INSTRUCTION-LEVEL PARALLELISM IN ASYNCHRONOUS PROCESSOR ARCHITECTURES

D. K. ARVIND and V. E. F. REBELLO
Department of Computer Science, The University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ, Scotland, U. K.
{dka,vefr}@dcs.ed.ac.uk

ABSTRACT. The Micronet-based Asynchronous Processor (MAP) is a family of processor architectures based on the micronet model of asynchronous control. Micronets distribute the control amongst the functional units which enables the exploitation of fine-grained concurrency, both between and within program instructions. This paper introduces the micronet model and evaluates the performance of micronet-based datapaths using behavioural simulations.

KEYWORDS. Instruction-level parallelism (ILP), asynchronous processor architecture, self-timed design.

1 INTRODUCTION

Centralised controls have been traditionally used to correctly sequence information within processor architectures. However, the ability to sustain this design style is under pressure from a number of directions [6]. This paper examines the effect of relaxing this strict synchrony on the design and performance of processor architectures. The reasons are the following. The the clock frequency of a synchronous processor is determined *a priori* by the speed of its slowest component (which takes into account worst-case timings for execution and propagation for pessimistic operating conditions). In contrast, the performance of an asynchronous processor is determined by actual operational timing characteristics of individual components (effectively the average delays), and overheads due to asynchronous controls. Secondly, an important consequence of asynchronous controls is the ability to exploit fine-grained Instruction-level Parallelism (ILP), and this is explored in greater detail in the rest of this paper.

ILP can be achieved either by issuing several independent instructions per cycle as in superscalar or VLIW architectures, or by issuing an instruction every cycle as in a pipelined

Published in the Proceedings of the 3rd International Workshop on Algorithms and Parallel VLSI Architectures, pp 203-215, Leuven, Belgium, August 1994.

architecture where the cycle time is shorter than the critical path of the individual operations [5]. This work concentrates on the design and evaluation of asynchronous pipelines for exploiting ILP, as a number of control issues resulting from data and structural dependencies between instructions have to be resolved efficiently.

A few asynchronous processors have recently been proposed [3, 8, 9]. These designs are based on a single micropipeline datapath [10]. One disadvantage of viewing a datapath as a linear sequence of stages is that, in general, only one of the functional units will be active in any cycle. Pipelining the functional units themselves is expensive both in terms of additional hardware and the resulting increase in latency.

We introduce an alternative model for an asynchronous datapath called a *micronet*. This is a network of elastic pipelines in which individual stages of the pipelines have concurrent operations, and stages of different pipelines can communicate with each other asynchronously. This allows for a greater degree of fine-grained concurrency to be exploited, which would otherwise be quite expensive to achieve in an equivalent synchronous datapath.

2 MICRONETS AND ASYNCHRONOUS ARCHITECTURES

Micronets are a generalisation of Sutherland's micropipeline [10], which dynamically control which stages communicate with each other. Thus micronets can be viewed not just as a pipeline but rather as a network of communicating stages. The operations of each of the stages are further exposed in the form of *microagents* which operate concurrently and communicate asynchronously with microagents in other stages. Each program instruction spends time only in the relevant stages and for just as long as is necessary. This is in contrast with synchronous datapaths in which the centralised control forces each instruction to go through all the stages, regardless of the need to do so (in effect a single pipeline). Furthermore, the microagents within a stage might operate on different program instructions concurrently.

Micronets are controlled at two levels: the data transfer between microagents is controlled locally, whereas the type of operation carried out by a microagent (called a microoperation) and the destination of its result is controlled by the sequencer or by other microagents. Microagents can communicate either across dedicated lines or via shared buses where arbitration is provided either by the sequencer or some other decentralised mechanism such as a token ring.

