A Micropipelined ARM

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

Department of Computer Science, The University of Manchester,
Oxford Road, Manchester, M13 9PL, England

Abstract

An asynchronous implementation of the ARM microprocessor is described. The design is based on Sutherland's Micropipelines, and allows considerable internal asynchronous concurrency. The rationale for the work is presented, the organisation of the chip described, and the characteristics of the chip described. The design displays unusual properties such as non-deterministic (but bounded) prefetch depth beyond a branch instruction. This work demonstrates the feasibility of building complex asynchronous systems and gives an indication of the costs and benefits of the Micropipeline approach.

Keyword Codes: C.1.1; B.1.1; B.7.1
Keywords: Processor Architectures, Single Data Stream Architectures;
Control Structures and Microprogramming, Control Design Styles;
Integrated Circuits, Types and Design Styles

1. INTRODUCTION

The power dissipation of high-performance CMOS VLSI microprocessors is becoming an increasing problem. Even when battery power and portability are not an issue the 20 to 30 Watt consumption of the latest high-end processors makes it difficult to keep the silicon at an acceptable operating temperature. At lower performance levels the designers of battery powered systems must make difficult trade-offs between the processing demands of, for example, hand-writing recognition software and the minimum acceptable battery life of their products.

The process advances which have caused CMOS to progress from a low power (and low performance) technology to a high power (and high performance) technology show no signs of abating and, if new approaches are not developed, state-of-the-art performance in ten years time will only be delivered at the cost of power dissipations one or two orders of magnitude higher than today's. While there are developments which alleviate the problem, such as the trend towards 3 volt (and later 2 volt) operation, these do not go far enough to remove the possibility that power dissipation might limit the performance that a chip can deliver.

One reason for the high power dissipation is the almost universal design approach which imposes global synchrony across a chip. This is achieved by applying a common clock to all the functional units on a chip and has the undesirable side effect of causing those units to dissipate power whether or not they are doing useful work. There is no doubt that the synchronous approach to logic design has been very effective over the last two decades, and has enabled great advances to be made in the productivity of designers and their design tools and in the performance of machines. However there is now a resurgence of opinion suggesting that it may be time to re-assess the merits of other design approaches. Higher speeds and larger chips are making the abstraction of global synchrony increasingly hard to sustain even on a single chip, with the power dissipation making this approach much less attractive. While it is possible to address the power issue by gating clocks to individual units, this makes the clock skew problem much worse and is therefore not a long term solution.

Asynchronous design approaches are therefore attracting renewed interest. New approaches to asynchronous design are overcoming some of the difficulties which had previously been impediments to cost-effective designs (much of the cost arising in the development rather than the manufacture of the design). Asynchronous designs tend naturally (in CMOS) to use power only when doing useful work and interfacing disciplines produce more modular designs with an inherent potential for component reuse.

For these reasons, the work described here was initiated to investigate whether an asynchronous approach might not offer significant advantages in the design of a RISC microprocessor. The particular asynchronous approach adopted is that described by Sutherland in his 1988 Turing Award Lecture entitled “Micropipelines” [1]. This is one of many possible approaches to asynchronous design, but was felt to offer the right balance of cost, performance and engineering practicality. Using this approach we have produced a full implementation of an existing commercial 32-bit microprocessor architecture (the ARM) with a design cost, die size and performance approaching that of the equivalent clocked product. Earlier work [2] has already demonstrated the feasibility of building an asynchronous microprocessor, but we believe this is the first application of asynchronous techniques to a commercial architecture which embraces all the practical difficulties such as exact exceptions and backwards instruction set compatibility.

2. BACKGROUND

2.1. The ARM

The ARM processor, originally developed at Acorn Computers Ltd in the UK in 1983-85, was the first commercial RISC and was inspired principally by the original Berkeley [3] and Stanford [4] work. ARM uses a load/store architecture and a register oriented instruction set [5]. It is characterised by being very small, simple (the original silicon used 25,000 transistors), low-cost and low-power (the current ARM6 macrocell delivers 100MIPS/Watt) and has been chosen as the processor for the Apple Newton pen-based computer.

