2002

A system-on-chip (SoC) digital interface IP core.

Huimei. Zheng
University of Windsor

A System-on-Chip (SoC) Digital Interface IP Core

By

Huimei Zheng

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through Electrical and Computer Engineering
in Partial Fulfillment of the Requirements for
the Degree of Master of Applied Science at the
University of Windsor

Windsor, Ontario, Canada
August 2002

0-612-75858-3

972992

Abstract

The work presented in this thesis describes a digital interface Intellectual Property (IP) core, for use in a System-on-Chip (SoC) environment. This IP core allows the main SoC bus to read and write to peripheral devices, such as A/D and D/A converters, respectively.

The microprocessor IP core typically defines the properties required for the main SoC bus. In the thesis the system bus requirements are based on the use of the ARM IP microprocessor core (V2.0). Both high performance bus (AHB) and peripheral bus (APB) specifications are available, together with state transaction specifications for bridging between the two bus systems. Peripheral IP cores connect to the APB bus, which interconnects to the APB port on the interface (bridging) IP core.

This thesis develops a high level Verilog description of an interface (bridging) IP core and a Verilog description of related interface circuitry that would be resident in the peripheral IP cores. The interface IP core has both an AHB port and an APB port. The Verilog description of the interface (bridging) IP core and the circuitry resident in the peripheral IP core have both been mapped to a 0.35-micron CMOS technology. The interface core is comprised of 1612 elements consisting of logic gates, multiplexors, inverters, and shift-registers.

Acknowledgements

There are several people who deserve my sincere thanks for their generous contributions to this project.

I would first like to thank my supervisor Dr. W.C. Miller for bringing this challenging project to my attention, and for his valuable guidance, encouragement as well as support throughout the period of my master degree program. I would also like to thank my committee members Dr. C. Chen and Dr. D. Antonelli, for their assistance and advice in completing this thesis.

A great deal of thanks also goes to Meinan, for his enormous help and comments during the progress of the thesis and will not be forgotten.

Finally I would like to thank all my colleagues at the RCIM Research Group for all the happy memories during the time of my study.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

A/D	Analog to Digital
AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ASB	Advanced System Bus
ASIC	Application-Specific Integrated Circuit
BBD	Block Based Design
CMOS	Complimentary Metal Oxide Semiconductor
CPU	Center Processor Unit
D/A	Digital to Analog
DCR	Device Control Register
DES	Digital Encryption Standard
DMA	Direct Memory Access
DRC	Design Rule Check
DSM	Deep Submicron
DSP	Digital Signal Processing
EOC	End of Conversion
FIFO	First In First Out

FPGA	Field Programmable Gate Array
FSM	Finite State Machine
IC	Integrated Circuit
I/O	Input/Output
IP	Intellectual Property
MEMS	Microelectronic Mechanical System
MPEG	Moving Picture Expert Group
PCI	Peripheral Component Interconnect
RAM	Random Access Memory
ROM	Read Only Memory
RTL	Register Transfer Level
SoC	System-on-Chip
SC	Start of Conversion
µP	Microprocessor
VC	Virtual Components
VHDL	VHSIC hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Chapter 1

Introduction

1.1 Introduction

The world market for electronic systems is following an exponential growth curve. The functionality and value of electronics-based products are increasingly being embedded in the integrated circuits that are the critical elements of these products, most noticeably in the areas of communications, multimedia, and transportation. The functionality of integrated circuits is growing to system-level proportions in which several complex sub-components, including logic, memory, numerical and signal processing, and radio frequency circuits reside on a single chip or chip set. Creating single chip solutions to system challenges is already the target of intense research and development effort around the world, and so-called "System-on-Chip" (SoC) designs, are becoming closer to a reality. Semiconductor manufacturing technologies are routinely capable of implementing chips comprising millions of gates, with hundreds of thousands of information bits spread over a constellation of embedded memories together with embedded communication networks and analog or mixed-signal interfaces.

We are now entering the era of block-based design (BBD), heading toward virtual component-based SoC design, which is driven by our ability to harness reusable virtual components (VC), a form of intellectual property (IP), and deliver it on interconnect-dominated deep submicron (DSM) devices.

1.2 Overview of SoC design

The conventional approach to integrated circuit design is to merge large numbers of simple components, through integration techniques, to produce complex chips, which become subsystems of a printed circuit board-based system implementation. The shift to SoC implementation involves a phenomenal increase in complexity by merging subsystems of chip-level complexity into a single chip. The subsystems that make up this SoC include standard interface blocks, reused design cores, embedded software, and new, innovative, custom designed "user blocks".

The SoC design methodology is a new paradigm in digital logic and microelectronics. It is a method by which whole systems are created on a single integrated circuit chip. In many case, this required the use of IP cores that have been designed by multiple IP core providers. SoC is similar to traditional microcomputer bus systems hereby the individual components are designed, tested and built separately. The components are then integrated to a finished system.

SoC design is defined as a complex IC that integrates the major functional elements of a complete end product into a single chip or chipset. In general, SoC design incorporates a programmable processor, on-chip memory, and accelerating function units implemented in hardware. It also interfaces to peripheral devices and/or the real world. SoC designs encompass both hardware and software components. Because SoC designs can interface to the real world, they often incorporate analog components, and can, in the future, also include MEMS components.

1.3 Reuse — The key to SoC design

The conventional Fundamental to the possibility of highly integrated SoC designs is the notion of design reuse. Reusing IP has long been touted as the fastest way to increase productivity. Chip designs have for the last 20 years reused design elements. What has been changing has been the level of reuse abstraction. In an ASIC style flow, involving RTL logic synthesis and automated standard cell place and route, the reuse abstraction has been at the basic cell level, where a cell represents a few gates of complexity. In addition, reuse of standard modules produced by generators or by hand, such as memories etc. has been common.

SoC design has involved the reuse of more complex elements at higher levels of abstraction. Block-based design, which involves partitioning, designing and assembling SoCs using a hierarchical block-based approach, has used the IP block as the basic reusable element. This might be an interface function such as a PCI or 1394 bus interface block; an MPEG2 or MP3 decoder; an implementation of data encryption or decryption such as a DES block, or some other complex function. The importance for SoC using an IP is increasing in modern design methodology. The past design method is not suitable.

1.4 SoC interface and communication

The main goal in the design of numerous microcomputer-based systems is to produce a reliable, high-performance, easy-to-use product at the lowest cost. An important part of the design is the interface between the microcomputer and the equipment it monitors or controls. Interfacing is a term that applies across a broad range of electronic implementation. It relates to systems as well as to individual transistors. The addition of interfacing circuitry to a microcomputer provides a useful means of expanding capability and enhancing performance. It gives the microcomputer the ability to communicate with a variety of external devices, or other computer systems.

Designing functional parts of large digital systems is no longer the key element of successful implementation. Instead, the on-chip communication and interconnects form the challenge. Communication using traditional methods such as on-chip buses is complex to design, verify and control. Especially, the traditional methods will neither have sufficient flexibility nor tolerate high on-chip noise. When the internal communication of the design gets more complex it will also increase the design productivity problems. It is a very well known fact that already now designers cannot use all the capacity that the processing technologies are providing. Thus in the future SoCs not only the functional parts but also the internal communication structures must be built using reusable blocks with configurable architectures and interfaces. That kind of interconnection network can be adjusted according to the application with minimum additional design costs. By using a reuse approach through the design process, including communication design, the design time and cost can be decreased. This development of methods is forming the design and implementation of SoC communication.

Fig.1.1 illustrates the generic architecture of a SoC. System designers obtain cores in hard (implemented in a particular technology) or soft (register-transfer-level) format from cores. Cores are integrated via a custom or commercial interconnection network with a controller and a timing and function interface to the external world. Cores can be either new or legacy cores — inherited from existing designs.

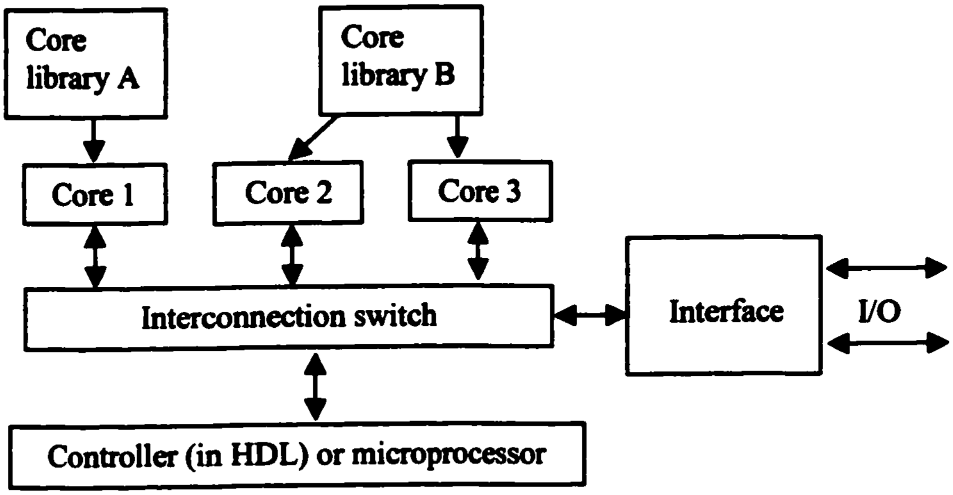


Figure 1.1: Generic SoC architecture

1.5 Thesis overview

The primary objective of this research is to develop a digital interface IP core, which enables the programmer to transfer a block of characters from peripheral devices (like MEMS) to the SoC main system bus. In addition, the interface IP core will also be implemented to enable the programmer to transfer data from main SoC bus to peripheral device, like a D/A converter.

Fig. 1.2 shows the flowchart of MEMS signal.

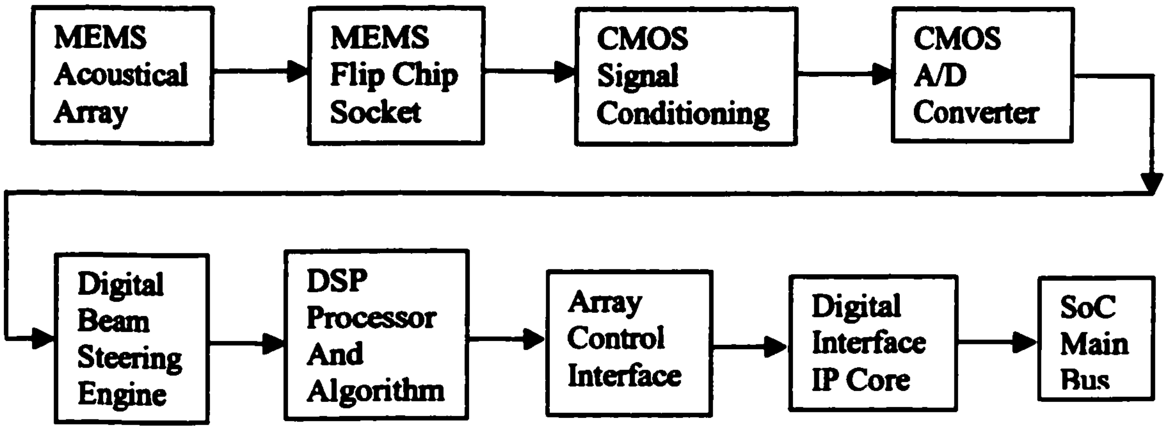


Figure 1.2: The flowchart of MEMS signal

The design of embedded SoC IP core will be based on giving a written specification for the on-chip interconnection.

1.6 Thesis organization

This thesis is organized as follows: Chapter 2 introduces some interface principle and methods. Chapter 3 provides information of some on-chip bus interconnect architectures. After the comparison, one of them will be selected as the basic architecture of the designed IP core. Chapter 4 describes the designed IP core's specification. Chapter 5 lists the design partitioning steps, including the description of core modules and testbench modules. Chapter 6 shows the synthesis (Synopsys) and physical design (Cadence) processes by using CMOSP35 technology. Chapter 7 concludes this thesis and gives a future direction.

Chapter 2

Interfacing Techniques

2.1 Introduction

The component elements of the microprocessor are interconnected by wiring paths known as *buses*. There are three types of buses: address, data, and control. The address bus carries binary-coded patterns (addresses) from the microprocessor to the ROM, RAM, and peripheral interface. The addresses identify unique locations that will be the source or destination of information transferred to or from the microprocessor. This information travels over the bi-directional data bus. The information falls into the four general categories of instructions, control words, status indicators, and data. Instructions contained in ROM are executed by the microprocessor, but intelligent peripheral devices that contain their own processors can execute instructions initiated by the microprocessor. Control words usually originate in the microprocessor and are transmitted to intelligent peripheral devices. Data are available from all elements of the microprocessor and travel on the data bus. The control bus contains a variety of signals such as reset, interrupt, halt, acknowledges, and so on.

The interface unit offers the following operations:

- Write a single data element or block write;

- Send a *Read-Request* for a single data element or *Read-Request* for a block;
- Transmit interrupt requests to the processor.

A programmer sees the interface as eight registers that are Data, Address, Command, Begin Address, Data Amount, Status, Interrupt Address and Interrupt Data. All of these registers are parameterized to width of the processor bus.

2.2 Design considerations

The most suitable method of interconnect will depend on the type of information to be transmitted, how frequently transmission occurs, the way in which the bus lines are shared, and the urgency of the communication. The main factors that influence the structure of the bus interconnections may be summarized as follows:

- How rapidly data are transferred;
- Whether concurrent communication is required;
- How often devices communicate;
- How long a delay between requesting and starting transmission is acceptable;
- How much information is transmitted during a single transfer;
- How likely are transmission errors;
- How reliable are the devices controlling the bus;
- How important is it that the bus continue to function correctly if one or more of the interconnected devices fails;
- Whether all devices and communications are to be treated equally;

- Whether it is important to minimize the number of interconnections and to simplify the bus interconnection logic;
- Whether the devices to be connected are physically remote;
- Whether the interconnect scheme must provide for expansion in terms of the number of interconnected devices.

2.3 Interface scheme

Selecting the best interface method for each IP is crucial to use the full power of IP. In this section, we explain the interface methods we support. The general interface method is shown in Figure 2.1. It can have in/out-buffers if needed and the in/out-controller that controls the interface scheme. The protocol transformer transforms IP specific various protocols into our standard synchronous one. We have selected to use the synchronous protocol as our standard one due to the fact that many IPs operate in synchronous mode. The operands in memory are fetched by the in/out-controller and passed to the in-buffer. The data in the in-buffer is passed to the IP via protocol transformer. The results from the IP are stored in the out-buffer, and then passed to memory under the control of the in/out-controller.

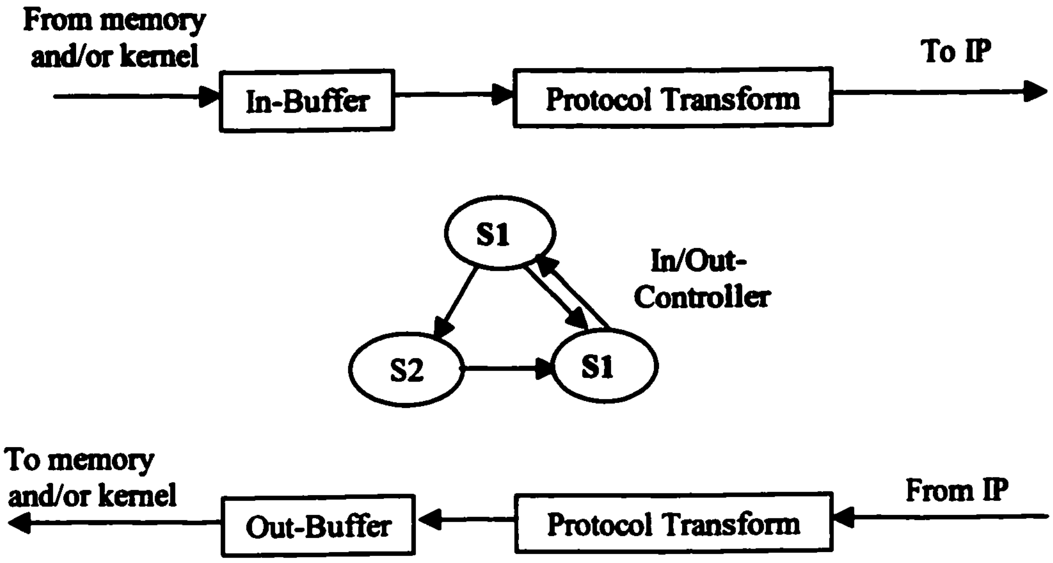


Figure 2.1: General interface method

By changing the in/out-controller and/or by inserting/eliminating the buffers, various interface methods are possible. The factors we consider in deciding a specific interface method for an IP are as follows. First, input and output characteristics of IP are considered: the number of input (output) ports, the input (output) data rate, the number of input (output) data, the latency from input to output, and whether the IP is pipelined or not. Second, parallel execution is considered. Parallel execution enables additional reduction in execution time by overlapping the execution of kernel with that of IP's as illustrated in Figure 2.2.

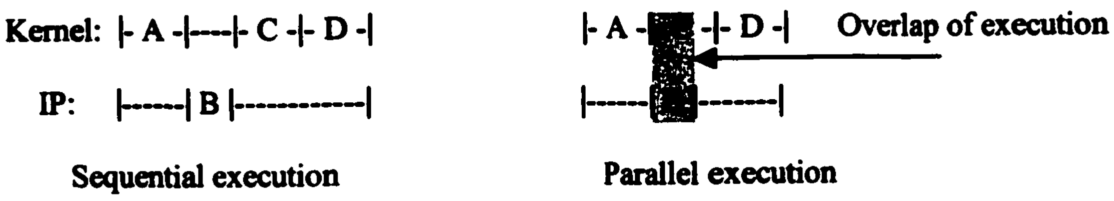


Figure 2.2: Execution of four code segments, A – D, in kernel and IP

According to the above analyzing, we support the interface type shown in Figure 2.3. It has very powerful performance. For this type, the in/out-controller is implemented in a FSM. The buffer enables to handle an IP having more than two in (out)-ports by assigning a buffer to each port and to transfer high-rate in/out-data. In fact, high-rate transfer occurs between the buffer and the IP, while low-rate transfer occurs between the buffer and FSM to fill the data required to start the IP into the buffer and to move results from the buffer to memory after the IP finishes its job. In addition, parallel execution without memory contention is possible because the IP accesses the buffers instead of memory.

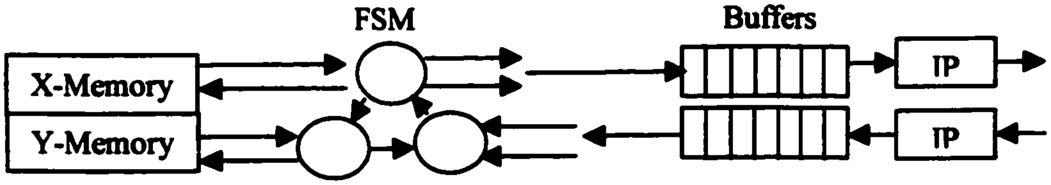


Figure 2.3: Hardware interface with buffer

In our specific IP core design, we need to transfer data between the high-speed system bus and low-speed peripheral devices (such as a A/D converter). Under this condition, the input data rate is different from output data rate. We use a kind of elastic buffer – FIFO to solve this problem. FIFO can form a bridge between subsystems with different clock rates and access requirements. A detail will be given in Chapter 4.

2.4 Device select signal

When interfacing a peripheral to a system master, each device needs to be given a unique address. The microprocessor selects the device it wants to communicate with by supplying a suitable 'device-selection' pulse to the device's enable line. There are normally many addresses available for the designer to use in deriving device-selection signal. One method of decoding is memory mapping, in which the microprocessor deals with a device as one or more memory locations. With this method, the communication is achieved by executing instructions such as 'move to' or 'load from' memory. Another method of decoding is input/output mapping, which is supported by some, but not all, microprocessors. With this method, the microprocessor executes IN and OUT instructions to produce and address of an input or an output device rather than an address of memory.

There are different ways of deriving selection signals. They depend on several factors, such as the type of microprocessor and the available addresses. We combining two types of signals produced by a microprocessor to generate device-selection signals. The first is the *device control signal*, which allows the microprocessor to control the communication process; the second signal is the *addressing signal*, which allows the microprocessor to identify a device with which it wishes to communicate.

Device-control Signal + Addressing Signal = Device-selection Signal

2.4.1 Deriving device-control signals

We consider two signals, which identify the direction of transfer of information, the read and write signals. They permit a microprocessor to indicate whether it wishes to read from an input or to write to an output destination. During communications, the microprocessor generates either of these two signals together with an addressing signal to identify the device it wishes to communicate with.

2.4.2 Deriving addressing signals

A microprocessor uses an addressing signal to identify the location of a device within the addressing area. When interfacing a device to a microprocessor, the designer needs to consult the memory map of the microprocessor to find the area available for use in deriving addressing signals. The size and boundaries of the area vary from one microprocessor to another. A large number of devices can be addressed by a single unique location. In some case, however, a block of sequential addresses can be reserved for the interface. In such cases, one or more address locations are allocated to each device to satisfy that interface requirements may grow or change in future. Therefore, the design depends on several factors, such as the size of the unused addressing area, the minimum number of addresses required per device, and future requirements.

Figure 2.4 shows a circuit, which can be used to generate the two signals. The address needed to set X1 to a low state is represented by 1000 0111 1111 1xxx, which corresponds to addresses 87F8H and 87FFH. Since output X2 is active only when both X1 and A2 are low, an addressing signal is produced through X2 when the address lines are represented by 1000 0111 1111 10xx, addresses 87F8H to 87FBH. Similarly, X3 is active only when A2 is high and X1 is low. When the address lines are 1000 0111 1111 11xx. This represents addresses 87FCH to 87FFH.

