Computer Science

Technical Report

REGISTER BYPASSING IN AN ASYNCHRONOUS SUPERSCALAR PROCESSOR

S J Davis, C J Elston, P A Findlay

Report No: 328

March 99

Register Bypassing in an Asynchronous Superscalar Processor

S. J. Davis[1], C. J. Elston[2], P. A. Findlay[3]

Abstract

Register bypassing, universally provided in synchronous processors, is more difficult to implement in an asynchronous design. Asynchronous bypassing requires synchronization between the forwarding and receiving units, with the danger that the advantages of asynchronous operation may be nullified by reintroducing the lock-step operation of synchronous processors. We present a novel implementation of register bypassing in an asynchronous processor architecture. Our technique of Decoupled Operand Forwarding provides centralized control over the bypassing operation, yet allows multiple execution units to function asynchronously. Our ideas are presented within the context of the development of Hades, a generic asynchronous processor architecture. We employ single-issue and dual-issue simulations of Hades to quantify the benefits of Decoupled Operand Forwarding and conclude that Decoupled Operand Forwarding yields significant speedups because of its success in removing register files from the critical timing path.

1. Introduction

In recent years there has been renewed interest in the asynchronous model of processor design. Asynchronous operation offers a number of potential advantages, including: low power, typical instead of worst case execution time, and the avoidance of clock skew through local communication. This latter advantage of asynchronous processors leads to the view that they will scale with technology far better than synchronous processors. Charles Seitz argues along these lines convincingly in [1], where he shows that as the scale of integration increases, the problems of clock skew will increase because propagation delays remain constant. Hence, at some time in the future it is likely to become too costly and difficult to increase the performance of synchronous processors by simply increasing the frequency of the clock. Asynchronous processors have no such asymptotic performance limitation and so in the medium term are likely to become an increasingly attractive proposition.

However, fundamental differences between synchronous and asynchronous operation make it undesirable to re-map a synchronous processor onto an asynchronous control framework. Designing asynchronous processors involves new problems that require novel solutions. In particular, operand forwarding or bypassing is recognized as problematic within asynchronous processor design. In Section 2, we give brief details of a number of recent asynchronous processor designs. Our own project [2] concentrated on the performance potential of asynchronous processors and developed Hades (Hatfield Asynchronous DESign) a generic asynchronous processor design. We present an overview of the Hades architecture in Section 3, including details of the asynchronous communication mechanisms investigated and the register file organization. We then reconsider operand forwarding in the light of asynchronous operation, in order to develop a scheme that will work harmoniously and efficiently within an asynchronous processor. In Section 4, we present our technique of Decoupled Operand Forwarding.

In Sections 5 and 6, we present simulation results based on the Stanford Integer Benchmark Suite. A single-instruction-issue (SII) version of Hades is simulated and used to assess the effectiveness of Decoupled Operand Forwarding. Hades is then extended to provide dual-instruction-issue. Comparisons are made with the SII versions and the effectiveness of Decoupled Operand Forwarding is assessed in a superscalar environment. Finally, the performance of various MII configurations is evaluated.

---

[1] Corresponding author. *Department of Computer Science, University of Hertfordshire, College Lane, Hatfield, Hertfordshire, AL10 9AB, England.* Fax: (01707) 284303. Email: S.J.Davis@herts.ac.uk
[2] Now at: [Details to follow].
[3] *ERDC, University of Hertfordshire, College Lane, Hatfield, Hertfordshire, AL10 9AB, England.* Email: P.A.Findlay@herts.ac.uk

2. Related Work

We present a brief overview of related work on asynchronous processors which has resulted in the development of novel asynchronous architectures.

The AMULET Group at the University of Manchester has given a major impetus to asynchronous processor design by demonstrating the feasibility of complex asynchronous processors. AMULET1 is based on the Acorn ARM architecture [3] and Sutherland's Micropipeline Methodology [4]. Limitations to the performance of AMULET1 include the lack of a bypassing mechanism, the length of time taken to resolve a branch, and the slow speed of the building blocks employed by the Micropipeline Methodology. These limitations appear to have been overcome in AMULET2 [5].

The CounterFlow Pipeline Processor (CFPP) architecture is the product of work by a team from the Sun Microsystems Laboratories [6]. It consists of a bi-directional pipeline, with instructions and data flowing in opposite directions. Instructions and data interact in each stage of the pipeline. If an instruction encounters a register value that it requires for execution, it copies that data. Once an instruction has obtained all the operands that it needs, it is able to execute in the next appropriate stage. Both instructions and data interact with the contents of each stage through which they pass. An instruction will encounter the results of all earlier instructions before the register operands that it has requested, which obviates the need for traditional operand forwarding. The drawback of this scheme is that an asynchronous arbiter is required to handle the synchronization that allows instructions and data to counterflow. Also, it is not clear how the counterflow pipeline can be generalized to multiple pipelines.

The Rotary Pipeline Processor Project, of the Computer Laboratory at Cambridge University, has resulted in the design and simulation of a generic, superscalar architecture that is particularly suited to asynchronous operation [7]. This processor consists of execution units arranged in a circular structure, with the inputs of each execution unit connected to the outputs from the preceding unit. The instruction fetch and decode mechanism, which delivers each instruction to an appropriate functional unit, is situated outside the ring. Functional units that require operands to execute an instruction, inspect the data circulating the ring, which includes register values. The data is then passed on, possibly after modification. Thus, synchronization is only required in one direction, overcoming the synchronization problems encountered by CFPP.

The SCALP architecture was developed at the University of Manchester to investigate the low power potential of MII asynchronous architectures [8]. Based on the idea of a Transport-Triggered Architecture [9] [10], it has multiple functional units that operate on data in FIFO queues that are visible to the compiler. The contribution of SCALP to asynchronous processor design is its simple model of complex superscalar execution.

3. Architectural Overview of Hades

Hades is a generic asynchronous processor architecture that was developed to explore the performance potential of asynchronous operation. Hades has a RISC instruction set comprising a small number of simple instructions. A generalized overview of the Hades architecture is shown in Fig. 1. Two register files are included, one for integer operands and one for Boolean values. The Boolean Register File stores conditions generated by integer comparisons that are used to resolve branches. The provision of multiple Boolean registers in a separate register file allows a number of conditions to be computed in advance of the instructions that will use them and reduces the number of accesses to the Integer Register File.

The Hades pipeline has four asynchronous *modules*, which have been named by analogy with the stages of a synchronous processor pipeline: Instruction Fetch (IF), Instruction Decode (ID), Execution (EX) and Writeback (WB). Separate instruction and data caches are provided to reduce memory contention. The pipeline can be instantiated for either SII or MII. All instantiations issue instructions in-order, but allow out-of-order instruction completion. The IF module supplies instructions to the pipeline; multiple instructions are fetched in multiple issue models.

The task of the ID module is twofold. Firstly, the ID module decodes instructions for the EX module. This includes providing control information, allocating resources and sign extending immediate data. Secondly, the ID module initiates register file accesses including those for subroutine return addresses. Initiating all register reads in the ID module avoids any need for arbitration before accessing the register file.

While non-branch instructions trigger a straightforward sequential fetch, branch instructions trigger control flow resolution. The ID module issues the branch instruction to the Address Generator and initiates a read for the branch condition from the Boolean Register File. Hades implements a delayed branch mechanism, but the number of sequential instructions that are executed after each branch instruction is not fixed by the architecture. Instead, this number is encoded directly into the branch instruction as a delay count. The Address Generator reacts initially by fetching the number of sequential instructions indicated by the delay count for this particular branch. The Address Generator then uses the Boolean value obtained from the Boolean Register File to determine the address of the next instruction to fetch.