The relative simplicity of the ARM makes it an attractive architecture for re-implementation in an asynchronous style. The very low power consumption of the synchronous implementation is perhaps a drawback for our work as it sets a very aggressive baseline to improve upon.

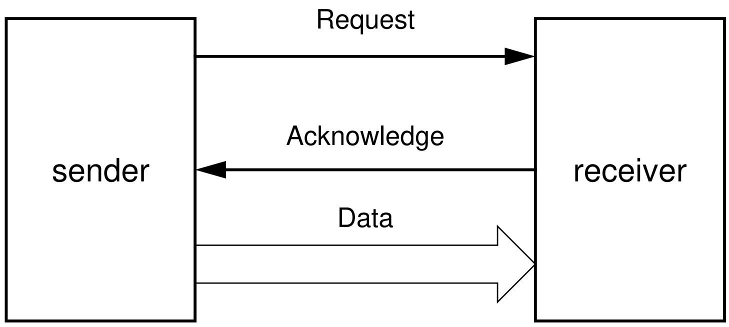


Figure 1. A two-phase bundled data interface

2.2. Micropipelines

The asynchronous communications used for this work employ a ‘two-phase bundled data protocol’. Here information is transferred from the sender to the receiver using a ‘bundle’ of data presented as a binary value on a set of wires, with two control wires (figure 1). When the data are valid a ‘request’ event is issued, and reception is indicated by an event on the ‘acknowledge’ wire. A two-phase event is a transition on a wire; there is no ‘return to zero’ phase.

3. THE ASYNCHRONOUS ARM

3.1. The processor interface

The top level interface is shown in figure 2. The MMU and memory have not yet been implemented, but are shown here to illustrate the environment in which the processor will be employed. The processor produces one output bundle containing the memory address, the ‘write’ data (if any) and control bits. The control bits include a read enable, a write enable, an

