Micronets: A Model for Decentralising Control in Asynchronous Processor Architectures

D. K. Arvind, R. D. Mullins and V. E. F. Rebello
Department of Computer Science, The University of Edinburgh
Edinburgh, EH9 3JZ, United Kingdom
E-mail: dka@dcs.ed.ac.uk

Abstract

Micronets model processor architectures as a network of communicating resources, in contrast to the traditional one of a linear pipeline. Micronets distribute the control to the functional units, which enables the exploitation of fine-grain concurrency between instructions. The overhead due to asynchrony is hidden with the four-phase protocol being used to implement scoreboarding and hazard avoidance mechanisms, without incurring additional control costs. This paper demonstrates the feasibility of micronet-based processors. Results are presented for SPICE-level simulations of a 0.7μm CMOS implementation of a datapath. The relationships between micronets and both the compiler and the computer architecture are also explored.

1 Introduction

Micropipelines [22] have been used to model linear asynchronous pipelines such as datapaths [6] [18], and two-dimensional pipeline structures [8]. However, viewing a datapath as a single linear pipeline has limitations [2]. A new paradigm called micronets has recently been proposed for the distribution of control in asynchronous processor architectures [1]. Micronets model datapaths as a network of communicating functional units which allows the efficient exploitation of both fine-grained instruction-level parallelism and the actual execution costs of instructions.

The choice of a four-phase communication protocol [19] between the functional units allows the efficient utilisation of these resources, by avoiding the additional control costs (scoreboarding and hazard avoidance mechanisms) normally associated with processors which exploit ILP.

The design of an effective micronet-based system should also consider the interplay between the compiler and the processor architecture, i.e. does a micronet-based processor offer a good target for a parallelising compiler. The influence of this asynchronous target on compiler design is briefly discussed.

2 Micronets

Micronets are a generalisation of micropipelines. The operations within each of the micropipeline stages are exposed in the form of fine-grain *microagents*. The microagents in any "stage" can operate concurrently, and microagents in the different "stages" communicate with each other asynchronously. Program instructions only utilise the relevant microagents and for just as long as is necessary. More than one instruction may utilise different microagents within a "stage". Figure 1 compares the resource utilisation in micropipelined and micronet datapaths. In the former, the number of active instructions is never greater than the number of pipeline stages, and at any time only a subset of the resources in each of the stages is normally utilised. In micronets the number of instructions which may be active at any time is bounded by the number of microagents. An instruction which does not require any of the resources within a "stage" can skip it. Furthermore, the time spent by instructions in microagents may vary. Due to these reasons instructions may overtake. Synchronous processors which permit this do so at a significant control cost; the resulting scoreboarding mechanism is also used to resolve structural hazards (together with software or hardware interlocks to resolve data dependencies). All of this comes for free in a micronet-based datapath: the existing handshaking mechanisms and register locking provide these services [1].

In practice datapaths have to deal with conditions which interrupt the flow of instructions, such as condition branching and exception handling. The RISC architectures have popularised the "delayed branching" approach to reducing the performance degrada-

Published in the Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies, pp 190-199, London, UK, May 1995.

© *IEEE Computer Society Press.*

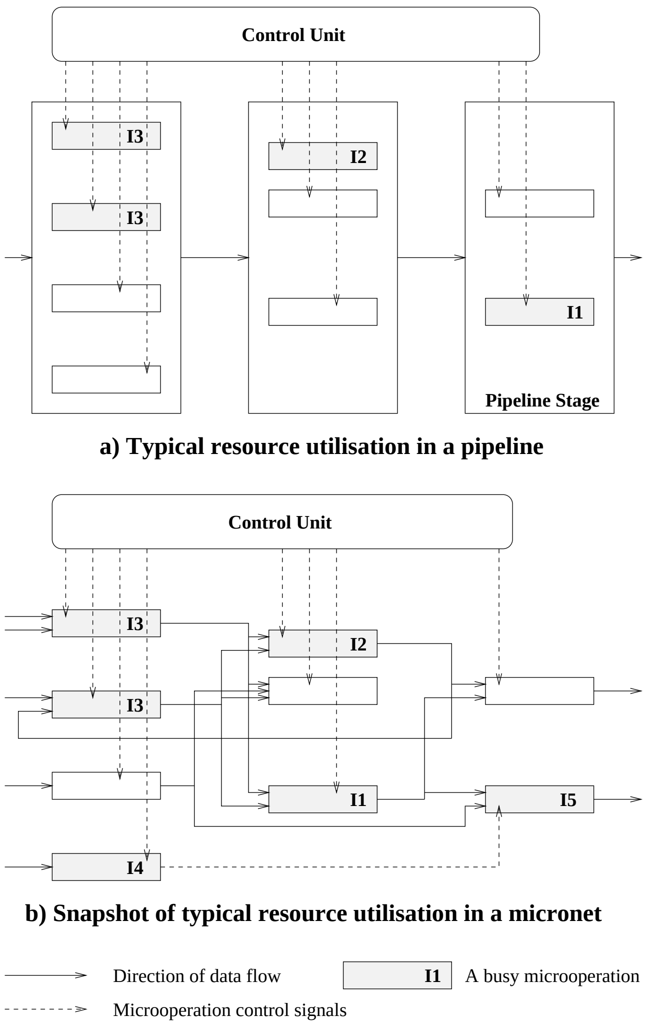

Figure 1: Contrasting a micropipeline with a micronet

tion of condition branching [9] [16] [17]. However, this technique is unsuitable for asynchronous datapaths because of the difficulty in estimating the time to resolve the branch condition (which is fixed in synchronous architectures). Therefore, the number of instructions which have to be fetched cannot be determined. For micronet-based architectures, the preferred techniques are ones which do not rely on fixed timing for their correct operation, such as branch prediction schemes [12] [20] or advanced branching mechanisms [15]. Precise exception handling and speculative execution are supported through the use of history and write-back buffers [21] [23].

