THE UNIVERSITY OF CALGARY

AMULET1: Specification and Verification in CCS

by

Ying Liu

A DISSERTATION
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA
SEPTEMBER, 1995

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

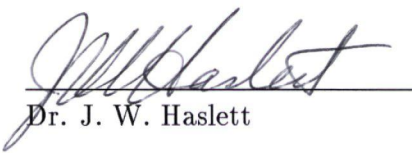
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a dissertation entitled, "AMULET1: Specification and Verification in CCS" submitted by Ying Liu in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Dr. J. Kendall, Supervisor & Chairman

Department of Computer Science

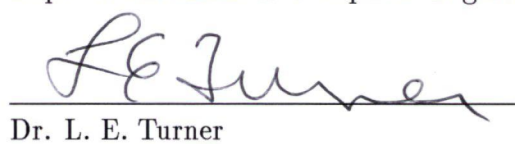
Dr. G. Birtwistle, Co-supervisor

Department of Computer Science



Dr. J. W. Haslett

Dept. of Electrical & Computer Engineering



Dr. L. E. Turner

Dept. of Electrical & Computer Engineering

Dr. D. Edwards, External Examiner

Manchester University

Date Sept. 11th 1995.

Abstract

There has been a dramatic resurgence of asynchronous hardware designs in recent years. Since state spaces multiply, asynchronous systems can be extremely state rich and it is correspondingly difficult to reason about their characteristic properties with the traditional techniques of simulation. The alternative, adopted here, is to use formal techniques of specification and property checking to reason about asynchronous designs and to test them thoroughly for desired behaviours.

This dissertation explores the feasibility of applying the CCS process algebra and its supporting tool, the Edinburgh Concurrency Workbench (CWB), to the formal specification and verification of a specific asynchronous microprocessor (AMULET1) developed by the AMULET group at Manchester University. AMULET1 is modeled at both the system level and the floor plan level. The system level model shows how each instruction class flows through the major components of the processor, and can be used outwards in modeling embedded applications and inwards to document the roles of the floor plan elements. The subsystem level modeling presents register transfer level detail of the floor plan elements and serves as an implementation guide to designers.

This is the first attempt to apply formal techniques to a full scale, practical, industrial-strength asynchronous design. CCS is demonstrated to be an appropriate and efficient notation for modeling such complex designs. In the main, property checking on the CWB proved to be a reliable and robust way of detecting and repairing specification flaws. Although this work is *post facto*, it suggests that the systematic incorporation of formal specification and verification techniques into the design cycle could shorten the overall design window and help reduce fabrication iterations.

Acknowledgments

This dissertation research was carried out in conjunction with Dr. Birtwistle as supervisor, and with the full cooperation of the Manchester AMULET team. Dr. Birtwistle not only initiated and supervised the dissertation work, but also actively participated throughout the specification iterations. It has been a great opportunity for me to learn from him and a great pleasure to work with him. Without his encouragement and support, this dissertation would not have been possible.

The Manchester AMULET team were extremely generous in explaining the architecture, reading, criticizing, and pointing out flaws as the various levels of abstraction were investigated and the specifications took shape. Dr. Jim Garside and Dr. Nigel Paver require particular thanks for their ever-generous help with an overall understanding of the entire architecture, as well as intuition and detail at every level.

Many thanks to my graduate committee, Dr. John Kendall, Dr. Jim Haslett, Dr. Laurence Turner, and Dr. Douglas Edwards, for their contributions and time.

Thanks also to John Aldwinckle, Bruce MacDonald, Steve Franks, Faron Moller, Carmen Rata, Camille Sinanan, Chris Tofts, Charles Tuckey, Dave Spooner, Ken Stevens, and Barry Yee, who were wonderfully supportive friends and colleagues.

This research could not have been completed without financial support from the Alberta Microelectronics Centre, the Killam Scholarship Committee, and the University of Calgary.

I am grateful to my parents and my brother for their long lasting support.

Finally, special thanks to Bo, for his thoughtfulness and patience with me all the time.

Dedication

To My Mom & Dad

Contents

List of Tables

List of Figures

Chapter 1
Introduction

There are two major circuit design styles, namely synchronous and asynchronous. Most of today's digital systems adopt the synchronous approach in which the correct operation of a system is governed by a global clock signal. However with the rapid advancement of VLSI technology, synchronous designers are finding it increasingly difficult to distribute clock signals and maintain functionality as more circuitry is packed onto a chip. The clocking system of the DEC Alpha chip [Dea92b] demonstrates the point that synchronous design might be beginning to approach its limits.

With potential advantages in distributed synchronization, composability and power consumption, the asynchronous circuit design style is attracting renewed interest after being neglected for over two decades. The resurgence of the asynchronous approach has produced several successful designs including asynchronous controllers [DCS93, MCS94, NDDH92], asynchronous datapaths [WH91, Gar93], and asynchronous processors [MBL+89, Dea92a, Pav94, Fur95]. Accordingly, the analysis and verification of asynchronous circuits is becoming more and more important as designers seek ways to shorten the overall design interval and reduce the probability of fabrication iterations.

It is very difficult to use the traditional technique of simulation to cope with the verification of asynchronous systems. In particular, locating deadlocks (to which asynchronous systems are prone) is extremely difficult since simulations are non-exhaustive. Further, it is almost unheard of to use simulation for checking other important properties associated with asynchronous designs such as freedom from livelock, safety and liveness. However, associated with compact but efficient and expressive hardware description languages, formal techniques open the way to reason about asynchronous designs and test them thoroughly

for their characteristic properties. Importantly, the verification results hold over all possible input sequences and for all possible timing variations.

This dissertation is concerned with the formal specification and verification in CCS of the AMULET1 asynchronous microprocessor, and shows that the techniques advocated can cope with an industrial strength microprocessor.

1.1. Asynchronous Circuit Design

The asynchronous circuit design style has a number of potential advantages over its synchronous counterpart especially in terms of local synchronization, composability, and power consumption.

- **Local Synchronization**

  For large modern synchronous circuit designs, clock distribution and clock skew are becoming increasingly costly and difficult to handle. The DEC Alpha CPU devoted 30% of its total area to clocking circuitry. Since asynchronous circuits do not feature a global clock and are designed as separate subsystems with standard communications amongst them, the problems associated with clock distribution and clock skew do not arise.

- **Composability**

  Asynchronous systems have simple and standard interfaces. This composability of asynchronous design not only provides a simple way of building larger structures hierarchically, but also makes it easy for system upgrading when improved circuitry becomes available. Large asynchronous systems can be composed of subsystems operating at widely different speeds. Each of these subsystems can be optimized according to its own needs, and how often it is used.

- **Power Consumption**

  All parts of synchronous systems dissipate power whether they are in operation or not since transitions of the clock are distributed across the entire chip on every clock cycle, while asynchronous systems only draw power when they are doing actual work. Although asynchronous systems often require more signal transitions in a

given transaction, these transitions usually occur only in areas actively involved in the current computation.

The above advantages have been evaluated in detail by Gopalakrishnan and Akella [GA92], Hauck [Hau95], and Davis and Nowick [DN95].

Although possessing some attractive advantages, asynchronous systems have disadvantages in consuming more area and lacking the supporting CAD tools that are widely available for synchronous designs. The design of asynchronous systems is usually considered to be very difficult due to the problems associated with producing hazard free circuits [Ung69] which are deadlock free, and most circuit designers have shied away from it. Thus the synchronous approach has held sway for over thirty years.

But regardless of how successful synchronous systems are, there will always be a need for asynchronous interfaces. As a simple example, it is natural for the interface between a synchronous system and its operating environment to be asynchronous. Thus, despite its difficulties, asynchronous design has never been dropped entirely and many efforts have been made towards producing hazard-free circuits [Ung69, Bre75, Kun92, ND92, LD94, Ste94] (a topic not covered in this dissertation). In the past few years, mathematical techniques have being developed and applied with increasing success to address the difficult topic of characterizing the properties of asynchronous designs. This dissertation is a first attempt to use these techniques for a large practical design.

1.2. Formal Techniques for Asynchronous Circuit Design

Formal techniques for asynchronous circuit design are concerned with the specification and verification of asynchronous circuits. A specification states what should happen. By using some formal reasoning framework, the verification investigates whether a specification possesses desired behaviours including freedom from deadlock, freedom from livelock, safety, and liveness, as well as whether an implementation conforms to its corresponding specification. Informally

- deadlock means that a system may evolve into a state from which no further action is possible.

- livelock means that a system may get into an internal loop and make no further progress in terms of visible inputs and outputs.
- a safety property means that nothing bad will happen when a system operates, e.g. at any time, there may not be more than one user on a shared bus.
- a liveness property means that something good will eventually happen when a system operates, e.g. each instruction class of a processor can be executed next, and none of them loops forever.

If a specification doesn't have these desired properties, it is pointless to implement it. Modifications should be made to the specification until satisfactory results have been achieved before embarking on implementations. At the moment, there are two formal approaches, based upon Hoare's CSP [Hoa78] and Milner's CCS [Mil89] respectively.

1.2.1. CSP Approach

CSP (Communicating Sequential Processes) is a programming notation for describing concurrent systems developed by Hoare [Hoa78]. In CSP, a concurrent system is described as a collection of processes running concurrently. The communication primitive used in CSP is a synchronizing model where the sending process is suspended from further execution until the receiving process is able to receive the information being sent so that the two processes synchronize and the communication takes place as a single atomic action.

Several programming languages based upon CSP have been developed to support the design of asynchronous systems. Amongst these are OCCAM used by Brunvand [BS89], Tangram used van Berkel [vb93], and CHP used by Martin [Mar90].

Trace theory was first proposed by Hoare [Hoa81] for specification and formal verification, and later developed by Rem, Snepscheut and Udding [RdSU83] to reason about the correctness of circuits. In trace theory, the behaviour of a concurrent system is described by the set of possible traces (sequences of events) that can be observed. Each trace represents one possible interleaved behaviour of the system. The observable behaviour of the system is defined by the individual traces combined into a set. Based upon trace theory, Ebergen developed formal models that can be used to describe individual modules, and combine these modules into larger systems [Ebe87, Ebe91, Ebe93].

Dill's verifier [Dil89, DNS92], also based upon trace theory, effectively checks for the safety properties of a concurrent system [DNS92]. However, the liveness properties that are equally important to concurrent systems can not be checked by this verifier although a theory of complete trace structures introduced in [Dil89] can model general liveness properties. With this verifier, Dill has successfully uncovered flaws in published circuits.

1.2.2. CCS Approach

CCS (Calculus of Communicating Systems) is a process algebra for describing and reasoning about concurrent systems developed by Milner [Mil89]. In CCS, a concurrent system is described as a collection of interacting processes which sometimes proceed independently and sometimes need to synchronize with others before they can carry on. CCS provides well-defined syntax and semantics for specifying processes, together with a set of laws for reasoning about these processes and how they communicate with each other.

Compared to CSP, CCS is a coarser language whose descriptions generate fewer states. For modeling asynchronous hardware that can easily enter an enormous number of states, CCS is perhaps more appropriate. In addition, CCS comes together with a mechanized support tool namely the Edinburgh Concurrency Workbench (CWB) [Mol91] which provides a model checker to verify whether a system behaves as expected. Thus, once a concurrent system is specified in CCS, it becomes possible to reason about its processes and verify the correctness of the system.

The key advantage of CCS/CWB is that we can investigate the consequences of a design specification before embarking upon an implementation. Provided with suitable propositions, the CWB model checker can be used to check the important characteristics of a concurrent system, such as freedom from deadlock, freedom from livelock, safety and liveness. Compared with the normal practice of circuit simulation, verification results proved by the CCS/CWB hold over all input sequences, while circuit simulation results are only valid for limited testing sequences.

Specification and verification techniques for asynchronous circuit designs based upon CCS have been proposed and developed at the University of Calgary [Liu92, SABL93, LABS93, Ste94]. CCS has been successfully used in specifying and testing a wide spectrum of small or

moderately sized asynchronous designs [Liu92, BLS+94a, BLS+94b] including flow-through architectures such as Sutherland's micropipeline [Sut89] and Ebergen's stack [EG91], token ring architectures such as Martin's distributed arbiter [Mar85], datapath modules such as Brunvand's carry completion adder [Bru91], and prototype microprocessors such as the MOVE Machine [RKDV92].

This dissertation focuses on the application of CCS to the formal specification and verification of the AMULET1 chip [FDG+93, Pav94, FDG+94, Fur95] which is a practical asynchronous microprocessor developed by the AMULET group at Manchester University, England.

1.3. Contributions of the Dissertation

This dissertation explores the feasibility of applying the CCS process algebra to AMULET1 at two different levels of abstraction: the top level and the subsystem level.

The main contribution of this dissertation is a systematic way of modeling asynchronous microprocessors. The specifications presented are by far the most ambitious specifications of asynchronous hardware yet attempted. At the top level, we successfully modeled AMULET1 in terms of how each instruction class flows through the processor, where the processor is abstracted into several major floor plan modules. At the subsystem level, we successfully modeled each of those major floor plan modules in register transfer level detail. The top level specification provides not only a useful guide to system designers when using AMULET1 in an embedded system, but also insights to how the major floor plan modules function. The register transfer level specification provides a useful guide to circuit implementors of the major functional units of AMULET1. The specifications achieved are succinct and efficient, but accurately capture the functionality details at each level of abstraction and avoid the potential intermediate state explosions that could easily happen.

Although the specifications presented are targeted for a particular asynchronous microprocessor with limited instruction classes, they can be easily expanded should more functionality be added to the processor. Further, the methodology presented is equally applicable to the modeling of other asynchronous designs.

This dissertation also property checks AMULET1 at both levels of specifications. With the Edinburgh Concurrency Workbench, we use a macro-based testing style to check the specifications thoroughly for desired characteristic properties including freedom of deadlock, freedom of livelock, and desired safety and liveness properties. Property checking proves to be a fast and efficient way in detecting specification flaws. The validation of testing results over all possible input sequences and for all possible timing variations can not be achieved through traditional simulation techniques.

Throughout this specification and property checking work, CCS was demonstrated to be an appropriate and efficient tool for modeling complex practical asynchronous designs.

1.4. Structure of the Dissertation

This dissertation is structured as follows:

Chapter 2 describes the CCS notation, and the Hennessy-Milner Logic and modal μ-calculus associated with it. The notation and the associated process logics are explained with small asynchronous design examples. We show that the parallel CCS specification style matches well with the structure of asynchronous systems, and present useful macros for property testing such as freedom from deadlock, freedom from livelock, safety, and liveness.

Chapter 3 gives an overview of the AMULET1 chip. We start with a brief summary of the ARM processor which is a synchronous counterpart of AMULET1. It is followed by AUMLET1's external interface and typical working environment, internal organization and major functional units, and the instruction set and instruction classification.

Chapter 4 examines the specification of practical asynchronous designs at various levels of abstraction. It presents specifications of the register bank at the top level, the register transfer level, and the implementation level. The purpose is to explore both the value of modeling and the appropriateness of the CCS process algebra as a notation at each of these specification levels. The conclusions drawn from this chapter are used as a guide for modeling AMULET1 in the sequel.

Chapter 5 is the top level specification of AMULET1. It models how the processor interacts with its off-chip memory and how the major functional units of the processor

interact with each other. We show how each instruction class (as classified in chapter 3) flows through the processor, clarify the role of each functional unit (floor plan module), present the abstract specifications of these modules, and finally give the top level specification of AMULET1. The specifications are tested for freedom from deadlock, freedom from livelock, and desired safety and liveness properties.

Chapter 6 is the register transfer level specification of the address interface which is one of the major floor plan modules. It models how instruction addresses and data addresses are produced and transferred to the off-chip memory via the address interface. We show the three distinct purposes of the address interface, and how they interact with each other.

Chapter 7 is the register transfer level specification of the data interface which is the simplest floor plan module in AMULET1. It models how data flows between the processor and its off-chip memory. We show how data loaded from the memory is dispatched to the processor either as a new instruction or as regular data during the data input phase, and how data generated by the processor is dispatched to the memory during the data output phase.

Chapter 8 is the register transfer level specification of the execution unit which is another major floor plan module in AMULET1. It models how each instruction class (as classified in chapter 3) flows through the pipelined execution unit. We tabulate the access needs of each instruction class from the data processing hardware associated with the pipeline stages, and to the floor plan modules external to the execution unit.

Finally, Chapter 9 summarizes the dissertation work and presents suggestions for future research.

CHAPTER 2
CCS and Process Logics

CCS (The Calculus of Communicating Systems) [Mil89] is a process algebra developed by Milner over the last 20 years for describing and reasoning about concurrent systems. It provides well-defined syntax and semantics for specifying processes, together with a set of laws for reasoning about these processes and how they communicate with each other. Once a concurrent system is specified in CCS, we can reason its correctness using the Edinburgh Concurrency Workbench [Mol91, CPB90] which provides a very powerful property checker in Hennessy-Milner Logic [HM80, HM85] and its extension, the modal μ-calculus [Koz83].

CCS is used in the rest of the dissertation for the specification and property checking of an asynchronous microprocessor (the Manchester AMULET1 chip [FDG+93]) at various levels of abstraction. My MSc thesis [Liu92] contained an extensive account of how to specify and property check asynchronous hardware in which CCS was demonstrated to be appropriate at various key levels of hardware description: the architectural level, the register transfer level, and the gate level. In this chapter, we content ourselves with describing just the notation and the associated process logics with some small examples.

2.1. CCS

CCS facilitates object oriented descriptions in which a concurrent system is specified as a collection of interacting agents (processes, or hardware blocks) each of whose behavior consists of interleaved, discrete actions.

2.1.1. Notation

The syntax of CCS follows:

$$
\begin{array}{lll}
\mathrm{E} ::= & \mathrm{Nil} & \\
\mid & \mathrm{A} & \textit{constant} \\
\mid & \alpha.\mathrm{E} & \textit{prefix} \\
\mid & \mathrm{E}_1 + \mathrm{E}_2 + \ldots + \mathrm{E}_n & \textit{summation} \\
\mid & \mathrm{E}_1 \mid \mathrm{E}_2 \mid \ldots \mid \mathrm{E}_n & \textit{composition} \\
\mid & \mathrm{E} \setminus \mathrm{L} & \textit{restriction} \\
\mid & \mathrm{E}\,[\,\mathrm{f}\,] & \textit{relabeling}
\end{array}
$$

where

$\mathrm{A} \in$ Const, some fixed infinite set of agent constants,

$\alpha \in$ Act, the set of actions,

L is a subset of Names, and

f is a relabeling function.

The simplest agent in CCS is *0*, the agent that can perform no actions. Complex agents are built from smaller agents using the following CCS operators:

- Prefix If α is an action and E an agent then $\alpha.E$ is an agent which is capable of performing action α and then behaving as the agent E, i.e.,

  $$\text{if } E = a.E' \text{ then } E \xrightarrow{a} E'$$

  Prefixing is used to order sequences of actions. The process $a.b.P$ must first carry out action a and then will carry out action b "some arbitrary time later". Note that delay insensitive designs maps naturally into this semantics.

- **Summation** If E_1 and E_2 are agents, then $E_1 + E_2$ is an agent which behaves non-deterministically like E_1 or E_2, i.e.,

  $$\text{if } E_1 = a.E_1\text{'} \text{ and } E_2 = b.E_2\text{'} \text{ then } E_1 + E_2 \xrightarrow{a} E_1\text{'} \text{ or } E_1 + E_2 \xrightarrow{b} E_2\text{'}$$

  Summation is used to model choice.

- **Composition** If E_1 and E_2 are agents, then $E_1 \mid E_2$ is an agent whose behaviour is such that either of E_1 and E_2 may act independently of the other, i.e.,

  $$\text{if } E_1 = a.E_1\text{'} \text{ and } E_2 = b.E_2\text{'} \text{ then } E_1 \mid E_2 \xrightarrow{a} E_1\text{'} \mid E_2 \text{ or } E_1 \mid E_2 \xrightarrow{b} E_1 \mid E_2\text{'}$$

  or together they may engage in a communication (whenever they are able to perform complementary output and input actions), i.e.,

  $$\text{if } E_1 = a.E_1\text{'} \text{ and } E_2 = \text{'}a.E_2\text{'} \text{ then } E_1 \mid E_2 \xrightarrow{\tau} E_1\text{'} \mid E_2\text{'} \ (a \text{ is normally restricted})$$

  Composition enables us to define a hardware system as "the sum of its parts".

- **Restriction** If E is an agent and L is a set of labels, then $E \setminus L$ is an agent which behaves like E except that it cannot perform any of the actions (as well as the corresponding complementary actions) lying in L externally, although each pair of these complementary actions can be performed for communication internally. Restriction is used to specify internal (hidden) wires.

- **Relabeling** If E is an agent and f is a relabeling function, then $E[f]$ is an agent which behaves like E except that the labels are relabeled as specified by the function f. Relabeling is used to derive instantiations (which will have different names on connections) from a single template.

The above CCS operators have decreasing binding power in the following order:

Restriction and Relabeling > Prefix > Composition > Summation

so that $a.E_1[x/a] \,||\, E_2 + E_3$ should be interpreted as: ((a . ($E_1[x/a]$)) || E_2) + E_3

2.1.2. Example

A CCS process may evolve in three ways: sequentially via the prefix operator (.), non-deterministically via the summation operator (+), and in parallel via the composition operator (|). These are now illustrated by some basic library modules for asynchronous design.

Sequential operation

A Toggle routes an input transition (a) to its two outputs ($'z0$, $'z1$) alternately, as illustrated in Figure 2.1.

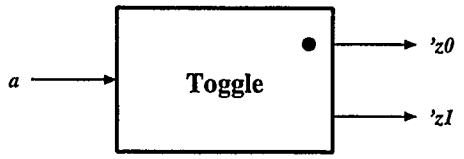


FIGURE 2.1. Asynchronous design library module: Toggle

After initialization, the first input transition is routed to $'z0$ and the subsequent input transition is routed to $'z1$. This behaviour cycle then repeats. In CCS we have

$$Toggle \stackrel{def}{=} a.'z0.a.'z1.Toggle$$

By convention, output names will be quoted, e.g. $'z0$.

Non-deterministic choice

A C-element as illustrated in Figure 2.2 serves as the "AND" function for transition signals.

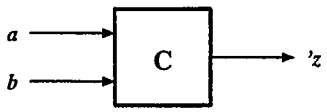


FIGURE 2.2. Asynchronous design library module: C-element

It accepts inputs on a and b in any order and then outputs a transition on $'z$. In CCS we have

$$C \stackrel{def}{=} (a.b + b.a).'z.C$$

If the C-element receives a transition on a, it evolves into the agent $b.'z.C$. If the C-element receives a transition on b, it evolves into the agent $a.'z.C$. If the C-element receives a transition on a and on b, it evolves into one of the above two agents non-deterministically.

Parallel operation

A two-way transition RGD arbiter as illustrated in Figure 2.3 guarantees the mutually exclusive access to a single resource contended for by two independent users.

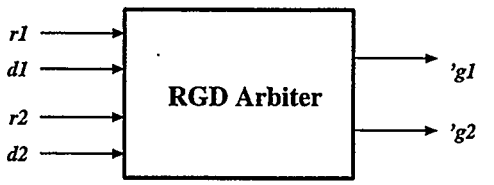


FIGURE 2.3. Asynchronous design library module: RGD arbiter

The two users are required to obey the RGD protocol:

$$U1 \stackrel{def}{=} r1.\ 'g1.\ d1.\ U1$$
$$U2 \stackrel{def}{=} r2.\ 'g2.\ d2.\ U2$$

Where *'g1.d1* and *'g2.d2* must be mutually exclusive. The necessary constraints are enforced by a semaphore agent.

$$\begin{array}{lll} U1 & \stackrel{def}{=} & r1.\ \clubsuit.\ 'g1.d1.\ \spadesuit.\ U1 \\ & & \Uparrow \qquad\qquad \Downarrow \\ Sem & \stackrel{def}{=} & \clubsuit. \qquad\quad \spadesuit.\ Sem \\ & & \Downarrow \qquad\qquad \Uparrow \\ U2 & \stackrel{def}{=} & r2.\ \clubsuit.\ 'g2.d2.\ \spadesuit.\ U2 \end{array}$$

The specification of an arbiter module is a typical parallel specification. It is achieved by composing the two users together with the semaphore agent in parallel. Formally in CCS we have

$$\begin{array}{lll} U1 & \stackrel{def}{=} & r1.\ 'g.\ 'g1.d1.\ 'p.\ U1 \\ U2 & \stackrel{def}{=} & r2.\ 'g.\ 'g2.d2.\ 'p.\ U2 \\ Sem & \stackrel{def}{=} & g.p.Sem \end{array}$$

$$Arbiter \stackrel{def}{=} (\ U1 \mid U2 \mid Sem\ ) \setminus \{\ g,\ p\ \}$$

Notice that even if access is already granted to one user, the arbiter will still be able to accept a fresh request from the other user, i.e., requests are neither held up nor forgotten.

The above specification can also be spelled out in steps using the prefix operator . and the summation operator + as illustrated in Figure 2.4. States shown with bold circles are states in which the arbiter is free. *S00, S01*, and *S10* are duplicated to avoid drawing wrap-around lines.

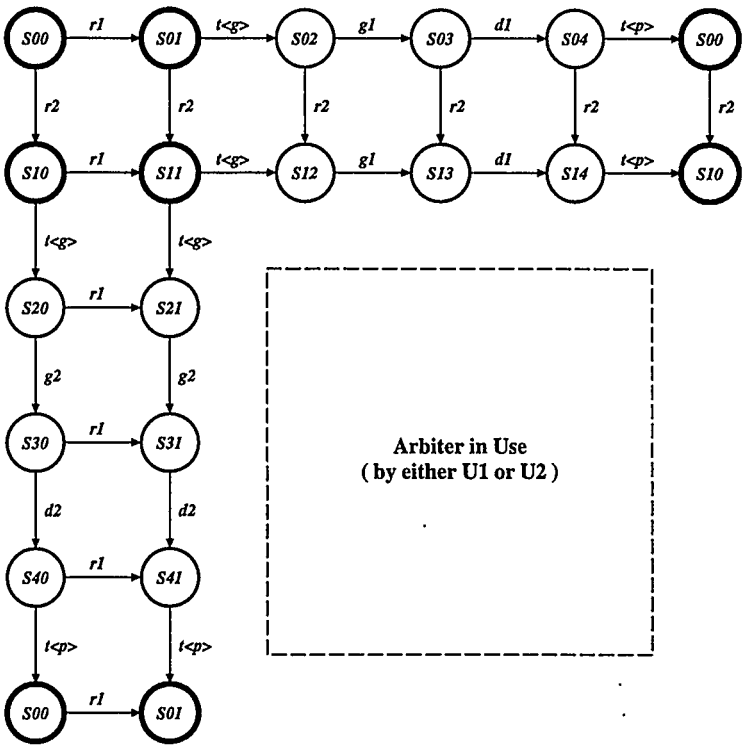


FIGURE 2.4. A two-user RGD arbiter evolves from state to state

The lattice is straightforward but is already long and tedious even for the case of the two-user arbiter. Note that the specifications developed in terms of the parallel operator | are linear in the number of interactions and are textually much shorter. Table 2.1 displays the specifications of arbiters with up to five users. The corresponding number of states increases exponentially.

System Definition	Number of States (minimized)					
	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = k\ (k > 1)$
$(\Pi_{k=1}^{n}\ U_i \mid Sem) \setminus \{ g, p \}$	3	12	32	80	192	$(n+1) \times 2^n$

TABLE 2.1. State space of the RGD arbiter

In addition to its compactness, the parallel specification style is also easier to understand because it presents the structure of the individual communications. [Liu92] and [SABL93] explain the art of specification in CCS with many examples.

2.2. Process Logics

We can never know whether a specification is correct, but we can gain confidence in its appropriateness if its consequences continue to fulfill our expectations when the specification is subjected to tests. Associated with CCS are two process logics: Hennessy-Milner Logic [HM80, HM85] and its extension the modal μ-calculus [Koz83]. They may be used to examine specifications for their consequences such as deadlock, livelock, safety and liveness. [ANB91] gives introduction to both Hennessy-Milner Logic and μ-calculus, and their applications on the Concurrency Workbench. See also [Sti91b, MP92] for more details in wider contexts.

2.2.1. Hennessy-Milner Logic

Hennessy-Milner Logic (HML) is a special type of modal logic that uses labeled transition systems as a model. Labeled transition systems have the form:

$$(\mathcal{P}, \mathcal{A}, \{ \mathcal{T} \mid \alpha \in \mathcal{A} \})$$

where

$\mathcal{P}$ is a non-empty set of agents,

$\mathcal{A}$ is an action set,

$\mathcal{T}$ is the set of transition relations, $\stackrel{\alpha}{\rightarrow} \subseteq \mathcal{P} \times \mathcal{P}$ for each $\alpha \in \mathcal{A}$.

For example, given the specification that describes the behaviour of a regular FIFO with depth of two:

$$\begin{aligned} FF0 &\stackrel{def}{=} incF.FF1 \\ FF1 &\stackrel{def}{=} incF.FF2 + decF.FF0 \\ FF2 &\stackrel{def}{=} decF.FF1 \end{aligned}$$

then we have,

$$\begin{aligned} \mathcal{P} &\stackrel{def}{=} \{ FF0, FF1, FF2 \} \\ \mathcal{A} &\stackrel{def}{=} \{ incF, decF \} \\ \mathcal{T} &\stackrel{def}{=} \{ FF0 \stackrel{incF}{\rightarrow} FF1, FF1 \stackrel{incF}{\rightarrow} FF2, FF1 \stackrel{decF}{\rightarrow} FF0, FF2 \stackrel{decF}{\rightarrow} FF1 \} \end{aligned}$$

Syntax of HML

Let K range over subsets of an action set $\mathcal{A}$. The syntax of HML is defined,

$$\mathrm{A} ::= T \mid \neg A \mid A \wedge B \mid [\, K \,] A$$

where

A is a formulae of HML,

T is the constant true, the only predefined atomic formula in HML,

$\neg$ is the negation of a formula,

$\wedge$ is the conjunction of two formulae,

$[\, K \,] A$ means: A holds after every action in K.

Other common operators can be derived, for example:

F	$\stackrel{def}{=}$	$\neg \mathrm{T}$	F is the constant formula false
$A \vee B$	$\stackrel{def}{=}$	$\neg(\neg A \wedge \neg B)$	$\vee$ is the disjunction of two formulae
$< K > A$	$\stackrel{def}{=}$	$\neg[\, K \,] \neg A$	$< K >$ is the dual of $[\, K \,]$

Satisfaction of HML

For every formula A of HML, we interpret $E \models A$ as meaning "process $E \in \mathcal{P}$ satisfies the property A", and $E \not\models A$ as meaning "E fails to have the property A". The satisfaction relation $\models$ is defined inductively over the structure of HML formulae (5, 6 and 7 are included for ease of reference, they could be inferred):

(1) $E \models T$ $\quad \forall\, E$

(2) $E \models \neg A$ $\quad$ iff $E \not\models A$

(3) $E \models A \wedge B$ $\quad$ iff $E \models A \wedge E \models B$

(4) $E \models [\, K \,] A$ $\quad$ iff $\forall\, E' \in \mathcal{P}, \forall\, \alpha \in K.$ if $E \xrightarrow{\alpha} E'$ then $E' \models A$

(5) $E \models F$ $\quad$ iff $E \not\models T$

(6) $E \models A \vee B$ $\quad$ iff $E \models A \vee E \models B$

(7) $E \models < K > A$ $\quad$ iff $\exists\, E' \in \mathcal{P}, \exists\, \alpha \in K.\ E \xrightarrow{\alpha} E'$ and $E' \models A$

Their interpretations are:

(1) Every process in $\mathcal{P}$ has property T.

(2) A process has property $\neg A$ when it fails to have property A.

(3) A process has property $A \wedge B$ when it has both properties A and B.

(4) A process satisfies $[\, K \,]\, A$ if after every performance of any action in K, all the resulting processes have property A

(5) No process has property F.

(6) A process has property $A \vee B$ when it has either property A or property B.

(7) A process satisfies $< K > A$ if it is possible to perform an action in K such that the resulting process has property A.

Expressing properties in HML

Using the satisfaction relation, we can show whether an agent can carry out a certain trace one move at a time. This is realized by expressing agent properties at a certain state in HML and checking for correctness using HML satisfaction. Several useful formulae for expressing agent properties are listed below, together with their interpretations.

$E \models [\, a \,] F$		E cannot do an a action
$E \models < a > T$		it is possible for E to do an a action
$E \models [\, - \,] F$		E cannot do any action (it is deadlocked)
$E \models < - > T$		E can do some action (it is live)
$E \models < - > T \wedge [\, - \, a \,] F$		E can do an a action and nothing else
$E \models < - \, a > < b > T$		E can do a non-a action then a b action
$E \models [\, a \,] < b > T$		after all a actions from E, one can do a b action
$E \models [\, a \,] T$		always true, even if E cannot make an a move
$E \not\models < a > F$		always false

For the regular FIFO with depth of two, properties we can test using HML include:

- $FF0 \models < incF > T \wedge \neg(< decF > T)$ *FF0* can accept, but not remove an item
- $FF1 \models < decF > T \wedge < incF > T$ *FF1* can both remove and accept an item
- $FF2 \models < decF > T \wedge \neg(< incF > T)$ *FF2* can remove, but not accept an item

The distinct states *FF0, FF1*, and *FF2* are characterized by modal formulae that hold in that state but in no other states. This is true in general (the Modal Characterization Theorem, [Mil89, section 10.5]).

2.2.2. Modal μ-calculus

HML is good for asking questions a few moves ahead, but cannot cope with recursive definitions. Unfortunately, all hardware agents are recursive and most interesting propositions associated with recursive agents are themselves recursive. For example, the FIFO with depth of two at state *FF1* has the property that all we can do is either an *incF* followed by a *decF*, or a *decF* followed by an *incF* and back to state *FF1* again. This infinite behaviour can be expressed by the following *fix point* equation:

$$FF1 \stackrel{def}{=} (< incF > < decF > \vee < decF > < incF >) \; FF1$$

Fix points

In general, fix point equations may have no solutions ($X = \neg X$) or several solutions. There is a simple syntactic check for the existence of at least one solution:

There will always be at least one solution provided that each fix point variable is within the scope of an even number of negations.

From now on, we assume all our modal formulae pass this simple syntactic check. There are no shortcuts for finding all the solutions to a fix point equation. One has to try all possibilities systematically: the empty set, the sets of singletons, two states at a time, ..., all the way up to $\mathcal{P}$. However, it turns out that fix point solutions form a lattice and that the *least* and *largest* of the solutions are not only unique, but also have interesting physical interpretations and fast direct algorithms.

- The minimum (least) fixpoint includes only that which is necessarily true. It can be found by iteration starting from the empty set of states. The minimum fixpoint expresses *liveness.*
- The maximum (largest) fixpoint includes everything except that which is necessarily false. It can be found by iteration starting from all possible states and paring away those found wanting. The maximum fixpoint expresses *safety.*

See [ANB91, Liu92] for a full tutorial account.

Raw modal μ

Modal μ extends HML with fix points. Assuming X is a fix point variable, we have:

$$A \ ::= \ HML \mid min(X.A) \mid max(X.A)$$

N.B., *min* and *max* are duals, so strictly, only one is required.

As an example, the deadlock free property of a system in raw modal μ is:

$$max\ (\ X.\ <->T \wedge [-]X\ )$$

This expresses the set of states X which can themselves make a move ($<->T$) and from which all moves ($[-]$) take us to members of the set of states X which can ... Thus, if every member of this set of states is capable of making a move, the system is free from deadlock.

It can be seen that properties written in raw modal μ are rather hard to read. It will be more difficult if we need to describe "properties within properties" which require nested fix point equations. But the modal μ-calculus is a very expressive logic and it has been shown that all the time honoured classical temporal logic operators can be expressed within it

[Dam90]. These operators make modal formulae considerably more intuitively understandable than their raw modal μ equivalents. Amongst the basic operators are[1]:

$$\begin{aligned} BOX\ P &\stackrel{def}{=} max\ (\ Z.\ P \wedge [-]Z\ ) \\ PATH\ P &\stackrel{def}{=} max\ (\ Z.\ P \wedge <->Z\ ) \\ POSS\ P &\stackrel{def}{=} min\ (\ Z.\ P \vee <->Z\ ) \\ EVENT\ P &\stackrel{def}{=} min\ (\ Z.\ P \vee [-]Z\ ) \end{aligned}$$

with interpretations

BOX	needs all states on all paths as witnesses
PATH	needs all states on a single path to be witnesses
POSS	needs only a single state on a single path as a witness
EVENT	needs a single witness on all paths

Thus, the deadlock free property can be expressed as:

$$BOX\ <->T$$

Other useful operators include:

- *ONLY a* $\stackrel{def}{=}$ (< a > T $\wedge$ [-a] F)

  This tests to see if it is possible to do an a action but no other actions.
- *ONLY-THEN a P* $\stackrel{def}{=}$ (ONLY a $\wedge$ [a] P)

  This tests to see if the only possible move is to perform an a, and we move to a state satisfying P after a is performed.
- *MUST-DO a* $\stackrel{def}{=}$ EVENT (ONLY a)

  This tests to see if we will eventually reach a state where a is the only possible move.
- *NEC-FOR a z* $\stackrel{def}{=}$ $max(X.[z]F \wedge [-a]X)$

  This tests to see if it is impossible to perform a z without an a. But it does not guarantee that after an a is performed, we will definitely have a z move. This macro can be further extended to:

$$NEC\text{-}FOR'\ K\ z \stackrel{def}{=} max\ (\ X.\ [z]F \wedge [-K]X\ )$$

[1] *PATH* and *EVENT* are presented in slightly simplified form here. Usually *EVENT* has an extra term forbidding deadlock and *PATH*, being its dual, permits deadlock.

where K is an action list which may consist of any number of actions. It states that at least one of the actions in the action list K is necessary for producing a z.

2.2.3. Property Testing in Modal μ

This section summaries the most useful macros for testing system properties such as freedom from deadlock, freedom from livelock, safety, and liveness.

- **deadlock** means that a system may reach a state at which it cannot make any move. For any system *SYS*, absence of deadlock may be expressed as:

$$SYS \models \text{BOX} < - > \text{T}$$

or by its dual:

$$SYS \models \neg (\text{POSS} [-] \text{F})$$

- **livelock** means that a system can "spin" forever on internal moves (τ actions in CCS) without ever doing an input or an output action. For any system *SYS*, this may be expressed as:

$$SYS \models \text{POSS} (\text{PATH} < \tau > \text{T})$$

- **safety** properties state that something bad never happens, this is, the system never enters an unacceptable state (such as deadlock). Different systems will have different classes of safety properties tailored to them. For the FIFO with depth of two, we may want to check that it is never possible to output three times without doing an input.

$$FF0 \models \text{BOX} [\text{incF}] [\text{incF}] (\text{NEC-FOR decF incF})$$

It expresses that *in every state (BOX) it is not possible to perform three consecutive incF actions from FF0.*

- **liveness** properties state that something good eventually happen, this is, it is always possible for the system to enter a desirable state. For any system *SYS*, and a particular action a, liveness of action a may be expressed as:

$$SYS \models \text{BOX} (\text{POSS} < \text{a} > \text{T})$$

this is read as *always possible to move to a state where a action can be carried out.*

Many other interesting applications can be found in [MP92]. For practical testing examples on asynchronous hardware, see [Liu92, LABS93].

2.3. The Edinburgh Concurrency Workbench

The Edinburgh Concurrency Workbench (CWB) [Mol91, CPB90] is an automated tool for analyzing concurrent systems expressed in CCS. With CCS, concurrent systems can be specified as a hierarchy of subsystems composed of objects or agents. After the CCS specifications are submitted, the CWB can be used to check the specifications for well formedness via *sort, sequence* and *states*, as well as for more sophisticated properties including *freedom from deadlock, freedom from livelock, safety* and *liveness* expressed in modal μ macros. Compared with the normal practice of circuit simulation, verification results proved by the CWB hold over all input sequences, while circuit simulation results are only valid for limited testing sequences.