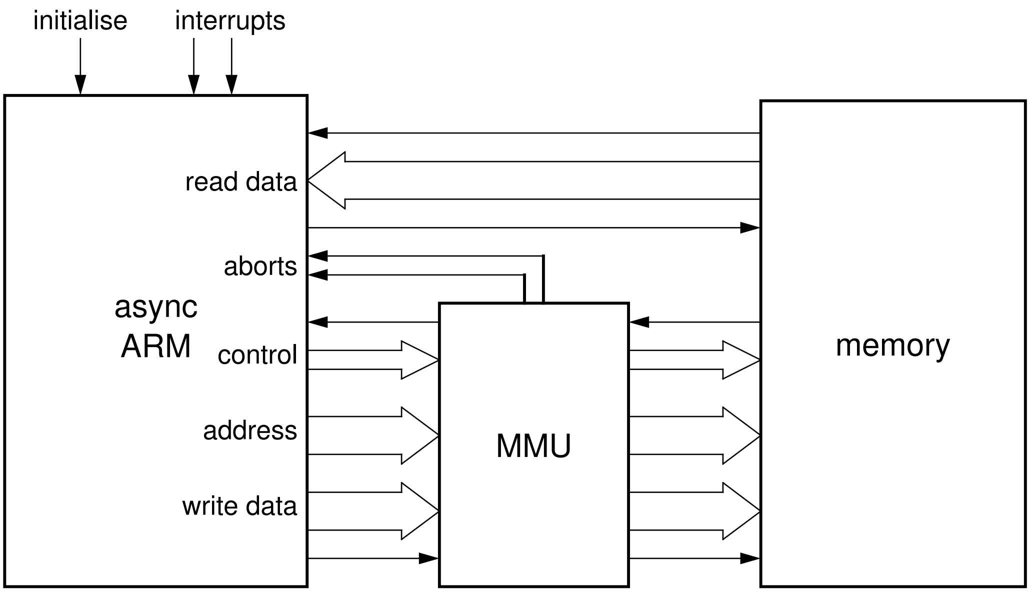


Figure 2. The processor interface

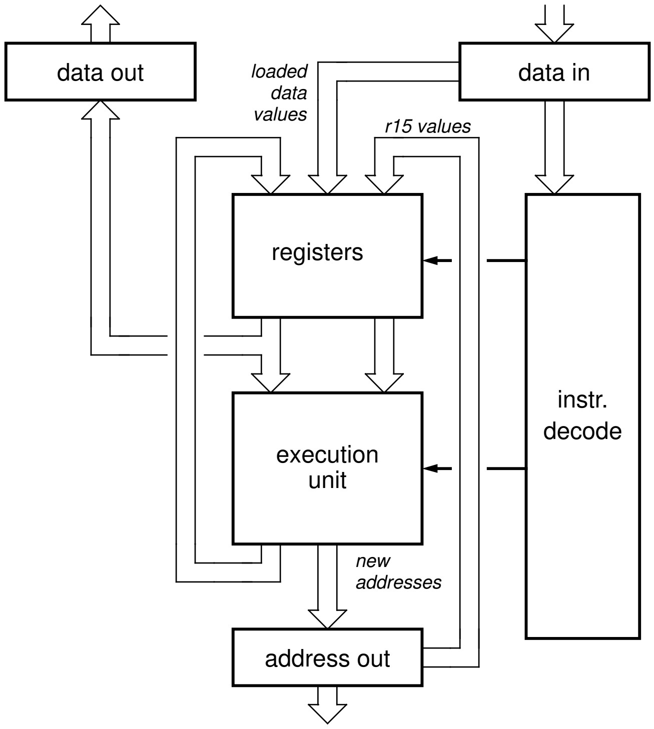


Figure 3. The processor organisation

opcode fetch bit (to indicate whether an instruction or a data item is to be fetched), a privilege mode bit and two bits which hint at sequential address behaviour. A second (input) bundle is used to transfer read data back to the processor.

Note that at this stage no assumption is made about the pipeline depth of the memory subsystem. Indeed, if the memory includes a cache, the effective pipeline depth may depend on whether the cache is hit or not. The memory must, however, return results in the same order as the requests were issued.

As the processor handles memory faults as exact exceptions, it must internally prevent any state change after issuing a request for data from memory until it knows that the request will succeed. Therefore a fault/no fault response from the MMU is time critical on data transfers, and the design employs a dual-rail encoded abort response signal from the MMU. A transition on one wire signifies ‘abort’, causing exception entry, whereas a transition on the other wire signifies ‘no abort’, allowing the processor to proceed. (For instruction pre-fetches a Boolean flag returned with the instruction is sufficient to indicate a successful access; the trap is then initiated when the instruction enters the decode stage.)

Conventional level sensitive interrupt inputs are provided for compatibility with existing peripheral chips, though this is not really a satisfactory model for an asynchronous system as there is no control on the timing of the release of an interrupt line. A system initialisation input completes the top level interface.

3.2. Processor organisation

The internal organisation of the processor is shown in figure 3. The processor state is held in the register bank, which has two read ports for operand access. The operands are processed by the execution unit and the result written back through a write port. A second write port allows data to be written into the register bank from memory. The third input port is used to allow the PC (which resides in the address interface unit) to be available as register 15 as required by the ARM instruction set

The address interface unit produces sequential addresses autonomously and only requires input from the execution unit when the address sequence is changed; this may be a temporary change for a data access or a permanent change for a branch.

Write data are copied from one of the register bank read operand buses and then synchronised with the appropriate address for issue to memory. Values returned from memory are split into data and instruction streams and processed accordingly.

3.3. The address interface

The address interface includes a PC word-increment loop (see figure 4). The ARM instruction set specifies that (in most cases) r15 has the value PC+8 (exposing the depth of the synchronous pipeline implementation in the original ARM), so for code compatibility that behaviour is copied here. The first address, 0, is produced in the memory address register. After passing through the incrementer, the first value offered to the PC pipeline (figure 5) is therefore 4. However the first value is ‘thrown away’, so the first value actually copied into the PC pipeline is 8. This skew between the PC pipeline and the fetched instruction stream persists and maintains the alignment of all future instructions with PC+8. The skew causes incorrect pairing of the instructions immediately following a branch, but as these are not executed this is of no consequence. Correct pairing is re-established by the time the new instruction stream begins execution.

