DEPENDENCY AND EXCEPTION HANDLING IN AN ASYNCHRONOUS MICROPROCESSOR

A thesis submitted to the University of Manchester for the degree of Doctor of Philosophy in the Faculty of Science and Engineering

1997

DAVID ALAN GILBERT

Department of Computer Science

Contents

List of figures

List of tables

Abstract

Dependency and exception handling mechanisms are an important part of modern high-performance microprocessors. In a pipelined microprocessor, dependency and exception handling require different stages of the pipeline to interact with each other to determine the current state of the processor as a whole. In a synchronous processor interactions between separate pipeline stages are managed using a global clock. Communication between non-neighbouring pipeline stages is more complex in an asynchronous microprocessor which does not have a global clock.

This thesis describes a solution to this problem in the context of a third generation asynchronous implementation of the ARM instruction set architecture. The architecture described provides powerful and efficient dependency resolution while simultaneously providing a flexible, low overhead exception handling mechanism. The mechanism provides the basis for the architecture of the AMULET3 microprocessor.

Existing exception handling and dependency resolution mechanisms are re-evaluated in the context of asynchronous implementation and the ARM architecture. The Reorder Buffer is chosen as the basis of the architecture and novel enhancements are proposed which enable its use in an asynchronous environment.

Simulation results are presented that show that the proposed architecture is significantly faster and more flexible than comparable architectures while still providing complete compatibility with the ARM instruction set architecture.

Declaration

No portion of the work referred to in the thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright and intellectual property rights

Acknowledgments

I would like to thank all the members of the AMULET research group and of the Department of Computer Science who have helped and encouraged me while I carried out the research described in this thesis.

Special thanks are due to my supervisor, Dr. Jim Garside, who has endured many hours of questioning and arguments relating to this research and this thesis. I am also especially grateful to the head of the AMULET group, Professor Steve Furber, for many interesting discussions on the material presented in this thesis and other aspects of Computer Science.

I am grateful to the authors of a number of free software packages used during the course of his research. These include the GNU set of tools, the Linux operating system and its associated tools, the TGIF drawing package and the VILE text editor.

I would also like to thank Nigel Paver for allowing the use of figure 2.5 from his thesis **[Pave94]** and Professor Steve Furber for the use of figures B.1-B.4, which were taken from his book *ARM System Architecture* **[Furb96]**.

Finally, I would like to thank Dr. Philip Endecott, Andrew Bardsley, Dr. Steve Temple and Professor Steve Furber for proof reading this thesis.

The author

The author graduated from the University of Manchester Department of Computer Science with a 1st class honours degree in Computer Science in 1993. He received the ICL award for the top single honours student in the final year.

In 1995 he was awarded an M.Sc. in Computer Science for research titled “Rendering Architectures for use in Virtual Reality Systems” which was carried out in the Advanced Interfaces Group of the University of Manchester Department of Computer Science.

Since then the author has been conducting research on computer architecture in the AMULET research group, culminating in this thesis.

Chapter 1: Introduction

This thesis describes the architectural design of a third generation asynchronous implementation of the ARM microprocessor architecture **[ARM96]**. In particular the thesis describes techniques for dependency and exception handling. The work described was performed in the AMULET research group at the University of Manchester and will be used (with some modification) in the AMULET3 microprocessor.

Implementations of a commercial processor, such as the ARM, must include solutions to many problems which do not exist in idealised processor designs. The most important of these problems is maintaining total compatibility with the original processor specification. Maintaining compatibility often causes added complication and a loss of efficiency but is required to make the implementation commercially useful. The techniques described in this thesis address these compatibility issues and show that it is possible to produce an efficient asynchronous implementation of the ARM compatible with the synchronous ARMs.

1.1 Synchronous and asynchronous design

Conventional synchronous design styles rely on a single synchronising clock. The clock cycle is sufficiently long to ensure that all blocks have read their input data, processed it and placed their result on the output before the next clock edge.

In asynchronous design there is no clock and so another mechanism must be used to indicate when data is available for a block to process. A *handshake* is used to synchronise two blocks before data is transferred; various types of handshake are described in section 1.4. The handshake ensures that a block is told when data at its inputs are valid, when the blocks connected to its outputs are able to accept data, and when they no longer require the data. The handshake produces local synchronisation between blocks during data transfer; this has replaced the global synchronisation of all blocks in a synchronous system.

1.2 Arguments for asynchronous design

Many possible advantages have been claimed for asynchronous design, including:

1.2.1 Clock skew avoidance

Synchronous designs rely on clock edges reaching all sections of the design 'simultaneously'. However, in practice propagation delays make this impossible. The difference in time between the clock reaching various sections of a design is referred to as the *clock skew*. Since the data must be available to all blocks by the time the clock edge reaches the destination, the clock skew must be subtracted from the total cycle time to give the amount of cycle time available for processing. As designs get more complex the clock must be distributed to increasing numbers of gates in the design; also, as designers simplify pipeline stages so that clock speeds can be increased the fraction of the clock cycle taken up by clock skew increases and thus the impact of clock skew on performance increases. These requirements lead to increasingly large, complex and power hungry clock drivers.

1.2.2 Better than worst case execution time

In a pipelined microprocessor each stage of the pipeline may take a different time to complete its work. In the synchronous world the pipeline must be clocked slowly enough for the worst case time of the slowest stage. An asynchronous design, however, can take advantage of variations in the completion times of the stages caused by varying data. For example, an ALU can complete an addition with a small carry propagation distance faster than one with a long carry propagation distance**[Gars93]**. It is therefore possible that given suitable data the pipeline will execute instructions faster than the worst case.

1.2.3 Power considerations

In CMOS VLSI most of the power is expended when a node changes voltage level. In a conventional clocked system all sections of the design are clocked simultaneously, including those which are not in use on all cycles (for example a multiplier in a typical CPU). This causes many nodes to toggle unnecessarily. Clock gating can be used to reduce this effect but at the expense of increased complexity and clock skew. In an asynchronous design transitions only occur when a block is being used, thus reducing the power consumption.

1.2.4 Electromagnetic compatibility (EMC)

Clocked systems radiate harmonics of their clock over a wide range of the radio frequency spectrum. The interference manifests itself as a large number of relatively high intensity signals at multiples of the clock frequency. In systems which utilise radio communication (such as mobile telephones) this can degrade the performance of the radio receivers significantly. It is strongly believed that asynchronous systems produce broadband distributed interference without the high amplitude peaks. This may make such designs more suitable for use in radio communication equipment.

1.2.5 Modularity of design

It is claimed that asynchronous design leads to more highly modular designs, since each block can be designed without knowledge of the timing characteristics of the other blocks in the design **[Suth89]**.

1.3 Problems with asynchronous design

While asynchronous design has many potential advantages, it also has a number of disadvantages which can make it harder to design with; some of these disadvantages will be explained in this section.

1.3.1 Control logic complexity

In asynchronous designs the control logic performing the synchronisation between blocks often becomes significant in area, power and time. Every block of the design needs hardware to perform synchronisation to wait for input data and to trigger other blocks when it has produced its data. To take advantage of data dependent processing times completion detection circuitry is needed to detect the end of computation in each block.

1.3.2 Testability

At the time of writing, testing asynchronous circuits to check for the presence of fabrication faults is difficult **[Petl96]**. The problems stem from the number of state holding elements used to implement the asynchronous handshakes and from the absence of a clock. In testing synchronous circuits it is useful to stop the clock, apply a test pattern, cause another clock edge and then observe the results. Without such a clock it becomes complex to apply test patterns.

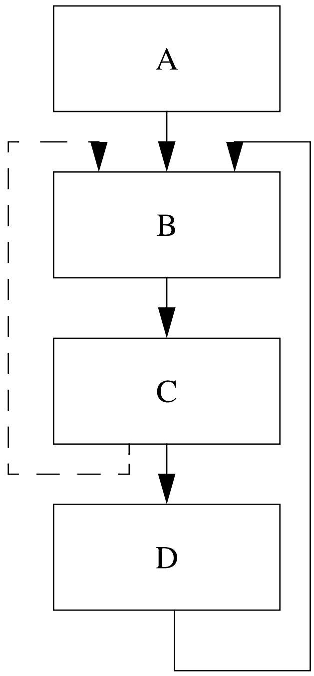


Figure 1.1 Global synchronisation

1.3.3 The risk of deadlock

The use of explicit communications between blocks makes it easy to introduce design errors which cause deadlocks in asynchronous systems. Such errors are difficult to reason about, difficult to detect during simulation and often involve situations caused by the interoperation of many parts of the design.

1.3.4 The loss of implied knowledge

The clock in a synchronous system can be thought of as providing both local and global synchronisation. Local synchronisation (i.e. between two adjacent blocks in a pipeline) is easily replaced by the handshake. However, the use of the clock as a global synchronisation mechanism is more difficult to replace. Figure 1.1 shows a model of a pipeline similar to that of a simple microprocessor with result forwarding of the type which will be encountered later in this thesis. In this system results generated by block C can either pass through block D before returning to block B or can be forwarded on the fast (dashed) path directly from block C. In a synchronous system the position of an instruction and its result is deterministic, so that in a particular system an instruction issued in the present cycle by block A will be at the output of block B in the next cycle. Thus if block B wishes to use the result of the instruction which it processed on the previous instruction it can wait for just one cycle and know that it can use the result via the fast path from C. In an asynchronous system once a packet is put into the head of a section of pipeline there is no way of knowing where it will be at any time later. To be able to reuse a result from later in the pipeline an explicit synchronisation is needed. However, causing the pipeline always to be synchronised causes lockstep operation with the pipeline operating at the speed of the slowest block. The alternative is to synchronise only when the result is needed; however this introduces complicated control logic which can reduce the throughput of the pipeline and it may not be possible to know that synchronisation is required before sending the current packet down the pipeline.

This thesis treats asynchronous design as a technique which requires investigation and examines some of the potential problems associated with it. However, it is not the intent of the author to judge whether asynchronous design is better or worse than synchronous design for any particular task.

Very few of the theoretical benefits of asynchronous design have been demonstrated. However the AMULET2e microprocessor **[Furb97]** shows that asynchronous designs can be competitive with their synchronous counterparts.

1.4 Asynchronous handshaking

This section explains two handshaking mechanisms which have been used in asynchronous designs; these are *two phase* and *four phase* handshakes **[Suth89]**. Both mechanisms use a pair of control lines, called *request* and *acknowledge,* between the two blocks which must synchronise. Handshakes can be used on their own to synchronise a pair of blocks or they can be used with a bus carrying data to indicate the validity of the data in a *bundled data protocol*.

In the two phase handshake the sender informs the receiver that it has data to transmit by inverting the state of the *request* line; the actual voltage on the line is irrelevant, as is the direction of the transition. When the receiver observes this change it processes or stores the data and then toggles the *acknowledge* line to indicate that it has finished with the data. An example of this type of handshake is shown in figure 1.2.

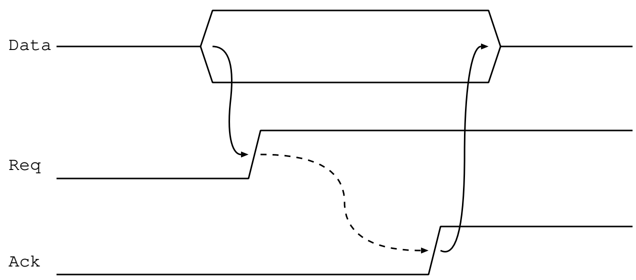


Figure 1.2 A two phase handshake

The case just described represents the operation of a block *pushing* data to the next block. An alternative is for the destination to *pull* data from the sender by placing an event on the request line and waiting for the sender to acknowledge.

The *four phase* handshake is shown in figure 1.3. Here data is placed on the output of a block and the *request* line is raised, the receiver acknowledges by raising *acknowledge*. The sender then lowers request and finally the receiver lowers acknowledge. While there are more transitions in the handshake causing it to expend more energy, this technique leads to the simplification of many common control structures **[FuDa96]**.

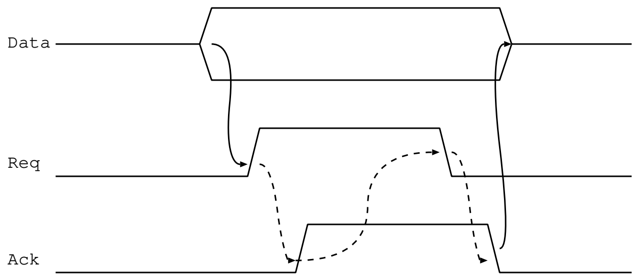


Figure 1.3 A four phase handshake

There are a number of variations on the basic four phase handshake model which differ in the period in which the data is valid; these are illustrated in figure 1.4. In the *early protocol* data is valid between Req rising and Ack rising. The *broad protocol* defines data validity being from Req rising until Ack falls and the *late protocol* defines data validity being between Req falling and Ack falling. The choice of the appropriate protocol to use varies with the particular application. In particular the *broad protocol* is useful in systems incorporating dynamic logic **[FuLi96]**.

1.5 Micropipelines

Sutherland **[Suth89]** suggests a form of asynchronous pipeline called *micropipelines* based on the handshaking mechanisms described above; this is shown in figure 1.5. Each stage consists of a latch and a controller which produces the request out, acknowledge in and latch control signals based on the incoming request in and acknowledge out signals.[1] The micropipeline acts as an *elastic* FIFO where the number of values in the pipeline is variable. Micropipelines can be enhanced by logic placed between the stages as in figure 1.6. In this model logic is placed between latches and the request line is delayed to account for the logic delay.

1. In common with the normal convention in the AMULET group, the *Request in* and *Acknowledge in* lines are the lines from the current block to the previous block while *Request out* and *Acknowledge out* refer to the lines between the current block and the next block. Thus (confusingly) *Acknowledge in* is an output of the current block.

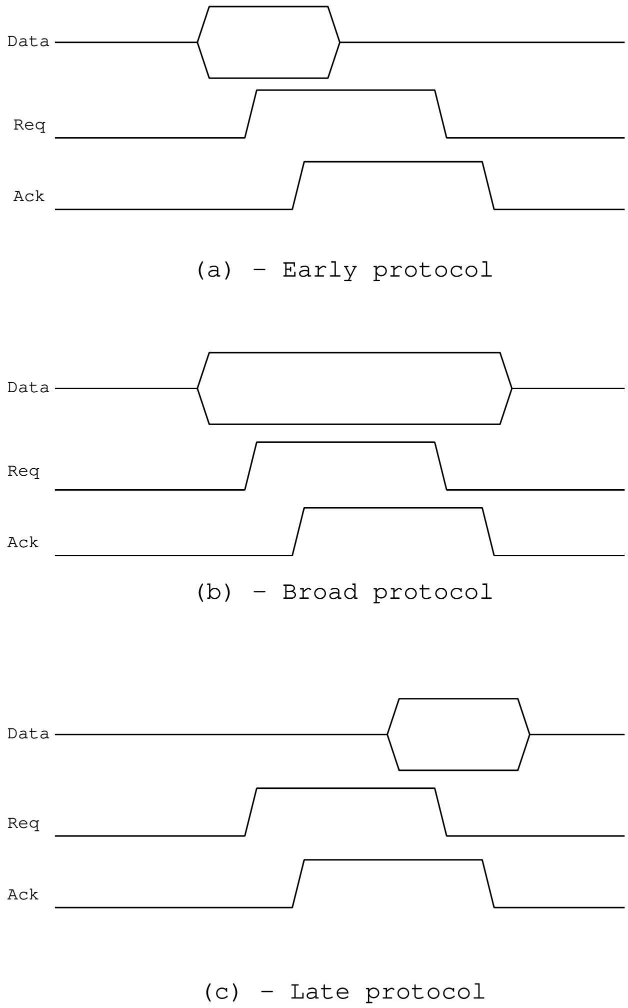


Figure 1.4 Variations on the 4 phase handshake bundled data protocol

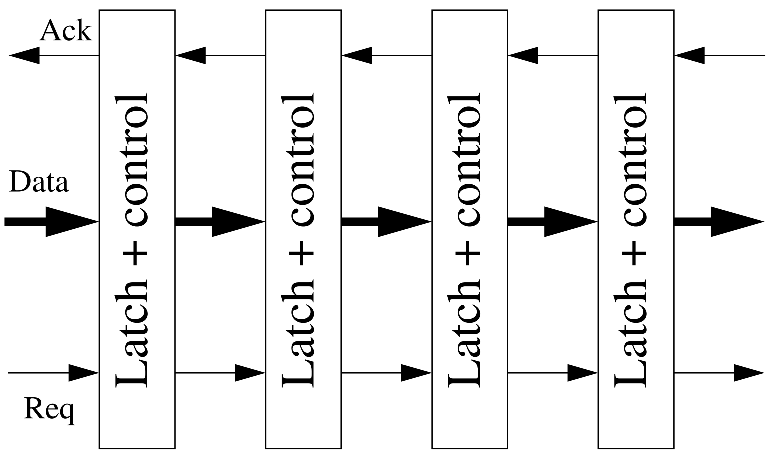


Figure 1.5 Micropipeline

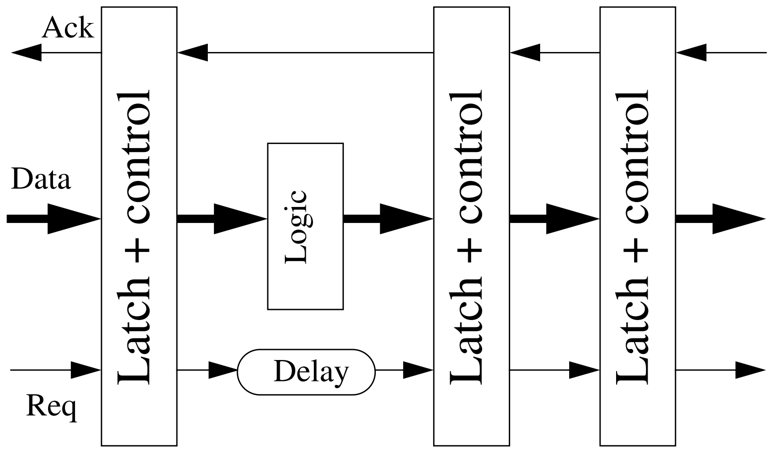


Figure 1.6 Micropipeline with logic

1.6 The ARM Microprocessor

The ARM microprocessor was designed in the early 1980s by Acorn Computers Ltd as a low cost, high performance microprocessor. Its primary design goals were simplicity (due to a small design team) and low power (so that it could utilise a low cost plastic package). In 1990 the ARM design team was split off into a separate company, Advanced RISC Machines Ltd., which has continued to develop the ARM. Since then the ARM has proved popular in embedded systems due to its low power consumption, small die size, and the fact that ARM licenses the processor core to be incorporated into ASICs.

The ARM instruction set is RISC based with only explicit load/store instructions accessing memory and many instructions executing in a single cycle. However, unlike most other RISC processors the ARM has several additional features that increase the expressiveness of each instruction.

Over the years since the ARM was first developed the instruction set has evolved and new features have been added. The work described in this thesis aims to produce a microprocessor that complies with the ARM architecture version 4, which is the latest version of the architecture at the time of writing. The instruction set and its semantics are defined in the ARM Architecture Reference Manual **[ARM96]** (subsequently referred to as the ARM ARM). Backward compatibility with the earlier 26 bit addressing format of the ARM is omitted for simplicity.

Appendix A describes the ARM instruction set architecture (ISA) in detail while chapter 4 describes some of the problems that the ARM design creates. Various features of the ARM ISA will be introduced throughout the thesis as they are needed.

Table 1.1 provides an overview of the ARM instruction set. *Op2* refers to either a constant or a (potentially shifted) register value. Coprocessor instructions are not shown and have not been implemented in this design; discussions on possible extensions to provide coprocessor instructions are in section 8.4.1.

The use of the ARM in this project provides the challenge of solving problems encountered in real microprocessors such as exception handling and coping with instruction set design choices which suited the original design very well but which are hard to copy in different implementations.

1.7 Existing AMULET processors

The AMULET research group was established in 1990 to investigate asynchronous microprocessor design. The head of the group, Professor Stephen Furber, was the hardware designer of the original ARM and thus the ARM was a natural choice for the project.

Data Operations	
ADC Rd,Rn,Op2	Rd := Rn + Op2 + Carry
ADD Rd,Rn,Op2	Rd := Rn + Op2
AND Rd,Rn,Op2	Rd := Rn AND Op2
BIC Rd,Rn,Op2	Rd := Rn AND NOT Op2
CMN Rn,Op2	CPSR flags := Rn + Op2
CMP Rn,Op2	CPSR flags := Rn - Op2
EOR Rd,Rn,Op2	Rd := Rn EOR Op2
MOV Rd,Op2	Rd := Op2
MVN Rd,Op2	Rd := 0xFFFFFFFF EOR Op2
ORR Rd,Rn,Op2	Rd := Rn OR Op2
RSB Rd, Rn,Op2	Rd := Op2 - Rn
RSC Rd,Rn,Op2	Rd := Op2 - Rn - 1 + Carry
SBC Rd,Rn,Op2	Rd := Rn - Op2 - 1 + Carry
SUB Rd,Rn,Op2	Rd := Rn - Op2
TEQ Rn,Op2	CPSR flags := Rn EOR Op2
TST Rn,Op2	CPSR flags := Rn AND Op2
Flow of Control	
B address	PC := address
BL address	R14 := PC; PC := address
BX Rn	PC := Rn; CPSR T bit := Rn[0]
Memory access	
LDR Rd,address	Rd := (address)
LDM Rd,{ register list }	Load list of registers from memory
STR Rd,address	(address) := Rd
STM Rd,{ register list }	Store list of registers to memory
SWP Rd, Rm, [Rn]	Rd := (Rn), (Rn) := Rm
Miscellany	
MLA Rd,Rm,Rs,Rn	Rd := (Rm*Rs) + Rn
MRS Rn,PSR	Rn := PSR (Program status register)
MSR PSR, Rm	PSR := Rm
MUL Rd,Rm,Rs	Rd := Rm * Rs
SMLAL RdLo,RdHi,Rm,Rs	(RdHi,RdLo) := (RdHi,RdLo) +Rm * Rs
SMULL RdLo,RdHi,Rm,Rs	(RdHi,RdLo) := Rm * Rs
SWI value	Perform OS call
UMLAL RdLo,RdHi,Rm,Rs	(RdHi,RdLo) := (RdHi,RdLo) +Rm * Rs
UMULL RdLo,RdHi,Rm,Rs	(RdHi,RdLo) := Rm * Rs

Table 1.1: An overview of the ARM instruction set

The AMULET1 processor was designed as part of the Esprit OMI-MAP project. It was a microprocessor core without cache or other peripherals which presented an asynchronous interface to the outside world, and nearly complied with ARM Architecture version 3. The structure of AMULET1 is described in more detail in section B.4 and **[Pave94]**. AMULET1 was fabricated in 1993-4 and was found to be functional; however its performance was lower than expected.

The AMULET2 microprocessor, designed as part of the Esprit OMI-DE project, built on the architecture of the AMULET1 with minor reworking of sections and the addition of branch prediction and result forwarding. Many stages which used two phase handshaking were redesigned using four phase handshaking in the belief that this would simplify the control logic and increase speed. The AMULET2 core was used in the AMULET2e embedded microprocessor which includes a cache, a synchronous memory interface, and a number of I/O ports. This was fabricated in the summer of 1996 and works well and lives up to the performance expected from simulation results obtained during the later stages of design. The performance is competitive with similar synchronous ARM processors although initial predictions made at the start of the design cycle were overly optimistic.

1.8 Aims of AMULET3

The initial design of AMULET3 was part of the Esprit OMI-DE2 project which had as its objective the development of applications of technology from earlier projects. The implementation of AMULET3 is part of the Esprit OMI-ATOM project.

The initial design criteria were:

- To increase performance by architectural modifications using experience from the AMULET1 and AMULET2 designs.
- To provide the added features which have been introduced in to the ARM family of processors since the AMULET1 design. These include 64 bit multiplica-

tion, 16 bit loads and stores, the Thumb compressed instruction set **[ARM96]**, and debug and breakpoint facilities.

A number of the architectural choices for AMULET3 were made prior to the start of this work, based on experience gained from earlier designs. One of the major choices is the use of separate data and instruction interfaces to memory (or at least to the cache). This decision should allow higher bandwidth memory access and simplify the memory address interface. In AMULET1 and AMULET2 the memory address interface had to generate address streams for both instruction and data requests, and the complexity of this interface proved to be one of the main limiting factors in the performance of the AMULET2.

It was also decided that AMULET3 was to be a major redesign rather than an incremental change to AMULET2. This provided the opportunity to investigate the ideas presented within this thesis.

1.9 Other asynchronous microprocessors

The AMULET is one of a number of asynchronous microprocessors which have been developed. The first was developed by Alain Martin et al. at the California Institute of Technology in 1988 **[Mart93]**. It was a 16-bit RISC-like processor which was designed using a semi-formal approach. The design was initially described as a sequential program which was then converted by a series of transformations into the circuit description. This is a simple microprocessor which does not attempt to handle exceptions.

Over the last few years other asynchronous microprocessors have been designed including Fred **[RiBr96]**, ST-RISC **[DaGY93],** ECSTAC **[ApML95]**, the Counterflow Pipeline Processor Architecture **[SpSM94]**, the Rotary pipeline processor **[MoPW96]**, SCALP **[Ende96]**, HADES **[ECFS95]**, and TITAC-2. The mechanisms used by some of these processors to handle inter-instruction dependencies will be described later in this thesis.

These asynchronous processors range from asynchronous implementations of conventional architectures (in the AMULET designs), novel architectures for implementing existing instruction sets asynchronously (in the Rotary pipeline processor and Counterflow pipeline processor) to completely novel designs using new types of instruction set (in SCALP).

1.10 The structure of this thesis

Chapters 2 and 3 describe the general problems of exceptions and dependencies and explain how existing designs have addressed them. Chapter 4 relates the dependency and exception problems to the ARM processor. Chapters 5 and 6 describe the mechanisms that the author has used to address these problems. Chapter 7 describes performance measurements taken from simulations which show the benefits of the design. Chapter 8 presents conclusions and topics for future work.

Chapter 2: Dependencies

This chapter investigates dependencies between instructions and how they affect the design of a processor pipeline. The various types of dependency and their effect will be discussed, together with a number of ways of reducing their impact on performance.

2.1 Types of dependency

In a simple non-pipelined processor, each instruction completes before the next one starts and so any inter-instruction dependencies are irrelevant. The performance of such a processor is related purely to the time taken to execute each instruction. In a pipelined processor the peak performance is related to the time taken for a stage to perform its processing on the instruction packet; however this peak performance is rarely achieved due to dependencies. The following sections classify dependencies into a number of distinct types.

2.1.1 Procedural dependencies

One of the simplest forms of pipelining is to separate the fetch, decode and execute units into separate stages as shown in figure 2.1. Each stage is separated by a register; these registers are not shown on the diagram. This scheme was used in all ARM microprocessors prior to the ARM 8 **[Furb96]**. All operations involved in the actual execution of an instruction, including register read, arithmetic, memory access and register write back, are confined to the execute stage, thus before the next instruction begins exe-

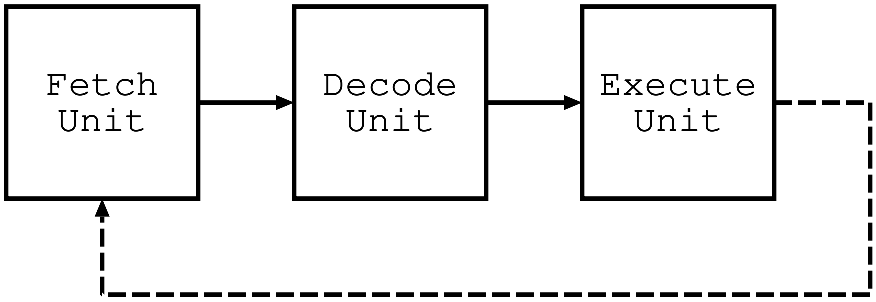


Figure 2.1 A simple pipeline

cution the current one must have completed. The dashed line in the diagram represents a path from the execute unit to the fetch unit to pass the destination address of a branch instruction.

An instruction, such as a branch, which affects the stream of instructions being fetched, causes a dependency. The segment of code shown below illustrates the problem:

```
1)                B exit
2)    alabel      ADD R1,R2,#42
3)                B errorcondition
4)    exit        MOV R6,#54
```

This type of structure might be found in the execution of a multiway branch in a high level language (e.g. a switch statement in C). Instruction 1 in the sequence above represents the end of one branch of the statement, 2 and 3 represent another branch and instruction 4 is the first instruction after the multiway branch.

The fetch unit cannot know the address of the next instruction to fetch until the current one has completed execution since that may be a branch which changes the current instruction stream. Thus the fetching of one instruction is *dependent* on the previous instruction. It is not until the instruction has reached the decode unit that a branch can be detected and not until reaching the execute unit that a conditional branch can be tested to see if it will really execute, so it is not possible to stall the fetch unit when a branch is fetched. Instead of stalling the fetch unit, instructions are always fetched sequentially until the execute unit presents a new address to the fetch unit. Thus the fetch unit is *speculating* that in most cases instructions are fetched in sequence; when a branch occurs the speculation is wrong and the incorrectly fetched instructions must be discarded.

In the simple pipeline design shown in figure 2.1, when instruction 1 (B exit) reaches execute, instruction 2 has reached decode and instruction 3 is being fetched. However, neither instruction 2 or 3 must be executed since they follow the branch. These instructions are said to *lie in the shadow of the branch*. One common way to process the branch is to calculate the branch destination address in the execute stage and transmit it back to the fetch stage, and then to force the execute stage to disregard the instructions in the shadow of the branch. Thus speculative fetching requires extra hardware in the execute stage to deal with cases where instructions have been fetched erroneously.

In the synchronous ARM this branch instruction would take three clock cycles to execute, one for the execution of the branch itself and two more for the new instruction stream to fill the pipeline. It is also possible to branch to an address held in a register or to load an address from memory as the branch target. The last case takes even longer to execute.

Branches constitute around 20% of all instructions fetched on a typical microprocessor **[HePa90]** and thus more effective ways of handling the branch are needed; some approaches are discussed in section 2.2.1.

2.1.2 Read after write (RAW) data dependencies

To increase the throughput of the processor shown in figure 2.1 it may be desirable to pipeline the execute stage and to enable one instruction to start executing before the previous one has finished. One possible structure with pipelined execution is shown in figure 2.2 and is based on that given in **[SmWe94]**. In this model the registers are read in the same pipeline stage as the instructions are decoded and the calculation (labelled *ALU*), memory access and register write back have been moved into separate stages.

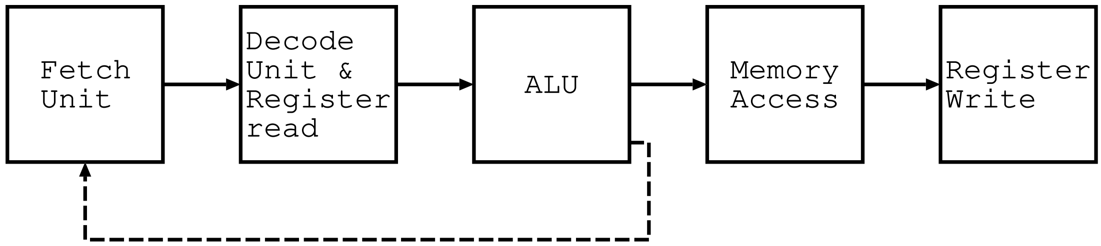


Figure 2.2 Five stage pipeline

A read after write (RAW) dependency (also known as a *true* dependency) occurs when a source operand of an instruction is the destination operand of a previous instruction; for example:

```
1) ADD R1,R2,#8   ; equivalent of R1 := R2 + 8
2) ADD R3,R1,#32  ; Dependent on R1 result of previous
                    instruction
```

The second instruction would reach decode (and thus register read) when the first instruction had only reached the ALU, and if allowed to continue would read an old version of R1. Section 2.2.2 describes how to handle this form of dependency.

Procedural dependencies can be viewed as a form of RAW dependency on the program counter. However, since the program counter is highly predictable, procedural dependencies are managed using different techniques.

2.1.3 Write after write (WAW) data dependencies

Another method of pipelining the execute stage is shown in figure 2.3. In this design memory operations are performed in a separate block off to one side of the pipeline; this design could be compared to a simple superscalar design with a separate functional unit for the memory, however a lot of the complexity is removed since all

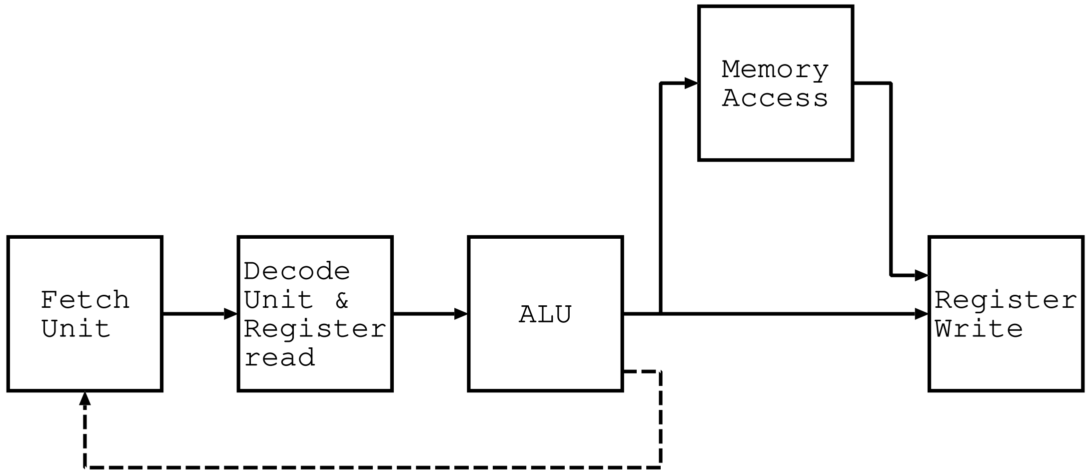


Figure 2.3 Pipeline with out of line memory unit

instructions must pass through the single execute stage. This reduces the latency of normal arithmetic operations by allowing them to pass through one stage less before being written back. Memory operations are sidelined in the memory stage and complete some time later while other instructions continue executing in the main pipeline in parallel. This is advantageous since memory operations are often slower than normal ALU instructions.

This modification introduces *out of order completion*, where an instruction earlier in the instruction stream may complete after its successor; for example:

```
1) LDR R1,[address]
2) ADD R3,R2,#10
```

In this example instruction 1 might write its result back after instruction 2. In this example this out of order write back does not cause a problem. However if two instructions write to the same register and can write back out of order, the wrong final result may be produced:

```
1) LDR R1,[address]
2) ADD R1,R2,R3
3) ADD R4,R1,#65
```

If the load in instruction 1 was to write its result back after the ADD in instruction 2 then instruction 3 will use the wrong version of R1. This sequence of instructions is redundant in that the load generates a result for R1 which is never used, and thus it might be expected that such a sequence would never be used. However such sequences do exist where the second write is after a conditional branch and so is not always executed.

Redundant sequences of instructions are legal in most instruction set architectures and if a design is to be made compatible with such an architecture it must be able to resolve write after write dependencies.

2.1.4 Write after read (WAR) dependencies

Figure 2.4 shows a pipeline with an instruction window which allows instructions to be *issued* out of order, thus allowing non-dependent instructions to issue in the shadow of an instruction which is stalled. In the following example:

```
1)    LDR R1,[address]
2)    ADD R2,R3,R1
3)    ADD R8,R4,R5
```

instruction 3 could be issued while instruction 2 is stalled waiting for the result of instruction 1. This is a powerful technique which is most often used when a large number of pipeline stalls due to dependencies are expected, such as in a superscalar design. There are various mechanisms available for implementing out of order issue; some of these are described in section 2.3.4.

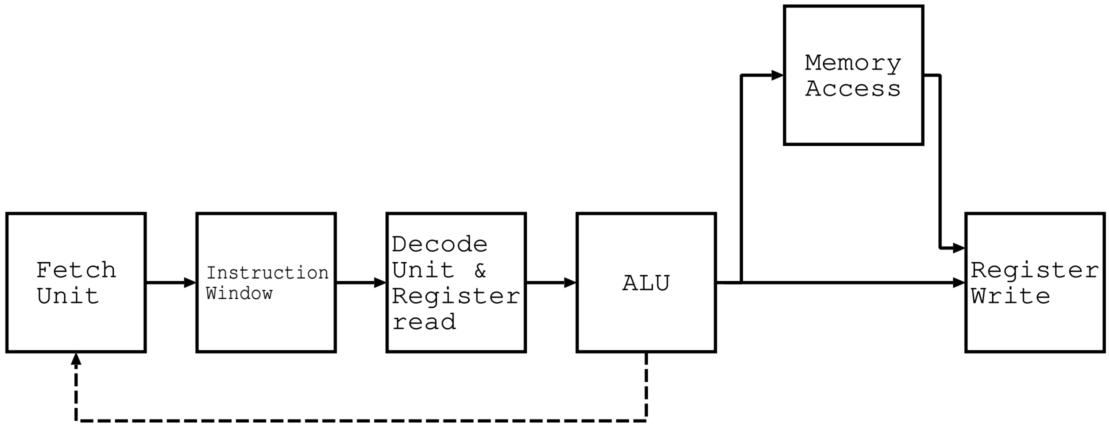


Figure 2.4 Pipeline with out of order issue

The introduction of out-of-order issue has introduced a new type of dependency, the *write-after-read dependency (WAR)* in which a result which has not yet been *read* is overwritten by a later instruction as in the following example:

```
1)    LDR R1,[address]
2)    ADD R2,R3,R1
3)    ADD R3,R4,R5                 Note: destination is now
                                   R3
```

In this example instruction 1 is issued first, instruction 2 has to wait for its result. In the meantime instruction 3 can be issued, however it writes a result into R3 which instruction 2 needs to read to begin execution. Thus instruction 3 would be writing into a register (R3) which has not yet been read. Register renaming, described in section 2.3.3 can avoid WAR hazards completely given enough hardware.

