shown in Fig. 19. Only a brief description of its operation is possible here; a full description of its design and operation is given elsewhere [17]. For a read operation, a new instruction has its availability signaled by *I-Req*, and presents two register addresses to be read (*a* and *b*) and a register address to be written (*w*) once the execute unit result is available. *I-Req* is stalled until the register bank is ready to start a new read operation when the read decoders are enabled. Concurrently with the read address decoding, the write register address

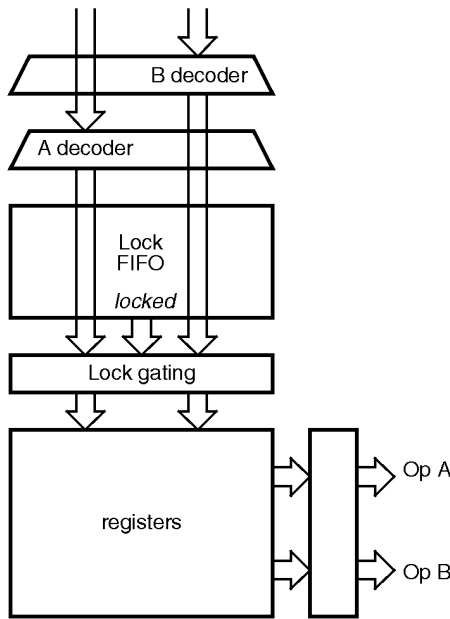


Fig. 18. The register read logic.

mize the cell size, with charge retention circuits to give pseudo-static operation. An extra 33rd bit line gives a matched completion signal (*A Done / B Done*). When both register values are available, they are latched and passed to the execution path (via the *D-Req* signal) which can process the data with no further delay.

Once the data has been latched, the read decoders are disabled and the read bus precharge is turned on to prepare for the next access. Normally the write address will be latched well before this time, and the instruction acknowledge (*I-Ack*) is issued so that a new instruction can be prepared during the register recovery time.

The write decoder is disabled during the read operation to present inactive inputs to the lock FIFO. Once the read data is latched, the write decoder is enabled and the destination register is locked. As soon as the lock FIFO has accepted the new address, a new instruction may be allowed to start its read operation. The lock logic will continue by disabling the write decoder, it will then free the write address latch for the next value.

For a register write operation, a data value (signaled on *W-Req* in Fig. 19) is paired with the decoded write address at the output of the lock FIFO, this rendezvous being implemented by means of a single Muller C-gate. The appropriate write word line is then enabled and the data written, following which the destination register is unlocked for reading by removing its address from the FIFO. The write operation is self-timed by detecting the transitions on the word line with a wide dynamic OR gate; the same circuit ensures that writes are fully disabled before the write data is allowed to change.

The interfaces use a bundled-data convention using transition signaling. Internally, the design employs a combination of two-phase and four-phase techniques, the latter being well matched to the precharge-active cycle of the dynamic circuits used in the basic register cell.

5.4 The Execute Unit

The Execute unit comprises two pipelined function units as shown in Fig. 20. The register operands first encounter a multiplier which passes them on immediately, if unactivated, or performs a carry save multiplication process and passes on, instead, the partial product and partial carry outputs. One operand path then passes through a barrel shifter which may modify one of the operands before both are placed in a pipeline latch. The operands are then combined in an ALU which has a data dependent delay and a latch to allow a dynamic structure to operate with pseudo-static external behavior. A result latch passes the output to its destination, either the register bank or the address unit.

The sequential organization of these functional units is not optimal for performance but is determined by adherence to the ARM instruction set which supports both shift and ALU operations in a single instruction thus requiring the barrel shifter to be in series with an ALU input. Multiplication, as implemented, also uses the ALU for the final carry-propagate addition of the partial product and carry components; if no multiplication is specified the delay in passing the operands through the multiplier is equivalent to that of passing them through a multiplexer to bypass this unit.

