FP6-IST-507219

PROSYD:

Property-Based System Design

Instrument: Specific Targeted Research Project

Thematic Priority: Information Society Technologies

A PSL Specification for WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores (Deliverable: Annex of D1.4/1)

Due date of deliverable: October 31, 2006
Actual submission date: October 31, 2006

Start date of project: January 1, 2004 Duration: Three years

Organisation name of lead contractor for this deliverable: STM

Revision 1.0

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	☒
PP	Restricted to other programme participants (including the Commission Services)	☐
RE	Restricted to a group specified by the consortium (including the Commission Services)	☐
CO	Confidential, only for members of the consortium (including the Commission Services)	☐

Notices

For information, contact Anthony McIsaac.

This document is intended to fulfil the contractual obligations of the PROSYD project concerning deliverable: Annex of D1.4/1 described in contract number 507219.

Table of Revisions

Version	Date	Description and reason	By	Affected sections
0.1	October 9, 2006	Complete PSL annotation of the English specification	Dana Fisman	All
0.2	October 11, 2006	Editing	Anne Lustig-Picus	All
0.3	October 16, 2006	Editing and addition of executive summary etc.	Dana Fisman	All
1.0	October 18, 2006	Editing	Anne Lustig-Picus	All

Authors*

Gadiel Auerbach
Dana Fisman

*We would like to thank Dmitry Pidan for various discussions regarding the specification of Wishbone in PSL using the Verilog flavor.

Executive Summary

This report annotates the English specification for WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores with a mathematically precise PSL specification. The PSL specification follows the methodology described in deliverables 1.1/1 and 1.1/2 of the Prosyd project.

This document takes the English specification and annotates it with a PSL specification. The PSL specification is always put on a shaded separate page, which is placed at the right of the English specification it describes. It is sometimes necessary to understand the context of the PSL specification (especially if it is used as input to a verification tool). Thus, the PSL page contains PSL remarks as well as PSL code.

As noted in D1.1/1 and D1.1/4 not every piece of English specification should/can be annotated with a corresponding PSL specification. For example, the parts of the specification defining required documentation, and non-discrete timing requirements such as RULE 5.10 (p.88) "The clock input [CLK_I] MUST have a duty cycle that is no less than 40%, and no greater than 60%."

The Wishbone protocols specifies how different IP cores (which may be soft-, firm- or hard-cores) should be interconnected together in order to accomplish standard data exchange between IP cores such as read/write cycles, block transfer cycle, and read-modify-write cycle. The specification supports various IP core interconnection means, including: point-to-point, shared bus, crossbar switch and data flow interconnection. The purpose of the protocol is to work for any number and any of

the above interconnection means of IP cores (that meets the defined requirements). The challenge in specifying the Wishbone protocol is to provide a generic specification that can be applied in any of the infinite number of the allowed settings.

To accomplish for this the specification was phrased using top level and low level properties. A top-level property is concerned with some main aspect of a correct data transfer cycle between all participating masters and slaves. For each such aspect a property, taking as parameters the relevant signals of one master and one slave, was phrased assuming the master and slave are the only ones participating and assuming they are interconnected in a point-to-point fashion. The top-level property was then phrased using quantification over all masters and slaves of the property describing the point-to-point connection of one master and one slave. This modularity is enabled since (1) the Wishbone specification makes the point-to-point interconnection an abstraction of all other interconnection types and (2) the point-to-point property is phrased to pass vacuously in case the master and slave are not connected or the master did not request the given slave.

Purpose

The purpose of this document is to provide a PSL specification for WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores.

Intended Audience

This document is intended for (1) anyone who is interested in the Wishbone specification, and (2) anyone who is interested in seeing an example of how a protocol describing the interconnections between several IP core modules is specified in PSL.

Background

PSL is a property specification language, which is an official IEEE since October 2005. PSL provides a means to write specifications that are both easy to read and mathematically precise. Thus, one important use of PSL is for documentation, either in place of or, more typically, in conjunction with a natural language specification. The PSL specification can then be used as input to the design phase, which benefits from a more precise starting point for logic design. In the verification phase, the PSL specification is reused to support the increasingly popular assertion based verification (ABV) style of verification.

WISHBONE COMPATIBLE

Specification for the:

WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores

Revision: B.3, Released: September 7, 2002

OPENCORES.ORG

Silicore™

Electronic Design • Sensors • IP Cores

This Page is Intentionally Blank

Stewardship

Stewardship for this specification is maintained by OpenCores Organization (hereafter OpenCores). Questions, comments and suggestions about this document are welcome and should be directed to:

Richard Herveille, OpenCores Organization
E-MAIL: rherveille@opencores.org   URL: www.opencores.org

OpenCores maintains this document to provide an open, freely useable interconnect architecture for its own and others' IP-cores. These specifications are intended to guarantee compatibility between compliant IP-cores and to improve cooperation among different users and suppliers.

Copyright & Trademark Release / Royalty Release / Patent Notice

Disclaimers

Document Format, Binding and Covers

This document is formatted for printing on double sided, 8½" x 11" white paper stock. It is designed to be bound within a standard cover. The preferred binding method is a black coil binding with outside diameter of 9/16" (14.5 mm). The preferred cover stock is Paper Direct part number KVR09D (forest green) and is available on-line at: www.paperdirect.com. Binding can be performed at most full-service copy centers such as Kinkos (www.kinkos.com).

Acknowledgements

Like any great technical project, the WISHBONE specification could not have been completed without the help of many people. The Steward wishes to thank the following for their ideas, suggestions and contributions:

Ray Alderman
Yair Amitay
Danny Cohan
Marc Delvaux
Miha Dolenc
Volker Hetzer
Magnus Homann
Brian Hurt
Linus Kirk
Damjan Lampret
Wade D. Peterson*
Barry Rice
John Rynearson
Avi Shamli
Rudolf Usselmann

(*) Wade D. Peterson from Silicore Corporation is the original author and steward. Without his dedication this specification would have never been where it is now.

Revision History

The various revisions of the WISHBONE specification, along with their changes and revision history, can be found at www.silicore.net/wishbone.htm.

Table of Contents

This page is Intentionally Blank

Chapter 1 - Introduction

The WISHBONE[1] System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores is a flexible design methodology for use with semiconductor IP cores. Its purpose is to foster design reuse by alleviating System-on-Chip integration problems. This is accomplished by creating a common interface between IP cores. This improves the portability and reliability of the system, and results in faster time-to-market for the end user.

Previously, IP cores used non-standard interconnection schemes that made them difficult to integrate. This required the creation of custom glue logic to connect each of the cores together. By adopting a standard interconnection scheme, the cores can be integrated more quickly and easily by the end user.

This specification can be used for soft core, firm core or hard core IP. Since firm and hard cores are generally conceived as soft cores, the specification is written from that standpoint.

This specification does not require the use of specific development tools or target hardware. Furthermore, it is fully compliant with virtually all logic synthesis tools. However, the examples presented in the specification do use the VHDL hardware description language. These are presented only as a convenience to the reader, and should be readily understood by users of other hardware description languages (such as Verilog®). Schematic based tools can also be used.

The WISHBONE interconnect is intended as a general purpose interface. As such, it defines the standard data exchange between IP core modules. It does not attempt to regulate the application-specific functions of the IP core.

The WISHBONE architects were strongly influenced by three factors. First, there was a need for a good, reliable System-on-Chip integration solution. Second, there was a need for a common interface specification to facilitate structured design methodologies on large project teams. Third, they were impressed by the traditional system integration solutions afforded by microcomputer buses such as PCI bus and VMEbus.

In fact, the WISHBONE architecture is analogous to a microcomputer bus in that that they both: (a) offer a flexible integration solution that can be easily tailored to a specific application; (b) offer a variety of bus cycles and data path widths to solve various system problems; and (c) allow products to be designed by a variety of suppliers (thereby driving down price while improving performance and quality).

[1] Webster's dictionary defines a WISHBONE as "the forked clavicle in front of the breastbone of most birds." The term 'WISHBONE interconnect' was coined by Wade Peterson of Silicore Corporation. During the initial definition of the scheme he was attempting to find a name that was descriptive of a bi-directional data bus that used either multiplexers or three-state logic. This was solved by forming an interface with separate input and output paths. When these paths are connected to three-state logic it forms a 'Y' shaped configuration that resembles a wishbone. The actual name was conceived during a Thanksgiving Day dinner that included roast turkey. Thanksgiving Day is a national holiday in the United States, and is observed on the third Thursday in November. It is generally celebrated with a traditional turkey dinner.

PSL Remark

The PSL specification in this document uses the Verilog flavor. The specification can be easily changed to use other flavors (e.g. VHDL flavor).

However, traditional microcomputer buses are fundamentally handicapped for use as a System-on-Chip interconnection. That's because they are designed to drive long signal traces and connector systems which are highly inductive and capacitive. In this regard, System-on-Chip is much simpler and faster. Furthermore, the System-on-Chip solutions have a rich set of interconnection resources. These do not exist in microcomputer buses because they are limited by IC packaging and mechanical connectors.

The WISHBONE architects have attempted to create a specification that is robust enough to insure complete compatibility between IP cores. However, it has not been over specified so as to unduly constrain the creativity of the core developer or the end user. It is believed that these two goals have been accomplished with the publication of this document.

1.1 WISHBONE Features

The WISHBONE interconnection makes System-on-Chip and design reuse easy by creating a standard data exchange protocol. Features of this technology include:

- Simple, compact, logical IP core hardware interfaces that require very few logic gates.
- Supports structured design methodologies used by large project teams.
- Full set of popular data transfer bus protocols including:

  - READ/WRITE cycle
  - BLOCK transfer cycle
  - RMW cycle

- Modular data bus widths and operand sizes.
- Supports both BIG ENDIAN and LITTLE ENDIAN data ordering.
- Variable core interconnection methods support point-to-point, shared bus, crossbar switch, and switched fabric interconnections.
- Handshaking protocol allows each IP core to throttle its data transfer speed.
- Supports single clock data transfers.
- Supports normal cycle termination, retry termination and termination due to error.
- Modular address widths

- Partial address decoding scheme for SLAVEs. This facilitates high speed address decoding, uses less redundant logic and supports variable address sizing and interconnection means.

- User-defined tags. These are useful for applying information to an address bus, a data bus or a bus cycle. They are especially helpful when modifying a bus cycle to identify information such as:

  - Data transfers
  - Parity or error correction bits
  - Interrupt vectors
  - Cache control operations

- MASTER / SLAVE architecture for very flexible system designs.

- Multiprocessing (multi-MASTER) capabilities. This allows for a wide variety of System-on-Chip configurations.

- Arbitration methodology is defined by the end user (priority arbiter, round-robin arbiter, etc.).

- Supports various IP core interconnection means, including:

  - Point-to-point
  - Shared bus
  - Crossbar switch
  - Data flow interconnection

  - Off chip

- Synchronous design assures portability, simplicity and ease of use.

- Very simple, variable timing specification.

- Documentation standards simplify IP core reference manuals.

- Independent of hardware technology (FPGA, ASIC, etc.).

- Independent of delivery method (soft, firm or hard core).

- Independent of synthesis tool, router and layout tool technology.

- Independent of FPGA and ASIC test methodologies.

- Seamless design progression between FPGA prototypes and ASIC production chips.

1.2 WISHBONE Objectives

The main objectives of this specification are

- to create a flexible interconnection means for use with semiconductor IP cores. This allows various IP cores to be connected together to form a System-on-Chip.

- to enforce compatibility between IP cores. This enhances design reuse.

- to create a robust standard, but one that does not unduly constrain the creativity of the core developer or the end user.

- to make it easy to understand by both the core developer and the end user.

- to facilitate structured design methodologies on large project teams. With structured design, individual team members can build and test small parts of the design. Each member of the design team can interface to the common, well-defined WISHBONE specification. When all of the sub-assemblies have been completed, the full system can be integrated.

- to create a portable interface that is independent of the underlying semiconductor technology. For example, WISHBONE interconnections can be made that support both FPGA and ASIC target devices.

- to make WISHBONE interfaces independent of logic signaling levels.

- to create a flexible interconnection scheme that is independent of the IP core delivery method. For example, it may be used with ‘soft core’, ‘firm core’ or ‘hard core’ delivery methods.

- to be independent of the underlying hardware description. For example, soft cores may be written and synthesized in VHDL, Verilog® or some other hardware description language. Schematic entry may also be used.

- to require a minimum standard for documentation. This takes the form of the WISHBONE DATASHEET, and allows IP core users to quickly evaluate and integrate new cores.

- to eliminate extensive interface documentation on the part of the IP core developer. In most cases, this specification along with the WISHBONE DATASHEET is sufficient to completely document an IP core data interface.

to allow users to create SoC components without infringing on the patent rights of others. While the use of WISHBONE technology does not necessarily prevent patent infringement, it does provide a reasonable safe haven where users can design around the patent claims of others. The specification also provides *cited patent references*, which describes the field of search used by the WISHBONE architects.

- to identify critical System-on-Chip interconnection technologies, and to place them into the public domain at the earliest possible date. This makes it more difficult for individuals and organizations to create proprietary technologies through the use of patent, trademark, copyright and trade secret protection mechanisms.

- to support a business model whereby IP Core suppliers can cooperate at a technical standards level, but can also compete in the commercial marketplace. This improves the overall quality and value of products through market forces such as price, service, delivery, performance and time-to-market. This business model also allows open source IP cores to be offered as well.

- to create an architecture that has a smooth transition path to support new technologies. This increases the longevity of the specification as it can adapt to new, and as yet unthought-of, requirements.

- to create an architecture that allows various interconnection means between IP core modules. This insures that the end user can tailor the System-on-Chip to his/her own needs. For example, the entire interconnection system (which is analogous to a backplane on a standard microcomputer bus like VMEbus or cPCI) can be created by the system integrator. This allows the interconnection to be tailored to the final target device.

- to create an architecture that requires a minimum of glue logic. In some cases the System-on-Chip needs no glue logic whatsoever. However, in other cases the end user may choose to use a more sophisticated interconnection method (for example with FIFO memories or crossbar switches) that requires additional glue logic.

- to create an architecture with variable address and data path widths to meet a wide variety of system requirements.

- to create an architecture that fully supports the automatic generation of interconnection and IP Core systems. This allows components to be generated with parametric core generators.

- to create an architecture that supports both BIG ENDIAN and LITTLE ENDIAN data transfer organizations.

A further objective is to create an architecture that supports one data transfer per clock cycle.

- to create a flexible architecture that allows address, data and bus cycles to be tagged. Tags are user defined signals that allow users to modify a bus cycle with additional information. They are especially useful when novel or unusual control signals (such as parity, cache control or interrupt acknowledge) are needed on an interface.

- to create an architecture with a MASTER/SLAVE topology. Furthermore, the system must be capable of supporting multiple MASTERs and multiple SLAVEs with an efficient arbitration mechanism.

- to create an architecture that supports point-to-point interconnections between IP cores.

- to create an architecture that supports shared bus interconnections between IP cores.

- to create an architecture that supports crossbar switches between IP cores.

- to create an architecture that supports switched fabrics.

- to create a synchronous protocol to insure ease of use, good reliability and easy testing. Furthermore, all transactions can be coordinated by a single clock.

- to create a synchronous protocol that works over a wide range of interface clock speeds. The effects of this are: (a) that the WISHBONE interface can work synchronously with all attached IP cores, (b) that the interface can be used on a large range of target devices, (c) that the timing specification is much simpler and (d) that the resulting semiconductor device is much more testable.

- to create a variable timing mechanism whereby the system clock frequency can be adjusted so as to control the power consumption of the integrated circuit.

- to create a synchronous protocol that provides a simple timing specification. This makes the interface very easy to integrate.

- to create a synchronous protocol where each MASTER and SLAVE can throttle the data transfer rate with a handshaking mechanism.

- to create a synchronous protocol that is optimized for System-on-Chip, but that is also suitable for off-chip I/O routing. Generally, the off-chip WISHBONE interconnect will operate at slower speeds.

- to create a backward compatible registered feedback high performance burst bus.

1.3 Specification Terminology

To avoid confusion, and to clarify the requirements for compliance, this specification uses five keywords. They are:

- **RULE**
- **RECOMMENDATION**
- **SUGGESTION**
- **PERMISSION**
- **OBSERVATION**

Any text not labeled with one of these keywords describes the operation in a narrative style. The keywords are defined as follows:

RULE
Rules form the basic framework of the specification. They are sometimes expressed in text form and sometimes in the form of figures, tables or drawings. All rules MUST be followed to ensure compatibility between interfaces. Rules are characterized by an imperative style. The upper-case words MUST and MUST NOT are reserved exclusively for stating rules in this document, and are not used for any other purpose.

RECOMMENDATION
Whenever a recommendation appears, designers would be wise to take the advice given. Doing otherwise might result in some awkward problems or poor performance. While this specification has been designed to support high performance systems, it is possible to create an interconnection that complies with all the rules, but has very poor performance. In many cases a designer needs a certain level of experience with the system architecture in order to design interfaces that deliver top performance. Recommendations found in this document are based on this kind of experience and are provided as guidance for the user.

SUGGESTION
A suggestion contains advice which is helpful but not vital. The reader is encouraged to consider the advice before discarding it. Some design decisions are difficult until experience has been gained. Suggestions help a designer who has not yet gained this experience. Some suggestions have to do with designing compatible interconnections, or with making system integration easier.

PERMISSION
In some cases a rule does not specifically prohibit a certain design approach, but the reader might be left wondering whether that approach might violate the spirit of the rule, or whether it might lead to some subtle problem. Permissions reassure the reader that a certain approach is acceptable and will not cause problems. The upper-case word MAY is reserved exclusively for stating a permission and is not used for any other purpose.

OBSERVATION
Observations do not offer any specific advice. They usually clarify what has just been discussed. They spell out the implications of certain rules and bring attention to things that might otherwise be overlooked. They also give the rationale behind certain rules, so that the reader understands why the rule must be followed.

1.4 Use of Timing Diagrams

Figure 1-1 shows some of the key features of the timing diagrams in this specification. Unless otherwise noted, the MASTER signal names are referenced in the timing diagrams. In some cases the MASTER and SLAVE signal names are different. For example, in the point-to-point interconnections the [ADR_O] and [ADR_I] signals are connected together. Furthermore, the actual waveforms at the SLAVE may vary from those at the MASTER. That's because the MASTER and SLAVE interfaces can be connected together in different ways. Unless otherwise noted, the timing diagrams refer to the connection diagram shown in Figure 1-2.

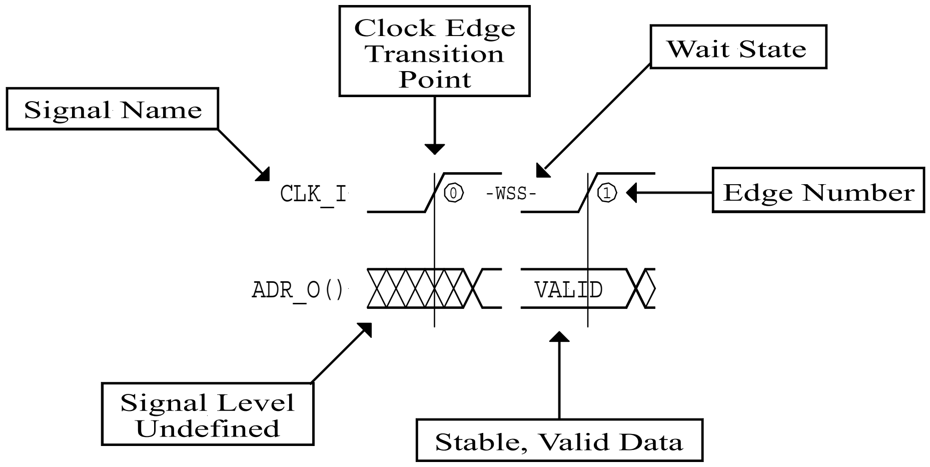


Figure 1-1. Use of timing diagrams.

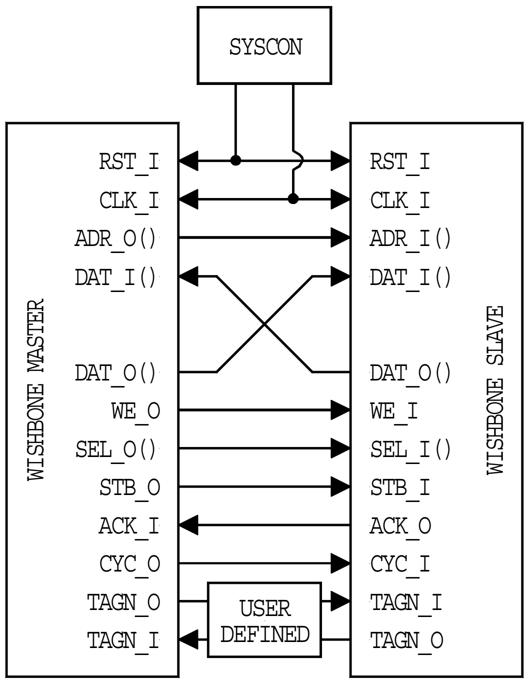


Figure 1-2. Standard connection for timing diagrams.