2.1.5 Resource contention

Although not strictly a dependency, resource contention is another source of complexity and performance loss in pipelined microprocessor designs. In the design shown in figure 2.3 one implementation of the write back stage could consist of a single write bus to the register bank which is used for both the ALU and memory results. Hardware must be added to stop two instructions which complete at the same time attempting to use this bus simultaneously. In superscalar designs hardware must be added to stop instructions being issued to functional units which are still executing previous instructions.

2.1.6 Summary of dependency types

In a processor which uses any form of instruction level parallelism, dependencies between instructions can degrade overall performance. In simple pipelined processors where all register modification and access is performed within the same stage the only type of dependency is the procedural dependency. As the level of parallelism is increased (by increased pipelining or by superscalar issue) dependencies on the register bank become more complex.

2.2 Enforcing dependencies

To produce a working pipelined processor it is necessary to add hardware to stall parts of the processor until dependencies have been resolved. Where speculative execution has taken place it is also necessary to add hardware to cope with incorrect speculation. This section presents relatively simple mechanisms to produce a functional pipeline. More complex solutions which attempt to avoid stalling the pipeline at the cost of added complexity will be introduced in section 2.3.

2.2.1 Procedural dependencies

As has previously been stated, it is difficult to stall the fetch stage of the pipeline until the branch has been resolved without losing all the benefits of pipelining and thus it is necessary to ensure that instructions in the shadow of a branch do not affect the operation of the processor. In a simple three stage pipeline (as shown in figure 2.1) this can be done by discarding instructions at the execute stage.

Usually this is simple since the number of instructions in the branch shadow is fixed and thus it is a simple matter to discard this number of instructions in the execute stage. In the AMULET microprocessors this is not possible since the elastic nature of micropipelines causes the depth of prefetch to be variable. The solution is to give each instruction stream a tag known as a *colour*. Each instruction stream present in the microprocessor at any time has a particular colour. When a branch is executed the execute unit generates a new colour and sends this together with the address to the fetch unit. The execute unit then discards instructions which do not match the new colour **[Pave94]**. In AMULET1 and AMULET2 the colour is implemented as a single bit added to the instruction. The added complexity needed is an example of the way asynchronous designs are penalised through the loss of implied knowledge (the fixed prefetch depth) available in synchronous designs. However, the asynchronous design is more modular since changes in the pipeline structure do not affect the operation of this process.

Where the depth of pipelining is greater it may not be possible to discard the instructions before they start executing. In this case it is necessary to ensure that instructions in the shadow of a branch do not change the processor's state.

2.2.2 RAW dependencies

The simplest technique for handling RAW dependencies is to place a *busy* flag on each register. As each instruction is issued it sets the flag on the register it will write its result into. The flag is cleared when the result is returned to the register bank. Instructions are prevented from issuing until the busy bits for all their operands are clear. This scheme is simple to implement in a synchronous system and uses little area.

AMULET1 uses *Lock FIFOs* **[Pave94]** to avoid RAW dependencies. Each lock FIFO is a micropipeline which holds a stream of unary-encoded register numbers. Whenever an instruction is issued the destination register address is placed in the top of the FIFO and whenever a result is returned the bottom element of the FIFO is removed and used as the write enable in the register bank.

In the following example

```
1) ADD R1,R2,R3
2) SUB R4,R5,R6
3) SUB R7,R6,R4
```

at the start of instruction 3 the state of the lock FIFO would be as shown in figure 2.5. Any column containing a 1 represents a register with an outstanding write, and since R4 is outstanding the execution of instruction 3 is stalled. AMULET1 uses two lock FIFOs where one FIFO is used for ALU results and the other for results from memory. The outputs of the two FIFOs are combined to form an overall lock flag; this allows out-of-order completion of instructions flowing down the main pipeline and being processed by the memory.

R7			R4				R0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0

Figure 2.5 The AMULET1 Lock FIFO

The lock FIFO simultaneously solves the problem of RAW dependencies and provides a place to store the write register identifiers for all outstanding writes. More details can be found in **[PDFG92]**.

2.2.3 WAW dependencies

Write-after-write dependencies can be handled by a simple extension to the RAW techniques given above. Instructions are only allowed to issue when their destination register is not busy. This is fairly simple to integrate into both systems described in section 2.2.2 and is incorporated in the AMULET2 microprocessor.

2.2.4 WAR dependencies

One method of solving WAR dependencies is to attach a *Read flag* to each register which is set whenever an instruction is considered for execution which needs to read that register. The flag is cleared when the register reads are completed. WAR dependencies are handled correctly if instructions are not allowed to write their results back until the read flag for their destination is clear **[Ibbe82]**.

2.2.5 Resource contention

Smith and Pleszkun describe a mechanism called the *Result shift register* which can be used to solve contention problems in synchronous designs **[SmPl88]**. The Result shift register consists of a FIFO holding a flag stating where the next result will arrive from and which register it should be written to. At each cycle the last entry is read from the FIFO and used to read a result from a particular functional unit and write it into the register bank. As instructions are issued each instruction type will be known to take a fixed number of cycles (for example an integer add might take two cycles). An instruction taking ‘n’ cycles reserves the *n*th element in the FIFO (i.e. the element ‘n’ cycles from the head of the FIFO). If the element is full it is known that another functional unit will be using the result bus in that cycle and thus the current instruction must be delayed for a cycle.

This technique is not suitable for use in the asynchronous world since it is not possible to predict how long each functional unit will take to complete its calculation and thus the order in which results will return. However, in an asynchronous environment arbitration can be used to resolve the contention when the results arrive at the register bank as shown in figure 2.6. Arbitration is often difficult to reason about and difficult to test, but it does provide a simple solution to this problem. The solution is simpler and less expensive than the synchronous case which requires storage in the form of the result shift register.

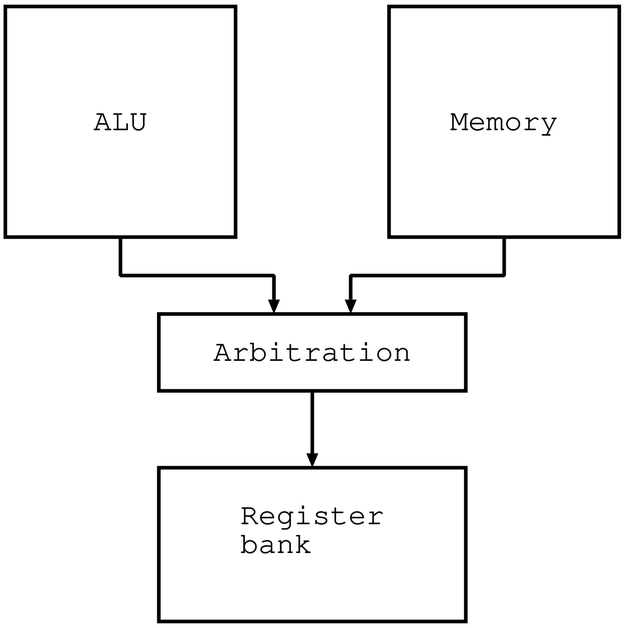


Figure 2.6 Arbitration for a shared register write bus

2.2.6 Summary of dependency enforcement techniques

The greater the potential parallelism in a design the more dependencies are exposed and the more complex the level of hardware needed to enforce dependencies. While enforcing dependencies ensures correct functionality it can easily reduce the overall performance. The next section illustrates ways of reducing the effects of the dependencies to regain some of the performance lost.

Although solutions to procedural dependencies, various forms of data dependency and resource contention have been described separately, it is normally not possible or desirable to implement them in isolation. In particular the effect of branches on other dependency mechanisms can be quite complex, for example:

```
1)                 ADD R4,R2,R3
2)                 B end
3)                 ADD R4,R5,#9
4)                 ADD R9,R1,#1
5)    end          ADD R8,R4,#2
```

In this example if instruction 3 is issued it would lock register R4, however instruction 3 will never complete execution since it is in the shadow of a branch and will be abandoned. When instruction 5 is issued it would wait for the new version of R4 being generated by instruction 3. Clearly the hardware which handles procedural dependencies must ensure that the other dependency mechanisms are kept consistent with the set of instructions which have actually executed.

2.3 Reducing the effect of dependencies

This section describes some of the techniques available to reduce the effects of dependencies. These techniques often trade off circuit complexity against performance. Their suitability often depends on the particular circumstances as to whether the extra complexity can be justified. The techniques either attempt to reduce the time that the pipeline is stalled due to a dependency or they attempt to execute other instructions while one instruction is stalled.

2.3.1 Procedural dependencies

Techniques for reducing performance losses due to procedural dependencies can be split into those techniques which reduce the number of instruction stream changes and those techniques which reduce the performance loss due to each change.

2.3.1.1 Branch delay slots

In synchronous processors where the depth of prefetch is known it is possible to define a branch so that a certain number of instructions in the shadow of a branch are always executed. Compilers can normally reorganise the instruction stream by moving an instruction from before the branch into the delay slot. This technique hides some of the cost of the dependency and allows the processor design to be simplified since it no longer requires hardware to deal with the dependency. Compilers normally can only take advantage of one branch delay slot **[John91]** and so this is not a complete solution to the problem.

Branch delay slots expose the detailed design of the processor to the programmer. This is dangerous since while branch delay slots may be beneficial in one implementation they may be difficult to add in another. Any variation between implementations would remove code compatibility.

2.3.1.2 Split branches

A conventional branch instruction can be split into two separate subinstructions. The first calculates the address to branch to, often by adding or subtracting an offset from the current program counter value. The second operation is informing the fetch mechanism of the new address. One way to reduce the performance impact of procedural dependencies is to use separate address calculation and stream change instructions. The program may include unrelated instructions between the address calculation and the stream change; and thus the address calculation can be performed in parallel with these other instructions. In addition the instruction which causes the stream change (often called *doit*) can be encoded in a way simple enough to be detected by the fetch unit thus reducing or removing the need to cancel wrongly prefetched instructions.

Split branches can be used in asynchronous processors; however, they do require special features in the instruction set and so are not applicable to the reimplementation of conventional processors. The *Fred* processor is one asynchronous microprocessor which uses split branches **[Rich96]**.

2.3.1.3 Branch prediction

As explained in section 2.1.1, the fetch unit works autonomously fetching a stream of instructions, although it may be that a branch has occurred which renders that stream invalid. Prefetching is based on the premise that the most likely address for the instruction after the current one is the next sequential address. This is a simple form of prediction which presumes that branches are never taken; when branches are taken some recovery mechanism must be used. An improvement on this scheme is to predict the outcome of each branch and to predict the destination address of the branch; this is *branch prediction*.

One branch prediction technique is the *Branch Target Cache* (BTC) **[York94]**. This technique attempts to reduce the number of instruction stream changes due to branches, but does nothing to reduce the effects of each change. In a conventional prefetch unit an incrementer generates each new address which is passed to memory; a new address stream may replace the value of the incrementer when a branch has occurred. Figure 2.7 shows a prefetch unit with a BTC; the BTC contains a mapping from addresses of branch instructions to the addresses of their targets. When a branch instruction is encountered by the execute unit it passes the address of the branch and its destination to the BTC which stores it. As each instruction address is generated the address is fed back to the incrementer and the BTC; if the address of the instruction corresponds to a branch which is in the BTC the next instruction will be fetched from the destination of that branch. A flag is added to the branch instruction to indicate whether it

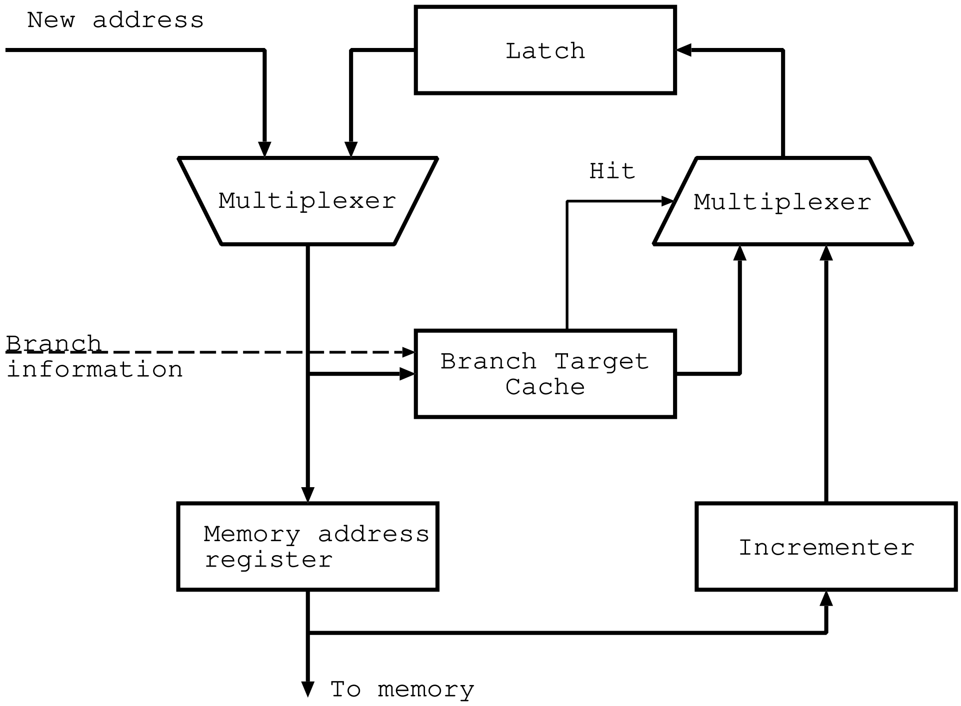


Figure 2.7 Instruction fetch unit with BTC

was predicted; this is used to ensure that predicted branches really were taken and to cause their effects to be reversed if they were not taken. The BTC has only a finite amount of space and various algorithms can be used to determine which entry to replace when a new branch is executed. In many programs a BTC can predict 80-90% of the branches **[HePa90]**, thus significantly reducing the number of instruction stream changes and hence reducing the cost of branches.

This technique is unique to procedural dependencies since the instructions fetched are highly predictable; however there are some common cases which the BTC cannot deal with. In particular branches which use addresses held in a register or that are loaded from memory cannot be predicted; such branches are commonly used as subroutine return instructions.

2.3.2 RAW dependencies

Unlike the program counter value, normal register values are unpredictable, so the technique used to reduce procedural dependency costs is not applicable to normal RAW dependencies. The simplest approach to reducing RAW dependencies is to make the compiler aware of the costs of each instruction and for it to rearrange the instructions to move dependent instructions as far apart as possible, for example:

```
       (a)                            (b)
1) LDR R1,[address]           1) LDR R1,[address]
2) ADD R3,R3,R1               2) LDR R2,[address]
3) LDR R1,[address]           3) ADD R3,R3,R1
4) ADD R4,R4,R1               4) ADD R4,R4,R2
```

In the code fragment (a) in the above example instruction 2 is dependent on instruction 1, and instruction 4 is dependent on instruction 3. Fragment (b) performs exactly the same task, however each instruction which consumes a value is one instruction further from the instruction which generates the value than in fragment (a). Note how fragment (b) requires an extra register and so may be more difficult to implement on architectures with a small register bank. Rearranging instructions at compile time helps to reduce the problem, but does not completely eliminate it.

2.3.2.1 Result forwarding

One method for reducing the cost of RAW dependencies is to make results available for reuse sooner by using *result forwarding* as shown in figure 2.8. With result forwarding the result of an instruction can be reused before it has reached the write back stage. Extra result paths are added between the output of each stage and the register read stage.

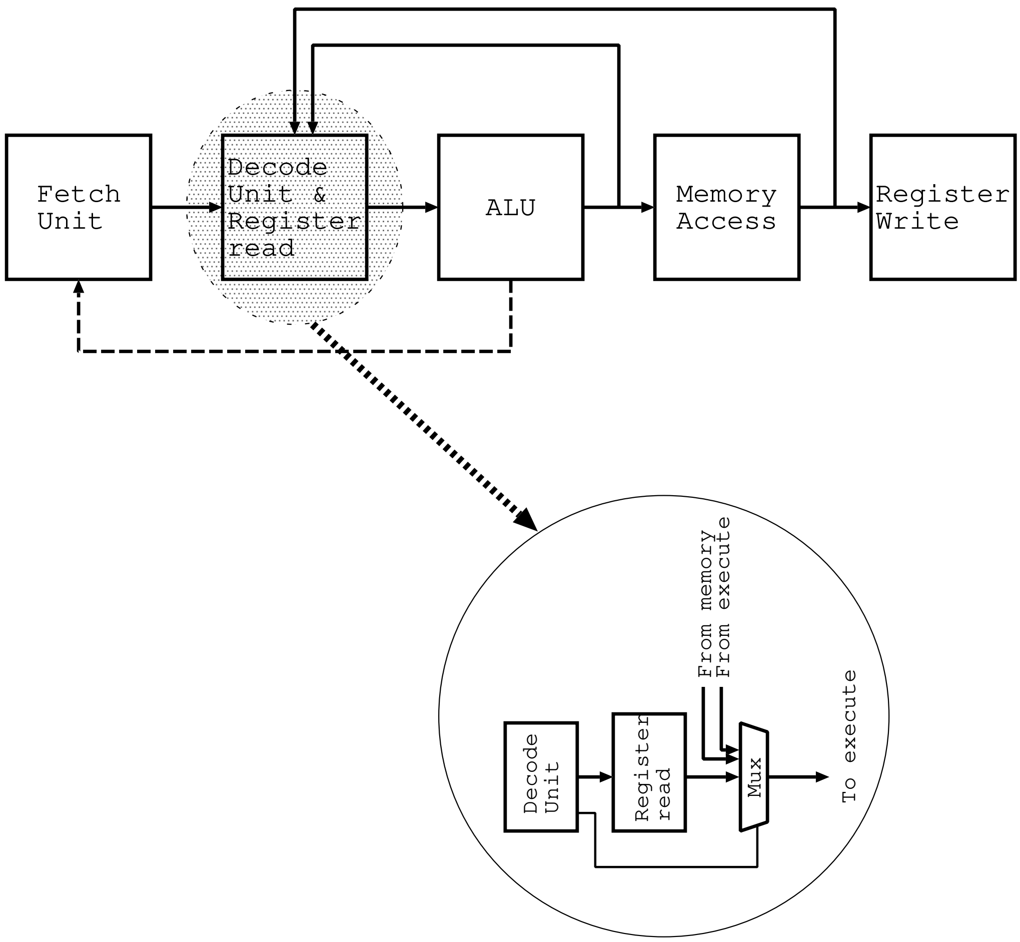


Figure 2.8 Pipeline with result forwarding

The decode unit keeps track of which stage each instruction is in; if an instruction whose operands have been calculated but not yet written back enters decode, decode reads the operand from the output of the stage which the source instruction has just passed through. A set of multiplexers placed after the register bank substitutes the result in place of the value read from the register bank.

This mechanism requires a significant amount of wiring to route the results to earlier stages in the pipeline and logic to keep track of the position of results. As the number of stages in the pipeline increases and the number of register read ports is increased the amount of hardware required increases considerably.

Result forwarding as previously described cannot be used in asynchronous systems since once an instruction has been inserted into the pipeline its rate of progress along the pipeline is variable and thus the decode cannot know which stage to read its result from. One solution to this is to force synchronisation between the register read of the following instruction and the result production of the current instruction. This effectively causes lockstep operation reducing the advantage of variable execution time present in asynchronous systems. Ideally synchronisation would only be used where the result of an instruction actually needs to be forwarded.

2.3.2.2 Last result reuse on AMULET2

AMULET2 uses a technique called *Last Result Reuse* as an alternative to result forwarding. The decoder keeps a record of the destination register of the previous instruction. When an instruction is decoded its source register numbers are compared with the destination of the previous instruction and if they match the register read is bypassed and the operand is collected from the output of the execute stage. No synchronisation is needed because the reuse path is held completely within the execute stage so that results are forwarded from the output of the ALU back to its input. Since the ALU is not pipelined, the output of the ALU at the start of one instruction is the result of the previous one. This technique is only used if the instruction from which the result will be forwarded is unconditionally executed.

In addition AMULET2 includes a *Last Memory Result Register* which holds the last result returned from memory. Results are forwarded from this register when an instruction following a load requires the loaded value. A lock FIFO is maintained for this register in a similar manner to registers within the register bank **[Furb96]**.

2.3.2.3 Fred's R1 queue

The Fred processor designed at the University of Utah uses a conventional scoreboard mechanism **[HePa90]** to manage general dependencies, however it also incorporates a queue of results to be used by later instructions known as the *R1 Queue* **[Rich96]**. The register identifier R1 is used to access a FIFO queue instead of the register bank. An instruction specifying R1 as the destination causes the result to be stored in a FIFO queue which can be read by another instruction reading from R1. Read after write dependencies are handled by simply stalling the processor if the R1 queue is read while empty. A similar mechanism is provided to hold branch destination addresses using a split branch mechanism similar to that described in section 2.3.1.2.

2.3.3 WAR and WAW dependencies

The performance degradation caused by both WAR and WAW dependencies can be reduced by *register renaming*. Consider the following example which causes a WAR dependency:

```
        (a)                     (b)
1)      LDR R3,[address]        LDR a,[address]
2)      LDR R1,[address]        LDR b,[address]
3)      ADD R2,R3,R1            ADD c,a,b
4)      ADD R3,R4,R5            ADD d,e,f
```

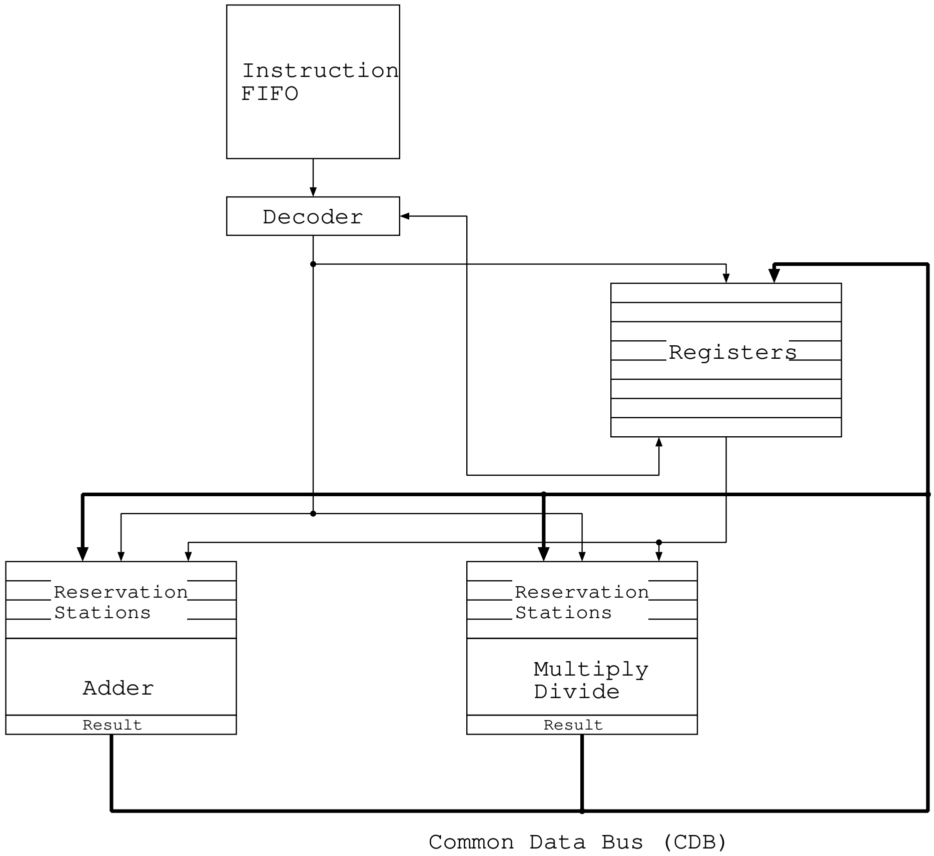
The decoder must prevent instruction 4 from completing before instruction 3 has started execution because instruction 4 would overwrite R3 which is needed by instruction 3. This is a WAR dependency and it is the only reason that the issue of instruction 4 would be delayed in an out-of-order issue architecture executing this example. This delay can be eliminated by providing separate logical and physical register identifiers and a larger number of physical registers than logical registers. As each instruction is issued a physical register is allocated to the result that the instruction will generate and a mapping is created between that physical register and the logical register identifier in the instruction destination field. Also during issue logical register numbers in operands are replaced using the logical to physical register identifier mapping. Section (b) of the example above shows the same instruction sequence after this mapping (lower case letters have been used to represent physical register numbers, ‘e’ and ‘f’ having been allocated prior to the code shown). Thus R3 in instructions 1 and 3 is mapped to physical register ‘a’, while R3 in instruction 4 is mapped to physical register ‘d’. Thus since instruction 4 overwrites a physical register which is not currently in use the WAR dependency has been eliminated.

One form of register renaming, used in the MIPS R10000 **[SmSo95]**, consists of a large register bank containing more registers than are needed by the logical register file and a mapping table which translates logical register addresses to addresses in the physical register file. The *Reorder buffer*, discussed in section 3.4.4, can also be regarded as a form of register renaming.

2.3.4 Out of order issue

In superscalar processor designs which allow large amounts of parallelism, pipeline stalls due to dependencies degrade performance severely. Since the effect of stalls in these systems is so high, extra hardware may be added to perform useful work while instructions are stalled. *Out of order issue* allows instructions to be issued while others are stalled due to dependencies.

One of the best known out of order issue techniques is *Tomasulo’s algorithm* **[Toma67]** which is shown in a simplified form in figure 2.9(a); this was first used in the floating point unit of the IBM 360/91.



(a)

(b)

Figure 2.9 Tomasulo's algorithm

As the decoder issues instructions they are placed in *Reservation Stations* associated with each functional unit with a copy of any of their source operands which are already available; the format of a reservation station entry is shown in figure 2.9(b). Any operand which is not available is replaced by a *Tag* which points to the reservation station holding the instruction which will produce that operand.

When all operands for an instruction are present and its functional unit is free it may execute. Thus a functional unit may have an instruction which is waiting for operands and another instruction (issued later) which already has operands available; this second instruction may execute even though the earlier instruction is stalled. After execution an instruction broadcasts its result to all the reservation stations and the register bank along the *Common Data Bus (CDB)* together with the reservation station number in which it was stored prior to execution. All reservation stations include comparators which compare the tag on the CDB with the tags in their entry and if a match is found the data on the CDB is stored in the reservation station.

Tomasulo's Common Data Bus algorithm allows multiple functional units to execute instructions, potentially out of program order. The inherent tagging system is another example of a register renaming technique which allows WAR and WAW dependencies to be overcome. Two disadvantages of the system are that with a large number of high throughput functional units contention for the CDB could slow the system down and that a large number of tag comparisons are required.

Tomasulo's algorithm is difficult to implement efficiently in an asynchronous design due to the way values are broadcast to multiple units. In an asynchronous implementation the producer of a result would have to synchronise with all consumers (including all other functional units) to ensure that they have had time to read the data from the bus.

2.4 Dependencies and external state

The processor registers are not the only state holding elements in a computer; the memory subsystem, processor extensions (i.e. coprocessors) and input/output devices also hold state and dependency mechanisms must include support for them.

An example of dependencies occurring in state external to the processor is a store followed by a load some time later from the same location. Many processor designs treat the memory as a functional unit which processes requests in the order which they are passed to it, and leave it to the memory to resolve dependencies. However, as with state internal to the processor, the problem grows as the level of parallelism grows.

One example of a memory dependency common to all pipelined processors is that of a program which loads a new section of code to be executed. The running program writes code to memory using conventional memory access instructions and then calls the code it has just written; this may happen whenever a program is loaded from storage, or it may be used to generate segments of code specific to a problem encountered at run time; the latter is known as *self-modifying code*. The dependency occurs because the processor may already have prefetched from the destination of the writes before the writes have completed and thus the processor will execute the instructions which were previously in memory.

In practice, self modifying code is rare and when it does occur the code is often executed a long time after it was written and so the writes have already occurred; however when separate instruction and data caches are present this can become a problem. Since loading code is reasonably rare this problem is normally solved by requiring the operating system to perform a special instruction after it has written the code to be executed.

Other forms of external dependency can affect just data accesses. The inclusion of write buffers and write back caches to increase the performance of the memory system can both cause RAW dependency problems.

Some of the techniques used to process register dependencies are unsuitable for use on the memory subsystem. In particular locking individual registers is unlikely to be useful for a memory system since adding a lock to each memory location would be prohibitively expensive.

2.5 Novel approaches to dependency resolution

A number of novel approaches to the problem of dependency resolution have been introduced in asynchronous microprocessors. Although these techniques are aimed at asynchronous implementation they are often also applicable to synchronous implementation.

2.5.1 SCALP

SCALP is an asynchronous superscalar processor which takes a novel approach to dependency resolution **[Ende96]**. SCALP consists of a number of functional units interconnected by a routing network and an instruction issuer which distributes instructions among the functional units. Each instruction states only which functional unit it should be executed by, which subfunction of that unit is needed, and where to send the result. In the following example the LOAD instruction tells the memory unit to wait for a memory address (which we will assume a previous instruction has generated) and use it to load a value from memory. This result is then sent to the input port on the duplicator. The DUPLICATE instruction tells the duplicator to wait for a value and then send it to both the input ports on the ALU. The ADD instruction tells the ALU to wait for two input values, add them and then pass the result to the memory unit.

```
LOAD -> duplicator
DUPLICATE -> alu_a, alu_b
ADD -> mem_addr
```

Read after write dependencies are handled locally by each unit; thus the ALU waits for a value on both of its inputs. It is possible that more than one instruction passes results to the same functional unit and that there is no other constraint which determines the arrival order of the results; in this case the program would execute non-deterministically. To solve this problem a sequencing instruction is provided which the compiler must use wherever non-determinism is present.

The instruction issuer in SCALP is not simple, however it would be significantly more complex if it had to determine inter-instruction dependencies.

SCALP has reduced the hardware costs for dependency analysis by increasing the burden on the compiler. It is up to the compiler to find non-deterministic instruction sequences and insert sequencing instructions and also for the compiler to distribute instructions among functional units. One disadvantage is that code compiled for one implementation of SCALP would not execute on a SCALP with a different mix of functional units.

2.5.2 The counterflow pipeline processor architecture

The Counterflow Pipeline Processor Architecture (CFPP) developed at Sun Microsystems Laboratories is a novel processor architecture which enforces dependencies using only local communication **[SpSM94]**; Figure 2.10 is a simplified representation of the CFPP. The lack of global communication makes it an interesting architecture for implementation using asynchronous techniques. The pipeline has two streams of data running in opposite directions; instructions flow upwards and results flow downwards.

A block of logic called the *Cop* between each stage regulates the flow of data between the stages. As instructions and results meet in the *Cop* results are copied into instructions which use them, and newer results from executed instructions replace older

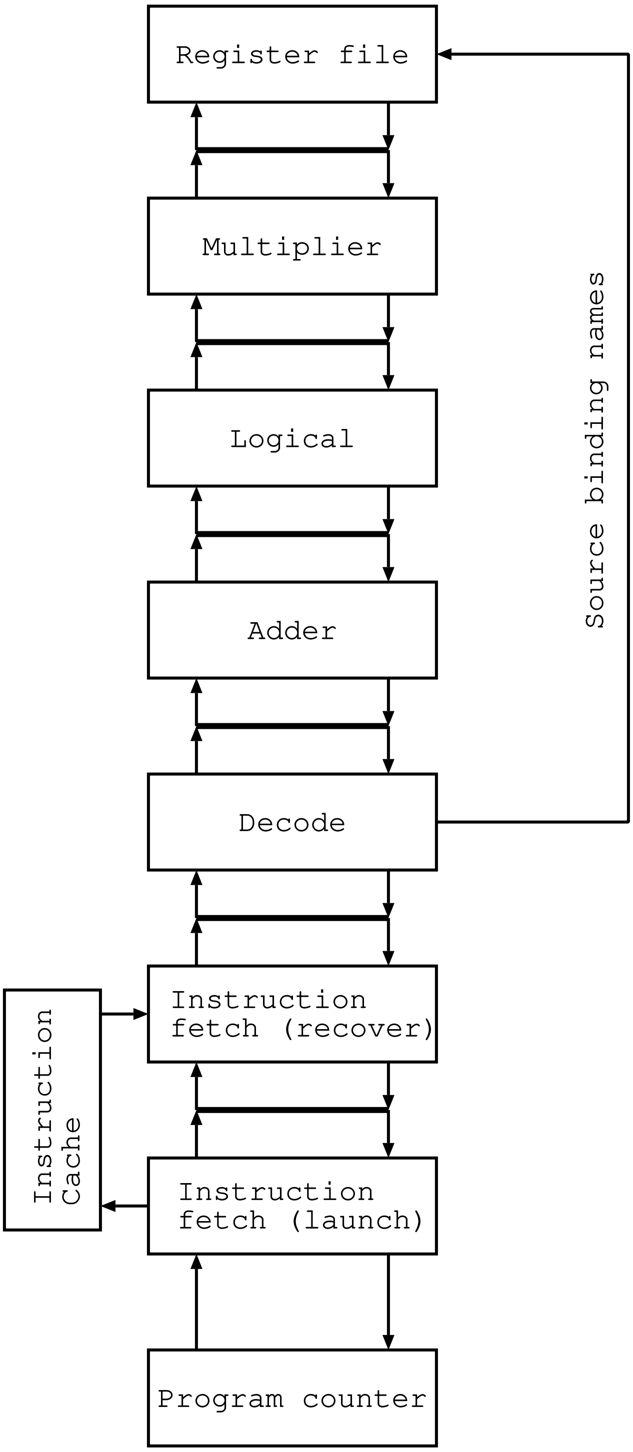


Figure 2.10 A simplified model of the CFPP

results in the result stream. The logic is such that results cannot overtake instructions which use the results and so all instructions are guaranteed to receive the latest version of the result. More details can be found in **[SpSM94]**.

The main disadvantage of the design is the amount of logic needed between each stage to route results between instructions.

2.5.3 Hades

The *Hades* processor developed at the University of Hertfordshire is an asynchronous superscalar processor **[ECFS95]**. Its main features are:

- Boolean register file

  Hades has two register files; an integer file and a boolean file. All comparisons write their results into the boolean register file and conditional branches use boolean registers as their condition.

- Explicitly declared delayed branching

  Each branch instruction has a delay count incorporated into the instruction, thus the compiler can state how many branch delay slots each branch is to use. This enables the compiler to tune the number of delay slots to the number of instructions it can find to put in the branch delay slots on a per branch basis.

- Decoupled Operand Forwarding

  Each functional unit has associated with it a forwarding register. Whenever a result is written back to the register file it is also written into the forwarding register. During the instruction decode stage a note is kept of the last register written to by each functional unit and thus the value which will be found (or which is due to be written) in each functional unit's forwarding register. In the instruction decode stage as each instruction is decoded its source operand registers are compared to the register numbers known to be held in the forwarding registers; if a match is found the register bank read is cancelled and the value is read from the forwarding register. The forwarding register may not yet have received the result to be forwarded and thus will stall before providing the forwarded data. The same piece of data can be forwarded from a forwarding register by multiple instructions since the value is not removed from the register once forwarded. A traditional register locking mechanism is used to perform RAW dependency resolution on register bank reads; however this forwarding mechanism enables instructions to be issued before their operands have been generated and thus avoid stalling on a locked register. They can, however, stall later waiting for an operand from a forwarding register.

2.5.4 The Micronet-based asynchronous processor (MAP)

The MAP processor is based on the concept of *Micronet*s which are a generalisation of micropipelines **[ArRe94]**. Each stage of the micronet has its own independent control in the form of *microagents* which communicate with microagents in other stages. The control for these communications is local to the microagents involved. The aim of this generalisation is to increase the parallelism in the design. The designers of MAP implement RAW dependency resolution mechanism with locking in the register bank. Requests to access a register are not acknowledged unless the register is unlocked. A token ring mechanism is used to provide distributed arbitration of access to a results bus. The results on this bus are tagged with information which enables forwarding off this bus in a manner similar to that described in section 2.3.4; however it is not clear how the data validity on this write back bus is guaranteed, thus allowing forwarding from the bus by other sections of the processor. The most notable feature of MAP is the decentralisation of control.

2.6 Special registers

While the discussion in this chapter has treated all dependencies on registers in the same way, real-world processors often have a number of special registers which have different access characteristics from normal data processing registers. These include processor status registers which determine the privilege level of the processor and registers holding carry and sign flags from arithmetic computation. The access patterns of these registers is often very different from normal data registers and thus the mechanisms chosen for handling dependencies may be different.

