# Parallel Structures for Asynchronous Microprocessors

Philip B. Endecott

Department of Computer Science, University of Manchester, U.K.

pbe@cs.man.ac.uk

## Abstract

***This paper considers the implementation of pipelining and superscalar parallelism in asynchronous processors. The performance of simple pipelines and superscalar structures is improved by asynchronous implementation. The organisation of general purpose processors is more complex: they include forwarding paths joining non-adjacent pipeline stages. Unfortunately the additional synchronisation required by forwarding paths is detrimental to performance, yet not having forwarding is also detrimental. A solution called conditional forwarding is proposed but it is found that conventional instruction sets do not permit conditional forwarding. To allow this technique to be used an alternative programming model called explicit forwarding is introduced. In a processor with explicit forwarding the destination to which the result of an instruction must be sent is indicated explicitly by the instruction; in contrast in a conventional instruction set the routing of the result of an instruction is deduced from the register specifiers of adjacent instructions. The paper concludes by describing an experimental processor called SCALP (Superscalar Asynchronous Low-Power Processor) which uses explicit forwarding.***

## 1. Introduction

The work described in this paper was motivated by the following observation: many of the fundamental architectural features of conventional instruction sets are based solidly on the expectation that their implementation will be synchronous. Despite this, most recent asynchronous processor designs choose to implement a conventional instruction set: the AMULET processor [10] implements the ARM instruction set, the Counterflow Pipeline Processor [14] implements the SPARC instruction set, and others implement similar "RISC" instruction sets.

This paper sets out to re-evaluate some architectural structures with a view to asynchronous implementation. Many areas deserve attention, but the specific focus here is on two familiar forms of concurrency, pipelining and superscalar parallelism.

## 2. Simple Pipelining

Figure 1 shows the familiar structure of a synchronous pipeline with latches (L) and combinational logic blocks (CL). All latches are controlled by a single global clock signal and so operate simultaneously.

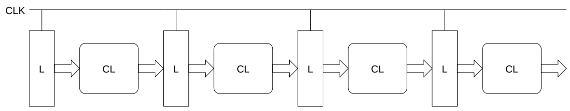


**Figure 1: A Simple Synchronous Pipeline**

This form of pipeline is readily implemented asynchronously as shown in figures 2 and 3. The latches (L) and combinational logic blocks (CL) are the same as in the synchronous pipeline but the timing is controlled quite differently. Each latch has an associated latch control circuit (LC). The latch control circuit opens and closes the latch in response to request (Req) signals from the previous stage and acknowledge (Ack) signals from the following stage.

The request signal from the latch control circuit must be delayed by an amount greater than the corresponding data delay in the combinational logic. This may be done either using a matched path delay element (D) as shown in figure 2 or by using some form of completion detection as shown in figure 3.

The asynchronous pipeline's performance can potentially exceed that of the synchronous pipeline. This is a result of at least three effects:

- When delays are data dependent, as in the case of an adder for example, the asynchronous pipeline's

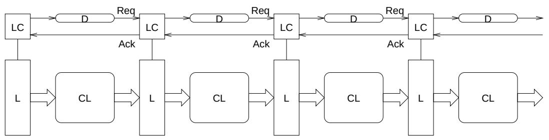


**Figure 2: A Simple Asynchronous Pipeline with Matched Delays**

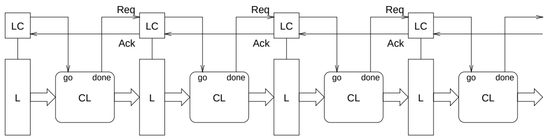


**Figure 3: A Simple Asynchronous Pipeline with Completion Detection**

speed can match the actual delay in that case. A synchronous pipeline must always allow for the worst case delay.

- When the delays in the different pipeline stages are unequal the pipeline's latency is limited by the sum of the individual stage delays. A synchronous pipeline is limited by the delay in the slowest stage multiplied by the number of stages.
- The clock speed in a synchronous system must be chosen to allow for the worst case supply voltage, temperature, process variation, and clock skew conditions. In an asynchronous system the performance obtained at a particular time will match the conditions in that system at that time.

# 3. Pipelines in Processors

The simple pipeline described above is common in applications such as digital signal processing. On the other hand in general purpose processors pipelines are made more complex by the presence of "forwarding" paths. Forwarding paths allow results to be used by closely following instructions without having to pass through the entire pipeline. The synchronous implementation of forwarding is shown in figure 4.

