DIVISION OF COMPUTER SCIENCE

Hades - Towards the Design of an Asynchronous Superscalar Processor

C. J. Elston
D. B. Christianson
P. A. Findlay
G. B. Steven

Technical Report No 218

February 1995

Hades - Towards the Design of an Asynchronous Superscalar Processor

C. J. Elston, D. B. Christianson, P. A. Findlay, G. B. Steven
Division of Computer Science
University of Hertfordshire, Hatfield, HERTS AL10 9AB, UK.
email: comqgbs@herts.ac.uk Tel: 0707-28-4319 Fax: 0707-28-4303

Abstract
Hades is a processor architecture aimed at single and multiple-instruction-issue asynchronous implementations. This paper uses a baseline version of Hades to illustrate some of the difficulties encountered in asynchronous processor design and demonstrates why it is undesirable to design a processor which is simply an asynchronous version of an existing synchronous processor. Particular emphasis is placed on an explicitly declared delayed branch mechanism and a decoupled register forwarding mechanism developed for Hades. The branch mechanism allows instruction fetching to proceed while branch insructions are being resolved, while the forwarding mechanism allows the last result of each functional unit to be bypassed to following instructions, yet completely separates bypassing from the register write back operation.

1. Introduction

The previous four decades have been dominated by synchronous processors with global clocks. However, recently there has been renewed interest in asynchronous designs. This resurgence of interest in asynchronous techniques is driven by advancing technology which is increasingly exposing the inherent limitations of synchronous techniques.

High performance synchronous processors are devoting increasing amounts of silicon area, design effort and power to the global clock. For example the master clock on the Dec Alpha APX 21064 draws a peak switching current of no less than 43A [McClellan 93]. Routing the global clock is also non-trivial, requiring careful simulation to ensure that clock skew is within manageable bounds [Seitz 80]. Furthermore, the problems associated with global clocks are magnified by both higher clock frequencies and increasing levels of integration.

All operations in a synchronous processor occur in concert with the global clock. The clock period must therefore cater for the longest latency operation, even if it occurs only rarely. All other operations, possibly requiring only a fraction of a clock cycle, are constrained to complete in a single cycle. The performance potential of synchronous processors is therefore artificially limited by forcing all operations to complete in an integral number of clock cycles. The clock cycle must also allow for worst case gate delays at worst case temperatures. Effectively the clock of a synchronous processor must always cater for a worst case scenario, with worst case clock skew and worst case gate delays at worst case temperatures.

Asynchronous processors employ no global clock. Instead synchronisation is achieved through the interaction of adjacent elements. Completion of one operation therefore initiates logically dependent operations. The advantages of asynchronous techniques are manifold. Since there is no global clock to consider designs can be partitioned into completely independent modules. Clock circuitry is not required, reducing silicon area. Power consumption is lowered since there is no global clock consuming power and since idle elements are quiescent. In contrast, registers in a synchronous processor tend to be clocked in every cycle. Finally, asynchronous processors have a greater performance potential because operations are not constrained by the clock cycle. For example, addition can have a latency proportional to the carry chain length.

The Hades project is an investigation into asynchronous processor design. The differences between synchronous and asynchronous techniques make it undesirable to produce an asynchronous processor by simply mapping an existing synchronous processor onto an asynchronous control structure. Designing asynchronous processors involves new problems which in turn require novel solutions. No aspect of design should therefore be sacrosanct in the move from synchronous to asynchronous techniques.

The Hades project aims to develop and assess schemes that increase the performance of asynchronous processors. A baseline Hades processor has been designed using the formal specification language, CSP [Hoare 85]. The concurrency inherent in CSP allows an asynchronous processor to be modelled as a hierarchy of concurrent processes communicating asynchronously. Hardware simulations of Hades in VHDL (VHSIC Hardware Description Language) [IEEE 87] are at an early stage. VHDL is used as a simulation environment because of the concurrency features offered. A relationship is being cultivated between CSP and VHDL to facilitate rapid construction of VHDL simulators.

Hades derives from two other computer architecture projects at the University of Hertfordshire. The main progenitors of Hades are HARP (Hatfield Advanced RISC Project) [Steven 92], a VLIW (Very Long Instruction Word) architecture and HSP (Hatfield Superscalar Processor) [Steven 94], a superscalar architecture. A common feature of both the HARP and HSP projects is the importance of instruction scheduling. In both cases the full potential of the architecture is only realised through the use of static code reordering at compile time. Static instruction scheduling is also an integral part of the Hades project.

In this paper a baseline version of Hades is presented to illustrate specific aspects of asynchronous processor design including an explicitly declared delayed branch mechanism and a decoupled operand forwarding mechanism. Where necessary the baseline model is expanded to show how other instantiations of Hades can be used to explore the asynchronous design space. Section 2 briefly reviews related work. Section 3 provides an overview of Hades. Section 4 provides an overview of the Hades implementation. Section 5 describes the Hades' explicitly declared delayed branching mechanism. Sections 6 and 7 give further details of the Secondary Decode Unit and Register File. Section 8 describes the Hades decoupled branch mechanism. Finally, Section 9 offers some concluding remarks.

2. Review of Related Work

This section places Hades in context by briefly descibing other work in the area. Research into asynchronous processor design was initially stimulated by a seminal paper on micropipelines by Sutherland [Sutherland 89].