The EX module performs the operations specified by the instructions. It is composed of a number of Execution Units or functional units each of which executes a subset of the instruction set. The minimum set of Execution Units consists of an Arithmetic, a Shift, a Multiply, a Relational and a Memory Unit. The operation of Execution Units depends only on the supply of instructions and data. Each Execution Unit can therefore process a separate instruction in parallel. Relational operations are carried out in a separate Relational Unit. This arrangement decouples the Execution Units that produce Boolean results from those that produce Integer results, so each Execution Unit is less complex [11]. Additionally, unlike the Arithmetic Units, the Relational Units do not take up result or bypass bus bandwidth.

The WB module updates the Integer Register File with results produced in the Execution Units. Results are routed to the Integer Register File across one or more shared result buses that require arbitration for exclusive access. However, because Boolean results are only one-bit wide, dedicated connections are provided between Relational Units and the Boolean Register File.

The base model just described was extended using a variety of techniques, including Decoupled Operand Forwarding which is the main topic of this paper.

3.1 Optimized Communication

In asynchronous systems, synchronization is achieved explicitly through the exchange of signals, rather than implicitly via the global clock. This communications overhead is one disadvantage commonly associated with asynchronous processors. Hades attempts to reduce this overhead by employing an optimized form of asynchronous communication.

Hades uses a four-phase, bounded delay protocol. This option offers the best performance potential, the lowest silicon area overhead and the ability to use the absolute state of control lines. However, because control and data are separate, only half of the events in the protocol are necessary for communication, the remaining events simply return the REQ and ACK signals to their initial state. When one communication initiates another the situation shown in Fig. 2 therefore arises; four phases of the first communication are followed by the four phases of the second even though the data transfer is complete after the rising portion of the protocol. Such a situation arises many times in the operation of a processor, for example, a buffer inputs a datum and then outputs it, or an Arithmetic Execution Unit inputs an instruction and then inputs the required operands.

Performance can be improved by overlapping the data transfer and recovery portions of communication (Fig. 3). The falling edges of the first communication are effectively overlapped with the rising edges of the second. This optimization significantly reduces the latency of asynchronous communication by overlapping the various stages of the protocol with other operations.

3.2 Asynchronous and Optimized Register Files

The Hades register file organization allows concurrent read and write accesses whenever possible. Correct operation is ensured in the presence of Read-After-Write (RAW) and Write-After-Write (WAW) hazards by a register locking mechanism. Similar mechanisms have been used in other asynchronous projects [12] [13].

All read accesses are initiated by the ID module which sends the source and destination register fields of an instruction to the register file. A lock is associated with each register and is set, concurrently with the read access for source operands, on the register that is specified as the destination for the instruction. Once set, this lock stalls read and subsequent lock accesses to this particular register. The lock is reset when data is written to the register. Register locking resolves RAW hazards by stalling reads until the data is available and WAW hazards by stalling lock accesses to registers that are already locked. This relatively simple scheme for dealing with WAW hazards was considered sufficient because of the low frequency of WAW hazards in typical code [14].

In a synchronous processor, a major justification for bypassing is that if data is not available from the register file at the beginning of a clock cycle, a read must be stalled for a complete clock cycle. In contrast, in an asynchronous processor an instruction need only be stalled until the data is actually available from the register file. It therefore seemed reasonable to explore the impact of register file optimizations on versions of Hades that did not support bypassing. In particular, we attempted to reduce the impact of RAW hazards by minimizing the time taken for data to be communicated between instructions via the Integer and Boolean Register Files. The critical path was reduced by overlapping the resetting of the destination register lock with the completion of the protocol mediating the write to the destination register. Furthermore, if a read encountered a lock, a path through the register file was preset to allow arriving data to pass straight through and to remove the delay of resetting the lock.

The use of an Optimized Register File is intended to tune the design so that register file access is less of a performance bottleneck. Optimized Register Files have a very low overhead in terms of additional hardware requirements. It is therefore natural to combine asynchronous operation with an Optimized Register File to reduce the performance-sapping effects of RAW hazards in a pipelined environment.

4. Decoupled Operand Forwarding

Bypassing is one of the most troublesome aspects of asynchronous processor design because the synchronization implicitly available in a synchronous pipeline is not present in an asynchronous pipeline. Furthermore, introducing the same level of synchronization in an asynchronous pipeline would dramatically degrade performance by re-introducing lockstep operation. It is therefore extremely difficult to design an effective bypassing mechanism that does not compromise performance in some manner. The intractability of the problem has led some projects to alter the structure of the pipeline drastically, in order to eliminate the need for bypassing, or to allow radically different forms of bypassing [3] [7].

Hades provides an unconventional bypassing mechanism called Decoupled Operand Forwarding. Although a register locking mechanism has the appeal of simplicity, RAW hazards occur so frequently that simply stalling instruction issue severely degrades performance [15]. An Optimized Register File will reduce the performance penalty but not remove it. Decoupled Operand Forwarding aims to reduce the frequency of stalls, by removing the register file from the critical path when two dependent instructions are executed. WAW hazards continue to be detected using the register file lock bits, since an instruction must still not be issued to an Execution Unit if the destination register lock is set. Decoupled Operand Forwarding is a fully general scheme that works for any combination of Execution Units in both SII and MII environments. The central idea is to avoid performance-sapping high-level synchronization by separating the forwarding from other pipeline operations.

4.1 Centralized Control

Decoupled Operand Forwarding can be viewed as a form of distributed register caching in which the most recently generated results are retained locally within the Execution Units. The functionality required by Decoupled Operand Forwarding comprises Forwarding Registers (FRs) and the associated data paths (Fig. 1), together with Forwarding Tags in the ID module. Each Execution Unit has an individual FR that holds a copy of the most recent result produced by that unit. These results are forwarded to subsequent instructions under the explicit control of the ID module.

Instructions are allocated to Execution Units by the ID module which can therefore track the contents of the FRs. The ID Unit maintains a Forwarding Tag for each FR that uniquely identifies the logical contents of the corresponding FR. When an instruction is issued, the appropriate Forwarding Tag is updated and an Overwrite signal is sent to the relevant FR, which invalidates the existing data and allows the register to be updated with the next result generated by its associated Execution Unit.

During ID, each instruction compares its source register fields with all the Forwarding Tags. If there is no match, the source operand is obtained from the register file. If a match does occur, Decoupled Operand Forwarding is initiated by sending a Forward Request signal to the appropriate FR, which then makes data available for input by an Execution Unit. Only valid contents will be output by the FR; once the register has received an Overwrite signal, it will wait until a valid result has been loaded from its Execution Unit before forwarding it.

Communication between the ID module, which provides central control, and each FR is sequential. There is only one channel for both the Overwrite signal and the Forward Request signal. In general, an instruction may receive a forwarded result from a particular FR and also use the same FR to hold its own result. In this case, a Forward Request signal initiates bypassing and a subsequent Overwrite signal allows the FR to load the result from the instruction. The single communication channel ensures that these events occur in the correct sequence, that is, they are synchronized. At the same time, the inherent delay through the Execution Unit will ensure that the execution of an instruction is not held up by the Overwrite signal. A more detailed discussion of Decoupled Operand Forwarding may be found in [2].