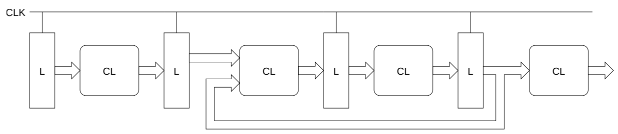


**Figure 4: A Synchronous Pipeline with Forwarding**

In figure 4, stage 2 uses as inputs the results of stages 1 and 3. Because all latches are synchronised the data from stages 1 and 3 are available simultaneously at the input to stage 2.

In an asynchronous pipeline this simple scheme cannot be used because the results of stages 1 and 3 are not generally available at the same time. Additional timing logic is required as shown in figure 5[1]. A Muller C element[2] is used to combine the requests from stages 1 and 3. The acknowledge from stage 2 is sent to both stage 1 and stage 3. Stage 3 must combine the acknowledge signals from both stages 4 and 2 using a second Muller C element.

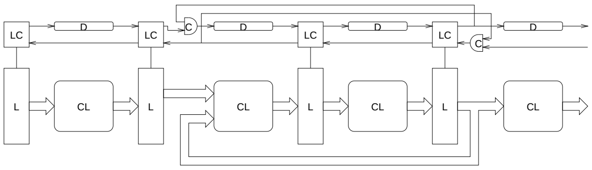


**Figure 5: An Asynchronous Pipeline with Forwarding**

The additional timing constraints in this circuit make it adopt a form of locally synchronous behaviour, constraining the progress of instructions along the pipeline. This leads to a reduction in performance compared with the simple asynchronous pipeline.

# 4. Forwarding in Other Asynchronous Processors

The problems with forwarding lead to a number of interesting solutions in existing asynchronous processors. AMULET1 [10], NSR [3] [12], Fred [13] and ECSTAC [9] all detect data dependencies during instruction decoding and stall the pipeline until the required result has been written back to the register bank. This leads to significant performance degradation as data dependencies between closely adjacent instructions are common.

AMULET2 provides a single forwarding path around the ALU. Because this path is around only one pipeline stage the timing issues are somewhat simplified. AMULET2 also provides more complex conditional synchronisation on the result of the last load instruction.

1. Some additional logic is required to ensure correct initialisation.

2. A Muller C element behaves as follows: when both inputs are high, the output becomes high. When both inputs are low, the output becomes low. When the inputs differ the output retains its previous level.

The Counterflow Pipeline Processor [14] uses a second pipeline flowing in the opposite direction to the main instruction pipeline to carry results from earlier instructions to be used as operands by later instructions. This solution is very general but relies on large numbers of register specifier comparators to detect values to be used as operands and complex synchronisation between the two pipelines.

# 5. Conditional Forwarding

Operands provided via forwarding paths are not always used by the stages to which they are sent. Typically the receiving stage uses a multiplexor to select between the forwarded value and a value received from the previous stage. This multiplexor is controlled by a bit generated during instruction decode that indicates whether forwarding should be used.

In principle this conditional nature of forwarding could be used to improve the performance of asynchronous forwarding pipelines. When forwarding is not to be used no additional synchronisation is carried out and performance is similar to that of a simple asynchronous pipeline. When forwarding is required temporary additional synchronisation can be introduced to allow the forwarded data to be transferred.

Such a scheme could be implemented as shown in figure 6: the instruction decode generates two bits for each instruction, one (USE_FWD) indicating that the instruction must use a forwarded value produced by a preceding instruction and another (GEN_FWD) indicating that the instruction must send its result to be used by a following instruction. At the pipeline stage where the forwarded data is used a “conditional pipeline merge” (CPM) element synchronises with forwarded data only when the USE_FWD bit is asserted. At the pipeline stage where the forwarded value is created a “conditional pipeline fork” (CPF) element sends the forwarded value only when the GEN_FWD bit is asserted.

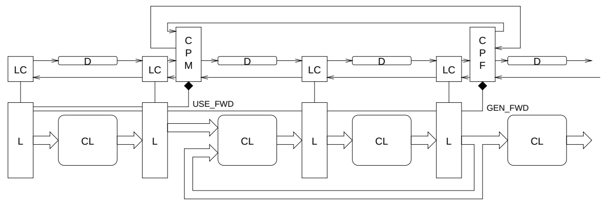


**Figure 6: An Asynchronous Pipeline with Conditional Forwarding**

The flaw with this proposal is that with conventional instruction sets it is impossible for the instruction decoder to generate the GEN_FWD bit. If a record of previously issued instructions is maintained the USE_FWD bit can be computed by comparing the operands of the current instruction with the results of the previous instructions - this technique is used by synchronous forwarding mechanisms. To compute the GEN_FWD bit it would be necessary to know the operand registers of future instructions before they are issued.

