

DIPLOMARBEIT

AMBA4SPEAR2: An AMBA Extension Module for the SPEAR2 Processor Core

ausgeführt am Institut für
Technische Informatik, Embedded Computing Systems Group
Technische Universität Wien

unter der Anleitung von
Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
und
Univ.Ass. Dipl.-Ing. Dr.techn. Martin Delvai

von

Josef Mosser
Matr.-Nr. 0126655
Bruggen 20
9761 Greifenburg

Wien, 6. März 2008

Acknowledgements

I want to thank my parents and my grandparents who offered me the possibility to study computer science at the technical university of Vienna. They supported me on many occasions and also offered financial help.

My thanks also go to Martin Delvai for supervising my thesis, offering me help whenever I needed it. I would also like to thank him, Andreas Steininger and Heidemarie Rice for proofreading my thesis.

Kurzfassung

Die immer mächtiger und kostengünstiger werdenden FPGAs haben den Entwurf digitaler Schaltungen revolutioniert - zum einen können ganz maßgeschneiderte System in einem einzigen Chip integriert werden zum anderen haben nun auch klein- und mittelständische Unternehmen die Möglichkeit ihre eigenen "ICs" zu bauen. Diesem Trend Rechnung tragend werden einerseits immer mehr vorgefertigte und getestete IP-Cores angeboten und andererseits bieten Werkzeughersteller sogenannte "Frameworks" an, um solche Systeme besser assemblieren zu können. Um verschiedene IP-Cores untereinander bzw. mit der selbstentworfenen Hardware zu verbinden, bedarf es einer normierten Schnittstelle oder eines normierten Bussystems. Im Rahmen dieser Diplomarbeit werden die gängigsten Bussysteme untersucht und miteinander verglichen und anschließend ein Bussystem, der AMBA Bus, näher vorgestellt. Im praktischen Teil der Diplomarbeit wurde für den am Institut entwickelten Prozessorkern SPEAR2 ein AMBA Interface entworfen und speziell auf letzteren zugeschnitten. Abschließend wurde die korrekte Funktionalität durch Anbinden verschiedener IP Module (SPI Interface, UART, etc.) an den Prozessor getestet und verifiziert.

Abstract

FPGAs are getting more powerful and more economical, leading to a revolution in designing digital circuits - on the one hand whole custom made systems can be integrated on a single chip, on the other hand small and medium-sized enterprises have the possibility to design their own ICs. Driven by this trend, more and more predesigned and pretested IP cores are available and the tool suppliers now offer so called frameworks to help assembling those systems. To connect different IP cores and/or its own custom hardware, standardized interfaces or standardized bus systems are needed. Within the scope of this thesis, common bus systems will be analyzed and compared followed by a more detailed description of one bus system: the AMBA bus. In the practical part of this thesis, an AMBA interface for the already existing SPEAR2 processor developed by our institute will be developed. Finally, the functionality of this new interface will be tested and verified by connecting IP cores (SPI Interface, UART, etc.) to the SPEAR2.

Contents

1	Introduction	1
2	State of the Art bus systems	4
2.1	Different types of integrating components into a system	4
2.1.1	Memory-mapped components	4
2.1.2	Port-mapped components	5
2.1.3	Shared Memory components	6
2.2	Different bus systems	8
2.2.1	CoreConnect	8
2.2.2	Wishbone	11
2.2.3	System Interconnect Fabric	15
2.3	Comparison of the State of the Art bus systems	17
2.4	Conclusion of the State of the Art bus systems	20
3	AMBA 2.0	21
3.1	Overview of the AMBA 2.0 bus system	21
3.2	AMBA 2.0 architecture	22
3.3	Features of AMBA 2.0	26
3.4	Traps of AMBA 2.0	27
3.5	Simple AHB/ASB data transfers	27
3.6	Complex AHB/ASB data transfers	29
3.6.1	Burst data transfers	29
3.6.2	Retry (or split) data transfers	32
3.7	APB data transfers	33
3.8	Usage of AMBA 2.0	35

4	AMBA Extension Module for SPEAR2	36
4.1	AMBA and the GRLIB	36
4.1.1	What is GRLIB?	36
4.1.2	GRLIB and its extensions in regard to AMBA 2.0 . . .	38
4.1.3	GRLIB and its address space	39
4.2	SPEAR2 Extension Modules	42
4.2.1	Overview of the SPEAR2 extension concept	42
4.2.2	Limitations of the SPEAR2 extension concept	44
4.3	AMBA integration approaches for SPEAR2	44
4.3.1	First approach	45
4.3.2	Limitations of the first approach	51
4.3.3	Second approach	52
4.3.4	Integration of the second approach into SPEAR2 . . .	54
4.3.5	Two different state machines	55
4.3.6	Summary of the whole AMBA Extension Module inte- gration	56
5	Results and Experiments	58
5.1	Integrating the AMBA Extension Module into the SPEAR2 .	58
5.1.1	Interfaces of the AMBA Extension Modules	58
5.1.2	Integrating the new Extension Modules	60
5.1.3	Transparent mode transfer and the address space . . .	65
5.2	Comparison of SPEAR2 and SPEAR2 with AMBA	66
5.2.1	Setup of the experiment environment	66
5.2.2	Result of the experiment	67
5.2.3	Summary of the experiment	68
	Conclusion	70
A	Signal description	72
A.1	Shared memory	72
A.2	AMBA state machine	73
A.3	AMBA Extension Module	76

B	Integration code	79
B.1	Internal signals	79
B.2	AMBA components integration	80
B.3	SPEAR2 code modification	83

List of Figures

2.1	schematics of a memory-mapped system	5
2.2	schematics of a port-mapped system	6
2.3	schematics of a MESI system	7
2.4	CoreConnect bus system schematic[11]	9
2.5	Wishbone on a SoC example	13
2.6	System Interconnect Fabric on a FPGA example[2]	16
3.1	typical AMBA system[4]	23
3.2	AMBA multiplexer interconnect	24
3.3	AHB master requests bus[4]	28
3.4	example of AHB data transfer[4]	28
3.5	example of AHB data transfer with stall[4]	29
3.6	difference between wrapping burst and incremental burst . . .	31
3.7	example of AHB incremental burst data transfer with stall[4] .	32
3.8	example of a AHB retry data transfer[4]	33
3.9	APB state diagram[4]	34
3.10	APB write transfer[4]	34
3.11	APB read transfer[4]	35
4.1	GRLIB interconnect[8]	40
4.2	Plug & Play information register layout[8]	41
4.3	schematics of the AMBA state machine	45
4.4	schematic of AMBA Extension Module - first approach	50
4.5	schematic of AMBA extension of SPEAR2	51

5.1	Interconnect of AMBA modules	59
5.2	Internal structure of a SPEAR2 with AMBA Extension Module	61

List of Tables

2.1	Comparison of state of the art bus systems	18
3.1	Comparison of burst modes	30
5.1	Result of the experiment	67

Chapter 1

Introduction

In these days the complexity of digital circuits is rising dramatically fast. Driving forces for this trend are the introduction of new technologies such as FPGAs, higher integration densities and resulting from that lower costs[10]. In contrast to that, high level design tools cannot keep up with the technological progress. This results in the so called “productivity” gap as mentioned in the *International Rechnology Roadmap for Semiconductors 1999*.

To reduce the productivity gap, new approaches for faster and less error prone design methods were needed: the birth of the IP based design. The IP based design uses predesigned function blocks like UARTs, processors, memories ... and put them together on a chip. The big advantage of this method is the breakdown of the complexity since the whole design can be broken down into smaller sub designs. Another advantage is that those components can be reused or, if the component is not available in the company’s own library, can be bought from a 3rd party vendor that specialises on IP component design. This gives the advantage that the bought component is functioning and far less error prone than someone’s own design written under time pressure since the 3rd party vendor makes it’s living by selling functioning components. Another side effect of this design method is the time saving in the coding phase and thus this time can be used to enhance the design.

But this step alone is not enough because the blocks are not connected. So new interconnection methods that take general purpose communication

systems into consideration were needed: the bus systems.

Now having these two fundamental new design approaches, the limitation of the new idea was quickly reached. There is no such thing as a general purpose communication system that is good in every field of application. Having reached that conclusion, a differentiation of the bus systems was introduced. This differentiation takes different properties of the bus systems into consideration and categorizes the bus system based on those properties.

The most common of those differentiations is the performance based categorization: high performance bus systems versus low to middle range performance bus systems. Another reason why this property is willingly chosen is because of the dependency of a lot of other properties on this property. In most cases the other properties like overhead and complexity are dependent on the performance of the bus. Normally a high performance bus system needs a higher overhead and a more complex structure to ensure high performance, low latency and good bus utilization when lots of components use the bus. In contrast to high performance, low and middle range performance components normally do not need such complex structures, because the components are not so fast as necessary to utilize the bus to its fullest.

Today there are quite a few bus systems to choose among. The range of those bus systems cover error prone bus systems to low latency bus systems, specialized bus systems to common bus systems, proprietary bus systems to open bus systems.

The choice of the bus system should also be influenced by its popularity and the available supply of 3rd party components.

This thesis is about bus systems with a focus on the AMBA bus and its implementation in the SPEAR2 project.

In Chapter 2 an overview of the state of the art bus systems will be given. Three different bus systems will be described where every bus system covers another area of application. The AMBA 2.0 bus system will be described in Chapter 3. This description will be more detailed than in the one in Chapter 2 since the AMBA 2.0 bus will be needed in the 4th Chapter, where the

integration of the SPEAR2 processor core into an AMBA 2.0 communication network will be presented. Having presented different design approaches for integration the SPEAR2 into an AMBA 2.0 network with all its limitations and solutions, the results and experiments of the integration will be presented. In Chapter 5 a short summary of the whole thesis will be given, pointing out the most important aspects of this thesis. Finally a short conclusion will sum up everything and point out the most important findings of this thesis.

Chapter 2

State of the Art bus systems

This chapter is about state of the art bus systems and different ways of integrating components. After presenting different methods of tying components to the existing design, selected state of the art bus systems will be described with a following comparison of those. Since there are lots of different bus systems, three representative bus systems were chosen.

2.1 Different types of integrating components into a system

Knowing that IP based designs are a way to reduce complexity, design patterns for tying components to the core are needed. Since different needs cannot be solved with one pattern in a satisfying way, this Section will cover the most common ones and describe their characteristics.

2.1.1 Memory-mapped components

Memory-mapped components use the same data and address bus as the memory. This means, that for the CPU there is no difference between writing (or reading) to the memory or writing (or reading) to a component[7, 15] (see Figure 2.1 for a schematic).

The advantage of this method is its practically non existing complexity.

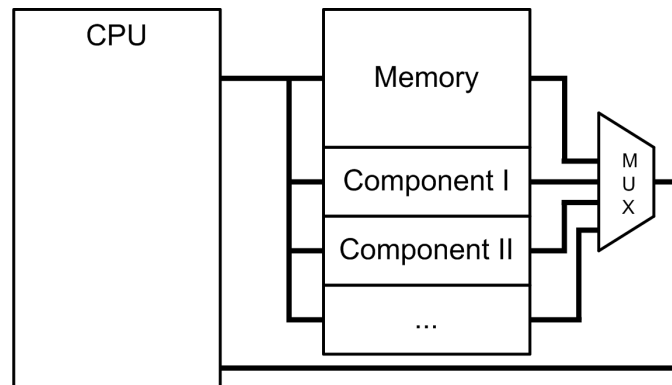


Figure 2.1: schematics of a memory-mapped system

Besides that, it is transparent for software, too. The CPU needs only an additional decoder to join the response of the memory and the memory mapped components. The CPU applies the same address and data bus to both, memory and memory mapped components. The response of the components and the memory is unified by a simple MUX. In this way the CPU gets only a single response. Besides that, all memory access methods are also available for memory-mapped components (address with an offset ...).

The disadvantage of this method is the address space that is wasted since no data can be memory stored in those address spaces. In modern 32 bit CPU's or higher ones this does not matter, since the address space is big enough. In 8 bit or 16 bit CPU's this matters, since the address space is already limited. Another disadvantage occurs when using slow components and fast memories. In this constellation, the slow components block the CPU and caching becomes tricky.

2.1.2 Port-mapped components

Port-mapped components have a dedicated address space accomplished by using an extra bus dedicated to the components or by using an extra component - signal that indicates if a component is meant or the memory[7, 18, 15] (see Figure 2.2 for a schematic). To write to (or read from) a component, special dedicated CPU instructions are needed.

The advantage of port-mapped components is the distinction between

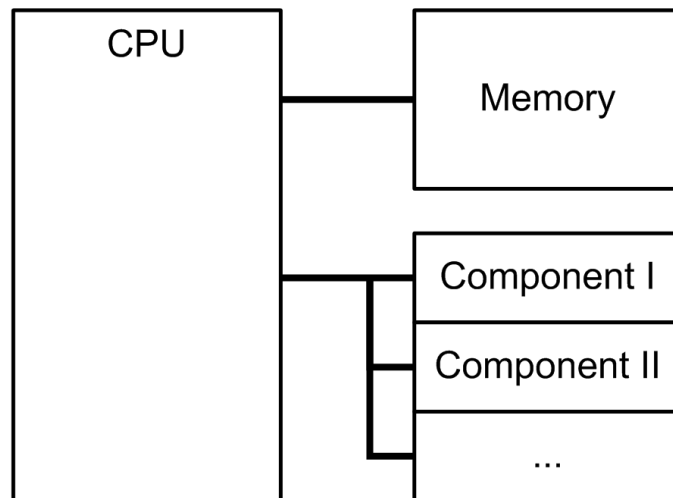


Figure 2.2: schematics of a port-mapped system

memory address space and component address space. This is an enormous improvement for CPU's with limited addressing capabilities, because now the whole address space can be used for accessing memory. This integration method is more secure, too. For example it is not possible for a stack to grow outside its own memory space and push data into a components address space. Port-mapped components also produce more understandable code since own instructions are needed for using those components.

The main disadvantage of this integration type is its overhead. It results in a more complex structure of the CPU, because of the additional data and address bus, and it also needs its own decoder logic. Another disadvantage is the larger ISA ¹ because own instructions are needed to write to port-mapped components.

2.1.3 Shared Memory components

Shared Memory components share, like the name suggests, a memory where every component has access to. The advantage of this integration method is that nearly every component can be connected using this method (i.e. two CPUs)[7, 18, 10]. It also supports the sharing of large amounts of data

¹Instruction Set Architecture

between components. But in order to function properly a coordinated access is required. A method for coordinating the access to the shared memory is the MESI protocol.

The MESI protocol was invented at the Illinious University and thus also known as Illinois protocol. It is used to assure memory (or cache) coherency. It uses status flags for every memory word to define its current status and a shared memory that holds the data that every connected component has access to. The four states that a memory word can have are: **M**odified, **E**xclusive, **S**hared or **I**nvalid. Another precondition that has to be met is that every component can “hear” what another component wants to read[7, 18, 15] (and thus a shared bus is often used for connecting all components; see Figure 2.3 for a schematic).

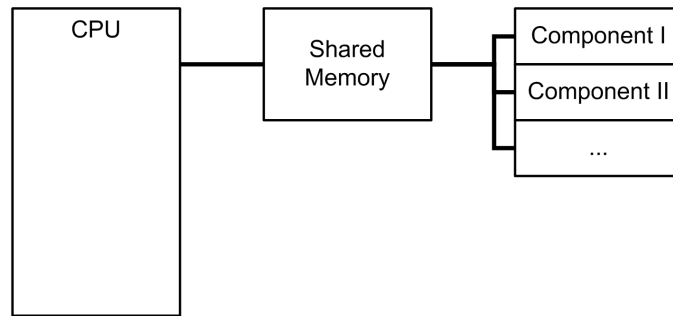


Figure 2.3: schematics of a MESI system

The function of the protocol is simple: If component X wants to use the data at a given address A, it signals this to the bus. In the next cycle, the shared memory sends the data on the bus, component X receives it and marks it with an E since no other component has the data at address A. Sometime later, component Y wants the same data, too. It also sends it request to the bus. The shared memory sends the desired data back, but since X already got the data, X signals that it has the data, too. So now component X and Y mark the data with an S since both of them hold the data. After some time component X finishes its manipulation of the data and marks it now with an M since it has another value than in component Y. At the same time is tells every other component that is has changed the value and thus every other component marks the data at the address A with an I. Now, if

Y wants to use the data, it has to get it again since it is marked with an I. So Y sends the request on the bus again. Hearing the request of Y, X sends a retry back, signalling that Y has to send the request again. At the same time X sends the new data to the shared memory and marks the data it holds with an E (since it is again the only component that has the current value of address A). Now Y sends its request again, this time receiving the new data from the shared memory. X signals again that it has the value too, so both components mark the data with an S again, and so on.