Data dependencies in synchronous pipelines are resolved by using either hardware or software interlocks [4], which increases the complexity of the controls. Micronets use their handshaking mechanisms together with simple register locking to achieve the same effect, but with trivial hardware overheads. In synchronous designs the structural hazards are normally avoided in hardware by using a scoreboarding mechanism. In micronets this is provided by existing handshaking protocols. Out-of-order instruction completion can be supported in synchronous designs, but at a non-trivial cost. Micronets are able to relax the strict ordering of instruction completions and thereby further exploit ILP. The result is to effectively increase the utilisation of the functional units by reducing their idle times or stalls. Better program performances can be achieved by exploiting both ILP and actual instruction execution times.

2.1 Asynchronous Architectures

Figures 1-3 illustrate micronet models of a generic asynchronous RISC datapath. The intention is not to focus on the functional units themselves but rather on their asynchronous control and investigate their effect on the performance. The number of units and their functionality may be changed without side-effects.

The architecture can be described as a network of microagents (denoted by solid boxes) which are connected via ports. The microagents which are labelled in the figures, called Functional Microagents (FMs), perform microoperations which are typical of a datapath. On each of their ports are Communicating Microagents (CMs) which are responsible for asynchronous communications between FMs and the rest of the micronet. The FMs are effectively isolated and only communicate through their CMs, and can therefore be modified without affecting the rest of the micronet.

2.2 Measuring Performance

We next introduce a few metrics for measuring improvements due to the distribution of control. There are two principal characteristics which affect performance - the microoperation latency (the time between initiating the operation and the result being available), and the microoperation cycle time (the minimum time between successive initiations of the same operation, i.e. throughput). The metrics defined for MAP are as follows:

Minimum Datapath Latency (MDL) - The time between asserting the control signals (i.e. initiating instruction issue) and receiving the final acknowledgement of the instruction's completion.

Instruction Cycle Time (ICT) - The time between two identical instruction issues once that instruction's pipeline is full. In asynchronous pipelines which usually have non-uniform stage delays, the time between successive instruction issues is influenced by the slowest stage *currently* active in the pipe.

Program Execution Time (PET) - The actual execution time of the program.

A more detailed exposition of performance-related issues is presented in [1].

To study the effectiveness of the micronets, it is sufficient to focus on the LD, ST, and ALU instructions. Five simple test programs were devised to exercise the design. The Alu, Load and Store test programs measure the maximum attainable utilisation of their respective FMs. Each of these programs contain a number of identical instructions, such that only structural dependencies exist between instructions (in effect setting up a static pipeline or a fixed path through a network of components). The number of instructions in the test programs are sufficient to fill the pipeline, i.e. enough instructions exist for the Control Unit (CU) to achieve a steady issue rate. The Hennessy Test (HT1) consists of a mix of the previously-mentioned instructions but without any data dependencies, which exercises the spatial concurrency and out-of-order completion, both of which are provided by the micronet, for a particular schedule devised by the compiler. HT2 is a variant of HT1, with data dependencies, which exercises the data forwarding mechanism as well. This program represents a "typical" basic block of compiled code (actually a line of code in C from [4]).

To facilitate the simulation of instruction sequences within reasonable run-times and without sacrificing accuracy, the timing characteristics of the architecture (in 1.5 μm CMOS) were extracted from a post-layout simulation tool within a commercial VLSI design package called SOLO 1400 [2] and incorporated into a mixed-level (mainly register-transfer level) model. The processor was described in Occam2 and simulated on a parallel asynchronous event-driven simulation platform, on a transputer-based ME*i*KO Computing Surface.

3 REFINEMENTS

The following sections discuss a number of refinements which were made in three stages to the base design as shown in Figure 1. This highlights the ease with which the micronet model can efficiently exploit ILP and without the difficulties normally encountered in synchronous datapath design, such as implementing hazard avoidance, data-forwarding or balanced pipeline-stage design.

The processor design as illustrated in Figure 1 only exploits the actual execution timings of microoperations (Stage 1), whereas later designs exploit both this property and the available concurrency between the microoperations of different instructions. The execution of each instruction requires a predetermined set of microoperations, each initiated by signals from the CU. These are four-phased controls whose acknowledgement signals are used as status flags for mimicing a scoreboarding mechanism. In general, the microoperations for an instruction are initiated as soon as possible by asserting the necessary control signals. The receipt of an acknowledgement confirms that the associated microoperation has begun and the initiating control signal is de-asserted. The instruction is said to be issued once all the asserted control signals have been acknowledged, which then allows the next instruction issue to begin.

