AN ANALYSIS

OF

ASYNCHRONOUS PROCESSOR

PIPELINES

A thesis submitted to the University of Manchester

for the degree of Master of Science

in the Faculty of Science and Engineering

September 1998

By

Daranee Hormdee

Department of Computer Science

Contents

List of Tables

List of Figures

Abstract

This thesis reports on a comparative study of asynchronous processor pipelines. Three-stage and five-stage pipelined asynchronous implementations of a simple RISC-like architecture, the DLX, proposed by Hennessy and Patterson, were modelled and evaluated using LARD (Language for Asynchronous Research and Design). Mechanisms for solving pipeline hazards - structural, data and control hazards - that occur in asynchronous pipelined processors owing to data dependencies or resource conflict were investigated. The mechanisms evaluated were stall, register locking, data forwarding and colour mapping. Simulation was employed to investigate the relative merits of each of these approaches and to evaluate the benefits of adapting a five-stage pipeline to asynchronous implementation for a future AMULET design.

Declaration

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

Copyright

Dedication

To my parents, my sisters - Ning and Noot - and my grandma.

Acknowledgements

This thesis could not have come about without the help and encouragement from some very special people.

First of all, I would like to thank my supervisor, Prof Steve B. Furber, for all his advice and support since I started this project.

My most heartfelt thanks go to the following people in the AMULET research group which I have worked with:

To Doug Edwards for his reading and commenting on the draft of this thesis. To "Mr. LARD", Phil Endecott, who has invented a very useful programming language for asynchronous design, for his inspiring explanation and help in dealing with LARD. To John Bainbridge, Andrew Bardsley and Peter Riocruex for their advice. Also the discussions with them on many topics have widely broadened my mind. And also the rest of the group.

I am also indebted to Chatchai Jantaraprim for his constant assistance with my questions and difficulties in many miscellaneous things which were inconvenient to discuss in non-mother tongue. His devotional friendship is also much appreciated.

I would like to thank Damien Desmicht, my Linux expert. His effort at the installation of Linux on my computer has tremendously facilitated my work from home.

My special thanks go to Renzo Stheins, who has helped me to proof read this thesis. My improvement of the English language is partly due to the conversations I had with him.

My thanks also go to a number of friends in the Chemistry Society at MMU, loads of French classmates, a few Thai friends and especially my dear friend, Luisa Quinti, for their company and acquaintance during the year far away from my home town have brought me cheerfulness and enjoyment dominating my homesickness.

I would like to take this opportunity to give special thanks to my beloved parents who have brought me up the way I am and also my lovely sisters.

I would like to acknowledge to the Royal Thai Government and KhonKaen University for the grant which enabled me to do this MSc.

And last, but by no means least, I acknowledge with great pleasure the help received from many other nice people that have directly or indirectly helped me in any way at all this year.

Preface

The author was born and brought up in KhonKaen, Thailand. She graduated from KhonKaen University, Thailand, in April 1996, obtaining a bachelor's degree (BEng) in Computer Engineering. From 1996 to 1997 she worked for the university as a lecturer. In 1997 she was awarded a grant from the Royal Thai Government.

An MSc in Advanced Computer Science was started at the University of Manchester in September 1997. This thesis reports on the work undertaken during a six month project, forming part of that MSc course.

Chapter 1

Introduction

1.1 Background, Motivation and Objectives

The majority of current digital design is based on a synchronous approach, where a central clock signal controls the operation of the system. Since synchronous design seems to be facing increasing difficulties (as will be discussed in section 2.2.1), it is natural to look at other techniques for VLSI [1] design; the use of asynchronous logic is established here.

The project presented in this thesis took place over a period of 6 months (April-September 1998) and was part of the ongoing research into asynchronous design being undertaken by the AMULET group at the University of Manchester.

[1] Very Large Scale Integration

The aim of this project is to investigate the relative merits of three-stage and five-stage asynchronous pipelines and to evaluate the benefits of adapting a five-stage pipeline to asynchronous implementation for a future AMULET design.

1.2 AMULET Projects

The AMULET (Asynchronous Microprocessor Using Low Energy Technology) group, a part of the Computer Science Department at the University of Manchester, under the direction of Professor Steve B. Furber, was established late in 1990 in order to investigate the claimed advantages and the feasibility of designing large asynchronous systems. The aim of this group is to realize asynchronous microprocessors with lower power consumption than are currently available using synchronous design techniques.

In 1994 the group delivered the *AMULET1* microprocessor which is the world's first implementation of a commercial microprocessor architecture (ARM) using Sutherland's micropipeline design style [Sut89]. The instruction throughput was increased whilst the electrical power consumption was reduced in the *AMULET2* processor, a major redesign of *AMULET1*. In 1996 *AMULET2e* (an embedded system chip with an *AMULET2* core and a self-timed cache), the second generation asynchronous ARM processor, was delivered.

The first two generations of processors were fundamentally designed as low power

processors and performance was a secondary consideration. *AMULET3* (which is now under development, the third generation asynchronous ARM processor) will try to maintain the high power efficiency of the earlier devices while significantly improving the MIPS [2] rate.

Apart from the AMULET microprocessors, there are other projects in the group concerned with both the reduction of electrical power consumption, and the synthesis ability of asynchronous circuits to allow the simple and quick production of devices for consumer applications.

1.3 Organisation of the Thesis

This thesis comprises 7 chapters and includes 3 appendices. Chapters 2, 3 and 4 provide a theoretical background to the area related to the subject of the thesis. Then Chapters 5 and 6 describe the author's work and evaluation. All the results presented in Chapter 6 were generated by the author. Chapter 7 summarises the conclusions drawn from the project presented in the thesis. Further work and possibilities are also introduced. Appendix A describes the DLX architecture which was implemented in this project. The main test program for this work, the Dhrystone benchmark program, is described in Appendix B. Finally details of the tools which were used in this project are given in Appendix C.

[2] Million Instructions Per Second

1.4 DLX as an Instructional Tool

The DLX architecture is a simplified version of the MIPS R3000 architecture [Kan89, GM93] introduced in the textbook *Computer Architecture : A Quantitative Approach* [HP96]. It was chosen as the architecture for this project because its simplicity makes it easy to work with to demonstrate the principles of pipelining. This textbook is accompanied by a comprehensive software analysis set, including a compiler, a simulator, and a group of benchmark programs. As DLX provides a great package for architectural experiments, those same tools, discussed in the textbook, have been applied in this project to analyse different kinds of asynchronous pipeline models based on the DLX pipeline.

The first model, using a five-stage pipeline, was similar to the five-stage pipeline DLX exploys. For comparison, another model was developed with only three pipe stages.

In addition, to simplify the models the DLX floating-point instructions were not included; adding them does not involve new concepts, it merely increases the complexity. Appendix A gives full details of the DLX architecture as implemented in this project.

1.5 The LARD Programming Language

LARD [3], developed at the AMULET group, is an asynchronous hardware description language which uses CSP-like [4] channel communication [Hoa78] to describe the behaviour of asynchronous VLSI systems. This communication abstraction makes LARD a much more productive language for this type of modelling than conventional languages such as VHDL.

[3] The Language for Asynchronous Research and Development. More details are presented in Appendix C

[4] Communicating Sequential Processes

Chapter 2

Processor Pipelining

A major concept that has received considerable attention in the design of high-speed computers is pipelining. The purpose of this chapter is to provide a brief overview of processor pipelining. Section 2.1 describes the basic tasks of a processor, then an overview of synchronous and asynchronous design is presented. Section 2.3 describes a method for improving processor speeds, i.e. pipelining. Finally, the differences between synchronous and asynchronous pipelines are discussed emphasising the different control principles involved in their construction.

2.1 An Overview of Processor Design

The general task of a processor is to execute a series of instructions. The three main processor functions are arithmetic operations, logical operations, and data accessing.

In order to execute each instruction, the processor must first fetch it from memory, then it must decode it and obtain the necessary operands. Finally, it must execute that instruction and write any results to their appropriate destination.

The basic elements of a processor are an instruction register (IR), data registers, an arithmetic and logic unit (ALU) and the system control logic.

The processor controls the transfer of data between the basic elements, executes the commands that modify the data or control program execution, maintains system status, and controls the sequence and timing of instruction execution.

A diagram which illustrates an example of the processor datapath for the DLX architecture is shown in Figure 2.1. Instructions and data generally move from left to right through this datapath as they complete execution. However, there are three exceptions to this left-to-right flow of data and instructions:

1. The write-back scheme, which places the result back into the data register file.

2. Writing a branch target address to the program counter (PC) instead of incrementing the PC by 4.

3. Incrementing the PC

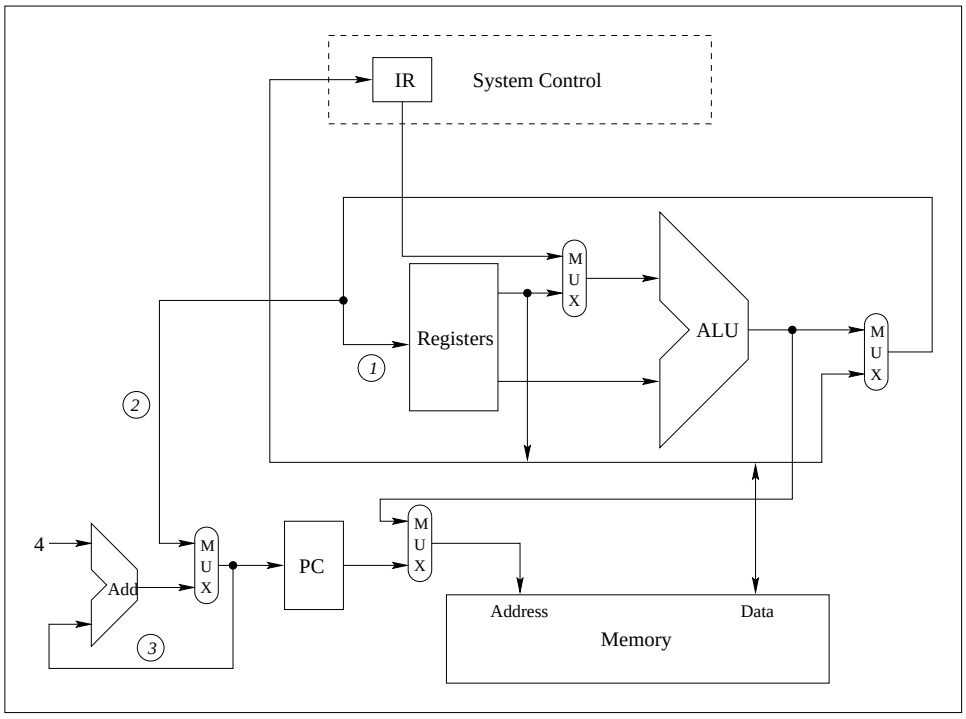

Figure 2.1: An example of a processor datapath

The movement of data from right to left does not affect the execution of the instruction which produces the right-to-left value , only following instructions in the pipeline are effected by these data flows. The details of this simple datapath can vary depending on the instruction set architecture [Dev94, PH97].

2.2 Synchronous and Asynchronous Design

This section discusses the comparative advantages and disadvantages of both synchronous and asynchronous design. At the end of this section the mechanisms used in asynchronous systems for transferring data between logic blocks will be discussed.

2.2.1 An overview of synchronous design

Synchronous design styles rely on distributed clock edges reaching all of the design concurrently. Each time the clock issues an active edge, a data transfer occurs.

The clocked-logic concept is widely used because [Sut89]:

- It offers a simple way to design.
- It is widely taught and understood.
- Parts that operate with a clock are widely available.
- System noise has gone away by the time a clock event occurs.

Despite these advantages there are a number of disadvantages to synchronous design which could potentially be resolved by using a different approach:

- The ideal concept for synchronous design is to synchronise all parts to work simultaneously. However, in practice propagation delays make this concept impossible. There is a difference in time between the clock reaching various parts of a design, called *clock skew*. This skew restricts the operating speed of a synchronous system.
- Synchronous circuits must wait until all possible computations have finished before latching the results. This yields worst-case performance.
- In the synchronous world, all sections of the design are clocked simultaneously, including those which are not active in every cycle. This causes many

nodes to change voltage level thereby toggling them unnecessarily.

- In a pipelined microprocessor, each stage of the pipeline may take a different time to finish its work. The clock period of the synchronous pipeline is limited by the slowest pipeline stage to finish. Synchronous pipelining will be discussed in more detail later on in this chapter.

2.2.2 An overview of asynchronous design

Asynchronous systems perform their calculations without being clocked globally. There is no clock in this style so another mechanism is needed to indicate when data is available for the next block to process. These systems act independently whenever local events permit [Hau93].

There are some benefits claimed for asynchronous design:

- Since by definition asynchronous systems have no globally distributed clock, clock skew is eliminated.

- Unlike synchronous systems, which yield worst-case performance as noted above, asynchronous systems exhibit average-case performance because they can sense when a computation has completed.

- In an asynchronous system, transitions occur only when a block is being used, thereby reducing the power consumption.

- In asynchronous systems, every pipeline stage is independent and can take a variable time to complete.

Whilst asynchronous design has a few possible advantages, it also has a number of disadvantages which can make it harder to design, such as [Gil97]:

- *Control logic complexity:* The control logic is more complex since each block of the design needs hardware to perform synchronisation to wait for data and to trigger other blocks when it has produced its data.

- *The risk of deadlock:* The use of explicit communications between blocks increases the risk of deadlock which can introduce design errors in asynchronous systems.

- *The loss of implied knowledge:* The clock system in a synchronous design can be thought as both local and global synchronisation. The use of a local synchronisation mechanism, i.e. between adjacent pipe stages in a pipeline, is quite simply replaced by the handshaking in an asynchronous design (which will be described in the following section). However, global synchronisation is more complicated to replace. Figure 2.2 shows a model of a simple pipeline with result forwarding (which will be discussed later in Chapter 4). In a synchronous system, the positions of an instruction and its result are deterministic, so that in a particular system an instruction issued in the previous clock cycle will be at the output of the next block in the next clock cycle. Thereby if block B wishes to use the result of

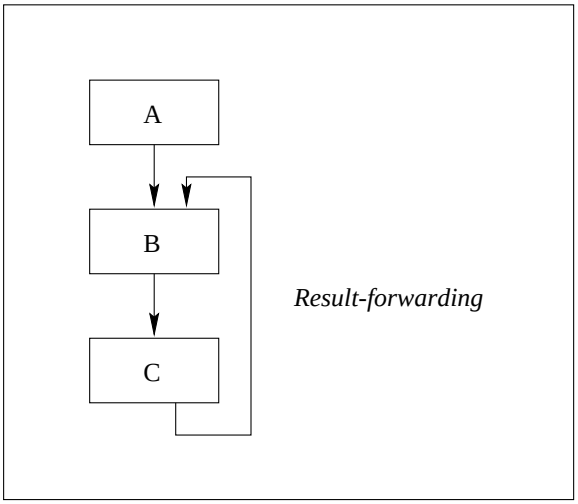

Figure 2.2: A simple pipeline

the preceding instruction (from block *C*) it must wait for only one clock cycle. In an asynchronous system, once a series of instructions is put into the beginning of a section of pipeline there is no way of knowing where each instruction will be at any time later. To be able to reuse a result later in the pipeline explicit synchronisation is needed. However, always being synchronised in an asynchronous system causes lockstep operation which leads to the pipeline operating at the speed of the slowest block. To synchronise only when the result is needed is a better solution.

2.2.3 Asynchronous handshaking

This section explains two handshaking mechanisms [Fur96] which have been used in asynchronous designs. These are the *two-phase* and *four-phase* handshake. When a sender and a receiver communicate, no matter which handshake is being used to control the flow of data, there will be two control wires, *request* and

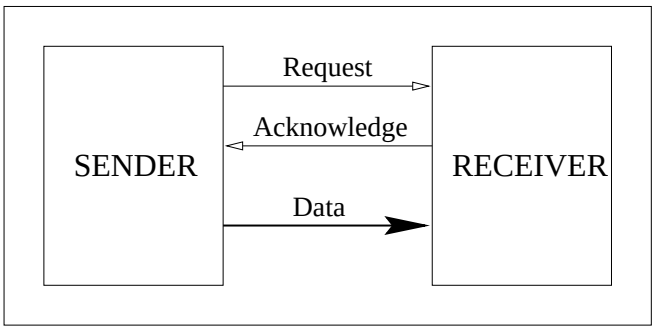

Figure 2.3: A bundled data interface

acknowledge, and some data wires between them, as illustrated in Figure 2.3.

Two-phase handshaking

Handshakes can be used on their own to synchronise a pair of blocks. In a two-phase handshake when the data is ready to be sent to the receiver, the sender informs the receiver that it has data to transmit by inverting the state of the request control wire. In some cycles the request event will be a rising transition and in some it will be a falling transition. Any transition event, either rising or falling, has the same meaning in the two-phase handshake. The receiver accepts the data and then produces an event on the acknowledge control wire to indicate that the data has been accepted. Figure 2.4 shows the bundled data interface with a two-phase handshake which is sometimes called *a transition signalling protocol*.

