# AN ASYNCHRONOUS ARM COMPATIBLE MEMORY MANAGEMENT UNIT DESIGN AND IMPLEMENTATION

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

September 2002

By
Frank Weigel
Department of Computer Science

# Contents

# List of Tables

# List of Figures

# Listings

# Abstract

This thesis proposes two architectures for an asynchronous ARM compatible Memory Management Unit (MMU): a very simple Baseline Architecture which operates mainly sequentially and a pipelined Performance Architecture which uses more concurrency.

The MMU architectures are based around a proposed modified coprocessor interface which integrates the processor and coprocessor more closely than the standard ARM model and is better suited for asynchronous operation. The thesis also covers an extension to the interface which allows reordering of coprocessor instructions and out-of-order completion of coprocessor instructions, something which is not possible with the existing synchronous interface.

The Memory Management Units, as well as the necessary controlling coprocessor, were implemented using Balsa, an asynchronous hardware description language developed at the University of Manchester. The thesis explores the suitability of Balsa for the implementation of complex designs and details problems and solutions to asynchronous specific problems such as 'pipeline colour', as well as solutions to MMU specific problems, such as TLB entry invalidation and coprocessor register locking.

Simulation of the synthesised transistor netlists show that both MMU designs add substantial overhead to each memory access. The faster Performance Architecture is predicted to supply instructions to a processor at a rate of around 10Mhz, indicating the need for a faster Balsa backend and more investigation into asynchronous MMU architecture.

# Declaration

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

# Copyright

# Acknowledgements

My thanks goes to my supervisor, Jim Garside, for his guidance and continuous support over the last two years. Peter Riocreux and Luis Plana both played a big part in the design of the new coprocessor interface, which they ended up using in the asynchronous SPA processor; their effort and help, not just on the coprocessor interface, was much appreciated.

It also was an unforgettable experience to work with Andrew Bardsley, the author of Balsa. Without his help on Balsa issues and quick bug fixes to the Balsa tool suite, the work presented in this thesis would have been impossible. I highly recommend that you only rely on tools that are maintained by people you can bribe with beer!

My parents and Nassima also deserve a big 'thank you' for their never ending support. Without them this thesis would never have gotten finished.

Last, but not least, my appreciation goes to all the fine folk at Transitive Technologies who proofread the final draft: Dave, Miles, Marek, John T., John F. and Adrian.

# Chapter 1

# Introduction

## 1.1 Thesis Guide

This thesis covers a wide range of topics: Design and implementation of two memory management units, a new coprocessor interface and a system coprocessor as well as the required evaluation system and software. All pieces aim at reaching three goals:

- *Development of the first asynchronous Memory Management Unit.* At the time of writing, no asynchronous table based memory management units have been described in the literature. However, with the advent of more and more powerful asynchronous processors, it should only be a matter of time before some interest in fully asynchronous computer systems will arise. This thesis aims to illustrate some of the problems involved and point out possible solutions and their effects.

- *Modernisation and asynchronous adaptation of the ARM coprocessor interface.* The Memory Management Units in this thesis are designed for use in conjunction with an ARM microprocessor. In the ARM architecture, the MMU is controlled via a dedicated coprocessor. The goal is to design a new, more powerful interface, suitable for synchronous and asynchronous operation.

- *Evaluation of Balsa as an asynchronous HDL for large, performance critical designs.* Implementing two memory management units and a coprocessor is no small task and would be unfeasible in the given timeframe, without the use of logic synthesis. Finding out how suitable Balsa, a recent asynchronous hardware description language from the University of Manchester, is for a task of such complexity and performance is the final question this thesis sets out to answer.

Chapter 1 gives the obligatory introduction to the fields covered: Asynchronous logic and memory management. Chapter 2 is dedicated to logic synthesis and especially to Balsa, the asynchronous synthesis language which was used to implement all designs in this thesis. The next three chapters cover the main portions of the work done. Chapter 3 describes a new coprocessor interface and a system coprocessor (necessary to control the memory management unit) which implements the interface. Chapter 4 covers design and implementation of two memory management units, whilst Chapter 5 explains the test setup required to evaluate the memory management sub-system and introduces the specifically developed verification software. Chapter 6 uses the performance data gathered with the setup described in Chapter 5 to analyse the performance of the memory management units, whilst Chapter 7 concludes the results.

A list of acronyms is provided in the preface and a glossary can be found (at the end), before the appendices. These are intended to aid the reader and keep the main text more concise. However, all entries only provide explanations of the terms to the depth required by the material presented here.

## 1.2 Asynchronous Logic

### 1.2.1 Advantages of Asynchronous Logic

Almost all modern computer chips are synchronous; a global clock is distributed to all parts of the chip and is used as a timing reference to keep the system in lockstep. The speed of the clock has become almost synonymous with performance: any PC buyer is lured by higher and higher clock speeds, with the latest processors running at more than 2Ghz. However, not all is well in 'clock land', as synchronous designers encounter an increasing number of problems:

1. *Clock skew.* The synchronous paradigm requires global synchrony. The clock edge, which is the global reference point, needs to arrive at all parts

of the chip simultaneously. However, shrinking feature size, lower supply voltages and higher clock speeds make this increasingly difficult. The clock signal is slowed down by wires and the small differences in distance traveled introduce delays, resulting in the clock edge being observed at different times. With very fast clocks, the maximally allowed clock skew becomes very small in absolute terms, which is difficult to achieve. As a result various *clock buffering techniques* such as H trees, buffer chains, etc. [10] need to be used. This in turn leads to increasing area being allocated to clock distribution and higher power consumption in the clock circuits.

2. *Modular composition* is becoming of increasing importance due to highly integrated chips with a processor core and many peripheral and memory devices. Each peripheral requires one or more clocks at various speeds. This leads to a need for multiple clock distribution networks, with the different clock signals often requiring a fixed phase relationship. As a result, the overall area allocated to clocks and their power consumption increases. Furthermore, additional problems arise for inter-peripheral communication. Connections between the different clock domains become necessary and introduce further overhead.

3. Many modern systems impose limits on EMR *(Electro-Magnetic Radiation)*. EMR can cause interference with devices such as televisions and radios. This is especially of concern in communication systems like mobile phones; making a radio transceiver to operate correctly only centimeters away from a clock running at several hundred megahertz is very difficult. Even several meters away from the clocked system interference can still affect sensitive equipment. Airlines have banned the operation of laptops during take-off and landing as interference from the laptop may endanger correct operation of crucial equipment. EMR may also compromise security as it can allow external observers to deduce data and/or instructions from the frequency distribution. This is of concern to very secure systems like smart-cards.

Due to these problems there has been a renewed interest in *asynchronous systems* which are entirely *self-timed* and do not require any regular clocks. Many early computers were asynchronous, but as asynchronous systems are more difficult to design they were quickly superceded by synchronous designs. However, the asynchronous paradigm claims several key benefits compared to clocked systems:

1. *Data dependent execution time.* Any operation which puts a result into a clocked register needs to compute the result sufficiently far ahead of the next clock edge to meet the required setup constraint. However, in many cases the time needed for an operation can vary depending on the data values being processed; alternatively the choice of operation might be dependent on the data. Synchronous designs always have to cater for the worst case delay to guarantee correct operation and hence need to choose architectures and implementations which minimize the worst case critical path. In an asynchronous system each operation is free to take as long as it requires, therefore the system can be optimized for the *typical case* rather than for the worst case. This influences both the architecture and implementation of the system. Depending on the frequency distribution of worst and average case data, it can lead to performance gains and, in many cases, to smaller implementations.

2. *Simple modular composition* is the immediate result of the absence of clock domains. Blocks become easily "composable" which is an advantage for a building block approach to system design as used in SoC (System on Chip). Timing verification can be done independently for each block, making it easier and faster to generate working systems. These advantages are becoming increasingly interesting to synchronous designers and there is now research into combining asynchronous and synchronous designs. Globally asynchronous, locally synchronous (GALS) designs use asynchronous interconnects to connect synchronous components to benefit from the modular composition properties of asynchronous systems.

3. *Low power consumption.* Clock distribution and buffering consume significant amounts of power, but are not needed in asynchronous designs. Furthermore, in most CMOS implementations, circuits only consume power when they are switching and in asynchronous circuits no transitions occur when a circuit is inactive. Clocked components consume power on every cycle, even when no useful work is performed. Synchronous designers can resort to *clock gating* strategies to save power, however these introduce added difficulties and pitfalls.

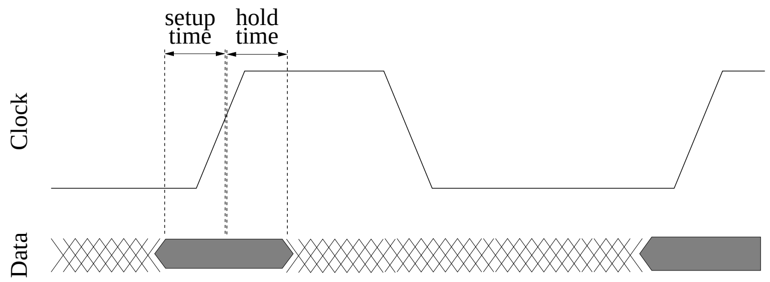


Figure 1.1: Data transfer in a synchronous system

4. *Better electro-magnetic compatibility* than synchronous systems. Because circuit activity is spread more evenly over time than in synchronous systems, peak levels of Electro-Magnetic Radiation are usually lower. Lower peak radiation reduces the interference problem and makes asynchronous systems interesting for use in communication devices (e.g. Amulet3 [20] is intended to be used in a cordless telephone system).

### 1.2.2 Handshakes

To transfer data between blocks, synchronization is required. The receiver needs to know when the incoming data is stable and valid, whilst the sender has to know when the transfer has finished and when it can initiate the next one. In synchronous designs the clock acts as the global reference for transfers and usually either the rising or the falling edge is used as synchronization point. The sender needs to have the data ready a certain time before the active clock edge arrives and will send more data at the next clock edge. The receiver has a guarantee that the incoming data will be ready and stable at the clock edge (refer to figure 1.1).

Asynchronous systems have no global synchronization. Each block works at an individual, often varying, pace. When transferring data explicit synchronization is necessary. The most common form of synchronization is the *handshake*. A handshake uses *request* and *acknowledgement* signals to indicate data validity and transfer completion. A basic handshake is illustrated in figure 1.3. All signal transitions, i.e. rising and falling edges, are equivalent and also referred to as *events*. A transfer is always initiated by a request event. Completion of the transfer is signalled by the recipient via an acknowledgement event. Data needs to be valid a certain time before request is asserted (*setup time*) and needs


Figure 1.2: 2 and 4 phase push handshakes

to remain steady for a certain time after acknowledgement was signalled (*hold time*). The form of handshake described is called *two phase handshake*.

Another handshake protocol is the four phase handshake. It differs from the two phase handshake in that it does not treat all transitions identically. Instead, it is level sensitive. Hence, acknowledgement and request need to *return to zero* after they have been asserted, before a new handshake can be initiated. An example is given in figure 1.2. A handshake is initiated by a rising request signal. The recipient raises the acknowledgement signal when it has seen the rising request signal and is ready to proceed. When the sender observes the high acknowledgement signal, it is free to lower the request signal. The recipient in turn needs to wait for the request to have returned to zero before it may lower the acknowledgement. Different schemes exist for data validity. Some are illustrated in figure 1.2. *Broad validity* requires the data to remain stable until both handshake signals have returned to zero. *Early validity* leaves the data stable from the rise of the request to the rise of the acknowledgement signal. *Late validity* requires the data to remain steady from the falling edge of the request signal to the falling edge of the acknowledgement. Most systems which are implemented in CMOS choose the four phase handshake over the two phase handshake, as it leads to simpler designs due to the level sensitivity and is the protocol used by all implementations covered in this thesis.

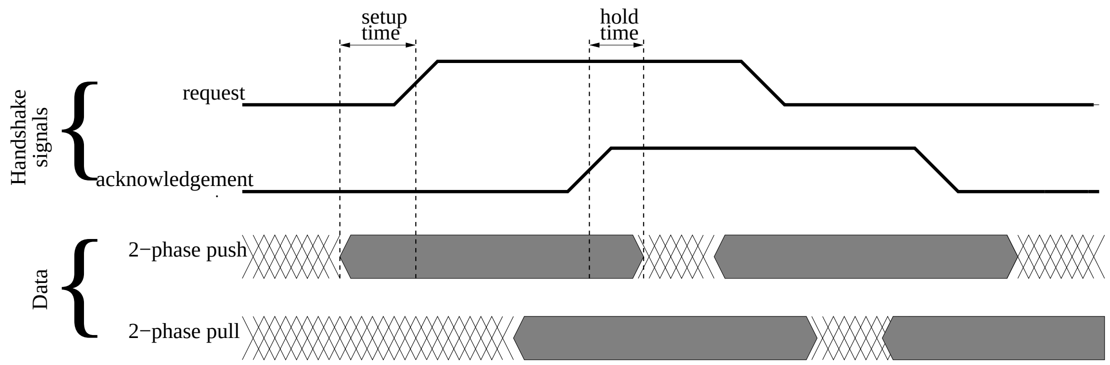


Figure 1.3: Data transfer with handshakes

The combination of data and the associated handshake signals is also named a *channel*. One common implementation of a channel uses the *bundled data* approach. Data is transferred on independent wires alongside the handshake signals. The delay on the data wires needs to be matched to the delay on the handshake wires. It requires the least amount of implementational overhead, however it can never be Delay Insensitive (refer to section 1.2.3) and hence requires careful timing verification. Other implementations of channels exist, which encode data and handshake differently and offer DI data transfer.

The description of the handshake so far assumes that data is transfered from the active initiator to the passive receiver. If this is the case, the channel is called a *push channel*. However, sometimes data needs to be requested by the recipient, i.e. the initiator of the handshake wants to receive data. Accordingly, such channels are named *pull channels*. Handshakes on pull channels are performed in the same way as handshakes on push channels, only the data validity scheme needs to be different, as illustrated in figure 1.3.

### 1.2.3 Delay Models

All logic circuit designs have inherent assumptions about timing. Assumptions about setup and hold times, propagation delays and restrictions on relative signal timing are necessary for synchronous and asynchronous systems to operate correctly. *Timing closure* is relatively easy to achieve in synchronous systems with a single clock. However, achieving timing closure in asynchronous systems or synchronous systems with multiple interacting clock domains is more difficult, as there is no single global timing control which would allow slowing down of the

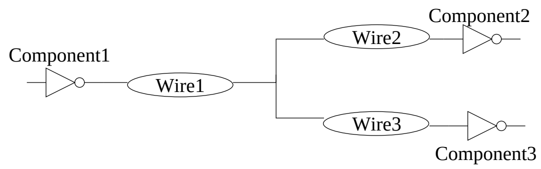


Figure 1.4: Example circuit to differentiate delay models

entire system.

In some asynchronous systems timing closure can be guaranteed by placing constraints on the allowed timing for the external system supplying the input data. However, the timing closure problem also applies to each sub-circuit, where usually no external control is possible [4]. To aid achieving timing closure, different *delay models* were introduced with specific timing constraints and assumptions. These models lead to easier timing verification and give the designer precise constraints to work with. Figure 1.4 depicts a circuit which is used to illustrate the assumptions of some common delay models:

- *Delay Insensitive.* No assumptions or constraints exist. Components and wires may have arbitrary delays without affecting correct operation. This is nearly impossible to achieve *within* components such as gates. Hence it is more commonly applied at the inter-component level.
  $Delay_{Component1}...Delay_{Component3} := arbitrary$
  $Delay_{Wire1}...Delay_{Wire3} := arbitrary$

- *Quasi Delay Insensitive.* Like delay insensitive but an *isochronic fork* [5] is assumed, i.e. it is required that a fork in a wire leads to a signal being observed at both ends of the fork simultaneously (or within an acceptable small margin).
  $Delay_{Component1}...Delay_{Component3} := arbitrary$
  $Delay_{Wire1} := arbitrary$
  $Delay_{Wire2} = Delay_{Wire3} = 0$

- *Speed Independent.* Only components are assumed to have delays, wire delays are assumed to be insignificant.
  $Delay_{Component1}...Delay_{Component3} := arbitrary$
  $Delay_{Wire1} = Delay_{Wire2} = Delay_{Wire3} := 0$

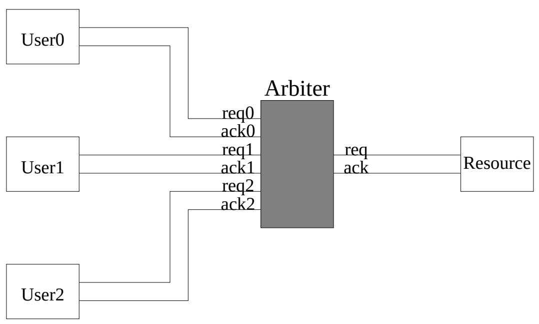


Figure 1.5: Example for use of an arbiter

### 1.2.4 Arbitration

In many systems a resource needs to be shared amongst several users. The implementation of such a resource often requires that access to it is mutually exclusive. Sometimes non-overlapping access is guaranteed because the blocks requiring access are mutually exclusive by design. Alternatively, if blocks are not inherently mutually exclusive, they can perform explicit synchronization to decide which block is allowed to access a contended resource. However, in many cases such explicit synchronization introduces considerable overhead or slows down the system significantly as it forces the blocks to wait for each other. Therefore, when access to a shared resource is potentially overlapping and explicit synchronization is not feasible, *arbitration* becomes necessary.

An *arbiter* receives requests to access the components from several users, but passes only one to the resource at any time. All but the winning request are delayed until the granted handshake has terminated. Only then will the arbiter grant access to another user. Arbiters are *non-deterministic* in their behaviour. There may be an arbitrary delay between receiving the request and passing it to the contended resource and 'Fairness' is not guaranteed. If the resource in figure 1.5 is contended by three users ($user0..user2$) and they request the resource in the order $req0$ then $req1$ then $req2$, it is not guaranteed that they will receive access to the resource in the same order. This can lead to unpredictable, but correct, behaviours in the system as a whole.

Due to these problems the number of arbiters in a circuit is usually kept as low as possible, however there are some cases where they are - practically - unavoidable (refer to section 3.2.3 for an example).

## 1.3 Memory Management

In the early days of computing, memory management was the responsibility of the programmer. If a program did not fit entirely into the available memory, programmers had to save currently unused portion of the program to disk, and replace it with the needed piece of code, so called 'overlays' [21]. This process was error prone, and complicated program development. The idea of *virtual memory*, then called 'one-level storage', was first proposed by Kilburn [22]. It relieved the programmer of the burden of managing memory by integrating the available disk storage (or drum storage at that time) with the physical memory to make them appear as a single piece of memory. The programmer did not need to know how big the physically available memory was, or where in physical memory his program resided, all management was automatic and transparent to the application programmer.

Modern virtual memory still follows the same basic idea. The entire addressable memory space is mapped onto the available physical memory and the disk storage is used to store those parts of the memory space not currently resident in physical memory. The required mapping of *virtual addresses* to *physical addresses* is performed by the *Memory Management Unit* (MMU) (figure 1.6). However, there is usually no need to remap individual addresses to an arbitrary location and the description of such a finely grained translation would occupy far too much space[1]. Instead memory is relocated in blocks. Two major types of such blocks exist: *pages* and *segments* [21]. Pages are of fixed size, whilst segments are of variable size. Most modern systems, including the ARM MMU, only offer support for pages, as they don't lead to fragmentation.

As pages are of fixed size, each memory address can be seen as consisting of two parts. One portion - the offset - describes which word within the page is addressed, whilst the remainder selects the required page (refer to figure 1.7). Only the portion selecting the page, also called *page number*, needs to be translated,

[1]Ignoring management and permission information, each word in the virtual address would require one word to hold the translated address. Hence a 32-bit machine would require 4GB to store translation information alone.

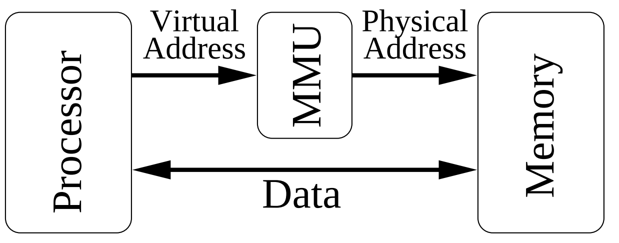


Figure 1.6: Position of the MMU in a system

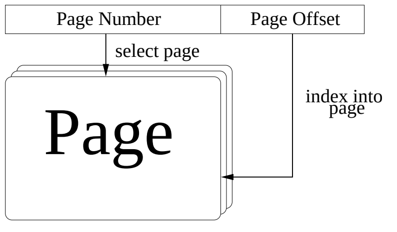


Figure 1.7: Partitioning of memory address into page number and page offset

as the offset - see figure - into the page is independent of its absolute position in memory.

The necessary translation information is stored in a *page table*, within which the page number is used as an index. The resulting table entry, or *descriptor*, contains the translation for the page number along with necessary management information (figure 1.8). However, a single level table gets rather large for small pages and, at the same time, only a relatively small portion of the entries would actually be used. To alleviate this, a *two-level table* is usually employed for the translation of smaller pages. If the first level entry refers to a used portion of memory, it contains a reference to the second level table which contains the actual mapping. This successive lookup process is called *table walking* and adds two memory accesses to each requested memory transaction. As memory latency and bandwidth is already one of the major performance bottle-necks of modern processors, this would lead to an unacceptable performance decrease. To improve performance, the translation information is cached in a so called TLB *(Translation Lookaside Buffer)*. Only if the required translation is not cached will table walking be required. (figure 1.9).

So far I have described the translation process, but ignored the fact that virtual memory was invented to simplify programs which require more memory than is physically available. Therefore, each page table entry also needs to state

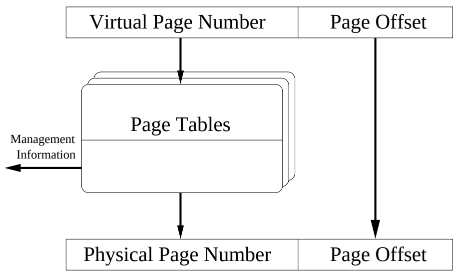


Figure 1.8: Table based translation

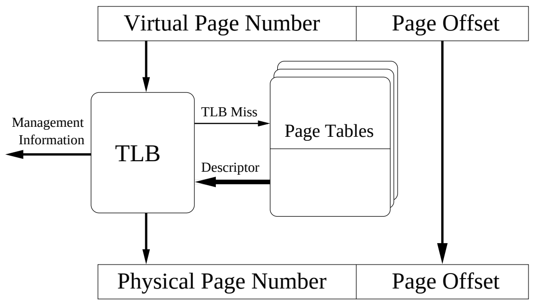


Figure 1.9: TLB based translation

whether the associated page currently resides in physical memory. If a requested page is not in memory, the MMU signals a *memory abort* to the processor. The OS (Operating System) then loads the required page into physical memory, before re-executing the instruction which caused the abort. The MMU has to provide sufficient information to allow the abort handler to determine the cause of the abort and the address of the memory location which caused the abort. To guarantee that aborts can be handled transparently for the user, all instructions which access memory need to be *restartable*, i.e. executing the instruction again has to yield the same result as if the instruction had executed without causing a memory abort.

Besides translation, *permission enforcement* is the other main memory management function performed by the MMU. Access permissions are necessary to protect concurrently running processes from accidentally or maliciously interfering with each other, whilst still allowing the sharing of memory when necessary. Furthermore, certain areas of memory, especially those accessing memory mapped peripherals, are often restricted to only allow access from a privileged mode. Access permissions for each page are normally also stored in the page tables and cached by the TLB.

Finally, for some embedded systems only permission enforcement and not translation is required, for example in security related applications like smartcards. For those applications, a simpler MPU *(Memory Protection Unit)* may be used instead of a MMU to save silicon area and simplify management by the OS.

### 1.3.1 Existing Asynchronous MMUs

As asynchronous processors mature and become more powerful (e.g. Amulet3), an MMU becomes desirable as it is required by many operating systems to implement virtual memory and memory protection (as outlined in section 1.3). However, to date, no fully functional asynchronous MMU has been implemented. The most likely reason is that so far, no asynchronous processor, other than Amulet3, has offered the features and performance necessary to implement systems complex enough to require memory management.

Myers [27] describes a very simple MMU which he uses as a design example to illustrate techniques to derive a circuit from a CSP (Communicating Sequential Processes) description. However, his proposed MMU only concatenates an 8-bit offset to a 16-bit memory address. It uses different offsets for reads and writes but it does not offer any table based translation or memory protection facilities.

A synthesized, asynchronous MPU was developed by Plana at the University of Manchester as part of a third generation smartcard project [19]. The design is compatible with existing ARM MPUs and illustrates that memory management is starting to become of interest to fully asynchronous systems. However, as it is an MPU it does not provide any of the facilities necessary to implement a virtual memory system. Like the MMU described in this thesis, it was developed to be compatible with the SPA processor. SPA is an ARM compatible processor which is synthesized from Balsa and implements a coprocessor interface described in section 3.1.3.

# Chapter 2

# Balsa

This chapter serves mainly as an introduction to the asynchronous HDL used to implement the memory management units (chapter 4) and the system coprocessor (chapter 3).

Section 2.1 gives a general introduction to logic synthesis and the associated advantages and problems whilst section 2.1.1 introduces a few asynchronous synthesis systems. All subsequent sections then cover aspects of Balsa:

- A general overview of Balsa (section 2.2)
- A commented code example which introduces the most common language constructs (section 2.3)
- The Balsa design flow (section 2.4)
- A brief justification of the choice of back-end (section 2.4.1)
- A short description of the directed translation approach and its consequences (section 2.4.2)
- A short overview of LARD, and why it was not used for behavioural simulations (sections 2.4.3, 2.4.4)

Readers already familiar with Balsa should feel free to skip most of this chapter. They are pointed to section 5.1.2 in a later chapter, which describes the integration of Balsa generated circuits into Timemill simulations.

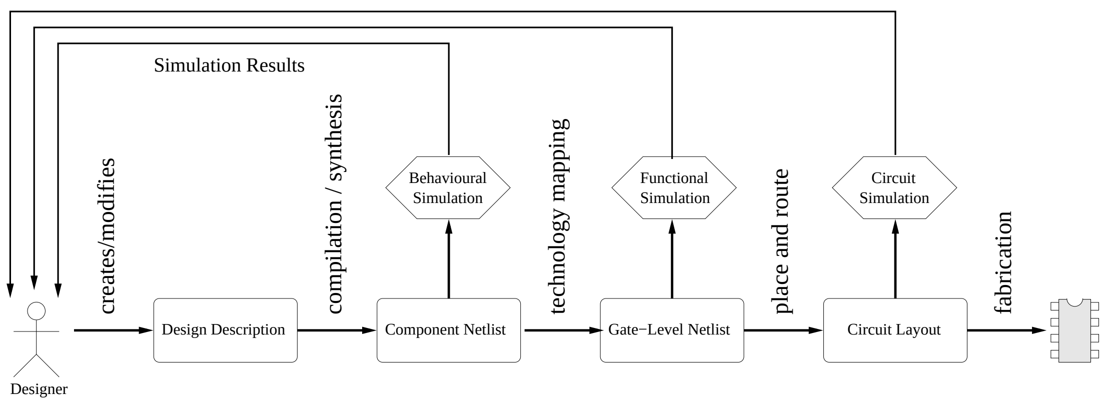


Figure 2.1: Generic design flow for logic synthesis

## 2.1 Logic Synthesis

Logic synthesis has become a very common way of generating logic circuits. With the exception of very ‘performance oriented’ designs, it has become commonplace to use *hardware description languages* (HDLs), like VHDL or Verilog to implement chips. A generic design flow for synthesis requires the following steps (refer to figure 2.1):

1. The design description (which does not need to be in a HDL: it can be a schematic or a graph) is *compiled* into an intermediate high-level component format, specific to the synthesis tool. Usually *behavioural simulation* is possible at this point.