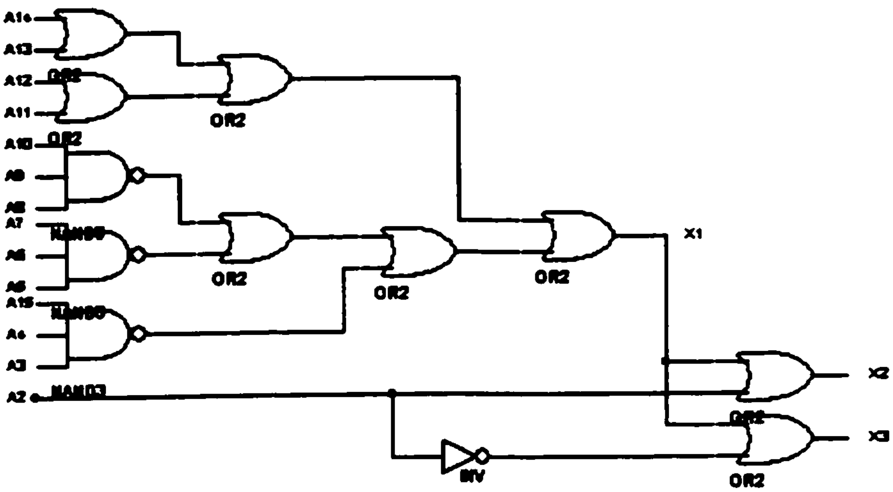


Figure 2.4: deriving addressing signals

2.5 Device synchronization methods

Three basic approaches are commonly used in synchronization design. They are program-controlled synchronization, interrupt-controlled synchronization and interface-controlled synchronization.

2.5.1 Program-controlled synchronization

The device is informed when data communication is required by setting one or more flags in a command register. It then synchronizes its operation to that of the bus and indicates that it's ready to supply or accept the data by setting one or more flags in a status register. The interfaced device behaves as a reluctant slave. The command and status registers are directly synchronized devices that are part of the bus interface. The interface must implement the slave talker and listener data transfer control logic for both the device and the two synchronization registers. Figure 2.5 shows the flowchart.

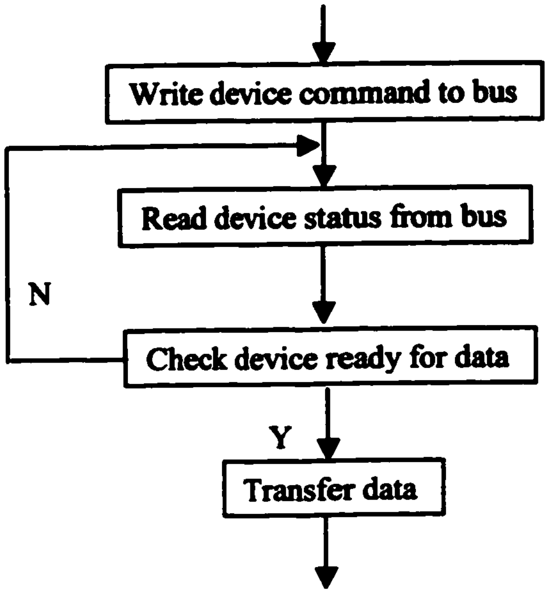

Write device command to bus
Read device status from bus
N
Check device ready for data
Y
Transfer data

Figure 2.5: The flowchart of program-controlled synchronization

2.5.2 Interrupt-controlled synchronization

Where the device wishes to be a slave that has the ability to initiate bus communications, in addition to the slave talker and listener data transfer control logic, the interface must implement the slave device synchronization control logic of the bus. When the slave device is ready for bus communication, the device synchronization logic sends an interrupt request to the microprocessor, which then organizes the appropriate data transfers over the bus. The interface device behaves as a partially independent slave.

2.5.3 Interface-controlled synchronization

In certain circumstances, the interface design must allow the device itself to control the transfer of data over the bus; where the interrupt response line would be too long for a slave device; if the data transfer rate is too slow when under control of the master; when master is overloaded with other tasks that must have priority over bus communication with a slave device, the interfaced device behaves as an independent master and the interface must implement the bus allocation control logic and the master and slave, talker and/or listener, data transfer control logic. The device and its interface are synchronized under the control of local synchronization logic. Program-controlled or interrupt-controlled synchronization is used to indicate completion of the transfer to the other master device that is talking part in the communication. The method is known as DMA if the communication is between an I/O device and the memory of a microprocessor. system.

In this thesis, we will apply the program-controlled synchronization to the IP core design.

2.6 Interfacing A/D converters to microprocessor

The operation of numerous systems is based on obtaining a single result regularly through an A/D converter. This result is used as an input to a software task that is executed immediately after the end of the A/D conversion process. An A/D converter provides the microprocessor with a digital value equivalent to the amplitude of the analogue signal at the time of conversion.

2.6.1 Operating A/D converters

Most A/D converters are designed to accept a command from a controlling device before commencing a digitization process. This command is called the start of conversion (SC). It may be in the form of a pulse, a logic level, or a software command. The provision of such a command is very useful, particularly when digitizing waveforms. It allows a controlling device to read the magnitude of an analogue signal at a particular instant of time. It also makes it possible for that device to reconstruct the signal.

Consider two methods of operating an A/D converter. The first is operation under the control of a microprocessor. In this method, the microprocessor starts an A/D converter by supplying the SC command. The second method is often referred to as the free-running operation. It is based on operating the converter without the intervention of the microprocessor. In this method, the SC command can be supplied to the A/D converter from external, or onboard, circuitry. Alternatively, the converter can be connected so that it does not need an SC command, i.e. one conversion starts immediately after the completion of another. Use can also be made of a converter that does not need an SC signal. Many A/D converters produce an end-of-conversion (EOC) signal to indicate the completion of a conversion. A microprocessor can therefore examine that signal to detect when to read the digital result.

In the thesis, the A/D converter uses the free-running approach to digitize a value without receiving commands from a microprocessor. Such an approach removes from the microprocessor the need for generating the timing signals required for the operation of the converter. This, however, implies that the microprocessor will no longer be able to specify the instant of starting a conversion.

In such a situation, the microprocessor controls the data transfer through FIFO. The A/D converter has three signals connected to the microprocessor: one is enable signal to select the A/D converter; one is FIFO control signal to control FIFO to read or write data; the other one is the transferred FIFO data. The detail will be discussed in Chapter 4.

2.6.2 Reducing the involvement of the microprocessor

There are two methods to fulfill the requirement. One method is to allow the EOC signal to interrupt the microprocessor. This approach permits the microprocessor to read the digital value as soon as it is obtained without introducing unnecessary delays. The interrupt activates an interrupt-serve routine, which can be very short, as all it needs to do is to read the digital result and store it in a specified memory location. The interrupt routine can also set a software flag to indicate to other parts of the program the availability of a new value. The procedure consists of two parts, the initialization and the interrupt-service routine. The interrupt-controlled approach, however, can only take place if there is access to a hardware interrupt line of the microprocessor to which the EOC line can be connected. The use of correctly organized interrupts is attractive, especially in real-time control applications. However, explains that the use of interrupts may not be attractive when the rate of their occurrence is very high.

Instead of employing the interrupt-controlled approach, a change in the state of the EOC signal can be detected by the use of the program-controlled input method. One way of applying this method is to program the microprocessor to continuously interrogate the EOC line, seeking the detection of a change of state. This, however, leaves the microprocessor waiting for the conversion to complete. If the conversion time is long, then enhanced performance can be realized if the microprocessor is permitted to execute useful tasks while the conversion is taking place.

There are two methods that a microprocessor adopt the program-controlled approach to detect the EOC signal generated by an A/D converter and be able to execute useful tasks during the conversion time.

Solution 1:

One way of fulfilling the requirement is to adopt an approach based on estimating the time needed for the converter to perform a conversion. The microprocessor can therefore execute a suitable routine after supplying the SC command to the converter. When this routine is complete, the microprocessor can examine the EOC line before reading the digital value.

The success of the implementation of this approach depends on the correct estimation of the conversion time. In a ramp A/D converter, the conversion time depends on the magnitude of the analogue-input signal, which is unknown prior to the conversion. In other conversion methods, such as when using a hardware-based successive-approximation conversion, the conversion time can be estimated by the designer. If it takes a relatively long time, the designer can arrange the software to make the microprocessor execute a suitable routine during the conversion period.

Solution 2:

It is possible to adopt an alternative approach which is independent of whether or not the conversion time can be estimated. It is base on storing the digitized result after the end of conversion without the intervention of the microprocessor. If the converter has no storage ability, then an external latch can be included. The operation of the converter can be initiated by an enable command from the microprocessor. When the conversion is accomplished, the EOC signal instructs a latch to store the results without the intervention of the microprocessor. The up-to-date conversion result is therefore held in the latch and will be updated after completing the next conversion.

With such an approach, the microprocessor does not need to read the digital value as soon as it is produced. Instead, it can command the start of conversion and then execute a suitable routine. The time required to process that routine could be longer than the maximum specified conversion time. Since the interval between the microprocessor applying the SC command and being ready to read the result is relatively long, there will be no need to examine the EOC line. This reduces the processing time and frees the associated input line of the microprocessor to be used for another purpose. Therefore, this solution is adopted in the thesis.

2.7 The operation of the bus

Devices wishing to use a bus must adhere to a set of rules or protocols. These protocols define the operation of the bus in a precise way. Each bus has a different set of protocols, although there are overall similarities. In general, bus operation can be divided into three distinct phases.

- Device synchronization. The slave device sends a request to the master device asking for data transfer to occur. The request is acknowledged and its priority examined. When the request has the highest priority and the master is ready, it initiates a data transfer over the bus.

- Bus allocation. The master requests use of the bus. The request is acknowledged and its priority examined by the arbitration logic. When request has the highest priority and the bus is not being used by any other device, the master is given control of the bus by the allocation logic.

- Data transfer. The master signals to the slave that data transfer will take place. On completion, the master relinquishes control of the bus and considers the next highest priority request.

During each phase of the bus operation, the master and slave devices are required to generate the relevant bus control signals at particular times and in a particular sequence. The precise nature of the control signal interactions is defined by the bus protocols. Each device has a bus interface, which interprets and generates the appropriate control signals in the correct sequence. The bus interface is a sequential logic circuit whose logical state reflects both the condition of the bus and the device that it interfaces to the bus.

In the following chapter, we will have an overview of three bus interconnect architectures. One of them will be selected to design the interface IP core.

Chapter 3

Interconnect Architectures

3.1 Background

Interconnect on a chip is used for signal communication, as well as power, ground and clock distribution. Interconnect effects that impact performance, compromise signal integrity, and increase power dissipation are becoming more pronounced as technology moves deeper into sub-micron feature sizes, and designs are operated at higher frequencies. In this work, we investigate inter-module signal communication techniques for high performance in the context of t he SoC application domain.

Industry experts agree that IP integration is the ideal technology for rapid SoC design development in a cost-efficient and fast time-to-market manner. This technology has not been widely adopted yet (full-custom designs are still favored) mostly because many issues still need to be resolved in terms of interfacing these components together. One idea, now finding wide acceptance in the community, is to register-bound IP's, thus temporally decoupling the inside of the block from the outside. This will allow IP's to be treated as "black-boxes" that are immune to glitches at the input, and do not generate any at their outputs, as well as to support "plug-and-play" where system developers can substitute one black box IP by another, given that they have the same functionality.

In an IP-based SoC environment the interconnect strategy should be able to support:

a. Connection of heterogeneous components with a relatively large number of pins in a "point-point" fashion;

b. "Plug-and-play" and the preservation of the synchronous design assumptions.

In this section, we introduce concepts and terminology associated with on-chip interconnect architectures and describe and compare some popular interconnect architecture used in commercial SoC designs.

The interconnect architecture topology consists of a network of shared and dedicated interconnection channels, to which various SoC components are connected. These include (1) masters, components that can initiate a communication transaction (e.g., CPUs, DSPs, DMA controllers etc.), and (2) slaves, components that merely respond to transactions initiated by a master (e.g., on-chip memories). When the topology consists of multiple channels, bridges are employed to interconnect the necessary channels.

The interconnect architecture plays a key role in SoC design by enabling efficient integration of heterogeneous system components (CPUs, DSPs, application specific cores, memories, custom logic, etc). In addition, the interconnect architecture also significantly influences the system performance and power consumption directly, since the delay and power in global interconnect is known to be an increasing bottleneck with shrinking feature sizes, and through its significant indirect impact on the computation time and power consumption in the system components.

I have read three SoC interconnect specifications: IBM CoreConnect, ARM AMBA and Silicore Corp Wishbone. All of these address the same basic goal: connecting IP cores. They all provide basic handshaking and variable data bus sizes. None specifies a clock frequency, which could be a problem when connecting cores from different vendors.

The purpose of this review is to choose a SoC bus for the design, that we would adopt and use in any core development. Standardizing on a common SoC will help us as a community to produce cores that can be easily integrated. Bridges to other SoC standards could be developed and would allow for our cores to be used with other SoC standards as well.

3.2 CoreConnect

It appeared to be the most complete set of documentation and technically very well though through. IBM has provided specs for each possible building block PLB, OPB, DCR1, Arbiter and 64+ bit extensions. IBM also provides a testsuite could not find any information if they actually charge for it or not).

3.2.1 Logical Bus Structure

Figure 3.1 illustrates the structure of CoreConnect bus.

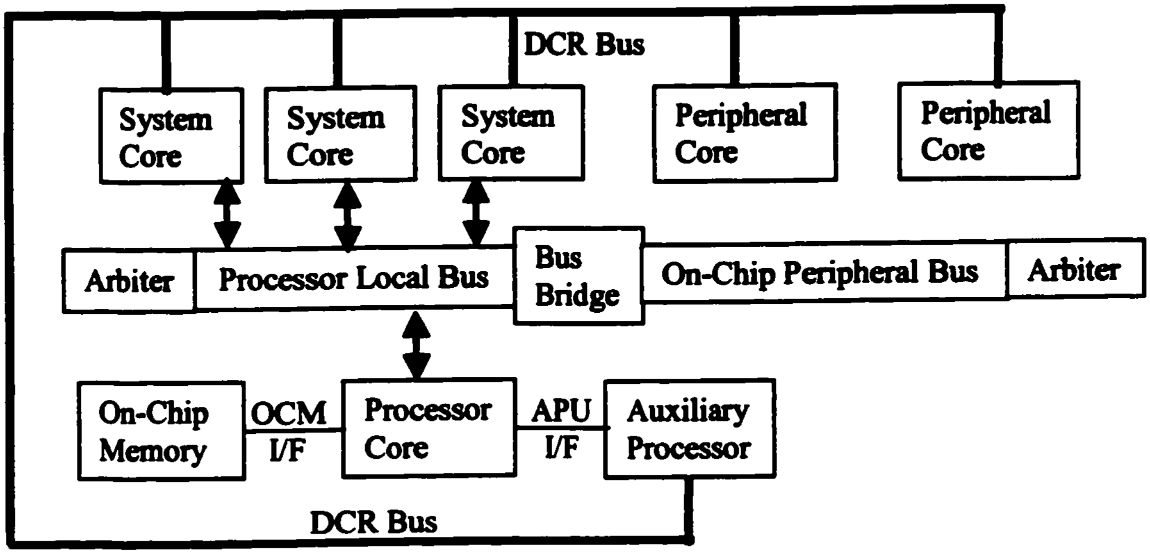


Figure 3.1: CoreConnect bus architecture

3.2.2 Technical Details

Below is a summary of main features of each CoreConnect bus.

3.2.2.1 Processor Local Bus

- High Performance bus
- Overlapped read and write (up to two transfers per cycle)
- Split transfer support
- Address pipelining (reduces latency)
- Separate read and write data
- 32-64+ bit data bus with 32 bit address space
- Support for 16-64 byte bursts
- Supports byte enabling (unaligned and 3 byte transfers)
- Late and hidden arbitration (reduces latency)
- 4 levels of arbitration priority

- Special DMA modes, such a flyby and memory to memory
- Address and data phase throttling
- Latency timer (ensures latency is kept to a desired level)

3.2.2.2 On-Chip Peripheral Bus

- Multiple masters
- 32 bit address space
- Separate read and write data bus
- 8-32 bit data bus
- Dynamic bus sizing
- Retry support
- Burst support
- DAM support
- Devices may be memory mapped (DMA support)
- Bus time out function (in arbiter)
- Arbitration support, REQ, GNT and LOCK
- Bus parking support

3.2.3 Applications

CoreConnect defines a clear structure for all system components and how they connect. The DRC bus wraps in a daisy chain configuration through all components attached to the Processor Local Bus.

I can see CoreConnect to be an important part of a true high performance system, like a workstation. My feeling is that CoreConnect might be too complicated and offer too many features that will be unused in simple embedded applications.

3.3 AMBA

AMBA is very similar to CoreConnect. ARM includes specifications for AHB, ASB, and APB. The "A" in the abbreviations stands for "Advanced". Descriptions of arbitration and 64+ bit extensions are integrated in to the main specifications.

3.3.1 Logical Bus Structure

Figure 3.2 illustrates the structure of AMBA bus.

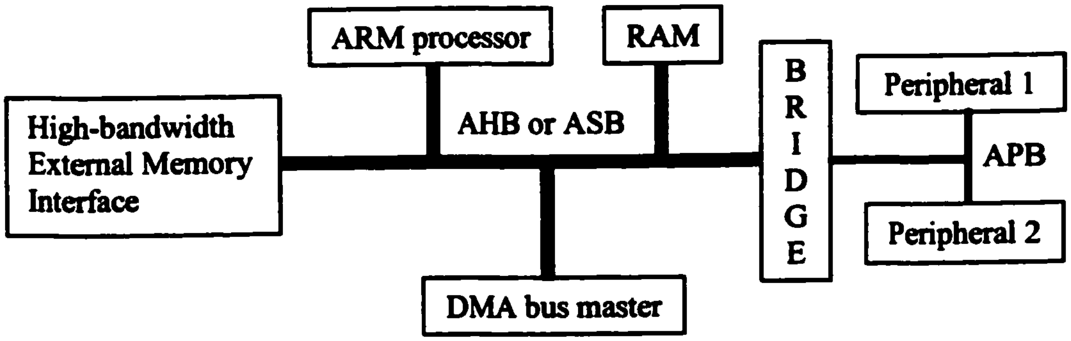


Figure 3.2: A typical architecture of an AMBA based SoC

The high performance bus can be either AHB or ASB. Both service the same goal: High Performance System Interconnect. A bridge from AHB or ASB is required to interface to APB. The function the bridge provides is a simpler interface. Any latency presented by low performance peripherals is reflected by the bridge to the high performance

(AHB/ASB) bus. The bridge itself appears to be a simple APB bus master that addresses the attached slaves and controls them through a subset of the control signals available on the high performance busses.

3.3.2 Technical Details

Below is a summary of main features of each AMBA bus.

3.3.2.1 AHB

The AHB is the advanced system bus. Its main purpose is for interconnecting high performance, high clock frequency devices, such as CPU, DMA and DSP. Its main features are:

- High Performance Bus (New Generation Bus)
- Multi Master
- Split transfers
- Single cycle bus master handover
- Non tristate implementation
- 32 - 128+ bit bus width
- Includes an access protection mechanism, to distinguish between such access as privileged and non privileged modes, instruction and data fetch, etc.

3.3.2.2 ASB

The ASB is the general-purpose system bus. It is a high performance interconnection for micro controller and system peripherals. The main features are:

- First Generation System Bus
- Multiple Masters

- Burst Transfers
- Pipeline Transfers
- 32 - 128+ bit bus width

3.3.2.3 APB

The APB is the peripheral interconnect bus. Focus here was minimal power consumption and ease of use. The main features include:

- Low performance, low power peripheral bus
- Single Master
- Very Simple, only 4 control signals (plus clock and reset)
- 32 bit address space
- Up to 32 bit data bus
- Separate read and write data bus

3.3.3 Applications

AMBA is a basic SoC bus divided into three different sub busses. Depending on requirements, the system designer has to choose which of the three busses he will interface to. All three busses consist of an address and one or multiple data phases. Technically I can see it as sufficient architecture for small-embedded systems, which are not necessarily performance driven.

3.4 Wishbone

The Wishbone specification is comprised of *Rules*, *Suggestions*, *Permissions*, and *Observation* etc. Not having implemented an interface bus, it is hard to say how complete it might be and whether the bus will cover all needs or not. This is especially true for wishbone.

3.4.1 Logical Bus Structure

Figure 3.3 illustrates the structure of wishbone bus.

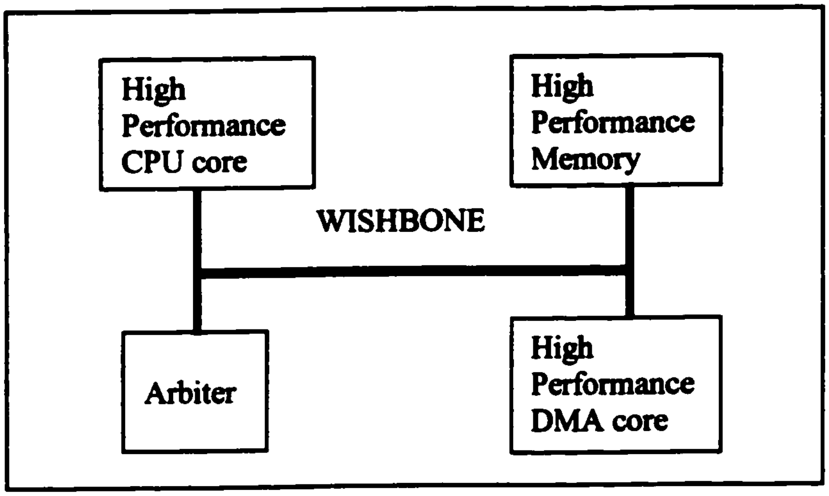


Figure 3.3: The structure of wishbone bus

Wishbone architecture is as simple as one can imagine. A system with many components, might want to include two wishbone interfaces: one for high performance blocks, and one for low performance peripherals.