4.2 A Decoupled Operand Forwarding Example

The Decoupled Operand Forwarding example (Fig. 4) illustrates the role of the ID module in tracking the logical contents of the FRs and in controlling their operation with the Overwrite and Forward Request signals. In the diagram, thick lines denote data paths and thin lines denote communication channels. The following sequence of instructions is being executed in the example.

```
ADD   R1,R2,R3
SUB   R4,R3,R1
ADD   R6,R5,R4
```

In Fig. 4(a), the ID module is decoding the ADD R1,R2,R3 instruction. The instruction is about to be allocated to EU1, so FT1 is set to indicate that FR1 will receive a copy of the result destined for R1. An Overwrite signal has also been sent to FR1. That this has been received, is indicated by 'OVERWRITE = Y'.

In Fig. 4(b), EU1 is executing the ADD R1,R2,R3 instruction and will pass the result to FR1. Meanwhile, the ID module has decoded the next instruction, SUB R4,R3,R1, and, by comparing the Forwarding Tags with its source register fields, has determined that one of its source operands, R1, will be available from FR1. A Forward Request signal has therefore been sent to FR1. 'FOR. REQ. = Y' shows that this signal has been received.

In Fig. 4(c), EU1 has completed the first ADD operation and has output the result to the Result Bus and to FR1, 'DATA = ADD Result'. In response to the Forward Request, FR1 places the result on a dedicated bypassing bus, for input by the Execution Unit that is waiting for an operand. In this case, the SUB R4,R3,R1 instruction has been allocated to the same Execution Unit, EU1, so FT1 is set to indicate that FR1 will receive a copy of the result for R4. An Overwrite signal has also been sent to FR1; 'OVERWRITE = Y' shows that it has been received.

In Fig. 4(d), EU1 accepts the data placed on the bypassing bus as the source operand, 'R1', for the SUB R4,R3,R1 instruction. The ID module is decoding the following instruction, ADD R6,R5,R4 and, after checking the Forwarding Tags, sends a Forward Request signal to FR1. This has been received, as shown by 'FOR. REQ. = Y'.

In a single instruction issue processor, two dedicated forwarding buses are provided, one for each instruction source operand. When a Forwarding Register performs a forwarding operation, data is simply driven onto the appropriate forwarding bus that serves all Execution Units. If an Execution Unit is waiting for a first or second source operand, it will therefore use the next data sent across the relevant bus. As a result, the forwarded data does not require any associated address information and Execution Units do not have to examine each data item to determine its origin.

With the Decoupled Operand Forwarding bypassing mechanism, the last result produced by each Execution Unit is always saved for possible forwarding. There are three possibilities for the state of the processor when Decoupled Operand Forwarding is initiated. Firstly, the operand required is available from both the Integer Register File and from one of the FRs. In this case, the operand will be obtained directly from the FR. Since the forwarding operation is faster than a register access, performance will be improved provided that a register access is not required for a second register operand. Secondly, the operand is present in one of the FRs, but not in the Integer Register File. An attempt to read from the Integer Register File would therefore cause a stall. In this case, Decoupled Operand Forwarding provides the operand directly from the FR, thereby increasing performance. Thirdly, the operand is still being produced and is therefore not available from either the Integer Register File or from one of the FRs. Once again, Decoupled Operand Forwarding will provide the operand directly from the FR and boost performance because the operand will be forwarded as soon as it is produced.

5. SII Simulations and Results

Hades has been designed as a generic processor architecture using CSP [16]. Detailed design has been carried out on two specific versions of Hades, an SII instantiation and an MII instantiation that decodes and issues two instructions concurrently. Simulations of these designs have been constructed using VHDL [17]. To quantify performance a reliable estimate of the execution time for a benchmark program is required. VHDL simulations are composed of elements that react to various events by generating further events. A delay occurs between the original event and the generated events. These delays were estimated and introduced into the simulations so that performance could be estimated. All delays are expressed in terms of a generalized gate delay and execution times in the results section are also expressed as generalized gate delays.

The SII simulations are referred to as Hades1. Hades1_nov is the most basic, as it incorporates no optimizations at all. Hades1_ov differs only in the communication protocol employed and yet Optimized Asynchronous Communication increases performance by 59%. Because of the overwhelming performance benefits of Optimized Asynchronous Communication, it is employed by all subsequent Hades simulations and Hades1_ov is used as the base model for all subsequent comparisons. The techniques of Optimized Register Files and Decoupled Operand Forwarding were simulated separately and in combination. The various SII simulations developed are shown in Table 1.

Our results are based on simulated runs of the Stanford Integer Benchmark Suite, which comprises eight programs that are illustrative of real-world, non-numeric applications. The programs are easy to understand and manipulate, at both source and assembly levels, because the static code size is reasonably small. At the same time, many of the programs are recursive and computationally intensive and are suitable for obtaining comparative performance figures. The programs (Fig. 5) are written in C and compiled using a GNUCC compiler originally developed for HSA [18]. All the instructions generated fall within the subset supported by Hades.

5.1 Results for SII Hades

SII results are shown in Fig. 6 and summarized in Table 2. We were particularly interested in quantifying the benefits of Optimized Register Files and Decoupled Operand Forwarding. All the speedups are relative to the baseline simulation, Hades1_ov, with Optimized Asynchronous Communication.

The first comparison was between Hades1, with Optimized Register Files (Hades1_orf), and the baseline simulation. Optimized Register Files improve performance, by overlapping independent operations, and reduce the latency of passing data between instructions via the Integer or Boolean Register Files. WB has a reduced latency and the duration of stalls is reduced. The relatively modest performance increase of 4% achieved with Optimized Register Files will be considered further in relation to Decoupled Operand Forwarding.

Next Hades1 with Decoupled Operand Forwarding, Hades1_dof, was compared with the baseline simulation. Decoupled Operand Forwarding provides a significant performance increase of 22%. It would therefore appear that in an SII model, Decoupled Operand Forwarding offers an attractive and successful method of bypassing, appreciably improving performance at a reasonable cost.

In an SII environment, the performance increase due to Decoupled Operand Forwarding is over five times greater than for Optimized Register Files. Two possibilities exist for the differing degrees of success of these two schemes. Firstly, the Integer and Boolean Register Files are bottlenecks to performance that cannot be addressed successfully merely by optimization; the very nature of the operations performed dictates a minimum latency that will adversely affect performance. Secondly, delays in the path from the output of an Execution Unit to the input of an Execution Unit dominate register file delays. These path delays could include asynchronous arbitration for Integer Result Buses and the cumulative effects of several communications protocols. Based on an examination of the behaviour of Hades, it is suggested that the register file is a bottleneck to performance that can only be improved, and not solved, by optimization. Decoupled Operand Forwarding increases performance because it removes register file delays in performance critical situations.

Decoupled Operand Forwarding and Optimized Register Files are combined in Hades1_orf_dof. A comparison with the baseline simulation yields a speedup of 22%, which is the same as that for Decoupled Operand Forwarding alone. No further performance benefit has therefore been gained from combining Optimized Register Files with Decoupled Operand Forwarding. This suggests that Decoupled Operand Forwarding negates the benefits of register file optimization. In the majority of cases where a register read access would encounter a locked register, Decoupled Operand Forwarding provides the operand. Hence, the register file has been successfully removed from the critical path for dependent instructions.

6. MII Simulations and Results

Fig. 7 shows a generalized diagram of the Hades architecture, with extensions to provide for MII. Resources are duplicated to allow two instructions to be issued in parallel. The IF module is now configured to fetch instructions in groups of two and the ID module is configured to operate on and issue two instructions concurrently. The Integer and Boolean Register Files are also required to provide operands for two instructions concurrently. The Interconnection Network already provided two dedicated forwarding buses between FRs and Execution Units to implement Decoupled Operand Forwarding. Four dedicated buses must now be provided to handle the bypassing requirements of two instructions. The MII models, collectively known as Hades2, are shown in Table 3.