The University of Manchester AMULET project investigates asynchronous logic, methodologies and processors. A micropipelined processor, AMULET1 [Furber 93], has been fabricated and another, AMULET2, is under development using the lessons of AMULET1. The project aim is to exploit the low power potential of asynchronous processors.

Alain Martin's group has designed and fabricated several asynchronous processors [Martin 89]. The aim is to develop a methodology for designing asynchronous processors using a basic processor architecture as a case study. Recent work has involved a GaAs processor implementation [Tierno 94].

The Counterflow Pipeline Processor Architecture (CFPP) [Sproull 94] is a radically different asynchronous pipeline structure where instructions and data flow through the processor pipeline in opposite directions. The aim is to develop processor organisations that allow efficient asynchronous realisations.

In spite of a steadily increasing amount of research in the area, there is still a widely held belief that asynchronous processors are inherently slower than their synchronous counterparts, partly because it is held that register bypassing can not be efficiently implemented in asynchronous designs. The aim of the Hades project is to develop processor organisations, including efficient bypassing mechanisms, that will allow the full performance potential of asynchronous processors to be realised.

3. An Overview of the Architecture

Hades has evolved from ideas developed with synchronous processors. However, the change in focus from synchronous to asynchronous techniques has necessitated alteration of several key areas of processor design. In particular a branch mechanism and an operand forwarding mechanism have been introduced to solve problems created by the move from synchronous to asynchronous techniques.

Hades has a RISC instruction set with a small number of simple instructions. The instruction set is compatible with the HSP instruction set, allowing software developed for HSP to be reused. In particular we plan to use the HSP instruction scheduling software on the Hades project.

Hades has two register files, one integer and one boolean. The boolean file stores conditions that are generated by integer comparisons. All conditional branches then test a boolean condition. Comparison instructions are restricted to generating a single boolean value thus ensuring that both comparison operations and branch resolution can be completed as quickly as possible [Steven 93]. Unconditional jumps correspond to a branch-on-false using boolean register B0 which is always false. Branch to subroutine, move register to PC and trap instructions are also provided.

4. Implementation Overview

Underlying Hades implementations is the concept of an asynchronous FIFO (First-In-First-Out) buffer [Sutherland 89]. Control in an asynchronous FIFO is localised and operation is elastic; throughput is determined directly by the availability of data. The elasticity in an asynchronous FIFO allows different input and output data rates to be sustained for short periods. An asynchronous FIFO therefore provides an organisation with localised control that can continue useful operations in the face of temporary interruption to the supply of instructions and data through cache misses or while branch instructions are being resolved.

All Hades implementations use the following five pipeline stages:

IF	- Instruction Fetch
ID1	- Primary Decode
ID2	- Secondary Decode

EX	- Execution Stage
WB	- Writeback

Separate instruction and data caches are adopted to reduce memory access bottlenecks. The basic pipeline can be instantiated as either a single-instruction-issue or multiple-instruction-issue processor. All instantiations of the architecture use in-order instruction issue but allow out-of-order instruction completion. In-order instruction issue allows significant simplification of the instruction decode stages. Furthermore, since compile-time instruction scheduling is envisaged, the performance benefits of out-of-order instruction issue are minimal.

The organisation of the baseline Hades processor is shown in Figure 1. The Instruction Fetch (IF) stage has the task of supplying instructions to the pipeline. A multiple-instruction-issue version of Hades can issue instructions in concurrent groups increasing the demand for instructions in proportion to the issue rate. To prevent a bottleneck the instruction bandwidth of the instruction cache must be maximised. There are two ways of achieving this: First, instructions can be fetched in groups, masking the latency of each individual cache access. Secondly, instruction fetching can be pipelined to increase throughput. Both alternatives will be modelled and the advantages and disadvantages of each assessed by simulation.

The Primary Decode Stage (ID1) is involved in the explicitly declared delayed branch mechanism explained in section 5. The task of Secondary Decode Stage (ID2) is twofold: First, ID2 produces executable instructions for the Execution Stage. Secondly, ID2 executes branch instructions. Producing an executable instruction for the Execution Stage includes providing control information, initiating register file accesses, allocating execution units and sign extending immediates and address offsets contained in the instruction. Executing branch instructions in ID2 resolves branches early in the pipeline yet avoids arbitration for access to the boolean register file by concentrating all reads from the boolean register file in the ID2 stage.The Execution Stage (EX) performs the operations specified by the instructions. The stage is composed of a number of independent functional units each of which executes a subset of the instruction set. One or more of each of the following functional units are provided:

- Arithmetic unit.
- Shift unit.
- Multiply unit.
- Relational unit.
- Memory unit.

Operand address formation is performed by a dedicated adder within the memory unit. The operation of a functional unit depends only on the supply of instructions and data. This allows concurrent execution of instructions, up to a maximum equal to the number of units.