3.4.2 Technical Details

- One Bus Architecture for all applications
- Simple, compact architecture
- Multi master support

- 64 bit address space
- 8 - 64 bit data bus (expandable)
- Single read and write cycles
- Event cycles
- Supports retry
- Supports memory mapped, FIFO and crossbar interface
- Throttling of data for slower devices provided
- Arbitration defined by the end user

3.4.3 Applications

Wishbone appears the simplest of the three busses I have reviewed. It defines only one bus - a high-speed bus. Users of wishbone might have to create their own substandard of wishbone. Additional features and functionality might also have to be added. Therefore, it might leave a few things to wish for, when implementing high performance systems.

3.5 Summary

Table 3.1 shows a technical comparison among CoreConnect, ARM AMBA and Wishbone architectures. All three busses are fully synchronous, using the rising edge of the clock to drive and sample all signals. There is almost no difference in basic operations between the busses. The most differences are in the feature set provided and completeness/relaxation of the specification.

Table 3.1: The comparison among architectures

	IBM CoreConnect Processor Local Bus	**ARM AMBA 2.0**	**Silicore Corp. Wishbone**
Bus Architecture	32-, 64-, and 128-bits Extendable to 256-bits	32-, 64-, and 128-bits	Up to 64-bits
Data Buses	Separate Read and Write	Separate Read and Write	Combined Read and Write
Key Capabilities	Multiple Bus Masters 4 Deep Read Pipelining 2 Deep Write Pipelining Split Transactions Burst Transfers Line Transfers	Multiple Bus Masters Pipelining Split Transactions Burst Transfers Line Transfers	One high-speed bus Single Read/Write cycles Supports retry Arbitration defined by the end user
	On-Chip Peripheral Bus	**Advance Peripheral Bus**	**No Peripheral Bus**
Masters Supported	Supports Multiple Masters	The APB Bridge	
Bridge function	Master on PLB or OPB	APB master only	
Data Bus	Separate Read and Write	Separate or 3-state	

Both CoreConnect and AMBA, offer a choice of system busses to the designer. An integrator might face a problem, when he tries to connect devices designed for the different portions of those interconnections. Bridges might be required to build a complete system. With wishbone, all cores connect to the same standard interface. A system designer may choose to implement two wishbone interfaces in a microprocessor core, one for high speed low latency devices and one for low speed, low performance devices.

At the end I feel it would be a wise choice to adopt wishbone as a primary interface to our cores. It's signaling appears to be very intuitive and should be easily adopted to the other interfaces when needed.

According to our specific system requirement, my solution to this interface problem is Advanced Peripheral Bus (APB), which is based on ARM AMBA specification. AMBA is an on-chip bus standard that defines a signal protocol for the connection of multiple blocks in an on-chip system. It provides the "digital glue" that binds IP cores together and is a key enabler of IP reuse. By designing to the standard AMBA interface, IP developers an implement and test modules without prior knowledge of the system into which the component will be finally integrated.

Chapter 4

Design Specification

4.1 Introduction

According to the above comparison, we decide to adopt APB as the design architecture. APB is part of the ARM AMBA hierarchy of buses and is optimized for minimal power consumption and reduced interface complexity to support peripheral functions. It is used to interface to any peripherals which are low-bandwidth and do not require the high performance of a pipelined bus interface. APB can be used in conjunction with either version of the system bus. Connection to the main system bus is via a system-to-peripheral bus, APB bridge. Figure 4.1 shows the APB in a typical AMBA system.

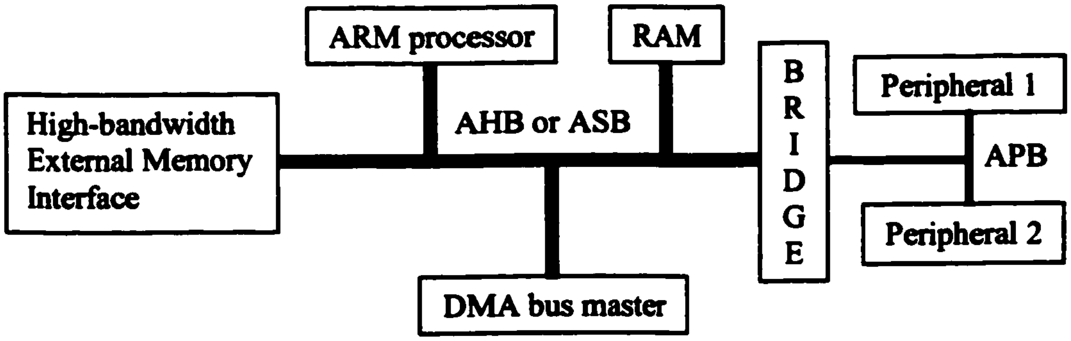


Figure 4.1: The APB in a typical AMBA system

APB has the following advantages:

- A simple bus

    a. Unpipelined architecture;

    b. Easy to implement with all the peripherals acting as slaves;

    c. Low gate count.

- Low power

    a. Reduced loading of the main system bus by isolating peripherals behind the bridge;

    b. Peripheral bus signals only active during low bandwidth peripheral transfers.

APB is recommended for:

- Simple register-mapped slave devices;
- Very low power interfaces where clocks can not be globally routed;
- Grouping narrow-bus peripherals to avoid loading the system bus.

4.2 Architecture

The whole interface IP core consists of two units: APB bridge and APB slaves. FIFO buffers transfer data between APB bridge and APB slave.

4.2.1 APB Bridge

In this IP core design, the main part is APB bridge, which is required to convert AHB transfers into a suitable format for the slave devices on the APB. The bridge performs the following functions:

- Latches the address and holds it valid throughout the transfer.
- Decodes the address and generates a peripheral select, PSELx. Only one select signal can be active during a transfer.
- Drives the data onto the APB for a write transfer.
- Drives the APB data onto the system bus for a real transfer.
- Generates a timing strobe, PENABLE, for the transfer.

The APB bridge behaves like an AHB slave; it uses one clock domain for both AHB and APB bus side. Figure 4.2 shows the architecture of the APB bridge.

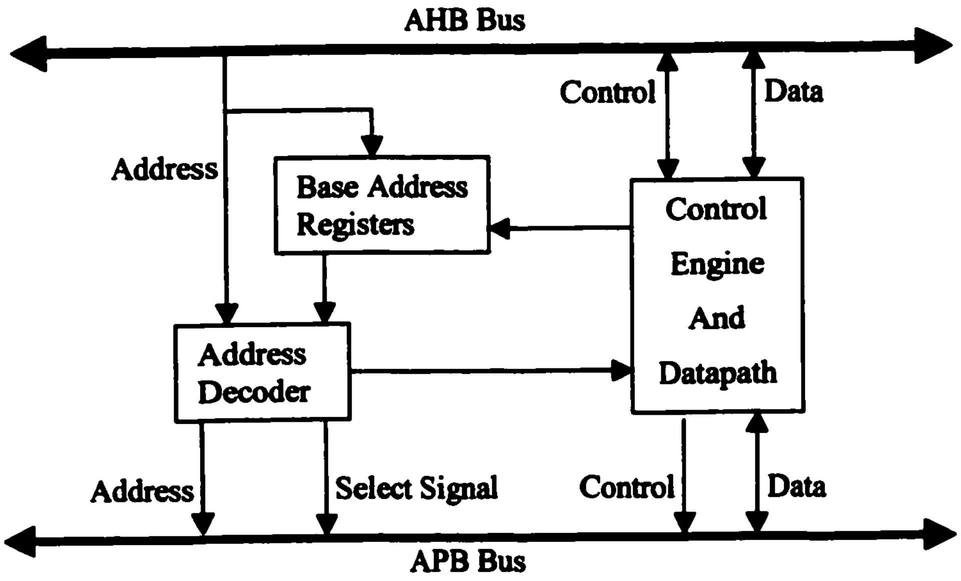


Figure 4.2: The architecture of the APB bridge

4.2.1.1 Control engine

Control engine has the following functions:

During AHB write, the data flows from AHB to the local peripheral.

- Interacts with the AHB to execute a write.
- Transfers data from AHB to APB.
- Interacts with the APB to write data into the APB-to-peripheral FIFO.

During AHB read, the data flows from peripheral to the AHB bus.

- Interacts with the AHB to execute a read.
- Interacts with the APB to fetch data from the peripheral-to-APB FIFO.
- Transfers the data from the APB to the AHB.

4.2.1.2 Base address register

Base address register is set to reserve system address space to map to the peripherals and the descriptor FIFO buffer. ARM recommends word (32-bit) alignment of peripheral registers even if they are 16-bit or 8-bit peripherals. The main reason for this is to make the hardware interface easier to implement.

4.2.1.3 Address decoder

Address decoder is used to perform a centralized address decoding function, which improves the portability of peripherals, by making them independent of the system memory map. The address decoder provides a select signal, *PSELx*, for each slave on the bus. The select signal is a combinatorial decode of the high-order address signals, and simple address decoding schemes are encourage to avoid complex decode logic and to ensure high-speed operation.

Figure 4.3 shows the APB signal interface of an APB bridge.

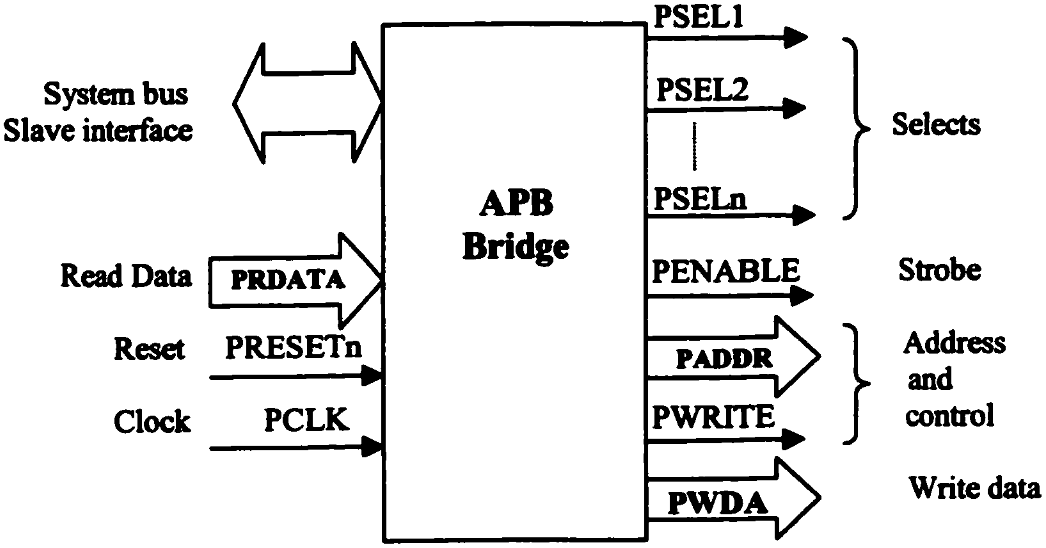


Figure 4.3: APB bridge interface diagram

4.2.2 APB Slave

All other modules on the APB are APB slaves. The APB slaves have the following interface specification:

- Address and control valid throughout the access (unpipelined);
- Zero-power interface during non-peripheral bus activity (peripheral bus is static when not in use);
- Timing can be provided by decode with strobe timing (unlocked interface);

- Write data valid for the whole access (allowing glitch-free transparent latch implementations).

Figure 4.4 shows the signal interface of an APB slave.

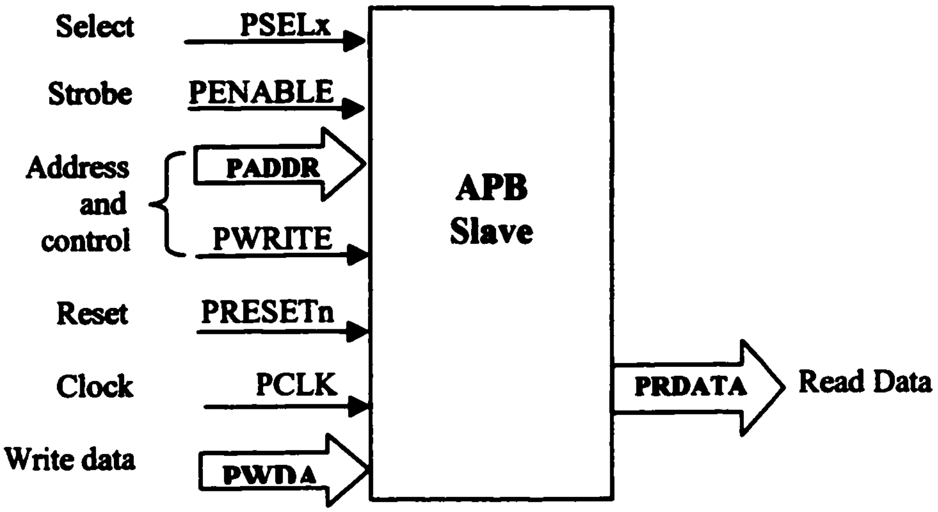


Figure 4.4: APB slave interface description

4.2.3 FIFO

In digital systems, it is sometimes required to transfer data between two asynchronous clock regimes; that is, between first and second groups of logic where the first group is controlled by a first clock signal and the second group is controlled by a second clock signal which is not synchronized with the first clock signal. In such a system, if no special precautions are taken, there is a possibility that the output data from the first clock regime may change at approximately the same time as it is transferred into the second clock regime. Because of variations in tolerances, the individual bits of a data word transferred in parallel between the first and second clock regimes may come from different clock beats of the first clock regime, resulting in corruption of the data.

A known solution to this problem is to use a FIFO memory to buffer data between the two clock regimes. A FIFO is a special type of buffer. The name FIFO stands for first in first out and means that the data written into the buffer first comes out of it first. FIFOs are usually used for domain crossing, and are therefore dual clock designs. Figure 4.5 illustrates the data flow in a FIFO.

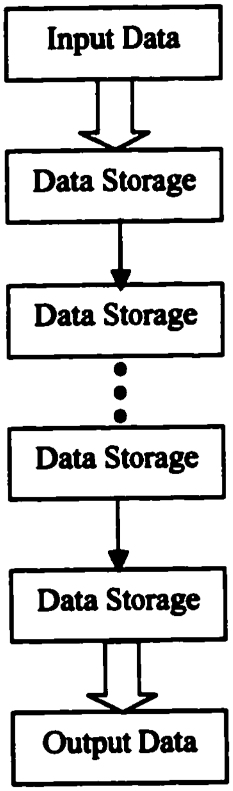


Figure 4.5: First-In First-Out data flow

In this FIFO design, there is no dependence between the writing and reading of data. Simultaneous writing and reading are possible in overlapping fashion or successively. This means that two systems with different frequencies can be connected to the FIFO. Synchronizing the two systems is taken care of in the FIFO. The block diagram in Figure 4.6 shows the control lines of a FIFO.

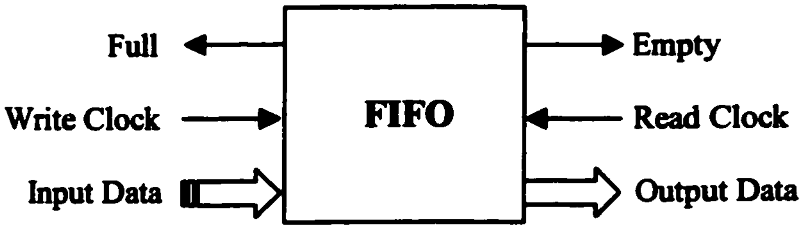


Figure 4.6: Connections of a FIFO

In order to operate a FIFO with independent Read and Write clocks, some asynchronous arbitration logic is needed to determine the status flags. The previous EMPTY/FULL generation logic and associated flip-flops are no longer reliable, because they are now asynchronous with respect to one another, since EMPTY is clocked by the read clock, and FULL is clocked by the write clock. Using Gray-code to synchronize the clock will solve this problem. The detail will be given in section 4.6.

4.2.4 Connection

Most processors are considerably faster than peripherals that are connected to them. FIFOs can be used so that the processing speed of a processor need not be reduced when it exchanges data with a peripheral. Even if the peripheral is sometimes faster than the processor, a FIFO can again be used to resolve the problem. Different variations of circuitry are possible, depending on the particular problem.

In this case, the processor reads input data over an A/D converter (see Figure 4.7). A FIFO can buffer a certain amount of input data, then set a flag so that the processor reads the data.

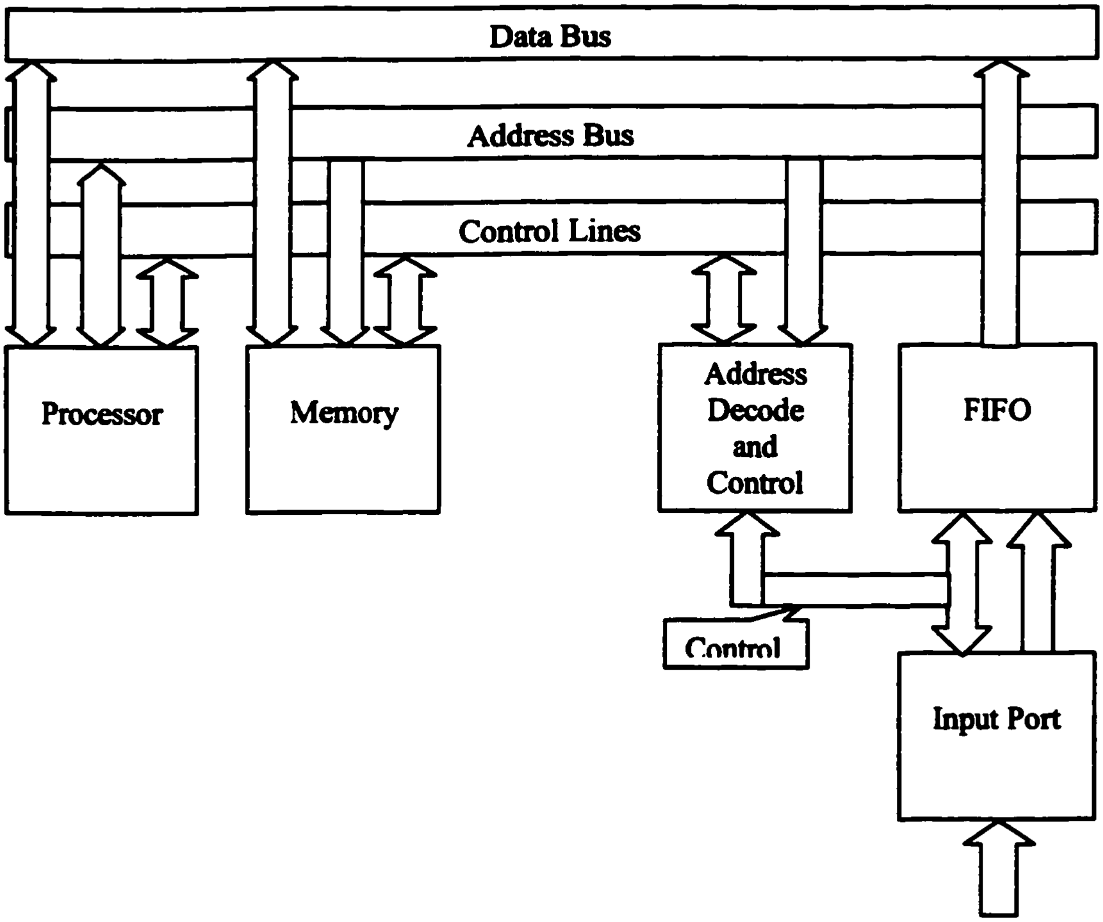


Figure 4.7: Connection of an A/D with a FIFO

4.3 I/O ports

This section contains an overview of all signals used in this interface IP core. It includes AHB signal list, APB bridge signal list and APB slave signal list. In order to completely implement the APB bridge, Not only do I connect an A/D converter to APB through FIFO, but also I connect a read-only device, like a D/A converter, to APB too.

Table 4.1, 4.2 and 4.3 show the AHB signal list, APB bridge signal list and APB slave signal list respectively.

Table 4.1: AHB signal list

Name	**Width**	**Source**	**Description**
hclk	1	Master	System clock, which times all bus transfers. All signal timings are related to the rising edge of hclk.
hresetn	1	Master	Reset signal, which is active low and is used to reset the system and the bus.
Haddr	32	Master	The 32-bit system address bus.
Htrans	2	Master	Indicate the type of the current transfer.
hwrite	1	Master	Transfer direction. When HIGH it indicates a write transfer, when LOW a read transfer.
hsize	3	Master	Transfer size, which is typically byte (8-bit), halfword (16-bit) or word (32-bit).
hwdata	32	Master	AHB write data bus.
Hrdata	32	Bridge	AHB read data bus.
Hready	1	Bridge	Transfer done.

Table 4.2: APB bridge signal list

Name	**Width**	**Source**	**Description**
paddr	8	AHB	APB address signal.
psel0	1	AHB	APB slave0 (A/D converter) select signal, coming from the secondary decoder.
psel1	1	AHB	APB slave1 (D/A converter) select signal, same as psel0.
penable	1	Bridge	APB strobe signal, used to time all accesses on the APB bus.
pwrite	1	AHB	When HIGH it indicates a APB write access, when LOW a read access.
prdata	32	APB slave	APB read data bus, output from slave0 (A/D converter).
pwdata	32	AHB	APB write data bus, input to slave1 (D/A converter).
sbe	4	AHB	Write byte enable, used to control FIFO.
devready0	1	APB slave	APB slave0 (A/D converter) is ready for the transfer.
devready1	1	APB slave	APB slave1 (D/A converter) is ready for the transfer.