6.1 MII Hades versus SII Hades

To investigate the benefits of multiple instruction issue, comparisons were made between various versions of Hades1 (Table 1) and Hades2 (Table 3). Firstly, the baseline versions were compared. This comparison between Hades1_ov and Hades2_ov is shown in Fig. 8. The 32% speedup obtained, in the absence of Decoupled Operand Forwarding and Optimized Register Files (Table 4), illustrates the potential of MII to increase the performance of asynchronous processors. Since no additional Execution Units were provided in the dual issue model, the Execution Units were clearly under-utilized by the single issue model.

Secondly, a comparison was made using both Optimized Register Files and Decoupled Operand Forwarding, Hades1_orf_dof and Hades2_orf_dof. These results are shown in Fig. 9. The speedup of 11% (Table 4) again indicates that MII can be applied successfully to asynchronous processors. However, this is a substantial reduction from the 32% speedup achieved with the baseline models. Since Hades2_orf_dof has only a single Arithmetic Execution Unit and a single Integer Result Bus, these results suggest that Hades2 does not include sufficient resources to exploit the parallelism in the benchmark programs fully.

A second Arithmetic Unit and a second Result Bus were therefore added to Hades2 to give the Hades2_res_rsc model. The results for a comparison of Hades1_orf_dof and Hades2_res_rsc are shown in Fig. 9. The speedup of 21% obtained confirms that the performance of the previous MII model was constrained by the lack of Execution Units. Further improvements could be achieved through static instruction scheduling, although variable execution times make it more difficult to schedule code for an asynchronous processor than for a synchronous one.

6.2 Results for MII Hades

In this section we examine the performance of various dual-issue Hades processor models. The baseline model, Hades2_ov has only one instance of each Execution Unit and includes no optimizations apart from the standard Optimized Asynchronous Communication.

Firstly, an Optimized Register File was added to the baseline model to give Hades2_orf. Optimized Register Files improve dual-issue performance by 4% (Table 5), an identical speedup to that obtained in the SII simulations (Table 2). Decoupled Operand Forwarding and Optimized Register Files were then combined in Hades2_orf_dof. The speedup against the baseline MII simulation was still only 5% (Table 5). This disappointing result suggests that the basic dual-issue model had insufficient resources to exploit Decoupled Operand Forwarding.

Various enhanced MII architectures were therefore explored to see if performance could be improved by providing additional resources. Execution times for these variants are shown in Fig. 11.

The ID module in Hades2 operates on instructions in pairs, rather than singly. This in turn introduces greater concurrency into the Execution Units. The obvious corollary is that there is greater potential for concurrency in the WB module. To assess the impact of an additional Integer Result Bus, and dual write ports on the Integer Register File, an instance of Hades2 was simulated with two Integer Result Buses, Hades2_res. The two buses were available to all Execution Units and asynchronous arbitration was required to gain exclusive access.

An additional Integer Result Bus increases performance by a modest 1%. The small size of the increase can be accounted for by a number of factors. In a synchronous processor, the number of Integer Result Buses can be related to the instruction issue rate because the number of integer instructions issued equals, on average, the number of integer results written back. However, in an asynchronous processor the situation is less precise because the execution latency of an instruction is variable and depends on many non-deterministic factors. Therefore, in general, asynchronous processors require less resource duplication, because demand is not as regulated as in a synchronous system and tends to be smoothly spread over time rather than fixed around a clock edge.

The benchmark programs frequently contain pairs of arithmetic operations that could execute concurrently. To exploit this, a version of Hades2 with two Arithmetic Execution Units, Hades2_rsc, was simulated. Providing two Arithmetic Units increases performance by 6% even though a single Result Bus is retained. If a second Integer Result Bus is reintroduced to give Hades2_res_rsc, the speedup over the current baseline is increased to 8%. A MII version of Hades with duplicate Arithmetic Execution Units therefore requires duplicate Integer Result Buses if it is to achieve its full potential. It is this enhanced version that achieves a 21% speedup over SII Hades (Fig. 9). The overall result from a comparison of a MII Hades having both optimizations and additional resources, Hades2_res_rsc, with the baseline MII simulation, Hades2_ov, yields an overall speedup of 13%.

6.3 Additional Resources and Decoupled Operand Forwarding

We now return to the benefits of Decoupled Operand Forwarding. With a minimal dual-issue configuration, the performance improvement was only 5%. In this section we quantify the benefits of Decoupled Operand Forwarding in a fully resourced system, by comparing a Hades2 model with two Arithmetic Units and two Result Buses, Hades2_res_rsc, against a cut-down model without Decoupled Operand Forwarding (Fig. 12). The speedup for Hades2 with more available resources is now 11%, over double that obtained in the previous comparison. Even so the speedup in a dual-issue environment attributable to Decoupled Operand Forwarding is still only half the speedup achieved with single instruction issue.

One possible reason for the reduction in the speedup may concern the nature of asynchronous operation; MII Hades works more efficiently than SII Hades. Superficially, MII exposes more RAW hazards in programs, suggesting that a mechanism reducing the impact of these hazards should have a greater affect in a MII environment than in a SII environment. However, in a SII environment, instructions following a RAW hazard are also stalled because a stall in the ID module affects not only the dependent instruction but also any following instructions. In a MII environment this restriction is relieved to some extent because an instruction in the same group as the stalled instruction can still be issued. It may be that a MII version of Hades that issues two instructions concurrently is, in this respect, a special case because the benefit provided by concurrent issue to some extent overshadows the benefits of Decoupled Operand Forwarding. With higher issue rates Decoupled Operand Forwarding may have a greater effect in a MII environment than in a SII environment, mirroring the situation encountered in synchronous processor design [15]. Further work on Hades, concentrating on instances that issue a greater number of instructions concurrently, would provide a more definitive answer.

A feature of the operation of Decoupled Operand Forwarding in Hades is that instructions from the following instruction groups are prevented from issuing until all bypassing for the previous instruction group has been initiated. This was a conscious design decision rather than a requirement of asynchronous operation. A request from the ID module to a Forwarding Register in the EX module has the form "Forward Data", which constitutes only a single bit of information. Two other elements of the architecture allow the request to be fulfilled. Firstly, there are dedicated connections to communicate each operand in the ID module to the EX module for each instruction in the ID module. An instruction arriving at an Execution Unit in the EX module consumes the operands offered to it without verifying that they are correct; this is controlled by the ID module. Secondly, to ensure that an instruction does not have to verify its operands before execution, a second instruction cannot be issued from the same instruction slot in the ID module until the preceding instruction has obtained all its operands. Otherwise the two instructions could attempt to input the same operands leading to non-deterministic effects. The configuration described provides the minimum hardware and functionality required to realize both MII and bypassing in an asynchronous environment. Furthermore, the effects on performance have been minimized because it is not necessary for an instruction to have actually obtained its operands before a further instruction can be issued; it is sufficient if a Forwarding Register has indicated that data is ready for transfer, and an Execution Unit has indicated its readiness to receive it. It is not necessarily the case that the data will have been communicated, only that both parties involved in the data transfer be committed to that transfer.