Some signals may or may not be present on a specific interface. That's because many of the signals are optional.

Two symbols are also presented in relation to the [CLK_I] signal. These include the positive going clock edge transition point and the clock edge number. In most diagrams a vertical guideline is shown at the positive-going edge of each [CLK_I] transition. This represents the theoretical transition point at which flip-flops register their input value, and transfer it to their output. The exact level of this transition point varies depending upon the technology used in the target device. The clock edge number is included as a convenience so that specific points in the timing diagram may be referenced in the text. The clock edge number in one timing diagram is not related to the clock edge number in another diagram.

Gaps in the timing waveforms may be shown. These indicate either: (a) a wait state or (b) a portion of the waveform that is not of interest in the context of the diagram. When the gap indicates a wait state, the symbols '-WSM-' or '-WSS-' are placed in the gap along the [CLK_I] waveform. These correspond to wait states inserted by the MASTER or SLAVE interfaces respectively. They also indicate that the signals (with the exception of clock transitions and hatched regions) will remain in a steady state during that time.

Undefined signal levels are indicated by a hatched region. This region indicates that the signal level is undefined, and may take any state. It also indicates that the current state is undefined, and should not be relied upon. When signal arrays are used, stable and predictable signal levels are indicated with the word 'VALID'.

1.5 Signal Naming Conventions

All signal names used in this specification have the '_I' or '_O' characters attached to them. These indicate if the signals are an input (to the core) or an output (from the core). For example, [ACK_I] is an input and [ACK_O] is an output. This convention is used to clearly identify the direction of each signal.

Signal arrays are identified by a name followed by a set of parenthesis. For example, [DAT_I()] is a signal array. Array limits may also be shown within the parenthesis. In this case the first number of the array limit indicates the most significant bit, and the second number indicates the least significant bit. For example, [DAT_I(63..0)] is a signal array with upper array boundary number sixty-three (the most significant bit), and lower array boundary number zero (the least significant bit). The array size on any particular core may vary. In many cases the array boundaries are omitted if they are irrelevant to the context of the description.

Special user defined signals, called *tags*, can also be used. Tags are assigned a *tag type* that indicates the exact timing to which the signal must adhere. For example, if a parity bit such as [PAR_O] is added to a data bus, it would probably be assigned a tag type of TAG TYPE: TGD_O(). This indicates that the signal will adhere to the timing diagrams shown for [TGD_O()], which are shown in the timing diagrams for each bus cycle. Also note that, while all tag types are specified as arrays (with parenthesis '()'), the actual tag does not have to be a signal array. It can also be non-arrayed signal.

When used as part of a sentence, signal names are enclosed in brackets '[ ]'. This helps to discriminate signal names from the words in the sentence.

1.6 WISHBONE Logo

The WISHBONE logo can be affixed to SoC documents that are compatible with this standard. Figure 1-3 shows the logo.

Figure 1-3. WISHBONE logo.

PERMISSION 1.00
Documents describing a WISHBONE compatible SoC component that are 100% compliant with this specification MAY use the WISHBONE logo.

1.7 Glossary of Terms

0x (numerical prefix)
The '0x' prefix indicates a hexadecimal number. It is the same nomenclature as commonly used in the 'C' programming language.

Active High Logic State
A logic state that is 'true' when the logic level is a binary '1' (high state). The high state is at a higher voltage than the low state.

Active Low Logic State
A logic state that is 'true' when the logic level is a binary '0' (low state). The low state is at a lower voltage than the high state.

Address Tag
One or more user defined signals that modify a WISHBONE address. For example, they can be used create a parity bit on an address bus, to indicate an address width (16-bit, 24-bit etc.) or can be used by memory management hardware to indicate a protected address space. All address tags must be assigned a tag type of [TGA_I()] or [TGA_O()]. Also see *tag, tag type*, *data tag* and *cycle tag*.

ASIC
Acronym for: Application Specific Integrated Circuit. A general term which describes a generic array of logic gates or analog building blocks which are programmed by a metalization layer at a silicon foundry. High level circuit descriptions are impressed upon the logic gates or analog building blocks in the form of metal interconnects.

Asserted
(1) A verb indicating that a logic state has switched from the inactive to the active state. When active high logic is used it means that a signal has switched from a logic low level to a logic high level. (2) *Assert:* to cause a signal line to make a transition from its logically false (inactive) state to its logically true (active) state. Opposite of *negated*.

Bit
A single binary (base 2) digit.

Bridge
An interconnection system that allows data exchange between two or more buses. The buses may have similar or different electrical, mechanical and logical structures.

Bus
(1) A common group of signals. (2) A signal line or a set of lines used by a data transfer system to connect a number of devices.

Bus Interface
An electronic circuit that drives or receives data or power from a bus.

Bus Cycle
The process whereby digital signals effect the transfer of data across a bus by means of an interlocked sequence of control signals. Also see: *Phase (bus cycle)*.

BYTE
A unit of data that is 8-bits wide. Also see: *WORD*, *DWORD* and *QWORD*.

Crossbar Interconnection (Crossbar Switch)
Crossbar switches are mechanisms that allow modules to connect and communicate. Each connection channel can be operated in parallel to other connection channels. This increases the data transfer rate of the entire system by employing parallelism. Stated another way, two 100 MByte/second channels can operate in parallel, thereby providing a 200 MByte/second transfer rate. This makes the crossbar switches inherently faster than traditional bus schemes. Crossbar routing mechanisms generally support dynamic configuration. This creates a configurable and reliable network system. Most crossbar architectures are also scalable, meaning that families of crossbars can be added as the needs arise. A crossbar interconnection is shown in Figure 1-4.

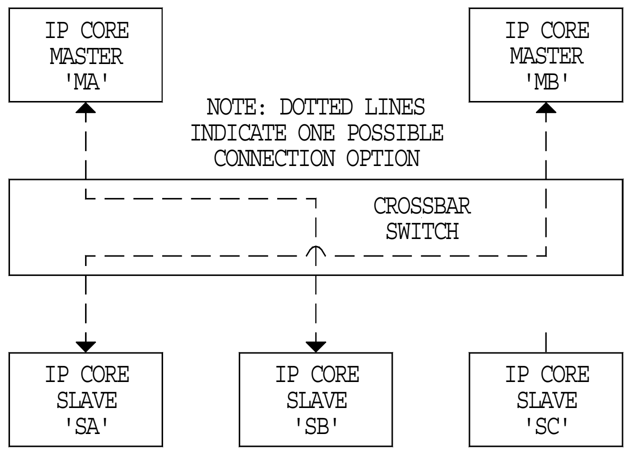


Figure 1-4. Crossbar (switch) interconnection.

Cycle Tag
One or more user defined signals that modify a WISHBONE bus cycle. For example, they can be used to discriminate between WISHBONE SINGLE, BLOCK and RMW cycles. All cycle tags must be assigned a tag type of [TGC_I()] or [TGC_O()]. Also see *tag type, address tag* and *data tag*.

Data Flow Interconnection
An interconnection where data flows through a prearranged set of IP cores in a sequential order. Data flow architectures often have the advantage of parallelism, whereby two or more functions are executed at the same time. Figure 1-5 shows a data flow interconnection between IP cores.

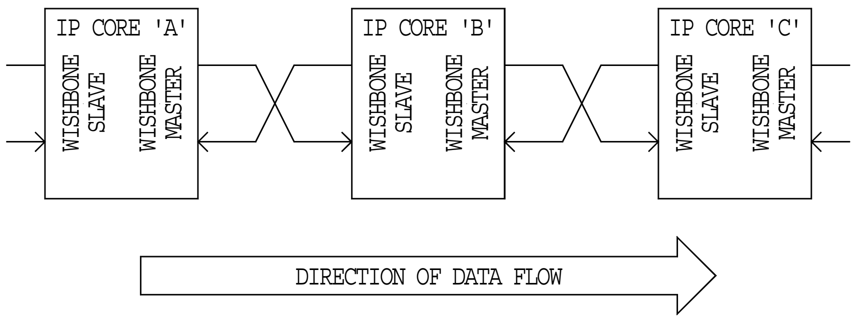


Figure 1-5. Data flow interconnection.

Data Organization

The ordering of data during a transfer. Generally, 8-bit (byte) data can be stored with the most significant byte of a mult-byte transfer at the higher or the lower address. These two methods are generally called BIG ENDIAN and LITTLE ENDIAN, respectively. In general, BIG ENDIAN refers to byte lane ordering where the most significant byte is stored at the lower address. LITTLE ENDIAN refers to byte lane ordering where the most significant byte is stored at the higher address. The terms BIG ENDIAN and LITTLE ENDIAN for data organization was coined by Danny Cohen of the Information Sciences Institute, and was derived from the book Gulliver's Travels by Jonathan Swift.

Data Tag
One or more user defined signals that modify a WISHBONE data transfer. For example, they can be used carry parity information, error correction codes or time stamps. All data tags must be assigned a tag type of [TGD_I()] or [TGD_O()]. Also see *tag type*, *address tag* and *cycle tag*.

DMA Unit
Acronym for Direct Memory Access Unit. (1) A device that transfers data from one location in memory to another location in memory. (2) A device for transferring data between a device and memory without interrupting program flow. (3) A device that does not use low-level instructions and is intended for transferring data between memory and/or I/O locations.

DWORD
A unit of data that is 32-bits wide. Also see: *BYTE*, *WORD* and *QWORD*.

ENDIAN
See the definition under 'Data Organization'.

FIFO
Acronym for: First In First Out. A type of memory used to transfer data between ports on two devices. In FIFO memories, data is removed in the same order that they were added. The FIFO memory is very useful for interconnecting cores of differing speeds.

Firm Core
An IP Core that is delivered in a way that allows conversion into an integrated circuit design, but does not allow the design to be easily reverse engineered. It is analogous to a binary or object file in the field of computer software design.

Fixed Interconnection
An interconnection system that is fixed, and *cannot* be changed without causing incompatibilities between bus modules (or SoC/IP cores). Also called a *static interconnection*. Examples of fixed interconnection buses include PCI, cPCI and VMEbus. Also see: *variable interconnection*.

Fixed Timing Specification
A timing specification that is based upon a fixed set of rules. Generally used in traditional microcomputer buses like PCI and VMEbus. Each bus module must conform to the ridged set of timing specifications. Also see: *variable timing specification*.

Foundry
See silicon foundry.

FPGA
Acronym for: Field Programmable Gate Array. Describes a generic array of logical gates and interconnect paths which are programmed by the end user. High level logic descriptions are impressed upon the gates and interconnect paths, often in the form of IP Cores.

Full Address Decoding
A method of address decoding where each SLAVE decodes all of the available address space. For example, if a 32-bit address bus is used, then each SLAVE decodes all thirty-two address bits. This technique is used on standard microcomputer buses like PCI and VMEbus. Also see: *partial address decoding*.

Gated Clock
A clock that can be stopped and restarted. In WISHBONE, a gated clock generator allows [CLK_O] to be stopped in its low state. This technique is often used to reduce the power consumption of an integrated circuit. Under WISHBONE, the gated clock generator is optional. Also see: *variable clock generator*.

Glue Logic
(1) Logic gates and interconnections required to connect IP cores together. The requirements for glue logic vary greatly depending upon the interface requirements of the IP cores. (2) A family of logic circuits consisting of various gates and simple logic elements, each of which serve as an interface between various parts of a computer system.

Granularity
The smallest unit of data transfer that a port is capable of transferring. For example, a 32-bit port can be broken up into four 8-bit BYTE segments. In this case, the granularity of the interface is 8-bits. Also see: *port size* and *operand size*.

Hard Core
An IP Core that is delivered in the form of a mask set (i.e. a graphical description of the features and connections in an integrated circuit).

Hardware Description Language (HDL)
(1) Acronym for: Hardware Description Language. Examples include VHDL and Verilog®. (2) A general-purpose language used for the design of digital electronic systems.

Interface
A combination of signals and data-ports on a module that is capable of either generating or receiving bus cycles. WISHBONE defines these as MASTER and SLAVE interfaces respectively. Also see: *MASTER and SLAVE interfaces.*

INTERCON
A WISHBONE module that interconnects MASTER and SLAVE interfaces.

IP Core
Acronym for: Intellectual Property Core. Also see: *soft core*, *firm core* and *hard core*.

Mask Set
A graphical description of the features and connections in an integrated circuit.

MASTER
A WISHBONE interface that is capable of generating bus cycles. All systems based on the WISHBONE interconnect must have at least one MASTER interface. Also see: *SLAVE.*

Memory Mapped Addressing
An architecture that allows data to be stored and recalled in memory at individual, binary addresses.

Minimization (Logic Minimization)
A process by which HDL synthesis, router or other software development tools remove unused logic. This is important in WISHBONE because there are optional signals defined on many of the interfaces. If a signal is unused, then the logic minimization tools will remove these signals and their associated logic, thereby making a faster and more efficient design.

Module
In the context of this specification, it's another name for an IP core.

Multiplexer Interconnection
An interconnection that uses multiplexers to route address, data and control signals. Often used for System-on-Chip (SoC) applications. Also see: *three-state bus interconnection*.

Negated
A verb indicating that a logic state has switched from the active to the inactive state. When active high logic is used it means that a signal has switched from a logic high level to a logic low level. Also see: *asserted*.

Off-Chip Interconnection

An off-chip interconnection is used when a WISHBONE interface extends off-chip. See Figure 1-6.

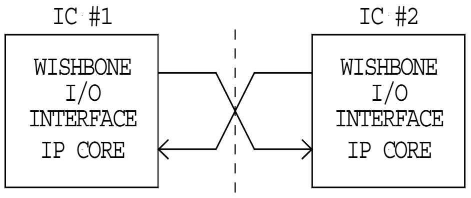

IC #1
IC #2
WISHBONE I/O INTERFACE IP CORE
WISHBONE I/O INTERFACE IP CORE

Figure 1-6. Off-chip interconnection.

Operand Size
The operand size is the largest single unit of data that is moved through an interface. For example, a 32-bit DWORD operand can be moved through an 8-bit port with four data transfers. Also see: *granularity* and *port size*.

Parametric Core Generator
A software tool used for the generation of IP cores based on input parameters. One example of a parametric core generator is a DSP filter generator. These are programs that create lowpass, bandpass and highpass DSP filters. The parameters for the filter are provided by the user, which causes the program to produce the digital filter as a VHDL or Verilog® hardware description. Parametric core generators can also be used create WISHBONE interconnections.

Partial Address Decoding
A method of address decoding where each SLAVE decodes only the range of addresses that it requires. For example, if the module needs only four addresses, then it decodes only the two least significant address bits. The remaining address bits are decoded by the interconnection system. This technique is used on SoC buses and has the advantages of less redundant logic in the system. It supports variable address buses, variable interconnection buses, and is relatively fast. Also see: *full address decoding*.

PCI
Acronym for: Peripheral Component Interconnect. Generally used as an interconnection scheme between integrated circuits. It also exists as a board level interconnection known as Compact PCI (or cPCI). While this specification is very flexible, it isn't practical for SoC applications.

Phase (Bus Cycle)
A periodic portion of a bus cycle. For example, a WISHBONE BLOCK READ cycle could contain ten phases, with each phase transferring a single 32-bit word of data. Collectively, the ten phases form the BLOCK READ cycle.

Point-to-point Interconnection
(1) An interconnection system that supports a single WISHBONE MASTER and a single WISHBONE SLAVE interface. It is the simplest way to connect two cores. See Figure 1-7. (2) A connection with only two endpoints.

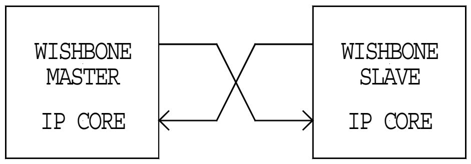


Figure 1-7. Point to point interconnection.

Port Size
The width of the WISHBONE data ports in bits. Also see: *granularity* and *operand size*.

QWORD
A unit of data that is 64-bits wide. Also see: *BYTE*, *WORD* and *DWORD*.

Router
A software tool that physically routes interconnection paths between logic gates. Applies to both FPGA and ASIC devices.

RTL
(1) Register-transfer logic. A design methodology that moves data between registers. Data is latched in the registers at one or more stages along the path of signal propagation. The WISHBONE specification uses a synchronous RTL design methodology where all registers use a common clock. (2) Register-transfer level. A description of computer operations where data transfers from register to register, latch to latch and through logic gates. (3) A level of description of a digital design in which the clocked behavior of the design is expressly described in terms of data transfers between storage elements (which may be implied) and combinatorial logic (which may represent any computing logic or arithmetic-logic-unit). RTL modeling allows design hierarchy that represents a structural description of other RTL models.

Shared Bus Interconnection
The shared bus interconnection is a system where a MASTER initiates addressable bus cycles to a target SLAVE. Traditional buses such as VMEbus and PCI bus use this type of interconnection. As a consequence of this architecture, only one MASTER at a time can use the interconnection resource (i.e. bus). Figure 1-8 shows an example of a WISHBONE shared bus interconnection.

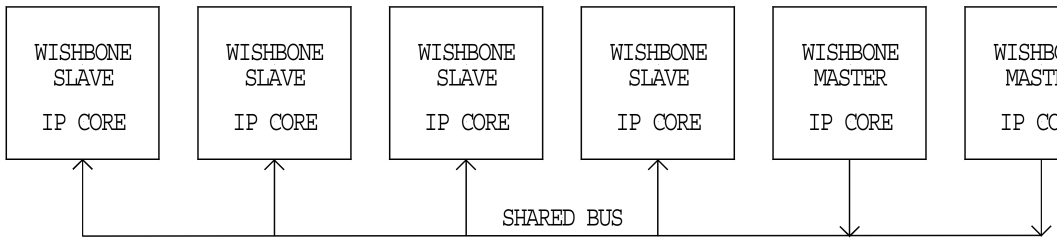


Figure 1-8. Shared bus interconnection.

Silicon Foundry
A factory that produces integrated circuits.

SLAVE
A WISHBONE interface that is capable of receiving bus cycles. All systems based on the WISHBONE interconnect must have at least one SLAVE. Also see: *MASTER*.

Soft Core
An IP Core that is delivered in the form of a hardware description language or schematic diagram.

SoC
Acronym for System-on-Chip. Also see: *System-on-Chip*.

Structured Design
(1) A popular method for managing complex projects that is often used with large project teams. When structured design practices are used, individual team members build and test small parts of the design with a common set of tools. Each sub-assembly is designed to a common standard. When all of the sub-assemblies have been completed, the full system can be integrated and tested. This approach makes it much easier to manage the design process. (2) Any disciplined approach to design that adheres to specified rules based on principles such as modularity and top-down design.

Switched Fabric Interconnection
A type of interconnection that uses large numbers of crossbar switches. These are organized into arrays that resemble the threads in a fabric. The resulting system is a network of redundant interconnections.

SYSCON
A WISHBONE module that drives the system clock [CLK_O] and reset [RST_O] signals.

System-on-Chip (SoC)
A method by which whole systems are created on a single integrated circuit chip. In many cases, this requires the use of IP cores which have been designed by multiple IP core providers. System-on-Chip is similar to traditional microcomputer bus systems whereby the individual components are designed, tested and built separately. The components are then integrated to form a finished system.

Tag
One or more characters or signals associated with a set of data, containing information about the set. Also see: *tag type*.

Tag Type
A special class of signals that is defined to ease user enhancements to the WISHBONE spec. When a user defined signal is specified, it is assigned a tag type that indicates the precise timing to which the signal must conform. This simplifies the creation of new signals. There are three basic tag types. These include address tags, data tags and cycle tags. These allow additional information to be attached to an address transfer, a data transfer or a bus cycle (respectively). The uppercase form TAG TYPE is used when specifying a tag type in the WISHBONE DATASHEET. For example, TAG TYPE: TGA_O() indicates an address tag. Also see: *address tag, data tag* and *cycle tag.*

Target Device
The semiconductor type (or technology) onto which the IP core design is impressed. Typical examples include FPGA and ASIC target devices.

Three-State Bus Interconnection
A microcomputer bus interconnection that relies upon three-state bus drivers. Often used to reduce the number of interconnecting signal paths through connector and IC pins. Three state buffers can assume a logic low state (‘0’ or ‘L’), a logic high state (‘1’ or ‘H’) or a high impedance state (‘Z’). Three-state buffers are sometimes called Tri-State® buffers. Tri-State® is a registered trademark of National Semiconductor Corporation. Also see: *multiplexer interconnection*.

Variable Clock Generator
A type of SYSCON module where the frequency of [CLK_O] can be changed dynamically. The frequency can be changed by way of a programmable phase-lock-loop (PLL) circuit or other control mechanism. Among other things, this technique is used to reduce the power consumption of the circuit. In WISHBONE the variable clock generator capability is optional. Also see: *gated clock generator* and *variable timing specification.*

Variable Interconnection
A microcomputer bus interconnection that *can* be changed without causing incompatibilities between bus modules (or SoC/IP cores). Also called a dynamic interconnection. An example of a variable interconnection bus is the WISHBONE SoC architecture. Also see: *fixed interconnection*.