Table 4.3: APB slave signal list

	Name	**Width**	**Source**	**Description**
Slave0 **(A/D converter)**	adenable	1	FIFO	A/D strobe
	ffwr	1	A/D	A/D write FIFO clock
	fffulla	1	FIFO	FIFO is full
	ffdata	32	A/D	A/D write FIFO data
Slave1 **(D/A converter)**	daenable	1	FIFO	D/A strobe
	daclk	1	D/A	D/A read FIFO clock
	ffemptyd	1	FIFO	FIFO is empty
	dadata	32	FIFO	D/A read FIFO data

4.4 Operation

The APB state diagram, shown in Figure 4.8, can be used to represent the activity of the peripheral bus.

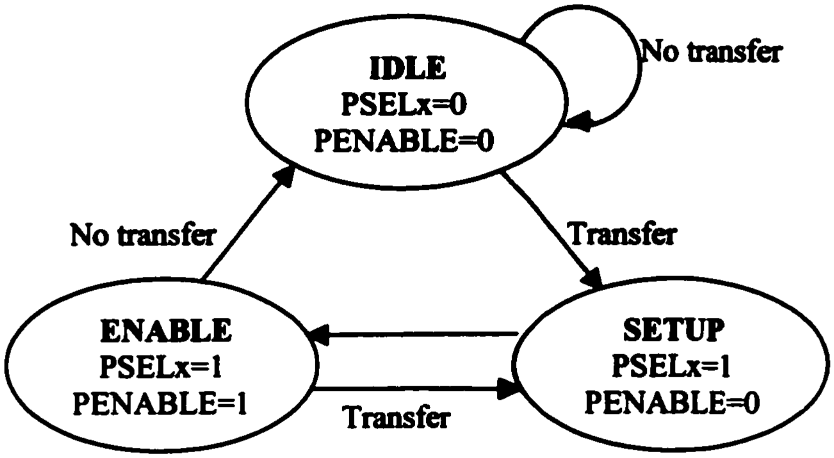


Figure 4.8: APB state diagram

Operation of the state machine is through the three states described below:

IDLE The default state for the peripheral bus.

SETUP When a transfer is required the bus moves into the SETUP state, where the appropriate select signal, *PSELx*, is asserted. The bus only remains in the SETUP state for one clock cycle and will always move to the ENABLE state on the next rising edge of the clock.

ENABLE In the ENABLE state the enable signal, *PENABLE* is asserted. The address, Write and select signals all remain stable during the transition from the SETUP to ENABLE state. The ENABLE state also only last for a single clock cycle and after this state the Bus will return to the IDLE state if no further transfers are required. Alternatively, if another transfer is to follow then the bus will move directly to the SETUP state. It is acceptable for the address, write and select signals to glitch during a transition from the ENABLE to SETUP states.

4.5 Registers

Table 4.4 describes 32-bit AHB address space mapping.

Table 4.4: AHB address space mapping

Bit #	**Name**	**Description**
[31:28]	APB bridge access signal	When bits are set to 1000, AHB is going to access APB bridge.
[27:11]	APB slave select signal	Which slave is selected to have a transform. In this case, A/D converter is set to 00000H; D/A converter is set to 00001H.
[9:2]	APB address signal	The address mapping of selected slave. For both A/D and D/A converters, when bits are set to 00H, the FIFO data output port is selected; when bits are set to 01H, the FIFO status output is selected.
[1:0]	Write byte enable	Which bit is enabled to write.

4.6 Gray-code counters

4.6.1 Synchronization: Solving the reliability problem

In the FIFO problem, we need to sample the value of a counter with a clock that is asynchronous to the AHB system clock. Thus we could land up in a situation where the counter is changing from, say FFFF to 0000, and every single bit goes metastable. However, this means the FIFO will not work. Synchronization will save counter samples from going metastable, but we still may get sampled values that are wildly off the mark.

The important thing that we must do is to make sure that not all bits of the counter can change simultaneously. In fact we have to make sure that precisely one bit changes every time the counter increments. This implies that if you catch the counter transitioning, only one bit may be in error. This is the best we can do, since we need at least one bit transition if the counter itself is to work. What we need therefore is a counter that counts in the Gray-code. This is because the Gray-code is a unit distance code; that is, every next value differs from the previous in only one bit position. A common use of gray codes is in reducing quantization errors in various types of A/D coversion systems. It can reduce ciruit hazards caused by glitches, as well as bring a low power design.

Let now examine how this helps us. Firstly, synchronization means that we will rarely have the sampled value of the counter go metastable, and second, the value that we do sample will at most have one bit error. This means that if the counter's actual value changed from N-1 to N, you will read either N-1 or N, but no other value. This is absolutely correct behavior for reading a counter, since at the time of change you need to make a decision about the value. As long as you decide that the value is the old value or the changed value, you are OK. Any other value is not OK.

4.6.2 Pessimistic Reporting: Handling errors gracefully

According to the above, we now analyze how it can be applied to the read and write pointers of the FIFO. We need to know the status of the FIFO buffer. For this design, the AHB system clock is always stable. Therefore, for the A/D converter, we synchronize the write pointer (Gray-coded) to the read pointer. This means we may have a stale value of the write pointer, since the actual write pointer may have changed to a different value while we were synchronizing it. If this is so, then the read side thinks that less writes have been performed (than actually have), and if conditions match, that the FIFO is empty. In truth, the FIFO may have some data because writes may have taken place that the read side did not "see". However, we merely block additional reads and this is OK. It would be incorrect if we did not block reads when the FIFO was actually empty.

Similarly for the D/A converter, we synchronize the read pointer (Gray-coded) to the write pointer. The write side sees "delayed" reads, and may decide that the FIFO is full when it actually has some space. The effect of this is that writing will be blocked till the reads "become visible" to the write side. In the meanwhile, it will not allow further writes.

This is called pessimistic reporting. In short, reporting to the read side that the FIFO is empty when it is not is fine, and so is reporting to the write side that the FIFO is full when it is not. This acts as if the FIFO had dynamically shrunk a little, and is quite harmless.

In case of the word count, we use the same technique, providing a read-side word count and a write-side word count. The read-side word count is likely to be lesser than the actual word count in the FIFO, and this is quite alright because the only effect it is allowed to have is to block further reads. Similarly the write side word count may be greater than the actual word count, and that too is OK.

This mechanism of pessimistic reporting takes care of gracefully handling errors in the synchronized value. In fact even if the sampled write pointer value were to remain metastable for a while, the effect would be to block reads, causing the FIFO to "hang" for that period for reads, but not causing data errors. The same applies to writes.

4.6.3 Creating the Empty and Full conditions

Remember from the last section that the pointers are not the only things that affect the empty and full flags. The empty condition is when a read caused the pointers to be equal and full is when a write caused the pointers to be equal. In other words, to generate the full and empty correctly, we need to sample the read and write signals themselves with respect to the other clock.

Since it is not possible to design one circuit that will satisfy pulse sampling regardless of frequency, we bypass the problem by encoding the read or write information in the pointers themselves. We keep a pointer width of N+1 for a FIFO that has a depth of 2^N words. The FIFO is deemed full when the most significant bits of the counters differ and the remaining N bits are equal. The FIFO is deemed empty when the pointers are exactly equal.

We can convert from Gray to binary and binary to Gray using the simple equations:

$g_n = b_n$

$g_i = b_i \oplus b_{i+1} \;\forall i \neq n$

And

$b_n = g_n$

$b_i = g_i \oplus b_{i+1} \;\forall i \neq n$

In the equations above, the subscript refers to the bit number in an n+1 bit binary or Gray value.

Also knowing that a counter is nothing more than a set of flip-flops and an incrementer, we do the following – convert the Gray value to binary, increment it, convert it back to Gray and store it. This is the general solution to the thorny problem of generalized n-bit Gray arithmetic. This generalized counter is shown in Figure 4.9.

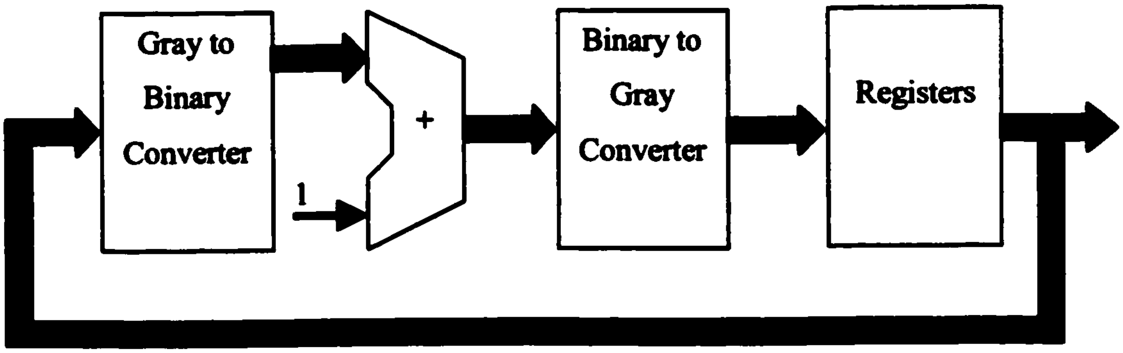


Figure 4.9: Generalized Gray counter architecture

Chapter 5

Design Document

5.1 Overview

The IP core provides an interface between the AHB main SoC bus and APB slaves. It consists of two independent units, one handling transactions between the AHB system bus and APB bridge; another one handling transactions between the APB bridge and APB slaves. The core has been designed to offer as much flexibility as possible to all kinds of applications. An instruction of the IP core is shown in Figure 5.1.

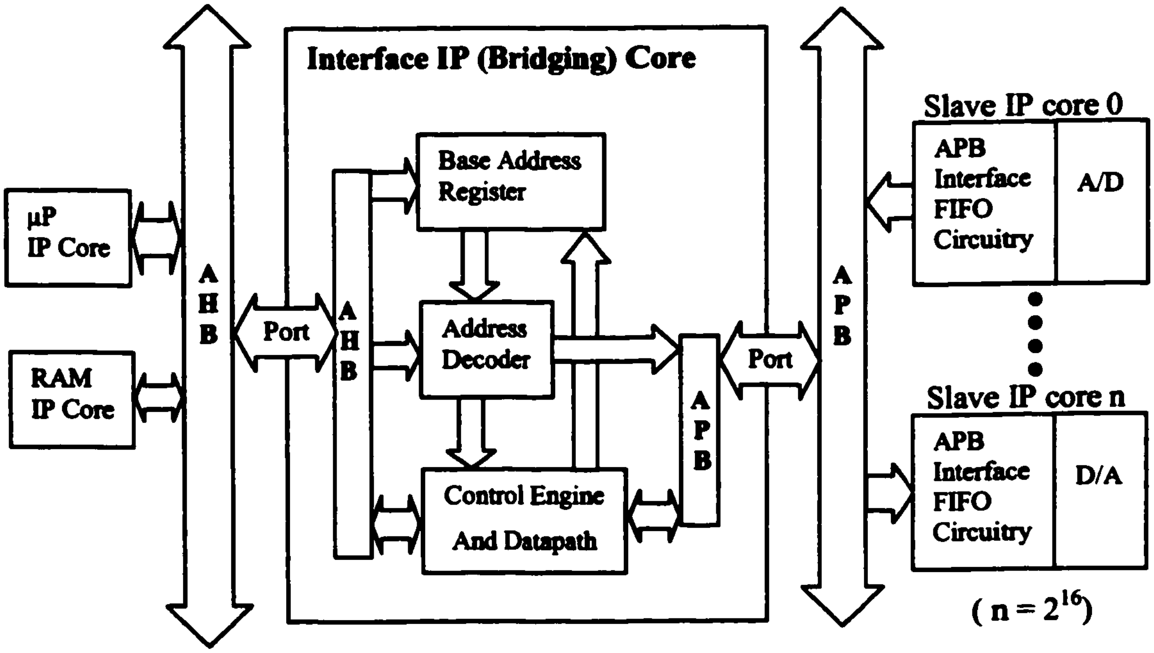


Figure 5.1: The conceptual block diagram of the interface IP core

The interconnection of the designed interface IP core is shown in Figure 5.2. The microprocessor acts like a master on the AHB bus. RAM and APB bridge are two slave IP cores controlled by the microprocessor, which drives out the address and control signals indicating the transfer it wish to perform. A central decoder is required to control the read data and ready signal multiplexors, which selects the appropriate signals from the slave core that is involved in the transfer. The maximum slave amount APB bridge can take is 2^{16}.

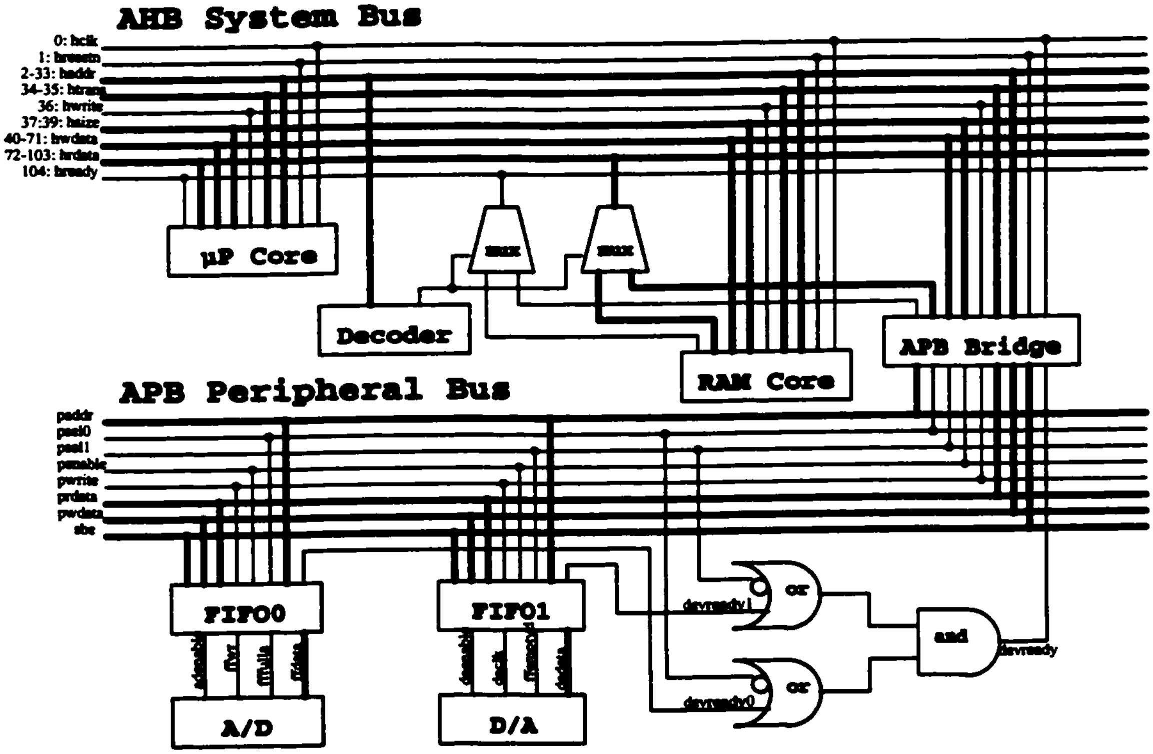


Figure 5.2: The interconnection of the interface IP core

A secondary decoder within the APB bridge unit outputs select signals to each peripheral slave IP core A/D and D/A converters. Through FIFO buffers, data transfer between the

APB bridge and peripheral slaves. The ready signals of both slaves act on the APB bridge together.

5.2 Core file hierarchy

The hierarchy of modules in the interface IP core is shown here with file tree.

top.v

- bridge.v
- adc.v
  - adfifoctl.v
  - fifowr.v
  - fiford.v
- dac.v
  - dafifoctl.v
  - fifowr.v
  - fiford.v

5.3 Description of core modules

5.3.1 The top module

The module *top.v* consists only module *bridge.v* and two APB slaves: *adc.v* and *dac.v*. It doesn't have any logic inside.

5.3.2 The bridge module

The module *bridge.v* is used for transacting data between AHB and APB. It consists 9 parts:

a. AHB-to-APB state machine;

b. HREADY signal generation;

c. AHB read data HRDATA generation;

d. APB write data PWDATA generation;

e. Write byte enable signal generation;

f. APB write signal PWRITE generation;

g. APB selection signal generation;

h. APB strobe signal PENABLE generation;

i. APB address signal PADDR generation.

The bridge state machine has totally 6 statuses. The default status is idle. When AHB selects the APB bridge to access, the state goes to address preparing status. At this state, AHB checks the status of APB bridge. If the corresponding APB bridge is ready, the state goes to write or read cycle directly, otherwise, the state moves to write wait or read wait status till APB bridge is ready.

For write operations, the AHB master will hold the data stable throughout the extended cycles. While for read transfers, the APB bridge does not have to provide valid data until the transfer is about to complete.

5.3.3 The A/D converter module

The A/D converter is an APB slave on the APB bridge. The module *adc.v* is used to write data to APB bridge. When PSEL0 = 1, A/D converter is selected. APB address 00H is connected to FIFO data output port from which the A/D converter writes valid data to the APB bridge; while APB address 01H is for the FIFO status output, which shows the A/D converter enable bit and the number of data inside the FIFO buffer.

5.3.4 The A/D FIFO control module

The module *adfifoctl.v* is used to control the FIFO buffer to read and write. The A/D converter is static when not in use. When it gets a command from the APB bridge, it starts to transfer data to the FIFO buffer, then APB bridge read data from the FIFO buffer. During the processing, in order to descript the status of FIFO, we apply two pointers, read pointer and write pointer. The memory address of the incoming data is in the write pointer. The address of the first data word in the FIFO buffer that is to be read out is in the read pointer. After reset, both pointers indicate the same memory location. After each write operation, the write pointer is set to the next memory location. The reading of a data word sets the read pointer to the next data word that is to be read out. The read pointer constantly follows the write pointer. When the read pointer reaches the write pointer, the FIFO is empty. If the write pointer catches up with the read pointer, the FIFO is full, having 8 words stored. Figure 5.3 illustrates the principle of a circular FIFO with two pointers.

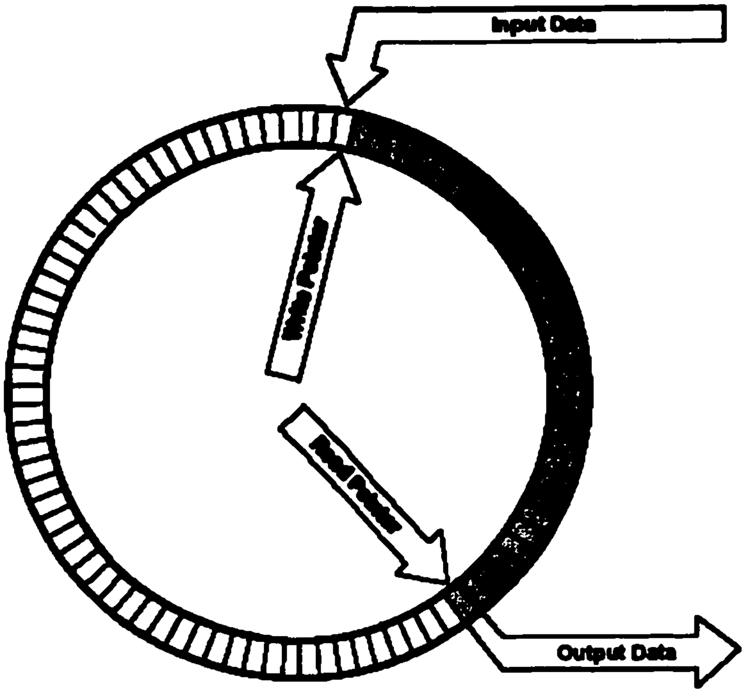


Figure 5.3: Circular FIFO With Two Pointers

The module uses a gray-coder pointer to synchronize the write pointer with the AHB system clock. A length-3 gray-code is a simple cyclic list of distinct binary 3-tuples, called codewords, with the property that any two adjacent codewords differ in exactly one component. The binary to gray-code conversion table is shown in Table 5.1.

Table 5.1: Binary to gray-code conversion table

Nth step after Reset	Binary	Gray-code
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

5.3.5 The FIFO write module

The module *fifowr.v* is used in both A/D and D/A converter. It's used to write data to the FIFO buffer. According to the different write pointer, the data is input to different FIFO address.

5.3.6 The FIFO read module

The module *fiford.v* is used to read data from the FIFO buffer. According to the different read pointer, the data is output from different FIFO address.

5.3.7 The D/A converter module

The D/A converter is another APB slave on the APB bridge. The module *dac.v* is used to read data from APB bridge. When PSEL1 = 1, D/A converter is selected. APB address 00H is connected to FIFO data output port from which the D/A converter reads valid data from the APB bridge; while APB address 01H is for the FIFO status output, which shows the D/A converter enable bit and the number of data inside the FIFO buffer.

5.3.8 The D/A FIFO control module

The module *dafifoctl.v* is used to control the FIFO buffer to read and write. Same as the A/D converter, D/A converter is also static when not in use. When the AHB master wants to transact a write operation to D/A converter, it firstly transfer data to the APB bridge, then the APB bridge input data to the FIFO buffer, so that the D/A converter can read data from the FIFO buffer.

The read pointer and the write pointer are also applied. The gray-coder pointer was used as the read pointer to synchronize AHB system clock.

5.4 Testbench

The design testbench consists of a whole environment for testing the interface including APB bridge and FIFO buffers with bus monitors and test-cases which use those models to stimulate transactions through the interface.