Four-phase handshaking

The four-phase handshake, sometimes called *a level signalling protocol*, is shown in Figure 2.5. Here, when data is available to be sent the sender produces a

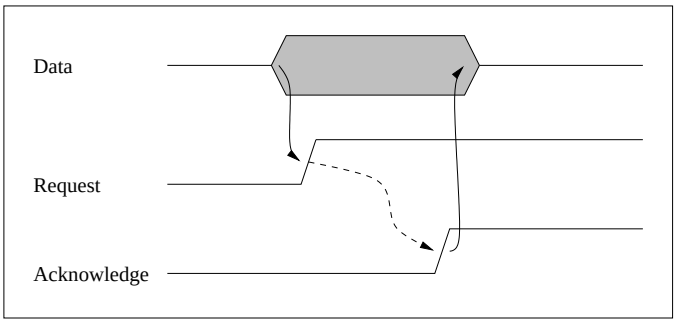

Figure 2.4: A two phase handshake

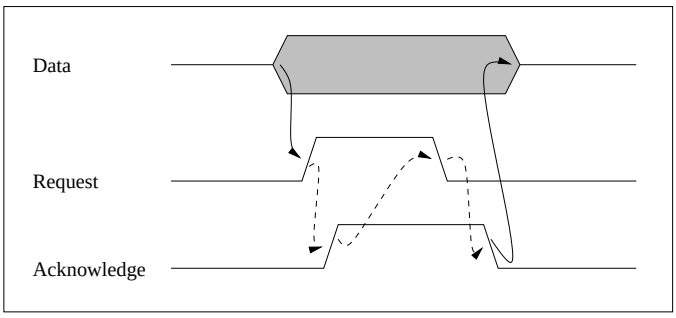

Figure 2.5: A four phase handshake

rising request signal. The receiver acknowledges by producing a rising signal on the acknowledge control wire. The sender then lowers the request signal which is acknowledged by the receiver by a falling signal on the acknowledge wire. With this kind of handshake, there are more transitions causing it to consume more energy.

2.3 Pipelining

Pipelining is an implementation technique that exploits parallelism among the instructions in a sequential instruction stream, and is used to improve the performance of processors. It comes from the observation that instruction execution can be split into a number of independent stages (*pipe stages* or *pipe segments*), allowing a number of instructions to be concurrent, each in a different stage of execution.

Pipelined processing is beneficial when all of the following are true [Kog81]:

- Each task is relatively independent of the previous one.
- Each task requires approximately the same sequence of stages.
- Those stages are closely related.
- The lengths of time to compute different stages are approximately equal. (For asynchronous pipelining, the time per stage is not necessarily constant but rather a function of both the stage and the data passing through it.)

For instance, assume there are fours tasks, A, B, C, and D. Each task can be split into four similar small jobs with subscripts 1 to 4. The non-pipelined approach has each task proceed sequentially as shown in Figure 2.6. When the first task, A, is done, the next task, B, starts, and so on. If each small job takes one unit of time to finish, the sequential approach takes 16 units of time for the four tasks in total.

The pipelined approach takes much less time. According to Figure 2.7 as soon as the first small job, *A1*, has finished and the second job, *A2*, has started, the first small job of the next task, *B1*, can begin. Then *C1* can begin when *A2* and *B1* have finished, and continue with *A3* and *B2*. At this point, all stages in the pipeline are operating concurrently, assuming the pipeline has separate resources for each stage. Since the real pipeline generally does not have totally separate resources for each stage, there are some situations which reduce the performance from the ideal speedup theoretically gained by pipelining. These situations are discussed later on in Chapter 4.

Pipelining improves performance by increasing the number of data values processed in a given time (the instruction *throughput*), as opposed to decreasing the time taken for an individual element to traverse the pipeline (the *latency*), but instruction throughput is the important metric because real programs execute billions of instructions.

Under normal conditions, if instruction execution could be split into n perfectly balanced stages, then an n stage pipeline would give n times the throughput of a non-pipelined version. Usually, however, the stages will not be perfectly equal. Furthermore, pipelining does involve some overhead. This is due to the starting and finishing of the parallel execution of tasks, as can be seen in Figure 2.7, where

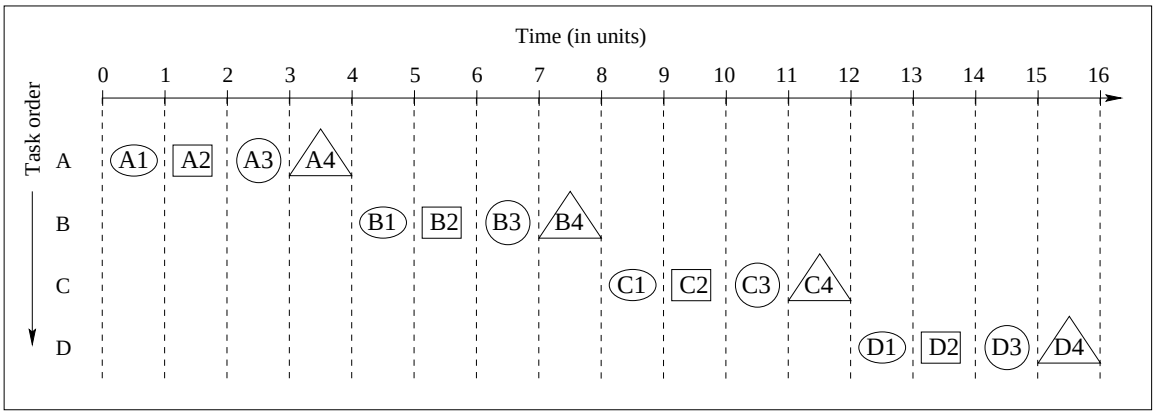

Figure 2.6: The non-pipelined approach

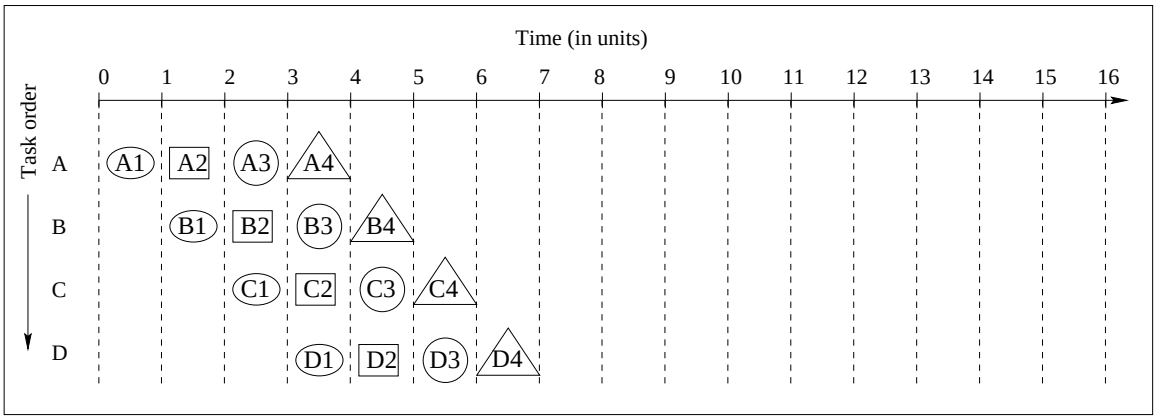

Figure 2.7: The pipelined approach

only at time interval 4 do the four tasks run simultaneously, yielding optimum performance.

In a computer pipeline, each pipe step completes a part of an instruction. Like an assembly line, different steps are completing different parts of different instructions in parallel. The stages are connected one to the next to form a pipe. Instructions enter at one end, progress through the stages, and exit at the other end.

2.4 Synchronous and Asynchronous Pipelines

This section takes a brief look at the differences between synchronous and asynchronous pipelines.

A conventional computer pipeline is a synchronous pipeline which is controlled by a global clock signal. In synchronous systems each operation of an arithmetic or logic unit has to be finished within a time slot given by the overall clock signal. Data signals have to be stable at latching time (at the edge of the control signals). The clock period of the synchronous pipeline is limited to a minimum of the time taken for the slowest pipeline stage to complete its processing. An example of a synchronous pipeline is shown in Figure 2.8

By contrast an asynchronous pipeline does not have any global clock, hence every stage can take a variable time to finish and can work independently. Therefore the next stage can begin after the previous stage has finished which theoretically makes an asynchronous pipeline faster than a synchronous pipeline. Since an asynchronous pipeline uses an asynchronous implementation approach, it also gets the benefits inherent in asynchronous circuits such as lower power and improved modularity. Figure 2.9 gives an example of how the timing of an asynchronous pipeline may look.

There are several hazards connected to pipelining which can result in a reduced

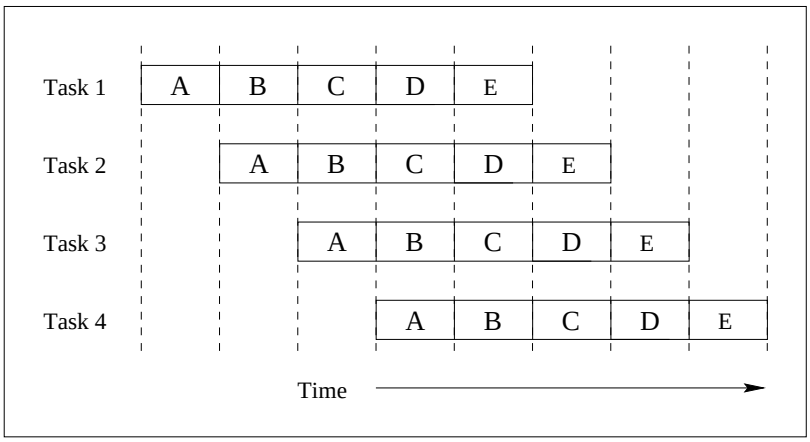

Figure 2.8: An example of a synchronous pipeline

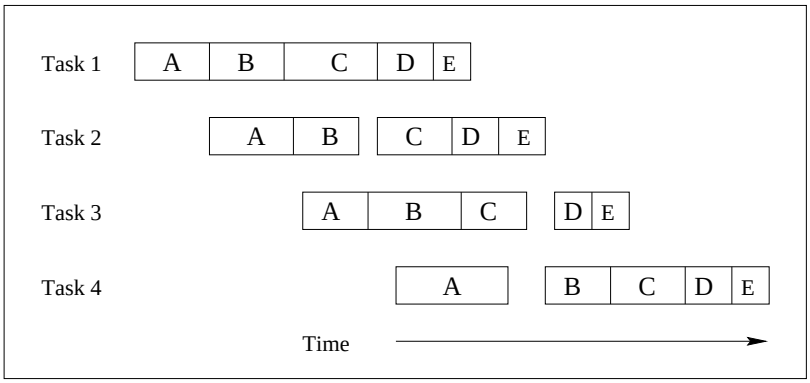

Figure 2.9: An example of an asynchronous pipeline

performance. The basic concepts for avoiding their occurrence are similar for both synchronous and asynchronous pipelines, although they are more complex for the asynchronous type. The various hazards and the techniques for avoiding them will be discussed in Chapter 4.

2.5 Summary

Pipelining is an approach to the way a processor deals with its tasks in which these are dealt with in a parallel fashion rather than using a simple sequential approach.

This improves processor performance since less time is needed to finish off a certain number of tasks. Furthermore, pipelining can be done either synchronously or asynchronously. The former relies on the classic external clock to synchronise its operation whereas the latter relies on a handshaking system. The asynchronous pipeline has some advantages over the synchronous version which make it attractive for implementing the DLX architecture.

Chapter 3

The Basic DLX Pipeline

An initial pipeline structure can be made by splitting the datapath into a number of independent stages which should take similar lengths of time to execute. Exactly how many pipeline stages are needed depends on what the costs of various pipeline depths are for the architecture being implemented.

A typical instruction can be split into several phases. For instance in DLX, as shown in Figure 3.1, the implementation of the DLX datapath allows every instruction to be executed in four or five clock cycles. In this implementation, branch instructions need four clock cycles to finish and all other instructions need five. That means up to five instructions will be in execution during any single clock cycle. To implement this datapath as a pipeline, each cycle refers to a stage of the pipeline. This may be called a five-stage pipeline.

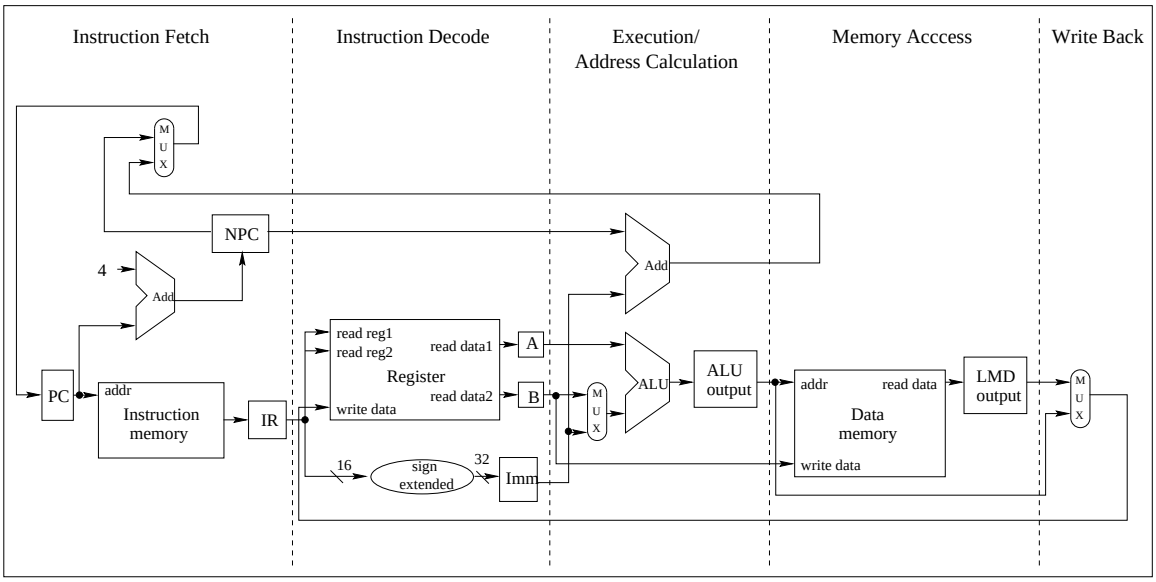

Figure 3.1: An implementation of a DLX datapath

To separate the datapath into five sections, each section has the name corresponding to a stage of the instruction execution:

- **IF** : Instruction fetch from memory.
- **ID** : Instruction decode and register read from the register file.
- **EXE** : Execution of the operation or address calculation.
- **MEM** : Data memory access for loading and storing instructions.
- **WB** : Write the result back into the register file (if needed).

When a DLX datapath is pipelined a set of pipeline registers is added. The result of pipelining the simple DLX datapath from Figure 3.1 is shown in Figure 3.2. The pipeline registers (shown as gray blocks) are used to separate the stages. They are labelled by the stages they separate *(IF/ID, ID/EXE, EXE/MEM,*

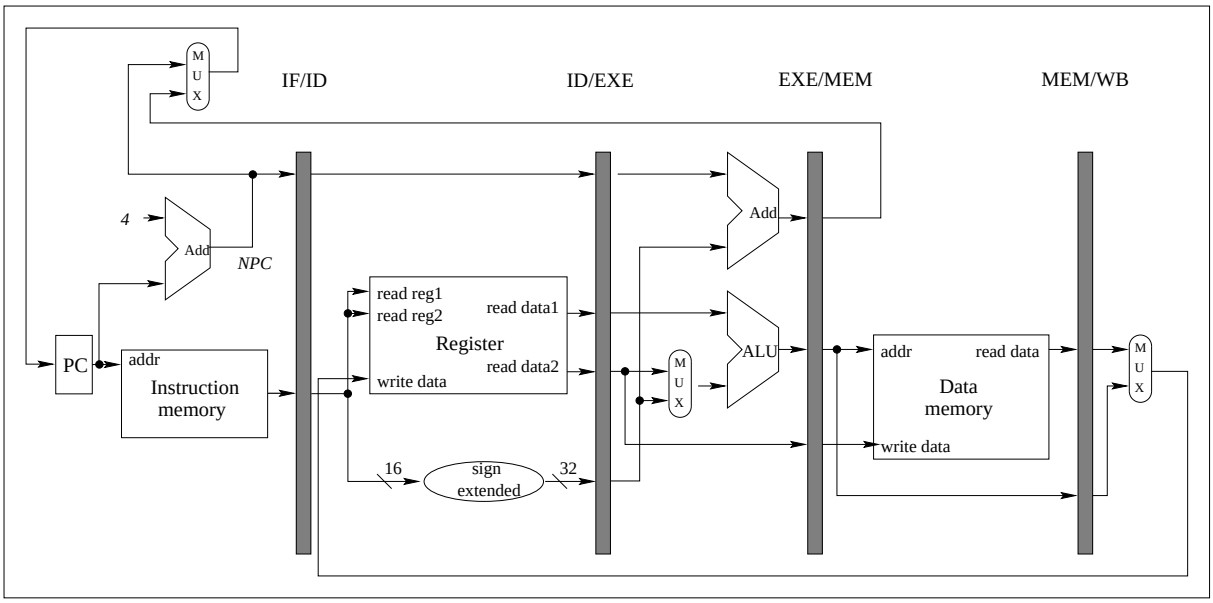

Figure 3.2: The pipelined DLX datapath

MEM/WB).

The rest of this chapter explains the basic pipeline for DLX in detail for each cycle.

3.1 IF: Instruction Fetch

```
IR ← Mem[PC]

NPC ← PC + 4
```

The instruction is read from the instruction memory into the instruction register (IR) using the address in the PC and then placed in the IF/ID pipeline register. The PC address is generally incremented by 4 to address the next sequential instruction and is then written back into the PC ready to be used in the next

data cycle. The IR, which is effectively part of the IF/ID register, is used to hold the instruction which will be needed in the next clock cycle. The incremented PC is saved as NPC in an IF/ID pipeline register as well, in case it is needed later as an operand (for example in a branch instruction). The IF stage occurs before the type of the instruction is identified. Hence it is the same for any instruction, apart from an earlier branch instruction which may cause a change in the PC to another target address.

- *Branch instruction:*

```
if (BranchCondition)
        then PC ← ALUoutput
        else PC ← NPC
```

If the previous instruction is a branch instruction and the condition of that branch instruction is true then the PC is replaced by the branch destination address in the register `ALUoutput`, computed in the execution cycle of the previous branch instruction, otherwise it is replaced by the incremented PC in the register NPC.