2. The intermediate format can then be *technology mapped*. Technology mapping results in a *gate-level* netlist where the high level components are mapped onto available library cells. *Functional simulation* is performed at this stage.

3. *Place and route* often requires several steps as well as some manual intervention and leads to a finished circuit layout. All process and technology constraints are incorporated and the resulting designs allows accurate *timing and power simulation*.

Simulation results from each level are fed back to the designer and might instigate modifications of the design description to meet specific performance or functional targets.

The use of logic synthesis has several advantages for the designer:

1. *Ease of implementation.* Implementing designs in a HDL is much simpler than designing circuits manually, as the language description is less complex. It requires substantially fewer resources and enables small teams to develop solutions.

2. *Ease of modification.* Modifying a language description is very fast and simple. Changes in encodings or change of algorithms are usually easy to incorporate into a HDL description whilst a manual design might require substantial rework. Ease of modification also results in *fast design turnaround*. It lowers the cost of customer specific modifications and helps fast product development by easily allowing new revisions.

3. *Inherent retargetability.* Once a language description exists, it can be resynthesized to a new process or a new library, without requiring much manual intervention. However porting an existing manually designed circuit to a new library is quite labour intensive and might require design changes.

4. *Better manageability.* As an HDL description is similar to a piece of software, revision control, code reuse and division of labour becomes easier.

However circuit synthesis also has some drawbacks. The expressiveness of the description language needs to be considered - not all circuits can be expressed in all HDLs. Language limitations might impose constraints which are unsuitable for the envisaged design. Furthermore, in many cases, the synthesized design performs worse than a custom designed circuit, although, often this may not matter, if the design performs within the requirements.

### 2.1.1 Asynchronous Logic Synthesis

Manual design of asynchronous circuits is difficult. Logic synthesis simplifies asynchronous logic design and a robust and reliable synthesis system which produces well performing circuits would make asynchronous designs more viable for commercial use. Hence many more or less fully featured asynchronous synthesis systems have been developed or proposed, using a wide range of strategies. A good discussion of the different classes of systems and their merits may be found in [4]. Some of the more commonly used systems, other than Balsa which is covered in the next section, are:

- *Petrify* [8]. Based on Petri nets [26] which are used as design descriptions. Allows description of very fine granularity of concurrency. State space exploration techniques are required for synthesis and make the approach too computationally expensive for larger designs.

- *Minimalist* [15]. Uses burst mode machines [28] which are very similar to classical asynchronous state machines. Well suited to model the arbitrary order of input changes often occurring in asynchronous designs, but the approach requires several state machines to model other forms of concurrency.

- *Null Convention Logic* [23]. A commercial asynchronous synthesis solution. Utilizes delay insensitive encodings in combination with threshold gates [25] with hysterises properties to generate fully delay insensitive circuits. The inherently necessary completion detection often leads to very big circuits.

- *Tangram* [31]. Hardware description language which, like Balsa, is based on *handshake components* [6] developed by Philips Research. It was the starting point for the development of Balsa and hence is quite similar to it.

## 2.2 Introduction to Balsa

Balsa[1] [4, 3] is an asynchronous hardware description language developed by Bardsley at the University of Manchester. It uses a *macromodular* [7] approach originally developed by Clark. Library modules of low to moderate complexity which follow a specific delay model are connected by delay-insensitive (DI) interconnects to form the overall circuit. As each module consists of only a few gates, it can be more easily constructed and optimized to meet the constraints of the delay model than an entire design.

In Balsa the building blocks are called handshake components [6]. Handshake components were originally invented by Philips Research for use in their Tangram [31] synthesis language. Balsa uses all of the original handshake components with some additions [4]. All components communicate via handshake channels. A wide range of possible implementations for handshake components

[1] It is freely available on the Internet at *http://www.cs.man.ac.uk/amulet/projects/Balsa*

exist and Balsa offers a flexible circuit generation system, which allows it to generate circuits in a range of different implementation styles.

## 2.3 Commented Balsa Code Example

For a complete description of Balsa and its use refer to [3] or the Balsa user manual [11]. However, an illustrated example of a simple Balsa program will be presented to aid the reader unfamiliar with the language. The following code snippet is part of the baseline architecture implementation, some parts were omitted for clarity (marked by `[...]`). Numbers between asterisks (`*1*, *2* ... *14*`) refer to the explanations following the code and are not part of the program.

```
import [spaExternalTypes]                                          *1*
 [...]

procedure alignmentCheck(                                          *2*
        -- interface to MMU main                                   *3*
        input virtBundle        : memAccess;                       *4*
         [...]
        output abort            : bit;                             *5*
         [...]
        sync reqTableBase                                          *6*
)
is
         shared noAbort is begin abort <- 0 end                    *7*
begin
         loop                                                      *8*
           select virtBundle, procMode, isFetch then               *9*
             alignmentEnabled -> then                              *10*
               accInfo <- {virtBundle.Address, (virtBundle.RorW as bit)} *11*
               || if alignmentEnabled = true then                  *12*
                         [...]
                  else
                        noAbort();                                 *13*
                        sync reqTableBase                          *14*
                        || tlrOut <- {virtBundle, procMode,
                                      isFetch ,0,0,0,  undef,undef}
                  end
             end
           end
         end
end
```

Listing 2.1: Balsa code example

Explanatory notes:

1. import [<*filename*>]
   Import statements allow the inclusion of other files, usually containing type declarations or library procedures (similar to #include in C)

2. procedure <*name*> (<*ports*>) is <*declarations*> begin ... end
   Procedures are the basis of all Balsa descriptions. They define a component (circuit) which can then optionally be instantiated (called) by other procedures.

3. Comments: -- and (- ... -)
   -- comments out everything until the end of the line, (- ... -) is used for multi-line comments

4. input <*port name*>;
   An input port defines the name and type of an externally visible channel. The type used in this example is a record (defined in the imported types file). A Balsa record is similar to a struct in C. Balsa is a *strongly typed* language.

5. output <*port name*>;
   Output ports follow the same pattern as input ports. bit is a built-in type. It is the fundamental type in Balsa, all other types can be cast into an array of bits.

6. sync <*port name*>;
   Sync ports declare a channel over which no data can be transferred, only handshakes can be performed, hence they are only used for signalling synchronisation. There is no distinction between sender and receiver at declaration time.

7. shared <*name*> is <*declarations*> begin ... end
   Definition of a *shared procedure*. If a 'regular' procedure is 'called', the circuit it defines will be instantiated for each such invocation. A shared procedure will only be instantiated once. Therefore access to the shared procedure needs to be mutually exclusive at all times and further restrictions apply; refer to [11] for details.

8. `loop ... end`

   Most Balsa procedures need to evaluate more than once. The `loop` construct causes infinite re-execution of the enclosed code. Other loop types exist.

9. `select` *<channel list>* `then ...`

   `[|` *<channel list>* `then ... ]* end`

   `Select` waits for all channels in the list to initiate handshakes before proceeding. The handshakes will not terminate before the end clause of the select block is reached. Therefore any handshakes performed within the select block are *enclosed* within the handshakes on the outer channels. This is one of the differences between Tangram and Balsa, Tangram does not allow enclosed handshakes. `Select` can also be used to allow *choice* (the '`|`' denotes the alternative choices) between several *mutually exclusive* transactions on different channels. All ports selected will become *passive*, i.e. they will wait for the sender to *push* data through. If non-exclusiveness cannot be guaranteed `arbitrate` can be used to provide arbitration between *two channels*[2].

10. *<channel>* `-> then ... end`

    *Active selection.* It is similar to select, but it doesn't allow choice or lists of signals. The port will become *active*, data will be *pulled* from the passive sender.

11. *<channel>* `<-` *<variable or expression or selected channel>*

    *Channel assignment.* In the example the right side expression demonstrates record composition, the commas separate the different fields of the record.

12. *<statements>* `||` *<statements>*

    The *parallel composition* operator ('`||`') results in the statements on either side being executed concurrently. The code also illustrates an `if` *<guard>* `then ... else` construct.

13. *<statements>* `;` *<statements>*

    A semicolon is used as the *sequence* operator. Only when the statement on the left has completed (i.e. performed all channel transactions) the

[2]No lists of channels are allowed

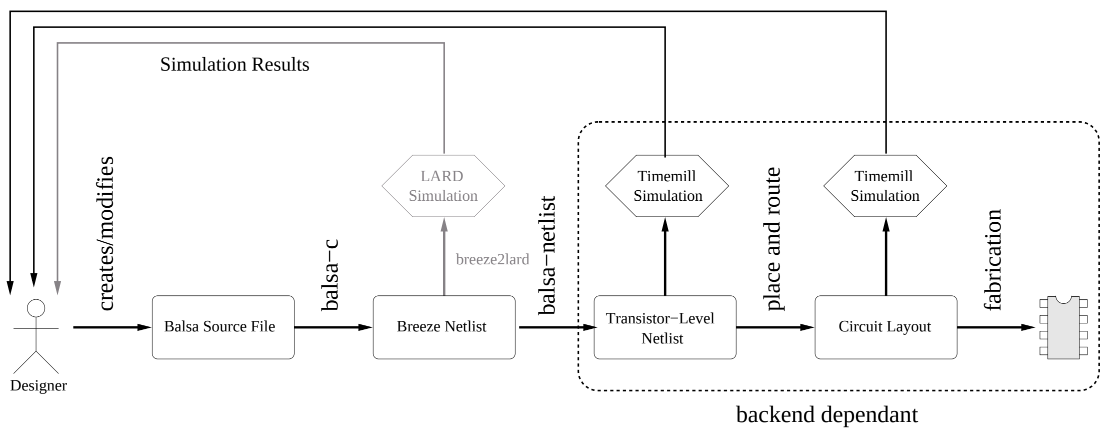


Figure 2.2: Balsa design flow

statement on the right will be executed. Also illustrates invocation of a shared procedure.

14. `sync <channel>`
Handshakes on sync channels do not transmit data and therefore the regular channel assignment operators do not apply. Hence the explicit `sync` keyword is used.

The resulting handshake component schematic of the complete procedure is included in the appendix.

## 2.4 The Balsa Design Flow

The Balsa design flow is depicted in figure 2.2. A Balsa description is compiled by *balsa-c*, the Balsa compiler, into a Breeze handshake component netlist. Breeze is the intermediate netlist format used by the Balsa suite. If behavioural simulation is required at this point, *breeze2lard* can be used to convert the Breeze description into a LARD program which can be simulated (refer to section 2.4.3 for further detail). However this step was skipped in the development of the MMUs described in this thesis (section 2.4.4).

Breeze netlists are also used as the input into *balsa-netlist* which converts the handshake component netlist into a library cell netlist. Balsa-netlist allows choice between a range of different *back-ends*. A back-end is the part of balsa-netlist which generates the gate-level description for specific Breeze components (section 2.4.2). The available back-ends generate circuits which differ in their implementation style for channels and the cell libraries used.

The details of the remaining design flow depends on the back-end chosen. For the bundled data back-end, which was used for the circuits in this thesis, Timemill was chosen to perform simulation of the transistor-level netlist as well as of the routed circuit (refer to section 5.1.2 for further detail about the integration with Timemill). The place and route steps were performed with Compass and resulted in a fully laid out circuit, as shown in appendix A.

### 2.4.1 Choice of Back-end

A range of different back-ends exist for Balsa. They can be divided into two groups:

- *Bundled data back-end.* It uses channels where handshake signals and data are transmitted on separate wires. These channels are not inherently DI, and as a result timing verification is required to guarantee that the delays on the data is matched with the delay on the handshake wires.

- *Inherently DI back-ends.* Channels are implemented in a way that data and handshake signals are encoded together and the resulting implementation is guaranteed to be DI. However, the encoding requires a substantial overhead and the resulting circuits are around two to three times bigger and around five times slower [30] than the equivalent bundled data implementations.

Memory access times are critical for overall performance of a processor system and the MMU introduces additional overhead for every memory access. Therefore the bundled data back-end was chosen, even though it does require additional verification before the design could be deployed.

### 2.4.2 Syntax Directed Translation

Balsa uses syntax directed translation to generate its handshake circuits. There is a fixed mapping from language elements to handshake components, which gives the designer good control over the final circuit. This contrasts with many non-syntax directed translated languages like VHDL or Verilog. In these languages small changes in the source description can lead to significant changes in the resulting circuit [4]. This gives the designer less control over the actual implementation and may make it more difficult to reach specific performance targets or comply with other constraints. Furthermore it may lead to valid language descriptions which may be simulated, but which cannot be synthesised into circuits.

### 2.4.3 LARD

LARD[3] [14] was developed by Endecott at the University of Manchester. It is a fully fledged programming language, which was specifically developed to perform behavioural simulation of asynchronous systems. A tutorial and language reference may be obtained from the LARD Documentation Home Page [12].

Balsa uses LARD descriptions of handshake component circuits (generated by breeze2lard) to perform behavioural simulation. It enables the Balsa user to use the available LARD tools to aid in analyzing and debugging the Balsa description. Furthermore, it facilitates easy integration with other LARD programs; this is especially useful for designing complex test benches for Balsa generated circuits. Even though Balsa itself can be used to generate test benches, it does not include any file input/output capabilities. This often becomes a problem for testing circuits of reasonable complexity, as necessary test vectors and correct answers need to be either hard-coded or generated on-the-fly. Therefore, for greater flexibility, LARD test benches are often used which allow the test data to be accessed from files.

Any LARD simulation is only behavioural. LARD supports some forms of back annotation of timing data but, according to the author of the Balsa system, no reliable absolute or relative timing information can be derived from the LARD simulation of Balsa circuits at present. However LARD can be helpful in performance analysis in other ways, as detailed in a paper by Endecott [13].

### 2.4.4 Why were no LARD simulations used ?

No LARD simulation was performed during the design of the MMUs described in this thesis. All behavioural simulation was skipped in favour of functional

[3] It is available on the Internet from http://www.cs.man.ac.uk/amulet/projects/lard

simulation of a transistor-level netlist with Timemill only, described in chapter 5.

Performance comparison of the developed MMU implementations requires some timing analysis and estimations of actual delays. Besides the performance aspect, additional timing verification of the synthesized circuits is necessary due to the specific back-end chosen (also refer to section 2.4.1). As outlined in the previous section, LARD simulations cannot provide reliable absolute timing simulations and therefore it became clear early on that transistor-level simulation needs to be performed. As the necessary testbench to allow testing of the memory management sub-system is quite complex (refer to chapter 5) and no additional information could be gained from performing the behavioural simulation, it was decided to forgo the convenience of the behavioural LARD simulation and only perform Timemill transistor level simulations.

The next two chapters cover the design and implementation of the system coprocessor and two memory management units, commencing with the introduction of the coprocessor architecture and implementation.

# Chapter 3

# The System Coprocessor

In the ARM architecture control of the MMU is performed via the system coprocessor (cp15). Even though it is not solely dedicated to control of the MMU, a cp15 implementation had to be developed to allow use of the MMU.

The existing ARM coprocessor interface is described in section 3.1.1 and an alternative is proposed in section 3.1.3. These sections are followed by a description (section 3.2) of the system coprocessor implementation using the new interface and a discussion (section 3.3) on various aspects of the interface and the coprocessor implementation.

## 3.1 The Coprocessor Interface

### 3.1.1 ARM Coprocessor Interface

The ARM architecture allows up to sixteen coprocessors to be present in a system. They all interact with the processor via the coprocessor interface. All ARM coprocessors use a load-store architecture and are accessed via dedicated coprocessor instructions. There are three different categories of coprocessor instructions in the ARM instruction set:

- *Coprocessor register transfer* instructions to transfer data between registers on the processor and the coprocessor.
- *Coprocessor data transfer* instructions to transfer data between the memory and the coprocessor. The addresses are generated by the processor.

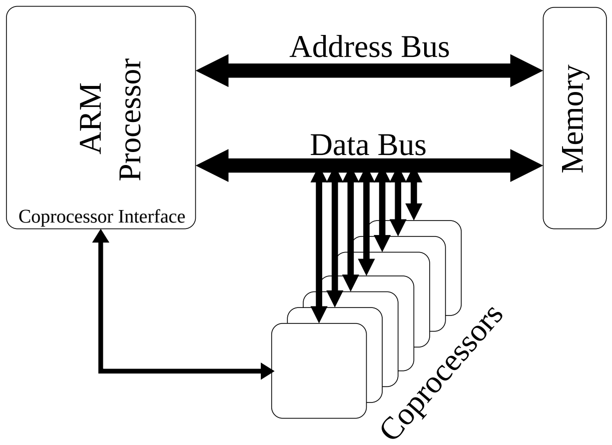


Figure 3.1: Coprocessors in an ARM system

- *Coprocessor data operations.* A coprocessor internal processing instruction, which only affects registers on the coprocessor.

The physical interface (figure 3.1) between the processor and coprocessor was designed around the idea of allowing easy and flexible off-chip addition of coprocessors. It is based on the coprocessors observing instructions on the data bus (also called *bus snooping*).

When a coprocessor instruction is fetched, it enters the pipeline of both the processor and the coprocessor, if present. In a subsequent cycle, when the processor is expected to execute the instruction, both the processor and the coprocessor assert dedicated handshake signals. Depending on the combination of these signals (see table 3.2), the following actions may be taken:

- The instruction is *discarded*. The instruction may have failed the condition test or be in a branch shadow.

- An *undefined instruction trap* is taken if the coprocessor being accessed in the instruction is not present in the system. This enables software emulation of coprocessors.

- The instruction is *delayed*. If the coprocessor is present but busy with another operation, the processor is free to wait or to continue and retry later.

- The instruction is executed *(it commits)*. Depending on the type of instruction, data might be transferred or a memory access might take place.

| CPU | Coprocessor | Result |
|---|---|---|
| discard | any Response | instruction discarded |
| commit | can't handle | undefined instruction trap |
| commit | busy | processor waits or retries later |
| commit | can handle | coprocessor commits |

Table 3.1: Possible handshakes in the synchronous coprocessor interface

The coprocessor may start execution of the instruction at any point, however it has to be able to recover its state if the instruction does not ultimately commit [16].

### 3.1.2 Drawbacks of the ARM Coprocessor Interfaces

The existing ARM coprocessor interface has some potential shortcomings. In a synchronous implementation, a coprocessor requires precise knowledge of the pipeline structure, to know when an instruction has reached the execution stage. This ties coprocessor implementations to a specific architecture implementation.

The interface also requires in-order, single instruction issue, because otherwise, the computation of the number of cycles between fetch and execution becomes very complex. Bus snooping can lead to up to 16 additional instruction decode units (inside the coprocessors) attached to the data bus, increasing power consumption and bus loading. Furthermore, as the processor does not know which coprocessors are present it has to start execution of all coprocessor instructions.

The interface also does not lend itself to an asynchronous implementation as counting cycles is not possible in an asynchronous system (which has no global synchronised cycles). An alternative method of synchronisation between processor and coprocessor is required to emulate the behaviour of the synchronous interface.

The off-chip expansion capabilities, one of the main motivations for the existing design, is unfeasible in most modern implementations due to the high bus speeds used. It is also unnecessary as embedded designs move towards the integration of the entire system on a single chip.

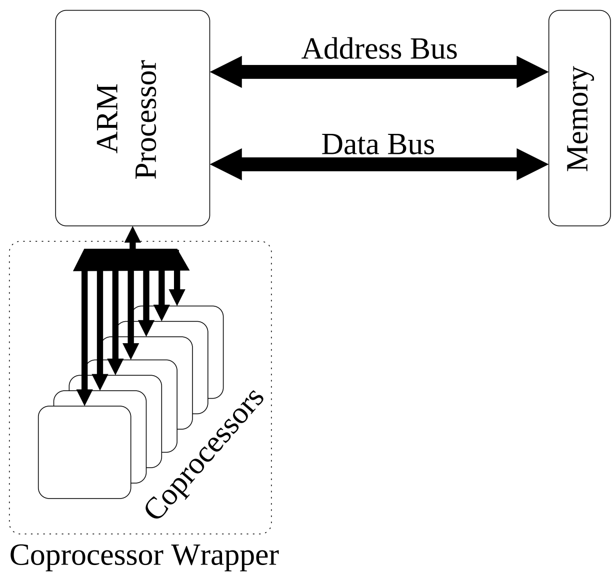


Figure 3.2: The new coprocessor interface

### 3.1.3 New Coprocessor Interface

A new coprocessor interface was developed to be used on the SPA [24] line of asynchronous processors. It replaces the obsolete off-chip expansion capabilities of the original interface with closer integration of processor and coprocessors which results in a static coprocessor map (i.e. the number and type of coprocessors is fixed). The coprocessors are not observing the data bus anymore, but instead are interfaced only to the processor (figure 3.2).

Execution of a coprocessor instruction requires three steps, of which each is associated with a part of the coprocessor interface:

1. *Instruction Decode.* If a coprocessor instruction is decoded by the processor, a copy is passed to the coprocessor wrapper. The coprocessor wrapper either sends a negative reply to the processor (if the addressed coprocessor is not present) or steers the instruction to the appropriate coprocessor. The coprocessor's decode unit sends a reply back to the processor, indicating whether it will accept the instruction or not.

2. *Instruction Execute.* After a coprocessor instruction has been accepted, the coprocessor performs a handshake with the processor's execution unit. Not all fetched instructions will commit as they might be in a 'branch

| Signal Name | Width | Direction | Notes |
|---|---|---|---|
| cpInstruction | 25 | input | Not all instruction bits are needed by the coprocessor |
| procMode | 4 | input | Some coprocessors only allow access from specific processor modes |
| cpReply | 1 | output | Indicates acceptance of instruction |
| Commit | 1 | input | Indicates whether instruction is to commit |
| procToCp | 32 | input | Transfer data to coprocessor |
| cpToProc | 32 | output | Transfer data to processor |
| memReadData | 32 | input | Receive data from memory (via processor) |
| memWriteData | 32 | output | Send data to memory (via processor) |
| memIsLastAccess | 1 | output | Boolean flag transferred with every word |

Table 3.2: Signals and their function in the new coprocessor interface

shadow' or the execution predicate might not be met[1]. A flag is transferred which indicates to the coprocessor whether the instruction should commit. Depending on the type of instruction, data might then be transferred either to or from the coprocessor.

3. *Memory Access*. This portion of the interface is only used for coprocessors which require memory accesses. Unlike the original design, coprocessors are not directly connected to the data memory bus, instead memory is accessed via the processor. All memory data is transferred via dedicated ports on the coprocessor interface. The processor generates consecutive memory addresses and a flag is sent by the coprocessor, indicating when the memory transfer has been completed.

Table 3.2 gives a complete list of all coprocessor interface signals and their use.

### 3.1.4 Interface Additions

Currently all ARM implementations execute instructions in the same order as they appear in the code being executed, one instruction at a time (*single issue*, with *in-order execution*). However many other processors reshuffle the order of instructions and attempt to execute several of them at the same time (*multiple issue*, with *out-of-order execution*) in a bid to achieve higher performance.

The standard ARM coprocessor interface, as well as the interface proposed in the preceding section, is only really suited to in-order single issue processors. This section describes a possible addition to the new coprocessor interface

[1]In the ARM architecture *all* instructions can be predicated, i.e. their execution depends on the state of the condition codes (similar to conditional branch instructions).

| Signal Name | Width | Direction | Notes |
|---|---|---|---|
| cpInstruction | 25 | input | Not all instruction bits are needed by the coprocessor |
| ***decodeTag*** | ***IMP*** | ***input*** | ***Tag to identify the instruction send*** |
| procMode | 4 | input | Some coprocessors only allow access from specific processor modes |
| cpReply | *3* | output | Indicates acceptance of instruction *and further detail* |
| Commit | 1 | input | Indicates whether instruction is to commit |
| ***executeTag*** | ***IMP*** | ***input*** | ***Indicates the instruction to execute*** |
| procToCp | 32 | input | Transfer data to coprocessor |
| cpToProc | 32 | output | Transfer data to processor |
| memReadData | 32 | input | Receive data from memory (via processor) |
| memWriteData | 32 | output | Send data to memory (via processor) |
| ***memoryTag*** | ***IMP*** | ***input*** | ***Indicates the instruction accessing memory*** |
| memIsLastAccess | 1 | output | Boolean flag transferred with every word |

Table 3.3: The extended new coprocessor interface

which makes it suitable for use with multiple-issue out-of-order execution based processor implementations. The required overhead is quite small enabling it to be used with current designs to guarantee interoperability with future processors.

In any out-of-order scheme an instruction must be uniquely identifiable, as its position in the fetch stream does not correspond anymore to its position in the execution stream. The new interface solves the problem by associating *tags* with coprocessor instructions. At each of the three stages of coprocessor instruction execution the tag is transferred to indicate the coprocessor instruction concerned.

In the current implementation the tag is a sequence of bits which needs to be unique for all instructions which could be present in the pipeline at the same time. Hence if sequential tags are used starting from zero, the largest required tag is therefore $\lceil log_2(\text{number of pipeline stages})\rceil$ bits in length.

The processor also requires knowledge of dependencies between instructions to reorder them and still guarantee correct execution. Problems arise with coprocessor instructions due to the side effects of some instructions (e.g. enabling MMU, disabling MPU, etc.) and due to the flexibility of coprocessor instruction interpretation by the coprocessor.

Unlike the instructions for the processor, where it is precisely defined which bits of the instructions specify source and target registers, the instructions for use by coprocessors are more flexible, only suggesting which fields are to be used

| System Coprocessor Register | Function | Notes |
|---|---|---|
| c1 | MMU Control Flags | Enable, Alignment, SR Flags |
| c2 | Translation Table Base | Start of page tables |
| c3 | Domain Access Control | Permissions for all domains |
| c5 | Fault Status (FS) | Error Code and Domain |
| c6 | Fault Address (FAR) | Virtual address |
| c8 | TLB Flush | Invalidate TLB entries |
| c10 | TLB Lockdown | Lock entries in TLB |

Table 3.4: MMU related registers in the system coprocessor

for target and source registers. Due to the special requirements of individual coprocessors, the recommended scheme cannot always be applied.

Therefore the coprocessor reply needs to include a flag specifying whether the instruction is allowed to be reordered. Any instruction which has side-effects or whose register allocation does not adhere to the standard scheme needs to be flagged non-reorderable. A second flag is necessary to specify whether the coprocessor itself allows out-of-order execution.

Even if the coprocessor requires 'its' instructions to stay in-order the processor can still reorder reorderable coprocessor instructions, as long as it does not change the order of instructions for the non-reorder capable coprocessor.

Table 3.3 gives an overview over all signals in the extended interface. All changes with respect to the interface described in section 3.1.3 are marked in bold italics (IMP means the size is implementation dependent).

## 3.2 System Coprocessor Implementation

The system coprocessor (cp15) provides access to the registers necessary to control the MMU. Table 3.4 gives an overview over the MMU related registers it contains[2]. The following sections (section 3.2.2 and 3.2.3) describe the architecture and implementation of the developed system coprocessor.

[2]Register c10 was not implemented as the TLB implementations do not support entry lockdown.

| | c0 | c1 | c2 | c3 | c5 | c6 | c8 |
|---|---|---|---|---|---|---|---|
| **Processor readable** | X | X | X | X | X | X | |
| **Processor writeable** | | X | X | X | X | X | X |
| **MMU readable** | | X | X | X | | | X |
| **MMU writeable** | | | | | X | X | |

Table 3.5: Access constraints on system coprocessor registers

### 3.2.1 Requirements

The system coprocessor is a relatively simple coprocessor as it only supports coprocessor register transfer instructions. Furthermore it is accessed very infrequently and hence instruction execution speed is not crucial to overall system performance. However, access to registers on the system coprocessor is expected to occur frequently in successive statements (i.e. when changing mappings, or after an abort) and this should not overly impair processor performance.