All the specifications presented in this thesis have been tested on the CWB and this thesis work would not have been possible without it. These are some of the most complicated specifications experimented with and tested on the CWB.

2.4. Constraining Intermediate States

Controlling the state space is a major problem when modeling concurrent systems. Although describing hardware systems as compositions of objects makes for clear and succinct specifications, it is standard practice to apply the CWB minimization algorithm and expand a specification into its equivalent (unique) minimal state machine prior to testing for most properties (all except livelock). Informally, the algorithm works by first assuming that composed processes run free and then using hidden line information to constrain on handshakes. Nearly all of our specifications are so large that trying to minimize the whole composition in one step is not feasible. Minimizing parallel specifications, such as

$$(O_1 \mid O_2 \mid ... \mid O_n) \setminus \{ l_1, l_2, ..., l_k \}$$

has to be carried out piecemeal, first minimizing $O_{12} = (O_1 \mid O_2) \setminus \{p_1, ..., p_j\}$, where $\{p_1, ..., p_j\}$ are those internal lines joining O_1 to O_2, and then O_{12} with O_3 etc. Even this strategy may fail if the stepwise compositions are carried out according to some hierarchical

strategy (which is probably the natural way to start). Instead, it is best to carry out the compositions following the flow of data and, wherever possible, to start with a natural bottleneck (e.g. an arbiter) which will cause states hanging off its outputs to sum rather than multiply. Early experiments with the MOVE machine [BLS+94b] reduced minimization times from 4 days to 4 hours following this rule of thumb.

A second major improvement comes from noticing that the semantics of CCS are rather more general than those of actual hardware. As a simple example, consider a wire that forks to both P and Q. When a signal travels down the wire P and Q are initiated, but in either order. Informally, this could be expressed in CCS by

$$\begin{aligned} FORK &= a.(\ 'sP.'sQ + 'sQ.'sP).FORK \\ P &= sP.\ body\_of\_P.\ 'dP.P \\ Q &= sQ.\ body\_of\_Q.\ 'dQ.Q \\ PQ &= (\ FORK \mid P \mid Q\ ) \setminus \{\ sP, sQ\ \} \end{aligned}$$

The key observation here is that CCS permits all possible interleavings, so that even if the handshake on sP occurs first, Q may complete before the very first action of *body_of_P* even commences. For certain "well structured" designs (reference below), forks may be redefined by $FFORK = a.'sP.'SQ.FFORK$ without loss of variety. This simple observation has a dramatic impact on minimization time. For *each* fork in a composition, it effectively halves the number of intermediate states produced on the CWB, but minimizes to exactly the same finite state machine as would be found using the usual definition of a fork with choice. Besides simplifying splitters (such as forks), we may also simplify the corresponding collectors (such as C-elements) with the same effect.

This work is the subject of an ongoing collaboration with Birtwistle and Tofts (see working paper *State space reduction for asynchronous micropipelines*, Tofts, Liu, Birtwistle, June 1 1995) and is not further described in this dissertation. Several common asynchronous design structures (serial, parallel, choice, looping) have been formalized, analyzed and shown to obey this intermediate state reduction property. In practice, the MOVE and AMULET1 designers unwittingly followed these design rules, and we have used this **Reduction Theorem** to good advantage many times. It further reduces the minimization time of the

MOVE machine down to 30 minutes. Comparable figures are not available for AMULET1 which is so much more complex, that nothing of any consequence minimizes unless both the strategies outlined in this section are adopted.

2.5. Summary

In this chapter we have briefly summarized the CCS notation, the Hennessy-Milner Logic and modal μ-calculus associated with it, and hinted at how they are used on the CWB. CCS/CWB supports hierarchical system specification at various levels of abstraction, and the parallel specification style matches well with the compositional structure of asynchronous systems. Using CCS/CWB, the consequences of the specification of a design can be tested thoroughly before embarking upon an implementation, and the characteristic properties proved by CCS/CWB hold over all possible timing variations, both external and internal. Full accounts on these topics can be found in [Mil89, MP92, Sti91a, Sti91b, Sti92].

Although the CCS process algebra can be used to specify the structure of concurrent systems accurately and succinctly, it is not entirely satisfactory for several reasons:

- Functionality is omitted — full value passing is not allowed in the CWB so that the behavioural aspects of a design have to be abstracted away. However, this turned out to be a bonus since the CCS descriptions of AMULET1 are already so large that we doubt whether a more powerful calculus could, at the time of writing, mechanize our models.
- No broadcasting – except for the complementary handshake actions, individual actions are not allowed to happen simultaneously. This makes it impossible to model isochronous forks or the innards of low level cells in CCS since they rely on locally non-delay-insensitive behaviour [BE90].
- No explicit timings – all actions that can fire will fire in due course, but we do not know how long this may take. Thus, as observed above, a CCS specification permits more variety than specific hardware would.

However, within these limitations, CCS is an appropriate specification language because of its generality, readability, compactness, and the mechanized support tool associated with it. The value passing CCS described in Milner's book [Mil89] is supported by the CWB but

is too state rich to be used as the specification language for asynchronous hardware. SCCS (Synchronous Calculus of Communicating Systems) [Mil83] and TCCS (Temporal Calculus of Communicating Systems) [MT89] support broadcasting and hard timings respectively, but again are too state rich to be used for this work.

CHAPTER 3
AMULET1 Overview

Chapter 3 gives an overview of the AMULET1 chip [FDG+93, Pav94, FDG+94, Fur95] developed by the AMULET group at Manchester University, England. AMULET1 is the asynchronous version of the well-known ARM processor [Fur89] – the biggest selling RISC processor in the 1980s, and now a leading macrocell for low power applications. This overview includes a top-level view of AMULET1's typical working environment, the major functional units of its internal organization, and a classification and tabulation of its instruction set.

3.1. The ARM Processor

The Advanced RISC Machine (ARM) is a *32*-bit general purpose reduced instruction set microprocessor. It was the world's first commercial development of the pioneering RISC architecture [Rad83, PS81, HJBG81]. The original design was developed at Acorn Computers, Cambridge, England in 1983. In 1992, the development of ARM6 for the Apple Newton PDA product marked the 130 MIPS per watt ARM processor as a world standard for a range of low-cost and low-power applications.

The ARM instruction set is based upon a load/store architecture and features *16* visible registers each of thirty-two bits. In common with other RISC processors, the instructions that perform data processing functions are separated from those that move data between the memory and the registers. Amongst the more unusual features of the ARM instruction set are: conditional execution of all instructions, pre- and post- indexing of load and store operations, general use of program counter (PC), and multiple register load/store operations. In practice, conditional execution reduces the number of branch operations from about *1* in *5* to about *1* in *7*, pre- and post- indexing speed up array access and renders their coding easier, PC can be assigned to in both load and data operations, and load/store multiple improves the programming efficiency of procedure entry and exit. More details of the ARM processor are given in [Fur89, vSA94].

3.2. AMULET1: An Asynchronous ARM

AMULET1 is a re-implementation of the ARM6 microprocessor using a fully two-phase asynchronous design style based upon Sutherland's micropipelines [Sut89]. It preserves all the functionality of the ARM6 macrocell, except for the MLA (MuLtiply Accumulate) instruction which was too troublesome to implement and certain instructions which have been maintained merely for backwards compatibility with previous ARM chips.

3.2.1. AMULET1 Interface

When communicating with the environment, AMULET1 employs an asynchronous micropipelined interface. As illustrated in Figure 3.1 (following page 66 of [Pav94]), the asynchronous processor has three groups of signals including an output bundle, an input bundle, and controls for initialization, interrupts and aborts.

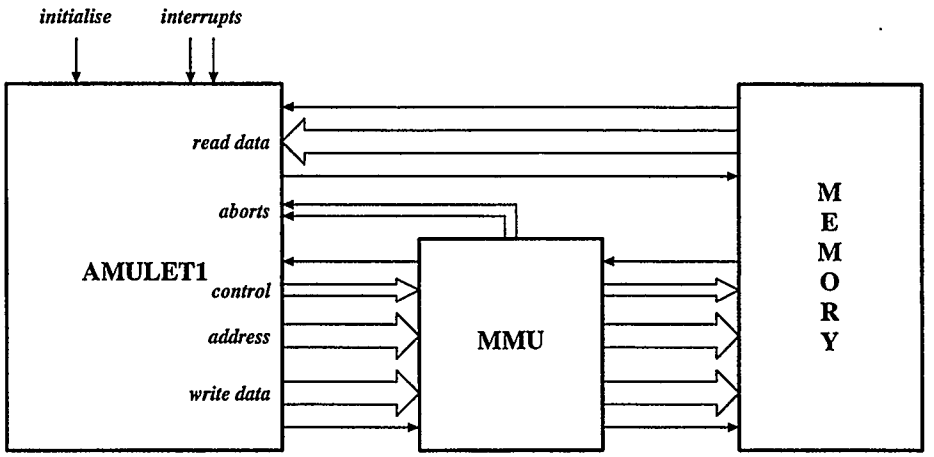


FIGURE 3.1. AMULET1 interface

- The output bundle contains the memory address, the write data, and the control bits from the processor to the Memory Management Unit (MMU).
- The input bundle contains the read data from the memory to the processor.
- The external control includes the processor initialization, interrupt requests, and memory abort response from the MMU.

3.2.2. AMULET1 Organization

The internal organization of AMULET1 is shown in Figure 3.2 (following page 69 of [Pav94]). It consists of five major functional units: the address interface, the register bank, the execution unit, the data interface, and the decode unit. The major bus interconnections between these units are also shown.

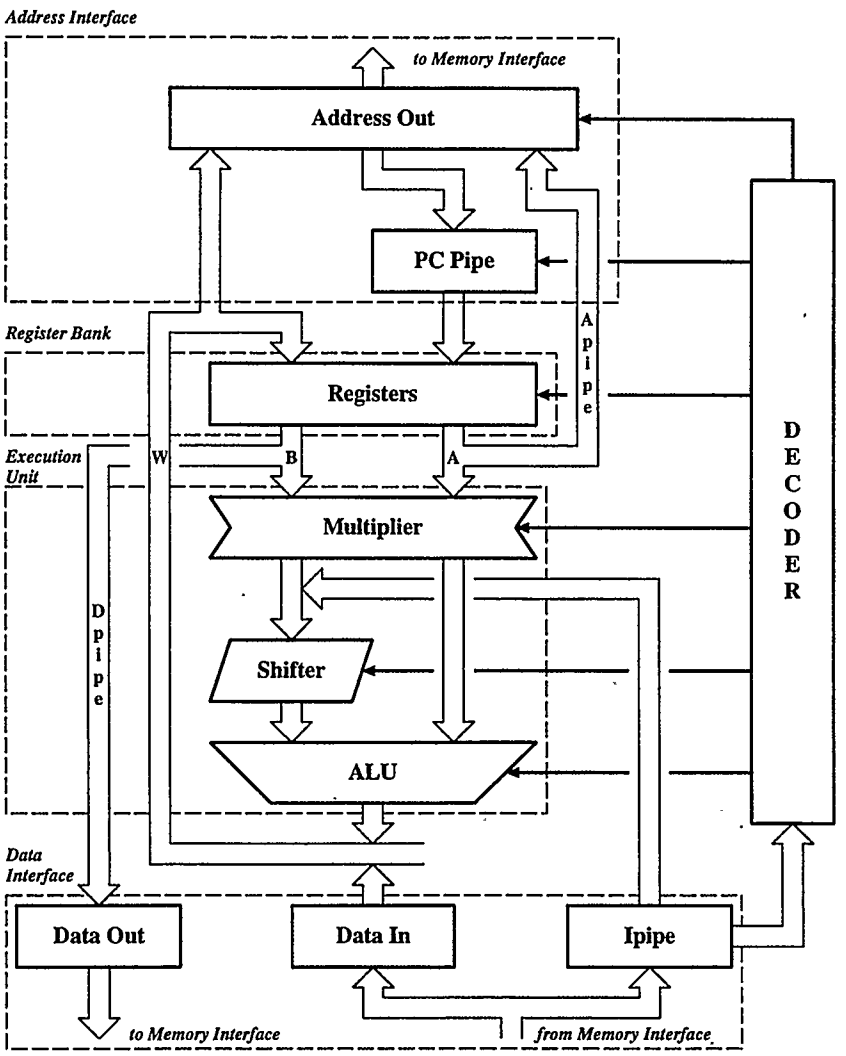


FIGURE 3.2. AMULET1 internal organization

- **Address interface**

  As abstracted in Figure 3.3, the address interface presents all requests to the memory interface (except the right hand sides of store operations which are presented by the data interface), and the PC value to execution unit. Inputs to the address interface are supplied either by the execution unit (via the W bus or the APIPE) or by the data interface (via the W bus).

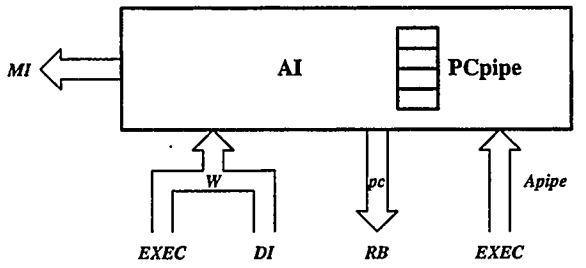


FIGURE 3.3. Address interface

The address interface autonomously generates sequential instruction addresses and forwards them to the memory interface. In addition, it also transfers addresses of load operations (right hand sides) and store operations (left hand sides), and generates the addresses for load/store multiple operations. On initialization, the address interface presents *zero* as the first value to the memory interface.

- **Register bank**

  The register bank contains all the visible registers except PC which is maintained in the PCpipe associated with the address interface. It has two read ports for operand access (A and B buses), a single write port (W bus) for modifying its contents, and an input port supplying the PC value. The PC value can appear on either the A bus or the B bus. For post-index operations, the value on the A bus is also dispatched to the address interface via the APIPE bus. For store operations, the value on the B bus is dispatched to the data interface via the DPIPE bus. RB is illustrated in Figure 3.4.

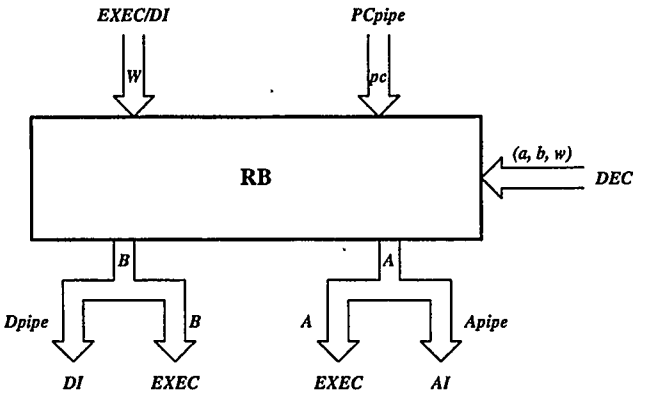


FIGURE 3.4. Register bank

- **Execution unit**

  The execution unit has a pipelined structure for execution efficiency. It accepts operands from the register bank, evaluates the decoded expression, and writes the execution result back either to the register bank or to the address interface via the W bus (which is shared with the data interface). If needed, an immediate value k is supplied to the execution unit by the data interface. EXEC is illustrated in Figure 3.5.

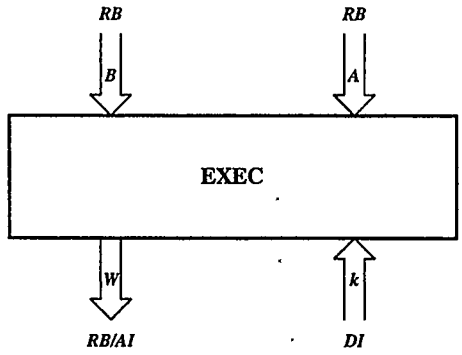


FIGURE 3.5. Execution unit

As showns in chapter 4, the register bank is merely a delay, never a source of contention in AMULET1. Hence we can abstract it away and hide it inside the execution unit. That being so, we combine Figure 3.4 and 3.5 together into the XC unit as illustrated in Figure 3.6.

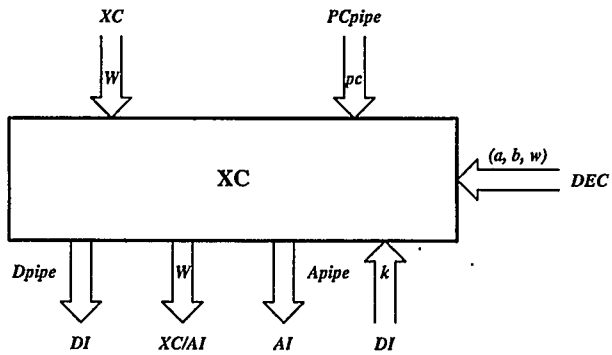


FIGURE 3.6. Abstraction of execution unit and register bank

- **Data interface**

  The data interface manages the data flow between the memory and the processor. Incoming values can be either new instructions, or the results of load operations from the memory, or the right hand side of store operations from the execution unit (register bank) which outputs to the memory. Immediate value k is offered to the XC unit via Ipipe. This is illustrated in Figure 3.7.

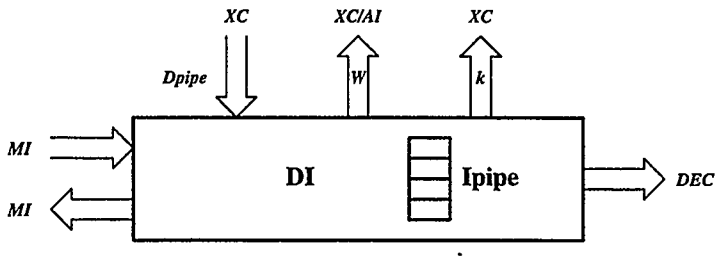


FIGURE 3.7. Data interface

- **Decode unit**

  The decode unit performs instruction decode on the instructions supplied by the Ipipe associated with the data interface. It also provides control information to the address interface, the register bank, and the execution unit. This is abstracted is Figure 3.8.

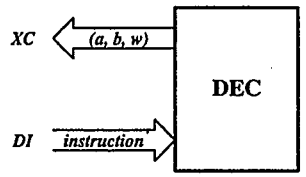


FIGURE 3.8. Decode unit

3.2.3. AMULET1 Abstraction

Combining the abstracted address interface, execution unit, and data interface, we present the abstraction of AMULET1 in Figure 3.9.

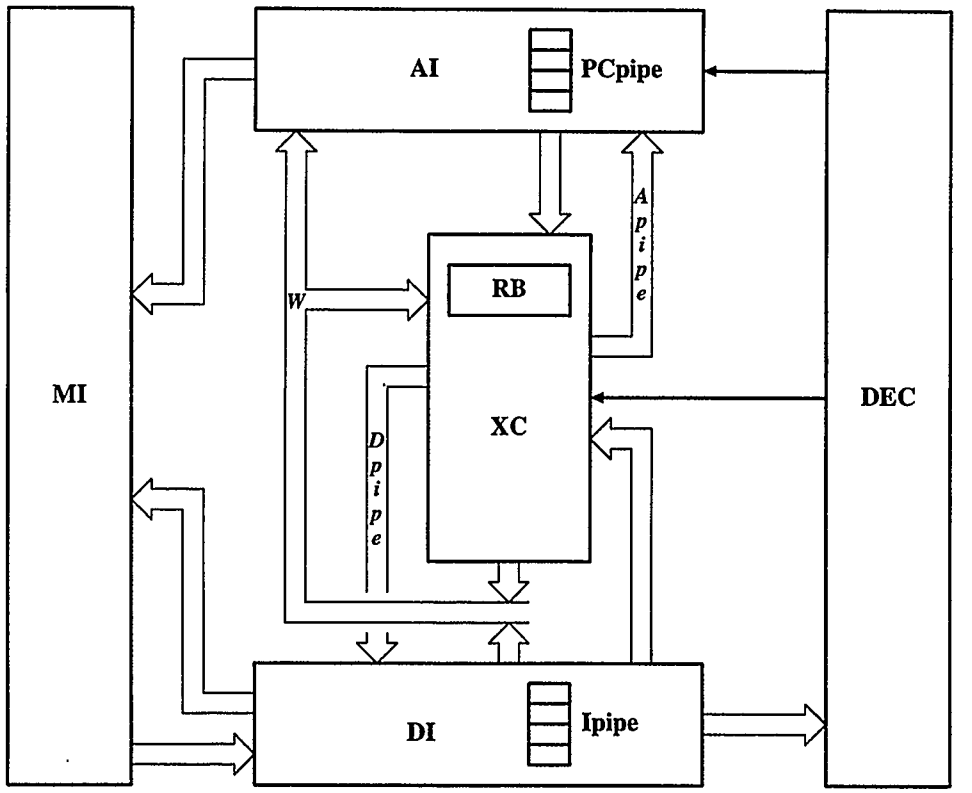


FIGURE 3.9. AMULET1 abstraction

Compared to Figure 3.2, the main simplifications in Figure 3.9 lie in folding the register bank into the execution unit, and abstracting away the multiplier, the shifter and the ALU from the execution unit. These simplifications were the key to reducing the size of the top level model down to the (almost) tractable.

3.2.4. Instruction Classification

The instruction set of AMULET1 is a direct mapping of the ARM instruction set [Gin91, vSA94] onto the asynchronous organization of AMULET1. The instructions are grouped into eight classes according to their datapath activities[1]. The following summary of datapath activity for each instruction class is based upon Figure 3.9.

[1]At this level of abstraction, the multiply operation could be subsumed as a data operation, and the software interrupt operation as a branch operation. However they seem to be worth distinguishing at very low levels of abstraction, and are left as separate categories here.

- **Multiply operation** ($w := a * b$)

  The multiplier accepts two input operands from the register bank. The partial sum and partial carry produced by the multiplier are added together in the ALU. The final result is written back to the register bank via the W bus.

- **Data operation** ($w := a\ op\ b$)

  A data operation takes two operands and performs one of the *16* arithmetic or logic operations supported by the ALU. It differs from the multiply operation in that the multiplier is not activated during it execution and the final result can be dispatched either to the register bank or, in the case when w is R15, to the address interface.

- **Branch operation** ($R15' := R15 + k\ \{R14 := R15 + 4\}$)

  The simple branch operation is a PC relative jump. It adds the current PC value and the offset together in the ALU and sends the result back to the address interface as the new non-sequential instruction address.

  Branch and link requires two phases. The first is the same as branch. In addition, it saves the address of the next instruction in the register bank as the subroutine return address.

- **Load operation** ($w := MEM[a + b]$)

  A load operation is split into an address calculation phase and a data transfer phase which follows the calculation phase immediately down the execution pipeline. The first phase calculates the load data address in the execution unit, and then sends it to the memory and, perhaps the register bank via the W bus and the address interface. The second phase transfers the data read from the memory via the data interface and the W bus either to the register bank or to the address interface. The load operation may additionally allow the pre-indexing or post-indexing of a register. The calculated address may be written back to the register bank in a pre-indexing load operation, and will always be written back to the register bank in a post-indexing load operation.

- **Store operation** ($MEM[a + b] := Dpipe$)

  A store operation is split into an address calculation phase and a data dispatch phase. The store data address is calculated in the execution unit and sent to the

memory via the W bus and the address interface. The store data is dispatched to the memory via the Dpipe and the data interface. The address and the data rendezvous at the memory to complete the store operation. The store operation also allows pre-indexing or post-indexing of a register.

- **Load/store multiple operation**

  The load/store multiple operation supports the load or store operation of any subset of the *16* visible registers. It differs from individual load/store operations in that, once the base address is calculated, the remaining addresses are produced autonomously by the address interface. Since the registers are operated on from low index to high index, if the PC (R15) is loaded or stored, it is always taken last. The load/store multiple operation is by far the most complex instruction in ARM's repertoire.

- **Swap operation** ($w := MEM[b],\ MEM[b] := a$)

  The swap operation swaps the contents of a register and a memory location. It carries out two memory accesses atomically without the possibility of being interrupted.

- **Software interrupt operation** ($R15' := k,\ R14 := R15 + 4$)

  The software interrupt operation causes the processor to break off from the current program by introducing an exception vector k to the memory via the execution unit, the W bus and the address interface. The return address of the instruction after the interruption happens is saved in the register bank. During the operation, the processor is switched to its supervisor mode. Thus, this operation is similar to the branch and link operation described above, but with mode switching.

3.3. Summary

This chapter discussed the external interface, internal organization and instruction set of AMULET1, and classified instructions into eight distinct categories. The instruction classification is summarized below in Table 3.1. We use this table to guide our descriptions of the top level in chapter 5 and of the execution pipeline in chapter 8.

Instruction Class	Example Instruction	Operation
Multiply	MPY Rd, Rn, Rm	Rd := Rn × Rm
Data	ADD Rd, Rn, Rm	Rd := Rn + Rm
Branch	B k	R15' := R15 + k
	BL k	R15' := R15 + k; R14 := R15 + 4
Load	LDR Rd, [Rn, Rm]	Rd := MEM[Rn+Rm]
	LDR Rd, [Rn, Rm]!	Rd := MEM[Rn+Rm]; Rn := Rn + Rm
	LDR Rd, [Rn], Rm	Rd := MEM[Rn]; Rn := Rn + Rm
Store	STR Rd, [Rn, Rm]	MEM[Rn+Rm] := Rd
	STR Rd, [Rn, Rm]!	MEM[Rn+Rm] := Rd; Rn := Rn + Rm
	STR Rd, [Rn], Rm	MEM[Rn] := Rd; Rn := Rn + Rm
Load/Store Multiple	LDM Rn! {regs}	load {regs} from MEM[Rn]; update Rn
	STM Rn! {regs}	store {regs} in MEM[Rn]; update Rn
Swap	SWP Rd, Rn, Rm	Rd := MEM[Rm], MEM[Rm] := Rn
Software Interrupt	SWI k	R15' := k, R14 := R15 + 4

TABLE 3.1. AMULET1 instruction classification

The modeling of AMULET1 starts in chapter 4 by presenting specifications of the register bank unit at three different levels of abstraction: the top level, the register transfer level, and the implementation level. A useful result is that the register bank is a source of delay but not of conflict. Since delay is a natural part of the CCS semantics, AMULET1 is modeled in chapter 5 at the top level in terms of its major floor plan modules with the register bank being abstracted away. The chapters following model the major floor plan modules (the address interface, the data interface, and the execution unit) at the register transfer level, one by one.

CHAPTER 4
Register Bank

The purpose of this chapter is to exhibit how CCS can be used to model a practical asynchronous system, and to investigate at which levels of abstraction it is appropriate and efficient. The conclusions drawn from this chapter are used as a guide for modeling AMULET1 in the sequel.

This chapter presents specifications of the register bank, one of AMULET1's major functional units, at various levels of abstraction, namely the top level, the register transfer level, and the implementation level. The top level specification focuses on the behaviour of asynchronous reads and writes to the register bank. The register transfer level specification focuses on the mechanism of register locking detection, and the possible instruction overtaking with the dual lock FIFO technique. The implementation level specification gives gate level detail of how the control sequences of register read and register write are generated.

The major difficulty in specifying the register bank arises from its data-dependent operations. Unfortunately, accurate modeling of data information using CCS is impractical due to the resulting state explosion. Accordingly, the specifications presented here emphasize the register bank's operational behaviour rather than its data movement details.

4.1. Register Bank Operation

The register bank in AMULET1 contains all the visible registers (except the PC) and stores most of the processor's state. It supports concurrent read and write operations with arbitrary timing dependencies between them, and is implemented with a novel arbiter-free circuit with a locking mechanism that enables efficient read operations in the presence of multiple pending writes [PDF+92]. This section discusses the operating environment of the register bank, as well as its potential operating hazards.

4.1.1. Operating Environment

The register bank consists of a number of registers with common read and write buses. Figure 4.1 illustrates its operating environment:

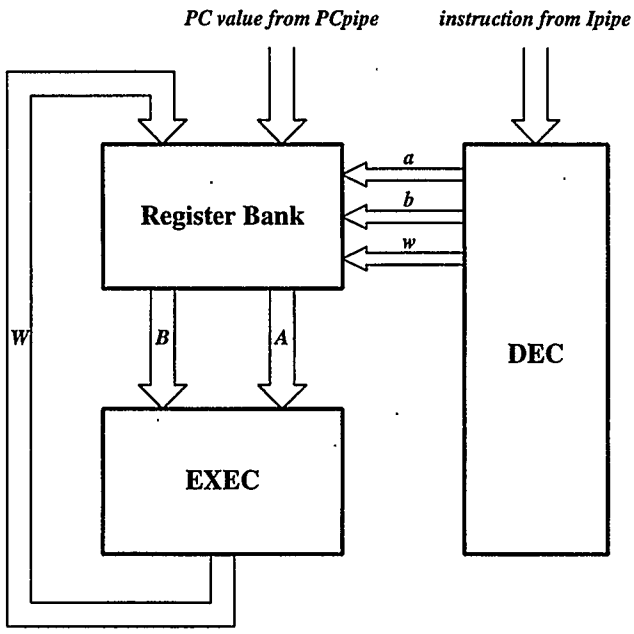


FIGURE 4.1. Operating environment of register bank

Inputs to the register bank include read and write addresses (a, b, w) from the decode unit, current PC value from the PCpipe, and write back data from the execution unit. Outputs from the register bank are operand A and operand B that correspond to read addresses a

and b. For an individual operation, reading from and writing to the register bank operate in the following sequence:

(1) The decode unit extracts read and write addresses (a, b, w) from the current instruction supplied by Ipipe.

(2) The corresponding operands A and B are read out from the register bank provided neither a nor b is pending a write back operation (the read stalls if a write back is pending). The write back register (w) is then locked and becomes unreadable.

(3) Operands A and B are passed to the execution unit and the instruction is executed (the execution of each instruction class will be detailed in chapter 8).

(4) The execution result is written back to the register bank via the W bus. The written register is then unlocked (if this is the only write pending for the given register).

4.1.2. Potential Hazards

Due to the asynchronous nature of register reads and register writes, interaction between the operations could cause metastability problems. Further, because of the pipelined processor structure, it is possible to have multiple outstanding write operations which need to be correctly matched with the corresponding execution results. These potential hazards associated with the asynchronous register bank are eliminated by special register locking and read stalling techniques.

- **Lock FIFO and register locking**

  A record of the destination register addresses is maintained in a FIFO allowing the returning result to be paired with its corresponding address. This FIFO is called a lock FIFO since it is equipped with a locking mechanism that stops reads being carried out from a register with a write pending.

- **Read stalling**

  An instruction may try to read a register whose contents are pending for a write. Under this circumstance, the read has to be stalled until the corresponding write has been completed, and the updated datum is available.

4.2. Top Level Specification

The top level model focuses on the relation between register reads and register writes which are autonomous and almost independent. Every read is evaluated as to whether it can proceed or has to be stalled. A read proceeds directly when the operand registers are not locked. A read has to be stalled when the operand registers are locked, and cannot proceed until the registers have been unlocked by a write.

4.2.1. Register Read

Assume that the (a, b, w) triple is supplied to the register bank by the decode unit. The behaviour of a read operation can be abstracted to:

```
BEGIN
          request for read, but proceed only if a and b are not locked;
          read a and b;
          lock w (and forward a and b to the execution pipeline);
END
```

N.B., letting "read a and b" proceed before "lock w" ensures assignments such as *w:= w+1* are possible. In CCS we have:

```
bi READ
   rReq.(isLK.unlock.PROCEED + ntLK.PROCEED)

bi PROCEED
   read.lockW.READ
```

4.2.2. Register Write

Given that the write back result has been computed, the behaviour of a write operation can be abstracted to:

```
BEGIN
          wait until the execution result (a OP b) is available;
          request for write;
          write ( w :=  result );
          unlock w;
END
```

In CCS we have:

```
bi WRITE
   wReq.write.'unlock.WRITE
```

4.2.3. The Register Bank

The high level specification of the register bank is achieved by composing register read and write in parallel. However, the above specification of the register write implies an execution pipeline with depth of one – the locked register (w) will be unlocked immediately after one write back operations via 'unlock. It can be easily extended to an execution pipeline with depth of N to reflect the actual implementation. The following is a top level specification of the register bank with an execution pipeline with depth of two (trace value unLOCK is added in deliberately for property testing).

```
bi READ
   rReq.(isLK.unlock.unLOCK.PROCEED + ntLK.PROCEED)

bi PROCEED
   read.lockW.READ

bi WRITE
   wReq.write.('unlock.WRITE + WRITE1)

bi WRITE1
   wReq.write.'unlock.WRITE

bi RBANK
   ( READ | WRITE) \ { unlock }

sort RBANK
**{isLK,lockW,ntLK,rReq,read,unLOCK,wReq,write}

min RBANK
RBANK'
**RBANK' has 29 states.
```

```
fd RBANK'
No such agents.
```

The critical property possessed by this specification is that at least one write back has to be done to unlock the register (unLOCK) that is pending for a write (isLK) before the register contents can be read out (read). This is verified on the CWB:

```
cp RBANK
BOX ( [isLK] ( NEC_FOR unLOCK read ))
**true
```

4.3. Register Transfer Level Specification

The top level specification described in the previous section focused on the behavioural level for getting the action sequences correct. It answers questions such as *"when can a read proceed if the corresponding register is locked?"* rather then *"how to detect whether a register is locked?"*. This section presents two register transfer level specifications of the register bank. One focuses on the locking detection mechanism itself, the other focuses on possible instruction overtaking in the AMULET1 implementation which actually includes two lock FIFOs.

4.3.1. Specification I: Register Bank with Locking Detection

The overall organization of this model consists of three major functional blocks as illustrated in Figure 4.2.

- A locking detection unit that stalls a read if the related addresses are pending for write back but proceeds directly when no related locking is detected.
- A lock FIFO that holds the outstanding write back addresses while instructions flow through the execution pipeline.
- A regular FIFO that mimics the execution pipeline stages for the process and storage of the write back data.

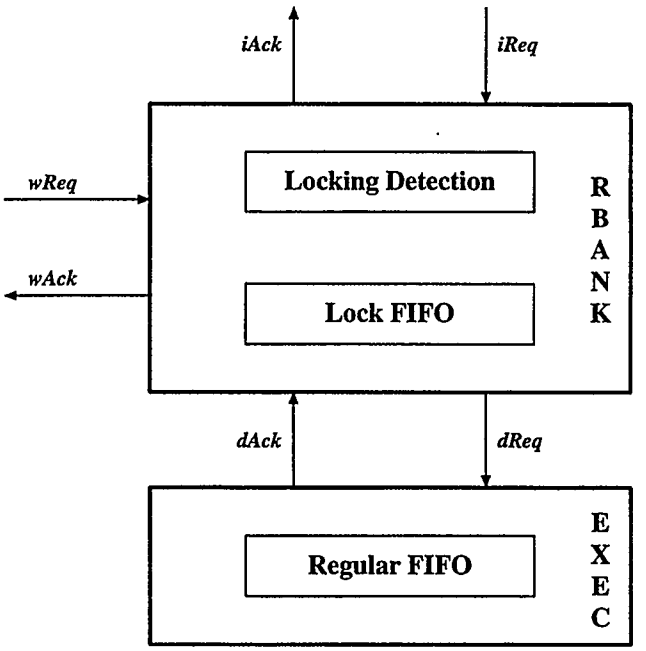


FIGURE 4.2. Register bank organization with locking detection

The register bank consists of three sets of external signals. These external signals coordinate the relationship between asynchronous read and write operations.

- iReq/iAck pair is the read request from the instruction decode unit to the register bank and the corresponding acknowledgement;
- dReq/dAck pair is the execution request from the register bank to the execution pipeline and the corresponding acknowledgement;
- wReq/wAck pair is the write request from the execution pipeline to the register bank and the corresponding acknowledgement.

Control flow

The control flow, which is not shown as an explicit block in Figure 4.2, provides the action sequences that maintain the register bank's asynchronous read and write operations, and coordinates the three major functional blocks displayed. It is generated based upon three major action sequences operating in parallel with internal constraints:

$$\text{CONTROL} \stackrel{def}{=} (\ \text{READ} \mid \text{EXEC} \mid \text{WRITE}\ ) \setminus \text{internal constraints}$$

(1) **READ operation**

A read request proceeds when no operands specified in the current instruction are pending for a write back operation. During a read operation, the action sequence is:

```
BEGIN
        read operands;
        store write back address in the lock FIFO;
        acknowledge back;
END
```

In CCS we have:

$$\mathsf{READ} \stackrel{def}{=} \mathsf{read.'incL.lockW.'iAck.READ}$$

(2) **EXEC operation**

An execution request is generated when both operands of the current instruction have been read out successfully. During an execution operation, the action sequence is:

```
BEGIN
        generate a request for execution;
        data entering the execution pipeline;
        acknowledge back;
END
```

In CCS, we have:

$$\mathsf{EXEC} \stackrel{def}{=} \mathsf{'dReq.'incF.exec.dAck.EXEC}$$

(3) **WRITE operation**

A write request is generated when both write back data and write back address are available. During a write operation, the action sequence is:

```
BEGIN
        generate a write request;
        write back to the register bank;
        unlock the corresponding register;
        acknowledge back;
END
```

In CCS, we have:

$$\mathsf{WRITE} \stackrel{def}{=} \mathsf{wReq.write.'decF.'decL.'wAck.WRITE}$$

(4) **Internal Constraints**

Each of the above individual action sequences could operate in parallel with the following internal constraints:

- As soon as the operands are read out from register bank (read), the saving of write back address in lock FIFO ('incL) and the initiating of execution ('dReq) can proceed simultaneously. The specification is thus modified as:

$$\begin{aligned} \mathsf{READ} &\stackrel{def}{=} \mathsf{read}.\ \mathit{ready}.\ \mathsf{'incL.lockW.'iAck.READ} \\ \mathsf{EXEC} &\stackrel{def}{=} \mathit{'ready}.\ \mathsf{'dReq.'incF.exec.dAck.EXEC} \end{aligned}$$

  where the action *ready* provides the synchronization to fire up EXEC.

- A write operation can only proceed when both write back data and write back address are available. This is expressed as neither the lock FIFO nor the regular FIFO is empty. The specification is thus modified as:

$$\mathsf{WRITE} \stackrel{def}{=} \mathsf{'neL.'neF.wReq.write.'decF.'decL.'wAck.WRITE}$$

Locking detection

The locking detection unit is used to check the lock FIFO and stall the current instruction if its operands refer to registers with pending writes. Assume the lock FIFO has the capacity of holding N write back addresses, and isK stands for K addresses currently in it. The following algorithm abstracts the locking detection mechanism.

```
BEGIN
          read request;
   Check: CASE number of items in lock FIFO OF
            ''is0'' :  do read
            ......

            ''isK'' :  if    not_locked
                       then  do read;
                       else  wait for a write back;
                             goto Check;
            ......

   Read:  read out operands and acknowledge back;
END
```

As an example, we give the CCS specification of the locking detection mechanism assuming that the lock FIFO has a maximum capacity of two write back addresses.

```
bi IREQ
   iReq.( 'is0.READ
        + 'is1.(READ + 'is0.READ)
        + 'is2.(READ + 'is1.(READ + 'is0.READ)))

bi READ
   ......'iAck.IREQ
```

The above specification can also be written as:

```
IREQ  = iReq.(LOCK0 + LOCK1 + LOCK2)
LOCK0 = 'is0.READ
LOCK1 = 'is1.(READ + LOCK0)
LOCK2 = 'is2.(READ + LOCK1)
READ  = ......'iAck.IREQ
```

Figure 4.3 illustrates the transition graph of performing the locking detection mechanism on a lock FIFO with depth of two.