The drawback of this integration method is its memory overhead. For every data word additional 2 bits of status information have to be stored. Thus the MESI protocol is mainly used in caches. In addition to that, the component always has to listen to the bus and check / update its memory if an invalid or a data request appears. Thus power saving techniques are not as easy to implement as in other integration methods.

2.2 Different bus systems

After reading about different integration approaches, this Section will discuss different State of the Art bus systems. All below mentioned bus systems can be implemented using one of the above mentioned integration methods.

2.2.1 CoreConnect

The CoreConnect bus system[12] was invented by IBM². Its original purpose was to ensure communication between macros in IBM Blue LogicTM design. Soon realizing, that those macros and the bus system are powerful design patterns, IBM expanded their use. Today it is a powerful bus system with lots of components available for it, either provided by IBM or 3rd party vendors.

²International Business Machines

CoreConnect buses

The CoreConnect architecture provides three different buses to connect user logic, cores and library macros:

- Processor Local Bus (PLB)
- On-Chip Peripheral Bus (OPB)
- Device Control Register (DCR) Bus

For a visualization of the schematic of the CoreConnect bus system, see Figure 2.4

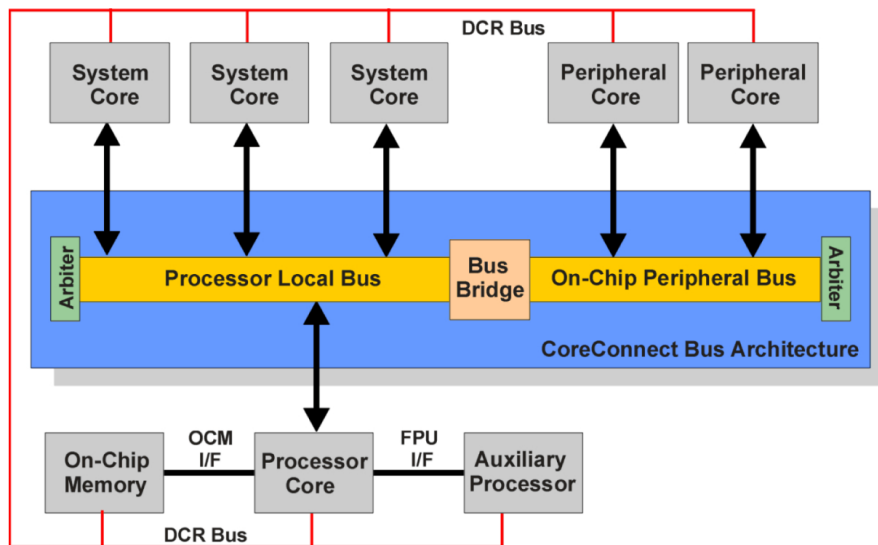


Figure 2.4: CoreConnect bus system schematic[11]

Of these three buses only two are used for transferring data between connected components: the Processor Local Bus (PLB) and the On-Chip Peripheral Bus (OPB)[12]. As the name suggests, the PLB is designed for high bandwidth, low latency and highly integrated components like processors, external memory controllers and DMA³ controllers. The OPB is used to prevent bottlenecks implied by high latency or low bandwidth components. So the primary usage of this bus is for I/O - components like serial ports,

³Direct Memory Access

timers, UARTs⁴ or parallel ports. To ensure an ordered access to the bus, every PLB and every OPB bus needs its own arbiter (see Chapter 3.2 for an example of an arbiter) that controls the usage of the bus. To connect those two buses, a so called OPB bridge is needed. Its function is simple: to act as a PLB slave and as an OPB master and establish a connection between those two “virtual” components.

The third of the above mentioned buses is the Device Control Register (DCR) Bus[12]. It is used for transferring status and configuration data between the connected components. Since in most cases the status and configuration data does not need to be accessed in low latency context, there is not a distinction between high performance and low performance DCR. Every component (either PLB or OPB) uses the same DCR bus.

Features of CoreConnect

After describing the different buses in CoreConnect, the features of CoreConnect will be mentioned next.

The advantages of CoreConnect[11] are:

- fully specified and published CoreConnect specification
- no-fee, no-royalty license
- 3 different CoreConnect version to support a variety of applications (32-, 64- and 128 bit version)
- 2 data buses to prevent bottlenecks: a high speed bus (PLB) and a low speed bus (OPB)
- dedicated bus for status and configuration data (connects all components for status information sharing or ...)

⁴Universal Asynchronous Receiver Transmitter

Traps of CoreConnect

There is no advantage without a disadvantage. The same applies to the CoreConnect bus system, too.

The main disadvantages of CoreConnect[13, 11] are:

- complex bus system resulting in more error prone designs due to its complexity
- different versions of CoreConnect are not compatible (they can only be made compatible by adding a new complexity level to the master that supports transactions)
- every bus (in highest degree usage three buses) has to be managed

Usage of CoreConnect

Since CoreConnect is an invention of IBM, it is used in nearly every PowerPC based application. This also includes Xilinx FPGA's, which uses the PowerPC 440 as its hard processor core.

Summary of CoreConnect

To sum everything up, the CoreConnect bus system is a complex and powerful bus system. It combines speed, high bandwidth and low latencies for applications at the cost of more overhead and more error prone designs due to its complexity.

2.2.2 Wishbone

The main difference between Wishbone⁵ and the above mentioned bus systems is that Wishbone is an open source bus system. No fees or licensing

⁵The term "WISHBONE interconnect" was coined by Wade Peterson of Silicore Corporation. During the initial definition of the scheme he was attempting to find a name that was descriptive of a bi-directional data bus that used either multiplexers or three-state logic. This was solved by forming an interface with separate input and output paths. When these paths are connected to three-state logic it forms a "Y" shaped configuration that resembles a wishbone.

costs emerge from using it and no patent violations emerge[14].

Wishbone was designed to utilize only one general purpose interface. Its design was influenced by three major factors[14]:

- to define a good and reliable bus system
- to provide a common interface specification for structured design methods and large project teams
- to take the idea of traditional system integrating microcomputer buses like PCI⁶ bus or VME⁷ bus and adapt it

Wishbone bus

As mentioned before, Wishbone took the idea of microcomputer buses and thus also provides its advantages[14]:

- a flexible integration solution
- it offers a wide variety of different data bus width and different bus cycles
- components can also be designed by 3rd party companies

The Wishbone architecture does not specifically define how the components are connected. This can be done through a point-to-point connection, through a shared bus (two data paths or a tri-state data path) or through a crossbar switch. Although different interconnects are possible, Wishbone only defines one interface, making it the same access pattern on every interconnect system. Since the most common usage of Wishbone is a shared bus, designers have to be aware that using only one bus can be a bottleneck for the application. If a DMA controller and an UART are connected with a processor core, the whole system will have a high latency and a slow throughput since the UART is in most systems the slowest component of all. Therefore it is quite common that two Wishbone shared bus systems are

⁶Peripheral Component Interconnect

⁷Versa Module Eurocard

used on the same system: one for high performance components, one for low performance components.

For a visualization of an example on employing Wishbone in a SoC, see Figure 2.5 showing a Wishbone system on a SoC using the tri-state shared bus as an interconnect.

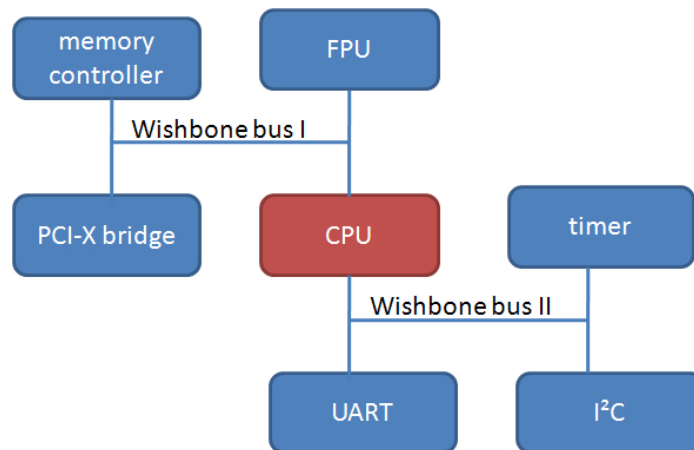


Figure 2.5: Wishbone on a SoC example

Features of Wishbone

Since Wishbone has only got one interface[14], its features differ from the above mentioned CoreConnect. Below, the more important features of Wishbone are listed:

- since it is open source it is absolutely free
- simple and compact bus system resulting in usage of few resources
- supports different interconnect methods: point-to-point, shared bus, switch fabric and crossbar switch
- supports user defined tags for marking the current purpose for data transfer (i.e. data transfer, interrupt vector transfer, parity or error correction bits ...)
- supports structured design approaches since there is only one interface

Traps of Wishbone

As there are advantages of Wishbone, there are also its disadvantages[14]. Below the most important ones are mentioned:

- connecting components with different latency / bandwidth with the same bus may result in a bottleneck
- specification is not as complete as the CoreConnet specification
- specification defines 5 different degrees of implementation:
 - RULE: this has to be implemented as written in the documentation
 - RECOMMENDATION: this should be implemented by the designer but is not a must
 - SUGGESTION: these are optional guidelines for designers
 - PERMISSION: if a RULE is not as specific as needed, the PERMISSION should give the designer some insight to understand why and if he can implement the not strictly specified functionality
 - OBSERVATION: they just remind the designer of already implied facts that can be overseen
- tri-state bus signals are not explicitly forbidden (can result in slower speeds)

Usage of Wishbone

The Wishbone bus system is mainly used in Open Cores components and open source designs. But there is a rising number of 3rd party vendors that modify their design to support Wishbone.

Summary of Wishbone

Wishbone is a simple alternative to CoreConnect, but to gain its simplicity, a number of performance optimizing design patterns were not implemented.

The big advantage is that it is open source and it is patent free. It is sufficient for normal design, but when low latencies and high bandwidth is needed, it is hard to achieve without taking heavy constraints on the desired design.

2.2.3 System Interconnect Fabric

The System Interconnect Fabric is used by Altera and was prior known as Avalon switch fabric[3]. In contrast to the above mentioned bus systems, the System Interconnect Fabric was designed for SOPC⁸-applications. Thus, its implementation in a SOPC-application is done by the SOPC Builder of Altera which connects components that implement the Avalon interface. A speciality of this SOPC optimized bus system is its streaming capability (see next few lines for explanation) besides the standard data transfers.

System Interconnect Fabric bus

As already mentioned, the System Interconnect Fabric supports streaming capability. Streaming is used when a data transfer between two components is needed, where the receiver has to process a continuous data stream. Since a lot of algorithms (for example video encoding / decoding) process data blocks instead of continuous arriving data, the streaming data is queued and the receiver has the possibility to access the data in bursts. The drawback of streaming is the fact that streaming is only available between a sender and a receiver (point-to-point). Therefore two different buses are available to ensure flexibility on connecting components: one streaming bus and a shared bus.

To understand the parallel usage of both buses (streaming bus and shared bus) see Figure 2.6

The setup of the interconnect system differs from application to application since it is done by the SOPC builder. So adding a new component to the application implies using the SOPC builder of Altera[2, 3]. This is a major difference to the above mentioned bus systems.

⁸system-on-a-programmable-chip

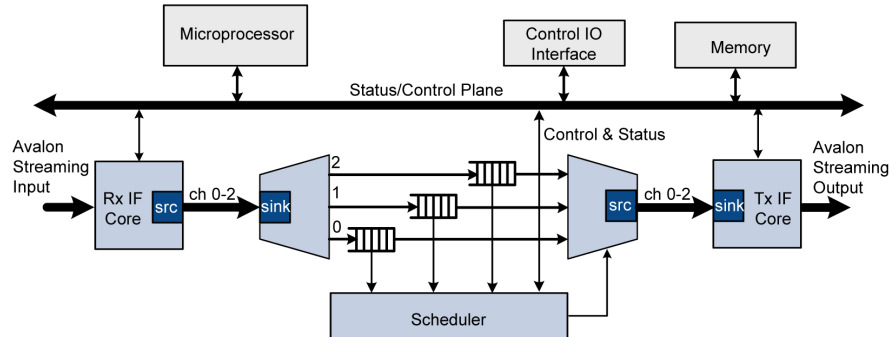


Figure 2.6: System Interconnect Fabric on a FPGA example[2]

Advantages of System Interconnect Fabric

One of the greatest advantages of the System Interconnect Fabric is the optimized implementation of the buses on a FPGA and thus resulting in the usage of only few resources. Other advantages of the Altera bus[2, 1] system are:

- dynamic bus sizing (meaning that the SOPC builder ensures data transfers between components without the developer's interaction, even if the data widths of the components differ)
- separate address, data and control paths (components do not need separate address and data decoding cycles)
- dedicated streaming bus:
 - low latency and high bandwidth
 - multiple channel support with flexible packet interleaving

Traps of System Interconnect Fabric

The optimization has also its price: when using other FPGAs than those from Altera it results in a higher resource usage. Other disadvantages of the Altera bus system[2, 1, 3] are:

- the connection of the components is only reasonably possible with the SOPC builder of Altera

- the shared bus can be a bottleneck since there is no distinction between high speed and low speed shared bus components
- streaming only supports one receiver (no single sender multiple receiver support)
- Licensing of the System Interconnect Fabric is not clearly stated on the Altera homepage

Usage of System Interconnect Fabric

The System Interconnect Fabric is mainly used in Altera FPGA-designs. There are 3rd party components which have the Avalon interface implemented, but the main source for components is the Altera IP center.

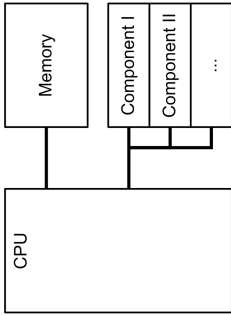
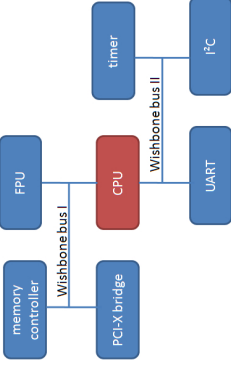
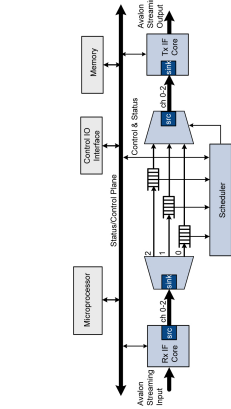
Summary of System Interconnect Fabric

System Interconnect Fabric is an interesting FPGA optimized bus system that has streaming as a feature. Since the licensing is not clearly stated on the homepage, it is hard to tell if this bus is interesting for non Altera designs. A major drawback of the bus system is its major dependency on the SOPC builder.

2.3 Comparison of the State of the Art bus systems

The following table will give a summary of the previously mentioned bus systems. The attributes of the buses that are compared are those which make up the characteristics of the previously mentioned buses and are widely used for comparing.

Table 2.1: Comparison of state of the art bus systems

	CoreConnect	Wishbone	System Interconnect Fabrice
Structure			
Shared bus interconnect	has three different buses: one for high and one for low speed communication one for status / configuration communication	defines only one interface bottleneck if all components use the same bus	two different buses: one bus for normal data communication one bus for streaming
Complexity	average since bus is pipelined high if all buses are used	low since only one interface is defined high if a component uses two buses	average since normal bus is pipelined high if streaming is used to its extent

Continued on Next Page...

Table 2.1 – Continued

	CoreConnect	Wishbone	System Interconnect Fabrice
Overhead	high if all three buses are used average if only one bus is used bridging the two buses results in a higher overhead	low if the minimum is implemented average if tagging feature is used high if a component uses two buses	average when only pipelined bus used high if streaming is used, too
Licensing	no fees are necessary no license is needed	bus system is open source Wishbone can be used commercially	on Altera designs it is free no clear information on other platforms
Integration type	specification is optimized for memory-mapped components	specification leaves integration up to the designer most designs are memory-mapped or/and port-mapped	specification is optimized for memory-mapped components streaming bus uses direct connection between components

2.4 Conclusion of the State of the Art bus systems

The presented state of the art bus systems show that the ideal bus system for every possible field of application does not exist. Every bus system has its pros and cons. It is up to the designer to choose which is the best one for his project. To be able to make a qualified decision, a lot of literature has to be read and compared to find out about the advantages and disadvantages of a system. There are still a lot of other bus systems which are not described in this thesis. Only three different characteristic bus systems have been presented.