Access to the coprocessor registers is subject to certain constraints. Most of the registers can be read and written to by the processor and only read by the MMU, whilst others are also writable by the MMU, or only allow write-only access from the processor. The constraints are summarised in table 3.5.

### 3.2.2 Architecture

The register bank containing the MMU control registers is the central feature of the system coprocessor. Two access paths need to be provided: One for the MMU and the other for the processor. This section covers the access path for the processor, whilst the next section covers the register bank itself and the MMU's access method.

The processor manipulates coprocessor registers via dedicated coprocessor register transfer instructions. As outlined in the section on the coprocessor interface (section 3.1.3) there are (maximally) three steps necessary to perform a coprocessor instruction. However the system coprocessor never needs to access memory, leaving only two steps: instruction decode and instruction execution (the transfer of data). Even though execution speed of coprocessor instructions is not crucial for cp15, each step is performed by a dedicated stage connected via a FIFO.

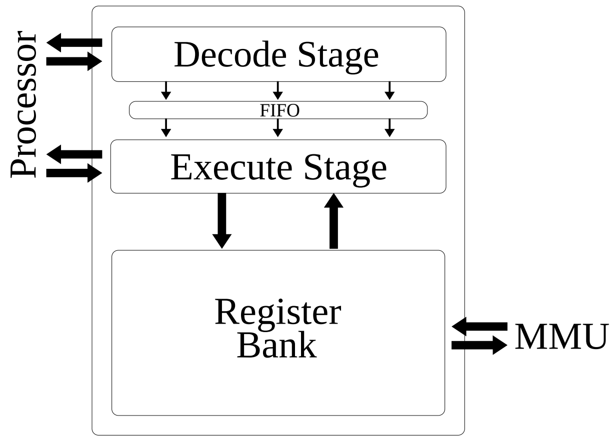


Figure 3.3: Architecture of the system coprocessor implementation

The first stage, the *decode stage*, decodes the coprocessor instruction and decides whether it is a valid system coprocessor instruction; (the instruction needs to be of type 'coprocessor register transfer' and the processor needs to be in a privileged mode for the instruction to be accepted.). The appropriate reply is sent back to the processor and, if the instruction is accepted, it is passed on to the second pipeline stage via a FIFO.

The *execute stage* waits for the processor to send the instruction tag and commit flag. Because the coprocessor registers are a contended resource (coprocessor and MMU require access), access to it is controlled by a locking mechanism. Hence, if the pending instruction is to commit the execute stage needs to acquire the lock for the register bank, before it can perform the required data transfer and finish execution of the instruction (the locking implementation is detailed in the next section). The processor is not allowed to proceed until the coprocessor access has completed.

Separating instruction decode and instruction execute via a FIFO allows both stages to do work simultaneously. This is important as system coprocessor instructions are often expected to occur in quick succession. If the coprocessor implementation is not pipelined the coprocessor can only accept a new instruction after it has completed execution of the previous one. This would force the processor to stall instruction decoding whilst waiting for the coprocessor to accept its next instruction[3].

[3] In an asynchronous system an incomplete handshake cannot terminate, there is no concept

### 3.2.3 MMU $\leftrightarrow$ Coprocessor Communication

The register bank in the system coprocessor needs to be accessed from the processor (via the coprocessor) and the MMU. These accesses to the coprocessor registers are not mutually exclusive, i.e. the MMU might want to access the cp15 registers while the coprocessor is accessing them and vice versa. An instruction prefetch, for example, could require an address translation through the MMU whilst the MMU control registers are being modified by the processor. Concurrent read and write of a register is not possible and hence some form of *arbitration* (refer to section 1.2.4) between the access requests becomes necessary.

Arbitration on a per-access basis is suitable for processor operations, as each access represents a single instruction. The MMU, however, requires more than a single read access per translation. It needs to be guaranteed that all of the reads refer to the same state of cp15 registers. Hence a locking mechanism was introduced to allow the MMU to prevent access by the coprocessor's execute stage.

The register bank implementation used by the base architecture is shown in listing 3.1. The `arbitrate` statement on line 10 grants access to either the coprocessor (lines 11-25) or a locking request of the MMU (line 28)[4]. If the MMU wins arbitration, it locks the register bank, by setting the `locked` variable (line 28), the `while` loop (lines 31-49) will then grant sole access to the registers to the MMU until a handshake on the `cpRegRelease` port (line 32-33) has completed. If the arbitration process was won by the processor, the `locked` variable is not set when the `while` loop (line 31-49) is reached which forces the loop body to be skipped. While the register bank is locked by the MMU, it can access the cp15 registers via dedicated ports (lines 34-47). These accesses have to be guaranteed to be non-overlapping, as the `select` statement (line 32) does not allow concurrent activation of its clauses. Once the `while` loop has terminated, the outer `loop` (lines 9-50) restarts the arbitration cycle, waiting for the next coprocessor register access request.

of a 'timeout'.

[4]The explanation treats Balsa as a regular programming language. This is only partially correct, as it describes an logic circuit, which does not have a notion of execute, call, etc. However, to simplify the explanations this distinction will only be made explicit where necessary.

```
procedure registerBank(
    [...]
)
is
local  [...]
begin
    init ();
    loop
        arbitrate regAddr then
            if (regAddr.rw = read) then
                case regAddr.register of
                     [...]
                else
                    regRead <- 0 --Always return 0 for unimplemented registers
                end
            else
                select regWrite then
                    case regAddr.register of
                         [...]
                    else
                        continue --Ignore writes to unimplemented registers
                    end
                end
            end

        | cpRegLock then -- Arbitration between regAddr and cpRegLock !
            locked:=1
        end;

        while locked then
            select cpRegRelease then
                locked:=0
            | mmuEnableOut <- mmuEnable then
                continue
            | alignmentCheckingOut <- alignmentChecking then
                continue
            | srPermsOut <- {s,r} then
                continue
            | tableBaseOut <- (#c2[14..31] as MMUTableBase) then
                continue
            | domainRegOut <- c3 then
                continue
            | faultStatus  then
                c5:= faultStatus
            | faultAddress then
                c6:= faultAddress
            end
        end
    end
end
```

Listing 3.1: Baseline Architecture system coprocessor register bank implementation

## 3.3 Discussion

### 3.3.1 Coprocessor Interface Comparison

#### Capabilities

The capabilities of the standard synchronous interface and the proposed asynchronous coprocessor interface in its basic form are nearly identical. The only difference is that in the original design the processor is able to interrupt a busy-waiting coprocessor. Busy-waiting occurs if the coprocessor is busy with a previous instruction and forces the processor to stall until the coprocessor can accept the next instruction. In the synchronous interface the processor can force the coprocessor to abandon a pending coprocessor instruction, by using a dedicated signal. This enables the processor to guarantee a maximum interrupt latency. The synchronous system can interrupt the coprocessor because the coprocessor samples the interface status lines every clock cycle. Hence the processor gets a chance every clock cycle to abort the pending coprocessor instruction to take an interrupt.

The asynchronous interface has no such option for the processor to force the coprocessor to abandon an instruction after it has issued the instruction to the coprocessor. Once a handshake (either at the execute stage or at the decode stage) has been initiated, it has to be completed. Due to the arbitration inside the register bank, the execute stage could, theoretically, take infinitely long to commit. This in turn results in a theoretically infinite interrupt latency for the attached processor. However, in practise arbitration resolves in a finite time and the interrupt latency of an actual system should not be noticably afftected by the arbitration delay.

The extended asynchronous interface (section 3.1.4) adds capabilities not found in the standard interface. It is suitable for use with multiple issue, out-of-order execution implementations of the ARM architecture. With the current interface, a superscalar processor would need to force in-order execution of coprocessor instructions (refer to section 3.1.4). This could result in significant system performance degradation for accesses to coprocessors which are used more frequently than the system coprocessor, i.e. the floating point coprocessor. Even

| | Synchronous Standard Interface | Asynchronous Basic Interface | Asynchronous Extended Interface |
|---|---|---|---|
| **Data** | 109 | 160 | 171 |
| **Synchronisation** | 1 | 18 | 24 |

Table 3.6: Number of wires needed by the coprocessor interfaces

though the extended interface presented here is an asynchronous implementation, it is equally suited for a synchronous implementation.

#### Complexity

The number of wires necessary to transfer the information across the interface depends on the asynchronous implementation style. Different data encodings result in different numbers of wires. In the comparison with the synchronous standard interface, a bundled data implementation (which was used for all designs in this thesis) is assumed. Table 3.6 shows the number of wires needed for each interface. The synchronous interface refers to an ARM9E-S [2] processor, which also only supports on-chip coprocessors and uses uni-directional buses. The extended asynchronous interface assumes a tag length of three bits (which is enough for processors with up to 8 pipeline stages).

The asynchronous interface requires more data signals, as it transfers memory data and register data via different ports, whilst the ARM9E-S uses the data bus to transfer all information. This uses fewer wires, but prevents data memory accesses during that cycle. The synchronisation overhead of the asynchronous implementation is high, as two signals are needed for handshake signals on each port of the interface. The synchronous implementation in contrast only requires the clock signal for global synchronisation.

However, as long as the coprocessors remain on-chip the number of wires should not pose a problem, with the difference between the interfaces being only around 70 wires.

### 3.3.2 MMU $\leftrightarrow$ Coprocessor Communication Issues

Writes to registers in the system coprocessor usually are intended to change operation of the MMU. The MMU might be activated or deactivated, permission information or the address mapping may be changed. The question is 'when do the changes take effect?'

Several features of modern processors make it non-trivial to determine which memory access will be the first to use the modified MMU state:

- *Pipelining.* Processors are divided into several pipeline stages, each performing only one step of a complete instruction. This results in several instructions pending execution at most times. All pending instructions have already been fetched from memory and, if one of the pending instructions changes MMU state, none of the other pending instructions that should be affected will be.

- *Prefetching.* Because memory accesses are slow compared to the cycle time of modern processors, instructions are fetched from memory speculatively. They are placed in a prefetch buffer from which the processor takes the instruction at some later time. *Some* of the prefetched instructions will have been fetched before MMU state has changed others, possibly, afterwards.

- *Interrupts.* When an interrupt occurs the processor proceeds to execute an appropriate interrupt handler. This is transparent to the application programmer as the processor will resume execution of the application code at the point it was interrupted. However, this in effect means that – as long as interrupts are enabled – an arbitrary number of instructions might be executed between any two instructions of a program.

- *Cache Misses.* Most instructions will be fetched from an instruction cache (a cache is a very fast piece of memory buffering a working set of instructions and data for fast access.). Occasionally however, a requested memory item will not be present in the cache and needs to be fetched from memory. Accesses to memory (cache-fills) take many more cycles than cache hits, hence the number of instructions (pre-)fetched in a specific amount of time can vary greatly. If a change of MMU state takes, say, 20ns to take effect, the number of instructions fetched/executed depends on the state of the cache.

Due to these complications, changes in MMU state require implementation dependent sequences of instructions, to guarantee safe and predictable behaviour.

These need to be executed in privileged mode and hence are usually implemented as system calls. The code should be located in a part of memory which is flat-mapped, i.e. where virtual addresses and physical addresses are identical. This guarantees that the code does not suddenly 'dissappear' when the MMU starts translating addresses.

In synchronous implementations a fixed number of cycles pass, from execution of an instruction which changes a register in the system coprocessor until it affects MMU operation. If caches and interrupts are disabled this translates into a predictable number of instructions which execute before changes take effect.

Asynchronous implementation do not have global synchrony and there are no system cycles to count. An asynchronous pipeline moves at a variable speed and the number of instruction pending execution is not fixed. Hence the number of instructions executed before a change in system coprocessor state takes effect is variable, even if caches and interrupts are disabled. An asynchronous implementation can only be designed to guarantee an *upper bound* for the number of instructions which might be fetched or executed before a new MMU state is active. As long as the piece of code changing MMU state resides in a piece of memory which is not relocated during its execution and is executed in privileged mode, safe operation can still be guaranteed.

The system coprocessor implementation detailed in this chapter forces the coprocessor execution to stall until the coprocessor write has completed (by not completing the handshake on the write data). This is necessary as writes to the system processor's register bank take an arbitrary time to complete (refer to section 3.2.3). If the processor was allowed to proceed before a write to the system coprocessor had completed, the number of instructions executed before the changes of MMU state become effective, would be unbounded. The actual bound is dependent on the processor implementation, especially its pipeline structure and prefetch buffer size. However, when using the proposed cp15 implementation it is guaranteed that the instruction after the cp15 write instruction can only *execute*[5] when system coprocessor state has already changed. If this instruction following the cp15 write instruction forces the prefetch buffer and pipeline to be

[5]It might even be fetched with the new MMU state, as we do not know when it was fetched!

invalidated (i.e. returning from the system call via a branch with mode change), the MMU state changing code becomes independent of the processor implementation, as all instructions which might have been prefetched with the old memory map will be discarded.

Now that the control of the MMU and its activation has been covered, it is time to move on to the main focus of this thesis. The next chapter proposes two asynchronous memory management units and their implementation.

# Chapter 4

# The Memory Management Unit

The previous chapter introduced the system coprocessor, which controls operation of the MMU, and described its interaction with the MMU. This chapter introduces the memory management units.

After a short overview of specifics of ARM memory management (sections 4.1.1 - 4.1.3) two MMU architectures are described. The Baseline Architecture (sections 4.2 - 4.2.4) is a simple, non-pipelined design, whilst the Performance Architecture (sections 4.3.1 - 4.3.5) uses pipelining and attempts to extract more concurrency for better performance. Some aspects of the colour implementations are described in more depth in section 4.4. A detailed description of the TLB implementation finishes the chapter. The performance of the MMU implementations and their merits are analysed and discussed in chapter 6.

## 4.1 ARM Memory Management

Memory Management on ARM systems has to accommodate a wide range of requirements. Embedded systems may have only a relatively small amount of memory and require fine mapping and protection granularity. Other systems might have to meet stringent real-time OS constraints. These different requirements are all reflected in the MMU and TLB specifications.

| Page size | Memory Size | Notes |
|---|---|---|
| Section | 1MB | Requires only single descriptor fetch |
| Large | 64KB | Possible in *coarse* or *fine* page table |
| Small | 4KB | Possible in *coarse* or *fine* page table |
| Tiny | 1KB | Only possible in *fine* page table |

Table 4.1: Available page sizes and their constraints

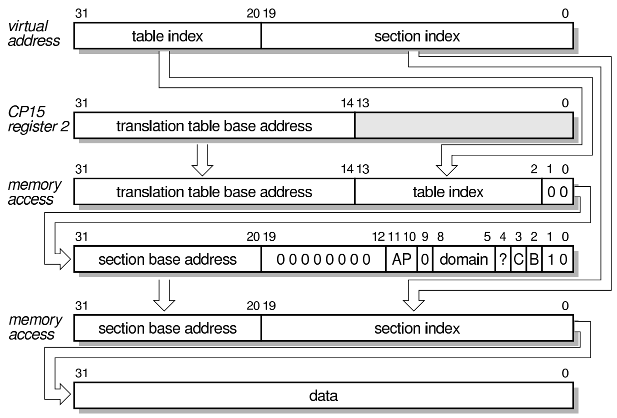


Figure 4.1: Translation of a Section from: S. Furber, ARM System Architecture, Addison-Wesley

### 4.1.1 Address Translation

The ARM architecture distinguishes two types of pages: those which use a single level page table and those which require a two level page table. Single level page tables are used only by *sections* which are very large pages, remapping 1MB portions of memory. Two level page tables come in two varieties: *coarse* and *fine*. Coarse page tables support *large* (64KB) and *small* (4KB) pages, whilst fine page tables also offer support for *tiny* (1KB) pages (table 4.1).

All table lookups start with a *first level descriptor* (FLD) fetch from memory. The FLD indicates the type of page table used, along with table-type specific information; alternatively the address can be marked as invalid. If the FLD is part of a single level table it contains all information to complete the address translation (figure 4.1).

If the FLD does not refer to a section and is not marked as invalid, a *second level descriptor* (SLD) or *page table entry* (PTE) needs to be fetched from memory. Figure 4.2 illustrates the entire address translation process for large pages.

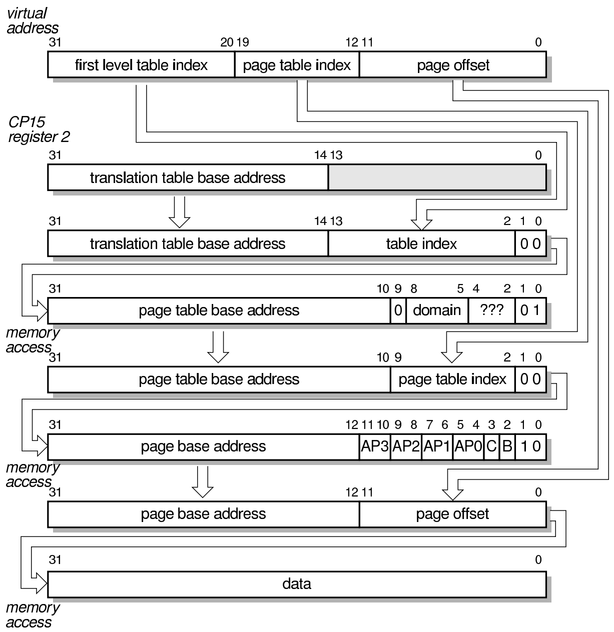


Figure 4.2: Translation of a Page from: S. Furber, ARM System Architecture, Addison-Wesley

| **Access Permissions** | **System flag** | **ROM flag** | **Privileged Permissions** | **User Permissions** |
|---|---|---|---|---|
| 0b00 | 0 | 0 | No access | No access |
| 0b00 | 1 | 0 | Read only | No access |
| 0b00 | 0 | 1 | Read only | Read only |
| 0b00 | 1 | 1 | Unpredictable | Unpredictable |
| 0b01 | x | x | Read/Write | No access |
| 0b10 | x | x | Read/Write | Read only |
| 0b11 | x | x | Read/Write | Read/Write |

Table 4.2: Access permission dependencies

### 4.1.2 Access Permissions

Access permission information for a page is contained in the descriptors. The interpretation of the access permission stored in the page tables depends on the processor mode and two flags (System Flag and ROM flag) stored in the system coprocessor (table 4.2). Sections and tiny pages only support access permissions on a per page basis. Large and small pages allow *subpages* (each being a quarter page) to have different access permissions, resulting in finer control of access permission than for address translation.

In addition to the page access permission, the ARM architecture features so called '*domains*' to allow fast and flexible overriding of page access permissions. Each FLD is marked as a member of one of sixteen available domains. A register in the system coprocessor contains the *domain access value* for each domain, applicable to the currently running process.

Domain access values can make the process a *domain manager*, a *domain client* or deny all access to pages which are part of the domain. Domain managers are granted full read/write access to all pages, overriding the page permissions, whilst domain clients are bound by them.

### 4.1.3 Memory Aborts

If a memory translation does not complete successfully, a memory abort is sent to the processor. There are five classes of aborts, which are checked in a fixed order (figure 4.3):

- *Alignment Aborts*. Alignment checking (i.e. checking whether an address is aligned to the access size) is not mandatory and can be disabled in the control registers.
- *Translation Aborts* occur when a descriptor is marked as invalid by the OS.
- *Domain Aborts* occur if the domain access value prohibits access to the domain.
- *Permission Aborts* occur if the access permission prohibits the requested type of access.
- *External Aborts*. These type of aborts are not generated within the MMU, but by the actual memory subsystem. The MMU only passes them on to the processor. They might be caused by a memory address being out-of-bounds or some parity mechanism. Depending on their cause, they might not be recoverable.

The abort behaviour of the processor and MMU depend on the type of memory access. Aborts on a data access cause the MMU to write the virtual address of the aborted access, along with an error code indicating the abort cause, to the Fault Address Register and Fault Status registers in the system coprocessor. If a data access is aborted the processor takes an exception and commences execution from the data abort exception vector. The operating system then typically either resolves the cause of the exception (e.g. maps in a page) or terminates the process

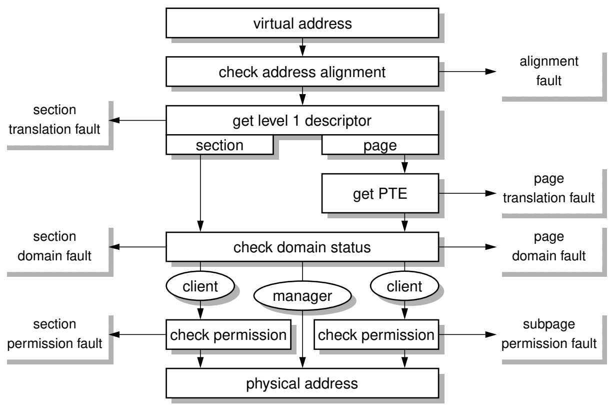


Figure 4.3: ARM MMU abort check sequence from: S. Furber, ARM System Architecture, Addison-Wesley

which caused the exception (i.e. a 'segfault') using the information in the FAR and FS registers.

Aborted instruction fetches need to be treated differently, due to *prefetching*. Modern processors usually fetch instructions speculatively to achieve better performance, as memory accesses are comparatively slow. Hence if an instruction fetch is aborted it is not yet known whether the instruction should be executed. Only if the aborted instruction is to be executed does the OS need to investigate the abort reason; hence there is no immediate action by the processor on an instruction abort. Because the processor does not always act on prefetch aborts, the MMU does not need to update the error code and abort address in the system coprocessor. The prefetch abort handler is able to deduce the address of the aborted instruction from the program counter at the time of the exception (which is saved by the processor as part of the exception entry).

### 4.1.4 Translation Lookaside Buffer (TLB)

The TLB is an integral part of the MMU address translation process, however implementations can differ substantially in features. The number of TLB entries and their organisation (associativity, and replacement strategy) varies between systems, to accommodate cost and performance constraints. Even the number of TLBs varies, with some systems offering separate TLBs for data and instruction accesses whilst others use a unified TLB.

TLB operation is transparent to the application programmer. The operating system has to manage two aspects of TLB operation: *TLB entry invalidation* (TLB flush) and *entry lockdown*.

TLB entry invalidation is necessary because the ARM MMU does not detect when page tables have been modified and hence it cannot know whether cached translations are still valid. The OS has to invalidate the cached mappings in the TLB explicitly when the active page table changes. The ARM architecture supports the invalidation of individual entries, however implementation support for this is optional. The TLB is allowed to invalidate all entries instead.

TLB entry lockdown is another optional feature, which prevents individual TLB entries from being replaced. It is used to ensure predictable latencies for time critical routines such as interrupt handlers. If this feature is not supported, worst case interrupt latency has to include the possibility of a TLB miss for an access to the interrupt handler. Like all memory management features it is controlled via a register in system coprocessor.

## 4.2 Baseline Architecture

The Baseline architecture (figure 4.4) is based on the abort check sequence diagram in figure 4.3. It is intended as a point of comparison to examine the effect of architectural and implementational changes. All steps are performed in order, without any overlap, to guarantee that no unnecessary activation of blocks occurs. The number of registers in the data path is kept to a minimum, instead data is only passed to blocks as required.

### 4.2.1 System Coprocessor Communication

Many components of the baseline architecture require information from registers in the system coprocessor to operate. As outlined in the previous chapter (section 3.2.3) the MMU has to get a lock for the coprocessor register bank, before it is allowed to access any cp15 registers.

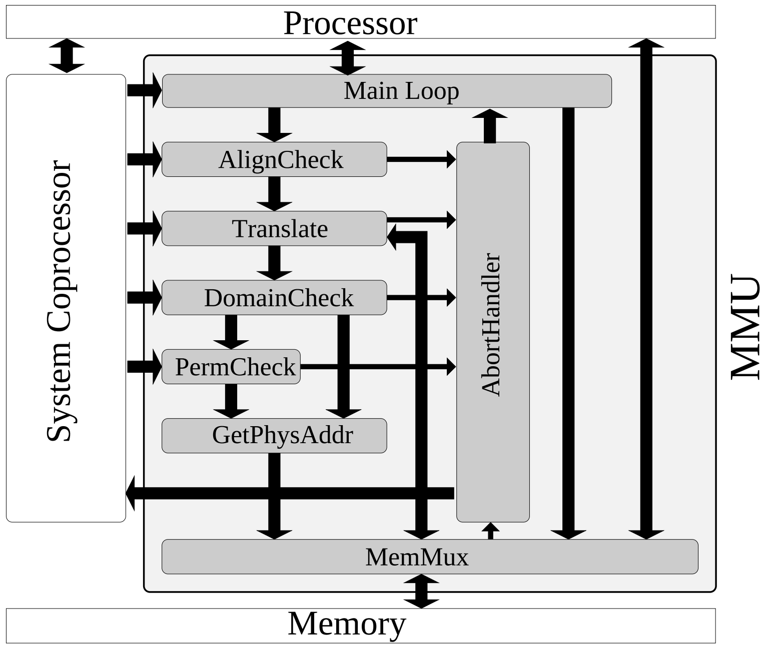


Figure 4.4: MMU Baseline Architecture

The baseline architecture acquires the lock when it receives a memory access request, and does not release it, until the access has completed (or aborted). This is necessary, as all values read from the system coprocessor have to refer to the same state of the coprocessor. Keeping the lock is necessary to prevent the processor from writing to the coprocessor registers and modifying MMU state during a memory access translation. There is no need for arbitration between coprocessor register accesses from the MMU, as components of the baseline MMU are only ever activated sequentially.

### 4.2.2 Untranslated Accesses

Once the register bank is locked, register c1 is read from the system coprocessor. If the MMU is not enabled, the memory access request is passed directly to the memory multiplexer. When the access has completed – successfully or with an external abort – the memory multiplexer releases the data access handshake with the processor. This guarantees that no further memory access can be initiated by the processor until the pending access has completed.

### 4.2.3 Translated Accesses

If the MMU is enabled, the memory access address needs to be translated before it can be passed on to memory. Each translation step shown in figure 4.3 is performed by a separate component. When the operation performed by a component completes successfully the next component is activated. If an abort condition is detected an abort signal is sent to the Abort Handler (section 4.2.4). The components of the Baseline Architecture (as shown in figure 4.4) are:

1. *AlignCheck* performs an alignment check, if alignment checking is enabled in the system coprocessor (Byte sized accesses are always aligned, but words or half words have constraints on their addresses).

2. *Translate* performs the translation of virtual page numbers to physical page numbers. It contains the TLB and table walking hardware. If the TLB has not cached a translation the table walking hardware performs a page table lookup in memory.

3. *DomainCheck* checks the domain access permissions for the pending memory access. Domain Client accesses are passed to the permission check, Domain Manager accesses are passed straight to the physical address computation (as they do not require a permission check).

4. *PermCheck* performs the access permission check. The processor mode information (priviliged/non-priviliged) required is sent with each memory access by the processor.

5. *GetPhysAddr* computes the physical address by concatenation of the physical page number with the page offset.

6. *MemMux* is the memory multiplexer. Memory can be accessed for three different reasons: descriptor fetches (from Translate block), translated memory accesses (from GetPhysAddr block) and untranslated memory accesses (from processor). The MemMux multiplexes the access requests from each source to physical memory. (As all accesses are guaranteed to be non overlapping there is no need for arbitration in this block.)