3.2 ID: Instruction Decode

$$\texttt{A} \leftarrow \texttt{Regs}[\texttt{IR}_{6..10}]$$

$$\texttt{B} \leftarrow \texttt{Regs}[\texttt{IR}_{11..15}]$$

$$\texttt{Imm} \leftarrow ((\texttt{IR}_{16})^{16} \#\# \texttt{IR}_{16..31})$$

This stage is also executed by all instructions, since it is still too early to identify the type of instruction. The instruction in the IF/ID pipeline register is decoded to generate the necessary data, such as the 16-bit immediate field which is sign-extended to 32-bits (`Imm`) and the register numbers (`A`, `B`) for the next stage. Apart from decoding, using the register numbers (`A`, `B`) as indices to access the register file to obtain register values occurs in this stage. These values are all stored in the ID/EXE pipeline register, along with the incremented PC address. In case any of these values are needed by the instruction further down the pipeline, everything is transferred.

3.3 EXE: Execution and Address Calculation

This stage changes depending on the type of DLX instruction. The ALU operates on the operands prepared in the previous cycle, performing one of the following four functions.

- *Memory reference (loads and stores):*

  `ALUoutput` $\leftarrow$ `A + Imm`

To form the effective address, the ALU adds the value in register `A` and the sign-extended immediate value in register `Imm`. Then the result is placed into the temporary register `ALUoutput` (which is part of EXE/MEM register).

- *Register-register ALU instruction:*

```
ALUoutput ← A  op B
```

The ALU performs the function defined by the opcode *op*, on the value in register `A` and on the value in register `B`. Then the result is placed into the register `ALUoutput`.

- *Register-immediate ALU instruction:*

```
ALUoutput ← A  op Imm
```

Similar to the register-register ALU instruction, but the second operand is the value in register `Imm` instead of register `B`.

- *Branch instruction:*

```
ALUoutput ← NPC + Imm

BranchCondition ← A  op 0
```

The branch target address can be computed by adding the values in register `NPC` and register `Imm`. Register `A` is used to determine whether the branch is taken. The comparison operation instruction *op* is determined by the branch opcode. For instance, *op* is '==' for the instruction `BEQZ` and is '!=' for the instruction `BNEZ`. The various forms of jump instructions are similar to branches.

3.4 MEM: Data Memory Access

This stage is executed only by the instructions which need to access the memory, load and store instructions, and branch instructions in order to forward the branch

target address back to the fetch stage. For all other instructions, the data from the EXE stage is passed directly to the WB stage.

- *Load instruction:*

```
LMDoutput ← Mem[ALUoutput]
```

If the instruction is a load, the data returns from memory, addressed by the `ALUoutput` register which is calculated in the prior cycle, and is placed in the `LMDoutput` (load memory data) register.

- *Store instruction:*

```
Mem[ALUoutput] ← B
```

If the instruction is a store, then the data from `B` is written into the memory addressed by the value in the `ALUoutput` register from the previous cycle.

3.5 WB: Write Back

This stage is used for reading the data from the MEM/WB pipeline register and writing it into the register file if necessary. Although some instructions such as the store instructions have no active function in this stage, it is more complicated to speed-up those instructions by using only four clock cycles to execute them. To keep the implementation simple, an instruction will pass through the write-back stage even if there is nothing to be written back.

- *Register-register ALU instruction:*

$$\texttt{Regs[IR}_{16..20}\texttt{]} \leftarrow \texttt{ALUoutput}$$

- *Register-immediate ALU instruction:*

$$\texttt{Regs[IR}_{11..15}\texttt{]} \leftarrow \texttt{ALUoutput}$$

- *Load instruction:*

$$\texttt{Regs[IR}_{11..15}\texttt{]} \leftarrow \texttt{LMDoutput}$$

The operation of this stage is to write the result into the register file, whether it comes from the memory system, which is `LMDoutput`, or from the ALU, which is `ALUoutput`. The register destination field depends on the opcode. Figure 3.3 shows a simplified version of the DLX datapath, drawn in pipeline style, which allows every instruction to be executed in five stages.

3.6 Summary

A pipelined implementation of the DLX architecture has been presented. This was done by splitting the datapath into the five distinct stages of the execution of an instruction and inserting four pipeline registers. Each stage and its principle of operation has been discussed in detail.

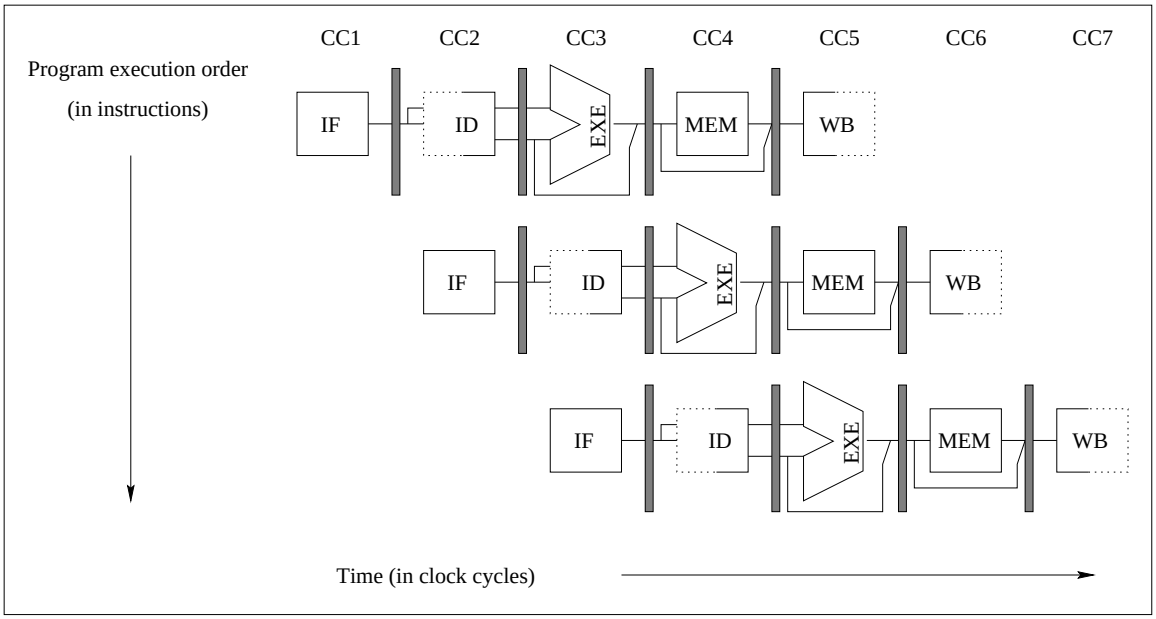

Figure 3.3: The pipelined DLX

Chapter 4

Pipeline Hazards

The last two chapters show the power of pipelined execution. This chapter presents what happens to pipelining with real programs.

Figure 4.1 shows how consecutive instructions would overlap in this pipeline. If each of the pipe stages was completely independent and took an identical time, then this diagram would show exactly how consecutive instructions complete.

However, since these pipe stages are not completely independent, there are a number of events, known as *pipeline hazards* or *pipeline dependencies*, which decelerate the speed of a pipeline when the next instruction cannot execute in the following clock cycle. A hazard in a pipeline is basically an aspect of its design or use that prevents new data from continually entering at the maximum possible rate. Hazard must be resolved to create a normal working pipeline structure.

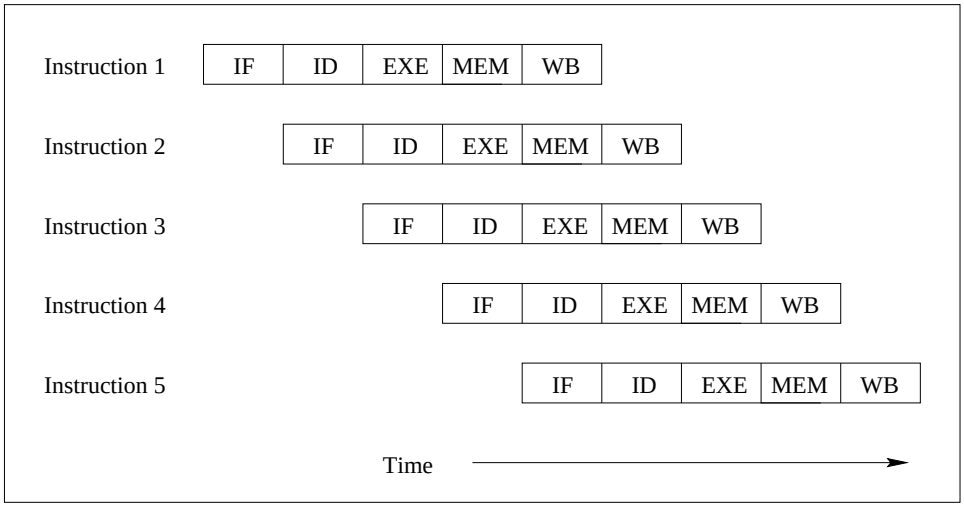

Figure 4.1: An example of a 5-stage pipeline structure in DLX

There are three classes of pipeline hazards:

- *Structural hazards* arise from resource conflicts which occur when the machine cannot support all possible combinations of instructions in concurrent overlapped execution.

- *A data hazard* is a situation in which the later instruction is supposed to use the results from an earlier instructions in its calculation.

- *Control hazard* arises from the modification of the PC, mainly by branch/jump instructions and other instructions.

The following discussion on resolving pipeline hazards assumes each pipeline stage takes an identical time to complete, based on the five-stage pipeline for DLX as discussed in the previous chapter. Whilst this may be true for a synchronous

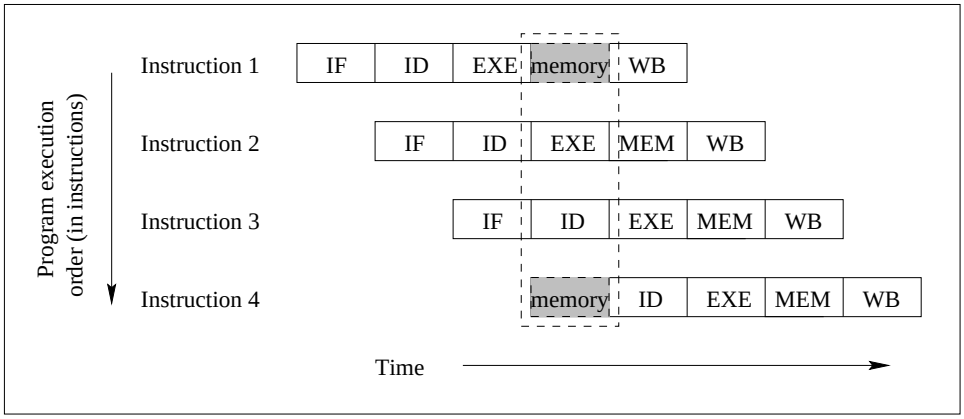

Figure 4.2: An example of a memory conflict

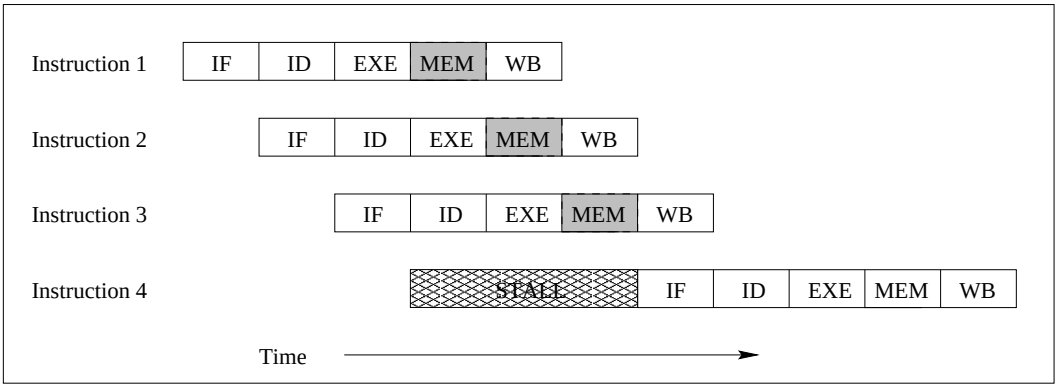

Figure 4.3: A pipeline stalled for memory conflict

implementation, it is unlikely to be true for an asynchronous system. However the general arguments remain true with both implementation approaches.

4.1 Structural Hazards

When a machine is pipelined, if some combination of instructions cannot be processed because of resource conflicts between instructions (which occur whenever more than one pipeline stage wishes to use the same resource at the same time) this will generate a structural hazard.

4.1.1 Memory Conflict

Some pipelined machines have a shared single-memory pipeline for data and instructions. As a result, a memory conflict can occur when two stages attempt to access the memory simultaneously. Figure 4.2 shows an example of this type of conflict. Here a memory stage (MEM) needs to be loaded with data from the memory in the first instruction, whilst at the same time an instruction fetch stage (IF) in the fourth instruction needs to fetch an instruction from memory.

Memory Conflict and Stalling

When a sequence of instructions encounters this hazard the simple method for resolving it is to stall one of the instructions until the required resource is available. A stall, since it floats through the pipeline taking space but carrying no useful work, is commonly called a *pipeline bubble* or just *bubble*. It causes the pipeline performance to degrade from the ideal performance by increasing the CPI [1] from its usual ideal value of 1. Figure 4.3 illustrates a pipeline stalled for a structural hazard in the case of a memory conflict.

Memory Conflict and Separating Memory

An alternative solution to this structural hazard would be to separate memory access for instructions and data, either by splitting the cache into separate instruction and data caches, or by using a set of buffers, generally called *instruction*

[1] Clocks Per Instruction

buffers, to hold instructions. The use of split caches eliminates the conflict for a single memory that would arise between the instruction fetch (IF) and data memory access (MEM).

4.1.2 Register File Conflict

The register file is used in two stages: for reading in the instruction decode stage (ID) and for writing in the write-back stage (WB). When considering the five-stage pipeline for DLX from the previous chapter, the following observations regarding register file read and write should be noted:

- Read : The DLX integer pipeline always does register fetches in the instruction decode stage.
- Write : The DLX integer pipeline does register write in the write-back stage.
- The write-back stage comes further down the pipeline than the instruction decode stage.

Accordingly neither two writes nor two reads can occur simultaneously. Hence a register file conflict can only arise from the need for one instruction to write and another instruction to read at the same time.

For example, whilst in the instruction decode stage (ID), the processor normally needs to read the required data from the register file. At the same time there is

another stage, the write-back stage (WB) as shown in Figure 4.4, which wishes to write a result back to the register file. In this case the system is said to have a structural hazard as well.

Register File Conflict and Stall

In a similar way to stalling for a memory conflict, this hazard can be removed by stalling one of the instructions that produces the conflict in the pipeline, when the register access occurs, until the resource is free.

Register File Conflict and Half Read-write

There is another simple implementation technique used to avoid register file conflicts. This technique is to perform the register file writes in the first half of the cycle and the register file reads in the second half. Then data hazards caused by register conflict will be eliminated since the two kinds of access, read and write, do not occur at the same time.

Figure 4.5 shows the overlap among the parts of the datapath. Since the register file is used as a source in the ID stage and as a destination in the WB stage, it appears twice. This figure shows that it is read in one stage and written in another by using a solid line, on the left or right, respectively, and a dashed line on the other side. Memory is split into instruction memory (IMEM) and data memory (DMEM).

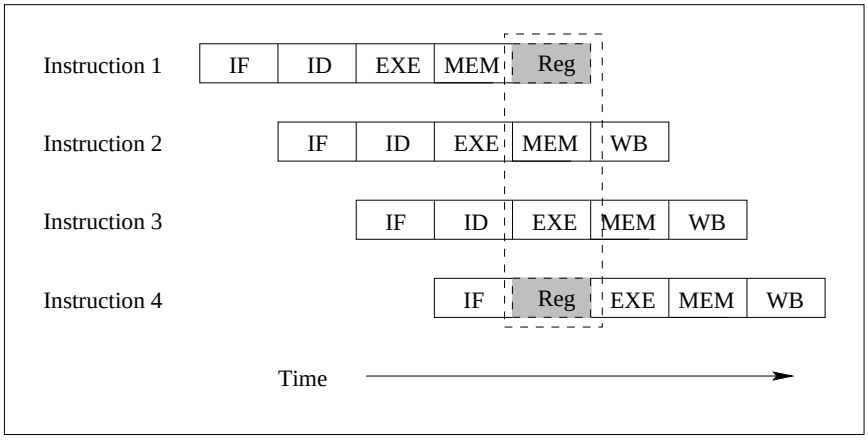

Figure 4.4: An example of a register file conflict

In general, the choice of the solution used to resolve a structural hazard would depend on how often the resource clash is likely to occur, and what the cost of using each solution would be.

4.2 Data Hazards

This hazard arises from data dependencies which are the most common type of dependencies. It occurs when an instruction depends on the result of a previous instruction in a way that is exposed by the overlap of instructions in the pipeline. It is the main reason that high-performance pipelines are hard to design. Correct operation in a pipelined processor requires that data hazards between instructions are resolved.