In Chapter 3 another state of the art bus will be introduced: the AMBA bus, while in Chapter 4 the implementation adaptations for the SPEAR2 are discussed in detail.

Chapter 3

AMBA 2.0

The AMBA bus has been chosen for the SPEAR2 processor core, because of the many available components and because it is free. This chapter will give a more detailed insight into the AMBA 2.0 bus systems, including the description of different data transfer modes. In contrast to the previous chapter, the bus access control will be described, too. This information is required to understand the limitations of the AMBA 2.0 bus discussed in Chapter 4.

3.1 Overview of the AMBA 2.0 bus system

The AMBA¹ bus architecture was introduced by ARM² Limited in 1995. Four years later, ARM redesigned the AMBA bus architecture and released the AMBA 2.0 specification. Today, the AMBA 2.0 specification is a de-facto standard for 32 bit processors. In 2003, ARM introduced the 3rd generation of the AMBA bus system, the AMBA 3 AXI bus system. In every new generation of AMBA, a new bus was introduced, resulting in four different buses for the AMBA 3 AXI bus system.

AMBA 2.0 defines two interface protocols that cover data intensive processing components and low bandwidth/latency components. The essential requirements for data intensive processing components are a high bandwidth

¹Advanced Microcontroller Bus Architecture

²Advanced RISC Machines

and a low latency whereas slow components like an UART only need a moderate latency and bandwidth. The low bandwidth interface protocol was also designed to assure little power consumption and little resource consumption whereas the focus of the high bandwidth interface protocols was set on data throughput.

Goals of the AMBA 2.0 specifications (and this also applies to all AMBA specifications) are:

- to assure a specification that is technology independent
- to support larger design teams that work on the same project
- to enhance the reusability of already designed components
- to specify a bus system that meets today's and future requirements of designers
- to provide a framework that can break down the complexity of a design
- to provide a bus system that has the right amount of tradeoffs between flexibility and overhead

The AMBA 2.0 bus system can be used without paying royalty fees, which makes the bus system interesting for 3rd party vendors. For example, Infineon uses the AMBA bus system in some of its MIPS³ based SoC applications.

The next section is about the AMBA 2.0 architecture, its mechanism for bus control and its limitations given by the specification.

3.2 AMBA 2.0 architecture

The AMBA 2.0 specification defines 3 buses:

- Advanced High-performance Bus (AHB)
- Advanced System Bus (ASB)

³Microprocessor without Interlocked Pipeline Stages

- Advanced Peripheral Bus (APB)

Between the Advanced High-performance Bus (AHB) and the Advanced System Bus (ASB) there is no big difference from a functional point of view. Both are used for a fixed one stage pipelined data transfer. The main difference between them is that the AHB bus uses 2 data paths (one for reading and one for writing) whereas the ASB bus uses only one bus (and thus implying a “tristate” bus⁴).

In contrast, the Advanced Peripheral Bus (APB) was introduced to prevent bottlenecks when using high and low performance components. The main usage of the APB bus is for low performance peripherals such as UARTs, timers and so on. To combine the high performance bus (or system bus) and the peripheral bus, a bridging component was introduced. This bridging component (like in CoreConnect) is an AHB (or ASB) slave and an APB master at the same time. The APB bus itself is simple due to the fact that it does not use pipelining for data transfer. This yields to a lower resource usage for the APB bus and components.

For a schematic example of a typical AMBA system, see Figure 3.1.

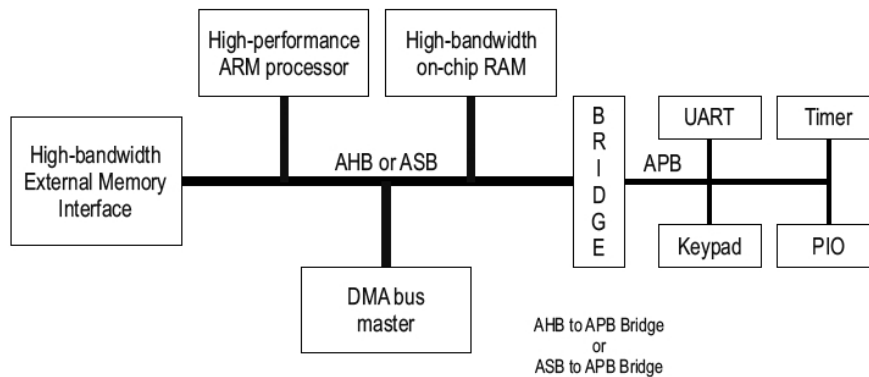


Figure 3.1: typical AMBA system[4]

Besides the components that are connected via the AHB, ASB or APB bus, components for bus access control are needed:

⁴In most cases there are not real tristate interconnect lines on a chip. Today, real tristates are only implemented in I/O's since tristates need powerful drivers for high clock rates

- the arbiter
- the decoder

The AMBA arbiter is responsible for controlling the bus access of the AHB master components since more than one AMBA masters are allowed. On the one hand the AMBA decoder is needed to drive the select signal of the corresponding slave component and on the other hand to filter the responses of the slave components. For a detailed view of the interconnect between the AHB masters, the AHB slaves, the AHB bus arbiter and the AHB bus decoder see Figure 3.2.

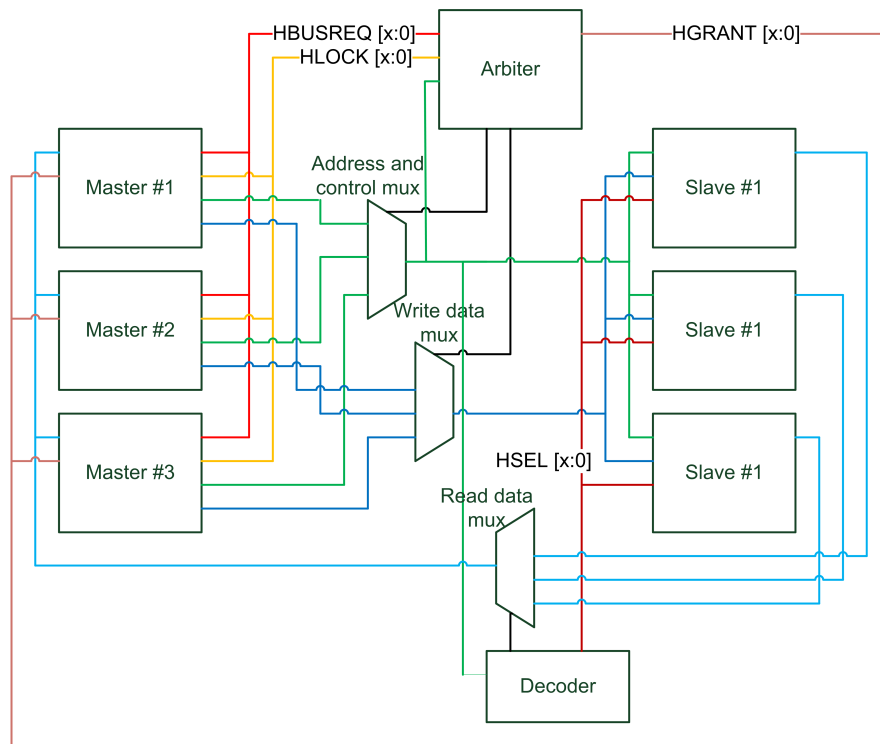


Figure 3.2: AMBA multiplexer interconnect

The access control, performed by the arbiter, is realised through two central multiplexors (one for the control and address signals and one for the write data signals). If a master component wants to use the bus, it has to send a request through its request signal (HBUSREQ_x). The arbiter receives

the request and decides if the master can invoke the bus. The decision is normally made on a priority base or a round robin base (it is up to the designer to decide or to extend the decision algorithms). After the decision has been made, the arbiter signals its decision to the master through the separate grant signal (HGRANTx). If the master gets the bus (HGRANTx is high) it can start its transfer, otherwise (HGRANTx remains low) the master has to wait until it is its turn. Besides that, the arbiter also sets the two multiplexer to pass through the data of the selected master to the slaves.

The arbiter also gets the current address and control signals of the active master. This is a precaution to prevent a component from monopolizing the bus or it can be used for an “intelligent” decision algorithm where those information is used as parameters, too. In most cases the arbiter is just programmed to prevent components from monopolizing the bus (for example, if a master just holds the line without using it thus blocking the bus for transfers of other masters).

The second component needed for bus control, the AMBA decoder, is responsible for selecting the right slave component and routing the right output of the slave components back to the master components. Based on the address, it decides which select signal has to be set. A slave component only responds to AMBA requests if its select signal has been set. If a slave component has been selected, it responds to the AMBA request by sending back control signals and, if a read memory takes place, data too. The AMBA decoder also masks out all other responses from the slave components in case that faulty slave components send data back even though they are not selected.

This however implies that the address space of all components has to be defined a priori and has to be encoded into the AMBA bus system[4]. If a new AMBA slave component has to be added to the existing design, not only the top entity of the design has to change, but also the whole AMBA decoder and arbiter logic has to be changed for the following reasons:

- the multiplexors have to accept another component
- the decoder has to consider another address space

- the decoder has to be extended by another select signal
- the arbiter has to take another component into consideration in its decision algorithm

In Chapter 4 it will be explained how the AMBA module used in the GRLIB was modified to overcome this weakness.

3.3 Features of AMBA 2.0

The previous section already described some features of the AMBA 2.0 bus. To complete the list of the most important features, the following additional features are provided by AMBA 2.0[4], too:

- no royalty fees necessary
- supports burst data transfer for more data throughput
- supports split transaction for a better utilization of the bus
- 2 categories of buses (one for high bandwidth, one for low bandwidth)
- 2 different high bandwidth buses (a tristate and a non tristate bus)
- data width is configurable
- supports multiple masters
- supports flexible address modes (word aligned or word wrap)
- offers a combination of high and low bandwidth bus through a bridge component
- APB bus utilizes only few resources and has a simple interface

3.4 Traps of AMBA 2.0

AMBA 2.0 also has its disadvantages compared to other bus systems:

- complex transfer modes cannot so easily be implemented
- components with different data width cannot so easily be combined
- implementation of all features is not easy (for example split transfer) and thus more error prone
- although they have the same functional basis, the AHB and ASB components cannot be connected on the same bus

This section has presented the architecture, the advantages and the disadvantages of AMBA, the next section will go into more detail. In the following section the AMBA data transfers and their illustrations will be given.

3.5 Simple AHB/ASB data transfers

Every access to the AMBA bus can only be initiated by an AHB (or ASB) master. From now on, only the data transfer of an AHB master will be described (but since the AHB and ASB master are similar, it is also valid for an ASB master). Thus, an AMBA AHB master has to signal the arbiter that it wants to use the bus⁵ (see 3.2). In the next clock cycle, the arbiter can grant the AHB master bus access if the bus is free⁶. Otherwise the AHB master has to wait until the arbiter grants bus access (see Figure 3.3).

After the arbiter has granted access to the bus, the AHB master has to start with the transfer. For this purpose the master has to write the address, the access type (write or read), the transfer type (8 bit, 16 bit, 32 bit, burst mode, normal mode, wrapping mode) and the status information (transfer in progress, stall) to the bus in the next clock cycle⁷. This is also called the address phase.

⁵the signal is called HBUSREQ_x where x is the id of the master

⁶this is indicated by the HGRANT_x signal

⁷see the AMBA 2.0 specification for a complete list of the signal names

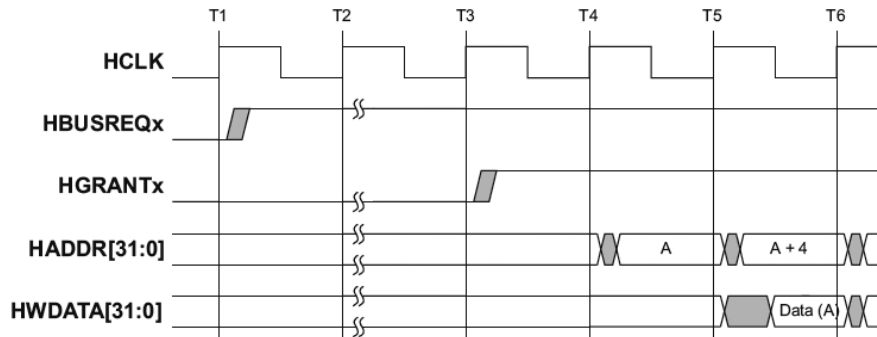


Figure 3.3: AHB master requests bus[4]

In the next clock cycle, the master must provide (or receive) the data on the bus. This is called the data phase. Since address and data phase are separated, the AMBA AHB bus is a fixed length pipelined⁸ bus (see Figure 3.4).

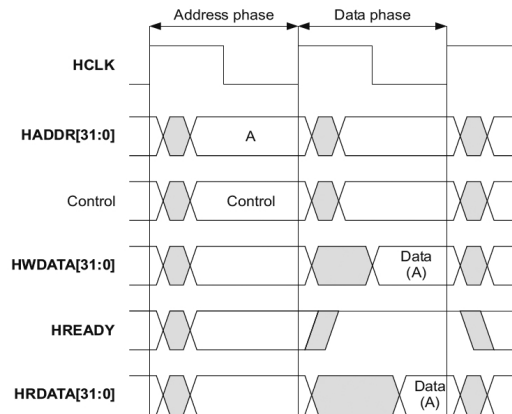


Figure 3.4: example of AHB data transfer[4]

Since it is not easy to write complex components that can keep up with the speed of the bus, AMBA 2.0 uses a method that allows the AHB slave to stall the bus. This is realized with a ready signal⁹. So if a slave sets the

⁸Pipelining means that a data processing task is broken down into smaller subtasks where the output of the current subtask is the input of the next subtask. All of those smaller subtasks are computed concurrent and need one cycle to finish. Pipelining was introduced to utilize the hardware better and to increase the speed of complex systems. The down side of this technique is a higher overhead for preventing data inconsistency.

⁹in the AMBA 2.0 specification the signal has the name HREADY

ready signal to low, the master has to stall its data transfer until the slave sets the ready signal to high again (see Figure 3.5).

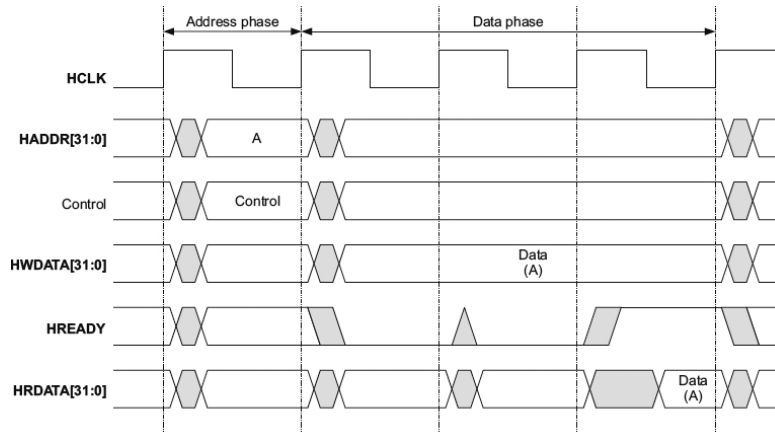


Figure 3.5: example of AHB data transfer with stall[4]

3.6 Complex AHB/ASB data transfers

Besides the simple AHB/ASB data transfers mentioned above, AMBA 2.0 also supports more complex data transfers. This section will cover the more complex data transfers since those are the ones that boost the performance of an AMBA 2.0 bus system.

3.6.1 Burst data transfers

The first more complex data transfer is the burst mode. In contrast to the simple data transfer, the burst mode was designed to transfer data bulks with a predefined length or with no predefined length. The burst mode is available for every simple transfer mode (8 bit data transfer, 16 bit data transfer and 32 bit data transfer). A restriction introduced by the burst mode is the necessity of continuous data. The burst mode must not cross a 1kB address boundary.

AMBA supports two different burst modes:

- incrementing burst

- wrapping burst

The incrementing burst accesses sequential locations and thus every transfer in the burst is just an increment of the previous address by the data width.

The wrapping burst is only available if the length of the burst is specified. In contrast to the incrementing burst, the wrapping burst writes (or reads) only data from a size aligned address space.