# 6. Explicit Forwarding

One solution to this problem is to change the nature of the instruction set so that the need to forward a value can be detected by the instruction decoder. This may be done by giving more information about the way in which a result will be used in the instruction that produces that result. This technique is referred to here as “explicit forwarding”.

A number of simple instruction set extensions have been considered to permit explicit forwarding. In the simplest case an additional bit is associated with each destination register specifier indicating whether this value should be forwarded. The pipeline structure needed is similar to that shown in figure 6, but the GEN_FWD bit is read directly from the instruction.

When branch instructions occur between instructions producing results and those consuming them, with this simple scheme the GEN_FWD bit can be thought of as a “forwarding prediction” bit. If GEN_FWD is asserted but a subsequent branch means that the forwarded value is not actually needed, no harm is done except that some additional unnecessary synchronisation has been performed. On the other hand if a branch means that a value that could have been forwarded has not been, the value can be read from the register bank with a small performance penalty.

More complex proposals take the idea further. A significant proportion of the results computed by a processor are used only by instructions that follow closely after. This property has been measured for SPARC code in [5] and is summarised in tables 1 and 2. When the result of an instruction is used only by means of forwarding there is no need to write that result into the register bank at all, and hence no need for a register specifier. This forms the basis for another form of explicit forwarding: a special register identifier is set aside for explicit forwarding. When written to the value is placed into the forwarding path and when read from the value is read from the forwarding path.

| Number of uses | Proportion of results |
|---|---|
| 0 | 17 % |
| 1 | 64 % |
| 2 | 11 % |
| 3 | 4 % |
| 4 | 2 % |
| >4 | 2 % |

**Table 1: Number of times each instruction result is subsequently used**

| Interval / instructions | Proportion of results |
|---|---|
| 1 | 45 % |
| 2 | 10 % |
| 3 | 8 % |
| 4 | 3 % |
| >4 | 34 % |

**Table 2: Interval before the last use of the result of an instruction**

The limitation of this scheme is that each value may be read only once. This can be overcome by allowing reads from the forwarding paths to optionally leave a copy of the value they have read in place for a subsequent instruction to read.

To allow greater freedom in the use of the mechanism, fifo queue stages may be added to the forwarding path to provide storage for several values. It is also possible to incorporate more than one such path that can be accessed using different register identifiers.

When extended in this way it seems possible that the storage in the forwarding paths could become the main form of short-term storage, with the register bank simplified to a single ported memory accessed only by a subset of the instructions and used for medium term storage. The organisation of this sort of processor is shown in figure 7.

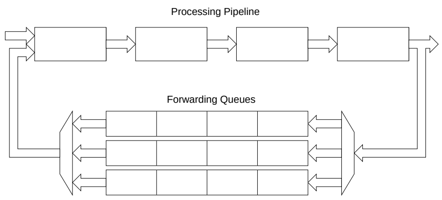


**Figure 7: Processor Organisation with Explicit Forwarding**

One interesting property of this type of processor is that incorrect code can cause the processor to deadlock by reading from a queue which is empty or by writing to a queue which is full. It may be desirable to use a watchdog timer or similar to reset the processor after some significant period of inactivity indicating internal deadlock. This may seem dramatic but it is the author's opinion that faulty code is easier to debug when it stops in this way than when it continues to execute beyond the point of error without indicating a problem.

# 7. Superscalar Parallelism

Pipelining is an attractive form of parallelism because introducing pipelining to a datapath has a relatively low hardware cost. Unfortunately the concurrency that can be obtained using pipelining is limited and at some point other more expensive forms of concurrency must be considered. Superscalar parallelism is one example.

A superscalar processor has several functional units and provides concurrency by having instructions in some stage of execution in more than one functional unit at a time. The amount of concurrency available is often limited by the processor's ability to find independent instructions that may be executed in parallel. To extend this limit complex techniques including out of order issue and register renaming must be employed. This results in significant complexity in the processor's instruction issuer.

In one way superscalar parallelism is more attractive to asynchronous implementation than pipelining. In an asynchronous pipeline the throughput is limited by the speed of the slowest stage; however when the stages are arranged in parallel the typical throughput is determined by the typical speed. All synchronous systems are limited by the slowest stage's worst-case speed.

Superscalar processors must employ forwarding both within and between the functional units. As with pipelining the presence of forwarding can make the efficient asynchronous implementation of conventional instruction sets difficult. The idea of explicit forwarding can be used to overcome this problem.