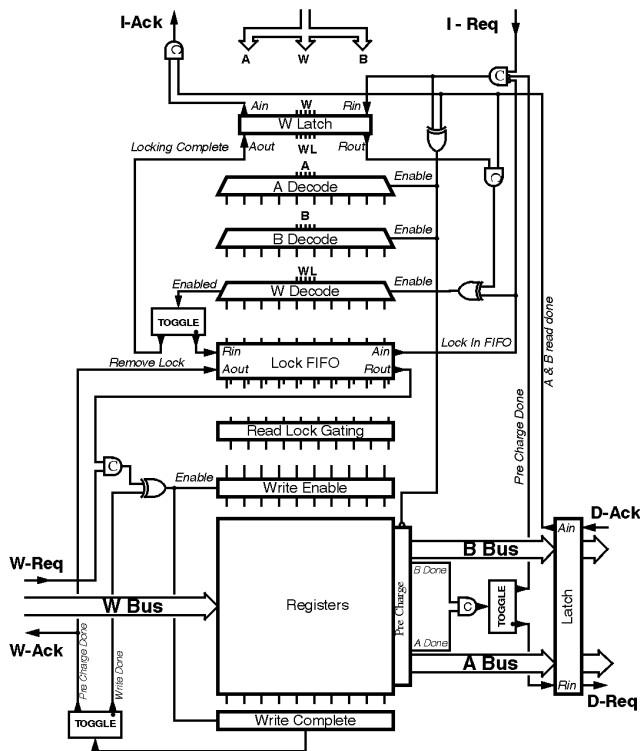


Fig. 19. Organization of the register bank.

is latched (in *W Latch*).

The decoded read addresses present *enables* for the selected registers which are gated with the locked register information. A read of an unlocked register will proceed, whereas a read of a locked register will stall at this point until a write operation clears the lock.

The register read circuitry uses dynamic logic to mini-

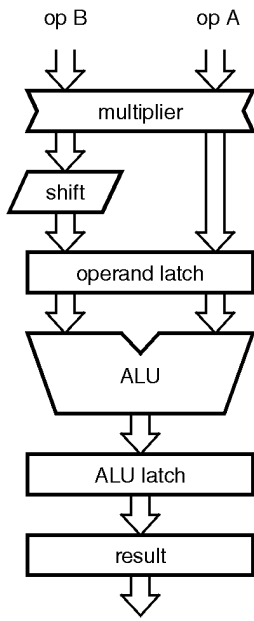


Fig. 20. The execution pipe.

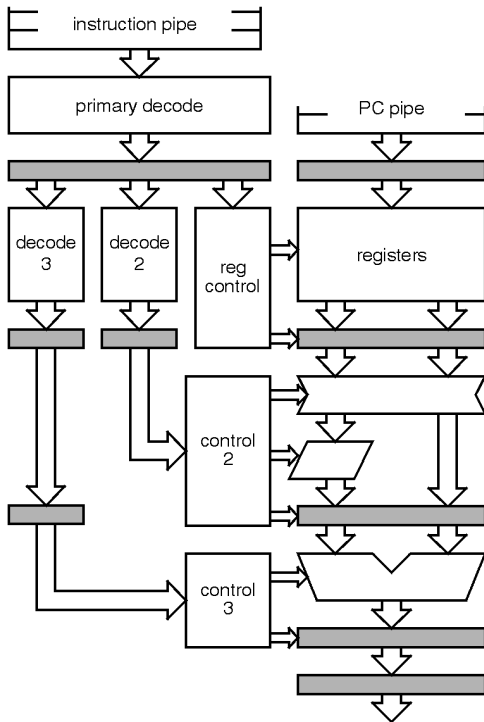


Fig. 21. Control structure.

The instruction decode and execute-pipe control logic are illustrated in Fig. 21 where the pipeline latches are shaded to clarify the structure. Prefetched instructions are queued before being passed to the primary decode logic whose function includes interrupt arbitration, the early discarding of redundantly prefetched instructions, and generating multiple pipeline entries to implement the more complex instructions; it also sends appropriate read and write addresses to the register bank for each pipeline entry and

passes information on to the secondary and tertiary decode logic. The execution pipeline of Fig. 20 is reproduced in the bottom right of Fig. 21 and the secondary and tertiary decoders control their respective stages of the execution pipe. Note that although the shaded pipeline latches are aligned to emphasize the matching of pipeline depths of the parallel structures, this does not imply that these latches operate at the same time. Synchronization occurs only when the pipelines interact, for example, where *control 2* meets the multiplier and where *control 3* meets the ALU.

The ALU implementation illustrates a particular opportunity to exploit the asynchronous nature of the processing pipeline. A clocked processor, such as the ARM6, must have an ALU capable of any operation on any operands within the clock cycle time; this usually entails the design of a fast carry path for additions to handle cases when a “carry” propagates the length of the addition chain. Such cases are rare; with a 32-bit bus and random data the average carry propagation is fewer than five bits, with typical data it is approximately 11 bits [19].