3.1 Stage 1

Figure 1 illustrates a naïve implementation of the datapath of an asynchronous processor, which does not as yet fully exploit the full repertoire of micronets. The control signals generated by the CU for Stage 1 are described in greater detail below:

Rx, (Ry) - This signal identifies the source register for the X (Y) Bus. The corresponding acknowledgement is asserted once the register has been accessed, and cleared once the data has been transferred to the operand fetch CM.

Rz - Same as above. The ST microoperation obtains the third operand over the Z Bus.

Rof - Same as above, but the value in the offset register is output onto the X bus.

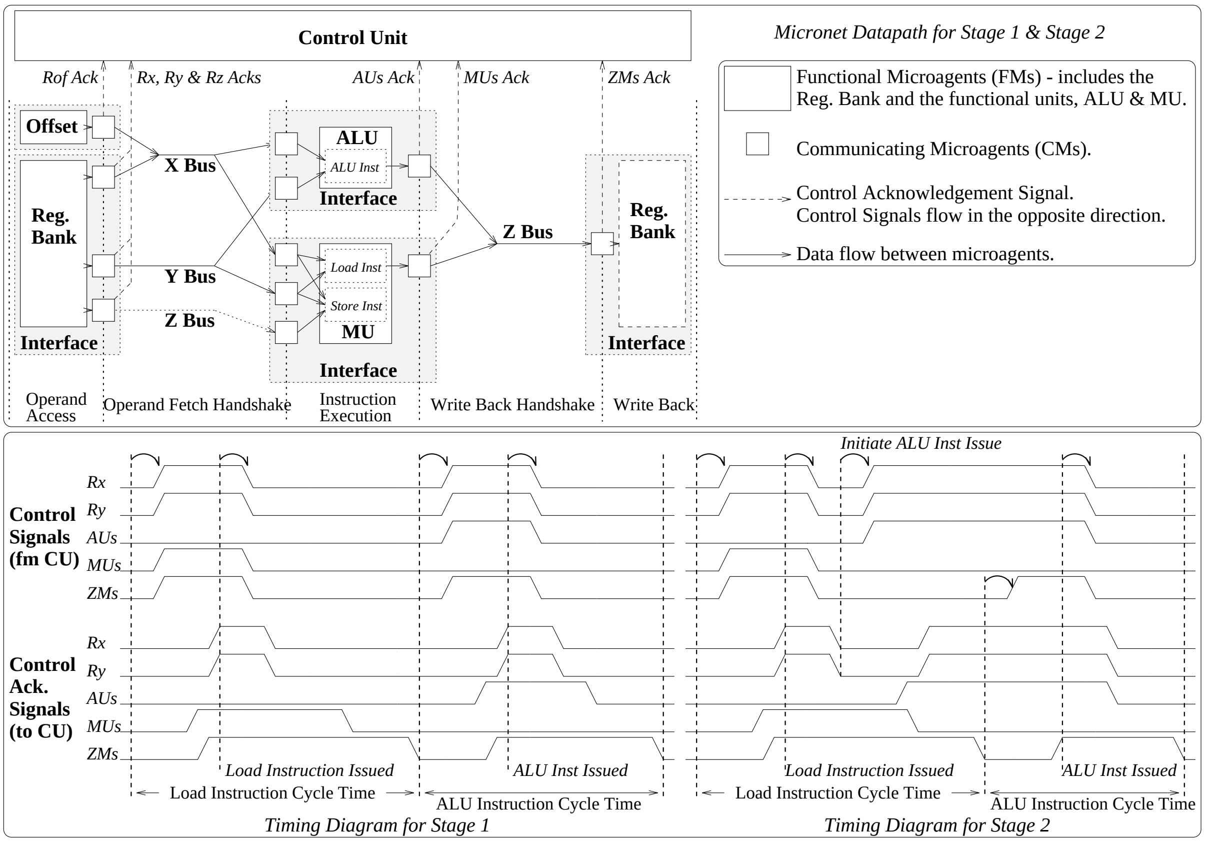

Figure 1: The micronet model of Stages 1 & 2

AUs - This signal identifies the next operation of the ALU. The corresponding acknowledgement is asserted when the interface is ready to fetch the ALU's operands from the registers and is cleared when it initiates the write-back handshake.