According to the property of the interface, there are several test-cases should be simulated:

1) System reset, all outputs are set to zero;
2) AHB starts to access APB bridge, select the A/D converter to transact a read operation;
    a) AHB enables the A/D converter;
    b) AHB reads data from FIFO;

c) AHB reads the status of the FIFO buffer;

d) AHB doesn't access FIFO, read data become zero;

e) AHB reads indefinite outputs when the A/D converter isn't selected or there is no read transaction.

3) AHB select the D/A converter to transact a write operation.

a) AHB enables the D/A converter;

b) AHB write data to FIFO;

c) AHB reads the status of the FIFO buffer;

d) AHB doesn't access FIFO, read data become zero;

e) AHB reads indefinite outputs when the D/A converter isn't selected or there is no read transaction.

Chapter 6

Synthesis & Layout

6.1 Up-front issues

This chapter shows the synthesis (Synopsys) and physical design (Cadence) processes by using CMOSP35 IC fabrication technology. The standard cell library selected for this design is from the technology foundry (TSMC or Taiwan Semiconductor Manufacturing Company) and the cells are often referred to as " Black Box" or sometimes phantom cells.

In this specific design, there are totally 176 I/O pads. 106 of them are inputs; the other 70 pins are outputs. In addition, 24 power pads (4 pair for core, 8 pair for ring power) will be added to provide power to the core of the chip, the other I/O cells, and dive signals off-chip. This will allow for an average flow of 240 mA of current to the core, which at 3.3 Volts gives nearly 800 mW of power consumption.

According the pin number, the length of die is decided to be 54 times 84μm plus two 365μm corner cell lengths. This makes a total length of 5.266 mm. The width of die is decided to be 4.594 mm based on the same calculation. Lastly, a 50 MHz (20 ns period) clock speed will be applied to the design.

The whole synthesis and physical design processes will base on CMC's Digital IC Design Flow, which is shown in Figure 6.1.

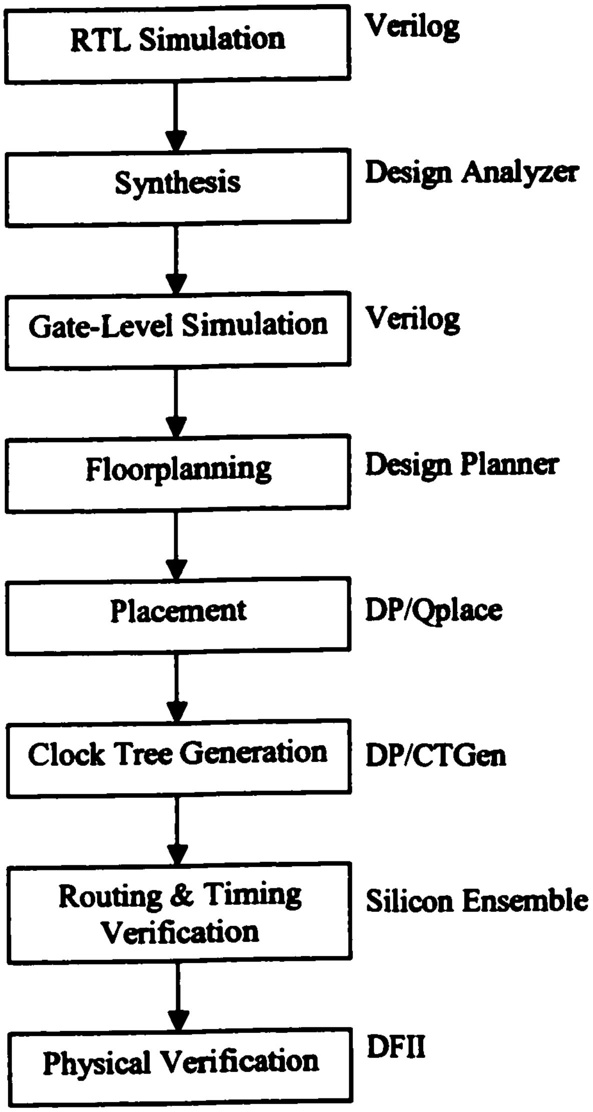


Figure 6.1: Digital IC Design Flow

The design flow shown in Figure 6.1 has been elaborated in the following steps:

Step 1:

In order to verify the Verilog code for interface (bridging) IP core, run behavioral simulation (RTL) with a test bench under a Verilog-XL environment.

Step 2:

The Verilog code was imported into the Synopsys Design Analyzer environment and the behavioral Verilog description (RTL) was converted to a gate-level Verilog description. Run the simulation again to verify the functionality of the gate-level netlist.

Step 3:

Imported the gate-level verilog netlist into the Cadence Design Planner environment. The logic gates were described in terms of 0.35-micron CMOS standard cells, floorplanning was carried out (I/O pads, standard cells).

Step 4:

After floorplanning, placement was carried out. Standard cells were placed but not interconnected; the clock tree structure was added. Also generated an updated Verilog netlist to be used for final verification.

Step 5:

The resulting placed design (.lef and .def file) was then imported into the Cadence Silicon Ensemble environment; routing (interconnection) and timing verification was performed. The produced .def described the routing of the IP core in the 0.35-micron CMOS technology.

Step 6:

The Verilog file produced from Design Planner and the DEF file created in Silicon Ensemble were imported into the Cadence DFII environment, Design Rule Check (DRC) and Layout Vs. Schematic (LVS) check were performed. The DFII tool converted the final design into a stream format. After compressing, it was ready to transmit to CMC for fabrication.

6.2 RTL Simulation

In order to verify the functionality of the RTL code, the first step is to simulate this RTL code with testbench. Using the **Signalscan** tool, we can view all input and output signals. The waveforms under different test cases are shown from Figure 6.2 to Figure 6.7.

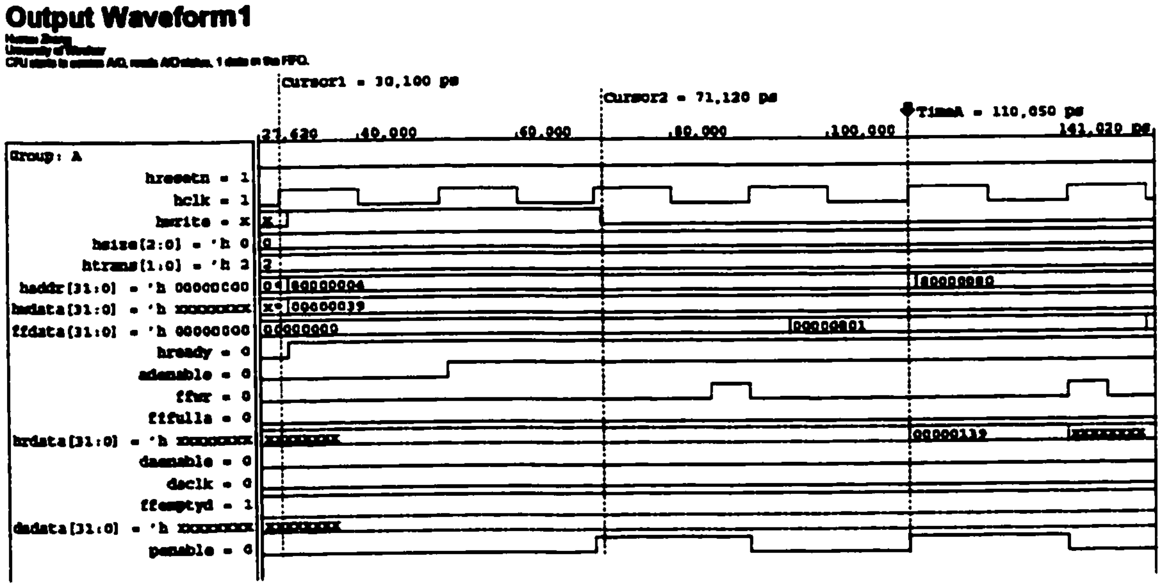


Figure 6.2: CPU reads the A/D status

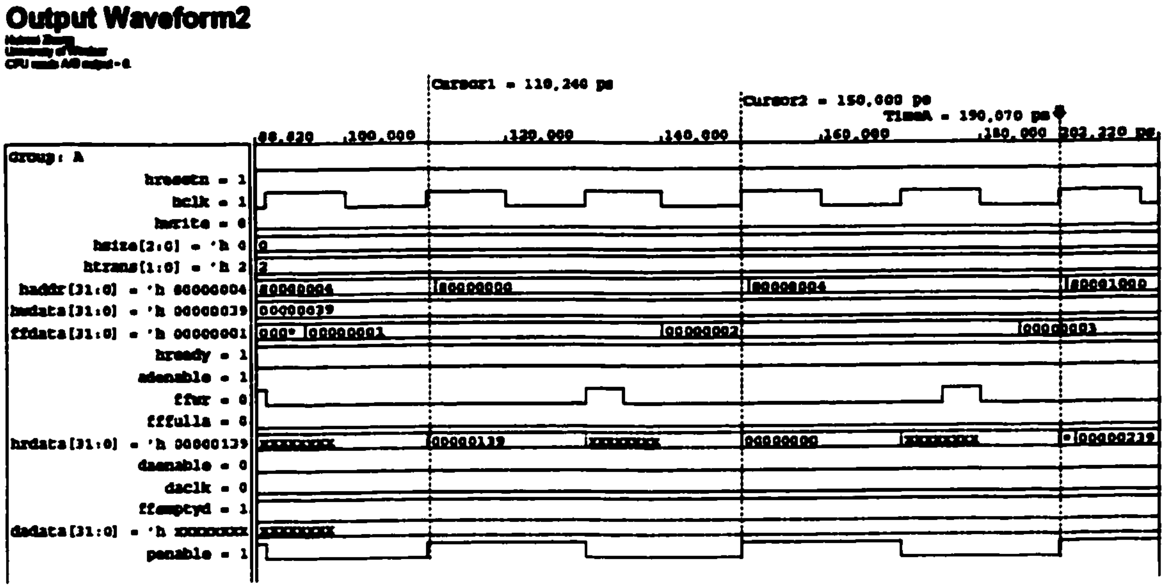


Figure 6.3: CPU reads the A/D output

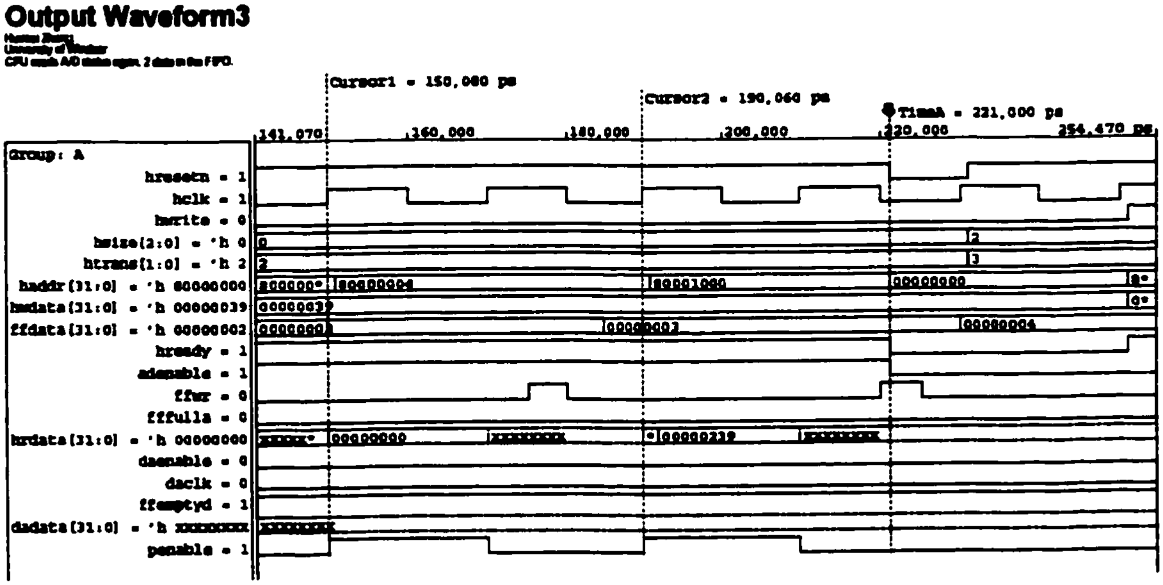


Figure 6.4: CPU reads the A/D status once more

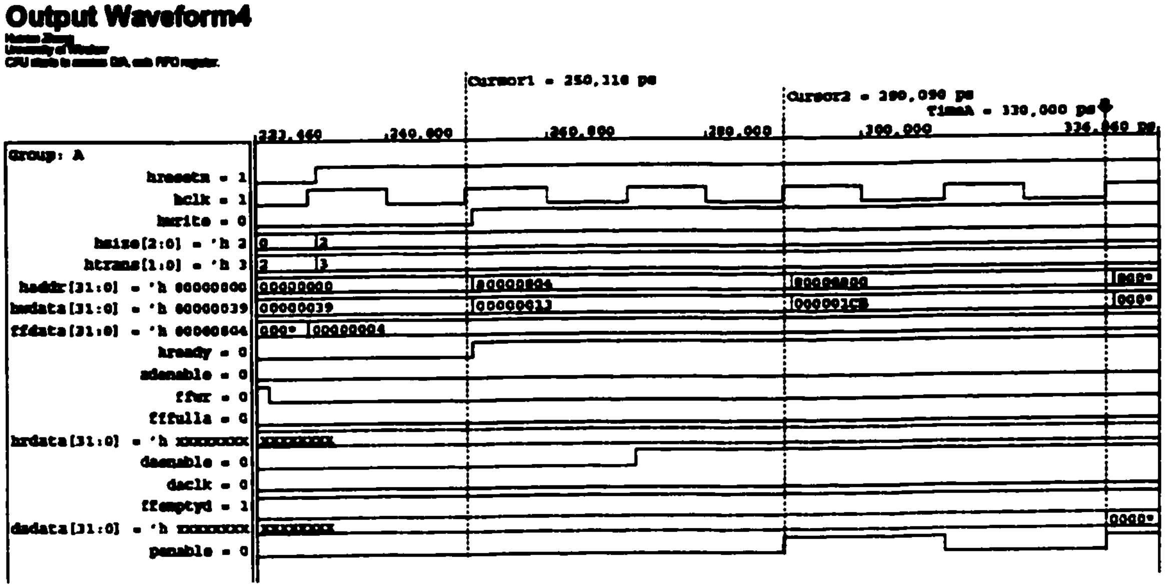


Figure 6.5: CPU starts to access the D/A

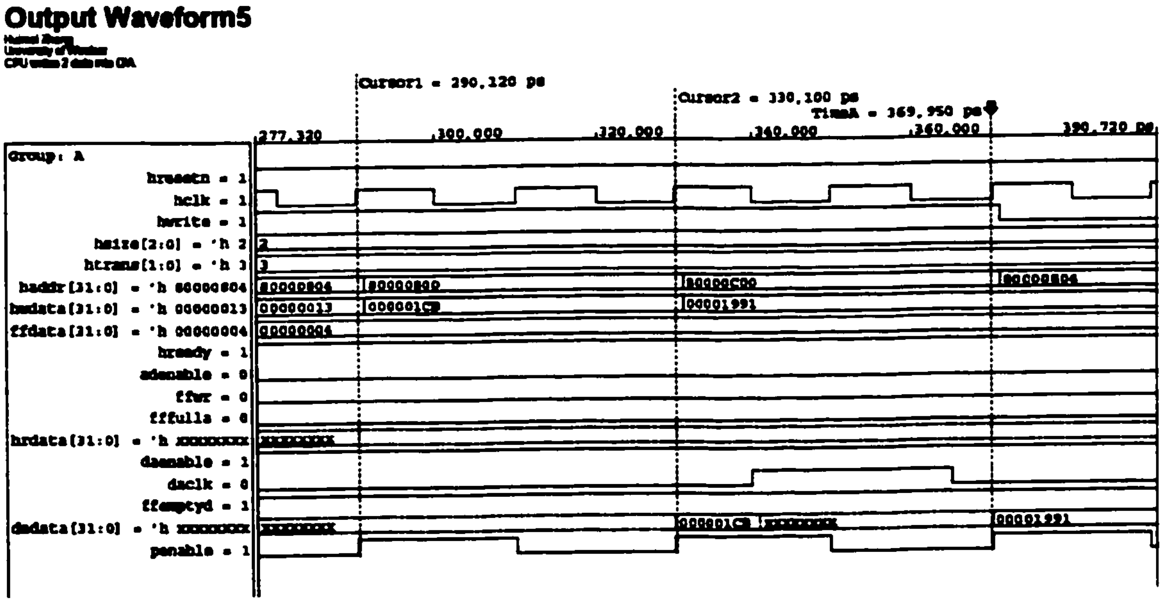


Figure 6.6: CPU writes data into the D/A

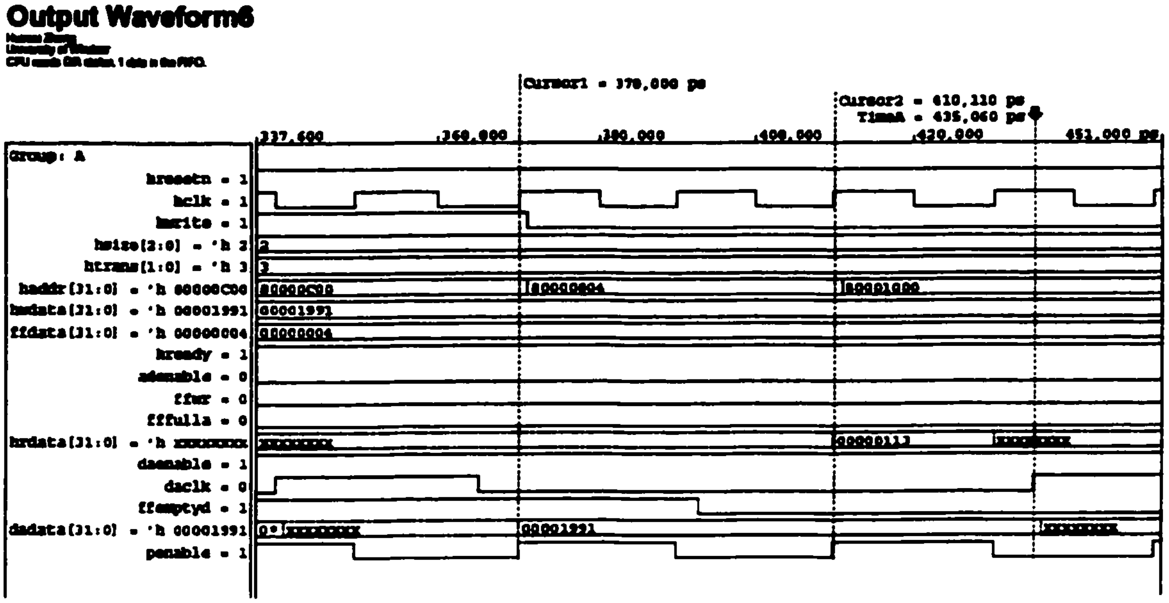


Figure 6.7: CPU reads the D/A status

6.3 Synthesis

At the second step we import the design into a Synopsys database, use **Design Analyzer** tool to synthesize the design into gates, making it meet all pre-set constraints. Soft-core synthesis usually consists of instantiating the netlist directly into the customer's design. Elements within soft-core netlists used in the Synopsys synthesis environment carry the company's "don't_touch" annotation to prevent changes to the soft-core design during synthesis optimization. The functionality of the gate-level netlist should be verified. The gate-level netlist contains totally 1612 references. The gate-level area and reference report is shown in appendix B.

This step performs the symbol view of each module. Figure 6.8, 6.9, 6.10 and 6.11 show the top interface module, APB bridge module, A/D FIFO module and D/A FIFO module blocks respectively. We can see all input and output pins from here.