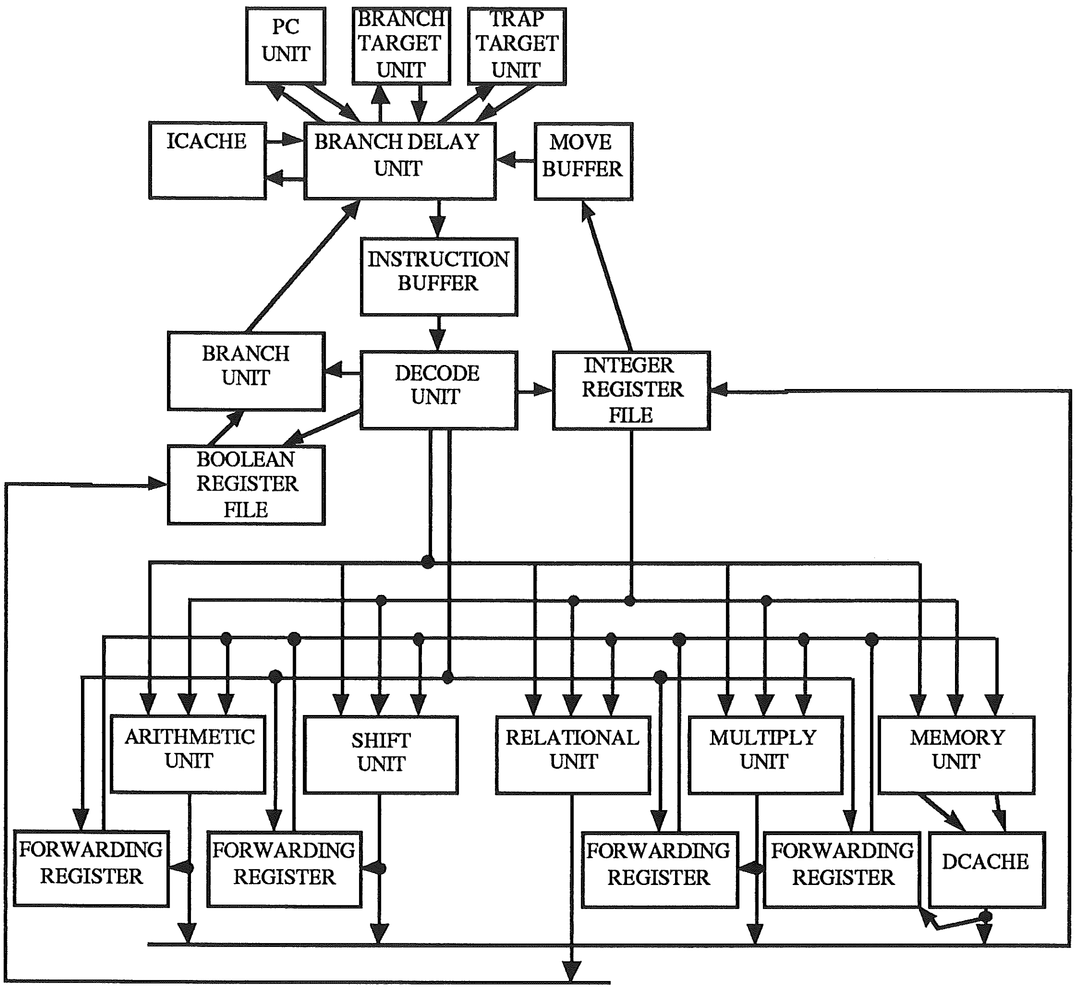

Figure 1: An Overview of the Hades Processor.

The Writeback Stage (WB) updates the state of the integer register file with results produced in the functional units. A result is sent to a register files across one of a number of shared result buses. A result from a functional unit consists of a bundle of data comprising a destination identifier and the data itself. The arrival of the data component is used to arbitrate for exclusive access to a result bus. The Boolean register file is also updated during the Writeback Stage. Since only relational and memory units generate boolean values, dedicated interconnections are provided between functional units and the boolean register file.

5. Explicitly Declared Delayed Branching

The Hades explicitly declared delayed branch mechanism aims to provide an effective method for resolving branches in an asynchronous processor. Explicitly declared delayed branches are a generalisation of the standard RISC delayed branch mechanism whereby one or more instructions following each branch is always executed before the branch is taken. In Hades the number of instructions after a branch instruction which are executed regardless of the outcome of the branch is not fixed by the architecture but is encoded directly in the branch instruction [Collins 94].

A count, called the delay count, is associated with each branch indicating the size of the delay region. A zero delay count is initially associated with each branch during compilation. Then during scheduling, the instruction scheduler attempts to move instructions into the branch delay region to provide the pipeline with useful work while the branch is being resolved. Finally, the scheduler adjusts the delay count to reflect the number of instructions after the branch which must now be executed at run time before the branch can be taken.

Explicitly declared delayed branching is implemented as two separate operations: The delay count of the branch is satisfied in the Primary Decode Stage (ID1) while the branch is resolved in the Secondary Decode Stage (ID2). Instructions from the instruction cache are returned to ID1 which determines if the instruction is a branch. Branches trigger ID1 to satisfy the branch delay count while non-branch instructions cause fetching to continue sequentially. In either case the instruction is forwarded to ID2. ID2 in turn issues a branch instruction to the Branch Unit for execution and initiates a parallel read of the branch condition from the boolean register file. Branch execution, performed concurrently with delay satisfaction, resolves the branch and produces a condition, called a target condition, indicating the address of the instruction following the delay region. When delay satisfaction is complete the Branch Delay Unit uses the target condition to determine the address of the next instruction fetch.