2.7 Summary

In all processors with instruction level parallelism some dependency resolution mechanism is needed. The techniques described in section 2.2 provide functional solutions to the dependency resolution problem at the cost of performance while those in section 2.3 provide increased performance at the cost of added complexity. This increased complexity may cause the cycle time of the processor to increase and thus it may actually reduce the overall throughput of the design. It is therefore necessary to consider the costs and benefits of various pipelining schemes and dependency avoidance mechanisms carefully to determine the best solution for a particular application.

The inability of asynchronous implementations to construct simple result forwarding mechanisms makes the linear five stage pipeline shown in figure 2.2 undesirable because of its high latency even for instructions which do not access memory. This is one reason why AMULET1 and AMULET2 used a pipeline structure similar to figure 2.3. However, as is evident from the variety of techniques shown in section 2.5, other techniques are available to the asynchronous designer.

Chapter 3: Exceptions

Most instruction sets make provision for handling rare or *exceptional* occurrences such as interrupts, bus errors and arithmetic overflow. To handle these occurrences (which we will describe under the general heading of *exceptions*) extra hardware must be added which can severely complicate the design of a pipelined processor. This chapter investigates the causes of exceptions, their appearance to the programmer and the hardware mechanisms needed to handle them. Later in this thesis these mechanisms will be developed for use in an asynchronous environment.

3.1 Causes of exceptions

Exceptions are caused by many different sections of the processor and system under a number of different circumstances. The exceptions can be characterised according to how often they occur and how often they can *potentially* occur. For example, memory access faults are very rare. However, every memory operation could potentially generate one, and as section 7.1.1 shows these make up around 25% of all instructions.

3.1.1 External interrupts

Interrupts are often generated by I/O devices requesting service from the operating system to transfer some data or initiate a new operation. Systems typically provide a number of different external interrupts with different priorities; a fast, high priority interrupt might be provided for servicing high bandwidth network connections while a low priority interrupt may be provided for handling a keyboard.

3.1.2 Arithmetic errors

Some systems require that an operation that produces a result which cannot be correctly represented must produce an exception to report the error; an example of this might be an arithmetic overflow. Other systems avoid producing an exception by requiring the application to explicitly test for errors after each operation. The IEEE 754 standard for floating point arithmetic (which is widely used) states that both a flag based and a trap based mechanism should be available, though the flag based error reporting is the default **[IEEE85]**.

3.1.3 Undefined/unimplemented instructions

Most instruction sets do not provide definitions for all possible instruction encodings, and many architectures require these *Undefined Instructions* to cause an exception. This allows a program which has crashed and executed these instructions unintentionally to be dealt with by the operating system in an appropriate manner. Similarly some implementations of a particular ISA may not implement all instructions. Complicated instructions like multiplication or floating point operations may be omitted; when one of these instructions is executed the operating system is called to emulate the instruction. This allows the same program to run on a variety of implementations with different sets of features.

3.1.4 Memory access errors

The memory subsystem would produce an exception if it was not *able* to complete an access due to a fault (such as a parity error) or, for example, due to an access to an area of virtual memory not presently mapped to a region of physical memory. It would also produce an exception if it was not *willing* to complete an access if the memory request was made in a non-privileged execution mode to an area of memory requiring privilege. Some memory subsystems can be built with reduced hardware by requiring software intervention for rare occurrences (such as TLB reloads **[STB94]**); these interventions would be signalled using exceptions.

Instruction and data accesses can both cause memory access errors; the effects on the processor of failed data accesses are often different from the effects of failed instruction accesses due to the stage in the pipeline in which the access originated.

3.1.5 Software interrupts

A program running in a non-privileged mode may require the operating system to perform a privileged task (such as accessing hardware); typically the program signals to the operating system by a mechanism similar to an exception often called a *trap* or *software interrupt*.

3.1.6 Unpredicted/mispredicted branches

In a system with branch prediction, branches may cause flushing of the pipeline so rarely that they can be treated in a similar manner to exceptions.

3.1.7 Breakpoints

Many processors and operating systems provide facilities to aid debugging of software under development. The most common provision is that of a *Breakpoint* which is a mechanism to cause an exception when a particular instruction is executed or a particular memory access is made. Normally, after a breakpoint the program is suspended and a monitor is run which allows the programmer to examine the current state of the program before possibly allowing it to continue. In many ways the effect of a breakpoint is similar to a software interrupt.

3.1.8 Reset

Reset of the processor can also be regarded as an exception; however unlike many of the other exceptions it is not necessary to continue execution of the original program after the reset has occurred.

3.2 The effect of exceptions

An exception will cause the processor to:

Prioritise exceptions: Since many architectures support a variety of different exceptions, some of which may occur simultaneously, the exceptions are prioritised so that an exception occurring during the processing of another exception may or may not be acted upon depending on the priority rules. For example, a low priority interrupt occurring during the execution of a high priority interrupt handler may have to wait for the handler to return before its own handler is called.

Complete previously issued instructions: Instructions issued prior to the instruction which caused the exception are normally forced to complete before exception processing continues; however some designs store the state of partially completed instructions.

Save the state of the running program: This is necessary to enable the operating system to continue execution of the program after the cause of the exception has been discovered and dealt with.

Enter a privileged mode: In many architectures programs run in an unprivileged mode without access to I/O devices. Since recovery from many exceptions requires access to I/O devices or protected areas of the operating systems data, the processor is placed in a privileged mode before the exception handling routine is executed.

Entry of the exception handling routine: The section of code which processes the exception is called an *Exception Handler*. The processor calls this routine either by jumping to a predefined location in memory or by reading the address of the handler from a predefined location.

After the exception handler has finished execution the state will generally be restored from that previously saved to allow the original program to continue execution. The system state as seen by the program should be unaffected by the exception. A possible exception to this rule is that the emulation of an unimplemented instruction might cause the visible state to be changed.

Some of these steps may be unnecessary when handling certain exceptions; for example it might not be necessary to save the state for a reset, and it is usually not necessary to enter a privileged mode to recover from a mispredicted branch.

3.3 Cost and frequency of different types of exception

Given the various types of exception described above it is interesting to consider how often the different types occur and how important it is to process them quickly. This information would allow the designer to choose from the various forms of exception handling hardware and trade off the amount of hardware against the expected performance benefit it would bring. Different approaches may be employed for handling each type of exception within the same processor design.

3.3.1 Frequency of exceptions

In a system with branch prediction which correctly predicted 90% of branches and which had programs running upon it which caused a branch every 5 instructions, a mispredicted branch would occur approximately every 50 instructions. Thus if branch misprediction is treated as an exception it must be handled efficiently and speedily since it occurs frequently.

The frequency of external interrupts varies greatly depending on the application and on the external hardware providing support to the processor; however except in the most heavily loaded system the frequency of external interrupts is much lower than the frequency of mispredicted branches. The processor on which the author is writing this thesis is receiving approximately 120 external interrupts per second, with 100 of those originating in the system timer and the other 20 being caused by the keyboard as the author enters this text. Since the processor clock runs at 90MHz this implies that an interrupt occurs approximately once every 750 000 cycles. From these figures it can be seen that on a system being used for text editing interrupts are rare. However, if a large amount of data is read from a hard disc with a controller requiring a large amount of processor intervention this figure increases to well over 2000 interrupts per second. Similar data read from a disc with an intelligent controller causes only a couple of hundred extra interrupts per second to the host.

The time taken to process arithmetic errors and undefined instruction errors can be large since they are expected to be caused solely by errors and thus do not occur in normal use.

Unimplemented instruction exceptions may occur frequently on a system making use of an instruction which is not present in the particular implementation but may appear infrequently on a similar system with a hardware extension. Thus a processor with an optional floating point coprocessor must implement unimplemented instruction exceptions efficiently so that it can work well in the case the coprocessor is not present;

however, the hardware needed to implement the exception is wasted when the coprocessor is present.

Memory access errors due to paging in a virtual memory environment are rare in a system with a large enough amount of real memory for the task it is performing; however some operating systems may use memory protection systems for other tasks such as demand loading of binaries, uncompressing sections of data or programs and other system specific tasks; these all increase the frequency with which memory access errors occur.

3.3.2 Cost of exceptions

As well as varying in frequency of occurrence, the cost of processing each exception varies. While a mispredicted branch usually has no cost other than the cost of discarding wrongly executed instructions, a memory access error in a virtual memory system is often much more expensive.

The cost of processing the exception can be split into the time taken:

- for the processor to enter the exception handler.
- for the exception handler software to perform system management tasks necessary to handle the exception.
- to perform what ever action is necessary to recover from the cause of the exception.
- to return to executing the original program.

In a system with virtual memory an access to a page which is presently stored on disc rather than main memory will cause a *page fault* exception. To process this exception requires that a page of memory is written to disc and another loaded from it. This will take many milliseconds since head movement and disc rotation are relatively slow. Since processing a page fault is so slow an extra few microseconds to enter the exception handler and store state will make very little overall difference to a simple system. However some multitasking systems may be able to find useful work to do while waiting for the disc system and thus the time taken to perform the exception entry is important again.

3.3.2.1 Exception latency

The performance of systems using externally generated interrupts is sensitive to both the total time taken to process an exception and the interrupt *latency*. Latency in this situation means the time from the external device notifying the processor of an interrupt to the time the appropriate interrupt handler is called. In some situations the amount of latency is constrained by the external hardware, for example an unbuffered UART which causes an interrupt when a character arrives must be serviced before the next character has arrived otherwise it will lose one of the characters. Unbuffered devices such as these are very demanding on exception handling mechanisms since slow response is penalised by failure rather than by a reduction in performance.

3.4 Mechanisms for saving state

In a processor which executes one instruction at a time implementation of exceptions is relatively simple. Once the exception is detected the state, i.e. the contents of the register bank, program counter and other internal registers is stored and the instruction stream changed to the address of the handler. This form of exception handling is known as a *precise exception* since the stored state represents the state of the processor at a particular point in the logical execution of the program.

In processors which execute more than one instruction at a time the register bank and program counter represent only part of the processor state; in addition there is the state of the dependency enforcing mechanisms, the state of each functional unit and (in a pipelined processor) the state of the pipeline latches. In principle it is possible to design a system which stores all this state when an exception occurs, and then allows it all to be restored after an exception, allowing execution of the original program to continue. The stored state would however be *imprecise* since it does not correspond to a particular point in the sequential model of program execution; some instructions may be partially executed.

Most microprocessor architecture definitions require precise exception handling and so in parallel processing implementations special effort must be made to generate a state corresponding to a precise exception when an exception occurs.

3.4.1 In-order, lookahead and architectural state

When considering different exception handling mechanisms it is useful to keep in mind the classifications of state made by Johnson **[John91]**. His definitions are:

- "The *in-order state* is made up of the most recent assignments performed by the longest continuous sequence of completed instructions."
- "The *lookahead state* consists of all assignments, starting with the first uncompleted instruction, to the end of the sequence."
- "The *architectural state* consists of the most recently completed and pending assignments to each register, relative to the end of the known instruction sequence, regardless of which instructions have been issued or completed."

For example, if an instruction was outstanding (which possibly causes an exception) the contents of all registers as seen at the start of that instruction represent the in-order state. The lookahead state would consist of those assignments made by instructions after the potentially failing instruction and the architectural state would be a combination of the lookahead and in-order states giving the view of the processor state as seen by the next instruction to be decoded.

3.4.2 Saving state using checkpoints

One approach to the problem of saving state is to take a snapshot of the current in-order processor state, known as a *checkpoint*, every n instructions. These copies can be used in the event of an exception to return the state of the processor to the state at the time of the checkpoint. Figure 3.1 represents an instruction stream with checkpoints taken every four instructions (represented by the thick bars). Instruction 11 causes an exception, perhaps due to a memory fault. Before the exception handler is entered the processor state is restored to that stored in the second checkpoint and instructions 9 and 10 are re-executed to produce the state corresponding to that prior to the execution of instruction 11.

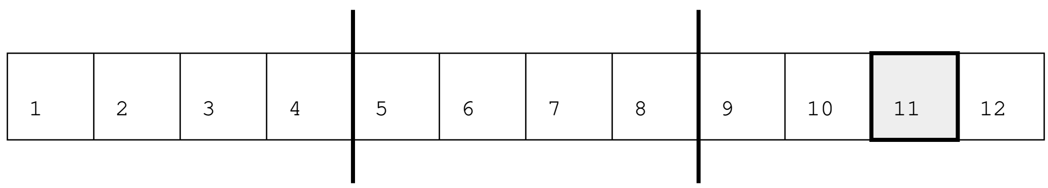


Figure 3.1 Checkpoints for exception recovery

To produce a precise exception state it is necessary to stall instruction issue and wait for all outstanding instructions to complete before taking the checkpoint; this leads to a performance loss. This mechanism is also expensive in terms of hardware to store the various copies of the state and in the complexity caused by the re-execution of the instructions between the checkpoint and exception. Furthermore the mechanism as described does not take account of state outside the processor.

Hwu and Patt describe an enhanced mechanism which reduces the need to stall the pipeline **[HwPa87]**. The mechanism that they present stores *Pending Consistent States (PCS)* which correspond to the architectural state at the time of the checkpoint without allowing outstanding instructions to complete. As instructions complete they update both the current state and all checkpoints taken after the instruction was issued. Before the state can be restored all outstanding instructions prior to the checkpoint must complete.

3.4.3 The history buffer

The *history buffer* is one of three mechanisms proposed by Smith and Pleszkun to handle precise interrupts in pipelined processors **[SmPl88]**. Like checkpointing mechanisms, the history buffer maintains some state to be restored when an exception is encountered. However, unlike checkpointing only the part of the state which has changed recently is stored. Figure 3.2 shows the basic structure of a system with a history buffer while figure 3.3 shows an entry in the history buffer.

As each instruction is issued:

- The old value of the instruction's destination register is placed in the *old value* field of the next free slot in the history buffer.
- The destination register number and program counter are stored in the slot and the exception and valid flags in the slot are cleared.

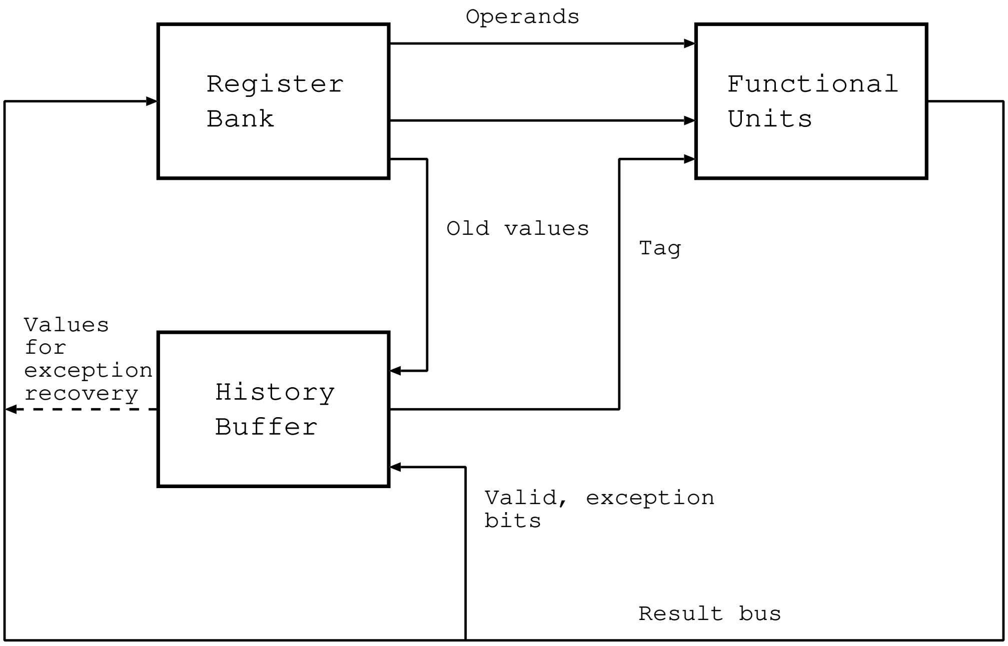


Figure 3.2 The history buffer

Destination	Old Value	Program Counter	Valid	Exception

Figure 3.3 Format of a history buffer entry

- The position of this slot is passed as a tag with the instruction down the rest of the pipeline.

As results return from the functional units to the register bank the exception flag is set if the instruction caused an exception and the valid flag is set to indicate that the instruction has completed.

In normal operation old results are discarded from the buffer in a FIFO manner once the valid flag is set. If an instruction causes an exception, issue is stopped and all outstanding instructions are allowed to complete, then the old values held in the history buffer are written back to the register bank with the newest value being written back first. The program counter of the instruction which caused the exception is restored from the entry in the history buffer. This restoration removes the effect of any instruction issued after the instruction causing the exception.

In addition to the storage and control logic for the history buffer itself, the system requires an extra read port on the register bank to supply the old values of the destination register.

If the history buffer does not contain enough entries, the decode and issue stages will be stalled waiting for a free space in the buffer before issuing the instruction, so the history buffer can cause a performance degradation beyond that caused by the added complexity of the control logic.

The history buffer does not help in the resolution of dependencies and it imposes a dependency itself: it may be necessary to wait for the old value of the destination register at issue before it is written into the history buffer. Thus an additional dependency resolution mechanism is needed.

The history buffer has two advantages over checkpointing systems. The first is that only a small part of the state needs to be stored, reducing the amount of storage needed. The second is that the cost of periodically copying the whole state is removed.

In a system using a history buffer the register bank holds the architectural state, as can be seen by the fact that the functional units access values directly from the register bank. The history buffer is used to restore the in-order state when an exception occurs.

3.4.4 The reorder buffer

The *reorder buffer* is another mechanism suggested by Smith and Pleszkun. The structure of a system with a reorder buffer is shown in figure 3.4. The reorder buffer is a queue holding values returned from the functional units; an entry is shown in figure 3.5.

The entry is similar to the history buffer except that the old value field has been replaced by a 'result' field.

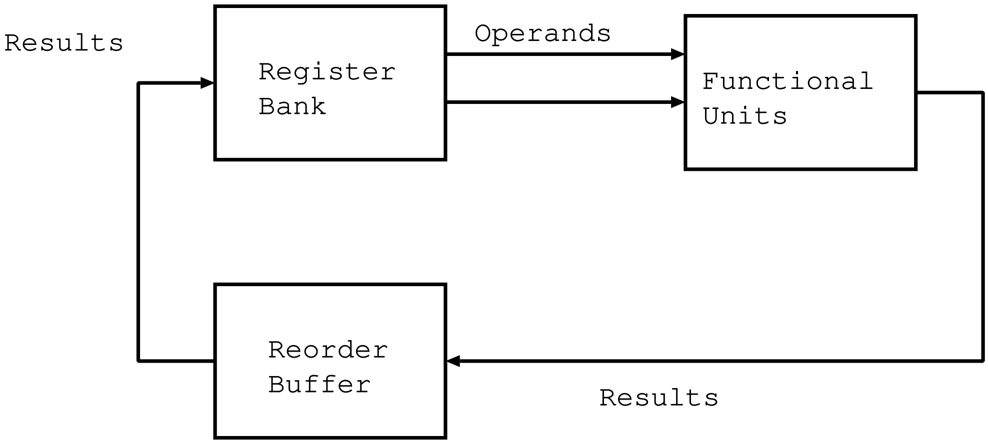


Figure 3.4 Processor organisation with a reorder buffer

Destination	Result	Program Counter	Valid	Exception

Figure 3.5 Format of the reorder buffer entries

During instruction issue a space is reserved in the reorder buffer into which the current program counter is written together with the destination register identifier. The index for the allocated entry is then associated with the instruction. As with the history buffer, if the reorder buffer is too small the issue stage will stall waiting for a space, leading to performance loss. Results returning from the functional units write their results into the allocated spaces in the reorder buffer rather than writing the results directly into the register bank.

Results are written to the register bank in the order in which they were allocated once the instruction is known to have completed without error. When a result is encountered which causes an exception, issue stops and all remaining results (corresponding to instructions issued after the instruction with the exception) are discarded and the program counter is restored from the reorder buffer entry of the instruction with the exception. Once all outstanding instructions have completed (and their results discarded) instruction issue can continue at the exception handler.

The reorder buffer holds lookahead state while the register file holds in-order state. As operands are read from the register file the performance degradation due to RAW dependencies is increased since it is necessary to wait for the in-order state to be resolved as results drain into the register bank.

3.4.5 Reorder buffer with forwarding paths

The primary disadvantage of the reorder buffer described above is its effect on RAW dependencies. This effect can be reduced (or removed) by adding forwarding paths from the reorder buffer around the register bank as shown in figure 3.6 **[SmPl88]**.

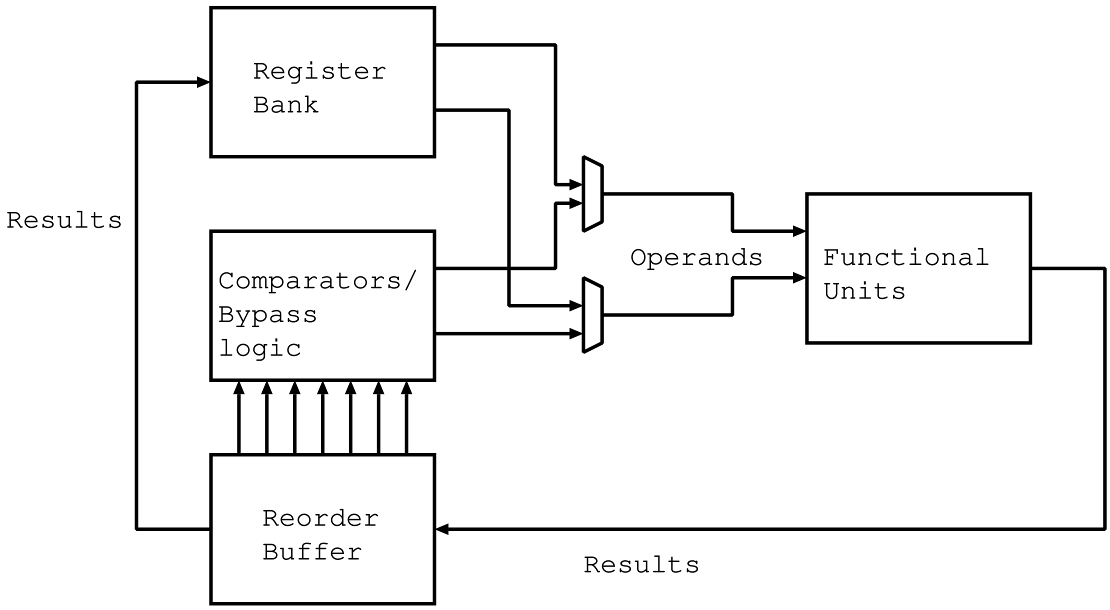


Figure 3.6 Processor organisation with a reorder buffer with forwarding

If it is possible to forward from every entry in the buffer a new mechanism for enforcing RAW dependencies is formed; this is similar to the model presented by Sohi and Vajapeyam **[SoVa87]**.

When an instruction is being issued the reorder buffer is searched for entries whose destination register field corresponds to a source operand that the instruction needs. If no entry in the reorder buffer matches the register number then the result has already reached the register file and the operand is read from there. If one match is found the instruction issue may be stalled until the reorder buffer entry contains a valid result, whereupon it is used as the operand for the instruction being issued. If more than one reorder buffer entry matches the latest version is used.

The primary disadvantage of this mechanism is the complexity of the logic required to test for the presence of registers in the reorder buffer. This consists of a Content Addressable Memory (CAM) whose size increases with the number of entries in the buffer and the number of operands that are forwarded. The comparison is complicated by the need to select the latest version of a register if multiple matches are found.

The main advantage of the reorder buffer with forwarding is that in addition to providing a mechanism for exception handling it also resolves RAW and WAW dependencies. Its handling of WAW dependencies can be seen to be a form of register renaming. Effectively each entry in the buffer is another register, and multiple versions of each register may be present in the buffer at any time. The buffer reorders values so that where there are WAW dependencies the values are written back to the register bank in the correct order, and the search mechanism ensures that instructions which are issued read the most recently allocated version of registers rather than the most recently completed version.

To summarise, the register file still holds in-order state, the reorder buffer holds lookahead state, but now the functional units read from the architectural state formed by a combination of the in-order and lookahead state. This combination is done with the aid of the CAM and other control logic.

3.4.6 The future file

The *future file* is the final mechanism described by Smith and Pleszkun. The mechanism described here is a modified form described by Johnson **[John91]**.

The organisation of a system using the future file is shown in figure 3.7. It consists of a model similar to the simple reorder buffer with the addition of an extra register file known as the *future file*. As in the simple reorder buffer system the reorder buffer holds lookahead state and the register bank holds in-order state. In normal operation the future file holds the architectural state, however upon recovery from exception the architectural state is formed by a combination of the future file and the register bank.

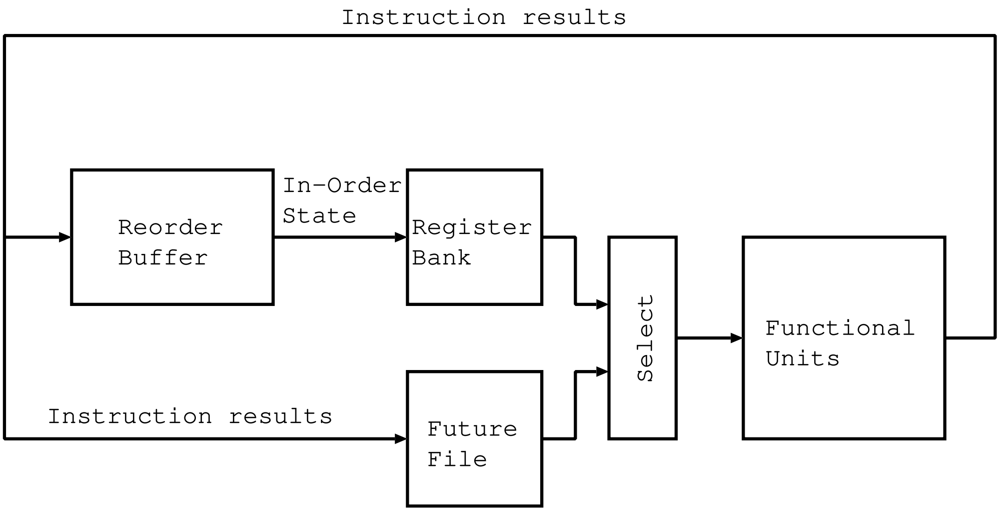


Figure 3.7 Processor organisation with a future file

Each entry in the future file consists of a validity flag and either a register value or a tag indicating that an outstanding instruction is due to place its result in the future file. As each instruction is issued the location in the future file corresponding to its destination register is marked to indicate that it is valid, and a tag is stored corresponding to the instruction which is to write the result.

As results arrive from the functional units they enter the reorder buffer and the future file. If the instruction returning the result does not match the future file tag the result is discarded; this allows WAW dependencies to be resolved by discarding older versions of the register irrespective of the order that they return from the functional units. If the tag matches the result overwrites it.

When an instruction is being issued and needs to read its operands it reads the same location in the register file and the future file. If the future file location is marked as invalid the operand is read from the register bank, otherwise the tag or value is read from the future file. If the future file contains a tag the instruction issue must stall to wait for the result to arrive. This lookup and flag check replaces the tag comparator lookup used in the reorder buffer.

During normal operation the validity flags in the future file are set and results are read directly from the future file without having to wait for the in-order state to be determined. When an exception occurs the valid results in the reorder buffer before the exception drain into the register file, completing the in-order state. The valid flags in the future file are then cleared. Any instruction which is now executed will read from the in-order state in the register bank.

The future file removes the need for the associative lookup of the reorder buffer while still not incurring cost in saving state. In a system with a future file recovery from an exception is simple since it is only necessary to drain the reorder buffer and clear a set of flags.

The cost in terms of hardware is a duplicate register bank (the future file itself), and the validity and tag logic. It is the area cost of the duplicate register bank which is the main disadvantage of the future file.

The future file cannot, by itself, solve WAR dependencies and so is unsuitable for out-of-order execution without the addition of extra hardware.

3.5 Exceptions in the AMULET1 processor

In AMULET1 exceptions fall into two broad categories: those that can be detected before instruction issue and those which can only be detected afterwards. External interrupts, software interrupts, and memory exceptions while fetching instructions are all in the first class, while failure to complete data memory operations (data aborts) are in the second.

Exceptions which can be detected during decode cause the next instruction to be substituted by an exception entry; in effect a special instruction. Consequently this form of exception needs very little extra hardware. Instructions which have already been prefetched are discarded as the substitute instruction behaves in a similar fashion to a subroutine call.

The processing of data aborts is more complex because other instructions may have been issued after the memory access instruction. A memory operation proceeds down the pipeline in the same manner as any other operation until it reaches the bottom of the ALU, which is used to calculate the address on which the memory operation is performed. The address is passed to the *address interface* which performs all address interactions with the memory system. However, the ALU is not yet freed for further instructions. The address interface then waits for an exception response from the memory subsystem. If the response indicates that there was no error, the actual data access is allowed to continue and the ALU is freed to allow further instructions to execute in the shadow of the memory operation. If the response was that a data abort should occur (for example on a page fault) the ALU changes the instruction colour (section 2.2.1) and informs the primary decode of this. Since the colour has changed no instructions after the aborting instruction complete.

With this mechanism the memory operation has been split into two suboperations. The first is a test of the ability of the memory operation to perform the access and the second is the actual memory access itself. If the exception test is expensive the performance of the processor would be significantly impaired since other operations would not be able to execute in the shadow of memory operations. In complex memory controllers (such as the one used in the ARM600 **[ARM91]**) determining whether a memory access is possible may itself require other memory accesses which could take a significant amount of time. In addition the separation of the two stages of the memory operation is not always possible; this is true if a system must be able to detect errors such as parity errors which are only detectable after the entire access has completed.

3.6 Exceptions in the Fred processor

The *Fred* asynchronous processor provides a *functionally precise exception model* which splits the problem of exception handling between the hardware and the operating-system [**Rich96**].

The basis of Fred's exception handling mechanism is the *instruction window* which forms part of Fred's dispatch unit and is based on the instruction window by Torng and Day **[ToDa93]**. The instruction window holds status information on all current instructions which is used during the dispatch of instructions and during exception processing.

The instruction window operates as a circular queue with instructions being issued in order and being removed when they reach the top of the queue having completed successfully. The status of an instruction moves through *not issued*, *issued*, and then arrives at either *complete* or a status reflecting an exception.

When an instruction with an exception reaches the head of the instruction window all instruction issue is halted and the processor waits for all outstanding instructions to complete. At this point the instruction window holds a copy of all instructions which have caused an exception and all instructions which have been fetched but not yet executed.

A copy of the instruction window is then stored in the *shadow instruction window* and the instruction window is cleared. The exception handling code is then entered. It is now up to the exception software to analyse the contents of the shadow instruction window, save all other state (including the R1 queue and branch queue described in section 2.3.2.3) and handle the exception. Once it has finished it issues a return from exception instruction which causes the shadow instruction window to be copied back to the main instruction window and execution to continue.

Since Fred allows out-of-order completion it is possible that an instruction after the one causing the exception could finish execution and overwrite one of the registers used by the failing instruction. Richardson gives an example similar to the following **[Rich96]**:

```
LDR R2,[R3,R4]
ADD R4,R5,R6
```

If the load was to fail but the add was to complete, R4 would be corrupted so that it would not be possible to re-execute the load after the exception had been handled. This problem is solved by holding the values of the operands to each instruction in the instruction window. In this example the load would keep a copy of R3 and R4 at the time of issue while the add would keep a copy of R5 and R6. It is then a job for the exception handling software to restore the registers (by examining the shadow instruction window) before returning from the exception.

This system requires significant additional hardware to implement the instruction window and the shadow instruction window. Each entry in the window is larger than would be otherwise necessary due to the storage of the instruction operands in addition to other control information. The addition of the shadow instruction window then doubles the amount of hardware required. Various methods for reducing this overhead are discussed in **[RiBr95]**.

The designers of Fred had the advantage that they could design their own instruction set and exception handling semantics. This has enabled the use of functionally precise exception semantics rather than conventional precise exception semantics. This option is not available when reimplementing existing systems.

3.7 Exceptions and external state

As well as a consistent state in the registers a consistent state in the memory system must be provided at the entry to an exception handler. This in itself is not sufficient in all systems since the *act* of reading or writing to a memory location may have side-effects (for example accessing I/O devices) making it impossible to undo the effect of an access. An example of this might be a memory mapped UART which has a buffer where received characters are stored. Reading a location in the processor's memory space causes this buffer to be read and to be freed for the next character. If a load instruction in the shadow of an exception was executed and caused this location to be read a character would be lost. Some systems solve this problem by having special I/O instructions, others have the ability to mark areas of memory as being I/O areas and force special consideration of accesses in these areas.

Techniques such as the history buffer and the reorder buffer can be adapted to preserve or restore state in the memory system. Utilising a history buffer requires that the old contents of the memory location is read before each modification, and in many systems this would double the time taken to perform the access and so would be unacceptable. A reorder buffer can be used to implement a *write buffer*. In a system with a write buffer write operations to memory enter the buffer immediately upon execution of the write instruction but only move from the buffer to memory when all preceding instructions are known to have completed successfully.

3.7.1 Exceptions in a pipelined memory

Another problem which often affects exception handling in memory systems is pipelining within the memory. Modern memory systems are often highly suited to pipelining, for example a system with a cache might be split into CAM lookup and RAM access stages which allow two memory operations to be in progress at any one time. The problem comes when the operation in a later stage causes an exception, thus:

```
LDR R1,[bad address]
STR R2,[good address]
```

When the store is issued to memory the processor may not yet have received the response from memory stating whether the load has completed successfully, so the store has already been passed to memory even though it should not execute as it is in the shadow of the aborting load. To solve this problem the memory system itself must perform some level of exception handling and discard operations in the shadow of operations which have failed. This might be done with the aid of a flag which is set when an exception occurs and causes all future memory operations to be ignored until the exception is recognised by the processor.

If more than one stage in the memory pipeline holds state which is modified by the memory accesses then there is the question of where the exception flag is stored and where it is examined. For example, consider a memory pipeline with two stages; an exception can be generated in the second stage and there is state which can be modified in both stages. Now consider two consecutive memory accesses, the first causes an exception in the second stage and the second access causes a modification to the state in the first stage. The exception in the second stage will set the exception flag in the memory and thus cause no more modifications to the state, however the modification of state by the second instruction is being carried out in parallel with the first access and thus the exception flag will not have been set by the time the state modification is carried out.

The author believes this is a potentially interesting problem, however it lies outside the scope of the processor itself and is thus not explored further in this thesis.

3.7.2 Multiple forms of external state

Where there are many different types of external state, processed by separate (potentially concurrent) execution units the problem is compounded since an exception caused by one section of external state may affect operations already issued to one of the other sections. An example of this might be an external floating point coprocessor and the memory subsystem. A store in the shadow of a floating point operation which caused an exception must not execute, and similarly a floating point operation in the shadow of a memory operation which caused an exception must not execute.

3.8 Summary

There are many different forms of exception in a typical system. Each form has different characteristics, (such as the frequency with which it occurs), which are affected not just by the forms of exception but also by the application and the environment the processor is being used in.

In a pipelined (or otherwise parallel) processor design the most difficult problem to be solved in exception handling is saving a consistent copy of the processor's state which can be used to continue execution after the exception has been dealt with.

Various exception handling schemes have been proposed. Each scheme has its own advantages and disadvantages in terms of performance (both in degradation of normal execution speed and in the cost of exception handling), hardware requirements and the burden placed on the software.

The reorder buffer (with forwarding) and the future file have a major advantage over the other techniques presented in this chapter as they integrate a dependency enforcement mechanism with the exception handling mechanism. It is hoped that by using one mechanism to solve both problems a simpler design might be produced.

Chapter 4: Issues in implementing the ARM architecture

The previous two chapters have discussed the twin problems of dependencies and exceptions in a processor-independent manner. This chapter describes the ARM and the problems involved in implementing its instruction set.

Since exception and dependency handling are both related to the maintenance of processor state, the chapter starts by describing the state held in the ARM processor.

4.1 Processor modes

The ARM operates in one of seven processor modes:

- *User* - for normal user programs
- *SVC* - for general purpose OS work
- *System* - Similar to SVC.
- *FIQ* - Used by the high priority (FAST) interrupt handler
- *IRQ* - Used by the low priority interrupt handler

- *Abort* - Used by the handler for aborted memory operations

- *Undef* - Used by the handler for undefined instructions.

All modes other than User mode are privileged in that a program running in that mode is allowed to change to any other mode and is given privilege when accessing memory (thus in most systems is allowed to access I/O devices). The privileged modes are entered automatically during exception entry.

4.2 Registers

The ARM architecture defines thirty general purpose registers, the Current Processor Status Register (CPSR), five Saved Processor Status Registers (SPSRs) and a Program Counter (PC).

At any given instant 15 of the general purpose registers are visible and are referred to as R0-R14. R15 is used to access the program counter. The current set of visible registers is defined by the current operating mode as shown in figure 4.1. This mechanism provides a number of temporary registers for use in each mode; in particular FIQ mode is provided with seven private registers allowing interrupt routines to be written without needing to stack and restore registers. Register values are preserved between uses of each mode, allowing the operating system to initialise R13 of each mode as a private stack or data pointer. The *System mode* is a privileged mode with the same register set as user mode.

The registers other than R0-R7 are referred to as *banked registers* and particular versions of a register are referred to by appending the mode name to the register identifier - for example R13_svc refers to the R13 register as seen in SVC mode while R13 would refer to R13 in the current mode.