Variable Timing Specification
A timing specification that is not fixed. In WISHBONE, variable timing can be achieved in a number of ways. For example, the system integrator can select the frequency of [CLK_O] by enforcing a timing specification during the circuit design. Variable timing can also be achieved during circuit operation with a variable clock generator. Also see: *gated clock generator* and *variable clock generator*.

Verilog®
A textual based hardware description language (HDL) intended for use in circuit design. The Verilog® language is both a synthesis and a simulation tool. Verilog® was originally a proprietary language first conceived in 1983 at Gateway Design Automation (Acton, MA), and was later refined by Cadence Corporation. It has since been greatly expanded and refined, and much of it has been placed into the public domain. Complete descriptions of the language can be found in the IEEE 1364 specification.

VHDL
Acronym for: VHSIC Hardware Description Language. [VHSIC: Very High Speed Integrated Circuit]. A textual based computer language intended for use in circuit design. The VHDL language is both a synthesis and a simulation tool. Early forms of the language emerged from US Dept. of Defense ARPA projects in the 1960’s, and have since been greatly expanded and refined. Complete descriptions of the language can be found in the IEEE 1076, IEEE 1073.3, IEEE 1164 specifications.

VMEbus

Acronym for: Versa Module Eurocard bus. A popular microcomputer (board) bus. While this specification is very flexible, it isn't practical for SoC applications.

WISHBONE
A flexible System-on-Chip (SoC) design methodology. WISHBONE establishes common interface standards for data exchange between modules within an integrated circuit chip. Its purpose is to foster design reuse, portability and reliability of SoC designs. WISHBONE is a public domain standard.

WISHBONE Classic
WISHBONE Classic is a high performance System-on-Chip (SoC) interconnect.
For zero-wait-state operation it requires that the SLAVE generates an asynchronous cycle termination signal. See chapter 3 for WISHBONE Classic bus cycles.
Also see: *WISHBONE Registered Feedback*

WISHBONE DATASHEET
A type of documentation required for WISHBONE compatible IP cores. This helps the end user understand the detailed operation of the core, and how to connect it to other cores. The WISHBONE DATASHEET can be included as part of an IP core technical reference manual, or as part of the IP core hardware description.

WISHBONE Registered Feedback
WISHBONE Registered Feedback is a high performace System-on-Chip (SoC) interconnect.
It requires that all interface signals are registered. To maintain performance, it introduces a number of novel bus-cycles. See chapter 4 for WISHBONE Registered Feedback bus cycles.
Also see: *WISHBONE Classic*

WISHBONE Signal
A signal that is defined as part of the WISHBONE specification. Non-WISHBONE signals can also be used on the IP core, but are not defined as part of this specification. For example, [ACK_O] is a WISHBONE signal, but [CLK100_I] is not.

WISHBONE Logo
A logo that, when affixed to a document, indicates that the associated SoC component is compatible with the WISHBONE standard.

Wrapper
A circuit element that converts a non-WISHBONE IP Core into a WISHBONE compatible IP Core. For example, consider a 16-byte synchronous memory primitive that is provided by an IC vendor. The memory primitive can be made into a WISHBONE compatible SLAVE by layering a circuit over the memory primitive, thereby creating a WISHBONE compatible SLAVE. A wrapper is analogous to a technique used to convert software written in 'C' to that written in 'C++'.

WORD
A unit of data that is 16-bits wide. Also see: *BYTE*, *DWORD* and *QWORD*.

1.8 References

IEEE 100: The Authoritative Dictionary of IEEE Standards Terms, Seventh Edition. IEEE Press 2000.

Feustel, Edward A. “On the Advantages of Tagged Architecture”. IEEE Transactions on Computers, Vol. C-22, No. 7, July 1973.

Chapter 2 – Interface Specification

This chapter describes the signaling method between the MASTER interface, SLAVE interface, and SYSCON module. This includes numerous options which may or may not be present on a particular interface. Furthermore, it describes a minimum level of required documentation that must be created for each IP core.

2.1 Required Documentation for IP Cores

WISHBONE compatible IP cores include documentation that describes the interface. This helps the end user understand the operation of the core, and how to connect it to other cores. This documentation takes the form of a WISHBONE DATASHEET. It can be included as part of the IP core technical reference manual, it can be embedded in source code or it can take other forms as well.

2.1.1 General Requirements for the WISHBONE DATASHEET

RULE 2.00
Each WISHBONE compatible IP core MUST include a WISHBONE DATASHEET as part of the IP core documentation.

RULE 2.15
The WISHBONE DATASHEET for MASTER and SLAVE interfaces MUST include the following information:

(1) The revision level of the WISHBONE specification to which is was designed.

(2) The type of interface: MASTER or SLAVE.

(3) The signal names that are defined for the WISHBONE SoC interface. If a signal name is different than that defined in this specification, then it MUST be cross-referenced to the corresponding signal name which is used in this specification.

(4) If a MASTER supports the optional [ERR_I] signal, then the WISHBONE DATASHEET MUST describe how it reacts in response to the signal. If a SLAVE supports the optional [ERR_O] signal, then the WISHBONE DATASHEET MUST describe the conditions under which the signal is generated.

(5) If a MASTER supports the optional [RTY_I] signal, then the WISHBONE DATASHEET MUST describe how it reacts in response to the signal. If a SLAVE supports the optional [RTY_O] signal, then the WISHBONE DATASHEET MUST describe the conditions under which the signal is generated.

PSL Remark

The PSL specification assumes the following constants have been defined: NUM_OF_MASTERS, NUM_OF_SLAVES. These constants (representing the number of slaves and number of masters, respectively) can be defined in a separate file so that the file will be referred by both the design files and the specification files. For example, define file named Wishbone_Env.v containing the following code:

PSL Code

```
`define NUM_OF_MASTERS 5
`define NUM_OF_SLAVES 7
```

PSL Remark

The PSL specification assumes an array of slaves and an array of masters have been defined. The PSL specification further assumes that each signal of a master/slave (other than CLK_I and RST_I) has been defined in an array of the corresponding size. The name of a master array for signal XXX is appended by an 'm' (i.e. mXXX) and the name for the slaves array for signal XXX is appended by a 's' (i.e. sXXX). For example:

PSL Code

```
input  [0:`NUM_OF_MASTERS-1] mACK_I;
input  [0:`NUM_OF_SLAVES-1]  sCYC_I, sSTB_I;
output [0:`NUM_OF_MASTERS-1] mSTB_O, mWE_O,  mRTY_O, mCYC_O;
output [0:`NUM_OF_SLAVES-1]  sWE_O, sERR_O, sRTY_O, sACK_O;
```

(6) All interfaces that support tag signals MUST describe the name, TAG TYPE and operation of the tag in the WISHBONE DATASHEET.

(7) The WISHBONE DATASHEET MUST indicate the port size. The port size MUST be indicated as: 8-bit, 16-bit, 32-bit or 64-bit.

(8) The WISHBONE DATASHEET MUST indicate the port granularity. The granularity MUST be indicated as: 8-bit, 16-bit, 32-bit or 64-bit.

(9) The WISHBONE DATASHEET MUST indicate the maximum operand size. The maximum operand size MUST be indicated as: 8-bit, 16-bit, 32-bit or 64-bit. If the maximum operand size is unknown, then the maximum operand size shall be the same as the granularity.

(10) The WISHBONE DATASHEET MUST indicate the data transfer ordering. The ordering MUST be indicated as BIG ENDIAN or LITTLE ENDIAN. When the port size equals the granularity, then the interface shall be specified as BIG/LITTLE ENDIAN. [When the port size equals the granularity, then BIG ENDIAN and LITTLE ENDIAN transfers are identical].

(11) The WISHBONE DATASHEET MUST indicate the sequence of data transfer through the port. If the sequence of data transfer is not known, then the datasheet MUST indicate it as UNDEFINED.

(12) The WISHBONE DATASHEET MUST indicate if there are any constraints on the [CLK_I] signal. These constraints include (but are not limited to) clock frequency, application specific timing constraints, the use of gated clocks or the use of variable clock generators.

2.1.2 Signal Naming

RULE 2.20
Signal names MUST adhere to the rules of the native tool in which the IP core is designed.

PERMISSION 2.00
Any signal name MAY be used to describe the WISHBONE signals.

OBSERVATION 2.00
Most hardware description languages (such as VHDL or Verilog®) have naming conventions. For example, the VHDL hardware description language defines the alphanumeric symbols which may be used. Furthermore, it states that UPPERCASE and LOWERCASE characters may be used in a signal name.

PSL Remark

In a parameterized PSL specification using as parameters master/slave signals, the signal name is appended with 'm' or 's' in its beginning, respectively. For example sACK_O or mSTB_O. Such a property is usually later quantified by all masters and all slaves. In the quantified property the actual parameters are elements of the above defined arrays.
The PSL specification assumes the information from the datasheet is gathered in arrays whose lengths are the same as the number of masters/slaves in the environment. In particular it assumes the following arrays have been defined and instantiated with the corresponding values from the datasheet.

PSL Code

```
// port granularity
integer mPORT_GRAN[0:`NUM_OF_MASTERS-1];
integer sPORT_GRAN[0:`NUM_OF_SLAVES-1];

// port size
integer mPORT_SIZE[0:`NUM_OF_MASTERS-1];
integer sPORT_SIZE[0:`NUM_OF_SLAVES-1];

// the ratio between mPORT_SIZE and mPORT_GRAN
integer mNUM_OF_PORTS[0:`NUM_OF_MASTERS-1];
integer sNUM_OF_PORTS [0:`NUM_OF_SLAVES-1];

// max operand size
integer mMAX_OP_SIZE[0:`NUM_OF_MASTERS-1];
integer sMAX_OP_SIZE[0:`NUM_OF_SLAVES-1];

// ordering type
// where 0 indicated LITTLE ENDIAN
// and 1 indicates BIG ENDIAN (or BIG/LITTLE ENDIAN)
integer mORDERING[0:`NUM_OF_MASTERS-1];
integer sORDERING[0:`NUM_OF_SLAVES-1];
```

RECOMENDATION 2.00
It is recommended that the interface uses the signal names defined in this document.

OBSERVATION 2.05
Core integration is simplified if the signal names match those given in this specification. However, in some cases (such as IP cores with multiple WISHBONE interconnects) they cannot be used. The use of non-standard signal names will not result in any serious integration problems since all hardware description tools allow signals to be renamed.

PERMISSION 2.05
Non-WISHBONE signals MAY be used with IP core interfaces.

OBSERVATION 2.15
Most IP cores will include non-WISHBONE signals. These are outside the scope of this specification, and no attempt is made to govern them. For example, a disk controller IP core could have a WISHBONE interface on one end and a disk interface on the other. In this case the specification does not dictate any technical requirements for the disk interface signals.

2.1.3 Logic Levels

RULE 2.30
All WISHBONE interface signals MUST use active high logic.

OBSERVATION 2.10
In general, the use of active low signals does not present a problem. However, RULE 2.30 is included because some tools (especially schematic entry tools) do not have a standard way of indicating an active low signal. For example, a reset signal could be named [#RST_I], [/RST_I] or [N_RST_I]. This was found to cause confusion among users and incompatibility between modules. This constraint should not create any undue difficulties, as the system integrator can invert any signals before use by the WISHBONE interface.

2.2 WISHBONE Signal Description

This section describes the signals used in the WISHBONE interconnect. Some of these signals are optional, and may or may not be present on a specific interface.

2.2.1 SYSCON Module Signals

CLK_O
The system clock output [CLK_O] is generated by the SYSCON module. It coordinates all activities for the internal logic within the WISHBONE interconnect. The INTERCON module connects the [CLK_O] output to the [CLK_I] input on MASTER and SLAVE interfaces.

RST_O
The reset output [RST_O] is generated by the SYSCON module. It forces all WISHBONE interfaces to restart. All internal self-starting state machines are forced into an initial state. The INTERCON connects the [RST_O] output to the [RST_I] input on MASTER and SLAVE interfaces.

2.2.2 Signals Common to MASTER and SLAVE Interfaces

CLK_I
The clock input [CLK_I] coordinates all activities for the internal logic within the WISHBONE interconnect. All WISHBONE output signals are registered at the rising edge of [CLK_I]. All WISHBONE input signals are stable before the rising edge of [CLK_I].

DAT_I()
The data input array [DAT_I()] is used to pass binary data. The array boundaries are determined by the port size, with a maximum port size of 64-bits (e.g. [DAT_I(63..0)]). Also see the [DAT_O()] and [SEL_O()] signal descriptions.

DAT_O()
The data output array [DAT_O()] is used to pass binary data. The array boundaries are determined by the port size, with a maximum port size of 64-bits (e.g. [DAT_I(63..0)]). Also see the [DAT_I()] and [SEL_O()] signal descriptions.

RST_I
The reset input [RST_I] forces the WISHBONE interface to restart. Furthermore, all internal self-starting state machines will be forced into an initial state. This signal only resets the WISHBONE interface. It is not required to reset other parts of an IP core (although it may be used that way).

TGD_I()
Data tag type [TGD_I()] is used on MASTER and SLAVE interfaces. It contains information that is associated with the data input array [DAT_I()], and is qualified by signal [STB_I]. For example, parity protection, error correction and time stamp information can be attached to the data bus. These tag bits simplify the task of defining new signals because their timing (in rela-

tion to every bus cycle) is pre-defined by this specification. The name and operation of a data tag must be defined in the WISHBONE DATASHEET.

TGD_O()
Data tag type [TGD_O()] is used on MASTER and SLAVE interfaces. It contains information that is associated with the data output array [DAT_O()], and is qualified by signal [STB_O]. For example, parity protection, error correction and time stamp information can be attached to the data bus. These tag bits simplify the task of defining new signals because their timing (in relation to every bus cycle) is pre-defined by this specification. The name and operation of a data tag must be defined in the WISHBONE DATASHEET.

2.2.3 MASTER Signals

ACK_I
The acknowledge input [ACK_I], when asserted, indicates the normal termination of a bus cycle. Also see the [ERR_I] and [RTY_I] signal descriptions.

ADR_O()
The address output array [ADR_O()] is used to pass a binary address. The higher array boundary is specific to the address width of the core, and the lower array boundary is determined by the data port size and granularity. For example the array size on a 32-bit data port with BYTE granularity is [ADR_O(n..2)]. In some cases (such as FIFO interfaces) the array may not be present on the interface.

CYC_O
The cycle output [CYC_O], when asserted, indicates that a valid bus cycle is in progress. The signal is asserted for the duration of all bus cycles. For example, during a BLOCK transfer cycle there can be multiple data transfers. The [CYC_O] signal is asserted during the first data transfer, and remains asserted until the last data transfer. The [CYC_O] signal is useful for interfaces with multi-port interfaces (such as dual port memories). In these cases, the [CYC_O] signal requests use of a common bus from an arbiter.

ERR_I
The error input [ERR_I] indicates an abnormal cycle termination. The source of the error, and the response generated by the MASTER is defined by the IP core supplier. Also see the [ACK_I] and [RTY_I] signal descriptions.

LOCK_O
The lock output [LOCK_O] when asserted, indicates that the current bus cycle is uninterruptible. Lock is asserted to request complete ownership of the bus. Once the transfer has started, the IN-

TERCON does not grant the bus to any other MASTER, until the current MASTER negates [LOCK_O] or [CYC_O].

RTY_I
The retry input [RTY_I] indicates that the interface is not ready to accept or send data, and that the cycle should be retried. When and how the cycle is retried is defined by the IP core supplier. Also see the [ERR_I] and [RTY_I] signal descriptions.

SEL_O()
The select output array [SEL_O()] indicates where valid data is expected on the [DAT_I()] signal array during READ cycles, and where it is placed on the [DAT_O()] signal array during WRITE cycles. The array boundaries are determined by the granularity of a port. For example, if 8-bit granularity is used on a 64-bit port, then there would be an array of eight select signals with boundaries of [SEL_O(7..0)]. Each individual select signal correlates to one of eight active bytes on the 64-bit data port. For more information about [SEL_O()], please refer to the data organization section in Chapter 3 of this specification. Also see the [DAT_I()], [DAT_O()] and [STB_O] signal descriptions.

STB_O
The strobe output [STB_O] indicates a valid data transfer cycle. It is used to qualify various other signals on the interface such as [SEL_O()]. The SLAVE asserts either the [ACK_I], [ERR_I] or [RTY_I] signals in response to every assertion of the [STB_O] signal.

TGA_O()
Address tag type [TGA_O()] contains information associated with address lines [ADR_O()], and is qualified by signal [STB_O]. For example, address size (24-bit, 32-bit etc.) and memory management (protected vs. unprotected) information can be attached to an address. These tag bits simplify the task of defining new signals because their timing (in relation to every bus cycle) is defined by this specification. The name and operation of an address tag must be defined in the WISHBONE DATASHEET.

TGC_O()
Cycle tag type [TGC_O()] contains information associated with bus cycles, and is qualified by signal [CYC_O]. For example, data transfer, interrupt acknowledge and cache control cycles can be uniquely identified with the cycle tag. They can also be used to discriminate between WISHBONE SINGLE, BLOCK and RMW cycles. These tag bits simplify the task of defining new signals because their timing (in relation to every bus cycle) is defined by this specification. The name and operation of a cycle tag must be defined in the WISHBONE DATASHEET.

WE_O
The write enable output [WE_O] indicates whether the current local bus cycle is a READ or WRITE cycle. The signal is negated during READ cycles, and is asserted during WRITE cycles.

2.2.4 SLAVE Signals

ACK_O
The acknowledge output [ACK_O], when asserted, indicates the termination of a normal bus cycle. Also see the [ERR_O] and [RTY_O] signal descriptions.

ADR_I()
The address input array [ADR_I()] is used to pass a binary address. The higher array boundary is specific to the address width of the core, and the lower array boundary is determined by the data port size. For example the array size on a 32-bit data port with BYTE granularity is [ADR_O(n..2)]. In some cases (such as FIFO interfaces) the array may not be present on the interface.

CYC_I
The cycle input [CYC_I], when asserted, indicates that a valid bus cycle is in progress. The signal is asserted for the duration of all bus cycles. For example, during a BLOCK transfer cycle there can be multiple data transfers. The [CYC_I] signal is asserted during the first data transfer, and remains asserted until the last data transfer.

ERR_O
The error output [ERR_O] indicates an abnormal cycle termination. The source of the error, and the response generated by the MASTER is defined by the IP core supplier. Also see the [ACK_O] and [RTY_O] signal descriptions.

LOCK_I
The lock input [LOCK_I], when asserted, indicates that the current bus cycle is uninterruptible. A SLAVE that receives the LOCK [LOCK_I] signal is accessed by a single MASTER only, until either [LOCK_I] or [CYC_I] is negated.

RTY_O
The retry output [RTY_O] indicates that the indicates that the interface is not ready to accept or send data, and that the cycle should be retried. When and how the cycle is retried is defined by the IP core supplier. Also see the [ERR_O] and [RTY_O] signal descriptions.

SEL_I()

The select input array [SEL_I()] indicates where valid data is placed on the [DAT_I()] signal array during WRITE cycles, and where it should be present on the [DAT_O()] signal array during READ cycles. The array boundaries are determined by the granularity of a port. For example, if 8-bit granularity is used on a 64-bit port, then there would be an array of eight select signals with boundaries of [SEL_I(7..0)]. Each individual select signal correlates to one of eight active bytes on the 64-bit data port. For more information about [SEL_I()], please refer to the data organization section in Chapter 3 of this specification. Also see the [DAT_I(63..0)], [DAT_O(63..0)] and [STB_I] signal descriptions.

STB_I
The strobe input [STB_I], when asserted, indicates that the SLAVE is selected. A SLAVE shall respond to other WISHBONE signals only when this [STB_I] is asserted (except for the [RST_I] signal which should always be responded to). The SLAVE asserts either the [ACK_O], [ERR_O] or [RTY_O] signals in response to every assertion of the [STB_I] signal.

TGA_I
Address tag type [TGA_I()] contains information associated with address lines [ADR_I()], and is qualified by signal [STB_I]. For example, address size (24-bit, 32-bit etc.) and memory management (protected vs. unprotected) information can be attached to an address. These tag bits simplify the task of defining new signals because their timing (in relation to every bus cycle) is pre-defined by this specification. The name and operation of an address tag must be defined in the WISHBONE DATASHEET.

TGC_I()
Cycle tag type [TGC_I()] contains information associated with bus cycles, and is qualified by signal [CYC_I]. For example, data transfer, interrupt acknowledge and cache control cycles can be uniquely identified with the cycle tag. They can also be used to discriminate between WISHBONE SINGLE, BLOCK and RMW cycles. These tag bits simplify the task of defining new signals because their timing (in relation to every bus cycle) is pre-defined by this specification. The name and operation of a cycle tag must be defined in the WISHBONE DATASHEET.