To understand the difference of these two burst modes, they will be compared in the following table (Table 3.1), followed by an illustration of the more complicate wrapping burst mode.

	incremental burst	wrapping burst
size of transfer	can be predefined (two, four or eight data junks) undefined burst length are also supported	can only be predefined (two, four or eight data junks)
address of data	address of data starts at given start address and increments with every new data junk until the transfer is finished or the 1k transfer boundary is reached	Address of data starts at given address, but wraps at the higher transfer size aligned start address to the lower transfer size aligned start address (see following example)
complexity of transfer	complexity is low since the next address is the previous address plus the size of a data junk	complexity is high since the transfer size aligned address boundaries have to be taken into consideration
usage of transfer	incremental mode is mainly used for transferring large amounts of data this mode is the more common transfer mode	wrapping mode can be used for efficient data shifting within a given boundary this mode is rarely used and often not implemented

Table 3.1: Comparison of burst modes

Having compared the two different burst modes, an illustration of the burst mode will be given with the help of an example:

Let us take a four beat wrapping burst of word length access (meaning four 4 byte data junks are transferred, making up a 16 byte data transfer) and let

the start address be 0x38. 0x38 is not a 16 byte aligned address. The 16 byte aligned addresses before and after 0x38 are 0x30 and 0x40. Thus, the first 4 byte data is stored at 0x38 and the second 4 byte data is stored at 0x3C. So far there is no difference to the incrementing burst. The third 4 byte data would be stored at 0x40, but since the boundary of the size aligned address space has been reached, it is placed at the beginning of the size aligned address space: 0x30. This is the difference to the incrementing burst. The fourth and last 4 byte data is stored at 0x34, back in correspondence with the incrementing burst. For an illustration of the wrapping mode, see Figure 3.6.

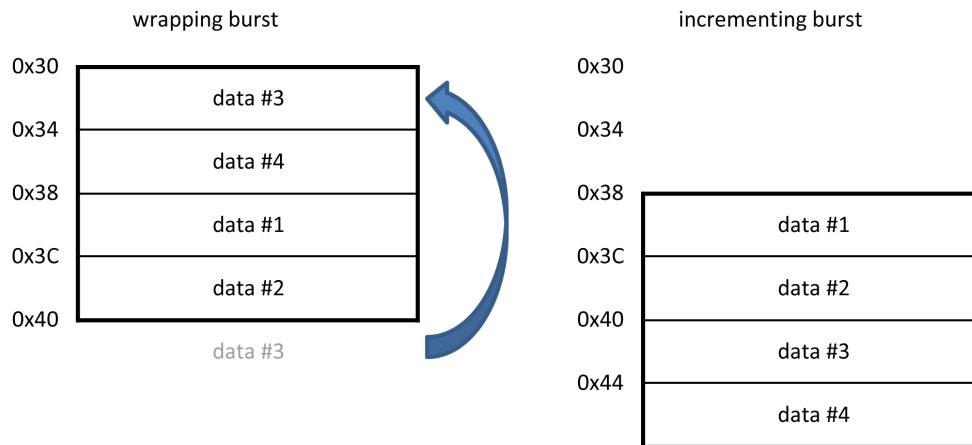


Figure 3.6: difference between wrapping burst and incremental burst

As previously mentioned, the incrementing burst mode is the more common burst mode and thus the following more detailed example will describe an incrementing burst.

To start an incrementing burst mode, the master must simply set the corresponding signal (in the specification called HBURST) and then provide the data in the data phase in every clock cycle. As in the simple data transfer, the slave also has the possibility to stall the data transfer. Thus, the burst mode must be stalled until the slave asserts the HREADY signal to high again (see Figure 3.7).

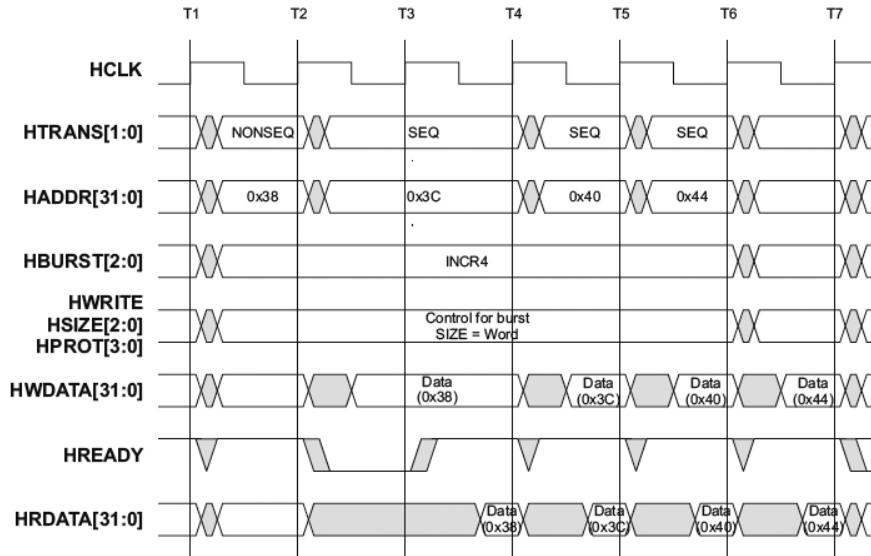


Figure 3.7: example of AHB incremental burst data transfer with stall[4]

3.6.2 Retry (or split) data transfers

The retry data transfer is something special and can only be initiated by a slave component. It stalls the transfer for two more clock cycles and then restarts it again. To do so, the slave has to send a retry response¹⁰ to the master. Then, the master has a time span of 2 clock cycles to prepare the transfer again. For more details, see Figure 3.8.

The retry data transfer is also the basis for the split data transfer. As mentioned in Section 3.3 (Features of AMBA 2.0), the split transaction is used to provide a better bus utilization. If a component *does* know it needs some time before it can provide the data, it can initiate a split transfer, meaning that another master should make its transfer in the meantime and after that, the original master can continue hoping, that the component has the required data now.

¹⁰in the specification, the bus for sending the information back is called HRESP

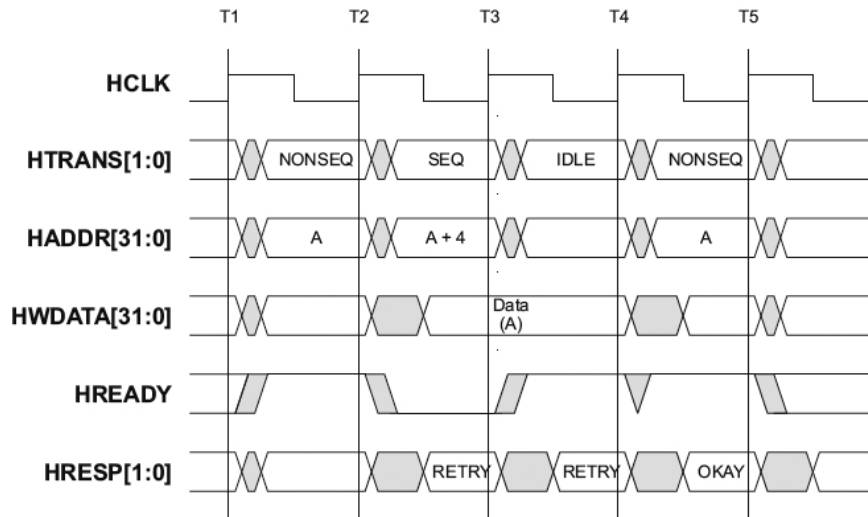


Figure 3.8: example of a AHB retry data transfer[4]

3.7 APB data transfers

In the previous section, AHB/ASB data transfers were explained in more detail. In contrast to those data transfers, the APB transfer differs fundamentally: it is not pipelined. Another difference is its simple interface. The data transfers on the APB bus can be visualized in a simple state diagram (see Figure 3.9).

Since there is only one APB master allowed, the bus arbitration is not needed. In most cases, the bridging component represents the APB master. To initiate a write transfer, the master must write the address and the data on the bus in the same cycle. Since no central decoder unit is defined, the master must also set the select signal of the corresponding slave in the same clock cycle. Now, the signals are held on the bus until the slave component asserts the enable signal¹¹ to high to indicate that the data has been received. After that, the write transfer has been completed and the slave component resets the enable signal to low again (see Figure 3.10).

The read transfer is nearly the same on an APB bus. The signals have to be set up the same way (except the PWRITE signal: it indicates if a write

¹¹in the specification known as PENABLE

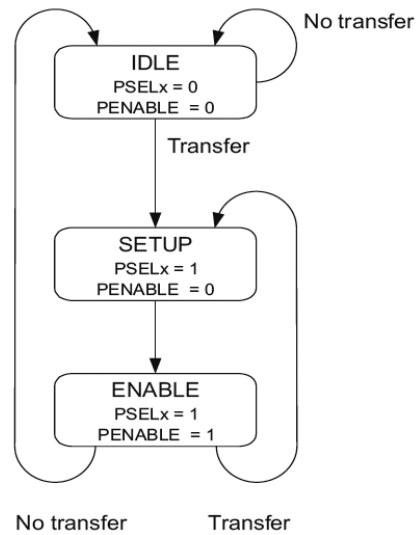


Figure 3.9: APB state diagram[4]

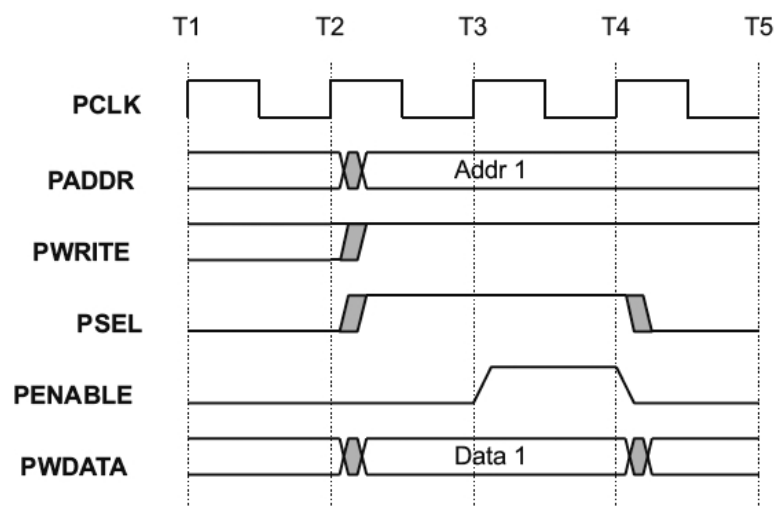


Figure 3.10: APB write transfer[4]

or a read takes place), the only difference is that not the master has to write the data on the bus, but the slave has to provide the data during its assertion of the enable signal to high (see Figure 3.11). This also means that the APB master must always accept the data a slave provides in one cycle, since there is no way of stalling the data transfer for the master.

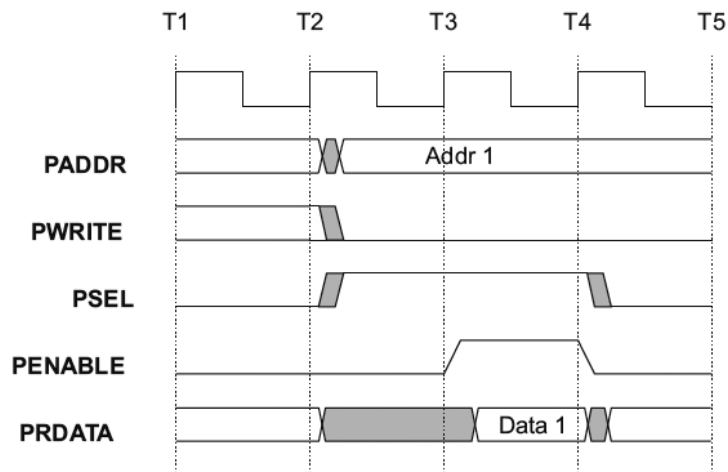


Figure 3.11: APB read transfer^[4]

3.8 Usage of AMBA 2.0

The AMBA 2.0 bus system is used in nearly every ARM design. Besides that, AMBA is also used in some of Infineon's SoC applications (mostly MIPS based). Another implementation of the AMBA bus system is in the GRLIB IP-package of Gaisler Research¹² which uses the LEON3 processor core as a basis. Some companies like EADS Astrium¹³, Ball Aerospace¹⁴ or Vineyard Technologies¹⁵ use the LEON3 processor for their applications. It is safe to say that the AMBA bus is one of the most widely used bus systems and in 32 bit processors a de-facto standard. Many 3rd party vendors equip their components with AMBA interfaces for a better marketing chance.

¹²<http://www.gaisler.com/cms/>

¹³<http://www.astrium.eads.net/>

¹⁴<http://www.ballaerospace.com/page.jsp?page=1>

¹⁵<http://www.vineyardtechnologies.com/>

Chapter 4

AMBA Extension Module for SPEAR2

In the previous chapter AMBA 2.0 was described. In this chapter, a framework for the AMBA 2.0 implementation will be introduced: the GRLIB. After giving a more detailed view of the GRLIB, the extension concept of the SPEAR2 will be described. Those constraints are the basis for the design approach taken to extend the SPEAR2 processor core to be able to operate in an AMBA 2.0 network. The resulting problems and the implemented solutions are presented in the second part of this chapter.

4.1 AMBA and the GRLIB

As already mentioned in Chapter 3, the AMBA 2.0 bus system was chosen for the SPEAR2 processor core. But instead of implementing the pure AMBA 2.0 specification, it was decided to implement the extended AMBA 2.0 bus system, defined by the GRLIB¹.

4.1.1 What is GRLIB?

The GRLIB was introduced by Gaisler Research. Its purpose is to help designers to reuse existing IP cores and to simplify software development

¹Gaisler Research IP Library

and debugging on those designs. This was achieved by extending the AMBA 2.0 bus in the following manners:

- distributed address decoding
- interrupt steering
- Plug & Play capability

But the GRLIB does not solely consist of a new specification. It includes also a number of already designed components, such as:

- AHB arbiter/multiplexer
- AHB/APB bridge
- 10/100 MBit Ethernet MAC
- 32 bit PC133 SDRAM controller
- PCI interface
- PROM and SRAM controller
- UART
- timer
- interrupt controller
- fully pipelined single and double precision IEEE-754 FPU²

Those components are a major help for SoC designers since those components are tested and ready to use. The library itself is available under the GNU GPL license and, if required, under commercial licensing conditions[9], too.

²Floating Point Unit

4.1.2 GRLIB and its extensions in regard to AMBA 2.0

As mentioned in the previous chapter (Chapter 4.1.1), GRLIB uses the AMBA 2.0 bus as a basis and introduces new signals. The combination of those put new features into effect. Now let us analyze those new features[8] in regard to extending AMBA 2.0:

distributed address decoding: As mentioned in Chapter 3.2, AMBA uses two centralized multiplexors for controlling the AMBA bus. Adding or removing components always results in modifying the multiplexer logic and thus also affects the arbiter. Besides that, it can also be necessary to modify the decoder if new slave components were added.

GRLIB avoids this dependency by utilizing a distributed address decoding. To provide this mechanism, the GRLIB cores and the AMBA AHB/APB controller have to be modified. The modification enfolds by introducing generics and constants. Instead of adding new components by adding a new AHB bus, changing the control logic of the multiplexors and changing the arbiter/decoder, GRLIB defines two AHB bus vectors (one for the connection between the masters outputs and the multiplexor, one for the connection between the slaves outputs and the multiplexor) that contain a number of AHB buses (value can be changed by changing the constant NAHBMST and NAHBSLV). Now every AHB component gets an index (defined by generic) and thus it knows, which AHB bus has to be used and the decoder knows which master it has to forward to the slaves.

Additionally, every AHB slave component has generics that define which address spaces the component belongs to. With those generics, the arbiter knows which slave is meant and which slave response has to be forwarded to the masters.

interrupt steering: AMBA 2.0 does not support interrupt steering. In modern architectures this is a necessity.

The GRLIB provides a unified interrupt handling scheme. 32 different interrupts can be driven by the AMBA components (including APB components) since all components share the same view on the interrupt bus³. It is up to the component how many interrupts it needs, but for most components one interrupt is sufficient. Those interrupts are monitored by one component that is connected to the global interrupt register of the processor. Now if any interrupt is driven, this component maps them to the corresponding processor interrupt(s).

Plug & Play capability: In the AMBA 2.0 specification there is no room for an operating system Plug & Play support of component.

The GRLIB package offers dedicated read-only information register where every used component is registered. The register information consists of a unique IP core ID, the corresponding memory mapping of the component and the interrupt vector it uses. Those values are set by each components generics. A maximum of 64 master and 64 slave components are supported. The information registers are mapped to the end of the AMBA address space, consuming 4096 byte of address space (for an overview of the GRLIB address space see 4.1.3). Those registers are automatically filled at synthesise time.