A micronet-based datapath, as illustrated in Figure 2, is composed of a network of microagents (denoted by solid boxes) which are connected via ports. The Functional Microagents (FMs) perform microoperations which are typical of a datapath. On each port of a FM is a Communicating Microagent (CM) which is responsible for communications among the FMs, and with the Control Unit (CU). The FMs are effectively isolated and only communicate through their CMs, and can therefore be modified without affecting the rest of the micronet. The protocol used in the design of micronet-based datapath is discussed in the following section.

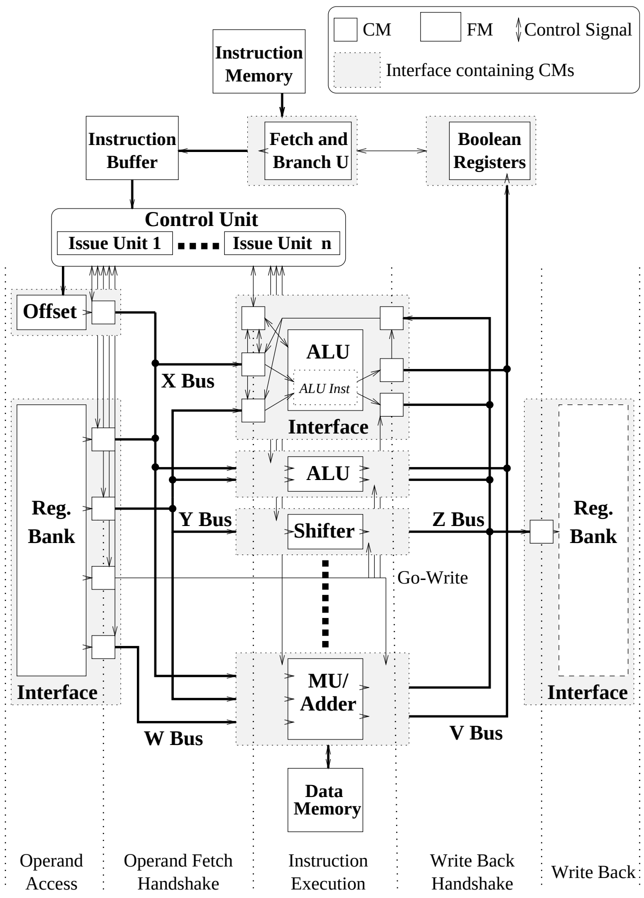

Figure 2: A micronet-based processor architecture

2.1 Choice of protocol

Both transitions in a generic four-phase protocol (the assertion and the return-to-zero) are accompanied by additional acknowledgements from the receiver. The principal advantage of this approach is a simpler circuit implementation. However, it uses twice as many transitions than is necessary and whenever the wire delay is a substantial fraction of the operation time, the extra trip required by a single communication can be a serious performance penalty. In fact, the reset phase of the handshake does not signal any event, thus leading some designers to modify the protocol to simultaneously reset the two signals after the active phase to reduce the handshake cycle time [5]. The micronet is only concerned with the external communications between microagents, which might use a different protocol internally. Micronets employ the traditional four-phase handshaking protocol for both control and local bundled data transfer. Other reasons, more specific to micronets, have influenced this choice, and these are discussed next.

2.1.1 Fast instruction issue

One of the significant features of micronets is its ability to exploit spatial concurrency within the datapath. This requires a fast instruction issue rate to keep the microagents busy. The CU initiates the microoperations for each of the instructions individually and as early as possible. The acknowledgements from the CMs (after a delay of one C-element) confirm that the corresponding microoperations will be initiated. The instruction is considered to have been issued once the CU has received all the acknowledgements. This corresponds to the first half of the four-phase protocol. The CU is free to issue the next instruction, while the reset phase of the protocol completes. This is done when the corresponding acknowledgement signal is deasserted which signifies that the particular resource is ready for the next request. The instruction releases the resources individually as soon as the respective microoperations have completed, freeing the resources for another instruction. Figure 3 shows the activity of two resources in micronets in comparison to a similar synchronous pipeline and micropipeline.

The microoperations of different instructions may overlap leading to potential hazards. Since the acknowledgement signals denote the busyness of resources, they can be collectively used as a scoreboard. Hazard avoidance due to data dependences is implicit in the orderings of the assertions of the control signals [1]. These *pre-issue conditions* stall the assertion of the respective control signal until the completion of one of the halves of the handshake protocol of the dependent microoperation control signal(s).