MUs - This signal identifies a load instruction to the MU and is asserted and cleared in the same manner as above. (Control signals for the other MU microoperations have been omitted for the sake of clarity).

ZMs - This signal identifies the destination register for data write-backs from the ALU or MU via the Z bus. The corresponding acknowledgement signal is asserted when the register is ready to receive data and cleared once the data has been written back.

In Stage 1, all the microoperations for a particular instruction are initiated together, and the next set cannot be initiated until the completion of the set of microoperations of the previous one. This effectively serialises the instruction execution, as illustrated in the timing diagram in Figure 1. In successive refinements the rôle of the CU is diminished by distributing the control of the micronet to local interfaces and microoperations are individually initiated as early as possible.

Instruction	Inst. Cycle Time (ICT)	Datapath Latency (MDL)
ALU	24nS	24nS
LD	43nS	43nS
ST	23nS	21nS

Table 1: Instruction Execution on Stage 1

In the base stage, the ICT is determined by the slowest control signal handshake since the next instruction issue cannot begin until all the previous handshakes have been completed. The results in Table 1 show that the ICT is equal to the MDL (except for the ST instruction), which is not surprising as instructions execute sequentially but only take as long as is necessary. The higher value for the ST instruction is due to a handshake delay, which in the LD instruction is hidden by the write-back stage. Although there is no explicit pipelining of the datapath, different phases of the handshaking may occur at the same time, e.g. a CM may initiate a handshake with another CM while completing one with its FM. As was expected the execution times of the test programs (Table 5) are the sum of their individual instruction execution times together with startup overheads.

3.2 Stage 2

The strict condition which was employed in Stage 1 for initiating a set of microoperations after the completion of the previous set is now relaxed. Furthermore, the CU can now assert any of the individual microoperations for an instruction asynchronously, where previously the set of microoperations for an instruction were initiated in unison. This allows microoperations relating to different instructions to overlap (Stage 2 in Figure 1). Note that a control signal which is related to an instruction can only be de-asserted once all of the relevant control signals have been acknowledged. The effect of relaxing this constraint is to introduce possible hazards and efficient mechanisms have been devised to avoid them. Fortunately, these hazard avoidance mechanisms are implicit in the orderings of the assertions of the control signals, known as the *pre-issue conditions* and these are discussed below:

Read-after-Write (RAW) - A register locking mechanism is implemented in the register bank without the CU having to keep track of the “locked” registers. The acknowledgement signal **ZMs** is asserted after the locking operation, and is de-asserted once the data is written back (signaling the unlocking of the register). By definition an instruction is issued once all the acknowledgements of the relevant microoperations have been received. This implies that the destination register of the previous instruction will have been locked before the CU initiates any of the current instruction's microoperations.

Write-after-Read (WAR) - This hazard is avoided without additional hardware overheads. When a register is used as both source and destination within the same instruction, then it is necessary to ensure that the source data is obtained before the register is locked, otherwise deadlock will occur. The CU stalls the assertion of **ZMs** until the source operand control signals **Rx** and **Ry** have been asserted.