An asynchronous processor need not constrain its ALU to produce a result within a particular time and therefore may allow rare, worse-case, operations to take longer to complete; resources can thus be dedicated to the optimization of the typical cases. The self-timed ALU in AMULET1 was implemented as a simple ripple-through-carry adder with completion detection; each stage of this adder is identical. An illustration of the circuit is provided in Fig. 22; the actual implementation uses dynamic CMOS logic so that all the signals, including the output of the NOR gate, are initially zeroed.

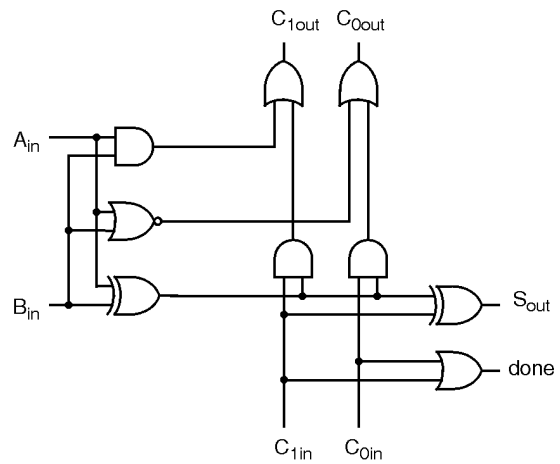


Fig. 22. Self-timed adder circuit.

Each stage of the adder has two “carry” signals activated when and if the carry output of the stage has been calculated as “0” or “1,” respectively. If possible, the two input bits are used immediately to indicate the carry output state, otherwise, the carry output must be forwarded from the appropriate carry input when it arrives. The sum is calculated when the carry input is known, as is the “done” signal. The “done” signals from all the stages are then combined using an AND gate tree to yield an overall completion signal which terminates the addition process. All the signals are zeroed again before the next operands are presented.

ALU logical operations, in contrast, are allowed to complete at their faster, natural, speed by use of a separate, fixed, timing signal to indicate their completion.

The multiplication process also benefits from the application of asynchronous techniques. The fastest iterative multiplication process utilizes carry-save addition techniques where each subproduct addition delay is determined by that of a single bit addition only; this is very small in comparison with a clock cycle time determined by a worse-case full addition delay. Fig. 23 shows a block diagram of the AMULET1 multiplier design.

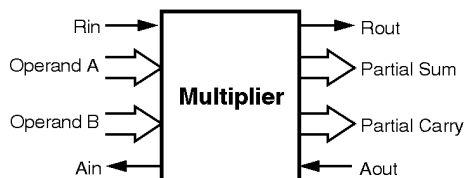


Fig. 23. AMULET1 multiplier block diagram.

The asynchronous multiplier circuit employed on AMULET1 takes two 32-bit operands, A and B, and performs an iterative carry-save addition technique to produce 32-bit partial sum and carry results. These are subsequently added using the main ALU to yield the required product.

Fig. 24 shows a block diagram of the AMULET1 multiplier datapath design. This is centered around a block of full-adder stages with an input latch stage, controlled by the signal *Ck1*, and an output latch stage controlled by the signal *Ck2*. The input latch stage holds the A and B operand values and the sum and carry input values to the full adder. The output latch holds copies of the A and B operands and the resulting sum and carry outputs from the full-adder stage. Data from the output latch is then fed back to the input latch with an appropriate shift applied to operands A, B, and carry out. The full-adder circuits conditionally add the shifted B operand to the input sum and carry operands dependent on the least significant bits of the shifted A operand.

Signals *Ck1* and *Ck2* are generated by an asynchronous control circuit and are guaranteed to be nonoverlapping. In AMULET1, three subproduct additions are performed in each asynchronous iteration using three cascaded levels of carry-save addition; the minimum delay between register stages dictated by the control logic implementation is long enough to accommodate the delay of three cascaded carry-save adders, which are therefore incorporated to optimize performance. Early termination, detected when the shifted B operand is all zeros, signals when the multiplication process is complete. As indicated in Table 1, the resulting AMULET1 multiplier circuit yields a very small delay per multiplier-bit time of 3 ns; in a synchronous system, where the clock period is constrained by the performance of an integer adder, full exploitation of the clock period may require the cascading of an unacceptably large number of carry-save adders.