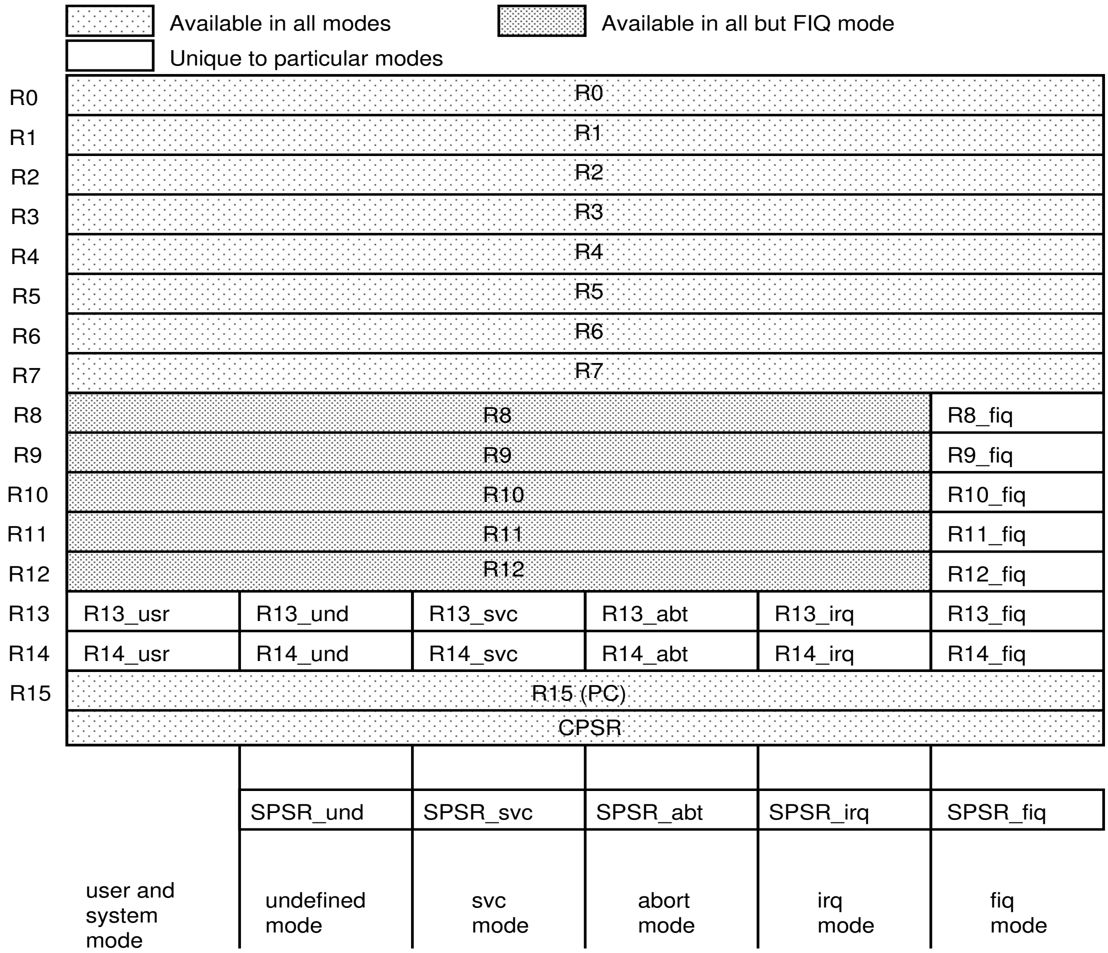


Figure 4.1 The ARM Register set

While R14 is a general purpose register it also serves as a return address register. The BL (Branch and Link) instruction, used to perform subroutine calls, stores the address of the instruction after the call in the current R14 register. Exception entry causes R14 of the exception mode to be set to the address at which execution must continue after the exception has been serviced (plus a predetermined offset determined by the exception).

4.3 The Program counter

The program counter (PC) is accessed in the same way as a general purpose register using register number R15. When read, R15 holds the address of the current instruction plus 8; this offset is an artifact of the pipeline implementation in the earlier ARMs.

R15 can also be used as the destination of almost any instruction. A write to R15 causes a branch to the address written. Combined with the BL mechanism described above this provides the complete subroutine entry and exit mechanism, thus:

```
        BL subroutine       R14:=addr+4, PC:=subroutine
        .
  subroutine
        do work
        MOV PC, R14         PC := R14
```

This mechanism does not allow nesting of subroutine calls, but since R14 is a general purpose register it can be stacked using the LDM and STM instructions described in section 4.9:

```
        BL subroutine
  subroutine
        STMFD R13!,{ R0-R3,R14 }      Stack R0,R1,R2,R3
                                      and R14 (use R13 as
                                      base pointer)
        BL another
        LDMFD R13!,{ R0-R3,PC }       Unstack R0,R1,R2,R3
                                      & the program counter
```

In this example the return address of the current subroutine is stacked while another is called. This example also shows that the program counter can be loaded directly from memory.

The flexibility allowed by writing directly to PC is useful while performing subroutine calls, complex multiway branches and vector indirections.

4.4 The CPSR

The CPSR, shown in figure 4.2, holds the current mode, a number of control flags and the ALU result flags. The ALU result flags, N - Negative, V - oVerflow, Z - Zero, C - Carry, can be written in any mode, while the other bits may only be written in the privileged modes. The I and F flags determine whether interrupts and fast interrupts are enabled. On ARMs with the Thumb extension the T bit enables the Thumb instruction set.

31	30	29	28	27 — 8	7	6	5	4 — 0
N	Z	C	V	Reserved	I	F	T	Mode

Figure 4.2 Organisation of the CPSR and SPSRs

The ALU result flags are written by any data operation or multiply with its S bit set. Thus:

`ADD R1,R2,R3` Does not change the flags,
`ADDS R1,R2,R3` Changes the flags.

The CPSR can also be modified explicitly by the use of the MSR instruction which enables a general purpose register to be transferred to the CPSR. This is infrequent and is generally only used by operating system code. The MSR instruction allows partial writes of the CPSR, thus:

`MSR CPSR_all, R0` Write all bits of the CPSR from R0
`MSR CPSR_flg, R0` Write only the ALU flags
`MSR CPSR_ctl, R0` Write only the control and mode bits

The contents of the CPSR are used by many instructions. The current mode affects all instruction and data fetches and all banked register operations. The ALU flags are used by instructions such as add with carry and some of the shift operations. In addition all ARM instructions can be executed conditionally on the basis of the ALU flags, so:

`ADDVC R0,R1,R2`

would only modify R0 if the overflow flag was clear before the instruction executed. As will be seen later this complicates many of the exception and dependency mechanisms.

The CPSR can also be read with the MRS instruction which provides the complementary operation to MSR.

4.5 The SPSRs and their role in exception entry

Each privileged mode has associated with it an SPSR which is identical in structure to the CPSR. The current SPSR can be read and modified in the same way as the CPSR using MSR and MRS instructions.

On an exception entry on the ARM the following steps are taken:

1. The CPSR is saved into the SPSR of the destination mode.
2. The CPSR mode bits are changed to the destination mode. In the case of some exceptions the I and F flags are modified to disable interrupts. The T flag is cleared.
3. R14_destination is set to the return address of the exception (plus an offset appropriate to the exception).
4. The program counter is set to the address of the appropriate exception handler entry point.

Thus to return from the exception, R14 must be copied to PC and the SPSR must be copied to the CPSR. Since modifying the CPSR would hide the relevant R14, one instruction must be used to perform both tasks. This instruction is a form of data operation which is special because its destination is the program counter and it has the S flag set (indicating it should update the ALU flags):

```
MOVS PC,R14        or
SUBS PC,R14,#4
```

The SUBS is used where an offset has previously been added to R14 during exception entry or where the instruction which called the exception is to be reexecuted. As expected these instructions write their result to the PC but instead of updating the ALU flags they restore the SPSR to the CPSR.

4.6 External state

The sections above have described all state internal to the processor, however both memory and coprocessors are part of the system state which must be correctly maintained.

4.6.1 Memory

All instructions in the shadow of a branch or exception must avoid accessing memory even by performing a read. Given:

```
LDR R0,[bad address]      Causes an abort
LDR R1,[good address]
STR R2,[good address]
```

if the first instruction causes an exception the access by the following load and store must not be seen externally to the processor. As explained in section 3.7, although a load may not have any obvious effect on normal memory, a read from a location in memory mapped I/O space could affect its state. Prefetch of instructions in the shadow of an exception is however allowed.

4.6.2 Coprocessors

The ARM has provision for up to 16 coprocessors which can be attached to perform extensions to the instruction set, for example floating point or DSP type operations. As in the AMULET1 and AMULET2, the issue of coprocessors has been ignored in the design presented in this thesis, mainly because of the added complexity to the dependency and exception mechanisms.

Ideally instructions destined for a coprocessor should begin execution as soon as possible, i.e. while preceding ARM instructions carry on executing. However if a preceding instruction causes an exception the coprocessor instruction must be prevented from changing the system state. Thus some mechanism for informing the coprocessors of the status of instructions sent to them must be provided. In practice with the addition of exceptions caused by the coprocessor this complicates the architecture significantly. Mechanisms for handling this problem are discussed in section 8.4.1

4.7 Exceptions on the ARM

The classes of exception are:

- IRQ - Externally generated interrupt.
- FIQ - Externally generated high priority ‘fast’ interrupt.
- Prefetch abort - Failure/refusal by the memory system to fetch an instruction.
- Data abort - Failure/refusal by the memory system to perform a data memory transfer.
- SWI - Software interrupt instruction encountered.
- Undef - Undefined or unsupported coprocessor instruction encountered.

The IRQ, FIQ, Prefetch abort and SWI exceptions can all be handled at instruction issue time by treating them as special instructions which perform an exception. These instructions are similar to a subroutine entry (BL) instruction but also cause the processor mode to be changed and the CPSR to be stored.

Undefined instructions fall into three classes. The first is a class of instructions set aside for future expansion known as the *undefined instructions*, these are offered to coprocessors for execution. The second class are instructions specifically intended for coprocessor execution. At execution time these first two classes of instructions are offered to coprocessors which may accept or reject them. If no coprocessor accepts the instruction the undefined exception vector and mode is entered.

The third class of undefined instructions is instruction encodings which do not fall into any other class of instruction. The behaviour of these instructions is *not* defined and they may perform any operation other than causing the processor to crash irrecoverably or performing an operation which cannot be performed in the current mode; e.g. executing such an instruction in user mode may not disable interrupts. While the behaviour of these instructions has not been defined it is regarded as desirable that they enter the undefined instruction vector. This can also be handled prior to instruction issue.

Thus only two forms of exception can occur after instruction issue: data aborts and undefined coprocessor instructions. As mentioned previously coprocessor instructions have not been implemented and always cause the undefined instruction exception to be entered: this is decided before issue. This leaves only data aborts.

4.8 Conditional execution

Perhaps the most unusual feature of the ARM instruction set is the fact that any instruction can be made conditional. It was introduced in the design of the original ARM to reduce the number of branches and so reduce the number of stalls due to procedural dependencies.

Conditional execution complicates the implementation of many exception and dependency handling techniques. There are several potential methods of solving the conditional execution problem, most of which severely degrade performance for code which uses conditional instructions. For example, one method would be to stall all conditionally executed instructions at decode and only allow them to continue once the condition code could be determined (i.e. any outstanding CPSR writes had been concluded). Although this would work, and code could be reoptimised to avoid using conditional execution, it goes against the aims of the AMULET project. The aim of the project is to produce a processor compatible with the ARM to the extent that existing software, tools and knowledge can be applied to the resulting processor. If significant changes were made to the relative performance of different instruction set features it would be necessary to recompile or recode applications to maintain efficiency. This would be contrary to the aims of reusing existing software. While it is clear that the relative performance of some features is going to change in a new implementation, it seems wrong to turn a feature which was a strong performance optimization into a strong performance penalty.

4.8.1 Conditional execution and the use of future files.

Of the exception handling mechanisms described in chapter 3, the *future file* described in section 3.4.6 is one of the most attractive because it solves the exception and dependency handling problems while avoiding the extra register read port required by the history buffer and the associative lookup of the reorder buffer. This section considers the difficulties associated with using a future file in an implementation of the ARM architecture.

Given the following fragment of C:

```
if (a==6) {
      b = c;
} else {
      b = d + e;
}
g = b + 5;
```

and the following mapping of variables to storage:

Variable	Stored in
a	R0
b	R1
c	memory
d	R2
e	R3
g	R4

Table 4.1: Mapping of variables to storage

the following piece of ARM assembler could be produced by a compiler:

```
1) CMP R0,#6               if (a==6) {
2) LDREQ R1,[..]               b = c;
                           } else {
3) ADDNE R1,R2,R3              b = d + e;
                           }
4) ADD R4,R1,#5            g = b + 5;
```

This example makes good use of the ARM's conditional instruction set, thus avoiding the need for several branches. At the time of issue, the CPSR flags needed for the evaluation of an instruction's condition codes may not be available since they may have been set by the preceding instruction which has not yet completed execution. The approach taken by most ARM implementations is to issue all instructions, irrespective of their condition codes, and then to check the condition codes later in the pipeline. So both instruction 2 and instruction 3 would be issued. The condition code is calculated in parallel with the execution of the instruction and is thus available together with the result of the instruction. If both instructions 2 and 3 are allowed to execute in parallel it is difficult to know which will complete first. For the purpose of this example assume that instruction 2 (the load) returns its result after instruction 3 has been issued but before instruction 3 completes, and assume that R0 is 6 and thus instruction 3 returns an invalid result since it failed its condition code.

Instruction 2 will discard its result because instruction 3 has overwritten R1's result tag in the future file. Instruction 3 fails its condition code, generating an invalid result, and so does not write its result. This leaves instruction 4 in a state where it cannot issue as it is waiting for the result of instruction 3 which will never arrive.

One possible solution would be for returning instructions, which would have discarded their result due to newer result tags in the future file, to wait until the instruction corresponding to the tag in the future file completes, and for the other instructions only to discard their results if a later instruction produces a valid result.

The management of such a scheme would quickly become complex and it complicates the result write back mechanism which was previously simple. For this reason the future file has not been considered as a viable mechanism in this thesis for implementing the ARM ISA.

4.8.2 Conditional execution and the Hades forwarding mechanism

The forwarding mechanism proposed for the Hades processor suffers from similar limitations when faced with conditional execution **[ECFS95]** (described briefly in section 2.5.3). A conditional instruction could potentially generate a result and store it in one of Hades' forwarding registers. A subsequent instruction might intend to use this result but would first have to determine whether the result really was generated by the conditional instruction. In the case of the mechanism used in Hades this problem could be solved by giving up the ability to forward results generated by conditional instructions.

4.9 Load/store multiple instructions

Another unusual feature of the ARM is the *load multiple registers* and *store multiple registers* instructions (LDM and STM respectively). LDM allows any set of the currently visible general purpose registers (including PC) to be loaded from sequential memory locations in one instruction. Similarly STM allows any set of the currently visible registers (including PC) to be stored. The instructions also allow the base register to be pre- or post-incremented or decremented. These instructions are very flexible, having many different modes of operations selected by flags in the instruction word. This flexibility leads to a number of cases which significantly complicate the entire processor design.

LDM and STM are commonly used in two ways in ARM programs. The first (as was shown in section 4.3) is in subroutine entry and exit where they are used to store and restore registers and to return from subroutines. The second use is in routines which initialise or copy large areas of memory.

The problem presented by these instructions is that they must access a large number of registers in one instruction. While most other instructions access no more than three registers an STM can read 17 registers. It is clearly not possible to do this in one pipeline packet so it is necessary to generate multiple packets for each LDM and STM instruction at some point in the pipeline where each packet will cause one (or more) registers to be processed.

4.9.1 LDM with base register in the transfer list

Any of the memory transfers making up the LDM or STM can cause a memory abort and thus an exception entry. This leads to a number of difficult cases for the exception recovery mechanism. In particular, the following instruction includes the base register in the transfer list:

```
LDMIA R3,{R0,R1,R3,R4,R5}
```

This instruction loads R0,R1,R3,R4 and R5 from memory with R0 being loaded from the address in R3 and each subsequent register coming from the next sequential word in memory. Registers are transferred in numerical order. The difficulty is encountered if a data abort occurs while transferring R4 or R5. In this case a new value of R3 (which was the base register) has already been loaded from memory, but the exception handler must be able to recover the old R3 value in order to re-execute the instruction. The ARM ISA defines the architecture in such a way that all instructions can be re-executed after an exception handler has corrected the cause of an exception, so care must be taken in preserving the base register across the exception. In this case however, it allows registers which will be reloaded by the re-executed instruction to be left corrupted after the exception, and so the value of R0 and R1 are unimportant since they will be reloaded by the instruction as it is re-executed.

4.9.2 LDM with PC in the transfer list

As with most other classes of ARM instruction the program counter can be used in an LDM. For example the following instruction loads all currently visible registers (including the PC) from memory:

```
LDMIA R0,{R0-R14,PC}
```

The instruction set is defined in such a way that registers must be loaded in order so that the PC is loaded last. Although this ordering rarely makes a difference in practice, LDM instructions are sometimes used to fetch data out of FIFO buffers in I/O devices, so the order is important.

An LDM with the PC in the transfer list produces a procedural dependency which, because the PC is loaded last, can cause a large performance loss. This can be significant as these instructions are used to return from nested subroutine calls. In section 6.2.3 a technique is presented which overcomes this problem, given the assumption that an LDM ..,{..PC} would never be used to load data from I/O space.

4.9.3 Conditional LDM/STM instructions

Like all other instructions an LDM or STM can be conditionally executed. One issue that this leads to is whether the instruction is expanded into multiple cycles before or after the condition code has been tested. If the expansion is carried out before the test then an LDM which fails its condition code will cause multiple cycles to be wasted in the pipeline. Expansion is, however, easiest to perform during decode, before the condition test can be carried out.

4.9.4 User mode register access

A final feature of the LDM and STM instructions is that they can access user mode registers while in a privileged mode, for example:

```
LDMIA R13,{R0-R14}^
```

when executed in a privileged mode causes user mode registers R0-R14 to be loaded from the address held in the privileged R13. This feature, which is normally used only in operating system code, can significantly complicate the way in which registers are accessed in the pipeline, since registers from two different modes must be accessed in the same instruction.

4.10 Summary

The ARM processor presents a number of interesting implementation challenges:

- *Banked registers* - The ARM's banked register mechanism complicates result forwarding mechanisms since (for example) register R11 written in one mode may or may not be forwarded as a value for R11 in the current mode depending on the particular combination of modes.

- *Program counter visible as a register* - Since most instruction classes can write to the program counter most instruction classes can potentially constitute a branch. This complicates instruction decoding and makes efficient branch prediction more difficult.

- *CPSR update* - The ability to update the CPSR in data operations complicates the operation of the processor. Data operations are the most common form of instruction. However, particular forms of data operation can change the operating mode of the processor; a rare and expensive operation.

- *Split SPSRs* - The SPSR registers are unusual in that it is possible to write to part of a register in an instruction; for example to update just the ALU flags while leaving the mode bits unchanged. This makes it difficult to treat the SPSRs as general purpose registers since many register forwarding schemes can not handle sections of a single read coming from different outstanding results.

- *Exception handling* - The ARM defines a conventional exception mechanism with precise exceptions. This limits the type of exception mechanism that can be used.

- *Conditional execution* - Conditional execution significantly complicates implementation of the ARM instruction set. Many techniques are difficult to use with conditional execution without making conditional instructions inefficient. Conditional execution also causes a RAW dependency on the CPSR, which is potentially updated by every data operation. Since the CPSR is read so frequently and potentially updated frequently, the normal register dependency mechanisms may not be suitable for use on the CPSR.

- *Load/store multiple* - The LDM and STM instructions provide an efficient mechanism for large data transfers on the ARM; however they are difficult to implement due to the large number of registers that they access. They also have a large number of different operating modes which present extra complexity, such as accessing user mode registers in privileged modes.

Chapter 5: An Asynchronous Reorder Buffer

This chapter describes the asynchronous reorder buffer, designed by the author, which forms the core of the dependency and exception handling mechanism in the architecture described in this thesis. The description is evolutionary in style, starting at earlier designs, stating the problems that they present, and then working forward until the current stage of the design is given.

5.1 Dependency and exception handling in AMULET2

AMULET2 uses two lock FIFOs of the type described in section 2.2.2, to manage dependencies. One lock FIFO contains a list of the destination registers of the outstanding results from the ALU and the other holds a similar list for outstanding memory operations.

Before reading the operands for an instruction, the lock FIFOs are interrogated to determine whether there is a write pending on a source operand register; if there is a pending write there is an unresolved RAW dependency and the instruction must be stalled until that write completes.

During decode the destination registers for the current instruction are placed in the appropriate lock FIFOs to indicate the pending write that the instruction will cause.

WAW dependencies are managed by testing for pending writes on the destination registers before inserting new registers into the FIFO.

As results return from either the ALU or memory subsystem they use the bottom element of the appropriate lock FIFO to select their destination register. The use of two separate lock FIFOs allows instructions for the ALU and memory to execute concurrently and it allows either the ALU instruction or the memory instruction to complete first.

Exceptions caused by the memory subsystem are managed by the *exec control* block. Memory operations are split into two stages, *exception detection* and *data access*. During the exception detection stage the memory system is presented with the required operation and address and must produce a fault/no fault response indicating whether that operation will cause an exception. The *data access* stage actually performs the operation. When executing an instruction which includes a memory access, the exec control block first performs the exception detection stage. Until the memory has responded the data access stage for the current instruction and any operations in future instructions cannot proceed.

The main disadvantage of this mechanism is that instructions after a memory access instruction cannot complete execution until the exception detection stage has completed. In systems with complex memory management systems, systems where the entire memory access must be completed before an exception can be detected and in systems where the memory is significantly slower than the ALU, this can inflict a large performance penalty.

5.2 A new pipeline model

An alternative to stalling the pipeline until an exception response has been returned from the memory system is to allow subsequent instructions to complete but to hold their results back from the register bank until the exception response has been received and thus it is clear that the results are valid. This is the basis of a reorder buffer as described in section 3.4.4. It is the author's development of this idea into a practical solution in an asynchronous environment which forms the basis of this thesis. Figure 5.1 shows a simple architecture based on this idea.

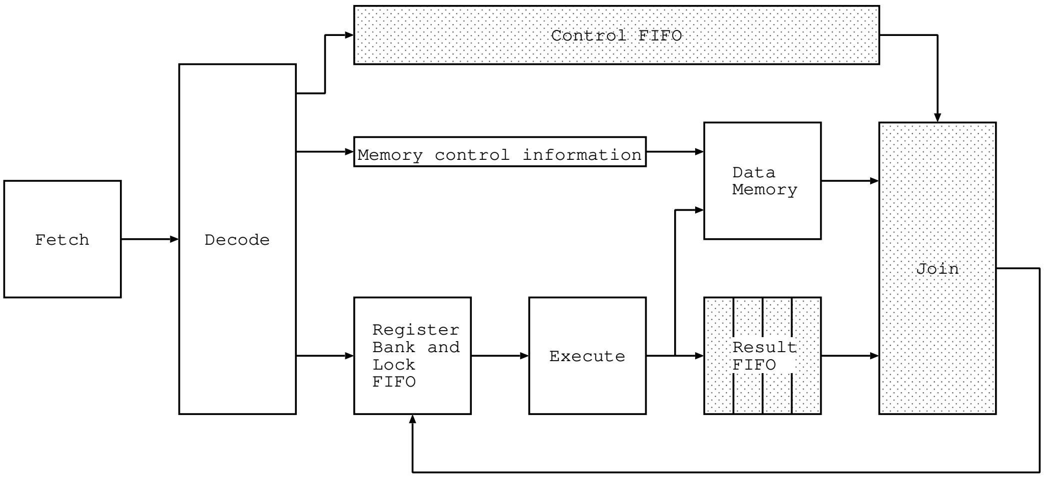


Figure 5.1 Initial pipeline model

Figure 5.1 shows a simple pipeline with the instruction and data caches hidden inside *fetch* and *data memory.* At this stage assume that each block represents one or more independent pipeline stages and is decoupled from previous stages allowing parallel execution; the position of the pipeline latches will be described later. When the decode block sends a packet down the pipeline for execution it places a packet in the control FIFO stating whether the result of that instruction will come from the memory or from the execute pipeline. The *join* block removes an item from the control FIFO; this indicates the source of the next result to be written to the register bank. The join block then waits for the appropriate result and writes it back to the register bank from where it can be used by later instructions. Store instructions must produce a result packet to indicate whether they caused an exception, but this result packet is discarded by the join block.

The job of detecting and handling exceptions is given to the *join* block which, after detecting an exception coming from the data memory, can discard all results until instructions from the exception handler are detected. A colour mechanism similar to that described in section 2.2.1 can be used to mark the new stream. The *result FIFO* holds multiple outstanding results from the execute pipe so that a number of instructions can execute in the shadow of a memory operation.

This mechanism is similar in operation to a simple reorder buffer without forwarding (section 3.4.4) with the exception that only results from the execute unit, and not the memory, are stored in the buffer, because only the memory can generate exceptions and thus there need never be more than one memory result which has arrived at the processor and has not been written into the register bank. The result FIFO holds lookahead state while the register bank holds in-order state.

As in the simple reorder buffer, this mechanism increases the performance loss due to RAW dependencies since results produced by instructions in the shadow of a memory operation cannot be used until the memory operation has completed. The impact of this depends both on the program being executed and the memory system available. The trace driven analysis of ARM programs presented in section 7.3.1 will show that around 25% of instructions are memory operations and around 50% are register to register data operations. Thus the likelihood of data operations and memory operations being interleaved is high. The same analysis shows that a high percentage of operands are generated by recently executed instructions, indicating that stalls due to

RAW dependencies would be common with this architecture. The effect of memory loads will be less than the effect of stores since the result of the load is often used shortly after and so imposes a RAW dependency, whereas since stores do not produce a result they cannot cause a dependency. The following example illustrates the problem:

```
STR R1,[R2,#64]
ADD R3,R4,R5
ADD R6,R3,#16
```

The data operations have no direct dependency on the preceding store, but they cannot be allowed to complete because the store may cause an exception. The first ADD may pass through the execution unit and place its result in the result FIFO. However, its result will not be available to the second ADD until after the store completes.

Even instructions which are not in the shadow of a memory operation are affected since the results have to pass through the result FIFO and join block before reaching the register bank. In particular the result FIFO, if implemented as a micropipeline, introduces a latency proportional to its length.

5.3 Parallel access FIFOs

As stated in section 2.3.2.1, once a block places a data item into a micropipeline the item's position at a later time is unknown to the block since the data moves independently down the pipeline. For this reason it is impossible to forward directly out of a result FIFO implemented as a micropipeline, since it is impossible to determine which stage to forward the value out of and also because the value may be moving between stages as the forwarding operation is performed. The author's first observation is that, by replacing the micropipeline implementation of the result FIFO by a circular queue implementation, the problems of latency are reduced and the forwarding problem becomes manageable.

A circular queue can be implemented as a set of latches together with a pair of counters; one counter selects which location the next input packet is stored in and the other selects which location provides the output **[Suth89]**; this is called a *Parallel Access FIFO*. It is necessary to design the queue in such a way that one counter cannot 'lap' the other when data is provided faster than it is removed or removed faster than it is provided. Figure 5.2(a) shows a micropipeline FIFO, and a circular queue is shown in figure 5.2(b); the two are presented together for comparison. The dashed line on the diagram of the circular queue represents the control necessary to stop the two counters lapping each other.

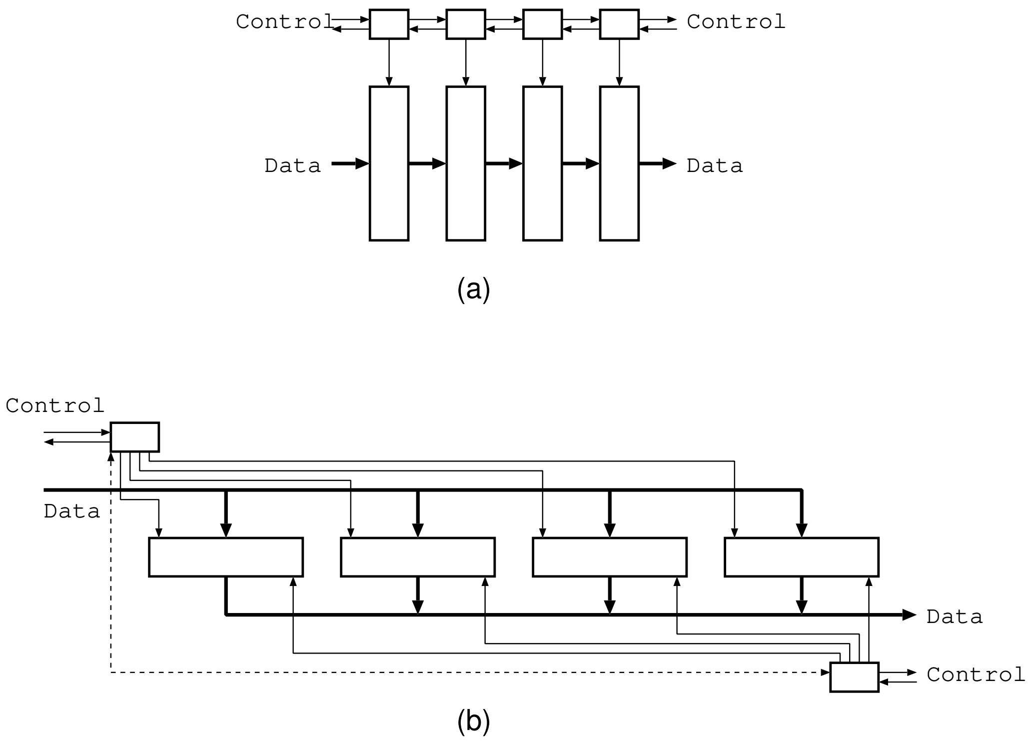


Figure 5.2 Micropipeline and parallel FIFO implementations

The external interfaces to a micropipeline and a parallel access FIFO may be identical, but there are a number of important differences in the behaviour of the two designs. In the micropipeline, data must pass through all elements of the FIFO and interact with all control elements before being available at the output, so the larger the FIFO the higher the latency, while in the parallel implementation the data passes through only one storage element. This gives the parallel implementation the potential advantage of a lower, constant, latency. The control logic on the parallel implementation is, however, more complex.

The main disadvantage of the parallel FIFO implementation is the large fan out of the circuit providing data to the FIFO and of each of the latches which must drive the common output bus (although this could be implemented as a multiplexer). As the FIFO grows in size this problem becomes worse. This disadvantage can (with the complexity of the control circuitry) lead to a higher cycle time for the parallel implementation. The trade off of latency against cycle time is discussed in more detail in **[Yant95]**.

The most useful feature of the parallel FIFO in relation to this work is that when data is placed in the FIFO it stays in the same position until it is overwritten by a new data value a *known number of input data packets later*. For the purposes of forwarding the parallel FIFO implementation is ideal since the input process can record the position at which it stored data and use this position to specify that the data be forwarded for use in later computation. Thus the FIFO evolves into the structure given in figure 5.3. In this figure the control for the data forwarding is not shown.

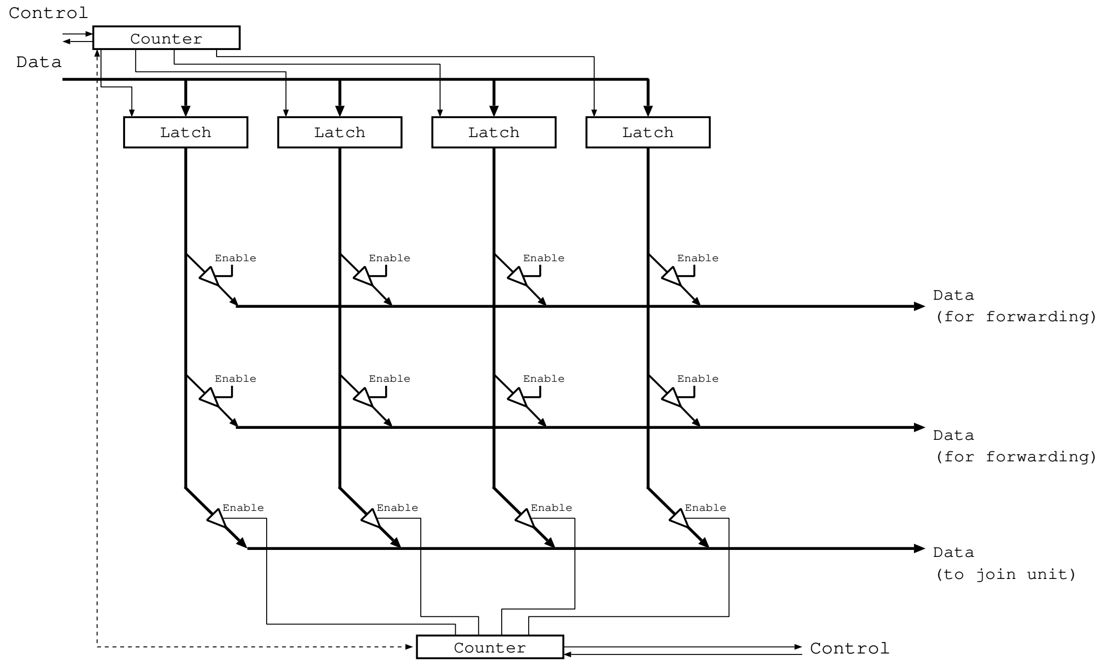


Figure 5.3 Parallel FIFO with forwarding

5.4 Three process view of the parallel FIFO buffer

The parallel FIFO buffer as described above is accessed by three types of process: the input process, the write back process and the forwarding process (there may be multiple forwarding processes).

- Input - The input process is connected to the output of the execute pipeline and thus data is inserted when a new result is generated.

- Write back - The write back process is connected to the join unit and produces a result when the join unit is ready to write a new value and when the value in the next available FIFO place has arrived from the input process.

- Forwarding - Finally, the forwarding processes produce data for reuse. The result of these processes is used in the same stage as the register read, just before the execute unit.

The design of the control for the buffer is heavily influenced by the need to reduce the synchronisation between the different processes within the microprocessor in order to take advantage of the asynchronous nature of the design.

In a conventional FIFO when an item is written out (by the write back process) that data is no longer available; thus it is necessary to synchronise the write back process and the forwarding process which must also read data out of the FIFO. However, the addition of the forwarding paths mean that this is no longer a simple FIFO, and the parallel implementation means that the data is still available once the data has been written back. Data can be forwarded before or *after* the data has been written out of the FIFO (or even while the write back is in progress), so the need for synchronisation between write back and forwarding is removed. This is important because the write back process and forwarding processes are at logically opposite ends of the pipeline and synchronisation would force most of the pipeline to act in lockstep.

Synchronisation is still required between the input process and the forwarding processes. This synchronisation is necessary to cause the forwarding process to wait for the arrival of data. The input process is at the base of the execute pipeline and the forwarding process at the head of the execute pipeline and so strict synchronisation may make it difficult to read the operands of one instruction while executing the previous one.

This synchronisation can be reduced by the realisation that the input process is performing two separable tasks. The first is the allocation of space in the buffer and the second is the arrival of data from the execute pipe. Thus the input process can be split into an ‘allocate’ process and an ‘arrival’ process. These two processes can be called from separate stages of the processor pipeline as long as some information is passed from allocate to arrival. Similarly the forwarding process is performing two separable tasks, ‘lookup’, where the buffer is searched to find the data to be forwarded, and ‘read’ where the data is read out.

5.5 Five process view of the buffer

There are now five processes involved in the management of the buffer:

- Allocate - The allocate process is called by the decode stage to allocate a space in the buffer for the result of the instruction being decoded. At this time it stores in a control field of the buffer the register identifier of the result. The Allocate process produces a token which is carried along with the instruction indicating where in the buffer the result must be written.

- Lookup - The lookup process is also called by decode and is used to search the buffer for results which need to be forwarded.

- Read - This process happens in parallel with the register read and reads the data from the buffer using information produced by Lookup.

- Arrival - As data arrives from the execute pipeline the Arrival process places it in the buffer.

- Write back - Write back writes data out of the buffer and into the register bank.

Essentially the Allocate and Lookup processes are operating on information concerned with register identifiers and the Read, Arrival and Write back processes are operating on result data which has arrived from the execute pipeline.

This reorganisation has moved the job of organising where to forward from into the lookup process rather than being external to the buffer. Further functionality can be encapsulated within the operation of the buffer. By adding an extra arrival process to write data arriving from the data memory system, and by merging the tasks of the Join block into the write back process, most of the work involved in exception handling has been hidden inside the operation of this parallel buffer. This produces an overall pipeline organisation as shown in figure 5.4.