WE_I
The write enable input [WE_I] indicates whether the current local bus cycle is a READ or WRITE cycle. The signal is negated during READ cycles, and is asserted during WRITE cycles.

Chapter 3 – WISHBONE Classic Bus Cycles

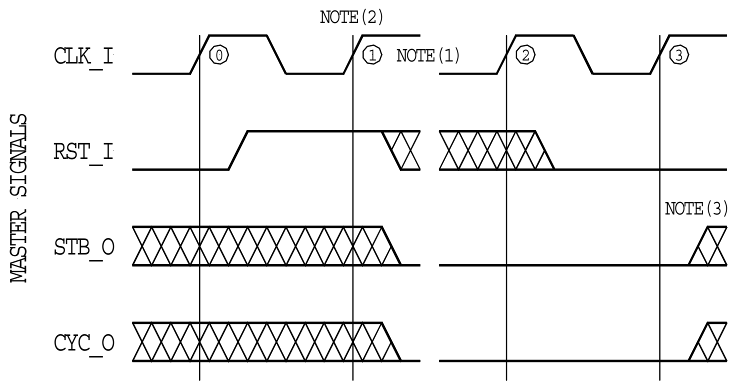
WISHBONE Classic bus cycles are described in terms of their general operation, reset operation, handshaking protocol and the data organization during transfers. Additional requirements for bus cycles (especially those relating to the common clock) can be found in the timing specifications in Chapter 5.

3.1 General Operation

MASTER and SLAVE interfaces are interconnected with a set of signals that permit them to exchange data. For descriptive purposes these signals are cumulatively known as a *bus*, and are contained within a functional module called the INTERCON. Address, data and other information is impressed upon this bus in the form of *bus cycles*.

3.1.1 Reset Operation

All hardware interfaces are initialized to a pre-defined state. This is accomplished with the reset signal [RST_O] that can be asserted at any time. It is also used for test simulation purposes by initializing all self-starting state machines and counters which may be used in the design. The reset signal [RST_O] is driven by the SYSCON module. It is connected to the [RST_I] signal on all MASTER and SLAVE interfaces. Figure 3-1 shows the reset cycle.



NOTES:
(1) Reset cycles can be extended for any length of time.
(2) Self-starting state machines & counters reset themselves at the rising [CLK_I] edge following the assertion of [RST_I]. On MASTERs, [STB_O] and [CYC_O] are negated at the same time.
(3) On MASTERs, [STB_O] and [CYC_O] may be asserted at the rising [CLK_I] edge following the negation of [RST_I].

Figure 3-1. Reset cycle.

This page is Intentionally Blank

RULE 3.00
All WISHBONE interfaces MUST initialize themselves at the rising [CLK_I] edge following the assertion of [RST_I]. They MUST stay in the initialized state until the rising [CLK_I] edge that follows the negation of [RST_I].

RULE 3.05
[RST_I] MUST be asserted for at least one complete clock cycle on all WISHBONE interfaces.

PERMISSION 3.00
[RST_I] MAY be asserted for more than one clock cycle, and MAY be asserted indefinitely.

RULE 3.10
All WISHBONE interfaces MUST be capable of reacting to [RST_I] at any time.

RULE 3.15
All self-starting state machines and counters in WISHBONE interfaces MUST initialize themselves at the rising [CLK_I] edge following the assertion of [RST_I]. They MUST stay in the initialized state until the rising [CLK_I] edge that follows the negation of [RST_I].

OBSERVATION 3.00
In general, self-starting state machines do not need to be initialized. However, this may cause problems because some simulators may not be sophisticated enough to find an initial starting point for the state machine. Furthermore, self-starting state machines can go through an indeterminate number of initialization cycles before finding their starting state, thereby making it difficult to predict their behavior at start-up time. The initialization rule prevents both problems by forcing all state machines to a pre-defined state in response to the assertion of [RST_I].

RULE 3.20
The following MASTER signals MUST be negated at the rising [CLK_I] edge following the assertion of [RST_I], and MUST stay in the negated state until the rising [CLK_I] edge that follows the negation of [RST_I]: [STB_O], [CYC_O]. The state of all other MASTER signals are undefined in response to a reset cycle.

OBSERVATION 3.05
On MASTER interfaces [STB_O] and [CYC_O] may be asserted beginning at the rising [CLK_I] edge following the negation of [RST_I].

PSL Code

```
// RULE 3.00

// [property regarding a given IP]

property initialize_interface(boolean init_state) =
always ({ RST_I } |=>
        { init_state[+] && {RST_I[*];!RST_I}}!)@rose(CLK_I);

// The top level assertion should quantify over all interfaces.
// For each interface a Boolean expression [init_state] defining the
// initial state of the system should be defined.
// For example for all masters the initial condition is !STB_O&&!CYC_O

// [assertion regarding a group of IPs]

assert
forall i in {0:`NUM_OF_MASTERS-1}:
initialize_interface(!mSTB_O[i] && !mCYC_O[i]);

// RULE 3.05

// [global property]

property reset_signal =
  always { rose(RST_I) } |=> {(RST_I && !CLK_I)[*]; RST_I && CLK_I};

// [global assertion]

assert reset_signal;

// RULE 3.15

// [property regarding a given IP]

property initialize_fsms(boolean init_state) =
always { RST_I } |=> {init_state[+] && {RST_I[*];!RST_I}}@rose(CLK_I);

// The top level assertion should quantify over all finite state
// machined and all counters.
// For each of them a Boolean expression [init_state] defining its
// initial state should be defined.
// For example for all counters we need to assert:
```

This page is Intentionally Blank

```
// [assertion regarding a group of IPs]

assert
forall i in {0:`NUM_OF_COUNTERS-1}:
initialize_interface(CNTRS[i].COUNTER.init_state);

// RULE 3.20

// This rule is already covered by 3.00
// However, it can be coded as follows as well:

// [property regarding a given master]

property initialize_master(boolean mSTB_O, boolean mCYC_O) =
always ({RST_I} |=> { {(!mSTB_O && !mCYC_O)[+]} &&
                      {RST_I[*];!RST_I}            }
        )@rose(CLK_I);

// [assertion regarding all masters]

assert
forall i in {0:`NUM_OF_MASTERS-1}:
initialize_master(mSTB_O[i], mCYC_O[i]);
```

OBSERVATION 3.10
SLAVE interfaces automatically negate [ACK_O], [ERR_O] and [RTY_O] when their [STB_I] is negated.

RECOMENDATION 3.00
Design SYSCON modules so that they assert [RST_O] during a power-up condition. [RST_O] should remain asserted until all voltage levels and clock frequencies in the system are stabilized. When negating [RST_O], do so in a synchronous manner that conforms to this specification.

OBSERVATION 3.15
If a gated clock generator is used, and if the clock is stopped, then the WISHBONE interface is not capable of responding to its [RST_I] signal.

SUGGESTION 3.00
Some circuits require an *asynchronous* reset capability. If an IP core or other SoC component requires an asynchronous reset, then define it as a non-WISHBONE signal. This prevents confusion with the WISHBONE reset [RST_I] signal that uses a purely synchronous protocol, and needs to be applied to the WISHBONE interface only.

OBSERVATION 3.20
All WISHBONE *interfaces* respond to the reset signal. However, the IP Core connected to a WISHBONE interface does not necessarily need to respond to the reset signal.

3.1.2 Transfer Cycle initiation

MASTER interfaces initiate a transfer cycle by asserting [CYC_O]. When [CYC_O] is negated, all other MASTER signals are invalid.
SLAVE interfaces respond to other SLAVE signals only when [CYC_I] is asserted.
SYSCON signals and responses to SYSCON signals are not affected.

RULE 3.25
MASTER interfaces MUST assert [CYC_O] for the duration of SINGLE READ / WRITE, BLOCK and RMW cycles. [CYC_O] MUST be asserted no later than the rising [CLK_I] edge that qualifies the assertion of [STB_O]. [CYC_O] MUST be negated no earlier than the rising [CLK_I] edge that qualifies the negation of [STB_O].

PERMISSION 3.05
MASTER interfaces MAY assert [CYC_O] indefinitely.

PSL Code

```
// OBSERVATION 3.10

// [property regarding a given slave]

property slave_initialization(boolean sSTB_I,sACK_O,sERR_O,sRTY_O) =
always ( !sSTB_I  -> (!sACK_O && !sERR_O && !sRTY_O));

// [assumption regarding all slaves]

assume
forall j in {0:`NUM_OF_SLAVES-1}:
slave_initialization(sSTB_I[j],sACK_O[j],sERR_O[j],sRTY_O[j]);
```

```
// RULE 3.25

// [property regarding a given master]

property CYC_O_signal(boolean mm,mCYC_O) =
always {rose(mSTB_O) && CLK_I} |=>
       {mCYC_O[+] && {fell(mSTB_O)[->] :CLK_I}};
```

This page is Intentionally Blank

```
// [assertion regarding all masters]

assert
forall i in {0:`NUM_OF_MASTERS-1}
CYC_O_signal(mSTB_O[i], mCYC_O[i]);
```

RECOMMENDATION 3.05
Arbitration logic often uses [CYC_I] to select between MASTER interfaces. Keeping [CYC_O] asserted may lead to arbitration problems. It is therefore recommended that [CYC_O] is not indefinitely asserted.

RULE 3.30
SLAVE interfaces MAY NOT respond to any SLAVE signals when [CYC_I] is negated. However, SLAVE interfaces MUST always respond to SYSCON signals.

3.1.3 Handshaking Protocol

All bus cycles use a handshaking protocol between the MASTER and SLAVE interfaces. As shown in Figure 3-2, the MASTER asserts [STB_O] when it is ready to transfer data. [STB_O] remains asserted until the SLAVE asserts one of the cycle terminating signals [ACK_I], [ERR_I] or [RTY_I]. At every rising edge of [CLK_I] the terminating signal is sampled. If it is asserted, then [STB_O] is negated. This gives both MASTER and SLAVE interfaces the possibility to control the rate at which data is transferred.

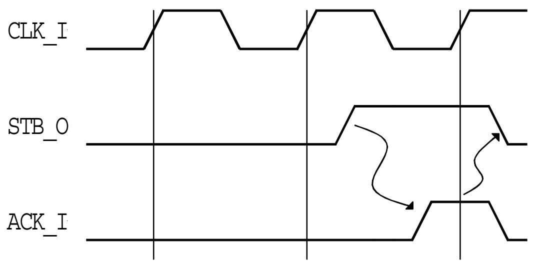


Figure 3-2. Local bus handshaking protocol.

PERMISSION 3.10
If the SLAVE guarantees it can keep pace with all MASTER interfaces and if the [ERR_I] and [RTY_I] signals are not used, then the SLAVE's [ACK_O] signal MAY be tied to the logical AND of the SLAVE's [STB_I] and [CYC_I] inputs. The interface will function normally under these circumstances.

OBSERVATION 3.25
SLAVE interfaces assert a cycle termination signal in response to [STB_I]. However, [STB_I] is only valid when [CYC_I] is valid.

RULE 3.35
The cycle termination signals [ACK_O], [ERR_O], and [RTY_O] must be generated in response to the logical AND of [CYC_I] and [STB_I].

PSL Code

```
// RULE 3.30

// The following assertion demands this rule for the mandatory signals
// Similar assertions are needed for the optional signals

// [property regarding a given slave]
Property slave_no_response(boolean sACK_O,sCYC_I,sSTB_I)
always (!sCYC_I -> ( sACK_O == prev(sACK_O) &&
                     sCYC_I == prev(sCYC_I) &&
                     sSTB_I == prev(sSTB_I)     );

// [assertion regarding all slaves]
assert
forall i in {0:`NUM_OF_SLAVES-1}:
initialize_interface(sACK_O[i],sCYC_I[i], sSTB_I[i]);

// From initial description of Section 3.1.3 (Handshaking Protocol)

// [property regarding a given master and a given slave]

property slave_response_to_master
(boolean mSTB_O,sACK_I,sERR_I,sRTY_I,sSTB_I) =
always {rose(mSTB_O) } |->
        { {!sSTB_I} |
          { mSTB_O[*];
            mSTB_O && (rose(sACK_I) || rose(sERR_I) || rose (sRTY_I));
            !mSTB_O}};

// [assertion regarding all masters and all slave]

assert
forall i in {0:`NUM_OF_MASTERS-1}:
forall j in {0:`NUM_OF_SLAVES-1}:
slave_response_to_master
(mSTB_O[i], sACK_I[j], sERR_I[j], sRTY_I[j], sSTB_I[j]);


// RULE 3.35

// takes into account observation 3.40
// See Rule 3.50 as well
```

This page is Intentionally Blank

```
// [property regarding a given slave]

property slave_response(boolean sSTB_I,sCYC_I,sACK_O,sERR_O,sRTY_O) =
always {CLK_I && sCYC_I && rose(sSTB_I)} |->
       {rose(sACK_O) || rose(sERR_O) || rose (sRTY_O)};

// [assertion regarding all slaves]

assert
forall j in {0:`NUM_OF_SLAVES-1}:
slave_response(sSTB_I[j], sCYC_I[j] ,sACK_O[j], sERR_O[j], sRTY_O[j]);
```

PERMISSION 3.15
Other signals, besides [CYC_I] and [STB_I], MAY be included in the generation of the cycle termination signals.

OBSERVATION 3.30
Internal SLAVE signals also determine what cycle termination signal is asserted and when it is asserted.

Most of the examples in this specification describe the use of [ACK_I] to terminate a local bus cycle. However, the SLAVE can optionally terminate the cycle with an error [ERR_O], or request that the cycle be retried [RTY_O].

All MASTER interfaces include the [ACK_I] terminator signal. Asserting this signal during a bus cycle causes it to terminate normally.

Asserting the [ERR_I] signal during a bus cycle will terminate the cycle. It also serves to notify the MASTER that an error occurred during the cycle. This signal is generally used if an error was detected by SLAVE logic circuitry. For example, if the SLAVE is a parity-protected memory, then the [ERR_I] signal can be asserted if a parity fault is detected. This specification does not dictate what the MASTER will do in response to [ERR_I].

Asserting the optional [RTY_I] signal during a bus cycle will terminate the cycle. It also serves to notify the MASTER that the current cycle should be aborted, and retried at a later time. This signal is generally used for shared memory and bus bridges. In these cases SLAVE circuitry asserts [RTY_I] if the local resource is busy. This specification does not dictate when or how the MASTER will respond to [RTY_I].

RULE 3.40
As a minimum, the MASTER interface MUST include the following signals: [ACK_I], [CLK_I], [CYC_O], [RST_I], and [STB_O]. As a minimum, the SLAVE interface MUST include the following signals: [ACK_O], [CLK_I], [CYC_I], [STB_I], and [RST_I]. All other signals are optional.

PERMISSION 3.20
MASTER and SLAVE interfaces MAY be designed to support the [ERR_I] and [ERR_O] signals. In these cases, the SLAVE asserts [ERR_O] to indicate that an error has occurred during the bus cycle. This specification does not dictate what the MASTER does in response to [ERR_I].

This page is Intentionally Blank

PERMISSION 3.25
MASTER and SLAVE interfaces MAY be designed to support the [RTY_I] and [RTY_O] signals. In these cases, the SLAVE asserts [RTY_O] to indicate that the interface is busy, and that the bus cycle should be retried at a later time. This specification does not dictate what the MASTER will do in response to [RTY_I].

RULE 3.45
If a SLAVE supports the [ERR_O] or [RTY_O] signals, then the SLAVE MUST NOT assert more than one of the following signals at any time: [ACK_O], [ERR_O] or [RTY_O].

OBSERVATION 3.35
If the SLAVE supports the [ERR_O] or [RTY_O] signals, but the MASTER does not support these signals, deadlock may occur.

RECOMMENDATION 3.10
Design INTERCON modules to prevent deadlock conditions. One solution to this problem is to include a watchdog timer function that monitors the MASTER's [STB_O] signal, and asserts [ERR_I] or [RTY_I] if the cycle exceeds some pre-defined time limit. INTERCON modules can also be designed to disconnect interfaces from the WISHBONE bus if they constantly generate bus errors and/or watchdog time-outs.

RECOMMENDATION 3.15
Design WISHBONE MASTER interfaces so that there are no intermediate logic gates between a registered flip-flop and the signal outputs on [STB_O] and [CYC_O]. Delay timing for [STB_O] and [CYC_O] are very often the most critical paths in the system. This prevents sloppy design practices from slowing down the interconnect because of added delays on these two signals.

RULE 3.50
SLAVE interfaces MUST be designed so that the [ACK_O], [ERR_O], and [RTY_O] signals are asserted and negated in response to the assertion and negation of [STB_I].

PERMISSION 3.30
The assertion of [ACK_O], [ERR_O], and [RTY_O] MAY be asynchronous to the [CLK_I] signal (i.e. there is a combinatorial logic path between [STB_I] and [ACK_O]).

PSL Code

```
// RULE 3.45

// [property regarding a given slave]

Property response_signals(boolean sERR_O,sRTY_O,sACK_O) =
never ((sERR_O && sRTY_O) || (sERR_O && sACK_O) || (sACK_O && sRTY_O));

// [assertion regarding all slaves]

assert
forall j in {0:`NUM_OF_SLAVES-1}:
response_signals(sERR_O[j], sRTY_O[j], sACK_O[j]);
```

```
// RULE 3.50
// complements rule 3.35
// takes into account observation 3.40

// [property regarding a given slave]

property slave_negated_response
(boolean sSTB_I,sCYC_I,sACK_O,sERR_O,sRTY_O) =
always {CLK_I && sCYC_I && fell(sSTB_I) } |->
       {fell(sACK_O) || fell(sERR_O) || fell (sRTY_O) };
```

This page is Intentionally Blank

```
// [assertion regarding all slaves]

assert
forall j in {0:`NUM_OF_SLAVES-1}:
slave_negated_response
(sSTB_I[j], sCYC_I[j], sACK_O[j], sERR_O[j], sRTY_O[j]);
```

OBSERVATION 3.40
The asynchronous assertion of [ACK_O], [ERR_O], and [RTY_O] assures that the interface can accomplish one data transfer per clock cycle. Furthermore, it simplifies the design of arbiters in multi-MASTER applications.

OBSERVATION 3.45
The asynchronous assertion of [ACK_O], [ERR_O], and [RTY_O] could proof impossible to implement. For example slave wait states are easiest implemented using a registered [ACK_O] signal.

OBSERVATION 3.50
In large high speed designs the asynchronous assertion of [ACK_O], [ERR_O], and [RTY_O] could lead to unacceptable delay times, caused by the loopback delay from the MASTER to the SLAVE and back to the MASTER. Using registered [ACK_O], [ERR_O], and [RTY_O] signals significantly reduces this loopback delay, at the cost of one additional wait state per transfer. See WISHBONE Registered Feedback Bus Cycles for methods of eliminating the wait state.

PERMISSION 3.35
Under certain circumstances SLAVE interfaces MAY be designed to hold [ACK_O] in the asserted state. This situation occurs on point-to-point interfaces where there is a single SLAVE on the interface, and that SLAVE always operates without wait states.

RULE 3.55
MASTER interfaces MUST be designed to operate normally when the SLAVE interface holds [ACK_I] in the asserted state.

3.1.3 Use of [STB_O]

RULE 3.60
MASTER interfaces MUST qualify the following signals with [STB_O]: [ADR_O], [DAT_O()], [SEL_O()], [WE_O], and [TAGN_O].

PERMISSION 3.40
If a MASTER doesn't generate wait states, then [STB_O] and [CYC_O] MAY be assigned the same signal.

OBSERVATION 3.55
[CYC_O] needs to be asserted during the entire transfer cycle. A MASTER that doesn't generate wait states doesn't negate [STB_O] during a transfer cycle, i.e. it is asserted the entire transfer

cycle. Therefore it is allowed to use the same signal for [CYC_O] and [STB_O]. Both signals must be present on the interface though.

3.1.4 Use of [ACK_O], [ERR_O] and [RTY_O]

RULE 3.65
SLAVE interfaces MUST qualify the following signals with [ACK_O], [ERR_O] or [RTY_O]: [DAT_O()].

3.1.5 Use of TAG TYPES

The WISHBONE interface can be modified with user defined signals. This is done with a technique known as tagging. Tags are a well known concept in the microcomputer bus industry. They allow user defined information to be associated with an address, a data word or a bus cycle.

All tag signals must conform to set of guidelines known as TAG TYPEs. Table 3-1 lists all of the defined TAG TYPEs along with their associated data set and signal waveform. When a tag is added to an interface it is assigned a TAG TYPE from the table. This explicitly defines how the tag operates. This information must also be included in the WISHBONE DATASHEET.

Table 3-1. TAG TYPEs.