The operation of explicitly declared delayed branching is illustrated in Figure 2, a series of snapshots of a simplified Figure 1. Reality is more fluid than Figure 2, only the logical sequence of operations is guaranteed to occur. In the baseline model, speculative instructions, which may have to be later removed from the pipeline, are avoided to avoid unnecessary complexity. As a result the branch delay region must always contain at least one instruction. This restriction will be removed in later Hades models.

Figure 2(a) - An instruction is input from memory and the next sequential instruction address is immediately output to the instruction cache giving a delay region of at least one instruction. The Branch Delay Unit recognises the instruction as a branch, saves a copy of the delay count and initiates the calculation of the branch target address. Meanwhile, the branch is output to the Decode Unit in the Secondary Decode Stage.

Figure 2(b) - The branch instruction is input by the Decode Unit which in turn issues the instruction to the Branch Unit for execution and initiates a read access from the boolean register file to obtain the appropriate boolean condition. The Branch Delay Unit continues delay satisfaction: The instruction following the branch is input from memory, the delay count is decremented and a sequential address output.

Figure 2(c) - The branch has been issued to the Branch Unit and branch execution begins. The Decode Unit is free to accept another instruction.

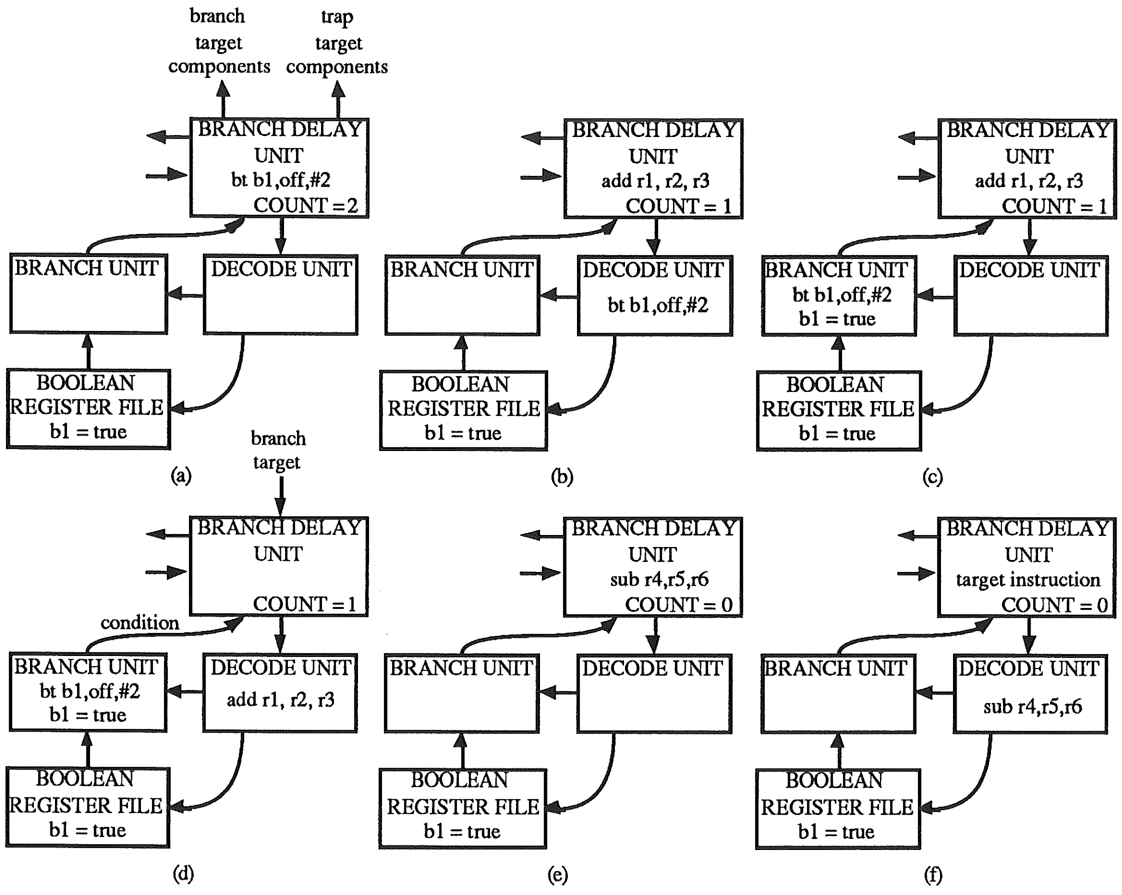

Figure 2: Example Branch Execution.

Figure 2(d) - Branch execution has completed producing a boolean condition to resolve the branch. Since the delay count in the Branch Delay Unit is zero indicating that the final delay region instruction fetch has been initiated, the boolean condition returned from the boolean register file is used to select the next instruction address.

Figure 2(e) - The final delay region instruction is input from the instruction cache and the branch target address is output.

Figure 2(f) - Execution continues with the branch target instruction returning from memory.

The aim of explicitly declared delayed branching is to hide the latency of branch execution behind the latency of delay satisfaction which continues to supply the pipeline with useful instructions. The success of all delayed branch mechanisms depends on the instruction scheduler being able to fill the delay region of a branch instruction with sufficient useful instructions to allow operation to continue in the presence of an unresolved branch. During the last few years considerable experience has been gained in instruction scheduling for multiple-instruction-issue processors, both at the University of Hertfordshire [Steven 95] and elsewhere [Moon 92, Lam 88]. This work suggests that filling a small number of branch delay slots is no longer a significant problem.