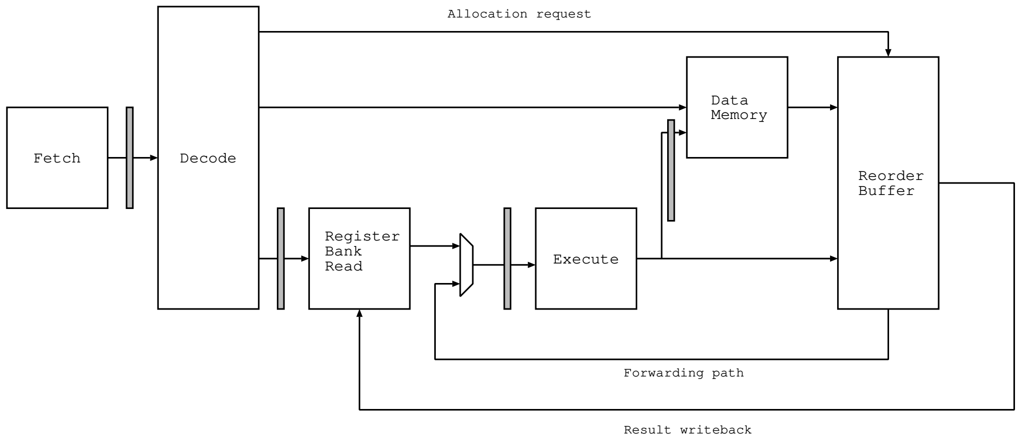


Figure 5.4 Pipeline with reorder buffer

Effectively this design process has evolved a mechanism very similar to the reorder buffer described in section 3.4.5. The five process view, together with the state mechanism, show that the reorder buffer is highly suitable for use in an asynchronous environment. In addition, it will be shown that it is highly suitable for use with the ARM's conditional instructions.

5.6 Operation with the five process model

This section describes the operation of the pipeline using the five process view of the reorder buffer described above. The description follows the progress of an instruction down the pipeline and describes the way in which it uses the five reorder buffer processes.

5.6.1 The pipeline stages

The grey bars in figure 5.4 represent pipeline latches; not all the pipeline latches in the design are shown, but sufficient detail is given to support the following description. In particular, it should be noted that the four main stages of the pipeline are fetch, decode, operand access and execute.

5.6.2 A data operation

This section will follow the ARM instruction AND R0,R1,R2 as it flows down the pipeline. This instruction performs a bitwise AND of the contents of the registers R1 and R2 and places the result in R0.

5.6.2.1 The decode stage

During the decode stage of the pipeline the operand and result register numbers are extracted from the instruction. Each operand register is passed to one instance of the lookup process. This interrogates the list of registers held in the reorder buffer and generates a *forwarding key* which lists the locations in the reorder buffer which potentially contain the required register. The list of registers in the reorder buffer is held in a CAM allowing fast implementation of the lookup process.

The forwarding key consists of a bit vector in which each bit is set if the corresponding location in the reorder buffer contains a value for the required register. The key also contains the address of the last location which was allocated. The forwarding key needs to be this complex because multiple versions of the same register may be contained within the reorder buffer. For this instruction two forwarding keys are generated (in parallel), one for R1 and the other for R2. The forwarding keys are passed to the next stage of the pipeline for use in the read process.

After the decode stage has received the forwarding keys from the lookup process it calls the allocate process to allocate a space for R0. A space can only be reallocated if its previous contents have been written back to the register bank, thus the allocate process may stall waiting for the write back process. Once a location becomes free in the reorder buffer the allocate process updates the register identifier for that location and marks the location as allocated. The allocate process then returns a *tag* which identifies the location just allocated; this tag is placed in a FIFO for later use.

Once the allocate process has finished modifying the state of the reorder buffer the decode process is free to start processing the next instruction. The allocate and lookup processes are called sequentially so that the lookup process is never examining data being modified by the allocate process.

5.6.2.2 The read stage

The read stage has been passed the forwarding keys generated by the lookup processes in the decode stage of the pipeline. In addition, it has been passed the register identifiers which are to be read. The register identifiers are used to start register bank reads while in parallel the forwarding keys are passed to the read processes in the reorder buffer. One read process exists for each operand; in this way all operands are read from the buffer in parallel with all register reads.

The read stage must use the forwarding key to determine whether a value for the register is held in the buffer and if there is more than one value it must determine which is the most recent. Consider the forwarding key below:

	0	1	2	3
(2)	0	1	0	1

Table 5.1: Example forwarding key for R1

This key represents part of the state of a four entry reorder buffer and would have been generated for the register R1 during the lookup process of the last instruction in the following sequence:

```
        ADD R1,R4,R5
        ADD R4,R4,#1
        ADD R1,R1,#8
        ADD R5,R5,R9
*       AND R0,R1,R2
```

At the time the key was generated by the lookup process the last entry to be allocated was entry 2 (shown in parentheses) and thus entry 3 is the oldest entry in the reorder buffer. Entries 1 and 3 both contain values for the appropriate register; signified by the 1 in those locations in the key. The read stage must search backwards from the last allocated entry to find the most recent version of the register, in this case the value held in location '1', generated by the ADD R1,R1,#8 instruction. Having identified location '1' as holding the latest version of the register, the read process must wait for the arrival of the value in that location to be signalled by one of the arrival processes.

The read process returns to the operand read stage a value and a flag indicating whether the value is valid; this flag is set if the register value was found in the reorder buffer. This flag is used to control a multiplexer which selects either the value from the register bank or the value from the reorder buffer.

The act of searching through the reorder buffer to find the latest valid version of the register is potentially complex and expensive to implement. However, high performance implementations of reorder buffers have been produced in the past **[WaDB94]**. An asynchronous implementation of this mechanism may be able to take advantage of the fact that some of the cases of operation are rare. Thus an implementation might be possible which was slower when there were multiple valid versions of a register present and very fast when only one version was present.

5.6.2.3 The execute stage

The values generated by the read process are used by the ALU to generate a result which is passed to the execute arrival process of the reorder buffer. The result will rendezvous with the tag generated by the allocate process. The tag is used to place the result in the correct location of the reorder buffer; that location is then marked as ‘full’ enabling it to be forwarded by the read process and written back by the write back process.

5.6.2.4 Write back

Unlike all the other processes within the reorder buffer, the write back process is autonomous and is not directly triggered by processes outside the reorder buffer. The write back process examines the locations of the buffer in order and waits for the locations to be allocated and for the results to arrive. The process then writes these results to the register bank and marks the entry as written back, allowing it to be reallocated.

5.6.2.5 Conditional execution and invalid reorder buffer entries

A result generated by the execute stage may be marked as invalid upon its arrival at the reorder buffer. This invalidation is performed to discard invalid results in the shadow of branches or results generated by ARM conditional instructions which failed their condition check. The mechanism for marking these results as invalid is described in more detail in chapter 6.

To process invalid data each reorder buffer entry has associated with it a validity flag. The write back process does not perform register writes for entries marked as invalid. The most complex impact of invalidation is with the result forwarding process, where the read process must avoid forwarding an invalid result. Referring to the example in section 5.6.2.2, if the R1 value in location ‘1’ was invalid the result from location ‘3’ would be forwarded. If both values were invalid no value would be forwarded and the value in the register bank would be used. This modification to the read process provides the correct values even when ARM conditional instructions are used. However, the read process is now much more complex to implement. The complexity of the read process is acceptable since it occurs in parallel with the register bank read. As long as the search of the reorder buffer can be carried out in a comparable time to the register bank read no performance loss will be incurred. This is only possible with a small reorder buffer. Section 7.3.2 will investigate the size of the reorder buffer required.

5.6.3 Memory operations

This section will examine the operation of the following instruction sequence:

```
LDR R0,[R1,R2]!        Base write back indicated by the !
ADD R3,R4,R5
SUB R3,R3,#9
ADD R6,R0,R3
```

The LDR instruction shown will load R0 from the address given by the sum of R1 and R2 writing the sum back to R1. During decode the allocation process of the reorder buffer is called twice, once for the result from memory and once for the base write back. The tags for the two operations are put in separate tag FIFOs. At this point in the description it will be assumed that the load succeeds and that the base write back result is allocated after the reorder buffer entry for the result from memory.

The operands are read in the same way as for the previous example; this produces values for R1 and R2 which are passed to the execute stage. The ALU adds R1 and R2, producing the address of the memory operation. This address is passed to the data memory and is also passed to the ALU result arrival process of the reorder buffer as the base register write back value. After the address has been passed to memory the execute stage is free to execute following instructions. For the purpose of this example it is assumed that the reorder buffer is large enough so that the first add instruction executes successfully and completes before the load result has arrived from memory. The forwarding process enables the SUB instruction to be executed by forwarding the R3 value generated by the first ADD from the reorder buffer; this R3 value will not have been written back to the register bank at this stage.

The second add instruction stalls during the read process for its R0 value. Eventually the memory returns a result which is placed in the reorder buffer by the memory result arrival process at the location specified by the tag allocated for it earlier. Once this has happened the second ADD instruction can proceed. Once the load result has been written back the new base value and the results of subsequent instructions can be written back to the register bank.

5.6.3.1 An abort

Using the same example as in the previous section it will now be assumed that the LDR fails due to an exception in the memory - this is a ‘Data abort’. The first ADD and the SUB instruction execute successfully in the shadow of the LDR. However, the value returned from memory is marked as being the result of an exception. This value is marked as invalid in the reorder buffer in a similar way to instructions in the shadow of a branch; it is, however, also specially marked to indicate the exception. No change in execution takes place until the write back process examines the location allocated for R0 in the reorder buffer.

Having encountered an aborted result the write back process must:

- Modify R14_abort to hold the address of the instruction causing the exception.
- Signal an earlier stage of the pipeline to cause it to jump to the abort exception vector (and change mode).
- Ensure that no results after the aborting result are written back to the register bank.
- Ensure that no results after the aborting result are forwarded by future instructions.

Since the write back process has access to the register bank write port it can easily write the aborting instruction's address into R14 (the address is passed to it down the Xpipe introduced in section 6.3.1), although there are issues involving data validity of R14 discussed in section 5.6.4.

Ensuring that no result in the shadow of the aborted result is written back or forwarded could be easily accomplished in a synchronous design by invalidating all entries in the reorder buffer. Unfortunately in an asynchronous design it is impossible for the write back process to guarantee that the forwarding process is not examining the valid bits at the time it intends to clear them. If this was happening the forwarding process might enter an undefined state having read a value from the valid bits which was neither true nor false since it was being modified at the time it was examined. To provide this functionality in an asynchronous implementation two extra mechanisms are required which are described in the following subsections.

5.6.3.2 Inhibiting write back of results in the shadow of aborted operations

To avoid writing back results which have already been placed in the reorder buffer but which are invalidated by an exception, an extra colour flag, known as the *exception colour*, is associated with each instruction. This flag is maintained by the reorder buffer write back process and is toggled each time an exception is encountered. The write back process will only write a result which has the same colour as the current exception colour. The following example is used to explain this:

Instruction	Exception colour during fetch	Exception colour in write back	
`ADD R0,R1,R2`	0	0	
`LDR R0,[abort!]`	0	0	
`ADD R2,R3,R4`	0	1	
`STMFD R13!,{...}`	1	1	1st instruction of exception handler

Initially instructions are fetched with the same colour as the current exception colour. When their results arrive they are written back. When the aborting load is encountered by the write back process it toggles its exception colour. Further results are then ignored because their exception colour differs from the current exception colour. When the write back process forces the jump to the exception vector it simultaneously passes back the new exception colour to the fetch mechanism and thus the instructions fetched from the exception vector execute in the new exception colour and are thus written back.

5.6.3.3 Inhibiting forwarding of results in the shadow of aborted operations

In the example given in section 5.6.3.2, the first instruction of the abort handler could read R2 and it is important that the R2 value generated by the ADD in the shadow of the aborted load is not forwarded for use in the abort handler.

To avoid this the decode stage detects the changing exception colour and issues a number of dummy instructions down the pipeline which generate invalid results. Thus each of these instructions' results occupies a reorder buffer location but since it is marked as invalid it will not be forwarded. Decode issues enough dummy instructions to ensure that the reorder buffer is clear of valid values before the first instruction of the exception handler is executed.

5.6.4 Data validity in the read process

Since the write back process is not synchronized with the operand read stage (which reads the register bank and performs operand forwarding) there is the possibility that a register may be read from the register bank while it is being written to by the write back process of the reorder buffer, thus the value read is unpredictable. This situation is, avoided by the forwarding mechanism implemented by the read and lookup processes. Since the reorder buffer always contains the latest version of the register's value and since the forwarding mechanism is always used, any value not written back to the register bank or in the process of being written back will be forwarded from the reorder buffer. This value will always replace the value read from the register bank whenever there is any possibility that it is being written to.

Unfortunately there is an exception to this rule caused by the abort mechanism described above. When the write back process writes R14_abort during exception entry there is no entry for it allocated in the reorder buffer and thus any attempt to read it will not cause a forwarded value to be used. Thus it is possible for an instruction to read a value from the register bank while it is being written to; potentially this would cause metastability issues if the result was used in a control circuit[1].

1. The same problem affects the base register written by the base restore mechanism described in section 6.5.

The instructions which could be reading this invalid result lie in the shadow of an aborting memory operation and so their own results are marked as invalid and the invalid data does not get propagated far through the processor. Thus while the data is never used in a valid instruction, care must be taken in the implementation to ensure that this invalid data can not pass through control circuitry and cause metastability problems.

Although not elegant, this approach is a valid engineering solution to the problem. A more elegant solution would be to cause the dummy instructions generated by decode to clear the reorder buffer to perform the R14_abort write. This is significantly more complex to implement in decode but does solve the problem completely.

5.6.5 Lack of synchronisation between the read and allocate processes

The reader may have noticed that in the example in section 5.6.2.2 the result of the AND instruction is allocated to location '3' which is also a location containing a result which is potentially forwarded. This is a perfectly valid (if somewhat counterintuitive) situation since the allocation process modifies the register identifier storage which is not used by the read process. This approach does, however, present two problems which are described in the following two subsections.

5.6.5.1 Result arrival indication

One method for indicating result arrival would be to store a single 'arrived' bit with each piece of data. This bit would be cleared during allocation and set by the arrival process. The 'read' process would have to wait for the bit to be set. This is insufficient in this system since the read process may examine the 'arrived' bit for an operand while the 'allocate' process resets the bit for a future result.

An alternative is to hold two flags for each reorder buffer element. The first flag (the ‘allocation colour’) is toggled by the allocation process when the element is reallocated, the second flag (the ‘arrival colour’) is toggled by the arrival process which receives the result for the element. Initially both colours are false; upon allocation the allocation flag is toggled and upon arrival the arrival flag is toggled. Thus when the two flags are equal the current result matches the current allocation. A copy of the current ‘allocation colour’ is included in the lookup keys. The read process now waits for the arrival colour to match the allocation colour captured at the time of lookup. Thus the read now waits for the result for which the lookup was performed.

5.6.5.2 Overwriting of data

It is, of course, important that an instruction does not overwrite one of its operands. Normally this can not happen since all operands are needed before any result can be produced. Thus in the example of AND R0,R1,R2 the instruction will stall in the operand read stage until both R1 and R2 are available and thus there is no possibility that R0 will overwrite one of the operands. However, there is one instruction which can potentially produce a result before all operands have been read; this is STR Rn,[Rn,Rm]!. In this instruction the register Rn is read twice, once as a piece of store data and once as the base register in the address calculation. This instruction includes a base write back and thus Rn is modified. If the three forwarding and register read processes were implemented independently the read for the store data could be performed after the base had been modified. It is important that in this situation a deadlock is not created due to the read process waiting for the original Rn value to become available as the store data. The precise effects of this instruction are classed as *unpredictable* by the ARM ARM because of the way it reads a register which is being modified; however it is unacceptable for the instruction to deadlock the processor.

5.7 Process synchronisation

Seven constraints regulate the interaction of the processes:

1. The ‘allocate’ and ‘lookup’ processes share access to the register number CAM and thus access must be mutually exclusive. This is implemented by calling the allocate process after all lookups for an instruction are complete and not performing any more lookups until allocation has completed.
2. The ‘read’ process requires the key generated by the ‘lookup’ process. These two processes are used in separate pipeline stages and once the lookup process has generated its key it may continue with the next operation.
3. The ‘read’ process must wait for data arrival. This can be implemented using the ‘colour’ mechanism described in section 5.6.5.1.
4. The ‘arrival’ process must wait for an allocation key stating where to store the data. This is implemented via the tag FIFOs.
5. The ‘write back’ process must not write back a location multiple times. This could happen if the write back process was much faster than the allocate process and thus the write back process would write all entries in the reorder buffer and arrive back at the first location and write this again. This constraint can be implemented by the addition of a ‘write back colour’ bit to each location which is toggled as each location is written back. If the colour does not match the location’s allocation colour it has not been written back.
6. The ‘write back’ process must wait for data arrival. This can be achieved by waiting for the allocation colour to match the arrival colour.
7. The ‘allocate’ process must not reallocate a space which has yet to be written back. A location which has been written back is signalled by the allocation colour and write back colour matching and thus that location may be reallocated.

It should be noted that there are no conventional request/acknowledge handshakes taking place directly between any of the processes, so while one process may need to wait for another to complete a task it never has to acknowledge the end of the wait to the original process.

5.8 The reorder buffer entries

The fields in each reorder buffer entry are:

- Data - The result of the instruction which will be written back to the register bank by the write back process.

- Register identifier - The register which the result is written to and which is matched during the forwarding comparison.

- Valid flag - This flag is returned from the execute or memory unit with the result data and indicates whether the instruction passed its condition code. It is also used to invalidate instructions in the shadow of a mispredicted branch in a similar manner to that used in **[WaDB94]**; this is described in detail in section 6.2.2.

- Abort flag - This flag is set if the result is invalid because the instruction which generated it produced an exception. The Abort and Valid flags together distinguish valid results, invalid results due to condition code failures and invalid results due to exceptions.

- Exception colour - The instruction's exception colour is stored in the reorder buffer; this is used in the write back process to discard results in the shadow of an aborted memory operation.

- State - These indicate whether the entry is allocated, whether its result has arrived or whether it has been written back. The state consists of the allocated, arrived and written back colour bits described above.

- Control - Some information is needed to mark entries with special requirements; for example place holders for store operations must be marked since they never generate a real result. This control information is stored here.

5.9 Summary of constraints

The design of the asynchronous reorder buffer relies on a number of constraints, imposed by the inter-working of a number of separate blocks, to remove synchronisation and arbitration from the design. These are:

1. Data is left in the reorder buffer after being written back to the register bank. This removes the need for synchronisation between the write back process and the forwarding and lookup process.
2. The forwarding mechanism. The forwarding mechanism is not just a mechanism to speed up access by bypassing the delayed register write; in addition it ensures that the latest result is always available from the reorder buffer and thus enables the removal of arbitration between register read and register write back as described in section 5.6.4.
3. The allocation and arrival mechanisms ensure that no attempt will be made to read a location which is awaiting the arrival of a result, thus no arbitration is needed to avoid reading a location while it is being written.
4. The allocation mechanism ensures that no more than one instruction is due to write to each reorder buffer entry; this removes the need for arbitrating between multiple write attempts at each reorder buffer location.

5. The separation of ‘allocation’ and ‘arrival’ has removed the need for synchronisation of the decode and execute stages.

These constraints save large amounts of hardware required for synchronisation and arbitration but make the entire architecture much more difficult to reason about. With these constraints in place it is no longer possible to analyse each block on its own, define and check interface specifications and argue about the correctness of larger blocks of the architecture; instead it is the combined operation of all blocks which make the operation of each block correct.

The aim of many high level asynchronous hardware description languages is to allow the description of individual blocks and to enable automatic synthesis by the addition of arbitration on shared variables and synchronisation between processes. However these languages often do not have the mechanisms to describe the higher level constraints which make this arbitration and synchronisation unnecessary.

5.10 Summary

This chapter has described the derivation of an asynchronous reorder buffer, its integration into a pipeline and the constraints necessary to make it work. It is the realisation that the reorder buffer can be implemented as a parallel access FIFO, that forwarding can continue after a result has been written back, and the constraints listed above, that enable the reorder buffer to function in an asynchronous environment.

The reorder buffer simultaneously solves the problems of exception and dependency handling while providing result forwarding and mechanisms for dealing with the ARM’s conditional instructions. It will be shown in section 7.3.2 that the reorder buffer provides a significant performance increase over the lock FIFO mechanism used in AMULET2.

Chapter 6: Auxiliary mechanisms

The reorder buffer is a powerful mechanism for resolving dependencies and handling the effects of exceptions on the main registers. However, it does not solve all the processor's dependency and exception handling problems. This chapter describes other mechanisms which provide an environment in which the reorder buffer can operate and which provide dependency resolution and exception handling for other parts of the processor's state.

6.1 The instruction colour

As described in section 2.2.1 the *colour* matching mechanism solves the problem of invalidating instructions in the shadow of branch instructions. In this design the expected instruction colour is stored in the *commit block*. This is a block immediately after the execute stage whose primary role is to ensure that instructions fetched in error and instructions that have failed their condition code test do not change the processor's state. The colour is stored in the commit block as a collection of flags that are modified whenever an instruction which changes the instruction stream passes through. The fetch unit appends the current instruction colour to instructions as they are fetched and this propagates down the pipeline. If, at the commit block, the instruction's colour does not match the expected colour, the instruction's result is marked as invalid, so preventing any permanent state changes occurring. Figure 6.1 shows the position of the commit block in the pipeline and how it routes addresses to the data address interface. Since it is positioned before the data memory in the pipeline the commit block can stop data accesses by instructions which have failed their condition code or have mismatched colour values.

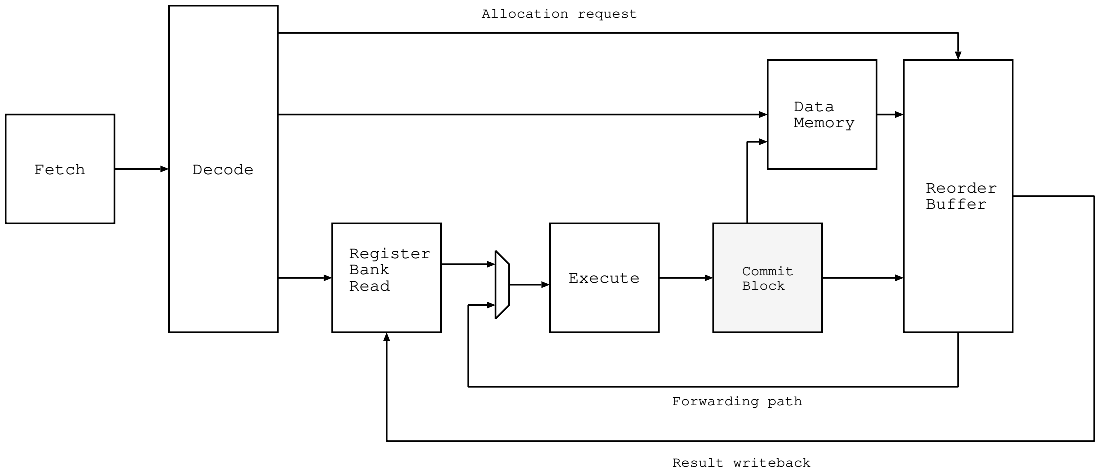


Figure 6.1 The commit block in the pipeline

6.2 The program counter

The ARM is unusual in its treatment of the program counter in that it can be used in much the same way as a conventional register. Many instructions can use the program counter as their source or destination register. For example, a subroutine return can be implemented as a normal move instruction with the destination set to the program counter. In this design the program counter is stored in the fetch unit.

6.2.1 Reading the program counter

In the early synchronous ARMs the program counter was read by the execute stage directly from a register in the fetch stage. Since the currently executing instruction had been fetched two cycles earlier the program counter was always two words ahead[1] of

1. On the ARM 2 and ARM 3 there were a number of instructions in which PC was three words ahead of the fetch address; this behaviour was removed in later ARMs.

the address from which the executing instruction had been fetched. In an asynchronous design it is not possible for one stage to read a value from a register in another stage without extra synchronisation which is not considered desirable.

Figure 6.2 illustrates an alternative solution to this problem. As each instruction is fetched a copy of the address used to fetch it is sent down a pipeline to the decode unit. As the decode unit receives the instruction from the fetch mechanism it also reads the associated program counter value which can then be used wherever the PC is required as a source operand. The address must first be modified by adding eight to simulate the pipelining of the early ARM designs.

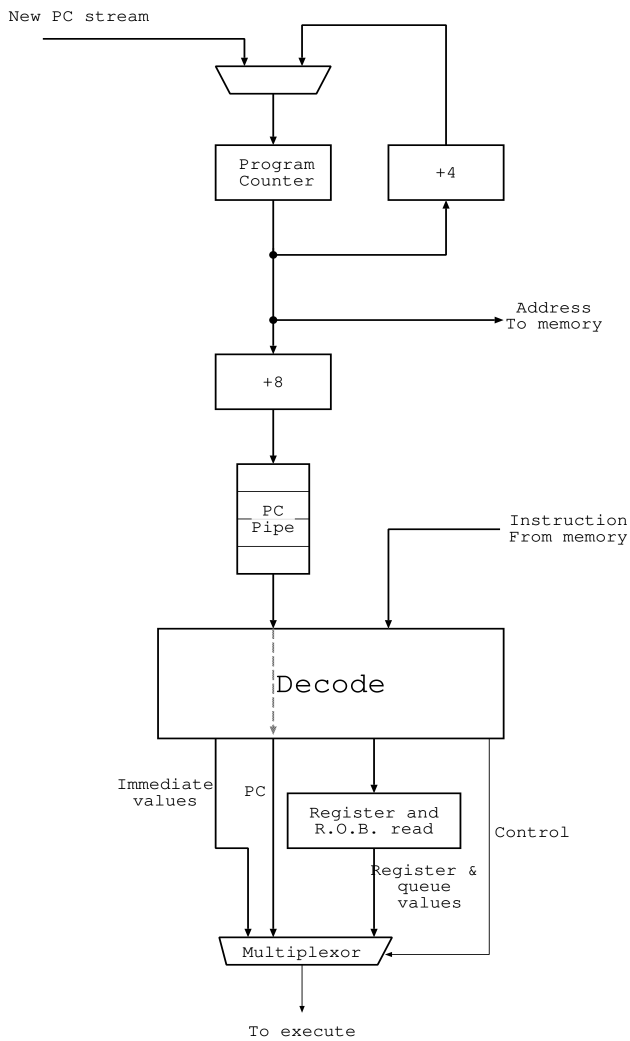


Figure 6.2 Adding the offset to the program counter

6.2.2 Changing the program counter

The first design considered treats modifications to the program counter in the same manner as modifications to any other register by passing them through the reorder buffer. Figure 6.3 shows how an extra path (shown dashed) is added to the design shown in figure 6.1 to route new PC values to the fetch unit rather than to the register bank.

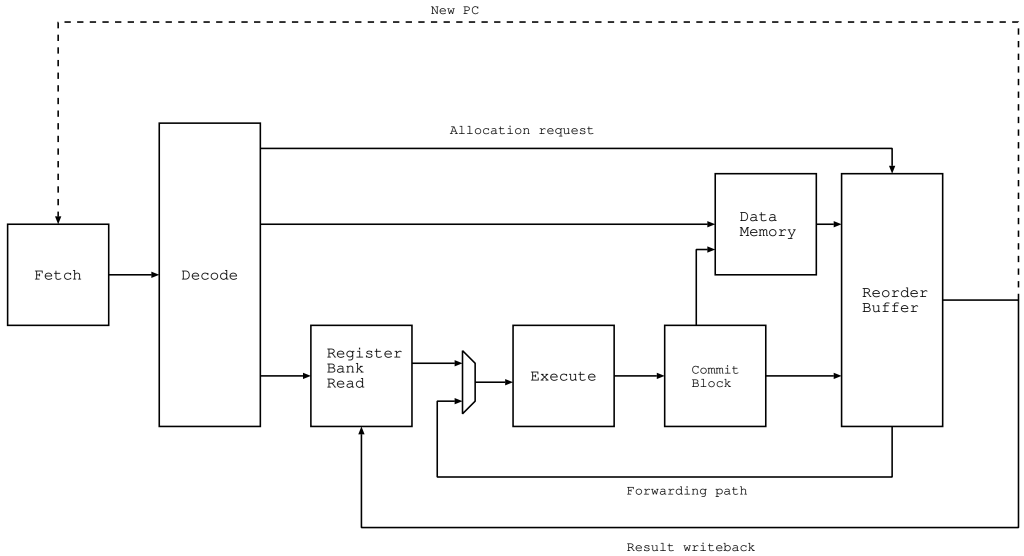


Figure 6.3 Writing to the PC via the reorder buffer

An advantage of this design is that the reorder buffer can cause instruction stream changes due to exceptions simply by passing the new address to the fetch unit. The disadvantage is that changes to the program counter have a high latency because they must travel the entire length of the pipeline before reaching the fetch unit. This would cause excessive prefetching which wastes power and reduces performance. The problem is especially severe when a branch follows a slow instruction such as a load.

Figure 6.4 shows an alternative arrangement in which the commit block routes PC modifications directly from the execute unit to the fetch unit. This eliminates the delay caused by passing the modifications through the reorder buffer. Commit is the most appropriate block to do this since it holds the expected instruction colour flag and so can

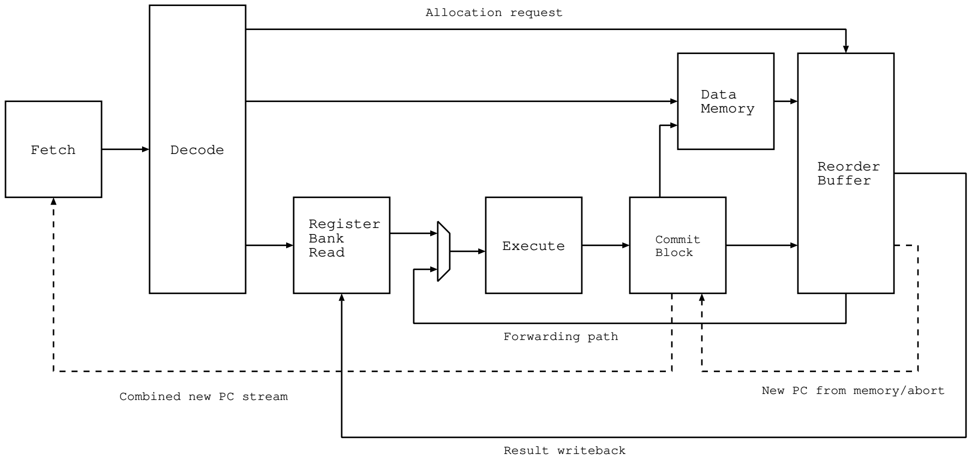


Figure 6.4 Writing to the PC via the commit block

determine the validity of the instruction producing the new PC value. The example below shows how this mechanism processes branches (and other instructions which generate a new PC value via the execute stage) with the aid of a single bit colour flag:

Instruction	Fetch Colour	Expected colour at Commit when the result arrives	
1 CMP R0,R1	0	0	
2 BEQ label1	0	0	Assume this branch is taken.
3 B label2	0	1	Colour toggle caused by previous branch
4 ADD R2,R3,R4	0	1	
5 label1: MOV R2,#6	1	1	First instruction at branch destination.

In this example we assume that the comparison causes the branch in instruction 2 to be taken while instructions 3 and 4 are prefetched in the shadow of the branch. When the first branch reaches commit its condition code is checked and found to be valid and so the branch will be executed. Commit toggles the *expected colour* and sends this new colour value together with the branch destination address to the fetch unit. Instructions 3 and 4 which were incorrectly prefetched are discarded by the commit block because their fetch colour, 0, is different from the current expected colour which is 1.

Discarding instruction 3 is simply a matter of ignoring it since it has no other effect on the system. Discarding instruction 4 involves writing its result into the reorder buffer marked as invalid; it can not simply be discarded because a place has been reserved for its result during the decode stage. When instruction 5, which is the first instruction of the new instruction stream, reaches commit it is known to be valid because its colour, 1, matches the expected colour.

A single colour bit can also manage PC values loaded from memory and returned to the commit block via the reorder buffer:

Instruction	Fetch Colour	Expected colour at Commit when the result arrives	
1 LDR PC,[....]	0	0	
2 MOV R0,R1	0	1	
3 ADD R3,R4,R5	1	1	Destination of LDR PC

In this example when instruction 1 reaches commit the current colour is changed and all further instructions in the current stream (in this case instruction 2) are discarded. At some point commit receives the new PC value from the reorder buffer which it then passes to the fetch unit with the current colour.

Exceptions complicate the colour mechanism significantly; the following example shows the problem.

Instruction	Fetch colour	Expected colour at commit when the result arrives	
1 LDR R1,bad address	0	0	
2 B dest	0	0	
3 ADD R0,R2,R3	0	1	Toggle due to branch
4 ADD R4,R5,R6	0	1	
5 SUB R7,R8,R9	0	0	Toggle due to exception reaching commit
6 B dest2	0	0	

In this example, at instruction 5 the reorder buffer has notified commit of the exception. Commit then toggles the colour and issues the exception vector address to the fetch unit. However, since instructions from the destination of the previous branch had not yet arrived, the effect of toggling the colour was to revalidate the remaining instructions in the shadow of the original branch. This would enable instruction 5 to write its result into the reorder buffer[1] and instruction 6 to cause yet another instruction stream change - this time incorrectly.

This problem has arisen because there are now three streams of instructions in the system:

1. The original stream including the load and branch.
2. The stream fetched at the destination of the branch.
3. The stream fetched at the abort vector caused by the load's exception.

The role of the colour is to select which of the streams currently in the system is the valid one and it is clear that a single bit is no longer sufficient for this. The solution is a two bit colour; one bit is toggled by commit when an instruction from the execute stage causes a PC change and the other bit is toggled when the stream is changed by a value from the reorder buffer. Instructions are only valid when both of the colour bits match the current expected colour.

---

1. However this incorrect value will be thrown away due to the mechanisms described in section 5.6.3.

Considering the previous example with this change:

Instruction	Fetch colour	Expected colour at commit when the result arrives	
1 LDR R1,bad address	0 0	0 0	
2 B dest	0 0	0 0	
3 ADD R0,R2,R3	0 0	0 1	Toggle due to branch
4 ADD R4,R5,R6	0 0	0 1	
5 SUB R7,R8,R9	0 0	1 1	Toggle due to exception reaching commit
6 B dest2	0 0	1 1	
7 SUB R0,R0,R1	0 1	1 1	Destination of the first branch
8 BIC R0,R14,#3	1 1	1 1	The first instruction of the exception handler

The example shows that the first instruction to produce a valid result after the load is instruction 8 which is the first instruction of the exception handler. Instruction 6 which was previously executed incorrectly is now not executed because its fetch colour does not match the current expected colour.

6.2.3 Loading PC via the fetch unit

The mechanism above has reduced the branch latency by removing the delay through the reorder buffer. However, it has not reduced the delay for instructions that load the PC from memory. An approach that does reduce this delay involves allowing the fetch unit to load the PC value from memory instead of using the normal data memory access mechanism. Since this design is based on a modified Harvard architecture, the PC load can proceed in parallel with the data accesses for the rest of the LDM instruction. This modification is particularly useful in the case of LDM instructions which load the PC value after loading a number of other values from memory; as illustrated in section 4.3 this is commonly used for returning from nested subroutines.

A modified fetch unit, as shown in figure 6.5, accepts the address from which the program counter must be loaded rather than the program counter value itself. It then performs a load and routes the result back as a replacement program counter value. An advantage of this technique is that the bus carrying the loaded program counter value becomes a short local bus in the instruction fetch unit; loading the value via the data memory interface requires a bus travelling to the commit block from the data interface. With the forwarding paths from the reorder buffer there is little space left for buses in the main part of the data path and so this optimisation is useful in removing the need for yet another 32 bit bus. In addition, with this mechanism the only stream changes caused by the reorder buffer are due to exceptions so the 2nd bit of the colour described above and the *exception colour* described in section 5.6.3.2 can be implemented by the same colour bit.

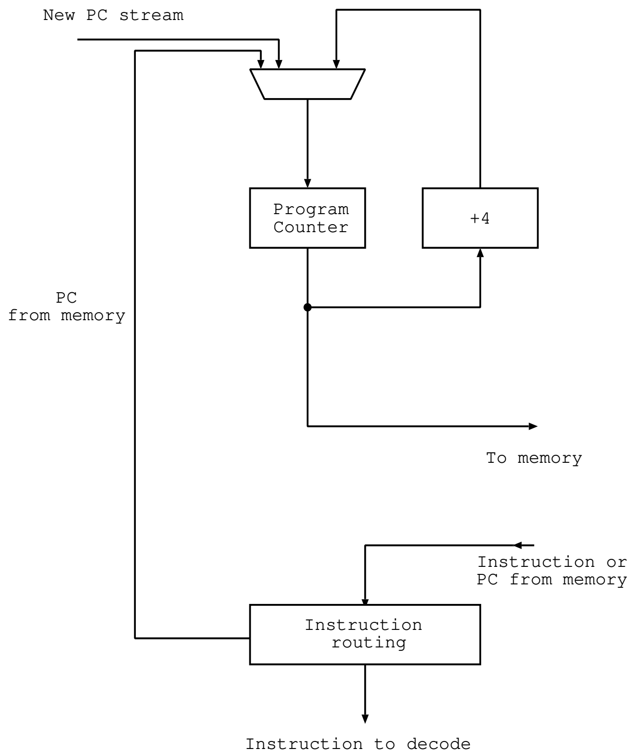


Figure 6.5 Reading the PC from memory via the fetch unit

The order of memory accesses in an LDM instruction is defined in the ARM ISA; this enables LDM to be used to load data from memory mapped input/output devices. However, the order of accesses is not important when loading from normal memory and a load which includes the program counter is unlikely to be performed from an input/output device. It can be argued that, although this deviates from the behaviour of existing ARMs, it is therefore reasonable to allow the program counter to be loaded via a separate mechanism.