Write-after-Write (WAW) - Although concurrent instruction execution can now take place, write-backs are still enforced in order. It is necessary to ensure that destination register has been locked, and that data is then written to its correct location. These conditions are met by simply preventing a functional unit (FU) from writing data back until the control signal from the CU has been de-asserted (an implicit go-write signal). This is sufficient since the control signals cannot be de-asserted before **ZMs** is asserted (see Figure 1). Note that if the CU attempts to lock a register which is already so, then the acknowledgement signal cannot be asserted and the current request will stall. This mechanism guarantees that write-backs to the same register occur in the correct order without stalling the instruction issue, and thereby allowing the instructions to execute concurrently with only the write-backs being sequential. The CDC6600 [11] used a similar go-write signal which sequentialised the execution of the offending instructions.

Operand Fetch - Simultaneous operand requests by FUs to the same Register Bank CM microoperation can lead to one of them acquiring the wrong operand. This can be avoided by the CU by delaying the assertion of the control signal to a FU until the previous FU has made its operand request(s) to the registers, i.e. until the acknowledgement signals of "operand fetch" microoperations have been de-asserted.

Bus Contention - Due to the mechanism to avoid WAW hazards only the Register Bank and either the ALU or Memory Unit can write onto the Z Bus simultaneously. Thus bus access is arbitrated by the CU through mutually-exclusive assertions of **Rz** and **ZMs**.

Instruction	Inst. Cycle Time (ICT)	Datapath Latency (MDL)
ALU	21nS	24nS
LD	42nS	43nS
ST	23nS	21nS

Table 2: Instruction Execution on Stage 2

The improvements in the instruction cycle times, as shown in Table 2, are small. This can be explained by the limited overlap between the operand access of the current instruction and the write-back of the previous one. In the design under consideration there can only be two program instructions active in the datapath simultaneously.

3.3 Stage 3

In Stage 3, the rôle of the CU is diminished further by distributing the control of the micronet to individual CMs. The CMs have been enhanced to more than just controlling local communications between FMs. They effectively buffer the initiations of the microoperations from the CU until their respective FMs are ready to perform. Also, the write-back to the register bank is no longer controlled by the CU, but directly by the CMs of the FMs which require the service, i.e. the write-back microoperation is initiated by the microoperations in the previous stage.

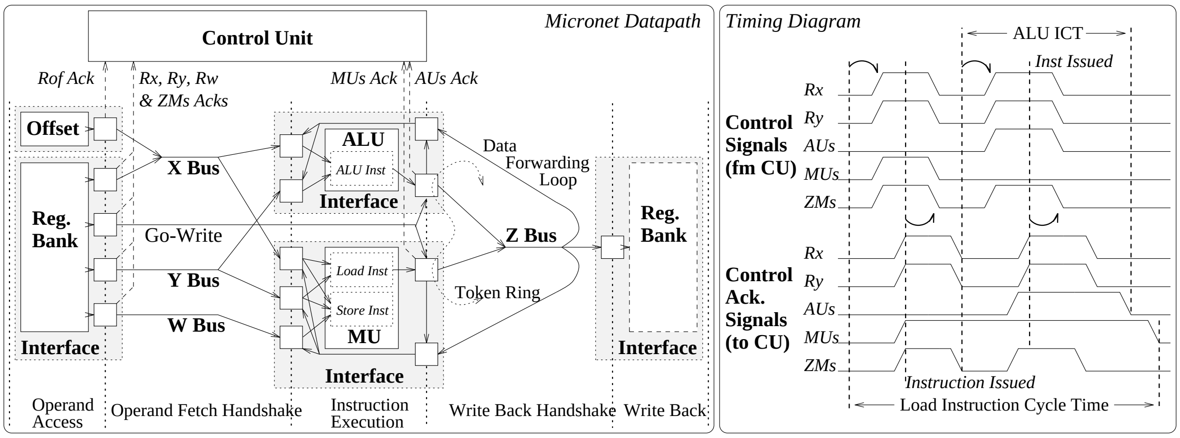

Figure 2: The micronet model of Stage 3

Enforcing write-backs in order restricts the degree of concurrency which can be exploited, especially when the FU executions times vary significantly. However supporting out-of-order completion of instructions in an asynchronous environment is more difficult than under synchronous control. Determining the precise order in which results will be available is virtually impossible since microoperation delays vary.