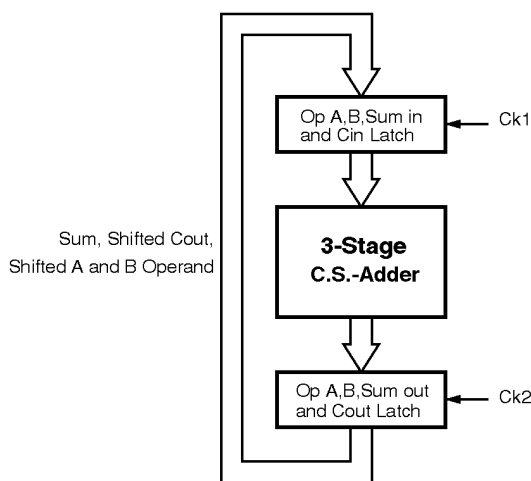


Fig. 24. AMULET1 multiplier datapath.

TABLE 1
CHARACTERISTICS OF AMULET1 AND ARM6

	AMULET1 (ES2)	AMULET1 (GPS)	ARM6
Process	1 μ m	0.7 μ m	1 μ m
Area (mm ²)	5.5 × 4.1	3.9 × 2.9	4.1 × 2.7
Transistors	58,374	58,374	33,494
Performance	20.5 kDhry.	40 kDhry.*	31 kDhry.
Multiplier	5.3 ns/bit	3 ns/bit	25n s/bit
Conditions	5V, 20°C	5V, 20°C,	5V, 20MHz
Power	152mW	N/A**	148mW
MIPS/W	77	N/A	120

* Estimated maximum performance.

** No separate core supply for power measurement.

6 IMPLEMENTATION

The AMULET1 chip was developed with the aid of existing (synchronous) design tools and using a conventional design flow. High-level simulation used a proprietary simulator from ARM Limited. Transistor level simulation and all layout was performed with commercial VLSI CAD tools from Compass Design Automation. The chip layout contains both full-custom components, primarily in the datapath, and compiled standard cell circuits, used mostly for control logic. The control circuits include two PLAs which were generated using a synthesis tool supplied by ARM Limited and modified to provide external access to the completion signal; this already existed inside the ARM design where it controlled power-down and precharging. The VLSI layout of the chip is shown in Fig. 25.

In line with standard practice, simulations were run on transistor-level netlists extracted from the physical layout in all the process corners to cover all four combinations of fast and slow n- and p-transistors. Any serious difference between matched paths in their dependence on rising and falling delays is exposed by these skewed simulations, and on AMULET1, a few paths were adjusted to compensate for asymmetries of this sort.

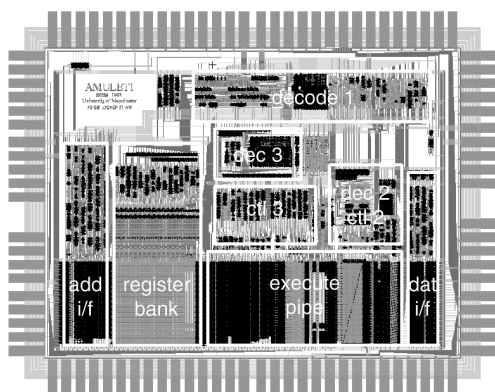


Fig. 25. AMULET1 VLSI layout.

In addition to the standard tools, software was developed to check that bundled data values never changed within the *Request-Acknowledge* period (with added margins) to ensure that the bundling constraints were not violated. These checks were made under the timing conditions which applied for the simulations undertaken, it was not possible to check for all possible timings under all possible conditions.

It was noted that the tendency for incorrect circuits to deadlock was a significant help during debugging. The behavioral simulator could be set up to maintain a circular buffer of past events. Then, when the model deadlocked, this buffer would contain enough history to identify the source of the problem. When similar problems arise with a clocked design, all that may emerge at the end of a long run is the wrong answer; identifying the time at which the error arose and its cause is then difficult.

7 TESTING

To evaluate the AMULET1, it was necessary to demonstrate it executing realistic programs. A processor card was therefore constructed which was functionally equivalent to ARM Limited's Platform Independent Evaluation (PIE) card [20]; this comprises an ARM6 CPU with an EPROM, static RAM, and a serial line for communication and loading programs. This provided a convenient environment for testing the AMULET1 processor and a demonstration of code compatibility in that an identical ROM could be used.