The branch resolution scheme described above was designed for later comparisons with more comprehensive schemes. Future extensions will allow branch instructions to be scheduled in branch delay slots and branch delay counts of zero. This last feature will be provided by ensuring that any speculatively fetched instructions immediately after a branch are squashed in the Primary Decode Unit if they are not required.

6. Secondary decode (ID2)

The operation of ID2 can be decomposed into four sub-operations:

- Resource Allocation: Since multiple functional units are provided, all instructions must be allocated to functional units. Where multiple instances of the same functional unit are provided, instructions are allocated dynamically to functional units on a rotating basis.
- Register File Access: Two types of access are required, one to read the source registers and a second to reserve or lock a destination register. The operation of the register file is described in Section 7.
- Operand Forwarding: Decoupled operand forwarding is initiated during ID2. This mechanism is explained in detail in Section 8.
- Instruction Issue: Instructions are issued to functional units together with any literal operands embedded in the instruction.

It is possible to arrange these operations sequentially, to maximise throughput, or concurrently, to minimise latency. In Hades resource allocation, operand forwarding and instruction issue are grouped together into a three stage pipeline which operates in parallel with the register file pipeline. This organisation aims to maximise the throughput of ID2, while masking the latency of the register file accesses.

Instructions are issued in-order simplifying implementation at the expense of greater reliance on compile-time instruction scheduling. True data dependencies (RAWs) and output dependencies (WAWs) are resolved by the register locking mechanism described in Section 7. As a result, although the decoupled operand forwarding described in Section 8 increases performance, it is not essential for correct operation.

7. Asynchronous register files

The Hades register file organisation has been designed to allow read and write accesses to proceed concurrently and independently whenever possible. To ensure correct operation a register locking mechanism similar to [Paver 92] and [Martin 89] is employed.

During ID2 the source and destination register fields of an instruction are sent to the register file, and a register read is initiated to obtain the source operands. To ensure correct operation a lock bit is associated with each register. During the read access, a lock operation is also performed by setting the lock bit associated with the destination register. The locked register is only subsequently unlocked when the result of the instruction is returned to the destination register. The lock associated with each register mediates both read and lock accesses to the register file, stalling any subsequent instructions with a locked source or destination register. Register locking resolves RAW hazards by stalling subsequent read accesses until the data is available and WAW hazards by stalling an instruction whose destination register is locked.

Register read accesses in Hades therefore proceed as follows:

- Accept next instruction.
- Stall until all source and destination registers are unlocked.
- Access source operands from register file.
- Forward source operands to functional unit.
- Lock destination register.

The final lock operation occurs in parallel with the two preceding operations. Register writes which return instruction results to the register file proceed completely independently of the read operations. When a write is complete, the destination register lock bit is also unlocked.

8. Operand Forwarding in Hades

It is usual to structure a pipeline so that once a result has been produced it travels through one or more stages before it becomes available to a subsequent instruction from the register file. When an immediately following instruction wishes to use the result, the pipeline may therefore have to be stalled until the data becomes available. Although this situation appears undesirable, the alternative of reducing the number of pipeline stages is also likely to reduce performance. Instead a mechanism called register bypassing or operand forwarding is usually implemented to route data directly from the ouput of the functional unit where it is produced to the input of the functional unit which requires the new data. Such is the frequency of operand reuse, particularly in a multiple-instruction-issue processor [Fernandes 84], that the absence of some form of bypassing is likely to degrade the performance of any synchronous or asynchronous processor.

8.1 Traditional Register Bypassing Mechanisms

In a synchronous system two independent activities can to be synchronised by associating each activity with a clock edge or other specific time in the clock cycle. Such synchronisation is implicit since there is no explicit communication between the two activities. Implicit synchronisation allows operand forwarding to be implemented with minimal overhead.

Figure 3 shows a pipeline diagram for a synchronous version of Hades. All stages have a latency equal to the clock cycle and all data are latched on the clock edge. Data can therefore be communicated between non-adjacent stages using the implicit synchronisation provided by the clock edge. As a result data generated in the execution stage of one instruction can be forwarded directly to the beginning of the execution stage of a following instruction requiring the data. (see dotted lines in Figure 3). Comparisons between the source and destination fields of the two instructions are used to control the additional data paths required.

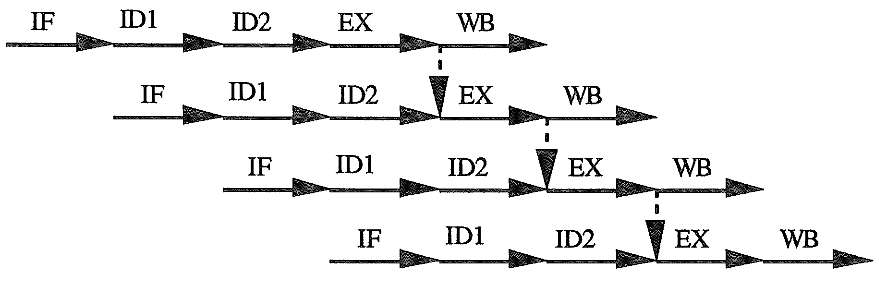

Figure 3: A Synchronous Hades Pipeline Diagram.