Although a four-phase protocol would be considered twice as expensive as a two-phase one, the same efficiency is obtained as two back-to-back, two-phase handshakes by representing two events in each cycle. The recovery transitions are used by the control unit for scoreboarding and hazard avoidance. This is necessary for efficient exploitation of ILP, since the control unit has to issue each instruction before the previous one completes its execution. Furthermore, a four-phase protocol exposes more concurrency by effectively decoupling the sender's and receiver's operations from their communication [1].

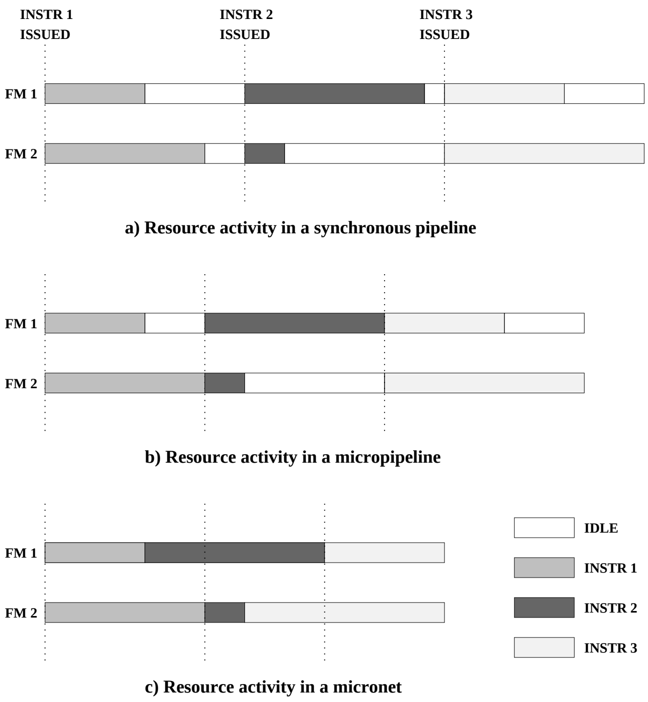

Figure 3: Resource activity

2.1.2 Routing data in micronets

Although the actual data transfer between microagents is controlled locally via handshake protocols, the access to shared resources, such as data highways, may be controlled either globally by the CU or locally by an arbitration scheme. Global control is used in cases where the order of granting resources is known in advance and has to be enforced. This is again achieved through the use of pre-issue conditions [1]. Otherwise, a local mutual exclusion scheme such as in token rings or arbiters will grant requests. For example, the writing back to the register bank is controlled directly by the CMs of the FMs which require this service. As a consequence of this and also due to the differences in the execution times of microoperations, instructions may complete out of order. Therefore data has to be tagged with its destination which also enables data-forwarding to be supported.

The reader is referred to [1] for further information on micronets, and to [2] for the performance evaluation of micronet-based datapaths.

3 Implementing a micronet-based datapath

The Control Unit (CU) is essentially an instruction queue which is now reduced to simply issuing instructions. The CU makes requests to the interfaces of the functional units, with the instruction's completion being controlled by these interfaces, thereby leaving the issue unit free to rapidly initiate the next instruction. The microoperations associated with these instructions operate concurrently, subject to dependence rules.

The destination registers can be locked to ensure that they are accessed in the correct order. In addition, a number of pre-issue constraints or dependence rules are implemented in the control unit to prevent Write-after-Read (WAR) and Read-after-Write (RAW) pipeline hazards. The rules may be relaxed by checking for dependencies between specific registers which are required by the current instruction. A Go-write mechanism maintains the correct ordering of write-backs to the same register, thus avoiding Write-after-Write (WAW) errors, without stalling the issue of instructions [1].

Figure 4 illustrates the instruction issue logic. Only a selection of microagents are required on each instruction issue cycle. A dual-rail encoder is used to produce both the requests for the necessary microagents and a complemented signal to mask those which are not required. Microagents which are not involved in the current instruction issue will not block due to this masking mechanism.

The dependence rules, P_0 to P_n, represent the specific conditions that must be met before that microoperation is issued. Each wait operation is implemented as a single asymmetric C-element, which ensures that any subsequent clearing of the pre-issue constraint does not result in another issue request.

The ALU and its interfaces are shown in Figure 5. The interface or control logic for the ALU can be decomposed into two main operations - execution, and write-back, which operate concurrently. Execution involves the fetching of operands, the go-write mechanism and the ALU operation itself, while the write-back allows the result to be written to the destination register. On the issue of an ALU microoperation, execution progresses as follows:

- The acknowledge to the control unit is asserted to signal that the microoperation has begun.

- The execute interface initiates the go-write and both operand requests.

