A Fully Asynchronous Superscalar Architecture

D. K. Arvind and Robert D. Mullins
Division of Informatics
The University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ, Scotland.
dka,rdm@dcs.ed.ac.uk

Abstract

An asynchronous superscalar architecture is presented based on a novel architectural feature called instruction compounding. *This enables efficient dynamic scheduling and forwarding of data based on local information, while maintaining the advantages of asynchrony in terms of exploiting actual delays. Results are presented in which statically and dynamically compounded architectures are compared against an equivalent synchronous superscalar architecture.*

1. Introduction

The design of high clock frequency processors leads to considerable physical problems in distributing the clock signal, high power dissipation and poor electromagnetic (EM) interference characteristics. The asynchronous design approach has been proposed as a solution to these problems [8], although the potential of multiple issue asynchronous architectures has not yet been fully explored. This paper introduces a technique called instruction compounding which better enables the advantages of asynchrony to be exploited in a superscalar architecture.

2. Synchronous Superscalar Architecture

This section highlights some features of a typical synchronous superscalar pipeline (see Figure 1) with out-of-order instruction issue. The pipeline is capable of fetching and executing multiple instructions on each clock cycle, and is typically supported by branch prediction and speculative execution in order to maintain a high instruction bandwidth.

The *instruction-issue buffer* implements, in essence, a limited dataflow capability, in holding instructions while their operands are being generated, and allowing ready instructions to issue out-of-order. The buffer may issue multiple instructions in a clock cycle to a number of functional units which operate concurrently. The operation of the instruction issue buffer can be split into two phases: *wakeup* and *selection*. The wakeup logic matches results generated by the functional units to the operands in the issue buffer; the selection logic determines which of the ready instructions should be issued to free functional units. These architectures may issue dependent instructions in consecutive clock cycles by waking instructions in the same cycle as their final operand is being produced. A network of result buses and bypass logic is used to obtain the correct operand values on the subsequent clock cycle, which is commonly termed as data forwarding.

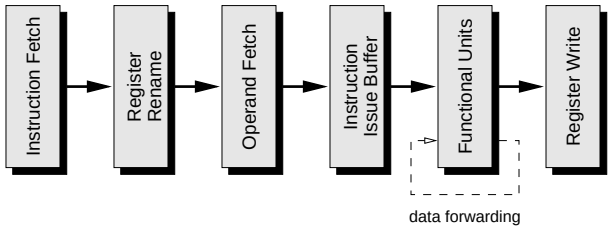

Figure 1. Synchronous Superscalar Pipeline

3. Asynchronous Superscalar Architecture

A number of synchronous implementations of the architectural features described previously already exist. Unfortunately imitating these designs within an asynchronous environment limits the extent to which the advantages of asynchrony may be exploited. To appreciate this statement we need to understand better the influence of the control paradigm on the architecture.

In synchronous architectures, the control mechanism has a rigid, periodic interaction with the datapath. Operations are initiated by the control unit and must complete within fixed multiples of clock cycles. This produces predictable and deterministic behaviour which may be exploited.

However the components of such a system must be designed to minimise the critical path to ensure a low clock period, even if this path is rarely taken. As a result, functional components lie idle for a proportion of the clock period, even though utilisation is high when measured in clock cycles. This is essentially a time-driven approach to the design of the interface between the control and the datapath. In contrast, one can implement an event-driven version of this interface using asynchronous circuits. This exposes actual delays within the datapath and results in components being active only when performing useful computations. A good asynchronous architecture is one which translates these local timing benefits to a better overall system performance. One way in which this may be achieved is by exploiting greater sub-instruction parallelism.

In synchronous implementations, both the instruction buffer and data forwarding mechanisms exploit global synchronisation. In the absence of a clock, a naive implementation would require a large number of local synchronisations - swamping any gains of exposing actual delays. We propose novel architectural ideas for efficiently realising dynamic scheduling and data forwarding in a fully asynchronous environment.