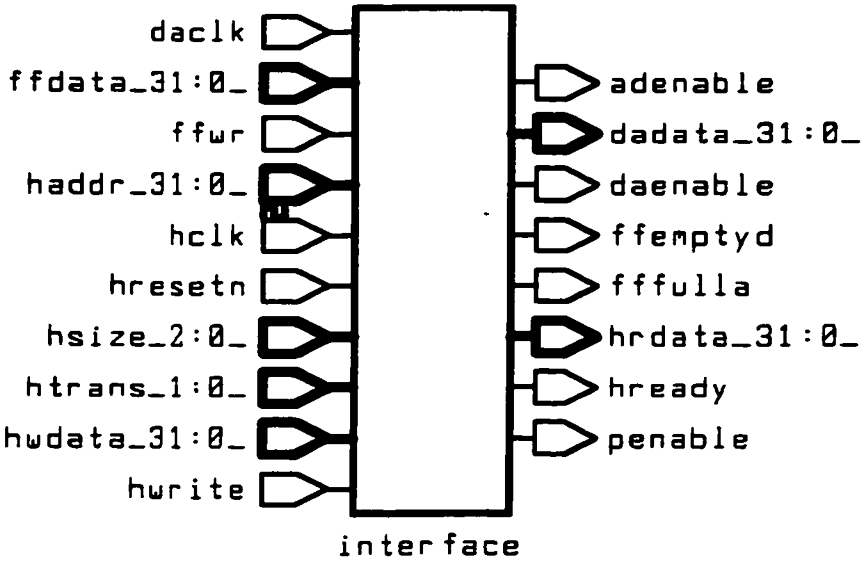


Figure 6.8: The top interface symbol view

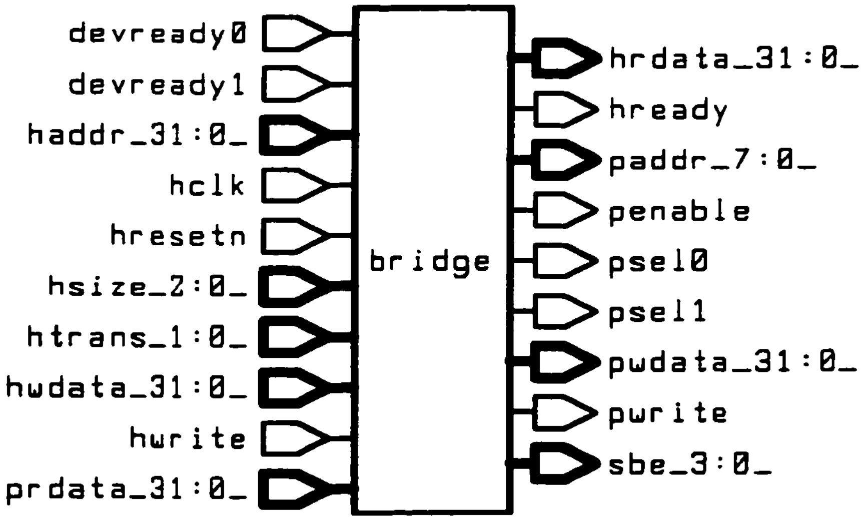


Figure 6.9: The APB bridge symbol view

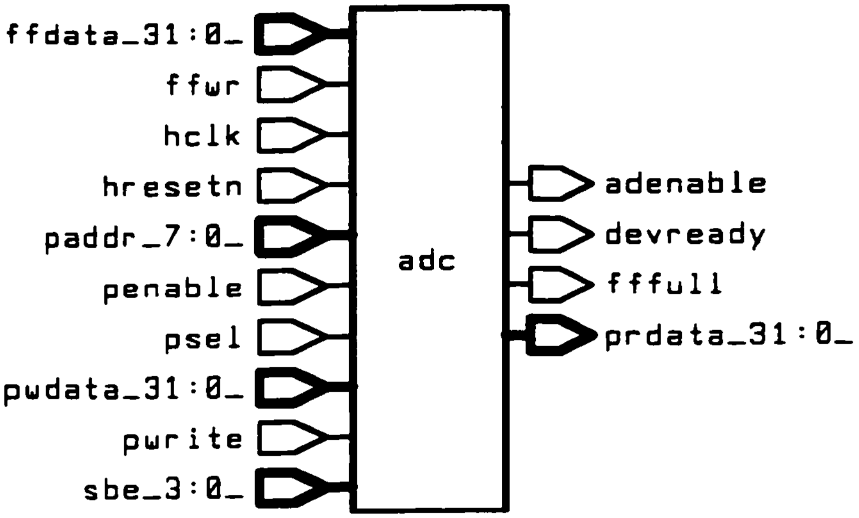


Figure 6.10: The A/D FIFO symbol view

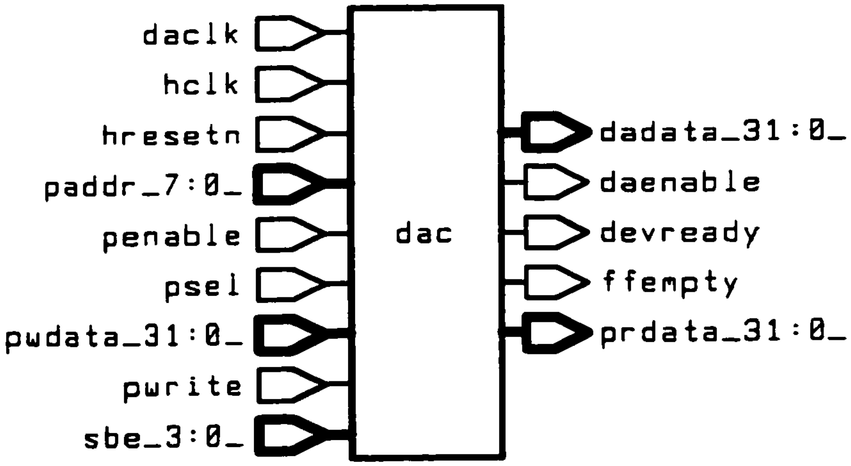


Figure 6.11: The D/A FIFO symbol view

6.4 Floorplanning

This section uses the Cadence tool **Physical Design Planner** for physical placement of a design. After inserting the netlist, we place and route each firm core and time it to meet the same timing assertions used to generate the static timing model. An updated Verilog netlist is also generated in order to have the final verification. We remove the netlist information and replace it with the firm-core library element before returning the design to the customer for postlayout functional verification.

Hard cores complicate floorplanning because of their size, wiring blockage, noise isolation requirements, peripheral test circuitry, and proximity to chip I/O pads. The core can be places anywhere on the chip, but placement in a corner of the die allows for maximum wiring of the remaining chip logic. Corner placement also minimizes the chance of splitting a functional block and placing the parts on opposite sides of the core. Figure 6.12 and 6.13 show the final floorplanned design and the zoom in view of the core area respectively.

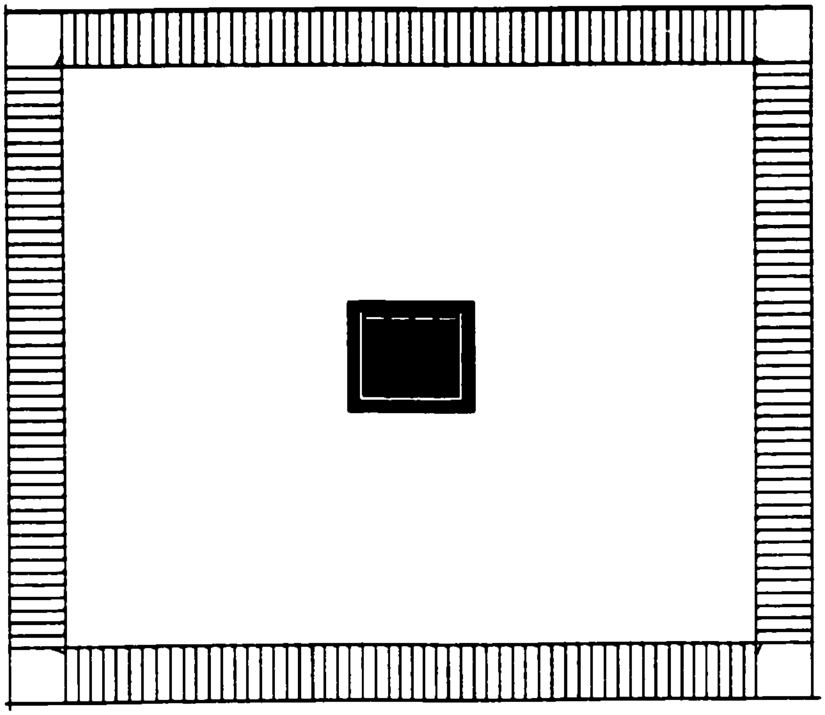


Figure 6.12: The final floorplanned design

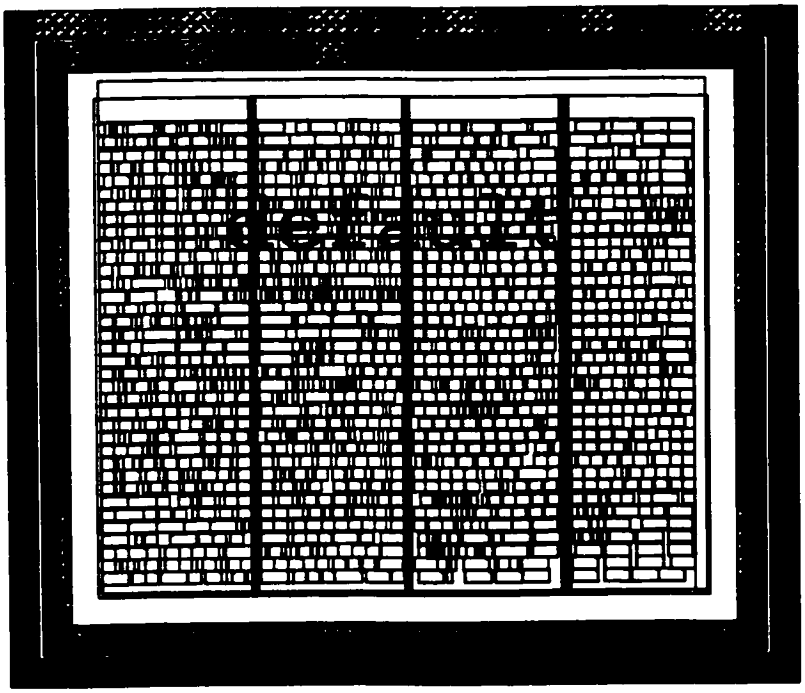
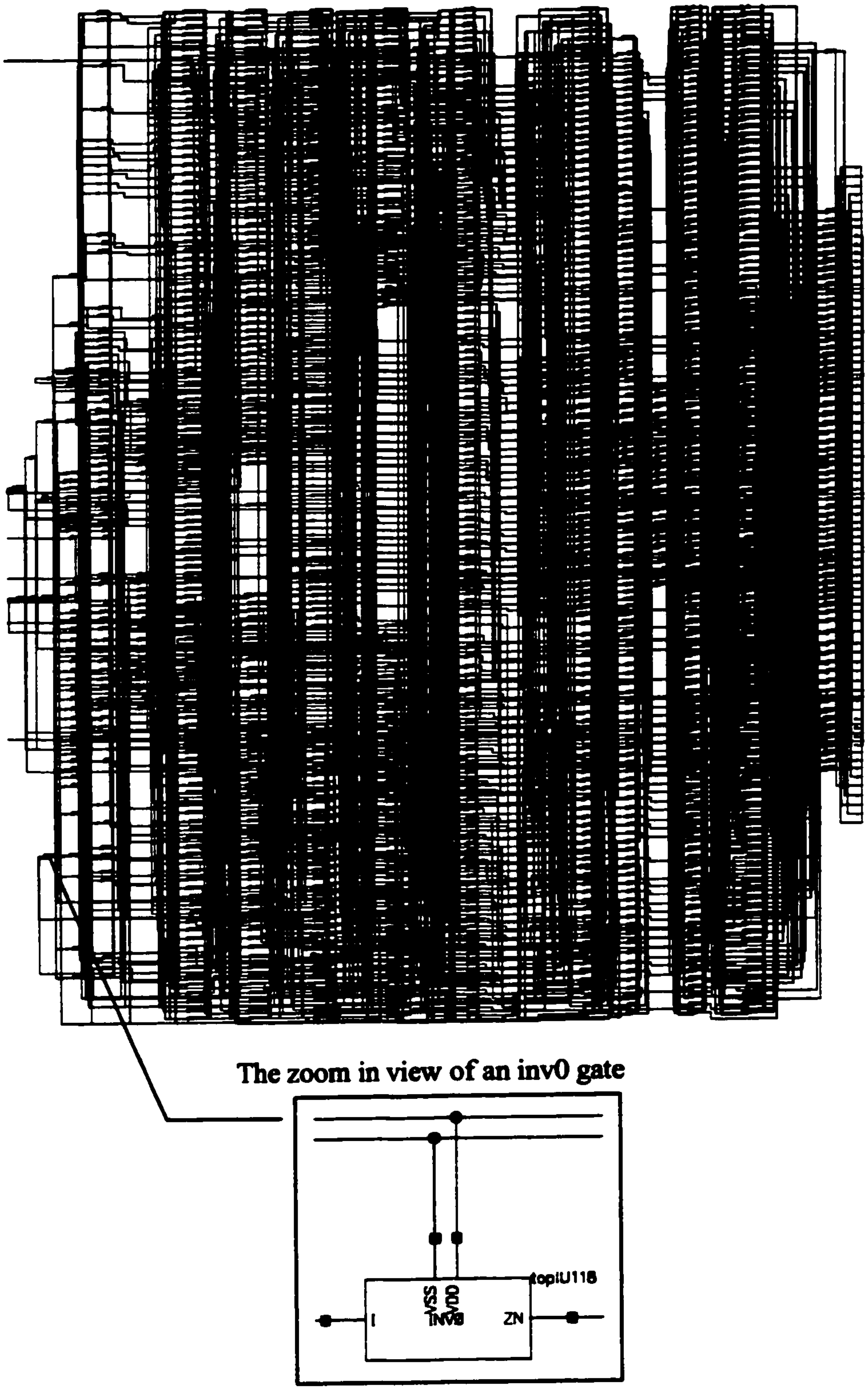


Figure 6.13: The zoom in core area view

6.5 Physical verification

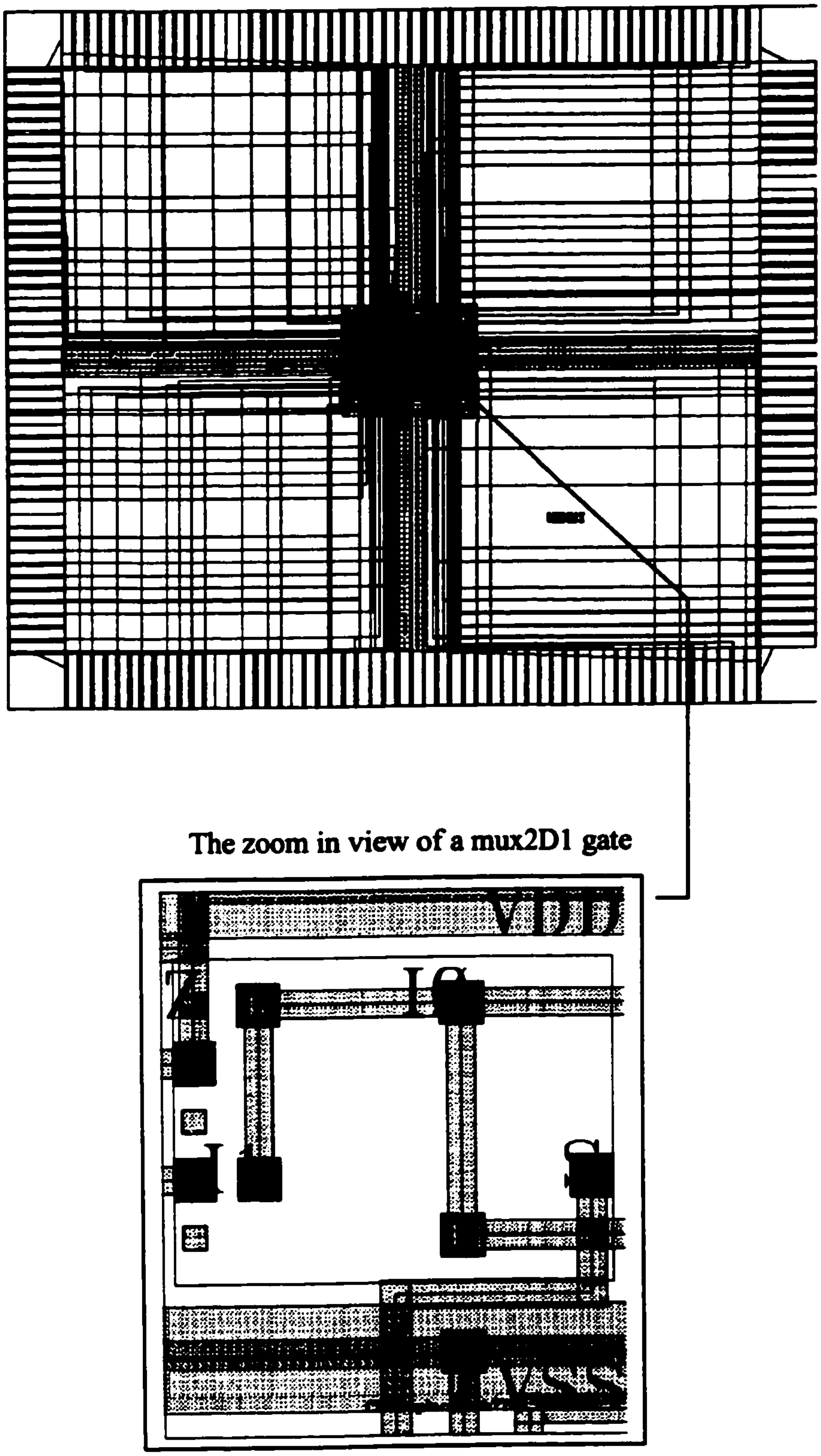
The resulting placed design (.lef and .def file) is then imported into the Cadence Silicon Ensemble environment; routing (interconnection) and timing verification is performed. The produced .def describe the routing of the IP core in the 0.35-micron CMOS technology.

The Verilog file produced from Design Planner and the .def file created in Silicon Ensemble are imported into the Cadence DFII environment. At this step, we verify the physical (Placed & Routed) version of the design; Design Rule Check (DRC) and Layout Vs. Schematic (LVS) check are performed. The DFII tool converted the final design into a GDSII stream format. After compressing, it was ready to transmit to CMC for fabrication. This is the last step; the final schematic view is shown in Figure 6.14, if zoom in the schematics, some symbols of gates could be viewed, such as an inverter, shown in Figure 6.14 too. Figure 6.15 is a final layout view showing the abstract representation of the proprietary standard cells supplied by the foundry. The layout dimensions are: 5266 × 4594 um for an area of 24192004 square um. Since it's a black-box library, only abstract symbols and some interconnection could be seen from the layout. A zoom in view of a multiplexor gate is also shown in Figure 6.15.



The zoom in view of an inv0 gate

Figure 6.14: The schematic view of the design



The zoom in view of a mux2D1 gate

Figure 6.15: The final layout view

Chapter 7

Conclusions & Future work

7.1 Conclusions and contributions

In this thesis, we describe a digital interface IP core, for use in a SoC environment. This IP core provides a bridging capability between the main SoC bus (AHB) and the peripheral bus (APB), as well as peripheral FIFO circuitry that must be inclubed in peripheral IP cores. The IP core allows the main SoC bus to read and write to peripheral devices, and is developed using Verilog for ease of mapping to technology and design reuse. The Verilog language is used to implement the state transition specifications required for the bridging architecture.

The interface IP core is compliant with AMBA specifications that define the interface protocol for the connection of multiple IP cores in a System-on-Chip environment.

The thesis develops a Verilog HDL description of the interface IP core (bridging) and the required FIFO circuitry that must be implemented in peripheral IP cores so that they may interface with the APB bus. The Verilog description has been mapped to a 0.35-micron CMOS technology process. The resulting hardware layout is comprised of 1612 elements consisting of logic gates, inverters, multiplexors and shift registers. The architecture supports a 50MHz system clock speed. The circuitry has been extendedly simulated and the layout has undergone design rule checking satisfactorily. The final GDSII stream file is ready to transmit to CMC for fabrication.

7.2 Suggestions for future work

The SoC design methodology is still evolving. An important area of future research is the development of standards to foster IP core development, exchange, and interoperability. As part of future work, the interface IP core presented in this thesis could be easily redesigned to produce an interconnection capability between bus architectures with different protocols.