7. *AbortHandler*. This is the central component of the baseline architecture. All blocks which can cause a memory abort are connected to it. On data

| Component | cp15 Register | cp15 Access Type | Possible Successors | Abort Cause |
|---|---|---|---|---|
| AlignCheck | c1 | read | Translate, AbortHandler | Alignment Fault |
| Translate | c2 | read | DomainCheck, MemMux, AbortHandler | Translation Fault |
| DomainCheck | c3 | read | PermCheck, GetPhysAddr, AbortHandler | Domain Fault |
| PermCheck | c1 | read | GetPhysAddr, AbortHandler | Permission Fault |
| GetPhysAddr | none | na | MemMux | none |
| MemMux | none | na | (Memory), AbortHandler | External Abort |
| AbortHandler | c5, c6 | write | none | none |

Table 4.3: Baseline Architecture component details

| Fault | Error Code |
|---|---|
| Alignment Abort | 0x1 or 0x3 |
| External Abort FLD fetch | 0xC |
| External Abort SLD fetch | 0xE |
| Invalid FLD | 0x5 |
| Invalid SLD | 0x7 |
| Section Domain Abort | 0x9 |
| Page Domain Abort | 0xB |
| Section Permission Abort | 0xD |
| Subpage Permission Abort | 0xF |
| Section Data Access External Abort | 0x8 |
| Page Data Access External Abort | 0xA |

Table 4.4: Abort reasons and their error codes

access aborts it updates the fault address and fault status registers in cp15. It is covered in more detail in the next section.

### 4.2.4 The Abort Handler

The Abort Handler is responsible for reporting memory aborts back to the processor and updating the error cause (FS) and abort address (FAR) registers in the system coprocessor. The error cause register is updated with a combination of a four bit error code detailing the reason for the abort (table 4.4) and the number of the domain accessed. (As mentioned previously, the system coprocessor registers are only updated on data aborts, not on prefetch aborts.)

On every translated access the AlignCheck component transfers the access address to the AbortHandler and the Translate component sends the page table type[1] as well as the accessed domain to the Abort Handler where they are stored in registers. All remaining MMU components are connected to the Abort Handler via simple control handshake channels (sync channels) which do not transfer any

[1]Error codes are different for sections and other pages (see table 4.4).

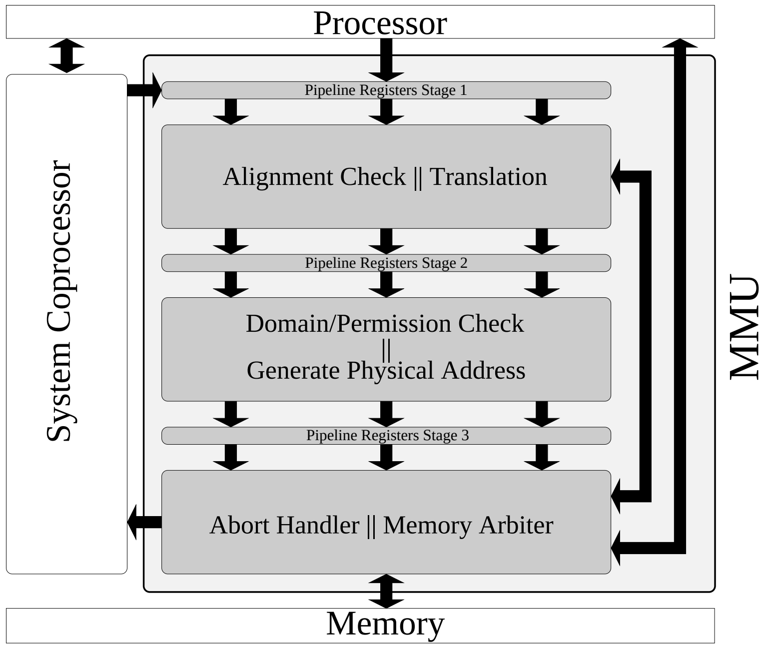


Figure 4.5: MMU Performance Architecture

data. The initiation of a handshake on one of these channels signals an abort to the Abort Handler and is sufficient for the handler to deduce the correct error code and update the system coprocessor accordingly (if required).

## 4.3 Performance Architecture

Due to the entirely sequential operation of the baseline architecture, it is optimised for memory aborts. In case of an abort in any block, execution of subsequent blocks is skipped and the next access can commence as early as possible. However memory aborts are extremely infrequent, around 0.00001% - 0.001% [21] of all memory accesses. Hence Amdahl's law predicts that good abort performance is unimportant for overall system performance as the non-abort behaviour accounts for over 99.99% of all accesses. A memory abort also results in the execution of the necessary abort code by the OS which requires thousands of instruction and, potentially, very slow disk accesses. This reduces the significance of MMU abort efficiency on overall system performance further.

The Performance Architecture (figure 4.5) aims to achieve higher performance than the Baseline Architecture by speculatively performing operations in parallel and pipelining (section 4.3.1) to allow multiple outstanding memory accesses. (Even though the architecture is designed to cope with multiple outstanding memory operations at a time, it makes no provision for allowing out-of-order completion of those accesses, as this requires architectural support from the processor and memory subsystem). Pipelining the design introduces an asynchronous specific problem: this problem, and its solution, is described in section 4.3.4.

### 4.3.1 Pipeline Structure

The Performance Architecture is divided into three pipeline stages (figure 4.5):

1. *Alignment Check and Translation.* Both these operations are performed concurrently. If a translation is not cached in the TLB, table walking commences. Descriptor fetches are performed via the memory arbiter in stage 3.

2. *Domain/Permission Check and Generate Physical Address.* The entire permission checking (domain and page access permissions) has been integrated into a common block. The physical address generation is separate from the translation stage to allow the next TLB access to occur as soon as possible.

3. *Abort Handler and Memory Arbiter.* Due to the pipelining, descriptor fetches and translated accesses do not exclude each other anymore, as in the Baseline Architecture. Hence arbitration becomes necessary.

Each pipeline stage only performs work if the preceding stage has not generated an abort. However, aborts are only reported to the processor by the abort handler in the third pipeline stage. Aborts cannot be reported to the processor at every stage, as this could result in aborts being send out-of-order to the processor.

### 4.3.2 Untranslated Accesses

Untranslated memory accesses also have to pass through the pipeline, just like translated accesses. Providing a separate access path to memory, which side-steps the pipeline would have introduced a race condition. Extra synchronisation would have become necessary to guarantee that memory accesses are not reordered[2]. As computer system with an MMU are expected to have memory management enabled most of the time the performance penalty of putting untranslated accesses through the pipeline does not justify the necessary synchronisation overhead.

Furthermore, the asynchronous design of the system aids the MMU performance for untranslated accesses. If the MMU is disabled each pipeline stage only passes data through. As the asynchronous implementation does not impose a fixed cycle time for each pipeline stage (as synchronous systems do), it leads to the pipeline running at a higher speed for untranslated accesses than for translated accesses.

### 4.3.3 System Coprocessor Communication

The Performance Architecture uses a different approach to communication with the system coprocessor from the Baseline Architecture. Multiple memory accesses may be pending and each may refer to a different state of the system coprocessor. Hence each access requires its own copy of the coprocessor registers.

Copies of all MMU control registers are taken at the first pipeline stage and progress down the pipeline together with the access information. Besides the mandatory read from the system coprocessor at the first pipeline stage, further write access to the system coprocessor may be necessary from the TLB or the abort handler. The TLB requires access to the system coprocessor after entry invalidation (section 4.5.3) and the abort handler has to update the Fault Address and Fault Status registers in the system coprocessor.

These accesses are not mutually exclusive and hence arbitration is needed for sending the locking request to the system coprocessor. Release of the lock however *is* mutually exclusive (as the system coprocessor lock can only be released by a component which 'holds' the lock and only one component can lock the coprocessor at any time) and therefore no arbitration is required for the release requests.

[2] For example a translated instruction could still be in the pipeline when the first untranslated access request is issued. Due to the asynchronous implementation the order in which these accesses occur would not be determined.

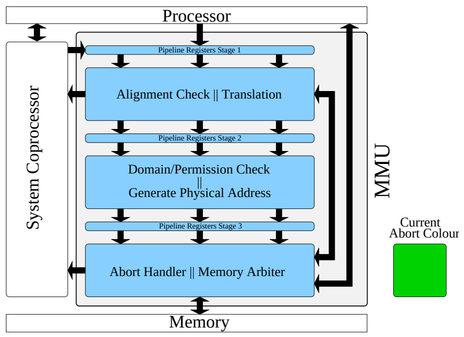


Figure 4.6: Initial state. Access colour: blue; Abort colour: red

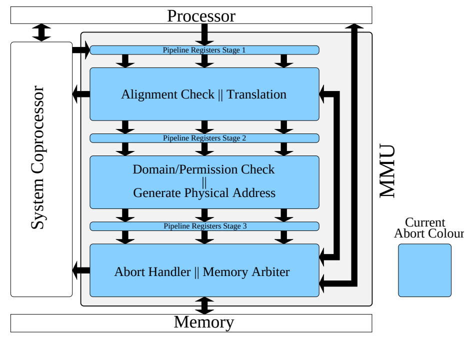


Figure 4.7: Abort occured. Access colour: blue; Abort colour: blue

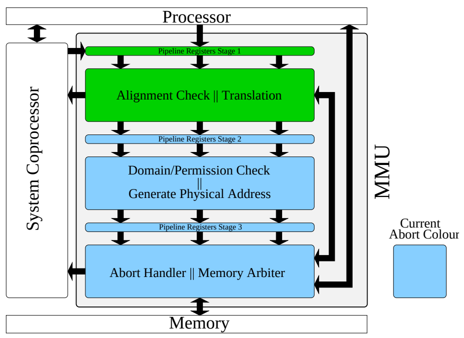


Figure 4.8: Processor changes access colour. Access colour: red; Abort colour: blue

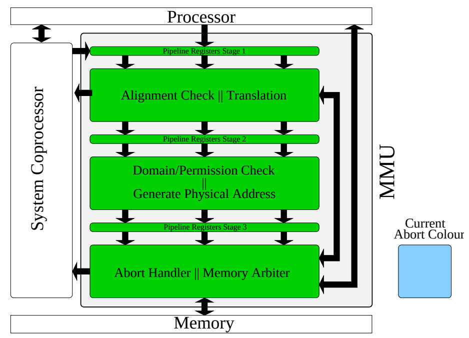


Figure 4.9: Colour change completed Access colour: red; Abort colour: blue

### 4.3.4 Memory Access Colour

In an asynchronous pipeline it is impossible to know how many pipeline stages are filled, i.e. how many outstanding memory accesses are in the pipeline, as movement between pipeline stages is not synchronised. In the MMU this results in problems when a memory access is aborted. If there are any outstanding accesses (memory accesses which were submitted by the processor before it received the abort) they should not be allowed to complete, as they could modify system state.

In synchronous system a pipeline advances one access every clock cycle. Hence cycle counting can be used to 'recognise' the first memory access which was issued *after* an abort. In an asynchronous pipeline it is problematic to know when speculative memoru accesses are flushed and memory accesses may complete again.The asynchronous MMU pipeline does not hold a fixed number of outstanding memory accesses at any time and hence (access-)cycle counting is not possible.

One way of resolving this problem of asynchronous pipelines is the notion of 'colour' developed at the University of Manchester by Nigel Paver [29]. Memory access passing through the pipeline are tagged with a colour. Only accesses tagged with a certain colour are allowed to complete, all others are aborted without updating the state of the system coprocessor. On an abort, the colour which determines which accesses are allowed to complete changes, to prohibit accesses tagged with the same colour as the aborted access.

A short example will illustrate the colour mechanism. Assuming the initial memory access colour is defined to be 'blue' all initial accesses will be tagged as 'blue' and all accesses tagged 'red' will be aborted (figure 4.6). When an instruction causes an abort, the abort manager sends an abort signal to the processor and changes the abort colour to 'blue' (figure 4.7). Once the processor receives the abort signal it tags all new valid accesses 'red' (figure 4.8). If there are still pending (blue) memory accesses in the pipeline, issued before the colour change, they are silently aborted by the MMU. Only when the first 'red' access reaches the final pipeline stage are memory accesses allowed to complete again (figure 4.9).

### 4.3.5 Colour and Instruction Fetches

Applying the colour mechanism as outlined in the previous section to all memory accesses (i.e. data accesses and instruction fetches) would lead to incorrectly aborted instructions. Three complications arise:

- *Instruction fetches and data accesses are not ordered.* After an abort, execution needs to restart at the 'earliest' instruction which caused an abort, to guarantee correct program behaviour. However, as instruction fetches and data memory accesses occur at different points in the processor pipeline they do not reflect the order of instructions in the program. If an aborted instruction fetch causes a colour abort for a pending data access, the data access refers to an instruction which executed several instruction before the instruction fetched. Hence there is no reason to prohibit the access. In fact, prohibiting the access requires the processor to restart execution from the unnecessarily aborted instruction. The processor needs to cater for these special cases and guarantee that a prefetch abort is only acted on when no 'earlier' data access is pending.

- *Not all aborted prefetched instructions are executed.* Prefetching is usually speculative. Not all prefetched instruction are executed. Hence invalidating any other memory accesses based on a prefetch abort is also speculative. When the processor receives a colour abort on a data access, it is not known whether the aborted prefetched instruction would execute.

- *No error code for colour aborts exists.* In the ARM archietcture there is no colour abort, and hence, no error code exists. Furthermore, the system co-processor's memory error status only needs to be updated for a colour abort on the first data access following a aborted prefetch access. Therefore the OS abort handler would need to be aware of the implementation-dependent colour aborts to react correctly.

There is also a potential for infinite abort loops. A sequence of instructions could be devised, which leads to a specifc instruction prefetch always occuring just before a specifc data access commits. If the instruction fetch causes a prefetch abort, the processor is forced to re-execute the instruction causing the wrongly aborted data access. However re-excution of the code can again result in the aborted instruction being prefetched (and aborted) just before the data access commits. Breaking such loops requires non-overlapping of memory accesses of different types (i.e. fill the MMU pipeline with only prefetch accesses, drain it, fill it only with data accesses, etc.).

Solutions can be devised for the problems mentioned, however they lead to considerable overhead, slow down overall system performance, require OS support and force non-ARM architecture conform, MMU implementation specific system behaviour. Providing separate colours for data and instruction accesses resolves the points outlined so far, but it can still lead to instructions being aborted unnecessarily (e.g. a prefetch might be aborted due to colour, even though the instruction causing the abort is never executed).

Due to the speculative nature of prefetching, the colour mechanism is not necessary for instruction fetches. Prefetching does not alter the state of a computer system and the processor discards any unnecessarily prefetched instructions in any case. Not having a colour mechanism in place for instruction fetches may result in unnecessary memory accesses after an instruction abort. However, instruction prefetching is speculative and, as prefetch aborts take many instructions to resolve, the performance impact of not using colour is negligible. On the other side implementing colour on prefteching can force unnecessary abort handler execution and requires the OS memory abort handler to be colour aware. Hence, no colour tagging is used for instruction fetches, only data accesses are tagged with a colour.

The implementation of the colour mechanism in the abort handler and memory arbiter is covered in the next section.

## 4.4 Colour Management Implementation

The colour mechanism outlined in the previous section results in the following requirements for third stage behaviour:

1. Only data memory accesses not tagged with the abort colour are allowed to change memory state.

2. The abort colour changes on an aborted data memory access.

3. The system coprocessor fault address and fault status registers are not updated on colour aborts (as they need to be transparent to the operating system).

The third stage consists of two components: the memory arbiter and the abort handler. They run concurrently but are not always active at the same time. The implementation of the colour mechanism is split between both. The abort handler is responsible for the colour management and error reporting, whilst the memory arbiter prevents wrongly coloured memory accesses from committing.

Colour information is sent from the abort handler to the memory arbiter when a (not already aborted) memory access request reaches the third pipeline stage. The memory arbiter in turn returns an abort status for each access, indicating whether the access was performed successfully. An accesses may not have completed either because it was aborted due to an incorrect colour tag, or because

```
function isColourAbort (
  accessColour : memColour;
  allowedColour : memColour
)
= accessColour /= allowedColour
```

Listing 4.1: TLB colour abort detection

```
shared abort
 is begin
    if isColourAbort(virtBundle.accessColour, allowedColour) or accessType=fetch then
        continue -- Don't update cp15 or allowedColour
    else
        toggleColor() || [update cp15]
    end
    ||
    sendAbort()
 end
```

Listing 4.2: MMU Colour abort implementation

an external abort occurred.

The colour check is performed by a simple function (listing 4.1) which compares the currently allowed colour (received from the abort handler) with the colour tag of the pending access. By separating the colour check into a separate function, changes in the colour encoding and abort rules can easily be facilitated. The abort handler uses the same function to control updates of the fault information in the system coprocessor in case of an abort (listing 4.2). The ‘allowed colour‘ is only updated on genuinely aborted data accesses, not on colour aborts (or instruction fetches). In the current implementation the colour is implemented as a single bit, which only needs to be inverted to generate the next allowed colour (listing 4.3).

## 4.5 Translation Lookaside Buffer Implementation

The translation lookaside buffer (TLB) is the central part of the Translate component. Without it every memory access would require one or two additional

```
type memColour is enumeration
    red, blue
over bit

variable allowedColour : memColour

shared toggleColor
is begin
    allowedColor := (not (allowedColor as bit) as memColour)
end
```

Listing 4.3: Toggling current MMU colour

memory accesses to perform page table look-ups. Three aspects of the TLB, as implemented in the performance architecture, will be introduced:

- *Entry Organisation* (section 4.5.1) covers the information stored in each TLB entry
- *Entry Lookup* (section 4.5.2) describes the entry look-up procedure
- *Entry Invalidation* (section 4.5.3) introduces the TLB entry invalidation mechanism

### 4.5.1 Entry Organisation

The TLB contains a user configurable even number of entries. Half the entries cache instruction fetch translations and the remaining half are dedicated to data access translations. Both halves of the TLB are direct mapped to keep the hardware simple and small.

Each entry is divided into the following fields:

- *Tag.* A TLB entry look-up is a hashing problem. A hashing function is applied to a virtual address to select the entry which can contain the cached translation (only one entry is selected due to the direct mapped organisation). However, several virtual addresses can result in the same hash. Hence, an entry found via the hash needs to contain some additional information to determine whether the selected entry contains the translation for the current virtual address. The additional required information is called the 'tag'. The MMU translates page numbers, the page offset remains unchanged. Hence each entry in the TLB should transfer a single page number, to maximise the amount of page translation cached in the TLB. This goal is unattainable with a fixed sized tag, as the ARM MMU supports multiple page sizes. The tag needs to be large enough to cover the largest page number field (i.e. the page number field for the smallest page size). The tag therefore consists of the top 22 bits of the virtual address. However not all bits used as the hash need to be stored, as some are implicit in the entry index (refer to next section). Hence the actual tag size is dependent on the number of bits used to encode the TLB index.

- *Valid Flag*. The flag specifies whether the information contained in the entry is valid. Refer to section 4.5.3 for more details.

- *Descriptor*. The current TLB implementation does not hold translated addresses and permission information separately. Instead it caches the last required descriptor (First Level Descriptor for sections, Second Level Descriptor for other pages).

- *Table Type*. This field contains the type of page table: 'section', 'coarse' or 'fine'. This information is only contained in the FLD and hence needs to be preserved for page translations when the descriptor field holds a SLD.

- *Page Type*. The type of page (section, large, small, tiny) can be deducted from the table type and the descriptor, however to speed up the permission check, the information is stored in a decoded form.

- *Domain*. The domain number of the translated page.

Listing 4.4 shows the relevant part of the Balsa implementation. The number of TLB entries (`TLB_ENTRIES`) is a constant which can be adapted to the intended application. (This nicely illustrates the benefit of using logic synthesis, as varying the number of TLB entries in a manual design requires tedious design modifications.)

### 4.5.2 Entry Lookup

A direct mapped entry organisation was chosen for the TLB. In a direct mapped TLB, each translation for a virtual address can only be cached in one TLB entry (however, many virtual addresses can refer to the same entry). Hence only one

```
constant TLB_ENTRIES = 8

constant TLB_INDEX_SIZE = (log TLB_ENTRIES)
constant TLB_INDEX_MSB = TLB_INDEX_SIZE − 1

variable tlb : array 0..(TLB_ENTRIES−1) of TlbEntry
variable tlbIndex : TLB_INDEX_SIZE bits


type TlbEntry is record
  tag : TAG_SIZE bits;
  isValid : bit;
  descriptor : 32 bits;
  tableType : TableType;
  pageType : PageType;
  domain : 4 bits
end
```

Listing 4.4: TLB Entry

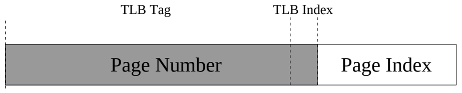


Figure 4.10: Address part of TLB index

tag comparison is necessary to determine whether an appropriate translation is cached.

Due to the fixed tag size (refer to previous section), each TLB entry caches the translation information for one kilobyte of virtual address space, indepenent of the page sized used. The index into the array of TLB entries is formed from parts of the virtual address (figure 4.10). It is taken from the least significant bits of the TLB entry tag portion of the virtual address (see previous section). These bits were chosen as they are likely to change often and their distribution is nearly random. This is important as it maximises the usage of the available TLB entries. For example, if the most significant address bits were chosen instead, most virtual addresses would refer to the same TLB entry, leaving most other entries unused. This would result in the one entry being replaced very frequently, often evicting useful translations. This heavily degrades performance (also known as *thrashing*).

Code and data usually reside in different parts of virtual memory, separated by many kilobytes. To avoid competition for the same TLB entries, the TLB is divided into seperate portions for data and instruction accesses. An access

```
constant TLB_ADDR_UPPER_BIT = 10 + (log (TLB_ENTRIES/2)) − 1
constant TAG_SIZE = 32 − (TLB_ADDR_UPPER_BIT + 1)

tlbIndex:=(#virtBundle.Address[10..TLB_ADDR_UPPER_BIT]
           @ #isData as TLB_INDEX_SIZE bits)

shared tlbLookup
is
begin
    if ( tlb [tlbIndex ]. tag = (#virtBundle.Address[(TLB_ADDR_UPPER_BIT+1)..31] as TAG_SIZE bits)
        and tlb[tlbIndex]. isValid ) then

        tlbHit:= 1
      || descriptorOut <− tlb[tlbIndex].descriptor
      || domainOut <− tlb[tlbIndex].domain
      || pageTypeOut <− tlb[tlbIndex].pageType
      || abortOut <− noAbort
    else
        tlbHit:= 0
    end
end
```

Listing 4.5: TLB Lookup

type bit is added to the TLB index to effectively split the available entries evenly between translations for instructions and translations for data accesses. This measure prevents instruction access TLB lookups from evicting data access TLB entries and vice versa.

### Entry Lookup Implementation

The number of TLB entries (`TLB_ENTRIES`) is a (compile time) user definable constant and hence the size of the TLB index (`TLB_INDEX_SIZE`) and the size of the entry tag (`TAG_SIZE`) need to be defined relative to it (listing 4.5). On each translation, the index of the TLB entry is formed from the least significant bits of the virtual page number and an additional bit indicating the type of access (data access/instruction access). The resulting index is stored in a variable, `tlbIndex`, which is used in the entry lookup and entry replacement procedures.

The `tlbLookup` procedure is also shown in listing 4.5. The tag of the TLB entry is compared against the relevant bits of the virtual address of the pending access and the boolean result of the comparison is stored in the `tlbHit` variable.

```
shared getFld
is
begin
-- compute FLD address
descFetch <- {read, (b2Array{0,0} @ #virtBundle.Address[20..31]
                        @ #cpVars.tableBase as memAddress), word, 0,0,0}
|| select descriptor then
       descriptorVar := descriptor
    || tableTypeVar := (#descriptor[0..1] as TableType)
    || if (#descriptor [0..1] as TableType) = undef then -- translation Abort
           wasAborted:=1
        || abortOut <- FLD
        || noValidDescriptor()
       else
           wasAborted:=0
        || domainOut <- (#descriptor[5..8] as 4 bits)
        || tlb [tlbIndex]. tag := (#virtBundle.Address[11..31] as 21 bits)
        || tlb [tlbIndex]. domain:= (#descriptor[5..8] as 4 bits)
        || tlb [tlbIndex]. tableType:=(#descriptor[0..1] as TableType)
        || if (#descriptor [0..1] as TableType) = section then
               tlb [tlbIndex]. pageType:=undef
            || tlb [tlbIndex]. descriptor:=descriptor
            || tlb [tlbIndex]. isValid :=1
            || descriptorOut <- descriptor
            || pageTypeOut <- section
            || abortOut <- noAbort
           end
       end

   | extAbort then
       abortOut <- extFLD
    || noValidDescriptor()
    || wasAborted:=1
   end
end
```

Listing 4.6: getFLD Procedure

If a TLB hit occurred the cached translation is forwarded to the permission check unit[3].

If no translation for the pending memory access is stored in the TLB, the MMU performs a page table lookup. Table walking commences with a fetch of the first level descriptor (FLD) by the `getFLD` procedure (listing 4.6). Once the

[3] Unlike other architectures, the ARM architecture has several features which make it hard to store the result of a permission check in the TLB. Namely the processor mode and the Domain access permissions may change in between accesses, resulting in potential changes in effective access permissions, without requiring a TLB flush [1]

```
shared getSld
is
begin
  if tableTypeVar = coarse then
     sldAddr:= (b2Array{0,0} @ #virtBundle.Address[12..19]
                             @ #descriptorVar[10..31] as memAddress)
  else
     sldAddr:= (b2Array{0,0} @ #virtBundle.Address[10..19]
                             @ #descriptorVar[12..31] as memAddress)
  end;
  descFetch <− {read,sldAddr , word,0,0,0}
  ||
  select descriptor then
     descriptorOut <− descriptor
   || pageTypeOut <− (#descriptor[0..1] as PageType)
   || if ((#descriptor [0..1]  as PageType) = undef) then
         wasAborted :=1
       || abortOut <− SLD
     else
         wasAborted:=0
         || abortOut <− noAbort
           −− All other TLB fields have already been filled by getFld !
         || tlb [tlbIndex]. descriptor:=descriptor
         || tlb [tlbIndex]. pageType:=(#descriptor[0..1] as PageType)      || tlb [tlbIndex]. isValid:=1
     end

  |  extAbort then
         wasAborted:=1
       || noValidDescriptor()
       || abortOut <− extSLD
  end
end
```

Listing 4.7: getSLD Procedure

FLD is read, the domain, table type and tag fields of the TLB entry are updated. If the FLD describes a Section, no further descriptor fetch is necessary and the remaining TLB fields (pageType, descriptor) are filled in and the entry is marked valid.

The `getSld` procedure (listing 4.7) is very similar to the `getFld` procedure. It fetches the second level descriptor (SLD) from the page tables, based on the second level table address and page table type taken from the FLD. When the SLD has been fetched from memory, the TLB fields not already filled in by `getFld` are updated.

### 4.5.3 Entry Invalidation

Invalidation of TLB entries is requested via writes to a system coprocessor register. Due to the asynchronous nature of the system, such writes are not synchronised with operation of the MMU. As entry invalidation changes the state of the MMU it is similar to a change in the page tables and suffers all the associated problems, as outlined in section 3.3.2. However, TLB entry invalidation requests differs in two ways from page table modifications:

1. Requesting TLB Entry invalidation is an *event which modifies MMU state*, whilst the other MMU related registers in the system coprocessor contain MMU state directly. This is an important distinction. Information like the page table base address can be sampled at any time and will always contain the current page table address. A write to the TLB entry invalidation register on the other hand requires the MMU to invalidate the corresponding entries once[4].

2. Invalidation requests may occur when the MMU is not activated. All other MMU state can be modified when the MMU is not activated without requiring any action on behalf of the MMU. TLB invalidation requests however need to be honoured even if the MMU is inactive (at least prior to the next translated access).

The obvious approach is to employ a similar system as for the MMU coprocessor communication: All access to the TLB could be arbitrated, to allow the system coprocessor and the MMU access. This way writes to the system coprocessor could not complete before the requested TLB entries were invalidated. The problems listed above would be solved, as the TLB entries are invalidated directly on every request.