In a superscalar processor explicit forwarding can have other benefits. A significant proportion of the complexity of the instruction decoding and issuing logic is concerned with comparing register specifiers to detect dependencies between instructions, activating forwarding paths, and delaying execution until operands are available. With explicit forwarding this complexity can be significantly reduced.

Figure 8 shows the structure of an asynchronous superscalar processor with explicit forwarding. There is one explicit forwarding queue associated with each operand input to each functional unit. All operands are provided by explicit forwarding; there is no global register bank. Functional units are activated when an instruction and the appropriate operands have arrived at its input. Information about where instruction results should be sent is indicated explicitly in the instructions. The role of the instruction issuer is consequently significantly simplified: it merely distributes instructions from the input stream to the appropriate functional units' instruction queues.

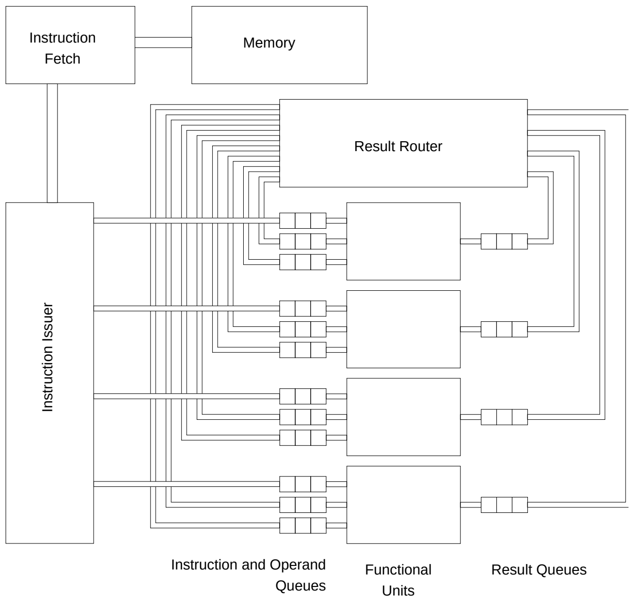


**Figure 8: A Superscalar Processor with Explicit Forwarding**

This architecture has significant similarities to the idea of transport triggered architectures (TTAs) [4] [11]. TTAs are synchronous processors whose instructions specify move operations from functional unit outputs to functional unit inputs. Functional units are activated when operands are moved to their "trigger" inputs. These authors have developed a "C" compiler for their processor and report encouraging results.

# 8. SCALP

Many of the ideas presented here are embodied in an asynchronous processor design undertaken by the author called SCALP. SCALP stands for "Superscalar Asynchronous Low-Power Processor". The principal objective of the SCALP design is high power efficiency. SCALP derives high power efficiency from four main architectural ideas:

- Asynchronous implementation. Asynchronous logic is believed to be more power efficient than synchronous logic because asynchronous circuits can waste fewer signal transitions than equivalent synchronous circuits. Furthermore asynchronous logic is better suited to operation in variable demand systems and in systems with dynamic supply voltage scaling. A good example of the power saving that can be obtained through the use of asynchronous logic is given in [1] and [7].
- Parallelism. In CMOS, power consumption is proportional to the square of the supply voltage whereas performance is directly proportional to the supply voltage. If performance is kept constant by increasing the parallelism in the processor as the supply voltage is reduced, the power consumption will decrease as parallelism is increased. [8]
- High code density. A significant proportion of the power consumption in a processor is proportional to the instruction fetch bandwidth, and by increasing the code density the amount of instruction fetch traffic per unit of computation performed is reduced. [2] [5]
- Variable width datapath operations. Many of the values operated on by a processor are small numbers or characters, yet in conventional systems the whole of the 32 bit datapath is always activated. SCALP provides narrower operations that activate only as much of the datapath as is necessary.

Explicit forwarding aids the first three of these requirements. Preceding sections have explained how explicit forwarding helps asynchronous implementation by permitting forwarding and helps superscalar implementation by simplifying the superscalar instruction issuer. Code density is improved because the explicit forwarding information requires fewer bits than the register specifiers of conventional instruction sets.

SCALP's potential for power saving is still unknown. The current implementation's performance is limited by two factors: as with all superscalar processors the branch latency is significant yet SCALP does not have branch prediction. This problem is easy enough to remedy. More seriously SCALP allows each result to be used by only one following instruction; to use a result more than once an explicit duplicate operation is required.