3.1. Novel ideas for instruction execution

In this section, we describe the design of an asynchronous superscalar processor, with emphasis on its out-of-order instruction execution and data-forwarding capabilities.

The basic pipeline, outlined in Figure 2, differs from the synchronous one described previously in the way that operands are obtained and instructions are scheduled. These operations are now distributed to execution units associated with each functional unit.

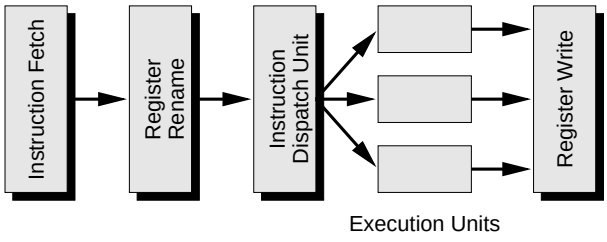

Figure 2. Asynchronous Superscalar Pipeline

The efficient operation of these execution units relies on information being obtained from the compiler in identifying possible candidates for data forwarding. The mechanism used to provide such information is called *instruction compounding* [1]. Instruction compounds provide additional information regarding the dependences between instructions. This information is used to reduce synchronisations between functional units when required to perform data forwarding and dynamic scheduling. A compound can be simply defined as a group of dependent instructions. A more precise definition with respect to the architecture is given below.

A basic block is partitioned into compounds by grouping adjacent dependent instructions. The only constraint in the selection of compounds is that the resulting graph of compounds must be a DAG. Within the compounding architecture results may only be forwarded between successive instructions within a compound. The example in Figure 5 illustrates a possible compounding for the code fragment. Instructions 2,3 and 4 are grouped together to form a compound, each instruction within the compound must be scheduled consecutively as shown. This allows membership of a particular compound to be indicated by a single *compounding bit* for each instruction. When the bit is set the instruction and the following instruction are both part of the same compound.

The architecture of an execution unit is shown in Figure 3. Instructions are issued out-of-order (and asynchronously) from the instruction buffer as soon as it is safe to do so. This is indicated by forwarding requests from other execution units, or the setting of future bits as other instructions issue. Once an instruction is ready and has successfully arbitrated for issue then its operands are obtained and its result is generated. Concurrently, a pipeline determines whether the result is to be forwarded, both finally converge in the forwarding unit from where data is actually forwarded. A more detailed description of the operation of these units follows.

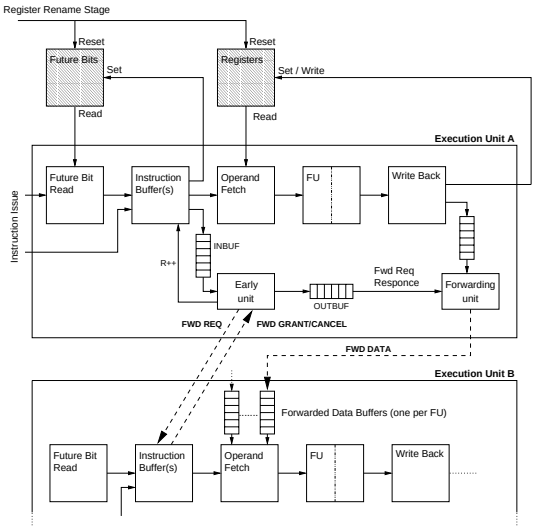

Figure 3. Execution Unit

Execution units receive results either via the shared re-

gister bank or directly from other units over the forwarding network.

In the absence of global synchronisation, communication via the register bank is implemented through the use of a register locking mechanism [6]. A status bit is attached to each physical register to indicate when its contents is valid. In addition, a *future bit* is associated with each register to indicate whether the instruction which will write to the register has been issued. Future bits guarantee the availability of results and are used to determine when an instruction may issue safely, without resulting in a deadlock. Both register status and future bits are reset during register renaming when a new physical register is mapped.