The external interface to AMULET1 is basically a micropipeline interface. The test board, also self-timed, needed to translate two-phase request and acknowledge signals into the four-phase "enables" required by conventional devices. Timing for devices such as the RAMs is provided by delay lines.

Although it proved possible to design the requisite two-phase control circuits with conventional programmable logic devices, this was cumbersome and, in some circumstances, limits the system performance.

8 RESULTS

AMULET1 has been fabricated on two CMOS processes: a $1\mu\text{m}$ process at ES2 and a $0.7\mu\text{m}$ process at GEC Plessey

Semiconductors (GPS). Both devices have been tested on the card described above which connects, via the serial line, to a workstation running development tools from ARM Limited [21]. Both prototype devices are functional and execute programs produced by standard ARM development tools such as the assembler and C compiler. A summary of the devices' characteristics [22] is shown in Table 1 with those of ARM6 for comparison.

Although the prototype asynchronous implementations do not show an overall gain compared with ARM6 the performance of the multiplier demonstrates the advantage of independence from a global clock signal dictated by worst-case ALU performance. The addition process in AMULET1 is also freed from worst-case constraints; its self-timed data-dependent operation signals completion once the carry has ceased to propagate [19]. The development of the AMULET design continues and subsequent versions will address the performance limitations of the prototype and should present a greater challenge to the synchronous design.

The devices have been characterized over voltage and temperature ranges and were found to display the usual property of asynchronous devices, namely the ability to adapt automatically to changing environmental conditions. The performance and power-efficiency of the $1\mu\text{m}$ part with varying voltage is shown in Fig. 26, using the Dhrystone benchmark as an indicator of performance. The $0.7\mu\text{m}$ part operates at twice the speed but does not have the facility to measure core power consumption. The voltage range used for these tests is limited to a minimum of 3.5V by the other components on the test card. In isolation, the processor appears to operate to below 2.5V.

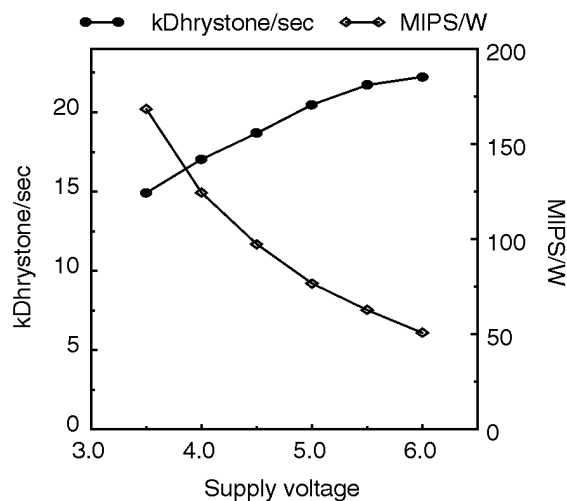


Fig. 26. AMULET1 performance and power efficiency.

The variation of operating speed with temperature was also measured; the test device exhibits the expected increase in delays of 0.3% per $^{\circ}\text{C}$ and operates correctly between -50°C and 120°C .

Four minor design errors were eventually identified in the AMULET1 prototype silicon; these related to the operation of interrupts and had relatively straightforward workarounds. One was in the implementation where a latch had been wired

incorrectly and was exhibited only by the part from one source. It was dependent on the relative speed of the attached memory which in the test card was critical to the correct operation of one source's part but not to the other's. The other errors affected the preservation and restoration of processor status registers under certain conditions; these were errors in design which should have been eliminated by a more thorough simulation of the design prior to fabrication. None of the problems identified could be attributed to the asynchronous nature of the design approach and all will be eliminated in subsequent designs.

9 CONCLUSIONS

The AMULET1 exercise has implemented a commercial RISC microprocessor architecture as a fully asynchronous system based on Sutherland's Micropipeline approach and using conventional VLSI design tools. The prototype is competitive with the best synchronous design in power efficiency and is close in performance and silicon area. As a system component, the AMULET1 device is flexible, automatically adjusting its performance to changes in temperature and supply voltage and using power only to perform useful work.

Experience gained in the design of AMULET1 and in subsequent analyses suggests that there is considerable scope for improvement in both power-efficiency and performance, but the area penalty for the asynchronous control logic is more difficult to address and will remain. An exact estimate of the cost of the asynchronous control overhead is difficult because the synchronous ARM implementation used for comparison employs a much simpler pipeline. However, the area occupied by asynchronous control elements is less than 20% of the total [16]; some synchronous designs require a similar proportion of the chip area to be dedicated to the clock drivers.