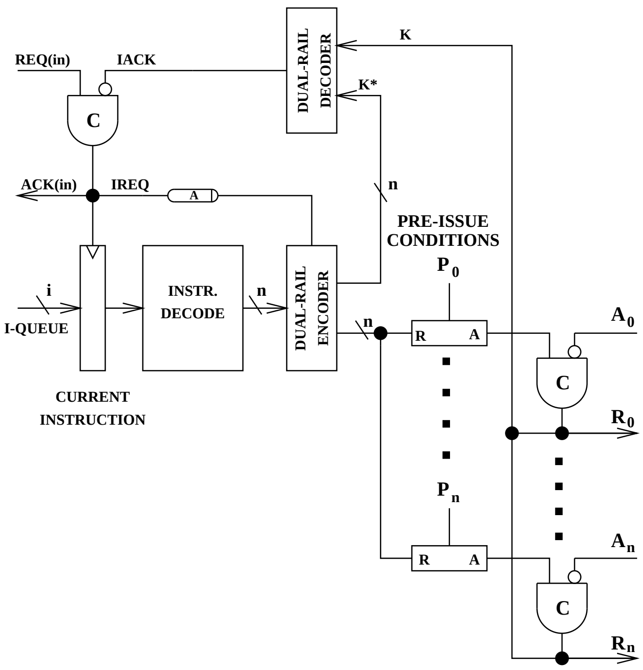

Figure 4: Instruction issue logic

- The operand interfaces make requests to the register bank for their operands (operand register details are provided directly by the control unit). A go-write request is also made by the go-write interface.

- The ALU's operand fetch interfaces signal the receipt of data to the ALU's execute interface, and complete their handshakes with the register interface.

- Operands are dual-rail encoded to allow a completion signal to be generated.

- The ALU operation begins, the result is detected and the execute interface attempts to issue a write back operation to the ALU's Z-bus interface.

- If the Z-bus interface is free then the result is latched and the ALU acknowledge to the control unit is deasserted. The control unit may now issue the next ALU microoperation.

- If a go-write permission has been received, then the write back occurs as soon as it obtains access to the Z-bus.

A closer inspection of the behaviour of the interfaces shows that the overhead in using a four-phase handshaking protocol is indeed hidden, as shown in the example that follows for the operand interface.

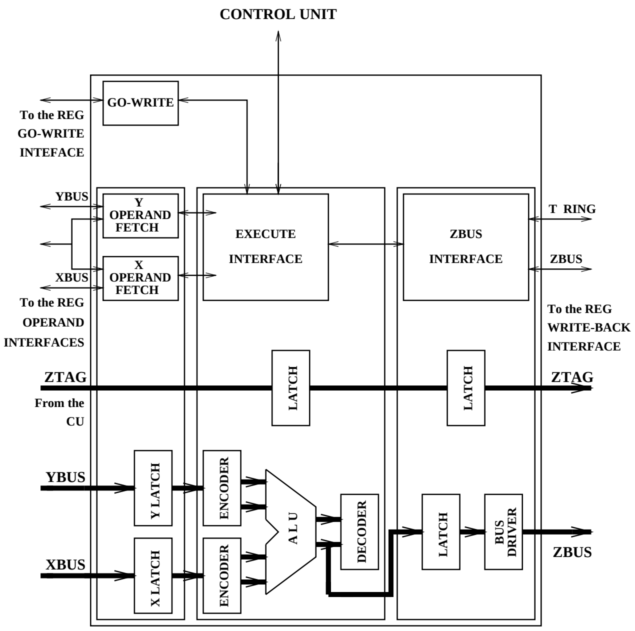

Figure 5: Interfaced ALU

Phase 1 - Request is made to the register bank for an operand. This phase is usually hidden since it takes place concurrently with the operand's register access.

Phase 2 - Acknowledge or data valid signal is received, the receipt of operands is now detected by the control unit and the ALU operation may begin.

Phases 3 and 4 - Handshake completes concurrently with the ALU operation, and these extra phases are effectively hidden.

The destination register for each ALU operation is stored in two tag latches, as shown in Figure 5. The tag and data are sent together to the register block allowing the correct destination register to be selected.

The functions of the register interfaces as shown in Figure 6 are listed below:

Operand Interfaces - These interfaces communicate with the control unit, the register bank and the operand fetch interfaces of other functional units, to control the supply of operands. An operand may only be sent to a functional unit when the following operations have been successfully completed:

- A request has been made to the operand interface by the CU.
- The register's lock bit has been read as clear.
- A request has been made by the functional unit for the operand.

Lock Interface - The behaviour of the lock interface is similar to that of the operand interfaces:

- A request is made by the CU to lock the destination register, which is locked when ready.
- The Go-Write permission is then granted to the requesting functional unit.

Write-Back Interface - This interface accepts data and register tags from the Z-bus and writes the data back to the appropriate register. The corresponding lock bit is then cleared.

Accesses to both the data registers and the lock bits is controlled by using a fixed delay. As the register access time is almost constant, there are few advantages to be gained by implementing a "data valid" signal.