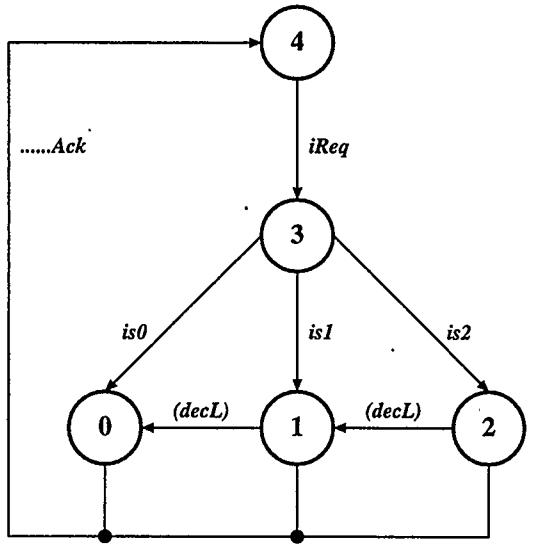


FIGURE 4.3. Locking detection for lock FIFO with depth of two

The interpretation follows:

(1) **Case "is0"**

Lock FIFO is empty hence there are no pending writes. A register read can always proceed.

(2) **Case "is1"**

Lock FIFO holds one write back address which may or may not block the current read request. If a register read does not depend on the pending write, it may proceed directly. Otherwise, a register read will be blocked until the corresponding write back has been completed. This is indicated by no items in the lock FIFO (is0).

(3) **Case "is2"**

Lock FIFO is full and holds two write back addresses which may or may not block the current read request. If register read does not depend on the pending writes, it can proceed directly. Otherwise, register read will be stalled until one write back has been completed and then step *2* is repeated for further checking.

The above specification is mechanically easy to expand to lock FIFO with depth of N, and the expansion is linear in N.

```
IREQ   = iReq.(LOCK0 + LOCK1 + ... + LOCKn)

LOCK0  = 'is0.READ
......
LOCKk  = 'isI.(READ + LOCK(K-1))
......
LOCKn  = 'isN.(READ + LOCK(N-1))

READ   = ...:..'iAck.IREQ
```

Figure 4.4 illustrates the transition graph of performing lock detection on a lock FIFO with depth of N.

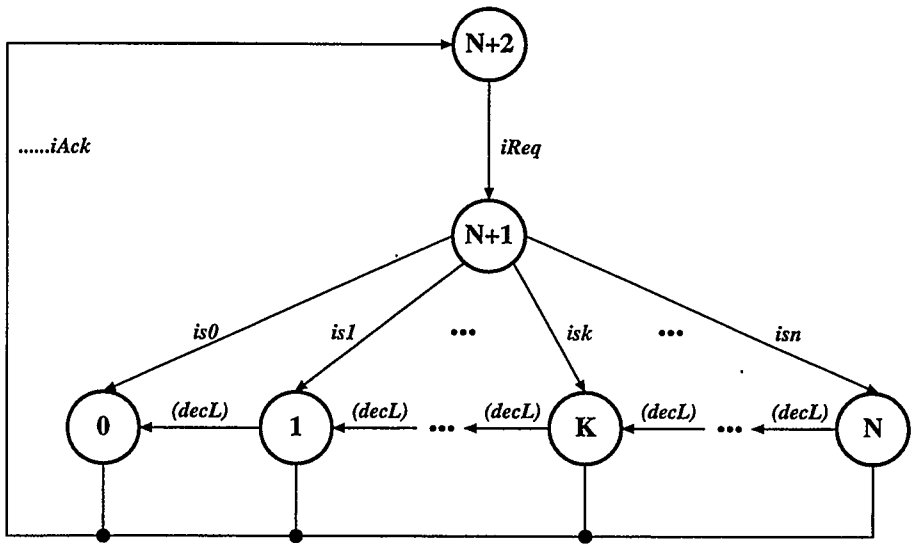


FIGURE 4.4. Locking detection for lock FIFO with depth of N

Lock FIFO

The lock FIFO in register bank is used to hold all the outstanding write back addresses. It is testable for the number of items currently in it (isK) as well as its non-emptiness (neL) at each state. Every time a read is carried out, the corresponding write back address is locked in the lock FIFO (incL). The address will be unlocked for access until the corresponding write back has been completed (decL). The isK flags are the complementary signals for locking detection described above. As an example, the specification of a lock FIFO of depth two is given:

```
bi LF0
   incL.LF1                    + is0.LF0

bi LF1
   incL.LF2 + decL.LF0 + neL.LF1 + is1.LF1

bi LF2
              decL.LF1 + neL.LF2 + is2.LF2
```

In each state, this specification allows movement to adjacent states or the nondestructive testing of the current state. It is easy to extrapolate this specification to a lock FIFO of depth N.

Regular FIFO

The functionality of the execution unit is to process the operands read from the register bank, and generate the write back data. Since the focus here is the register bank rather than the execution pipeline, the execution details are irrelevant. A regular FIFO which can be incremented if it is not full and decremented if it is not empty is used to abstract away the processing details but maintain the storage function of the execution unit.

The specification of the regular FIFO is traditional, and has been presented and tested thoroughly in chapter 2. In order to test the FIFO's non-emptiness (neF), an extra item is added in to each FIFO status. As an example, a regular FIFO of depth two is given:

```
bi FF0
   incF.FF1

bi FF1
   incF.FF2 + decF.FF0 + neF.FF1

bi FF2
              decF.FF1 + neF.FF2
```

This template is used frequently in later chapters.

On the Workbench

After evaluating the individual functional blocks and the control flow that coordinates them, the complete CCS specification for the register bank with locking detection can be presented. The following specification assumes that both the lock FIFO and the execution FIFO are of depth two.

```
bi IREQ
   iReq.( 'is0.READ
        + 'is1.(READ + 'is0.READ)
        + 'is2.(READ + 'is1.(READ + 'is0.READ)))
```

```
bi READ
   read.ready.'incL.lockW.'iAck.IREQ

bi EXEC
   'ready.'dReq.'incF.exec.dAck.EXEC

bi WRITE
   'neL.'neF.wReq.write.'decF.'decL.'wAck.WRITE

bi RBANK
   ( IREQ | EXEC | WRITE | LF0 | FF0 )
   \ { is0, is1, is2, ready, neF, neL, incL, decL, incF, decF }

sort RBANK
**{dAck,exec,iReq,lockW,read,wReq,write,'dReq,'iAck,'wAck}

min RBANK
RBANK'
**RBANK' has 115 states.

fd RBANK'
**No such agents.
```

4.3.2. Specification II: Register Bank with Dual Lock FIFO

Based upon the above specification, a more detailed specification of the register bank at the register transfer level is presented. This specification features two lock FIFOs as implemented in AMULET1, namely the ALU lock FIFO and the memory lock FIFO. The dual lock FIFO structure is the key to instruction overtaking in the AMULET1 implementation. The overall organization of this model consists of four major functional blocks as illustrated in Figure 4.5.

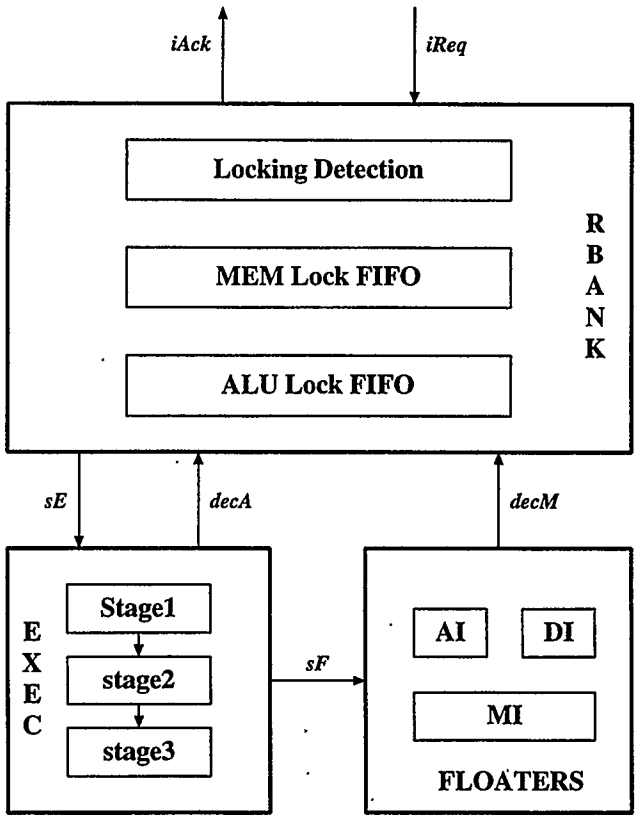


FIGURE 4.5. Register bank organization with dual lock FIFO

- A locking detection unit that stalls a read if the related addresses are pending for write back at either the ALU lock FIFO or the memory lock FIFO, but proceeds directly when no locking is detected.
- Dual lock FIFO: a memory lock FIFO that holds the outstanding write back addresses waiting for data resulting from accessing memory and an ALU lock FIFO that holds the outstanding write back addresses waiting for data achieved from ALU processing.
- A three-stage execution pipeline that represents the process and storage of write back data in the execution unit.
- A FLOATERS unit including the address interface, the data interface, and the memory interface that will only be invoked by instructions related to memory access.

iReq/iAck is the only pair of visible external signal remaining from the previous specification. dReq/dAck and wReq/wAck are omitted in this organization since they are for the sole purpose of modeling asynchronous reads in relation to asynchronous writes and this was evaluated thoroughly in the previous specification. In this specification, more trace values will be added in to display the possible instruction overtaking.

The three-stage execution pipeline and the floaters unit (includes address interface, data interface, and memory interface) are included in this model as the register bank's operating environment to show the possible instruction overtaking that allows the faster internal ALU instructions to overtake the slower external memory access instructions.

Locking detection

In the dual lock FIFO structure, locking detection has to be performed on both FIFOs before the operands can be read out from the register bank. The corresponding specification is developed based upon that of the single lock FIFO. To make life easier, the detection is performed sequentially (in either order) on the two lock FIFOs. The following specification assumes that the detection is performed on the memory lock FIFO first then the ALU lock FIFO, and both FIFOs are of depth two.

```
bi IREQ_M
   iReq.('is0_M.IREQ_A
        +'is1_M.(IREQ_A + 'is0_M.IREQ_A)
        +'is2_M.(IREQ_A + 'is1_M.(IREQ_A + 'is0_M.IREQ_A)))

bi IREQ_A
   'is0_A.READ  +
   'is1_A.(READ + 'is0_A.READ) +
   'is2_A.(READ + 'is1_A.(READ + 'is0_A.READ))

bi READ
   ......'iAck.IREQ
```

Figure 4.6 illustrates the transition graph of performing locking detection on the dual lock FIFO both of depth two.

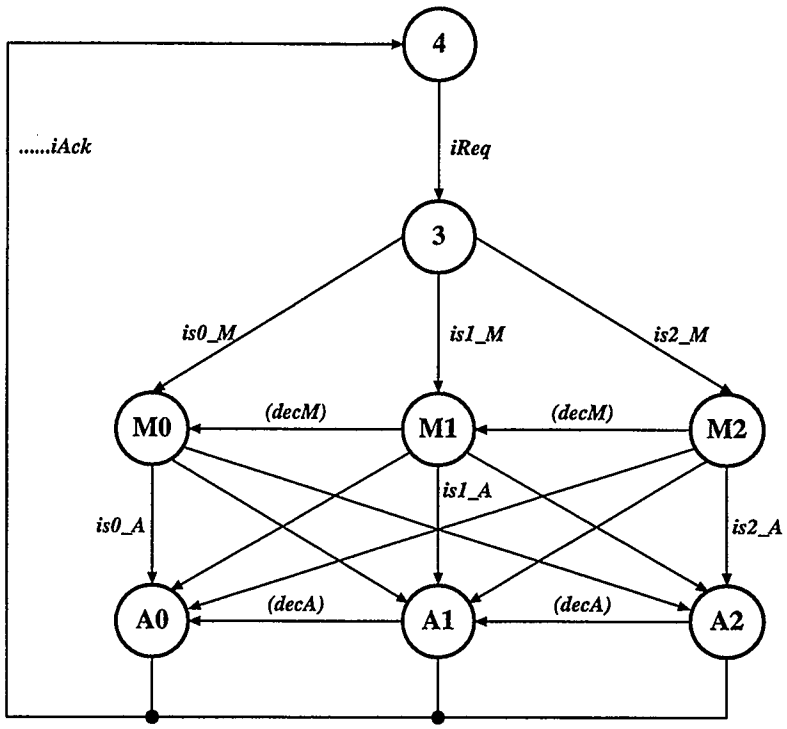


FIGURE 4.6. Locking detection for dual Lock FIFO both with depth of two

Similarly, the above specification is mechanically easy to expand to a dual lock FIFO both with depth of N, and the expansion is linear.

Dual Lock FIFO

The specification of the lock FIFO is explained in the previous section. For an ALU lock FIFO and a memory lock FIFO both at depth of two, we have:

```
bi ALU_LF
   LF0 [ incA/incL, decA/decL, neA/neL, is0_A/is0, is1_A/is1, is2_A/is2 ]

bi MEM_LF
   LF0 [ incM/incL, decM/decL, neM/neL, is0_M/is0, is1_M/is1, is2_M/is2 ]
```

Execution Pipeline

In the previous specification, a regular FIFO was used to simulate the storage function of the execution unit. In order to demonstrate the possible instruction overtaking, more details of the execution unit are needed so that the whereabouts of each instruction can be traced. We now abstract the execution unit into three pipeline stages. The specification of this three-stage execution pipeline follows:

```
bi EXEC
   ( S1_E | S2_E | S3_E )

bi S1_E
   gs1E.ps1E.S1_E

bi S2_E
   gs2E.ps2E.S2_E

bi S3_E
   gs3E.ps3E.S3_E
```

The corresponding control sequences will be discussed later as part of the global control.

Floaters

The floaters include the address interface, the data interface, and the memory interface. Accesses to them are required by some of the instruction classes for the purpose of loading data back to the register bank from the memory which then unlocks the corresponding register address stored in the memory lock FIFO. Although not all the three interfaces have to be accessed during an instruction execution, they are represented as a three-stage pipeline since at this level of specification we don't really need to know to which specific interface the instruction needs access. The specification follows:

```
bi FLOATERS
   ( S1_F | S2_F | S3_F )

bi S1_F
   gs1F.ps1F.S1_F
```

```
bi S2_F
   gs2F.ps2F.S2_F

bi S3_F
   gs3F.ps3F.S3_F
```

The corresponding control sequences will also be discussed later as part of the global control.

Global control

The global control for this level of register bank specification includes three parts. They are instruction read, access to execution unit, and access to floaters. Additional trace values added in to demonstrate instruction overtaking will be discussed together the corresponding control sequence.

(1) **Instruction read**

When the locking no longer exists, an instruction read proceeds directly, and the write back address is saved in the lock FIFO. For an implementation with dual lock FIFO, we need to check which lock FIFO will be used to save the write back address:

(a) For instruction classes that collect write back data directly from the execution unit, the write back address is saved in the ALU lock FIFO only. The beginning of the read is marked by the trace value aluB.

(b) For instruction classes that collect write back data only from the memory, the write back address is saved in the memory lock FIFO only. The beginning of the read is marked by the trace value memB.

(c) For instruction classes that collect write back data from both the execution unit and the memory, write back addresses are saved in both the ALU lock FIFO and the memory lock FIFO. The beginning of the read is marked by the trace value a_mB.

As soon as the instruction enters execution unit via 'gs1E, the read request is acknowledged via 'iAck. The execution continues via 'sALU, 'sMEM, 'sA_M respectively and a new read operation starts from locking detection on the memory lock FIFO. In CCS we have,

```
bi READ
     aluB.'incA.'gs1E.'iAck.'sALU.IREQ_M
   + memB.'incM.'gs1E.'iAck.'sMEM.IREQ_M
   + a_mB.'incA.'incM.'gs1E.'iAck.'sA_M.IREQ_M
```

(2) **Access to execution unit**

The control sequences for accessing the execution pipeline follow the same entering/releasing protocol discussed in chapter 7. When the write back address is locked only in the ALU lock FIFO (aluB), the FIFO is unlocked ('decA) at the final stage of execution pipeline. This marks the end of execution (aluE). When the write back address is also locked in the memory lock FIFO (memB, a_mB), the floaters are invoked ('gs1F) at the final stage of execution. In CCS we have,

```
bi EXEC_CNTR
     sALU.'gs2E.'ps1E.'gs3E.'ps2E.'gW.'decA.'pW.'ps3E.aluE.EXEC_CNTR
   + sMEM.'gs2E.'ps1E.'gs3E.'ps2E.'gW.'gs1F.'cMEM.'pW.'ps3E.EXEC_CNTR.
   + sA_M.'gs2E.'ps1E.'gs3E.'ps2E.'gW.'decA.'gs1F.'cA_M.'pW.'ps3E.EXEC_CNTR
```

(3) **Access to floaters**

Access to floaters starts upon receiving cMEM and cA_M from the execution unit. If the instruction does not need access to a certain interface, it will walk through that stage without any new information being added in. When the corresponding operation completes, write back data is dispatched to register bank via the W bus, and the memory lock FIFO will then be unlocked ('decM). This marks the end of executing the corresponding instruction (memE, a_mE). In CCS we have,

```
bi FLOATERS_CNTR
     cMEM.'gs2F.'ps1F.'gs3F.'ps2F.'gW.'decM.'pW.'ps3F.memE.FLOATERS_CNTR
   + cA_M.'gs2F.'ps1F.'gs3F.'ps2F.'gW.'decM.'pW.'ps3F.a_mE.FLOATERS_CNTR
```

On the Workbench

Based upon the above explanation, this section puts everything together and presents a register bank specification that models instruction overtaking. In order to run the CWB efficiently, a reasonable partition divides the overall specification into three major parts.

```
bi PART1
   ( IREQ_M | ALU_LF | MEM_LF )
   \ { incA, incM, is0_A, is0_M, is1_A, is1_M, is2_A, is2_M, neA, neM }

sort PART1
**{a_mB,aluB,decA,decM,iReq,memB,'gs1E,'iAck,'sALU,'sA_M,'sMEM}

min PART1
PART1'
**PART1' has 111 states.

bi PART2
   ( EXEC | EXEC_CNTR | EXEC_CNTR ) \ {gs2E, gs3E, ps1E, ps2E, ps3E }

sort PART2
**{aluE,gs1E,sALU,sA_M,sMEM,'cA_M,'cMEM,'decA,'gW,'gs1F,'pW}

min PART2
PART2'
**PART2' has 248 states.

bi PART3
   ( FLOATERS | FLOATERS_CNTR | FLOATERS_CNTR )
   \ { gs2F, gs3F, ps1F, ps2F, ps3F }

sort PART3
**{a_mE,cA_M,cMEM,gs1F,memE,'decM,'gW,'pW}

min PART3
PART3'
**PART3' has 132 states.
```

```
bi RBANK
   ( PART1' | PART2' | PART3' | WBUS )
   \ { cA_M, cMEM, decA, decM, gW, gs1E, gs1F, pW, sALU, sA_M, sMEM }

sort RBANK
**{a_mB,a_mE,aluB,aluE,iReq,memB,memE,'iAck}

min RBANK
RBANK'
**RBANK' has 632 states.
```

The specification is tested to be deadlock free and livelock free:

```
Command: cp RBANK'
Proposition: BOX (~Deadlock)
**true

Command: cp RBANK
Proposition: BOX (~Livelock)
**true
```

The following trace sequences show the possible instruction overtaking:

```
iReq memB 'iAck iReq aluB 'iAck aluE ......

iReq a_mB 'iAck iReq aluB 'iAck aluE ......
```

Although instructions that need memory accesses start first (memB/a_mB), the following instruction with write back comes directly from ALU (aluB) could be completed (aluE) before the memory related instructions (memE/a_mE).

4.4. Implementation Level Specification

The implementation of the control sequences for the asynchronous read and write operations to the register bank is illustrated by Paver in [Pav94, Figure 54]. It is replicated in Figure 4.7 to give an overview of the implementation level detail.

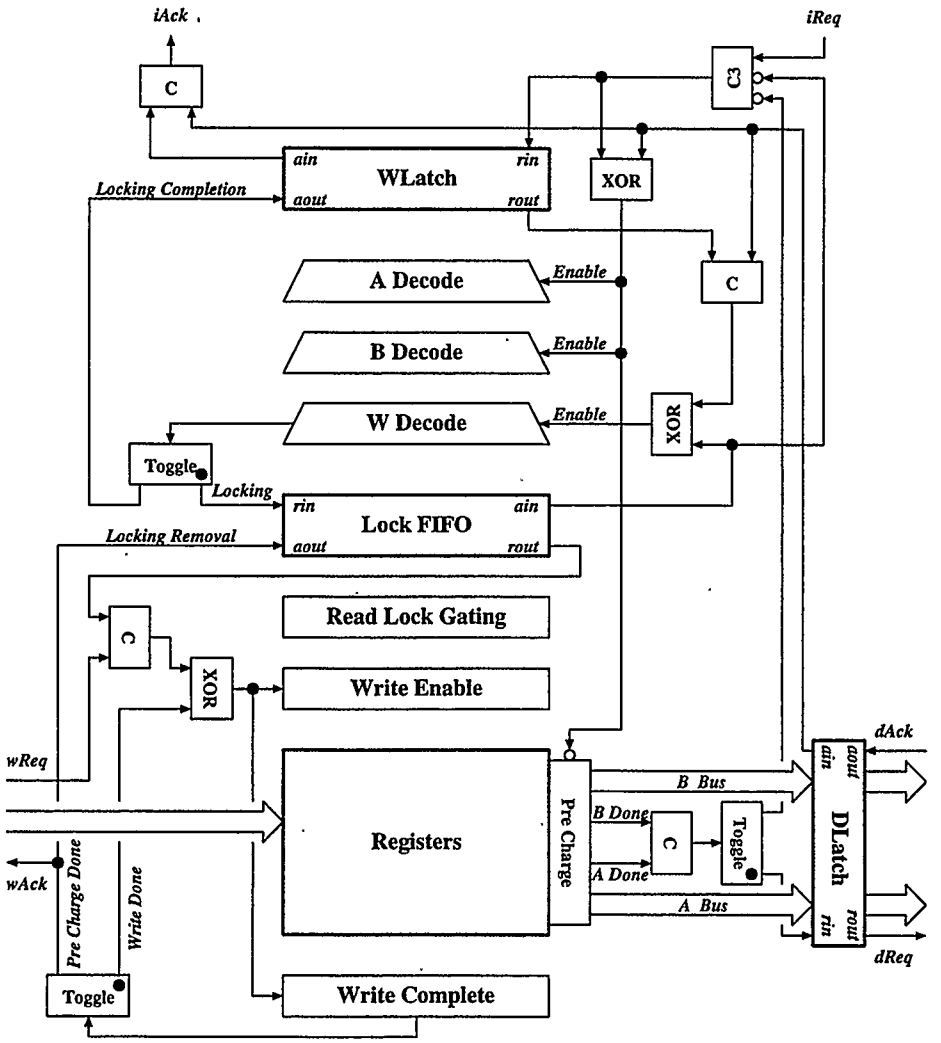


FIGURE 4.7. Register bank organization at the implementation level

This implementation employs a combination of two-phase and four-phase techniques. For example, the two-phase bundled-data convention with transition signaling is used for accessing a FIFO, and the four-phase return-to-zero signaling is used for accessing the decode

units and the registers (read and write). TOGGLE, one of the basic modules for asynchronous circuit design, is the key element in realizing the conversion from one protocol to another.

For simplicity, the following read protocol and write protocol at the implementation level abstract away the details of the decode units for a, b and w addresses, and combines operations on read addresses (a and b) together. Since the decode unit for w address in Figure 4.7 is abstracted away, the XOR/TOGGLE pair associated with the w decode for two-phase to four-phase conversion is also abstracted away. The Read Lock Gating unit is modeled using a Locking Detection unit. Figure 4.8 illustrates the overall organization of this abstracted implementation.

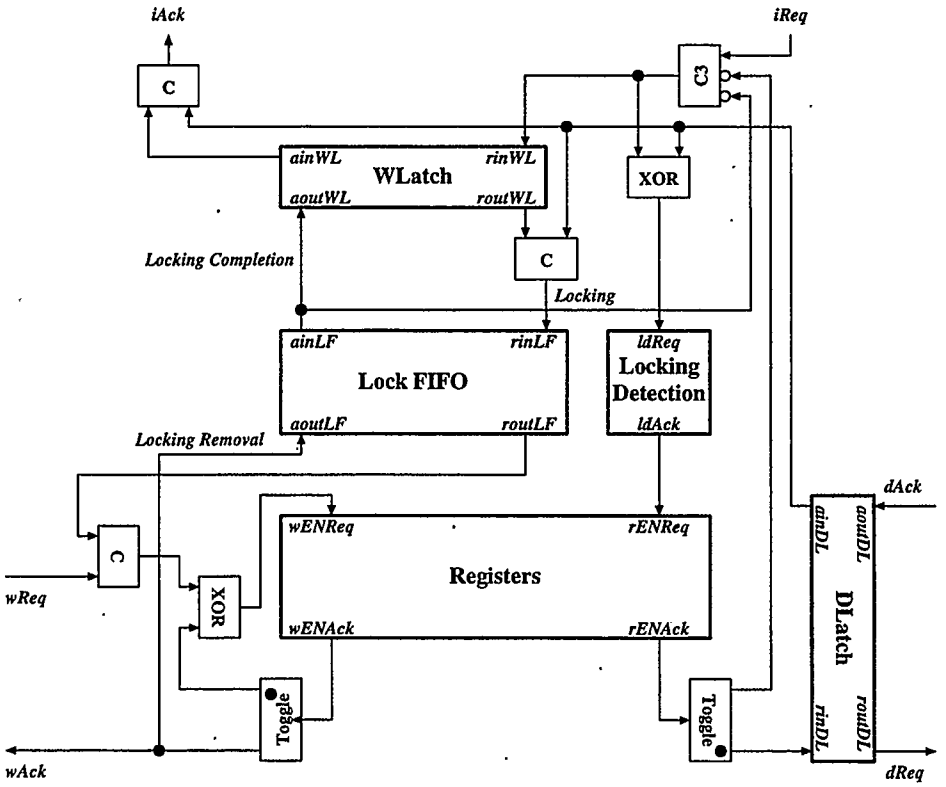


FIGURE 4.8. The overall organization of the abstracted register bank implementation

4.4.1. Read Protocol

Register read starts upon receiving the read request (iReq) from the decode unit when the (a, b, w) triple for the new instruction is available. Figure 4.9 highlights the control sequences of a read operation.

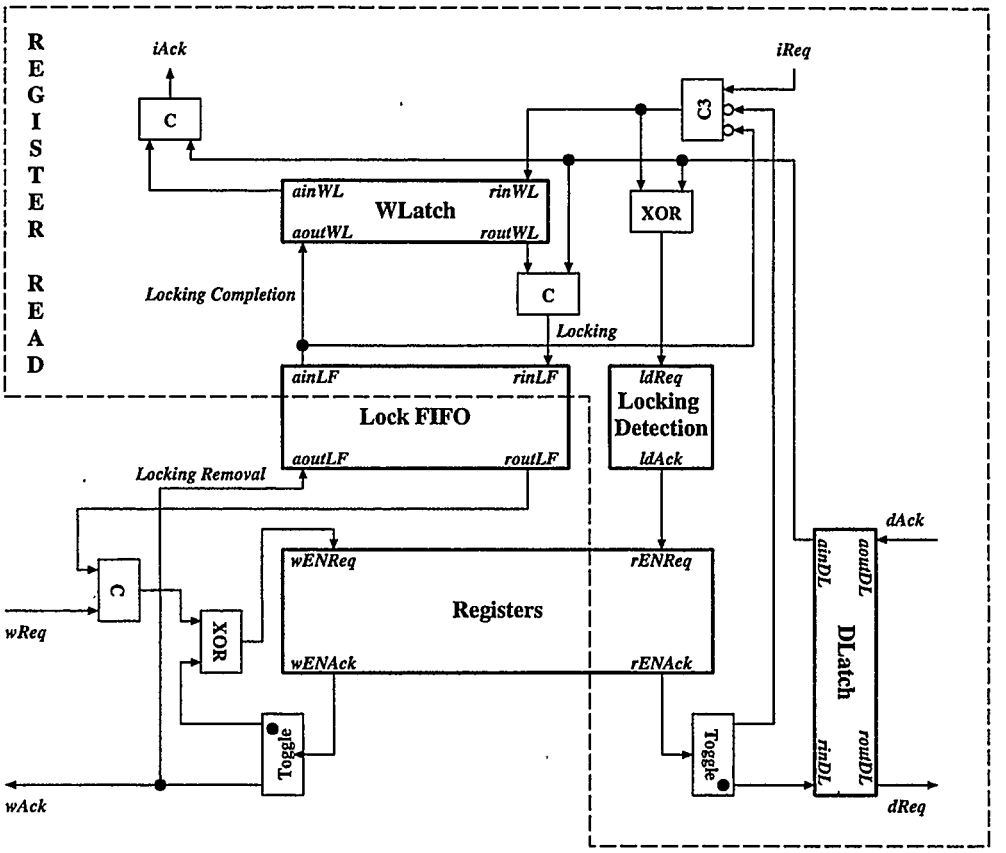


FIGURE 4.9. Control sequences for a register read operation

A read request iReq is stalled until the register bank is ready to start a new read operation: the w address of the previous instruction has been safely latched in the lock FIFO (ainLF), the previous register read operation has been completed successfully and the registers involved are quiescent again. This is guaranteed by a three-way C-element. When this C-element fires, it passes the read request to the locking detection unit (ldReq) through an XOR gate (along with the read register addresses), meanwhile, sends the write register

address to the WLatch (rinWL).

(1) **Locking detection and register read**

The read register addresses are subjected to locking detection before the read can proceed. An unlocked operand proceeds immediately (rENReq), but a locked operand is blocked until a write operation has cleared the lock FIFO. When the read is completed (rENAck), both operands are latched into the DLatch (rinDL) via a TOGGLE. The operands are then passed to the execution unit by signaling on dReq whose corresponding acknowledgment dAck comes from the execution unit after the execution is completed.

(2) **Write register locking and read registers reset**

As soon as the WLatch receives the register write address (rinWL), it acknowledges (ainWL) the C-element which is responsible for generating the acknowledgement to the read request (iAck). This is carried out concurrently with the first step described above.

However, in order to ensure assignments such as *w:= w+1* to be carried out correctly, the write register address (w) has to be delayed from entering the lock FIFO while the register read is in progress. Only when ainDL is received from the DLatch signaling that the operands have been safely latched, can the write address be sent to the lock FIFO (rinLF). After w is latched in the lock FIFO, the WLatch is acknowledged (ainLF). Meanwhile, the three-way C-element is also acknowledged on one of its inputs.

After ainDL is received, read register reset proceeds in parallel with write register locking. Both of the four-phase locking detection unit and the read registers will be reset to their initial states. The completion of this procedure is signaled by the second output of the TOGGLE and collected by the three-way C-element.

4.4.2. Write Protocol

Register write starts upon receiving the write request wReq from the execution unit when the execution result is available. Figure 4.10 highlights the control sequences of a write operation.

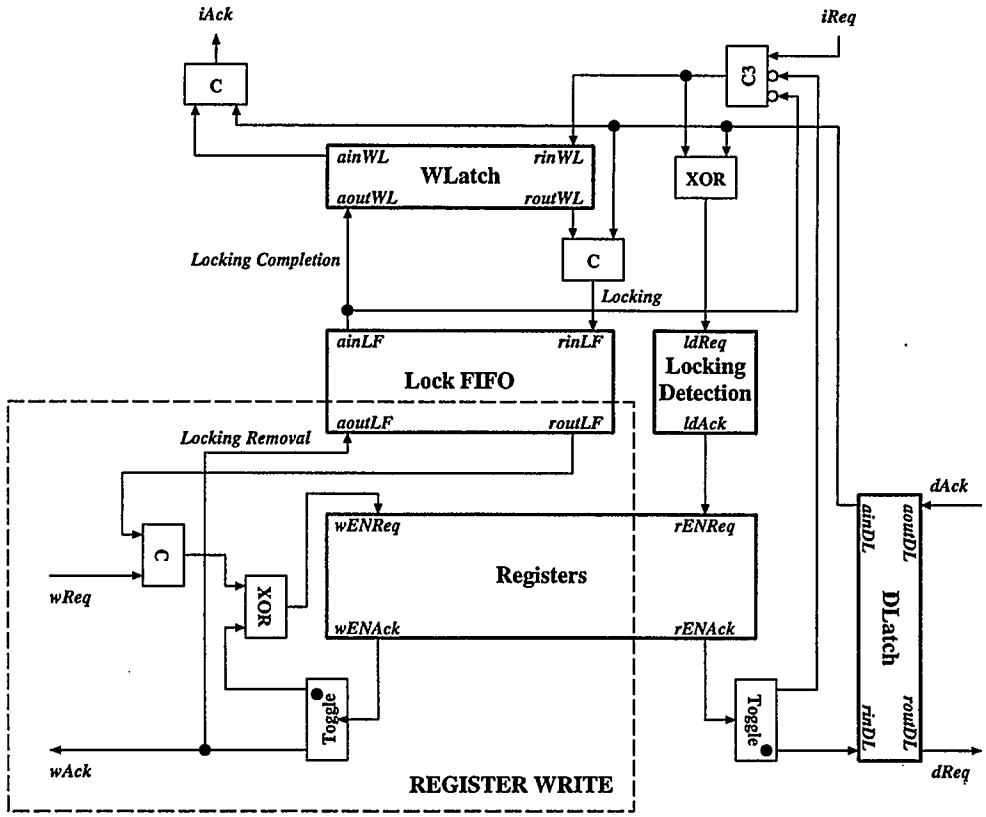


FIGURE 4.10. Control sequences for a register write operation

A write request wReq has to be paired with the availability of a register write address that is in the lock FIFO (routLF). This is guaranteed by a two-way C-element. When this C-element fires, the write register is enabled.

(1) **Register write**

When both routLF and wReq have been received, the write enable (wENReq) is passed by an XOR gate to the write register. A write is carried out, and completed by signaling on wENAck.

(2) **Write register reset**

The wENAck signal generated from the above step is also the input to a TOGGLE. The first TOGGLE output is routed to the write register again through the XOR gate to reset the write register to its original status. The second TOGGLE output triggered by the second wENAck is split into two copies: one acknowledges the

lock FIFO on aoutLF, the other acknowledges the execution unit that the write is completed on wAck.

4.4.3. The Register Bank

The complete register bank operation is implemented as the autonomous register read and register write operating on their own, but communicate between each other through the lock FIFO (routLF, aoutLF): a stalled register read operation cannot proceed until the corresponding register write operation has been completed; a register write request has to be paired with the availability of a register write address. This has been illustrated in Figure 4.8.

4.4.4. Specification and Testing

This subsection presents CCS specifications for the implementation described above. The specifications and testings for the basic library modules (XOR, C-element, TOGGLE, FIFO, etc.) can be found in [Liu92].

Basic functional blocks

(1) WLatch and DLatch

The Wlatch and the DLatch are both regular micropipeline FIFO with depth of one.

```
bi FF0
   incF.FF1

bi FF1
              decF.FF0 + neF.FF1

bi FF_IN
   rinFF.'incF.'ainFF.FF_IN

bi FF_OUT
   'neF.'routFF.aoutFF.'decF.FF_OUT

bi FIF01
  ( FF_IN | FF0 | FF_OUT ) \ { incF, decF, neF }
```

```
sort FIFO1
**{aoutFF,rinFF,'ainFF,'routFF}

min FIFO1
FIFO1'
**FIFO1' has 8 states.

bi WLatch
   FIFO1' [ rinWL/rinFF, ainWL/ainFF, routWL/routFF, aoutWL/aoutFF ]

bi DLatch
   FIFO1' [ rinDL/rinFF, ainDL/ainFF, routDL/routFF, aoutDL/aoutFF ]
```

(2) Lock FIFO

At the implementation level, the lock FIFO operates in a two-phase transition signaling environment. The specification is similar to the regular micropipeline FIFO but with detecting of how many items currently are in it. The following is a lock FIFO with depth of two.

```
bi LF0
   incL.LF1                        + is0.LF0

bi LF1
   incL.LF2 + decL.LF0 + neL.LF1 + is1.LF1

bi LF2
              decL.LF1 + neL.LF2 + is2.LF2

bi LF_IN
   rinLF.'incL.'ainLF.LF_IN

bi LF_OUT
   'neL.'routLF.aoutLF.'decL.LF_OUT

bi LFIFO
  ( LF_IN | LF0 | LF_OUT ) \ { incL, decL, neL }
```

```
sort LFIFO
**{aoutLF,is0,is1,is2,rinLF,'ainLF,'routLF}

min LFIFO
LFIFO'
**LFIFO' has 21 states.
```

(3) Locking detection mechanism

This has been discussed thoroughly at the register transfer level detail. Here we use the same specification with one lock FIFO only. Operating in a four-phase signaling environment, the locking detection unit has to be reset to its initial status after the locking detection is completed.

```
bi LDU
   ldReq.( 'is0.PROCEED
         + 'is1.(PROCEED + 'is0.PROCEED)
         + 'is2.(PROCEED + 'is1.(PROCEED + 'is0.PROCEED)))

bi PROCEED
   'ldAck.ldReq.'ldAck.LDU
```

(4) Read and write protocol

This is used to abstract the four-phase read/write enable and disable procedure.

```
bi EN_RD
   rENReq.'rENAck.rENReq.'rENAck.EN_RD

bi EN_WT
   wENReq.'wENAck.wENReq.'wENAck.EN_WT
```

Register read

The modeling of the register read part is partitioned into three parts for minimization efficiency on the CWB. The overall specification is achieved by composing the minimized parts together with constraints from the environment. The environment ensures that another read request (reqIREQ) can not be produced until the previous one has been acknowledged (ackIREQ).

```
bi PART1
   ( C3'     [   iReq/a,    t2c/b, ainLFb/c,    zC3/z ]
   | FFORK   [    zC3/a,   zC3b/b,  rinWL/c           ]
   | FFORK3 [  ainDL/a, ainDLb/b, ainDLc/c, ainDLd/d ]
   | XOR2    [   zC3b/a, ainDLb/b,  ldReq/z           ]
   ) \ { zC3, zC3b, ainDLb }

sort PART1
**{ainDL,ainLFb,iReq,t2c,'ainDLc,'ainDLd,'ldReq,'rinWL}

min PART1
PART1'
**PART1' has 88 states.

bi PART2
   ( C2       [ ainDLd/a,  ainWL/b,   iAck/z  ]
   | WLatch
   | C2       [ ainDLc/a, routWL/b,  rinLF/z  ]
   | FFORK    [  ainLF/a, ainLFb/b, aoutWL/c  ]
   | LFIFO'
   ) \ { ainLF, ainWL, aoutWL, rinLF, routWL }

sort PART2
**{ainDLc,ainDLd,aoutLF,is0,is1,is2,rinWL,'ainLFb,'iAck,'routLF}

min PART2
PART2'
**PART2' has 350 states.

bi PART3
   ( LDU     [ rENReq/ldAck                      ]
   | EN_RD
   | TOGGLE [ rENAck/a,     rinDL/b,     t2c/c  ]
   | DLatch [ dReq/routDL,   dAck/aoutDL         ]
   ) \ { rENAck, rENReq, rinDL }
```

```
sort PART3
**{dAck,ldReq,'ainDL,'dReq,'is0,'is1,'is2,'t2c}

min PART3
PART3'
**PART3' has 118 states.

bi ENV_RD
   reqIREQ.'iReq.iAck.'ackIACK.ENV_RD

bi READ
   ( PART1' | PART2' | PART3' | ENV_RD )
   \ { ainDL, ainDLc, ainDLd, ainLFb, iAck, iReq, is0, is1, is2,
       ldReq, rinWL, t2c }

sort READ
**{aoutLF,dAck,reqIREQ,'ackIACK,'dReq,'routLF}

min READ
READ'
**READ' has 58 states.

fd READ'
**No such agents.
```

Register write

The register write is simple and needs no further partition. It also operates with constraints from the environment. The environment ensures that another write request (reqWREQ) can not be produced until the previous one has been acknowledged (ackWREQ).

```
bi PART
   ( C2     [   wReq/a, routLF/b,     w1/z ]
   | XOR2   [     w1/a,     w2/b, wENReq/z ]
   | EN_WT
   | TOGGLE [ wENAck/a,     w2/b,    tWc/c ]
   | FFORK  [    tWc/a, aoutLF/b,   wAck/c ]
   ) \{ w1, w2, wENReq, wENAck, tWc }
```

```
sort PART
**{routLF,wReq,'aoutLF,'wAck}

min PART
PART'
**PART' has 44 states.

bi ENV_WT
   reqWREQ.'wReq.wAck.'ackWACK.ENV_WT

bi WRITE
   ( PART' | ENV_WT )  \ { wAck, wReq }

sort WRITE
**{reqWREQ,routLF,'ackWACK,'aoutLF}

min WRITE
WRITE'
**WRITE' has 4 states.

fd WRITE'
**No such agents.
```

The register bank

The register bank is specified as register read and register write operating in parallel, where a stalled read proceeds when the corresponding write has been completed.

```
bi RBANK
   ( READ' | WRITE' ) \ { aoutLF, routLF }

sort RBANK
**{dAck,reqIREQ,reqWREQ,'ackIACK,'ackWACK,'dReq}

min RBANK
RBANK'
**RBANK' has 74 states.
```

```
fd RBANK'
**No such agents.
```

4.5. Summary

This chapter presents models of the register bank at three levels of abstraction: the top level, the register transfer level, and the implementation level. Each model was described in CCS and tested on the CWB.