An asynchronous system contains no global clock and therefore no implicit synchronisation. Instead, synchronisation must involve explicit communication between elements. Synchronisation of adjacent stages in an asynchronous pipeline is essential to maintain the flow of data. However, more comprehensive synchronisation of non-adjacent stages is undesirable and can lead to a lockstep operation of pipeline stages and reduced performance. Unfortunately, the implementation of traditional bypassing schemes in an asynchronous environment tends to lead to an undesirable level of high-level synchronisation. However, as noted earlier, without an effective form of operand forwarding the performance of a multiple-instruction-issue asynchronous processor is likely to be compromised.

8.2 Decoupled Operand Forwarding

Hades therefore provides an unconventional decoupled operand forwarding mechanism. The central idea is to avoid high-level synchronisation by separating the forwarding from other pipeline operations. The aims of the mechanism are fourfold:

- To forward results directly between functional units, rather than indirectly through the register file.
- To remove the register file access from the critical execution path whenever a chain of dependent instructions is being exceuted.
- To eliminate high-level synchronisation from the pipeline, allowing operations to continue asynchronously.
- To prevent a locked register from stalling not only the current instruction attempting to access the lock but also subsequent, possibly independent instructions.

Section 7 details a register locking mechanism that resolves RAW hazards by stalling instruction issue until the required operand is available from the register file. Decoupled operand forwarding virtually eliminates these stalls by providing a mechanism for obtaining operands directly from the functional units that produced them. The objective is to increase performance. Correct operation is still ensured by the register locking mechanism.

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 functional units. Each functional unit includes a forwarding register that holds a copy of the last result produced by the unit. These results are then forwarded to subsequent instructions under the explicit control of the Secondary Decode Stage.

Instructions are allocated to functional units during the secondary decode stage, ID2. This in turn identifies the forwarding register that will contain the data produced by the instruction. A tag is maintained by ID2 for each forwarding register which uniquely identifies the current contents for the purposes of operand forwarding. When an instruction is issued the appropriate forwarding tag is updated and an overwrite signal is sent to the corresponding forwarding register, invalidating the existing data and allowing the register to be updated with a new result.

During ID2 each instruction compares its source register fields with the forwarding tags. If no match occurs the source operand is obtained from the register file. If a match occurs decoupled operand forwarding is initiated by requesting the matched forwarding register to send data directly to the instruction's execution unit. The forwarding register will only output valid contents. If the register has received an overwrite signal, it will wait until the next result has been loaded from the functional unit before forwarding data. In this case the forwarding register transmits the operand directly to the execution units. If the functional unit is both forwarding data and executing the instruction, the forwarding register will receive both a forward and overwrite signal. Note that WAW hazards must still be detected using the register file lock bits. An instruction must still not be issued to a functional unit if the destination lock bit is set.

As an example of decoupled operand forwarding consider the following code fragment:

```
Instruction 1:      add r1, r2, r3      /* r1 := r2 + r3 */
Instruction 2:      sub r4, r3, r1      /* r4 := r3 - r1 */
```

Here the subtraction uses the result of the addition. The execution of these instructions is described with reference to Figure 4, a series of snapshots of a simplified version of Figure 1.

Figure 4(a) - Instruction 1 (add r1,r2,r3) is available to the Decode Unit in ID2 and source and destination identifiers (r1,r2,r3) are available to Integer Register File. This is assumed to be the initial state of the pipeline All elements of state marked 'x' are not pertinent to the example and can be ignored.

Figure 4(b) - In ID2, instruction 1 has been allocated to ALU1. The Decode Unit has compared the source register fields of instruction 1 to the tags identifying potential forwarding data (FOR_REG1 and FOR_REG2). No match has been obtained, indicating that both operands must be obtained from the register file. Signals are therefore sent to the register file to initiate read register accesses for r2 and r3. A lock destination register access is also initiated for r1. Finally, FOR_REG1 is updated to contain r1, the destination register of instruction 1, and an overwrite signal is sent to the ALU1 forwarding register.