Although much work remains to be carried out before asynchronous design is reestablished as a viable technology for widespread commercial use, the AMULET1 project shows that asynchronous design is feasible on a commercially interesting scale. The reign of the global clock is not about to be ended, but micropipelines represent a viable alternative approach.

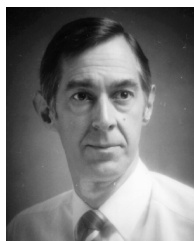
ACKNOWLEDGMENTS

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.

REFERENCES

- [1] I.E. Sutherland, "Micropipelines, The 1988 Turing Award Lecture," *Comm. ACM*, vol. 32, no. 6, pp. 720-738, June, 1989.
- [2] D.W. Dobberpuhl et al., "A 200-MHz 64-b Dual-Issue CMOS Microprocessor," *IEEE J. Solid-State Circuits*, vol. 27, no. 11, pp. 1,555-1,565, Nov. 1992.
- [3] S.B. Furber, *VLSI RISC Architecture and Organisation*. New York: Marcel Dekker Inc., 1989.
- [4] A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, and P.J. Hazewindus, "Design of an Asynchronous Microprocessor," *Advanced Research in VLSI 1989: Proc. Decennial Caltech Conf. VLSI*, C.L. Seitz, ed. MIT Press, 1989.
- [5] F. Asai, S. Komori, T. Tamura, H. Sato, H. Takata, Y. Seguchi, T. Tokuda, and H. Terada., "Self-Timed Clocking Design for a Data-Driven Microprocessor," *IEICE Trans.*, vol. E-74, no. 11, pp. 3,757-3,765, Nov. 1991.
- [6] G. Gopalakrishnan and P. Jain, "Some Recent Asynchronous System Design Methodologies," Technical Report UU-CS-TR-90-016, Dept. of Computer Science, Univ. of Utah, Oct. 1990.
- [7] A.J. Martin, "Formal Program Transformations for VLSI Circuit Synthesis," *UT Year of Programming Inst. Formal Developments of Programs and Proofs*, E.W. Dijkstra, ed. Reading, Mass.: Addison-Wesley, 1989.
- [8] K. van Berkel, *Handshake Circuits: An Asynchronous Architecture for VLSI Programming*, Int'l series parallel computation. Cambridge Univ. Press, 1993.
- [9] R.E. Miller, "Sequential Circuits," chapter 10, *Switching Theory*, vol. 2. New York: Wiley, 1965.
- [10] C. Mead and L. Conway, *Introduction to VLSI Systems*. London: Addison-Wesley, 1980.
- [11] M.G.H. Katevenis, *Reduced Instruction Set Computer Architectures for VLSI*. Cambridge, Mass.: MIT Press, 1985.
- [12] J.J. Hennessy, N. Jouppi, F. Baskett, and J. Gill, "MIPS: A VLSI Processor Architecture," *Proc. CMU Conf. VLSI Systems and Computations*, pp. 337-346. Rockville, Md.: Computer Science Press, 1981.
- [13] A. van Someren and C. Atack, *The ARM RISC Chip, A Programmer's Guide*. Addison-Wesley, 1993.
- [14] S.B. Furber, P. Day, J.D. Garside, N.C. Paver, and J.V. Woods, "A Micropipelined ARM," *Proc. IFIP TC 10/WG 10.5 Int'l Conf. Very Large Scale Integration (VLSI '93)*, T. Yanagawa and P.A. Ivey, eds. North Holland, Sept. 1993.
- [15] S.B. Furber, P. Day, J.D. Garside, N.C. Paver, and J.V. Woods, "AMULET1: A Micropipelined ARM," *Proc. IEEE Computer Conf.*, Mar. 1994.
- [16] N.C. Paver, "The Design and Implementation of an Asynchronous Microprocessor," PhD thesis, Univ. of Manchester, U.K., June 1994.
- [17] N.C. Paver, P. Day, S.B. Furber, J.D. Garside, and J.V. Woods, "Register Locking in an Asynchronous Microprocessor," *Proc. 1992 IEEE Int'l Conf. Computer Design, VLSI in Computers & Processors*, Oct. 1992.
- [18] N.C. Paver, "Condition Detection in Asynchronous Pipelines," UK Patent no 9114513, Oct. 1991.
- [19] J.D. Garside, "A CMOS VLSI Implementation of an Asynchronous ALU," *Proc. IFIP Working Conf. Asynchronous Design Methodologies*, S.B. Furber and M.D. Edwards, eds. North Holland, Apr. 1993.
- [20] C. Atack, D. Flynn, D. Jaggar, P. Magowan, and Stuart-Avery-Design, *ARM PIE User Guide*, doc. no. ARMDUI0001a, Advanced RISC Machines Limited, Apr. 1992.
- [21] "ARM Software Development Toolkit," doc no. ARM DUI 0002b, Advanced RISC Machines Limited, Oct. 1993.
- [22] S.B. Furber, P. Day, J.D. Garside, N.C. Paver, and J.V. Woods, "The Design and Evaluation of an Asynchronous Microprocessor," *Proc. Int'l Conf. Computer Design (ICCD '94)*, pp. 217-220, IEEE CS Press, 1994.