- The top level model concentrates on how a register read is related to a register write. The corresponding specification results in very few states when minimized on the CWB, and a sufficient number of trace values can be added for a variety of testing purposes. These trace values can be hidden away later when all the desired properties of the system have been proved. The resulting specification can be used as a "black box" component for building larger systems.

  Although this level of specification gives no implementation detail, this simple model provides system designers with a clear understanding of how the major parts of this system communicate with each other (communication between register read and register write) and insights into the functionality of its major functional units.
- The register transfer level model concentrates on the details of the major building blocks in the register bank design: the locking detection mechanism, the lock FIFO, and the control flow that coordinates these building blocks. The corresponding specification results in a reasonable number of states even when suitable trace values are added in.

  Although this level of specification is distinctly higher than the implementation level, the feeling it gives for the role and for the local environment of each component makes it suitable to serve as a practical guide for circuit designers.
- The implementation level model gives gate level detail of how the control sequences of register read and register write are generated. When fitted with suitable environmental constraints and composed efficiently, the specification results in a reasonable number of states when submitted to the CWB. However, since the implementation

level specification is a direct composition of basic library modules, adding in trace values is awkward, and leads to a state explosion.

This modeling of the register bank at three different levels of abstraction serves as a good introductory guide on how practical asynchronous designs can be modeled in CCS, and provides a stiff test for the appropriateness and efficiency of CCS when applied to real hardware at various levels of abstraction As evaluated above, CCS is a suitable notation and is equipped with a support tool (the CWB) sufficient for modeling asynchronous designs at both the top level and at the register transfer level, but it is not powerful enough to cope with full details at the implementation level due to the state explosion incurred by adding traces. Without traces, all we can test are deadlock and livelock, properties that are necessary but not sufficient.

In the light of our experience, AMULET1 will be modeled at both the top level in its entirety and at the register transfer level for its major functional blocks in the following chapters. The register bank itself will be abstracted away since its thorough modeling in this chapter has shown that it can be viewed simply as a delay.

CHAPTER 5

Top Level Modeling of AMULET1

This chapter is concerned with the top level specification of AMULET1. The purpose of this model is to provide an abstraction of how the processor interacts with its off-chip memory and how the major functional units of the processor interact with each other.

The model presented here is based upon previous work [BL95] which proved to be intractable due to the explosive number of states generated by its model of the execution pipeline. This new model abstracts away from the inner details of the major functional units, clarifies their roles, and shows clearly how they are accessed by each instruction class. The first step partitions AMULET1 into several floor plan modules. Next the accesses to each of these modules are tabulated instruction by instruction. The specification of each floor plan module is then pieced together and a complete top level specification follows by composition.

5.1. Top Level Abstraction

The top level abstraction is based upon the model illustrated in Figure 3.9 in chapter 4. Figure 5.1 shows the major floor plan modules of this abstraction.

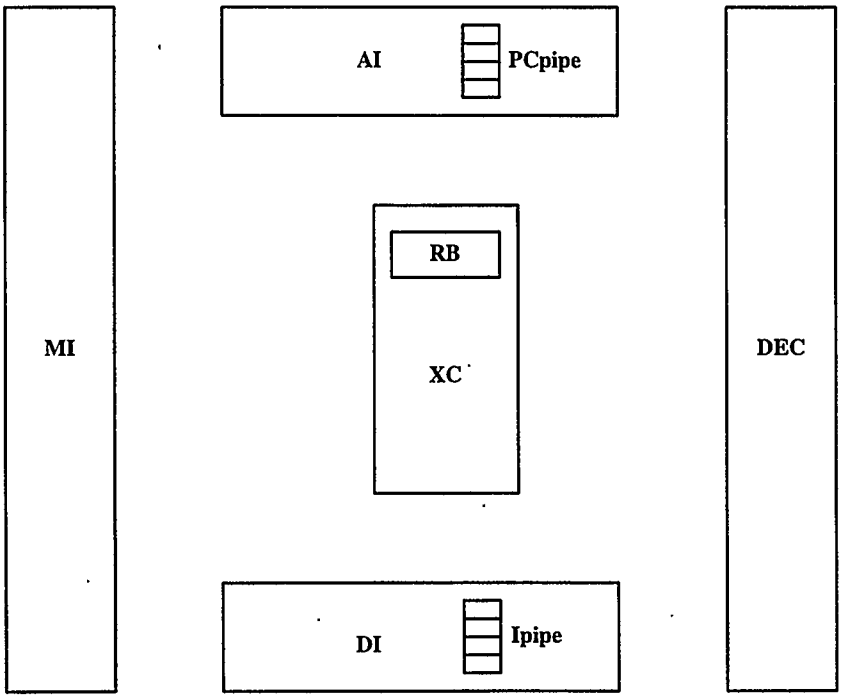


FIGURE 5.1. Floor plan modules for AMULET1

This abstraction expresses the internal structure of AMULET1 together with its external interface as five major floor plan modules, namely the address interface (AI), the data interface (DI), the memory interface (MI), the decode unit (DEC) and the execution unit (XC). Associated with the address interface is the program counter pipeline (PCpipe) which contains a set of PC values generated by the address interface itself. Associated with the data interface is the instruction pipeline (Ipipe) which contains those instructions fetched from memory but not yet decoded. The register bank (RB) is hidden away inside the execution unit.

5.2. Instruction and Hardware Interplay

The classification of the instruction set was discussed in chapter 4. The instruction fetch and decode cycles are identical for all the instruction classes. However, different instruction classes have different access needs to the major functional units during the rest of their execution cycles, and so the corresponding execution specifications vary from one instruction class to another. This section evaluates these access needs instruction by instruction. More can be found in [BL95] which is pitched at a lower level of abstraction and contains details unnecessary for this chapter, but pertinent to chapter 8.

5.2.1. Notation

The signal notation adopted in the following description follows the convention:

content_source_destination

where

content is one of a (address), d (data), i (instruction) or p (PC value);
source is one of AI, DI, MI, XC;
destination is also one of AI, DI, MI, XC.

e.g. p_AI_MI is used to represent the movement of the PC from the AI to the MI.
In addition:

- The decoder presents two operand registers and one write back register to the XC per instruction. We represent these by the triple (a, b, w).
- The PC value for the current instruction is available from PCpipe.
- Should the instruction contain a constant, it will in practice be forwarded from Ipipe. This detail is not described in our model since it does not cause any contentions or conflicts.
- Data forwarded from either the DI or the XC share the same bus (W bus). Since accesses to it are not mutually exclusive, it is expressed as a semaphore agent:

  $$\text{WBUS} \stackrel{def}{=} \text{gW.pW.WBUS}$$

  Provided the obvious protocol is obeyed by each user, this guarantees that at most one user can access the W bus at any time.

5.2.2. Instruction Fetch

For instruction fetch and decode, the AI sends the current PC value to the MI; the MI sends the fetched instruction MEM[PC] to Ipipe via the DI; the instruction is then decoded and presented for execution to the XC:

$$\mathsf{p\_AI\_MI} \longrightarrow \mathsf{i\_MI\_DI} \longrightarrow \mathsf{instruction} \longrightarrow (\mathsf{a}, \mathsf{b}, \mathsf{w})$$

The DEC unit not only presents register addresses (a, b, w) to XC, but also evaluates the instruction and extracts the constant value k from Ipipe (should it be needed). At this stage, the current PC value is accessible to the XC via PCpipe.

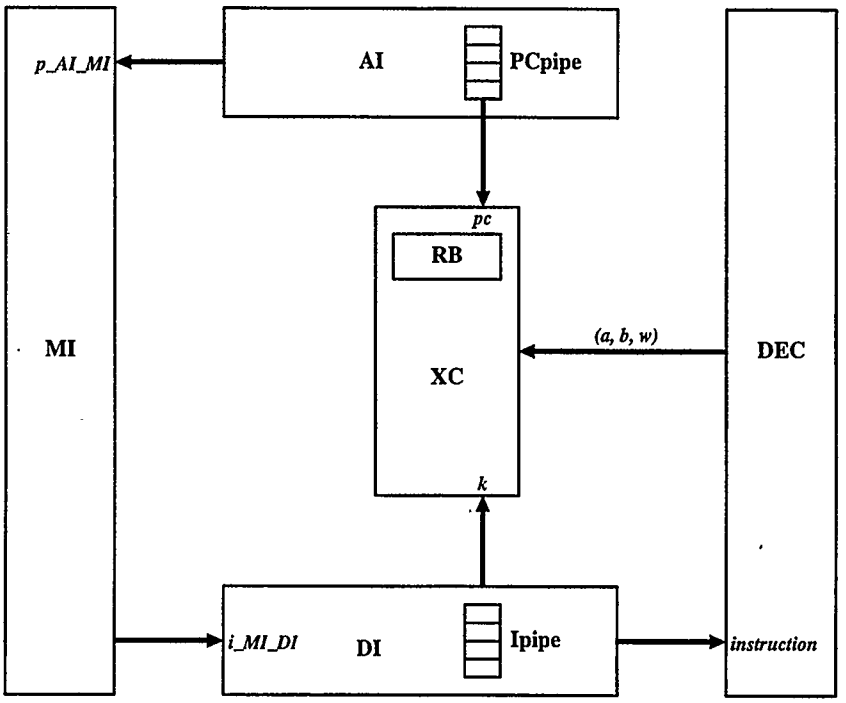


FIGURE 5.2. Instruction fetch and decode

5.2.3. Individual Instruction Classes

The precise details of how instructions flow through inside the execution unit are hidden away at this level of abstraction. Our focus is to show how each instruction class affects the major functional units and uses the buses whilst being executed. The specifications for

each instruction class are described first one by one and then composed together. Since each instruction is fetched in the same way, we start our specification for the execution unit after it has received (a, b, w) from DEC (and k from Ipipe if need be). The matching PC value of the current instruction is also supplied to the execution unit via PCpipe.

Multiply operation (MPY Rd, Rn, Rm)

The multiply operation sends the execution result back to the register bank via the W bus as illustrated in Figure 5.3. It has no effect on any other functional units.

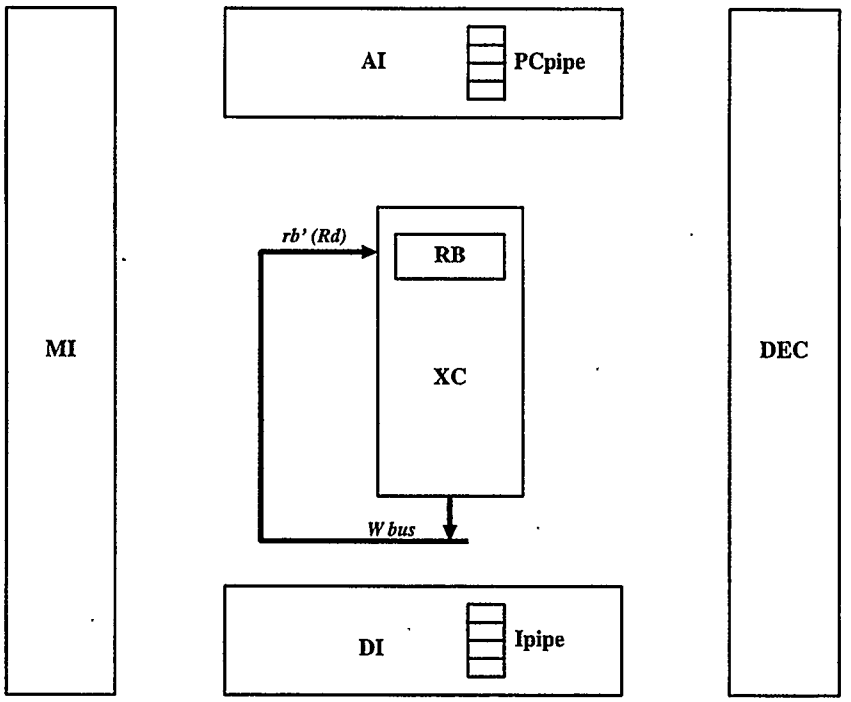


FIGURE 5.3. Multiply operation

Arbitration between the EXEC and the DI is taken care of by the W bus. The CCS specification is simply:

```
EXEC_MPY = 'gW.rb'.'pW.EXEC
```

i.e. EXEC gains control of the W bus, updates RB, and then releases the W bus.

Data operation (e.g. ADD Rd, Rn, Rm)

Data operations differ from multiply operations in that a new PC value may be calculated, so the execution result may be written back to either the register bank or to the address interface.

ADD Rd, Rn, Rm $\Longrightarrow$ Rd via W bus
| p_XC_AI via W bus

If the result is written back to the register bank, it is encoded in the same way as the multiply operation. If the result is a new PC value, it is destined for the address interface, from whence it is forwarded in the normal way (see previous description of instruction fetch). The overall data flow is illustrated in Figure 5.4:

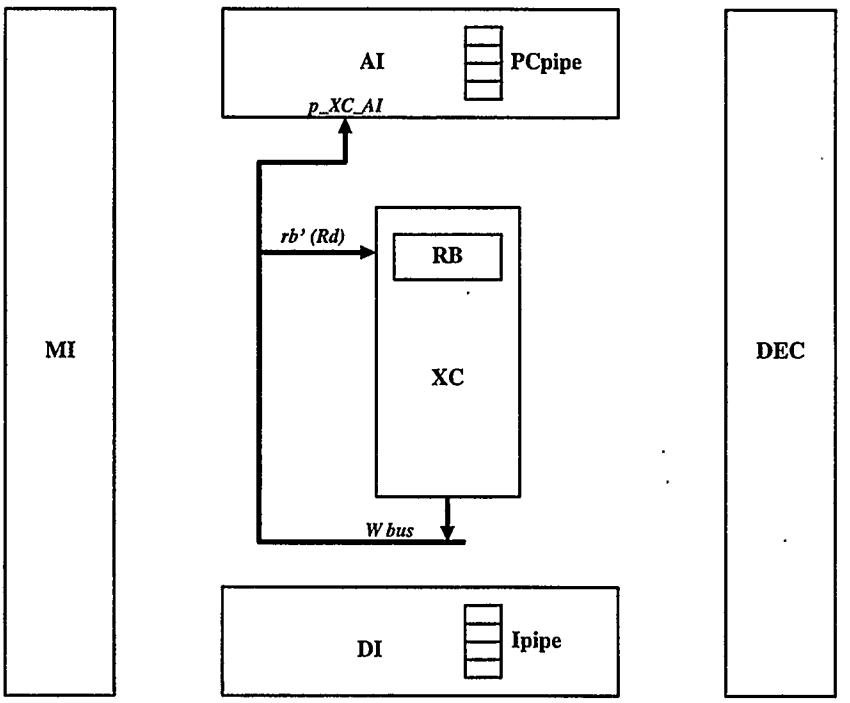


FIGURE 5.4. Data (add) operation

In CCS we have

```
EXEC_ADD  = 'gW.(rb'.'pW.EXEC + 'p_XC_AI.'pW.EXEC)
```

Branch operation (B k & BL k)

The branch operation can be viewed as a special type of data operation whose execution result always becomes a new PC value and will thus be destined for the address interface.

$$\text{B k} \implies \text{p\_XC\_AI via W bus}$$

Branch and link needs two steps in the execution unit. The first step is exactly the same as for the simple branch operation, and the second step is a data operation whose execution result is destined for R14 in the register bank.

$$\text{BL k} \overset{1}{\implies} \text{p\_XC\_AI via W bus}$$
$$\overset{2}{\implies} \text{R14 via W bus}$$

The overall data flow is illustrated in Figure 5.5:

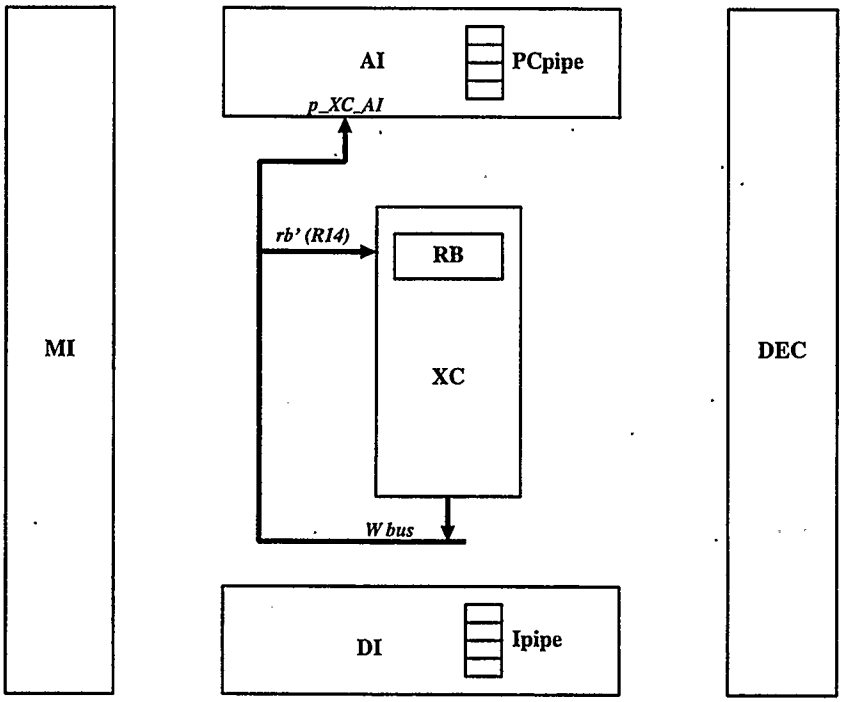


FIGURE 5.5. Branch operation & branch and link operation

In CCS we have

```
EXEC_B          = 'gW.'p_XC_AI.'pW.EXEC

EXEC_BL_PHASE1 = 'gW.'p_XC_AI.'pW.EXEC_BL_PHASE2
EXEC_BL_PHASE2 = 'gW.rb'.'pW.EXEC
```

Load operation (LDR Rd, Rn, Rm)

There are three major varieties of load operations as summarized in Table 3.1 of chapter 4:

(1) pre-indexing without write back (LDR Rd, [Rn, Rm]);
(2) pre-indexing with write back (LDR Rd, [Rn, Rm]!);
(3) post-indexing (which always writes back) (LDR Rd, [Rn], Rm).

In pre-indexing, the base value read from the register bank is modified before being used as the data load address. Thus the write back value (if there is a write back) is the same as the load address. In post-indexing, the base value is used directly as the data load address. Thus in this case, the write back value (write back always occurs) differs from the load address.

All these varieties have two phases when being executed: an address calculation phase and a data transfer phase. They differ from each other only in the address calculation phase, i.e. their data transfer phases are identical.

- **Address calculation phase**

  (1) For pre-indexing without write back, the address is calculated in the execution unit and then dispatched to the address interface via W bus (a_XC_AI).

  LDR Rd, [Rn, Rm] $\Longrightarrow$ a_XC_AI via W bus

  (2) For pre-indexing with write back, the address is calculated in the execution unit and then dispatched to both the address interface (a_XC_AI) and the register bank (Rn) via W bus.

  LDR Rd, [Rn, Rm]! $\Longrightarrow$ a_XC_AI via W bus || Rn via W bus

  (3) For post-indexing, the address is the base value read directly from the register bank and is dispatched to the address interface via Apipe (a_Apipe_AI). The write back result is calculated in the execution unit, and is dispatched to register bank via W bus (Rn).

  LDR Rd, [Rn], Rm $\Longrightarrow$ a_Apipe_AI || Rn via W bus

In summary:

$$\begin{aligned} \text{LDR} \Longrightarrow\ & \text{'gW. 'a\_XC\_AI. 'pW} \\ +\ & \text{'gW. ('a\_XC\_AI || Rn). 'pW} \\ +\ & \text{'a\_Apipe\_AI || 'gW.Rn.'pW} \end{aligned}$$

Formally in CCS, the parallel operations in the address calculation phase may be simplified to sequential operations according to the Reduction Lemma:

```
EXEC_LDR = 'gW.'a_XC_AI.'pW.EXEC
         + 'gW.'a_XC_AI.rb'.'pW.EXEC
         + 'a_Apipe_AI.'gW.rb'.'pW.EXEC
```

- **Data transfer phase**

  The calculated address sent to the AI is then dispatched to the MI for loading the corresponding data. This data will be dispatched to either the register bank or the AI (as a new PC value) via the DI and the W bus.

$$\text{d\_MI\_DI} \longrightarrow \begin{array}{l} \text{Rd via W bus} \\ \mid \quad \text{p\_DI\_AI via W bus} \end{array}$$

  In CCS we have

```
DI = d_MI_DI.'gW.(rb'.'pW.DI + 'p_DI_AI.'pW.DI)
```

The data flow activity for all these varieties of load operation is shown in Figure 5.6.

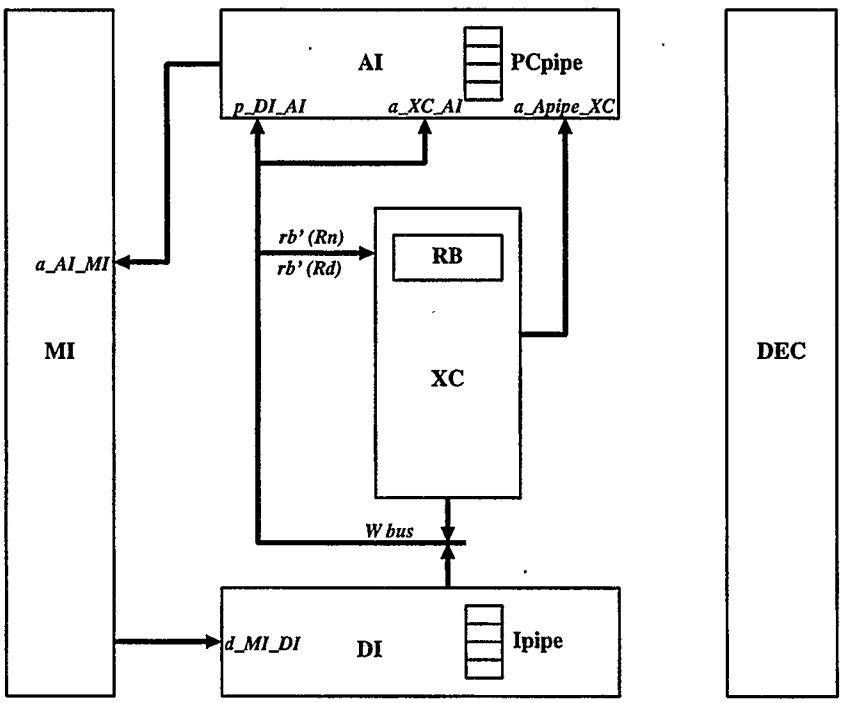


FIGURE 5.6. Load operation

Store operation (STR Rd, Rn, Rm)

The store operation also has three major varieties:

(1) pre-indexing without write back (STR Rd, [Rn, Rm]);
(2) pre-indexing with write back (STR Rd, [Rn, Rm]!);
(3) post-indexing which always writes back (STR Rd, [Rn], Rm).

As with the load operation, all these varieties have an address calculation phase and a data dispatch phase, and differ only in their address calculation phases.

- **Address calculation phase**

  (1) For pre-indexing without write back, the address is calculated in the execution unit and then dispatched to the address interface via the W bus (a_XC_AI).

  STR Rd, [Rn, Rm] $\Longrightarrow$ a_XC_AI via W bus

  (2) For pre-indexing with write back, the address is calculated in the execution unit and then dispatched to both the address interface (a_XC_AI) and to the register bank (Rn) via the W bus.

  STR Rd, [Rn, Rm]! $\Longrightarrow$ a_XC_AI via W bus || Rn via W bus

  (3) For post-indexing, the address is the base value read directly from the register bank and is dispatched to the address interface via the Apipe (a_Apipe_AI). The write back result is calculated in the execution unit, and is dispatched to the register bank via the W bus (Rn).

  STR Rd, [Rn], Rm $\Longrightarrow$ a_Apipe_AI || Rn via W bus

- **Data dispatch phase**

  During the data dispatch phase, data to be stored in the memory is dispatched to the MI via the Dpipe and the DI:

  d_Dpipe_DI.

In practice, the address calculation phase and the data transfer phase overlap:

$$
\begin{aligned}
\text{STR} \Longrightarrow\ & \text{'d\_Dpipe\_DI} \parallel \text{'gW. 'a\_XC\_AI. 'pW} \\
+\ & \text{'d\_Dpipe\_DI} \parallel \text{'gW. ('a\_XC\_AI} \parallel \text{Rn) .'pW} \\
+\ & \text{'d\_Dpipe\_DI} \parallel \text{'a\_Apipe\_AI} \parallel \text{'gW. Rn. 'pW.}
\end{aligned}
$$

Formally in CCS, the parallel operations can be simplified to sequential operations according to the Reduction Lemma. In the specification below, data is dispatched to the MI first, followed by three varieties of dispatching address to the memory interface.

```
EXEC_STR = 'd_Dpipe_MI.( 'gW.'a_XC_AI.'pW.EXEC
                       + 'gW.'a_XC_AI.rb'.'pW.EXEC
                       + 'a_Apipe_AI.'gW.rb'.'pW.EXEC)
```

The calculated address sent to the AI is then dispatched to the MI as the address where data will be stored (a_AI_MI). Data sent to the DI is then dispatched to the MI as the data to be stored (d_DI_MI). a_AI_MI and d_DI_MI rendezvous at the memory to complete the store operation. The data flow activity for all these store operation varieties is shown in Figure 5.7.

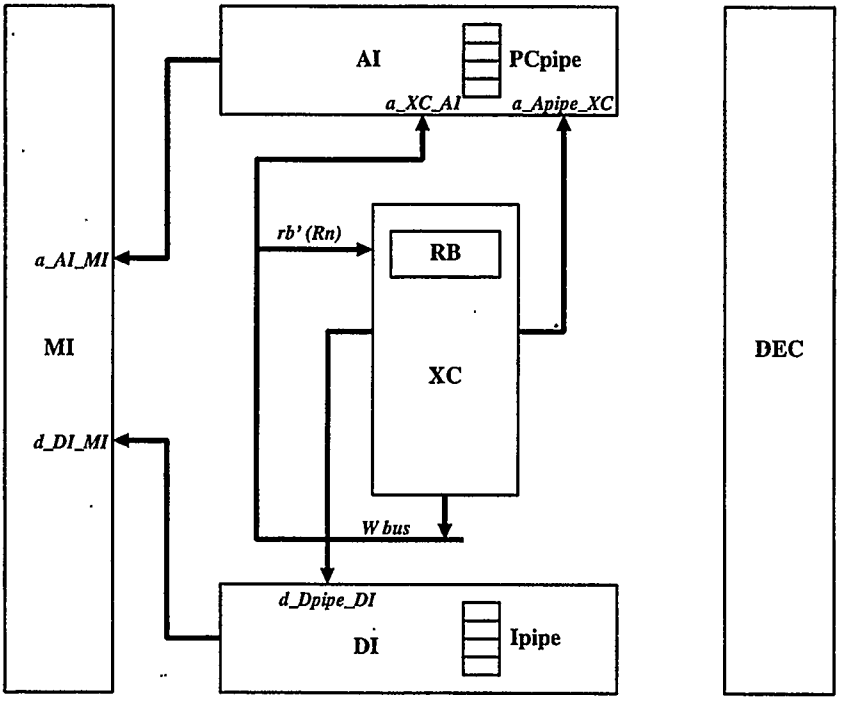


FIGURE 5.7. Store operation

Load/Store multiple operation (LSM Rn! {regs})

The load/store multiple operation has either three phases (store multiple) or four phases (load multiple). The first two phases are common for both load multiple and store multiple. The third and fourth phases vary according to the nature of the operation.

- **Phase 1: start address calculation**

  The load/store multiple operation begins by calculating the start address of the multiple transfer in the execution unit. Once the calculation is complete, the start address is forwarded to the AI via the W bus. The rest of the addresses for the multiple transfer are generated by the AI based upon this value.

  $$\text{LSM Rn! register list} \stackrel{1}{\Longrightarrow} \text{a\_XC\_AI via W bus}$$

- **Phase 2: final address calculation**

  The second phase of load/store multiple operation calculates the final address in the execution unit. Once the calculation is complete, the final address is written back to the register bank via W bus.

  $$\text{LSM Rn! register list} \stackrel{2}{\Longrightarrow} \text{rb' via W bus}$$

- **Phase 3: multiple transfers**

  The third phase of load/store multiple operation is the multiple data transfer section. The number of cycles needed in this phase depends linearly upon by the number of registers being transferred (one cycle for each register).

  - If it is a load multiple operation, this phase carries out the corresponding load operation in a similar manner to the load operation described before. It then signals the AI to prepare the next load address if it is not the end of the transfer.

    $$\text{LSM Rn! register list} \stackrel{3(LDM)}{\Longrightarrow} \text{a\_XC\_AI}$$

  - If it is a store multiple operation, this phase carries out the corresponding store operation in a similar manner to the store operation described before. It then signals the AI to prepare the next store address if it is not the end of the transfer.

    $$\text{LSM Rn! register list} \stackrel{3(STM)}{\Longrightarrow} \text{d\_Dpipe\_DI} \longrightarrow \text{a\_XC\_AI}$$

- **Phase 4: mode switch or base recovery**

  For a store multiple operation, the execution is completed after the third phase. For a load multiple operation, there is a fourth phase where either a mode switch or a base recovery will be carried out. Mode switch is carried out by copying the contents in SPSR to CPSR via the W bus. Base recovery is carried out by writing the base value (saved in ALU) back to the register bank via the W bus.

  LSM Rn! register list $\stackrel{4(LDM)}{\Longrightarrow}$ mode switch or base recovery via W bus

The corresponding data flow activity is illustrated in Figure 5.8.

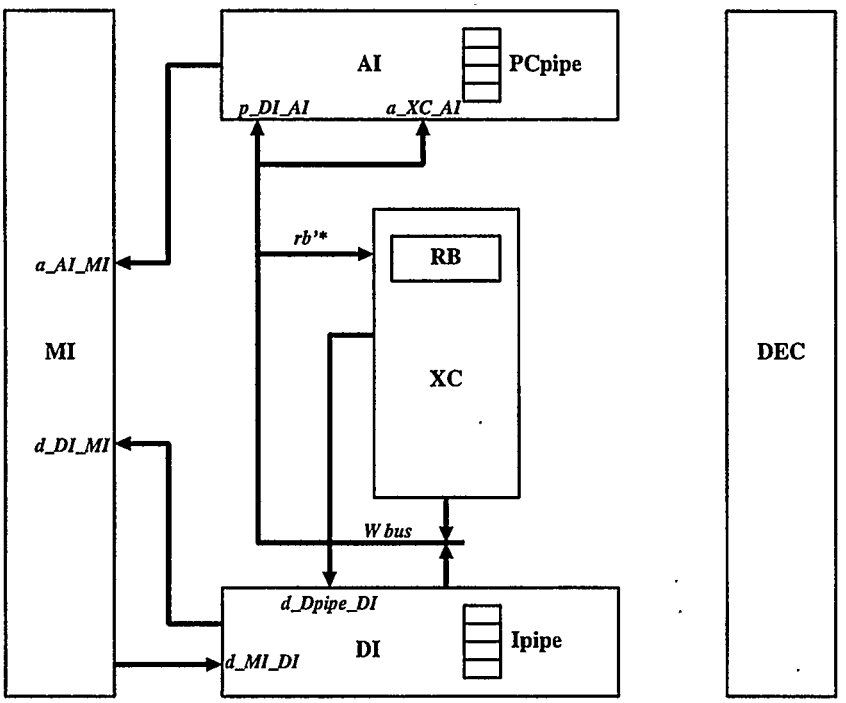


FIGURE 5.8. Load/store multiple operation

In CCS we have

```
EXEC_LSM_PHASE1 = 'gW.'a_XC_AI.'pW.EXEC_LSM_PHASE2
EXEC_LSM_PHASE2 = 'gW.rb'.'pW.(isLDM.EXEC_LDM_PHASE3 + isSTM.EXEC_STM_PHASE3)
EXEC_LDM_PHASE3 = ldr.('a_XC_AI.EXEC_LDM_PHASE3 + EXEC_LDM_PHASE4)
EXEC_STM_PHASE3 = str.'d_XC_DI_MI.('a_XC_AI.EXEC_STM_PHASE3 + EXEC_STM_PHASE3)
EXEC_STM_PHASE4 = 'gW.rb'.'pW.EXEC
```

Swap operation (SWP Rd, Rn, Rm)

Swapping the contents of a register and a memory address is an atomic operation. It is a load operation together with a store operation which share the same memory address. The data loaded from the memory is destined for the register bank only.

$$\text{SWP Rd, Rn, Rm} \xRightarrow{1} \text{a\_Apipe\_AI} \parallel \text{d\_Dpipe\_DI}$$
$$\xRightarrow{2} \text{d\_MI\_DI} \longrightarrow \text{d\_DI\_XC via W bus}$$

R15 is not allowed in any operand position in a swap operation. The data flow is shown in Figure 5.9.

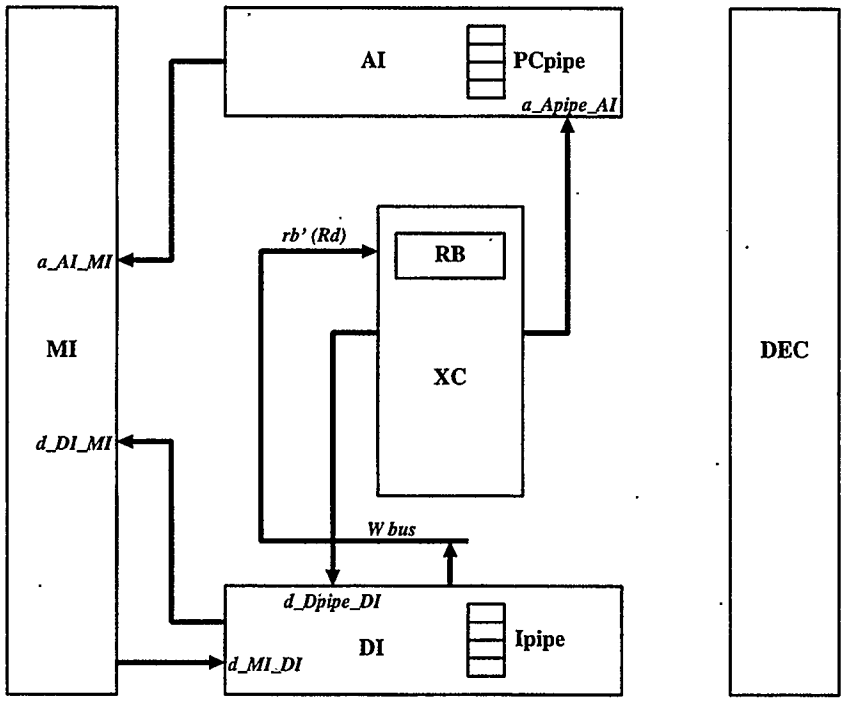


FIGURE 5.9. Swap operation

In CCS we have

```
EXEC_SWP = 'a_Apipe_AI.'d_Dpipe_DI.EXEC

DI       = d_MI_DI.'gW.rb'.'pW.DI
```

Software interrupt operation (SWI k)

The software interrupt operation has three phases. The first phase is comparable to the first phase in the branch and link operation. The second is a write back from execution unit to the register bank during which the processor is switched to its supervisor mode. The third phase is identical to the second phase of branch and link.

$$\begin{aligned} \text{SWI k} &\xRightarrow{1} \text{p\_XC\_AI via W bus} \\ &\xRightarrow{2} \text{mode switch via W bus} \\ &\xRightarrow{3} \text{R14 via W bus} \end{aligned}$$

The data flow activity of software interrupt operation is shown in Figure 5.10.

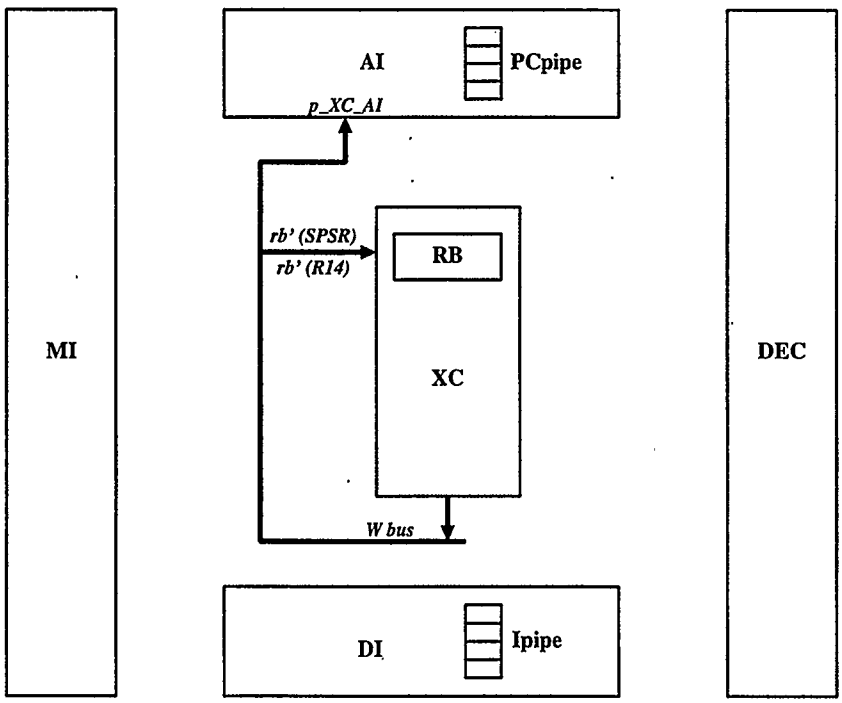


FIGURE 5.10. Software interrupt operation

In CCS we have

```
EXEC_SWI_PHASE1 = 'gW.'p_XC_AI.'pW.EXEC_SWI_PHASE2
EXEC_SWI_PHASE2 = 'gW.rb'.'pW.EXEC_SWI_PHASE3
EXEC_SWI_PHASE3 = 'gW.rb'.'pW.EXEC
```

5.3. Hardware Sharing

In section 5.2, we presented the flow of each instruction class through the major functional units. We now combine the separate instruction flows and present the complete top level abstraction of AMULET1 as illustrated in Figure 5.11.