The incrementer loop operates autonomously so a new address from the ALU arrives asynchronously. An arbiter is required to ensure safe interruption of the incrementer loop, and the precise point where the loop is interrupted is therefore non-deterministic, resulting in the non-deterministic depth of prefetching beyond a branch. The interruption may be transient (for a data access) or permanent (for a branch). In the former case the PC value is preserved in the

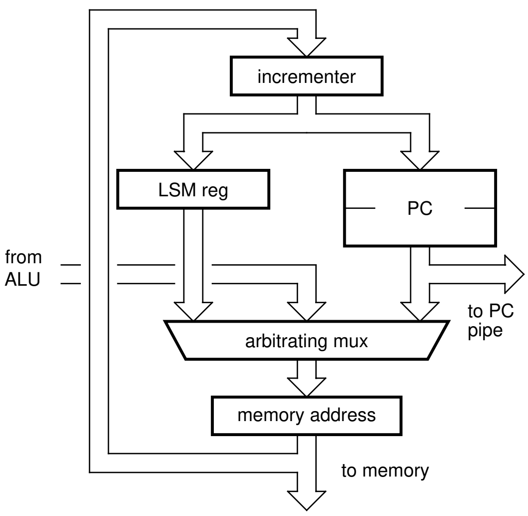


Figure 4. The address interface

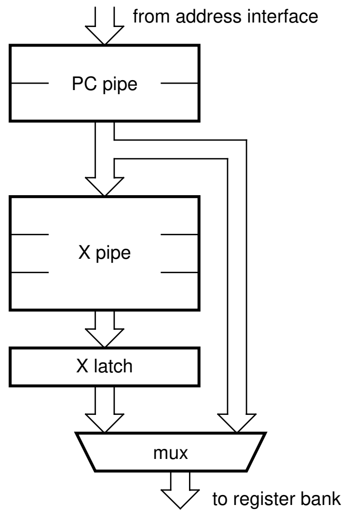


Figure 5. The PC pipelines

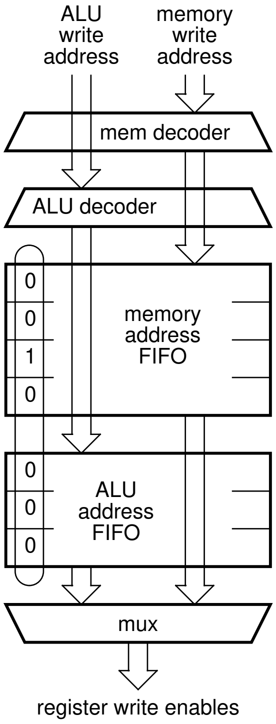


Figure 6. Lock FIFO

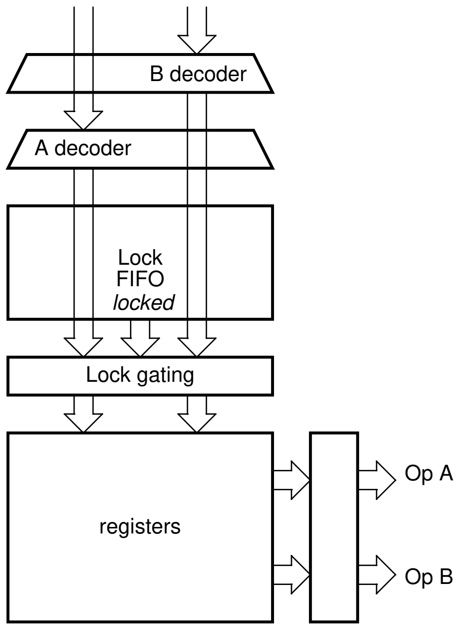


Figure 7. The register read logic

PC register; in the latter case the old value is thrown away and the new one takes over the loop.