However, this obvious solution has a serious drawback: It requires arbitration for the TLB on every memory access. As outlined in section 1.2.4, arbitration has some inherent disadvantages and most of all would slow down each TLB access. Instead another solution was sought and implemented, which does not carry the penalty of arbitration for the TLB on every access.

[4]multiple invalidation would not lead to incorrect results, it only reduces MMU performance.

```
shared flushTLB
is
begin
    for || i in 0..( TLB_ENTRIES−1) then
        if (#cpVars.tlbFlushFlags[#i[TLB_INDEX_MSB]]) = 1 then
            tlb [ i ]. isValid :=0
        end
    end
  ||
    −− Signal back to cp15 that TLB entries were flushed
    [
      sync cpRegLock;
      flushedFlags <− cpVars.tlbFlushFlags;
      sync cpRegRelease
    ]
end
```

Listing 4.8: TLB Entry Invalidation: TLB side

The implemented solution trades off performance in the common case, when no TLB entries are invalidated, against performance in the rare case, when entries need to be flushed. A write to the TLB invalidation register in the system coprocessor sets flags, indicating whether Data and/or Instruction TLB entries need to be invalidated[5]. These flags are part of the registers read by the MMU on every memory access from the system coprocessor (refer to section 4.3.3). On every *translated* access, the TLB checks the entry invalidation flags and invalidates the entries as requested. When invalidating entries, the TLB acquires the system coprocessor lock and resets the TLB invalidation flags.

This solution addresses the issues raised at the start of this section. In effect the flag setting and resetting communication changes the nature of the invalidation requests from an event to static MMU state, similar to all the other state registers in the system coprocessor. The scheme also results, to a limited extent, in *lazy* TLB entry invalidation: Entries are only invalidated when the MMU is active and multiple outstanding invalidation requests may be combined into a single operation (e.g. this occurs when multiple flush requests are sent when the MMU is inactive).

[5] The current scheme does not support invalidation of individual entries. However he scheme could easily be extended to include the necessary virtual address tag.

```
procedure registerBank(
    [...]
 )
 is
 local  [...]
 begin
     [...]
    loop
        arbitrate
           regAddr then
            if (regAddr.rw = read) then
                [...]
            else
                select regWrite then
                    case regAddr.register of
                        [...]
                       | 8 then --TLB flush register
                               -- Take care not to clear current flags.
                               c8:= c8 or (b2Array{#regWrite[1],#regWrite[0]} as 2 bits)
                        [...]
                end
            end
        | cpRegLock then -- Arbitration between regAddr and cpRegLock !
            locked:=1
        end;

        while locked then
            -- MMU has locked System Coprocessor
            select cpRegRelease then
                locked:=0
               [...]
            | tlbFlags then -- TLB was flushed notification
               -- Clear flush requests which were serviced
               c8:= c8 and (not tlbFlags)
            end
        end
    end
 end
```
Listing 4.9: TLB Entry Invalidation: Coprocessor side

The scheme does introduce additional arbitration when resetting the flags. However, this only happens in the rare case of entry invalidation, which carries a severe performance overhead in any case, as it results in TLB misses on subsequent memory accesses. The read of the TLB invalidation flags on every memory access is part of the arbitrated read of all other MMU registers which is performed on every memory access anyway and hence does not introduce any *additional* arbitration.

The resulting implementation on the MMU side is shown in listing 4.8 and the corresponding part of the system coprocessor implementation is given in listing 4.9. Because the number of TLB entries is a parameter, the entry invalidation logic is constructed by the '`for ||`' construct. This 'loop' is 'unrolled' at compile time and results in multiple instantiations of the logic described in the loop body, all operating concurrently (refer to the Balsa User Guide [11] for more detail on the '`for ||`' construct). The implementation uses the fact that the TLB entries are evenly split between data and instruction translations (see previous section), where the most significant entry index bit indicates the type of entry. This bit is used as an index into the TLB entry invalidation flags to decide whether the entry needs to be invalidated.

On the coprocessor side the flags are set by a write to the TLB entry invalidation register (c8). Care is taken as not to overwrite the current state of the flags, hence writes to it are always 'ored' with the existing flags. When the TLB has invalidated entries, it writes back the flags corresponding to the operation it just performed. The coprocessor then resets the flags accordingly. The flag field is also made available to software inspection and can be accessed by reading the entry invalidation register from the system coprocessor (listing 4.9).

This concludes the description of the TLB, as implemented in the performance architecture (The implementation in the Baseline Architecture differs slightly, however the high level design is identical). The code for the complete translation block, consisting of the TLB and the table walking hardware, may be found in Appendix B.

# Chapter 5

# Evaluation

The evaluation of the memory management subsystem can be divided into two aspects: verification of correct operation and performance estimation. This chapter describes the evaluation setup and the test suite used to verify correct operation of the MMUs and the System Coprocessor. The next chapter analyses the performance of the MMUs, using the data obtained with the evaluation setup.

Establishing that an MMU is operating according to some specification is a non-trivial task due to the number of variables the MMU operation depends on. The behaviour of the ARM MMU is dependent on:

1. *State of control registers in the system coprocessor.* The control registers control activation of the MMU and hold information necessary for permission checking and translation. The MMU also reports error causes and abort addresses through registers in the coprocessor (section 4.1.3).

2. *Virtual address trace.* MMU operation does not only depend on the memory address of the access being processed, but also on previous accesses. Past access addresses influence TLB state and contents and in the pipelined performance architecture they influence the allowed access colour (section 4.3.4).

3. *Page tables in memory.* The tables containing the translation information can be located at an arbitrary location in memory. Therefore the entire memory contents can potentially influence translation behaviour.

4. *Current TLB state.* MMU operation depends on the contents of the TLB

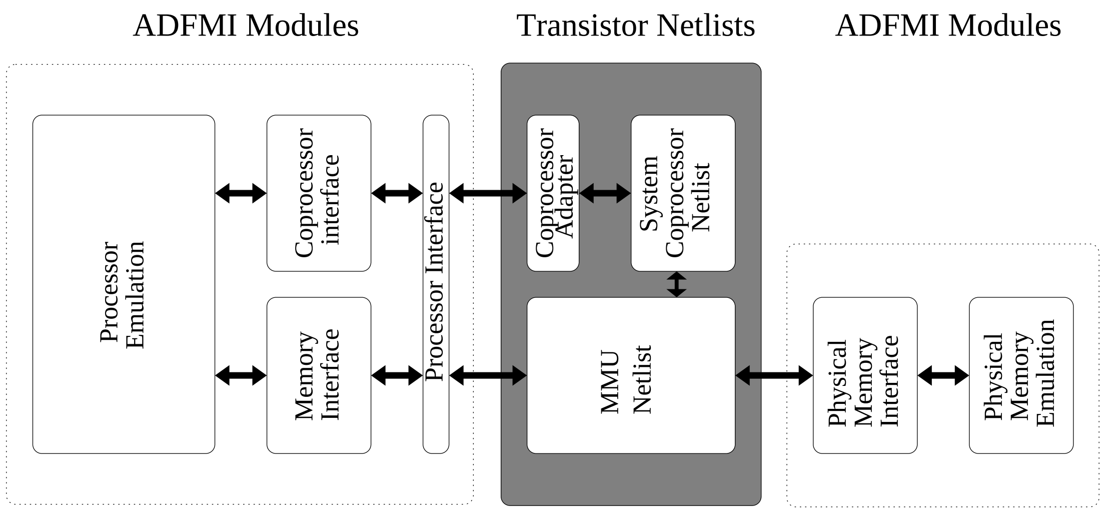


Figure 5.1: System evaluation configuration

entries as table walking is conditional on TLB misses. The state of the TLB is the result of the access address trace and system coprocessor state (section 4.1.4 ).

5. *Memory access mode and access size.* The current processor mode affects the permission check whilst the access size is relevant for alignment checking (section 4.1.2).

It is obvious that exhaustive testing is infeasible. Instead an MMU test suite was developed and implemented, which uses a series of tests to confirm that the MMU behaves as specified in the ARM Architecture Manual [1]. The tests, along with the mmuLib test kernel, are described in section 5.3 and subsequent. The test setup, necessary to execute the test suite, is detailed in section 5.1.1.

## 5.1 ADFMI Cosimulation

### 5.1.1 The Test Setup

A cosimulation approach was chosen to allow evaluation of the MMUs as part of a complete computer system. Cosimulation allows simulation of mixed level systems where some components are described by transistor level netlists whilst others are emulated by software. Cosimulation was chosen, because the processor and the memory were not available as netlists. They were emulated in software and interfaced to the transistor level netlists of the memory management components (MMU and the system coprocessor). The deployed system-level test setup is depicted in figure 5.1 and a description of all blocks can be found in section 5.1.4.

The Timemill simulator, which was used for all simulations, allows cosimulation via the ADFMI (Advanced Digital Functional Modelling Interface) (section 5.1.3). Even though the ADFMI incorporates delay and drive strength simulation, because no actual processor and memory implementation were available to base these values on, all *system level* measurements can only be used for relative performance comparisons between the architectures, not as predictions of actual system performance. An attempt at deriving system performance predicitions is undertaken in the next chapter.

### 5.1.2 Integrating Balsa with Timemill

The simulated transistor netlists were generated with balsa-netlist. They can readily be integrated into Timemill simulations, requiring only some netlist format conversion. The physical interface of the synthesized circuits consists of:

1. *All ports of the top-level Balsa procedure.* All Balsa procedure ports are divided into two signals for the handshake signals (<signalName>_a and <signalName>_r) and an array of data signals (<signalName>_d). The width of the array depends on the type of the port (i.e. the number of data bits required).

2. *An activation port.* Every Balsa procedure has an implicit sync port in its interface, used to activate the procedure. The activation request only gets acknowledged when the procedure terminates and, as the top-level design loops forever, no handshake will ever be completed on this port.

3. *Supply lines (*vdd *and* gnd*).* Naturally the circuit needs to be connected to power. However some problems arise due to the tools used to perform the necessary format conversion. Spice, which is used to convert the netlist, generates *all* signals as bi-directional and performs some simple optimizations. This often results in constant level *outputs* to be considered power

connections. Therefore, the supply lines may not add explicit ports to the interface, but rather use a existing constant level data ports[1].

The resulting converted netlists are instantiated in a top-level netlist, which describes the connections between all simulation blocks (transistor netlists and ADFMI models).

### 5.1.3 The Analog Digital Functional Modeling Interface

ADFMI allows the integration of circuit emulation blocks into a netlist level simulation. Each ADFMI block consists of a set of routines written in C, which describe the externally visible *ports* of the block, as well as its behaviour. The C source files of all blocks are compiled and combined into a library by tools from the Timemill environment. All blocks defined in that library can be instantiated by the user as part of a simulation netlist.

Whenever the data on a net which is connected to one of the block's inputs changes, the block's evaluation function is executed. The entire simulation executes as a single thread and hence the ADFMI model needs to give control back to Timemill to resume simulation. Only when the actual simulator is executing may simulation time advance. This needs to be considered when developing ADFMI models, as it means that a model cannot wait (in the sense of multiple threads) for a specific input to change. This became slightly awkward for the processor emulation, see section 5.1.4 for details.

### 5.1.4 ADFMI Block Descriptions

The MMU evaluation setup as depicted in figure 5.1 contains several ADFMI blocks to emulate all the parts of the system for which no netlists were generated. The processor and memory emulations were divided each into multiple parts to separate the functional emulation of the components from the necessary ADFMI interfacing routines. This separation was chosen to facilitate easy porting

[1]Some further manual modification is sometimes required. Blocks which pass-through data, get implemented by naming the input and output data signals identically when the block is instantiated (the handshake signals are not affected). If this occurs on the top level design, the necessary renaming needs to be performed manually.

to a different cosimulation environment and aid modifications of the emulation routines in the future. The following is a list of the ADFMI blocks used in the simulation setup:

1. *Processor Core*. This is the ARM compatible processor core emulation. It does not have any ADFMI ports. All interaction with the simulation occurs through the memory interface and the coprocessor interface. Several difficulties occurred due to the single simulation thread, which makes it impossible for the processor to wait for data to appear after it initiated a memory access. Instead the processor emulation code has to return without having completed the memory access to allow control to be passed back to the simulator. When the data is available, the processor has to repeat the access to collect the data read or the return status (i.e. whether an abort occured).

2. *Memory Interface*. This connects the processor to the MMU and is the central control block for the processor simulation. It starts off the simulation and then calls the processor emulation every time a memory access or coprocessor access has completed.[2] As the processor can never have more than one pending memory access, managing the current memory access colour is trivial and is performed entirely within the memory interface (refer to section 4.3.4 for explanation of access colour).

3. *Coprocessor Interface*. It connects the processor to the system coprocessor via the *Coprocessor Adaptor*. The adaptor is a piece of synthesised Balsa which is used to simplify handshaking between the processor emulation and the ystem coprocessor. The coprocessor interface has a large number of ports connected to the processor. As handshaking is relatively complex in ADFMI modules, all necessary information for the access is sent in one big data structure to the adaptor which then performs the separate handshakes

[2] To perform a memory read, the processor calls a function of the memory interface to start the read access and then returns control back to Timemill. The memory interface completes the necessary handshakes to perform the access and then calls the processor execute function again. The processor in turn calls the memory interface to receive either the requested data or an abort notification. Memory writes use a similar approach as success after every write needs to be confirmed. This is of course quite a wasteful approach as the execute function is called at least twice on every instruction fetch. However, the solution only required relatively small modifications to the existing ARM emulator.

with the coprocessor. Upon completion of the access it returns a reply to the processor containing the decode response and the value returned by the coprocessor on a read. The current coprocessor interface and adaptor blocks are limited to the specific requirements of accesses to the system coprocessor and therefore does not implement the coprocessor memory interface as it is not required by the system coprocessor.

4. *Processor Interface.* This is a wrapper for all processor emulation parts. It is necessary to satisfy ADFMI constraints.

5. *Physical Memory Interface.* This interfaces the MMU to the memory emulation. It calls the relevant functions in the memory emulation to perform the actual memory access and also initialises the memory at the start of the simulation. The complete implementation is given in Appendix A as an example for ADFMI models.

6. *Physical Memory Emulation.* This implements the physical memory. It can load an initial memory image from file and generates external aborts for accesses to memory addresses outside the simulated address range.

## 5.2 Handshaking in ADFMI

Neither Timemill nor the ADFMI make any special provisions for the simulation of asynchronous circuits. Interfacing to asynchronous circuits gets complicated because data needs to be sent or received via handshakes (refer to section 1.2.2).

Handshakes can be modelled as a state machine which can trivially be implemented in a programming language, such as C. The situation gets complicated by the concurrency of handshakes. Handshakes on different ports often overlap at least partially. This easily leads to a very large statespace, even for only a few concurrent handshakes, due to the number of possible transition permutations.

Rather than implementing the component handshake behaviour as a monolithic state-machine, the return-to-zero phase was modeled separately. This lead to a compact and easily understandable main state machine which modelled the component behaviour (albeit one which does not model all possible valid state transitions) and an additional small state machine to control the return-to-zero phase on each port. Refer to Appendix A for an example of an ADFMI component which uses the described approach to handshake modelling.

## 5.3 Test suite

An MMU test suite was developed to allow the verification of correct MMU operation. The suite was specifically designed for the test setup described in the previous section, but the system should run on any ARM platform with an ARM compatible MMU.

The test suite consists of a system kernel called mmuLib (section 5.3.1) and a variety of tests (section 5.3) which use the facilities provided by mmuLib to verify specific aspects of the MMU. To simplify the management of virtual memory, the suite uses the concept of *MMU contexts*.

### 5.3.1 The mmuLib System

Managing virtual memory is not easy. A slight error in the multilevel page tables can lead to wrong pieces of code (or even data) to be executed. These are often very hard to debug. The alignment of tables and table entries and the sometimes necessary repetition of descriptors within a page table differ between the various combinations of page sizes and table granularities. This makes ARM virtual memory management potentially prone to programmer errors. In a test suite designed to verify an MMU implementation, special care needs to be taken so that virtual memory management is done in a safe and robust manner. Errors in the implementations should, whenever possible, lead to meaningful debugging outputs and the test should keep executing rather than crashing silently.

The mmuLib system addresses these requirements in three ways. Firstly, all tests are executed in user mode and all interaction between the page tables and the MMU is performed via system calls to mmuLib kernel routines. Secondly, only a dedicated part of memory not holding any code can be remapped by the test routines. The parts of memory containing the system and test code are guaranteed to remain in a flat mapped part of memory, so that virtual and physical addresses are always identical. Finally, a higher-level view of memory mappings is provided to the tests through so called *MMU contexts*.

An MMU context describes a mapping for a specific part of memory, as well as a page size and a page table granularity. It is left to the system kernel to generate the appropriately aligned page tables. The system also adds the necessary entries to the page tables to maintain the *system mapping*, i.e. the flat mapping of the parts of memory containing the system and test code as well as the page tables.

### 5.3.2 The mmuLib Kernel

The mmuLib kernel is the common core of all tests. The services it provides can be divided into three areas:

1. *System initialisation*. The kernel is designed to run on the bare chip, as it needs complete control over the system coprocessor, abort handlers and memory. Necessary initialisation tasks include interrupt vector setup and provision of initial handlers as well as stack and heap allocation.

2. *Context management functions*. Context creation, activation and verification is done via a set of kernel functions. These kernel functions all aid the test designer by checking contexts for validity (there are constraints on context size, alignment and location) and maintaining the system mapping.

3. *System calls*. To guarantee that all manipulation of MMU setup and page tables can only occur via the dedicated kernel functions, all tests are executed in user space and need to use system calls (SWIs) to change MMU or page table state. The SWI functions and the necessary SWI handler are provided by the kernel.

As mentioned in the preceding section, only a dedicated part of memory is allowed to be remapped by the tests. The memory is conceptually divided into several areas (figure 5.2). The addresses of the page table area and the translation test area are not fixed, as they are dependent on the size of the test. The actual addresses are 'discovered' at system initialisation time and then aligned appropriately.

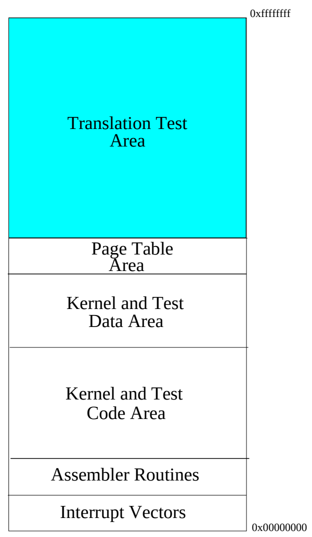


Figure 5.2: Memory map used by the mmuLib system

### 5.3.3 MMU Tests

A series of tests to verify the correct operation of the implemented MMUs was developed using the mmuLib kernel. All tests are incremental, i.e. later tests rely on all earlier tests to have completed successfully. Test results and status messages are displayed via a *memory tube*[3]. All tests are designed to detect whether they are working as predicted and supply debugging data in case of failure. Wherever sensible, tests always perform each operation on all possible page size and granularity combinations. The test are based on the MMU specified in the ARM Architecture Reference Manual [1] and are designed to test adherence to the specifications. Due to the lack of time no additional tests aimed at discovering faults (e.g. by creating memory access scenarios aimed at 'tricking' the MMU) were implemented.

The test suite is comprised of the following tests:

1. *SWI Test*. This test verifies the correct system coprocessor behaviour of the system. The system calls provided by the kernel are used to write and read back values from the MMU control registers in the system coprocessor. The MMU remains disabled throughout the test. This verifies processor $\leftrightarrow$ coprocessor communication and the coprocessor implementation, it does

[3] A memory tube is a specific memory address which outputs any data written to that address as an ASCII symbol on screen (or in a file).

not test whether the control registers have the desired effect on the MMU. This is done indirectly in the subsequent tests.

2. *Flat Mapping Test.* The first test which enables the MMU and the first of a series of address translation tests. As a working flat mapping is required for the part of memory containing the kernel and test code, it needs to be verified before any other mapping. However there is no way from within the test to verify that the MMU is doing any address translation[4]. A system without an MMU would actually pass this test!

3. *Inverse Mapping Test.* The mapping used maps the lowest virtual test page number to the highest physical test page number. This is the first test which can detect whether the MMU is translating correctly.

4. *Fixed Mapping Test.* This test maps *all* virtual test pages to a single physical page. It was developed to add more variations on the mappings used, to increase confidence into the mapping functionality of the MMU. It is the last of the translation tests.

5. *TLB Test.* TLB flushing is already used on every context switch. However no previous tests verify that any entries actually are cached at all. This test serves as an indirect proof of TLB caching by changing contexts without flushing the TLB and demonstrating that the old, cached, mapping remains in operation. *This test was not implemented due to time constraints and as it is dependent on specific TLB implementations (size, associativity, lock-down support, etc.). Correct TLB caching was verified with the use of memory traces, showing when descriptor fetches occur.*

6. *Abort Test.* This test verifies the correct abort behaviour and abort case detection, i.e. does the MMU only abort when necessary? And if it aborts, does it report the correct fault status in the system coprocessor? The mapping tests only ensure that the alignment, permission and domain checks do not incorrectly report errors, however those tests would pass even if those units were entirely absent. Hence, their correct behaviour in an abort case is needed to demonstrate the presence and correct operation of the alignment, domain and permission check units. This test, if successful, also

[4] externally it is possible to verify MMU operation by verifying that the necessary descriptor fetches occur.

verifies that the MMU control registers in the system coprocessor are read uncorrupted and interpreted correctly by the MMU, as well as the ability of the MMU to write to the system coprocessor.

The test suite was developed and verified on an existing software ARM emulator to which an MMU emulation was added. As the tests need to execute many millions of instructions, they were of only limited use in the early testing stages of the MMU implementations, because the netlist level simulation takes several days to perform even a single test. Hence a collection of short assembler programs was devised to test MMU operation, where each program causes a specific abort case to occur. These tests only verify a very specific abort case each and hence can not be considered as a verification of the MMU designs. However, they proved an excellent starting point for debugging the designs, as they only require the execution of tens of instructions each and allow easy trace analysis of the netlist simulations.

## 5.4 Discussion

### 5.4.1 Test Bench Alternatives

Originally the MMU implementations were intended to be evaluated as part of an entire asynchronous computer system. They were designed to be compatible with the SPA processor [24], a synthesised, asynchronous, ARM compatible processor developed at the University of Manchester. However at a relatively early stage of the project it became apparent that the processor would not be available early enough to allow it to be used for MMU verification. Hence an alternative testing method was required.

The use of a testbench which supplies predefined test vectors to the designs was considered. However, such a testbench is non-trivial. Test vector generation for an MMU verification is difficult, due to the number of signals, but during the initial phases of the project an MMU emulation was added to an existing ARM simulator, which could have been used to generate the necessary data. However, the problem of presenting the test vectors to the design remained an unsolved issue. A LARD (section 2.4.3) test bench was considered as it can easily supply the test data to the LARD level simulation of the MMUs. But due to the drawbacks described in section 2.4.4, the simulation could not have supplied any performance figures and was thus deemed unsuitable.

A cosimulation setup, similar to the one actually used, but only supplying predefined test vectors to the MMUs, was briefly considered. However, such a test bench would not have been of substantially lower complexity than generating the test vectors using the existing processor emulation when running a program. Executing a test program on the test bench has the distinct advantage that the running program can report process and supply debugging data in case of failure which greatly aids the debugging of such a complex system. Hence the integration of the processor emulation into the testbench was chosen as the method of generating the test inputs for the MMUs.

The implemented solution works well. As it implements the SPA compatible coprocessor and memory interface it can also be used to test other components implementing those interfaces. The main drawback of the system is the poor simulation speed on the available computer systems. The developed MMU test suite showed its value when it was used to increase confidence in a reference software MMU emulation – it uncovered many bugs. It offers a nice clean interface to add more tests to increase test coverage. This comes at a cost: each test requires the execution of millions of instructions. The test suite was developed on the software ARM emulator where it completes within two hours. The cosimulation setup on the other hand requires more than a day for *each* of the individual tests. No profiling was undertaken to decide whether the ADFMI blocks or the simulation of the netlists was responsible for the poor performance.

The next chapter uses the simulation results gathered with the evaluation setup and compares the performance of the Baseline Architecture and the Performance Architecture MMUs.

# Chapter 6

# Performance Analysis

This chapter analyses the predicted performance of the synthesized memory management units. The method used to arrive at the performance estimates is outlined in section 6.1. An initial estimate of system performance (section 6.2) is followed by a more detailed look at the overhead distribution responsible for the slow system access performance. Section 6.3 looks at the overheads, when the MMUs are not translating addresses. Further sections cover translation performance (section 6.4) and look at the performance impact of the different coprocessor access styles used (section 6.5). The chapter finishes with the identification of the crucial performance limiting factors (section 6.6) and proposes two performance optimisations for the current architectures (section 6.7).

## 6.1 Method

All measurements used in this chapter were performed on the transistor level co-simulation of the MMU architectures and the System Coprocessor described in the preceeding chapter. The simulated designs were placed and routed using the Compass design suite (refer to the appendix for some diagrams), however the resulting circuits did not function correctly. Further analysis showed that they require substantial manual modifications to achieve timing closure due to the non-delay insensitive characteristics of the chosen Balsa back-end (refer to section 2.4.1). Currently no tools exist to support this process and as it is very laborious, it was deemed to be outside the scope of this work. Hence, the simulations did not take capacitance into account. Instead all results given in this chapter were adjusted to take into account predicted capacitance effects, based on

| Access Scenario | Baseline Architecture | Perf. Architecture |
|---|---|---|
| Load: Untranslated | 12ns | 52ns |
| Load: Translated TLB Hit | 150ns | 96ns |
| Load: Translated TLB Miss Segments | 172ns | 118ns |
| Load: Translated TLB Miss Pages | 190ns | 122ns |
| Store: Untranslated | 20ns | 66ns |
| Store: Translated TLB Hit | 172ns | 106ns |
| Store: Translated TLB Miss Segments | 172ns | 126ns |
| Store: Translated TLB Miss Pages | 190ns | 136ns |

Table 6.1: Comparison of projected memory access overhead for the different MMU architectures (i.e. excluding actual memory access).

| Access Type | Cache | Main Memory |
|---|---|---|
| Read: Average Access Request to Data Handshake Completed | 10ns | 40ns |
| Write: Average Access Request to Abort and Data Handshake Completed | 12ns | 50ns |

Table 6.2: System memory access times

heuristics gained through past experience with the same cell library, technology and asynchronous design style for other asynchronous circuits.

## 6.2 Predicted System Peformance

Table 6.1 shows the projected overheads introduced by the two MMU designs for a variety of different memory access types. An estimate of average memory access latency was derived by making certain assumptions about the remaining system components and their usage. Assumed memory access times for the performance estimate are given in table 6.2[1].

The projections assume only a single outstanding memory access at all times (i.e. memory latency not memory throughput are compared. Maximum throughput of the Performance Architecture is analysed in section 6.6.2). All other assumptions are summarised in table 6.3. Cache miss rate as well as the load/store frequency and the typical TLB miss rates are based on [21]. The miss rate for the performance architecture TLB is projected to be three orders of magnitude worse than a typical TLB miss rate due to the limiting factors of the current TLB implementation:

[1]Times given are considered average access times, incorporating faster sequential accesses, etc. Times are estimates for the entire handshake, from beginning to end; data is assumed to be available about 65% through the cycle.

| System Parameter | |
|---|---|
| Cache Miss Rate | 0.02 |
| Projected TLB Miss Rate (Base. Arch. TLB) | 0.1 |
| Projected TLB Miss Rate (Perf. Arch. TLB) | 0.1 |
| Typical TLB Miss Rate | 0.0001 |
| Integer Load Frequency | 0.26 |
| Integer Store Frequency | 0.09 |

Table 6.3: Assumption about the system in which the MMUs are used. It was the basis for the performance projections in table 6.4.

| | **Untranslated Instruction Fetch** | **Translated Instruction Fetch** | **Untranslated Data Access** | **Translated Data Access** |
|---|---|---|---|---|
| **Base. Arch.** | 13ns | 155ns | 15ns | 160ns |
| **Perf. Arch. Implemented TLB** | 53ns | 100ns | 57ns | 102ns |
| **Perf. Arch. Ideal TLB** | 53ns | 97ns | 56ns | 99ns |

Table 6.4: MMU system performance projection

- *Small number of entries.* The current design was only synthesised for a 8-entry TLB (4 each for Data and Instruction Accesses), which would result in a high number of *capacity misses*. However, as the number of TLB entries is a compile-time constant, this can be easily increased to suit requirements.