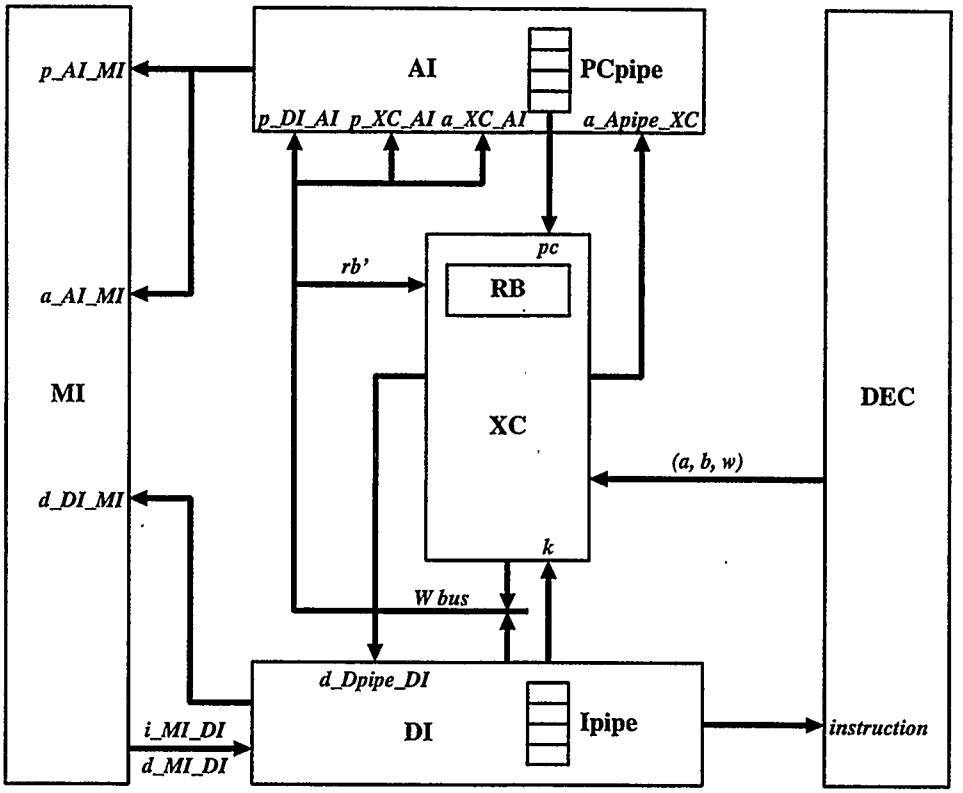


FIGURE 5.11. AMULET1 top level abstraction

5.4. Floor Plan Modules

This section presents formal CCS specification of all the floor plan modules compiled from their individual descriptions given in the previous section of how each instruction class uses the hardware.

5.4.1. Buses

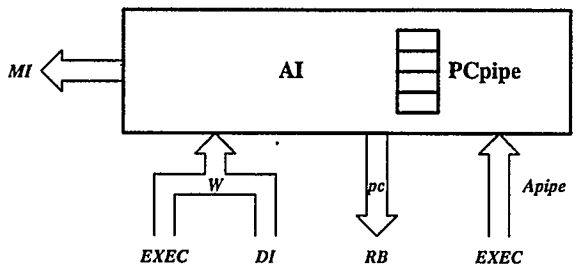
As illustrated in Figure 5.11, the buses at the top level abstraction fall into three categories according to the way they are accessed:

- single user access (e.g., Apipe, Dpipe);
- multi-user but naturally exclusive access (e.g., the bus between AI and MI shared by PC values and memory addresses);
- multi-user arbitrary access (e.g. W bus).

Buses for single user access are coded in CCS directly, buses for multi users with guaranteed exclusive access are expressed by "+" sign in CSS, and buses for multi users with arbitrary access use semaphore agents to guarantee exclusive access.

5.4.2. Address Interface and PC Pipe

In chapter 4 we abstracted the address interface as:



Since we cannot express value passing in CCS, we need to encode distinct accesses by distinct signals. Table 5.1 lists all possible accesses to address interface through each instruction class.

Instruction Class	Input to AI	Output from AI
Instruction Fetch	next PC	p_AI_MI, pc
Multiply Operation		
Data Operation	p_XC_AI	p_AI_MI
Branch Operation	p_XC_AI	p_AI_MI
Load Operation	a_XC_AI	a_AI_MI
	a_Apipe_AI	a_AI_MI
	p_DI_AI	p_AI_MI
Store Operation	a_XC_AI	a_AI_MI
	a_Apipe_AI	a_AI_MI
Load/Store Multiple	a_XC_AI	a_AI_MI
	p_DI_AI	p_AI_MI
Swap Operation	a_Apipe_AI	a_AI_MI
Software Interrupt	p_XC_AI	p_AI_MI

TABLE 5.1. The sharing of the address interface by instruction classes

The distinct cases are:

$$
\begin{aligned}
&\text{Instruction Fetch} && \Longrightarrow && \text{next PC} \longrightarrow \text{p\_AI\_MI, pc}\\
&\text{Address via W bus} && \Longrightarrow && \text{a\_XC\_AI} \longrightarrow \text{a\_AI\_MI}\\
&\text{Address via Apipe} && \Longrightarrow && \text{a\_Apipe\_AI} \longrightarrow \text{a\_AI\_MI}\\
&\text{PC from EXEC via W bus} && \Longrightarrow && \text{p\_XC\_AI} \longrightarrow \text{p\_AI\_MI}\\
&\text{PC from DI via W bus} && \Longrightarrow && \text{p\_DI\_AI} \longrightarrow \text{p\_AI\_MI}
\end{aligned}
$$

as depicted in Figure 5.12:

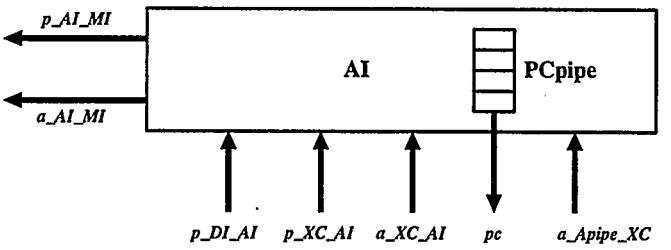


FIGURE 5.12. Top level abstraction of AMULET1 address interface

The address interface starts operation on being initialized. Its major function is to produce sequential instruction addresses to the memory interface autonomously through one of its sub-units AI1. In addition, it also transfers regular address supplied by the execution unit (a_XC_AI or a_Apipe_AI) to the memory interface through sub-unit AI2, or new PC value supplied by the execution unit (p_XC_AI) or the data interface (p_DI_AI) to the memory interface through sub-unit AI3. These three sub-units of the AI operate in parallel: each of them may accept new input while not in use, but whether or not the input value wins entry to the memory interface is competitive (the memory can only accept one input from the address interface at any time). In CCS we have

```
bi AI
   init.( AI1 | AI2 | AI3 )

bi AI1
   'incPC.'fetMEM.'p_AI_MI.AI1

bi AI2
   a_XC_AI.'a_AI_MI.AI2 + a_Apipe_AI.'a_AI_MI.AI2

bi AI3
   p_XC_AI.'newPC.'fetMEM.'p_AI_MI.AI3 + p_DI_AI.'newPC.'fetMEM.'p_AI_MI.AI3
```

The PCpipe associated with the address interface is incremented upon receiving a sequential PC value produced by the AI (incPC) or a new PC value dispatched via W bus (newPC caused by p_XC_AI or p_DI_AI), and decremented by the XC (decPC) when the PC value is no longer needed. To prevent deadlock, a PCpipe of depth n should always have a spare space saved for accepting a new PC value newPC, which leaves a maximum of $n-1$ spaces for the sequential PC value access incPC. The PCpipe should also be testable for its non-empty status via nePC. The CCS specification for a PCpipe of depth *2* follows.

```
bi PCpipe0
   incPC.PCpipe1 + newPC.PCpipe1

bi PCpipe1
                  newPC.PCpipe2 + decPC.PCpipe0 + nePC.PCpipe1

bi PCpipe2
                                  decPC.PCpipe1 + nePC.PCpipe2
```

5.4.3. Memory Interface

The off-chip memory serves as both instruction memory and data memory. As instruction memory, it supports instruction fetch. As data memory, it supports memory read and store. Table 5.2 lists all possible accesses to memory interface through each instruction class.

Instruction Class	Input to MI	Output from MI
Instruction Fetch	p_AI_MI	i_MI_DI
Multiply Operation		
Data Operation	p_AI_MI	i_MI_DI
Branch Operation	p_AI_MI	i_MI_DI
Load Operation	a_AI_MI	d_MI_DI
	p_AI_MI	i_MI_DI
Store Operation	a_AI_MI	
	d_DI_MI	
Load/Store Multiple	a_AI_MI	d_MI_DI
	p_AI_MI	i_MI_DI
	a_AI_MI	
	d_DI_MI	
Swap Operation	a_AI_MI	d_MI_DI
	d_DI_MI	
Software Interrupt	p_AI_MI	i_MI_DI

TABLE 5.2. The sharing of the memory interface by instruction classes

The distinct cases are:

$$\begin{aligned}
\text{Instruction fetch} &\Longrightarrow \text{p_AI_MI} \longrightarrow \text{i_MI_DI} \\
\text{Load} &\Longrightarrow \text{a_AI_MI} \longrightarrow \text{d_MI_DI} \\
\text{Store} &\Longrightarrow \text{a_AI_MI \& d_DI_MI} \\
\text{Swap} &\Longrightarrow \text{a_AI_MI \& d_DI_MI} \longrightarrow \text{d_MI_DI}
\end{aligned}$$

as depicted in Figure 5.13:

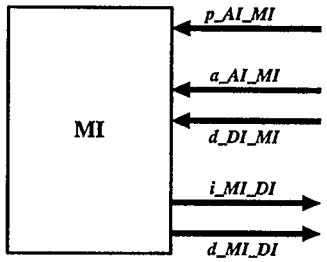


FIGURE 5.13. Top level abstraction of AMULET1 memory interface

Instruction fetch (fetMEM) starts receiving the current PC value (p_AI_MI), the corresponding instruction is dispatched to DI. Data load (ldrMEM) starts upon receiving the load address (a_AI_MI), the loaded data is also dispatched to DI. Data store strMEM) starts upon receiving both the data to be stored and location to store (d_DI_MI & a_AI_MI). Data swap (swpMEM) performs both data load and data store whilst keeping the memory bus locked until both the load and the store have been completed, data read out from the MI is dispatched to the W bus via DI. In CCS we have

```
bi MI
     fetMEM.p_AI_MI.'i_MI_DI.MI
   + ldrMEM.a_AI_MI.'d_MI_DI.MI
   + strMEM.d_DI_MI.a_AI_MI.MI
   + swpMEM.a_AI_MI.d_DI_MI.'d_MI_DI.MI
```

5.4.4. Data Interface and Instruction Pipe

In chapter 4 we abstracted the data interface as:

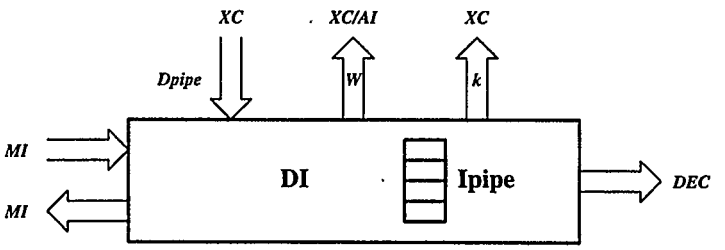


Table 5.3 lists all possible accesses to the data interface through each instruction class.

Instruction Class	Input to DI	Output from DI
Instruction Fetch	i_MI_DI	instruction
Multiply Operation		
Data Operation	i_MI_DI	instruction
Branch Operation	i_MI_DI	instruction
Load Operation	d_MI_DI	Rd or p_DI_AI via W bus
	i_MI_DI	instruction
Store Operation	d_Dpipe_DI	d_DI_MI
Load/Store Multiple	d_MI_DI	Rd or p_DI_AI via W bus
	i_MI_DI	instruction
	d_Dpipe_DI	d_DI_MI
Swap Operation	d_MI_DI	Rd via W bus
	d_Dpipe_DI	d_DI_MI
Software Interrupt	i_MI_DI	instruction

TABLE 5.3. The sharing of the data interface by instruction classes

The distinct cases are:

$$\begin{aligned}
&\text{Instruction fetch} \implies \text{i_MI_DI} \longrightarrow \text{instruction}\\
&\text{Load} \implies \text{d_MI_DI} \longrightarrow \text{Rd or p_DI_AI via W bus}\\
&\text{Store} \implies \text{d_Dpipe_DI} \longrightarrow \text{d_DI_MI}\\
&\text{Swap} \implies \text{d_MI_DI} \longrightarrow \text{Rd via W bus}
\end{aligned}$$

as depicted in Figure 5.14:

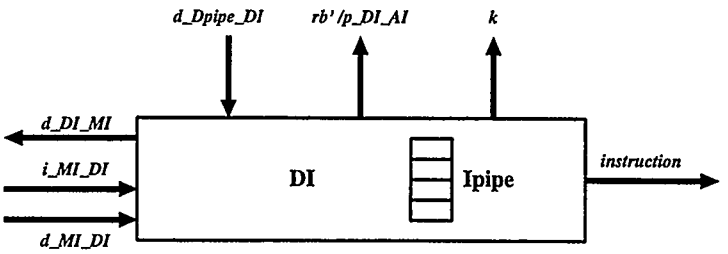


FIGURE 5.14. Top level abstraction of AMULET1 data interface

The data interface has a data input phase (DI_IN) and a data output phase (DI_OUT) operating in parallel. The data input phase (DI_IN) has two parts: DI_IN1 accepts an instruction and stores it in the Ipipe associated with DI; DI_IN2 accepts a datum loaded from the memory and dispatches it to the register bank or to the address interface according to its nature. These two parts of the data input phase also operate in parallel. The data output phase dispatches the incoming data value from Dpipe to MI. In CCS we have

```
bi DI
   ( DI_IN1 | DI_IN2 | DI_OUT )

bi DI_IN1
   i_MI_DI.'incIP.DI_IN1

bi DI_IN2
   d_MI_DI.'gW.(rb'.'pW.DI_IN2 + 'p_DI_AI.'pW.DI_IN2)

bi DI_OUT
   d_Dpipe_DI.'d_DI_MI.DI_OUT
```

The specification of Ipipe is similar to that of the PCpipe, except that all the increments are caused by the same source – the incoming instruction (incIP). The Ipipe is also testable for its non-empty status via neIP.

```
bi Ipipe0
   incIP.Ipipe1
```

```
bi Ipipe1
   incIP.Ipipe2 + decIP.Ipipe0 + neIP.Ipipe1

bi Ipipe2
                  decIP.Ipipe1 + neIP.Ipipe2
```

5.4.5. Decode Unit

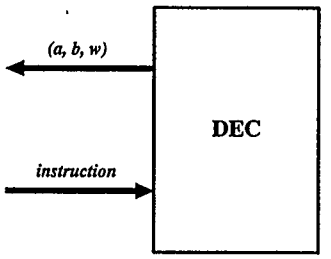


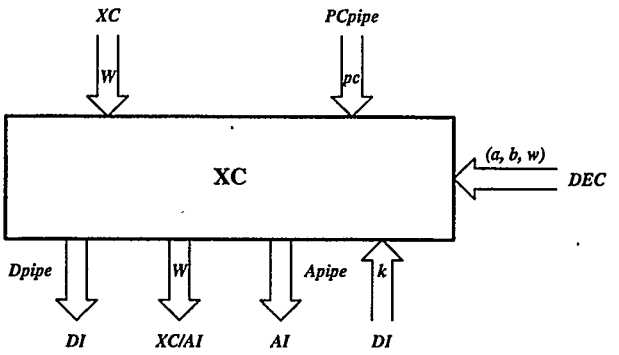
FIGURE 5.15. Top level abstraction of AMULET1 decode unit

The decode unit fetches instruction from Ipipe, provides register read and write back addresses (a, b, w) to the register bank, extracts the constant value k from Ipipe (not modeled and hence not shown in Figure 5.15). It also prepares control information to the execution unit (again, not shown in Figure 3.8) while instructions flow through the execution pipeline stages. Since every instruction has to be paired with its corresponding PC value for execution, the decode unit will not start operating until a matching pair has entered the PCpipe and Ipipe respectively. In the specification, this is expressed as neither PCpipe nor Ipipe being empty. The corresponding instruction is thrown away ('decIP) when the decode is completed. In CCS we have

```
bi DEC
   'nePC.'neIP.'decIP.( isMPY.'sMPY.DEC + isADD.'sADD.DEC
                      +   isB.  'sB.DEC +  isBL. 'sBL.DEC
                      + isLDR.'sLDR.DEC + isSTR.'sSTR.DEC
                      + isLSM.'sLSM.DEC + isSWP.'sSWP.DEC
                      + isSWI.'sSWI.DEC)
```

5.4.6. Execution Unit and the W Bus

In chapter 4 we abstracted the execution unit (with register bank) as:



The execution unit starts operating upon receiving the decoded information including (a, b, w) and k from DEC, and their matching PC value from PCpipe. The result produced by the XC is destined either for the AI via the W bus or the Apipe, or for the DI via the Dpipe. The execution results achieved through other floor plan modules (directly from the DI) are transferred to either the register bank (RB) or the AI via the W bus. By encoding the values as signals, we have,

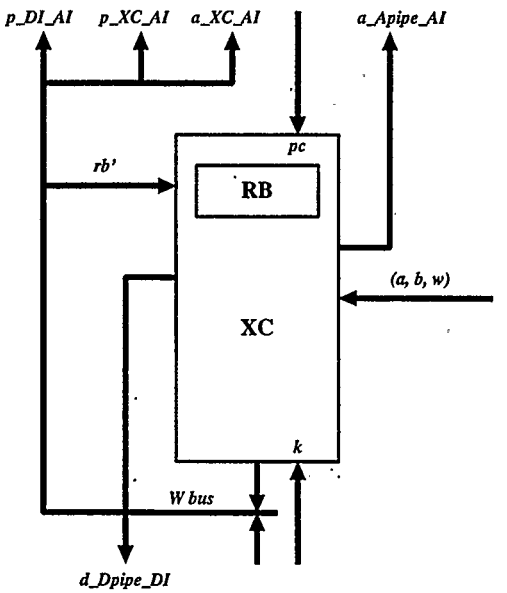


FIGURE 5.16. Top level abstraction of AMULET1 execution unit (with W bus)

Since all the instruction classes in AMULET1 can be conditionally executed, the corresponding CCS specification for the execution of each instruction class should all be modified by adding in a mutually exclusive item that permits instructions to be skipped. By doing so, we get the colour checking mechanism included for free – rejecting wrong colour instructions is just a subcase of conditional execution.

We have now detailed how each instruction class individually accesses the floor plan modules, and can present the modified CCS specification directly:

```
bi EXEC
     sMPY.EXEC_MPY + sADD.EXEC_ADD +   sB.EXEC_B
   +  sBL.EXEC_BL  + sLDR.EXEC_LDR + sSTR.EXEC_STR
   + sLSM.EXEC_LSM + sSWP.EXEC_SWP + sSWI.EXEC_SWI

bi EXEC_MPY
   'decPC.('gW.mpyRB.'pW.EXEC + noXC.EXEC)

bi EXEC_ADD
   'decPC.('gW.(addRB.'pW.EXEC + addAI.'p_XC_AI.'pW.EXEC) + noXC.EXEC)

bi EXEC_B
   'decPC.('gW.bAI.'p_XC_AI.'pW.EXEC + noXC.EXEC)

bi EXEC_BL
   'decPC.('gW.blAI.'p_XC_AI.'pW.'gW.blRB.'pW.EXEC + noXC.EXEC)

bi EXEC_LDR
   'decPC.('ldrMEM.ldrAI.( 'gW.'a_XC_AI.'pW.EXEC
                         + 'gW.'a_XC_AI.ldrRB.'pW.EXEC
                         + 'a_Apipe_AI.'gW.ldrRB.'pW.EXEC )
          + noXC.EXEC )

bi EXEC_STR
   'decPC.('strMEM.strDI.strAI.'d_XC_DI.( 'gW.'a_XC_AI.'pW.EXEC
                                        + 'gW.'a_XC_AI.strRB.'pW.EXEC
                                        + 'a_Apipe_AI.'gW.strRB.'pW.EXEC )
          + noXC.EXEC )
```

```
bi EXEC_LSM
   lsmAI.'gW.'a_XC_AI.'pW.'gW.lsmRB.'pW.
   (isLSM_LDR.EXEC_LSM_LDR + isLSM_STR.EXEC_LSM_STR)

bi EXEC_LSM_LDR
     noEND_LDR.'ldrMEM.'a_XC_AI.EXEC_LSM_LDR
   + isEND_LDR.'ldrMEM.'decPC.(lsmCPSR.EXEC + 'gW.lsmRB.'pW.EXEC)
   + noXC.EXEC

bi EXEC_LSM_STR
     noEND_STR.'strMEM.'d_XC_DI.'a_XC_AI.EXEC_LSM_STR
   + isEND_STR.'strMEM.'d_XC_DI.'decPC.EXEC
   + noXC.EXEC

bi EXEC_SWP
   'decPC.( 'swpMEM.swpAI.swpDI.'a_Apipe_AI.'d_XC_DI.EXEC
          + noXC.EXEC )

bi EXEC_SWI
   'decPC.('gW.swiAI.'p_XC_AI.'pW.'gW.swiRB1.'pW.'gW.swiRB2.'pW.EXEC)
          + noXC.EXEC )
```

5.5. Testing the Specification

So far, we have presented formal CCS specifications for all the floor plan modules of AMULET1 at system level, as well as how each instruction class is decoded and executed. In this section, we test the specifications on the Concurrency Workbench using the macro-based testing style proposed in [Liu92, LABS93]. The tests were carried out first on individual instruction classes, and then on the complete instruction set.

5.5.1. Individual Instruction Classes

The testing on individual instruction classes assumes that the processor only executes one class of instruction, i.e. once it is ready to accept another instruction, it will receive another from the same instruction class. The testing was carried out using the following steps:

(1) Specifying the library elements including floor plan modules and buses (these elements are common and shared by all the instruction classes).
(2) Specifying the execution flow of a specific instruction class.
(3) Specifying the processor in terms of one specific instruction class.
(4) Testing the specified processor for basic properties such as sort, size, traces and etc.
(5) Testing the specified processor for deadlock free, livelock free, and other desired safety and liveness properties.

The importance of testing on individual instruction classes is to verify that each of these instructions flow through the processor as desired. During the testing procedure, suitable trace values were added in to help test specific safety and liveness properties. These trace values could be deleted later on when correctness has been proved. As an example, we show how the load instruction is tested on the CWB. Testing details for other instruction classes are similar.

(1) Floor plan modules and buses with suitable trace values added in for testing:

```
bi AI
   init.( AI1 | AI2 | AI3 )

bi AI1
   'incPC.'fetMEM.'p_AI_MI.AI1

bi AI2
   a_XC_AI.'a_AI_MI.AI2 + a_Apipe_AI.'a_AI_MI.AI2

bi AI3
   p_XC_AI.'newPC.'fetMEM.'p_AI_MI.AI3 + p_DI_AI.'newPC.'fetMEM.'p_AI_MI.AI3

bi PCpipe0
   incPC.PCpipe1 + newPC.PCpipe1
```

```
bi PCpipe1
                    newPC.PCpipe2 + decPC.PCpipe0 + nePC.PCpipe1

bi PCpipe2
                                    decPC.PCpipe1 + nePC.PCpipe2

bi MI
     fetMEM.p_AI_MI.fetch.'i_MI_DI.MI
   + ldrMEM.a_AI_MI.read.'d_MI_DI.MI
   + strMEM.a_AI_MI.d_DI_MI.store.MI
   + swpMEM.a_AI_MI.d_DI_MI.read.'d_MI_DI.store.MI

bi DI
   ( DI_IN1 | DI_IN2 | DI_OUT )

bi DI_IN1
   i_MI_DI.'incIP.DI_IN1

bi DI_IN2
   d_MI_DI.'gW.(rb'.'pW.DI_IN2 + 'p_DI_AI.'pW.DI_IN2)

bi DI_OUT
   d_Dpipe_DI.'d_DI_MI.DI_OUT

bi Ipipe0
   incIP.Ipipe1

bi Ipipe1
   incIP.Ipipe2 + decIP.Ipipe0 + neIP.Ipipe1

bi Ipipe2
                   decIP.Ipipe1 + neIP.Ipipe2

bi WBUS
   gW.pW.WBUS
```

(2) Execution flow of the load instruction:

```
bi DEC
   'nePC.'neIP.sD.'decIP.isLDR.'sLDR.DEC

bi EXEC
   sLDR.EXEC_LDR

bi EXEC_LDR
   'decPC.( 'ldrMEM.ldrAI.( 'gW.'ldr_a_XC_AI.'pW.EXEC
                          + 'gW.'ldr_a_XC_AI.ldrRB.'pW.EXEC
                          + 'ldr_a_Apipe_AI.'gW.ldrRB.'pW.EXEC )
          + noXC.EXEC )
```

(3) AMULET1 in terms of the load operation only:

```
bi AMULET1_LDR
   ( AI | PCpipe0 | MI | DI | Ipipe0 | WBUS | DEC | EXEC )
   \ { a_AI_MI, a_Apipe_AI, a_XC_AI, d_DI_MI, d_Dpipe_DI, d_MI_DI, decIP,
       decPC, fetMEM, gW, i_MI_DI, incIP, incPC, ldrMEM, neIP, nePC, newPC,
       pW, p_AI_MI, p_DI_AI, p_XC_AI, sLDR, store, strMEM, swpMEM }
```

(4) Testing AMULET1_LDR for basic properties:[1]

```
sort AMULET1_LDR
**{fetch,init,isLDR,ldrAI,ldrRB,noXC,rb',read,sD}

min AMULET1_LDR
Save result in identifier: AMULET1_LDR'
**AMULET1_LDR' has 699 states. .
```

(5) Testing the specified processor for desired safety and liveness properties (selected examples):

(a) Deadlock free

```
cp AMULET1_LDR'
Proposition: BOX <->T
**true
```

[1]Nothing should be read into the size of the minimized AMULET1_LDR. It only reflects the number of trace values we are interested in.

(b) Livelock free

```
cp AMULET1_LDR
Proposition: ~ POSS BOX <t>T
**true
```

(c) The data read phase (read) in the load operation will always possible be performed, but not always eventually be performed (discarded).

```
cp AMULET1_LDR'
Proposition: BOX (POSS <read>T)
**true

cp AMULET1_LDR'
Proposition: BOX (EVENT <read>T)
**false
```

(d) It is always possible for the instruction to be discarded (noXC).

```
Command: cp AMULET1_LDR'
Proposition: BOX (POSS <noXC>T)
**true
```

The specification displayed here has the complexity measure tabulated in Table 5.4:

Component sizes								Number of	Minimized
AI	MI	DI	DEC	EXEC	PCpipe	Ipipe	WBUS	states	states
25	14	24	6	10	3	3	2	9,072,000	699

TABLE 5.4. Complexity measure for top level load operation (new)

where

Component sizes lists the number of states of each individual component;

Number of States is the number of unminimized states of AMULET1_LDR;

Minimized States shows the number of minimized states of AMULET1_LDR.

The corresponding specification for the load operation presented in [BL95] has the complexity measure tabulated below in Table 5.5. But note that the latter failed to minimize.

Component sizes							Number of states	Minimized states
AI	MI	DI	DEC	EXEC	EXECpipe	WBUS		
110	5	37	3	26^2	16	2	440,112,200	still running!

TABLE 5.5. Complexity measure for top level load operation (old)

The specification presented here successfully models this instruction class previously found intractable due to an explosive number of states mainly caused by the complexity of the execution unit and the execution pipeline.

Although the above specifications may seem obvious, they took many iterations to achieve. The following is one of our many attempts in modeling the load operation, which was found to have potential deadlocks on the CWB. The difference in the following version to the one presented above lies in who signals the memory interface to do a memory read. In the above presented version, it is the execution unit that signals the memory interface (ldrMEM) to wait for a load address so as to perform a memory read. In the following specification, it is the address interface that signals the memory interface (ldrMEM) to perform a memory read upon receiving a load address. Hence the specifications for the address interface and the execution unit are modified while the rest of the modules remain the same.

```
bi AI
   init.( AI1 | AI2 | AI3 )

bi AI1
   'incPC.'fetMEM.'p_AI_MI.AI1

bi AI2
   ldr_a_Apipe_AI.'ldrMEM.'a_AI_MI.AI2 + ldr_a_XC_AI.'ldrMEM.'a_AI_MI.AI2

bi AI3
   p_XC_AI.'newPC.'fetMEM.'p_AI_MI.AI3 + p_DI_AI.'newPC.'fetMEM.'p_AI_MI.AI3
```

```
bi EXEC_LDR
   'decPC.( ldrAI.( 'gW.'ldr_a_XC_AI.'pW.EXEC
                  + 'gW.'ldr_a_XC_AI.ldrRB.'pW.EXEC
                  + 'ldr_a_Apipe_AI.'gW.ldrRB.'pW.EXEC )
          + noXC.EXEC )

bi AMULET1_LDR
   ( AI | PCpipe0 | MI | DI | Ipipe0 | WBUS | DEC | EXEC )
   \ { a_AI_MI, d_DI_MI, d_Dpipe_DI, d_MI_DI, decIP, decPC, fetMEM, gW,
       i_MI_DI, incIP, incPC, ldrMEM, ldr_a_Apipe_AI, ldr_a_XC_AI, neIP, nePC,
       newPC, pW, p_AI_MI, p_DI_AI, p_XC_AI, sLDR, store, strMEM, swpMEM }
```

This specification is tested on the CWB and found to have potential deadlocks.

```
sort AMULET1_LDR
**{fetch,init,isLDR,ldrAI,ldrRB,noXC,rb',read,sD}

min AMULET1_LDR
AMULET1_LDR'
**AMULET1_LDR' has 1887 states.

fd AMULET1_LDR'
**--- init fetch sD isLDR t<incPC> t<decPC> fetch ldrAI t<ldr_a_Apipe_AI>
      sD isLDR t<pW> t<ldrMEM> ldrAI t<pW> fetch read sD isLDR t<pW>
      t<d_MI_DI> ldrAI t<gW> fetch read sD isLDR t<incPC> ldrAI t<gW>
      ---> AMULET1_LDR'minState7158
```

The cause of the deadlock can be deduced from the trace of the deadlock information provided by the CWB. The execution unit is free to accept another load address as soon as the first address is dispatched to the address interface, hence the W bus is seized for this purpose. However this new address cannot enter the address interface until the data loaded from the memory has been accepted by the data interface, but the data interface cannot accept the loaded data since it is waiting to use the W for dispatching the loaded data.

The real cause of deadlock is rooted in the fact that the W bus is shared by both the execution unit and the data interface. If the data interface had its own bus for data dispatching, this deadlock would vanish.

5.5.2. Complete Instruction Set

The testing of the complete instruction set is expected to reflect the fact that the processor must be prepared to execute any instruction in its repertoire as the next instruction. However, due to the enormous number of reachable states when all the instruction classes are put together, the minimization can not be carried out on the accessible hardware. The following is the processor with its complete instruction set tested on the CWB part by part.

(1) In order to reduce the number of reachable states during the minimization procedure, the DI_OUT sub-unit in the DI is replaced with a single bus d_XC_DI_MI from the EXEC via the DI to the MI since it is always accessed by the same single user. Further, the number of visible trace values in the floor plan modules and buses are reduced to minimal.

```
bi AI
   init.( AI1 | AI2 | AI3 )

bi AI1
   'incPC.'fetMEM.'p_AI_MI.AI1

bi AI2
   a_XC_AI.'a_AI_MI.AI2 + a_Apipe_AI.'a_AI_MI.AI2

bi AI3
   p_XC_AI.'newPC.'fetMEM.'p_AI_MI.AI3 + p_DI_AI.'newPC.'fetMEM.'p_AI_MI.AI3

bi PCpipe0
   incPC.PCpipe1 + newPC.PCpipe1

bi PCpipe1
                  newPC.PCpipe2 + decPC.PCpipe0 + nePC.PCpipe1

bi PCpipe2
                                  decPC.PCpipe1 + nePC.PCpipe2
```

```
bi MI
     fetMEM.p_AI_MI.'i_MI_DI.MI
   + ldrMEM.a_AI_MI.'d_MI_DI.MI
   + strMEM.d_XC_DI_MI.a_AI_MI.MI
   + swpMEM.a_AI_MI.d_XC_DI_MI.'d_MI_DI.MI

bi DI
   ( DI_IN1 | DI_IN2 )

bi DI_IN1
   i_MI_DI.'incIP.DI_IN1

bi DI_IN2
   d_MI_DI.'gW.('pW.DI_IN2 + 'p_DI_AI.'pW.DI_IN2)

bi Ipipe0
   incIP.Ipipe1

bi Ipipe1
   incIP.Ipipe2 + decIP.Ipipe0 + neIP.Ipipe1

bi Ipipe2
                  decIP.Ipipe1 + neIP.Ipipe2

bi WBUS
   gW.pW.WBUS
```

(2) Since the multiply operation and branch operation can be viewed as a subset of the data operation, only the data operation is included in the following testing to reduce the number of reachable states. For the same purpose, the load/store multiple operation is excluded in the following testing since it is covered by the load operation combined with store operation. By doing so, the nine instruction classes in AMULET1 are reduced to six. These six instruction classes are tested in two groups.

(a) Group I: ADD, BL and SWI

```
bi DEC
   'nePC.'neIP.'decIP.(isADD.'sADD.DEC + isBL. 'sBL.DEC + isSWI.'sSWI.DEC)

bi EXEC
   sADD.EXEC_ADD + sBL.EXEC_BL + sSWI.EXEC_SWI

bi EXEC_ADD
   'decPC.'gW.('pW.EXEC + 'p_XC_AI.'pW.EXEC)

bi EXEC_BL
   'decPC.'gW.'p_XC_AI.'pW.'gW.'pW.EXEC

bi EXEC_SWI
   'decPC.'gW.'p_XC_AI.'pW.'gW.'pW.'gW.'pW.EXEC

bi AMULET1_ADD_BL_SWI
   ( AI | PCpipe0 | MI | DI | Ipipe0 | WBUS | DEC | EXEC )
   \ { a_AI_MI, a_Apipe_AI, a_XC_AI, d_MI_DI, d_XC_DI_MI, decIP, decPC,
       fetMEM, gW, i_MI_DI, incIP, incPC, ldrMEM, neIP, nePC, newPC, pW,
       p_AI_MI, p_DI_AI, p_XC_AI,sADD, sB, sBL, sLDR, sLSM, sMPY, sSTR,
       sSWI, sSWP, strMEM, swpMEM }

sort AMULET1_ADD_BL_SWI
**{init,isADD,isBL,isSWI}

min AMULET1_ADD_BL_SWI
AMULET1_ADD_BL_SWI'
**AMULET1_ADD_BL_SWI' has 2 states.

vs 3 AMULET1_ADD_BL_SWI'
**=== init isADD isADD ===>
**=== init isADD isBL ===>
**=== init isADD isSWI ===>
**=== init isBL isADD ===>
**=== init isBL isBL ===>
```

```
**=== init isBL isSWI ===>
**=== init isSWI isADD ===>
**=== init isSWI isBL ===>
**=== init isSWI isSWI ===>
```

(b) Group II: LDR, STR and SWP

```
bi DEC
   'nePC.'neIP.'decIP.(isLDR.'sLDR.DEC + isSTR.'sSTR.DEC + isSWP.'sSWP.DEC)

bi EXEC
   sLDR.EXEC_LDR + sSTR.EXEC_STR + sSWP.EXEC_SWP

bi EXEC_LDR
   'decPC.'ldrMEM.('gW.'a_XC_AI.'pW.EXEC + 'a_Apipe_AI.'gW.'pW.EXEC)

bi EXEC_STR
   'decPC.'strMEM.'d_XC_DI_MI.
   ('gW.'a_XC_AI.'pW.EXEC + 'a_Apipe_AI.'gW.'pW.EXEC)

bi EXEC_SWP
   'decPC.'swpMEM.'a_Apipe_AI.'d_XC_DI_MI.EXEC

bi AMULET1_LDR_STR_SWP
   ( AI | PCpipe0 | MI | DI | Ipipe0 | WBUS | DEC | EXEC )
   \ { a_AI_MI, a_Apipe_AI, a_XC_AI, d_MI_DI, d_XC_DI_MI, decIP, decPC,
       fetMEM, gW, i_MI_DI, incIP, incPC, ldrMEM, neIP, nePC, newPC, pW,
       p_AI_MI, p_DI_AI, p_XC_AI,sADD, sB, sBL, sLDR, sLSM, sMPY, sSTR,
       sSWI, sSWP, strMEM, swpMEM }

sort AMULET1_LDR_STR_SWP
**{init,isADD,isBL,isSWI}

min AMULET1_LDR_STR_SWP
AMULET1_LDR_STR_SWP'
**AMULET1_LDR_STR_SWP' has 2 states.
```

```
vs 3 AMULET1_LDR_STR_SWP'
**=== init isLDR isLDR ===>
**=== init isLDR isSTR ===>
**=== init isLDR isSWP ===>
**=== init isSTR isLDR ===>
**=== init isSTR isSTR ===>
**=== init isSTR isSWP ===>
**=== init isSWP isLDR ===>
**=== init isSWP isSTR ===>
**=== init isSWP isSWP ===>
```

(3) Testing AMULET1 for safety and liveness: since the visible trace values have been reduced to minimal, only freedom from deadlock and freedom from livelock are tested.

(a) Deadlock free

```
cp AMULET1_ADD_BL_SWI'
Proposition: BOX <->T
**true

cp AMULET1_LDR_STR_SWP'
Proposition: BOX <->T
**true
```

(b) Livelock free

```
cp AMULET1_ADD_BL_SWI
Proposition:  ~ POSS BOX <t>T
**true

cp AMULET1_LDR_STR_SWP
Proposition:  ~ POSS BOX <t>T
**true
```

5.6. Summary

This chapter presents a top level specification of AMULET1 in terms of the processor's major floor plan modules. The behaviour of the processor is precisely and efficiently modeled

in terms of how each instruction class flows through the processor. The processor is further verified to be deadlock free and livelock free.

The main contribution of this chapter is finding the right level of abstraction which makes it possible to model the behaviour of a realistic processor at the system level, particularly, the abstraction of the decode unit and the execution unit. The decode unit was greatly simplified from the implementation in which it has three pipelined stages associated with the execution pipelines. The individual pipeline stages of the execution unit were also abstracted away. These are the keys to reduce the number of states and make the top level model tractable. Compared to the specifications presented in [BL95], the critical improvements are:

- **Suitable level of abstraction for execution unit**

  The execution unit specified in [BL95] models execution details on how instruction class flows through the execution pipeline stages: how each pipeline stage is seized and released. This level of detail made it intractable even for modeling the load operation, never mind the complete instruction set. The model presented in this chapter abstracted away all the details regarding how each pipeline stage is accessed, but preserved all the access needs of each instruction class while flowing through the execution unit.

- **Encoding conditional execution**

  The specification presented here systematically models conditional execution which was incomplete in [BL95]. With condition test, an operation can be cancelled even after the datum has been dispatched to the corresponding buses (e.g., Apipe, Dpipe).

- **Eliminating colour checking**

  Since all the instructions have to pass a conditional test before the execution results become valid (e.g. write back to the register bank), colour checking is naturally included in the conditional test. The explicit colour checking presented in [BL95] was redundant. It was also one of the major causes of its state explosion.

This top level abstraction of AMULET1 not only gives a good intuition of how instructions flow through the processor, but also provides a good understanding of the functionality of each floor plan module. It is a useful guide to system designers when using AMULET1

in an embedded system and to the implementors of floor plan modules who now have not only a good understanding of the role of each module but also complete tabulations of the source of each input signal and the destination of each output signal.

Through specification and property testing, CCS successfully demonstrates itself to be appropriate and efficiency in modeling a complex and practical asynchronous design at the system level. In the following chapters, the major floor plan modules in AMULET1 will be specified and tested in register transfer level detail.