SCALP's instruction issuer is simpler than it would have to be for a conventional instruction set, yet it is still complex. This complexity derives from the need to decode and issue several instructions in parallel and from SCALP's variable length instruction format.

A number of small programs have been written in SCALP assembly language to evaluate the potential of the explicit forwarding instruction set. In straight-line code the model works well, but the presence of branches can cause problems. The results in tables 1 and 2 show that it is very common for it to be possible to discard a result quite soon after it has been computed. Unfortunately these values do not allow for the possibility of an intervening branch - it may be the case that a value is needed later in the program only if a particular branch is taken.

The SCALP processor will be discussed further in [6].

# 9. Conclusion

Using asynchronous logic for general purpose processor design has several potential benefits, yet conventional instruction sets do not allow these benefits to be fully exploited. By considering execution models other than the conventional global register bank model architectures that are better suited to asynchronous implementation can be developed. One such alternative is the idea of explicit forwarding, implemented in the SCALP processor.

The SCALP implementation has shown that by using explicit forwarding an efficient asynchronous implementation is possible. On the other hand the usefulness of the explicit forwarding model for programming remains unproven; further work such as the development of a compiler is required to understand its potential.

# 10. References


**[1]** K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Ronken, F. Schalij, R. van de Weil, "A Single-Rail Re-implementation of a DCC Error Detector Using a Generic Standard-Cell Library", Proceedings of the Second Working Conference on Asynchronous Design Methodologies, London, U.K., 1995, pps 72-79.

**[2]** J. Bunda, W. C. Athas, D. Fussell, "Evaluating Power Implications of CMOS Microprocessor Design Decisions", Proceedings of the 1994 International Workshop on Low Power Design, Napa, California, pps 147-152.

**[3]** E. Brunvand, "The NSR Processor", Proceeding of the 26th Annual Hawaii International Conference on System Sciences, pps 428-435, Maui, Hawaii, 1993.

**[4]** H. Corporaal, "Evaluating Transport Triggered Architectures for scalar applications", Microprocessing and Microprogramming, No 38, pps 45-52, 1993. `http://einstein.et.tudelft.nl/~heco/documents/documents.html`

**[5]** P. B. Endecott, "Processor Architectures for Power Efficiency and Asynchronous Implementation", M.Sc. thesis, University of Manchester, U.K., 1993. `http://www.cs.man.ac.uk/amulet/publications/thesis/endecott93_msc.html`

**[6]** P. B. Endecott, "SCALP: A Superscalar Asynchronous Low-Power Processor", Ph.D. thesis, University of Manchester, U.K., 1995, to be submitted. `http://www.cs.man.ac.uk/~endecotp/research/`

**[7]** J. Kessels, "VLSI Programming of a Low-Power Asynchronous Reed-Solomon Decoder for the DCC Player", Proceedings of the Second Working Conference on Asynchronous Design Methodologies, London, U.K., 1995, pps 44-52.

**[8]** R. F. Lyon, "Cost Power and Parallelism in Speech Signal Processing", Proceedings of the IEEE 1993 Custom Integrated Circuits Conference, San Diego, California.

**[9]** S. V. Morton, S. S. Appleton, M. J. Liebelt, "ECSTAC: A Fast Asynchronous Microprocessor", Proceedings of the Second Working Conference on Asynchronous Design Methodologies, London, U.K., 1995, pps 180-189.

**[10]** N. C. Paver, "The Design and Implementation of an Asynchronous Microprocessor", Ph.D. Thesis, Department of Computer Science, University of Manchester, U.K., 1994. `http://www.cs.man.ac.uk/amulet/publications/thesis/paver94_phd.html`

**[11]** D. Pountain, "Transport-Triggered Architectures", Byte, February 1995, pps 151-152.

**[12]** W. F. Richardson, E. Brunvand, "The NSR Processor Prototype", Technical Report UUCS-92-029, University of Utah, 1992. `ftp://ftp.cs.utah.edu/techreports/1995/UUCS-92-029.ps.Z`

**[13]** W. F. Richardson, E. Brunvand, "Fred: An Architecture for a Self-Timed Decoupled Computer", Technical Report UUCS-95-008, University of Utah, 1995. `ftp://ftp.cs.utah.edu/techreports/1995/UUCS-95-008.ps.Z`

**[14]** R. F. Sproull, I. E. Sutherland, C. E. Molnar, "Counterflow Pipeline Processor Architecture", IEEE Design and Test of Computers, Volume 11, No 3, 1994. Also as Sun Microsystems Laboratories Inc. Technical Report SMLI TR-94-25.