Out-of-order instruction completion is supported by tagging the write-back data with the address of the destination register. The CU cannot predict the write-back order, therefore a decentralised bus arbitration scheme as in a token ring is employed. The ring is distributed amongst the CMs and is very simple to implement in VLSI. However, the ring's cycle time will increase with the number of FMs, and might be infeasible for larger numbers.

With data transfer on the Z bus being tagged, CMs can identify and intercept operands for which it may be waiting. This mechanism is reminiscent of the IBM 360/91 common bus architecture [12]. Data-forwarding has been implemented by exploiting the feedback loops of the micronet. In the event of data forwarding, where data is routed directly to the CM of a waiting FM, the CM's previous request for that operand is in effect cancelled by initiating a separate handshake. This frees the corresponding "operand fetch" CM to service its next request. An alternative approach would be to implement operand bypassing, where the operand is fed back to the "operand fetch" microoperation. This avoids the need for data forwarding CMs and the cancel handshake. The dual rôle of the Z bus can no longer be supported due to the data-forwarding mechanism. A separate operand fetch bus (W bus) is used, thereby making the Z bus purely a write-back one (see Figure 2).

As a result of these modifications, the acknowledgements to the control signals and the pre-issue conditions have to be revised as shown below:

Rx, (Ry, Rw) - The acknowledgement is asserted by the CM of the register bank when the X (Y, W) bus operand fetch microoperation is ready, and de-asserted once the operand fetch handshake is in progress.

Rof - Same as above. Note that both the control signals **Rx** and **Rof** cannot be active simultaneously.

AUs, MUs - The acknowledgement is now asserted when the corresponding CMs are ready to fetch the operands from the registers and is cleared once the FM microoperation has completed.

ZMs - The acknowledgement signal is asserted when the CM is ready and de-asserted once the operation has been completed (i.e. the register has been locked).

RAW -The CU delays the assertion of the operand fetch control signals **Rx**, **Ry** and **Rw** until the previous **ZMs** control acknowledgement signal has been de-asserted, which indicates the locking of the previous destination register.

WAW - The mechanism is unchanged except that the go-write signal originates from the register interface and not the CU (i.e. the mechanism has now been decentralised).

Write-back Contention - This is prevented by the use of a token ring to arbitrate accesses to the write-back (Z) bus. Of course, this problem could be obviated by using dedicated buses for small number of FMs, but is impractical for designs with larger numbers.

Further concurrency is achieved by applying these pre-issue conditions only when necessary by explicitly checking register addresses for dependencies between successive instructions.

Instruction	Inst. Cycle Time (ICT)	Datapath Latency (MDL)
ALU	15nS	24nS
LD	38nS	43nS
ST	23nS	21nS

Table 3: Instruction Execution on Stage 3

We observe an improvement in the cycle times of instructions which require to write data back to the registers, such as the LD and ALU instructions, as shown in Table 3. This is due to the de-centralisation of the write-back control to the relevant CMs. These improvements are reflected in the shorter PETs for Load, Alu and HT1 test programs, as shown in Table 5. Columns "HT2" and "HT2(DF)" refer to the cases without and with data-forwarding, respectively.

3.4 Stage 4

In this final stage, both the assertion and de-assertion of the control signals now occur independently of each other. The states of the FU acknowledgements no longer represent the activity of their FMs, but rather that of their operand-fetch CMs. All of this further increases the concurrency between microoperations which makes possible the exploitation of fine-grained concurrency between instructions.