The next stage in the development of Hades would be to remove the restriction that an instruction must be committed to all its operands before a further instruction can be issued from the same slot in the ID module. However, this demands extra resources and functionality that were not considered essential to the initial stage of Hades' development. The restriction can be removed in two ways: either, each instruction can be given sufficient information to vet each operand offered to it and can take only the operands it requires, or more information can be given to the Integer Register File and the Forwarding Registers so that they can target the data to a specific Execution Unit rather than to a slot in ID. Increasing the amount of information given to an instruction and allowing it to choose the operands it inputs is non-trivial because each item of data communicated to an Execution Unit must be uniquely tagged. It is not sufficient to use the Integer Register tags, because multiple instructions may require data from the same register and data from different sources may yield different values (i.e. the present state of the register and a result destined for the register). This means that the ID module must allocate and maintain unique tags for data in the pipeline and that the Execution Units must be able to check for correct tags.

Giving more information to the Integer Register File and the Forwarding Registers, to allow them to target a specific Execution Unit does not imply any additional operation latency because an instruction would again take the first operands offered to it, and the ID module already has all the required information when an instruction is issued. However, this scheme has a very high cost in additional hardware requirements. At present the Integer Register File and the Forwarding Registers each have dedicated, shared connections that can support each slot in the ID module. These connections broadcast data to all Execution Units in response to signals from ID. The data is broadcast over a one-to-many bus from the Integer Register File to the Execution Units in the EX module and a many-to-many connection from the Forwarding Registers to the Execution Units. The reason that these interconnections are guaranteed to work is that only one instruction, and hence Execution Unit, can be waiting for data at any time. Therefore, an instruction must be committed to its operands before a subsequent instruction can be issued. Removing this restriction with the present connection scheme would mean that two instructions could potentially accept the same data. For example, suppose an instruction is issued which then has to wait for bypassed data and, subsequently, a second instruction is issued from the same ID slot requiring data from the Integer Register File. Both instructions arrive at their respective Execution Units prepared to accept the first operands offered. Because of the non-determinism introduced by asynchronous operation it is not possible to predict which data will arrive first, so both instructions will attempt to accept the first data offered. The only alternative, if Execution Units are still to accept whatever data is offered, is to provide dedicated connections between the Integer Register File, Forwarding Registers and Execution Units to ensure that data can be targeted to the correct Execution Unit. Providing dedicated connections quickly inflates the number of buses required to unacceptable levels.

Hence, the next step in the development process for Hades would be to remove the restriction that all bypassing must occur in order, either by uniquely tagging each piece of data in the pipeline, or by informing the source of the data for which Execution Unit it is destined. In either case the introduction of shared data buses and arbitration for exclusive access would be required.

7 Conclusions

In this paper we consider performance issues in the asynchronous model, through the design and simulation of the Hades processor.

Communication was optimized for use in an asynchronous environment and was implemented in all subsequent simulations. Our experience with SII Hades showed that Optimized Register Files go some way towards reducing the impact of the register file. However, bypassing in the form of Decoupled Operand Forwarding is far more effective and improves performance by 22% regardless of whether an Optimized Register File is provided. This result suggests that Decoupled Operand Forwarding removes the register file from the critical path when dependent instructions are executed. The key to the success of Decoupled Operand Forwarding is to centralize control in the ID module without forcing synchronization on the Execution Units, which continue to operate asynchronously.

We began to investigate MII in an asynchronous environment with Hades2, a dual-instruction-issue version of Hades. An encouraging performance gain of 32% was achieved when the SII and MII baseline cases were compared. As this gain was reduced to 11% in the presence of both Optimized Register Files and Decoupled Operand Forwarding, we simulated other MII variants of Hades with additional resources. Results of a further comparison with SII Hades show a 21% speedup, indicating that some earlier resource constraints had been removed. Nonetheless, our results indicate that the speedup attributed to Decoupled Operand Forwarding is less in a MII environment than in a SII environment. Overall, Decoupled Operand Forwarding was found to be a very successful technique to improve the behaviour and performance of Hades.

References

[1] C. Seitz, System timing, in: Mead, C., Conway, L., (Eds), Introduction to VLSI systems, Addison-Wesley, 1980, pp. 218-262.

[2] C. J. Elston, Hades: an asynchronous superscalar processor, Ph.D. Thesis, University of Hertfordshire, 1996.

[3] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, J. V. Woods, AMULET1: a micropipelined ARM, Proceedings of CompCon'94, IEEE, San Francisco, March 1994, pp. 476-485.

[4] I. E. Sutherland, Micropipelines, Communications of the ACM, 32 (6) (1989) 720-738.

[5] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, S. Temple, AMULET2e, EMSYS '96 - OMI Sixth Annual Conference, IOS, Berlin, 23-25 September, 1996.

[6] R. F. Sproull, I. E. Sutherland, C. E. Molnar, CounterFlow Pipeline Processor architecture, Technical Report SMLI TR-94-25, Sun Microsystems Laboratories, 1994.

[7] S. Moore, P. Robinson, S. Wilcox, Rotary pipeline processors, IEE Proceedings Computers and Digital Techniques, 143 (5) (1996) 259-265.

[8] P. B. Endicott, SCALP, Ph.D. Thesis, University of Manchester, 1996.

[9] H. Corporaal, H. Mulder, Move: a framework for high-performance processor design, Supercomputing, 7 (1991) 692-701.

[10] J. Hoogerbrugge, H. Corporaal, Register file port requirements of transport triggered architectures, Micro27, ACM, San Jose, California, November 1994, pp. 191-195.

[11] G. B. Steven, F. L. Steven, ALU design and processor branch architecture, Microprocessing and Microprogramming, 36 (1993) 259-278.

[12] A. Martin, S. Burns, T. Lee, D. Borkovic, P. Hazewindus, The Design of an asynchronous microprocessor, Technical Report CS-TR-89-02, Caltech, 1989.

[13] N. Paver, P. Day, S. B. Furber, J. D. Garside, J. V. Wood, Register locking in an asynchronous microprocessor, ICCD 92: IEEE International Conference on Computer Design, October 1992, pp. 351-355.

[14] J. L. Hennessy, D. A. Patterson, Computer architecture: a quantitative approach, Morgan-Kaufmann, 1990, pp. 236-245.

[15] M. Franklin, G. S. Sohi, Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors, Micro25, ACM, Portland, Oregon, December 1992, pp.236-245.

[16] C. A. R. Hoare, Communicating sequential processes, Prentice Hall, 1985.

[17] IEEE Standard VHDL reference manual, IEEE Std 1076-1987.

[18] G. Steven, B. Christianson, R. Collins, R. Potter, F. Steven, A Superscalar architecture to exploit instruction level parallelism, Microprocessors and Microsystems 20 (1997) 391-400.