References

1. Abdallah Tabbara and Bassam Tabbara, *Inter-Module Interconnect Strategy for System on Chip Applications*, Technical Memoranda M99/45, Electronics Research Laboratory, University of California

2. ALTERA, *PCI MegaCore Function User Guide*, August 2001, http://www.altera.com

3. Anders Nordqvist, Patric Eriksson, *Design of Distributed Adaptive User Interfaces Using Digital Plant Technology*, The 24th annual conference of the IEEE Industrial Electronics Society, September 1998

4. Ann Marie Rincon, Cory Cherichetti, James A. Monzel, David R. Stauffer and Michael T. Trick, *Core Design and System-on-a-Chip Integration*, IEEE Design & Test of Computers, 1997, pp. 26 –36

5. ARM, *AMBA^{TM} Specification*, Rev 2.0, 1999, http://www.arm.com

6. Becker, J., Pionteck, T. and Glesner, M., *Adaptive systems-on-chip: architectures, technologies and applications*, 14th Symposium on Integrated Circuits and Systems Design, 2001, pp. 2 –7

7. Canadian Microelectronics Corporation, *Tutorial on CMC's Digital IC Design Flow*, Document ICI-096, V1.2, December 2000

8. Chang, H., Cooke, L. etc., *Surviving the SOC Revolution, A Guide to Platform-Based Design*, Kluwer Academic Publishers, 1999

9. Chauchin Su and Wenliang Tseng, *Configuration free SoC interconnect BIST methodology*, Proceedings of 2001 International Test Conference, 2001 pp. 1033 –1038

10. Clifford E. Cummings and Peter Alfke, *Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons*, Sunburst Design, Inc., Xilinx, Inc., 2002, http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO2_rev1_1.pdf

11. Dexter, L., *Microcmputer Bus Structures and Bus Interface Design*, Marcel Dekker Inc., 1986

12. Erno Salminen, Timo Hämäläinen, Tero Kangas, Kimmo Kuusilinna, and Jukka Saarinen, *Interfacing Multiple Processors in a System-on-Chip Video Encoder*, Proceedings of 2001 IEEE International Symposium in Circuits and Systems (ISCAS'2001), Vol. 4, 2001, pp. 478 –481

13. Fox Trevor Robert, *First-in-first-out buffer*, INT Computers Ltd., EP 0484652, May 1992, http://swpat.ffii.org/patents/txt/ep/0484/652/

14. Gale, D., *System-on-chip research infrastructure for Canadian Universities*, Proceedings of 2001 International Conference on Microelectronic Systems Education, 2001, pp. 46 –49

15. Hoon Choi, Ju Hwan Yi, Jong –Yeol Lee, In –Cheol Park, And Chong –Min Kyung, *Exploiting Intellectual Properties in ASIP Designs for Embedded DSP Software*, Design Automation Conference (DAC), 1999, pp. 939 –944

16. IBM, *The CoreConnect^{TM} Bus Architecture*, 1999, http://www.chips.ibm.com/products/coreconnect

17. Janet Wedgwood and Greg Buchanan, *A Model-Year Architecture Approach to Hardware Reuse in Digital Signal Processor System Design*, VHDL International Users Forum, October 1997, pp. 231 –240

18. Klapproth, P., *General architectural concepts for IP core re-use*, Design Automation Conference, 2002. Proceedings of ASP-DAC 2002. 7th Asia and South Pacific and the 15th International Conference on VLSI Design, 2002, pp. 325

19. Kreutz, M.E., Carro, L., Zeferino, C.A. and Susin, A.A., *Communication architectures for system-on-chip*, 14th Symposium on Integrated Circuits and Systems Design, 2001, pp. 14 -19

20. Lahiri, K., Raghunathan, A. and Dey, S., *System-level performance analysis for designing on-chip communication architectures*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 20 No. 6, June 2001, pp. 768 -783

21. Lahiri, K., Raghunathan, A. and Lakshminarayana, G., *LOTTERYBUS: a new high-performance communication architecture for system-on-chip designs*, Design Automation Conference, 2001, pp. 15 -20

22. Mark Genoe, Philippe Delforge and Eric Schutz, *Experience of IP-Reuse in System-on-Chip Design for ADSL*, 1999 IEEE international Solid-State Circuits Conference, February 1999, pp. 360 -362

23. Martin, G. and Chang, H., *System-on-chip design*, Proceedings of 4th International Conference on ASIC, 2001, pp. 12 –17

24. Martin, G. and Lennard, C., *Improving embedded software design and integration in SOCs*, Proceedings of the IEEE 2000 Custom Integrated Circuits Conference (CICC), 2000, pp. 101 –108

25. Miha Dolenc and Tadej Markovic, *PCI IP Core Specification*, Rev. 0.6, January 2002, http://www.opencores.org

26. Miha Dolenc and Tadej Markovic, *PCI IP Core Design Document*, Rev. 0.6, January 2002, http://www.opencores.org

27. Mustafa, A., *Microcomputer Interfacing and Applications (second Edition)*, Hartnolls, Bodmin, 1994

28. Navabi, Z., *Verilog Digital System Design*, McGraw-Hill, 1999

29. Roy Chapman and Tariq S. Durrani, *IP Protection of DSP Algorithms for System on Chip Implementation*, IEEE Transactions on Signal Processing, Vol. 48, No. 3, March 2000, pp. 854 –861

30. Roza, E., *Systems-on-chip: what are the limits?* Electronics & Communication Engineering Journal, Vol. 13, No. 6, December 2001, pp. 249 –255

31. Rudolf Usselmann, *OpenCores SoC Bus Review*, Rev. 1.0, 2001 http://www.opencores.org/wishbone/soc_bus_comparison.pdf

32. Saastamoinen, I., Siguenza-Tortosa, D.and Nurmi, J., *Interconnect IP node for future system-on-chip designs*, The First IEEE International Workshop on Electronic Design, Test and Applications, 2002, pp. 116 –120

33. Takanori Okuma, Koji Hashimoto and Kazuaki Murakami, *Development of PPRAM-Link Interface (PLIF) IP Core for High-Speed Inter-SoC Communication*, Proceedings of ASP-DAC 2001, January 2001, pp. 37 –38

34. Texas Instruments Inc., *FIFO Architecture, Functions, and Applications*, November 1999, http://www-s.ti.com/sc/psheets/scaa042a/scaa042a.pdf

35. Vijay A. Nebhrajani, *Asynchronous FIFO Architectures*, http://www.geocities.com/deepakgeorge2000/vlsi_book

36. Vijay K. Madisetti and Lan Shen, *Interface Design for Core-Based Systems*, IEEE design & Test of Computers, 1997, pp. 42 –51

37. Virtual IP Group, *V6001 AHB-APB Bus Bridge Data Sheets*, Rev 2.0, http://www.virtualipgroup.com

38. Xilinx, Inc., *Application Note XAPP175*, V1.0, November 1999, http://www.xilinx.com/xapp/xapp175.pdf

39. Xilinx, Inc., *Application Note XAPP261*, V1.0, January 2001, http://www.xilinx.com/xapp/xapp261.pdf

40. Zhao Junchao, Chen Weiliang and Wei Shaojun, *Parameterized IP core design*, Proceedings of 4th International Conference on ASIC, 2001, pp. 744 –747

Appendix A

Digital Interface IP Core

Software Code

A.1 The main verilog code

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////
//// File name: top.v
////
//// This file is the top file of the "A SoC Digital Interface IP Core" project
////
//// Author(s):     - Huimei Zheng, zhenghm@hotmail.com
////
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////
//// Copyright © 2002 Huimei Zheng, zhenghm@hotmail.com
////
//// This source file may be used and distributed without restriction provided that
//// this copyright statement is not removed from the file and that any derivative
//// work contains the original copyright notice and the associated disclaimer.
////
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

`timescale 1ns/100ps

module top(// input from AHB
            hclk, hresetn, haddr, htrans, hwrite, hsize, hwdata,
            // input to fifo
            ffwr, ffdata, daclk,
           // output to AHB
            hrdata, hready, penable,
          // output from fifo
           adenable, daenable, dadata, fffulla, ffemptyd
           );

 input          hclk;   //ahb clock
 input          hresetn; //active low speed
 input [31:0]   haddr;  //address bus
 input [1:0]    htrans; // Transfer Type Encoding - 00 = IDLE, 01 = BUSY,
 input         hwrite; //read/write signal 1=write 0 = read
 input [2:0]    hsize;  //000=byte
 input [31:0]   hwdata; //32 bit write data bus
 input         ffwr;  //fifo write clock
input [31:0]   ffdata; // input read data
input          daclk; // fifo read clock

output [31:0] hrdata;
output         hready;
output         penable;
output         fffulla;
output         ffemptyd;
output         adenable;
output         daenable;
output [31:0]  dadata; //output write data

wire           devready0, devready1, psel0, psel1, pwrite;
wire [31:0]  pwdata, rda, rad;
wire [3:0]   sbe;
wire [7:0]   paddr;

bridge ibridge(.hclk(hclk), .hresetn(hresetn), .haddr(haddr), .htrans(htrans),
            .hwrite(hwrite), .hsize(hsize), .hwdata(hwdata),
            .devready0(devready0), .devready1(devready1),
            .hrdata(hrdata), .hready(hready),
            .psel0(psel0), .psel1(psel1), .penable(penable),
            .pwdata(pwdata), .sbe(sbe), .pwrite(pwrite), .paddr(paddr),
            .prdata(prdata)
            );

adc iadc(.hclk(hclk), .hresetn(hresetn), .pwdata(pwdata),
       .sbe(sbe), .pwrite(pwrite), .paddr(paddr),
       .psel(psel0), .penable(penable),
       .ffwr(ffwr), .ffdata(ffdata), .fffull(fffulla),
       .prdata(prdata), .devready(devready0),
       .adenable(adenable)
       );

dac idac(.hclk(hclk), .hresetn(hresetn), .pwdata(pwdata),
       .sbe(sbe), .pwrite(pwrite), .paddr(paddr),
       .psel(psel1), .penable(penable),
       .daclk(daclk), .dadata(dadata), .ffempty(ffemptyd),
       .prdata(prdata), .devready(devready1), .daenable(daenable)
       );

endmodule
```

A.2 APB bridge

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////
////
//// File name: bridge.v
////
//// This file is part of the "A SoC Digital Interface IP Core" project
////
//// Author(s):    - Huimei Zheng, zhenghm@hotmail.com
////
//////////////////////////////////////////////////////////////////////////////////////////////////////////
////
//// Copyright © 2002 Huimei Zheng, zhenghm@hotmail.com
////
//// This source file may be used and distributed without restriction provided that
//// this copyright statement is not removed from the file and that any derivative
//// work contains the original copyright notice and the associated disclaimer.
////
//////////////////////////////////////////////////////////////////////////////////////////////////////////

`timescale 1ns/100ps

module bridge(//input from AHB
        hclk, hresetn, haddr, htrans, hwrite, hsize,
        hwdata, devready0, devready1,
        //output to AHB
        hrdata, hready,
        //register read and write, apb
        psel0, psel1, penable, pwdata, sbe, pwrite, paddr, prdata
        );

 input          hclk;   //ahb clock
 input          hresetn; //active low speed
 input [31:0]   haddr;  //address bus
 input [1:0]    htrans; // Transfer Type Encoding - 00 = IDLE, 01 = BUSY,
 input          hwrite; //read/write signal 1=write 0 = read
 input [2:0]    hsize;  //000=byte
 input [31:0]   hwdata; //32 bit write data bus
 input [31:0]   prdata; //register read data bus
 input          devready0; //come from individual device
 input          devready1; //come from individual device

 output [31:0]  hrdata;
 output         hready;
 output [31:0]  pwdata; //register data
 output [3:0]   sbe;   //register write-byte enable
 output         pwrite;   //register write signal
 output [7:0]   paddr; //APB address
 output         psel0;
 output         psel1;
 output         penable;

 parameter   delay = 1;
 reg      psel0, psel1;
 reg      intregwract;
```

```
wire      devready = (devready0 | ~psel0) & (devready1 | ~psel1);
wire      intregrdact = ~intregwract;
wire      bcycstrt; // Bus Cycle Start

assign bcycstrt = ((htrans[1] && ~htrans[0]) || (htrans[1] && htrans[0]));
wire cifaccess = (haddr[31] && ~haddr[30] && ~haddr[29] && ~haddr[28]);

parameter      stidle = 6'b000001,
               staddl = 6'b000010,  //address prepare
               stdat1 = 6'b000100,  //read cycle wait
               stdat2 = 6'b001000,  //write cycle wait
               stdat3 = 6'b010000,  //read point
               stdat4 = 6'b100000;  //write point

reg [5:0]     cifstate;
reg [5:0]     nextstate;

always @ (posedge hclk or negedge hresetn) begin
   if(~hresetn) begin
      cifstate <= stidle;
    end // if (~hresetn)
   else begin
      cifstate <= nextstate;
    end // else: !if(~hresetn)
 end // always @ (posedge hclk or negedge hresetn)

always @ (cifstate or bcycstrt or cifaccess or intregwract or intregrdact or devready)
 begin
   case (cifstate)      // synopsys parallel_case
                        // synopsys full_case
     stidle: begin
         if(bcycstrt && cifaccess)
           nextstate <= staddl;
         else
           nextstate <= stidle;
      end // case: stidle
     staddl: begin
         if(intregrdact && devready)  //read and device is ready
           nextstate <= stdat3;
         else if(intregwract && devready) //write cycle and device is ready
           nextstate <= stdat4;
        else if(intregwract) //write cycle
           nextstate <= stdat2;
         else nextstate <= stdat1;  //for read cycle only
      end // case: staddl
     stdat3: begin
        if(bcycstrt && cifaccess)
            nextstate <= staddl;
         else
          nextstate <= stidle;
      end // case: stdat3
     stdat2: begin  //write cycle only
         if(~devready) //keep on waiting
            nextstate <= stdat2;
         else
```

```
          nextstate <= stdat4;
      end
     stdat1: begin
         if(~devready) //keep on waiting
           nextstate <= stdat1;
         else
           nextstate <= stdat3;
      end
     stdat4: begin
        if(bcycstrt && cifaccess)
           nextstate <= stadd1;
         else
          nextstate <= stidle;
      end
     default: begin
         nextstate <= stidle;
      end // case: default
   endcase // case(cifstate)
 end // always @ (cifstate or bcycstrt or cifaccess or intregwract or intregrdact or devready)

always @ (posedge hclk or negedge hresetn) begin
   if(~hresetn) begin
      intregwract <= 1'b0;
    end
   else if(nextstate[1]) begin
      intregwract <= hwrite;
    end // if (nextstate[1])
 end

/*
 * Hready generation
 */

reg hready;
always @ (posedge hclk or negedge hresetn) begin
   if(~hresetn) begin
      hready <= 1'b0;
    end
   else begin
     hready <= #delay devready;
   end
 end

/*
 * APB write data generation
 */

reg [31:0]  wregdata, pwdata;
always @ (nextstate or hwdata or wregdata) begin
   if(nextstate[1])
    wregdata <= hwdata;
   else if(nextstate[3] || nextstate[5])
    pwdata <= wregdata;
 end // always @ (nextstate or hwdata or wregdata)

/*
```

```
 * selection signals generation
 */

wire sel0p = ({haddr[27:11]} == 17'h00000); //there can be many selection signals
wire sel1p = ({haddr[27:11]} == 17'h00001);

always @ (posedge hclk or negedge hresetn) begin
   if(~hresetn) begin
      psel0 <= 1'b0;
      psel1 <= 1'b0;
    end
   else if(nextstate[1]) begin
      psel0 <= #delay sel0p;
      psel1 <= #delay sel1p;
    end
   else if(nextstate[0]) begin
      psel0 <= 1'b0;
      psel1 <= 1'b0;
    end
 end

/*
 * sbe[3:0] generation
 */

wire xferbyted = (~hsize[2] && ~hsize[1] && ~hsize[0]);
wire xferhwrdd = (~hsize[2] && ~hsize[1] && hsize[0]);
wire xferwordd = (~hsize[2] && hsize[1] && ~hsize[0]);
reg  xferbyte, xferhwrd, xferword;

always @ (posedge hclk) begin
   if(nextstate[1]) begin
      xferbyte <= xferbyted;
      xferhwrd <= xferhwrdd;
      xferword <= xferwordd;
    end // if (nextstate[1])
 end // always @ (posedge hclk)

wire bythwdxfer = (xferhwrd || xferbyte);
reg [1:0] low2bits;

always @ (posedge hclk) begin
   if(nextstate[1]) begin
      low2bits <= haddr[1:0];
    end // if (nextstate[1])
 end // always @ (posedge hclk)

wire byte0act = (~low2bits[1] && ~low2bits[0]);
wire byte1act = (~low2bits[1] && low2bits[0]);
wire byte2act = (low2bits[1] && ~low2bits[0]);
wire byte3act = (low2bits[1] && low2bits[0]);
wire lhalfact = ~low2bits[1];
wire uhalfact = low2bits[1];

assign sbe[0] = (xferbyte && byte0act) || (xferhwrd && lhalfact) || (xferword);
assign sbe[1] = (xferbyte && byte1act) || (xferhwrd && lhalfact) || (xferword);
assign sbe[2] = (xferbyte && byte2act) || (xferhwrd && uhalfact) || (xferword);
assign sbe[3] = (xferbyte && byte3act) || (xferhwrd && uhalfact) || (xferword);

/*
 * pwrite signal generations
 */

reg pwrite;
always @ (posedge hclk or negedge hresetn)  begin
    if(~hresetn)  begin
        pwrite <= 1'b0;
     end // if (~hresetn)
    else if(nextstate[1])
          pwrite <= hwrite;
 end // always @ (posedge hclk or negedge hresetn)

/*
 * enable signal generation
 */

assign penable = cifstate[4] || cifstate[5];

/*
 * paddress generation
 */

reg [7:0]   prdd, paddr;
always @ (nextstate or haddr)  begin
    if(nextstate[1])
      prdd <= haddr[9:2];
 end // always @ (nextstate)

always @ (cifstate or prdd)  begin
    if(cifstate[1])  begin
        paddr <= prdd;
     end // if (cifstate[1])
 end // always @ (cifstate or prdd)

/*
 * hrdata, read out data from devices
 */

reg [31:0]  hrdata;
always @ (nextstate or prdata )
 begin
    if(nextstate[1])
     hrdata <= prdata;
    else hrdata <= 32'bz;
 end // always @ (nextstate or prdata)

endmodule
```

A.3 A/D FIFO code

```
////////////////////////////////////////////////////////////////////////////////////////////////
////
//// File name: adc.v
////
//// This file is part of the "A SoC Digital Interface IP Core" project
////
//// Author(s):     - Huimei Zheng, zhenghm@hotmail.com
////
////////////////////////////////////////////////////////////////////////////////////////////////
////
//// Copyright © 2002 Huimei Zheng, zhenghm@hotmail.com
////
//// This source file may be used and distributed without restriction provided that
//// this copyright statement is not removed from the file and that any derivative
//// work contains the original copyright notice and the associated disclaimer.
////
////////////////////////////////////////////////////////////////////////////////////////////////


/*****************************************************************************
 Analog to digital control block, including fifo interface to ahb-apb bridge
 1. when psel0 is 1'b1, paddr == 0 will select fifo data output port
 2. paddr == 1 will select fifo status output
 *****************************************************************************/

`timescale 1ns/100ps

module adc(hclk, hresetn, pwdata, sbe, pwrite, paddr,
        psel, penable, ffwr, ffdata, fffull,
        prdata, devready, adenable
        );
input           hclk;
input           hresetn;
input [31:0]    pwdata;
input [3:0]     sbe;
input           pwrite;
input [7:0]     paddr;
input           psel;
input           penable;
input           ffwr;
input [31:0]    ffdata;

output [31:0]   prdata;
output          devready;
output          adenable;
output          fffull;

wire            ffempty;
wire [2:0]      wptr, rptr, ffsts;
wire [31:0]     ff0d, ff1d, ff2d, ff3d, ff4d, ff5d, ff6d, ff7d, outdata;

wire        ffoportsel = (paddr == 8'b0);
wire        ffstsel = (paddr == 8'b1);
wire      regwr = pwrite & psel & ffstsel;
wire      ffrd = ~pwrite & psel & penable & ffoportsel;
wire      ffrst = ~pwrite & psel & penable & ffstsel;

assign     devready = ~ (ffempty & ffoportsel & psel);

// register write

reg [7:0]   ffregister; //fifo register setting, enable bit inside

always @(hresetn or sbe or pwdata or regwr) begin
  if(~hresetn) ffregister <= 8'b0;
  else if(sbe[0] & regwr) ffregister[7:0] <= pwdata[7:0];
end

assign adenable = ffregister[0];

adfifoctl ififoctl(.hresetn(hresetn), .ffwr(ffwr), .hclk(hclk),
            .ffrd(ffrd), .ffsts(ffsts), .fffull(fffull),
            .ffempty(ffempty), .wptr(wptr), .rptr(rptr)
            );

fifowr fifowra(.ffdata(ffdata), .ffwr(ffwr),.wptr(wptr),
            .ff0d(ff0d), .ff1d(ff1d), .ff2d(ff2d), .ff3d(ff3d),
            .ff4d(ff4d), .ff5d(ff5d), .ff6d(ff6d), .ff7d(ff7d)
            );

fiford fiforda(.rptr(rptr),
            .ff0d(ff0d), .ff1d(ff1d), .ff2d(ff2d), .ff3d(ff3d),
            .ff4d(ff4d), .ff5d(ff5d), .ff6d(ff6d), .ff7d(ff7d),
            .outdata(outdata)
            );

reg [31:0]  prdata;

always @(ffrd or ffrst or outdata or ffregister or ffsts) begin
  if(ffrd) prdata <= outdata;
  else if(ffrst) prdata <= {21'b0, ffsts[2:0], ffregister[7:0]}; //total 32 bits
  else prdata <= 32'bz;
end

endmodule
```

A.4 A/D FIFO control code

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////
////
//// File name: adfifoctl.v
////
//// This file is part of the "A SoC Digital Interface IP Core" project
////
//// Author(s):     - Huimei Zheng, zhenghm@hotmail.com
////
//////////////////////////////////////////////////////////////////////////////////////////////////////////////
////
//// Copyright © 2002 Huimei Zheng, zhenghm@hotmail.com
////
//// This source file may be used and distributed without restriction provided that
//// this copyright statement is not removed from the file and that any derivative
//// work contains the original copyright notice and the associated disclaimer.
////
//////////////////////////////////////////////////////////////////////////////////////////////////////////////


/*********************************************
 AD Fifo control module
 *********************************************/

`timescale 1ns/100ps

 module adfifoctl(hresetn, ffwr, hclk, ffrd, ffsts,
                  fffull, ffempty, wptr, rptr
                  );
 input          hresetn;
 input          ffwr;
 input          hclk;
 input          ffrd;

 output [2:0]   ffsts;
 output         ffempty;
 output         fffull;
 output [2:0]    wptr;
 output [2:0]    rptr;

 parameter    delay = 1;

 /*
  * delay ffwr twice to get rid of hold time hazard
  */

 wire        wrclk0, wrclk1;
 assign       wrclk0 = ffwr & hresetn; //delay purpose only
 assign       wrclk1 = wrclk0 & hresetn;  //delay purpose only

 /*
  * write pointer generation
  */
  reg [2:0] wptr;
```

```
always @(posedge wrclk1 or negedge hresetn) begin
  if(~hresetn) wptr <= 3'b0;
  else wptr <= #delay wptr + 3'b1;
end

/*
 * Read pointer generation on every falling edge of ffrd
 * read pointer increased by 1
 */

reg [2:0] rptr;

always @(negedge ffrd or negedge hresetn) begin
 if( ~hresetn )
  rptr <= #delay 3'b0;
 else
  rptr <= #delay rptr + 3'b1;
end // always

/*
 *  Gray coded read status pointer
 */

reg [2:0] gwptr;

// Gray coded write status pointer
always @(posedge ffwr or negedge hresetn) begin
  if( ~hresetn )
   gwptr <= 3'b0;
  else
   case( gwptr )  // synopsys full_case parallel_case
     3'b000: gwptr <= #delay 3'b001;
     3'b001: gwptr <= #delay 3'b011;
     3'b011: gwptr <= #delay 3'b010;
     3'b010: gwptr <= #delay 3'b110;
     3'b110: gwptr <= #delay 3'b111;
     3'b111: gwptr <= #delay 3'b101;
     3'b101: gwptr <= #delay 3'b100;
     3'b100: gwptr <= #delay 3'b000;
   endcase // case( gwptr )
 end // always @ (posedge ffwr or negedge hresetn)

 reg [2:0] gwptrsy;
 //gwptr sync on hclk, ignore timing check on the flip-flops
 always @(posedge hclk or negedge hresetn) begin
   if(~hresetn) gwptrsy <= 3'b0;
   else gwptrsy <= gwptr;
 end

 reg [2:0]  bgwptrsy;
 // Convert synchronized read status back to binary
 //  This is gray code to binary conversion table!!