	MASTER		SLAVE	
Description	TAG TYPE	Associated With	TAG TYPE	Associated With
Address tag	TGA_O()	ADR_O()	TGA_I()	ADR_I()
Data tag, input	TGD_I()	DAT_I()	TGD_I()	DAT_I()
Data tag, output	TGD_O()	DAT_O()	TGD_O()	DAT_O()
Cycle tag	TGC_O()	Bus Cycle	TGC_I()	Bus Cycle

For example, consider a MASTER interface where a parity protection bit named [PAR_O] is generated from an output data word on [DAT_O(15..0)]. It's an 'even' parity bit, meaning that it's asserted whenever there are an even number of '1's in the data word. If this signal were added to the interface, then the following information (in the WISHBONE DATASHEET) would be sufficient to completely define the timing of [PAR_O]:

SIGNAL NAME:	PAR_O
DESCRIPTION:	Even parity bit
MASTER TAG TYPE:	TGD_O()

RULE 3.70
All user defined tags MUST be assigned a TAG TYPE. Furthermore, they MUST adhere to the timing specifications given in this document for the indicated TAG TYPE.

PERMISSION 3.45
While all TAG TYPES are specified as arrays (with parenthesis '()'), the actual tag MAY be a non-arrayed signal.

RECOMMENDATION 3.15
If a MASTER interface supports more than one defined bus cycle over a common set of signal lines, then include a cycle tag to identify each type of bus cycle. This allows INTERCON and SLAVE interface circuits to discriminate between these bus cycles (if needed). Define the signals as TAG TYPE: [TGC_O()], using signal names of [SGL_O], [BLK_O] and [RMW_O] when identifying SINGLE, BLOCK and RMW cycles respectively.

This page is Intentionally Blank

3.2 SINGLE READ / WRITE Cycles

The SINGLE READ / WRITE cycles perform one data transfer at a time. These are the basic cycles used to perform data transfers on the WISHBONE interconnect.
Note that the [CYC_O] signal isn't shown here to keep the timing diagrams as simple as possible. It is assumed that [CYC_O] is continuously asserted.

RULE 3.75
All MASTER and SLAVE interfaces that support SINGLE READ or SINGLE WRITE cycles MUST conform to the timing requirements given in sections 3.2.1 and 3.2.2.

PERMISSION 3.50
MASTER and SLAVE interfaces MAY be designed so that they do not support the SINGLE READ or SINGLE WRITE cycles.

3.2.1 SINGLE READ Cycle

Figure 3-3 shows a SINGLE READ cycle. The bus protocol works as follows:

CLOCK EDGE 0:	MASTER presents a valid address on [ADR_O()] and [TGA_O()]. MASTER negates [WE_O] to indicate a READ cycle. MASTER presents bank select [SEL_O()] to indicate where it expects data. MASTER asserts [CYC_O] and [TGC_O()] to indicate the start of the cycle. MASTER asserts [STB_O] to indicate the start of the phase.
SETUP, EDGE 1:	SLAVE decodes inputs, and responding SLAVE asserts [ACK_I]. SLAVE presents valid data on [DAT_I()] and [TGD_I()]. SLAVE asserts [ACK_I] in response to [STB_O] to indicate valid data. MASTER monitors [ACK_I], and prepares to latch data on [DAT_I()] and [TGD_I()]. Note: SLAVE may insert wait states (-WSS-) before asserting [ACK_I], thereby allowing it to throttle the cycle speed. Any number of wait states may be added.
CLOCK EDGE 1:	MASTER latches data on [DAT_I()] and [TGD_I()]. MASTER negates [STB_O] and [CYC_O] to indicate the end of the cycle. SLAVE negates [ACK_I] in response to negated [STB_O].

PSL Remarks

The properties described in this section assume classic bus cycles. Registered feedback bus cycles is described in Chapter 4. The PSL specification there refines the specification here by considering the signals CTI coding information about registered feedback cycles. The PSL specification here ignores the CTI signals following the English specification.

The properties in this section describe correct protocols regarding classic bus cycles. To complete the specification, high level properties assuring the correct data has been transferred are needed. These high-level properties are formulates near Section 3.5 since they take data organization considerations into account.

The properties described in this section assume a fast clock has been defined. The fast clock is high in every edge of the clock CLK_I and every setup of an edge of CLK_I for example a fast clock can be defined as follows:

PSL Code

```
// defining a Boolean expression describing that holds
// in each setup and edge of CLK_I
property fast_clock = next(!stable(CLK_I));
```

PSL Code

```
// RULE 3.75

// Properties say nothing about tags since the specification
// requires nothing from them
// Section 3.2.1 single read cycle
// Figure 3-3
// Section 3.2.2 single write cycle
// Figure 3-4

// [property regarding a given master and a given slave]

property
read_write_cycle_slave_response1
(boolean mCYC_O,mSTB_O,sACK_O,sRTY_O,sERR_O,sSTB_I) =
always ({rose(mCYC_O && mSTB_O)}
  |=>
  {{!sSTB_I} |
   {(mCYC_O && mSTB_O)[*] :
    (rose(sACK_O) || rose (sRTY_O) || rose(sERR_O) }})@fast_clock;
```

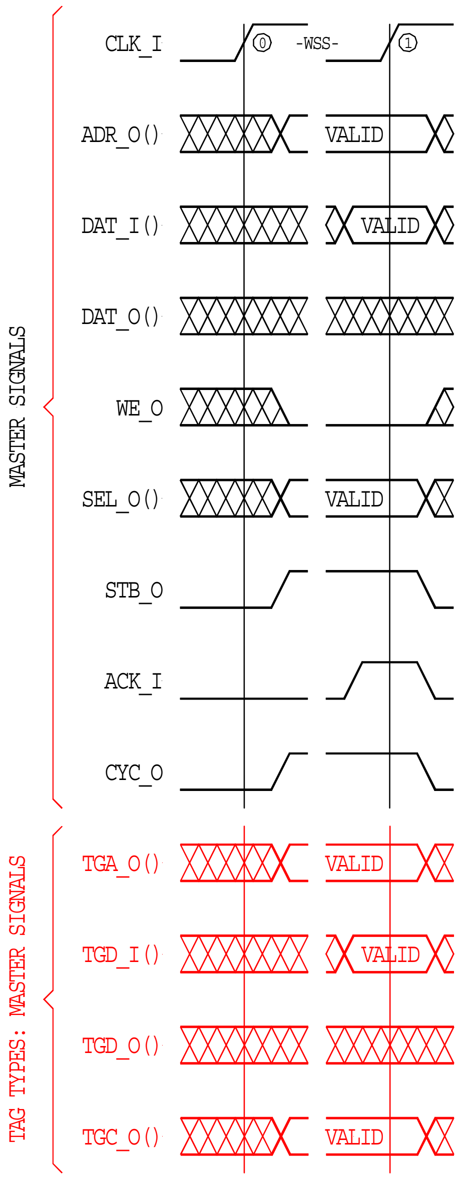


Figure 3-3. SINGLE READ cycle.

```
// [property regarding a given master and a given slave]

property
read_write_cycle_slave_response2
(boolean mCYC_O,mSTB_O,sACK_O,sRTY_O,sERR_O,sSTB_I) =
always ({sACK_O | sERR_O | sRTY_O} |->
                    {(mCYC_O && mSTB_O)})@fast_clock;

// [property regarding a given master and a given slave]

property
read_write_cycle_slave_response3
(boolean mCYC_O,mSTB_O,sACK_O,sRTY_O,sERR_O,sSTB_I) =
always ({ fell(sACK_O)} |->
        {{!mSTB_O && !m_CYC_O} |
         {sACK_O[->] && (mSTB_O && mCYC_O)[*]}
        }
       )@fast_clock;

// [property regarding a given master and a given slave]

property
read_write_cycle_slave_response
(boolean mCYC_O,mSTB_O,sACK_O,sRTY_O,sERR_O,sSTB_I) =
read_write_cycle_slave_response1
(boolean mCYC_O,mSTB_O,sACK_O,sRTY_O,sERR_O,sSTB_I) &&
read_write_cycle_slave_response2
(boolean mCYC_O,mSTB_O,sACK_O,sRTY_O,sERR_O,sSTB_I) &&
read_write_cycle_slave_response3
(boolean mCYC_O,mSTB_O,sACK_O,sRTY_O,sERR_O,sSTB_I);

// [assertion regarding all masters and all slaves]


forall i in {0:`NUM_OF_MASTERS-1}:
forall j in {0:`NUM_OF_SLAVES-1}:
read_write_cycle_slave_response
(mWE_O[i],mCYC_O[i],mSTB_O[i],sACK_O[j],sRTY_O[j],sERR_O[j],sSTB_I[j]);
```

3.2.2 SINGLE WRITE Cycle

Figure 3-4 shows a SINGLE WRITE cycle. The bus protocol works as follows:

CLOCK EDGE 0: MASTER presents a valid address on [ADR_O()] and [TGA_O()].
MASTER presents valid data on [DAT_O()] and [TGD_O()].
MASTER asserts [WE_O] to indicate a WRITE cycle.
MASTER presents bank select [SEL_O()] to indicate where it sends data.
MASTER asserts [CYC_O] and [TGC_O()] to indicate the start of the cycle.
MASTER asserts [STB_O] to indicate the start of the phase.

SETUP, EDGE 1: SLAVE decodes inputs, and responding SLAVE asserts [ACK_I].
SLAVE prepares to latch data on [DAT_O()] and [TGD_O()].
SLAVE asserts [ACK_I] in response to [STB_O] to indicate latched data.
MASTER monitors [ACK_I], and prepares to terminate the cycle.

Note: SLAVE may insert wait states (-WSS-) before asserting [ACK_I], thereby allowing it to throttle the cycle speed. Any number of wait states may be added.

CLOCK EDGE 1: SLAVE latches data on [DAT_O()] and [TGD_O()].
MASTER negates [STB_O] and [CYC_O] to indicate the end of the cycle.
SLAVE negates [ACK_I[in response to negated [STB_O].

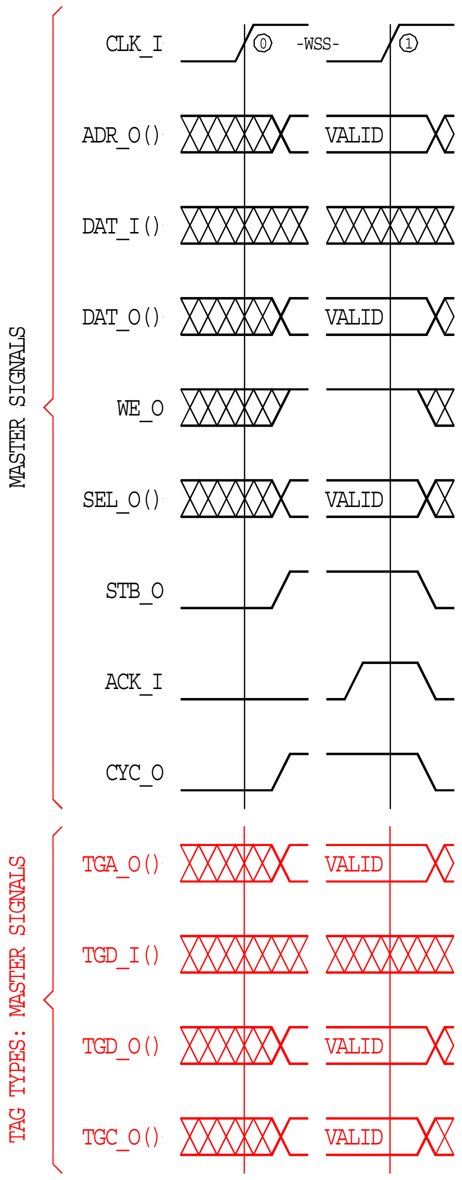


Figure 3-4. SINGLE WRITE cycle.

3.3 BLOCK READ / WRITE Cycles

The BLOCK transfer cycles perform multiple data transfers. They are very similar to single READ and WRITE cycles, but have a few special modifications to support multiple transfers.

During BLOCK cycles, the interface basically performs SINGLE READ/WRITE cycles as described above. However, the BLOCK cycles are modified somewhat so that these individual cycles (called *phases*) are combined together to form a single BLOCK cycle. This function is most useful when multiple MASTERs are used on the interconnect. For example, if the SLAVE is a shared (dual port) memory, then an arbiter for that memory can determine when one MASTER is done with it so that another can gain access to the memory.

As shown in Figure 3-5, the [CYC_O] signal is asserted for the duration of a BLOCK cycle. This signal can be used to request permission to access a shared resource from a local arbiter. To hold the access until the end of the cycle the [LOCK_O] signal must be asserted, as is shown. During each of the data transfer phases (within the block transfer), the normal handshaking protocol between [STB_O] and [ACK_I] is maintained.

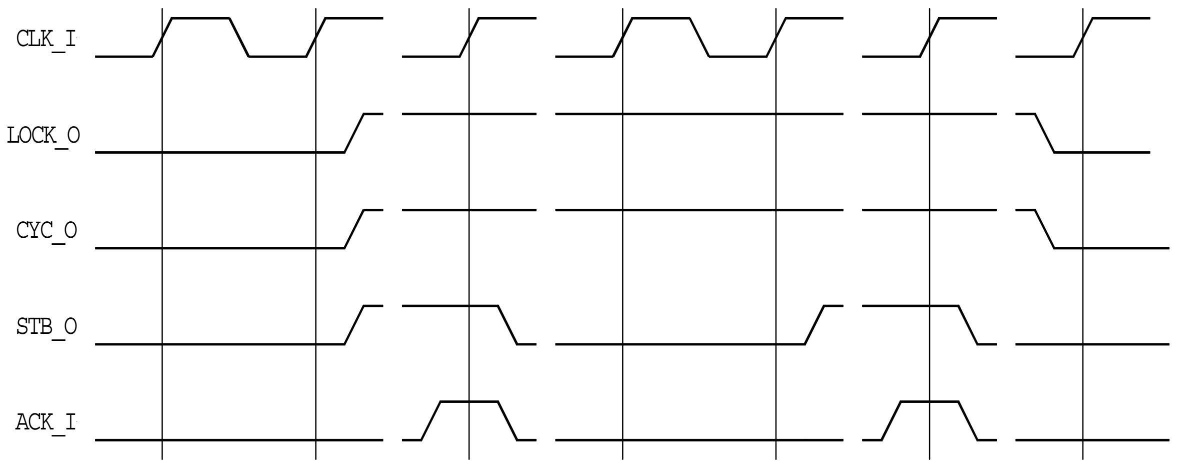


Figure 3-5. Use of [CYC_O] signal during BLOCK cycles.

RULE 3.80
All MASTER and SLAVE interfaces that support BLOCK cycles MUST conform to the timing requirements given in sections 3.3.1 and 3.3.2.

PERMISSION 3.55
MASTER and SLAVE interfaces MAY be designed so that they do not support the BLOCK cycles.

PSL Code

```
// High level property

// [property regarding all masters]

property lock_mechanism =
forall i in {0:`NUM_OF_MASTERS-1}:
      forall j in {0:`NUM_OF_MASTERS-1}:
            always !(i=j) -> !( mLOCK_O[i] && mLOCK_O[j]);

// [assertion regarding all masters]

assert lock_mechanism;
```

3.3.1 BLOCK READ Cycle

Figure 3-6 shows a BLOCK READ cycle. The BLOCK cycle is capable of a data transfer on every clock cycle. However, this example also shows how the MASTER and the SLAVE interfaces can both throttle the bus transfer rate by inserting wait states. A total of five transfers (phases) are shown. After the second transfer the MASTER inserts a wait state. After the fourth transfer the SLAVE inserts a wait state. The cycle is terminated after the fifth transfer. The protocol for this transfer works as follows:

CLOCK EDGE 0: MASTER presents a valid address on [ADR_O()] and [TGA_O()].
MASTER negates [WE_O] to indicate a READ cycle.
MASTER presents bank select [SEL_O()] to indicate where it expects data.
MASTER asserts [CYC_O] and [TGC_O()] to indicate the start of the cycle.
MASTER asserts [STB_O] to indicate the start of the first phase.

Note: the MASTER asserts [CYC_O] and/or [TGC_O()] at, or anytime before, clock edge 1.

SETUP, EDGE 1: SLAVE decodes inputs, and responding SLAVE asserts [ACK_I].
SLAVE presents valid data on [DAT_I()] and [TGD_I()].
MASTER monitors [ACK_I], and prepares to latch [DAT_I()] and [TGD_I()].

CLOCK EDGE 1: MASTER latches data on [DAT_I()] and [TGD_I()].
MASTER presents new [ADR_O()] and [TGA_O()].
MASTER presents new bank select [SEL_O()] to indicate where it expects data.

SETUP, EDGE 2: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE presents valid data on [DAT_I()] and [TGD_I()].
MASTER monitors [ACK_I], and prepares to latch [DAT_I()] and [TGD_I()].

CLOCK EDGE 2: MASTER latches data on [DAT_I()] and [TGD_I()].
MASTER negates [STB_O] to introduce a wait state (-WSM-).

SETUP, EDGE 3: SLAVE negates [ACK_I] in response to [STB_O].

Note: any number of wait states can be inserted by the MASTER.

CLOCK EDGE 3: MASTER presents new [ADR_O()] and [TGA_O()].
MASTER presents new bank select [SEL_O()] to indicate where it expects data.
MASTER asserts [STB_O].

SETUP, EDGE 4: SLAVE decodes inputs, and responds by asserting [ACK_I].

SLAVE presents valid data on [DAT_I()] and [TGD_I()].
MASTER monitors [ACK_I], and prepares to latch [DAT_I()] and [TGD_I()].

CLOCK EDGE 4: MASTER latches data on [DAT_I()] and [TGD_I()].
MASTER presents [ADR_O()] and [TGA_O()].
MASTER presents new bank select [SEL_O()] to indicate where it expects data.

SETUP, EDGE 5: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE presents valid data on [DAT_I()] and [TGD_I()].
MASTER monitors [ACK_I], and prepares to latch [DAT_I()] and [TGD_I()].

CLOCK EDGE 5: MASTER latches data on [DAT_I()] and [TGD_I()].
SLAVE negates [ACK_I] to introduce a wait state.

Note: any number of wait states can be inserted by the SLAVE at this point.

SETUP, EDGE 6: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE presents valid data on [DAT_I()] and [TGD_I()].
MASTER monitors [ACK_I], and prepares to latch [DAT_I()] and [TGD_I()].

CLOCK EDGE 6: MASTER latches data on [DAT_I()] and [TGD_I()].
MASTER terminates cycle by negating [STB_O] and [CYC_O].

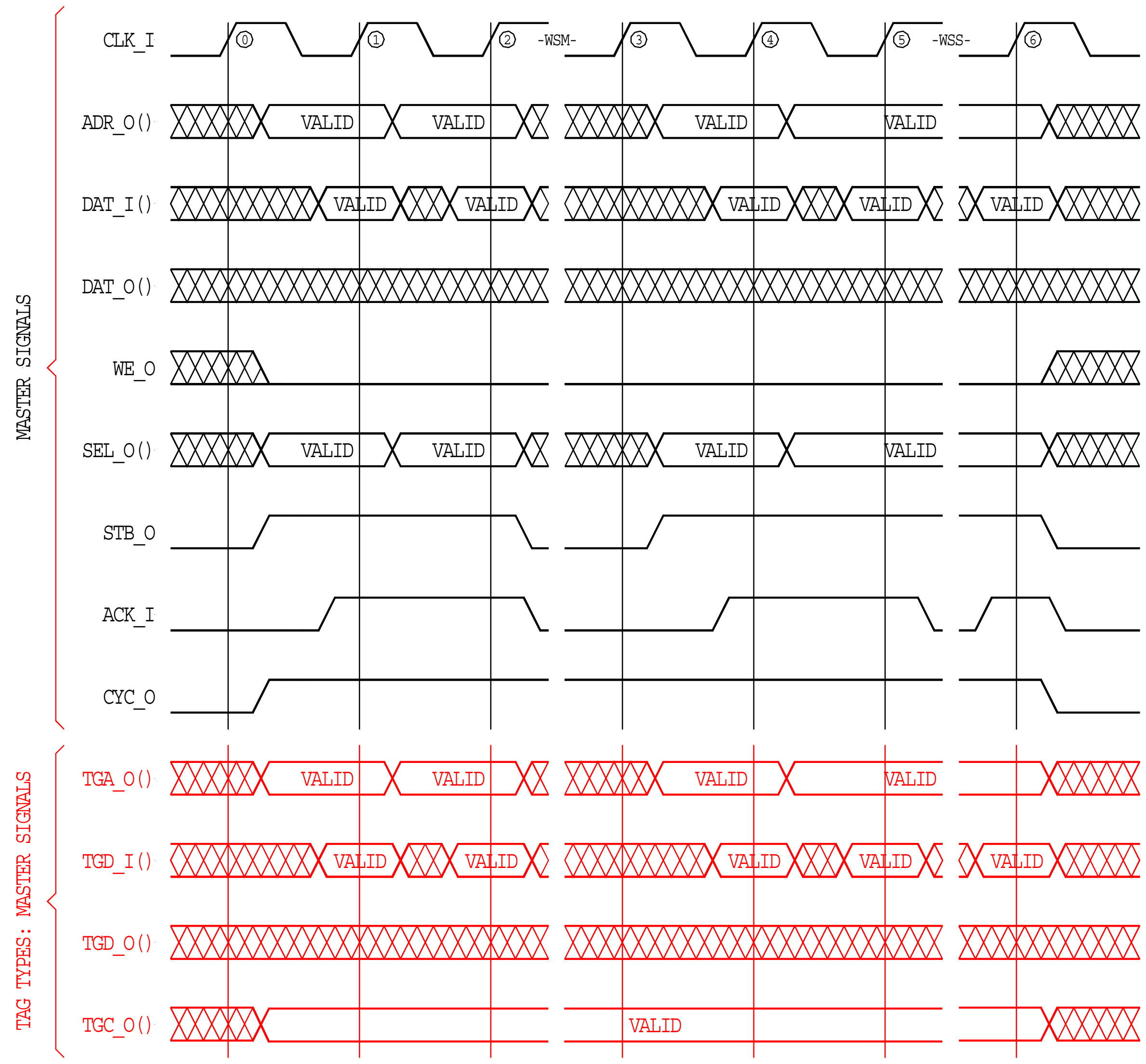


Figure 3-6. BLOCK READ cycle.