CHAPTER 6
Address Interface

This and the next two chapters are concerned with modeling the major floor plan modules in AMULET1 at the register transfer level. In these chapters, we reduce the complexity in so far as we can by abstracting away the details of all regular and well understood datapath modules (e.g. incrementer, ALU, and FIFOs), and by lifting out and collecting irregular decision logics in one place. The models presented focus on the communication and synchronization of netlists comprising these abstracted datapath modules and standard library blocks.

The first effort at specifying AMULET1 was at this level of abstraction. After a few iterations it became clear that a higher level of abstraction — the top level presented in chapter 5 — was needed in order to understand the operating environment of each floor plan module and how these modules interact with each other. However, the top level modeling was made much easier by this ground work.

This chapter presents a model of the address interface at the register transfer level. The address interface has three distinct purposes, namely producing sequential instruction addresses to the memory interface through an autonomous PC incrementing loop, transferring a single address for instructions such as branch, load, etc., and generating and transferring multiple addresses for load/store multiple operations.

The modeling of the address interface starts from understanding the data flow activities for each individual access to the address interface. The control flow of these accesses to the address interface is then developed, specified, and tested.

6.1. Internal Organization

The internal organization of the address interface is shown in Figure 6.1. Inputs to the address interface are regular addresses and PC values that come from either the W bus or the Apipe. All these regular addresses and PC values are destined for the MI, but PC values sent to the PC register are also dispatched to PCpipe associated with the address interface (this is not included in the following description since the flow through the PCpipe is straightforward).

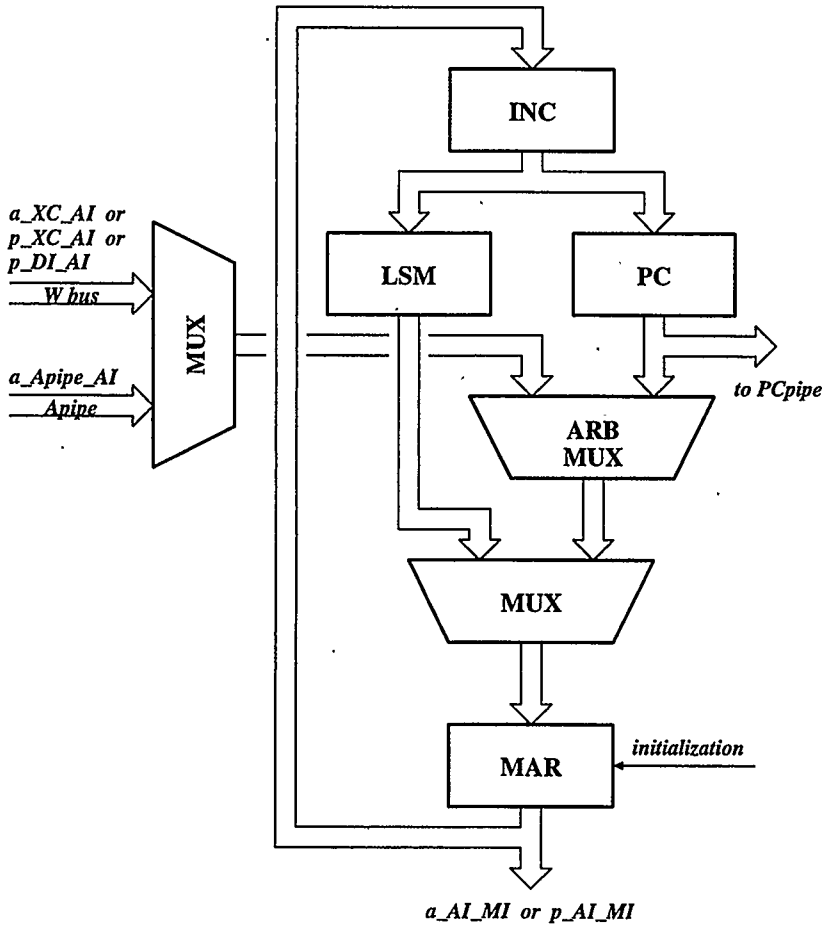


FIGURE 6.1. Internal organization of the address interface

The main data path modules inside the address interface include a PC register that stores the current PC value for fetching the corresponding instruction from the memory, an LSM register that stores the current address during a load/store multiple operation, and a memory address register MAR that contains the next address to be dispatched to the memory. The contents of the PC register and the LSM register are generated by incrementing the current MAR contents through the incrementer, INC.

Mutually exclusive requests to the MAR are filtered by a regular multiplexor; competing requests requiring access to the MAR are filtered by an arbitrating multiplexor. Exclusive access to the MAR is thus guaranteed at any time.

6.2. Data Flow of Accesses to the Address Interface

The accesses to the address interface can be split into three distinct cases according to how they use the data path modules inside the address interface. They are:

- PC incrementing loop (PCL) for instruction fetch;
- Single address transfer (SAT) for transferring a regular address in LDR, STR, and SWP operations, or a new PC value in ADD, B, BL, LDR, and SWI operations;
- Multiple address transfer (MAT) for transferring multiple address in LSM operation.

These cases are discussed below in separate subsections.

6.2.1. PC Incrementing Loop

The PC incrementing loop generates the next sequential PC value for instruction fetch by circulating the contents of PC register around an incrementing loop. Each PC value generated by the loop is also copied to the PCpipe associated with the address interface as it is dispatched to the memory interface. Figure 6.2 illustrates the data flow.

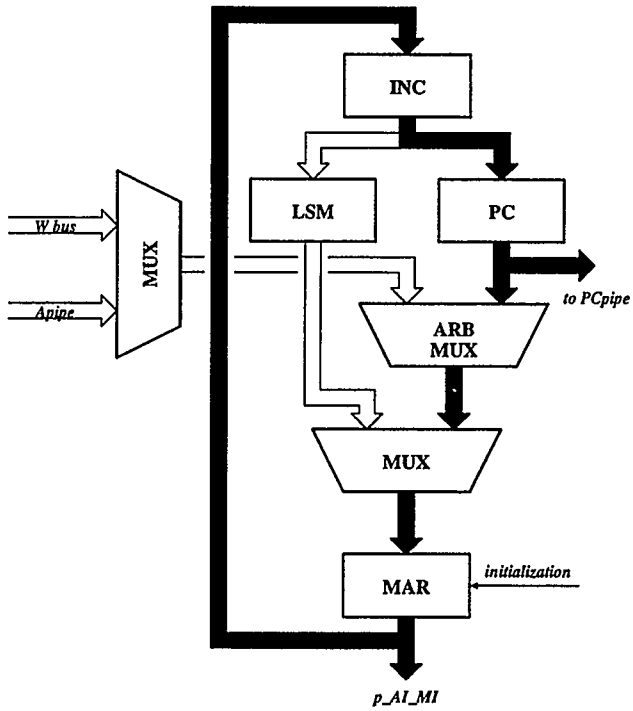


FIGURE 6.2. Data flow of PC incrementing loop

The PC incrementing loop has two phases: the initialization phase and the continuation phase.

- The *initialization phase* begins with the MAR being reset to zero. This is the first value to be dispatched to the MI. This first value is also simultaneously incremented and saved in the PC register[1].
- The *continuation phase* begins with the contents in the PC register being sent to the MAR via the arbitrating multiplexors. Once latched in the MAR, this PC value is then dispatched to the MI via the MAR, and to the PC register after being incremented (for the incrementing loop to continue on).

The behaviour of the PC incrementing loop can be informally summarized as:

$$\begin{aligned} \mathsf{PCL} &\overset{init}{\Longrightarrow} \mathsf{MAR(0)} \longrightarrow \mathsf{MI(mar)} \parallel \mathsf{INC(mar).PC(inc)} \\ &\overset{cont^*}{\Longrightarrow} \mathsf{MAR(pc)} \longrightarrow \mathsf{MI(mar)} \parallel \mathsf{INC(mar).PC(inc)} \end{aligned}$$

[1]This first PC value will not be placed in the PCpipe.

6.2.2. Single Address Transfer

Single address transfer interrupts the PC incrementing loop in order to transfer a regular address or a new PC value to the MI. The regular address is generated by the load, store, and swap operations. The new PC value is generated by add, branch, load, and software interrupt operations that supply a new PC address for fetching the next instruction. Figure 6.3 illustrates the possible data flows.

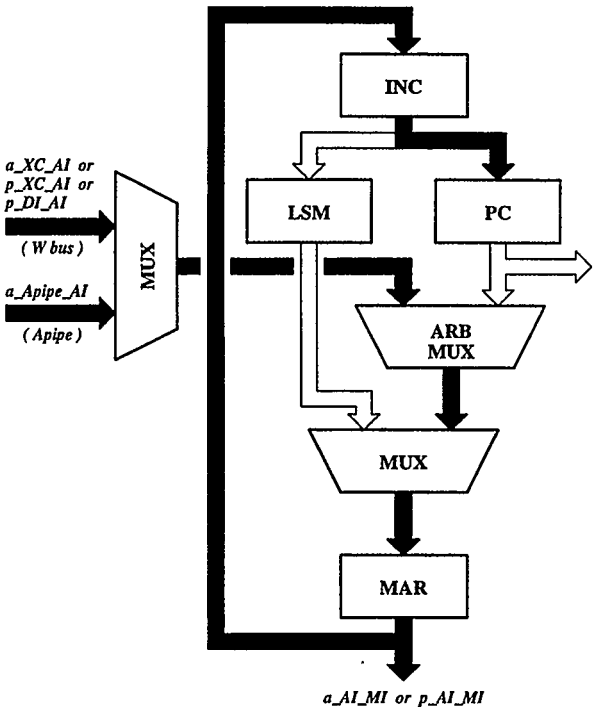


FIGURE 6.3. Data flow of single address transfer

The single address transfer begins with an address or a new PC value being supplied on either the W bus or the APIPE. When it wins the arbitrating multiplexor, the PC incrementing loop is temporarily interrupted and the regular address or the new PC value enters the MAR.

- If a regular address is to be transferred, it is dispatched to the MI only.
- If a new PC value is to be transferred, it is dispatched to the MI and also to the PC register after being incremented.

The behaviour of the single address transfer can be summarized as:

$$\begin{array}{lll} \mathsf{SAT} \overset{isPC}{\Longrightarrow} \mathsf{MAR(w)} & \longrightarrow & \mathsf{MI(mar)} \parallel \mathsf{INC(mar).PC(inc)} \\ \overset{ntPC}{\Longrightarrow} \mathsf{MAR(w,Apipe)} & \longrightarrow & \mathsf{MI(mar)} \end{array}$$

where

isPC represents a new PC value supplied by the W bus only;

ntPC represents a regular address supplied by either the W bus or the Apipe.

6.2.3. Multiple Address Transfer

Multiple address transfer generates and transfers a set of sequential instruction addresses based upon a base value supplied to the address interface while executing the load/store multiple operation. Figure 6.4 illustrates the data flow.

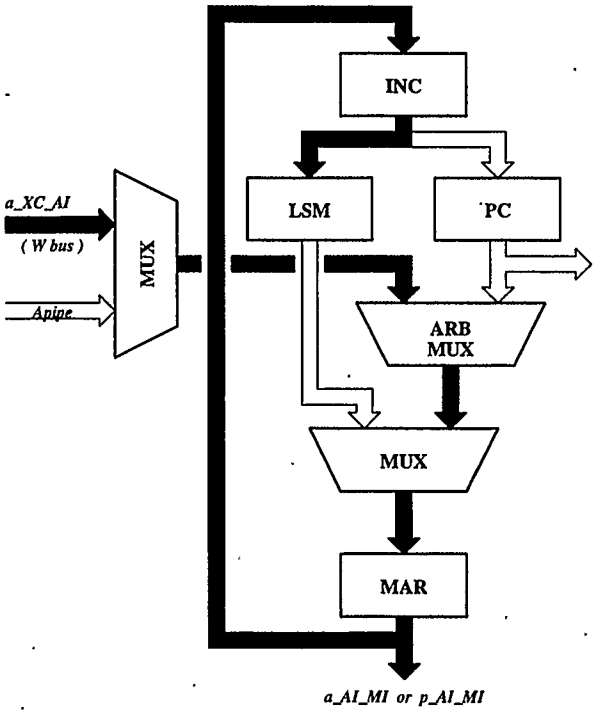


FIGURE 6.4. Data flow of multiple address transfer

The multiple address transfer also has a initialization phase and a continuation phase.

- The *initialization phase* begins with the base address arriving on the W bus. When it wins the arbitrating multiplexor, the PC incrementing loop is temporarily interrupted and the base address enters the MAR. The address interface then uses the information from the decode unit to detect whether this is the end of the transfer. If this is the last address to be transferred, it is dispatched to the MI only and the multiple address transfer completes. If not, it is dispatched to both the MI and to the LSM register after being incremented. The multiple address transfer then enters its continuation phase.
- The *continuation phase* of the multiple address transfer begins with the contents in the LSM register being dispatched to the MAR directly without any contention from other resources (the multiple address transfer is the sole user of the address interface until the end of the transfer is detected and completed). The address interface again uses the information from the decode unit to detect whether this is the end of the transfer. If the end is detected, the address is dispatched to the MI only and the multiple address transfer completes. If not, the address is dispatched to both the MI and the LSM register after being incremented, and the multiple address transfer continues.

The behaviour of the multiple address transfer can be summarized as:

$$
\begin{array}{lllll}
\mathsf{MAT} & \stackrel{init}{\Longrightarrow} & \mathsf{MAR(w)} & \stackrel{isE}{\longrightarrow} & \mathsf{MI(mar)} \\
 & & & \stackrel{ntE}{\longrightarrow} & \mathsf{MI(mar)} \parallel \mathsf{INC(mar).LSM(inc)} \\
 & \stackrel{cont}{\Longrightarrow}^{*} & \mathsf{MAR(lsm)} & \stackrel{isE}{\longrightarrow} & \mathsf{MI(mar)} \\
 & & & \stackrel{ntE}{\longrightarrow} & \mathsf{MI(mar)} \parallel \mathsf{INC(mar).LSM(inc)}
\end{array}
$$

where

isE represents the end of a multiple address transfer being detected;

ntE represents the end of a multiple address transfer not yet detected.

6.2.4. Summary of Accesses

In summary, the accesses to the address interface are

PCL	$\overset{init}{\Longrightarrow}$	MAR(0)	$\longrightarrow$	MI(mar) || INC(mar).PC(inc)	... 1
	$\overset{cont^*}{\Longrightarrow}$	MAR(pc)	$\longrightarrow$	MI(mar) || INC(mar).PC(inc)	... 2
SAT	$\overset{isPC}{\Longrightarrow}$	MAR(w)	$\longrightarrow$	MI(mar) || INC(mar).PC(inc)	... 3
	$\overset{ntPC}{\Longrightarrow}$	MAR(w,Apipe)	$\longrightarrow$	MI(mar)	... 4
MAT	$\overset{init}{\Longrightarrow}$	MAR(w)	$\overset{isE}{\longrightarrow}$	MI(mar)	... 5 (5a)
			$\overset{ntE}{\longrightarrow}$	MI(mar) || INC(mar).LSM(inc)	... 5 (5b)
	$\overset{cont^*}{\Longrightarrow}$	MAR(lsm)	$\overset{isE}{\longrightarrow}$	MI(mar)	... 6 (6a)
			$\overset{ntE}{\longrightarrow}$	MI(mar) || INC(mar).LSM(inc)	... 6 (6b)

The PCL, the SAT, and the MAT may require access to the address interface arbitrarily, but only one of these accesses can be granted whenever there is a contention. General operating sequences are

(1) The address interface is invoked by the initialization phase of the PCL during which the MAR is reset to zero.

(2) Once the address interface is initialized, it enters the continuation phase of the PCL. This is the main function of the address interface.

(3) However, after the initialization phase is completed, both the SAT and the MAT can compete with the continuation phase of the PCL for accessing the address interface. If the SAT or the MAT wins the access, the PCL will be temporarily interrupted.

6.3. Control Flow of Accesses to the Address Interface

The six distinct accesses to the address interface evaluated in the previous section share the common datapath modules in the address interface. The control flow of these accesses has to reflect this common sharing. This section presents the control flow of accesses to the address interface at the register transfer level.

6.3.1. Intuitive Overview

Figure 6.5 gives an intuitive presentation of the control flow of accesses to the address interface.

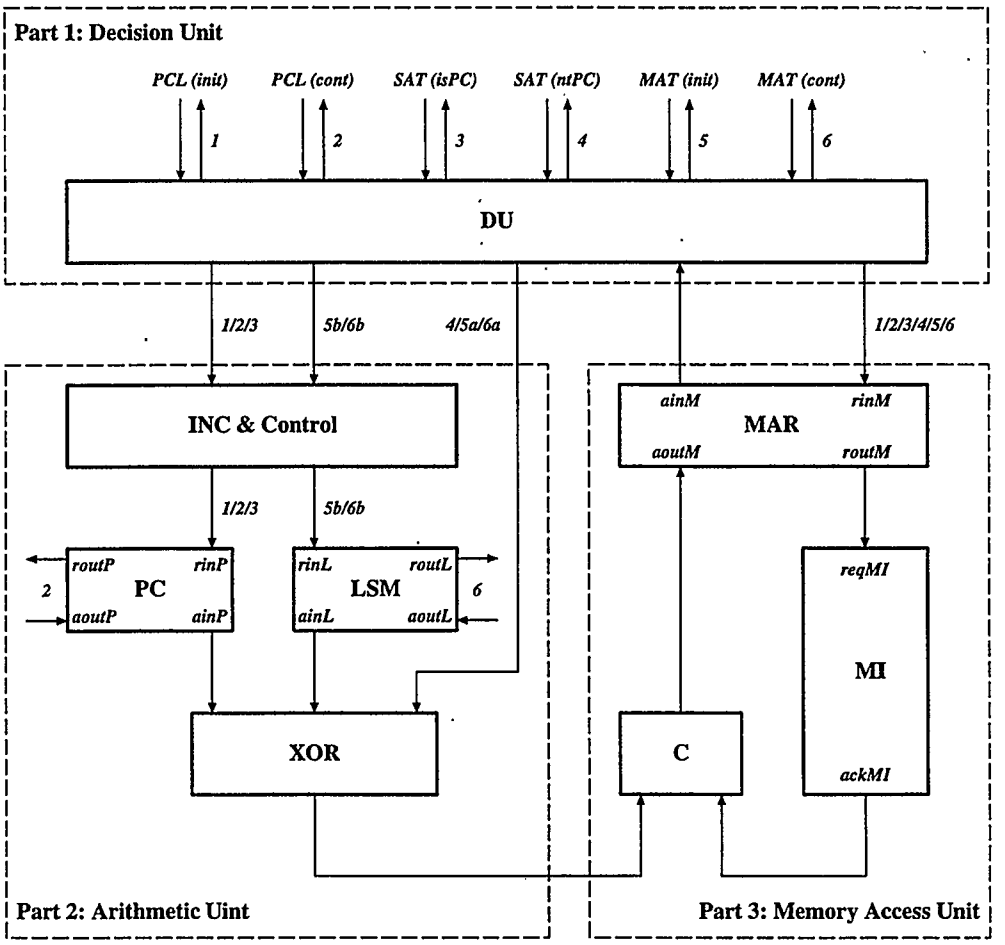


FIGURE 6.5. Intuitive overview: control flow of accesses to the address interface

In Figure 6.5, the address interface hardware is partitioned into three basic units, namely the decision unit, the arithmetic unit, and the memory access unit.

- The *decision unit* guarantees that at any time, only one of the six distinct access requests to the address interface can be granted. It also generates control signals for accessing the arithmetic unit and the memory access unit:
  - inputs of all the access requests, cases *1, 2, 3, 4, 5, 6*, are dispatched to the memory interface[2];
  - inputs of cases *1, 2, 3* are dispatched to the MAR and then to the PC register;
  - inputs of cases *5b, 6b* are dispatched to the MAR and then to the LSM register;
  - inputs of cases *4, 5a, 6a* are dispatched to the memory interface only, they have no access to the arithmetic unit.
- The *arithmetic unit* accepts access requests from the decision unit:
  - inputs of cases *1, 2, 3* are incremented and then dispatched to the PC register;
  - inputs of cases *5a, 6a* are incremented and then dispatched to the LSM register;
  - inputs of cases *4, 5b, 6b* bypass the INC, the PC register, and the LSM register.

  These access requests are routed through an XOR module after their corresponding accesses have been carried out.
- The *memory access unit* accepts access requests from the decision unit: inputs of cases *1, 2, 3, 4, 5, 6* are dispatched to the memory interface via the MAR.

Acknowledgements to the MAR are collected by a C-element after the corresponding access needs have been carried out in both the arithmetic unit and the memory interface.

[2]except the old PC value residing in the PC register after a new PC value has been transferred.

6.3.2. Register Transfer Level Detail

Based upon the intuitive overview, the basic units in Figure 6.5 are expanded to the register transfer level detail respectively.

Part I: Decision Unit

The register transfer level implementation of the decision unit is illustrated in Figure 6.6.

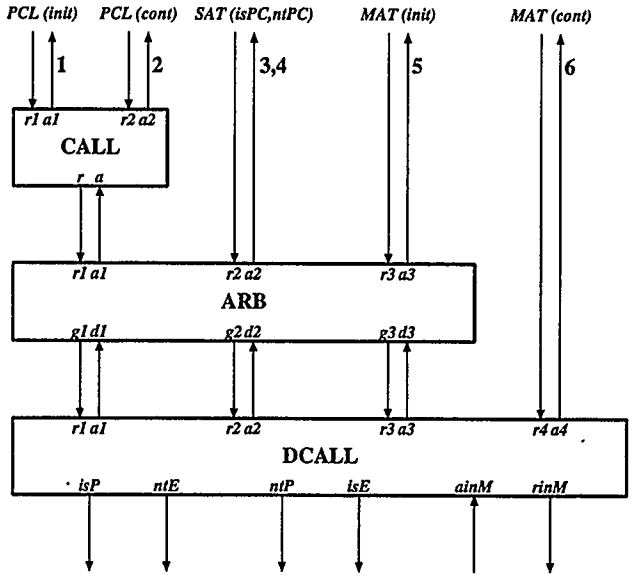


FIGURE 6.6. Control flow of accesses to the address interface: decision unit

Access requests to the address interface can be either mutually exclusive or arbitrary. The decision unit uses call modules for mutually exclusive requests and arbiter modules for arbitrary requests to guarantee the exclusive access to the address interface at any time.

- The initialization phase and the continuation phase of the PCL are mutually exclusive accesses between each other (the processor starts operating with the initialization phase, and moves on to the continuation phase). A two-way call module (CALL) is used for this purpose.
- The SAT and the initialization phase of the MAT are mutually exclusive between each other, but arbitrate with the PCL. A three-way arbiter (ARB) is used for this

purpose. To make the presentation simpler, the ARB used here adopts the RGDA protocol rather than the RGD protocol. In CCS we have

$$
\begin{aligned}
U1 &\stackrel{def}{=} r1.\ 'g.\ 'g1.d1.'a1.\ 'p.\ U1 \\
U2 &\stackrel{def}{=} r2.\ 'g.\ 'g2.d2.'a2.\ 'p.\ U2 \\
U3 &\stackrel{def}{=} r3.\ 'g.\ 'g3.d3.'a3.\ 'p.\ U3 \\
Sem &\stackrel{def}{=} g.p.Sem \\
ARB &\stackrel{def}{=} (\ U1 \mid U2 \mid U3 \mid Sem\ ) \setminus \{\ g,\ p\ \}
\end{aligned}
$$

Once the access to the address interface is granted to the initialization phase of the MAT, the ARB cannot be released until the continuation phase of the MAT has been completed. Thus, there is no need for the continuation phase of the MAT to compete for the address interface via the ARB.

- After the arbitration via the ARB, a decision call unit (DCALL) is used to accept the mutually exclusive access requests from the PCL, the SAT, the initialization phase of the MAT, and the continuation phase of the MAT. The DCALL is a finite state machine which generates control signals for accessing the arithmetic unit and the memory address unit according to the corresponding access needs. Its operation sequences in CCS are

$$
\begin{aligned}
DCALL &\stackrel{def}{=} r1.'rinM.ainM.'isP.'a1.DCALL \\
&\quad + r2.'rinM.ainM.('isP.'a2.DCALL' + 'ntP.'a2.DCALL) \\
&\quad + r3.'rinM.ainM.('isE.'a3.DCALL + 'ntE.LOOP) \\
LOOP &\stackrel{def}{=} r4.'rinM.ainM.('isE.'a4.'a3.DCALL + 'ntE.'a4.LOOP) \\
DCALL' &\stackrel{def}{=} r1.'a1.DCALL \\
&\quad + r2.'rinM.ainM.'ntP.'a2.DCALL' \\
&\quad + r3.'rinM.ainM.('isE.'a3.DCALL' + 'ntE.LOOP') \\
LOOP' &\stackrel{def}{=} r4.'rinM.ainM.('isE.'a4.'a3.DCALL' + 'ntE.'a4.LOOP')
\end{aligned}
$$

The interpretations are:

  - Access requests from the PCL ($r1$), the SAT ($r2$), and the initialization phase of the MAT ($r3$) are accepted by the DCALL; access request from the continuation phase of the MAT ($r4$) is accepted by the LOOP. If a PC value

is transferred (*isP*) during the SAT, the DCALL unit enters a new state expressed by the DCALL' and the LOOP'.

– The DCALL' may accept access request from the PCL (*r1*), but will throw away the PC value and return to the state expressed by the DCALL and the LOOP. It may also transfer a regular address (*ntP*) during the SAT, but not a PC value (*isP*). This new state has no effect on the MAT which is carried out via *r3* in DCALL' and *r4* in LOOP'.

Part II: Arithmetic Unit

The register transfer level implementation of the arithmetic unit is illustrated in Figure 6.7.

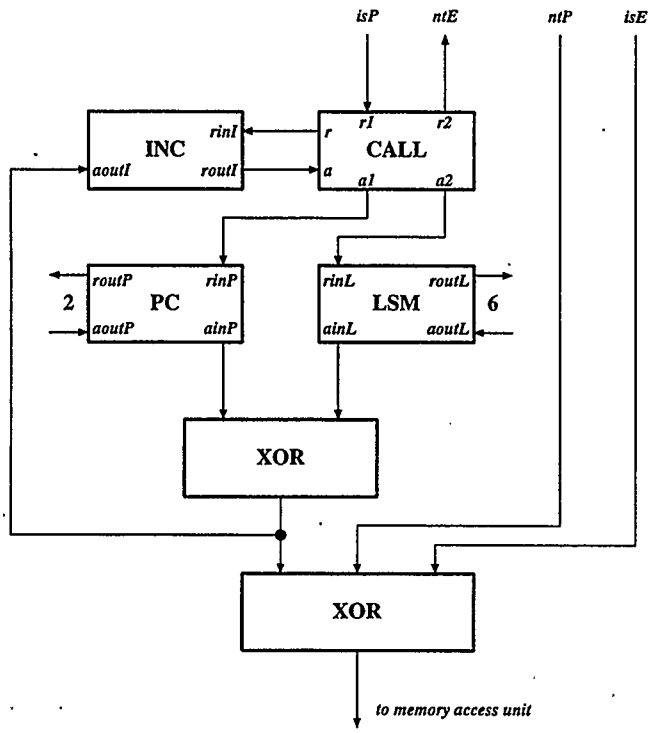


FIGURE 6.7. Control flow of accesses to the address interface: arithmetic unit

The arithmetic unit accepts mutually exclusive access requests from the decision unit. They are: *isP, ntP, isE* and *ntE.*

- if a PC value is transferred (*isP*), it is dispatched to the PC register after being

incremented by the INC;

- if a regular address is transferred (*ntP*), it bypasses the INC and the PC register;
- if a LSM address (not the last one) is transferred (*ntE*), it is dispatched to the LSM register after being incremented by the INC;
- if the last LSM address is transferred (*ntE*), it bypasses the INC and the LSM register.

The mutually exclusive accesses to the INC from *isP* and *ntE* are guaranteed by a two-way call module (CALL). The corresponding acknowledgments (*ainP, ainL*) to the INC (*aoutI*) are collected by an XOR module. These acknowledgments, together with *ntP* and *isE*, are collected by a three-input XOR module for acknowledging the MAR in the memory access unit.

Part III: Memory Access Unit

The register transfer level implementation of the memory access unit illustrated in Figure 6.8 remains the same as that illustrated in Figure 6.5.

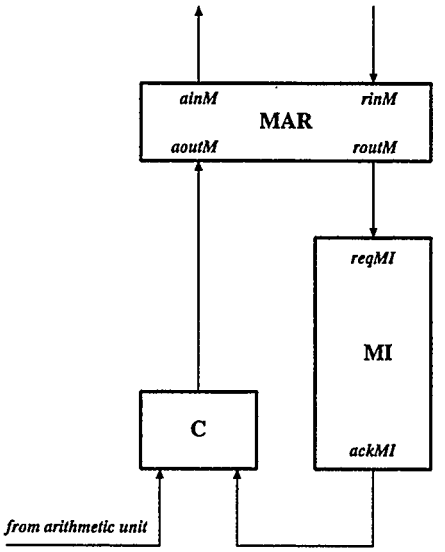


FIGURE 6.8. Control flow of accesses to the address interface: memory access unit

Input to the memory interface is first latched in the MAR upon receiving *rinM* from the decision unit. It is then dispatched to the memory interface via *routM*. Acknowledgments

to the MAR (*aoutM*): *ackMI* from the memory interface and the output of the three-input XOR module in the arithmetic unit, are collected by a C-element:

The complete picture

The register transfer level implementation of the complete control flow of accesses to the address interface is illustrated in Figure 6.9.

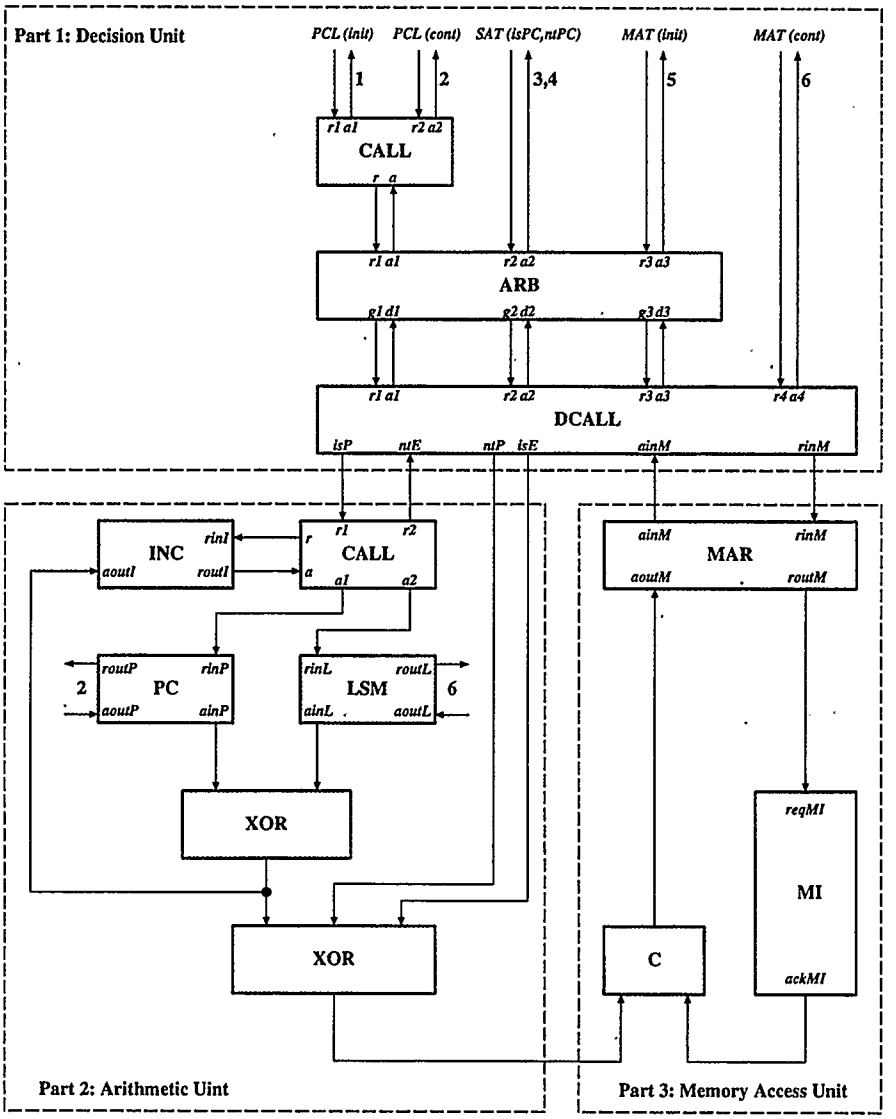


FIGURE 6.9. Complete control flow of accesses to the address interface

To access the address interface, the access requests for the initialization phase of the PCL, the SAT, and the initialization phase of the MAT come from the environment; the access requests for the continuation phase of the SAT and the continuation phase of the MAT come from the PC register (*routP, aoutP*) and the LSM register (*routL, aoutL*) respectively.

6.4. Specification and Testing

6.4.1. Operating Environment

Although the PC incrementing loop, the single address transfer, and the multiple address transfer may require accesses to the address interface arbitrarily, they have to obey the following constraints from the operating environment:

(1) The first access request to the address interface has to be the initialization phase of the PCL (*reqENV_PCL*);

(2) Once the address interface has been initialized (*ackENV_PCL*), both the single address transfer (*reqENV_SAT*) and the multiple address transfer (reqENV_MAT) may arbitrate with the continuation phase of the PC incrementing loop. However, (*reqENV_SAT*) and (*reqENV_SAT*) are mutually exclusive. The winner has the sole access to the address interface.

In CCS we have

```
bi INIT
   reqENV_PCL.'reqPCL.ackPCL.'ackENV_PCL.ENV


bi ENV
     reqENV_SAT.'reqSAT.ackSAT.'ackENV_SAT.ENV
   + reqENV_MAT.'reqMAT.ackMAT.'ackENV_MAT.ENV
```

6.4.2. First Round Testing

The specifications are developed by combining all the basic modules in the control flow illustrated in Figure 6.9 directly, with the constraints from the operating environment. They are then submitted to the CWB for testing.

```
bi PART1
   ( CALL  [ reqPCL/r1,  ackPCL/a1,  routP/r2,  aoutP/a2, r1/r, a1/a ]
   | ARB'  [ reqSAT/r2,  ackSAT/a2, reqMAT/r3, ackMAT/a3            ]
   | DCALL [     g1/r1,      d1/a1,     g2/r2,     d2/a2,
                 g3/r3,      d3/a3,  routL/r4,  aoutL/a4            ]
   ) \ { a1, d1, d2, d3, g1, g2, g3, r1 }

sort PART1
**{ainM,reqMAT,reqPCL,reqSAT,routL,routP,'ackMAT,'ackPCL,'ackSAT,'aoutL,
   'aoutP,'isE,'isP,'ntE,'ntP,'rinM}

min PART1
PART1'
**PART1' has 344 states.

bi PART2
   ( CALL   [ isP/r1, rinP/a1, ntE/r2, rinL/a2, rinI/r, routI/a ]
   | INC
   | PC
   | LSM
   | XOR    [   ainP/a,  ainL/b,  ainPL/z         ]
   | FFORK  [  ainPL/a, aoutI/b, ainPL'/c         ]
   | XOR3   [ ainPL'/a,   ntP/b,    isE/c, zXOR/z ]
   ) \ { ainL, ainP, ainPL, ainPL', aoutI, rinI, rinL, rinP, routI }

sort PART2
**{aoutL,aoutP,isE,isP,lsm',ntE,ntP,pc',sinc','routL,'routP,'zXOR}

min PART2
PART2'
**PART2' has 378 states.

bi PART3
   ( MAR
   | MI    [ routM/reqMI                ]
   | C     [ ackMI/b,     zXOR/a, aoutM/z ]
   ) \ { ackMI, aoutM, routM }
```

```
sort PART3
**{mar',mem',rinM,zXOR,'ainM}

min PART3
PART3'
**PART3' has 15 states.

bi INIT
   reqENV_PCL.'reqPCL.ackPCL.'ackENV_PCL.ENV

bi ENV
     reqENV_SAT.'reqSAT.ackSAT.'ackENV_SAT.ENV
   + reqENV_MAT.'reqMAT.ackMAT.'ackENV_MAT.ENV

bi AI
   ( PART1' | PART2' | PART3' | INIT )
   \ { ackMAT, ackPCL, ackSAT, ainM, aoutL, aoutP, isE, isP, ntE, ntP,
       reqMAT, reqPCL, reqSAT, rinM, routL, routP, zXOR }

sort AI
**{lsm',mar',mem',pc',reqENV_MAT,reqENV_PCL,reqENV_SAT,sinc','ackENV_MAT,
   'ackENV_PCL,'ackENV_SAT}

min AI
AI'
**AI' has 90 states.

fd AI'
--- reqENV_PCL mar' mem' sinc' pc' 'ackENV_PCL reqENV_SAT t<r1> mar' t<routP>
    mem' sinc' 'ackENV_SAT reqENV_MAT t<routP> ---> AI'minState335
```

6.4.3. Resolution of the Deadlock

The CWB testing shows that the control flow illustrated in Figure 6.9 has potential deadlock. Following the trace sequences generated by the CWB, we came to know the reason that causes the deadlock.

(1) After the continuation phase of a PCL is interrupted by a SAT during which a PC value is transferred (*isP*), the old PC value generated by the PCL remains in the PC register while the new PC transferred by the SAT enters the MAR.

(2) The new PC value in the PC register cannot circulate any further until the PC register is free: the old PC value wins the ARB and is then discarded immediately.

(3) At this point, if another SAT wins the ARB instead of the old PC value in the PC register, deadlock results:

- the address to be transferred by the SAT cannot enter the MAR because the new PC value is still occupying it and will not be free until the PC register is empty;
- the PC register will not be free until the old PC value wins the ARB;
- the ARB will not be free until the address to be transferred enters the MAR.

A simple solution used in the AMULET1 implementation is to employ a PC register with the depth of two (*PC2*).

```
bi PC2
   ( PC [ routP1/routP, aoutP1/aoutP ]
   | PC [ routP1/rinP,  aoutP1/ainP  ]
   ) \ { routP1, aoutP1 }

sort PC2
**{aoutP,pc',rinP','ainP,'routP}

min PC2
PC2'
**PC2' has 18 states.
```

6.4.4. Second Round Testing

The control flow specifications with two PC registers are tested to be free from deadlock on the CWB.

```
bi PART2
   ( CALL   [ isP/r1, rinP/a1, ntE/r2, rinL/a2, rinI/r, routI/a ]
   | INC
   | PC2'
   | LSM
   | XOR    [   ainP/a,  ainL/b,  ainPL/z         ]
   | FFORK  [  ainPL/a, aoutI/b, ainPL'/c         ]
   | XOR3   [ ainPL'/a,   ntP/b,    isE/c, zXOR/z ]
   ) \ { ainL, ainP, ainPL, ainPL', aoutI, rinI, rinL, rinP, routI }

sort PART2
**{aoutL,aoutP,isE,isP,lsm',ntE,ntP,pc',sinc','routL,'routP,'zXOR}

min PART2
PART2'
**PART2' has 756 states.

sort AI
**{lsm',mar',mem',pc',reqENV_MAT,reqENV_PCL,reqENV_SAT,sinc',
   'ackENV_MAT,'ackENV_PCL,'ackENV_SAT}

min AI
AI'
**AI' has 162 states.

fd AI'
**No such agents.
```

The control flow of the accesses to the address interface with two PC registers are also tested for the following desired safety and liveness properties in addition to freedom from deadlock.