This allows earlier availability of the PC value in an LDM ..,{...PC} instruction. This approach has four disadvantages:

1. The first is that if the PC load causes an exception it is not possible for the instruction fetch unit to signal this back to the rest of the pipeline. This is solved by also performing the load from the data memory and discarding the result if no abort occurs and processing it as a normal abort if it does occur. This solution will cause a small increase in the memory bandwidth used.
2. The second problem is that there is a dependency between data written from the data memory interface and the loaded PC value which is read by the instruction memory interface. To ensure correct operation the PC load must be delayed until all outstanding store instructions have completed.
3. The third problem is that the technique is incompatible with split instruction and data caches since any modifications to memory made through the data cache would not be seen by the instruction cache. In particular, where a return address had been placed on the stack recently an old stack value might be present in the instruction cache. This may be a serious limitation. However, split instruction and data caches are considered unsuitable for use with ARM code because data and instructions are often tightly interwoven. This causes contamination of the data

cache by instructions and of the instruction cache by data making poor use of the available cache size. (Note, however, that split caches are used by the Strong-ARM implementation of the ARM).

4. The mechanism produces behaviour which deviates from that of existing ARMs in the case of an LDM PC from an input/output device. This is however very unlikely to be used in practice.

6.2.4 Discarding instructions at the decode stage

In this design, complex instructions such as LDM are turned into multiple cycles in the decode stage. If these instructions should not be executed (due to a condition code failure or due to lying in the shadow of a branch) the decode stage (and potentially a number of stages below it) are still kept busy partially executing the instruction. To avoid this a mechanism is introduced that discards some instructions at decode rather than commit.

In this scheme the decode stage sets a flag whenever it decodes an instruction which definitely causes a change of instruction stream, e.g. an unconditional branch instruction. It then discards all following instructions until the flag is cleared by the arrival of a new instruction stream of a different colour. This new stream may have resulted from the execution of the instruction which set the flag or it may have resulted from an exception.

However, this mechanism does not address the problem created by conditional branch instructions since the decode must assume that the branch is not taken and carry on decoding later instructions. It has been suggested that one possible way of reducing this problem would be to send copies of the ALU flags back from the execute stage to the decode stage and so allow it to eliminate instructions earlier in the pipeline **[GaGi97]**.

6.3 The CPSR

The CPSR, described in section 4.4, consists of the ALU flags, the processor mode and a number of control flags. The ALU flags are stored in the commit block, while the mode and other CPSR control flags are stored in the fetch unit. A copy of the mode and control flags is passed down the pipeline with each instruction and its program counter value; this is used to control any instructions which behave differently in privileged modes, to permit banked register decoding and to provide access to the current mode when needed.

The reasons for these choices are:

1. The ALU flags of the CPSR are generated by the execute unit; these can be passed to the commit block along with the result from the execute unit which is already passed to the commit block.
2. Commit already invalidates instructions if their colour is wrong; with access to the ALU flags it can also invalidate instructions if their condition codes fail. This contains the task of processing ARM condition codes almost completely within the commit block.
3. The current processor mode is needed by the fetch unit to determine if instructions should be fetched from memory in a privileged mode. For this reason it is useful to store the mode in the fetch unit.
4. Almost all modifications to the CPSR control flags and mode are made during entry and exit from exception handlers when the program counter is also modified. Again it is useful to have the CPSR in the fetch unit since the commit block can send the new CPSR flags to the fetch unit at the same time as the new PC value.

5. The MRS and MSR instructions which are used by system code to explicitly read and modify the CPSR can be implemented in the commit block by routing results between the execute stage, the CPSR and the reorder buffer.

The commit block and the execute stage (including the ALU) form one pipeline stage as shown in a simplified form in figure 6.6. This single pipeline stage mechanism ensures that the previous instruction has stored its flags into the commit block before a new instruction begins execution. In this way the commit block can provide the previous instruction's ALU flag results to the new instruction.

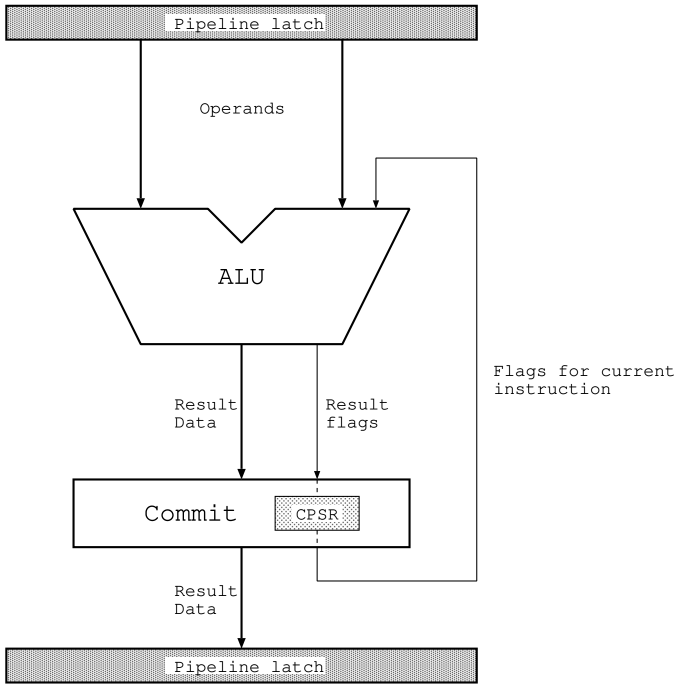


Figure 6.6 The relationship of the execute unit and the commit block

6.3.1 The Xpipe

A problem arises in situations similar to the following:

```
LDR R0,bad address
CMP R1,R2
```

The load is allowed to execute and the CMP executes and enters commit; since at this stage commit has not been notified of the failure of the load, the CMP is allowed to execute causing the ALU flags to change. By the time the load has completed the CPSR has been changed, but the exception handler must see the old value. This problem arises because access to the CPSR is not made through the reorder buffer.

The solution adopted requires the commit block to store the current CPSR in a FIFO, known as the *Xpipe*, as each memory operation is despatched. The write back process of the reorder buffer discards entries from the Xpipe as memory operations complete successfully. In the event of an exception the next Xpipe entry is passed back to commit and written back into the CPSR. The Xpipe also holds the PC value of the memory access instruction; this value is written into the R14_abort register during exception entry. The Xpipe is, in effect, a simple history buffer.

6.4 The SPSRs

The SPSRs hold a copy of the CPSR taken at the time of exception entry and are used to restore it after exception processing has completed.

Two exception handling mechanisms were considered for the SPSRs. The first is to treat the SPSRs in a similar manner to conventional registers, storing them in the register bank and using the reorder buffer to perform dependency and exception handling. Unfortunately the MSR instruction is able to modify a subsection of the current SPSR. This leads to the problem that it may be necessary to combine partial SPSR results from multiple reorder buffer entries and the register bank. This would significantly complicate the reorder buffer read process.

The second option is to store the SPSRs in a separate unit in the commit block. This makes copying between the SPSR and CPSR much easier but extra hardware must be added to manage SPSR values during exceptions.

6.4.1 The interaction of the SPSRs and data aborts

The SPSR is vulnerable to the same form of corruption as the CPSR during data aborts. For example:

```
LDR R0,bad address
SWI .....                     causes entry into SVC mode
```

In this code the SWI could enter the commit block before the abort response has been returned by the load and so update the SPSR_svc register to contain a copy of the current flags and mode. When the memory exception is detected, abort mode will be entered (with SPSR_abt set appropriately) but with the SPSR_svc incorrectly modified by the SWI which lies in the shadow of the aborting load.

Three mechanisms have been considered to handle this eventuality:

6.4.1.1 Expansion of the Xpipe

The Xpipe could be expanded to hold copies of *all* SPSRs (other than the abort SPSR which the data abort will overwrite anyway) and the Xpipe entry could then be used to restore all SPSRs upon the abort exception entry. Since SPSR writes are rare the performance impact is not significant, however the Xpipe has become much larger.

6.4.1.2 Locking the SPSRs

A semaphore could be created to represent the number of outstanding memory operations; when this is zero it would be safe to perform writes to the SPSR. Thus all instructions writing to the SPSR (other than memory operations which only cause a write in the case of a data abort) must wait for the semaphore to be zero. The semaphore would be decremented by the write back process of the reorder buffer as it checks each memory operation has successfully completed.

6.4.1.3 Duplicate copies of the SPSRs

Each SPSR could be represented by two registers and a generation bit could state which is the current copy; this is represented in figure 6.7. When an SPSR is written only one copy is modified, the other copy holds the old SPSR value, the generation bit is set to indicate the current valid copy. During a memory operation the current state of the generation bits is stored; if an exception occurs it is only these which need to be restored. Extra logic would be needed to ensure that given a number of memory operations and SPSR modifications that the old SPSR values were not discarded until the appropriate memory operations had completed.

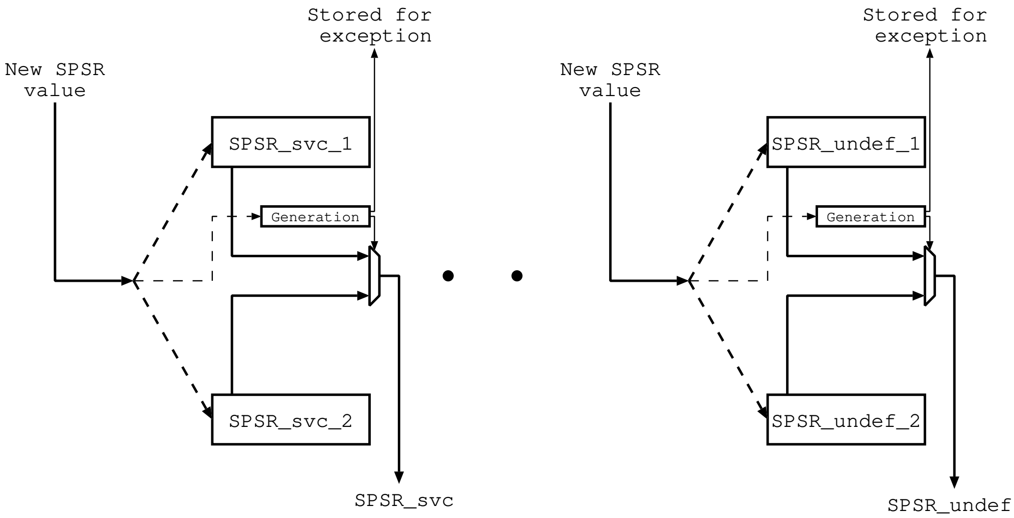


Figure 6.7 Duplicate copies of the SPSRs

6.4.1.4 Comparison of SPSR exception handling mechanisms

Expanding the Xpipe to hold SPSR values is similar in principle to a history buffer; it has the disadvantage that it would use a relatively large amount of silicon area and significantly complicate the SPSR writing mechanism. Keeping duplicate copies of the SPSRs is similar to register renaming with a mapping table. Keeping duplicate copies is rather complex and thus was not chosen. Locking the SPSRs is relatively simple and is suitable providing that the locking does not cause pipeline stalls very often. Section 7.3.4 analyses the impact of this locking.

6.5 Base restoration

Section 4.9 describes how an LDM instruction may load a new value for its base register and then cause an exception so losing the original base register value. Delaying write back of the loaded values into the register bank until after the last memory cycle has completed would solve this problem but it would require a reorder buffer of at least sixteen entries which would be expensive and slow.

The solution used in AMULET1, AMULET2 and this design is a FIFO, known as the *Base Restore Pipe* (BRP) into which the commit block places the value and name of the base register. The write back process of the reorder buffer removes entries from the BRP at the end of LDM instructions as shown in figure 6.8. If the LDM completes successfully the result is discarded; however if any element of the LDM caused an exception the value in the BRP is written back to the base register before the exception handler is entered.

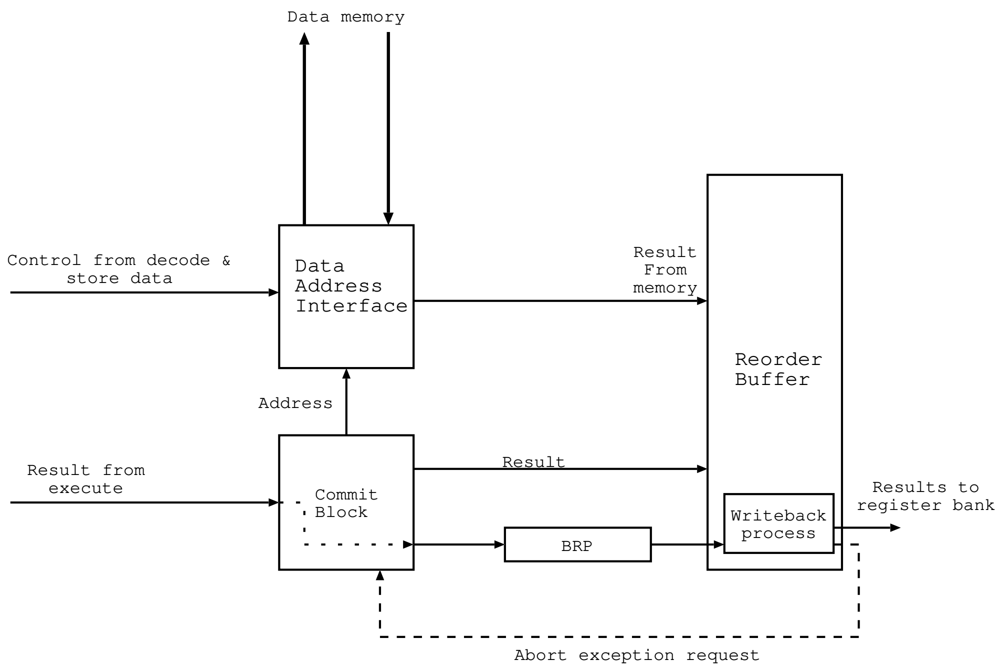


Figure 6.8 The Base Restore Pipe (BRP)

6.6 Exceptions in the memory

The model described above allows for a pipelined memory system as described in section 3.7 and so the commit block can issue a stream of requests to the memory without waiting for each request to return an exception response. Requests may be made to the memory subsystem for operations which lie in the shadow of other memory accesses which have caused exceptions.

This design places the responsibility for handling this problem with the memory subsystem which must discard all accesses in an exception shadow.

6.6.1 Pipelined memory and reorder buffer size

Pipelining the memory subsystem increases the number of instructions which have been issued but have not yet returned a result. This requires an increase in the size of the reorder buffer so that entries can be allocated for the extra outstanding results.

6.7 Interrupts

The external interrupt lines enter this model at the decode stage where they are combined with the interrupt enable flags flowing down with the instruction. If the interrupt is enabled the current instruction is replaced by an instruction which causes the exception entry. This instruction is similar to a SWI except that it causes a different mode to be entered and a different vector to be read.

The interrupt pseudo-instruction operates in the same way as any other instruction passing down the pipeline and is vulnerable to being invalidated if a previous instruction causes an instruction stream change. Thus an interrupt which replaces the instruction immediately after a branch is ignored. Since the interrupt system is level sensitive rather than edge sensitive this does not cause a problem as the decode stage will continue to replace all instructions by pseudo-interrupt instructions until either the interrupt source is removed or interrupts are disabled. Since the interrupt instruction itself disables interrupts on entry to the exception handler the decoder will start executing instructions from the interrupt handler as normal.

A consequence of this mechanism is that it is very difficult to calculate the maximum interrupt latency; this is a major disadvantage for time-critical embedded systems. Further research in the AMULET group is, at the time of writing, investigating improving the interrupt mechanism.

6.8 Long multiplication

ARM architecture 3M and ARM architecture 4 define a family of multiply instructions which take two 32 bit operands and produce a 64 bit result. The result is placed in two separate registers. One variant of this instruction is a 64 bit multiply and accumulate which reads two values to be multiplied, two registers as a 64 bit value to accumulate and writes two registers representing the 64 bit result. This instruction is unusual in that it can read up to 4 register values and write two non-memory results.

The long multiply instructions can be split into multiple pipeline packets in the decode stage in a similar way to LDM instructions. Care must be taken with the allocation of spaces in the reorder buffer so that a later cycle of the multiply does not wait for a result register allocated in an earlier cycle in such a way that a deadlock occurs. The details of this problem depend on the way in which the multiplier is implemented and in particular whether all four operand registers are needed to produce the first result register. This problem is not investigated further in this thesis.

6.9 Summary

Figure 6.9 is a more detailed diagram of the architecture showing the features described in this chapter. This diagram also incorporates a number of other details which were omitted from previous diagrams for clarity:

- All three register read ports are shown. Port A is the base register for memory operations. Together ports A and B form the operands for normal data operations. Port C provides store data and register based shift values for data operations.

- The data memory is now connected to the core via the *Data Address Interface* (DAI). This synchronises store data and control information with the address from the commit block and then accesses the data memory as appropriate.

- The *Apipe* (so named because it holds the values from the A register port) passes values to the commit block bypassing the execute unit. This is used during memory operations which require both an original base register value and an incremented value. The original base register is read from the Apipe while the modified value is received from the execute block.

- A third result port on the reorder buffer (Cancel) has been added; this is used for placing invalid results in the reorder buffer for memory operations which failed their colour or condition code tests. This is simpler than passing dummy operations down the memory pipeline.

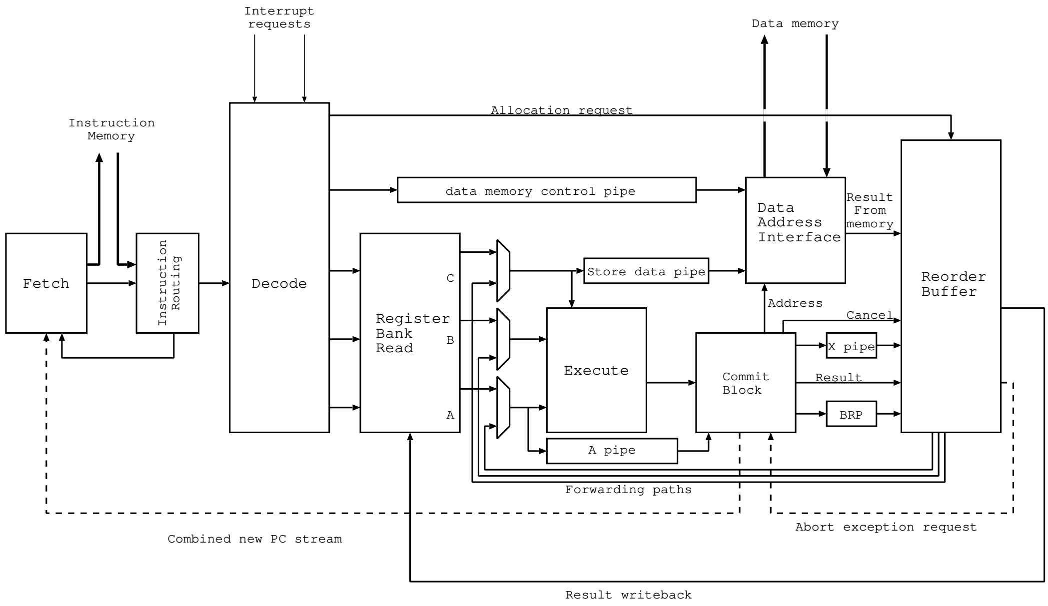


Figure 6.9 Architecture overview

The pipeline stages (although not shown explicitly in this diagram) can be summarised as:

- Instruction address generation (fetch 1)
- Instruction cache/memory read (fetch 2)
- Decode/Reorder buffer allocate and lookup
- Register read/Reorder buffer read
- Execute/commit
- Data memory access - only for memory access instructions.

The reorder buffer itself can also be regarded as a pipeline stage since it accepts a result and then writes it out sometime later to the register bank.

Many of the dependency and exception handling problems have been confined to the commit block which maintains the PSRs and the expected instruction colour and routes values between the execute unit, the memory, the reorder buffer and the fetch unit. The only reservation about this approach is that the commit block has become more complex than desired; further research is needed into ways of splitting this block into smaller, more manageable sections.

Chapter 7: Simulation and results

This chapter describes the simulation models which have been created to aid this research. These models are then used to compare the performance of the architecture proposed in earlier chapters with previous architectures. The simulations are also used to assess the benefits of small enhancements to the architecture.

7.1 The simulation environment

Two forms of simulation were undertaken, *trace based simulation* and *behavioural simulation*.

7.1.1 Trace based simulation

The first form of simulation was trace based analysis. This used an ARM emulator to produce a trace of instructions fetched and data accesses performed by benchmark programs. This trace was then fed into an analyser written in C. This analyser produced statistics which will be presented in section 7.3.1.

7.1.2 Behavioural simulation

To test the architectural design presented in chapter 5 and chapter 6 a behavioural model of the architecture was written in VHDL. This model consists of behavioural descriptions of each block of the design with a structural description of the interconnections between blocks. The model is sufficiently detailed to allow normal

ARM programs to execute. The memory model simulates memory mapped I/O allowing the ARM program to gain access to files on the host computer and provide equivalents of the standard input and output streams. This memory mapped I/O is typically accessed via a small OS written by the author (called ‘Davros’) which provides a system call interface compatible with the ARM ‘Demon’ OS **[ARM95]**. This enables ANSI standard C programs (which can call the ANSI C library) which were compiled for the ARM ‘PIE’ card to run unaltered under the simulation model. The model was parameterised to allow the size of the reorder buffer to be altered and the memory timings to be changed.

Three variants of the architecture were modelled:

- Reorder buffer with forwarding - This is the architecture described in chapter 5 and chapter 6. Figure 6.9 is a detailed block diagram of this model.

- Reorder buffer without forwarding - This architecture, shown in figure 7.1, uses no forwarding and has a single lock FIFO (see section 2.2.2) to resolve dependencies. This represents a combination of the old lock FIFO method and the method being proposed. Its advantage is that it does not need the complex searching mechanism used during forwarding.

- Dual lock FIFO model - This architecture uses no reorder buffer and instead has two lock FIFOs to resolve dependencies. The two result streams arbitrate for the result bus in the *join* block. The result, together with an indication of the source of the result, is passed to the *write control* block which removes an entry from the appropriate lock FIFO and writes the result into the register bank. This model uses dependency and exception handling mechanisms similar to those used in AMULET2. Figure 7.2 shows this architecture in more detail.

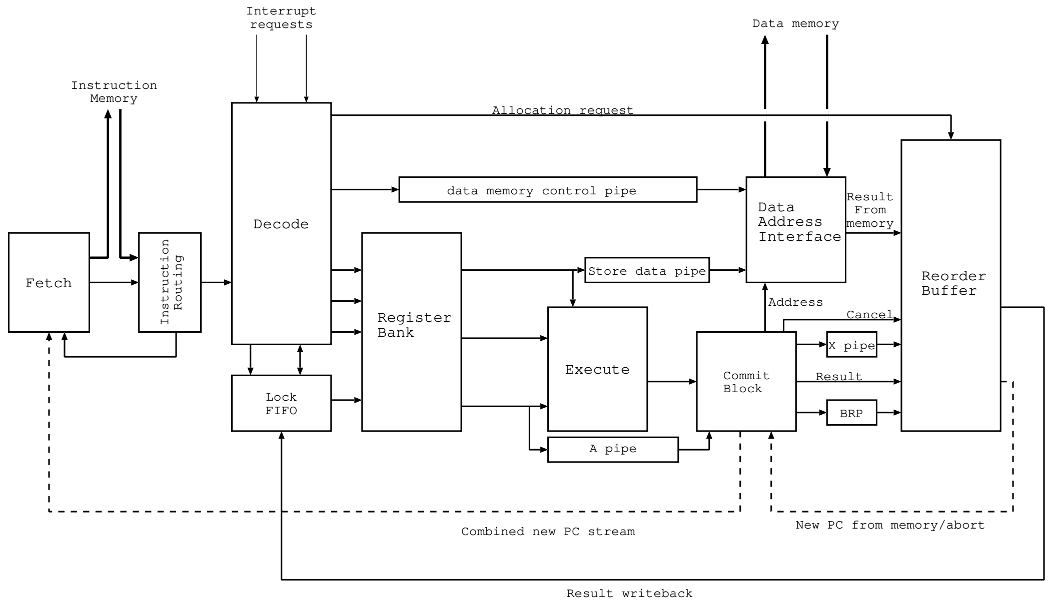


Figure 7.1 Model as simulated: Reorder buffer without forwarding

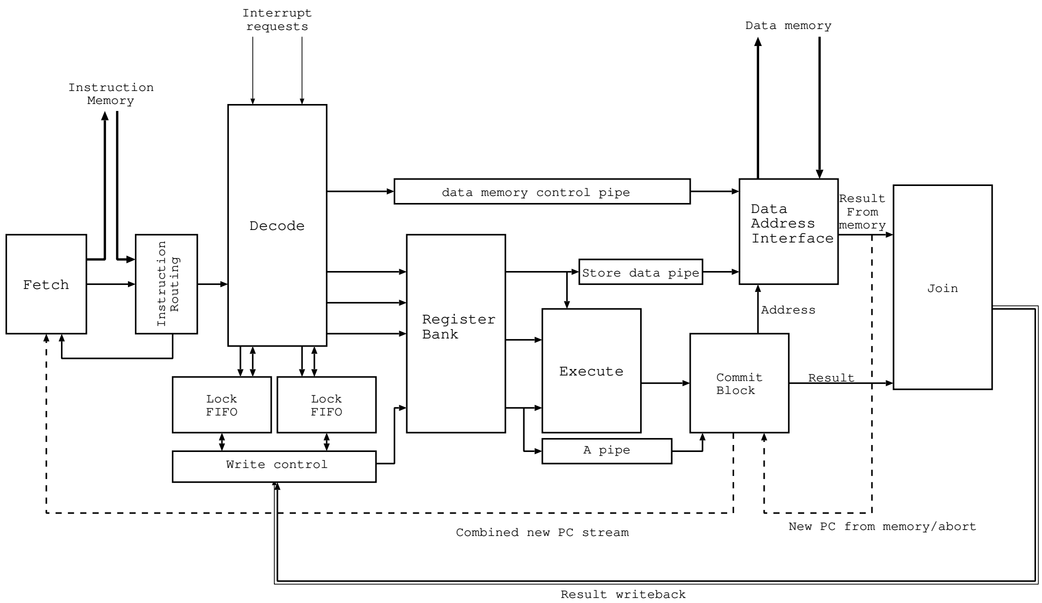


Figure 7.2 Model as simulated: No reorder buffer

Last result reuse, as used in AMULET2 and described in section 2.3.2.2, was used in the model without forwarding and in the lock FIFO model (although without a last memory result register). A 32 entry branch-target-cache, similar to that used in AMULET2, is incorporated in the fetch mechanism.

7.1.2.1 Timing

At the time of writing other members of the AMULET group are taking the architecture described in this thesis, building upon it and designing low level schematics to implement it; thus at this time there is no accurate timing information available. Due to this lack of timing information the simulation models used in this research have been written with the assumption that the designers' aim is to produce a balanced pipeline in which all pipeline stages take approximately the same time; this is common since a balanced pipeline leads to more efficient utilisation of the hardware. In practice pipelines are never balanced and in asynchronous systems pipeline stages take data dependent times. However, it is a common design goal.

Throughout the model two separate time units are used. One large unit is used to represent pipeline blocks and another very small unit is used to represent the logic involved in handshakes and synchronisation. In this model 100ns[1] was used for the large time and 1 ps for the small time; thus the handshake time can be effectively ignored.

One consequence of this overly simple model is that, as its size is changed, the time taken to search the reorder buffer remains constant. This should be considered when reading the results later in the chapter.

7.1.2.2 Choice of modelling language

In retrospect the choice of VHDL as a behavioural description language for an asynchronous microprocessor was poor. VHDL was chosen over a conventional programming language (such as C or C++) because of its concurrent programming capabilities, its inbuilt concept of time and its integration with available CAD tools. It was found that it is possible to model an asynchronous system using VHDL's model of concurrency

1. These figures are arbitrary - they are just used to distinguish calculation time from handshaking time during simulation.

but the resulting model is overly complex. The author believes that the main reason for this is that the act of creating a parallel process in VHDL is complex and is more suited to expressing large scale parallelism rather than the fine grain parallelism of the type needed to represent the handshaking mechanisms in asynchronous systems. The effect of this is that the model is dominated by code representing the parallelism in the handshaking and the actual problem is hidden from those examining the model. A language for asynchronous development should make this parallelism simple to express and hide the details of the handshaking mechanisms. Another difficulty with VHDL is a consequence of its 'wait until' statement. Often in an asynchronous design a block will need to wait for a request to be high (for example); initially it might be thought that the VHDL statement:

```
wait until Req='1';
```

would be suitable. However if 'Req' was already '1' this statement would wait until it changed away from '1' and then back, thus the required VHDL is:

```
if Req/='1' then
 wait until Req='1';
end if;
```

While this structure can be placed inside a function for frequent use there are many variations on this same problem.

At the time when the author started writing the model no asynchronous hardware description languages were available for use. However, there are now a number of languages such as BALSA **[Bard96]** and LARD under development in the AMULET research group. These appear to be better suited to the task of describing asynchronous systems.

7.2 Benchmarks

One of the limitations of the VHDL model is its execution speed: around 20 ARM instructions per second on a SUN SparcStation 5. The simulations were run mainly at night using a large number of workstations, each simulating a different configuration of the architecture. This limits the dynamic number of instructions in the simulation to that which can be achieved overnight to around 750 000 instructions. Fortunately the forwarding and dependency mechanisms operate over a short range of instructions and so simulation of small programs is a valid method of assessing the system's performance.

Given the constraint on the execution speed of the simulator, large benchmarks such as the SPECint95 suite could not be used. Instead the following six benchmarks were used:

- *Dhrystone Version 2.1* - This is the standard 'Dhrystone' benchmark, implemented in C (compiled without the 'register attribute' option). Ten loops of the benchmark itself are executed.

- *Numeric sort* - One thousand random numbers are generated and then sorted using the quick sort routine in the standard C library.

- *String sort* - This program reads a text file from its standard input stream, sorts it and then prints it out on its standard output stream. The sort is performed using the standard C library quick sort routine. The text to be sorted is the ctype.h header file from the ARM cross development toolkit.

- *Espresso* - The espresso logic minimisation tool with a small input file.

- *Mandelbrot* - A program to calculate and display a section of the Mandelbrot set. This program makes heavy use of floating point calculations which, for the purposes of these experiments, were performed via a floating point emulator which intercepted floating point instructions using the ARM's undefined instruction exception vector.

- *JPEG* - A program to decompress a JPEG image file. This benchmark consists of the independent JPEG group's *djpeg* program (version 6a) decompressing a small JPEG image. The decompressor is configured to use a fast integer mode of operation rather than the optional floating point mode.

All the benchmarks are written in C and have been compiled using the compiler in the ARM cross development toolkit **[ARM95]**.

7.3 Results

This section presents the results of the simulation models described above.

7.3.1 Results from the trace based simulation.

The first set of results from these simulations is the distribution of instruction types. Table 7.1 shows the distribution of instruction types for each of the benchmarks and the mean figure.

	Mean	Dhrystone	Espresso	JPEG	Mandelbrot	Numeric Sort	String sort
Data Op	56.83%	52.25%	46.45%	56.65%	71.38%	60.73%	53.52%
Single Load/ Store	22.12%	23.30%	32.48%	24.19%	6.28%	22.21%	24.26%
Multiple Load/Store	3.40%	4.00%	4.40%	3.94%	4.48%	0.11%	3.48%
Multiply	0.29%	0.01%	0.22%	0.54%	0.92%	0.07%	0.01%
Branch	17.18%	20.38%	16.41%	14.68%	16.04%	16.86%	18.70%
SWI	0.02%	0.06%	0.02%	0.01%	<0.005%	0.01%	0.04%
Coprocessor	0.02%	<0.005%	0.01%	<0.005%	0.13%	<0.005%	<0.005%
PSR transfer	0.14%	<0.005%	0.03%	<0.005%	0.77%	<0.005%	<0.005%

Table 7.1: Instruction set usage (dynamic)

The entries in the table represent[1]:

- Data Op – Data operations involving one or two registers with a register destination.
- Single Load/Store – LDR or STR instructions.
- Multiple Load/Store – LDM or STM instructions.
- Multiply – All ARM Multiply instructions.
- Branch – Explicit Branch or Branch and link instructions; this figure does not include branches caused by data operations or loads, such as procedure returns.
- SWI – Software Interrupt.
- Coprocessor – Any form of coprocessor instruction.
- PSR Transfer – The ARM MSR and MRS instructions.

1. More details of the ARM instruction set can be found in Appendix A.

These figures show that on average 26% of instructions are memory operations and so may cause an exception. The SWI and coprocessor instructions which cause exceptions are executed rarely, even in the Mandelbrot program which makes heavy use of emulated floating point coprocessor instructions. Thus instructions which *potentially* cause exceptions occur frequently and exception handling mechanisms must be able to handle these at least as efficiently as instructions which always cause exceptions.

The second important result from the trace based simulations relates to the age of operands. The X axis of figure 7.3 represents the age of the operand and is 1 if an operand was written in the previous instruction. The Y axis represents the percentage of operands written within the corresponding number of instructions; thus (from the mean result) 51% of operands were written 5 or fewer instructions prior to the current instruction. Last result reuse (as used in the AMULET 2) can only avoid the penalty of a RAW dependency if the operand was produced on the *previous* instruction; this case accounts for only 25% of operands and so half of the operands written recently cannot be forwarded using that mechanism.

In a system with a reorder buffer the return of results to the register bank is delayed; this graph shows that the results in the buffer are likely to be required by subsequent instructions. Not forwarding from the reorder buffer is therefore likely to stall the pipeline.

The graph shows that between 20% and 30% of operands are more than 20 instructions old. This behaviour can be accounted for by cases such as registers being written prior to a loop and being read many times from within it.

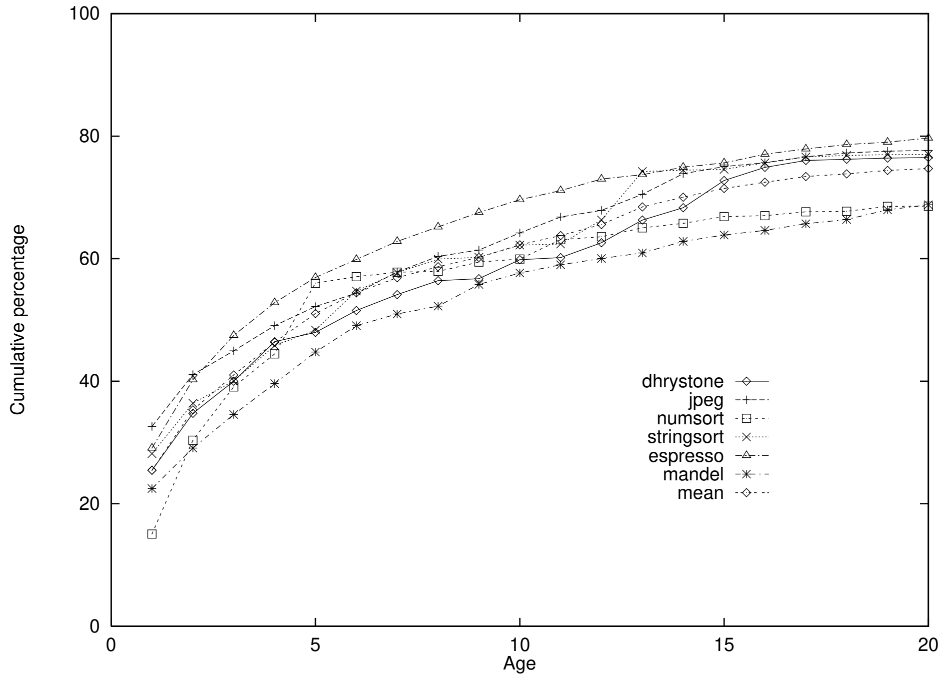


Figure 7.3 Operand age distribution

7.3.2 The benefits of the reorder buffer

Using the VHDL model described above simulations have been carried out to compare the reorder buffer architecture with the lock FIFO architecture. Figure 7.4 shows a comparison for the JPEG benchmark. All times are measured relative to the lock FIFO model using a memory model that returns an abort response in half the time allocated to the delay through one pipeline stage (i.e. comparable to 0.5 cycle); this model is represented by a horizontal line at 1 on the execution time axis; the other two horizontal lines represent lock FIFO systems with faster and slower abort responses. From this graph it can be seen that reducing the abort response time to 0.1 cycle time in the lock FIFO model produces a small performance change of approximately 2%.