These hazards may be classified into four types according to the order of read

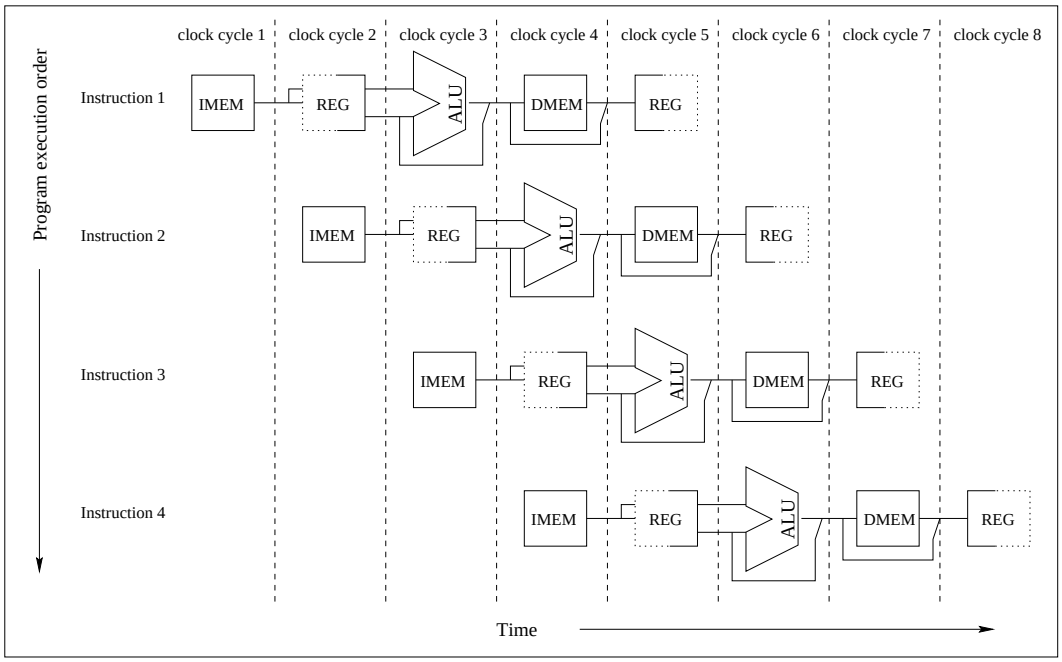

Figure 4.5: Pipeline with separate I&D memories and half read-write for the register file

and write accesses in the instructions.

4.2.1 RAR (read-after-read) Dependency

For a RAR hazard, since there is no modification of any object and reads all happen in the same stage in a DLX pipeline, they must occur in the order of instruction executions, thus there are no hazards of this type in the DLX pipeline.

4.2.2 WAW (write-after-write) Dependency

A WAW hazard exists when both instructions i and j (j assumed to logically follow i) attempt to update the same register or memory location, but i's work can finish after j's. The result is that after both instructions have completed, the

object may be left with an intermediate value (from i) and not the final value (from j).

Fortunately, since the DLX integer pipeline writes registers in the same stage (WB) and does the memory stores only in the memory stage (MEM), WAW hazards do not occur.

4.2.3 WAR (write-after-read) Dependency

A WAR hazard exists when instruction j (logically following i) wishes to modify some object that is read by i. If j modifies the object before i has accessed it, i will get the wrong value, although now the value is too new rather than too old.

According to the DLX integer pipeline, register reads occurring in the instruction decode stage (ID) are always before register writes in the write-back stage (WB), thus there is no problem with WAR hazards either.

4.2.4 RAW (read-after-write) Dependency

This hazard occurs when an instruction needs to read a register but the result of an earlier instruction has not yet been written back. Therefore that register will contain the wrong value. This is the most common type of hazard.

Consider the following sequence of instructions with some dependencies, shown

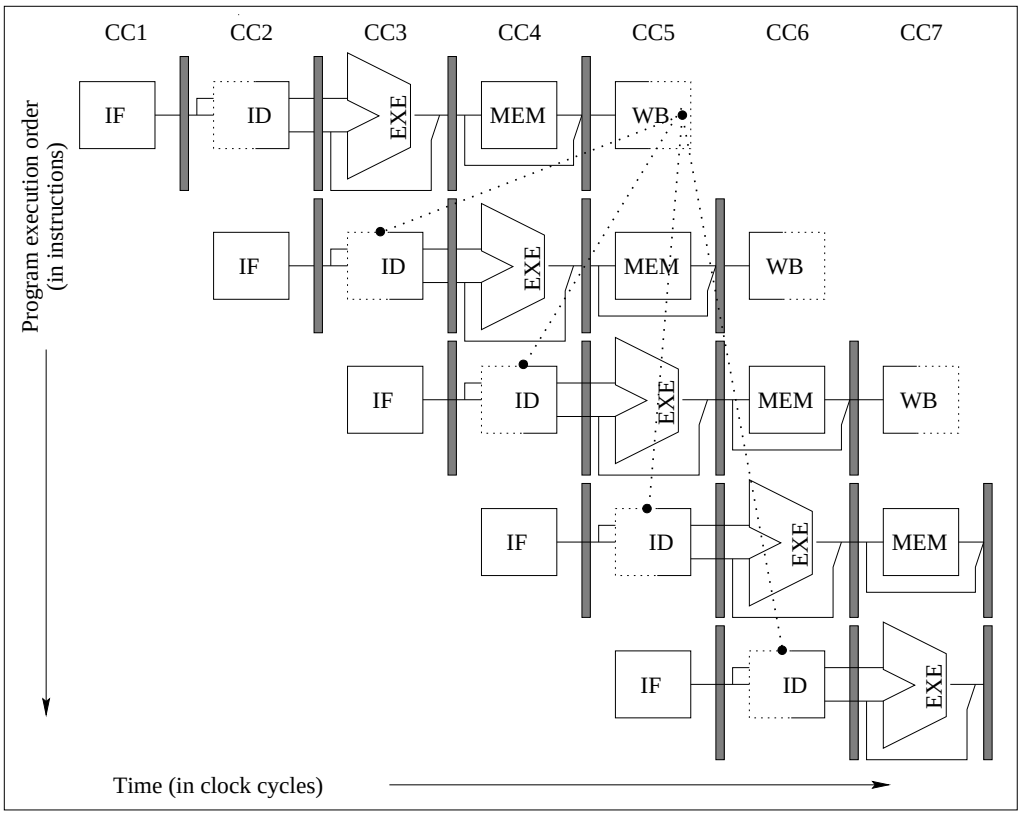

Figure 4.6: An example of a data hazard

in blocks:

```
ADD r1, r2, r3
AND r4, r5, r1
OR  r6, r1, r7
SUB r8, r1, R1
XOR r10, r1, r9
```

The last four instructions are all dependent on the result in register **r1** from the first instruction. Figure 4.6 shows the data hazard arising from the use of the result of the first instruction in the next three instructions, since register **r1** is not written until the write-back pipe stage at the end of clock cycle 5, which is after

those instructions read it. The `AND` instruction reads the register during clock cycle 3. Similarly the `OR` instruction tries to read the register during clock cycle 4. Unless precautions are taken, those two instructions will read the wrong value and use it. By using the technique that splits reads and writes to the register file into different halves of the cycle, the `SUB` and `XOR` instructions can operate without incurring any data hazard. The fifth instruction has no hazard since **r1** is already written back in the previous stage, just before the fourth instruction reads that data.

4.2.5 Stall

The most straightforward approach to solve data hazards in hardware is to prevent a new instruction issue (when a hazard arises), through the use of stalls, until the results of the previous instructions have been written back. This has a consequential performance degradation. Figure 4.7 shows the stalling approach to resolving a data hazard.

4.2.6 Register Locking

There is a process termed *locking* [PDF+92], which may be used to make sure that a location with a pending modification cannot be accessed until the write operation has completed. The stall frequency can be reduced with the addition of a lock to each register so that the pipeline only stalls if an instruction attempts to read a locked register. According to Figure 4.7, this stall will happen only

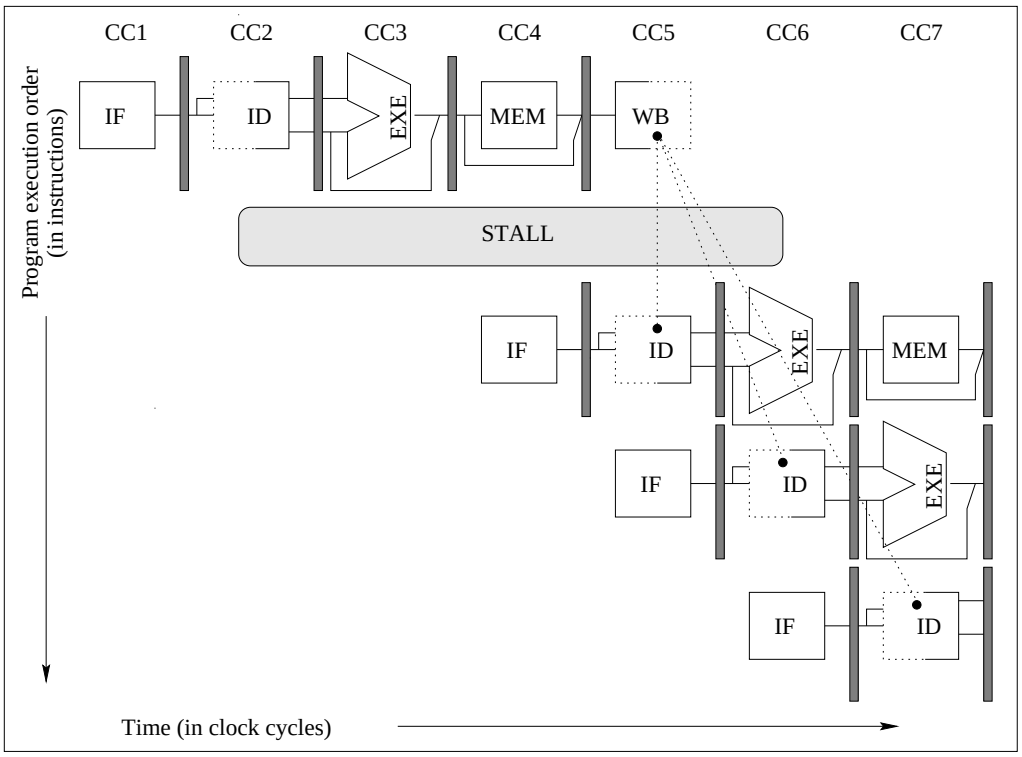

Figure 4.7: Stall to prevent data hazard

when the second instruction requires the result from the first instruction.

4.2.7 Forwarding

Instead of stalling instructions waiting for the result to be written back from the write-back stage (WB), which takes time, there is a technique called *forwarding* or *bypassing* which uses temporary results. As illustrated in Figure 4.8, for ALU instructions the result from the first instruction is valid at the end of the execution stage (EXE) whilst the second instruction needs the data in that register in its execution stage (EXE). Forwarding the temporary result could be done here from the end of the execution stage (EXE) in the first instruction to the beginning of the execution stage (EXE) in the second instruction. Similarly for the third

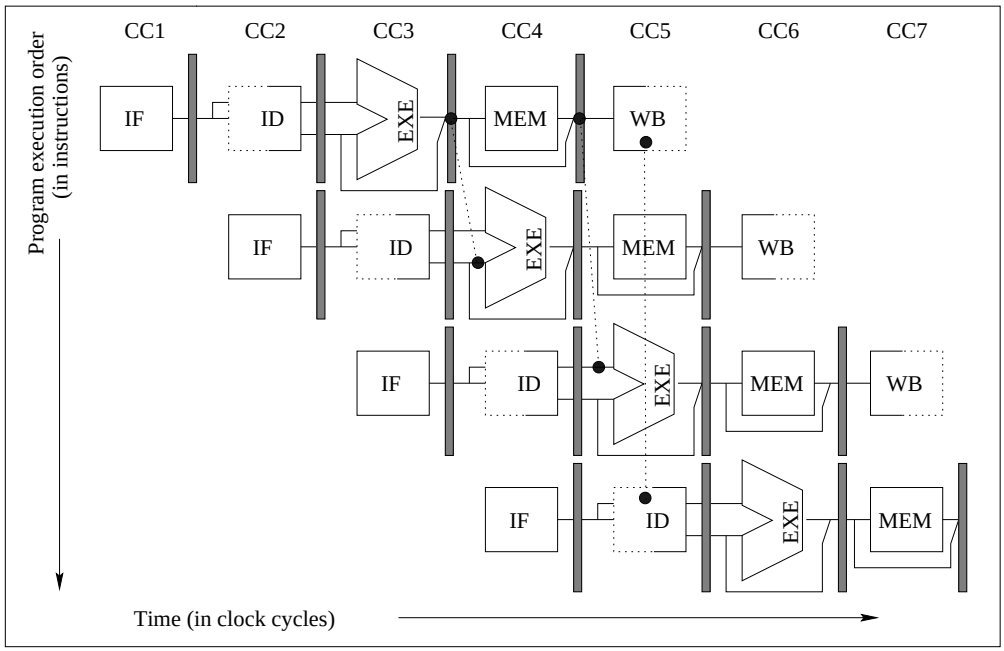

Figure 4.8: The forwarding technique to avoid the data hazard

instruction, which also requires the result from the first one, the temporary result can be forwarded from the memory stage (MEM) to where it is really needed, in the execution stage (EXE). Since the register read and write are split into two halves, in the fourth instruction there is no hazard as the result from the first instruction is written in the first half of the clock cycle and the instruction decode stage (ID) in the fourth instruction reads the required registers in the second half of that same clock cycle [GG97].

Unfortunately, not all data hazards can be handled just by forwarding. Consider the following sequence of instructions with some dependencies, shown in blocks:

```
LW  r1, 0(r0)
```

```
AND r2, r1, r3
SUB r5, r4, r1
ADD r7, r1, r8
```

The pipeline datapath with forwarding for this example is shown in Figure 4.9. Since load instructions provide the result after the memory stage (MEM), forwarding the result backward in time to the earlier clock cycle as shown in the figure is impossible. The data hazard in this case cannot be eliminated by just simple forwarding as used in the previous example. Stalling for at least one clock cycle is needed to make the `AND` instruction wait until the result from `LW` is valid which is at the end of the fourth clock cycle. Figure 4.10 illustrates how to handle data hazards caused by load instructions.

Since store data is needed only in the memory stage (MEM), it is not necessary to get the required register data before that stage. Hence in this case, it is more straightforward than a normal ALU instruction. Figure 4.11 shows that a store instruction executed after the load instruction can proceed without any stall if the appropriate forwarding path is available.

In synchronous systems, each stage works synchronously. Hence it is predictable how many instructions will be affected in a data hazard. Without synchronisation however, it is hard to solve data hazards by exploiting these techniques without additional mechanisms.

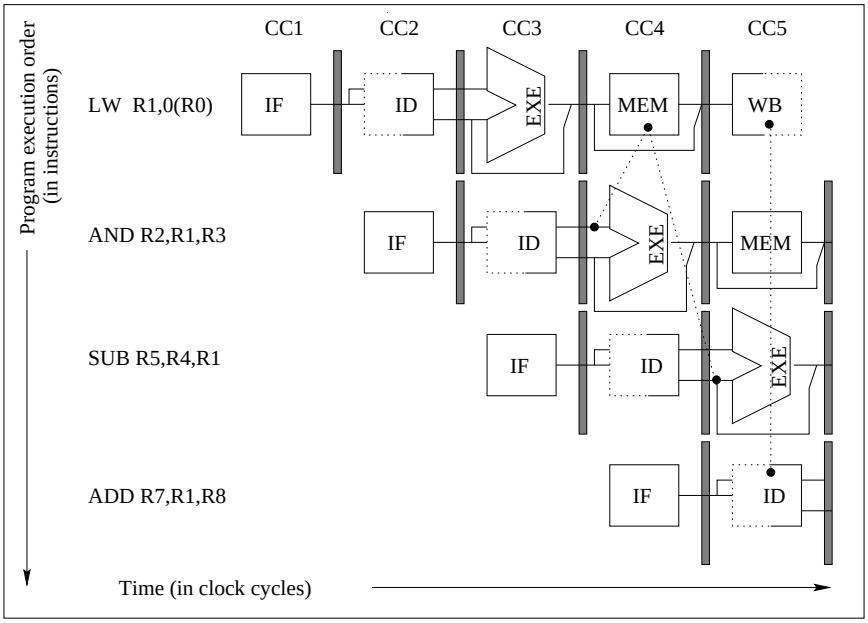

Figure 4.9: The impact from only forwarding for Load instructions

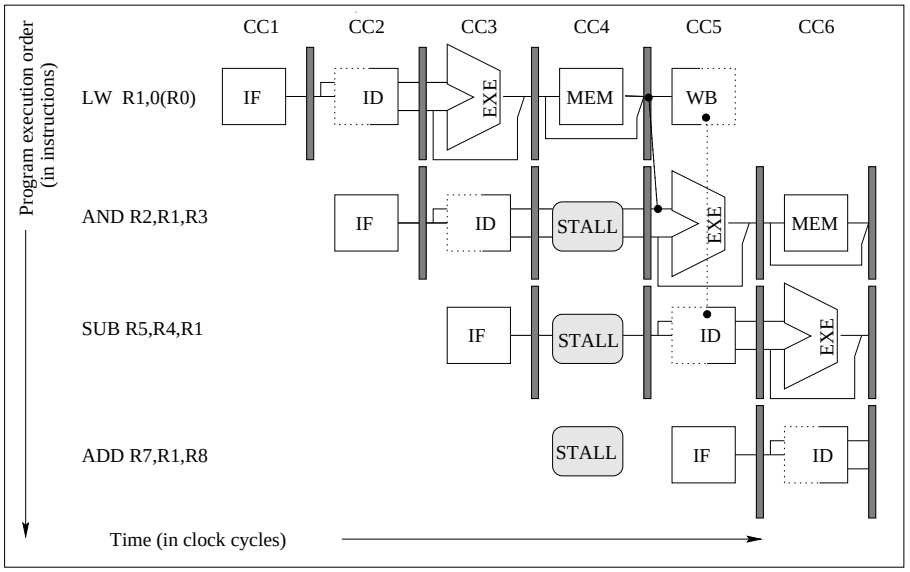

Figure 4.10: Combination of forwarding and stalling to prevent data hazard

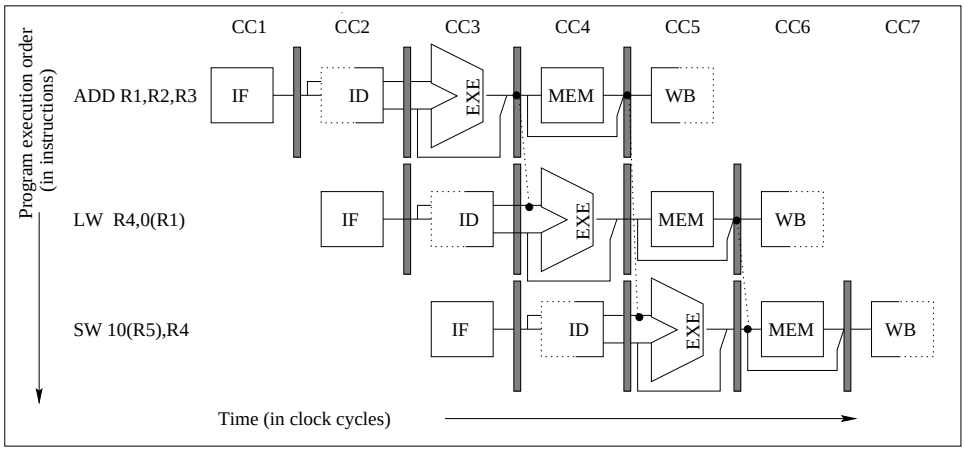

Figure 4.11: Load instruction followed by a store instruction

4.3 Control Hazards

A control hazard affects how branch instructions are handled, and arises from the need to decide the address of the subsequent instruction to fetch, determined from the results of the branch instruction, whilst other instructions are executing. When a branch is executed, it may or may not change the PC to something other than its normal next value. But the branch target address may not be calculated until the end of the memory stage. Meanwhile, an instruction must be fetched at every clock cycle to sustain the pipeline. Hence, there will be a number of instructions already in the pipeline which may not be needed.