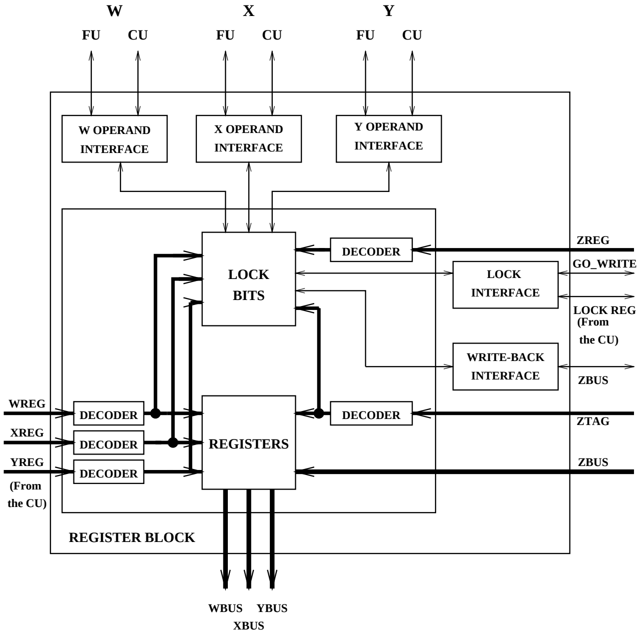

Figure 6: Interfaced Register Bank

Since the write-back bus is shared by a number of functional units, some form of arbitration mechanism must be used to avoid contention, like a token ring. Although easy to implement, its performance would degrade with an increase in resources sharing the bus.

4 Simulation Results

A prototype datapath was implemented in ES2's 0.7μm CMOS process using the Cadence design tools. They were used to create a library of self-timed components and datapath elements. The Cadence Design Framework provided interfaces to both VHDL and HSPICE. A VHDL model of the datapath was created from a high-level specification and synthesised. The HSPICE simulations of the entire datapath took approximately 17 hours on a SUN Sparc-10.

Figure 7 shows the execution of an ALU instruction with traces of the relevant control signals being numbered.

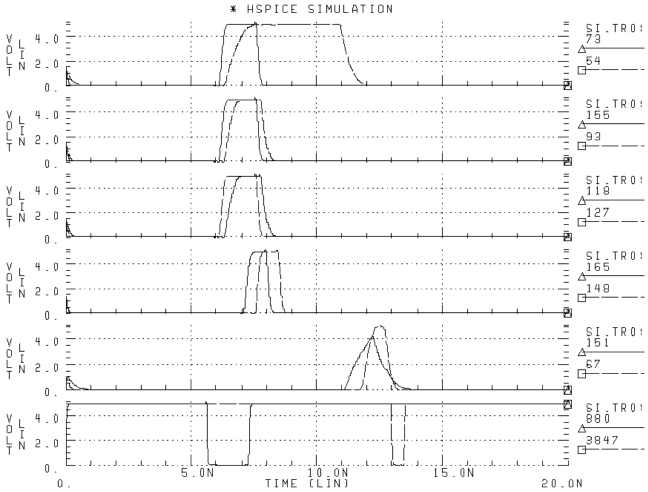

Figure 7: ALU Instruction

Panel 1 - An asserted request signal (**73**) from the control unit to the ALU initiates an add operation. The acknowledge signal (**54**) represents the period of ALU activity.

Panel 2 - The operand request signal (**155**) is sent by the control unit to initiate a register access microoperation. The acknowledge signal (**93**) is asserted by the register operand interface to prevent further operand requests until a functional unit has claimed the current operands. Only the signals for one of the operands is shown.

Panel 3 - A request to lock the destination register (**118**) and the register bank acknowledge signal (**127**) are shown. The acknowledge signal is lowered after the register has been locked and a go-write request has been received from a functional unit.

Panel 4 - Shows an ALU operand request (**165**) to the register bank, together with the corresponding data valid signal (**148**) from the register bank.

Panel 5 - After the add operation has completed, a request (**151**) is made to write the result to the destination register. The receipt of data at the register bank is signaled by the assertion of the acknowledge flag (**67**).

Panel 6 - Shows the instruction decode start signal (**880**). The duration of this signal indicates the instruction issue time. Also shown is the register write signal (**3847**), where data is written back on the final edge of this signal. The complete instruction execution time is represented by the delay between the first and last edges as shown in this panel. (Note that both signals are active low.)

The following subsections describe a number of measurements which influence the performance of micronet-based datapaths.

4.1 Handshaking

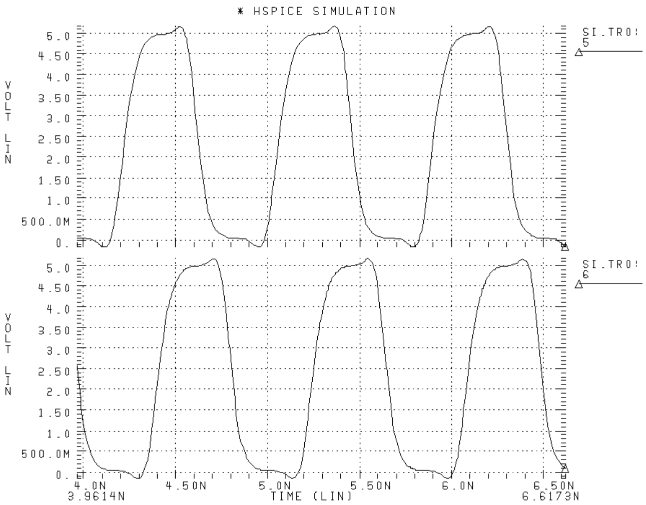

Figure 8: The handshake cycle

The handshake cycle is implemented by two back-to-back C-elements. This forms the basis for distributed control in micronets (a circuit commonly used for communication between microagents) and therefore a crucial factor which influences performance within the micronet. Ignoring the computation within a microagent, the throughput would then be limited by the control handshake cycle. Figure 8 shows a cycle time of 0.8nS, corresponding to a maximum throughput rate of 1.25GHz. This suggests that micronet control circuitry is unlikely to limit throughput in processing pipelines.