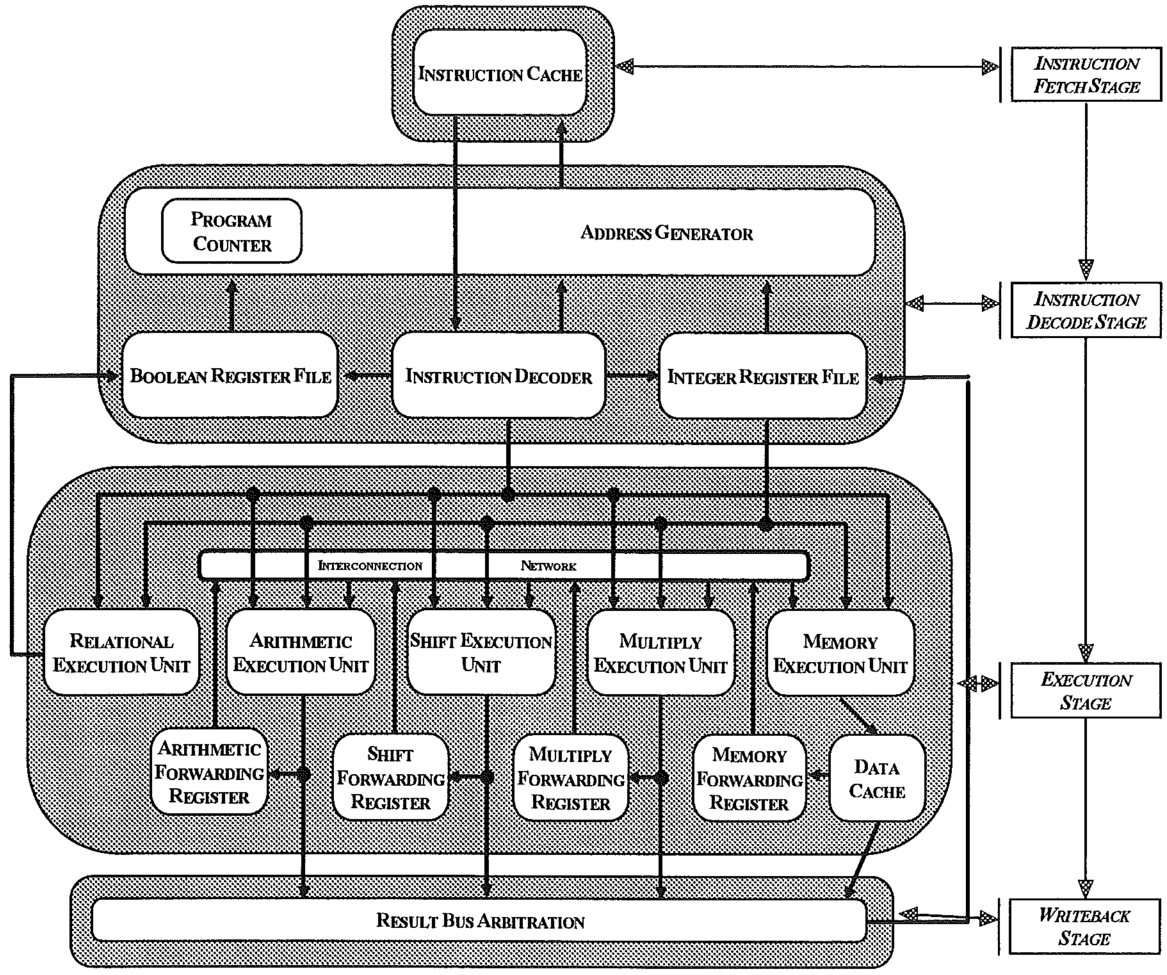

Fig. 1. The Hades Processor Architecture.

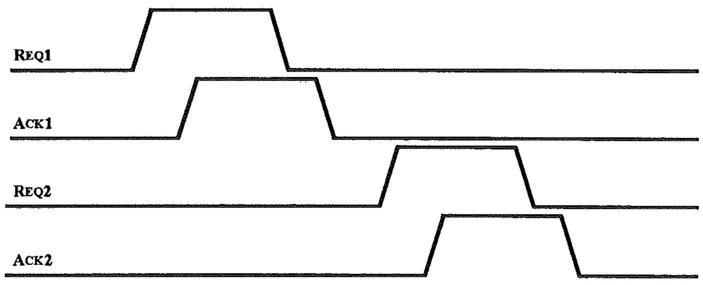

Fig. 2. Non-Overlapping Four-Phase Communication

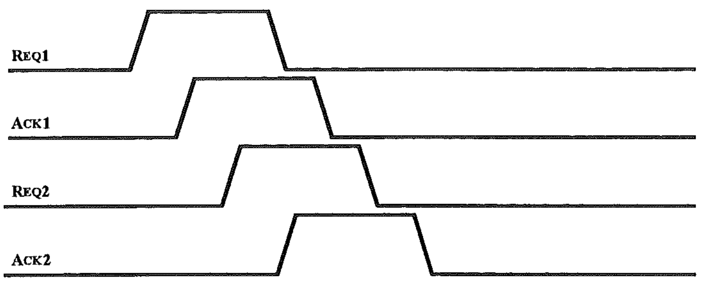

Fig. 3. Overlapping Four-Phase Communication

Table 1
SII simulations.

Hades1_nov	SII Hades, no Decoupled Operand Forwarding, no Optimized Register Files, Unoptimized Asynchronous Communication.
Hades1_ov	Hades1_nov, Optimized Asynchronous Communication.
Hades1_orf	Hades1_ov, Optimized Register Files.
Hades1_dof	Hades1_ov, Decoupled Operand Forwarding.
Hades1_orf_dof	Hades1_ov, Optimized Register Files, Decoupled Operand Forwarding.

Permute	Recursively computes permutations of a number of integer elements.
Matrix	Multiplies two integer matrices.
Queens	Solves the queens chess problem recursively.
Puzzle	Recursively solves a cube packing problem.
Quicksort	Recursive quicksort of a number of integer elements.
Tower	Solves the Towers of Hanoi problem recursively.
Bubble	Bubble sort of a number of integer elements.
Tree	Binary tree sort of a number of integer elements.