If a branch is in the pipeline, that part of pipeline behind the branch must be flushed and refilled with the appropriate instructions. In this case, time is wasted processing the wrong sequence of instructions. Worse yet, any instruction after the branch which should not have been executed can alter data that is still needed.

Although control hazards are comparatively less complicated to understand and occur much less frequently than data hazards, they can potentially cause a larger loss of performance than other hazards do. Since writing back incorrect results is harmful, this error must be prevented.

4.3.1 Stall

There are several things that can be done with the instructions sequentially following the branch. Since the branch target address will not be calculated before the memory stage (MEM), one of these solutions is to stall the next instruction after a branch instruction until the target address is calculated. This means that after a branch instruction is taken, the pipeline will not fetch any instructions until the branch target address is sent to the PC. Then the pipeline executes the instruction at that target address in the PC. This solution is quite simple to implement. But this makes the pipeline take much longer to finish, whether the branch is taken or not. Figure 4.12 illustrates this stall technique which is used to delay prefetching until the branch target address has been calculated.

To implement this approach, in fact it is impossible to recognise a branch until the end of the ID stage, so the pipeline cannot prevent at least one instruction fetch after the branch. These prefetched instructions must not be executed. Then the pipeline must be stalled until the branch target address is valid. The next executed instruction is the new instruction at that target address.

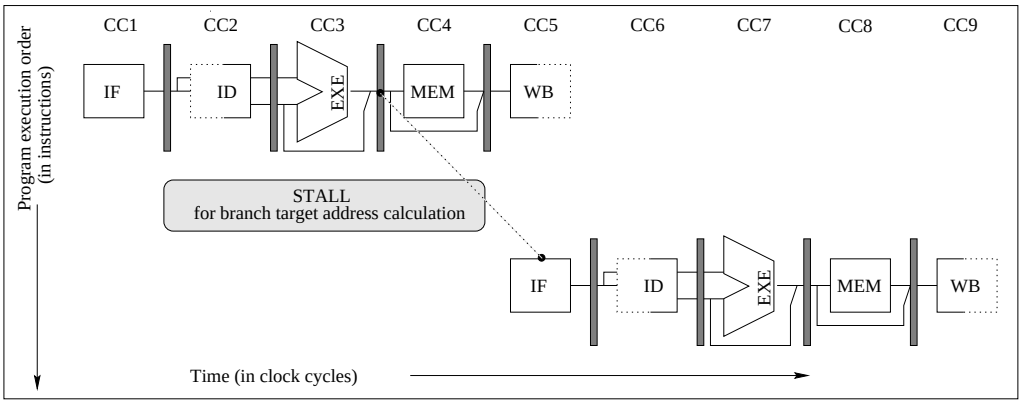

Figure 4.12: Stall to prevent control hazard

4.3.2 Speculative Execution

The other possible solution is always to execute the instructions following the branch speculatively, on the assumption that their execution will be useful, but prevent any state from being changed until the branch target address and condition have been calculated and it is known that they should be executed. This technique is known as *speculative execution* [Eng96]. If some speculatively executed instructions should not have been processed then they must be flushed, discarding any result that they created. Then the pipeline continues at the address of the branch target. If branches are not taken half of the time, and if it costs little to throw the instructions away, this technique could halve the cost of control hazards.

The dashed block in Figure 4.13 is a speculative execution which is processed regardless of whether or not the branch in the first instruction is taken. But writing results back from this speculative execution is prevented if the branch

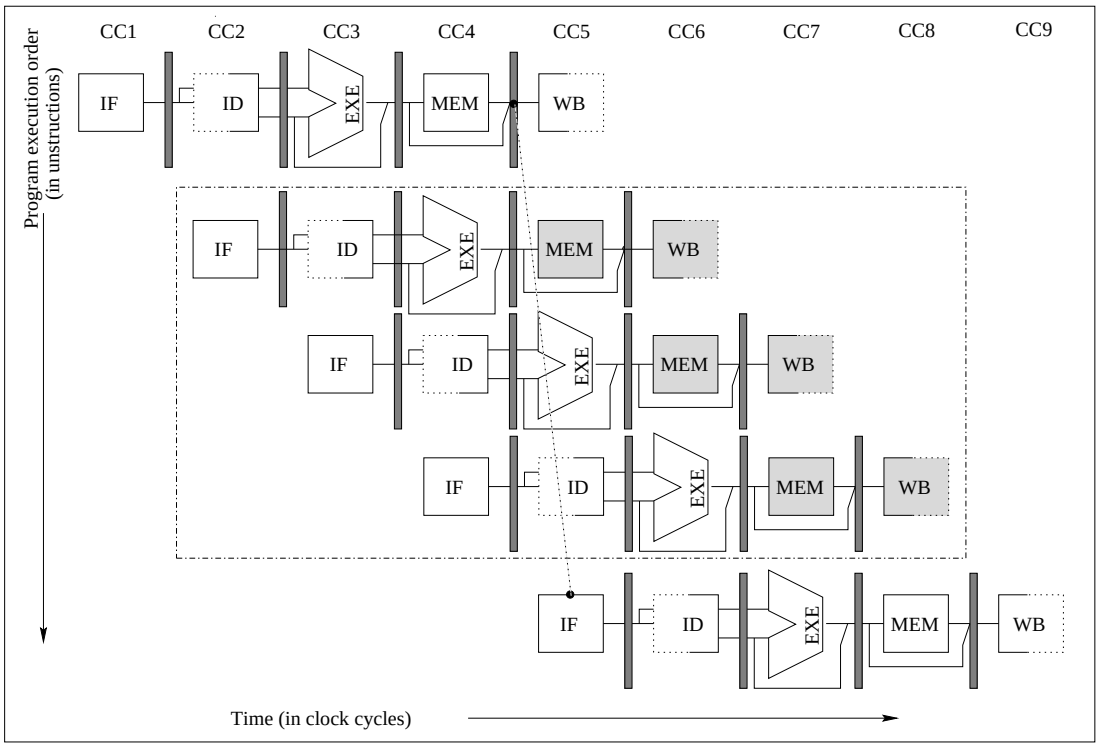

Figure 4.13: The speculative execution for the control hazard

instruction is not taken.

4.3.3 Moving up the Branch Address Calculation

If the branch execution is moved to an earlier stage in the pipeline, then fewer instructions need to be flushed when the branch is taken. So far, the branch address calculation has been done in the memory stage (MEM). It could be possible to save one or more clock cycles of penalty by moving up the branch address calculation to the end of the execution stage (EXE) instead, or even at an earlier stage in the pipeline such as in the instruction decode stage (ID). This can be implemented by simply moving the branch adder from the memory stage (MEM) to the earlier stage [PH97]. This will reduce the cost of the taken branch.

4.3.4 Branch Prediction

Another solution is to have a mechanism which attempts to predict the correct path of instructions based on past behaviour. For instance, some machines provide a *branch history table* or *branch prediction buffer*, a small amount of memory which maintains a record of previous decisions for each of several branch instructions that have been used in the program to guide the prediction. However, prediction is just a hint that is assumed to be correct. Hence, of course, when any branch prediction is taken incorrectly, there is time consumed to flush and refill the pipeline. Branch prediction mechanisms have proven to be effective, though they are often complex.

4.4 Summary

A classification has been offered for the various pipeline hazards into structural, data and control hazards. Each type of hazard has been extensively investigated as to their effect on processor performance, and ways of resolving them have been discussed.

However, caution should be taken when examining the methods for resolving these pipeline hazards, since they only apply to synchronous systems. When dealing with asynchronous pipelines, different measures are required in order to eliminate pipeline hazards.

Chapter 5

Asynchronous Processor Models

This chapter describes the asynchronous processor non-pipelined and pipelined models (three-stage and five-stage pipelines) of the DLX processor that were created for this project. First the objectives behind the models are given, followed by an introduction to the initial models, non-pipelining version and pipelining versions (both three-stage and five-stage). Finally, the development of five-stage pipelining models will be discussed.

5.1 Objectives Behind the Initial Models

When developing a new design, it is important to define the limits of possible design decisions by indicating the objectives of the design, what criteria are used and what hypotheses can be made. Here the first and most important objective is to create two initial asynchronous pipeline models of the same processor architecture using two different kinds of design styles, a three-stage and a five-stage

pipeline. The following standards and assumptions were applied to guide the design decisions:

- The designs implement the DLX architecture.
- The designs issue only one instruction per time cycle. (Note: Super-scalar pipelining was not considered)
- The environment for the initial models is the same. (The criteria of timing characteristics is the same.)
- Timing information for the various components of each initial model can be comparable.

The initial models of the asynchronous non-pipelined design and the pipelined designs had slightly different objectives. For the asynchronous non-pipelined and three-stage pipeline design, the aim was to attempt to design the final models straight away. These resultant models were not expected to be optimal. They were intended to provide the reference designs to be compared against each other and against the asynchronous five-stage pipeline version.

The primary aim of the initial five-stage pipeline model was to design it with an emphasis on simplicity. Consequently the design could be analysed easily and improved in order to provide better performance.

5.2 The Initial Models

There are three initial models: asynchronous non-pipelined, asynchronous three-stage pipeline and asynchronous five-stage pipeline models.

5.2.1 Asynchronous Non-Pipelining Design

The simple processor datapath discussed in Section 2.1 was modelled here using a self-timed logic design style instead of a global clock system as used in a synchronous design.

In this design the processor executes only one instruction at a time. Each instruction is fetched from the memory, decoded and executed. Then, after finishing the whole execution process of an instruction, the next instruction is fetched and processed by the processor. Hence there are no hazards in this design. The memory does not need to be split into separate instruction and data memories since memory conflicts will not occur. Similarly, it is unnecessary to have different cycle-halves to read from and write to the register file. Because there is no difficulty in dealing with any conflict, an asynchronous design is quite simple to implement.

The result from this model should be faster than a synchronous non-pipelined model since each instruction can be processed after the previous one has finished. It is not necessary to wait for the next clock tick as in a synchronous design.

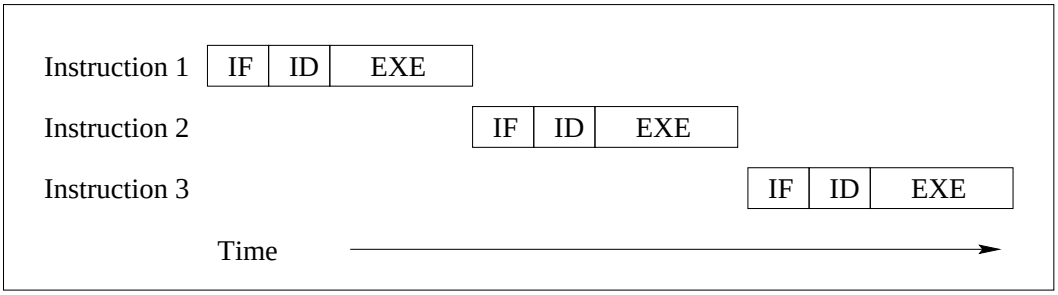

Figure 5.1: The timing diagram of a non-pipelined processor

However, the time needed to execute instructions in a non-pipelined model for an asynchronous processor, as illustrated in Figure 5.1, could be cut by using pipelining which will be discussed in the following sections.

5.2.2 Asynchronous Three-stage Pipeline Design

To allow higher execution rates pipelining techniques were implemented in the processor. In order to create a pipeline structure, each instruction needs to be split into subtasks for the asynchronous processor to handle. By using three separate subtasks - fetching, decoding and executing - the three-stage pipeline was created with three different stages: instruction fetch (IF), instruction decode (ID) and execution (EXE).

Whilst the non-pipelined processor deals with each instruction sequentially, the three-stage pipeline allows up to three *different* stages to be processing simultaneously. Hence, the three-stage pipeline approach should take less time than the non-pipelined approach to execute a given instruction sequence.

Whereas it can be assumed that there is only one simple block in the asynchronous non-pipelined model, some mechanisms have to be added in an asynchronous three-stage pipeline in order to be able to perform the same functions. These mechanisms handle PC-updating and write-back. Figure 5.2 shows the structure of a three-stage pipelining model. The PC-updating mechanism is used for updating the value of the PC, and comes into play when dealing with branch target addresses. The write-back mechanism is used for writing the result to the proper register in the register file. A timing diagram of the three-stage pipeline model is displayed in Figure 5.3.

As a result of dependencies between the various pipeline stages, several hazards will inevitably arise in the model, which do not arise in non-pipelined models. Mechanisms to solve these hazards will now be discussed.

Separate I&D Memories

There are two stages (IF and EXE) which need to access the memory, and usually they need to access it at the same time. This model uses the separate memory technique by splitting the memory into instruction and data memories (as discussed in Section 4.1.1) to resolve this conflict. Consequently both stages can access their dedicated memory without any delay, thus improving the operating speed and avoiding any memory conflicts.

Register Locking and Stalling

Similar to the memory dependency, register dependencies between consecutive instructions cause the pipeline to stall. Register locking is used to indicate which register is being updated and to prevent an out-of-date register from being read.

Colour Mapping

Due to the fact that a pipelined design executes instructions in parallel, the processor is required to prefetch the next instructions while executing the present ones. This is called a prefetch system. It is impossible, however, to predict the number of instructions that have been prefetched at any given moment. When a branch instruction is taken, the prefetched instructions will most likely be invalid. Consequently a system will be needed to keep track of the prefetched instructions since they will have to be flushed when they become invalid.

The addition of a very useful technique in asynchronous logic design, *'colour mapping'* is used in this model. This removes unnecessary prefetched instructions after any branch instruction is taken. The colour remains the same until there is a branch request. When it outputs the first address of the new instruction stream, the colour bit is inverted. Each instruction is prefetched with its 'colour', and arrives at the first stage of the pipeline for execution. The colour information indicates whether the instruction was from the original stream (to be flushed) or from the new stream (to be executed). After discarding those instructions which

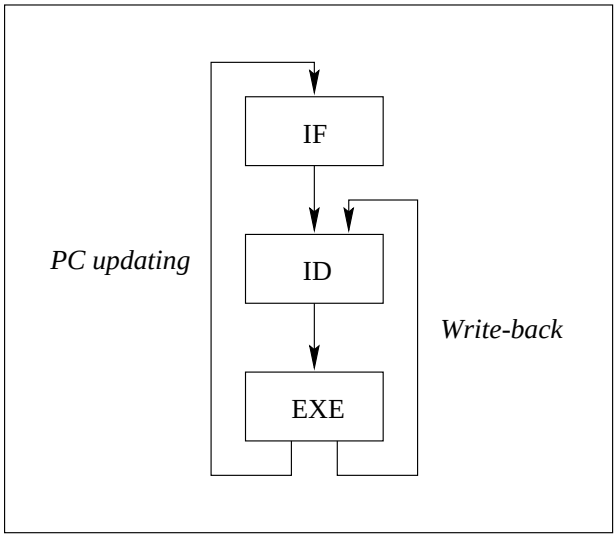

Figure 5.2: The structure of a 3-stage pipeline

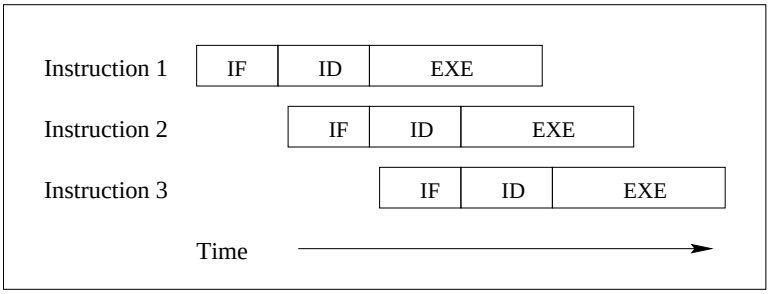

Figure 5.3: The timing diagram of a 3-stage pipeline

have a different colour from the new reference colour, the following instructions in the new stream are executed. The best position for colour checking is at the EXE stage which is the earliest stage at which it is known whether or not the branch will be taken. [Pav94]

5.2.3 Asynchronous Five-stage Pipeline Design

The next model was implemented as an asynchronous five-stage pipeline design which came from the observation that since the last stage in the three-stage

pipeline has much more work to process, it obviously takes much longer to finish. It could be practically possible to split this stage into three independent pipe stages; execution (EXE), memory access (MEM), and write-back (WB). (Note: the EXE stages in three-stage and five-stage pipelines have different meanings.)