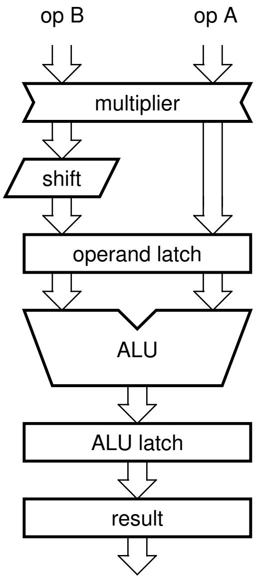
The ARM instruction set includes multiple register loads and stores of arbitrary subsets of the register set. Here the memory targets are sequential addresses, so the incrementer loop is temporarily usurped to produce these addresses. In this case the PC is held in the PC register, and the looping value passes through the load/store multiple (LSM) register.

The PC pipeline buffers PC+8 values for pairing with instructions as they go into execution. When the instruction is a data transfer type, the PC+8 value is further copied into the exception pipe (X pipe in figure 5) in case a memory fault is detected. Each MMU response causes the entry at the bottom of the exception pipe either to be discarded (no abort) or copied into the exception latch (if an abort was indicated). The presence of a value in the exception latch causes a data abort exception to be raised and the exception entry process uses the X latch value (accessed as r15) to form the return address.

Note that there is a crucial restriction on the length of the PC pipeline. If it is not at least three stages shorter than the instruction prefetch pipeline, the prefetch unit can congest the memory pipeline, potentially preventing a data access from completing and hence causing deadlock. The PC pipeline is therefore restricted in length to govern the behaviour of the prefetch unit.

3.4. The register bank

The design issues relating to the register bank have been described in detail elsewhere [6,7], but in summary the unit must handle multiple pending write operations, register locking to prevent access to stale register values and the asynchronous interaction between read and write operations. All these issues are resolved in a single regular structure, the register lock FIFO (figure 6; note that in our design there are two FIFOs to allow internal results to overtake data



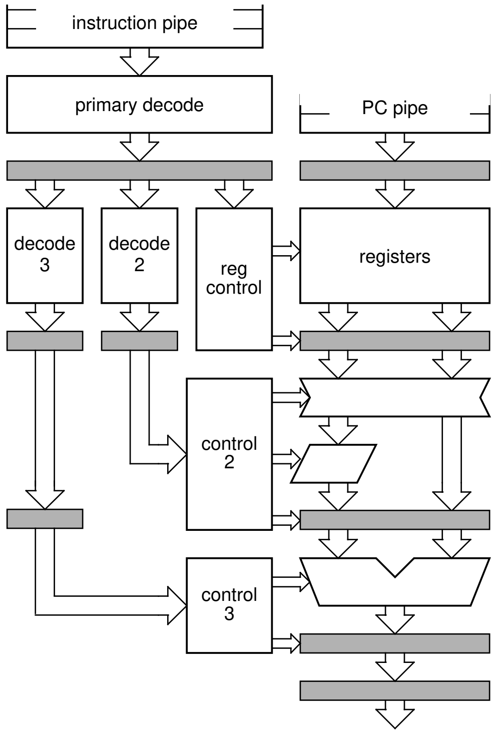


Figure 8. The execution pipe

Figure 9. Control structure

from - potentially much slower - memory). Here write addresses are queued in a fully decoded form, so each stage of the FIFO contains at most one ‘1’. A column of the FIFO contains all the information about pending writes to a particular register and a logical OR of the column provides the lock control. The OR function is not normally permissible across such an asynchronous structure but is possible here because, as data propagates through the FIFO, a ‘1’ is copied to the new latch before it is removed from the old one and the lock output is glitch-free.

The lock information is used to delay the decoded read word lines (figure 7) until the correct value is available. Multiple locks on a single register are handled correctly and no arbiter is required to manage the asynchronous interaction between reads and writes; these proceed independently when there is no interdependency and the lock mechanism synchronises them when a dependency occurs.