7.3.2.1 Reorder buffer without forwarding

From figure 7.4, and figure 7.5 (which show just the graphs relating to the reorder buffer without forwarding for all benchmarks), it can be seen that for small sizes of reorder buffer (marked ROB on the graph) *without* forwarding the execution time is

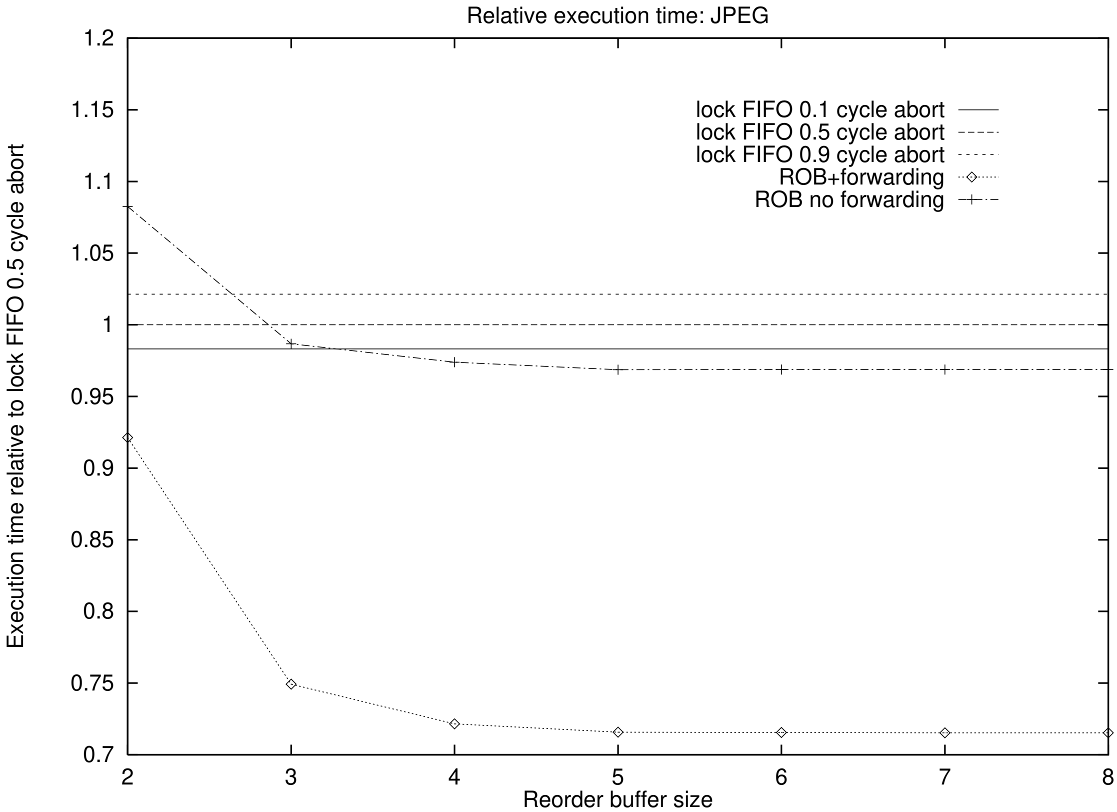


Figure 7.4 Relative execution times for the JPEG benchmark

worse than that of the lock FIFO model while for large sizes of reorder buffer the performance is slightly better than the lock FIFO model. The reason for the performance loss at small sizes is believed to be that a significant proportion of the reorder buffer allocation requests cause the processor to stall, waiting for spaces in the reorder buffer; this is substantiated by figure 7.6 which shows the percentage of allocation requests which had to stall waiting for a space.

At larger sizes of reorder buffer the abort response time penalty has been removed leading to much of the performance gain shown. In addition the processor is free to execute instructions free of dependencies in the shadow of memory operations which could potentially cause exceptions.

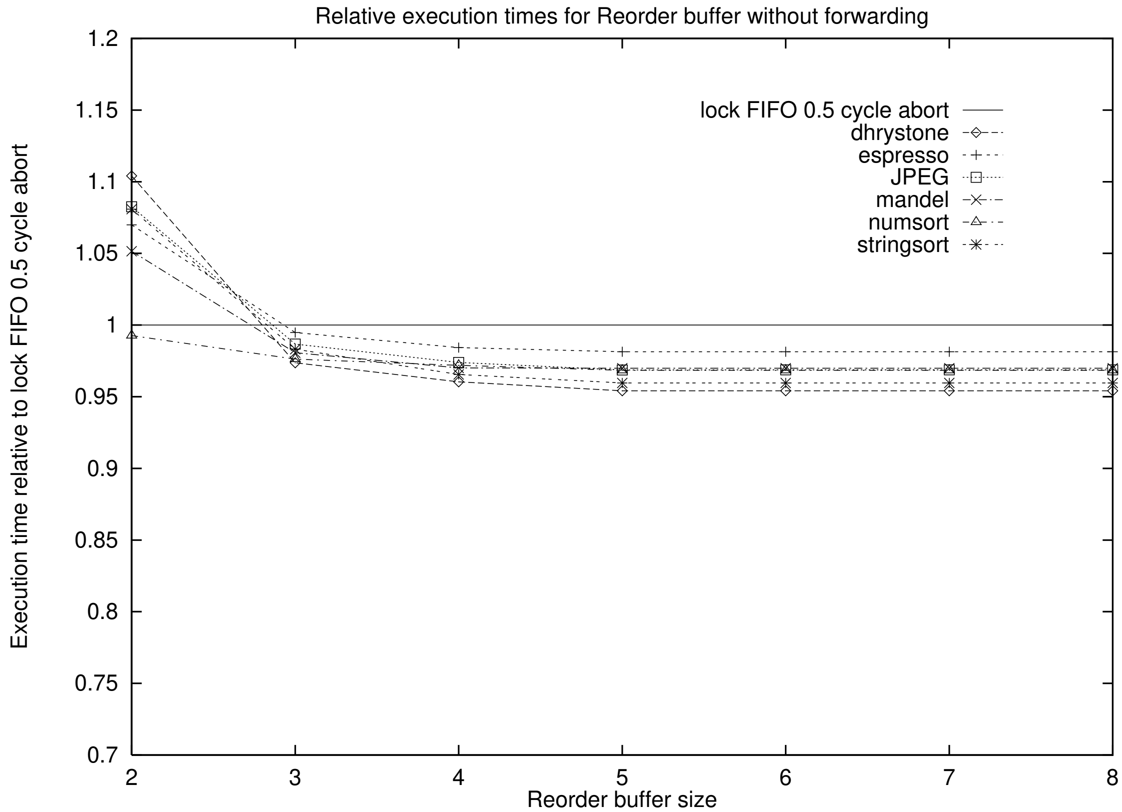


Figure 7.5 Summary of execution times for reorder buffer without forwarding

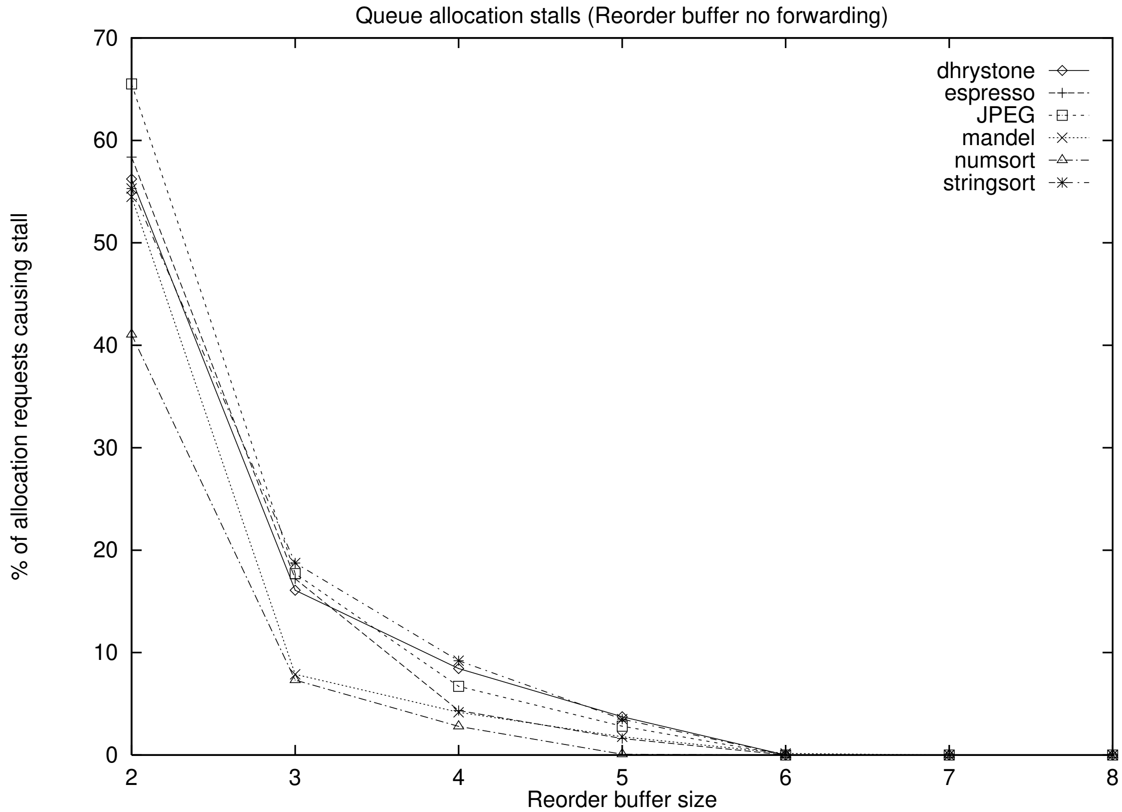


Figure 7.6 Percentage of reorder buffer allocations which had to stall (no forwarding)

7.3.2.2 Reorder buffer with forwarding

From figure 7.4, and figure 7.7 (which shows just the graphs relating to the reorder buffer with forwarding for all benchmarks), it can be seen that in all cases the reorder buffer with forwarding has produced a reduction in execution time; this is true even for small reorder buffer sizes where the benefits of forwarding more than offset the penalties of waiting for space in the buffer. At larger sizes of reorder buffer the reduction in execution time is significant at between 19% and 28%. Figure 7.8 shows the percentage of reorder buffer allocation requests which had to stall in the model with forwarding. Thus it can be seen that at a reorder buffer size of 6 very few allocation requests cause stalls since there is almost always some free space in the buffer.

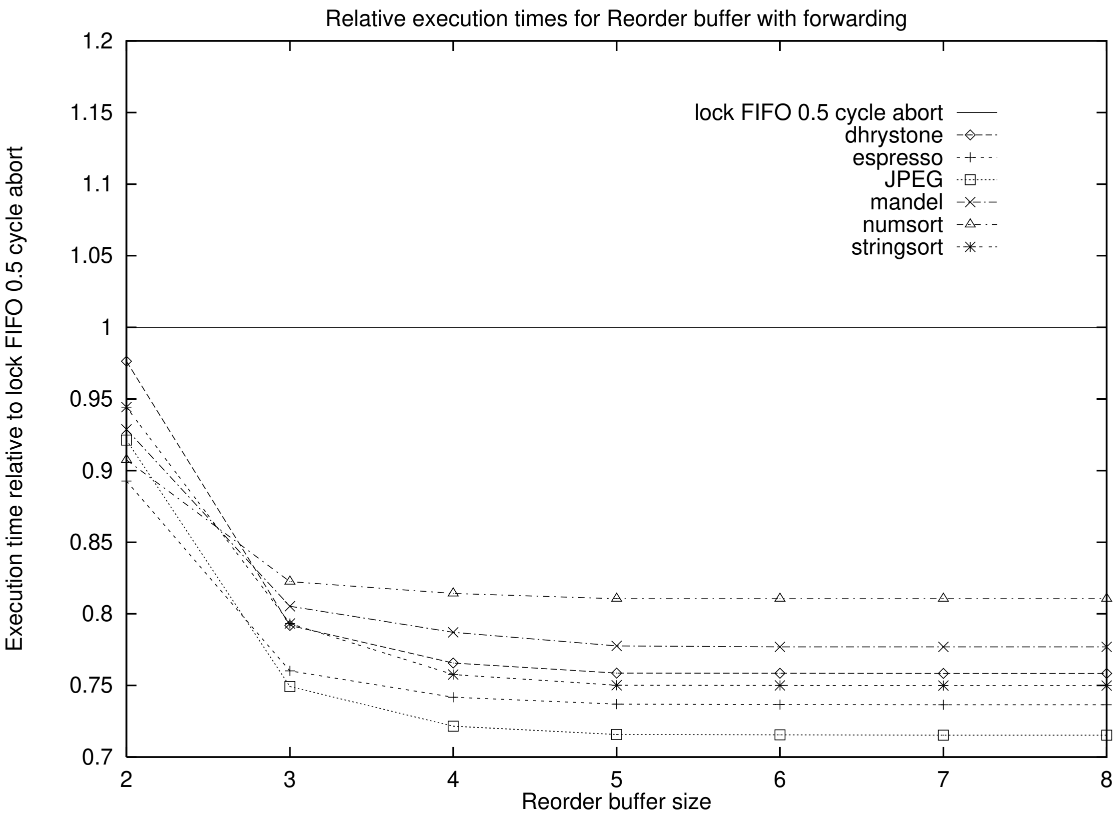


Figure 7.7 Summary of execution times for reorder buffer with forwarding

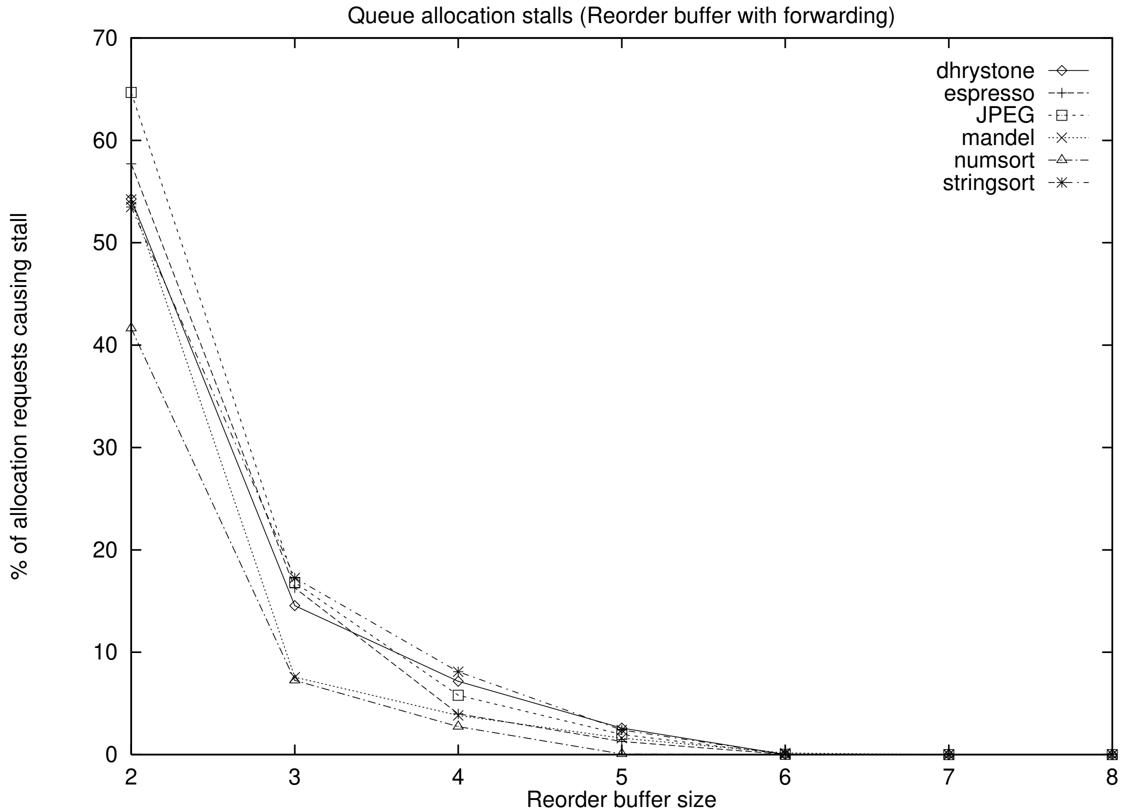


Figure 7.8 Percentage of reorder buffer allocations which had to stall (with forwarding)

7.3.2.3 Appropriate sizing of the reorder buffer

From figure 7.7 it can be seen that very little increase in performance is gained by increasing the reorder buffer above 4 entries in size. In practice increasing the size of the buffer will probably slow the implementation and so the small performance increases which can be seen for the 5 entry reorder buffer in the graph above will probably not be seen in an implementation.

For this reason a 4 entry reorder buffer would appear to be the best configuration in designs based on the architecture modelled.

7.3.3 Loading the PC via the instruction fetch mechanism

Section 6.2.3 described how the instruction fetch mechanism can be used to execute instructions which load a new PC value from memory. This mechanism is used partly because it is easier to implement and partly because it promises increased performance. It does, however, have the disadvantage that it is incapable of working with split instruction/data caches and it also uses more memory bandwidth. The simulations presented above have been performed with this mechanism enabled.

Table 7.2 shows the execution times of the benchmarks executing on an architecture with a 4 entry reorder buffer, with forwarding, without using the fast PC load mechanism relative to an architecture with the fast PC load mechanism. The performance benefit of this mechanism is small; this is because PC loads are relatively rare in the benchmarks used. However, it is still felt that this mechanism is useful because of the likely simplification of the implementation **[GaGi97]**.

Benchmark	Relative execution time
Dhrystone	1.024
Espresso	1.015
JPEG	1.012
Mandelbrot	1.002
Numeric sort	1.000
String sort	1.020

Table 7.2: Relative execution time: PC load via data interface/PC load via fetch interface

7.3.4 The penalty of SPSR locking

In section 6.4.1 a number of mechanisms were proposed for maintaining the correct value of the SPSRs after an exception. The simplest of these, SPSR locking, involves holding a count of the number of outstanding memory operations, and thus potential exceptions, and stalling updates of the SPSR until there are no outstanding memory operations. The simulations described above have used this locking mechanism. Although this mechanism will stall the processor it is expected to occur infrequently and thus cause a very small reduction in overall performance. To ascertain whether a more complex but more efficient mechanism is needed simulations were carried out with the mechanism disabled; this meant that the architecture was incapable of processing an exception correctly, but the penalty for checking the lock was not incurred. The performance difference between this model and the normal model with locking enabled shows the penalty due to the locking. The simulations were carried out only on a model with a reorder buffer of four entries with forwarding. For this model all benchmarks showed a performance loss of well under 0.01% when using SPSR locking. For this reason it is recommended that the SPSR locking mechanism be used since the performance loss is insignificant.

7.4 Absolute performance

The results presented so far in this chapter have been comparative timings between different versions of the author's own simulation models; this section compares the absolute performance of these models with the performance of existing ARM processors. To assess the performance of other ARMs the *ARMulator*, a cycle level ARM simulator produced by ARM, **[ARM95]** was used to simulate the execution of the benchmarks on a variety of ARM processors.

Comparing absolute performance at the cycle level is difficult because the length of the cycle is influenced by the implementation and by the different organisations of each processor's pipeline. For this comparison the cycle time of all processors was treated as equivalent by configuring the ARMulator to run with a cycle time of 100ns which was equivalent to the pipeline stage time of the VHDL models. This is unrealistic but is the best which can be achieved until implementations have been produced for the architecture described in this thesis. Table 7.3 shows the raw results.

	ARM7	ARM8	Strong-ARM	ROB 4 entry	ROB 8 entry
Dhrystone	32.88	27.86	25.37	27.64	27.38
Espresso	92.28	75.24	68.86	89.76	89.13
JPEG	134.87	115.91	103.87	115.55	114.56
Mandel	108.58	104.79	N/A[a]	110.86	109.42
Numeric sort	44.27	37.42	32.40	47.00	46.78
String sort	64.53	54.92	49.27	55.04	54.48

Table 7.3: Absolute execution times (in ms)

a. The ARMulator failed to complete the Mandel benchmark when configured for StrongARM; since the benchmark successfully completed on all other simulators and configurations it is believed this is due to a fault in the ARMulator. This has been reported to ARM but no solution has been found.

From these results it can be seen that the asynchronous model with a four entry reorder buffer is faster than the ARM7 for most benchmarks. For three of the 6 benchmarks the four entry model is comparable in performance to the ARM8 while for the other three benchmarks it is significantly slower than the ARM8. For all benchmarks the asynchronous model is slower than the StrongARM.

The speed relative to StrongARM is summarised in Table 7.4. The remainder of this section will explain the reason for the low performance of the model when compared with the StrongARM.

	% slower
Dhrystone	8.93
Espresso	30.35
JPEG	11.24
Numeric sort	45.06
String sort	11.72

Table 7.4: Percentage slower than StrongARM (ROB 4 entry)

7.4.1 Cost of stream changes

The model as presented has a stream change (branch) cost of 6 cycles compared with the StrongARM which can execute a branch in one or two cycles depending on the form of branch. However, this model has branch prediction which StrongARM lacks. The StrongARM has branch hardware using a specialised path for executing subroutine returns in one cycle and a separate branch adder to speed the execution of normal branch instructions.

Stream changes can be broken down into 7 types:

	StrongARM	VHDL
Unpredicted, untaken	1	1
Unpredicted, taken	2	6
Mispredicted	1[a]	6
Predicted	2	1
Other	2[b]	6
LDR PC	4	7

Table 7.5: Branch costs on StrongARM and the VHDL model (cycles)

a. This figure represents the time to execute a branch which would have been mispredicted in the VHDL model

b. This represents subroutine returns and other stream changes caused by arithmetic operations writing to the PC - for the purposes of this investigation these are assumed to be mainly due to subroutine returns. It is also assumed that no dependency exists on the value transferred to the PC.

From this point on unpredicted, untaken branches will be ignored since they execute in the same time on both processors. Table 7.6 shows the frequency with which each branch type occurs in the benchmarks. Table 7.7 shows the costs of branches in the benchmarks for the two architectures while Table 7.8 shows the total execution time for the VHDL models with the branch cost above a StrongARM removed; this table also includes the percentage by which these times are slower than the StrongARM.

	Dhrystone	Espresso	JPEG	Numeric sort	String sort
Unpredicted, taken	2515	17637	4725	4141	4342
Mispredicted	116	4464	2079	3004	647
Predicted	21104	32165	68399	17995	40168
Other	441	2481	347	26011	2023
LDR PC	3992	6538	7226	95	6916

Table 7.6: Branch type occurrence in benchmarks

	StrongARM cycle cost	VHDL ‘cycle’ cost	VHDL cycle penalty	VHDL time penalty
Dhrystone	64204	67480	3276	0.328 ms
Espresso	135182	225423	90241	9.02 ms
JPEG	177925	161887	-16038	-1.60 ms
Numeric sort	99678	217596	117918	11.79 ms
String sort	121377	130652	9275	0.93 ms

Table 7.7: Branch costs for benchmarks

	VHDL compensated time	% slower than StrongARM
Dhrystone	27.31 ms	7.64
Espresso	80.73 ms	17.25
JPEG	117.16 ms	12.79
Numeric sort	35.20 ms	8.66
String sort	54.12 ms	9.84

Table 7.8: Benchmark performance having compensated for branch costs

From these tables it can be seen that the numeric sort’s excellent performance on the StrongARM is primarily due to its fast path for return from subroutines which the numeric sort performs often.[1] In general it can be seen that the long branch latency in the VHDL models are far too high and contributes to a significant performance loss.

7.4.2 Load latency

A load followed by an instruction which accesses the data incurs a one cycle penalty on the StrongARM and a two cycle penalty in the architecture presented here. This difference appears to account for the majority of the remaining performance loss

1. This is mostly likely due to the comparison being performed in a separate small subroutine which will be called many times.

compared to StrongARM. This theory is reinforced by the observation that the Espresso benchmark, which has significantly worse performance compared to StrongARM, performs more memory operations than the other benchmarks (see Table 7.1). The benchmark programs were compiled with a compiler designed for an ARM7 processor which does not suffer from avoidable penalties due to load latency and which therefore does not need to schedule instructions between loads and instructions dependent on the load data. Thus optimisations which would remove some of the load latency penalty have not been performed.

7.4.3 Stage complexity

The reason for both the increased branch and load latency is the larger number of pipeline stages in the model presented when compared with any of the commercial ARM implementations. In particular the model as presented splits the decode and register read into separate stages which the StrongARM performs in one stage. Thus comparing the two models at the cycle level is bound to be inaccurate.

This pipeline stage was split because the author was concerned that the decoding of instructions (which is potentially complex and slow on the ARM), together with the logic to perform the reorder buffer search, might slow the stage down to the point where it became the limiting speed factor. If this is indeed the speed limiting stage the split may allow a faster cycle time and a time performance comparison (rather than a comparison at the cycle level) may come out more favourably in terms of the asynchronous architecture.

Determining which stage limits the cycle time is difficult without implementation and simulation at the gate level. Since producing a gate level implementation is a long and complex task it is not in general possible to correct the architecture (in terms of the number of pipeline stages) after the implementation is completed and it is found that a stage should be added or removed. At the same time it is difficult to determine the number of pipeline stages to use until the gate level implementation is completed. The author believes that this paradox can only be solved by the use of higher level simulation and synthesis which allows the designer to determine which stage was the limiting stage earlier in the design process and allow the architecture to be adjusted as appropriate.

7.5 Summary

The results given in this chapter show that the dependency and exception mechanisms described in this thesis provide a significantly higher level of performance than those found in the earlier AMULET designs. The key points are:

- The reorder buffer with forwarding provides a performance benefit of more than 20% compared to a model using lock FIFOs (as in AMULET1 and AMULET2) for the benchmarks used in this work.

- A reorder buffer of four entries is recommended. While these results show a slight performance increase for a five entry buffer this may be offset by the slower implementation and the increased cost in area.

- Loading the PC via the instruction fetch mechanism provides a small performance benefit based on these simulations. However, in a full system the extra cache bandwidth used may outweigh these benefits. The strongest argument for its use is that it will simplify the implementation by reducing the processor's bus complexity.

- SPSR locking, the simplest mechanism to maintain the SPSR values during exception processing, is adequate since the possible performance gain from any other mechanism is very small.

- The branch and load latency of the pipeline model given is too high, indicating that there are too many pipeline stages in the design and that, combining some of the stages (e.g. decode and register read) is likely to improve performance.

- Additional fast branch mechanisms (similar to those used in StrongARM) should be investigated as possible ways of reducing the branch cost.

Chapter 8: Conclusions and Future Work

This chapter summarises the work described in the thesis, draws conclusions and suggests directions for future work.

8.1 Summary

This thesis has described the design of exception and dependency handling mechanisms for a third generation asynchronous microprocessor which implements the ARM instruction set architecture.

Chapter 2 described the general problem of dependencies, the different forms of dependency and existing mechanisms for resolving dependencies. Traditional result forwarding mechanisms used in synchronous systems were shown to be unworkable in asynchronous systems due to the absence of a global timing reference. This influences the overall pipeline structure and makes the simple linear five stage pipelines used in the ARM8 and StrongARM (see Appendix B) unsuitable for use in an asynchronous implementation.

Chapter 3 described the problems of exceptions, the different types of exceptions and exception handling mechanisms which have been proposed. In particular it was shown that many forms of exception, such as interrupts, can be dealt with in the early stages of the pipeline in a similar way to normal instructions. However there are some forms of exception, particularly those generated by memory operations, which may be caused by a large proportion of instructions but which rarely occur.

Chapter 4 summarised the ARM ISA and described the unusual features of the ARM. The ARM's conditional execution facility proves to be a particularly difficult feature to implement efficiently. The presence of this feature makes the *future file*, one of the possible solutions to the exception problem, unworkable in the context of the implementation of an ARM.

Chapter 5 described the evolution of the asynchronous reorder buffer on which this thesis is based. A set of constraints was given which removed the need for synchronisation and enabled the creation of an efficient asynchronous implementation of a reorder buffer. In particular, allowing forwarding after the result has been written back to the register bank removes the need for a large amount of synchronisation between logically distant sections of the processor pipeline.

Chapter 6 described additional mechanisms that provide dependency and exception handling mechanisms for other parts of the processor state. In particular, a modification to AMULET1's colour mechanism is described which can be used to label instructions in all instruction streams currently in the processor.

Chapter 7 presented the results of a large number of simulations used to assess the effectiveness of the proposed mechanisms. These results show that the reorder buffer provides a performance increase of around 20% over the techniques used in AMULET1 and AMULET2. However, these results show that more work is needed to reduce the effect of procedural dependencies and load latency.

8.2 Conclusions

The work described in this thesis has led to a high performance data dependency and exception handling architecture for an asynchronous microprocessor. Existing techniques have been examined and it has been determined that the reorder buffer **[SmPl88]** is a suitable candidate for asynchronous implementation and is also suitable for use with the ARM's conditional instruction execution.

A number of problems have been identified which influence the implementation of an asynchronous reorder buffer. Solutions to these problems have been provided along with mechanisms for solving the other dependency and exception handling problems involved in implementing the ARM ISA. The resulting architecture has been modelled in VHDL and has been compared with models of previous architectures. The results have shown increased performance and flexibility when compared with earlier mechanisms. These results also show that the branch dependency mechanisms in the architecture need significant improvement.

At the time of writing the architecture is being refined and implemented by other members of the AMULET group and will form the basis of the AMULET3 microprocessor.

8.3 Advantages and disadvantages of the proposed architecture

The benefits of the dependency and exception handling mechanism described in this thesis are:

- **Increased performance**

  The reorder buffer provides increased performance compared with the mechanisms used in AMULET2.

- **Increased flexibility**

  The reorder buffer is significantly more flexible than the mechanism used in AMULET2. Unlike the mechanism used in AMULET2, the reorder buffer can be used with pipelined memory systems which generate exceptions and memory systems which take a long time to produce an exception response.

- **Unification of dependency and exception handling**

  In AMULET2 the inter-instruction dependency mechanisms were implemented using lock FIFOs while exceptions were processed using ad hoc logic. The reorder buffer with forwarding solves both problems in a single mechanism.

- **Precise exceptions and complete ARM compatibility**

  The mechanisms described in this thesis provide precise exception handling (as defined in section 3.4) unlike the mechanisms used in processors such as Fred. This is essential in providing complete compatibility with existing ARM code and operating systems. In addition, the architecture described is completely compatible with existing ARM programs and does not penalise the use of ARM's unusual features such as conditional instructions.

- **Generality**

  While the asynchronous reorder buffer was developed for use in an implementation of the ARM it is general enough to be used in other processor designs. However other techniques, which are unsuitable for use in an implementation of the ARM, such as the future file may also be applicable to other processor architectures.

The mechanisms described in this thesis do have a number of undesirable properties:

- **Lack of modularity**

  One of the potential benefits of asynchronous design suggested in chapter 1 is that it should be possible to make an asynchronous design more modular than its synchronous equivalent because the timing characteristics of each block are unrelated. Unfortunately this design is not as modular as might be hoped. For example, the implementation of the decode block is heavily dependent on the fact that a reorder buffer is being used; a change to a different mechanism would require the redesign of several parts of the processor.

- **The commit block is too large**

  The *commit block* (described in chapter 6) has become too large and complex and thus may be difficult to implement efficiently. Initially the purpose of this block was to combine stream changes from the execute stage and exception requests from the reorder buffer and generate packets representing the new streams to be sent to the fetch unit. It has, however, taken on the role of routing results between the execution unit, the reorder buffer and addresses to the memory interface, maintenance of the CPSR and SPSRs and maintenance of the colour. The block is perhaps the largest, most complex block in the design. The result is that it is difficult to implement efficiently and has been very difficult to debug. With care it should be possible to split the commit block resolving this problem.

- **Lack of proof**

  The architecture as presented is believed to be free of deadlocks and the simulations have executed large amounts of ARM code leading to high confidence that the design is correct. There is, however, no proof of its correctness. Formal proof of the architecture is a highly desirable goal which the author hopes others will undertake.

- **Complexity**

  The architecture described is probably more complex to implement than the equivalent mechanisms used in synchronous implementations of the ARM such as the StrongARM. In particular the use of a linear five stage pipeline with forwarding solves many of the problems described in this thesis in a simpler way, unfortunately unavailable to asynchronous designers.

  It may be more appropriate to apply the asynchronous reorder buffer to an implementation which already requires a powerful dependency handling mechanism. In particular a superscalar implementation of the ARM (or other microprocessor) would require a mechanism such as this and would be an interesting field of future research **[ChEd96]**.

The main direction of the research described in this thesis has been resolving data dependencies and ensuring data validity during exceptions; this research has resulted in an efficient mechanism for doing this. In retrospect more work is needed on efficient procedural dependency mechanisms which are still impacting processor performance.

8.4 Future work

A number of important issues have been left unresolved in the architecture described thus far. These issues (described below) are left for future research. In addition there are a number of potential sources of performance improvement which have not been investigated fully. These are described below to provide more opportunities for future work.

8.4.1 Coprocessors

The designers of the ARM2 made provision for extensions of the instruction set to be provided by a number of separate ‘coprocessors’. Sections of the ARM instruction set are presented to the coprocessors and if a coprocessor wishes to execute the instruction the ARM and the coprocessor cooperate during the execution of the instruction. If no coprocessor wishes to execute the instruction the ARM must enter the undefined instruction exception handler. Thus the first problem to be solved when designing a coprocessor interface is to provide a mechanism for passing instructions to the coprocessors and for the coprocessors to state whether or not they will execute the instruction.

There are four forms of coprocessor instruction:

- Coprocessor Register Transfer (CPR): A CPR instruction allows an ARM register to be transferred to the coprocessor or a coprocessor register value to be read and placed in an ARM register.

- Coprocessor Data Operation (CPD): A CPD instruction is used to ask the coprocessor to perform an operation within the coprocessor.

- Coprocessor Data Transfer (CDT): A CDT instruction is used to allow a coprocessor to store some data to memory or to load some data from memory. The ARM provides the addresses to the memory for each access; the coprocessor signals the ARM after each access to indicate whether any more are required.

- Undefined instructions: ARM undefined instructions are offered to the coprocessors for execution in a similar way to Coprocessor Data operations.

While the Coprocessor Data Operations and Undefined instructions are relatively simple to implement the CPR and CDT instructions are more complex because they require a larger degree of interaction between the main processor and the coprocessors. In particular the CDT accesses external memory and can cause data aborts and thus incur the complexity of handling an exception.

As with any other ARM instruction a coprocessor instruction may be prefetched in the shadow of a branch instruction and thus might not be executed after being fetched. If the instruction is passed to the coprocessors before it is known whether it passed its condition codes and whether its colour matched then a mechanism must be provided for informing the coprocessor whether the instruction should really be executed. In addition the coprocessor instruction could execute in the shadow of a data abort and thus there must be a way to ensure that any instruction in the data abort shadow does not change the coprocessor state permanently.

8.4.2 Thumb

ARM Ltd. defines an extension to the standard ARM instruction set known as *Thumb*. The Thumb instruction set is a 16 bit instruction set which provide a limited subset of the full ARM instruction set. Programs written in Thumb code are typically more compact than their ARM equivalents and are suitable for systems where memory is scarce or slow. The processor can switch between ARM and Thumb modes very quickly and programs can consist of mixtures of ARM and Thumb code using ARM code for system tasks or where the extra expressiveness of the full instruction set is required.

In the ARM 7, Thumb is implemented as a translation block placed between the instruction fetch mechanism and the decode unit. The block translates almost all Thumb instructions into standard ARM instructions. There are a few Thumb instructions which must be dealt with by extra instructions provided in the decoder.

Thumb has little direct effect on any part of the design detailed in this thesis. However, it would be interesting to perform simulations which compared the register usage of programs written in ARM code and programs written in Thumb code. Differences in register usage patterns, due to the design of the Thumb instruction set, may suggest different sizes of reorder buffer.

8.4.3 PC change prediction

Branch prediction mechanisms work well for predicting branches at the end of loops and calls to subroutines. Most cannot, however, provide any benefit for returns from subroutines or other branches which calculate their destination address. The instructions after these forms of branch are prefetched in error, wasting power and causing pipeline stalls which slow down the execution of the program.

The author believes that one possible solution to this problem is to enhance branch predictors so that they store the address of all instructions which cause a branch, not just static branch instructions. Although this would not allow the branch predictor to start prefetching from the destination of the branch instruction it would allow the prefetch to be halted and thus reduce the power wasted. The implementation of this technique would consist of a larger branch address CAM and added logic to route all instruction stream changing instructions to the predictor. As with many such techniques, research is needed to determine whether the added complexity produces a sufficient reduction in power usage or performance gained to justify the extra complexity and power used in the added logic.

8.5 The asynchronous future

The mechanisms described in this thesis provide the opportunity for future enhancements such as superscalar implementations and show that it is possible to adapt synchronous architectural techniques for use in asynchronous frameworks. It is hoped that this work will make a useful contribution to the current resurgence in asynchronous design.

References

[ApML95] S.S. Appleton, S.V. Morton, and M.J. Liebelt, “The Design of a Fast Asynchronous Microprocessor”, IEEE Technical Committee on Computer Architecture Newsletter, October 1995.

[ARM91] Advanced RISC Machines Ltd, “ARM600 Datasheet”, 1991.

[ARM95] Advanced RISC Machines Ltd, “ARM Software Development Toolkit Reference Manual”, 1995. ARM document number ARM DUI 0020.

[ARM96] Advanced RISC Machines Ltd, “ARM Architecture Reference Manual”, ARM Document Number: ARM DDI 0100B; ISBN 0 13 736299 4, Prentice Hall. 1996.

[ArRe94] D.K. Arvind and V.E.F. Rebello, “Instruction-Level Parallelism in Asynchronous Processor Architectures”, In M. Moonen and F. Catthoor, editors, Proceedings of the 3rd International Workshop on Algorithms and Parallel VLSI Architectures, pages 203-215, Leuven, Belgium, August 1994. Elsevier Science.

[Bard96] A.Bardsley, “An Asynchronous Logic Synthesiser”, 3rd year project report, University of Manchester, Department of Computer Science, 1996.