Moving up Branch Address Calculation

Unlike the five-stage pipeline for the DLX architecture discussed in Chapter 3, this initial model was implemented to reduce the branch penalty by moving up the branch target address calculation to the end of the EXE stage (from the dashed line to normal line in Figure 5.4). The structure of the five-stage pipeline model is illustrated in Figure 5.4. Figure 5.5 shows a rough example of the timing diagram of this model.

Details on the Added Pipe Stages

Without adding any complex channels, each instruction has to pass through the same five stages. Although some instructions (i.e. ALU and Branches) do not need to access memory, they still need to pass through the channel between the EXE and MEM stage to reach the WB stage. Furthermore, even though it is not necessary to write back a result, the WB stage is processed for each instruction in order to update validated results and / or unlock the register. Because of this, the five-stage pipeline has to waste time in order to keep the pipeline simple.

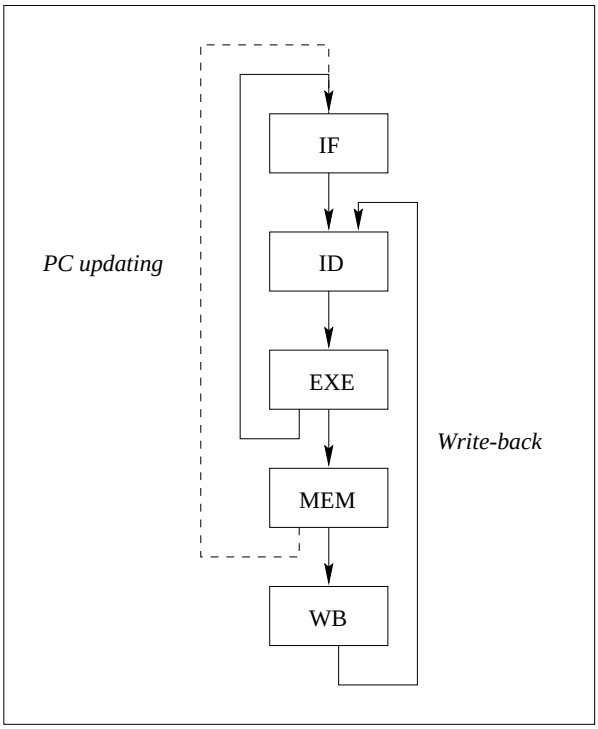

Figure 5.4: The structure of a 5-stage pipeline

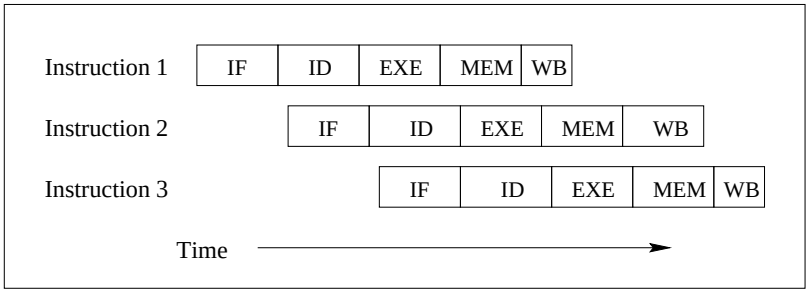

Figure 5.5: The timing diagram of a 5-stage pipeline

5.3 Implementations of Five-stage Pipeline

The aim of the five-stage pipeline design is to analyse and improve the initial five-stage pipeline model. This section deals with the actual five-stage pipeline models as they were implemented. They mainly differ from the initial design in the way the ID-stage reads the data from the register file. As will be discussed in Section 5.4, the ID-stage takes 8 units of time to finish. The first three units of time are used to decode the instruction type, while the rest of the time is used for reading values from the register file and assigning them to the relevant parameters. In the initial pipeline model the start of the ID-stage was stalled until the previous instruction produced a result (i.e. after the WB-stage has finished). Alternatively, the implemented five-stage pipeline models stall for a shorter time and start with the ID-stage prematurely so that the instruction-type may be decoded. Meanwhile the previous instruction has produced its result and the ID-stage can continue. This improves execution time for the implemented model compared to the initial model.

Three kinds of forwarding path (EXE-EXE, MEM-EXE and MEM-MEM forwarding) are described in the following sections. To implement a forwarding mechanism in an asynchronous design is trickier than in a synchronous design. Each pipeline stage can start at a different time and can finish independently, so without a buffer the stage at the beginning of the forwarding path has to wait until the stage at the end of the forwarding path is ready to receive the data. This

will reduce the performance of the asynchronous pipelined model by introducing additional synchronisation points. The solution is to store the forwarded data into a buffer and let the stage at the beginning of the forwarding path continue its work. In this project the pipeline always forwards the data as it is impossible to predict whether the later instructions need the forwarded data or not. Therefore the instruction in the stage at the end of the forwarding path has to throw the unnecessary forwarded data away if it does not need it.

5.3.1 EXE-EXE Forwarding

As shown in Figure 5.6, the ADD instruction could use the result from the previous instruction (SUB) when it really needs it (at the beginning of the EXE stage) right away after that data is valid which is at the end of the EXE stage of the previous instruction. Hence the simplest type of forwarding was implemented here, which is forwarding the valid result of the earlier instruction to where that result is needed. Figure 5.7 illustrates the effect of using EXE-EXE forwarding to resolve the data hazard. The EXE-EXE forwarding is shown in Figure 5.8 as an arc-arrow within the box because it was implemented by using a local variable. As this path is contained within a single pipeline stage it can be implemented without any synchronisation overhead.

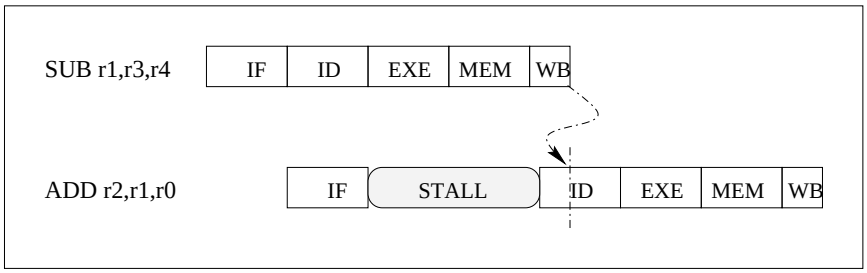

Figure 5.6: The diagram of the simplest data hazard

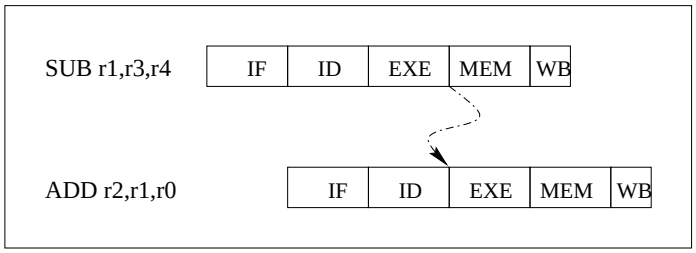

Figure 5.7: Using EXE-EXE forwarding to resolve data hazard

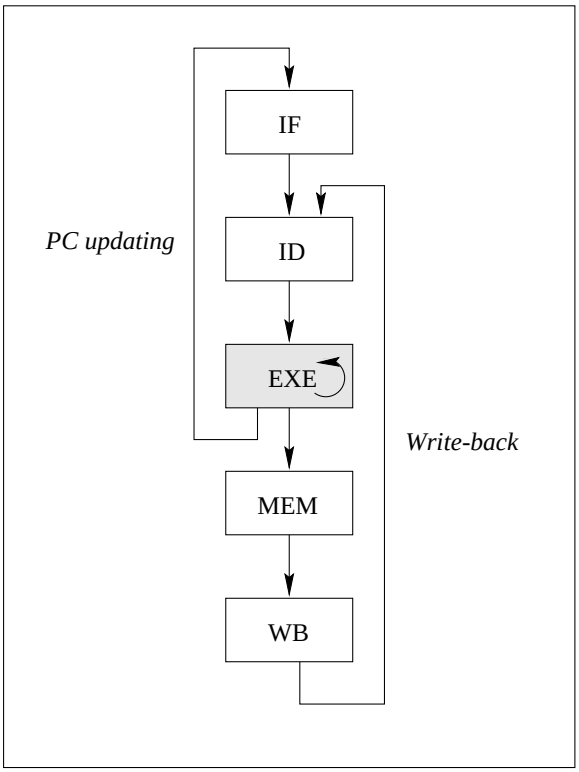

Figure 5.8: The 5-stage pipeline with EXE-EXE forwarding

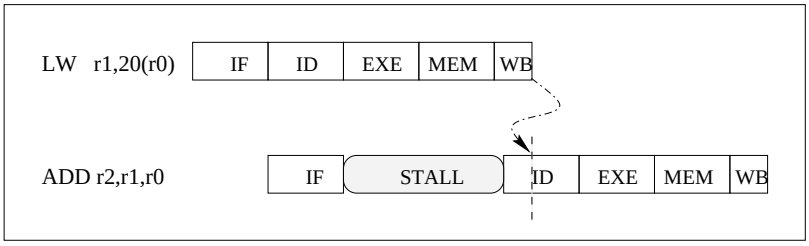

Figure 5.9: The diagram of the Load hazard

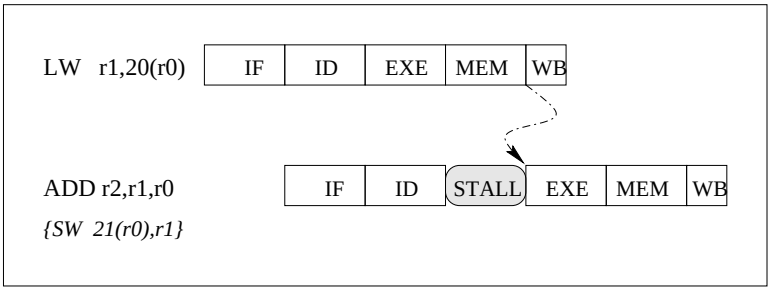

Figure 5.10: Using MEM-EXE forwarding to resolve a Load hazard

5.3.2 MEM-EXE Forwarding

Unfortunately not all instructions can take advantage of the EXE-EXE forwarding, since the result from Loads is valid at the end of the MEM stage as shown in Figure 5.9. Another forwarding path (MEM-EXE) has to be implemented, with stalling, to speed up the pipeline by forwarding the valid result of Loads from the MEM stage to the EXE stage where the next instruction may need that data. Figure 5.10 illustrates how to reduce Load hazards using MEM-EXE forwarding. The structure of the model with MEM-EXE forwarding is shown in Figure 5.11.

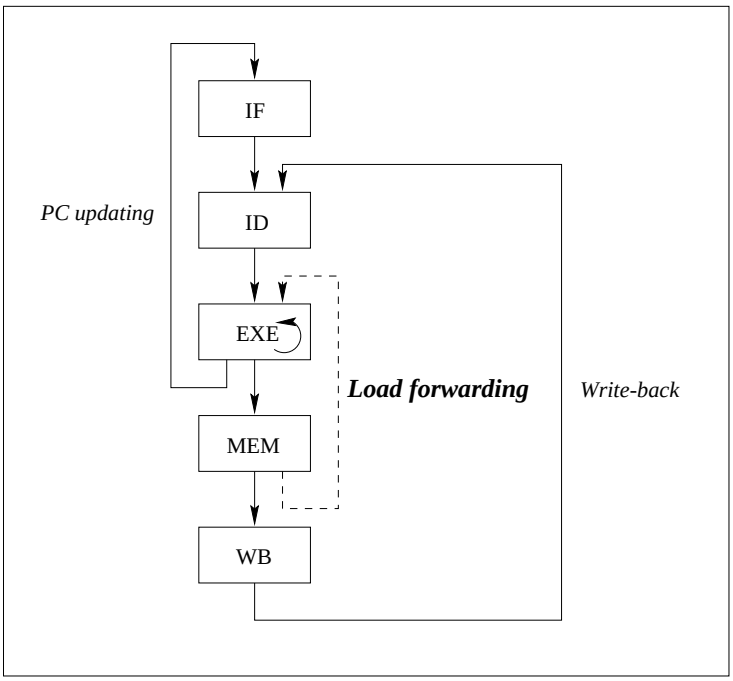

Figure 5.11: The 5-stage pipeline with MEM-EXE forwarding

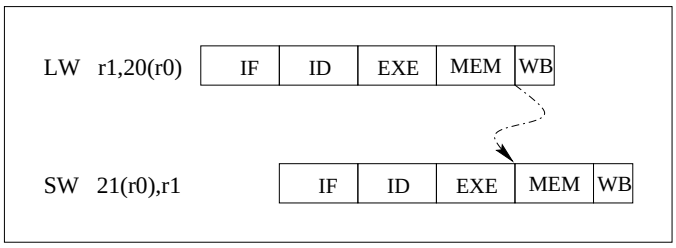

Figure 5.12: Using MEM-MEM forwarding to solve the Store-Load hazard

5.3.3 MEM-MEM Forwarding

However, according to Figure 5.12, when a Load instruction is followed by a Store instruction, it is not necessary to stall the Store since Store instructions need the result from the Load at the beginning of the MEM stage. Finally, Figure 5.13 illustrates the structure of the five-stage pipeline with MEM-MEM forwarding. Similar to EXE-EXE forwarding, MEM-MEM forwarding is shown as an arc-arrow within the box because it was implemented by using a local variable.

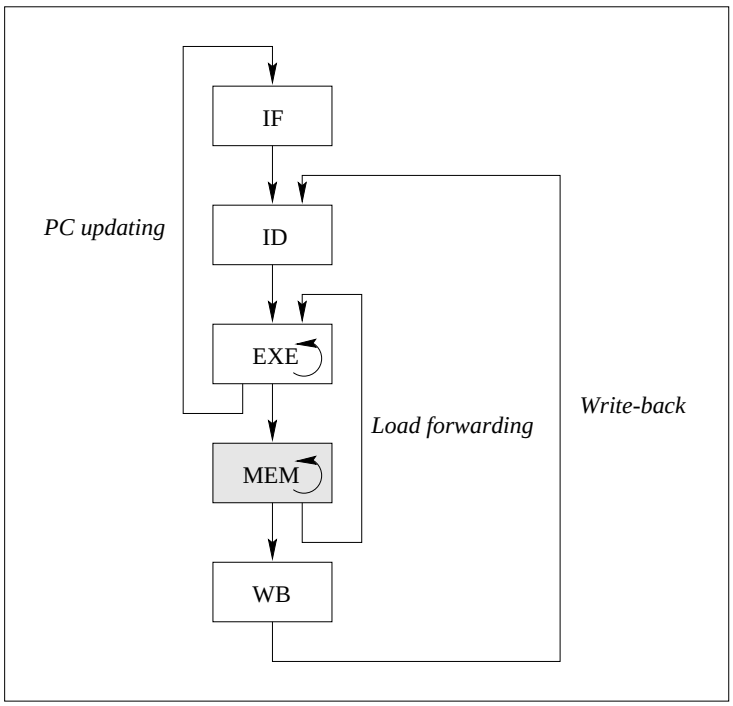

Figure 5.13: The 5-stage pipeline with MEM-MEM forwarding

5.4 Timing Information

These models were all designed subject to the same timing criteria as follows:

- Prefetching each instruction takes 8 time units in the LARD language.
- Decoding the instruction takes 8 time units.
- To execute (excluding memory access and write-back) the validated instruction takes 8 units.
- To Access memory for Loads and Stores takes 8 time units.
- Writing back to update the register file (which happens for every prefetched instruction in the pipelined models, or only for instructions that need to write the result back in the non-pipelined model) takes 1 time unit.

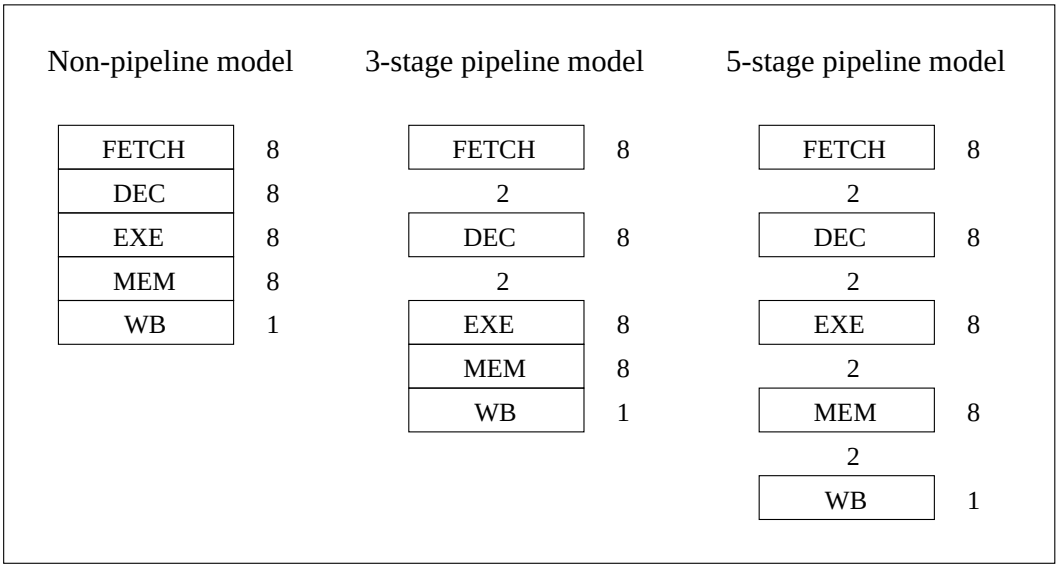

Figure 5.14: Time information for the initial models

- Each channel between two different stages takes 2 time units.