To sum this up, GRLIB uses the AMBA 2.0 bus as a basis and introduces some signals. The arbiter and multiplexors have been integrated into one component for the distributed address decoding (see Figure 4.1). The new features are meaningful and not only for marketing, making the GRLIB a powerful SoC bus system.

4.1.3 GRLIB and its address space

The address space can be broken down into four parts:

- AHB memory bank
- AHB I/O bank

³it is called hirq in the GRLIB package

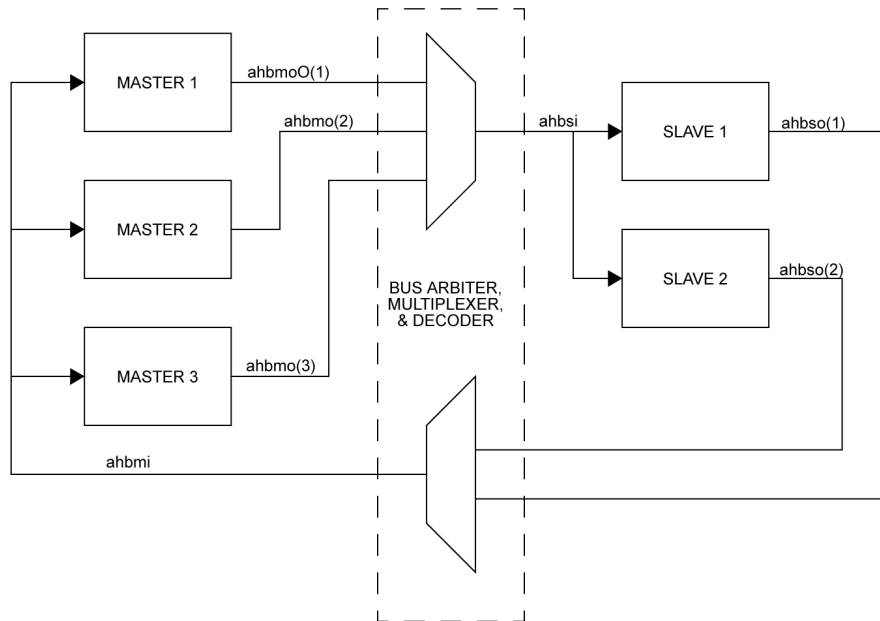


Figure 4.1: GRLIB interconnect[8]

- master information registers
- slave information registers

The AHB memory bank and AHB I/O bank are read and write spaces, whereas the information registers are solely for reading (and thus normally implemented as ROM's). Comparing those by their size, the AHB memory bank is the biggest with a size of 4293918720 bytes (= 4095 MByte) followed by the AHB I/O bank with a size of 1044480 bytes (= 1020 KByte). The information registers are the smallest with each a size of 2048 bytes (= 2 KByte).

The AHB memory bank resides on the top of the address space from 0x0000 0000 to 0xFFFF 0000. Components in this section have to reserve a minimum of 1 MByte address space and may use address space up to 2048 MByte. The actual size of a component is defined by its generics. As mentioned in Chapter 4.1.2, those generics are used to define the memory mapping and its size, which is stored in the Plug & Play information registers.

For a better understanding, the Plug & Play information register layout of a component (see Figure 4.2) will be described. It consists of eight 32 bit

words.

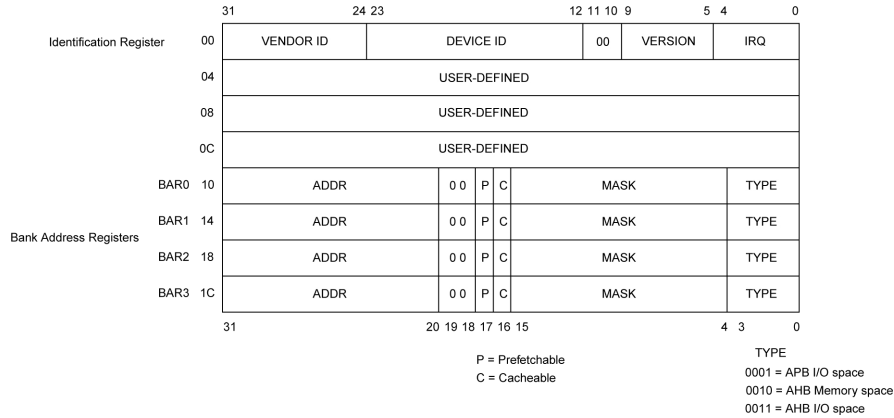


Figure 4.2: Plug & Play information register layout[8]

The first 32 bit word is for the identification of the component. It has a vendor ID which is administrated by Gaisler Research, a device ID which is administrated by the vendor, a version number which is administrated by the vendor and the interrupt it uses.

The second, third and fourth 32 bit words are free and can be used to store some additional read only information.

The 5th to the 8th 32 bit word is used for address mapping. Every component can have up to four address mappings. Every 32 bit word represents a mapping. The mapping consists of a base address which uses the upper 12 bit of an address, an address mask which is used for defining the size of the memory block, the type of the memory block and the information if the component supports prefetch or caches.

Let us illustrate the address mapping with an example. A component has to be placed at address 0x3600 0000 and it should have a memory size of 8 MByte. This means the base address is 0x360. If the arbiter just compared the highest 12 bits of the address to 0x360 now, this would result in a maximum memory size of 1 MByte. So, the mask register has to do the trick. It has to be set to 0xFF8. 0xFF8 is 1111 1111 1000 in binary representation. The zeros in the binary representation mask the comparison bits out, meaning that not the highest 12 bits are used for comparison but only the highest 9 bits. Now the component has an address space of 8 MByte.

Besides, in the memory bank this mapping is also used in the I/O bank. The I/O bank resides below the memory bank, using up the address space between 0xFFFF0 0000 and 0xFFFF F000. The minimum size of memory a component can allocate is 256 Byte. The maximum memory size a component may use is 1024 KByte. Like in the memory bank, components residing in the I/O bank have to set their generics for a precise address space definition, too. The only difference to the memory bank is the bits that are taken for the base address and the mask register. In the memory bank the highest 12 bits were used, in the I/O bank the following 12 bits are used (meaning bit 19 to 8). The bits 31 to 20 are already predefined with ones.

The last parts of the address space are the information registers. As above mentioned, both are equally big and have the same structure. The information registers concerning the master components reside between 0xFFFF F000 and 0xFFFF F800 whereas the information registers concerning the slave components reside between 0xFFFF F800 and 0xFFFF FFFF. The content of those memory spaces is solely made up by the Plug & Play configuration layout as described in Figure 4.2.

4.2 SPEAR2 Extension Modules

This section will give a brief overview of the SPEAR2 extension concept. Only the most important parts of the extension system will be given. For a more detailed explanation of the SPEAR2 extension system, refer to the master thesis about the SPEAR2 processor [6].

4.2.1 Overview of the SPEAR2 extension concept

The SPEAR2 extension concept uses a generic interface to map modules into the address space of the data memory. This generic interface is rather simple. It behaves like a synchronous RAM and thus has nearly the same signals as found in most basic synchronous rams. Besides those signals, it got two special signals: an interrupt signal and a so called "byte enable" signal. The reason behind the byte enable signal is that normally only 32 bit access

is allowed. To improve the granularity of the access, a four bit wide signal was defined that states which 8 bit parts of the 32 bit data word should be processed and which not. This is the byte enable signal.

So, to sum this up, the generic interface defines the following signals for information transfer from the SPEAR2 to the Extension Module:

- `write_en`: it indicates if a write (high) or a read (low) operation should take place
- `byte_en`: as mentioned, this 4 bit wide signal indicates which 8 bit part of the data word is valid and which is not (an example will be given shortly)
- `data`: this 32 bit wide signal holds the data that has to be written when the `write_en` signal is high
- `addr`: this 15 bit wide signal holds the address from which the data should be taken or where the new data has to be written

The information transfer from the Extension Module to the SPEAR2 is realised through the following signals:

- `data`: this 32 bit wide signal holds the data from a read operation
- `intreq`: this signal indicates if the Extension Modules has raised an interrupt or not

The next point is the address space that an Extension Module can take. By default an Extension Module has an address space of 32 byte. Thus, having a 15 bit wide address signal, it is possible to use up to 1024 Extension Modules at the same time. Of course, an Extension Module can take more than 32 byte of address space. To achieve that, it must only be mapped multiple times into the address space.

Instead of implementing a 15 bit comparator in each Extension Module a centralized Extension Module Access Control is used. Thus the component address mapping takes place in one location. This is also advantageous for

remapping components. Now only one signal is needed that tells the component if it is meant or not: the "select" signal. Every Extension Module has its own select signal that indicates if it is meant or not.

4.2.2 Limitations of the SPEAR2 extension concept

Like everything, the SPEAR2 extension concept also got its limitations. In the following lines the most important limitations will be described.

First: the not standardized interface. Although it is a quite simple interface, new Extension Modules always have to be written by the designer since the interface is not standardized. There is no possibility to use existing components like it is done in IP based designs today.

Second: the speed of the extension concept. It is always as fast as the SPEAR2. In modern bus systems the bus can operate at a lower (or higher) speed than the processor. This can be quite useful when the processor is so complex that it is rather slow while the bus and other components can operate at a higher speed.

Third: the absence of another bus. This can be in some cases a bottleneck. The bus can only operate as fast as its slowest component. If the Extension Modules are poorly designed, it can decrease the performance of the whole system.

To sum up, the SPEAR2 extension concept is a simple extension concept with little overhead. Because of this simplicity it has of course limitations that can decrease system performance, but this is always the trade off between complexity and speed.

4.3 AMBA integration approaches for SPEAR2

This section is about the approaches that were taken to implement another extension system for the SPEAR2. Besides that, the limitations of the design approaches and the solutions to them will be presented, too.

4.3.1 First approach

The AMBA state machine

The first thing to do was to read the AMBA 2.0 specification to get an overview on how AMBA 2.0 works and what means can be taken to implement it. Having read the AMBA 2.0 specification, the idea of using a state machine was concluded. This state machine should include the AMBA control logic. This state machine can also be seen as the core of every different implementation approach. So, building a solid and functioning state machine was the first objective to accomplish.

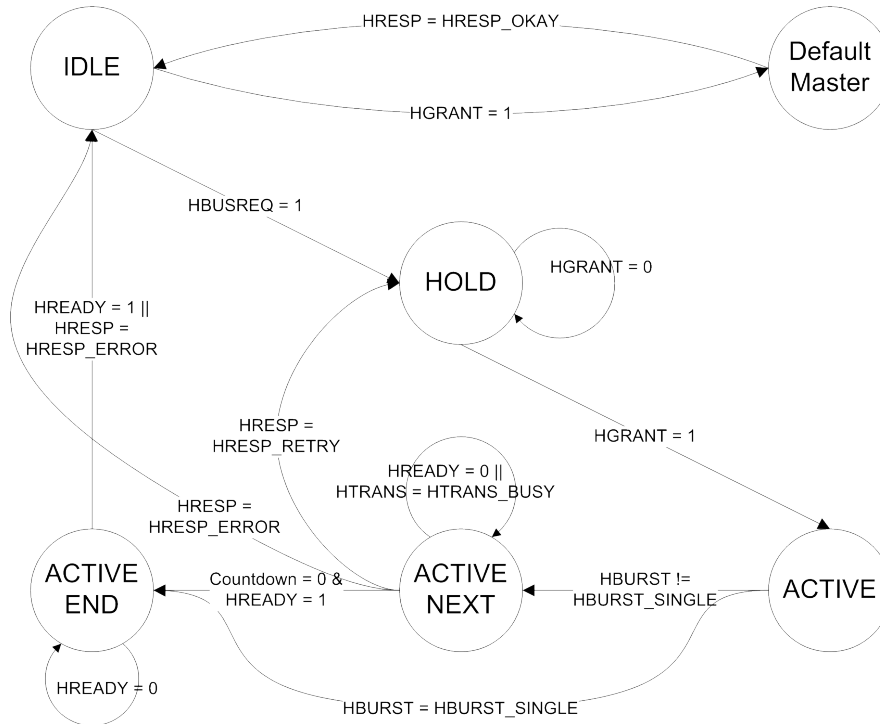


Figure 4.3: schematics of the AMBA state machine

The first idea was to write down the state machine (see Figure 4.3) and implement it. The first try of implementing the state machine was accomplished by using a visual tool that generates the VHDL code after drawing the state machine. The tool used was the Xilinx StateCAD Release 8.1i. But after some further use of the program it was clear that the produced VHDL

code was not user friendly and that changes always result in generating a new VHDL code, overwriting manual changes made to the previous VHDL code. Manual changes in the VHDL code were required for interfacing the state machine with the SPEAR2. So the idea of using a visual tool was discarded.

Still having the visualization of the state machine, the approach of coding was taken.

The next step was to define an interface between the AMBA state machine and the processor core. The requirements for this interface are:

- general use interface
- processor independent
- a way of transporting data between the state machine (and thus AMBA) and the processor
- a way of transporting configuration/status data between the state machine and the processor

We chose the already existing Extension Module concept for implementing the AMBA state machine into the SPEAR2.

for reading data from the processor: after steering the address, the corresponding data will be available in the next clock cycle

for writing data to the processor: after steering the address and the data, the memory will store it at the next rising clock edge

The configuration/status data that will be exchanged are:

AMBA bus request: This signal indicates if the processor wants to use the AMBA bus.

SPEAR2 base address: This 32 bit signal states the address where the sending or receiving data starts. If memory mapping is used, the address is the same as the AMBA address. If port mapping is used, the address can be different to the AMBA address.

AMBA address: The address on the AMBA bus where the data has to be sent or where the data has to be read from.

size of the AMBA transfer: This 3 bit signal indicates how much data will be transferred or should be received.

write or read transfer: This signal is used to indicate if the transfer is a write to AMBA or a read from AMBA transfer.

stall: This signal presents a possibility to stall the AMBA transfer if the processor is busy.

error: This signal gives information if the AMBA transfer is still OK or if an error occurred.

finished: If the transfer is finished, the signal will be driven from the state machine.

AMBA bus access granted: This signal signals the processor that the previous requested bus access has been granted.

interrupt: This 8 bit wide signal forwards 8 interrupts of the GRLIB system.

Now that the second interface of the AMBA state machine had been defined, the realization of the state machine started and finished quite smoothly.

The testbench for the AMBA state machine

As the AMBA state machine had been defined, the next step would be to integrate it into the SPEAR2 in form of an Extension Module. But since the AMBA state machine is the core and used in every design approach, the decision to postpone the integration had been made. Instead, the next step was writing a testbench for the just finished AMBA state machine.

The testbench contains all the necessary timing diagrams of the AMBA 2.0 AHB master specification. Further, to enhance the usability of the testbench, each necessary value is checked by assertions. It is also a goal of the testbench to write it once and never change it again.

So after writing the testbench, a test run in ModelSimTM was scheduled. This test run should show how well the implementation of the state machine was done. Like in most cases, there are always some errors. Since the checks are made by assertions, the errors of the state machine are shown immediately. After analysing the failure and correcting it, the testbench is started again. This “game” has to continue until the testbench finishes error free.

After checking the correctness of the AMBA state machine, the state machine represents a solid basis for further integration approaches.

It should also be mentioned that writing a testbench is very time consuming and error prone since forgetting to write an assertion is not so uncommon but introduces great consequences. Writing a good testbench needs nearly as long as writing the component.

The AMBA Extension Module for the AMBA state machine

Once a solid basis had been built, the next step was to integrate it into the SPEAR2 processor core in form of an Extension Module.

The idea of tying the SPEAR2 to the AMBA bus was to use an explicit send command, applied through the Extension Module Interface. Instead of transferring the data from the processors data RAM to the AMBA Extension Module Interface, we decided to use a (small) shared memory. From the processors point of view, the shared memory can be used in the same way as normal DRAM. Thus data, which is intended to be sent or received can be stored and manipulated directly in the shared memory. This significantly reduces the overhead when using the AMBA interface. However using a shared memory requires:

1. a true dual ported memory which is expensive in terms of area overhead and
2. some kind of access control in order to achieve data consistency (see MESI)

Thus we used a dual port memory, enhanced by a small access control unit.

So, an AMBA access would be realised if the following guide lines were observed:

1. if a send data should take place, then this data has to be written into the shared memory before the transfer takes place
2. write the AMBA address where the data has to be written (or read) into an Extension Module Register
3. write the SPEAR2 address where the data for transfer has to be read (or written) from the shared memory into an Extension Module Register
4. write the configuration data for the AMBA state machine into an Extension Module Register
5. set the start bit in the custom configuration register of the Extension Module
6. the finish bit will be set and, if not masked out, an interrupt will be generated after the AMBA state machine is finished

Since only one and a half registers are needed for storing all the necessary data and only half of the read only register, a second AMBA transfer slot is implemented for better utilization. The Extension Module would then look like in Figure 4.4.