- *Each entry only caches the translation for 1KB areas of memory.* This problem was already outlined in section 4.5.1.

- *Direct Mapped entry organisation.* Normally a higher level of associativity is chosen to reduce the number of *conflict misses* caused by multiple descriptors competing for the same TLB entry.

The figures assume that the cache access occurs in parallel with the MMU lookup, i.e. a virtual addressed cache. However, even in the case of a cache hit, the memory access has to stall until the permission check has completed, before the cached data can be used by the processor. Physical memory can only be accessed once the physical address has been computed, hence it cannot be overlapped with the address translation. Instruction fetches are listed seperately as they are slightly faster because fetches are always memory reads, never memory writes. On untranslated accesses each memory access is still assumed to synchronise with the MMU, even though no work is performed. For comparison, table 6.4 also includes predicted performance of the Performance Architecture with an idealised TLB, which never has a TLB miss.

| | Baseline Arch. | Performance Arch. |
|---|---|---|
| Cp Access (Lock and Var. Reads) | 7ns | 30ns |
| Pipeline Register Delay | 0ns | 12ns |
| MemMux / MemArbiter | 4ns | 6ns |
| Other | 1ns | 4ns |
| **Total** | **12ns** | **52ns** |

Table 6.5: MMU overheads distribution for untranslated memory accesses (figures refer to read access).

## 6.3 Untranslated Access Overheads

### 6.3.1 Baseline Architecture

Table 6.5 shows how the overheads are distributed on untranslated memory accesses. The coprocessor access is necessary to determine whether the MMU is activated. However, only one coprocessor register is read whilst all other stages of the MMU are bypassed, as they are not required, except for the memory multiplexer which adds a small delay.

Bypassing the unused components results in substantially faster untranslated accesses than in the Performance Architecture, where all stages are activated in turn, passing the data on. The coprocessor access could be sped up by adding buffers, however, worst-case overheads are unpredictable, due to the unbounded arbitration delay necessary to gain the coprocessor lock.

### 6.3.2 Performance Architecture

The Performance Architecture is substantially slower in performing untranslated accesses than the Baseline Architecture (table 6.5). The major slowdown is caused by the coprocessor access. Unlike the baseline architecture, which only accesses one register, the Performance Architecture always reads all coprocessor registers as one large 'packet'. In the current implementation, the data is actually read *twice* into the first stage pipeline register, causing the very large observed overhead (refer to section 6.5 for the reasons). The pipeline register delays are the result of the design decision to have untranslated memory accesses pass through all pipeline stages, even though no work is performed at any stage, to maintain pipeline coherency upon MMU activation (as outlined in section 4.3.2).

| | Base. Arch. Load | Perf. Arch. Load | Base. Arch. Store | Perf. Arch. Store |
|---|---|---|---|---|
| **TLB Hit** | 50ns | 32ns | 50ns | 32ns |
| **Return-To-Zero Ripple** | 36ns | 0ns | 46ns | 0ns |
| **Cp Access** | 16ns | 30ns | 16ns | 30ns |
| **Alignment Check** | 20ns | 0ns | 20ns | 0ns |
| **Permission + Domain Check** | 20ns | 14ns | 20ns | 14ns |
| **Pipeline Registers** | 0ns | 12ns | 0ns | 12ns |
| **Others** | 8ns | 8ns | 20ns | 18ns |
| **Total** | **150ns** | **96ns** | **172ns** | **106ns** |

Table 6.6: MMU overheads distribution for translated memory accesses.

Even though the Performance Architecture is very slow on untranslated accesses, this should not affect overall system performance. Assuming that a CPU with an MMU is only employed when an MMU is required, the system will only run with deactivated MMU for a very brief period during system startup, which makes untranslated performance of little importance for overall system performance.

## 6.4 Translated Access Overheads

By far the most important case for MMU performance is the common case of a translated access with a TLB hit. It is expected to occur for over 99% of all accesses [21]. Hence this is the case analysed in this section; TLB misses are not analysed further.

The individual causes for the MMU overhead for translated TLB hits are listed in table 6.6; figure 6.1 illustrates the data. The implementation specific overheads are broken down and analysed in the subsequent two subsections. A later section (section 6.6) also looks at the architectural factors which limit performance.

### 6.4.1 Baseline Architecture

The individual factors contributing to the MMU overhead on translated accesses in the Baseline Architecture are detailed in table 6.7. Comparison with the Performance Architecture shows that the Baseline TLB implementation is substantially slower. Even though the TLB architectures are identical, the implementations differ in style. The Baseline TLB Architecture does not contain

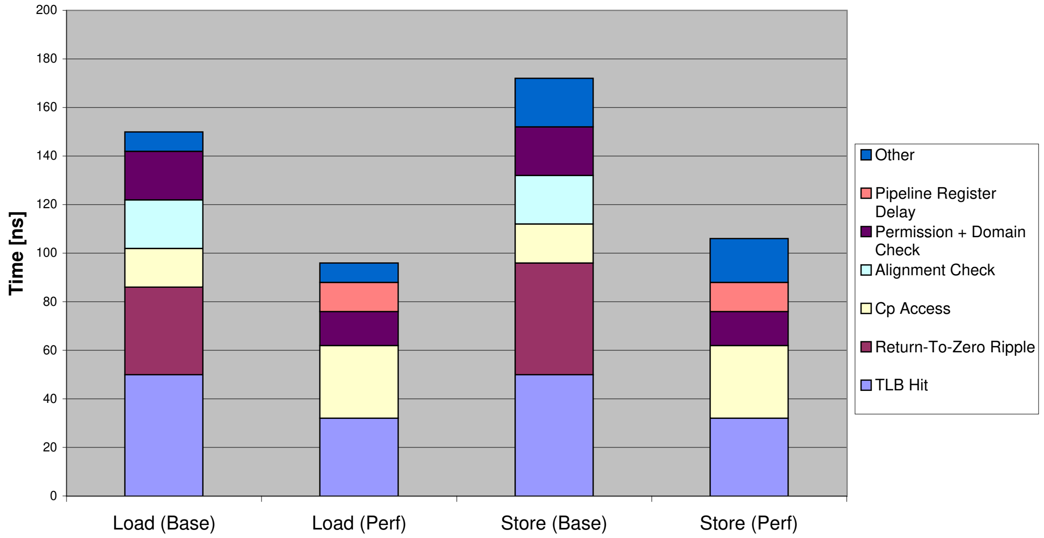


Figure 6.1: MMU overheads distribution for translated memory accesses.

any registers in the data path (other than the TLB entries). All communication between the TLB components occurs over internal channels, onto which data is steered as required. This results in multiple nested `select` statements. Each level of `select` needs to wait on its inputs and no handshake can complete before the inner most `select` clause has completed. Only then can all pending open handshakes start their return-to-zero phase. This leads to a 'ripple' of handshakes closing, one level after another, adding substantial delay overheads.

A similar effect occurs at the MMU system level. In the Baseline Architecture no registers are used to decouple functional components. Hence each component performs a `select` on its inputs and needs to keep the handshake with the preceding component open until the subsequent component has finished the handshake for its outputs. This 'return-to-zero ripple' adds substantial overhead (table 6.7). A solution that eliminates this ripple is proposed in section 6.7.1, later in this chapter.

The alignment check also adds substantial overhead. Two implementation

| | Baseline Arch. Load | Baseline Arch. Store |
|---|---|---|
| **TLB Hit** | 50ns | 50ns |
| **Return-To-Zero Ripple** | 36ns | 46ns |
| **Alignment Check** | 20ns | 20ns |
| **Cp Access** | 16ns | 16ns |
| **Permission Check** | 12ns | 12ns |
| **Abort Handler** | 0ns | 12ns |
| **Domain Check** | 8ns | 8ns |
| **Compute Physical Address** | 4ns | 4ns |
| **Memory MUX** | 4ns | 4ns |
| **Total** | **150ns** | **172ns** |

Table 6.7: Detailed MMU overheads distribution for translated memory accesses in the Baseline Architecture.

| | Performance Arch. Load | Performance Arch. Store |
|---|---|---|
| **TLB Hit** | 32ns | 32ns |
| **Cp Access** | 30ns | 30ns |
| **Permission + Domain Check** | 16ns | 16ns |
| **Abort Manager** | 0ns | 10ns |
| **Pipeline Registers** | 12ns | 12ns |
| **Memory Arbiter** | 6ns | 6ns |
| **Total** | **96ns** | **106ns** |

Table 6.8: Detailed MMU overheads distribution for translated memory accesses in the Performance Architecture.

details are the cause. Firstly, a Balsa function is used to determine whether an access is correctly aligned. The lengthy logical expression used in the function evaluates slowly, a switch statement implementation would be faster. Secondly, the Alignment Check component communicates with the Abort Handler on every translated access. The virtual address is transferred to the Abort Handler, where it is required in case of a potential subsequent data abort (it is necessary to update the exception address in the system coprocessor). To minimise the number of arbiters in the system external constraints must guarantee non-overlapping communication with the Abort Handler. Hence, the handshake from the Alignment Check needs to have completed before the next component can be activated, as it – in case of an abort – might need to communicate with the Abort Handler as well.

### 6.4.2 Performance Architecture

The MMU translation overheads, as detailed in table 6.8, are dominated by the TLB and coprocessor access times. The coprocessor access time is analysed in detail in the next section and is not covered here any further. The TLB access time is 36% faster than the Baseline TLB implementation. Therefore the use of

| | estimated delay |
|---|---|
| **CpInstruction → cpReply (request to request)** | 4ns |
| **Execute cycle Write (handshake completed)** | 20ns |
| **Execute cycle Read (handshake completed)** | 32ns |
| **locking the register bank (handshake completed)** | 4ns |
| **register read (request to acknowledge)** | 3ns |

Table 6.9: System Coprocessor performance estimates

'global' registers within components employed in the Performance Architecture should be preferred to a nested `select` implementation style used by the Baseline Architecture. This finding is supported by the integrated Permission and Domain Check implementation, which performs faster than the Baseline counterpart.

Each pipeline register causes a delay of around 4ns, leading to a total of 12ns overhead. There is a variance between the individual stages, due to different widths of the registers. The current implementation forces reads and writes to be sequential. A 'wagging' buffer [11] implementation, allowing overlapping of read and write access, was attempted, which would have reduced overheads in cases when the pipeline is filled with multiple accesses. However, the resulting synthesised circuits did not function correctly due to timing violations[2] and no further alternative implementations were persued.

## 6.5 Coprocessor Access Overheads

### 6.5.1 Coprocessor Data Transfers

The coprocessor is a shared resource, which is accessed from the processor as well as from the MMU (see chapter 3). This section examines the coprocessor access overheads for coprocessor access instructions which transfer values between the coprocessor register bank and the processor registers. The coprocessor access times though the coprocessor interface are shown in table 6.9. The times include the delay through the coprocessor wrapper, which steers the data between the processor and the coprocessors (see chapter 5).

[2]Manual resizing of transistors and additional buffering would have been needed, due to the non delay insensitivity of the bundled-data backend.

| | **Baseline Arch.** | **Performance Arch.** |
|---|---|---|
| Cp Lock | 4ns | 6ns |
| Single Register Read | 3ns | 10ns |
| Number of Reads | 4 | 2 (1) |
| Cp Release | 2ns | 4ns |
| **Total** | **16ns** | **30ns (18ns)** |

Table 6.10: MMU overheads distribution for coprocessor register access. Refer to section 6.5.3 for details about the figures in brackets.

Coprocessor reads are slower than coprocessor writes, because in a read cycle the data has to be read from the register bank before the transfer to the processor can commence. The data handshake needs to be completed, before the handshake with the register bank can continue. A write to the register bank overlaps the data handshake with the access to the register bank, resulting in the observed shorter cycle times.

The substantial difference in cycle times for different operations illustrates an advantage of an asynchronous implementation. A synchronous design would be forced to choose a clock speed which guarantees completion of a pipeline stage within one cycle. Assuming the execution times would refer to a synchronous implementation, it would force the design to use either a maximum clock speed of 31MHz or to split the execution stage into several pipeline stages to get more balanced execution times. Pipelining adds further overheads and using a slower clock leads to unnecessary delays (assuming the processor and coprocessor run at the same clock speed), as reads from the system coprocessor are expected to be quite rare (only occuring following aborts). The proposed asynchronous implementation is ready to perform the next cycle as soon as an operation has finished. With, for example, a ratio of three writes to one read of the system coprocessor, this would lead to the design performing equivalent to a synchronous design running at 42MHz.

### 6.5.2 Baseline Architecture

The Baseline Architecture strategy, reading coprocessor registers as needed, results in less MMU overhead for coprocessor accesses, compared to the Performance Architecture (table 6.10). The access times of the coprocessor registers depend on the width of the register read; times stated in the table refer to average access time. Releasing the coprocessor lock is faster than acquiring it because no arbitration is necessary (arbitration being necessary to acquire the lock, as the MMU competes with the processor for coprocessor access).

Even though the Baseline Architecture coprocessor access strategy results in smaller overheads than its Performance Architecture counterpart (refer to next section), it leaves the coprocessor locked during the entire memory access, hence it is a lot more likely to cause the processor to stall waiting for coprocessor access. The system performance impact of this behaviour is hard to estimate without more data on coprocessor access frequency.

### 6.5.3 Performance Architecture

Coprocessor accesses in the Performance Architecture are slower than Baseline Architecture accesses in all parts (table 6.10). The largest part of this difference however, is caused by the shortcomings of the bundled-data backend, used for synthesis. The Performance Architecture (unless an abort, or TLB flush occurs) reads all coprocessor registers in a single data transfer. The circuit initially synthesised did not manage to transfer the data correctly into the first stage pipeline register. The bundled data backend does not generate delay insensitive circuits (as outlined in section 2.4.1). In the synthesised circuit the handshake completed, before the read had succeeded. Manual insertion of buffers and possibly resizing of transistors should fix the problem and improved performance. However, due to time pressure, a simpler solution was implemented. Rather than just reading the data once, it is read *twice* into the first stage pipeline register, in succession. The generated circuit worked reliably, however it adds a 10ns overhead to the coprocessor access time. The numbers in brackets in table 6.10 are projected figures for a modified circuit which only performs a single coprocessor register read, resulting in a predicted 40% speedup.

Even the correct coprocessor register read estimates seem dis-proportionally high. The cause for the large overhead is the way the data is gathered in the coprocessor implementation (listing 3.1). When a read request is received, a handshake with all four coprocessor registers is initiated. The read data is then passed on to the MMU and the handshakes complete, once the MMU has acknowledged the data. These multiple concurrent handshakes cause the observed overhead.

Coprocessor locking is also slower in the Performance Architecture than in the Baseline Architecture, as an additional level of arbitration is required within the former's MMU. Pipeline Stage 1 reads all MMU related coprocessor registers on every memory access; Stage 2 needs to write to the system coprocessor when the TLB is flushed and Stage 3 updates the Fault Address and Fault Status registers in the system coprocessor for aborted data accesses. Hence an additional arbiter is necessary for the lock between the pipeline stages.

As in the Baseline Architecture, no arbitration is necessary upon lock release, neither inside the coprocessor, nor inside the MMU. However, a `select` block is necessary to steer the release signal.

Even though the locking strategy requires a slightly more complex implementation than in the Baseline Architecture, it leaves the coprocessor locked for shorter periods of time, i.e. it is only locked whilst it is accessed. This compares favourably with the Baseline Architecture which keeps the coprocessor locked for the duration of the access, as outlined in the previous section.

## 6.6 Performance Limiting Factors

The previous sections examined the causes for the MMU overheads of both MMU architectures. Faster implementation alternatives were proposed and shortcomings were pointed out. This section investigates the architectural features which limit performance of the designs and proposes improvements. Independently from such architectural modifications, performance of both implementations is also expected to improve significantly by manual modifications of the synthesised circuits, i.e. the additional buffering and resizing of transistors. These kinds of improvements were, however, considered to be outside the scope of this thesis.

### 6.6.1 Baseline Architecture

The major architectural shortcoming of the Baseline Architecture MMU, from a performance point of view, is its lack of concurrency. Sequential operation guarantees, as intended, that no unnecessary work is ever performed, but severely limits performance. It should also minimise power consumption (which is not

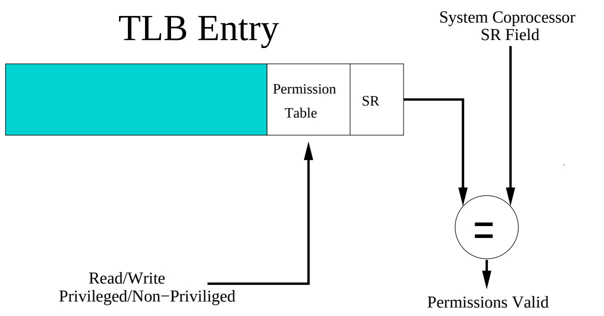


Figure 6.2: Improved TLB entry design, which also caches access permissions.

covered in this thesis), and resulted in a more compact design using around 20% less area than the Performance Architecture implementation.

Even without pipelining, concurrent activation of blocks would improve performance. Suggested concurrency improvements (all of which are realised in the Performance Architecture) are:

- *The Alignment Check should occur in parallel with the TLB lookup.* This effectively hides the cost of the alignment check, including the communication with the Abort Handler. Care needs to be taken in the Abort Handler to request abort status from both abort sources sequentially. This would save around 20ns per access.

- *Physical Address Computation, Domain and Access Permission checks should occur concurrently.* All three operations are dependent on the result of the TLB lookup, but not dependent on each other. An access time reduction of around 12ns is expected.

The other major performance limiting factor is the very long return-to-zero phase, as outlined in section 6.4.1. A solution is described in the final section of this chapter, which eradicates this overhead entirely, resulting in a reduction of access overheads by around 56%.

### 6.6.2 Performance Architecture

Identifying and improving the performance limiting factors in the Performance Architecture is more difficult. All obvious concurrency between components is already exploited, only the coprocessor access performance is obviously lacking, as explained previously. Nevertheless, several areas remain where modifications or extensions could lead to improvements:

1. *Use of pipelining to overlap accesses.* So far all access time measurements on the Performance Architecture only considered latency, ignoring throughput. However,the design is intended to cope with multiple outstanding accesses. Table 6.11 shows the delays per pipeline stage for four different kinds of TLB hits, including memory access (taken from table 6.2). If the processor can keep the pipeline filled at all times and no faults occur, *memory access times* (i.e. *not* MMU overheads) could be reduced to 66ns in the best case, which would be a reduction by over 33% for that case. However, the table also indicates that the pipeline is only ever utilised entirely in the case of a cache miss. In all cache hit cases the imbalance between the pipeline delays results in only two stages working simultaneously (this will, of course, vary depending on memory and cache access times). The pipeline could either be extended, by dividing the current first stage into two, or shortened through combination of the second and third stage. Which option results in greater performance depends on the average number of overlapping accesses the processor supplies and the memory and cache access times (as they are the dominant factor in third stage delay). As the majority of current ARM processors only allow overlap of two accesses, one instruction fetch and one data access, shortening the pipeline to two stages might be the better option.

2. *More speculation.* The concurrent execution of Alignment Check and TLB lookup already lead to speculation. The TLB lookup will be discarded if the alignment check fails. However, more 'aggressive' strategies could be implemented. One such strategy, speculative loads, is proposed in a later section.

3. *TLB improvements.* Two aspects of the TLB need to be reworked. The entry organisation, i.e. the number of entries and their associativity, should be increased. As soon as a non-direct mapped design is implemented, the choice of a suitable entry replacement algorithm also becomes an issue. Such an architectural change to the TLB would result in a higher hit rate compared to the current design. However, table 6.4 indicates that predicted system performance is limited by the overheads of the architecture, rather

| | Cache Hit Load | Cache Miss Load | Cache Hit Store | Cache Miss Store |
|---|---|---|---|---|
| **Stage1** | 66ns | 66ns | 66ns | 66ns |
| **Stage2** | 20ns | 20ns | 20ns | 20ns |
| **Stage3** | 20ns | 50ns | 32ns | 70ns |
| **Total** | **106ns** | **136ns** | **118ns** | **156ns** |

Table 6.11: MMU overheads distribution by pipeline stage for translated accesses. The memory access times are taken from table 6.2

than by the predicted TLB miss rate; so reducing the TLB translation overheads should be addressed first. One possible approach is the modification of the information contained within the entries. By caching more information in a TLB entry, less work needs to be done in the common case of a TLB hit. For example a decoded permission table can be stored in an additional 4-bit field. This would be indexed by the access type and the processor mode. To reduce the size of the lookup table, the system and ROM flags (SR fields) in the system coprocessor are assumed to be unchanged. This assumption is verified in parallel with the translation, by comparing the stored SR field with the current SR settings (figure 6.2).

4. *System architecture changes.* The current designs assume one MMU is used and shared between instruction and data accesses. However, as most performance ARM implementations use a split cache architecture, i.e. separate Instruction and Data caches, sharing a MMU would severely impair performance. Such systems are expected to have separate MMUs for each cache. Performance could be gained by optimising each MMU for its purpose as the Instruction MMU does not require all features of the Data MMU. Instruction MMUs do not need a colour mechanism (section 4.3.4) and do not require a full Abort Handler, as the system coprocessor does not need to be updated with any fault information in case of a prefetch abort. Furthermore, the permission checks can be simplified, as only read accesses are ever performed on the Instruction bus.

All of the suggested performance improvements reduce the overheads for translated TLB hits, the common case. Again, the advantages of asynchronous pipeline design become apparent. As the pipeline is not bound by a fixed cycle time, as in synchronous systems, it always performs at each pipeline stages data dependent maximum speed. Hence the common case, which all stages are optimised for, has minimised latency, whilst the rare cases simply cause the pipeline to slow down.

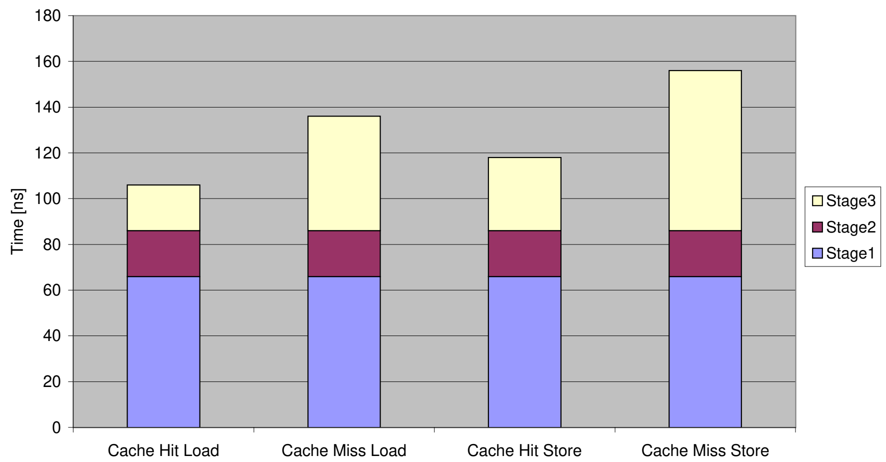


Figure 6.3: MMU overheads distribution by pipeline stage for translated accesses.

## 6.7 Performance Optimisation

This section describes two performance optimisations which address issues raised in the previous section. The Baseline Architecture optimisation is concerned with reducing the return-to-zero overhead (subsection 6.7.1). Finally, a speculative load mechanism is proposed (subsection 6.7.2).

The extension to the Baseline Architecture was successfully implemented and resulted in substantial overhead reduction. The speculative load mechanism however, was projected not to result in significant performance gains with the current implementation of the Performance Architecture and hence was not implemented.

### 6.7.1 Extended Baseline Architecture

The Baseline Architecture suffers from a long return-to-zero phase, as explained in section 6.4.1. Each MMU component does a `select` to wait for its input. Once its inputs are ready, the component generates its output and passes it on to the subsequent unit. As Balsa `select` statements 'enclose' the handshakes within their body (i.e. the input handshakes to a component remain pending, until all handshakes within the component's `select` body have completed). Hence, only when the memory access has finished all the nested handshakes can complete. Starting from the Memory Multiplexer component, one component after the other completes the pending output handshakes, which in turn, once all output handshakes are closed, completes the handshakes on its input channels. This behaviour leads to a 'ripple' of closing handshakes, which adds between 36ns and 48ns per access.

This problem was addressed by a register in the component which computes the physical address sent to the Memory Multiplexer. The register buffers the input to the Physical Address Calculation. The register immediately completes the handshake with the sending component (Permission Check). The return-to-zero ripple in all preceding components commences and occurs concurrently with the memory access. As a result, the biggest part of the rippling overhead is masked by the memory access time and does not contribute to the MMU overhead anymore.

Adding a register effectively pipelines the Baseline Architecture. If the ripple reaches the top level before the memory access has completed, a second access request could be accepted by the MMU. This needs to be avoided, as the Baseline Architecture was designed with the assumption of only one pending memory access:

- *Multiple pending access require a pipeline colour mechanism.* As outlined in section 4.3.4 a colour mechanism is necessary in asynchronous pipelines, as asynchronous pipelines cannot be 'flushed' in the same way as synchronous ones can.

- *The Memory Multiplexer is not arbitrated.* If multiple accesses can be pending in the pipeline, a TLB miss could lead to a memory access attempt, whilst the previously translated access is still pending. A similar problem arises from the bypass mechanism used when the MMU is deactivated. An untranslated access could attempt to access memory before the previous, translated access, has completed. Without arbitration within the memory multiplexer, the results of such attempts are undefined.

- *The Abort Handler is not arbitrated.* The Abort Handler was designed under the assumption, that none of the many abort sources could ever request an

> abort simultaneously, so that no arbitration was necessary. If a second access faults – for example during the alignment check – and the pending external memory access has also aborted, the result would be undefined.

Hence, a locking mechanism was implemented to prevent a new access being accepted by the MMU, whilst another access is still pending. The proposed modifications were implemented in an Extended Baseline Architecture, which performed in line with the projections, performing accesses 30ns to 40ns faster than the Baseline Architecture.

### 6.7.2 Speculative Loads

Speculative optimisations, by their nature, are always wasteful. Work is performed, which in some cases is unnecessary. Often a penalty occurs if the speculation was unsuccessful, as some 'cleanup' becomes necessary. Hence they need to be assessed carefully to guarantee that they produce a net gain in performance.

The most common memory access is an unaborted, translated load, which is cached in the TLB. The physical address, which is necessary to access memory, is effectively contained within the TLB entry[3]. However, the decision whether the cached memory access is allowed can only be taken when the permission and domain information have been evaluated. In the vast majority of cases this only introduces a delay, as the access is allowed to go ahead.

Speculative loads exploit this common case by performing the memory access as soon as the physical address is available. The memory access ends up being overlapped with the permission and domain checks, hiding those overheads. In the rare case that the permissions do not allow the read to occur, the read value is silently discarded. In those cases, however, a penalty is incurred as the MMU has to wait for the value from memory, even if it will be discarded.

This optimisation may only be performed on loads. The MMU has no way of recovering the state prior to a speculative store, in cases where a store needs to be aborted due to permission faults. As table 6.3 suggests that 93% of all memory

[3] The TLB entry only contains the physical page number. Concatenation with the page offset is required to form the final physical memory address.

accesses are loads[4], this limitation only has a small performance impact.