Figure 5.14 illustrates the timing information for the initial models in this project. According to this, the cycle time in the non-pipelined model is around 25 (without memory access) to 33 (with memory access) time units. The cycle time in the three-stage pipeline model is approximately 10 time units + extra time for memory access. In the five-stage pipeline model the cycle time is around 10 time units with no extra cycles for memory access.

5.5 Implementation using LARD

The models outlined above were described using the LARD language. Each box (IF, ID, EXE, MEM, WB) was modelled as a LARD process, with data passing between boxes through a LARD communication channel.

In LARD channel communication values are sent to a channel using the ! operator. The syntax for sending is *channel ! value*, which means:

- Wait until the receiver is ready to accept a communication.
- Send the value to the receiver.
- Wait until the receiver has finished with the value.

Values are received using the ? operator. The syntax for receiving is *channel ? expression*, which means:

- Wait until the sender is ready to initiate a communication.
- Evaluate the body expression.
- Indicate to the sender that the value is finished with.

Within the body expression the received value can be read using the expression *? channel*.

The effective pipeline depth of the model can be modified by changing the 'Acknowledge' phase of the communication:

- If the block processing and output communication is enclosed within the input channel communication, the block operates within the same pipeline stage as its predecessor.

- If the block copies the input value and acknowledges immediately, the block operates as a separate pipeline stage from its predecessor.

Hence a single LARD program can, with relatively little modification, model the behaviour of the different pipeline structures described above.

5.6 Summary

Processor designs have been developed based on the theory presented in the earlier chapters. Three models were created in order to compare their performance. All of the designs are asynchronous and two of them use pipeline structures. This is especially important for proving the pipeline to be superior to a non-pipelined design. Furthermore, both a three-stage and a five-stage pipeline were created to test the notion that the five-stage pipeline is better balanced in terms of block processing time and would therefore be superior to the three-stage design. The tests and their results will be discussed in the next chapter.

Chapter 6

Testing and Evaluation

The testing procedure was divided into three sections written in a different programming language. The first test used three very simple assembly programs to understand each model and to attempt to improve the performance of the five-stage pipeline. The second test used small programs written in the high-level programming language, C. The third test used the Dhrystone benchmark C program. Then Section 6.4 evaluates and compares the results from the different models.

6.1 Simple Assembly Test Programs

It is obvious that a pipelined model should be faster than non-pipelined model since it can operate in parallel. Since there are more pipe stages in a five-stage pipeline, and most of the pipe stages take a similar time, a five-stage model is expected to be faster than a three-stage pipeline. The latter has one stage - the

EXE-stage - which takes much longer than the other two, causing following instructions to be stalled. This line of reasoning will hold most of the time, although in some cases a three-stage pipeline is faster due to the extra time needed for the added stages to communicate in a five-stage pipeline (EXE-MEM, MEM-WB).

For the implementation of the five-stage pipeline, very simple programs were written in Assembly Programming Language with different kinds of data dependencies, in order to compare each model specifically.

6.1.1 Asm-1: Test Program

With the Assembly code for the DLX architecture shown below, this test gives the performance obtained from the initial five-stage pipeline and the one with EXE-EXE forwarding.

```
SUB     r1, r2, r3
ADD     r4, r1, r5
```

From the code, the `ADD` needs the result , **r1**, from the `SUB`. In both initial three-stage and five-stage pipelines, the `ADD` has to stall and wait until the result from the `SUB` is written back. By incorporating the EXE-EXE forwarding in the latter model, the `ADD` can get that value immediately after it is written in the EXE stage. This reduces the cost. Figure 6.1 shows the time views supplied by the LARD simulation system.

Note: The traces in LARD time view can be interpreted as follows:

- The upper half bars of the trace represent the sender's activity. The start of the bar indicates the point where the sender was ready to start a communication. The end of the bar indicates where the communication finished.
- The lower half bars of the trace represent the receiver's activity. The start of the bar indicates the point where the receiver became ready to accept a communication. The end of the bar indicates where the communication finished.

In case of a trace containing a lot of upper bars, it indicates a system where a sender is blocked by a relatively slow receiver. On the other hand, a trace containing a lot of lower bars indicates a system where a receiver is waiting for a relatively slow sender.

When observing these time views, the model with the EXE-EXE forwarding is faster than the normal one because it saves the time required to pass the result through the MEM stage and write back the result to the register file.

6.1.2 Asm-2: Test Program

```
LW      r1, 0(r3)
ADD     r4, r1, r5
```

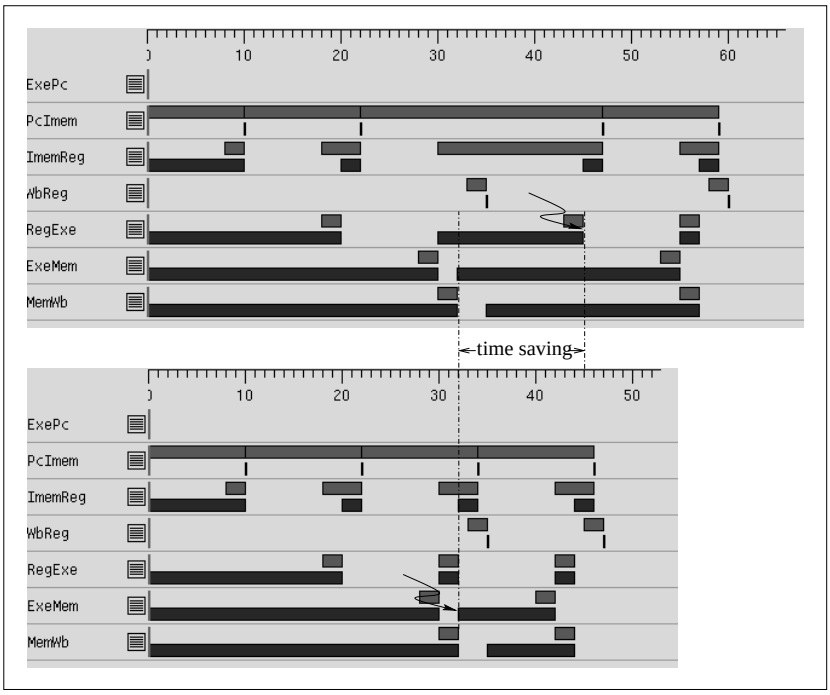

Figure 6.1: The effect of EXE-EXE forwarding

Since the instruction following the Load instruction needs the result from the Load, a data hazard has been introduced in this code. Unlike the *Asm-1* program, the dependency occurs in the MEM stage in which the result from `LW` is produced. In order to get the result right away, thereby reducing stall time, another forwarding technique was applied here. Figure 6.2 illustrates the reduced stall time after applying the MEM-EXE forwarding to pass the result.

6.1.3 Asm-3: Test Program

```
LW      r1, 0(r3)
SW      40(r2), r1
```

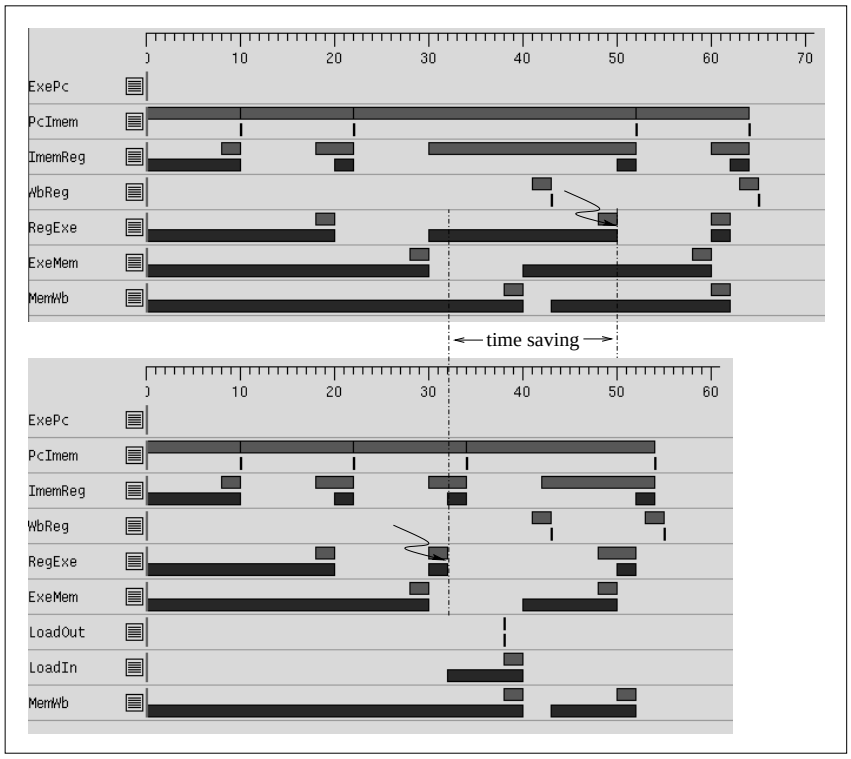

Figure 6.2: The effect of MEM-EXE forwarding

This is a special case, similar to the previous example, which occurs when the following instruction after Load is a Store which needs the result from the Load at the beginning of the MEM stage. By adding MEM-MEM forwarding, the stall time in the earlier model can be eliminated (or at least reduced). Figure 6.3 shows the effect of this forwarding.

In all of these models forwarding saves time overall even though generally forwarding itself takes time as well. (Although, in fact EXE-EXE and MEM-MEM forwarding do not consume any extra time since these are internal operations within the EXE-block and MEM-block consecutively.) Since it is impossible to

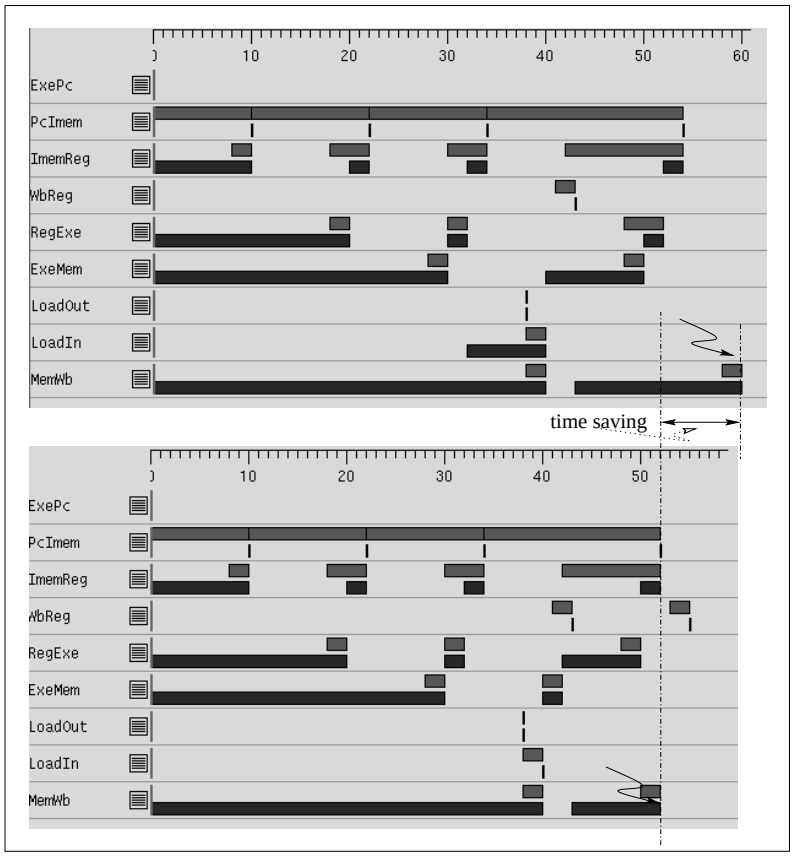

Figure 6.3: The effect of MEM-MEM forwarding

Program Name	Assembly Inst Count	Description
Bubble.c	2958	Bubble sorts for 10 items from minimum to maximum.
Fibonacci.c	3565	Calculates the 10^{th} Fibonacci number.
StrRev.c	1731	Convert a string "Test Reverse String Function" into reverse word order and print it out.

Table 6.1: The description of C test programs

predict whether the following instruction needs the result of the preceding instruction or not, the forwarding schemes in each model always pass the result of the current instruction.

6.2 Simple C Test Programs

A number of C test programs were used to measure the performance of each model. These programs were compiled with GCC [1] and assembled with a DLX assembler. Details about the GCC and the assembler can be found in Appendix C. Due to the incompatibility between the GCC compiler and DLX assembler, it was very difficult to compile the test programs from C programs.

A brief description of the test models is given in Table 6.1. Table 6.2 shows the results for each model of the C test programs. The time of operation was normalised by setting the three-stage model to unity. As can be seen, the use of any forwarding technique improves the operating speed greatly (13.5% to 20.8%).

[1] A C compiler

Model Name	Normalised Time		
	Bubble	Fibonacci	StrRev
Initial Models			
Non-pipelined Model	1.60	1.59	1.54
3-stage Pipeline Model	1.00	1.00	1.00
5-stage Pipeline Model	0.99	0.87	1.03
Modified Models of 5-stage Pipeline			
a) with EXE-EXE Forwarding	0.84	0.78	0.87
b) a plus MEM-EXE Forwarding	0.75	0.75	0.81
c) b plus MEM-MEM Forwarding	0.73	0.75	0.81

Table 6.2: The comparative results with C test programs

Model Name	Loop Time (time units)	Normalised Time
Initial Models		
Non-pipelined Model	59447	1.53
3-stage Pipeline Model	38746	1.00
5-stage Pipeline Model	36875	0.95
Modified Models of 5-stage Pipeline		
a) with EXE-EXE Forwarding	31803	0.82
b) a plus MEM-EXE Forwarding	30119	0.78
c) b plus MEM-MEM Forwarding	29891	0.77

Table 6.3: The results with the Dhrystone program

6.3 The Dhrystone Program

As well as testing the models in this project with small Assembly and C Programming Languages, the "DHRYSTONE" Benchmark Program (more details are given in Appendix B) was used to evaluate the performance of each model. Dhrystone is probably the smallest program that can reasonable be claimed to reflect the performance of large C applications realistically. The results are presented in Table 6.3. Here, as before, the modified models which made use of forwarding prove to be superior to the initial model.

6.4 Evaluation

According to Section 5.4, the cycle time for the non-pipelined, the three-stage pipeline and the five-stage pipeline models are 25 to 33, 10 + extra time for memory access and 10 time units respectively. The CPI of the non-pipelined model equals to 1. For the three-stage pipeline model the CPI can be calculated comparative to the CPI of the non-pipelined model as below:

$$CPI = \frac{(25 \quad to \quad 33)}{10 * 1.55} = 1.6 \quad to \quad 2$$

This CPI number (excluding the extra time for memory access) is similar to the 2 clocks per instruction of *ARM7* [ARM98a], a synchronous three-stage pipeline microprocessor. Similarly, the CPI of the five-stage pipeline model can be calculated as follows:

$$CPI = \frac{(25 \quad to \quad 33)}{10 * 2} = 1.25 \quad to \quad 1.65$$

The CPI number is close to the 1.5 value of *ARM9* [ARM98b], which is a synchronous five-stage pipeline microprocessor.

The asynchronous pipelined design is much faster than the non-pipelined version which operates sequentially. To take advantage of the pipeline design, it is important to ensure that each pipe stage is balanced. This makes the five-stage pipeline faster than the three-stage pipeline. A little overhead can be attributed to the five-stage pipeline limiting the maximum throughput, since in some cases

this pipeline has to waste time to pass data from the EXE stage through the MEM stage in order to keep the pipeline simple to design. This can be seen when comparing the results of the three-stage and five-stage pipelining models for the *StrRev.c* program. A higher performance pipeline design would be able to reduce these times.

To improve the performance of the five-stage pipeline, some modified models were implemented to reduce the pipe hazards by adding some mechanisms to support forwarding. There are several kinds of forwarding. Each of these resolves a different kind of data dependency and has a different cost to implement.

The results show that the EXE-EXE forwarding path is the most cost-effective because it is simple to implement and is applicable to the most frequently encountered data dependencies. The second most cost-effective forwarding mechanism is the MEM-EXE forwarding. MEM-MEM forwarding is used infrequently and provides little performance improvement when used.

However, even if in some cases the forwarding seems to be useless (when the forwarded results are not used by any instruction), it could be assumed from the test results that the forwarding mechanism is very useful to incorporate in the design. If unnecessary forwarding could be eliminated or at least reduced, then the pipelined model with forwarding schemes would become more competitive

than the others. To obtain the optimal performance, the modeller must trade-off which mechanisms are most cost-effective in the target application.

In summary, although one of the results from this project shows the asynchronous five-stage pipeline design to be slower than the three-stage approach, improvements to this design are likely to reduce this gap. Together, these suggest that future asynchronous five-stage implementations could be competitive with three-stage designs, although they are slightly more complicated to implement. However, performance is not the only issue which determines the value of a design style. Another issue is the ease of design. Although it is currently hard to predict the performance of each forwarding design because it depends on the 'hazard density' (i.e. the average number of times a hazard is likely to occur), omitting forwarding altogether seems to yield a much worse performance. The decision of choosing or rejecting any particular mechanism rests with the designer.