The shared memory for the AMBA Extension Module

As previously mentioned, a shared memory is needed since the AMBA Extension Module should make the transfer in the background while the SPEAR2 continues with its program. The shared memory is tied as an Extension Module to the SPEAR2. Besides that, it also has a second interface for a direct connection to the AMBA Extension Module. So, the schematic of the whole AMBA extension for the SPEAR2 looks like in Figure 4.5.

Due to this integration approach, the access to the shared memory has to follow some rules: either the SPEAR2 or the AMBA Extension Module uses the shared memory. Also, to ensure the safety of the AMBA transfer, the AMBA Extension Module will steer an interrupt if the SPEAR2 tries to access the shared memory while the ownership lies with the Extension Module.

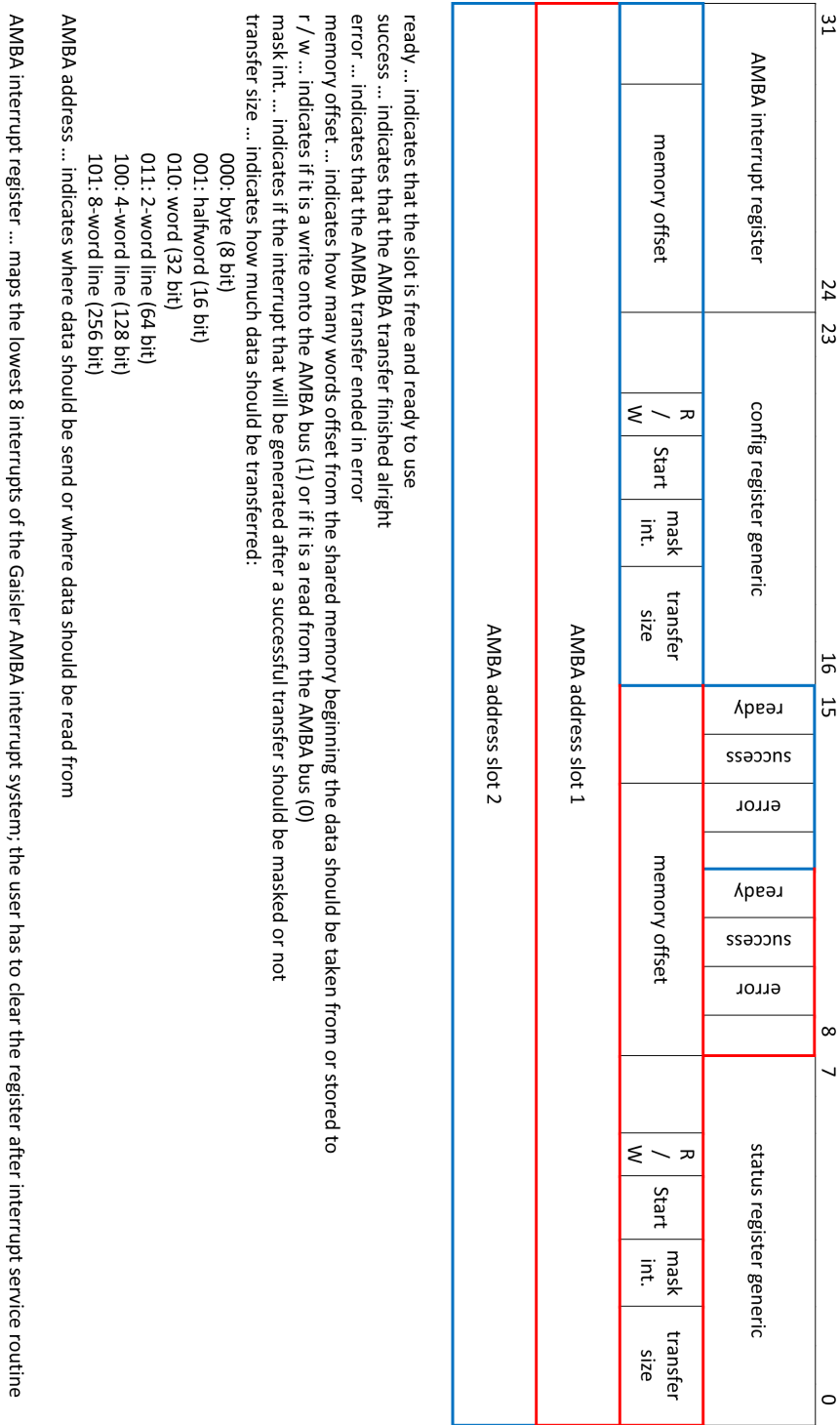


Figure 4.4: schematic of AMBA Extension Module - first approach

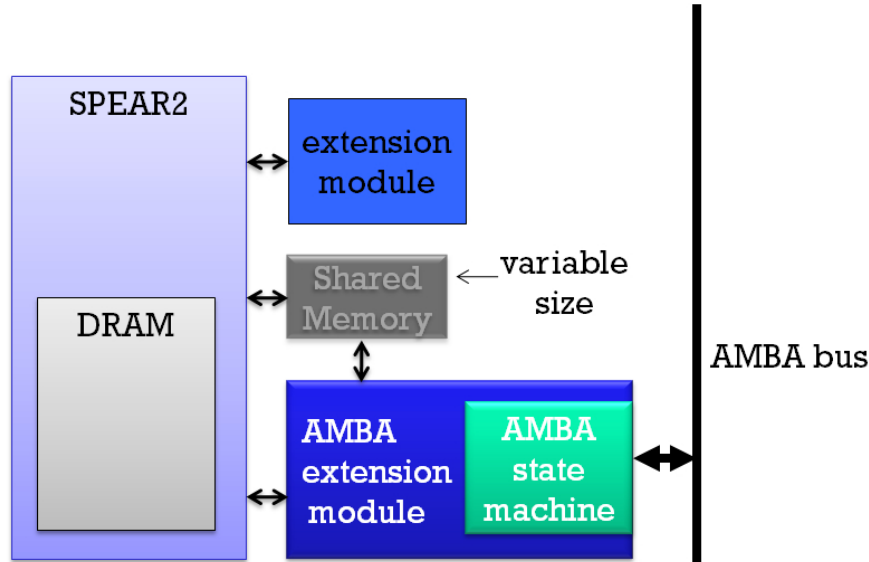


Figure 4.5: schematic of AMBA extension of SPEAR2

All the above mentioned rules, guidelines and the implementation make up the first approach of the AMBA Extension Module. Like every first approach, it has its limitations.

4.3.2 Limitations of the first approach

The first approach has some limitation in its practical use.

The SPEAR2 has no direct access to the AMBA state machine. Thus the entire data to setup an AMBA transfer must be provided to the AMBA state machine through the AMBA Extension Module Interface. This implies that accessing an AMBA component needs some time.

First: at least three store operations are required to setup an AMBA transfer:

- write the AMBA address to the register
- write the shared memory offset to the register
- write the config data to the register + set the start bit

Of course, if a send operation from the SPEAR2 to AMBA should take place, the data must already be adjusted.

Second: an access to AMBA has a big delay. As mentioned, three operations are needed to setup a data transfer in the AMBA Extension Module. Now a minimum of three cycles are needed for transferring the data itself:

1. one cycle for the state machine to request the bus
2. wait till the arbiter grants access to the AMBA bus (at least one cycle)
3. another cycle for steering the config data and address to the AMBA bus

When using the burst mode, this is not so important since a lot of data is transferred with a new data word each cycle. But when transferring only small amounts of data, this is an unacceptable delay.

These are the two most important limitations that the AMBA Extension Module introduced. Since the practical use of the Extension Module is not as good as it could be, a second approach is needed that takes those limitations into consideration.

4.3.3 Second approach

The first limitation originates from the fact that the SPEAR2 has no direct access to the AMBA address space. So, the first logic consequence is to unify those address spaces.

This sounds easy, but given the existing architecture it is not so easy to achieve. The big problem that makes it not so easy is the shared memory: every AMBA transfer takes or writes its data into the shared memory.

The first idea was to write a self organizing shared memory that maps the address space reserved for AMBA to its limited space. Then, based on the access to the data memory, the AMBA state machine should autonomously decide whether the transfer should take place or not. It is worth mentioning that this approach also supports complex data transfers. This decision algorithm would be as follows:

If the address refers to an AMBA component then:

- if a read on the AMBA bus is needed, make a burst transfer of four 32 bit words to the shared memory, beginning at the desired address and write back the first 32 bit word to the SPEAR2
- if a write to the AMBA bus is needed, store the AMBA address and the data word and wait if either:
 - two clock cycles passed and no new data is written to the AMBA address space \Rightarrow transfer the data to the address as a 32 bit data transfer
 - a new data word is written to the AMBA address space, but not directly after the previous AMBA address \Rightarrow transfer the previous stored data to the address as a 32 bit data transfer, and store the new data and AMBA address to a new space in the shared memory
 - a new data word is written to the AMBA address space directly after the previous AMBA address \Rightarrow store the data to the shared memory and wait again; if no new data follows or the new data should be transferred to another address segment, take the collected data and transfer it with a burst transfer

The self organizing shared memory would behave like a simple cache controller. But the goal was to write an AMBA Extension Module that is really just an add on to the SPEAR2 processor core and needs few resources. So this design approach was discarded.

Knowing that the first idea of unifying the address space was a failure, the decision to strip down the idea has been made. The controller logic of the shared memory was the reason why the new shared memory would need so many resources. So, why use such a complex logic? This was the birth of the final design approach: the hybrid approach.

The hybrid approach uses the old approach for large data transfers and the new approach for small data transfer (32 bit maximum). Thus the “intelligent” AMBA transfer mode is only available for simple access to the AMBA bus. This is easy to handle because small transfers occur and no temporal saving of data is needed since no burst transfers can occur. This transfer

would also be transparent to the SPEAR2 processor because when used, the SPEAR2 would be stalled and so it would seem that the transfer was made in an instant. From now on, this new transfer will be called “transparent mode”.

This new idea also solves the problem of the big delay for sending small amounts of data: no setup phase is needed for a transfer. The new transfer only needs three clock cycles for transferring data.

Complex transfers, such as bursts are not possible in this transparent mode. Thus, depending on the size of data which has to be transferred, either the normal mode or the transparent mode may be the better choice.

This is a big achievement in comparison to the first design approach and it also enhances the practical use.

4.3.4 Integration of the second approach into SPEAR2

The integration of the above mentioned idea is not so hard to accomplish. The first approach can be taken as basis and just needs some extensions.

First, a differentiation whether an internal SPEAR2 component (Extension Module or DRAM) or an AMBA component should be accessed is needed. This is achieved by adding a new signal.

After being able to distinguish between accessing a SPEAR2 component or an AMBA component, a stall signal for the processor core is needed.

The third step is extending the internal logic of the Extension Module. Besides the normal mode access for burst transfers, the transparent mode needs its own control logic. It can also happen that the SPEAR2 wants to initiate a transparent mode transfer, but another normal mode AMBA transfer is still running in the background. Then the SPEAR2 has to be stalled until the normal mode transfer finishes and the transparent mode transfer can take place. Besides that, status flags are needed, too. Suppose that something went wrong with the transfer. This is realised by adding another two status bits to the read only status register.

The fourth and last step in extending the already existing design is the address comparator that decides if a transparent mode takes place or not.

This address comparator is not realized in the Extension Module but in the top module of the SPEAR2. The reason is simple: The Extension Module only functions when the select signal of the module is driven. Since the address space of the Extension Module is now much greater (it also contains the address space of the transparent mode), it needs its own select steering logic. And this select steering logic can only be implemented on a higher level (for a more detailed explanation of the extension system see Section 4.2.1 or [6]).

4.3.5 Two different state machines

Having finished the AMBA Extension Module for the SPEAR2 processor core, one minor detail still exists: the state machine.

The implementation of the state machine is not meant for high speed grades. The reason is the optimization for low latency. When the input signals change, the state machine computes the new output signals and puts them immediately on the bus, meaning that on the next rising clock edge all components can already use the new signals.

This means that the timing tool takes takes following factors into consideration for computing the worst execution time of the state machine:

- the time that the latest possible input signal for the state machine of all AMBA components needs after the rising clock edge
- the time that the longest possible computation branch in the state machine needs
- the time that the farthest away AMBA component needs to receive the output signals of the AMBA state machine
- the worst setup hold time of all the AMBA components

Summing up all these factors, the time needed for one cycle will dramatically increase and thus the speed will dramatically decrease, too.

To solve this problem, it was decided to build a second state machine that solves this problem. The second state machine uses a fixed 1 stage pipeline,

meaning that the computed signals will be sent in the next clock cycle to the AMBA component. This increases the speed as the timing tool does not need to take into account the way of the output signals to the AMBA components, but increases the latency by a clock cycle.

The second state machine was also designed to have the same interface as the first state machine, meaning that changing the state machine only results in changing the VHDL file and nothing more.

4.3.6 Summary of the whole AMBA Extension Module integration

This chapter describes the way that led to the current design. It also shows that connecting an existing processor to a bus system sounds easy but can yield to some problems.

The approach that has been taken was to write a solid basis that is tested and then, based on this, write the “glue code” that ties the basis to the processor. But this “glue code” has to be designed well. Like in this case, the first approach that was taken to implement the glue code led to some limitations, making the Extension Module not user friendly and hard to handle in an interactive way. A failure in the first approach normally shows exactly what problems exist in combination with the existing processor architecture. After a redesign, in most cases a far better result will be achieved because of the previously gained information.

Implementing the second approach results in most cases in a usable, stable and quite well manageable component, ready for testing and, of course, for using.

In the last stage the speed factor should be analysed. As in this case, the state machine was too optimized on latency, resulting in lower speeds of the bus system. If the processor or some other components are still slower than the bus system, this does not matter. But if the bus system is responsible for the bottleneck, another redesign should be considered. The redesign is quite straight forward in most cases: just introducing pipelining or adding another pipeline stage. This normally boosts the speed of the bus system,

leaving the bottleneck at some other component.

Chapter 5

Results and Experiments

5.1 Integrating the AMBA Extension Module into the SPEAR2

This section is about the modifications that have to be made to integrate the AMBA Extension Modules into the SPEAR2. In the first part the interfaces of the Extension Modules needed will be described, in the second part the integration process itself will be presented.

5.1.1 Interfaces of the AMBA Extension Modules

Two Extension Modules are needed to make an existing SPEAR2 design AMBA capable: the shared memory and the AMBA Extension Module. Besides those two modules, another one will be described: the AMBA state machine. The AMBA state machine is embedded in the AMBA Extension Module, but since this is the core of the whole AMBA extension and since it is platform independent, it will be described, too. For a schematic of the connections between the modules, see the following Figure 5.1.

The purpose of the signals can be looked up in Appendix A.

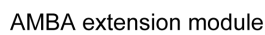


Figure 5.1: Interconnect of AMBA modules

5.1.2 Integrating the new Extension Modules

Integrating the above mentioned two Extension Modules, the SPEAR2 will be AMBA capable. Since the SPEAR2 is a flexible processor it was decided that integrating the AMBA modules should not decrease its flexibility. The most common way to achieve that is to integrate the AMBA modules into the SPEAR2 processor and tag them with a boolean value that states if the modules should be synthesized or not. This boolean value is declared in the SPEAR2 configuration file (see the master thesis about the SPEAR2 for further information [6]).

Besides that, it was also decided that the SPEAR2 is the only AHB master in an AMBA network in most cases, thus the AMBA arbiter, the AMBA decoder and the AMBA multiplexers will be integrated, too.

The last decision regarding the AMBA extension of the SPEAR2 that was made was the integration of the AMBA AHB/APB bridge. Since a lot of commonly used components like timers, UARTS, watchdogs and so on are APB slave components, it would definitely improve the usage of the SPEAR2 processor if the designer did not have to integrate the AHB / APB bridge too, but to provide the APB bus directly beside the AHB bus. Then he just has to attach the components to the provided AHB and APB buses. See the following Figure 5.2 for a simplified schematic of the internal structure of a SPEAR2 with the AMBA Extension Module.

The AHB master, the AHB decoder, the AHB multiplexors and the AHB/APB bridge used for integration are the ones provided by the GRLIB package since the AMBA Extension Module was also written by using the GRLIB extension specification of the AMBA bus.