Once an instruction is dispatched to an execution unit, each of its operands which cannot be forwarded must read its register future bit. This is achieved by queuing each read operation in one of two read queues. After a future bit is read, the status of the corresponding operand in the instruction buffer is updated. This write is made using an instruction buffer write port. This operation is illustrated in Figure 4.

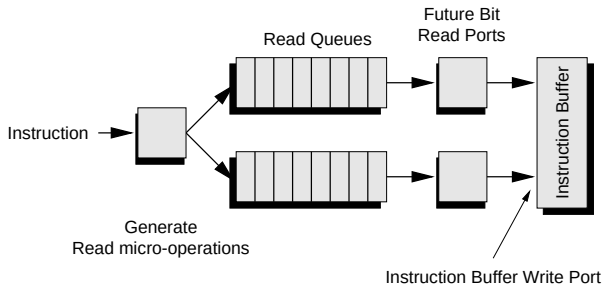

Figure 4. Future Bit Read Operation

The instruction buffer dispatches instructions out-of-order depending on the status of their operands. The operand status is updated via the instruction buffer write ports, either by successfully reading a future bit (as described before), or by receiving a forwarding request (to be described). These operations are equivalent to an instruction *wakeup* phase in a synchronous pipeline. Each write port is associated with an issue port. If a write wakes an instruction in the buffer, then the write port is blocked until the issue request is granted. This limits the number of arbitrating instructions to the number of write ports, which is desirable in asynchronous architectures (due to the delay of multi-way arbiters). Each write is made directly to a particular buffer entry - this is possible as both forwarding requests and future bit reads are tagged with the instruction's buffer entry.

Each entry in the buffer contains information about a particular instruction's operands, their status and forwarding bits, the operation to be performed at the functional unit, the instruction's compounding bit and the location of the next instruction in the compound. The forwarding bits associated with each operand indicate whether the result will be forwarded, or fetched from the register bank. These bits are initially set in the instruction dispatch unit.

Once an instruction has issued, it proceeds to the operand fetch stage, and should its compounding bit be set (it forwards its result), then it is also sent to the early unit input buffer. The future bit associated with its destination register will be set to indicate that the result is being generated.

The early unit queries the next instruction in a compound to determine if forwarding is possible. This query or *forwarding request* is also used to update the status of the operand. A detailed description of this operation is given below.

- The early unit receives each instruction which is a member of an instruction compound (bar the final instruction) and makes a forwarding request to the next instruction in the compound. The location (execution unit and buffer entry) of this instruction is obtained within the instruction dispatch unit.
- The forwarding request must arbitrate for access to the instruction buffer. Forwarding requests are then queued before they access a particular entry via a write port.
- When a forwarding request is made to a particular instruction in the buffer, then one of two situations will arise:
  - The status of all other operands has been updated through future bit reads. In this case, data forwarding is possible and the instruction may issue.
  - Future bit reads are pending for one or more operands. In this case it is not possible to issue the instruction and data forwarding must be cancelled. The operand which would have been forwarded is now obtained from the register bank, and its forwarding bit is reset to reflect this.
- The early unit will receive either an acknowledgement or cancellation signal. This information is used to determine whether or not to forward the data at the forwarding unit.

The order in which results are consumed from a particular execution unit must be guaranteed to be the same as the order in which they are sent. This is only possible by cancelling the forwarding of certain results. The alternative of issuing an instruction whenever it receives a forwarding request is not possible without introducing the possibility of deadlock.

Another potential deadlock condition involving the early unit is controlled by the release of instructions from the instruction buffer. Instructions are only released when there is no possibility of filling the early unit input buffer. The $R{+}{+}$

signal in Figure 3 is used to maintain a count within the instruction buffer and implement such a mechanism. If the queue was to block instruction issue, then deadlock could occur.

Operand fetch obtains register and forwarded result data. Forwarded results are received into an individual queue for each sender. This is necessary as the order in which forwarded results are sent is only guaranteed with respect to a single execution unit. In both the cases of register operands and forwarded results, operand fetch will stall until the data is available.