3.5. The execution unit

The functional units in the execution pipeline are shown in figure 8. The register operands first pass through a multiplier, which either passes them on immediately or replaces them by partial product and partial carry outputs from a carry-save multiplication unit. A barrel-shifter then modifies one of the operands before both are placed into a pipeline latch. The operands are then combined in the ALU which has a data-dependent delay [8] and a latch to allow a dynamic structure to operate with pseudo-static external behaviour. A result latch passes the output to its next destination (either a register or the address unit).

The sequential positioning of the function units is perhaps not ideal for performance. However the ARM instruction set supports shift and ALU operations in a single instruction, forcing the barrel shifter to be in series with one of the ALU inputs. Multiplications also use the

ALU for the carry propagate addition which is required to combine the partial product and partial carry. When no multiplication is required, the cost of passing the operands through the multiplier is equivalent to passing them through the multiplexer which would be needed to bypass the multiplier, so overall this arrangement would appear to be the best compromise for the target instruction set.

The instruction decode and execute pipe control logic are illustrated in figure 9. Here the pipeline latches are shaded to highlight the structure. Prefetched instructions are queued before being passed to the primary decode logic which produces multiple pipeline bubbles for the more complex instructions, sends appropriate read and write addresses to the register bank for each bubble and passes information onto the secondary and tertiary decode logic. Note that although the shaded pipeline latches are aligned to emphasise the matching of the pipeline depths of the parallel structures, synchronisation only occurs when the pipelines interact, for instance where ‘control 2’ connects into the multiplier and where ‘control 3’ governs the ALU.

4. THE SILICON

The design as described above has been implemented in silicon using tools from Compass Design Automation. Sutherland’s event control blocks (plus a few additional basic asynchronous elements) were added to a standard cell library supplied by Advanced RISC Machines Limited and the layout produced using hand-crafted full custom design for the datapath and compiled standard cells for the control areas. Two major control blocks were implemented as synthesized PLA layout using a modified version of a self-timed PLA generator from ARM Ltd. The resulting layout is shown in figure 10.

The prototype is in fabrication with GEC Plessey Semiconductors and samples are expected in the third quarter of 1993. The principle characteristics of the design are shown in table 1, where the corresponding characteristics of the ARM6 under similar simulated conditions are also noted for comparison.

Table 1
Characteristics of the micropipelined ARM compared with ARM6

	Micropipelined ARM	ARM6
Process	1μm DLM CMOS	1μm DLM CMOS
Cell area	5.5mm x 4.1mm	4.1mm x 2.7mm
No. of transistors	58,374	33,494
Performance	9Kdhrystones	14Kdhrystones @ 10MHz
Dissipation	83mW	75mW @ 10MHz