3.3.2 BLOCK WRITE Cycle

Figure 3-7 shows a BLOCK WRITE cycle. The BLOCK cycle is capable of a data transfer on every clock cycle. However, this example also shows how the MASTER and the SLAVE interfaces can both throttle the bus transfer rate by inserting wait states. A total of five transfers are shown. After the second transfer the MASTER inserts a wait state. After the fourth transfer the SLAVE inserts a wait state. The cycle is terminated after the fifth transfer. The protocol for this transfer works as follows:

CLOCK EDGE 0: MASTER presents [ADR_O()] and [TGA_O()].
MASTER asserts [WE_O] to indicate a WRITE cycle.
MASTER presents bank select [SEL_O()] to indicate where it sends data.
MASTER asserts [CYC_O] and [TGC_O()] to indicate cycle start.
MASTER asserts [STB_O].

Note: the MASTER asserts [CYC_O] and/or [TGC_O()] at, or anytime before, clock edge 1.

SETUP, EDGE 1: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE prepares to latch data on [DAT_O()] and [TGD_O()].
MASTER monitors [ACK_I], and prepares to terminate current data phase.

CLOCK EDGE 1: SLAVE latches data on [DAT_O()] and [TGD_O()].
MASTER presents [ADR_O()] and [TGA_O()].
MASTER presents new bank select [SEL_O()] to indicate where it sends data.

SETUP, EDGE 2: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE prepares to latch data on [DAT_O()] and [TGD_O()].
MASTER monitors [ACK_I], and prepares to terminate current data phase.

CLOCK EDGE 2: SLAVE latches data on [DAT_O()] and [TGD_O()].
MASTER negates [STB_O] to introduce a wait state (-WSM-).

SETUP, EDGE 3: SLAVE negates [ACK_I] in response to [STB_O].

Note: any number of wait states can be inserted by the MASTER at this point.

CLOCK EDGE 3: MASTER presents [ADR_O()] and [TGA_O()].
MASTER presents bank select [SEL_O()] to indicate where it sends data.
MASTER asserts [STB_O].

SETUP, EDGE 4: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE prepares to latch data on [DAT_O()] and [TGD_O()].
MASTER monitors [ACK_I], and prepares to terminate data phase.

CLOCK EDGE 4: SLAVE latches data on [DAT_O()] and [TGD_O()].
MASTER presents [ADR_O()] and [TGA_O()].
MASTER presents new bank select [SEL_O()] to indicate where it sends data.

SETUP, EDGE 5: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE prepares to latch data on [DAT_O()] and [TGD_O()].
MASTER monitors [ACK_I], and prepares to terminate data phase.

CLOCK EDGE 5: SLAVE latches data on [DAT_O()] and [TGD_O()].
SLAVE negates [ACK_I] to introduce a wait state.

Note: any number of wait states can be inserted by the SLAVE at this point.

SETUP, EDGE 6: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE prepares to latch data on [DAT_O()] and [TGD_O()].
MASTER monitors [ACK_I], and prepares to terminate data phase.

CLOCK EDGE 6: SLAVE latches data on [DAT_O()] and [TGD_O()].
MASTER terminates cycle by negating [STB_O] and [CYC_O].

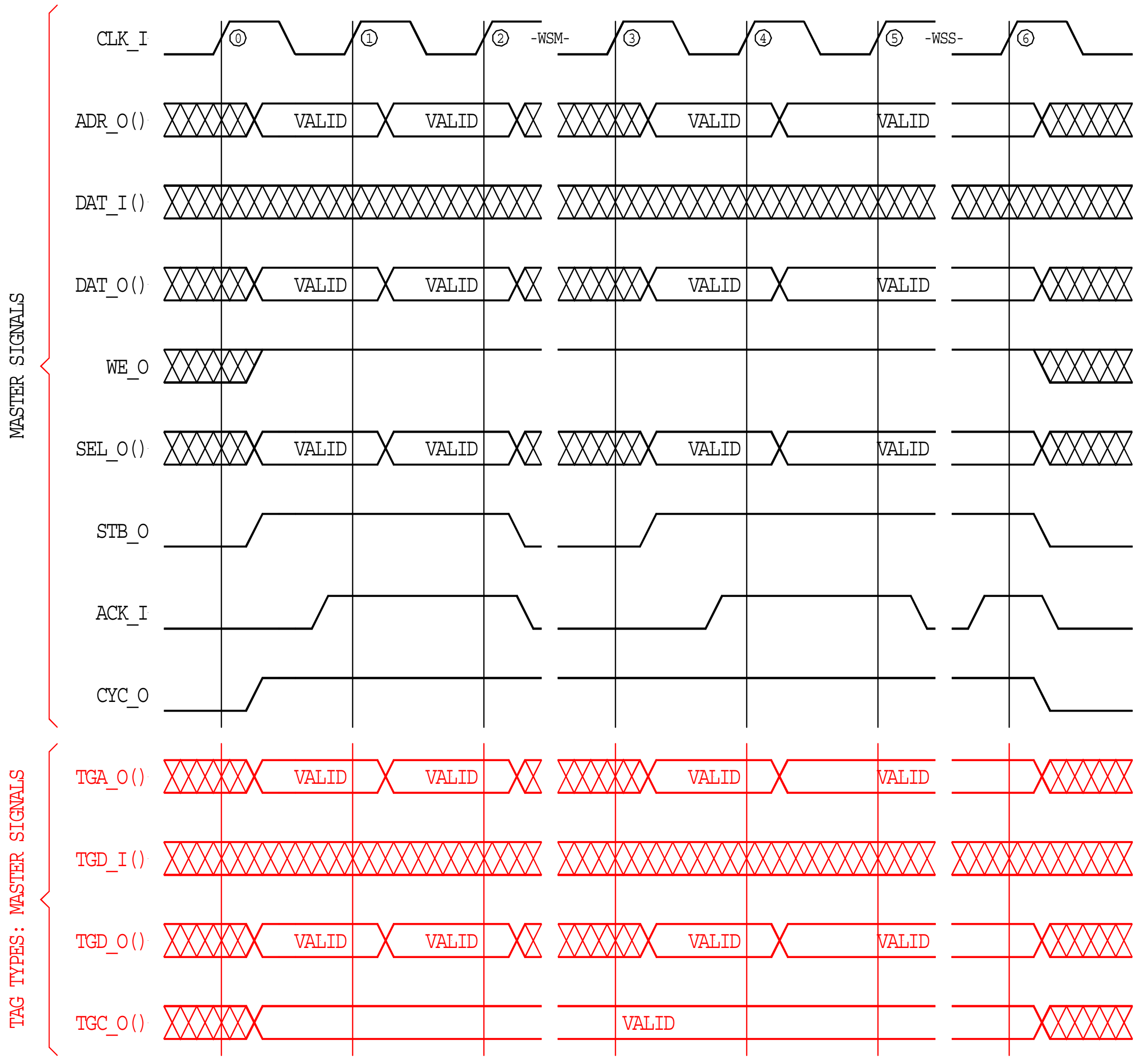


Figure 3-7. BLOCK WRITE cycle.

3.4 RMW Cycle

The RMW (read-modify-write) cycle is used for indivisible semaphore operations. During the first half of the cycle a single read data transfer is performed. During the second half of the cycle a write data transfer is performed. The [CYC_O] signal remains asserted during both halves of the cycle.

RULE 3.85
All MASTER and SLAVE interfaces that support RMW cycles MUST conform to the timing requirements given in section 3.4.

PERMISSION 3.60
MASTER and SLAVE interfaces MAY be designed so that they do not support the RMW cycles.

Figure 3-8 shows a read-modify-write (RMW) cycle. The RMW cycle is capable of a data transfer on every clock cycle. However, this example also shows how the MASTER and the SLAVE interfaces can both throttle the bus transfer rate by inserting wait states. Two transfers are shown. After the first (read) transfer, the MASTER inserts a wait state. During the second transfer the SLAVE inserts a wait state. The protocol for this transfer works as follows:

CLOCK EDGE 0: MASTER presents [ADR_O()] and [TGA_O()].
MASTER negates [WE_O] to indicate a READ cycle.
MASTER presents bank select [SEL_O()] to indicate where it expects data.
MASTER asserts [CYC_O] and [TGC_O()] to indicate the start of cycle.
MASTER asserts [STB_O].

Note: the MASTER asserts [CYC_O] and/or [TGC_O()] at, or anytime before, clock edge 1. The use of [TAGN_O] is optional.

SETUP, EDGE 1: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE presents valid data on [DAT_I()] and [TGD_I()].
MASTER monitors [ACK_I], and prepares to latch [DAT_I()] and [TGD_I()].

CLOCK EDGE 1: MASTER latches data on [DAT_I()] and [TGD_I()].
MASTER negates [STB_O] to introduce a wait state (-WSM-).

SETUP, EDGE 2: SLAVE negates [ACK_I] in response to [STB_O].
MASTER asserts [WE_O] to indicate a WRITE cycle.

Note: any number of wait states can be inserted by the MASTER at this point.

CLOCK EDGE 2: MASTER presents WRITE data on [DAT_O()] and [TGD_O()].

MASTER presents new bank select [SEL_O()] to indicate where it sends data.
MASTER asserts [STB_O].

SETUP, EDGE 3: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE prepares to latch data on [DAT_O()] and [TGD_O()].
MASTER monitors [ACK_I], and prepares to terminate data phase.

Note: any number of wait states can be inserted by the SLAVE at this point.

CLOCK EDGE 3: SLAVE latches data on [DAT_O()] and [TGD_O()].
MASTER negates [STB_O] and [CYC_O] indicating the end of the cycle.
SLAVE negates [ACK_I] in response to negated [STB_O].

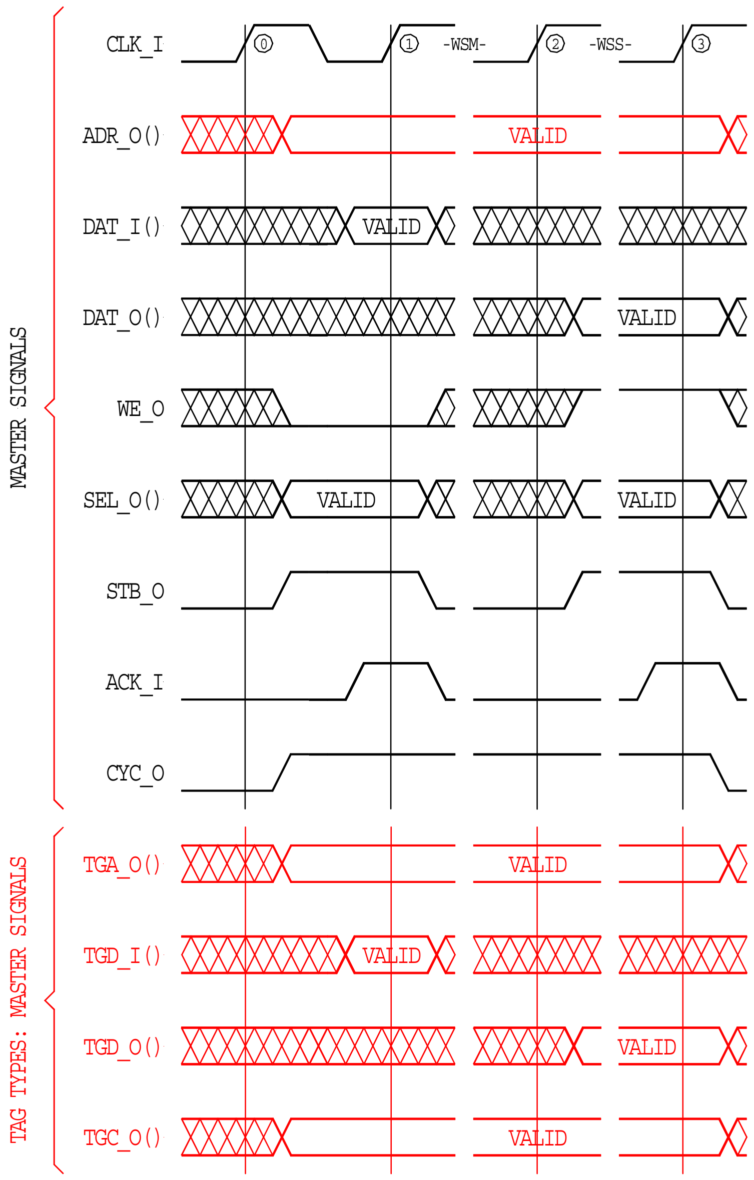


Figure 3-8. RMW cycle.

This page is Intentionally Blank

3.5 Data Organization

Data organization refers to the ordering of data during transfers. There are two general types of ordering. These are called BIG ENDIAN and LITTLE ENDIAN. BIG ENDIAN refers to data ordering where the most significant portion of an operand is stored at the lower address. LITTLE ENDIAN refers to data ordering where the most significant portion of an operand is stored at the higher address. The WISHBONE architecture supports both methods of data ordering.

3.5.1 Nomenclature

A BYTE(N), WORD(N), DWORD(N) and QWORD(N) nomenclature is used to define data ordering. These terms are defined in Table 3-1. Figure 3-9 shows the operand locations for input and output data ports.

Table 3-1. Data Transfer Nomenclature

Nomenclature	Granularity	Description
BYTE(N)	8-bit	An 8-bit BYTE transfer at address ‘N’.
WORD(N)	16-bit	A 16-bit WORD transfer at address ‘N’.
DWORD(N)	32-bit	A 32-bit Double WORD transfer at address ‘N’.
QWORD(N)	64-bit	A 64-bit Quadruple WORD transfer at address ‘N’.

The table also defines the granularity of the interface. This indicates the minimum unit of data transfer that is supported by the interface. For example, the smallest operand that can be passed through a port with 16-bit granularity is a 16-bit WORD. In this case, an 8-bit operand cannot be transferred.

Figure 3-10 shows an example of how the 64-bit value of 0x0123456789ABCDEF is transferred through BYTE, WORD, DWORD and QWORD ports using BIG ENDIAN data organization. Through the 64-bit QWORD port the number is directly transferred with the most significant bit at DAT_I(63) / DAT_O(63). The least significant bit is at DAT_I(0) / DAT_O(0). However, when the same operand is transferred through a 32-bit DWORD port, it is split into two bus cycles. The two bus cycles are each 32-bits in length, with the most significant DWORD transferred at the lower address, and the least significant DWORD transferred at the upper address. A similar situation applies to the WORD and BYTE cases.

Figure 3-11 shows an example of how the 64-bit value of 0x0123456789ABC is transferred through BYTE, WORD, DWORD and QWORD ports using LITTLE ENDIAN data organization. Through the 64-bit QWORD port the number is directly transferred with the most significant bit at DAT_I(63) / DAT_O(63). The least significant bit is at DAT_I(0) / DAT_O(0). However, when the same operand is transferred through a 32-bit DWORD port, it is split into two bus cycles. The two bus cycles are each 32-bits in length, with the least significant DWORD transferred at the lower address, and the most significant DWORD transferred at the upper address. A similar situation applies to the WORD and BYTE cases.

PSL Remark

The following properties assume a memory definition. For example the memory can be defined as follows.

PSL Code

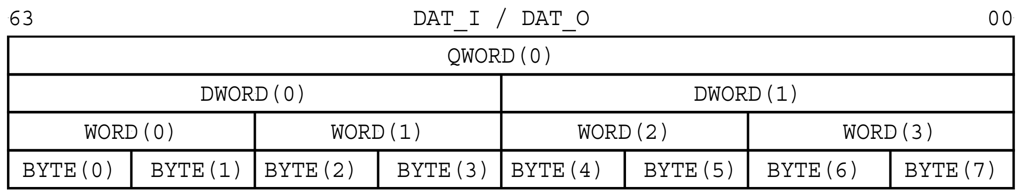
```
`define reg [7:0] mem [0:MAX_ADDR][7:0]
```

PSL Remark

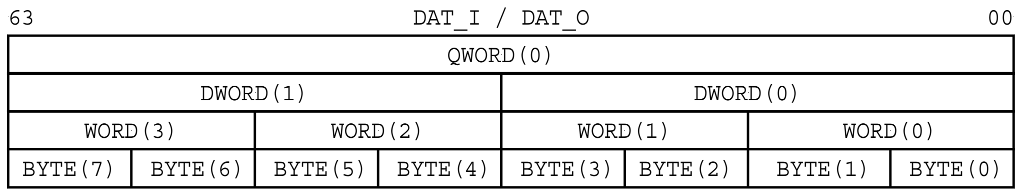
The following properties assume the DAT_I and DAT_O arrays are defined so that their finest granularity (determined by PORT_GRANULAIRTY) can be accessed. For example, DAT_I can be defined as follows, where NUM_OF_PORTS is defined as the ration between PORT_SIZE and PORT_GRANULAIRTY.

PSL Code

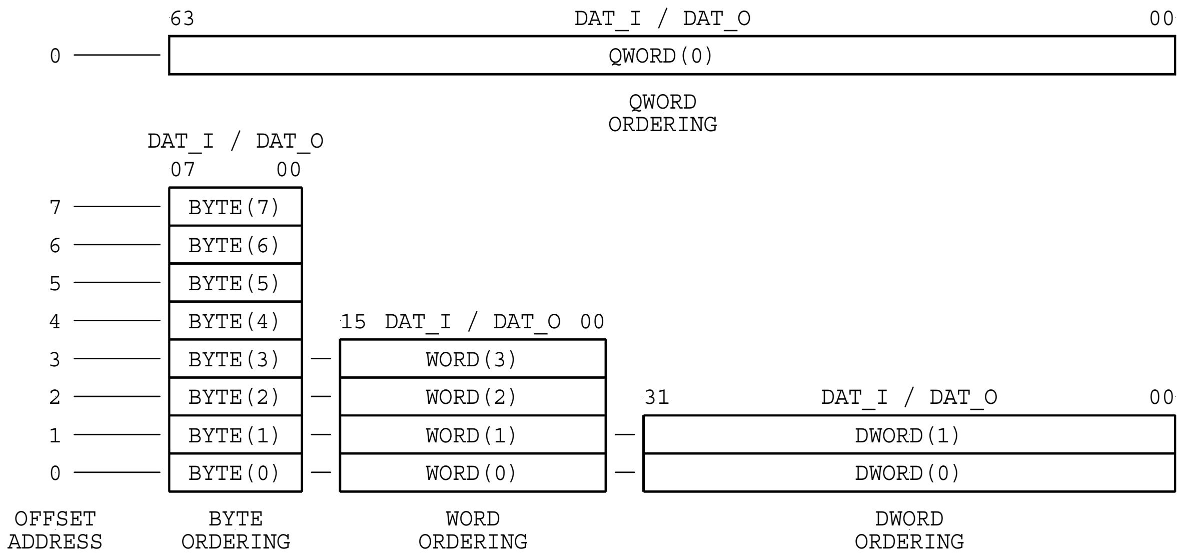
```
`define reg [PORT_GRAN-1:0] DAT_I [0:`NUM_OF_PORTS-1]

// Top-level-property regarding correct data transfer during read
// Taking into account data ordering as described in
// Rule 3.90 and Rule 3.95
// clock should be defined as fast_clock or slow_clock according
// to the whether this is a classic cycle or a registered bus cycle

// [property regarding one master and one slave]

property read_phase_correct_data_with_regard_to_ordering(boolean
mCYC_O, mSTB_O, mWE_O, sACK_I, ordering; bitvector  mSEL_O, mADR_O,
sDAT_I; numeric num_of_ports)
=
forall addr in {0:`MAX_ADDR-1}:
forall sel in {0:`NUM_OF_PORTS-1}:
always
({ mCYC_O && mSTB_O && !mWE_O && rose(sACK_I) &&
   mADR_O[0:MAX_ADDR]==addr && mSEL_O[0:`NUM_OF_PORTS -1] == sel}
    |->
    { (sDAT_I[sel]==mem[addr][sel] && ordering == 0) |
      (sDAT_I[sel]==mem[addr][num_of_ports-sel] && ordering == 1) }
)@clock

// [assertion regarding all masters and all slaves]

assert
forall i in {0:`NUM_OF_MASTERS-1}:
forall j in {0:`NUM_OF_SLAVES-1}:
```



(a) BIG ENDIAN BYTE, WORD, DWORD and QWORD positioning in a 64-bit operand.