 always @( gwptrsy ) begin
   case( gwptrsy ) // synopsys full_case parallel_case
```

```
    // This is gray code to binary conversion table!!
    3'b000: bgwptrsy <= #delay 3'b000;
    3'b001: bgwptrsy <= #delay 3'b001;
    3'b011: bgwptrsy <= #delay 3'b010;
    3'b010: bgwptrsy <= #delay 3'b011;
    3'b110: bgwptrsy <= #delay 3'b100;
    3'b111: bgwptrsy <= #delay 3'b101;
    3'b101: bgwptrsy <= #delay 3'b110;
    3'b100: bgwptrsy <= #delay 3'b111;
  endcase // case( gwptrsy )
end // always @ ( gwptrsy )

assign #delay ffsts = bgwptrsy - rptr;
assign #delay fffull = (ffsts == 3'b111) ? 1'b1 : 1'b0;
assign #delay ffempty = (ffsts == 3'b000) ? 1'b1 : 1'b0;


endmodule
```

A.5 D/A FIFO code

```
////////////////////////////////////////////////////////////////////////////////////////////////////
////
//// File name: dac.v
////
//// This file is part of the "A SoC Digital Interface IP Core" project
////
//// Author(s):     - Huimei Zheng, zhenghm@hotmail.com
////
////////////////////////////////////////////////////////////////////////////////////////////////////
////
//// Copyright © 2002 Huimei Zheng, zhenghm@hotmail.com
////
//// This source file may be used and distributed without restriction provided that
//// this copyright statement is not removed from the file and that any derivative
//// work contains the original copyright notice and the associated disclaimer.
////
////////////////////////////////////////////////////////////////////////////////////////////////////


/*******************************************************************************
 Digital to analog control block, including fifo interface to ahb-apb bridge
 1. when psel is 1'b1, paddr == 0 will select fifo data input port
 2. paddr == 1 will select fifo status output
 *******************************************************************************/

`timescale 1ns/100ps

 module dac(hclk, hresetn, pwdata, sbe, pwrite, paddr,
             psel, penable, daclk, dadata, ffempty,
            prdata, devready, daenable
             );
 input          hclk;
 input          hresetn;
 input [31:0]   pwdata;
 input [3:0]    sbe;
 input          pwrite;
 input [7:0]    paddr;
 input          psel;
 input          penable;
 input          daclk;
 output [31:0]  prdata;
 output         devready;
 output         daenable;
 output [31:0]  dadata;
 output         ffempty;

 wire           fffull;
 wire [2:0]     wptr, rptr, ffsts;
 wire [31:0]    ff0d, ff1d, ff2d, ff3d, ff4d, ff5d, ff6d, ff7d;

 wire       ffinportsel = (paddr == 8'b0);
 wire       regsel = (paddr == 8'b1);
 wire       regwr = pwrite & psel & regsel;
wire     read = ~pwrite & psel & penable & regsel;
wire     ffcpuwr = pwrite & psel & penable & ffinportsel;

assign    devready = ~(fffull & ffinportsel & psel);

// register write
reg [7:0]   ffregister; //fifo register setting, enable bit inside

always @(hresetn or sbe or regwr or pwdata) begin
  if(~hresetn) ffregister <= 8'b0;
  else if(sbe[0] & regwr) ffregister[7:0] <= pwdata[7:0];
end

assign daenable = ffregister[0];

dafifoctl idafifoctl(.hresetn(hresetn), .ffcpuwr(ffcpuwr), .hclk(hclk),
              .daclk(daclk), .ffsts(ffsts), .fffull(fffull),
              .ffempty(ffempty), .wptr(wptr), .rptr(rptr)
              );

fifowr fifowrd(.ffdata(pwdata), .ffwr(ffcpuwr),.wptr(wptr),
           .ff0d(ff0d), .ff1d(ff1d), .ff2d(ff2d), .ff3d(ff3d),
           .ff4d(ff4d), .ff5d(ff5d), .ff6d(ff6d), .ff7d(ff7d)
           );

fiford fifordd(.rptr(rptr),
           .ff0d(ff0d), .ff1d(ff1d), .ff2d(ff2d), .ff3d(ff3d),
           .ff4d(ff4d), .ff5d(ff5d), .ff6d(ff6d), .ff7d(ff7d),
           .outdata(dadata)
           );

reg [31:0] prdata;

always @(read or ffregister or ffsts) begin
  if(read) begin
    prdata <= {21'b0, ffsts[2:0], ffregister[7:0]}; //total 32 bits
  end
  else prdata <= 32'bz;
end

endmodule
```

A.6 D/A FIFO control code

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////
////
//// File name: dafifoctl.v
////
//// This file is part of the "A SoC Digital Interface IP Core" project
////
//// Author(s):     - Huimei Zheng, zhenghm@hotmail.com
////
////////////////////////////////////////////////////////////////////////////////////////////////////////////
////
//// Copyright © 2002 Huimei Zheng, zhenghm@hotmail.com
////
//// This source file may be used and distributed without restriction provided that
//// this copyright statement is not removed from the file and that any derivative
//// work contains the original copyright notice and the associated disclaimer.
////
////////////////////////////////////////////////////////////////////////////////////////////////////////////

/*******************************************
 DA Fifo control module
 *******************************************/

`timescale 1ns/100ps

module dafifoctl(hresetn, ffcpuwr, hclk, daclk, ffsts, fffull,
                 ffempty, wptr, rptr
                 );
input           hresetn;
input           ffcpuwr;
input           hclk;
input           daclk;

output [2:0]    ffsts;
output          fffull;
output          ffempty;
output [2:0]    wptr;
output [2:0]    rptr;
parameter     delay = 1;

 /*
  * delay ffwr twice to get rid of hold time hazard
  */
 wire        wrclk0, wrclk1;
 assign       wrclk0 = ffcpuwr & hresetn;  //delay purpose only
 assign       wrclk1 = wrclk0 & hresetn;  //delay purpose only

 /*
  * write pointer generation
  */

reg [2:0] wptr;

 always @(negedge wrclk1 or negedge hresetn) begin
```

```
  if(~hresetn) wptr <= 3'b0;
  else wptr <= #delay wptr + 3'b1;
end

/*
 * Read pointer generation on every falling edge of daclk
 * read pointer increased by 1
 */

reg [2:0] rptr;

always @(posedge daclk or negedge hresetn) begin
 if( ~hresetn )
  rptr <= #delay 3'b0;
 else
  rptr <= #delay rptr + 3'b1;
end // always

/*
 *  Gray coded read status pointer
 */

reg [2:0] grptr;

// Gray coded read status pointer
always @(posedge daclk or negedge hresetn) begin
  if( ~hresetn )
   grptr <= 3'b0;
  else
   case( grptr )  // synopsys full_case parallel_case
     3'b000: grptr <= #delay 3'b001;
     3'b001: grptr <= #delay 3'b011;
     3'b011: grptr <= #delay 3'b010;
     3'b010: grptr <= #delay 3'b110;
     3'b110: grptr <= #delay 3'b111;
     3'b111: grptr <= #delay 3'b101;
     3'b101: grptr <= #delay 3'b100;
     3'b100: grptr <= #delay 3'b000;
   endcase // case( grptr )
end // always @ (posedge daclk or negedge hresetn)

reg [2:0] grptrsy;
//grptr sync on hclk, ignore timing check on the flip-flops
always @(posedge hclk or negedge hresetn) begin
   if(~hresetn) grptrsy <= 3'b0;
   else grptrsy <= grptr;
end

reg [2:0]  bgrptrsy;
// Convert synchronized read status back to binary
always @( grptrsy ) begin
  case( grptrsy )  // synopsys full_case parallel_case
    3'b000: bgrptrsy <= #delay 3'b000;
    3'b001: bgrptrsy <= #delay 3'b001;
    3'b011: bgrptrsy <= #delay 3'b010;
    3'b010: bgrptrsy <= #delay 3'b011;
    3'b110: bgrptrsy <= #delay 3'b100;
    3'b111: bgrptrsy <= #delay 3'b101;
    3'b101: bgrptrsy <= #delay 3'b110;
    3'b100: bgrptrsy <= #delay 3'b111;
  endcase // case( grptrsy )
end // always @ ( grptrsy )

assign #delay ffsts = wptr - bgrptrsy;
assign #delay ffempty = (ffsts == 3'b000) ? 1'b1 : 1'b0;
assign #delay fffull = (ffsts == 3'b111) ? 1'b1 : 1'b0;

endmodule
```

A.7 FIFO write code

```
////////////////////////////////////////////////////////////////////////////////////////////////////
////
//// File name: fifowr.v
////
//// This file is part of the "A SoC Digital Interface IP Core" project
////
//// Author(s):     - Huimei Zheng, zhenghm@hotmail.com
////
////////////////////////////////////////////////////////////////////////////////////////////////////
////
//// Copyright © 2002 Huimei Zheng, zhenghm@hotmail.com
////
//// This source file may be used and distributed without restriction provided that
//// this copyright statement is not removed from the file and that any derivative
//// work contains the original copyright notice and the associated disclaimer.
////
////////////////////////////////////////////////////////////////////////////////////////////////////

`timescale 1ns/100ps

module fifowr(ffdata, ffwr, wptr,
              ff0d, ff1d, ff2d, ff3d, ff4d, ff5d, ff6d, ff7d
              );
 input [31:0]   ffdata;
 input          ffwr;
 input [2:0]    wptr;
 output [31:0]  ff0d;
 output [31:0]  ff1d;
 output [31:0]  ff2d;
 output [31:0]  ff3d;
 output [31:0]  ff4d;
 output [31:0]  ff5d;
 output [31:0]  ff6d;
 output [31:0]  ff7d;

 reg [31:0]    ff0d, ff1d, ff2d, ff3d, ff4d, ff5d, ff6d, ff7d;

 always @(ffwr or wptr or ffdata) begin
   if(ffwr) begin
     case(wptr[2:0])
       3'h0: ff0d <= ffdata;
       3'h1: ff1d <= ffdata;
       3'h2: ff2d <= ffdata;
       3'h3: ff3d <= ffdata;
       3'h4: ff4d <= ffdata;
       3'h5: ff5d <= ffdata;
       3'h6: ff6d <= ffdata;
       3'h7: ff7d <= ffdata;
     endcase // case(wptr[2:0])
   end
 end
 endmodule
```

A.8 FIFO read code

```
////////////////////////////////////////////////////////////////////////////////////////////////////
////
//// File name: fiford.v
////
//// This file is part of the "A SoC Digital Interface IP Core" project
////
//// Author(s):    - Huimei Zheng, zhenghm@hotmail.com
////
////////////////////////////////////////////////////////////////////////////////////////////////////
////
//// Copyright © 2002 Huimei Zheng, zhenghm@hotmail.com
////
//// This source file may be used and distributed without restriction provided that
//// this copyright statement is not removed from the file and that any derivative
//// work contains the original copyright notice and the associated disclaimer.
////
////////////////////////////////////////////////////////////////////////////////////////////////////

`timescale 1ns/100ps

module fiford(rptr,
              ff0d, ff1d, ff2d, ff3d,
              ff4d, ff5d, ff6d, ff7d,
              outdata
              );
 input [2:0]    rptr;
 input [31:0]   ff0d;
 input [31:0]   ff1d;
 input [31:0]   ff2d;
 input [31:0]   ff3d;
 input [31:0]   ff4d;
 input [31:0]   ff5d;
 input [31:0]   ff6d;
 input [31:0]   ff7d;
 output [31:0]  outdata;

 reg [31:0]     outdata;

 always @(rptr or ff0d or ff1d or ff2d or ff3d or ff4d or ff5d or ff6d or ff7d) begin
   case(rptr[2:0]) //synopsys full_case parallel_case
     3'b000: outdata <= ff0d;
     3'b001: outdata <= ff1d;
     3'b010: outdata <= ff2d;
     3'b011: outdata <= ff3d;
     3'b100: outdata <= ff4d;
     3'b101: outdata <= ff5d;
     3'b110: outdata <= ff6d;
     3'b111: outdata <= ff7d;
   endcase // case(rptr[2:0])
 end

endmodule
```

A.9 Testbench

```
//////////////////////////////////////////////////////////////////////////////////////////
////
//// File name: stimulus.v
////
//// This file is the testbench of the "A SoC Digital Interface IP Core" project
////
//// Author(s):     - Huimei Zheng, zhenghm@hotmail.com
////
//////////////////////////////////////////////////////////////////////////////////////////
////
//// Copyright © 2002 Huimei Zheng, zhenghm@hotmail.com
////
//// This source file may be used and distributed without restriction provided that
//// this copyright statement is not removed from the file and that any derivative
//// work contains the original copyright notice and the associated disclaimer.
////
//////////////////////////////////////////////////////////////////////////////////////////

`timescale 1ns/100ps

 module stimulus;

 reg         hclk;
 reg         hresetn;
 reg [31:0]  haddr;
 reg [1:0]   htrans;
 reg         hwrite;
 reg [2:0]   hsize;
 reg [31:0]  hwdata;
 reg         ffwr; //fifo write clock
 reg [31:0]  ffdata;
 reg         daclk; // fifo read clock

 wire [31:0] hrdata;
 wire        hready;
 wire        penable;
 wire        fffulla;
 wire        ffemptyd;
 wire        adenable;
 wire        daenable;
 wire [31:0] dadata; //output write data

interface inter(hclk, hresetn, haddr, htrans, hwrite, hsize, hwdata,
            ffwr, ffdata, daclk, hrdata, hready, penable,
            adenable, daenable, dadata,fffulla, ffemptyd
            );

initial begin
   $shm_open("stimulus.db");
   $shm_probe(hclk, hresetn, haddr, htrans, hwrite, hsize, hwdata,
            ffwr, ffdata, daclk, hrdata, hready, penable,
            adenable, daenable, dadata,fffulla, ffemptyd
            );
 end

initial  begin
 $display("\t\tTime  haddr  ad ffwr    ffdata     hrdata da daclk   hwdata   dadata  penable\n");
 $monitor("%d %h %b  %b %d %h   %b   %b %d %d    %b",
          $time, haddr, adenable, ffwr, ffdata, hrdata, daenable, daclk,
             hwdata, dadata, penable);
 end


initial  begin
   hclk = 1'b0;
   hresetn = 1'b0;
   haddr = 32'b0;
   daclk = 1'b0;
   ffdata = 32'b0;
 end
always #10 hclk = ~hclk;

initial
 begin
  // accesses A/D
   #15 hresetn = 1'b1; htrans = 2; hsize = 0;
   @(posedge hclk)
   #1
   hwrite=1'b1;
   haddr = 32'h80000004; hwdata = 57;
   $display("Access A/D converter, write register");
   #20
   @(posedge hclk);  //read status
   #1 hwrite = 1'b0;
   #30
   @(posedge hclk);  //read data
   #1 haddr = 32'h80000000;
   $display("Read A/D data");
   #30
   @(posedge hclk);  //read status
   #1 haddr = 32'h80000004;
   $display("Read A/D status");
   #30
   @(posedge hclk); //read status
   #1 haddr = 32'h80001000;

  //accesses D/A

   #30 hresetn = 1'b0; haddr = 32'h0;
   @(posedge hclk)
   #1 hresetn = 1'b1; htrans = 3; hsize = 2;
   @(posedge hclk)
   #1 hwrite = 1'b1;
   haddr = 32'h80000804; hwdata = 19;
   $display("Access D/A converter, write register");
   #20
   @(posedge hclk);  //write data
   #1 haddr = 32'h80000800; hwdata = 459;
   $display("Write data to D/A" );
  #20
  @(posedge hclk); //write data
  #1 haddr = 32'h80000C00; hwdata = 6545;
  $display("Write data to D/A");
  #20
  @(posedge hclk); //read status
  #1 hwrite = 1'b0; haddr = 32'h80000804;
  $display("Read D/A status" );
  #30
  @(posedge hclk);
  #1 haddr = 32'h80001000;

#40 $finish;
 end

  always begin
  #5 ffwr = 1'b0;
 if(adenable) begin
   #30 ffwr = 1'b1;
   #5  ffwr = 1'b0;
   #5  ffdata = ffdata + 1'b1;
   end
 else if (daenable) begin
  #65 daclk = 1'b1;
  #25 daclk = 1'b0;
   end
end

endmodule
```

Appendix B

The Synopsys Report for the Gate-level Design

B.1 The area report

```
****************************************
Report : area
Design : interface
Version: 2000.11-SP1
Date   : Thu Aug  8 14:47:03 2002
****************************************

Library(s) Used:

   tcb773pwc (File: /CMC/tools/synopsys/syn_2000.11-SP1/cmc/cmosp35/syn/tcb773pwc.db)
   tpd773pnwc (File: /CMC/tools/synopsys/syn_2000.11-SP1/cmc/cmosp35/syn/tpd773pnwc.db)

Number of ports:           176
Number of nets:            352
Number of cells:           177
Number of references:      3

Combinational area:        5505762.500000
Noncombinational area:     129482.500000
Net Interconnect area:     undefined  (Wire load has zero net area)

Total cell area:           5635245.000000
Total area:                undefined
```

B.2 The reference report

```
****************************************
Report : reference
Design : interface
Version: 2000.11-SP1
Date   : Thu Aug  8 14:47:03 2002
****************************************

Attributes:
    b - black box (unknown)
   bo - allows boundary optimization
    d - dont_touch
   mo - map_only
    h - hierarchical
    n - noncombinational
    r - removable
    s - synthetic operator
    u - contains unmapped logic

Reference    Library        Unit Area     Count    Total Area       Attributes
------------------------------------------------------------------------------
PDI          tpd773pnwc     30660.000000    106    3249960.000000
PDO08C       tpd773pnwc     30660.000000     70    2146200.000000
top                        239085.000000      1     239085.000000   h,n
-----------------------------------------------------------------------------
Total 3 references        5635245.000000

****************************************
Report : reference
Design : top
Version: 2000.11-SP1
Date   : Fri Aug  9 20:01:49 2002
****************************************

Attributes:
    b - black box (unknown)
   bo - allows boundary optimization
    d - dont_touch
   mo - map_only
    h - hierarchical
    n - noncombinational
    r - removable
    s - synthetic operator
    u - contains unmapped logic

Reference       Library       Unit Area     Count    Total Area     Attributes
------------------------------------------------------------------------------
AN2D1           tcb773pwc      70.000000        5     350.000000
AN3D1           tcb773pwc      87.500000       23    2012.500000
AN4D1           tcb773pwc     105.000000        2     210.000000
DFCN1N          tcb773pwc     332.500000        2     665.000000    n
DFCN1Q          tcb773pwc     332.500000       19    6317.500000    n
```

```
DFCNE1Q     tcb773pwc   402.500000     2     805.000000  n
DFCNS1Q     tcb773pwc   437.500000    11    4812.500000  n
DFSN1N      tcb773pwc   332.500000     1     332.500000  n
DFXCN1N     tcb773pwc   420.000000     1     420.000000  n
HA1D1       tcb773pwc   245.000000     4     980.000000  r
ICB0        tcb773pwc   350.000000   100   35000.000000
IND2D1      tcb773pwc    70.000000   350   24500.000000
IND2D2      tcb773pwc   122.500000     1     122.500000
IND3D1      tcb773pwc    87.500000    23    2012.500000
IND3D2      tcb773pwc   157.500000     3     472.500000
INV0        tcb773pwc    35.000000   112    3920.000000
INV1        tcb773pwc    52.500000     7     367.500000
INV3        tcb773pwc    70.000000     2     140.000000
INVTN1      tcb773pwc    87.500000    96    8400.000000  n
IOA22D2A    tcb773pwc   175.000000    43    7525.000000
IOA221D1A   tcb773pwc   210.000000    11    2310.000000
IOA221D1B   tcb773pwc   175.000000    85   14875.000000
ITB2        tcb773pwc   385.000000     1     385.000000
ITBN0       tcb773pwc   350.000000     2     700.000000
LH1N        tcb773pwc   175.000000     1     175.000000  n
LH1Q        tcb773pwc   175.000000   567   99225.000000  n
LN1Q        tcb773pwc   175.000000    40    7000.000000  n
MAOI22D0    tcb773pwc   105.000000     4     420.000000
MOAI22D0    tcb773pwc   105.000000     2     210.000000
MUX2D1      tcb773pwc   122.500000    10    1225.000000
ND2D0       tcb773pwc    52.500000     1      52.500000
ND8D1       tcb773pwc   210.000000    32    6720.000000
NR4D0       tcb773pwc   105.000000     1     105.000000
NR7D1       tcb773pwc   227.500000     1     227.500000
NR8D1       tcb773pwc   245.000000     2     490.000000
OAI21D0     tcb773pwc    70.000000     1      70.000000
OR2D1       tcb773pwc    87.500000    21    1837.500000
OR3D1       tcb773pwc   105.000000     1     105.000000
OR4D1       tcb773pwc   140.000000     3     420.000000
TFCN1Q      tcb773pwc   332.500000     4    1330.000000  n
XOR2D1      tcb773pwc   122.500000    15    1837.500000
-----------------------------------------------------------
Total 41 references         239085.000000
```

Vita Auctoris

Name:	Huimei Zheng
Place of Birth:	Shannxi, China
Year of Birth:	1973
Eduction:	M.A.Sc, University of Windsor, Canada 2000 – 2002
	B.Eng, Hunan University, China 1990 – 1994