Fig. 5. The Stanford Integer Benchmark Suite

Please note figure four is over the page

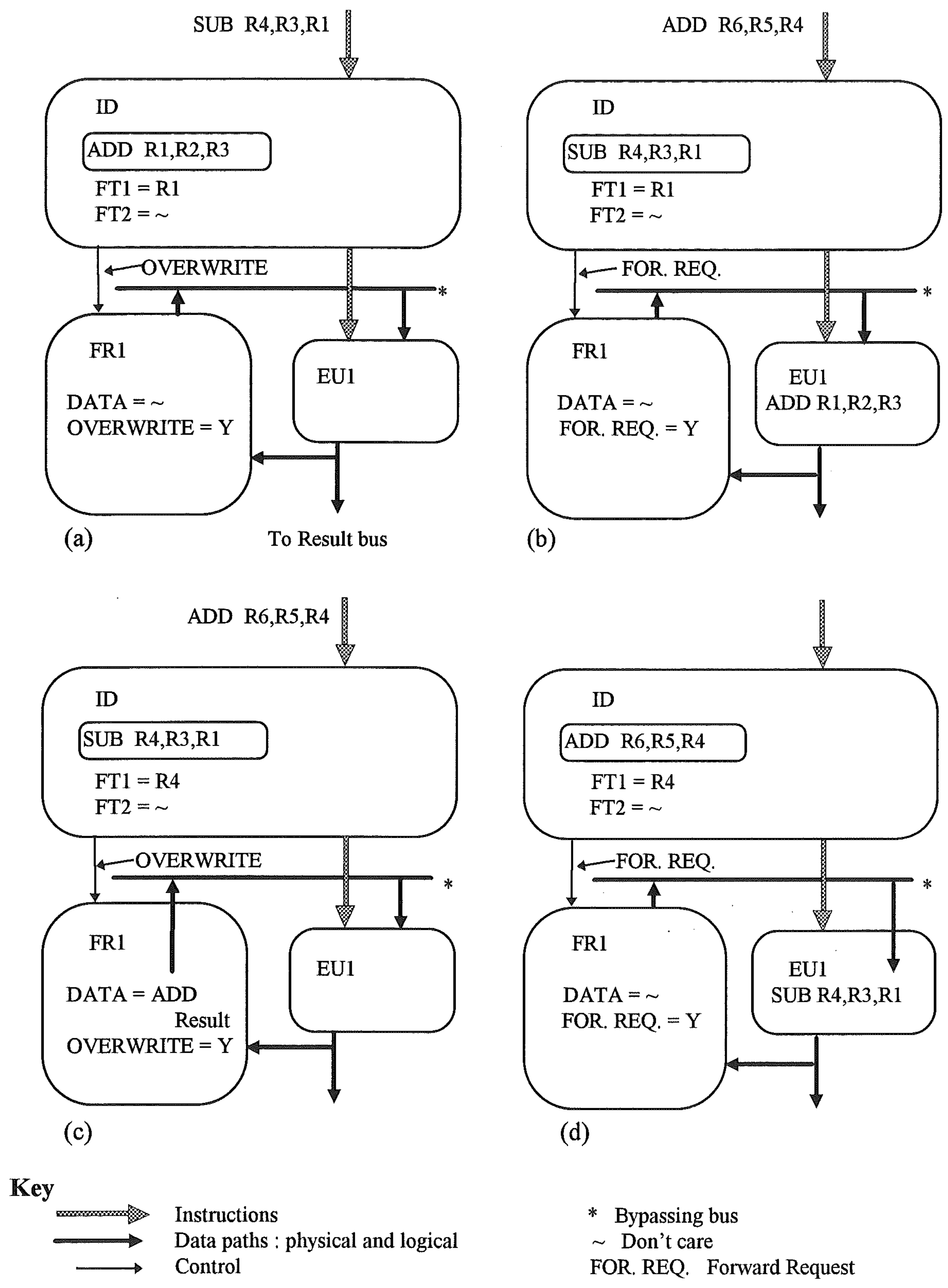

Fig. 4. An example of Decoupled Operand Forwarding, illustrating the use of a FR.

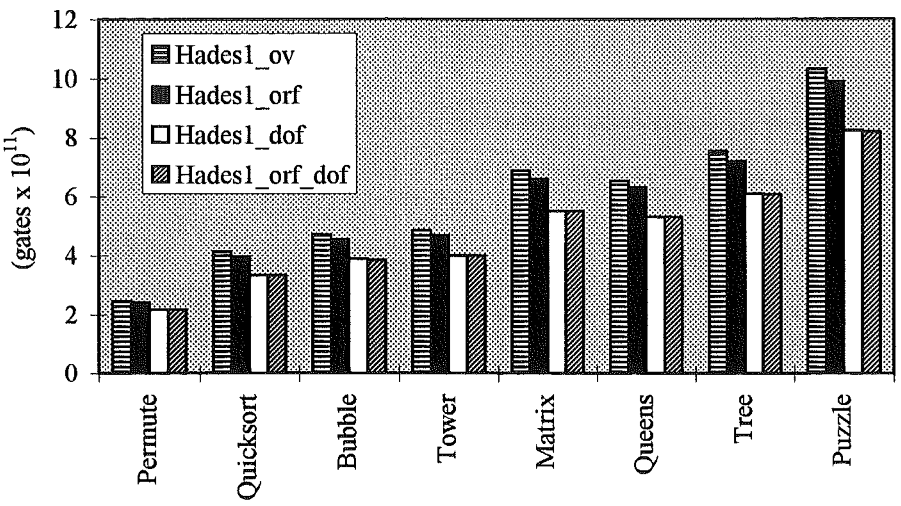

Fig. 6. Execution times for all SII simulations.

Table 2
Speedups in SII simulations when compared with Hades1_ov.

	Hades1_orf	**Hades1_dof**	**Hades1_orf_dof**
Permute	1.02	1.13	1.13
Quicksort	1.05	1.24	1.24
Bubble	1.04	1.21	1.23
Tower	1.04	1.21	1.21
Matrix	1.04	1.25	1.25
Queens	1.03	1.23	1.23
Tree	1.05	1.23	1.24
Puzzle	1.04	1.25	1.26
Geometric Mean	**1.04 (4%)**	**1.22 (22%)**	**1.22 (22%)**

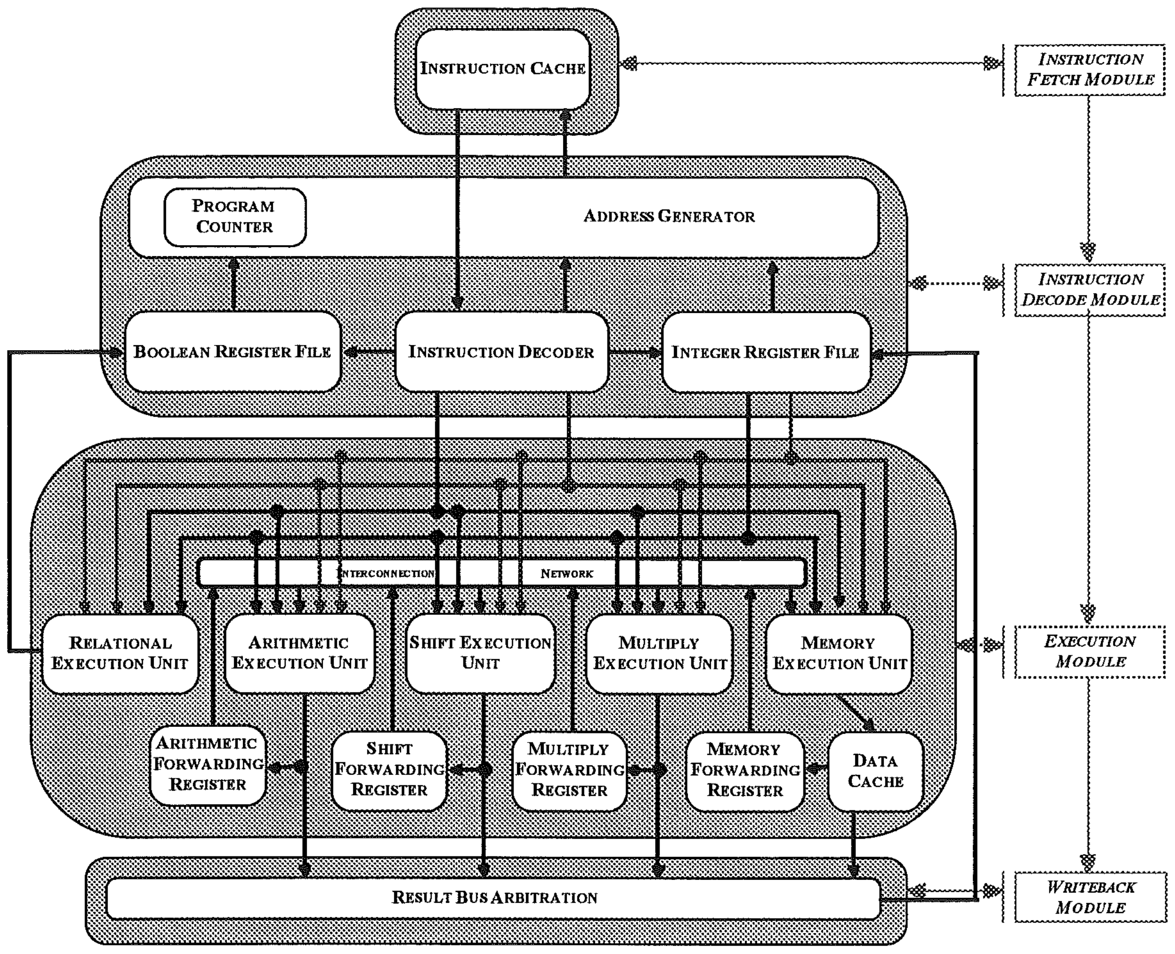

Fig. 7. The MII Hades Processor Architecture.

Table 3
MII simulations.