The integration of the AMBA bus system into the SPEAR2 requires the following three steps:

1. extending the SPEAR2 *config file*
2. altering the GRLIB AMBA definition file
3. altering the SPEAR2 (*spear.vhd*)

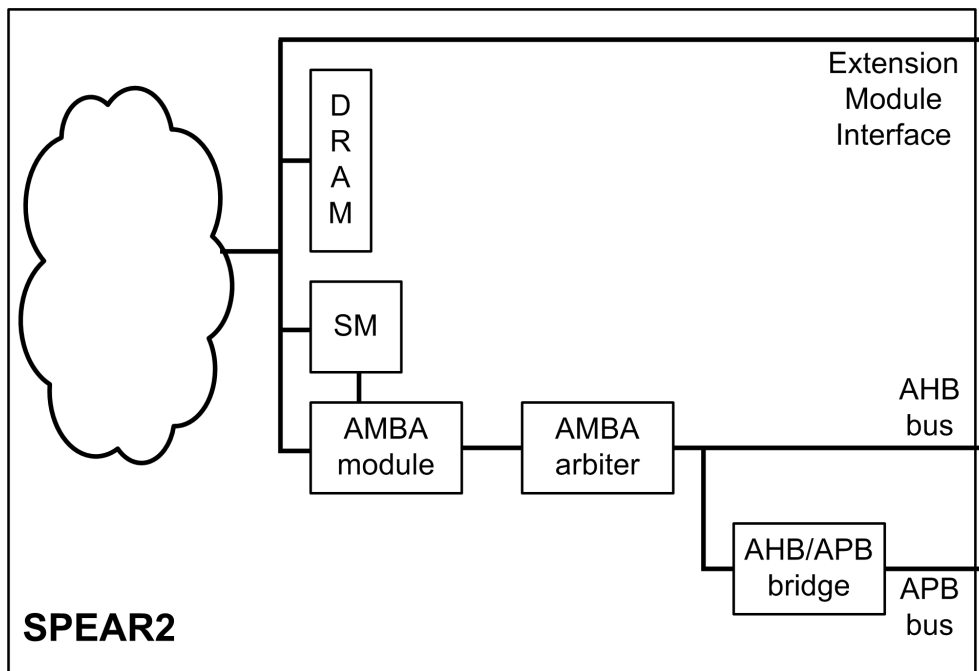


Figure 5.2: Internal structure of a SPEAR2 with AMBA Extension Module

Extending the SPEAR2 *config* file

The SPEAR2 *config* file has to be extended to provide the new config parameters that are needed for the AMBA components.

```
--
-- Defines if the AMBA modules should be loaded
--
constant ENABLE_AMBA : boolean := false;

--
-- Defines the start address of the AMBA APB bus
--
constant AMBA_APB_ADDRESS : integer := 16#800#;

--
-- Defines the number of AHB slaves
```

```
--
    constant AMBA_AHB_SLAVES : natural range 1 to 16 := 1;

--
--   Defines the number of APB slaves
--
    constant AMBA_APB_SLAVES : natural range 0 to 16 := 0;
```

The number of AHB slaves must be equal or higher than one since the AHB/APB bridge (an AHB slave component) is part of the integrating AMBA modules, too.

Altering the GRLIB AMBA definition file

Since in the GRLIB the maximum numbers of AHB masters, AHB slaves and APB slaves are each set to 16, a lot of warnings will be generated when synthesizing your design. Thus those values should be altered to reduce the number of warnings. The number of AHB masters is set to one since only one AHB master is present. The number of AHB slaves is set to two or higher since the AHB/APB bridge is one slave and the other one is needed for maintaining the port to the top level entity. If no other AHB slave is used, the port to the top level entity will be filled with dummy data. Last is the number of APB slaves. Since the AHB/APB bridge is no APB slave but an APB master, the exact number of APB slaves has to be filled in. All those values are taken from the SPEAR2 config file.

```
library work;
use work.spear_conf.all;

...

constant NAHBMST : integer := 1; -- maximum AHB masters
constant NAHBSLV : integer := AMBA_AHB_SLAVES+1; -- maximum AHB slaves
constant NAPBSLV : integer := AMBA_APB_SLAVES; -- maximum APB slaves
```

Altering the SPEAR2 (spear.vhd)

The next step is to integrate all components into the SPEAR2 processor.

The first step is to alter the interface of the SPEAR2. It has to be extended to support AHB and APB slaves. Thus the new interface looks as follows:

```
component spear
  port (
    clk          : in  std_ulogic;
    sysrst       : out std_ulogic;
    extrst       : in  std_ulogic;
    speari       : in  spear_in_type;
    spearo       : out spear_out_type;
    ahbsl2sp     : in  ahb_slv_out_vector_type(NAHBSLV-1 downto 1);
    ahbsp2sl     : out ahb_slv_in_type;
    apbsl2sp     : in  apb_slv_out_vector;
    apbsp2sl     : out apb_slv_in_type
  );
end component;
```

The AHB slave vector with index 0 is used for the AHB/APB bridge and internally connected. Of course, these changes have to be made in the SPEAR2 package, too.

Next the internal AMBA signals for interconnecting the AMBA components and the SPEAR2 core components have to be defined. For a complete list refer to Appendix B Section B.1 where the signals are listed.

Having defined all signals, the next step is to instantiate the following components:

- the AMBA Extension Module of the SPEAR2 which represents an AHB master
- the AMBA shared memory Extension Module of the SPEAR2 which is needed for non transparent transfers

- the AHB arbiter, the AHB multiplexors and the AHB decoder which are needed for controlling the AHB bus
- the AHB/APB bridge

In the framework developed with this Master thesis all those components are tagged with a boolean config parameter that states if those components are synthesized or not. For the listing refer to Appendix B Section B.2. Since the AHB/APB bridge is only small, the idea of using another tag for deciding if it should be synthesized or not was discarded.

As already mentioned in the previous lines, the SPEAR2 is the only AHB master. Thus the AHB arbiter / AHB multiplexor is parameterized with support for only one AHB master. Besides that, split support is disabled, the number of AHB slaves is 1 plus the number stated in the SPEAR2 config file and the fixed burst length is supported. All other parameters use the default value as described in the GRLIB IP library user's manual[8].

The AHB/APB bridge can be parameterized in the SPEAR2 config file, too. The parameter that can be set is the address where the bridge should be placed. As default value 0x800 is recommended for an easier implementation of the transparent mode. For details see the following Section 5.1.3.

The last step is to change the control code inside the SPEAR2. For a listing of the changes that have to be made, refer to Appendix B Section B.3.

In the following lines the most important code segments are explained.

As already written in the section about the SPEAR2 extension interface (see Section 4.2.1), the address has only a width of 15 bit. Since the transparent mode needs the full 32 bit, those other 17 bit have to be sent to the AMBA Extension Module when the 32 bit SPEAR2 is used. This is achieved by the following code:

```
if (ENABLE_AMBA = true and WORD_CFG_C = 2) then
    addr_high(31 downto 15) <= coreo.extaddr(31 downto 15);
else
```

```
    addr_high(31 downto 15) <= (others => '0');
end if;
```

The AMBA Extension Module also has the possibility to stall the processor. For this reason the hold signal of the SPEAR2 has to accept another input in case the AMBA modules are loaded:

```
if (ENABLE_AMBA = true) then
    exthold <= spearhold and spear1.hold;
else
    exthold <= spear1.hold;
end if;
```

Summary of the changes

The above mentioned steps sum up the changes that have to be made to the SPEAR2 to make it AMBA capable. The changes are only few in numbers and they also maintain the flexibility of the SPEAR2 processor.

For more details refer to the homepage of the SPEAR2 project where the altered SPEAR2, documentations and the necessary tools can be found.

5.1.3 Transparent mode transfer and the address space

As already described in Section 4.3.3 there is another transfer mode besides the standard transfer: the transparent mode.

This transparent mode is initiated when the transmode signal of the AMBA Extension Module asserts from low to high. But there must be a rule if a transparent mode transfer should happen or if not. For this reason it was decided that when writing to a certain address space it will be activated automatically. The address space that has been chosen starts at 0x8000 0000 and ends at 0xFFFF F800, the start address of the AMBA Extension Modules. This address space also simplifies the logic that asserts the transmode signal:

```
if (coreo.extaddr(WORD_W-1 downto 15) = EXTMODACT) then
```

```
    transmode <= '0';  
    ...  
else  
    ...  
    if (WORD_CFG_C = 2) then  
        transmode <= coreo.extaddr(31);  
    else  
        transmode <= '0';  
    end if;  
end if;
```

Of course, this address space is only available when using the 32 bit SPEAR2 version. When using the 16 bit SPEAR2 version, the transparent mode does not make any sense since the address space is too limited and thus it is deactivated by default.

5.2 Comparison of SPEAR2 and SPEAR2 with AMBA

Next to follow is a comparison between the SPEAR2 with a native extension UART and a SPEAR2 with an AMBA UART by Gaisler Research. This comparison should show the difference in the size and speed.

First the setup will be described, followed by the comparison itself. Finally a short conclusion will be given.

5.2.1 Setup of the experiment environment

Since size and speed will be compared first a suitable hardware has to be chosen. The example designs will be compared on an Altera Cyclone II Type EP2C35F484C6. This FPGA is attached to the Gleichmann Electronics development board "Hpe mini AC2" which will be used as hardware platform. As development tool Quartus II 6.0sp1 Web Edition was chosen for synthesizing, mapping and FPGA programming of the competing designs.

Besides that, no tuning attempts were made in regard of the synthesis and the mapping.

For the comparison I used a minimal SPEAR2 in its 32 bit configuration. The instruction memory and the programmer module were disabled. The data memory were the size of 8192 byte. The boot ROM was capable of storing 128 half words. In one case a UART with the Extension Module interface was used, in the other case the AMBA UART and the AMBA Extension Module were used.

After having described the setup of the experiment, the following part will present the results of the experiment.

5.2.2 Result of the experiment

The summary of the result can be taken from the table 5.1. Following this table, a more detailed explanation and a conclusion of the experiment will be given.

	SPEAR2 ext. UART	SPEAR2 AMBA UART
size	2760 logic elements 69632 memory bits 20 M4K memory blocks	3504 logic elements 71680 memory bits 24 M4K memory blocks
speed	55 MHz	State Machine without pipeline: 51 MHz State Machine with pipeline: 55 MHz

Table 5.1: Result of the experiment

As expected, the SPEAR2 with AMBA extension is larger. The difference is about 750 logic elements. Besides that, the SPEAR2 with AMBA interface uses 2048 bits of memory and 4 M4K memory blocks. Using 4 Cyclon II M4K memory blocks for only 2048 bit is extremely high. The reason for such a high degree of usage is the 4 independent 8 bit wide memory blocks that make up the shared memory. But since every memory block has to be controlled independently in regard to reading or writing, 4 blocks are needed.

The higher usage of the logic elements can easily be explained. In order to use AMBA, an AMBA master, an AMBA arbiter and an AMBA decoder are needed before a usable AMBA component can be used. These elements need, of course, logic elements. This is the price to pay to make an existing SPEAR2 design AMBA capable. But the advantages that come with the AMBA interface are manifold. Existing components can simply be added. For example the GRLIB¹[8] has over 70 IPs at the moment and the number is growing.

The reason of the speed difference between the two state machine is the critical path.

As in the comparison table stated, the SPEAR2 with the non pipelined AMBA state machine is 4 MHz slower than the SPEAR2 without AMBA. The reason is that:

- a signal has to start at the SPEAR2 dram,
- this signal goes through the AMBA Extension Module and the AMBA state machine,
- the state machine waits till the computation of the AMBA state machine is "final" and
- the computed signal finally goes through the AMBA bus and the AMBA multiplexer controlled by the AMBA arbiter to the AMBA UART.

The bottleneck in this design is the AMBA extension system.

The critical path can be broken by adding an additional pipeline inside the AMBA module. The resulting design is as fast as the SPEAR2 without the AMBA extension. But this also leads to a higher latency.

5.2.3 Summary of the experiment

The experiment was quite successful. It substantiated the initial assumption that the design with the AMBA extension is larger than the original design.

¹version 1.0.17, November 2007 was taken as reference

But it also shows that the amount of logic elements needed is not particularly high. The sacrifice in space can easily be justified by the fact that already existing components can be used and do not have to be modified or programmed again in order to accept the SPEAR2 extension interface.

The experiment also shows that the AMBA extension can be a bottleneck if it is poorly designed. But using current design rules like pipelining can easily prevent such bottlenecks.

Finally, this experiment shows that the SPEAR2 processor has to be altered in some of its elements (see Section 5.1.2 for more details) in order to make the SPEAR2 accept the AMBA Extension Module but the changes are few in numbers and can easily be changed.

Conclusion

A lot of different bus systems are available today. The reason for this is that there is no general purpose bus system, but each of them has got its advantages and disadvantages. Three characteristic bus systems were described, leading to the fourth one, the AMBA bus system. AMBA, like the other bus systems, has its advantages and disadvantages, leading to enhanced implementations of the bus to diminish these disadvantages.

The GRLIB package of Gaisler Research is such an enhancement of the AMBA 2.0 bus. This enhancement was chosen to implement on the SPEAR2 processor, leading to a more user friendly and more flexible extended SPEAR2. The reason to use GRLIB is because it is free and because a lot of already existing components (over 70 in number) are available.

A state machine and a shared memory represent the core components of the AMBA Extension Module for the SPEAR2 processor core. These components were customized in such a way that they fit in perfectly with the SPEAR2 design, leading to a better usability of the AMBA bus.

To enhance the usability of the SPEAR2 processor even more, the AMBA arbiter, the AMBA decoder and the AMBA AHB/APB bridge are integrated, too. Besides the standard Extension Module Interface the SPEAR2 has an AHB Interface and an APB interface where the designer can directly connect his AHB or APB components. He does not have to think about managing the bus or connecting the APB components to the AMBA master, he only has to connect the desired components to the bus. This makes the SPEAR2 processor extremely user friendly.

Of course, by extending the SPEAR2 with an AMBA interface, the processor becomes bigger. But the price for this extension (about 700 logic

elements) is justifiable when facing the fact that predesigned and pretested components can now simply be used with the SPEAR2 processor.

The experiment also shows that the implementation should not be done without thinking. If this is done, the danger arises that the extension can be the new bottleneck of the design. But if you apply the standard design rules like using pipelining or drawing down the outline of the design (for example the state machine with its transitions), the extension should not be the new bottleneck of the design.

An application for the SPEAR2 with the AMBA interface is the lecture “Hardware Software Co-Design” at the technical university of Vienna where the new AMBA interface opened a whole new segment of practical examples. Due to the AMBA interface, the IP based approach can be understood and experienced much better now.

Appendix A

Signal description

A.1 Shared memory

The shared memory is mainly needed for bulk data transfers. Besides the standard Extension Module Interface, it has another interface that is used for communication between the AMBA Extension Module and itself. The shared memory interfaces consist of the following signals:

clk: This is the clock signal of the module.

rst: This is the reset signal of the module

ambadramsel: This signal represents the select signal of the SPEAR2 Extension Module Interface.

exti: This bus is the connection of the SPEAR2 to the Extension Module as defined in the SPEAR2 Extension Module Interface.

exto: This bus represents the output signals of the Extension Module defined in the SPEAR2 Extension Module Interface with a small difference: if *ambadramlock* is active, the output is the default output (every one of the 32 signals is low).

ambadram-ren: If the shared memory is used by the AMBA Extension Module (*ambadramlock* is active), this signal controls if the memory of the shared memory is enabled or not.

ambadramlock: This signal controls if the AMBA Extension Module exclusively uses the shared memory or not. If the signal is active, the SPEAR2 core is not allowed to access the shared memory. If it still does, an interrupt is generated to prevent data inconsistencies.

adrami - write_en: This signal is used for writes from the AMBA Extension Module (*write_en* is active) to the shared memory or reads from it. This signal is only processed if *ambadramlock* is active.

adrami - byte_en: This 4 bit wide signal has the same purpose as in the Extension Module Interface. It signals the shared memory which memory banks are active and which are not. This signal is only processed if *ambadramlock* is active.

adrami - data: This 32 bit wide signal delivers data that has to be written to the shared memory. At the same time the address where the data has to be stored must be present. This signal is only processed if *ambadramlock* is active.

adrami - addr: This signal states where data has to be stored or where data has to be read from. Its width depends on the size of the shared memory. This signal is only processed if *ambadramlock* is active.

adramo - data: This 32 bit wide signal delivers the data of the shared memory from a read process. The data that has to be read from a given address is valid only in the following clock cycle.