4.2 Maximum instruction issue rate

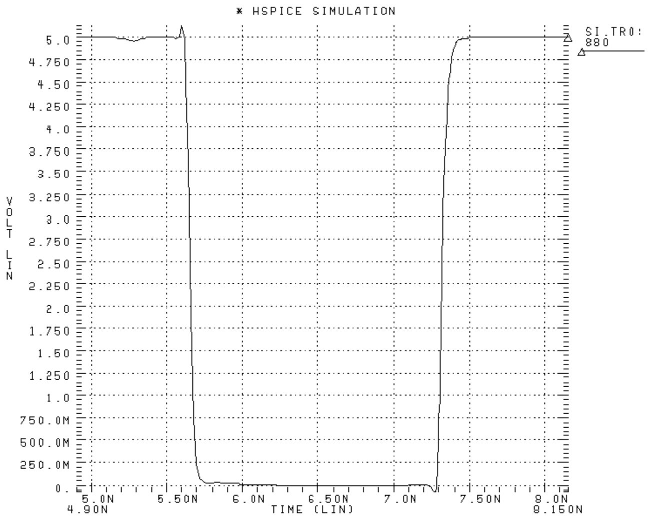

Figure 9: Maximum instruction issue rate

Figure 9 represents an instruction issue time of 1.85nS. The maximum instruction issue rate is determined by the earliest possible reassertion of the issue signal. Given sufficient instruction fetch bandwidth, the minimum cycle time for this signal is 2.05nS which equates to a maximum instruction issue rate of 488Mhz. This represents a theoretical upper limit on processor performance while ignoring datapath delays.

4.3 ALU throughput

Figure 10 shows the signal from the control unit (**73**) being asserted to initiate an ALU operation. The period when both the ALU and its interface are busy is represented by the duration of signal **54** (4.31nS). During this period the ALU interface requests both operands, initiates the operation, detects the result, obtains write-back (go-write) permission and writes the result to the Z-bus. The actual instruction execution time of the ALU is determined by the period between the operands arriving and the ALU's acknowledge being deasserted (3.11nS). This is the delay required to add without any carry propagation and thus represents the minimum time through the functional unit. The minimum ALU instruction cycle time is determined by the earliest possible reassertion of signal

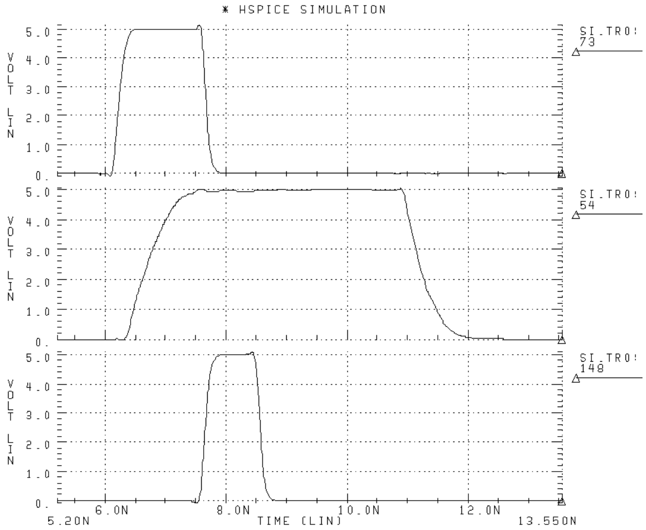

Figure 10: ALU Activity

73. This cycle time was estimated at 4.51nS, implying a peak processor performance of 222 MIPS for add instructions.

Only the FM latency should be considered as time spent in useful work, with the other delays being overheads of control paradigm and the architecture. In this implementation, the micronet overhead for this operation is 1.4nS (the difference between the operation's cycle time and its latency). This overhead can be effectively removed by modifying the ALU interface to deassert the microoperation acknowledge signal to the CU once the operands have been fetched [2].

4.4 Operand fetch

The operand fetch delay, as shown in Figure 11, was calculated as the period between the assertion of the operand request signal (**155**) and the assertion of the data valid signal (**148**) from the register bank (1.45nS). The actual time to access one of the registers is determined by the duration between the assertion of the operand request acknowledge (**93**) and the data valid signal (1.24nS).

4.5 Write-back

Figure 12 shows that the time, between the result becoming available at the output of the ALU and being written back in the destination register, is 2.48nS. The actual time taken to write data to a register is 0.5nS (duration of signal **3847**). The slow rising and falling edges of the write-back request (**151**) signal, limits the write-back rate to 474MHz.