(b) LITTLE ENDIAN BYTE, WORD, DWORD and QWORD positioning in a 64-bit operand.



(c) Address nomenclature.

Figure 3-9. Operand locations for input and output data ports.

```
read_phase_correct_data_with_regard_to_ordering(mCYC_O[i], mSTB_O[i],
mWE_O[i], sACK_I[j], sORDERING[j], mSEL_O[i], mADR_O[i],sDAT_I[j],
NUM_OF_PORTS[i]);

// Top-level-property regarding correct data transfer during write
// Taking into account data ordering as described in
// Rule 3.90 and Rule 3.95
// clock should be defined as fast_clock or slow_clock according
// to whether this is a classic cycle or a registered bus cycle

// [property regarding one master and one slave]

property write_phase_correct_data(boolean mCYC_O,mSTB_O, mWE_O, sACK_I,
ordering; bitvector mADR_O,mDAT_O,mSEL_O,sDAT_I)
=
forall addr in {0:`MAX_ADDR-1}:
  forall data in {0:`PORT_SIZE-1}:
    forall sel in {0:`NUM_OF_PORTS-1}:
     always
     ({mCYC_O && mSTB_O && mADR_O=addr && mWE_O && mDAT_O=data &&
       mSEL_O = sel; sACK_I[->] }
       |=>
      {sDAT_O[sel]==mem[addr][sel] && ordering == 0 |
       sDAT_O[sel]==mem[addr][`NUM_OF_PORTS] && ordering == 1}
     )@clock

// [assertion regarding all masters and all slaves]

assert
forall i in {0:`NUM_OF_MASTERS-1}:
forall j in {0:`NUM_OF_SLAVES-1}:
write_phase_correct_data
(mCYC_O[i], mSTB_O[i], mWE_O[i], sACK_I[j],mORDERRING[j], mADR_O[i],
 mDAT_O[i], mSEL_O[i], sDAT_I[j]);
```

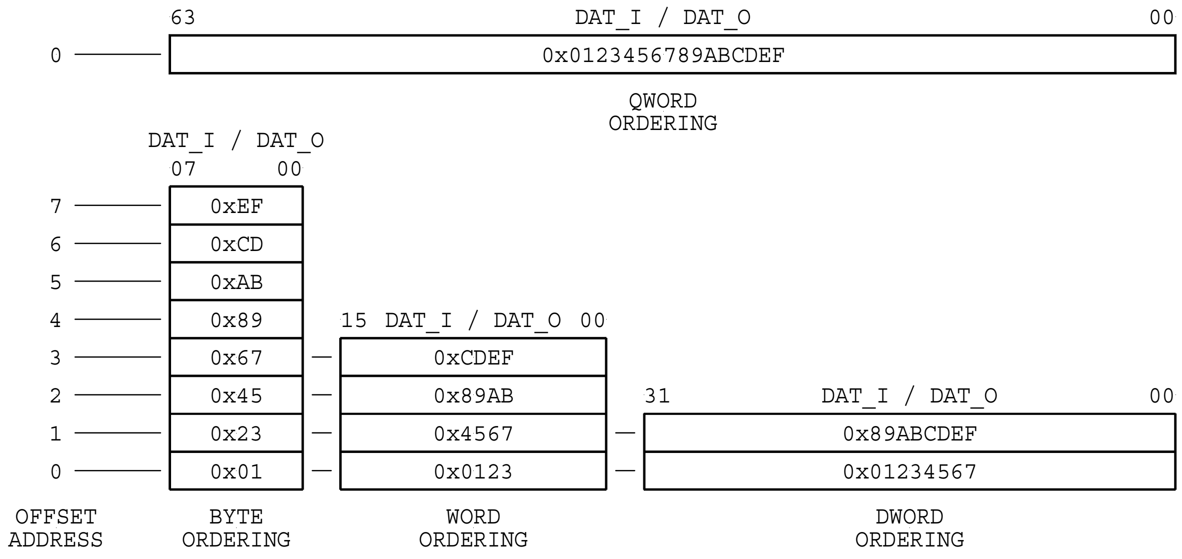


Figure 3-10. Example showing a variety of BIG ENDIAN transfers over various port sizes.

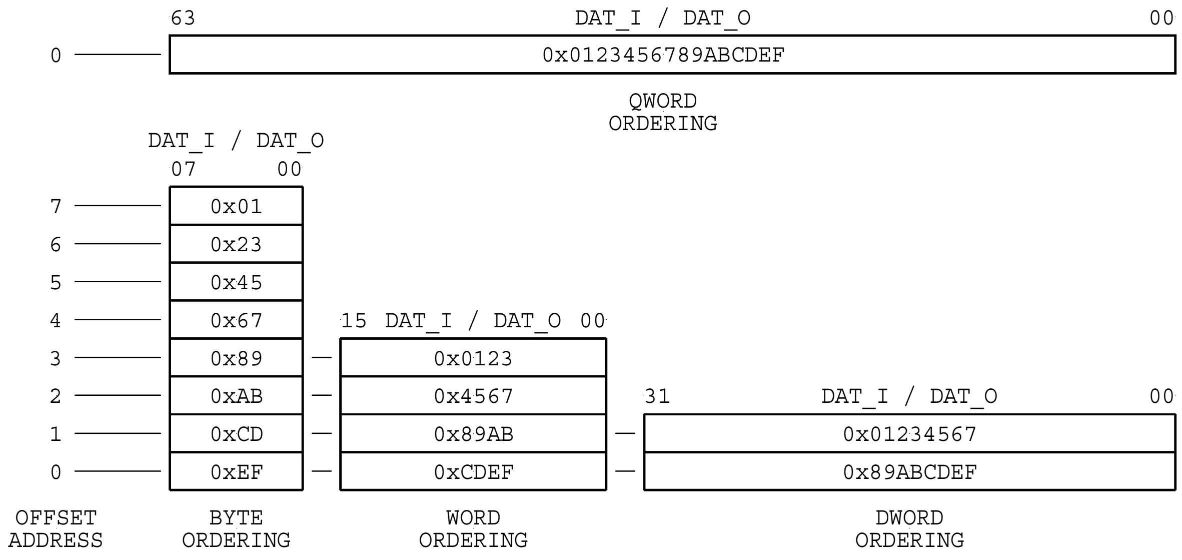


Figure 3-11. Example showing a variety of LITTLE ENDIAN transfers over various port sizes.

RULE 3.90
Data organization MUST conform to the ordering indicated in Figure 3-9.

3.5.2 Transfer Sequencing

The sequence in which data is transferred through a port is not regulated by this specification. For example, a 64-bit operand through a 32-bit port will take two bus cycles. However, the specification does not require that the lower or upper DWORD be transferred first.

RECOMMENDATION 3.20
Design interfaces so that data is transferred sequentially from lower addresses to higher addresses.

OBSERVATION 3.60
The sequence in which an operand is transferred through a data port is not highly regulated by the specification. That is because different IP cores may produce the data in different ways. The sequence is therefore application-specific.

3.5.3 Data Organization for 64-bit Ports

RULE 3.95
Data organization on 64-bit ports MUST conform to Figure 3-12.

64-bit Data Bus With 8-bit (BYTE) Granularity									
	Address Range:	Active Portion of Data Bus							
	ADR_I ADR_O (63..03)	DAT_I DAT_O (63..56)	DAT_I DAT_O (55..48)	DAT_I DAT_O (47..40)	DAT_I DAT_O (39..32)	DAT_I DAT_O (31..24)	DAT_I DAT_O (23..16)	DAT_I DAT_O (15..08)	DAT_I DAT_O (07..00)
	Active Select Line	SEL_I(7) SEL_O(7)	SEL_I(6) SEL_O(6)	SEL_I(5) SEL_O(5)	SEL_I(4) SEL_O(4)	SEL_I(3) SEL_O(3)	SEL_I(2) SEL_O(2)	SEL_I(1) SEL_O(1)	SEL_I(0) SEL_O(0)
BYTE Ordering	BIG ENDIAN	BYTE(0)	BYTE(1)	BYTE(2)	BYTE(3)	BYTE(4)	BYTE(5)	BYTE(6)	BYTE(7)
	LITTLE ENDIAN	BYTE(7)	BYTE(6)	BYTE(5)	BYTE(4)	BYTE(3)	BYTE(2)	BYTE(1)	BYTE(0)

64-bit Data Bus With 16-bit (WORD) Granularity					
	Address Range	Active Portion of Data Bus			
	ADR_I ADR_O (63..02)	DAT_I DAT_O (63..48)	DAT_I DAT_O (47..32)	DAT_I DAT_O (31..16)	DAT_I DAT_O (15..00)
	Active Select Line	SEL_I(3) SEL_O(3)	SEL_I(2) SEL_O(2)	SEL_I(1) SEL_O(1)	SEL_I(0) SEL_O(0)
WORD Ordering	BIG ENDIAN	WORD(0)	WORD(1)	WORD(2)	WORD(3)
	LITTLE ENDIAN	WORD(3)	WORD(2)	WORD(1)	WORD(0)

64-bit Data Bus With 32-bit (DWORD) Granularity			
	Address Range	Active Portion of Data Bus	
	ADR_I ADR_O (63..01)	DAT_I DAT_O (63..32)	DAT_I DAT_O (31..00)
	Active Select Line	SEL_I(1) SEL_O(1)	SEL_I(0) SEL_O(0)
DWORD Ordering	BIG ENDIAN	DWORD(0)	DWORD(1)
	LITTLE ENDIAN	DWORD(1)	DWORD(0)

64-bit Data Bus With 64-bit (QWORD) Granularity		
	Address Range	Active Portion of Data Bus
	ADR_I ADR_O (63..00)	DAT_I DAT_O (63..00)
	Active Select Line	SEL_I(0) SEL_O(0)
QWORD Ordering	BIG ENDIAN	QWORD(0)
	LITTLE ENDIAN	QWORD(0)

Figure 3-12. Data organization for 64-bit ports.

3.5.4 Data Organization for 32-bit Ports

RULE 3.100
Data organization on 32-bit ports MUST conform to Figure 3-13.

32-bit Data Bus With 8-bit (BYTE) Granularity					
	Address Range	Active Portion of Data Bus			
	ADR_I ADR_O (63..02)	DAT_I DAT_O (31..24)	DAT_I DAT_O (23..16)	DAT_I DAT_O (15..08)	DAT_I DAT_O (07..00)

	Active Select Line	SEL_I(3)
		SEL_O(3)
			SEL_I(2)
			SEL_O(2)
				SEL_I(1)
				SEL_O(1)
					SEL_I(0)
					SEL_O(0)

BYTE Ordering	BIG ENDIAN	BYTE(0)
		BYTE(4)
			BYTE(1)
			BYTE(5)
				BYTE(2)
				BYTE(6)
					BYTE(3)
					BYTE(7)

	LITTLE ENDIAN	BYTE(3)
		BYTE(7)
			BYTE(2)
			BYTE(6)
				BYTE(1)
				BYTE(5)
					BYTE(0)
					BYTE(4)

32-bit Data Bus With 16-bit (WORD) Granularity			
	Address Range	Active Portion of Data Bus	
	ADR_I ADR_O (63..01)	DAT_I DAT_O (31..16)	DAT_I DAT_O (15..00)

	Active Select Line	SEL_I(1)
		SEL_O(1)
			SEL_I(0)
			SEL_O(0)

WORD Ordering	BIG ENDIAN	WORD(0)
		WORD(2)
			WORD(1)
			WORD(3)

	LITTLE ENDIAN	WORD(1)
		WORD(3)
			WORD(0)
			WORD(2)

32-bit Data Bus With 32-bit (DWORD) Granularity		
	Address Range	Active Portion of Data Bus
	ADR_I ADR_O (63..00)	DAT_I DAT_O (31..00)

	Active Select Line	SEL_I(0)
		SEL_O(0)

DWORD Ordering	BIG ENDIAN	DWORD(0)
		DWORD(1)

	LITTLE ENDIAN	DWORD(0)
		DWORD(1)

Figure 3-13. Data organization for 32-bit ports.

3.5.5 Data Organization for 16-bit Ports

RULE 3.105
Data organization on 16-bit ports MUST conform to Figure 3-14.

16-bit Data Bus With 8-bit (BYTE) Granularity			
	Address Range	Active Portion of Data Bus	
	ADR_I ADR_O (63..01)	DAT_I DAT_O (15..08)	DAT_I DAT_O (07..00)

	Active
	Select
	Line
		SEL_I(1)
		SEL_O(1)
			SEL_I(0)
			SEL_O(0)

BYTE
Ordering
	BIG
	ENDIAN
		BYTE(0)
		BYTE(2)
		BYTE(4)
		BYTE(6)
			BYTE(1)
			BYTE(3)
			BYTE(5)
			BYTE(7)

	LITTLE
	ENDIAN
		BYTE(1)
		BYTE(3)
		BYTE(5)
		BYTE(7)
			BYTE(0)
			BYTE(2)
			BYTE(4)
			BYTE(6)

16-bit Data Bus With 16-bit (WORD) Granularity		
	Address Range	Active Portion of Data Bus
	ADR_I ADR_O (63..00)	DAT_I DAT_O (15..00)

	Active
	Select
	Line
		SEL_I(0)
		SEL_O(0)

WORD
Ordering
	BIG
	ENDIAN
		WORD(0)
		WORD(1)
		WORD(2)
		WORD(3)

	LITTLE
	ENDIAN
		WORD(0)
		WORD(1)
		WORD(2)
		WORD(3)

Figure 3-14. Data organization for 16-bit ports.

3.5.6 Data Organization for 8-bit Ports

RULE 3.1010
Data organization on 8-bit ports MUST conform to Figure 3-15.

8-bit Data Bus With 8-bit (BYTE) Granularity		
	Address Range	Active Portion of Data Bus
	ADR_I ADR_O (63..00)	DAT_I DAT_O (07..00)

	Active
	Select
	Line
		SEL_I(0)
		SEL_O(0)

BYTE
Ordering
	BIG
	ENDIAN
		BYTE(0)
		BYTE(1)
		BYTE(2)
		BYTE(3)
		BYTE(4)
		BYTE(5)
		BYTE(6)
		BYTE(7)

	LITTLE
	ENDIAN
		BYTE(0)
		BYTE(1)
		BYTE(2)
		BYTE(3)
		BYTE(4)
		BYTE(5)
		BYTE(6)
		BYTE(7)

Figure 3-15. Data organization for 8-bit ports.

3.6 References

Cohen, Danny. *On Holy Wars and a Plea for Peace.* IEEE Computer Magazine, October 1981. Pages 49-54. [Description of BIG ENDIAN and LITTLE ENDIAN.]

Chapter 4 – WISHBONE Registered Feedback Bus Cycles

4.1 Introduction, Synchronous vs. Asynchronous cycle termination

To achieve the highest possible throughput, WISHBONE Classic requires asynchronous cycle termination signals. This results in an asynchronous loop from the MASTER, through the INTERCONN to the SLAVE, and then from the SLAVE through the INTERCONN back to the MASTER, as shown in figure 4-1. In large System-on-Chip devices this routing delay between MASTER and SLAVE is the dominant timing factor. This is especially true for deep sub-micron technologies.

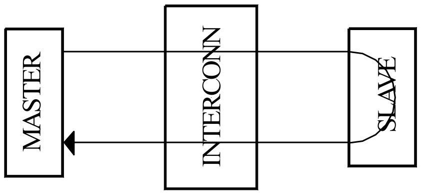


Figure 4-1 Asynchronous cycle termination path

The simplest solution for reducing the delay is to cut the loop, by using synchronous cycle termination signals. However, this introduces a wait state for every transfer, as shown in figure 4-3.

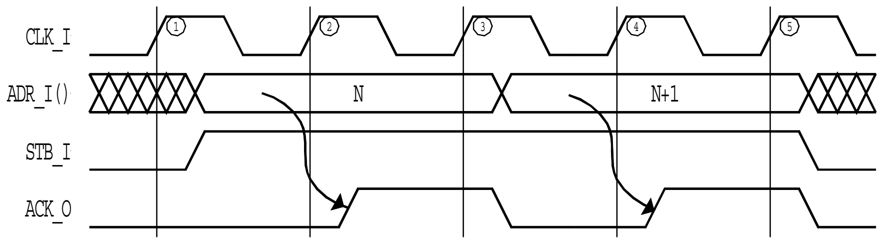


Figure 4-3 WISHBONE Classic synchronous cycle terminated burst

During cycle-1 the MASTER initiates a transfer. The addressed SLAVE responds in the next cycle with the assertion of ACK_O. During cycle-3 the MASTER initiates a second cycle, addressing the same SLAVE. Because the SLAVE does not know in advance it is being addressed again, it has to negate ACK_O. At the earliest it can respond in cycle-4, after which it has to negate ACK_O again in cycle-5.

Each transfer takes two WISHBONE cycles to complete, thus only half of the available bandwidth is useable. If the SLAVE would know in advance that it is being addressed again, it could already respond in cycle-3. Decreasing the amount of cycles needed to perform the transfers, and thus increasing throughput. The waveforms for that cycle are as shown in figure 4-4.

PSL Remark

The properties described in this section assume a slow clock has been defined. The slow clock is high in every edge of the clock CLK_I. For example a slow clock can be defined as follows:

PSL Code

```
// defining a Boolean expression that holds (once) in each edge of CLK_I
property slow_clock = next(fell(CLK_I));
```

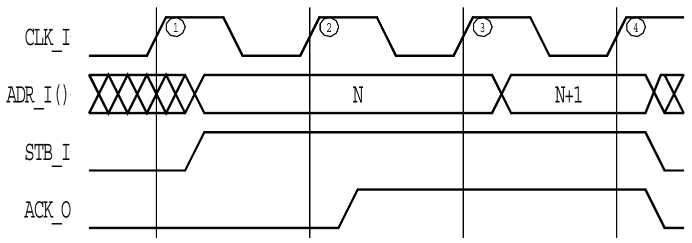


Figure 4-4 Advanced synchronous terminated burst

During cycle-1 the MASTER initiates a transfer. The addressed SLAVE responds in the next cycle with the assertion of ACK_O. The MASTER starts a new transfer in cycle-3. The SLAVE knows in advance it is being addressed again, therefore it keeps ACK_O asserted.

A two cycle burst now takes three cycles to complete, instead of four. This is a throughput increase of 33%. WISHBONE Classic however would require only 2 cycles. An eight cycle burst takes nine cycles to complete, instead of 16. This is a throughput increase of 77%. WISHBONE Classic would require eight cycles. For single transfers there is no performance gain.

Table 4-1 Burst comparison

Burst length	Asynchronous Cycle termination	Synchronous Cycle termination	Advanced Synchronous Cycle termination
1	1 (200%)	2 (100%)	2
2	2 (150%)	4 (75%)	3
4	4 (125%)	8 (62%)	5
8	8 (112%)	16 (56%)	9
16	16 (106%)	32 (53%)	17
32	32 (103%)	64 (51%)	33

Table 4-1 shows a comparison between the discussed cycle termination types, for zero wait state bursts at a given bus-frequency. Asynchronous cycle termination requires only one cycle per transfer, synchronous cycle termination requires two cycles per transfer, and the advanced synchronous cycle termination requires (burst_length+1) cycles. The percentages show the relative throughput for a burst length, where the advanced synchronous cycle termination is set to 100%.

Advanced synchronous cycle termination appears to get the best from both the synchronous and asynchronous termination schemes. For single transfers it performs as well as the normal synchronous termination scheme, for large bursts it performs as well as the asynchronous termination scheme.

NOTE that for a system that already needs wait states, the advanced synchronous scheme provides the same throughput as the asynchronous scheme.

Example 4-1
A given system, with an average burst length of 8, is intended to run at over 150MHz. It is shown that moving from asynchronous termination to synchronous termination would improve timing by 1.5ns. Thus allowing a 193MHz clock frequency, instead of the 150MHz.
The asynchronous termination scheme has a theoretical throughput of 150Mcycles per sec. For the given average burst length of 8, the advanced synchronous termination scheme has a 12% lower theoretical throughput than the asynchronous termination scheme. However the increased operating frequency allows it to perform more cycles per second. The theoretical throughput for the advanced synchronous scheme is 193M / 1.12 = 172Mcycles per sec.

Example 4-2
System layout requires that all block have registered outputs. The average burst length used in the system is 4.
Moving to the advanced synchronous termination scheme improves performance by 60 %.