A.2 AMBA state machine

The AMBA state machine is the core of the whole extension system. It asserts the AMBA master outputs in dependency of the AMBA master input and the AMBA Extension Module input. For this reason it has an interface to the AMBA bus and another one to the AMBA Extension Module.

HRESET: This is the reset signal of the module.

HCLK: This is the clock signal of the module.

BAtS - sMADDR: This 32 bit address signal is used to access the shared memory. The AMBA state machine also considers the access pattern of the shared memory: for reading, the address is asserted and in the next clock cycle the data will be present; for writing, the data and the address have to be asserted in the same clock cycle.

BAtS - sMWDATA: This 32 bit data signal is directly routed to the shared memory, forwarding the incoming AMBA data to the memory.

BAtS - sMWRITE: This signal is used to state if the data send to the memory should be read from or written to it. It is an obsolete signal in this AMBA implementation since the *byte_en* signal does the trick, but it has not been discarded because the module was created for general purpose use.

BAtS - sByteEn: This 4 bit signal has the same usage as the *byte_en* signal in the SPEAR2 Extension Module Interface specification.

BAtS - sERROR: This signal is used to indicate if an error occurred in the AMBA transfer. If so, the corresponding error flag in the AMBA Extension Module register is set.

BAtS - sFinished: This signal signals the successful transfer of an AMBA transfer. If it is asserted, the AMBA Extension Module sets the corresponding flag in the status register.

BAtS - sBusRequest: This signal indicates if the AMBA state machine has successfully requested the AMBA bus. It is needed to signal the AMBA Extension Module that an AMBA transfer will start shortly.

BAtS - sIRQ: This 8 bit signal represents the AMBA interrupts. It sends the information to the AMBA Extension Module where a logical or of the AMBA Extension Module interrupt register and the current AMBA interrupts takes place.

BStA - sHBUSREQ: This signal tells the AMBA state machine that an AMBA transfer is needed.

BStA - sBADDR: This 32 bit address signal indicates from which shared memory address the data should be taken from or written to. If a burst transfer takes place, this address is the base address for the data in the shared memory.

BStA - sHADDR: This 32 bit address signal indicates on the AMBA bus where the data should be sent to or read from.

BStA - sMRDATA: This 32 bit data signal holds the value of the shared memory read action. The data is from the address that was asserted in the previous clock cycle.

BStA - sHWRITE: This signal is used to tell the AMBA state machine if a read transfer or a write transfer on the AMBA bus should take place.

BStA - sHSIZE: This 3 bit signal indicates how much data will be transferred.

BStA - sWAIT: This signal can be used to stall the AMBA transfer. It is the equivalent of the *hready* signal of the AMBA AHB slave components.

AMBAI: This bus represents the AMBA AHB master in bus as described by the GRLIB specification. These signals are processed by the AMBA state machine module which is embedded in this module. For a detailed explanation of this bus refer to the GRLIB specification[8] and the AMBA 2.0 specification[4].

AMBAO: This bus represents the AMBA AHB master out bus as described by the GRLIB specification. These signals are asserted by the AMBA state machine module which is embedded in this module. For a detailed explanation of this bus refer to the GRLIB specification[8] and the AMBA 2.0 specification[4].

A.3 AMBA Extension Module

The AMBA Extension Module is the main Extension Module needed for making a SPEAR2 design AMBA capable. It has an Extension Module Interface, an interface to the shared memory and another interface for the AMBA bus. Although the AMBA Extension Module has the AMBA interface, the embedded AMBA state machine module is the part that drives these ports. Thus the AMBA Extension Module Interfaces consist of the following signals:

clk: This is the clock signal of the module.

rst: This is the reset signal of the module

extsel: This signal represents the select signal of the SPEAR2 Extension Module Interface.

exti: This bus is the connection of the SPEAR2 to the Extension Module as defined in the SPEAR2 Extension Module Interface.

exto: This bus represents the output signals of the Extension Module defined in the SPEAR2 Extension Module Interface with an enhancement: if a transparent mode read from AMBA transfer takes place, the result will be put on the bus when the hold signal is released again. Thus the data will be present only in the next clock cycle like when using a normal Extension Module.

gIRQ: This signal is used to indicate if an interrupt occurred on the AMBA bus. If an interrupt occurred, the interrupt register of the AMBA Extension Module has to be read because this signal is a logical or over the 8 possible AMBA interrupts. For more details regarding the AMBA interrupt, refer to section 4.1.2 where the GRLIB and its enhancements to AMBA 2.0 are explained.

spearhold: This signal is used to stall the SPEAR2 processor. It is used when a transparent mode transfer takes place. For more details regarding the transparent mode transfer, refer to section 4.3.3 where the transparent mode transfer is introduced.

ambadram_ren: If the shared memory is used by the AMBA Extension Module (*ambadramlock* is active), this signal indicates if the memory of the shared memory is enabled or not.

ambadramlock: This signal indicates if the AMBA Extension Module exclusively uses the shared memory or not. If the signal is active, the SPEAR2 core is not allowed to access the shared memory. If it still does, an interrupt is generated to prevent data inconsistencies.

AtD - write_en: This signal indicates if the AMBA Extension Module writes (*write_en* is active) to the shared memory or reads from it. This signal is only processed if *ambadramlock* is active.

AtD - byte_en: This 4 bit wide signal has the same purpose as in the Extension Module Interface. It signals the shared memory which memory banks are active and which are not. This signal is only processed if *ambadramlock* is active.

AtD - data: This 32 bit wide signal delivers data that has to be written to the shared memory. At the same time the address where the data has to be stored must be present. This signal is only processed if *ambadramlock* is active.

AtD - addr: This signal indicates where data has to be stored or where data has to be read from. Its width depends on the size of the shared memory. This signal is only processed if *ambadramlock* is active.

DtA - data: This 32 bit wide signal delivers the data of the shared memory from a read process. The data that has to be read from a given address is valid only in the following clock cycle.

AMBAI: This bus represents the AMBA AHB master in bus as described by the GRLIB specification. These signals are processed by the AMBA state machine module which is embedded in this module.

AMBA0: This bus represents the AMBA AHB master out bus as described by the GRLIB specification. These signals are asserted by the AMBA state machine module which is embedded in this module.

Appendix B

Integration code

B.1 Internal signals

```
-- signals for component amba shared memory
signal    ambadram_ren : std_ulogic;
signal    ambadramsel  : std_ulogic;
signal    ambadramlock : std_ulogic;
signal    ambadramexto : module_out_type;
signal    ambadrami    : sm_dram_in_type;
signal    ambadramo    : sm_dram_out_type;
-- signals for component amba extension module
signal    ambaexto     : module_out_type;
signal    ambasel      : std_ulogic;
signal    addr_high    : std_logic_vector(31 downto 15);
-- signal for transparent mode
signal    transmode    : std_ulogic;
-- signal for Gaisler interrupt system
signal    gIRQ         : std_ulogic;
-- signal for stalling spear
signal    spearhold    : std_ulogic;
-- signals for amba bus system + amba module outputs
signal    ahbmi : ahb_mst_in_type;
```

```

signal    ahbmo : ahb_mst_out_vector := (others => ahbm_none);
signal    ahbsi : ahb_slv_in_type;
signal    ahbso : ahb_slv_out_vector := (others => ahbs_none);

```

B.2 AMBA components integration

```

ena_amba : if (ENABLE_AMBA = true) generate

```

```

-----

```

```

--- AMBA DRAM-Module on Extension-Bus

```

```

-----

```

```

ambadram_unit: ext_AMBA_sharedmem
  port map(
    clk          => clk,
    rst          => intrst,
    ren          => ambadram_ren,
    -- control and select signals
    ambadramsel  => ambadramsel,
    ambadramlock => ambadramlock,
    -- Extension-Module Interface
    exti         => exti,
    exto         => ambadramexto,
    -- AMBA <-> Shared DRAM
    adrami       => ambadrami,
    adramo       => ambadramo
  );

```

```

-----

```

```

--- AMBA-Module on Extension-Bus

```

```

-----

```

```

amba_unit: ext_AMBA
  generic map(
    hindex      => 0,
    DRAMOffset  => (others => '0')
  )

```

```

)
port map(
    clk          => clk,
    rst          => intrst,
    extsel       => ambasel,
    ambadramlock => ambadramlock,
    transmode    => transmode,
    -- Extension-Module Interface
    exti         => exti,
    exto         => ambaexto,
    addr_high    => addr_high,
    -- Gaisler interrupt
    gIRQ         => gIRQ,
    spearhold    => spearhold,
    -- AMBA-Interface
    AMBAI        => ahbmi,
    AMBAO        => ahbmo(0),
    -- DRAM-Interface
    ambadram_ren => ambadram_ren,
    AtD          => ambadrami,
    DtA          => ambadramo
);

-----
--- AMBA AHB arbiter/multiplexer
-----

ahb0 : ahbctrl
generic map(
    defmast => 0,          -- default master
    split   => 0,          -- split support
    nahbm   => 1,          -- number of masters
    nahbs   => 1+AMBA_AHB_SLAVES, -- number of slaves
    fixbrst => 1          -- support fix-length bursts

```

```

)
port map(
    rst => intrst,
    clk => clk,
    msti => ahbmi,
    msto => ahbmo,
    slvi => ahbsi,
    slvo => ahbso
);

-----
--- AMBA AHB/APB Bridge
-----

apb0 : apbctrl
generic map(
    hindex => 0,
    haddr  => AMBA_APB_ADDRESS,
    hmask  => 16#fff#,
    nslaves => AMBA_APB_SLAVES
)
port map(
    rst => intrst,
    clk => clk,
    ahbi => ahbsi,    -- from master to bridge
    ahbo => ahbso(0), -- from bridge to master
    apbi => apbsp2sl, -- from bridge to slaves
    apbo => apbsl2sp  -- from slaves to bridge
);

-- route the remaining AHB slave outputs to the
-- AHB slave output vector (remaining slaves are present
-- in the top level entity)
ahbso(NAHBSLV-1 downto 1) <= ahbsl2sp(NAHBSLV-1 downto 1);

```



```

-- send the slave in signal to the top level entity
-- is needed for attaching other AHB slaves
ahbsp2sl <= ahbsi;

end generate;

no_amba : if (ENABLE_AMBA = false) generate
ahbsp2sl <= ((others => '0'),(others => '0'),'0',(others => '0'),
(others => '0'),(others => '0'),(others => '0'),(others => '0'),
'0',(others => '0'),'0',(others => '0'),'0',(others => '0'),
'0','0','0','0');
apbsp2sl <= ((others => '0'),'0',(others => '0'),'0',
(others => '0'),(others => '0'),'0','0','0','0');
end generate;
-----
-- End Configurable SPEAR Modules
-----

```

B.3 SPEAR2 code modification

```

spear2ext.data <= (others => '0');
spear2ext.addr <= coreo.extaddr(14 downto 0);
spear2ext.byte_en <= (others => '0');
spear2ext.write_en <= coreo.extwr;
if (ENABLE_AMBA = true and WORD_CFG_C = 2) then
    addr_high(31 downto 15) <= coreo.extaddr(31 downto 15);
else
    addr_high(31 downto 15) <= (others => '0');
end if;

...

syscsel <= '0';

```

```
progsel <= '0';
spearo.extsel <= '0';
dramsel <= '0';
ambasel <= '0';
ambadramsel <= '0';

v.ext_mod_sel := '0';
if (coreo.extaddr(WORD_W-1 downto 15) = EXTMODACT) then
  transmode <= '0';
  v.ext_mod_sel := '1';
  case coreo.extaddr(14 downto 5) is
    when "1111111111" =>
      --SYSC Module
      syscsel <= coreo.extwr or coreo.memen;
    when "1111111110" =>
      --PROG Module
      if (USE_IRAM_CFG_C = true) then
        progsel <= coreo.extwr or coreo.memen;
      else
        null;
      end if;
    when "1111111000" =>
      --AMBA Module
      if (ENABLE_AMBA = true) then
        ambasel <= coreo.extwr or coreo.memen;
      else
        null;
      end if;
    when "1000000000" =>
      --AMBA Shared Memory
      if (ENABLE_AMBA = true) then
        ambadramsel <= coreo.extwr or coreo.memen;
      else
```

```
        null;
    end if;
when "1000000001" =>
    --AMBA Shared Memory
    if (ENABLE_AMBA = true) then
        ambadramsel <= coreo.extwr or coreo.memen;
    else
        null;
    end if;
when "1000000010" =>
    --AMBA Shared Memory
    if (ENABLE_AMBA = true) then
        ambadramsel <= coreo.extwr or coreo.memen;
    else
        null;
    end if;
when "1000000011" =>
    --AMBA Shared Memory
    if (ENABLE_AMBA = true) then
        ambadramsel <= coreo.extwr or coreo.memen;
    else
        null;
    end if;
when others =>
    null;
end case;
spearo.extsel <= coreo.extwr or coreo.memen;
else
    dramsel <= coreo.extwr or coreo.memen;
    if (WORD_CFG_C = 2) then
        transmode <= coreo.extaddr(31);
    else
        transmode <= '0';
    end if;
end if;
```

```
    end if;
end if;

v.extdata := (others => '0');
for i in v.extdata'left downto v.extdata'right loop
    if (ENABLE_AMBA = true) then
        v.extdata(i) := speari.data(i) or syscexto.data(i) or
            progexto.data(i) or ambaexto.data(i) or ambadramexto.data(i);
    else
        v.extdata(i) := speari.data(i) or syscexto.data(i) or
            progexto.data(i);
    end if;
end loop;

...

if (ENABLE_AMBA = true) then
    exthold <= spearhold and speari.hold;
else
    exthold <= speari.hold;
end if;

r_next <= v;
```

Bibliography

- [1] Altera Corporation. *Avalon Memory-Mapped Interface Specification*, Apr. 2007. URL,http://www.altera.com/literature/fs/fs_avalon_streaming.pdf.
- [2] Altera Corporation. *Avalon Streaming Interface Specification*, June 2007. URL,http://www.altera.com/literature/fs/fs_avalon_streaming.pdf.
- [3] Altera Corporation. *SOPC Builder's System Interconnect Fabric*, 2007. URL,http://www.altera.com/products/software/products/sopc_avalon/nio-avalon_bus.html.
- [4] ARM Limited. *AMBATM Specification (Rev 2.0)*, May 1999. URL,http://www.arm.com/products/solutions/AMBA_Spec.html.
- [5] M. Barr. *Embedded Systems Glossary*, 1999-2007. URL,<http://www.netrino.com/Publications/Glossary/E.php>.
- [6] M. Fletzer. Spear2. Master's thesis, TU Wien, Institut für Technische Informatik, Vienna, Austria, Mar. 2008.
- [7] T. Flik. *Mikroprozessortechnik und Rechnerstrukturen*. Springer Verlag, 7. edition edition, 2004.
- [8] J. Gaisler and S. Habinc. *GRLIB IP Library User's Manual*, 2006. URL,<http://gaisler.com/products/grlib/grlib.pdf>.
- [9] Gaisler Research. *GRLIB product brief*, Sept. 2004. URL,<http://www.gaisler.com/doc/Leon3%20Grlib%20folder.pdf>.
- [10] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann Publishers, 4th edition edition, 2007.
- [11] International Business Machines Corporation. *CoreConnectTM Bus Architecture*, 1999. URL,http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF7785256991004DB5D9/file/crcon_pb.pdf.

- [12] International Business Machines Corporation. *The CoreConnectTM Bus Architecture*, 1999. URL,http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF77852569910050C0FB/file/crcon_wp.pdf.
- [13] International Business Machines Corporation. *CoreConnect FAQ*, Sept. 2002. URL,<http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/3C53ED93E1397EF487256B190068A363>.
- [14] OpenCores Organization. *WISHBONE, Revision B.3 Specification*, July 2002. URL,http://www.opencores.org/projects.cgi/web/wishbone/wbspec_b3.pdf.
- [15] D. A. Patterson and J. L. Hennessy. *Rechnerorganisation und -entwurf*. Spektrum, 3. auflage edition, 2005. translated by Elke Jauch and Judith Muhr.
- [16] W. Peterson. *An introduction to WISHBONE: A chip-level microcomputer bus*, 2004. URL,<http://www.vmebus-systems.com/pdf/Silicore.Feb04.pdf>.
- [17] SiliconFarEast. *SiliconFarEast*, 2005. URL,<http://www.siliconfareast.com/soc.htm>.
- [18] A. S. Tanenbaum. *Computerarchitektur*. Pearson Studium, 5th edition edition, 2006. translated by Dipl.-Ing. Frank Langenau.