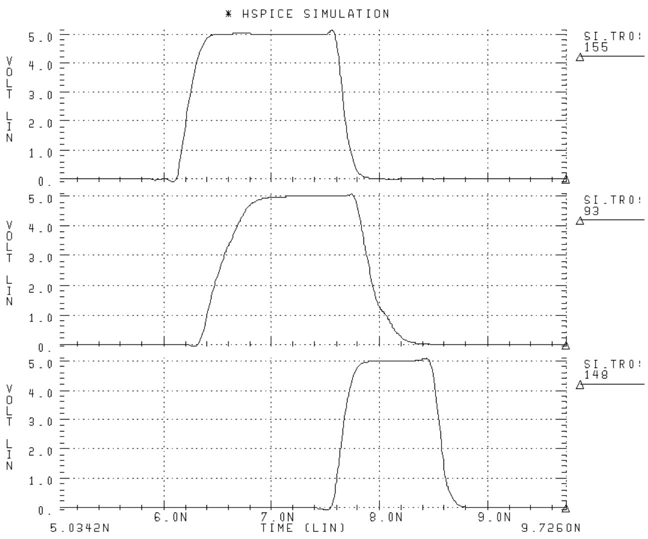

Figure 11: Operand Fetch

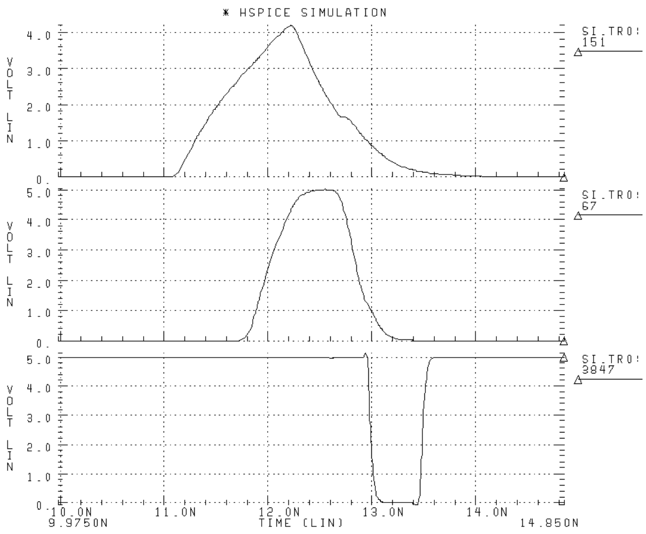

Figure 12: Register Write Back

Note that no circuit optimisation of transistor sizes have yet been made to either improve performance or sharpen edges. Micronet datapaths can be synthesised from high-level specifications using a custom-built library of four-phased self-timed components and interconnection cells.

5 Discussion

The emergence of VLSI technology, together with the maturing of optimising compiler techniques, had aided the development of early RISC architectures [9] [11] [16]. Their primary concern was the efficient usage of expensive silicon real estate, and careful consideration was given to the design of the instruction set architecture [13]. There have been two orthogonal trends in the evolution of synchronous processor architectures [10]: the deeply-pipelined architectures [14], i.e. ones which exploit temporal parallelism, and superscalar architectures which exploit spatial parallelism [3] ([4] is an example which exploits both). Both these classes have benefited from improvements in technology and the resulting faster clock frequencies. But these improvements have been sustained at a high price in terms of clock distribution, power consumption, and design complexity [4]. Furthermore, significant additional control costs are incurred in exploiting ILP in both cases.

Micronets offer an alternative model for the design of future processor architectures. Whereas the original RISC ideal was the efficient usage of the silicon space by identifying the critical resources, we are essentially concerned with their efficient utilisation over time. We achieve this in two ways: by removing the clock, and distributing control to the resources; and viewing the datapath not as a linear pipeline, but as a network of communicating resources. We are able to efficiently (the overheads due to asynchrony are hidden [1]) exploit a fine-grain ILP without the additional control costs (the protocol also implements a scoreboarding and hazard avoidance mechanisms).

The asynchronous and distributed nature of the control in micronets allows the processor to be easily extended with little effect on the rest of the design. For a given class of problems, the designer is able to easily explore the architectural design space more accurately by adding critical resources. This can be naturally extended to superscalar architectures by increasing the number of issue units. (Synchronous superscalar architectures replicate entire datapaths.) The same scoreboarding mechanism is shared between the issue units for determining the global state of the datapath. A micronet-based superscalar architecture has been designed and its performance is currently being evaluated.

5.1 The Micronet and the compiler

The micronet model exposes structural concurrency in the datapath, with fine-grained resources now being visible to the compiler. It is the task of the compiler to schedule instructions such that these resources are efficiently utilised. The instruction schedule is devised based on a model of the architecture; for synchronous architectures the model is simple: instructions do not interact and their execution times are fixed. In contrast, an asynchronous model is necessarily less accurate for the following reason: execution times for the same instruction may vary due to environmental parameters, data-dependent operations, and interactions between different instructions which are simultaneously executing in the micronet. We have considered models based on worst-case instruction execution times where the resulting schedule is treated as a first pass one. The instructions are dynamically reordered at run-time to tune this schedule and due to the asynchronous behaviour these instructions are issued as soon as possible, without the need for delays using NO-OP instructions.

A micronet-based datapath has several communicating "pipelines" which can all potentially be busy simultaneously. The control unit aims to issue those instructions successively which minimise resource contention. It will only stall if no instructions are available for issue, or all the instruction's resources are busy. The micronet's asynchronous behaviour minimises the duration of this stall. In the case of instructions with data or structural hazards, both instructions are issued without stalling, with the second instruction executing until the busy microagent. These fine-grain hazard avoidances are enforced at run-time by the pre-issue conditions of the micronet.