(1) Livelock free

```
Command: cp AI
Proposition: ~ POSS BOX <t>T
**true
```

(2) After the initialization phase of the PCL (*ackENV_PCL*), it is always possible for the SAT (*reqENV_SAT*) or the MAT (*reqENV_MAT*) to interrupt the continuation phase of the PCL.

```
Command: cp AI'
Proposition: [ackENV_PCL] BOX ( POSS <reqENV_SAT>T)
**true

Command: cp AI'
Proposition: [ackENV_PCL] BOX ( POSS <reqENV_MAT>T)
**true
```

(3) The SAT (*reqENV_SAT*) and the MAT (*reqENV_MAT*) are mutually exclusive between each other.

```
Command: cp AI'
Proposition: (BOX ([reqENV_SAT] [reqENV_MAT]F)) & (BOX ([reqENV_MAT] [reqENV_SAT]F))
**true
```

(4) All the inputs to the PC register and the LSM register are incremented by the INC.

```
Command: cp AI'
Proposition: NEC_FOR sinc' pc'
**true

Command: cp AI'
Proposition: NEC_FOR sinc' lsm'
**true
```

6.5. Summary

This chapter presents a register transfer level specification of the address interface which is one of the major floor plan modules of AMULET1 abstracted in chapter 5. The accesses to the address interface are classified into three cases, namely the PC incrementing loop, the single address transfer, and the multiple address transfer. The control flow of these accesses to the address interface is then developed, specified, and tested.

The main contribution of this chapter is the right level of functional abstraction and the suitable detail of circuitry. The control flows at the register transfer level comprise regular structures (e.g. incrementers, FIFOs) whose implementation techniques are well known, and of highly irregular structures (e.g. DCALL) whose implementations are efficiently supported by Stevens' automatic tool [Ste94]. This level of specification not only provides a good abstract view of how a floor-plan module can be implemented, but also serves as a good design guide for lower level implementations.

In addition, the potential deadlock with one PC register that has been found out through simulation by the AMULET1 group at Manchester university is successfully replicated on the CWB. The resolution of deadlock with two PC registers is also proved to be feasible.

CHAPTER 7
Data Interface

The data interface in AMULET1 manages the data flow between the processor and the memory. It comprises a data input phase and a data output phase. The data input phase accepts a datum from the memory and transfers it to the processor. The data output phase dispatches a datum generated by the processor to the memory. These two phases operate in parallel without interference.

This chapter is concerned with the modeling of data interface at register transfer level. Compared to the address interface discussed in the previous chapter, the specification of the data interface is trivial and straightforward. It is included for completeness.

7.1. Data Input

The internal organization of the data input phase is illustrated in Figure 7.1.

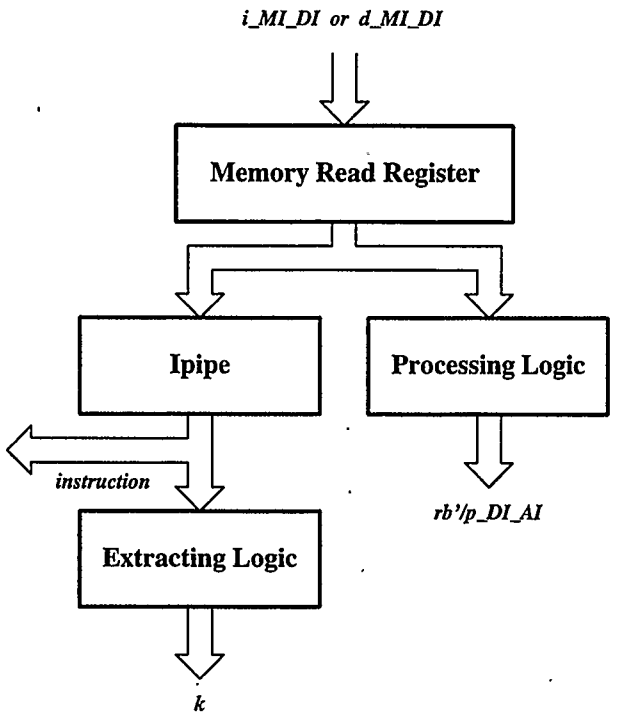


FIGURE 7.1. Internal organization of data input phase in the data interface

Inputs to the data interface can be either instructions or regular data both supplied by the memory interface. The incoming instruction will be stored in the instruction pipe (Ipipe) which is connected to the decode unit. While being dispatched to the decode unit, the instruction is also processed by the extracting logic unit in case an immediate value (k) is needed. The incoming data will be processed by the processing logic unit (PLU) for byte rotation, byte alteration, etc.

When specifying the data input phase, we model the processing logic unit as a regular FIFO, and abstract the Memory Read Register away. The extracting logic unit for immediate value extraction will be modeled in chapter 8 together with the execution pipeline stages. This abstraction is illustrated in Figure 7.2.

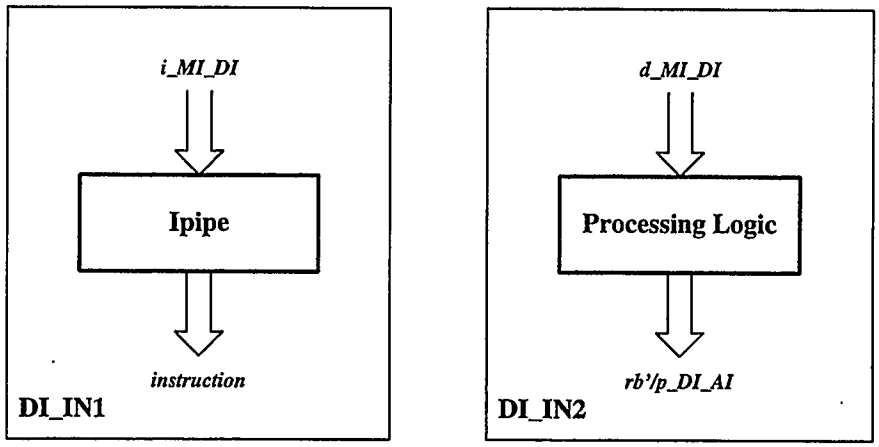


FIGURE 7.2. Abstraction of data input phase in the data interface

In AMULET1, the processing logic unit has two stages, and the Ipipe has five stages[1]. Here they are both modeled as regular FIFOs of depth two. In CCS we have

```
Ipipe = FIFO2 [  incIP/incF,  decIP/decF,  neIP/neF ]
PLU   = FIFO2 [ incPLU/incF, decPLU/decF, nePLU/neF ]
```

The specification of the data input phase (DI_IN) is simply the parallel operation of the instruction input (DI_IN1) associated with the Ipipe, and the data input (DI_IN2) associated with the PLU.

```
bi DI_IN
   ( DI_IN1 | DI_IN2 )
```

- DI_IN1 accepts the next instruction from the memory interface and increments Ipipe.

```
bi DI_IN1
   i_MI_DI_'incIP.DI_IN1
```

- DI_IN2 has two operations which can overlap. One receives data from the memory interface and passes it to PLU. The other passes the data processed by the PLU to either the register bank or the address interface via the W bus.

```
bi DI_IN2
   ( DI_IN2_1 | DI_IN2_2 | PLU ) \ { incPLU, decPLU, nePLU }
```

[1]Due to the potential deadlock caused by instruction overflow, the Ipipe must be three stages longer than the PCpipe [Pav94].

```
bi DI_IN2_1
   d_MI_DI.'incPLU.DI_IN2_1

bi DI_IN2_2
   'nePLU.'decPLU.'gW.(rb'.'pW.DI_IN2_2 + 'p_DI_AI.'pW.DI_IN2_2)
```

7.2. Data Output

The data output phase simply accepts the input from the Dpipe bus associated with the processor and dispatches it to the memory interface. It is implemented as a two stage FIFO in AMULET1. Since the data output phase only has a single user, there is no contention for access to it. In CCS we have

```
bi DI_OUT
   ( DI_OUT_1 | DI_OUT_2 | FIFO2 ) \ { incF, decF, neF }

bi DI_OUT_1
   d_Dpipe_DI.'incF.DI_OUT_1

bi DI_OUT_2
   'neF.'decF.'d_DI_MI.DI_OUT_2
```

7.3. Summary

This short chapter explains the specification of the data interface at register transfer level and is included for completeness. It contains no new techniques or new ideas. The property checking tests are trivial and follow the style presented the rest of the dissertation.

CHAPTER 8
Execution Pipeline

This chapter is concerned with modeling the execution unit at the register transfer level. The execution unit is the computational core of AMULET1. It has a pipelined structure.

The modeling of the execution pipeline is targeted at how each individual instruction class flows through the pipeline in terms of how it seizes a pipeline stage and how it releases the stage. Different instruction classes have different access needs from the data processing hardware associated with the pipeline stages and to the floor plan modules external to the execution unit.

Throughout the modeling of the execution pipeline, we assume that both operands for an instruction have already been read out from the register bank. The register bank itself has been modeled at various levels of abstraction in chapter 4.

8.1. Internal Organization

The execution unit contains the major data processing hardware of AMULET1, including a multiplier, a barrel shifter and an ALU. The internal organization of the execution pipeline is shown in Figure 8.1.

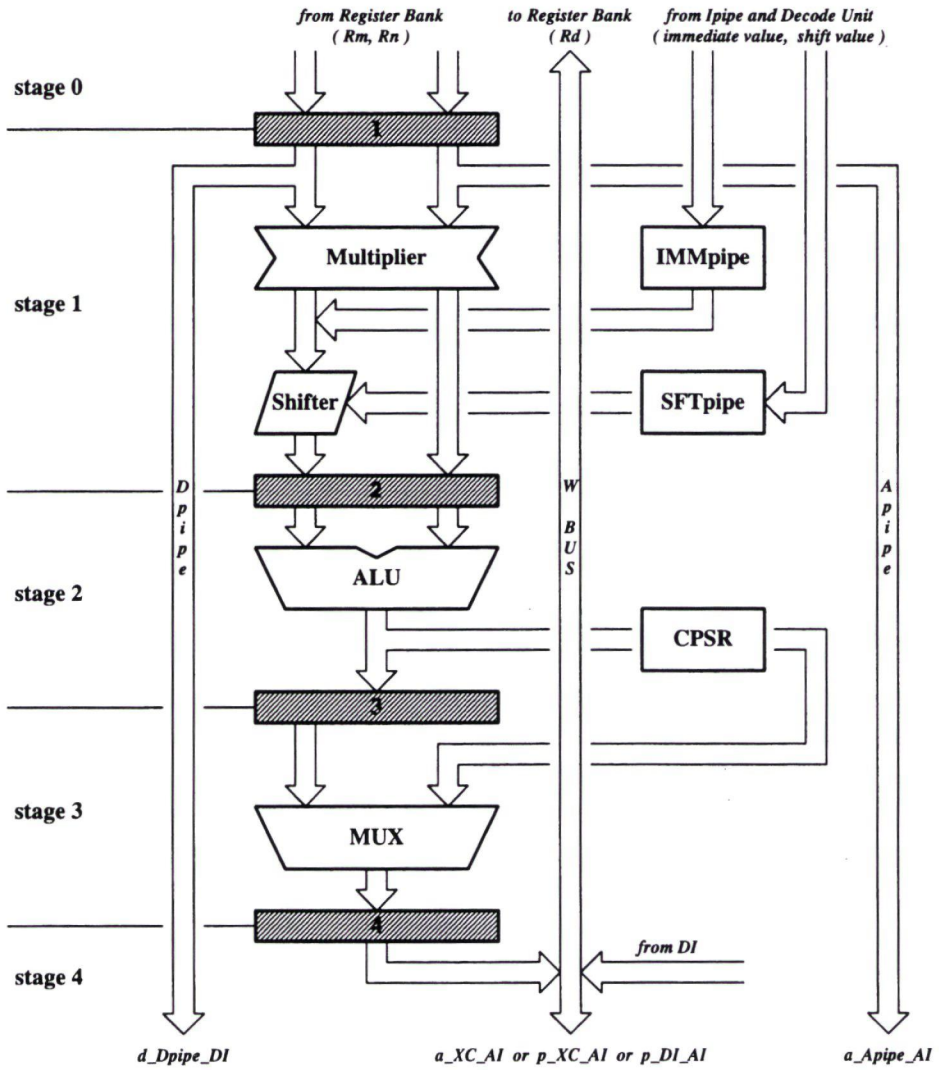


FIGURE 8.1. Internal organization of execution pipeline

Inputs to the execution unit come from register bank, Ipipe (controlled by the decode unit), and the data interface. Input from the Ipipe is decoded and stored in an immediate value pipeline (IMMpipe) and a shift value pipeline (SFTpipe) associated with the execution pipeline. Outputs from the execution unit can be written back to the register bank via the W bus, dispatched to the address interface via either the W bus or the Apipe, or to the data interface via the Dpipe. The shadowed boxes represent the latches between adjacent pipeline stages.

8.1.1. Pipeline Stages

As illustrated in Figure 8.1, the execution unit contains five pipeline stages (stage0 to stage4) if the register bank stage (stage0) is taken into account. Major stages include the multiply and shift stage, ALU processing stage, multiplexor selection stage, and the write back stage. Four latches are used to separate the adjacent pipeline stages.

Stage0

This stage performs instruction decode and operand read from register bank. It also extracts information from Ipipe, and prepares the immediate value and shift value for execution at stage1 in case they are needed. Details of register bank read were discussed in chapter 4.

Stage1

This stage contains a multiplier and a shifter. The multiplier is only used by multiply operations and is bypassed by the rest of the instruction classes. The shifter uses the value supplied by SFTpipe to perform a shift of the operand on the B bus. The operand on the A bus is dispatched to the address interface via Apipe in load operations, store operations, and swap operations. The operand on the B bus is dispatched to the ALU and the data interface via Dpipe in store operations, swap operations, and load/store multiple operations.

Stage2

This stage generates execution results for all the instruction classes through the ALU. The current status of the processor is saved in register CPSR. Condition test (whether an instruction will be executed or discarded) is carried out during this stage.

Stage3

This stage uses the multiplexor to select the normal execution result from ALU, or the current processor status from CPSR if an exception has occurred.

Stage4

This stage is not associated with any particular data processing hardware. It only uses the W bus for dispatching execution results to the register bank or the address interface. Since the data interface also uses the W bus for data dispatching, attention has to be paid to arbitration for access to the W bus.

8.1.2. Additional Pipeline Registers

Additional pipeline registers associated with the execution unit include the immediate value pipeline register IMMpipe, and the shift value pipeline register SFTpipe.

Immediate value pipeline register

Some of the operations use an immediate value k as one of their operands. This immediate value is extracted at the instruction decode stage, but will not be used until the corresponding instruction has entered stage1. The storage of k thus requires a pipeline of depth *2*. The two spaces in IMMpipe store the immediate values for instructions that are currently in stage0, and stage1 respectively.

Shift value pipeline register

The operand on the B bus for some of the operations will be shifted for a certain amount s during execution. This shift value is also extracted at the instruction decode stage, but will not be used until the corresponding instruction has entered stage1. The storage of s thus also requires a pipeline of depth *2*. The two spaces in SFTpipe store the shift values for instructions that are currently in stage0, and stage1 respectively.

8.2. Access Needs and Corresponding Notation

The access needs of an instruction while being executed generally fall into four categories.

8.2.1. Accessing Pipeline Stages

Whether or not an instruction class has needs to access the data processing hardware associated with each pipeline stage, it has to flow through all the pipeline stages sequentially while being executed. Each pipeline stage can be abstracted as a semaphore agent. For stage I, we have:

$$\mathsf{STAGE\_I} \stackrel{def}{=} \mathsf{gsI.psI.STAGE\_I}$$

The control sequence for executing an instruction at stage I is

$$\mathsf{'gsI} \longrightarrow \mathsf{'ps(I\text{-}1)} \longrightarrow \mathsf{exec} \longrightarrow \mathsf{'gs(I+1)} \longrightarrow \mathsf{'psI}$$

and is generated through the following steps:

(1) Entering the current stage via 'gsI, all the information prepared for execution at stageI from the previous stage has thus been latched.

(2) Releasing the previous stage via 'ps(I-1).

(3) Accessing the data processing hardware at the current stage as instructed. The execution (exec) includes accesses to pipeline registers, accesses to local data processing hardware and accesses to floor plan modules external to the execution unit.

(4) Dispatching execution results from the current stage to the next stage via 'gs(I+1) (first step in the next stage).

(5) Releasing the current stage via 'psI (second step in the next stage). StageI is now free for executing the next instruction upon request.

The five pipeline stages in the execution unit operate in parallel:

$$\mathsf{PIPE} \stackrel{def}{=} (\ \mathsf{S0} \mid \mathsf{S1} \mid \mathsf{S2} \mid \mathsf{S3} \mid \mathsf{S4}\ )$$

where

$$\mathsf{Sk} \stackrel{def}{=} \mathsf{gsk.psk.Sk},\ (\ k = 0, 1, 2, 3, 4\ )$$

8.2.2. Accessing Local Data Processing Hardware

Once an instruction class enters a pipeline stage, it will either be processed by the associated data processing units or bypass them as instructed.

Since the instruction currently being executed at stageI is the sole user of the data processing units resident in stageI and there is no contention from other users for accessing them; it is redundant to model how these units are accessed in further detail. We simply use instrI to represent the accesses as instructed, e.g., ldr2 represents a load operation being executed in stage2 (the data processing unit available for accessing is ALU).

Since the accesses to W bus in stage4 can be either destined for register bank or for address interface (when it is a PC value), we use instrRB or p_XC_AI as the notation respectively.

8.2.3. Accessing Pipeline Registers

Pipeline registers external to the execution unit include PCpipe and Ipipe, internal to the execution unit include IMMpipe and SFTpipe. Accesses to them either increment or decrement. Since Ipipe and PCpipe are incremented by floor plan modules external to the execution unit, the execution unit is only responsible for the decrementing of them (decIP and decPC). IMMpipe and SFTpipe are local pipeline registers associated with the execution unit. The execution unit is responsible for both incrementing (incK and incS) and decrementing (decK and decS) them. The specification of IMMpipe and SFTpipe follows:

$$\begin{aligned} \mathsf{IMMpipe} &\stackrel{def}{=} \mathsf{FF0}\ [\ \mathsf{incK/incF},\ \mathsf{decK/decF}\ ] \\ \mathsf{SFTpipe} &\stackrel{def}{=} \mathsf{FF0}\ [\ \mathsf{incS/incF},\ \mathsf{decS/decF}\ ] \end{aligned}$$

where FF0 is the initial state of a pipeline with depth of *2*:

$$\begin{aligned} \mathsf{FF0} &= \mathsf{incF.FF1} \\ \mathsf{FF1} &= \mathsf{incF.FF2} + \mathsf{decF.FF0} \\ \mathsf{FF2} &= \mathsf{decF.FF1} \end{aligned}$$

8.2.4. Accessing External Floor Plan Modules

Accesses to and from floor plan modules external to the execution unit are mainly via the W bus, but Apipe and Dpipe are also used. The notation used for this purpose follows the content_source_destination convention defined in chapter 5, e.g., a_XC_AI represents an address produced by the execution unit being dispatched to the address interface.

8.2.5. Summary of Notation

Table 8.1 summarizes the notation for all the access needs described above:

Access	Notation	
Pipeline Stages	Stage I:	STAGEI = gsI.psI.STAGEI
	Control:	'gsI.'ps(I-1).exec.'gs(I+1).'psI
Data Processing Hardware	stage0:	instr0 (register bank)
	stage1:	instr1 (multiplier and shifter)
	stage2:	instr2 (ALU)
	stage3:	instr3 (multiplexor)
	stage4:	instrRB or p_XC_AI via W bus
Pipeline Registers	PCpipe:	'decPC
	Ipipe:	'decIP
	IMMpipe:	'incK, 'decK
	SFTpipe:	'incS, 'decS
External Floor Plan Modules	content_source_destination	

TABLE 8.1. Notation for access needs in execution unit

8.3. Evaluation of Execution Details

This section evaluates and specifies the execution details of each instruction class at each pipeline stage. Each individual instruction class flows through the pipeline stages in a similar way. The control sequence of executing an individual instruction class (INSTR) is developed based upon the notation summarized in Table 8.1. When condition test is included (in stage2), we have

$$\begin{aligned} \text{CNTR\_INSTR} \stackrel{def}{=} \ & \text{isINSTR.} \\ & \text{'gs0.exec0.'gs1.'ps0.exec1.'gs2.'ps1.exec2.} \\ & \text{(doXC.'gs3.'ps2.exec3.'gs4.'ps3.'gW.exec4.'pW.'ps4.CNTR\_INSTR} \\ & \text{+ noXC.'ps2.CNTR\_INSTR)} \end{aligned}$$

where

> execI can be any access needs to the local data processing hardware associated with stageI, the four pipeline registers, and the floor plan modules external to the execution unit.
>
> doXC is a trace value which represents that the current instruction has passed the condition test and will be carried out.
>
> noXC is a trace value which represents that the current instruction is failed by condition test and will be discarded.

Since the item expressing condition test (noXC.'ps2.CNTR_INSTR) is identical for all the individual instructions, we ignore it in the following descriptions to keep the specifications compact. For the execution of each individual instruction, the control sequence is thus simplified to:

$$\begin{aligned} \text{CNTR\_INSTR} \stackrel{def}{=} \ & \text{isINSTR.} \\ & \text{'gs0.exec0.'gs1.'ps0.exec1.'gs2.'ps1.exec2.} \\ & \text{'gs3.'ps2.exec3.'gs4.'ps3.'gW.exec4.'pW.'ps4.CNTR\_INSTR} \end{aligned}$$

Following this, we evaluate and specify the individual instruction classes. For every instruction class being executed, we first tabulate its access needs at each pipeline stage, and then present the corresponding control sequences in CCS.

Multiply operation (MPY)

The multiply operation takes two register operands. The execution completes in one execution cycle (from stage0 to stage4). Table 8.2 displays the action sequences of multiple operation being executed through the pipeline stages.

MPY Operation	Stage0	Stage1	Stage2	Stage3	Stage4
MPY Rd, Rn, Rm	'decIP	'decPC			mpyRB

TABLE 8.2. Multiply operation on execution

The interpretation of Table 8.2 follows:

- stage0: decrementing Ipipe ('decIP) as soon as MPY enters stage0;
- stage1: decrementing PCpipe ('decPC) as soon as MPY enters stage1;
- stage2: no explicit accesses to external modules;
- stage3: no explicit accesses to external modules;
- stage4: write the multiplication result back to register bank (mpyRB) via W bus.

By adding in trace values mpyI (representing MPY currently being executed in stage I), we have the control sequences in CCS:

```
CNTR_MPY = isMPY.'gs0.mpy0.'decIP.'gs1.'ps0.mpy1.'decPC.'gs2.'ps1.mpy2.
           'gs3.'ps2.mpy3.'gs4.'ps3.'gW.mpyRB.'pW.'ps4.CNTR_MPY
```

Data operation (e.g. ADD)

The operands in data operations have more variety than those in multiply operations. Typically, the two operands can be both register values, or one register value and one immediate value (on the B bus). The operand on the B bus will be shifted if a shift value is provided by SFTpipe. Table 8.3 displays the action sequences of typical add operations being executed through the pipeline stages.

ADD Operation	Stage0	Stage1	Stage2	Stage3	Stage4
ADD Rd, Rn, Rm	'decIP	'decPC			addRB + 'p_XC_AI
ADD Rd, Rn, k	'incK.'decIP	'decPC	'decK		addRB + 'p_XC_AI
ADD Rd, Rn, Rm$\ll$s	'incS.'decIP	'decPC	'decS		addRB + 'p_XC_AI

TABLE 8.3. Add operation on execution

The interpretation of Table 8.3 is similar to that of Table 8.2, except:

- when one of the operands is an immediate value (k), k is prepared and stored in IMMpipe at stage0 ('incK), and released at stage2 ('decK);
- when one of the operands needed to be shifted, the shift value s is prepared and stored in SFTpipe at stage0 ('incS), and released at stage2 ('decS);
- the result of addition can be written back to either the register bank (addRB) or the address interface ('p_XC_AI) via the W bus.

In CCS we have the control sequences as:

```
CNTR_ADD     = isADD1.'gs0.add0.'decIP.'gs1.'ps0.add1.'decPC.
                        'gs2.'ps1.add2.'gs3.'ps2.add3.ADD_STAGE4
             + isADD2.'gs0.'incK.add0.'decIP.'gs1.'ps0.add1.'decPC.
                        'gs2.'ps1.'decK.add2.'gs3.'ps2.add3.ADD_STAGE4
             + isADD3.'gs0.'incS.add0.'decIP.'gs1.'ps0.add1.'decPC.
                        'gs2.'ps1.'decS.add2.'gs3.'ps2.add3.ADD_STAGE4

ADD_STAGE4   = 'gs4.'ps3.'gW.(addRB.'pW.'ps4.CNTR_ADD + 'p_XC_AI.'pW.'ps4.CNTR_ADD)
```

Branch operation (B & BL)

The branch operation is a special type of data operation. One of its two operands is the current PC value, the other is an immediate value supplied by IMMpipe. The addition result of the two operands becomes the new PC value and is sent to the address interface. Table 8.4 displays the action sequences.

B Operation	Stage0	Stage1	Stage2	Stage3	Stage4
B k	'incK.'decIP	'decPC	'decK		'p_XC_AI

TABLE 8.4. Branch operation on execution

In CCS we have the control flow as:

```
CNTR_B  = isB.'gs0.'incK.b0.'decIP.'gs1.'ps0.b1.'decPC.'gs2.'ps1.'decK.b2.
          'gs3.'ps2.b3.'gs4.'ps3.'gW.'p_XC_AI.'pW.'ps4.CNTR_B
```

Branch and link operation requires an extra cycle in the execution unit to save the return address in R14. Since the calculation of return address is based upon the current PC value, this PC value has to be maintained in the PCpipe until it has been latched in the first stage of the second execution cycle. Table 8.5 displays the action sequences for these two execution cycles.

BL Operation	Stage0	Stage1	Stage2	Stage3	Stage4
BL k	'incK		'decK		'p_XC_AI
	'decIP	'decPC			blRB

TABLE 8.5. Branch and link operation on execution

In CCS we have,

```
CNTR_BL1  = isBL1.'gs0.'incK.bl1_0.'gs1.'ps0.bl1_1.'gs2.'ps1.'decK.bl1_1.
            'gs3.'ps2.bl1_3.'gs4.'ps3.'gW.'p_XC_AI.'pW.'ps4.CNTR_BL1
CNTR_BL2  = isBL2.'gs0.bl2_0.'decIP.'gs1.'ps0.bl2_1.'decPC.'gs2.'ps1.bl2_2.
            'gs3.'ps2.bl2_3.'gs4.'ps3.'gW.blRB.'pW.'ps4.CNTR_BL2
```

The execution of these two cycles can be overlapped because of the pipelined nature of the execution unit. The second execution cycle starts as soon as the first cycle has finished its execution in stage0.

Load operation (LDR)

The major varieties of load operation have been summarized in chapter 5 in terms of their indexing styles and write back styles. Each of these varieties is now further classified according the typical operands they take. Similar to the data operation, the two operands can both be register values, or one register value and one immediate value (on B bus). The operand on the B bus will be shifted by a specified amount if a shift value is provided by SFTpipe. Table 8.6 displays the action sequences of the address calculation phase for typical load operations. Executions of all these varieties complete in one execution cycle.

LDR Operation	Stage0	Stage1	Stage2	Stage3	Stage4
LDR Rd, [Rn, Rm]	'decIP	'decPC			'a_XC_AI
LDR Rd, [Rn, k]	'incK.'decIP	'decPC	'decK		'a_XC_AI
LDR Rd, [Rn, Rm≪s]	'incS.'decIP	'decPC	'decS		'a_XC_AI
LDR Rd, [Rn, Rm]!	'decIP	'decPC			'a_XC_AI.ldrRB
LDR Rd, [Rn, k]!	'incK.'decIP	'decPC	'decK		'a_XC_AI.ldrRB
LDR Rd, [Rn, Rm≪s]!	'incS.'decIP	'decPC	'decS		'a_XC_AI.ldrRB
LDR Rd, [Rn], Rm	'decIP	'decPC.'a_Apipe_AI			ldrRB
LDR Rd, [Rn], k	'incK.'decIP	'decPC.'a_Apipe_AI	'decK		ldrRB
LDR Rd, [Rn], Rm≪s	'incS.'decIP	'decPC.'a_Apipe_AI	'decS		ldrRB

TABLE 8.6. Load operation on execution

The conclusions from Table 8.6 follows:

- All pre-indexing load operations dispatch the calculated addresses ('a_XC_AI) at their final stage (stage4) via W bus. If there is a write back to the register bank (ldrRB), it is also carried out in stage4 while the address is being dispatched.
- All post-indexing load operations dispatch the addresses read from register bank ('a_Apipe_AI) directly to address interface via Apipe at stage1. A write back to register bank is always associated with the post-indexing load operation, and carried out at stage4 via W bus.

The corresponding CCS specification is given by combining the pre-indexing load with and without write back together:

```
CNTR_LDR      = isLDR_PRE1.'gs0.ldr0.'decIP.'gs1.'ps0.ldr1.'decPC.
                           'gs2.'ps1.ldr2.'gs3.'ps2.ldr3.LDR_PRE_STAGE4
              + isLDR_PRE2.'gs0.'incK.ldr0.'decIP.'gs1.'ps0.ldr1.'decPC.
                           'gs2.'ps1.'decK.ldr2.'gs3.'ps2.ldr3.LDR_PRE_STAGE4
              + isLDR_PRE3.'gs0.'incS.ldr0.'decIP.'gs1.'ps0.ldr1.'decPC.
                           'gs2.'ps1.'decS.ldr2.'gs3.'ps2.ldr3.LDR_PRE_STAGE4
              + isLDR_PST1.'gs0.ldr0.'decIP.'gs1.'ps0.ldr1.'decPC.'a_Apipe_AI.
                           'gs2.'ps1.ldr2.'gs3.'ps2.ldr3.LDR_PST_STAGE4
              + isLDR_PST2.'gs0.'incK.ldr0.'decIP.'gs1.'ps0.ldr1.'decPC.'a_Apipe_AI.
                           'gs2.'ps1.'decK.ldr2.'gs3.'ps2.ldr3.LDR_PST_STAGE4
              + isLDR_PST3.'gs0.'incS.ldr0.'decIP.'gs1.'ps0.ldr1.'decPC.'a_Apipe_AI.
                           'gs2.'ps1.'decS.ldr2.'gs3.'ps2.ldr3.LDR_PST_STAGE4

LDR_PRE_STAGE4  = 'gs4.'ps3.'gW.( 'a_XC_AI.'pW.'ps4.CNTR_LDR
                                + 'a_XC_AI.ldrRB.'pW.'ps4.CNTR_LDR )

LDR_PST_STAGE4  = 'gs4.'ps3.'gW.ldrRB.'pW.'ps4.CNTR_LDR
```

The data transfer phase of load operations is carried out by the data interface which arbitrates with the execution unit for accessing the W bus.

Store operation (STR)

Similar to load operations, major varieties of store operations summarized in chapter 5 are also further classified according to the types of operands they take. Table 8.7 displays the action sequences of typical store operations.

STR Operation	Stage0	Stage1	Stage2	Stage3	Stage4
STR Rd, [Rn, Rm]					'a_XC_AI
	'decIP	'decPC.'d_Dpipe_DI			
STR Rd, [Rn, k]	'incK.'decIP	'decPC.'d_Dpipe_DI	'decK		'a_XC_AI
STR Rd, [Rn, Rm≪s]	'incS		'decS		'a_XC_AI
	'decIP	'decPC.'d_Dpipe_DI			
STR Rd, [Rn, Rm]!					'a_XC_AI.strRB
	'decIP	'decPC.'d_Dpipe_DI			
STR Rd, [Rn, k]!	'incK.'decIP	'decPC.'d_Dpipe_DI	'decK		'a_XC_AI.strRB
STR Rd, [Rn, Rm≪s]!	'incS		'decS		'a_XC_AI.strRB
	'decIP	'decPC.'d_Dpipe_DI			
STR Rd, [Rn], Rm		'a_Apipe_AI			strRB
	'decIP	'decPC.'d_Dpipe_DI			
STR Rd, [Rn], k	'incK.'decIP	'decPC.'a_Apipe_AI. 'd_Dpipe_DI	'decK		strRB
STR Rd, [Rn], Rm≪s	'incS	'a_Apipe_AI	'decS		strRB
	'decIP	'decPC.'d_Dpipe_DI			

TABLE 8.7. Store operation on execution

Unlike load operations, a store operation takes two execution cycles when both operands are read from the register bank. The second cycle is used to dispatch data to the data

interface ('d_Dpipe_DI) at stage1. In CCS we have

```
CNTR_STR1 = isSTR_PRE1_1.'gs0.str1_0.'gs1.'ps0.str1_1.
                         'gs2.'ps1.str1_2.'gs3.'ps2.str1_3.STR_PRE_STAGE4
          + isSTR_PRE2.  'gs0.'incK.str0.'decIP.'gs1.'ps0.str1.'decPC.'d_Dpipe_DI.
                         'gs2.'ps1.'decK.str2.'gs3.'ps2.str3.STR_PRE_STAGE4
          + isSTR_PRE3_1.'gs0.'incS.str1_0.'gs1.'ps0.str1_1.
                         'gs2.'ps1.'decS.str1_2.'gs3.'ps2.str1_3.STR_PRE_STAGE4
          + isSTR_PST1_1.'gs0.str1_0.'gs1.'ps0.str1_1.'a_Apipe_AI.
                         'gs2.'ps1.str1_2.'gs3.'ps2.str1_3.STR_PST_STAGE4
          + isSTR_PST2.  'gs0.'incK.'decIP.str0.
                         'gs1.'ps0.str1.'decPC.'a_Apipe_AI.'d_Dpipe_DI.
                         'gs2.'ps1.'decK.str2.'gs3.'ps2.str3.STR_PST_STAGE4
          + isSTR_PST3_1.'gs0.'incS.str1_0.'gs1.'ps0.str1_1.'a_Apipe_AI.
                         'gs2.'ps1.'decS.str2.'gs3.'ps2.str3.STR_PST_STAGE4

STR_PRE_STAGE4 = 'gs4.'ps3.'gW.( 'a_XC_AI.'pW.'ps4.CNTR_STR1
                               + 'a_XC_AI.strRB.'pW.'ps4.CNTR_STR1 )

STR_PST_STAGE4 = 'gs4.'ps3.'gW.strRB.'pW.'ps4.CNTR_STR1

CNTR_STR2      = 'gs0.str2_0.'decIP.gs1.'ps0.'decPC.'d_Dpipe_DI.str2_1.
                 'gs2.'ps1.str2_2.'gs3.'ps2.str2_3.'gs4.'ps3.'ps4.CNTR_STR2
```

The second cycle may start as soon as the first cycle completes its access to stage0 and releases it.

Load/store multiple operation (LSM)

Load/store multiple operation is the most complex instruction class. It needs four execution cycles to complete a load multiple and three execution cycles to complete a store multiple. The first two cycles in the execution pipeline are identical for both load multiple and store multiple, but they differ in the third and fourth cycles. Table 8.8 displays the action sequences of these cycles.

LSM Operation	Stage0	Stage1	Stage2	Stage3	Stage4
cycle 1					'a_XC_AI
cycle 2					lsmRB
cycle 3	(isLDM_ntE)*				'a_XC_AI
	isLDM_isE				
	(isSTM_ntE)*	'd_Dpipe_DI			'a_XC_AI
	isSTM_isE.'decIP	'decPC.'d_Dpipe_DI			
cycle 4	'decIP	'decPC			lsmRB

TABLE 8.8. Load/store multiple operation on execution

The interpretation follows:

- The first cycle always calculates the base address and dispatches it to address interface via W bus at stage4 ('a_XC_AI).
- The second cycle always calculates the final address and dispatches it to register bank via W bus at stage4 (lsmRB).
- The third cycle is carried out according to whether the operation is a load multiple (isLDM) or a store multiple (isSTM), and whether this is the end of the transfer (isE/ntE).
  - If a load multiple is being executed, the address is dispatched to the address interface via the W bus at stage4. This cycle loops until the end of the transfer

is detected (isLDM_isE). The execution then moves on to the fourth cycle where the PC value is released.

- If a store multiple is being executed, the data is dispatched to the data interface via the Dpipe at stage1, and address is dispatched to the address interface via the W bus at stage4. This cycle loops until the end of the transfer is detected (isSTM_isE). The execution completes after the last store is finished, and the PC value is released during the last transfer.

• Cycle 4 is only executed for load multiple operation. During this cycle, either a switch of mode or base recovery is carried out internally. For the switch of mode, the current processor status register CPSR is updated with the contents in the relevant stored program status register (SPSR) which specifies the new mode. For the base recovery, the base value is reloaded after a data abort happens. The updated CPSR value or the recovered base value is dispatched to the register bank via the W bus at stage4.

In CCS we have:

```
CNTR_LSM1  = isLSM.
             'gs0.lsm1_0.'gs1.'ps0.lsm1_1.'gs2.'ps1.lsm1_2.
             'gs3.'ps2.lsm1_3.'gs4.'ps3.'gW.'a_XC_AI.'pW.'ps4.CNTR_LSM1

CNTR_LSM2  = 'gs0.lsm2_0.'gs1.'ps0.lsm2_1.'gs2.'ps1.lsm2_2.
             'gs3.'ps2.lsm2_3.'gs4.'ps3.'gW.lsmRB.'pW.'ps4.CNTR_LSM2

CNTR_LSM3  = isLSM.(ntE.'gs0.lsm3_0.'gs1.'ps0.lsm3_1.'gs2.'ps1.lsm3_2.
                         'gs3.'ps2.lsm3_3.'gs4.'ps3.'gW.'a_XC_AI.'pW.'ps4.CNTR_LSM3
                    +isE.'gs0.lsm3_0.'gs1.'ps0.lsm3_1.'gs2.'ps1.lsm3_2.
                         'gs3.'ps2.lsm3_3.'gs4.'ps3.'ps4.CNTR_LSM3)
           + isSTM.(ntE.'gs0.lsm3_0.'gs1.'ps0.'d_Dpipe_DI.lsm3_1.'gs2.'ps1.lsm3_2.
                         'gs3.'ps2.lsm3_3.'gs4.'ps3.'gW.'a_XC_AI.'pW.'ps4.CNTR_LSM3
                    +isE.'gs0.lsm3_0.'gs1.'ps0.lsm3_1.'decPC.'gs2.'ps1.lsm3_2.
                         'gs3.'ps2.lsm3_3.'gs4.'ps3.'ps4.CNTR_LSM3)

CNTR_LSM4  = 'gs0.lsm4_0.'decIP.'gs1.'ps0.lsm4_1.'decPC.'gs2.'ps1.lsm4_2.
             'gs3.'ps2.lsm4_3.'gs4.'ps3.'gW.lsmRB.'pW.'ps4.CNTR_LSM4
```

Swap operation (SWP)

The swap operation completes both a load and a store within one execution cycle. After the address and the data have been dispatched via the Apipe and the Dpipe respectively, the next instruction can be started. Table 8.9 displays the simple action sequences.

SWP Operation	Stage0	Stage1	Stage2	Stage3	Stage4
SWP Rd, Rn, Rm	'decIP	'decPC.'a_Apipe_AI.'d_Dpipe_DI			

TABLE 8.9. Swap operation on execution

During the execution, the address and the store data rendezvous at the memory interface first. A load operation is then carried out followed by a store operation. In CCS we have:

```
CNTR_SWP    = isSWP.'gs0.swp0.'decIP.
                    'gs1.'ps0.swp1.'decPC.'a_Apipe_AI.'d_Dpipe_DI.'ps1.CNTR_SWP
```

The data transfer phase of the load operation is carried out by the data interface.

Software interrupt operation (SWI)

The software interrupt operation is similar to the branch and link operation, but it takes three execution cycles to complete. Table 8.9 displays the action sequences.