Some loads may also modify memory state. Reads from memory mapped peripherals may have side effects (i.e. starting an A/D conversion, advancing a FIFO, etc.), which make them irreversible. These types of accesses can be recognised, as they require the page to be marked uncachable. Even though other pages might be marked uncachable too, the speculative load strategy is guaranteed to be safe, if only loads from cachable pages are performed speculatively.

In the current implementation of the Performance Architecture however, the physical address only arrives 8ns before the result of the permission check at the Memory Arbiter. Hence the optimisation would only cut overheads by less than 10%. Alternative implementations and the combination of second and third pipeline stage are expected to increase the possible performance gain.

[4]Every instruction fetch is a load. Hence (1.0+0.26)/(1.0+0.26+0.09) of all memory accesses are loads.

# Chapter 7

# Conclusion

The introduction outlined three goals which motivated the research covered in this thesis:

1. *Development of the first asynchronous Memory Management Unit.* Two asynchronous MMU architectures were designed and implemented (chapter 4). A quite basic, mainly sequential design, the Baseline Architecture (section 4.2) was compared to a more performance oriented design, which uses pipelining and increased concurrency, appropriately named the Performance Architecture (section 4.3.1). A summary of the designs and their strengths and weaknesses follows later in this chapter (section 7.1).

2. *Modernisation and asynchronous adaptation of the ARM coprocessor interface.* Chapter 3 introduced a new ARM coprocessor interface, which integrates the coprocessor and processor more closely than the existing synchronous interface. An interface extension was proposed which allows the interface to cope with multiple outstanding coprocessor instructions and their out-of-order completion, allowing better integration of coprocessors into future ARM processors. The new interface became the basis of an implementation of a System Coprocessor, required by the MMU designs. Section 7.2 summarises the coprocessor related work and findings.

3. *Evaluation of suitability of Balsa as asynchronous HDL for large, performance critical designs.* Balsa, introduced in chapter 2, was used to implement the MMUs and the coprocessor. It turned out to be a very versatile and intuitive implementation language and was well suited to describe the designs. However, the generated circuits required manual modifications,

to operate correctly and at optimal speeds, due to the backend chosen. Section 7.5 sums up the Balsa experience in more detail.

The co-simulation evaluation setup and the MMU test suite and its suitability are discussed in section 7.3, whilst section 7.4 looks at the consequences the choice of asynchronous logic had on the MMU designs. The chapter concludes with a section suggesting follow-on work from this thesis, which also makes suggestions for improvements of the Performance Architecture.

## 7.1 The MMUs

Two MMU architectures were designed and implemented (refer to section 4.1.1 for information on ARM Memory Management):

- *Baseline Architecture (section 4.2).* The Baseline Architecture is a basic and mainly sequential implementation of the ARM MMU specification. It minimises the use of registers in the datapath, instead local channels are used to steer the data to the required place. No speculation is used and components are only activated when it is certain that their results are needed.

- *Performance Architecture (section 4.3.1).* The Performance Architecture uses pipelining and some speculation to improve performance over the Baseline Architecture. The MMU is divided into three pipeline stages: *Alignment Check and Translation*, *Permission and Domain Check and Physical Address Computation* and *Memory Arbiter and Abort Manager*.

The Performance Architecture results in an 46% reduction of memory latency on translated memory accesses compared to the Baseline Architecture. The latency reduction can be attributed to the additional concurrency, a faster TLB implementation (refer to section 7.5) and the absence of multiple levels of nested handshakes ('return-to-zero ripple' section 6.4.1).

The pipeline in the Performance Architecture uses a 'colour' mechanism to flush pending data memory accesses from the pipeline after a memory abort has occurred. Synchronous systems can trivially solve the problem, as a fixed number of pending accesses needs to be discarded; asynchronous systems cannot use such a system, as the number of pending accesses is not fixed (section 4.3.4). The colour system is only necessary for data memory accesses, instruction prefetches are not affected, as they are speculative in their nature and the processor will discard unnecessarily fetched instructions.

The pipelining improves the maximum throughput of the Performance Architecture by 33% compared to a Performance Architecture MMU which has only ever one memory access pending, or by 56% when compared to the Baseline Architecture. The maximum throughput is limited by the unequal latencies of the individual pipeline stages (table 6.11). In most cases the first pipeline stage causes the longest delay and limits the amount of concurrent work between pipeline stages. Further investigation into ways of achieving a more balanced pipeline (e.g. by reducing the number of pipeline stages or pipelining the first stage further) is required; such work also needs to consider the processor's capability to cope with multiple outstanding memory accesses.

Both designs use a simple TLB, with a parameterised number of direct mapped entries. The TLB entries are evenly split between entries caching instruction fetch translations and data access translations. Having the number of TLB entries parameterised takes full advantage of the HDL design process and it allows easy adaptation of the TLB to specific requirements. A 'lazy' TLB flushing mechanism is described (section 4.5.3) where the processor sets a 'flush request flag' which is reset by the MMU when the requested TLB entries are invalidated. This solution trades off arbitration for the TLB on every translated memory access against higher overheads on TLB flush. Thanks to the asynchronous implementation this only results in increased translation latency on TLB entry invalidation; the common case remains unaffected.

Further research on the TLB architecture and implementation is required. The current design is limited by the simple direct mapped entry organisation, which is expected to result in frequent conflict misses. Furthermore, each TLB entry only caches the translation for a one kilobyte area of virtual memory, independent of the page sized used (section 4.5.1). This results in inefficient use of TLB entries for small pages, large pages and sections, as multiple entries contain the same translation information. Due to time constraints no investigation into more efficient schemes (e.g. using per entry masks on tag comparison) was possible.

Performance projections, based on simulations of the synthesised transistor netlists, suggest an absolute, average, translated memory access latency of around 160ns for the Baseline Architecture and of around 100ns for the Performance Architecture (section 6.2). As MMUs are usually only employed on platforms which need substantial computing power, these figures need to be improved before the MMUs are of any practical use in an asynchronous system. Improvements to the Balsa synthesis system (e.g. an improved backend) and the MMU architecture will be necessary to achieve this goal. Balsa also offers the possibility to integrate custom designed circuits to be integrated into synthesised designs; this option seems most appropriate for the TLB and seems worthy of further investigation.

## 7.2 The System Coprocessor

A new ARM coprocessor interface was proposed in chapter 3. The new interface closely integrates the processor and coprocessors, interfacing at the processors decode, execute and memory access stage. It is intended to replace the existing synchronous interface, which was mainly designed for off-chip expansion capabilities, which are no longer required.

The new interface is better suited to an asynchronous system as it does not require 'cycle counting' by the coprocessor, to deduce when the coprocessor instruction has reached certain pipeline stages of the processor – there are no global synchronised cycles in a self-timed system. The new interface also does not require 'bus snooping' by the coprocessors anymore, which reduces the load on the instruction bus and it frees the coprocessor from all dependencies on the processors pipeline structure. However, the asynchronous interface does not offer the processor an option to force a coprocessor to abandon execution of a coprocessor instruction once it has begun, whilst the synchronous interface uses this feature to bound the interrupt latency.

An extension to the interface is also suggested (section 3.1.4) which allows out-of-order and/or multiple scheduling of coprocessor instructions, features which are not supported by the standard ARM coprocessor interface. The scheme is based on tagging coprocessor instruction to allow instruction identification independently of their position in the fetch stream. In combination with an extended coprocessor decode reply, containing information on coprocessor and instruction reordering, it is flexible enough to allow mixed use of reorder capable and non-reorder capable coprocessors with a superscalar processor. Whilst the interface was designed for use in an asynchronous system, it is equally applicable and easily transferable into synchronous designs. The interface is used on the SPA processor [24], a synthesised, asynchronous, ARM compatible processor developed at the University of Manchester, for use in secure smartcards.

A coprocessor, required for use with the MMU, was designed and implemented which uses the proposed interface (section 3.2). The dominant feature of the pipelined design is the register bank. The register bank is a contended resource, as it is accessible by the processor, via coprocessor instructions, and also required by the MMU on every memory access. The arbitration issue is complicated by the MMU coherency requirements. All coprocessor registers read during the translation of a memory access have to refer to the same coprocessor register state. A locking mechanism was implemented which allows the MMU to lock the register bank and prevent accesses by the processor (which stalls until the MMU releases the registers).

The projected performance of the design, based on transistor netlist simulation, suggests execution cycle times of 20ns for reads and 32ns for writes. These times increase if the coprocessor is locked by the MMU, as the processor access stalls. In an asynchronous system, those access times should be acceptable as the processor would only run at that speed when executing coprocessor instructions. As accesses to the system coprocessor are quite rare the impact on processor performance should be small. However, the slow coprocessor access times highlight the need for performance improvement in the Balsa synthesis system.

## 7.3 The Evaluation setup

Verification and evaluation of the MMU and coprocessor transistor netlists was performed using a co-simulation setup (chapter 5). The Balsa generated circuits were connected to software emulation of an ARM processor and a memory model. Such a complex test environment was chosen over simpler test vector based systems, as it allowed realistic simulation of the interactions between processor, coprocessor, MMU and memory. Furthermore, the test setup is easily adaptable and reusable for tests of other memory subsystem components, such as caches or Memory Protection Units, or other coprocessors which implements the new ARM coprocessor interface.

Due to the great amount of state (page tables, system coprocessor registers, past accesses, processor mode, etc.) which affects MMU operation, a test suite was developed to facilitate systematic testing. The suite is designed around a kernel (section 5.3), which provides coprocessor and page table interfaces to the test programs.

The test suite was invaluable in the development of a reference software MMU implementation, but the usability of the test suite for debugging of the synthesised MMU suffered from the very slow execution of the co-simulation setup. This was countered by the addition of simple, fast executing tests to the verification suite, which were used in the initial phases of the MMU design cycle.

The Balsa design flow allows simulation of Balsa designs via LARD, an asynchronous modelling language. Due to the complexity of the required LARD test setup and the lack of performance conclusions such simulation would have provided, it was chosen to omit this step (section 2.4.4). In retrospect it would have been useful to have had a LARD test bench capable of executing the MMU verification suite. As the behavioural LARD simulations are a lot faster than transistor netlist simulations they would have allowed more complete validation of the new MMU designs.

## 7.4 Asynchronous Design Benefits

The main motivation for choosing an asynchronous implementation style was the lack of asynchronous MMUs. As asynchronous processors mature and improve in performance, it seems likely that a time will come, when asynchronous systems require a MMU. The introduction to asynchronous logic, introduced several key benefits of asynchronous design compared to synchronous design methodology. This section analyses how these benefits apply to the designs described in this thesis.

The benefits of asynchronous logic, as introduced in section 1.2.1 were:

1. *Data dependent execution time.* The bundled data Balsa backend, used for all designs, does not perform any completion detection to detect the end of an operation (however other Balsa backends do). Hence no data dependent execution time was used on a per operation basis. Data dependent execution time however, is frequently taken advantage of on an architectural level, where components have data (or state) dependent execution times. The Translate component for example usually performs a TLB lookup; only in rare cases does a very slow lookup through page tables in memory become necessary. Thanks to the properties of asynchronous design, no fixed duration constraint (i.e. n-clock cycles) is imposed on the work of the component. This allows very clean architectural design, optimised for the common case. Where synchronous designs require worst-case oriented cycle times (and as a result additional pipeline stages to minimise cycle times again), asynchronous components only use as much time per cycle as required to perform the work during that cycle. Other examples of this feature include:

    - *Short per stage cycle time in the Performance Architecture on deactivated MMU.*
    - *Variable memory access timing in both MMU architectures (Cache, Main Memory).*
    - *Different cycle time on reads and writes in the coprocessor.*

2. *Simple modular composition.* The absence of a clock allows simple composition of asynchronous blocks to systems. This ability is nicely demonstrated

in the evaluation setup. Even though both MMU architectures perform at significantly different speeds, they were both connected to the same processor emulation. No changes to the test setup were necessary. This is expected to hold true for a computer system. A SoC designer can simply replace one MMU with another, to achieve higher performance. The same is valid for the MMU micro architecture. Components within the MMU can be replaced by alternative implementations with identical interfaces. A faster TLB, for example, could very easily be integrated into either MMU, without requiring any other changes to the designs.

3. *Low power consumption. Better electro-magnetic compatability.* As the laid-out designs required substantial manual modifications to operate, due to the backend chosen, no work was undertaken to investigate power consumption or electro-magnetic radiation. Hence no conclusions are possible.

The main disadvantage of asynchronous logic, the high design complexity, was partly overcome through the use of Balsa. Even though the use of an asynchronous HDL greatly simplified the implementation step (which is discussed in more detail in th next section), it does not address many of the other complexities of asynchronous design. Deadlocks, due to incorrect architectural design, can only be avoided through understanding of asynchronous methodology. The same applies to improving performance by maximising concurrency.

Nevertheless, the asynchronous design style chosen, worked well for the architectural design of the MMUs and coprocessor. A 'Lego brick' design style became possible. Initially, very simple implementations of MMU components were 'plugged' together. These, often functionally incomplete blocks, were then replaced with more sophisticated or functionally complete blocks as work progressed. For example, the TLB was only added after the MMUs already worked. Due to their asynchronous nature the resulting designs always performed at their best, no worrying about clock cycle times or pipeline rebalancing, was necessary. The only downside of the current asynchronous implementation is the poor performance. However, it is more likely that architecture and synthesised implementation are to blame, rather than their asynchronous nature, as fast asynchronous circuits have been demonstrated in the past [20].

## 7.5 Suitability of Balsa

Balsa, an asynchronous hardware description language, developed by Bardsley at the University of Manchester, was used as the implementation method for the MMU and coprocessor. The language was very easy to learn, especially the strong typing system, which incorporates user-defined types. This enabled a very structured implementation and was very useful in providing strong interfaces between components on all levels (i.e. between pipeline stages, within components, etc.).

The development tools were stable and integration of the individual tools was straightforward. At no point did the language seem insufficient to describe the intended designs, or enforce undue constraints. The change from synchronous HDLs to Balsa was reasonably simple, even though prior experience in multi-threaded programming seems a more useful background, as many similarities exist. Balsa's syntax directed translation approach turned out useful in the design cycle. Due to the previously described iterative design methodology, it was important to rely on the generated circuits to perform similar to previous iteration. An addition or modification to the design only affected performance of the modified component. This is unlike synchronous languages, like VHDL, where a small change can dramatically change the generated circuit and its characteristics [4].

The only problems arose due to the choice of backend. A bundled data backend was chosen, as it offers the best performance [30]. This backend however does not generate delay-insensitive circuits. Manual timing verification and possible modifications are necessary to guarantee timing closure. All synthesised circuits failed simulation when they were laid out and included capacitance. Even the transistor netlist, without capacitance consideration, required modifications to perform reliably (section 6.5.3).

No tools currently exist to aid timing verification or aid to perform the required modifications (inserting buffers, resizing transistors, etc.). Without some levels of automation, it is very difficult to achieve timing closure on complex designs. As long as such tools are unavailable, only the guaranteed delay-insensitive backends should be considered for complex designs. The best performing (quasi-) delay-insensitive backend generates circuits which run at around half the speed of the bundled data implementations and use about three times as much area [30].

As poor performance was already the major problems of the generated designs, use of such backends seems hardly feasible for performance oriented designs. Hence more work is needed on the synthesis aspect of the Balsa system to make it a feasible alternative for the design of complex performance oriented designs.

The lessons learned from the work with Balsa can be summarised:

- *Using registers to pass data between parts of a component is faster than using local channels.* Even though the use of registers within components leads to less concurrent designs (as reads and writes can only be sequential), it resulted in better performance than the use of local channels (section 6.4.2). Naturally this result depends on the amount of extra concurrency local channels can introduce.

- *Complex designs synthesised using the bundled data backend require manual modifications to operate.* Whilst smaller designs only require timing verification, larger circuits are expected to require modifications to work at all.

- *Transistor netlist cosimulation is suitable for all stages of system design, but slow on large designs.* As outlined previously (section 5.4.1) the cosimulation of Balsa generated transistor netlists was suitable for all steps of the design process, including debugging. It can minimise work, as it allows the skipping of the LARD simulation level, which might require a non-trivial test bench design. However, depending on the available computing power and the complexity of the design, a behavioural LARD simulation might be necessary to allow faster testing turn-around.

## 7.6 Future Work

As outlined in the previous section, the Balsa environment could benefit greatly from better performance of generated circuits. *Tools aiding timing closure* are necessary, to allow use of the fast bundled data backend in complex designs. A *comparative performance analysis of different Balsa constructs* would be very useful in designing higher performance circuits with Balsa. As in most languages, several ways exist of solving a design problem. Having performance figures for the different alternatives would aid the Balsa user and could also pinpoint weaknesses of current backends.

The current TLB implementations are mainly intended as a proof-of-concept. Work on the *design of asynchronous TLBs* is necessary to construct a competitive asynchronous MMU. Due to Balsa's capabilities to use compiled modules, such work is not constrained to use Balsa to derive a fast implementation. As TLBs are basically fast caches the work on asynchronous caches, such as [9], might point the way.

The new coprocessor interface also invites further investigation. Implementation of, for example, an *asynchronous ARM floating point coprocessor* using the new interface might offer interesting material on the suitability of the interface. Furthermore *further investigation into the performance impact of using the out-of-order capabilities of the extended coprocessor interface* is needed, to assess its value and analyse possible system performance gains.

Lastly, an *implementation of the improved MMU design*, proposed in the next subsection might prove an interesting starting point towards asynchronous MMU designs, feasible for use.

### 7.6.1 Suggested MMU Architecture Improvements

This section gathers the suggested improvements to the Performance Architecture MMU architecture, based on the findings in this thesis. The important suggested changes, compared to the Performance Architecture are:

- *Improved TLB.* The TLB needs further investigation. Each TLB entry needs to cache an entire page translation, not a fixed sized area and more entries should be combined in a set-associative, or fully-associative manner to make more efficient use of the available entries. The suggested extention that TLB entries should contain additional information, to improve performance of physical address translation and permission check (section 6.6) also requires further analysis.

- *Separate, specialised MMUs for instruction and data accesses.* As outlined in section 6.6 the MMU required for instruction accesses can be substantially simpler than the MMU used for data accesses. As instruction fetches are far more frequent than data accesses, providing a cut-down, faster instruction MMU should result in improved system performance.

- *Two stage pipeline.* Assuming that the processor cannot handle more than two outstanding data accesses (a load and a buffered store for example), there is no point in providing more pipeline stages, as they would not operate concurrently. Having a large number of pipeline stages in the instruction MMU might result in poor performance, as all pending fetches have to complete. Depending on branch predictor performance, having too many pending instruction fetches results in bad performance, as time is spend on waiting for prefetched instructions which will be discarded straight away.

- *Speculative Loads.* Section 6.7.2 outlined a speculative load mechanism. Whilst it was not suitable for the current MMU designs, it is predicted to result in significant performance gains, when combined with an improved TLB, as suggested above.

Besides all the architectural improvements, changes in the implementational style will probably result in greater performance gains. A faster Balsa backend or more traditional manual implementation of the MMU is needed to build an asynchronous MMU which performs on par with fast asynchronous processors. Once asynchronous processors will have matured enough to use MMUs it will be very interesting to look back on this thesis and compare the designs; and maybe this thesis will have played a tiny role in reaching that goal.

# Bibliography


[1] ARM Limited. *ARM Architecture Reference Manual*, February 2000.

[2] ARM Limited. *ARM9TDMI Technical Reference Manual*, March 2000.

[3] Andrew Bardsley. Balsa: An asynchronous circuit synthesis system. Master's thesis, University of Manchester, 1998.

[4] Andrew Bardsley. *Implementing Balsa Handshake Circuits*. PhD thesis, University of Manchester, 2000.

[5] Kees van Berkel. Beware the isochronic fork. *Integration, the VLSI journal*, 13(2):103–128, 1992.

[6] Kees van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5. Cambridge University Press, 1993.

[7] Wesley A. Clark and Charles E. Molnar. Macromodular computer systems. In Ralph W. Stacy and Bruce D. Waxman, editors, *Computers in Biomedical Research*, volume IV, pages 45–85. Academic Press, 1974.

[8] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, 1997.

[9] J.D. Garside D. Hormdee. Amulet3i cache architecture. In *Proceedings Async 2001, IEEE Computer Society Press*, pages 152–161, 2001.

[10] S. Dhar, M. Franklin, and D. Wang. Reduction of clock delays in vlsi structures. In *Proceedings of IEEE International Conference on Computer Design*, pages 778–783, 1984.

[11] Andrew Bardsley Doug Edwards. *Balsa 3.0 User Manual*. University of Manchester, 1999.

[12] P. B. Endecott. Lard documentation home page. http://www.cs.man.ac.uk/amulet/projects/lard/index.html.

[13] P. B. Endecott. Behavioural modelling of asynchronous systems for power and performance analysis. In *Proceedings PATMOS '98*, 1998.

[14] P. B. Endecott. Modelling and simulation of asynchronous systems using the lard hardware description language, 1998. ftp://ftp.cs.man.ac.uk/pub/amulet/papers/lard_esm98.ps.gz.

[15] R. Fuhrer, S. Nowick, M. Theobald, N. Jha, B. Lin, and L. Plana. Minimalist: An environment for the synthesis, verification and testability of burst-mode asynchronous machines. *Technical Report CUCS-020-99, Columbia University*, 1999.

[16] Steve Furber. *ARM System Architecture*. Addison Wesley Longman, 1996.

[17] C.A.R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice Hall International, 1985.

[18] V. Illingworth I.C.Pyle, editor. *Dictionary of Computing*. Oxford University Press, 4th edition, 1996.

[19] IST-1999-13515. http://www.g3card.org, 2001.

[20] A. Bardsley D.M. Clark D.A.Edwards S.B. Furber J. Liu D.W. Lloyd S. Mohammadi J.S. Pepper O.Petlin S. Temple J.V. Woods J.D. Garside, W.J. Bainbridge. Amulet3i - an asynchronous system-on-chip. *Proceedings Async2000*, IEEE Computer Society Press:162–175, April 2000.

[21] David A. Patterson John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2nd edition, 1996.

[22] T. Kilburn, D. Edwards, M. Lanigan, F. Summer, and s system. One-level storage system. *IRE Transactions on Electronic Computers*, 1962.

[23] S.A. Brandt K.M. Fant. Null convention logic: A complete and consistent logic for asynchronous digital design. http://www.theseus.com/PDF/ncl_paper.pdf, 1997.

[24] W.J. Bainbridge A. Bardsley J.D. Garside S. Temple L.A. Plana, P.A. Riocreux. Spa – a synthesisable amulet core for smartcard applications. In *Proceedings Async 2002, IEEE Computer Society Press*, pages 201–210, 2002.

[25] P. M. Lewis II and C. L. Coates. *Threshold Logic*. John Wiley & Sons, Inc., New York, NY, 1967.

[26] Mukund. Petri nets and step transition systems. *IJFCS: International Journal of Foundations of Computer Science*, 3, 1992.

[27] Chris J. Myers and Alain J. Martin. The design of an asynchronous memory management unit. *Technical Reports CS-TR-92-25*, 1992.

[28] Steven Mark Nowick. Automatic synthesis of burst-mode asynchronous controllers. Technical Report CSL-TR-95-686, 1995.

[29] Nigel C. Paver. *Design and Implementation of an Asynchronous Microprocessor*. PhD thesis, University of Manchester, 1994.

[30] W. B. Toms. Delay insensitive implementations of balsa synthesised circuits. *Transfer Report, University of Manchester*, 2001.

[31] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij. The vlsi programming language tangram and its translation into handshake circuits. In *Proceedings of the European Design Automation Conference*, pages 384–389, 1991.

# List of Acronyms

ADFMI ....... Advanced Digital Functional Modelling Interface  Software interface used by the *Timemill* circuit simulator. It allows software descriptions of circuits to be included into simulations (*cosimulation*).

CMOS ........ Complementary Metal Oxide Semiconductor  Currently the dominating implementation technology for integrated circuits.

CSP .......... Communicating Sequential Processes  A process algebra developed by C.A.R. Hoare [17] used to describe and reason about concurrent systems.

DI ............. Delay Insensitive  A Delay model in which components and wires are allowed to have arbitrary delays. Refer to section 1.2.3 for more detail.

EMR .......... Electro-Magnetic Radiation  Radiation given off by all pieces of electric equipment, to which they are also susceptible [18].

FAR .......... Fault Address Register  Register in the System Coprocessor containing the address of the last aborted data access.

FIFO ......... Here: Acronym for 'First in first out'. Refers to a buffering system which behaves like a queue.

FLD .......... First Level Descriptor  An entry in a first level page table in the ARM architecture.

FS ............ Fault Status  Register in the System Coprocessor containing a code describing the reason for the last aborted data access and its domain number.

GALS......... Globally Asynchronous, Locally Synchronous  Systems which use *asynchronous* interconnects to connect *synchronous* modules.

HDL.......... Hardware Description Language  Language used as the input into a hardware synthesis system. Common examples are *VHDL*, Verilog, *Balsa* and *Tangram*.

LARD........ An asynchronous modelling and simulation language developed at the University of Manchester[12].

MPU.......... Memory Protection Unit  Component which provides memory protection facilities without offering address translation as available with a *MMU*.

NCL.......... Null Convention Logic

OS............ Operating System  "The set of software products that jointly controls the system resources and the processes using these resources on a computer system" [18]. This includes management of the *page tables* and the *MMU*.

PTE.......... Page Table Entry  Same as an *SLD* in the ARM architecture.

QDI........... Quasi Delay Insensitive  A Delay model in which components and wires are allowed to have arbitrary delays, but it assumes an isochronic fork. Refer to section 1.2.3 for more detail.

ROM......... Read-Only Memory  Memory which can only be read and not be written to. Often used to contain system software.

SI............. Speed Independent  A Delay model in which only components are assumed to have delays, all wire delays are assumed to be non existent. Refer to section 1.2.3 for more detail.

SLD.......... Second Level Descriptor  An entry in a second level page table in the ARM architecture.

SoC . . . . . . . . . . . System on Chip    Designs on which an entire computer system, usually consisting of a processor core, several peripherals and some memory, is integrated on a single chip. Often employed in embedded applications.

TLB . . . . . . . . . . Translation Lookaside Buffer    Small cache used by the *MMU* to cache address mapping and permission information. Typically implemented as a n-way associative cache with tens or hundreds of entries.

VHDL . . . . . . . . Very High Speed Integrated Circuit Hardware Description Language

# Glossary

**abort** Here: Signal sent by the *MMU* to the processor indicating that a pending memory access was not allowed to complete. It is the responsibility of the *OS* to determine the cause and take appropriate action.

**acknowledgement** Here: One of the control signals required to perform a *handshake*. In this thesis all handshakes use the 4-phase protocol and hence acknowledgement is active when it is high. Also refer to *request*.

**Amulet3** Latest member of the *asynchronous* Amulet processor family developed by the Amulet group at the University of Manchester. All Amulet processors implement the commercial *ARM* instruction set. Amulet3 incorporates advanced architectural features such as a reorder buffer and a branch target buffer [20]. It was designed for use in a commercial cordless phone handset.

**arbiter** Here: *Asynchronous* component used to arbitrate between access requests for a contended resource. Refer to section 1.2.4 for further detail.

**ARM** *Advanced RISC Machines* Company which designs the microprocessors by the same name.

**associativity** Here: Set associativity in caches or TLBs. Refers to the number of possible locations (sets) in which a cached entry might be found.

**asynchronous** Refers to *asynchronous* circuits. Circuits are considered asynchronous when they are self timed, i.e. when they don't use a global synchronization reference, e.g. a *clock*.