6.5 Summary

Several different pipelined processor designs have been tested. In order to gain a better insight into their operation, the designs were first tested with simple programs write in assembly language. After modification, they were tested with programs written in C which would be more relevant to their future use. The modified designs make use of forwarding techniques which have proven to be beneficial for the operating speeds.

Chapter 7

Conclusions

Even though asynchronous logic design has been resurrected only a few years ago, it can potentially perform its task as well as or even better than synchronous systems, and it is continually moving forward.

This project has shown how an asynchronous pipelined processor can be implemented in both three-stage and five-stage pipelines. An analysis of the performance of several different designs has been presented, and an attempt made to improve the throughput for the asynchronous five-stage pipelined approach.

7.1 Assessment of Work

Due to the simplicity of the DLX architecture, both asynchronous non-pipelined and pipelined (three-stage and five-stage) designs were successfully implemented

and analysed. This simplicity contributed to the work by restricting the complexity of the instructions, and should ease further attempts to improve the performance of the design.

Having channel communication and high-level programming features, LARD was found to be a suitable language for modelling both the asynchronous non-pipelined design and even more so for the asynchronous pipelined design at an abstract level. It is not necessary to model the request and acknowledge signals used in inter-block communication explicitly as it is in other programming languages. Although LARD is harder to debug than, for example, C or VHDL owing to the channel communication, it offers excellent tools for maintenance and viewing results.

In conclusion, the results certainly suggest that the improvement of a simple asynchronous five-stage pipeline design is not only practical, but also positively beneficial to performance. It points to possibilities for future asynchronous pipelining processor design.

7.2 Suggestions for Further Work

Four issues remain to be resolved to make these models more realistic and to improve the performance of an asynchronous five-stage pipeline approach.

- First of all, more realistic time information should be applied to obtain more accurate result.

- Secondly for future improvement to these models of the DLX architecture, compatible tools (at least the standard version according to [HP96] of DLX assembler) are required in order to avoid tedious work in adjusting the output code of the compiler to suit the DLX assembler. Otherwise another well-supported architecture should be considered. The MIPS architecture, which is similar to DLX, is another choice for implementation because many practical tools for MIPS already exist.

- Implementation of the floating-point instruction set should be included to make these models more complete.

- Last, but not least, other mechanisms should be identified to solve pipeline dependencies in order to allow the pipeline to achieve optimal performance. These could include other kinds of forwarding and branch prediction as discussed in Chapter 4.

Bibliography

[ARM98a] Arm7tdmi-s datasheet. (ARM DDI 0084C), July 1998. Advanced RISC Machines Ltd (ARM).

[ARM98b] Arm9tdmi datasheet. (ARM DDI 0091A-02), January 1998. Advanced RISC Machines Ltd (ARM).

[Dev94] Ian V. Devereux. Synchronous and asynchronous processor design -a comparative study-. Master's thesis, Computer Science Dept. The University of Manchester, 1994.

[Eng96] I. Englander. *The Architecture of ComputerHardware Systems Software: An Information Technology Approach.* John Wiley & Sons, Inc., 1996.

[Fur96] S. B. Furber. Asynchronous logic. IberChip, Sao Paulo, Brazil, February 1996.

[GG97] D. A. Gilbert and J. D. Garside. A result forwarding mechanism for asynchronous pipelined systems. In *IEEE Computer Society Press*, pages 2–11, April 1997.

[Gil97] D.A. Gilbert. *Dependency And Exception Handling In An Asynchronous Microprocessor.* PhD thesis, Computer Science Dept. The University of Manchester, 1997.

[GM93] J. Goodman and K. Miller. *A Programmer's View of Computer Architecture: with assembly language examples from the MIPS RISC architecture.* Sauders College Publishing Inc., 1993.

[Hau93] S. Hauck. Asynchronous design methodologies: An overview. (UW-CSE-93-05-07), April 1993.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[HP96] J.L. Hennessy and D.A Patterson. *Computer Architecture: A Quantative Approach.* Morgan Kaufmann, 1996.

[Kan89] G. Kane. *MIPS RISC Architecture.* Prentice Hall, 1989.

[Kog81] P.M. Kogge. *The Architecture of Pipelined Computers.* Hemisphere Publishing Corporation, 1981.

[Pav94] N.C. Paver. *The Design and Implementation of an Asynchronous Microprocessor.* PhD thesis, Computer Science Dept. The University of Manchester, 1994.

[PDF+92] N.C. Paver, P. Day, S.B. Furber, J.D. Garside, and J.V. Woods. Register locking in an asynchronous microprocessor. In *1992 IEEE International Conference on Computer Design: VLSI in Computers & Processors*, October 1992.

[PH97] D.A. Patterson and J.L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, 1997.

[Sut89] I.E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

Appendix A

The DLX Architecture

A.1 Introduction

The DLX (pronounced "deluxe") architecture is a simplified version of the MIPS R3000 processor, as described in the text book *Computer Architecture, A Quantitative Approach* by Hennessy and Patterson as an instruction tool. The architecture of DLX was chosen based on observations of frequently used instructions from compiled programs. DLX provides a good architectural model for study, not only because of the recent popularity of this type of machine, but also because it is easy to understand.

DLX emphasises a simple load-store instruction set, design for pipelining efficiency, an easily decoded instruction set and efficiency as a compiler target. The complete version of the DLX architecture includes a floating-point instruction set.

However, in order to simplify the model the floating point was omitted in this project. This appendix describes the DLX architecture as it was implemented.

A.2 DLX Overview

The DLX architecture, which is used in this implementation, has thirty-two 32-bit general-purpose registers (GPRs), named R0 through R31. The value of R0 is always equal to zero.

The data types are 8-bit bytes, 16-bit half words and 32-bit words for integer data. The DLX operations work on 32-bit integers. Bytes and half words are loaded into registers with either zeros or the sign bit replicated to fill the 32 bits of the registers, depending on the opcode. Once loaded, they are operated on with the 32-bit integer operations. The only data addressing modes are immediate and displacement, both with 16-bit fields. The DLX memory is byte addressable in Big Endian mode with a 32-bit address. As it is a load-store architecture, all memory references are through loads or stores between memory and the GPRs. Supporting the data types mentioned above, memory accesses involving the GPRs can be to a byte, to a half word or to a word. All memory accesses must be aligned.

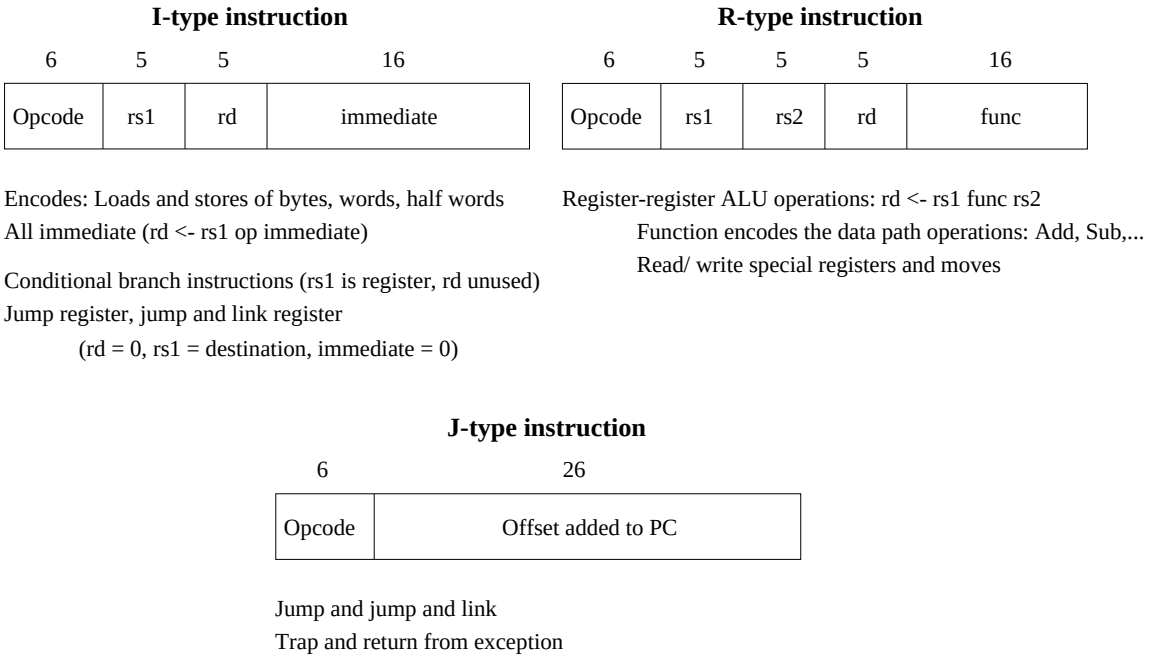

Figure A.1: Instruction formats for DLX

A.3 DLX Instruction Format

To meet the requirements for making the machine easy to pipeline and decode, all instructions are 32 bits with a 6-bit primary opcode. Figure A.1 shows the instruction layout. All instructions are encoded in one of three formats, the I, J or R-type instruction format. These formats are simple while providing 16-bit fields for displacement addressing, immediate constants or PC-relative branch addresses.

Instruction	Instruction name	Meaning
LW R1, 30(R2)	Load word	Regs[R1] $\leftarrow_{32}$ Mem[30+Regs[R2]]
LB R1, 30(R2)	Load byte	Regs[R1] $\leftarrow_{32}$ (Mem[30+Regs[R2]]$_0$)^{24} ## Mem[30+Regs[R2]]
LBU R1, 30(R2)	Load byte unsigned	Regs[R1] $\leftarrow_{32}$ 0^{24} ## Mem[30+Regs[R2]]
LH R1, 30(R2)	Load half word	Regs[R1] $\leftarrow_{32}$ (Mem[30+Regs[R2]]0)16 ## Mem[30+Regs[R2]] ## Mem[31+Regs[R2]]
SW R3, 500(R4)	Store word	Mem[500+Regs[R4]] $\leftarrow_{32}$ Regs[R3]
SB R3, 500(R4)	Store byte	Mem[500+Regs[R4]] $\leftarrow_{16}$ Regs[R3]$_{16..31}$
SH R3, 500(R4)	Store half word	Mem[500+Regs[R4]] $\leftarrow_{8}$ Regs[R3]$_{24..31}$

Table A.1: Examples of load and store instructions on DLX

A.4 DLX Operations

Apart from floating-point operations, there are three other broad classes of instructions: loads and stores, ALU operations and branches and jumps.

A.4.1 Load and store instructions

Any of the GPRs may be loaded or stored except that loading R0 has no effect. All load-store instructions use a single addressing mode and require that the memory value be aligned. Of course, both loads and stores are available for all data types shown. Table A.1 gives some examples of the load and store instructions.

Note :

- The statement $\leftarrow_n$ means transfer an n-bit quantity.
- A subscript is used to indicate selection of a bit or subrange from a field. Bits are labelled from the most significant bit starting at 0.
- Superscript is used to replicate a field.
- The statement ## is used to concatenate two fields.

Instruction	Instruction name	Meaning
ADD R1,R2,R3	Add	Regs[R1]←Regs[R2]+Reg[R3]
ADD R1,R2,#3	Add immediate	Regs[R1]←Regs[R2]+3
LHI R1,#42	Load high immediate	Regs[R1]←42##0^{16}
SLLI R1,R2,#5	Shift left logical immediate	Regs[R1]←Regs[R2]<<5
SLT R1,R2,R3	Set less than	if (Regs[R2]<Reg[R3]) then Regs[R1]←1 else Regs[R1]←0

Table A.2: Examples of arithmetic/logical instructions in DLX.

A.4.2 ALU operations

All ALU operations are register-register instructions. The operations include simple arithmetic operations (ADD and SUB) and logical operations (AND, OR, XOR and shifts). Immediate forms for all these instructions, with a 16-bit sign-extended immediate, are provided. The operation LHI (load high immediate) loads the top half of a register, while setting the lower half to zero. There are compare instructions, which compare two registers (=,!=,<,>,<=,>=). If the condition is true, these instructions place a 1 (true) in the destination register; otherwise they place the value 0 (false). There are also immediate forms of these compares.

Table A.2 gives some examples of arithmetic/logical instructions with and without immediate values.

Instruction	Instruction name	Meaning
J name	Jump	PC←(PC+4) + name
JAL name	Jump and link	Regs[R31]←PC+4 ; PC←(PC+4) + name
JR R5	Jump register	PC←Regs[R5]
JALR R5	Jump and link register	Regs[R31]←PC+4 ; PC←Regs[R5]
BEQZ R4, name	Branch equal zero	if (Regs[R4] == 0) then PC←(PC+4) + name
Bnez R4, name	Branch not equal zero	if (Regs[R4]!= 0) then PC←(PC+4) + name

Table A.3: Examples of typical control-flow instructions in DLX.

A.4.3 Branches and Jumps

The four jump instructions are differentiated in the way they specify the destination address and by whether or not a link, used for procedure calls, is made. All branches are conditional. The condition is specified by the instruction, which may test the register source for zero or non-zero. All control instructions, except jumps to an address in a register, are PC-relative. Table A.3 gives examples of typical control-flow instructions in DLX. Table A.4 contains a list of all DLX operations and their meanings that were implemented into the models of this project.

A.5 DLX Opcodes

These opcodes were taken from the DLXsim [1]. Table A.5 shows the opcode numbers used for the DLX instructions. Register-register instructions have the special opcode, and the instruction is specified in the lower six bits of the instruction

[1] A simulator for DLX. More detail in Appendix C

word. (Similarly, floating point instructions have the FPARITH opcode).

Instruction type/opcode	Instruction meaning
Data transfers	**Move data between registers and memory; only memory address mode is 16-bit displacement + contents of a GPR**
LB, LBU ,SB	Load byte, load byte unsigned ,store byte
LH, LHU, SH	Load half word, load half word unsigned , store half word
LW, SW	Load word, load word unsigned ,store word
Arithmetic/logical	**Operations on integer or logical data in GPRs; signed arithmetic trap on overflow**
ADD, ADDI, ADDU, ADDUI	Add, add immediate (all immediates are 16 bits); signed and unsigned
SUB, SUBI, SUBU, SUBUI	Subtract, subtract immediate; signed and unsigned
AND, ANDI	And, and immediate
OR, ORI, XOR, XORI	Or, or immediate, exclusive or, exclusive or immediate
LHI	Load high immediate - load upper half of register with immediate
SLL, SRL, SRA, SLLI,	Shift both immediate (S--I) and variable form (S---);
SRLI, SRAI	shifts are shift left logical, right logical, right arithmetic
S--, S--I	Set conditional: "--'' may be LT, GT, LE, GE, EQ, NE
Control	**Conditional branches and jumps; PC-relative or through register**
BEQZ, BNEZ	Branch GPR equal/not equal to zero; 16-bit offset from PC+4
J, JR	Jumps; 26-bit offset from PC+4 (J) or target in register (JR)
JAL, JALR	Jump and link; save PC+4 in R31, target is PC-relative (JAL) or a register (JALR)

Table A.4: All implemented DLX instructions

MainOp	$00	$01	$02	$03	$04	$05	$06	$07
$00	SPECIAL	FPARITH	J	JAL	BEQZ	BNEZ	bfpt	bfpf
$08	ADDI	ADDUI	SUBI	SUBUI	ANDI	ORI	XORI	LHI
$10	rfe	TRAP	JR	JALR	SLLI		SRLI	SRAI
$18	SEQI	SNEI	SLTI	SGTI	SLEI	SGEI		
$20	LB	LH		LW	LBU	LHU	lf	ld
$28	SB	SH		SW			sf	sd

SPECIAL	$00	$01	$02	$03	$04	$05	$06	$07
$00					SLL		SRL	SRA
$08					TRAP			
$10								
$18								
$20	ADD	ADDU	SUB	SUBU	AND	OR	XOR	
$28	SEQ	SNE	SLT	SGT	SLE	SGE		
$30	movi2s	movs2i	movf	movd	movfp2i	movi2fp		

Table A.5: Opcodes of the DLX architecture

Appendix B

The Dhrystone: Benchmark Program

In general, the "DHRYSTONE" Benchmark Program is used to measure the performance of a processor. The Dhrystone that was used in this project is version 2.1 (May 25, 1988), implemented by Reinhold P. Weicker. This benchmark program is the smallest reasonable program used for testing the performance of a processor. Table B.1 presents the structure of the dynamic Assembly instructions for one execution of the Dhrystone loop.

Instruction	Number
ALU Instruction	**1207**
OR	1
SLEI	2
SLT	2
SGTI	3
SNE	3
SEQI	5
SLE	8
SUB	10
TRAP	17
SLLI	18
SUBUI	20
SEQ	22
ADDUI	44
SNEI	118
ANDI	155
ADDI	251
ADD	528
Memory Instruction	**1061**
LBU	20
LHI	44
SB	111
LB	141
SW	294
LW	451
Branch Instruction	**351**
BEQZ	11
JAL	26
JR	26
J	138
BNEZ	150
Total	**2619**

Table B.1: Numbers of Instructions in Dhrystone

Appendix C

Detail about Tools

LARD was developed by Phil B. Endecott of the AMULET group. Its documentation home page is available at: http://*www.cs.man.ac.uk/amulet/projects/lard/*.

GCC compiler, GNU C compiler, version 2.7.x, developed at the University of Minnesota, which is used in this project, forms an ANSI C compiler for the DLX architecture. It is available by FTP to go with [HP96]; http://*www-mount.ee.umn.edu/mcerg/software.html*

The assembler for the DLX architecture used in this project is available by FTP from

ftp.lip6.fr/lip6.softs/alliance.

DLXsim was developed at the University of Illinois. It is available by FTP to go

with [HP96]; http://*galileo.dpi.inpe.br/pub/dlxvsim*