SWI Operation	Stage0	Stage1	Stage2	Stage3	Stage4
cycle 1					p_XC_AI
cycle 2					swiRB
cycle 3	'decIP	'decPC			swiRB

TABLE 8.10. Software interrupt operation on execution

Cycle 1 calculates the new PC value and dispatches it to address interface via W bus; cycle 2 saves the old CPSR contents in the relevant stored program status register SPSR by writing it back to SPSR via W bus; cycle 3 saves the return address in R14 via W bus before the exception happens. In CCS we have:

```
CNTR_SWI1 = isSWI.
            'gs0.swi1_0.'gs1.'ps0.swi1_1.'gs2.'ps1.swi1_2.
            'gs3.'ps2.swi1_3.'gs4.'ps3.'gW.'p_XC_AI.'pW.'ps4.CNTR_SWI1
CNTR_SWI2 = 'gs0.swi2_0.'gs1.'ps0.swi2_1.'gs2.'ps1.swi2_2.
            'gs3.'ps2.swi2_3.'gs4.'ps3.'gW.swiRB.'pW.'ps4.CNTR_SWI2
CNTR_SWI3 = 'gs0.swi3_0.'decIP.'gs1.'ps0.swi3_1.'decPC.'gs2.'ps1.swi3_2.
            'gs3.'ps2.swi3_3.'gs4.'ps3.'gW.swiRB.'pW.'ps4.CNTR_SWI3
```

8.4. Specification and Testing

For every instruction being executed, it is the responsibility of the decode unit to decide the nature of the instruction. The execution of each instruction class is thus specified by composing the control sequence described above together with the pipeline stages, the additional pipeline registers, and the corresponding decode information.

Instruction decode

Instruction decode is carried out in stage0. Although most instruction classes only need one execution cycle (from stage0 to stage4) to complete the execution, some instruction classes require more than one cycle in the execution unit to complete. When more than one execution cycle is needed, the following cycle enters the pipeline as soon as its preceding cycle has cleared stage0. Timings between these execution cycles are also coordinated by the decode unit. This is again carried out at stage0. In CCS we have

```
bi DEC
   sDEC.'gs0.( isMPY.DEC_MPY + isADD.DEC_ADD +   isB.DEC_B
             +  isBL.DEC_BL  + isLDR.DEC_LDR + isSTR.DEC_STR
             + isLSM.DEC_LSM + isSWP.DEC_SWP + isSWI.DEC_SWI )


bi DEC_MPY
   mpy0.'decIP.'sMPY.DEC


bi DEC_ADD
     isADD1.add0.'decIP.'sADD1.DEC
   + isADD2.add0.'incK.'decIP.'sADD2.DEC
   + isADD3.add0.'incS.'decIP.'sADD3.DEC
```

```
bi DEC_B
   'incK.'b0.'decIP.'sB.DEC

bi DEC_BL
   'incK.bl1_0.'sBL1.'gs0.bl2_0.'decIP.'sBL2.DEC

bi DEC_LDR
     isLDR_PRE1.ldr0.'decIP.'sLDR_PRE1.DEC
   + isLDR_PRE2.'incK.ldr0.'decIP.'sLDR_PRE2.DEC
   + isLDR_PRE3.'incS.ldr0.'decIP.'sLDR_PRE3.DEC
   + isLDR_PST1.ldr0.'decIP.'sLDR_PST1.DEC
   + isLDR_PST2.'incK.ldr0.'decIP.'sLDR_PST2.DEC
   + isLDR_PST3.'incS.ldr0.'decIP.'sLDR_PST3.DEC

bi DEC_STR
     isSTR_PRE1_1.str1_0.'sSTR_PRE1_1.'gs0.str2_0.'decIP.'sSTR2.DEC
   + isSTR_PRE2.'incK.str0.'decIP.'sSTR_PRE2.DEC
   + isSTR_PRE3_1.'incS.str1_0.'sSTR_PRE3_1.'gs0.str2_0.'decIP.'sSTR2.DEC
   + isSTR_PST1_1.str1_0.'sSTR_PST1_1.'gs0.str2_0.'decIP.'sSTR2.DEC
   + isSTR_PST2.'incK.str0.'decIP.'sSTR_PST2.DEC
   + isSTR_PST3_1.'incS.str1_0.'sSTR_PST3_1.'gs0.str2_0.'decIP.'sSTR2.DEC

bi DEC_LSM
   lsm1_0.'sLSM1.'gs0.lsm2_0.'sLSM2.(isLDM.DEC_LDM + isSTM.DEC_STM)

bi DEC_LDM
     isLDM_isE.'gs0.lsm3_0.'sLDM_isE.'gs0.lsm4_0.'decIP.'sLDM4.DEC
   + isLDM_ntE.'gs0.lsm3_0.'sLDM_ntE.DEC_LDM

bi DEC_STM
     isSTM_isE.'gs0.lsm3_0.'decIP.'sSTM_isE.DEC
   + isSTM_ntE.'gs0.lsm3_0.'sSTM_ntE.DEC_STM

bi DEC_SWP
   swp0.'decIP.'sSWP.DEC
```

```
bi DEC_SWI
   swi1_0.'sSWI1.'gs0.swi2_0.'sSWI2.'gs0.swi3_0.'decIP.'sSWI3.DEC
```

Instruction execution

The actual execution starts from stage1 in the execution pipeline. The control sequences of each instruction class are summarized here.

```
bi EXEC_MPY
   sMPY.'gs1.'ps0.mpy1.'decPC.'gs2.'ps1.mpy2.
        'gs3.'ps2.mpy3.'gs4.'ps3.'gW.mpyRB.'pW.'ps4.EXEC_MPY

bi EXEC_ADD
     sADD1.'gs1.'ps0.add1.'decPC.'gs2.'ps1.add2.'gs3.'ps2.add3.ADD_STAGE4
   + sADD2.'gs1.'ps0.add1.'decPC.'gs2.'ps1.'decK.add2.'gs3.'ps2.add3.ADD_STAGE4
   + sADD3.'gs1.'ps0.add1.'decPC.'gs2.'ps1.'decS.add2.'gs3.'ps2.add3.ADD_STAGE4

bi ADD_STAGE4
   'gs4.'ps3.'gW.(addRB.'pW.'ps4.EXEC_ADD + 'p_XC_AI.'pW.'ps4.EXEC_ADD)

bi EXEC_B
   sB.'gs1.'ps0.b1.'decPC.'gs2.'ps1.'decK.b2.
      'gs3.'ps2.b3.'gs4.'ps3.'gW.'p_XC_AI.'pW.'ps4.EXEC_B

bi EXEC_BL1
   sBL1.'gs1.'ps0.bl1_1.'gs2.'ps1.'decK.bl1_1.
        'gs3.'ps2.bl1_3.'gs4.'ps3.'gW.'p_XC_AI.'pW.'ps4.EXEC_BL1

bi EXEC_BL2
   sBL2.'gs1.'ps0.bl2_1.'decPC.'gs2.'ps1.bl2_2.
        'gs3.'ps2.bl2_3.'gs4.'ps3.'gW.blRB.'pW.'ps4.EXEC_BL2

bi EXEC_LDR
     sLDR_PRE1.'gs1.'ps0.ldr1.'decPC.'gs2.'ps1.ldr2.
               'gs3.'ps2.ldr3.LDR_PRE_STAGE4
   + sLDR_PRE2.'gs1.'ps0.ldr1.'decPC.'gs2.'ps1.'decK.ldr2.
               'gs3.'ps2.ldr3.LDR_PRE_STAGE4
   + sLDR_PRE3.'gs1.'ps0.ldr1.'decPC.'gs2.'ps1.'decS.ldr2.
```

```
                'gs3.'ps2.ldr3.LDR_PRE_STAGE4
   + sLDR_PST1.'gs1.'ps0.ldr1.'decPC.'a_Apipe_AI.'gs2.'ps1.ldr2.
                'gs3.'ps2.ldr3.LDR_PST_STAGE4
   + sLDR_PST2.'gs1.'ps0.ldr1.'decPC.'a_Apipe_AI.'gs2.'ps1.'decK.ldr2.
                'gs3.'ps2.ldr3.LDR_PST_STAGE4
   + sLDR_PST3.'gs1.'ps0.ldr1.'decPC.'a_Apipe_AI.'gs2.'ps1.'decS.ldr2.
                'gs3.'ps2.ldr3.LDR_PST_STAGE4

bi LDR_PRE_STAGE4
   'gs4.'ps3.'gW.('a_XC_AI.'pW.'ps4.EXEC_LDR + 'a_XC_AI.ldrRB.'pW.'ps4.EXEC_LDR)

bi LDR_PST_STAGE4
   'gs4.'ps3.'gW.ldrRB.'pW.'ps4.EXEC_LDR

bi EXEC_STR1
     sSTR_PRE1_1.'gs1.'ps0.str1_1.'gs2.'ps1.str1_2.
                 'gs3.'ps2.str1_3.STR_PRE_STAGE4
   +   sSTR_PRE2.'gs1.'ps0.str1.'decPC.'d_Dpipe_DI.'gs2.'ps1.'decK.str2.
                'gs3.'ps2.str3.STR_PRE_STAGE4
   + sSTR_PRE3_1.'gs1.'ps0.str1_1.'gs2.'ps1.'decS.str1_2.
                'gs3.'ps2.str1_3.STR_PRE_STAGE4
   + sSTR_PST1_1.'gs1.'ps0.str_1.'a_Apipe_AI.'gs2.'ps1.str1_2.
                'gs3.'ps2.str1_3.STR_PST_STAGE4
   +   sSTR_PST2.'gs1.'ps0.str1.'decPC.'a_Apipe_AI.'d_Dpipe_DI.
                'gs2.'ps1.'decK.str2.'gs3.'ps2.str3.STR_PST_STAGE4
   + sSTR_PST3_1.'gs1.'ps0.str1_1.'a_Apipe_AI.'gs2.'ps1.'decS.str2.
                'gs3.'ps2.str3.STR_PST_STAGE4

bi STR_PRE_STAGE4
   'gs4.'ps3.'gW.('a_XC_AI.'pW.'ps4.EXEC_STR1 + 'a_XC_AI.strRB.'pW.'ps4.EXEC_STR1)

bi STR_PST_STAGE4
   'gs4.'ps3.'gW.strRB.'pW.'ps4.EXEC_STR1

bi EXEC_STR2
   sSTR2.'gs1.'ps0.'decPC.'d_Dpipe_DI.str2_1.'gs2.'ps1.str2_2.
         'gs3.'ps2.str2_3.'gs4.'ps3.'ps4.EXEC_STR2
```

```
bi EXEC_LSM1
   sLSM1.'gs1.'ps0.lsm1_1.'gs2.'ps1.lsm1_2.
         'gs3.'ps2.lsm1_3.'gs4.'ps3.'gW.'a_XC_AI.'pW.'ps4.EXEC_LSM1

bi EXEC_LSM2
   sLSM2.'gs1.'ps0.lsm2_1.'gs2.'ps1.lsm2_2.
         'gs3.'ps2.lsm2_3.'gs4.'ps3.'gW.lsmRB.'pW.'ps4.EXEC_LSM2

bi EXEC_LSM3
     sLDM_ntE.'gs1.'ps0.lsm3_1.'gs2.'ps1.lsm3_2.
              'gs3.'ps2.lsm3_3.'gs4.'ps3.'gW.'a_XC_AI.'pW.'ps4.EXEC_LSM3
   + sLDM_isE.'gs1.'ps0.lsm3_1.'gs2.'ps1.lsm3_2.
              'gs3.'ps2.lsm3_3.'gs4.'ps3.'ps4.EXEC_LSM3
   + sSTM_ntE.'gs1.'ps0.'d_Dpipe_DI.lsm3_1.'gs2.'ps1.lsm3_2.
              'gs3.'ps2.lsm3_3.'gs4.'ps3.'gW.'a_XC_AI.'pW.'ps4.EXEC_LSM3
   + sSTM_isE.'gs1.'ps0.lsm3_1.'decPC.'gs2.'ps1.lsm3_2.
              'gs3.'ps2.lsm3_3.'gs4.'ps3.'ps4.EXEC_LSM3

bi EXEC_LDM4
   sLDM4.'gs1.'ps0.lsm4_1.'decPC.'gs2.'ps1.lsm4_2.
         'gs3.'ps2.lsm4_3.'gs4.'ps3.'gW.lsmRB.'pW.'ps4.EXEC_LDM4

bi EXEC_SWP
   sSWP.'gs1.'ps0.swp1.'decPC.'a_Apipe_AI.'d_Dpipe_DI.'ps1.EXEC_SWP

bi EXEC_SWI1
   sSWI1.'gs1.'ps0.swi1_1.'gs2.'ps1.swi1_2.
         'gs3.'ps2.swi1_3.'gs4.'ps3.'gW.'p_XC_AI.'pW.'ps4.EXEC_SWI1

bi EXEC_SWI2
   sSWI2.'gs1.'ps0.swi2_1.'gs2.'ps1.swi2_2.
         'gs3.'ps2.swi2_3.'gs4.'ps3.'gW.swiRB.'pW.'ps4.EXEC_SWI2

bi EXEC_SWI3
   sSWI3.'gs1.'ps0.swi3_1.'decPC.'gs2.'ps1.swi3_2.
         'gs3.'ps2.swi3_3.'gs4.'ps3.'gW.swiRB.'pW.'ps4.EXEC_SWI3
```

Testing on the CWB

The testing here is concerned with how each individual instruction class is decoded and then flows through the execution unit. Again, the load operation is used as an example to demonstrate how testing is carried out.

(1) Testing load operation (LDR) for basic properties:

```
bi LDR
   ( DEC | EXEC_LDR | PIPE | IMMpipe | SFTpipe | WBUS )
   \ { decK, decS, gW, gs0, gs1, gs2, gs3, gs4, incK, incS, pW, ps0, ps1, ps2,
       ps3, ps4, sLDR_PRE1, sLDR_PRE2, sLDR_PRE3, sLDR_PST1, sLDR_PST2, sLDR_PST3 }

sort LDR
**{isLDR,isLDR_PRE1,isLDR_PRE2,isLDR_PRE3,isLDR_PST1,isLDR_PST2,isLDR_PST3,
   ldr0,ldr1,ldr2,ldr3,ldrRB,sDEC,'a_Apipe_AI,'a_XC_AI,'decIP,'decPC}

min LDR
LDR'
**LDR' has 106 states.
```

(2) Testing load operation for desired safety and liveness properties:

(a) Deadlock free

```
Command: cp LDR'
Proposition: BOX <->T
**true
```

(b) Livelock free

```
Command: cp LDR
Proposition: ~ POSS BOX <t>T
**true
```

(c) Pre-indexing load may or may not do a write back; post-indexing load always performs write back, e.g.,

```
Command: cp LDR'
Proposition: BOX ([isLDR_PRE1] (POSS <ldrRB>T))
**true
```

```
Command: cp LDR'
Proposition: BOX ([isLDR_PRE1] (EVENT <ldrRB>T))
**false

Command: cp LDR'
Proposition: BOX ([isLDR_PST1] (EVENT <ldrRB>T))
**true
```

(d) While being executed, the instruction flows through the pipeline stages sequentially. None of the stages can be bypassed.

```
Command: cp LDR'
Proposition: NEC_FOR ldr0 ldr1
**true

Command: cp LDR'
Proposition: NEC_FOR ldr1 ldr2
**true

Command: cp LDR'
Proposition: NEC_FOR ldr2 ldr3
**true
```

Testing details for other instruction classes are similar to the testings of the load operation. The testing complexity of each instruction class is tabulated in Table 8.11.

Instruction Class	MPY	ADD	B	BL	LDR	STR	LSM	SWP	SWI
Minimized States	29	35	29	107	106	1226	1392	24	248

TABLE 8.11. Number of minimized states for each instruction class

8.5. Summary

This chapter presents a register transfer level specification of the execution unit which is another major floor plan module of AMULET1, abstracted in chapter 5. It details how each instruction class flows through the execution pipeline stages and what is carried out within each stage.

The main contribution of this chapter is a systematic way of modeling an execution unit succinctly and efficiently. The specification can be easily expanded/shrunk when more/less pipeline stages are needed. It is a useful guide to system designers when organizing the pipeline stages for execution efficiency, and to circuit designers for an actual implementation. The model suggests that pipeline stage3 can be combined together with stage2 since none of the instruction classes has any extraneous activity between these stages.

CHAPTER 9

Summary

9.1. History

The formal methods group at the University of Calgary started research on asynchronous design and its formalization in 1991. Initial efforts focused on learning how to specify concurrent systems and the intuition and techniques for checking their behaviour and properties [SABL93, LABS93, Liu92]. These techniques were applied to the specification and verification of a library of standard cells and then to an "academic strength" prototype processor, the Move Machine [BLS+94a, BLS+94b]. A four-phase Move Machine, based directly upon the CCS specifications, was successfully implemented by Tom Borsodi in 1993. It functioned on the first silicon.

In January 1993, visitors from Manchester University introduced their AMULET1 design and implementation work to Calgary. Compact (65,000–70,000 transistors) but of industrial strength, AMULET1 seemed to be a worthy challenge for further investigating the appropriateness of CCS to practical asynchronous systems. Further, Manchester University colleagues offered to sustain us with expert help and reciprocal visits from architects, designers and implementors. This input was an essential part of any success we may have achieved.

The modeling of AMULET1 was initiated in early 1993. It is a team effort between Dr. Birtwistle and the author, with direct cooperation from the AMULET group in Manchester University. Given the severity of the task, our initial plan was to specify and verify some of the major functional units in AMULET1 which range in complexity from the straightforward (e.g. the data interface, the memory interface, and the decoder) to the very complex (e.g.

the register bank, the execution unit, and the address interface).

9.1.1. First Round Modeling

The first round modeling focused on the register transfer level specification of the major functional units. We started with an early draft of Nigel Paver's PhD dissertation [Pav94] which, amongst other things, gave a clear overall view of the architecture of AMULET1 as well as the organization of major functional units, but very little on the synchronizations which seem to be documented solely by their schematics. Since CCS concentrates on synchronizations and ignores data values, we proceeded by understanding the data flow, building a CCS model that reproduced our understanding, and then explaining the model to Manchester colleagues — and iterating. The pictures and informal notations used throughout this dissertation served this purpose well. They convert trivially into CCS, and can be used as blueprints for actual implementations.

- **Address interface**

  The address interface was the very first unit we specified. Unfortunately, it turned out to be the hardest piece of hardware to understand since it is shared by several users contending for access to the memory address register. This difficult task forced us to devise a notation for communicating with the Manchester team in order to extract the synchronization details from them [LBP94a]. The informal notation and the block diagrams used in chapters 5–8 filled this role admirably.

- **Register bank**

  Initial attempts on modeling the register bank failed because they were too state rich — we included features of little consequence such as tracking both a and b operands through the lock FIFO in parallel. The breakthrough came when we found the abstractions described in chapter 4 which abstracted the lock FIFO down to a counter [LBP94b]. At the end of this stage, our models were still very detailed and very state rich. But we had the understanding that the register bank acted as a delay and never as a constraint, and was ripe for further simplification.

- **Execution pipeline**

  Although it was clear at the beginning that the AMULET1 instructions could be grouped into several typical classes, two models of the execution pipeline were explored [LBP94c] before we settled on the model presented in chapter 8.

  The first "throw-away" model described in detail how each instruction class flows through the pipeline, but only included hardware that was used by this particular class. The effort was an essential first step to understand the execution detail of each instruction class, but could not be composed because the hardware descriptions were different for each class.

  The second model attempted to overcome this difficulty by moving the instruction classes through the same hardware. To reduce the rich number of states in the first model, the pipeline stages were abstracted into "semaphores" that were seized and released as an instruction moved down the pipeline. The model was successfully used when communicating with the Manchester AMULET team for extracting the execution details accurately.

- **Simple modules**

  The memory interface, the data interface, and the decode unit were comparatively simple and were modeled without much difficulty.

By the end of this phase, we had done much of the groundwork and achieved a good level of understanding of how the major parts of AMULET1 worked in isolation. But it became clear that a top-level model of the complete AMULET1 chip was needed in order to understand how the individual functional units interacted and communicated with each other. This was the major challenge in the second round of modeling.

9.1.2. Second Round Modeling

The second round of modeling started aiming at a very abstract specification that would show how each instruction class flows through the hardware. This top level model made little direct use of the detailed work completed in the first round modeling, but was made much easier by the ground work completed.

The first top level model [BL95] completed was very instructive and fulfilled its purpose of giving us a very clear picture of how the floor plan elements interacted and it proved to be an excellent teaching tool for passing on our understanding. However, it was far too state rich to minimize a stand alone version of just the LDR operation on the complete hardware.

More efforts were then made on finding higher abstractions. This work eventually led to the model presented in chapter 5. These include the abstraction of the execution pipeline by noting that each instruction class could be described as a "block" rather than a number of phases at the top level, and the abstraction of conditional checking which covered colour checking naturally. Even with these improvements, the model was still on the boundary of the viable, and the visible trace variables have to be pared to a minimum. However, the importance of this new model is that we now had a complete overview on where every signal came from and could set up the complete tabulations of signals for each floor plan module.

9.1.3. Third Round Modeling

Based upon this top level modeling, the major floor plan modules were then reworked at the register transfer level. During this round of modeling, we came to realize that an abstract view of each floor plan module at the register transfer level was particularly useful in understanding the implementation detail at this level. These abstractions present intuitions behind the implementations. The register transfer level modeling of the register bank, the address interface, the data interface, and the execution pipeline are presented in chapters 4, 6, 7 and 8 respectively.

9.2. Contributions

This dissertation describes a systematic attempt to apply formal specification and verification techniques to a practical asynchronous design. A complex, industrial-strength, asynchronous microprocessor, the Manchester AMULET1 chip, was modeled with success at both the top level and the register transfer level. The top level specification gives a clear overview of how instructions circulate around the major floor plan modules and would be a suitable level for modeling the processor in an embedded system. With the top level

model making clear the roles and "contexts" of the floor plan modules, the register transfer level models give good implementation guidance to circuit designers. State spaces were minimized by simple abstractions of regular structures (e.g. incrementers, FIFOs) whose implementation techniques are well known, and of highly irregular structures (e.g. the decision call) whose implementations are efficiently supported by Stevens' automatic tool [Ste94].

The contributions and conclusions of this dissertation include:

- This is the first time a complete, practical, asynchronous microprocessor has been specified and verified. Although this is a post facto verification, we believe that the systematic incorporation of formal specification and verification techniques into the design cycle could shorten the overall design window and help reduce the probability of fabrication iterations.
- The main value of the top level model is that it presents a good understanding of the whole architecture, a very abstract view of major functional units, and most importantly, how these functional units interact with each other. It has been used to duplicate several of the well-known deadlocks of early versions of AMULET1, but has not uncovered any new design flaws. The succinct and efficient specifications accurately captured the functionality details. They would also serve as guides to system analysts and systems designers investigating AMULET1 embedded systems.
- The value of the abstract view of each floor plan module at the register transfer level lies in that it abstracts away the clumsy implementation details at this level but retains an admirable intuition of those details. It is easily mapped into the register transfer level design.
- The value of the register transfer level modeling for major functional units lies in that it presents implementation details ready to be floor planned by circuit designers. It is also mechanically easy for direct translation into CCS.
- The specifications were tested for freedom from deadlock, freedom from livelock, and specific safety and liveness properties. These are very difficult to prove via traditional simulation techniques. Property checking was demonstrated to be a fast and efficient way, with full state coverage, of detecting specification flaws.

- Throughout this specification and property checking work, CCS was demonstrated to be an appropriate notation for specifying practical asynchronous systems. Even with CCS, a very coarse specification language, the models are extremely state rich and on the limit of what can be handled at the time of writing. Other specification notations (CSP, value passing CCS, SCCS, ...) would generate even more states.
- The CWB was a robust and reliable tool for property checking and correctness verification. However, its minimization algorithm is not particularly efficient and what can be put through depends largely on the user's skill. The reduction theorem was one of the keys to reduce the number of intermediate states during minimization. Another key lies in choosing the right order to compose when minimizing — follow the flow of data rather than a design hierarchy.

9.3. Future Work

The work presented in this dissertation suggests the following future research topics:

9.3.1. Applications

Although the specifications presented in this dissertation are tailored to a specific asynchronous microprocessor, the methodology illustrated can be easily extended when more functionality is added in. Useful applications include:

- design guidance for next generation AMULET chip;
- specification and verification of AMULET1 embedded asynchronous systems.

Further, the specification style is also applicable to general purpose asynchronous designs.

The specification and verification work presented in this dissertation is a post facto approach which was started after the design had been fabricated and well-tested. The systematic incorporation of formal specification and verification techniques into the above mentioned design cycles could shorten the overall design period and help reduce the probability of fabrication iterations.

9.3.2. Notation Formalization

Although CCS process algebra has demonstrated its appropriateness and efficiency in modeling asynchronous designs at both the system level and the register transfer level, it is not a notation specifically tailored to asynchronous systems. In this dissertation, informal notations were used at both levels of specifications before the actual codes in CCS are presented. These intermediate notations are found to be both intuitive and expressive. The formalization of these notations into an extension of CCS targeted at the modeling of asynchronous hardware would be useful.

9.3.3. CCS Based Silicon Compilation

The specifications presented in this dissertation, especially the register transfer level specifications, demonstrate a direct map to the practical implementation. A CCS based compiler that translates specifications in CCS into asynchronous circuits seems to be feasible.

There have been several prototype silicon compilers for asynchronous circuit design using a variety of CSP based notations, including the CHP compiler by Martin and his colleagues [Mar86, Bur87, Mar90], the Tangram compiler by van Berkel and his colleagues [vBS88, NvBRS88], and the OCCAM complier by Brunvand [Bru91]. However, none of these compilers consider the formal verification of a design. By integrating the macro-based property testing mechanism, a CCS based silicon compiler could help put the design of asynchronous systems on a firm formal basis. The structure of the proposed silicon compiler is illustrated in Figure 9.1.

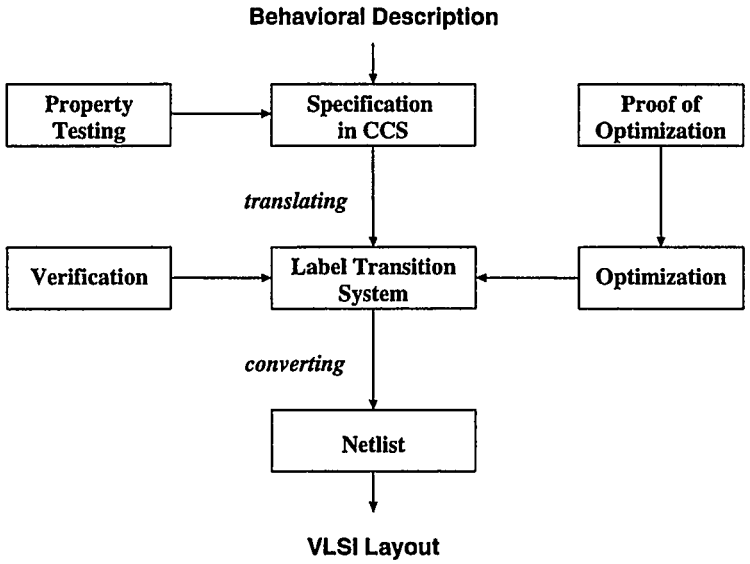


FIGURE 9.1. Silicon compilation based upon CCS

In this compiler, CCS process algebra is used as the programming notation for design specifications, a translator takes the specifications in CCS to an intermediate label transition system, and a converter takes the intermediate notation down to circuit netlist (e.g. FPGA netlist) which is ready for layout. The major supporting tools of this compiler include property testing of the specification, correctness verification of the translation procedure, optimization of the direct translation results, and also the proofs of these optimizations.

Bibliography

[ANB91] J. Aldwinckle, R. Nagarajan, and G. Birtwistle. An Introduction to Modal Logic and its Applications on the Concurrency Workbench. Technical Report, Computer Science Department, University of Calgary, 1991.

[BE90] J. Brzozowski and J. Ebergen. On the Delay-Sensitivity of Gate Networks. Technical Report 90–5, University of Eindhoven, 1990.

[BL95] G. Birtwistle and Y. Liu. Specification of the Manchester AMULET1: Top level. Computer Science Department Technical Report, University of Calgary, January, 1995.

[BLS+94a] G. Birtwistle, Y. Liu, D. Spooner, J.Aldwinckle, K. Stevens, and W. Yu. Case Studies in Asynchronous Design. Part I: AMM — Asynchrounous MOVE Machine. Computer Science Technical Report, University of Calgary, 1994.

[BLS+94b] G. Birtwistle, Y. Liu, D. Spooner, J.Aldwinckle, K. Stevens, and W. Yu. Case Studies in Asynchronous Design. Part II: a 4 stroke AMM. Computer Science Technical Report, University of Calgary, 1994.

[Bre75] J. G. Bredeson. Synthesis of Multiple-input Change Hazard-free Combinational Switching Circuits without Feedback. *International Journal of Electronics (GB)*, 39(6):615–624, December 1975.

[Bru91] E. Brunvand. Translating Concurrent Communicating Programs into Asynchronous Circuits. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1991.

[BS89] E. L. Brunvand and R. F. Sproul. Translating Concurrent Programs into Delay-insensitive Circuits. In *IEEE International Conference on Computer-Aided Design*, pages 262–265, Los Alamitos, CA, 1989. IEEE Comput. Soc. Press.

[Bur87] S. Burns. Automated Compilation of Concurrent Programs into Self-timed Circuits. MSc Thesis, Caltech, 1987.

[CPB90] R. Cleaveland, J. Parrow, and B.Steffen. The Concurrency Workbench. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, LNCS 407, pages 24–37. Springer Verlag, 1990.

[Dam90] M. Dam. Translating CTL into the Modal μ-calculus. Technical Report ECS-LFCS-90-123, Department of Computer Science, University of Edinburgh, Edinburgh, 1990.

[DCS93] A. Davis, B. Coates, and K. Stevens. The Post Office Experience: Designing a Large Asynchronous Chip. In *Proceedings of the 26th Annual Hawaii International Conference on System Science*, pages 409–418, Honolulu, 1993. IEEE.

[Dea92a] M.E. Dean. STRiP: A Self-timed RISC Processor Architecture. PhD thesis, Stanford University, 1992.

[Dea92b] D. Dobberpuhl and et al. A 200-MHz 64-b Dual Issue CMOS Microprocessor. *IEEE Journal of Solid-State Circuits*, 27(11):1555–1565, November 1992.

[Dil89] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* MIT Press, Cambridge, Massachusetts, 1989.

[DN95] A. Davis and S. Nowick. Introduction. In G. Birtwistle and A. Davis, editor, *Proceedings VII Banff Workshop: Asynchronous Digital Circuit Design*, pages 1–49. Springer Verlag, Workshops in Computing Series, 1995.

[DNS92] D. Dill, S. Nowick, and C. Sproull. Specification and Automatic Verification of Self-timed Queues. *Formal Methods in Systems Design*, 1(1):30–60, 1992.

[Ebe87] J. C. Ebergen. *Translating Programs into Delay-Insensitive Circuits.* PhD thesis, Eindhoven University of Technology, 1987.

[Ebe91] J. C. Ebergen. A Formal Approach to Designing Delay-Insensitive Circuits. *Distributed Computing*, 5(3):107–119, 1991.

[Ebe93] J. C. Ebergen. A Verifier for Network Decompositions of Command-based Specifications. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, pages 310–318. IEEE Computer Society Press, 1993.

[EG91] J. C. Ebergen and S. Gingras. An Asynchronous Stack with a Constant Response Time. Tech Report, Department of Computer Science, University of Waterloo, 1991.

[FDG+93] S.B. Furber, P. Day, J.D. Garside, N.C. Paver, and J.V. Woods. A Micropipelined ARM. In T. Yanagawa and P.A. Ivey, editor, *Proceedings of the IFIP TC 10/WG 10.5 International Conference on Very Large Scale Integration (VLSI'93)*. North Holland, 1993.

[FDG+94] S.B. Furber, P. Day, J.D. Garside, N.C. Paver, and J.V. Woods. AMULET1: A Micropipelined ARM. In *Proceedings of the IEEE Computer Conference*, March 1994.

[Fur89] S. Furber. *VLSI RISC Architecture and Organisation.* Marcel Dekker, Amsterdam, 1989.

[Fur95] S. Furber. Computing without Clocks: Micropipelining the ARM Processor. In G. Birtwistle and A. Davis, editor, *Proceedings VII Banff Workshop: Asynchronous Digital Circuit Design*, pages 211–262. Springer Verlag, Workshops in Computing Series, 1995.

[GA92] G. Gopalakrishnan and V. Akella. VLSI Asynchronous Systems: Specification and Synthesis. *Microprocessors and Microsystems*, 16(10):517–526, 1992.

[Gar93] J. D. Garside. A CMOS VLSI Implementation of an Asynchronous ALU. In S.B. Furber and M.D. Edwards, editors, *IFIP Working Conference on Asynchronous Design Methodologies*, North Holland, 1993. M. D. Publisher.

[Gin91] M. Ginns. *Archimedes Assembly Language.* Dabs Press, 1991.

[Hau95] S. Hauck. Asynchronous Design Methodologies: An Overview. *Proceedings of the IEEE*, 83(1), Janauary 1995.

[HJBG81] J. Hennessy, N. Jouppi, F. Baskett, and J. Gill. MIPS: A VLSI Processor Architecture. In *Proceedings of the CMU Conference on VLSI Systems and Computations*. Computer Science Press, 1981.

[HM80] M. Hennessy and R. Milner. On Observing Nondeterminism and Concurrency. In *Lect. Notes in Computer Science 85*. Springer, 1980.

[HM85] M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *J. Assoc. Comput. Mach.*, 32:137–161, 1985.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21:666–677, 1978.

[Hoa81] C. A. R. Hoare. A Model for Communicating Sequential Processes. Technical Report, PRG-22, Programming Research Group, Oxford University Computing Laboratory, Oxford, England, 1981.

[Koz83] D. Kozen. Results on the Propositional μ-calculus. *Theor. Comp. Sci.*, 27:333–354, 1983.

[Kun92] D. S. Kung. Hazard-non-increasing Gate-level Optimization Algorithms. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 631–634. IEEE Computer Society Press, November 1992.

[LABS93] Y. Liu, J. Aldwinckle, G. Birtwistle, and K. Stevens. Testing the Consequences of Specifications in modal μ. In *Proceedings of Canadian Conference on Electrical and Computer Engineering*, Vancouver, 1993.

[LBP94a] Y. Liu, G. Birtwistle, and N. Paver. Specification of the Manchester AMULET1: Address Interface. Computer Science Department Technical Report, University of Calgary, April, 1994.

[LBP94b] Y. Liu, G. Birtwistle, and N. Paver. Specification of the Manchester AMULET1: Register Bank. Computer Science Department Technical Report, University of Calgary, June, 1994.

[LBP94c] Y. Liu, G. Birtwistle, and N. Paver. Specification of the Manchester AMULET1: Execution Pipeline. Computer Science Department Technical Report, University of Calgary, June, 1994.

[LD94] B. Lin and S. Devadas. Synthesis of Hazard-free Multi-level Logic under Multiple-input Changes from Binary Decision Diagrams. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 542–549. IEEE Computer Society Press, November 1994.

[Liu92] Y. Liu. Reasoning about Asynchronous Designs in CCS. MSc Thesis, Department of Electrical and Computer Engineering, University of Calgary, 1992.

[Mar85] A. Martin. Distributed Mutual Exclusion on a Ring of Processes. *Science of Computer Programming*, 5:265–276, 1985.

[Mar86] A. J. Martin. Compiling Communicating Processes into Delay-Insensitive VLSI Circuits. *Distributed Computing*, 1:226–234, 1986.

[Mar90] A. J. Martin. Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, New York, 1990. Addison-Wesley.

[MBL+89] A. J. Martin, S. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus. The Design of an Asynchronous Microprocessor. In C. L. Seitz, editor, *Proc . Decennial CalTech Conference on VLSI.* MIT Press, 1989.

[MCS94] A. Marshall, B. Coates, and P. Siegel. The design of an asynchronous communications chip. *IEEE Design and Test*, June, 1994.

[Mil83] R. Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, London, 1989.

[Mol91] F. G. Moller. The Edinburgh Concurrency Workbench, Version 6.0. Tech Report, Computer Science Department, University of Edinburgh, 1991.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive Systems: specification.* Springer-Verlag, New York, 1992.

[MT89] F. Moller and C. Tofts. A Temporal Calculus of Communicating Systems. Technical Report ECS-LFCS-89-104, Department of Computer Science, University of Edinburgh, Edinburgh, 1989.

[ND92] S. Nowick and D. Dill. Exact Two-level Minimization of Hazard-free Logic with Multiple-input Changes. In *Proceedings of the IEEE/ACM International Con-*

ference on Computer-Aided Design, pages 626–630. IEEE Computer Society Press, November 1992.

[NDDH92] S. Nowick, M. Dean, D. Dill, and M. Horowitz. The Design of a High-Performance Cache Controller: a Case Study in Asynchronous Synthesis. In *Proceedings of the Twenty-Sixth Annual Hawaii Internatioanl Conference on System Sciences*, volume 1, Hawaii, 1992.

[NvBRS88] C. Niessen, C. H. van Berkel, M. Rem, and Ronald W.J.J. Saeijs. VLSI programming and silicon compilation: a novel approach from Philips Research. In *Proc ICCD*, New York, 1988.

[Pav94] N. C. Paver. The Design and Implementation of an Asynchronous Microprocessor. PhD thesis, Computer Science Department, 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. *Proceedingd of ICCD'92*, pages 351–355, October 1992.

[PS81] D. Patterson and C. Sequin. RISC I: A Reduced Instruction Set VLSI Computer. In *Proceedings of the 8th Annual Symposium on Computer Architecture, ACM SIGARCH CAN*, 1981.

[Rad83] G. Radin. The 801 Minicomputer. *IBM Journal of Research and Development*, 27(3), May 1983.

[RdSU83] M. Rem, J.L.A.van de Snepscheut, and J.T. Udding. Trace Theory and the Definition of Hierarchical Components. In R. Bryant, editor, *Proceedings of the Third Caltech Conference on VLSI*, pages 225–239, 1983.

[RKDV92] J. Roy, N. Kumar, R. Dutta, and R. Vemuri. DSS: A Distributed High-Level Synthesis System. *IEEE Design and Test of Computers*, 9(2):18–32, 1992.

[SABL93] K. Stevens, J. Aldwinckle, G. Birtwistle, and Y. Liu. Designing parallel specifications in CCS. In *Proceedings of Canadian Conference on Electrical and Computer Engineering*, Vancouver, 1993.

[Ste94] K. Stevens. Practical Verification and Synthesis of Low Latency Asynchronous Systems. PhD Thesis, Computer Science Department, University of Calgary, 1994.

[Sti91a] C. Stirling. Modal and Temporal Logics. Tech Report ECS-LFCS-91-157, Laboratory for the Foundations of Computer Science, Computer Science, University of Edinburgh, 1991.

[Sti91b] Colin Stirling. An Introduction to Modal and Temporal Logics for CCS. In A. Yonezawa and T. Ito, editors, *Concurrency: Theory, Language, and Architecture*, number 491 in LNCS, pages 2–20. Springer-Verlag, 1991.

[Sti92] C. Stirling. Modal and Temporal Logics for Processes. Tech Report ECS-LFCS-92-221, Laboratory for the Foundations of Computer Science, Computer Science, University of Edinburgh, 1992.

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

[Ung69] S. H. Unger. *Asynchronous Sequential Switching Circuits.* John Wiley & Sons, Inc., New York, 1969.

[vb93] K. van berkel. *Handshake Circuits.* Cambridge University Press, 1993.

[vBS88] C. H. van Berkel and Ronald W.J.J. Saeijs. Compilation of Communicating Processes into Delay-insensitive Circuits. In *IEEE Internal Conference on Computer Design: VLSI in Computers & Processors*, pages 157–162, 1988.

[vSA94] A. van Someren and C. Atack. *The ARM RISC Chip: A Programmer's Guide.* Addison Wesley, 1994.

[WH91] Ted E. Williams and Mark A. Horowitz. A zero-overhead self-timed 160-ns 54-b cmos divider. *IEEE Journal of Solid-State Circuits*, 26(11):1651–1661, November 1991.