**back annotation** A technique to feedback timing information from a routed circuit (or routed components) into the simulation of a design.

**back-ends** Here: Parts of the Balsa tools which convert the *breeze* netlist into a transistor level netlist. A variety of back-ends exist, using different implementation styles to implement the *handshake components* and their interconnects. The *bundled data* backend was used for the ciruits in this thesis.

**Balsa** Asynchronous hardware description language developed at the University of Manchester. Refer to section 2.2 for details.

**branch shadow** In some architectures the instructions immediately following a branch instruction are special. They are said to be in the 'branch shadow'. They usually are executed even if the branch is taken. The behaviour is usually tied to the pipeline organisation.

**Breeze** The proprietary Balsa netlist format, describing *handshake components* and their connections. It is used as the intermediate format, used by the various Balsa *back-ends*.

**C** Here: Procedural programming language. (Actually the sucessor to a language named B...)

**capacity miss** Reason for a cache or TLB miss caused by the cache or TLB not having enough entries to accomodate the entire working set. The frequency of capacity misses can be lowered by increasing the size of the cache or TLB.

**channel** Here: Combination of data and associated *handshake* signals. The handshake signals may be encoded together with the data, or be transferred on separate wires.

**clock** A "series of fixed width pulses having a fixed repetition frequency" [18]. The frequency is usually specified in Hertz (Hz). The rising or falling edge of the signal is used as global synchronization reference for *synchronous* circuits.

**clock domain** Parts of a circuit which use a particular *clock* signal as reference. Modern chips often require several clocks at different speeds.

**clock gating** Technique employed by designer of *synchronous* designs to stop the *clock* temporarily in parts of a circuit. Used to conserve power, as *CMOS* circuits mainly consume power when switching.

**clock skew** Delay of the *clock* signal. Leads to the clock edge being visible at different times in different parts of a circuit.

**completion detection** Logic which detects when an (*asynchronous*) circuit has computed its outputs, i.e. when a mathematical operation has completed. Used in asynchronous design to signal availability of a result. Necessary to take advantage of data-dependant execution times.

**conflict miss** Reason for a cache or TLB miss caused by several items in the working set competing for the same cache or TLB entry. The frequency of conflict misses can be lowered by increasing the *associativity* of the cache or TLB.

**cosimulation** Here: Approach to simulation of designs, which alows mixing of transistor level netlist and software emulations of functional blocks in a simulation setup. This allows the testing of netlists when the remaining, necessary parts of the system are not available as netlists.

**descriptor** Entry in a *page table*. Contains translation and management information for the *MMU*. Maintained by the operating system. Refer to section 1.3 for details.

**direct mapped** Here: Entry organisation used by the TLB. Each virtual address translation has exactly one possible location in the TLB. This is effectively a mapping with an *associativity* of one.

**domain access value** Here: A two bit value, associating certain domain access permissions with a domain number. The value can either forbid all access, or give *domain manager* or *domain client* access rights to all pages belonging to a domain.

**domain client** Here: A domain client uses the access permissions stored in the page tables, no special access rights are granted by this *domain access value*.

**domain manager** Here: One of the ARM *domain access values*. A domain manager is allowed full access to all pages belonging to that domain, independently of the access permissions stored in the page tables.

**handshake** Here: Most common way of inter-block communication in *asynchronous* designs. Can be used to synchronize a data transfer or as a control signal. Uses *request* and *acknowledgment* signals to indicate data availability and correct receipt. An example is given in figure 1.3. A wide range of possible implementations exist.

**handshake components** Usually paramterised, simple asynchronous library components, which are used to construct larger asynchronous circuits. *Balsa* and Tangram use these components as the basis of their synthesis.

**memory tube** Here: A very simple character device. All bytes written to the device address are interpreted as asciii characters and displayed. Used for simple processor tests on bare chips, when no operating system is present to facilitate output.

**page** Here: A fixed sized portion of memory. Usually the smallest unit which can be remapped by a *MMU*. Also refer to section 1.3.

**page number** Portion of a memory address which selects the *page*. The remainder of the address is used to index into the selected page. Also refer to figure 1.7.

**page table** Table which stores address translations and related management information in *descriptors*. Is used by an *MMU*. Refer to section 1.3 for details.

**Petri nets** A model of a concurrent system that is expressed in a specific graphical notation and can be used to explore certain properties of the system. A Petri net consists of a set of *places*, a set of *transition bars*, and a set of directed edges. A specific state is represented by a specific allocation of *tokens* to places, called a *marking*. They were devised in the early 1960s by C.A.Petri[18].

**physical address** Address used to access physical memory in a computer system. If a *MMU* is used to translate memory accesses, it might differ from the *virtual address*.

**program counter** A register which contains the address of the currently executing instruction. It is automatically updated by the processor on every instruction execution.

**propagation delay** Delay of a signal through a component.

**replacement strategy** Here: Policy which decides which entry of a cache or TLB will be overwritten when a new entry is added. Popular examples include *least recently used*, *random* or *cyclic*. A decision is only necessary in systems with an *associativity* greater than one, where more than one location may hold a specific entry.

**request** Here: One of the control signals required to perform a *handshake*. In this thesis all handshakes use the 4-phase protocol and hence request is active when it is high. Also refer to *acknowledgement*.

**segment** Here: A variable sized portion of memory which is remapped by a MMU. Usually described by a start address and maximal offset. Has some implementational disadvantages compared to *pages*.

**SPA** An asynchronous ARM compatible processor developed at the University of Manchester. It is fully synthesized from a *Balsa* description.

**split cache** A cache which is split into seperate parts for instructions and data. Having seperate instruction and data caches allows accesses to both in parallel.

**strongly typed** Here: Strongly typed languages. All data items (variables) need to be declared to be of a specific type. The language only allows operations on data items of compatible type. Balsa forces all type conversions to be explicit.

**subpage** Here: The ARM architecture has finer granularity for protection than for translation. There are four subpages in each large or small page.

**superscalar** Processors are considered superscalar, when they can issue and execute multiple instructions per cycle. They usually include sophisticated reorder logic, which reorders instructions to take advantage of instruction-level parallelism.

**synchronous** Refers to *synchronous* circuits. Circuits are considered synchronous when they use a *clock* as global timing reference to synchronize circuit operation.

**table walking** Process of looking up an entry in a *page table*. So called because many page tables are implemented as multi level tables and require several lookups, where the next table accessed is dependent on preceding table entry.

**tags** Here: An identifier. A fixed number of bytes which uniquely identify an item.

**test bench** Here: Setup used to test a circuit. Sets of input signals (test vectors) are applied to the circuit to be tested and the outputs of the circuit are compared to reference results. This allows automated testing of designs under various conditions.

**threshold gate** A logic element whose output is determined by comparing a weighted sum of inputs with a fixed threshold. If the threshold is exceeded, the output is a logic true otherwise a logic false.

**Timemill** Transistor netlist simulation tool used for all netlist simulations in this thesis.

**timing closure** Timing closure is achieved when all necessary timing assumptions for a circuit have been met, i.e. setup and hold times are guaranteed to be met under all circumstances.

**virtual address** Memory address as used by an application running on a processor. If a *MMU* is used to translate memory accesses, it will be translated into a *physical address* before is is used to actually access memory.

# Appendix A

# ADFMI Code Example

This appendix gives a complete example for one of the smaller ADFMI interface blocks used for the MMU evaluation setup. The example given is the ADFMI interface block for the asynchronous memory emulation. The actual memory model is seperated from the ADFMI interface and not included in the example. It illustrates the divided state machine. The main state machine (memoryAccess(), for FMSIMPHASE) only contains three states: AWAITING_ACCESS, PENDING_WRITE and PENDING_RETURN_TO_ZERO. The various possible permutations of handshakes closing are all covered managed by the returnToZero() function which allows pending handshakes to complete in any order.

---

```
/**
   ADFMI Memory Model Interface

   MMU Evaluation setup

   (c) 2001 Frank Weigel

   This is the ADFMi interface for the
   ADFMI Asynchronous Memory Model.
**/


#include <stdio.h>
#include <stdlib.h>
#include "vec.h"
#include "fmtype.h"
#include "fmuser.h"
#include "ctype.h"
#include "../adfmiUtil.h"
#include "memory.h"
```

```
/* State defines */
#define AWAIT_ACCESS 0
#define PENDING_READ 1
#define PENDING_WRITE 2
#define PENDING_ABORT 3
#define PENDING_RETURNS_TO_ZERO 4

/*Return to Zero MAsK states */
#define ALL_RESET 0

/* MASKs for return to zero flags */
#define BUNDLE 0x1
#define OUT   0x2
#define IN      0x4
#define ABORT 0x8

/* Handshake signals */
int reqBundleID, ackBundleID, weID, sizeID, addressID;
int reqOutID, ackOutID, outID;
int reqInID, ackInID, inID;
int reqAbortID, ackAbortID, abortID;

/* Global bus values */
unsigned currentAddr, data, we, size;

/* Memory Emulation Error Status */
extern int physMemoryErrorStatus;

/* State Machine Initialisation */
int currentState = AWAIT_ACCESS;
int pendingReturnsToZero = ALL_RESET;
char currentAccessPending = 0;


/* Register ADFMI Model with Timemill */
int MemoryRegisterUserModel ()
{
  FMHANDLE *body;

   fprintf (stderr,"[memory.c]: Registering memory model...\n");

  fmNewHandle (body);
  fmDefineModel ("memory", (FMFUNC) memoryAccess, 71,37 , 0, 0, body);
  /* Control signal "Bundle" Port: Handshake and Data signals */
  reqBundleID = fmDefinePort (body, "reqBundle", fmDigital, fmInport);
  ackBundleID = fmDefinePort (body, "ackBundle", fmDigital, fmOutport);
  weID = fmDefinePort (body, "we", fmDigital, fmInport);
  sizeID           = fmDefineVectorPort (body, "size",1,0, fmDigital, fmInport);
  addressID      = fmDefineVectorPort (body, "address",31,0, fmDigital, fmInport);
  /* Data OUT Port */
  reqOutID = fmDefinePort (body, "reqOut", fmDigital, fmOutport);
  ackOutID = fmDefinePort (body, "ackOut", fmDigital, fmInport);
  outID = fmDefineVectorPort (body,"out", 31, 0, fmDigital, fmOutport);
```

```
  /* Data IN Port */
  reqInID = fmDefinePort (body, "reqIn", fmDigital, fmInport);
  ackInID = fmDefinePort (body, "ackIn", fmDigital, fmOutport);
  inID = fmDefineVectorPort (body,"in", 31, 0, fmDigital, fmInport);
  /* Abort Signal Port */
  reqAbortID = fmDefinePort (body, "reqAbort", fmDigital, fmOutport);
  ackAbortID = fmDefinePort (body, "ackAbort", fmDigital, fmInport);
  abortID = fmDefinePort (body, "abort", fmDigital, fmOutport);

  fmFreeHandle (body);
  fprintf (stderr, "[memory.c]: Registration done.\n");
}

/* Memory Access State Machine */
int memoryAccess ()
{
  int *tmp=NULL;

  switch (fmSimPhase())
    {
    case FMDCINITPHASE:
      /* Simulation Initialisation */
      fmSetPortStateById (ackBundleID, UNDEF);
      fmSetPortStateById (ackInID, UNDEF);
      fmSetPortStateById (reqOutID, UNDEF);
      fmSetPortStateById (reqAbortID, UNDEF);
      fmSetPortStateById (abortID, UNDEF);
      fmSetPortStatesById (outID,31,0, undefWord);
      /* Initialise Memory Model */
      init_phys_mem(MEMORY_SIZE, MEMORY_IMAGE, 0,
            MEMORY_IMAGE_LOAD_OFFSET, "");
      break;

    case FMKICKOFFPHASE:
      /* SImulation Start */
      fmSetDigitalPortDelayDriveById(DELAY_ACK_BUNDLE_1_0,ackBundleID
            ,600.0,400.0,FMDRIVEDOWN);
      fmSetDigitalPortDelayDriveById(DELAY_ACK_BUNDLE_0_1,ackBundleID
            ,600.0,400.0,FMDRIVEUP);
      fmSetDigitalPortDelayDriveById(DELAY_ACK_IN_1_0,ackInID,600.0,400.0,
            FMDRIVEDOWN);
      fmSetDigitalPortDelayDriveById(DELAY_ACK_IN_0_1,ackInID,600.0,400.0,
            FMDRIVEUP);
      /* Make sure that the req's will ALWAYS be sent after the data */
      fmSetDigitalPortDelayDriveById(DELAY_REQ_OUT_1_0 + MAX(
            DELAY_OUT_1_0,DELAY_OUT_0_1),reqOutID,600.0,400.0,
            FMDRIVEDOWN);
      fmSetDigitalPortDelayDriveById(DELAY_REQ_OUT_0_1 + MAX(
            DELAY_OUT_1_0,DELAY_OUT_0_1),reqOutID,600.0,400.0,FMDRIVEUP);
      fmSetDigitalPortDelayDriveById(DELAY_REQ_ABORT_1_0 + MAX(
            DELAY_ABORT_1_0,DELAY_ABORT_0_1),reqAbortID,600.0,400.0,
            FMDRIVEDOWN);
```

```
fmSetDigitalPortDelayDriveById(DELAY_REQ_ABORT_0_1 + MAX(
      DELAY_ABORT_1_0,DELAY_ABORT_0_1),reqAbortID,600.0,400.0,
      FMDRIVEUP);
fmSetDigitalPortDelayDriveById(DELAY_ABORT_1_0,abortID,600.0,400.0,
      FMDRIVEDOWN);
fmSetDigitalPortDelayDriveById(DELAY_ABORT_0_1,abortID,600.0,400.0,
      FMDRIVEUP);
fmSetDigitalVectorPortDelayDriveById(DELAY_OUT_1_0,outID
      ,31,0,600.0,400.0,FMDRIVEDOWN);
fmSetDigitalVectorPortDelayDriveById(DELAY_OUT_0_1,outID
      ,31,0,600.0,400.0,FMDRIVEUP);
/* Ignore changes on datalines */
/* This speeds up simulation! */
fmSetPortIdleById(inID);
fmSetPortIdleById(outID);
fmSetPortIdleById(abortID);
fmSetPortIdleById(addressID);
fmSetPortIdleById(sizeID);
fmSetPortIdleById(weID);
fmSetPortStateById (ackBundleID, 0);
fmSetPortStateById (ackInID, 0);
fmSetPortStateById (reqOutID, 0);
fmSetPortStateById (reqAbortID, 0);
fmSetPortStateById (abortID, UNDEF);
fmSetPortStatesById (outID,31,0, undefWord);
break;

case FMSIMPHASE:
  /* Simulation is Running */
  if (DEBUGLEVEL > 5) fprintf(stderr, "\n[Memory] In Simphase\n");
  switch (currentState) {

    /* Main State Machine */
  case AWAIT_ACCESS:
    /* Wait for a memory access */
    if (fmGetPortStateById(reqBundleID) == 1 && !pendingReturnsToZero ){
      fmSetPortStateById(ackBundleID,1);
      pendingReturnsToZero|= BUNDLE;   /* Is safe as ack will be only reset
            AFTER req is low i.e. it was seen going high */
      currentState = PENDING_RETURNS_TO_ZERO;
      currentAccessPending = 1;
    }
    if (pendingReturnsToZero) returnToZero();
    break;

  case PENDING_WRITE:
    /* Waiting for Write to complete */
    if (fmGetPortStateById(reqInID) == 1) {
      fmSetPortStateById(ackInID,1);
      pendingReturnsToZero|= IN; /* Is safe as ack will be only reset AFTER
            req is low i.e. it was seen going high */
      currentState = PENDING_RETURNS_TO_ZERO;
    }
```

```
        break;

    case PENDING_RETURNS_TO_ZERO:
        /* Wait for Return-To_Zero Phase to complete before we can start new
             access */
        if (fmGetPortStateById(ackAbortID) == 1) {
          pendingReturnsToZero|=ABORT;
        }
        if (fmGetPortStateById(ackOutID) == 1) {
          pendingReturnsToZero|=OUT;
        }
        if (pendingReturnsToZero){
          returnToZero();
        }
        else if (!currentAccessPending) {
          /* Make sure all signals have actually returned to zero ! */
          /* All input signals were already observed to be low if we arrive here */
          if (   !fmGetPortStateById(ackBundleID) &&
                 !fmGetPortStateById(ackInID) &&
                 !fmGetPortStateById(reqAbortID) &&
                 !fmGetPortStateById(reqOutID)
                 ){
            currentState = AWAIT_ACCESS;
          }
          else {
            /* Make sure we get scheduled again! We ignore changes on data inputs
                 */
            fmScheduleEvent(fmevalelement, (fmCurrentTime() + 1.0),0,0);
          }
        }
        break;

    default:
        /* Sanity Check */
        fprintf(stderr, "\n[Memory] FATAL ERROR. Undefined state reached !\
             nBye !\n");
        exit (1);
        break;
    }
    break;
  }
}

/* Wait for all information about memory access to arrive */
void latchMemBundle() {
  int *tmp=NULL;
  /* Read Data values */
  tmp = fmGetPortStatesById(addressID, 31,0); /* Using it inside the function call
       would lead to memory leak ! */
  currentAddr = arrayToInt(tmp, 32);
  fmFree(tmp);
  we = fmGetPortStateById(weID);
```

```
  tmp = fmGetPortStatesById(sizeID, 1,0); /* Using it inside the function call would
        lead to memory leak ! */
  size = arrayToInt(tmp, 2);
  fmFree(tmp);

  if (DEBUGLEVEL) fprintf(stderr, "[Memory] Address received: 0x%08X   we: %x
       size: 0x%01X\n", currentAddr, we, size);

  if (we==0) { /* WRITE */
    currentState = PENDING_WRITE;
    fmScheduleEvent(fmevalelement, (fmCurrentTime() + 1.0),0,0); /* To avoid
         missing the change on reqIn, if it has already happened !*/
  }
  else {          /* READ */
    /* Read from memory model */
    data = read_phys_mem(currentAddr, size);
    tmp = intToArray(data, 32);
    if (DEBUGLEVEL) fprintf(stderr, "[Memory] Read 0x%08X. \n", data);
    if (physMemoryErrorStatus) {
      fprintf (stderr ,"[Memory] External Abort on address 0x%08X !\n",
           currentAddr);
      physMemoryErrorStatus = 0;
      fmSetPortStateById(abortID, 1);
      fmSetPortStateById(reqAbortID, 1);
    }
    else {
      /* Start handshake to return read data */
      fmSetPortStatesById(outID, 31, 0, tmp);
      fmSetPortStateById(reqOutID, 1); /* DELAY Guarantees that change will
           occur AFTER data is stable!!!! */
    }
    /* Read has completed => Close all handshakes */
    currentState = PENDING_RETURNS_TO_ZERO;
    fmFree(tmp);
  }
}

/* Wait for data which is going to be written to memory */
void latchWriteData() {
  int *tmp=NULL;
  /* Read Data values */
  tmp = fmGetPortStatesById(inID, 31,0); /* Using it inside the function call would
       lead to memory leak ! */
  data = arrayToInt(tmp, 32);
  fmFree(tmp);
  /* Write to memory model */
  write_phys_mem(currentAddr, data, size);
  if (physMemoryErrorStatus){
    fprintf (stderr ,"[Memory] External abort on write to address 0x%08X\n",
         currentAddr);
  }
  if (DEBUGLEVEL && (!physMemoryErrorStatus)) {
```

```
    fprintf(stderr,"[Memory] Succesfully wrote 0x%08X to address 0x%08X\n",
        data, currentAddr);
  }
  fmSetPortStateById(reqAbortID, 1);
  fmSetPortStateById(abortID, (physMemoryErrorStatus ? 1 : 0));
}


/* Wait for all handshakes to complete */
void returnToZero(){

  if (pendingReturnsToZero & BUNDLE) {
    if (!fmGetPortStateById(reqBundleID)) {
      latchMemBundle(); /* Broad data validity to be safe */
      fmSetPortStateById(ackBundleID, 0);
      pendingReturnsToZero&=~BUNDLE; /* Clear flag */
    }
  }

  if (pendingReturnsToZero & OUT) {
    if (fmGetPortStateById(ackOutID) == 1) {
      fmSetPortStateById(reqOutID, 0);
      pendingReturnsToZero&=~OUT; /* Clear flag */
      currentAccessPending = 0;
    }
  }

  if (pendingReturnsToZero & IN) {
    if (!fmGetPortStateById(reqInID)) {
      latchWriteData();
      fmSetPortStateById(ackInID, 0);
      pendingReturnsToZero&=~IN; /* Clear flag */
    }
  }

  if (pendingReturnsToZero & ABORT) {
    if (fmGetPortStateById(ackAbortID) == 1) {
      fmSetPortStateById(reqAbortID, 0);
      pendingReturnsToZero&=~ABORT; /* Clear flag */
      currentAccessPending = 0; /*access finihsed */
    }
  }
}
```

Listing A.1: ADFMI code example: Memory Interface

# Appendix B

# Performance Architecture: Address Translation Unit

This appendix contains the Balsa source code for the entire Translate component. Parts of the component were used in chapter 4 to illustrate the TLB implementation. The listing shows how the parts interact and serves as an example of a non-trivial Balsa procedure.

```
(-- perfTranslate.balsa

    Performance Architecture Address Translation Component
    Includes TLB and table walking hardware

    v1.0

    24/08/2001  ftw      final  version
    11/07/2001  ftw      initial  version

--)


import [spaExternalTypes]
import [cp15Types]
import [mmuInternalTypes]
import [implementationSpecificDefines]

procedure translatePerf (
  input virtBundle : memAccess;
  input cpVars : MMUVars;
  input isData : memIorD;

  output descriptorOut : 32 bits;
  output domainOut : 4 bits;
```

```
  output pageTypeOut : PageType;

  -- cp15 Interface
  sync cpRegLock;
  output flushedFlags : TlbFlushFlags;
  sync cpRegRelease;

  -- Memory Interface
  output descFetch : memAccess;
  input descriptor : 32 bits;
  sync extAbort;

  -- Abort source
  output abortOut : AbortSource
)

is

  variable tlbHit : bit

  type TlbEntry is record
    tag : 21 bits;
    isValid : bit;
    descriptor : 32 bits;
    tableType : TableType;
    pageType : PageType;
    domain : 4 bits
  end

  constant TLB_ENTRIES = 7
  -- TLB will have ONE MORE entries tham in this constant (indexing starts
     with 0 !)

  variable tlb : array 0..TLB_ENTRIES of TlbEntry
  variable tlbIndex : (log TLB_ENTRIES) bits
  constant TLB_ADDR_UPPER_BIT = 10 + (log (TLB_ENTRIES/2)) - 1
  constant TLB_INDEX_MSB = (log TLB_ENTRIES) - 1

  variable wasAborted : bit
  variable descriptorVar : 32 bits
  variable tableTypeVar : TableType

  type b1Array is array 1 of bit

      shared noValidDescriptor
      is
      begin
        domainOut <- 0
        || pageTypeOut <- undef
        || descriptorOut <- 0
      end

begin
```

```
for || i in 0.. TLB_ENTRIES then
    tlb [i ]. isValid :=0
end;

loop
    select  virtBundle, cpVars, isData then
        local

          shared flushTLB
          is
          begin
                for || i in 0.. TLB_ENTRIES then
                        if (#cpVars.tlbFlushFlags[#i[TLB_INDEX_MSB]])
                             = 1 then
                                tlb [i ]. isValid :=0
                        end
                end
                ||
                -- Signal back to cp15 that TLB entries were flushed
                [
                        sync cpRegLock;
                        flushedFlags <- cpVars.tlbFlushFlags;
                        sync cpRegRelease
                ]
          end



          ----------------------------------------------------------------

          -- Do TlbLookup and set tlrOut on hit
          ----------------------------------------------------------------

           shared tlbLookup
           is
           begin
              if ( tlb [tlbIndex]. tag = (#virtBundle.Address[11..31] as 21 bits)
                  and tlb[tlbIndex]. isValid ) then
                      tlbHit:= 1
                      ||descriptorOut <- tlb[tlbIndex].descriptor
                      ||domainOut <- tlb[tlbIndex].domain
                      ||pageTypeOut <- tlb[tlbIndex].pageType
                      ||abortOut <- noAbort
                 else
                      tlbHit:= 0
                 end

           end


          ----------------------------------------------------------------

          -- get the first level descriptor
```

```
-----------------------------------------------------------------

shared getFld
is
begin

 -- compute FLD address

     descFetch <- {read, (b2Array{0,0} @ #virtBundle.Address
           [20..31] @ #cpVars.tableBase as memAddress), word, 0,0,0}
 || select descriptor then
           [descriptorVar := descriptor ; descriptorVar := descriptor ]
       || tableTypeVar := (#descriptor[0..1] as TableType)
       || if (#descriptor [0..1] as TableType) = undef then --
           translation Abort
              wasAborted:=1
           || abortOut <- FLD
           || noValidDescriptor()
          else
              wasAborted:=0
           || domainOut <- (#descriptor[5..8] as 4 bits)
           ||
              [
                tlb [tlbIndex]. tag := (#virtBundle.Address[11..31] as
                      21 bits)
              || tlb [tlbIndex]. domain:= (#descriptor[5..8] as 4 bits)
              || tlb [tlbIndex]. tableType:=(#descriptor[0..1] as
                      TableType)
              ]
           || if (#descriptor [0..1] as TableType) = section then
                   tlb [tlbIndex]. pageType:=undef
                || tlb [tlbIndex]. descriptor:=descriptor
                || tlb [tlbIndex]. isValid:=1
                || descriptorOut <- descriptor
                || pageTypeOut <- section
                || abortOut <- noAbort
              end
          end

      | extAbort then
          abortOut <- extFLD
       || noValidDescriptor()
       || wasAborted:=1
     end
end


-----------------------------------------------------------------

-- get the second level descriptor
-----------------------------------------------------------------

shared getSld
```

```
is
  variable sldAddr : memAddress
begin
    if tableTypeVar = coarse then
        sldAddr:= (b2Array{0,0} @ #virtBundle.Address[12..19] @
                #descriptorVar[10..31] as memAddress)
    else
        sldAddr:= (b2Array{0,0} @ #virtBundle.Address[10..19] @
                #descriptorVar[12..31] as memAddress)
    end;

    descFetch <- {read,sldAddr , word,0,0,0}
    ||
    select descriptor then
        descriptorOut <- descriptor
      || pageTypeOut <- (#descriptor[0..1] as PageType)
      || if ((#descriptor [0..1] as PageType) = undef) then
            wasAborted :=1
          || abortOut <- SLD
        else
            wasAborted:=0
            || abortOut <- noAbort
            ||
            [
              -- All other TLB fields have already been filled by
                    getFld !
              tlb[tlbIndex].descriptor:=descriptor
              || tlb[tlbIndex].pageType:=(#descriptor[0..1] as
                    PageType)
              || tlb[tlbIndex].isValid:=1
            ]
        end

    |      extAbort then
        wasAborted:=1
        || noValidDescriptor()
        || abortOut <- extSLD
    end
  end

----------------------------------------------------

-- MAIN Component body
----------------------------------------------------

begin
  if (cpVars.tlbFlushFlags /= 0) then
        -- Flush TLB entries
        flushTLB()
  end;

  if cpVars.mmuEnable then
```

```
                tlbIndex:=(#virtBundle.Address[10..TLB_ADDR_UPPER_BIT]
                    @ #isData as (log TLB_ENTRIES) bits);
                tlbLookup();

                if tlbHit = 0 then
                    -- do Table Walking
                    getFld();
                    if (wasAborted = 0 and ((#descriptorVar[0..1] as
                        TableType) /= section)) then
                        -- do sld lookup
                        getSld()
                    end
                end
            else
                continue -- don't do anything when mmu is disabled
            end
        end
      end
    end
end
```

Listing B.1: Performance Architecture: Translate block