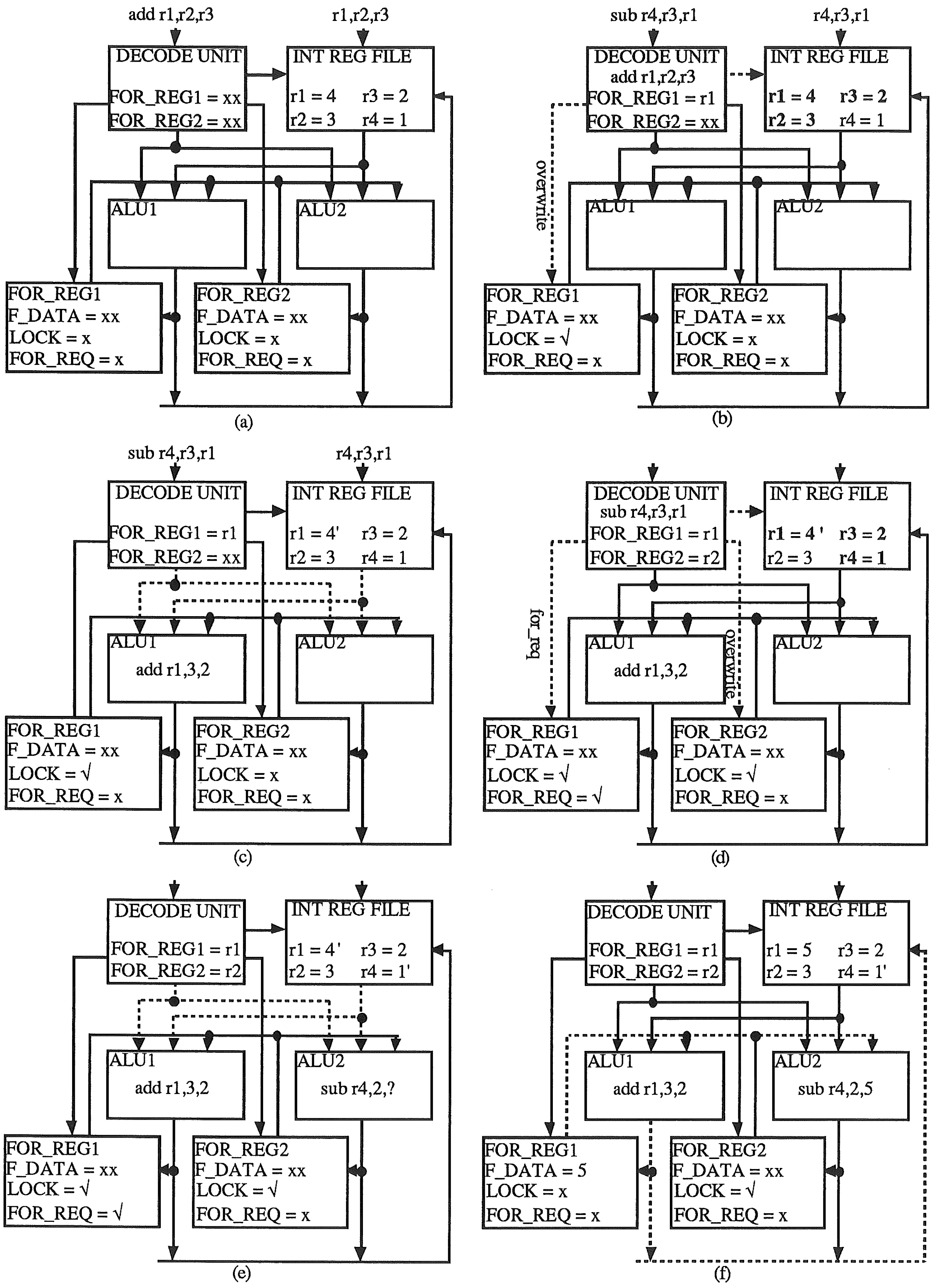

Figure 4: A Series of Diagrams Illustrating Decoupled Operand Forwarding.

Figure 4(c) - Instruction 1 is issued by the Decode Unit and the contents of r2 and r3 are output by the register file.

Figure 4(d) - ALU1 begins execution of the add instruction. The Decode Unit receives instruction 2 (sub r4,r3,r1) and the integer register file receives new source and destination register fields. This time r1 from instruction 2 matches FOR_REG1. A request is therefore sent to the forwarding register of ALU1 asking it to send data directly to ALU2. As a further result, a read access is initiated for r3 but not for r1. A lock register access is also required for r4. Finally, an overwrite signal is sent to the ALU2 forwarding register and the state of the forwarding tags is updated.

The ALU1 forwarding register cannot respond immediately since it is waiting the addition result from ALU1. However, as soon as this result is received it will be passed straight through the forwarding register to ALU2.

Figure 4(e) - Instruction 2 is issued to ALU2. The subtraction stalls until the forwarded value of r1 is received.

Figure 4(f) - Instruction 1 completes execution and the result is loaded into the forwarding register. This register is effectively transparent to the data which is passed immediately to ALU2, allowing the subtraction to start. The ALU1 result is also written to the register file concurrently. Note, however, that the most recent value of r1 remains in the forwarding register and is therefore available for subsequent reuse.

Decoupled operand forwarding saves the last result produced by each functional unit for possible forwarding. There are three possibilities for the state of the processor when decoupled result forwarding is initiated.

The first possibility is that the operand required is available from both the register file and a forwarding register. In this case the operand will be obtained directly from the forwarding register. Since the forwarding operation is faster than a register access, performance will be improved as long as a register access is not required for a second register operand.

The second possibility is that the data is present in a forwarding register but not in the register file. Accessing the register file would therefore cause a stall, affecting not only the current instruction but subsequent, possibly independent instructions. In this case, decoupled operand forwarding provides the operand directly from the forwarding register with a latency determined by the physical separation of the forwarding register and the execution unit. This increases performance on two fronts: First the operand is provided more quickly. Secondly, the issue of following instructions is not stalled.

The third possibility is that the operand is still being produced and is therefore available from neither the register file nor a forwarding register. This is the traditional data forwarding case.

Again decoupled operand forwarding boosts performance: First, the operand will be forwarded as soon as it is produced. Secondly, the issue of following instructions will not be unnecessarily delayed. It is therefore possible for subsequent instructions to overtake instructions waiting for their operands to be forwarded.