Hades2_ov	MII Hades, no Decoupled Operand Forwarding, no Optimized Register Files.
Hades2_orf	**Hades2_ov**, Optimized Register Files.
Hades2_orf_dof	**Hades2_ov**, Optimized Register Files, Decoupled Operand Forwarding.
Hades2_res	**Hades2_orf_dof**, two Integer Result Buses.
Hades2_rsc	**Hades2_orf_dof**, two Arithmetic Execution Units.
Hades2_res_rsc	**Hades2_orf_dof**, two result buses, two Arithmetic Execution Units.

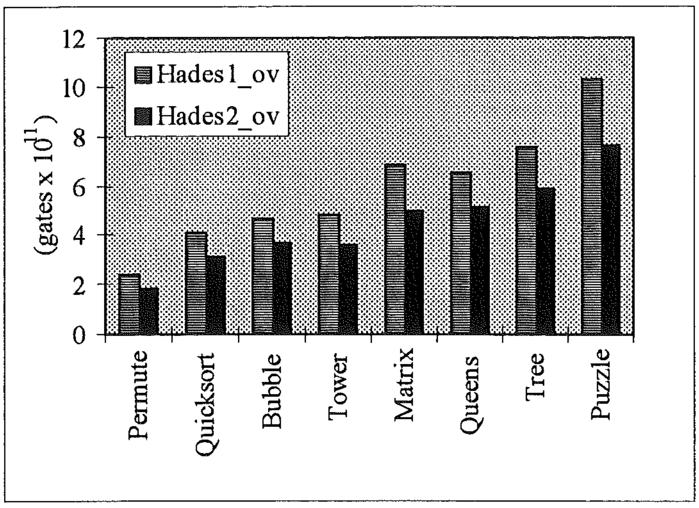

Fig. 8. Execution times for baseline SII and MII Hades.

Table 4
Speedups of MII over the equivalent SII simulations.

	Hades2_ov over Hades1_ov	**Hades2_orf_dof over Hades1_orf_dof**
Permute	1.29	1.17
Quicksort	1.31	1.12
Bubble	1.27	1.06
Tower	1.35	1.17
Matrix	1.37	1.13
Queens	1.26	1.07
Tree	1.28	1.10
Puzzle	1.35	1.09
Geometric Mean	**1.32 (32%)**	**1.11 (11%)**

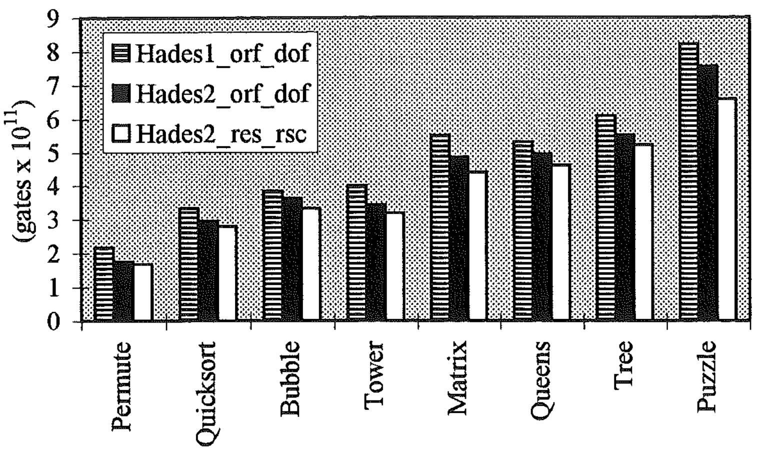

Fig. 9. Execution Times for a comparison of Hades2_orf_dof and Hades2_res_rsc with Hades1_orf_dof.

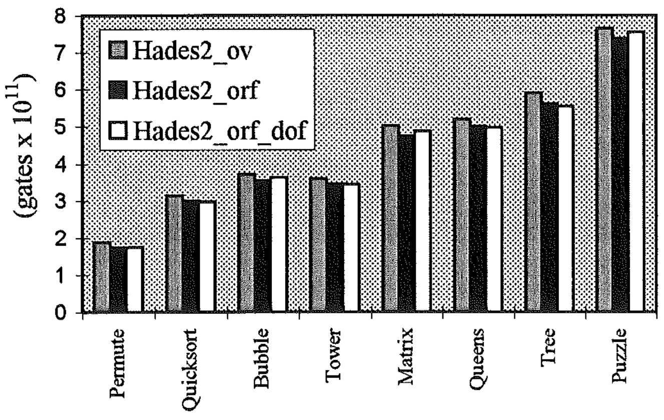

Fig. 10. Execution times for MII Hades simulations.

Table 5
Speedups from the MII base model, Hades2_ov.

	Hades2_orf	Hades2_orf_dof
Permute	1.05	1.08
Quicksort	1.05	1.06
Bubble	1.04	1.02
Tower	1.04	1.05
Matrix	1.06	1.03
Queens	1.04	1.04
Tree	1.05	1.06
Puzzle	1.04	1.05
Geometric Mean	**1.04 (4%)**	**1.05 (5%)**

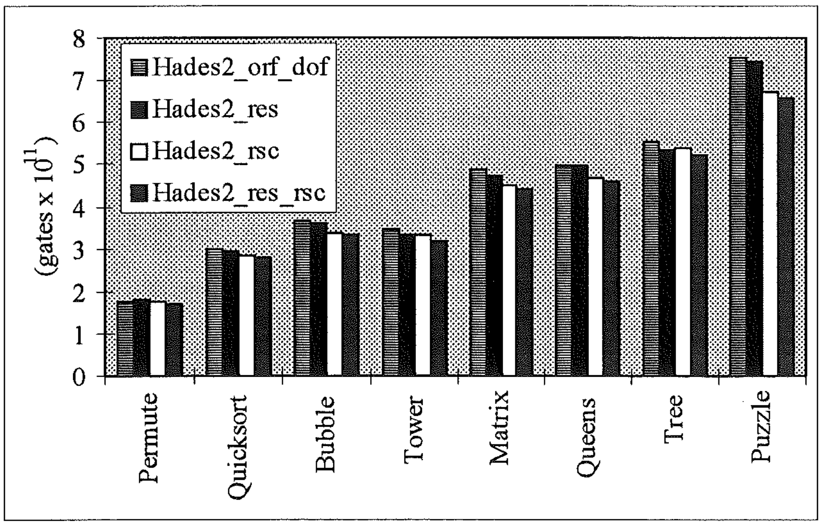

Fig. 11. Execution times for Hades2 variants.

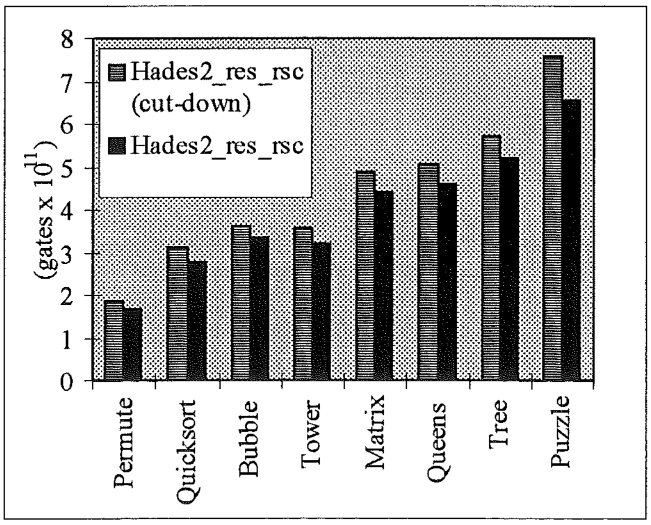

Fig. 12. Execution Times for a comparison of Hades2_res_rsc and a cut-down Hades2_res_rsc.