Other reasons contribute towards the complexity of compiler-time scheduling on micronets. An initial state of activity is assumed for scheduling a basic block within the micronet, which might well be different at run-time. The actual state can indeed be determined at run-time thanks to the implicit scoreboarding mechanism in the CU. This information is used to dynamically alter the static schedule by identifying an instruction which can be executed immediately (easily achieved using the control acknowledgement signal), after checking for independence from the previous instructions in the buffer, which is determined at compile-time and marked by a concurrency bit. The instruction issue is only limited by the availability of resources and operands, in the presence of out-of-order instruction issue. Micronets can therefore be viewed as a hybrid dataflow style of architecture which is limited to the window of instructions available in the instruction buffer, without the bookkeeping costs of traditional dataflow architectures [7].

6 Conclusions

We have presented a new model, called micronets, for decentralising controls in asynchronous processor architectures. They are viewed as a network of communicating functional units, which expose fine-grain concurrency between instructions. We have demonstrated that four-phase handshaking protocols enable the implementation of highly concurrent structures and in most cases the overheads can be hidden. Just as importantly, these protocols are used to efficiently avoid datapath hazards.

The modular nature of micronets eases modification and empowers the computer architect with finer control in the design, for example, of superscalar architectures. Some of the issues relating to micronets as targets for parallelising compilers have been discussed.

The control interfaces for the micronet-based datapaths are specified using a library of interconnection cells, and automatically synthesised in terms of simple C-elements. Results from SPICE simulations for an add ALU operation have been presented which demonstrates the feasibility of distributing controls.

In conclusion, the micronet model considers the interactions between the underlying implementation technology, the architecture and the compiler, and underlines our integrated approach to system design.

Acknowledgements

V. Rebello and R. Mullins were supported by postgraduate studentships from the U. K. Engineering and Physical Sciences Research Council (EPSRC). 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. Instruction-level parallelism in asynchronous processor architectures. In M. Moonen and F. Catthoor, editors, *Proceedings of the 3rd International Workshop on Algorithms and Parallel VLSI Architectures*, pages 203–215, Leuven, Belgium, August 1994. Elsevier Science Publishers.

[2] 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.

[3] K. Diefendorff and M. Allen. Organisation of the Motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, 12(2):40–63, April 1992.

[4] D. W. Dobberpuhl *et al.* A 200-MHz 64-bit dual issue CMOS processor. *IEEE Journal of Solid-State Circuits*, 27(11):1555–1567, November 1992.

[5] S. B. Furber. Lessons from AMULET1: Towards AMULET2. In *Computing Without Clocks: Asynchronous Microprocessor Design*. The Sun Annual Lecture in Computer Science at the University of Manchester, September 1994.

[6] 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.

[7] J.-L. Gaudiot and L. Bic. *Advanced Topics in Dataflow Computing*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1991.

[8] G. Gopalakrishnan. Some unusual micropipeline circuits. Technical Report UUCS-93-015, Department of Computer Science, University of Utah, Salt Lake City, UT, USA, December 1993.

[9] J. Hennessy, N. Jouppi, F. Baskett, and J. Gill. MIPS: A VLSI processor architecture. In *The Proceedings of the CMU Conference on VLSI Systems and Computations*, Rockville, Md. USA., October 1981. Computer Science Press.

[10] 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.

[11] M. G. Katevenis, R. W. Sherbourne, D. A. Patterson, and C. H. Séquin. The RISC II microarchitecture. In F. Anceau and E. J. Aas, editors, *The Proceedings of VLSI'83: VLSI Design of Digital Systems*, pages 349–359. North-Holland, 1983.

[12] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17(1):6–22, January 1984.

[13] Å. Lunde. Empirical evaluation of some features of instruction set processor architectures. *Communications of the ACM*, 20(3):143–153, March 1977.

[14] S. Mirapuri, M. Woodacre, and N. Vasseghi. The MIPS R4000 processor. *IEEE Micro*, pages 10–22, April 1992.

[15] Y.-J. Oyang, C.-H. Wen, Y.-F. Chen, and S.-M. Lin. The effects of employing advanced branching mechanisms in superscalar architectures. *ACM Computer Architecture News*, 18(4):35–51, December 1990.

[16] D. A. Patterson and C. H. Séquin. RISC I: A reduced instruction set VLSI computer. In *The Proceedings of the 8th International Symposium on Computer Architecture*, pages 443–457, May 1981.

[17] G. Radin. The 801 minicomputer. In *The Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 39–47, March 1982.

[18] 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.

[19] C. L. Seitz. System Timing. In C. Mead and L. Conway, editors, *Introduction to VLSI Systems*, chapter 7, pages 218–262. Addison-Wesley, 1980.

[20] J. E. Smith. A study of branch prediction strategies. In *The Proceedings of the 8th International Symposium on Computer Architecture*, pages 135–148, May 1981.

[21] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, May 1988.

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

[23] N. Ullah and M. Holle. The MC88110 implementation of precise exceptions in a superscalar architecture. *ACM Computer Architecture News*, 21(1):15–25, March 1993.