3.2. A Simple Example

In this section we illustrate the operation of the datapath through a simple example (see Figure 5) of forwarding and dynamic scheduling.

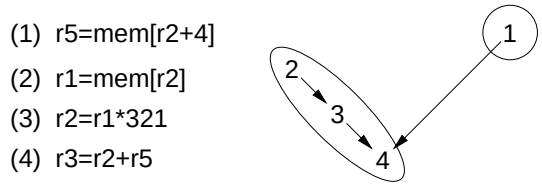

Figure 5. Example Compounds

Instructions 2,3 and 4 are compounded, while instruction 1 remains a singleton compound. Alternatively, compounds (2,3) and (1,4) could have been created. For simplicity, we assume in this example that the logical and physical registers used for each instruction have the same identifiers.

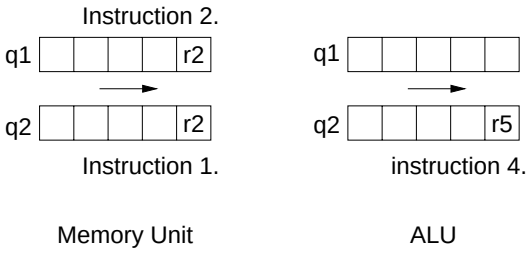

Figure 6. Future bit Read Queues

The following description shows how instructions are issued and obtain forwarded data.

- All instructions are dispatched to their respective execution units. In this case, a single memory unit (for instructions 1 and 2) and ALU (for instructions 3 and 4) are present.
- Future bit read operations are queued for all register operands (see Figure 6). Communication between instructions 2,3 and 4 are handled by forwarding operations and do not require future bit reads.
- We now concentrate on the execution of instructions 2 and 3. Instruction 3 requires no future bit read and only awaits a forwarding request from instruction 2. Instruction 2 issues after its operand's status bit is updated upon completion of the future bit read for register 2 (from q1 at the memory unit).
- The instruction proceeds to both the operand fetch stage and the early unit. The latter makes a forwarding request to instruction 3. Causing it to generate a forwarding acknowledge signal and to issue. Forwarding cannot be cancelled in this case as the instruction has no register operands.
- Once the result for instruction 2 is generated, the *forwarding unit* will receive both a result and forward request response - in this case an acknowledgement. The result will then be sent to the ALU's memory unit result queue, where instruction 3 will obtain the result during its operand fetch stage.

3.3. Dynamic Compounding

In the architecture presented so far instruction compounds are identified at compile-time. An alternative approach is to construct the compounds dynamically as instructions are read. This section describes an implementation of dynamic compounding, which extends compounding beyond basic block boundaries.

The implementation is based on a table being maintained within the register renaming, or issue stages of the datapath. An entry exists for each physical register and contains the following information:

- A *forward bit* to indicate that this result is to be forwarded. A destination in the form of a functional unit and instruction buffer entry is also present if the compounded bit is set.
- An *executed flag*, which is set once the instruction generating the result for this entry's register has queried the table.

An entry in the forwarding table is cleared when an instruction obtains its physical register destination. A subsequent read of this register may then be forwarded. This requires the compounded bit to be set in the table and the location of the instruction requiring the result to be recorded. A result may only be forwarded once, as in the static case, and only while the *executed flag* is clear. This flag is set when the instruction producing the result queries the table to see if the result is to be forwarded. This query takes place in an extra stage prior to the early unit. The details of the implementation have been omitted, as it is only used to explore the limits of compounding in this context.

4. Evaluation

We compare asynchronous architectures operating with statically and dynamically generated compounds to a synchronous superscalar machine. We also include results for a queue-based asynchronous architecture, which offers limited dynamic scheduling but lacks data forwarding.

All functional units share the same architectural parameters, as described (see Table 1). The delays used within the asynchronous architectures, as listed in Table 2, these are expressed as a percentage of the synchronous architecture's clock period.