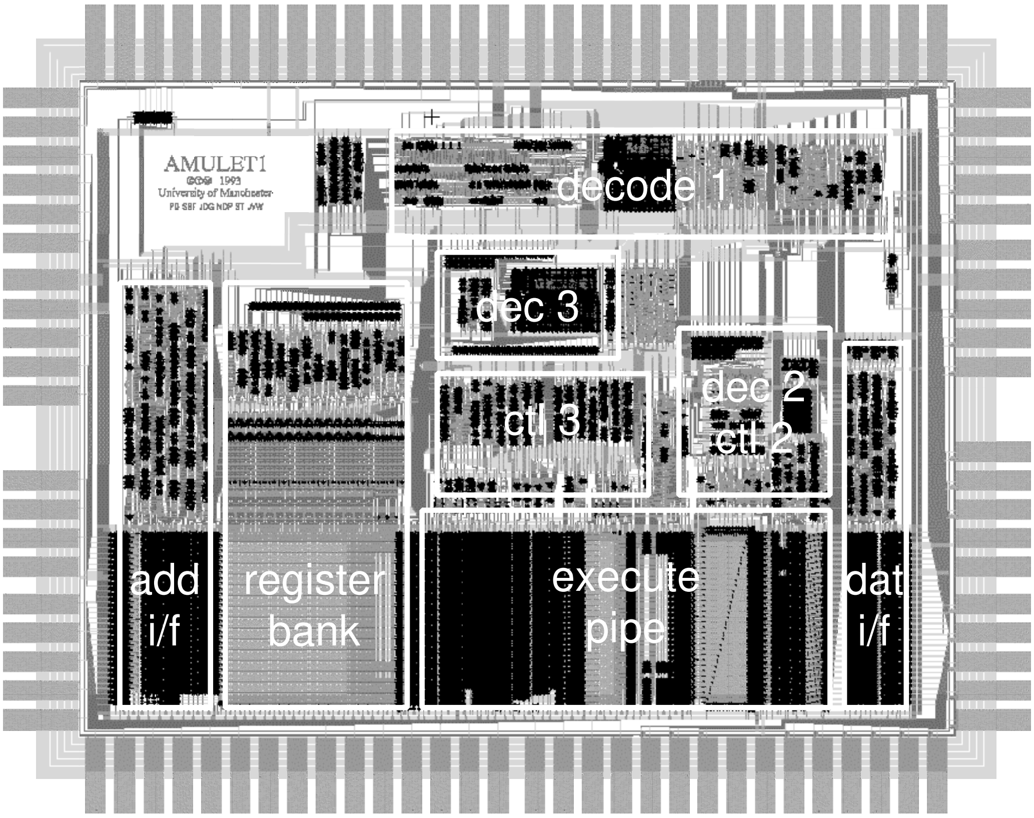


Figure 10. The asynchronous ARM chip

5. CONCLUSIONS

We have demonstrated the feasibility of designing a full functionality commercial RISC architecture in asynchronous logic. Micropipelines would appear to offer a good framework for this design task, as the modularity of the design allows a conventional engineering approach to be applied with most of the asynchronous complexities being hidden within pre-packaged ‘black boxes’.

Although our chip is rather larger than the clocked ARM6 cell (which has similar functionality), most of the difference can be accounted to the greater pipeline depth of our design. A clocked design with a similar depth of pipeline would have an area and transistor count perhaps only 10 or 20% smaller than our design.

At this stage our design does not outperform its clocked counterpart, but its performance is within a factor of two in all areas. As this is a first attempt at producing a design of this complexity in a micropipelined style we are not too discouraged that we have failed to demonstrate an immediate advantage over the ARM6 (which is the fourth generation synchronous ARM and is the leader of its class in low-power processing). The engineering of the micropipelined ARM is very conservative and there are several areas where we expect to recover more than the factor two disadvantage in future revisions of the design.

ACKNOWLEDGEMENTS

The work described in this paper was carried out as part of ESPRIT project 5386, OMI-MAP (the Open Microprocessor systems Initiative - Microprocessor Architecture Project), and 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 consortium, and particularly the prime contractor, INMOS Limited, is also acknowledged.

REFERENCES

1. I. E. Sutherland, “Micropipelines”, Communications of the ACM Vol. 32 No. 6 (1989) pp 720-738.

2. 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: Proceedings of the Decennial Caltech Conference on VLSI, ed. C. L. Seitz, MIT Press (1989) pp 351-373.

3. M. G. H. Katevenis, “Reduced Instruction Set Computer Architectures for VLSI”, MIT Press, Cambridge, MA (1985).

4. J. L. Hennessy, N. Jouppi, F. Baskett and J. Gill, “MIPS: A VLSI Processor Architecture”, Proceedings of the CMU Conference on VLSI Systems and Computations, Computer Science Press, Rockville, MD (1981) pp 337-346.

5. S. B. Furber, “VLSI RISC Architecture and Organization”, Marcel Dekker, New York (1989).

6. N. C. Paver, “Condition Detection in Asynchronous Pipelines”, UK Patent No. 9114513 (1991).

7. N. C. Paver, P. Day, S. B. Furber, J. D. Garside and J. V. Woods, “Register Locking in an Asychronous Microprocessor”, Proceedings of ICCD 92 (1992) pp 351-355.

8. J. D. Garside, “A CMOS Implementation of an Asynchronous ALU”, Proceedings of the IFIP Working Conference on Asynchronous Design Methodologies, Manchester, England (1993).