J. Viv Woods received his BSc and MSc degrees in electrical engineering from the University of Manchester Institute of Science and Technology in 1961 and 1963, respectively, and the PhD degree in computer science from the University of Manchester in 1973. He is a senior lecturer in computer science at the University of Manchester, having moved to a faculty position in 1968; from 1963 to 1968, he was with International Computers as a design engineer. Current research interests are in the exploitation of asynchronous digital system design for low power processor applications.

Past interests have included the developments of novel mainframe processors and parallel architectures for declarative languages. In 1991, the Council of the IEE awarded Dr. Woods and his coauthor the "Computing and Control Engineering Journal Premium" for 1989/90.



Paul Day received his BSc degree in electronics from the University of Manchester Institute of Science and Technology, Manchester, U.K., in 1982. Between 1982 and 1987, he worked in the field of VLSI test and evaluation at both ICL Computers and IBM. He received his MSc and PhD degrees in electrical engineering from the University of Manchester in 1988 and 1991, respectively. In 1990, he joined the Department of Computer Science at the University of Manchester as a researcher. His main research fields

were low-power circuit design, asynchronous systems, and microprocessor design. In 1995, he cofounded Cogency Technology Incorporated and is now engaged in the exploitation of asynchronous low-power circuits and self-timed techniques for commercial applications.



Steve B. Furber received his BA degree in mathematics in 1974 and his PhD in aerodynamics in 1980 from the University of Cambridge, U.K. From 1980 to 1990, he worked in the hardware development group within the R&D department at Acorn Computers Ltd, and was a principal designer of the BBC Microcomputer and the Acorn RISC Machine (a 32-bit RISC microprocessor for low-cost applications), both of which earned Acorn Computers a Queen's Award for Technology. Since moving to the ICL

Chair of Computer Engineering at the University of Manchester in 1990, he has established a research group with interests in asynchronous logic design and power-efficient computing. He is a member of the British Computer Society and a Chartered Engineer.



Jim D. Garside received his BSc in physics from the University of Manchester in 1983 and his MSc and PhD degrees in computer science from the University of Manchester in 1984 and 1987. Since then he has worked as a computer design engineer in parallel processing architectures and as a programmer in the software industry. He was appointed a lecturer in computer science at the University of Manchester in 1991; his main research interests are in the development of the AMULET asynchronous microprocessors.



Nigel C. Paver received his BSc degree in electronics from the University of Manchester Institute of Science and Technology (UMIST), Manchester in 1988 and his MSc and PhD degrees in computer science from the University of Manchester in 1989 and 1995, respectively. From 1990 to 1995, he worked as a researcher in the AMULET group in the Department of Computer Science at the University of Manchester. His main research interests are microprocessor organizations and self-timed system design. In 1995,

he left the University of Manchester and cofounded Cogency Technology Incorporated where he is engaged in the exploitation of low-power self-timed circuits and techniques for commercial applications.



Steve Temple received his BA in computer science from the University of Cambridge in 1980. In 1984, he received his PhD following research into local area networking at the University of Cambridge Computer Laboratory. He was then employed as a research fellow at the Cambridge Computer Laboratory before spending several years working for computer companies in the Cambridge area. In 1992, he took up a post as a research associate in the Amulet group in the Department of Computer Science at the University of Manchester. His current interests include microprocessor system design, asynchronous logic design, and apiculture.