[ChEd96] V.A.Chouliaras and D.A.Edwards, “A Superscalar AMULET”, Proceedings of the First UK Asynchronous Forum, D.K.Arvind and S.Furber (Editors), The University of Edinburgh, pp. 19-25, 1996.

[DaGY93] Ilana David, Ran Ginosar, and Michael Yoeli, “Self-Timed architecture of a Reduced Instruction Set Computer”, Proceedings of the IFIP Working Conference on Asynchronous Design Methodologies, Manchester, England, 1993.

[ECFS95] C.J.Elston, D.B.Christianson, P.A.Findlay and G.B.Steven, “Hades - Towards the Design of an Asynchronous Superscalar Processor”, the proceedings of the Second Working Conference on Asynchronous Design Methodologies, May 30-31, 1995, South Bank University, London, pp. 200-209.

[Ende96] P.B.Endecott, “Superscalar instruction issue in an asynchronous microprocessor”, IEE Proceedings on Computers and Digital Techniques, Volume 143, Number 5, September 1996, pp. 266-272.

[FuDa96] S.B. Furber and P. Day, “Four-Phase Micropipeline Latch Control Circuits”, IEEE Transactions on VLSI Systems, vol. 4 no. 2, June 1996 pp. 247-253. ISSN 1063-8210

[FuLi96] S.B. Furber and J. Liu, “Dynamic Logic in Four-Phase Micropipelines”, Proceedings: Async’96, Aizu-Wakamatsu, Japan, March 18-21 1996.

[Furb96] S.B. Furber, “ARM System Architecture”, Addison Wesley Longman, 1996. ISBN 0-201-40352-8.

[Furb97] S.B.Furber, J.D.Garside, S.Temple, J.Liu, P.Day, N.C.Paver, “AMULET2e: An Asynchronous Embedded Controller”, Proceedings of the Third International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async97), Eindhoven,

[GaGi97] J.D.Garside, D.A.Gilbert, “AMULET 3 ideas document”, internal AMULET group document.

[Gars93] J.D.Garside, “A CMOS VLSI Implementation of an Asynchronous ALU”, Proceedings of the IFIP Working Conference on Asynchronous Design Methodologies, Manchester, England, 1993.

[HePa90] John L. Hennessy and David A. Patterson, “Computer Architecture A Quantitative Approach”, Morgan Kaufmann Publishers Inc., ISBN 1-55860-069-8, 1990.

[HwPa87] W.W.Hwu and Y.N.Patt, “Checkpoint Repair for Out-of-order Execution Machines”, Proceedings of the 14th Annual International Symposium on Computer Architecture (ISCA87) pp 18-26.

[Ibbe82] Roland N. Ibbett, “The Architecture of High Performance Computers”, Macmillan Publishers, ISBN 0 333 33231 8, 1982.

[IEEE85] “IEEE Standard for Binary Floating-Point Arithmetic”, ANSI/IEEE Std 754-1985.

[John91] Mike Johnson, “Superscalar Microprocessor Design”, Prentice Hall, Englewood Cliffs, ISBN 0-13-875634-1, 1991.

[Mart93] Alain J. Martin, Chapter 9 of “Synthesis of Asynchronous VLSI Circuits”, in the Proceedings of the VII Banff Workshop: Asynchronous Hardware Design, Banff, Canada, August 28-September 3, 1993.

[MoPW96] S.Moore, P.Robinson, S.Wilcox, “Rotary pipeline processors”, IEE Proceedings on Computers and Digital Techniques, Volume 143, Number 5, September 1996, pp. 259-265.

[Pave94] Nigel Charles Paver, “The Design and Implementation of an Asynchronous Microprocessor”, PhD thesis, University of Manchester, 1994.

[PDFG92] N.C.Paver, P.Day, S.B.Furber, J.D. Garside and J.V.Woods, “Register Locking in an Asynchronous Microprocessor”, Proceedings of ICCD 92 1992 pp. 351-355.

[Petl96] Oleg A. Petlin, “Design for Testability of Asynchronous VLSI Circuits”, PhD thesis, University of Manchester, 1996.

[RiBr95] William F. Richardson and Erik Brunvand. “Precise Exception Handling for a Self-Timed Processor”, in 1995 IEEE International Conference on Computer Design: VLSI in Computers & Processors, October 1995, pp. 32-37.

[RiBr96] W.F. Richardson and E. Brunvand, “Architectural considerations for a self-timed decoupled processor”, IEE Proceedings on Computers and Digital Techniques, Volume 143, Number 5, September 1996.

[Rich96] William F. Richardson, “Architectural Considerations in a Self-Timed Processor Design”, PhD dissertation, University of Utah, Department of Computer Science, March 1996.

[SmPl88] James E. Smith, and Andrew R. Pleszkun, “Implementing Precise Interrupts in Pipelined Processors”, IEEE Transactions on Computers, Vol. 37, No.5, May 1988, pp.562-573.

[SmSo95] James E. Smith, and Gurindar S. Sohi, “The Microarchitecture of Superscalar Processors”, Proceedings of the IEEE, 1995, Vol.83, No.12, pp. 1609-1624

[SmWe94] James E. Smith, and Shlomo Weiss, “PowerPC 601 and Alpha 21064: A Tale of Two RISCs’, IEEE Computer, Volume 27, Number 6, June 1994, pp. 46-58.

[SpSM94] Robert F. Sproull, Ivan E. Sutherland, Charles E. Molnar, “Counterflow Pipeline Processor Architecture”. Sun Microsystems Laboratories technical report SMLI TR-94-25.

[SoVa87] Gurindar S. Sohi, and Sriram Vajapeyam, “Instruction Issue Logic for High-Performance, Interruptable Pipelined Processors”, Proceedings of The 14th Annual International Symposium on Computer Architecture (ISCA), Pittsburgh, Pennsylvania, 1987, pp. 27-34.

[STB94] SPARC Technology Business, “UltraSPARC-I Data Sheet”, April 1994.

[Suth89] I.E. Sutherland, “Micropipelines”, The 1988 Turing Award Lecture, Communications of the ACM, Vol. 32, No 6, pp 720-738, January, 1989.

[ToDa93] H.C. Torng and Martin Day, “Interrupt Handling for Out-of-Order Execution Processors”, IEEE Transactions on Computers, Vol. 42, No.1, January 1993, pp. 122-127.

[Toma67] R.M. Tomasulo, “An Efficient Algorithm for Exploiting Multiple Arithmetic Units”, IBM Journal, vol. 11, January 1967, pp. 25-33, Reprinted in Daniel P.Siewiorek, C.Gordon Bell, Allen Newll, “Computer Structures:Principles and Examples”, 1982.

[WaDB94] Steven Wallace, Nirav Dagli, and Nader Bagherzadeh, “Design and Implementation of a 100 MHz Reorder Buffer”, 37th Midwest Symposium on Circuit and Systems, August 1994, pp. 42-45.

[Yant95] J.T.Yantchev et. al., “Low-Latency Asynchronous FIFO Buffers”, in the proceedings of the Second Working Conference on Asynchronous Design Methodologies, May 1995, pp. 24-31.

[York94] Richard York, “Branch Prediction Strategies for Low Power Microprocessor Design”, M.Sc Thesis, University of Manchester, 1994.

Appendix A:The ARM instruction set

This appendix provides a brief description of the ARM instruction set for those not familiar with it. For more detail see **[ARM96].** The structure of the ARM's register bank has already been described in chapter 4.

A.1 Conditional execution

All ARM instructions can be executed conditionally. The condition is based on a combination of the ALU result flags; for example the CS condition causes the instruction to execute only if the carry flag is set. In the assembler notation the condition code is appended to the end of the instruction, thus the instruction:

```
MOVCS R0,R1
```

moves R1 into R0 if and only if the carry flag is set. The condition codes are shown in Table A.1:

EQ	Z(ero) set
NE	Z(ero) clear
CS	C(arry) set
CC	C(arry) clear
MI	N(egative) set (negative)
PL	N(egative) clear (plus)
VS	(o)V(erflow) set
VC	(o)V(erflow) clear
HI	C set and Z clear (unsigned higher)
LS	C clear or Z set (unsigned lower or same)
GE	N=V (greater or equal)
LT	N!=V (less than)
GT	Z clear and N=V (greater than)
LE	Z set or N!=V (less than or equal)
AL	Always

Table A.1: ARM Condition codes

A.2 Normal data processing operations

The most frequently used ARM instructions are the *data processing instructions* which perform register-to-register arithmetic and logical operations. These instructions have the assembly code format:

```
OPP Rd, Rn, Op2
```

Where *Rd* represents the destination register, *Rn* is the *first operand* and *Op2* is the second operand. The available operations are:

AND	Rd:= Bitwise AND of Rn and Op2
BIC	Rd:= Bitwise AND of Rn and NOT(Op2)
EOR	Rd:= Bitwise exclusive-OR of Rn and Op2
ORR	Rd:= Bitwise OR of Rn and Op2
MOV	Rd:= Op2 (Rn is ignored)
MVN	Rd:= NOT Op2 (Rn is ignored)
SUB	Rd:= Rn - Op2
RSB	Rd:= Op2 - Rn
ADD	Rd:= Rn + Op2
ADC	Rd:= Rn + Op2 + C
SBC	Rd:= Rn - Op2 + C -1
RSC	Rd:= Op2 - Rn + C - 1

Table A.2: ARM data processing operations

The second operand can either be a constant value (built from a shifted 8 bit constant) or an optionally shifted register value, for example the following are possible instructions:

```
ADD R0,R1,#5          Add 5 to R1 and place the result in R0
ADD R0,R1,R2          Add R2 to R1 and place the result in R0
ADD R0,R1,R2,LSL #2
                      Add the value of R2 shifted two places
                      to the left to the value of R1 and place
                      the result in R0.
ADD R0,R1,R2,LSL R3
                      Shift the value in R2 left by R3 bits,
                      add the result to R1 and place the
                      result in R0.
```

The shifter mechanism allows left shifts, right logical and arithmetic shifts and bit rotation. This mechanism makes these instructions a lot more flexible than the equivalent instructions on other RISC microprocessors.

Each of these instructions can optionally set the CPSR flags to represent characteristics of the result produced, thus:

`ADDS R0,R1,#5`	Add 5 to R1, place the result in R0 and set the CPSR flags as appropriate.

The flags are:

- V oVerflow
- C Carry
- N Negative
- Z Zero

In addition the data processing set of instructions include a number of comparison operations which are identical to other data operations except that they affect only the ALU flags and do not write a result into Rd. These instructions are:

TST	Equivalent to AND
TEQ	Equivalent to EOR
CMP	Equivalent to SUB
CMN	Equivalent to ADD

Table A.3: ARM Comparison operations

A.2.1 Data processing with PC write

Any of the data processing instructions which produce a result can write the result into the program counter. Examples of this are:

`MOV PC,R14`	Returning from a leaf subroutine
`ADD PC,PC,R0,LSL#2`	Jumping via a branch table

The case where the destination is the program counter and the S flag is set is treated specially. In this case the SPSR is copied into the CPSR after performing the data operation for example:

```
MOVS PC,R14
```
Return from exception

A.3 Branch

The ARM's branch instruction is written in the form

```
B label
```

and contains a 24 bit signed word offset and so can branch forward or backwards by 32 Mbytes relative to the current program counter.

A.3.1 Branch with link

For calling subroutines the ARM provides the *Branch and link* instruction which is written in the form:

```
BL subroutine
```

This stores the address of the next instruction (PC+4) into the current R14 register (also known as the *link register or LR*) and then performs the branch. *MOV PC,LR* is used to return from a subroutine called in this way.

A.4 Single value memory transfer

The LDR and STR instructions load and store single words or unsigned bytes to and from memory. The instructions also allow the base register to be updated. Examples of these instruction include:

`LDR Rd,[Rn,+/-off]`	Load the word at Rn+/-off into Rd.
`LDR Rd,[Rn,+/-off]!`	Load the word at Rn+/-off into Rd and then update Rn to be the address from which the word was loaded.
`LDR Rd,[Rn],+/-off`	Load the word at Rn into Rd and then update Rn to be Rn+/-off
`LDRB Rd,[Rn,+/-off]`	Load the byte at Rn+/-off into the low byte of Rd and clear bits 8 to 31.
`STR Rd,[Rn,+/-off]`	Store the word in Rd at the address Rn+/-off.
`STRB Rd,[Rn,off]`	Store the byte in the low 8 bits of Rd into the address Rd+/-Off

The offset can be either added to or subtracted from the base register (Rn) and is either an unsigned 12 bit value or a shifted register value similar to that used in the second operand of the data operations.

A.4.1 Halfword and signed byte accesses

In ARM Architecture version 4 instructions to perform signed byte loads and halfword (16 bit) transfers was added. These include the following operations:

`LDRSB Rd,[Rn,off]`	Load the byte from the address Rn+off, sign extend it and place the result in Rd.
`LDRH Rd,[Rn,off]`	Load the halfword from the address Rn+off, zero extend it and place the result in Rd.
`LDRSH Rd,[Rn,off]`	Load the halfword from the address Rn+off, sign extend it and place the result in Rd.
`STRH Rd,[Rn,off]`	Store the halfword in the bottom 16 bits of Rd at the address Rn+off

Pre- and post- indexed addressing is available using an 8-bit constant offset or an unshifted register value.

A.4.2 Swap

The swap instruction causes a pair of atomic memory operations. It is provided as a mechanism for implementing software semaphores. It has the form:

`SWP Rd,Rm,[Rn]`	Load Rd with the word from the address in Rn and then store Rm at the same address
`SWPB Rd,Rm,[Rn]`	Load Rd with the byte at the address Rn and then store the bottom 8 bits of Rm at the same address.

A.5 Multiple value memory transfer

The *LDM* and *STM* instructions allow a subset of the registers to be loaded from or stored to memory in a single instruction. The most common use of the LDM and STM instructions is in subroutine entry and exit:

```
STMFD R13!,{R0-R8,R14}
      .
body of subroutine
      .
LDMFD R13!,{R0-R8,PC}
```

In this example the STM is storing R0-R8 and R14 (the link register) onto a stack using R13 as the stack pointer which is updated to account for the entries added to the stack (this is signified by the ! which causes base write back). The corresponding LDM instruction causes these values to be loaded back from the stack, however this time instead of loading the old R14 value into R14 it is written into the program counter, causing a return from subroutine. Any combination of the 15 registers and the program counter can be loaded or stored by a single LDM or STM instruction.

The STM and LDM instructions can use pre or post indexed addressing with an offset corresponding to the number of bytes loaded or stored added or subtracted from the base register. Thus *LDMIA* corresponds to *I*ncrement *A*fterwards (i.e. post-increment) addressing. In the example above the *FD* option is used which indicates post-increment for load and pre-decrement for stores which is suitable for implementing a Full Descending stack.

The LDM and STM instructions also have a number of system management capabilities, in particular they can be used to access registers from outside the current mode and the LDM instruction can cause the SPSR to be copied back to the CPSR.

A.6 Access to the CPSR/SPSR

The MSR and MRS instructions provide explicit access to the CPSR and SPSR and are used in operating system code. For example:

`MRS Rd, CPSR`	Copy the CPSR to Rd.
`MSR CPSR, R0`	Write the value of R0 into the CPSR.
`MSR SPSR, R0`	Write the value of R0 into the current SPSR.
`MSR CPSR_flg, R0`	Write only the ALU flag field of the CPSR from the corresponding bits in R0.
`MSR SPSR_flg, R0`	Write only the ALU flag field of the current SPSR from the corresponding bits in R0.

The value to be written into the SPSR or CPSR is taken from a restricted form of the data operation second operand.

A.7 Multiplication

In ARM architecture version 4 there are two forms of multiplication. The first is implemented by the *MUL* and *MLA* instructions which multiply two 32 bit values together and provide a truncated 32 bit result; MLA also adds a third register value onto the result:

`MUL Rd,Rm,Rs`	Rd:=Rm*Rs
`MLA Rd,Rm,Rs,Rn`	Rd:=Rm*Rs+Rn

As with the data operations the S flag may be given to force these instructions to update the CPSR flags.

The second form of multiply instruction is the long multiply. These multiply together two registers and produce a 64 bit result, which is placed in two registers. A 64 bit multiply and accumulate is also provided which multiplies two register values and then adds this result to the contents of the two result registers. Separate signed and unsigned variants of both the normal and accumulating long multiply instructions are provided. Thus:

`UMULL RdLo,RdHi,Rm,Rs`	RdHi,RdLo:=Rm*Rs (unsigned)
`UMLAL RdLo,RdHi,Rm,Rs`	RdHi,RdLo:=Rm*Rs+RdHi,RdLo
`SMULL RdLo,RdHi,Rm,Rd`	RdHi,RdLo:=Rm*Rs (signed)
`SMLAL RdLo,RdHi,Rm,Rd`	RdHi,RdLo:=Rm*Rs+RdHi,RdLo

These instructions can have an S flag appended to set the CPSR ALU flags.

A.8 Coprocessor instructions

The ARM architecture defines three forms of coprocessor instructions, coprocessor data operations, coprocessor data transfers and coprocessor register transfers. Each instruction specifies which of the possible 16 coprocessors the instruction should be processed by. Alternative assembler mnemonics are commonly used for operations specific to particular coprocessors, for example there is a set of floating point mnemonics which map onto the mnemonics given below.

A.8.1 Coprocessor data operations

These instructions are designed to be executed completely within the coprocessor, for example a register to register operation in a floating point accelerator. For example:

`CDP pn,op,cd,cn,cm,x`	On coprocessor p*n* execute operation *op* on coprocessor registers *Cn* and *Cm* and place the result in coprocessor register *Cd* passing the extra constant *x*.

Other than the sections of the instruction required to identify it as a coprocessor instruction and the condition code, the ARM ignores all other bits in the instruction; this makes it possible for a coprocessor to interpret these fields in a completely different manner.

A.8.2 Coprocessor data transfers

The Coprocessor data transfers are used to load or store some of the coprocessors registers from or to memory. The ARM performs all addressing but the coprocessor supplies or consumes data while informing the ARM of the number of words to transfer. For example:

`LDC pn, cd, [Rn,#offset]`	Load coprocessor register cd in coprocessor pn from the address Rn+offset

As with the LDR and STR instructions the offset can be pre- or post- indexed with write back optional in the case of the pre-indexed mode. The coprocessor number, coprocessor register and an optional flag bit (not shown above) are not examined by the ARM and so may be interpreted differently by the coprocessor.

A.8.3 Coprocessor register transfers

The coprocessor register transfer instructions are used to transfer data between the ARM registers and the coprocessor. An important use of these instructions is to communicate with system control coprocessors which are used to control caches and memory management units on some versions of the ARM processors. Fields are provided to pass extra information to the coprocessor to inform it to perform extra operations on the data before transfer. An example of this instruction is:

`MRC pn,op,Rd,cn,cm,x`	Coprocessor pn will perform operation *op* (sub operation *x*) on registers cn and cm and transfer the result back to the ARM register Rd

This instruction allows Rd to be set to 15 which instead of writing the result to the program counter causes the result to be transferred to the CPSR flags register; this can be used, for example, to transfer information about the result of a floating point coprocessor operation to the ARM condition flags.

A.9 Software interrupts

The software interrupt (SWI) instruction on the ARM is used to allow unprivileged programs to call sections of the operating system which must execute in a privileged mode. The instruction is written:

```
SWI constant
```

The constant is a 24 bit constant which is ignored by the processor and is typically used by the operating system to determine the operation required.

The effect of the instruction is to preserve the current program counter in R14_svc, the current CPSR in SPSR_svc and then enter supervisor mode at the SWI exception vector at address 8.

A.10 Undefined instructions

In ARM terminology the *undefined instructions* refer to an area of the instruction encoding that is defined to cause an undefined instruction exception; other instructions which have not been defined may or may not cause such an exception. The exception causes the current program counter to be copied to R14_undef, the current CPSR to SPSR_undef and then the processor to jump to the undefined instruction vector at address 4. Coprocessor instructions that no coprocessor is willing to execute cause the same behaviour.

A.11 Summary

The ARM instruction has a small number of instructions, however some of the instructions (such as the data operations and LDM/STM instructions) are very powerful. This power comes at the expense of complexity with instructions such as LDM having many flags which change their behaviour in a complex manner.

Some instructions have special responsibilities in system management (such as MOVS PC,...). While these instructions perform a very specialised task they are encoded as part of normal, commonly used instructions. This can complicate the design of instruction decoders.

Appendix B:The structure of existing ARMs and AMULETs

This appendix provides an overview of the architecture of previous implementations of the ARM processor. Further details can be found in **[Furb96]**.

B.1 ARM 2/3/6/7

The ARM 2, ARM 3, ARM 6 and ARM 7 share a near identical organization shown in figure B.1. The processor is organised into three pipeline stages, fetch, decode and execute. Execute includes register read, processing and write back. Since register read and write back are performed within the same stage no complex dependency mechanisms are required. The register bank has two read ports; this means that multiple cycles are needed for some store and data operations thus complicating the decode stage. The program counter holds the address of the instruction being fetched and is normally two instructions ahead of the instruction being executed. A branch causes the execute unit to discard two instructions; these are the instructions which have been incorrectly prefetched in the shadow of the branch.

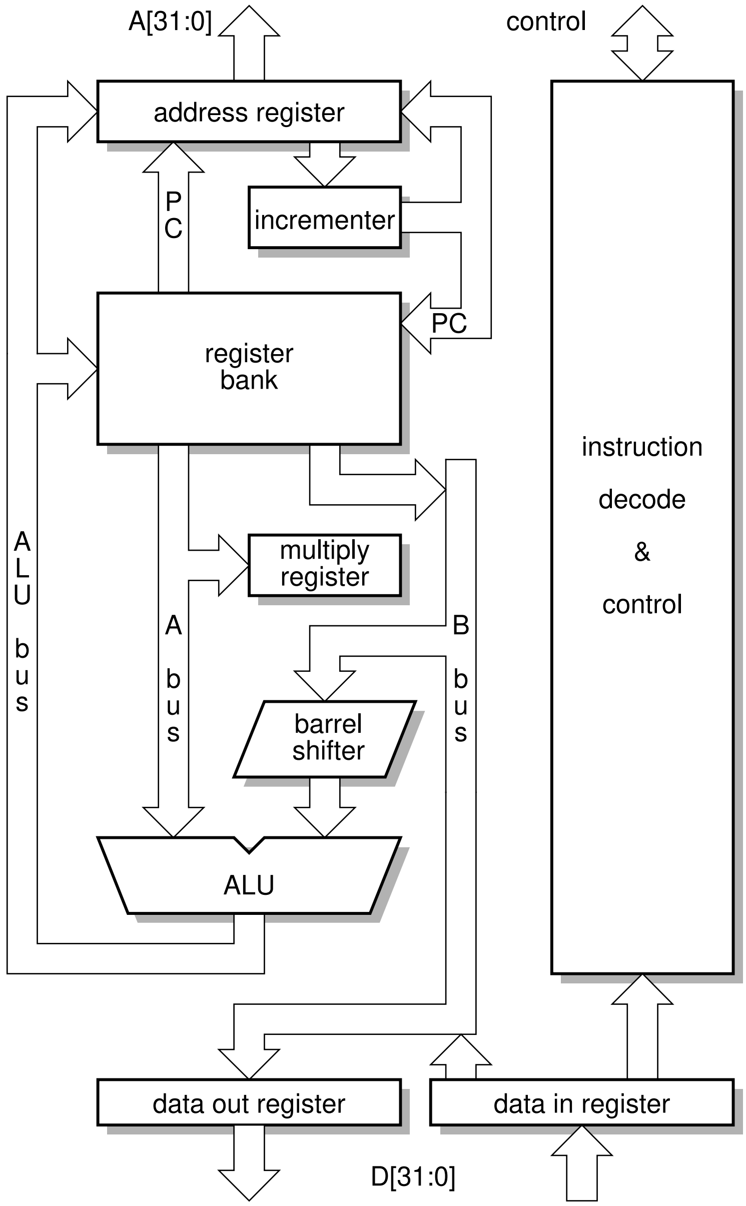


Figure B.1 ARM 2/3/6/7 organization

B.2 ARM 8

The ARM 8 provides higher performance than earlier ARMs through a number of separate techniques:

- Double-bandwidth memory

  The ARM8 transfers two words on each clock cycle during the sequential memory operations which make up the bulk of instruction fetches and the data transfers from ARM 'load multiple' instructions.

- Branch prediction

  The ARM 8 incorporates static branch prediction in which all backwards branches are predicted as taken, this proves to be a simple model for predicting the effect of branches in loops. This is implemented in a separate prefetch unit which provides a stream of instructions and PC addresses to the integer execution unit (shown in figure B.2).

- Smaller pipeline stages

  The ARM8 has a five stage pipeline consisting of prefetch, decode and register read, execute, data memory access and result write back which enables each stage to be simpler and thus faster than earlier ARM pipeline stages.

No documentation is available on the way in which the result forwarding paths (shown in the diagram) are used or on how the ARM 8 processes exceptions.

B.3 StrongARM

The StrongARM is an implementation of the ARM by Digital Equipment Corporation; a diagram of its pipeline organisation is given in figure B.3. It uses:

- A modified-Harvard architecture

  This has a separate data and instruction cache which provides a higher memory bandwidth to the core.

- A 5 stage pipeline

  Like the ARM8 the StrongARM uses a 5 stage pipeline consisting of fetch (from the instruction cache), decode and register read, execute, data cache access, and finally result write back. Forwarding paths are provided to for-

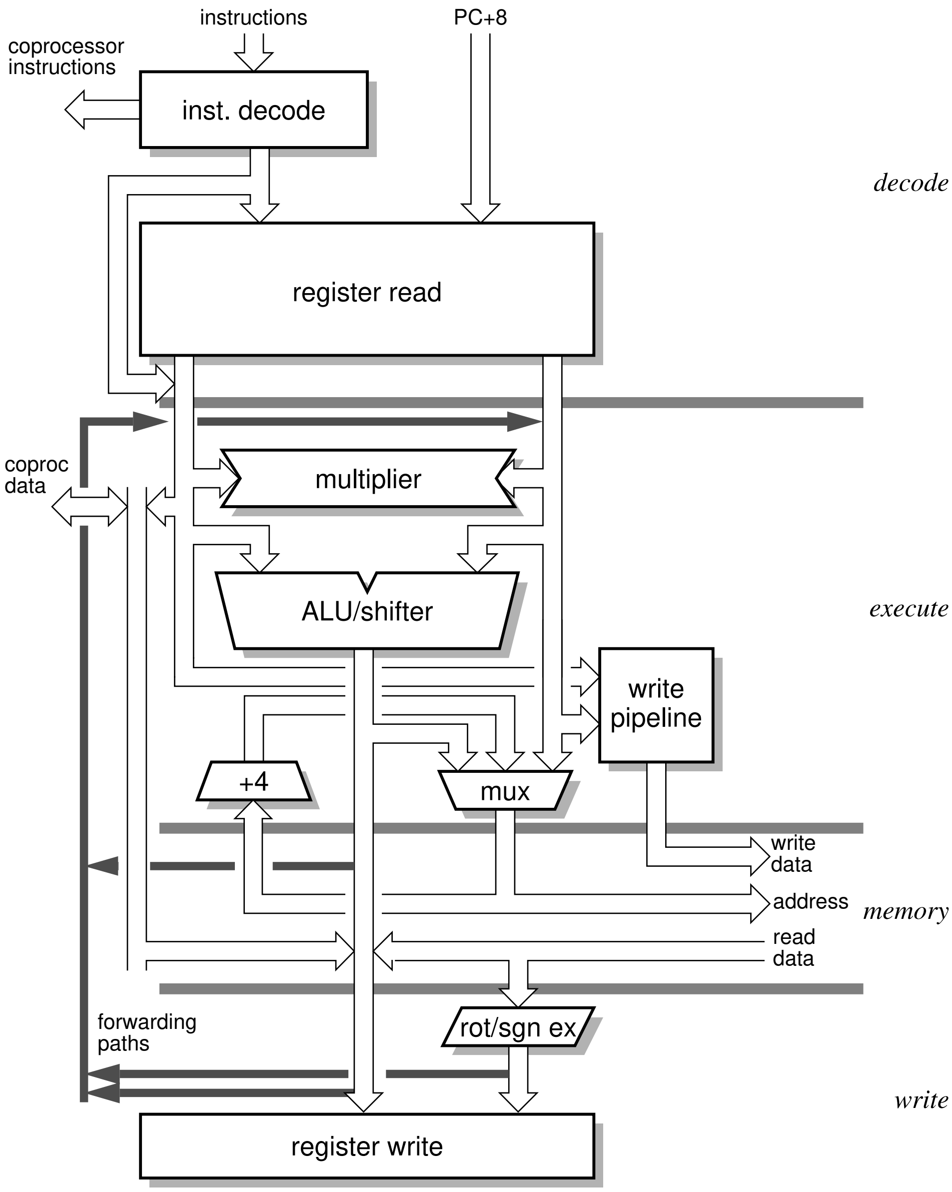


Figure B.2 ARM8 Integer Unit organisation

ward from later stages in the pipeline thus avoiding the delay before register write back.

- Branch adder

  The ARM8's method of reducing the branch penalty is to reduce the number of branches seen by the core via branch prediction. In contrast the StrongARM does not use branch prediction but instead attempts to reduce the cost of each branch by providing a separate branch adder and PC forwarding paths which enable branch target addresses to be calculated in the instruction decode stage of the pipeline and be quickly routed back to the fetch unit.

- 3 read port register bank

  In earlier ARM's added complexity in the decode stage was caused by some instructions requiring three operands while the register bank could only provide two per cycle. While these instructions are rare and thus reducing their execution time produces a small increase in overall performance the added complexity in the decode stage is significant. By adding a third read port this problem is removed at the cost of a larger register bank.

Both the ARM 8 and StrongARM use a linear pipeline with the data memory read as a stage in the main pipeline and thus no register is written back in the shadow of a memory operation which may cause an exception.

B.4 The AMULET1 and AMULET2

The AMULET1 and AMULET2 have a similar internal architecture shown in figure B.4. The *address interface* is responsible for generating instruction addresses and also routing data addresses to memory. The instructions return into a pipeline used as an instruction buffer and then into decode. Operands are read from the register file (after dependency information has been resolved using a lock FIFO - not shown on the diagram). In common with the earlier ARM's the AMULET has two read ports on its register bank and thus has to perform multiple register bank reads for the more complex instructions. In contrast to the other ARM implementations the AMULET has a memory pipeline separate from the execution pipeline. The justification for this is that it is difficult to implement traditional result forwarding paths and the increased pipeline length would have a detrimental effect on performance.

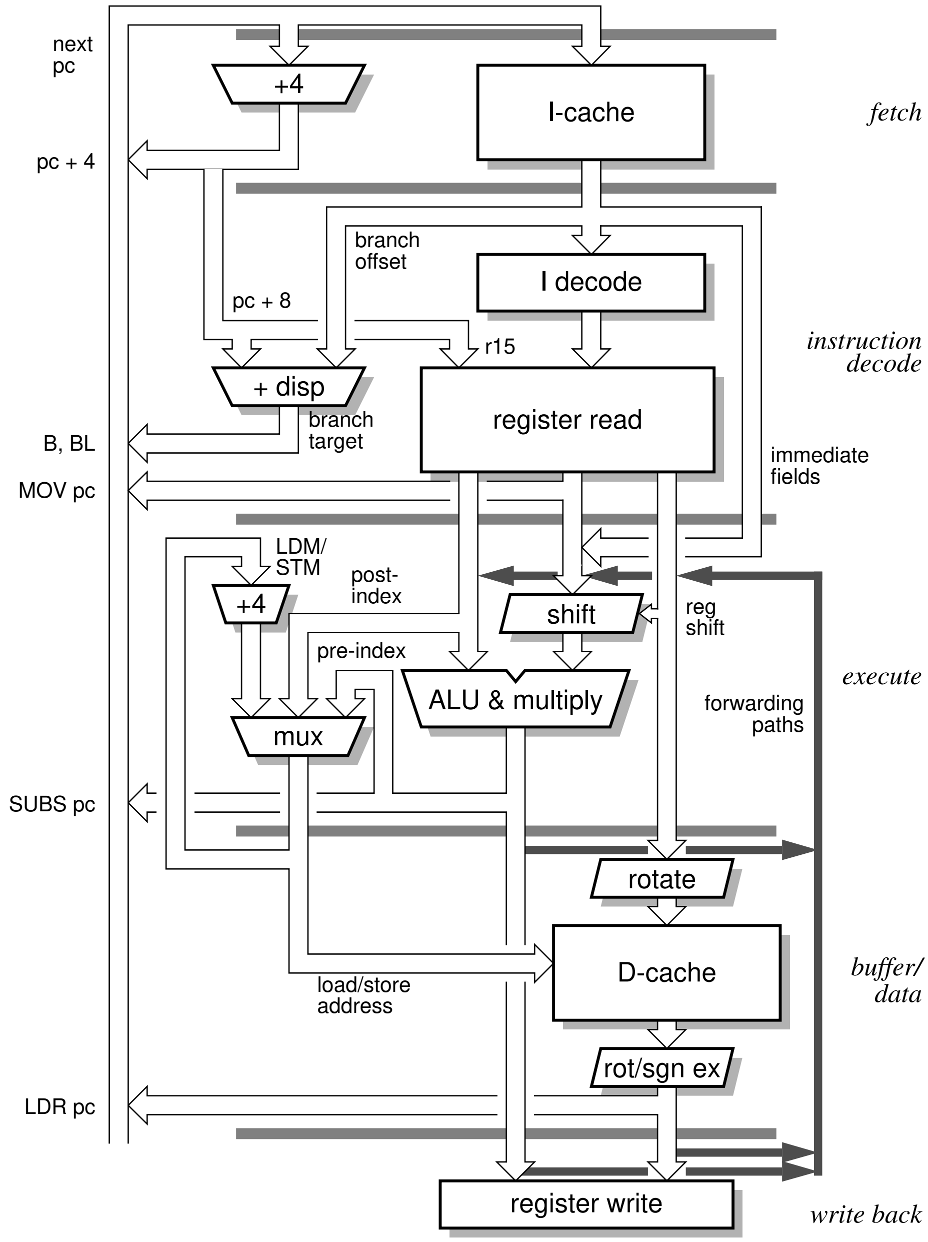


Figure B.3 StrongARM pipeline core organization

The AMULET2 includes a *last result reuse* mechanism within the execution pipeline to reduce the performance loss due to inter-instruction dependencies and a branch target cache to reduce the performance loss due to branches. The main performance limitation of the AMULET2 is believed to be the complexity of the address interface.

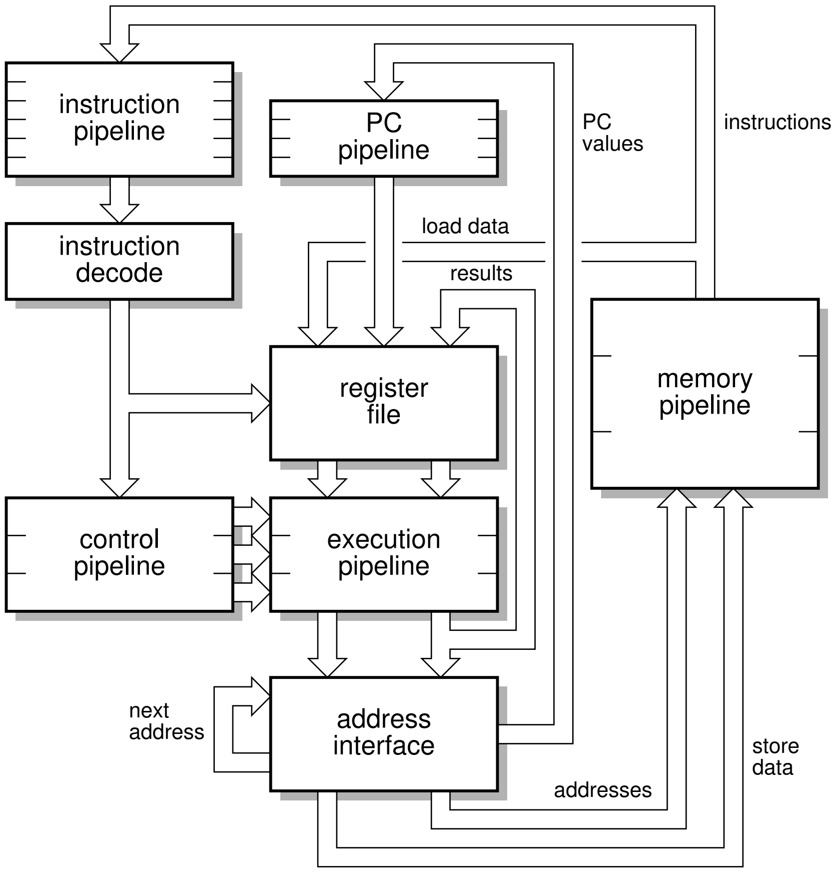


Figure B.4 AMULET Internal organisation

B.5 Summary

The various implementations of the ARM instruction set architecture described above show a wide range of architectural techniques. The earliest ARMs rely on a short, simple pipeline to remove the need for expensive forwarding mechanisms and dependency systems. The StrongARM and ARM8 have opted for longer pipelines to enable shorter pipeline stages at the cost of needing forwarding mechanisms and other added complexities.

The AMULET1 and AMULET2 have relatively long pipelines compared to the synchronous ARMs but do not have sophisticated forwarding mechanisms. This leads to a loss of performance that is improved by the architecture described in this thesis.