Parameter	Number
No. of instrs. fetched per memory cycle	4
Complex ALU (ALU, logic, shift., mult.)	1
ALU (ALU, logic)	2
Memory Unit	1
Logical Registers	32
Physical Registers	64
Instruction buffers per Functional Unit	16

Table 1. Architectural Parameters

Component	Delay (% Cycle time)
Memory Access	100
Register Access	100
Future Bit read/write	60
Instruction Buffer Issue	50
Instruction Buffer Write	30
FU to FU communication	
Request (requires arbitration)	45
Acknowledge	30
Data	30
Instruction Delays	
ALU (add/shift)	50
Logical	20
Set/Move/Clear	0
Load/Store	100

Table 2. Asynchronous Component Delays

The following list gives additional implementation details specific to each model.

- The queue-based asynchronous architecture simply issues instructions to execution units consisting of an instruction queue, operand fetch stage and functional unit pipeline.
- The synchronous machine's instruction buffer is distributed amongst the functional units.
- In the case of the compounding architecture, each instruction buffer is split in two. One buffer is used to hold instructions which may receive forwarded data, and the other for those which will not. Two future bit read queues and ports are shared between each buffer, within each execution unit. Read operations were assigned in a round-robin fashion to the queues.
- The forwarding table (for dynamic compounding) incurs no delay due to reading, writing or arbitration. In this case, dynamic compounding is simply used to explore the possible advantages of extending compiler based compounding beyond basic blocks.

Results where obtained using a trace-driven, event-based simulator. The benchmarks used are *cjpeg* (spec95), *bubble sort*, *queens*, *compress*(spec92), *xlisp* (spec92) and *fgrep*. Instruction compounds were selected using a greedy graph partitioning algorithm with a maximum compound length of 10. No optimisations were performed on the schedule of compounded instructions, or for the queue-based asynchronous model.

Results showing the IPC (a cycle is defined in terms of a memory access operation for all the models) for each processor model are presented in Figure 7. Perfect branch prediction, memory disambiguation and instruction fetch bandwidth are assumed.

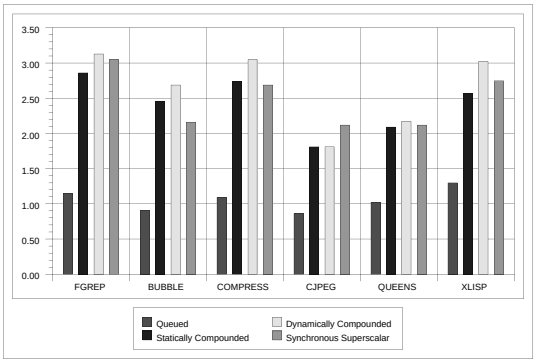

Figure 7. IPC for different processor models

The percentage of operands which were compounded either statically or dynamically and the actual percentage of operands obtained via forwarding are given in Figure 8. These differ due to the need to cancel some forwarding operations to avoid deadlock at run-time.

It can be seen from the results that the synchronous processor only outperforms the dynamically compounded model in one case (*cjpeg*). Static compounding performs worse than dynamic in all cases, only outperforming the synchronous model in the case of *bubble sort* and *compress*.

These preliminary results are encouraging, and they will improve with compiler optimised static compounding. We

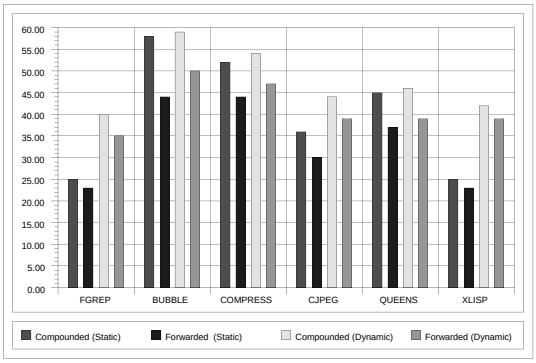

Figure 8. Percentage of operands compounded and results actually forwarded

are currently investigating a combined SW/HW approach to support forwarding across basic blocks.