The ICT value for the LD instruction in Table 4 is the best attainable as it represents the MU delay for the operation. These figures show that the micronet can exploit the

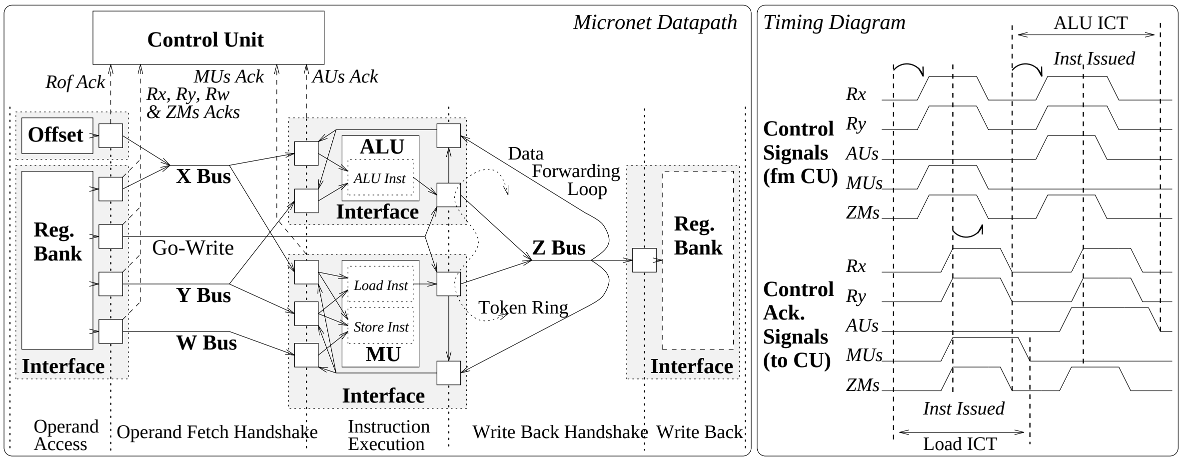

Figure 3: The micronet model of Stage 4

Instruction	Inst. Cycle Time (ICT)	Datapath Latency (MDPL)
ALU	12nS	24nS
LD	23nS	43nS
ST	12nS	21nS

Table 4: Instruction Execution on Stage 4

actual operational cost and effectively hide the overheads of self-timed design. The ICTs for the ALU and ST instructions are limited by their operand fetch cycle times. The overall improvements in the program execution times in Stage 4 over Stage 1 for the first three test programs (shown in Table 5 and Figure 4) are due to improvements in temporal concurrency due to the pipelining of the datapath. The actual speedup which is achieved is less than the maximum attainable improvement (the ratio of the ICTs in Tables 1 and 4), due to the MDL and the startup overheads (for longer test programs the speed-up will approach this maximum value). The speed-up for HT1 is due in part to pipelining of the instructions as observed in the other test programs, but also due to additional spatial concurrency due to the overlapping of different instructions in the same stage of the micronet. This further improvement is still significant (approximately 17% in this example) given that both successive instruction operand fetches and write-backs are effectively forced to take place sequentially due to resource constraints. (In fact, since these delays are larger than

PET	Alu Test	Load Test	Store Test	HT1	HT2	HT2(DF)
Stage 1	175nS	308nS	164nS	143nS	143nS	-
Stage 2	157nS	302nS	165nS	119nS	119nS	-
Stage 3	121nS	280nS	165nS	83nS	97nS	91nS
Stage 4	103nS	188nS	98nS	79nS	-	91nS
Effective Speed Up	1.75	1.66	1.71	1.89	-	1.62