9. Conclusions and Future Work

Hades is a test bed for developing and assessing alternative asynchronous processor organisations. In this paper we have presented a baseline model of Hades. In particular, we have outlined a proposed delayed branch mechanism and a decoupled operand forwarding mechanism.

The central feature of the operand forwarding is the complete separation of bypassing from the register file write back operation. This mechanism was developed to provide an efficient bypassing mechanism for asynchronous processors which could be easily extended to multiple-instruction-issue designs. However, as synchronous superscalar designs continue to require ever-more-complex register file organisations with an ever-increasing number of read and write ports, the separation of bypassing from write back could prove to be a useful technique for avoiding register file bottlenecks in synchronous as well as asynchronous designs.

The Hades model presented will undergo further refinement before being compared with more powerful multiple-instruction-issue models. Our aim is to develop an asynchronous processor organisation which will compete effectively with synchronous superscalar designs. Since future superscalar designs are likely to use aggressive compile-time instruction scheduling, an effective asynchronous design must also be amenable to instruction scheduling, in spite of the inevitable uncertainties associated with the timing of operations within an asynchronous processor.

We will continue to use CSP and VHDL to support our work, both to clarify our ideas and to support our hardware simulations. We also hope that CSP will allow us to demonstrate that all our Hades models are logically equivalent to a simple sequential Hades interpreter.

Acknowledgements

The authors would like to thank the rest of the Computer Architecture Group at the University of Hertfordshire, in particular, Roger Collins, Richard Potter, Fleur Steven and Simon Trainis from Computer Science and Dave McHale from ERDC. They would also like to thank Professor M Loomes, Dr S L Stott, J A Davis and Professor P Kaye for their support. Part of this research was supported by an EPSRC Research Studentship.

References

Collins 94)	Collins,R. & Steven,G.B. "An Explicitly Declared Delayed-Branch Mechanism for a Superscalar Architecture," EuroMicro94, Liverpool, September 1994.
IEEE 87)	"IEEE Standard VHDL Reference Manual," IEEE Std. 1076-1987, IEEE, Piscaway, N.J., 1987.
Fernandes 94)	Fernandes,E.S.T. & Vasconcelos,N.Q. "A Study of Associative Dispatch in Superscalar Processors," EuroMicro94, Liverpool, September 1994, pp346-352.
Furber93)	Furber,S.B., Day,P., Garside,J.D., Paver,N.C. & Woods,J.V., "A Micro-pipelined ARM," VLSI 93: International Conference on VLSI, Grenoble, France, September 1993, pp5.4.1 - 5.4.10.
Hoare 85)	Hoare,C.A.R. "Communicating Sequential Processes," Prentice-Hall, 1985.
Lam 88)	Lam,M. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," SIGPLAN88, Atlanta, June 1988, pp318-328.
Martin 89)	Martin,A., Burns,S., Lee,T., Borkovic,D. & Hazewindus,P. "The Design of an Asynchronous Microprocessor," Technical Report CS-TR-89-02, Caltech, 1989.
McLellan 93)	McLellan,E. "The Alpha AXP Architecture and 21064 Processor," IEEE Micro, June 1993, pp36-47.
Moon 92)	Moon,S. & Ebcioglu,K. "An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors", Micro 25, Portland, Oregon, December 1992, pp55-71.
Paver 92)	Paver,N., Day,P., Furber,S.B., Garside,J.B.& Wood,J.V. "Register Locking in an Asynchronous Microprocessor," ICCD 92: IEEE International Conference on Computer Design, October 1992, pp351-355.
Seitz 80)	Seitz,C. "System Timing, in Introduction to VLSI systems," C. Mead & L. Conway, Eds., Addison-Wesley, 1980, pp218-262.
Sproull 94)	Sproull,R.F., Sutherland,I.E. & Molnar,C.E. "Counterflow Pipeline Processor Architecture," Sun Microsystems Laboratories, April 1994, pp1-23.
Steven 92)	Steven,G.B., Adams,R.G., Findlay,P.A. & Trainis,S A. "iHARP: A Multiple Instruction Issue Processor," IEE Proceedings, Part E, Computers and Digital Techniques, Vol.139, No.5, September 1992, pp 439-449.
Steven 93)	Steven,G.B. & Steven,F.L. "ALU Design and Processor Branch Architecture," Microprocessing and Microprogramming, Vol.36, No.5, October 1993, pp 259-278.
Steven 94)	Steven,G.B. "The Hatfield Superscalar Architecture," Version 2, University of Hertfordshire Technical Report, September 1994, pp1-58.
Steven 95)	Steven,F.L., Steven,G.B. & Wang,L. "Using a Resource Limited Instruction Scheduler to Evaluate the iHARP Processor," to appear in IEE Proceedings Computers and Digital Techniques, 1995.
Sutherland 89)	Sutherland,I.E. "Micropipelines," Turing Award Lecture, CACM, June 1989, Vol.32, No.6, pp720-738.
Tierno 94)	Tierno,J.A., Martin,A.J., Borkovic,D. & Lee,T.K. "A 100-MIPS GaAs Asynchronous Microprocessor," IEEE Design & Test of Computers, Summer 1994, Vol.11, No.2, pp43-49.