Techniques also exist for exposing greater fine grained parallelism. For example, higher utilisation of the register read ports may be possible if the operand fetch stage is redesigned to permit each port to be accessed independently. This would allow both ports to be used if two instructions only require a single register fetch each. The overhead involved in implementing such aggressive techniques to expose further parallelism is current being evaluated.

5. Related Work

The effect of asynchrony on processor architecture has been explored in earlier work [3, 2], which introduced the notion of a fine-grained network of asynchronous agents called a *micronet*. Although this work was limited to scalar architectures many of the ideas and techniques for distributing control have been applied here.

Notable asynchronous processor implementations include an asynchronous MIPS R3000 [5] processor and the Amulet 2 [7], an asynchronous implementation of the ARM processor. Each makes some attempt to implement data forwarding, such as register bypassing in the case of the R3000 at the register bank, and by implementing last use registers in the Amulet 2 processor. A result forwarding mechanism designed for inclusion in the latest Amulet processor is presented in [4]. Here a small parallel FIFO is used to forward results between instructions currently in the pipeline. Each of these techniques have been developed for use within a scalar processor and their application to dynamically scheduled superscalar machines is limited. One reason for this is the large number of outstanding instructions and possible forwarding situations.

6. Conclusions

We have presented a novel architecture for exploiting asynchrony in superscalar architectures. To our knowledge this is the first detailed study into the performance advantages of an asynchronous multiple issue architecture.

We achieve better performance by two means: reducing run-time synchronisation and by exploiting fine-grained parallelism. Two techniques are used to achieve these aims. Firstly, instruction compounding reduces run-time synchronisations by generating forwarding information at compile time. Secondly, the early unit and future bits expose additional parallelism by allowing events to occur as early as possible while avoiding deadlock.

By understanding the interplay between compilers and architectures we aim to realise fully the performance potential of asynchronous multiple issue architectures.

References

[1] D. K. Arvind and R. D. Mullins. Instruction compounding. In *Proceedings of the 1st UK Asynchronous Forum*, Edinburgh, Scotland, Dec. 1996.

[2] D. K. Arvind, R. D. Mullins, and V. E. F. Rebello. Micronets: A model for decentralising control in asynchronous processor architectures. In *Asynchronous Design Methodologies*, pages 190–199. IEEE Computer Society Press, May 1995.

[3] D. K. Arvind and V. E. F. Rebello. Instruction-level parallelism in asynchronous processor architectures. In M. Moonen and F. Catthoor, editors, *Proc. of the Third Int. Workshop on Algorithms and Parallel VLSI Architectures*, pages 203–215. Elsevier Science Publishers, Aug. 1994.

[4] D.A.Gilbert and J.D.Garside. A result forwarding mechanism for asynchronous pipelined systems. In *Async '97*, pages 2–11. IEEE Computer Society Press, Apr. 1997.

[5] A. J. Martin, A. Lines, R. Manohar, M. Nystroem, P. Penzes, R. Southworth, and U. Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI*, pages 164–181, Sept. 1997.

[6] N. C. Paver, P. Day, S. B. Furber, J. D. Garside, and J. V. Woods. Register locking in an asynchronous microprocessor. In *Proc. International Conf. Computer Design (ICCD)*, pages 351–355. IEEE Computer Society Press, Oct. 1992.

[7] S.B.Furber, J.D.Garside, S.Temple, J.Liu, P.Day, and N.C.Paver. AMULET2e: An Asynchronous Embedded Controller. In *Async '97*, pages 290–299. IEEE Computer Society Press, Apr. 1997.

[8] C. H. K. van Berkel, M. B. Josephs, and S. M. Nowick. Scanning the technology: Applications of asynchronous circuits. *Proceedings of the IEEE*, 87(2):223–233, Feb. 1999.