Table 5: Execution Times of the Test Programs

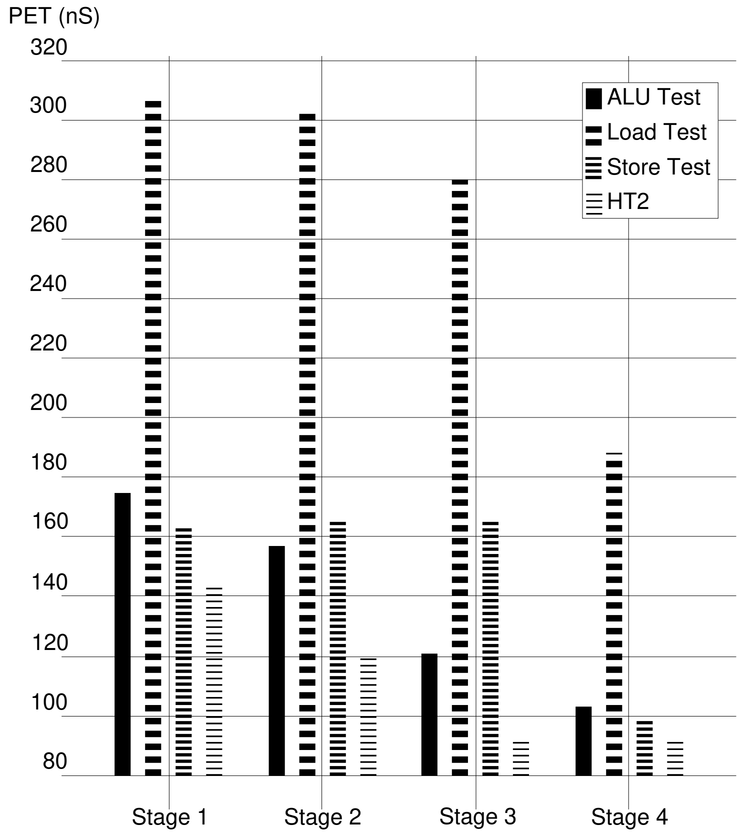

Figure 4: Comparison of Execution Times of the Test Programs

the FM delays for the Store and ALU operations, the scope for spatial concurrency in this particular example is quite small). As the number of microagents in each stage is increased, the spatial concurrency effect will be more pronounced. The speed-up for HT2 as expected reflects the reduced concurrency which can be exploited, due to the data dependencies in the program.

In summary, the rôle of the CU in an asynchronous processor has been considerably simplified to just initiating individual microoperations as early as possible. The control of the datapath is distributed to local interfaces, courtesy of the micronet.

4 CONCLUSIONS

This work has investigated the influence of an asynchronous control paradigm on the design and performance of processor architectures. By viewing the datapath as a network of microagents which communicate asynchronously, one can extract fine-grain concurrency between and within instructions. The micronet can be easily implemented using simple self-timed elements such as Muller C-elements [7] and conventional gates. Future work will investigate the suitability of asynchronous processors as targets for optimising compilers.

Acknowledgements

V. Rebello was supported by the U. K. Engineering and Physical Sciences Research Council (EPSRC) through a postgraduate studentship. This work was partially supported by a grant from EPSRC entitled *Formal Infusion of Communication and Concurrency into Programs and Systems* (Grant Number GR/G55457).

References

[1] D. K. Arvind and V. E. F. Rebello. On the performance evaluation of asynchronous processor architectures. In P. Dowd and E. Gelenbe, editors, *Proceedings of the 3rd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'95)*, pages 100–105, Durham, NC, USA, January 1995. IEEE Computer Society Press.

[2] European Silicon Structures Limited. *Solo 1400 Reference Manual.* ES2 Publications Unit, Bracknell, U.K., 1990.

[3] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. In T. Yanagawa and P. A. Ivey, editors, *The Proceedings of the IFIP International Conference on Very Large Scale Integration (VLSI'93)*, pages 5.4.1–5.4.10, Grenoble, France, September 1993.

[4] J. Hennessy and T. Gross. Postpass code optimisation of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[5] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *The Proceedings of ASPLOS III*, pages 272–282. ACM Press, April 1989.

[6] C. Mead and L. Conway. *Introduction to VLSI Systems.* Addison-Wesley, Reading, Mass., 1980.

[7] R. E. Miller. *Switching Theory. Volume II: Sequential Circuits and Machines.* John Wiley and Sons, 1965.

[8] W. F. Richardson and E. L. Brunvand. The NSR processor prototype. Technical Report UUCS-92-029, Department of Computer Science, University of Utah, USA., 1992.

[9] R. F. Sproull, I. E. Sutherland, and C. E. Molnar. Counterflow pipeline processor architecture. Technical Report SMLI TR-94-25, Sun Microsystems Laboratories Inc., April 1994.

[10] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[11] J. E. Thornton. *Design of a Computer: The Control Data 6600.* Scott Foresman and Company, 1970.

[12] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.