4.1 WISHBONE Registered Feedback

WISHBONE Registered Feedback bus cycles use the Cycle Type Identifier [CTI_O()], [CT_I()] Address Tags to implement the advanced synchronous cycle termination scheme. Both MASTER and SLAVE interfaces must support [CTI_O()], [CTI_()] in order to provide the improved bandwidth. Additional information about the type of burst is provided by the Burst Type Extension [BTE_O()], [BTE_I()] Address Tags. Because WISHBONE Registered Feedback uses Tag signals to implement the advanced synchronous cycle termination, it is inherently fully compatible with WISHBONE Classic. If only one of the interfaces (i.e. either MASTER or SLAVE) supports WISHBONE Registered Feedback bus cycles, and hence the other supports WISHBONE Classic bus cycles, the cycle terminates as though it were a WISHBONE Classic bus cycle. This eases the integration of WISHBONE Classic and WISHBONE Registered Feedback IP cores.

PERMISSION 4.00
MASTER and SLAVE interfaces MAY be designed to support WISHBONE Registered Feedback bus cycles.

RECOMMENDATION 4.00
Interfaces compatible with WISHBONE Registered Feedback bus cycles support both WISHBONE Classic and WISHBONE Registered Feedback bus cycles. It is recommended to design new IP cores to support WISHBONE Registered Feedback bus cycles, so as to ensure maximum throughput in all systems.

RULE 4.00
All WISHBONE Registered Feedback compatible cores MUST support WISHBONE Classic bus cycles.

PSL Remark

The following properties refine the properties of Chapter 3 (discussing classic bus cycles) by referring to the Cycle Type Identifier (STI) signals providing information about the registered bus cycles. In addition they refer to the slow clock rather than the fast clock. These properties check the protocol; the high-level properties asserting correct data transfer are given near section 3.5.

PSL Code

```
// [property regarding a given master and a given slave]

property
read_write_cycle_slave_response1
(boolean mCYC_O,mSTB_O,sACK_O,sRTY_O,sERR_O,sSTB_I) =
always
({rose(mCYC_O && mSTB_O)}
  |=>
  {{!sSTB_I} |
   {(mCYC_O && mSTB_O)[*] :
    (rose(sACK_O) || rose (sRTY_O) || rose(sERR_O) }}
)@slow_clock;

// [property regarding a given master and a given slave]

property
read_write_cycle_slave_response2
(boolean mCYC_O,mSTB_O,sACK_O,sRTY_O,sERR_O,sSTB_I,mCTI) =
always
( {sACK_O | sERR_O | sRTY_O}
  |->
  {(mCYC_O && mSTB_O) |  (mCTI_O[2:0]=3b`111)})@slow_clock;

// [property regarding a given master and a given slave]

property
read_write_cycle_slave_response3
(boolean mCYC_O,mSTB_O,sACK_O,sRTY_O,sERR_O,sSTB_I) =
always
({ fell(sACK_O)} |->
        {{!mSTB_O && !m_CYC_O} | {sACK_O[->] && (mSTB_O && mCYC_O)[*] }
)@slow_clock;
```

4.2 Signal Description

CTI_IO()
The Cycle Type Idenfier [CTI_IO()] Address Tag provides additional information about the current cycle. The MASTER sends this information to the SLAVE. The SLAVE can use this information to prepare the response for the next cycle.

Table 4-2 Cycle Type Identifiers

CTI_O(2:0)	Description
‘000’	Classic cycle.
‘001’	Constant address burst cycle
‘010’	Incrementing burst cycle
‘011’	*Reserved*
‘100’	*Reserved*
‘101	*Reserved*
‘110’	*Reserved*
‘111’	End-of-Burst

PERMISSION 4.05
MASTER and SLAVE interfaces MAY be designed to support the [CTI_I()] and [CTI_O()] signals. Also MASTER and SLAVE interfaces MAY be designed to support a limited number of burst types.

RULE 4.05
MASTER and SLAVE interfaces that do support the [CTI_I()] and [CTI_O()] signals MUST at least support the Classic cycle [CTI_IO()=’000’] and the End-of-Cycle [CTI_IO()=’111’].

RULE 4.10
MASTER and SLAVE interfaces that are designed to support a limited number of burst types MUST complete the unsupported cycles as though they were WISHBONE Classic cycle, i.e. [CTI_IO()= ‘000’].

PERMISSION 4.10
For description languages that allow default values for input ports (like VHDL), [CTI_I()] MAY be assigned a default value of ‘000’.

PERMISSION 4.15
In addition to the WISHBONE Classic rules for generating cycle termination signals [ACK_O], [RTY_O], and [ERR_O], a SLAVE MAY assert a termination cycle without checking the [STB_I] signal.

```
// [property regarding a given master and a given slave]

property
read_write_cycle_slave_response
(boolean mCYC_O,mSTB_O,sACK_O,sRTY_O,sERR_O,sSTB_I,mCTI_I) =
read_write_cycle_slave_response1
(boolean mCYC_O,mSTB_O,sACK_O,sRTY_O,sERR_O,sSTB_I) &&
read_write_cycle_slave_response2
(boolean mCYC_O,mSTB_O,sACK_O,sRTY_O,sERR_O,sSTB_I,mCTI_I) &&
read_write_cycle_slave_response3
(boolean mCYC_O,mSTB_O,sACK_O,sRTY_O,sERR_O,sSTB_I);

// [assertion regarding all masters and all slaves]

assert
forall i in {0:`NUM_OF_MASTERS-1}:
forall j in {0:`NUM_OF_SLAVES-1}:
read_write_cycle_slave_response
(mWE_O[i], mCYC_O[i], mSTB_O[i], sACK_O[j],sRTY_O[j], sERR_O[j],
 sSTB_I[j],mCTI_I[i]);
```

OBSERVATION 4.00
To avoid the inherent wait state in synchronous termination schemes, the SLAVE must generate the response as soon as possible (i.e. the next cycle). It can use the [CTI_I()] signals to determine the response for the next cycle. But it cannot determine the state of [STB_I] for the next cycle, therefore it must generate the response independent of [STB_I].

PERMISSION 4.20
[ACK_O], [RTY_O], and [ERR_O] MAY be asserted while [STB_O] is negated.

RULE 4.15
A cycle terminates when both the cycle termination signal and [STB_I], [STB_O] is asserted. Even if [ACK_O], [ACK_I] is asserted, the other signals are only valid when [STB_O], [STB_I] is also asserted.

BTE_IO()
The Burst Type Extension [BTE_O()] Address Tag is send by the MASTER to the SLAVE to provides additional information about the current burst. Currently this information is only relevant for incrementing bursts, but future burst types may use these signals.

Table 4-2 Burst Type Extension for Incrementing and Decrementing bursts

BTE_IO(1:0)	Description
‘00’	Linear burst
‘01’	4-beat wrap burst
‘10’	8-beat wrap burst
‘11’	16-beat wrap burst

RULE 4.20
MASTER and SLAVE interfaces that support incrementing burst cycles MUST support the [BTE_O()] and [BTE_I()] signals.

PERMISSION 4.25
MASTER and SLAVE interfaces MAY be designed to support a limited number of burst extensions.

RULE 4.25
MASTER and SLAVE interfaces that are designed to support a limited number of burst extensions MUST complete the unsupported cycles as though they were WISHBONE Classic cycle, i.e. [CTI_IO()= 000’].

4.3 Bus Cycles

4.3.1 Classic Cycle

A Classic Cycle indicates that the current cycle is a WISHBONE Classic cycle. The SLAVE terminates the cycle as described in chapter 3. There is no information about what the MASTER will do the next cycle.

PERMISSION 4.30
A MASTER MAY signal Classic Cycle indefinitely.

OBSERVATION 4.05
A MASTER that signals Classic Cycle indefinitely is a pure WISHBONE Classic MASTER. The Cycle Type Identifier [CTI_O()] signals have no effect; all SLAVE interfaces already support WISHBONE Classic cycles. They might as well not be present on the interface at all. In fact, routing them on chip may use up valuable resources. However they might be useful for arbitration logic, or to keep the buses from/to interfaces coherent.

Figure 4-5 shows a Classic read cycle. A total of two transfers are shown. The cycle is terminated after the second transfer. The protocol for this cycle works as follows:

CLOCK EDGE 0:	MASTER presents [ADR_O()].
	MASTER presents Classic Cycle on [CTI_O()].
	MASTER negates [WE_O] to indicate a READ cycle.
	MASTER presents select [SEL_O()] to indicate where it expects data.
	MASTER asserts [CYC_O] to indicate cycle start.
	MASTER asserts [STB_O].
SETUP, EDGE 1:	SLAVE decodes inputs.
	SLAVE recognizes Classic Cycle and prepares response.
	SLAVE prepares to send data.
	MASTER monitors [ACK_I] and prepares to terminate current data phase.
CLOCK EDGE 1:	SLAVE asserts [ACK_I]
	SLAVE presents data on [DAT_I()].
SETUP, EDGE 2:	SLAVE does not expect another transfer.
	MASTER prepares to latch data on [DAT_I()].
	MASTER monitors [ACK_I] and prepares to terminate current data phase.
CLOCK EDGE 2:	SLAVE negates [ACK_I].
	MASTER latches data on [DAT_I()]

	MASTER presents new address on [ADR_O()]
SETUP, EDGE 3:	SLAVE decodes inputs. SLAVE recognizes Classic Cycle and prepares response. SLAVE prepares to send data. MASTER monitors [ACK_I] and prepares to terminate current data phase.
CLOCK EDGE 3:	SLAVE asserts [ACK_I] SLAVE presents data on [DAT_I()].
SETUP, EDGE 4:	SLAVE does not expect another transfer. MASTER prepares to latch data on [DAT_I()]. MASTER monitors [ACK_I] and prepares to terminate current data phase.
CLOCK EDGE 4:	SLAVE negates [ACK_I]. MASTER latches data on [DAT_I()] MASTER negates [CYC_O] and [STB_O] ending the cycle

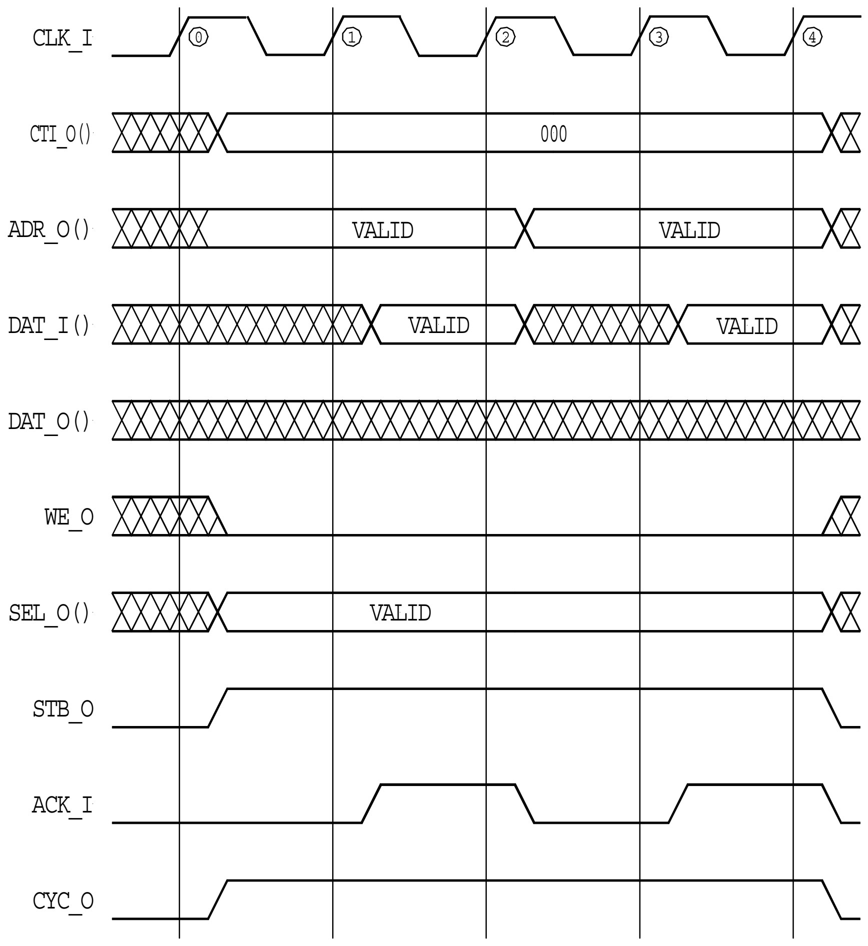


Figure 4-5 Classic Cycle

4.3.2 End-Of-Burst

End-Of-Burst indicates that the current cycle is the last of the current burst. The MASTER signals the slave that the burst ends after this transfer.

RULE 4.30
A MASTER MUST set End-Of-Burst to signal the end of the current burst.

PERMISSION 4.35
The MASTER MAY start a new cycle after the assertion of End-Of-Burst.

PERMISSION 4.40
A MASTER MAY use End-Of-Burst to indicate a single access.

OBSERVATION 4.05
A single access is in fact a burst with a burst length of one.

Figure 4-6 demonstrates the usage of End-Of-Burst. A total of three transfers are shown. The first transfer is part of a WISHBONE Registered Feedback read burst. Transfer two is the last transfer of that burst. The burst is ended when the MASTER sets [CTI_O()] to End-Of-Burst ('111'). The cycle is terminated after the third transfer, a single write transfer. The protocol for this cycle works as follows:

SETUP EDGE 0: WISHBONE Registered Feedback burst read cycle is in progress.
MASTER prepares to latch data on [DAT_I()]
MASTER monitors [ACK_I] and prepares to terminate current data phase.
MASTER prepares to end current burst
SLAVE expects another cycle and prepares response

CLOCK EDGE 0: MASTER latches data on [DAT_I()]
MASTER presents new [ADR_O()]
MASTER presents End-Of-Burst on [CTI_O()]
SLAVE presents new data on [DAT_I()]
SLAVE keeps [ACK_I] asserted to indicate that it is ready to send new data

SETUP EDGE 1: SLAVE decodes inputs.
SLAVE recognizes End-Of-Burst and prepares to terminate burst
SLAVE prepares to send last data.
MASTER prepares to latch data on [DAT_I()]
MASTER monitors [ACK_I] and prepares to terminate current data phase.

	MASTER prepares to start a new cycle
CLOCK EDGE 1:	MASTER latches data on [DAT_I()]
	MASTER starts new cycle by presenting End-Of-Burst on [CTI_O()]
	MASTER presents new address on [ADR_O()]
	MASTER presents data on [DAT_O()]
	MASTER asserts [WE_O] to indicate a WRITE cycle
	SLAVE negates [ACK_I]
SETUP, EDGE 2:	SLAVE decodes inputs
	SLAVE recognizes End-Of-Burst and prepares for a single transfer.
	SLAVE prepares response.
	MASTER monitors [ACK_I] and prepares to terminate current data phase.
CLOCK EDGE 2:	SLAVE asserts [ACK_I].
SETUP, EDGE 3:	SLAVE prepares to latch data on [DAT_O()]
	SLAVE prepares to end cycle.
	MASTER monitors [ACK_I] and prepares to terminate current data phase.
CLOCK EDGE 3:	SLAVE latches data on [DAT_O()]
	SLAVE negates [ACK_I]
	MASTER negates [CYC_O] and [STB_O] ending the cycle.

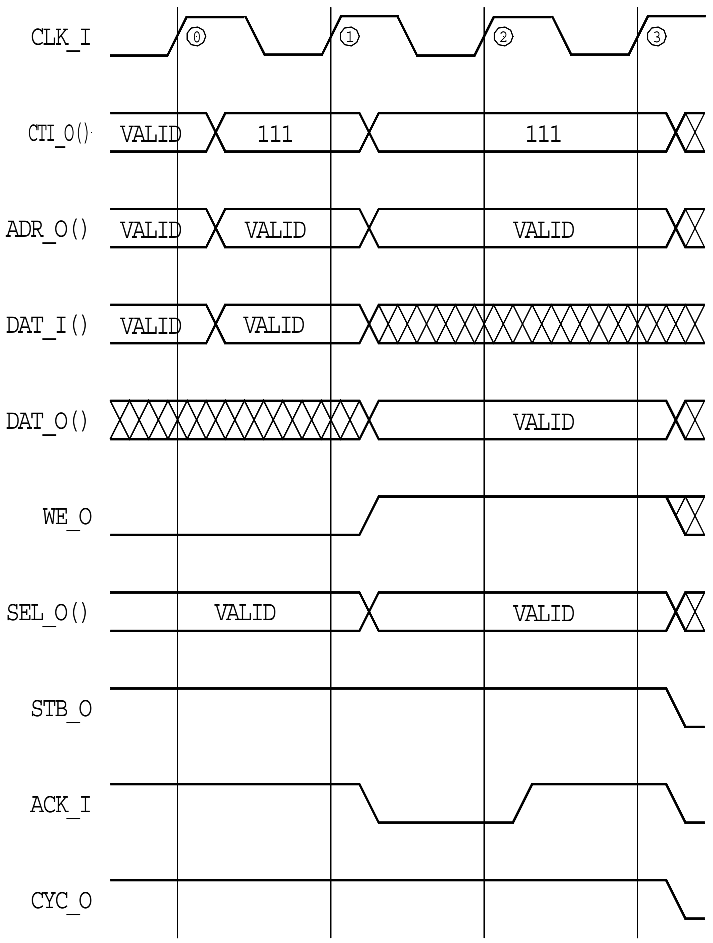


Figure 4-6 End-Of-Burst

This page is Intentionally Blank

4.3.3 Constant Address Burst Cycle

A constant address burst is defined as a single cycle with multiple accesses to the same address. Example: A MASTER reading a stream from a FIFO.

RULE 4.35
A MASTER signaling a constant address burst MUST initiate another cycle, the next cycle MUST be the same operation (either read or write), the select lines [SEL_O()] MUST have the same value, and that the address array [ADR_O()] MUST have the same value.

PERMISSION 4.40
When the MASTER signals a constant address burst, the SLAVE MAY assert the termination signal for the next cycle as soon as the current cycle terminates.

Figure 4-7 shows a CONSTANT ADDRESS BURST write cycle. After the initial setup cycle, the Constant Address Burst cycle is capable of a data transfer on every clock cycle. However, this example also shows how the MASTER and the SLAVE interfaces can both throttle the bus transfer rate by inserting wait states. A total of four transfers are shown. After the first transfer the MASTER inserts a wait state. After the second transfer the SLAVE inserts a wait state. The cycle is terminated after the fourth transfer. The protocol for this transfer works as follows:

CLOCK EDGE 0:	MASTER presents [ADR_O()].
	MASTER presents Constant Address Burst on [CTI_O()].
	MASTER asserts [WE_O] to indicate a WRITE cycle.
	MASTER presents select [SEL_O()] to indicate where it sends data.
	MASTER asserts [CYC_O] to indicate cycle start.
	MASTER asserts [STB_O].
SETUP, EDGE 1:	SLAVE decodes inputs.
	SLAVE recognizes Constant Address Burst and prepares response.
	MASTER monitors [ACK_I] and prepares to terminate current data phase.
CLOCK EDGE 1:	SLAVE asserts [ACK_I]
SETUP, EDGE 2:	SLAVE expects another transfer and prepares response for new transfer.
	SLAVE prepares to latch data on [DAT_O()].
	MASTER monitors [ACK_I] and prepares to terminate current data phase.
CLOCK EDGE 2:	SLAVE latches data on [DAT_O()].
	SLAVE keeps [ACK_I] asserted to indicate that it's ready to latch new data.
	MASTER inserts wait states by negating [STB_O].

PSL Code

```
// Rule 4.35

// [property regarding a given master]

property constant_address_burst_cycle
(boolean mWE_O; bitvector mCTI,mADR_O,mSEL_O) =
forall b in boolean:
forall addr[0:`MAX_ADDR-1] in boolean:
forall sel[0:`NUM_OF_PORTS-1] in boolean:
always
{mCTI[0:2]==3b'001 && mADR_O==addr && mWE_O == b && mSEL_O==sel} |=>
{mADR==addr && mWE_O == b && mSEL_O==sel };

// [assertion regarding all masters]

assert
forall i in {0:`NUM_OF_MASTERS-1}:
constant_address_burst_cycle(mWE_O[i], mCTI_O[i], mADR_O[i],mSEL_O[i]);
```

NOTE: any number of wait states can be inserted here.

SETUP, EDGE 3: MASTER is ready to transfer data again.

CLOCK, EDGE 3: MASTER presents [SEL_O].
MASTER presents new [DAT_O()].
MASTER asserts [STB_O].

SETUP, EDGE 4: SLAVE prepares to latch data on [DAT_O()]
MASTER monitors [ACK_I] and prepares to terminate current data phase.

CLOCK, EDGE 4: SLAVE latches data on [DAT_O()].
SLAVE inserts wait states by negating [ACK_I].
MASTER presents new [DAT_O()].

NOTE: any number of wait states can be inserted here.

SETUP, EDGE 5: SLAVE is ready to transfer data again.
MASTER monitors [ACK_I] and prepares to terminate current data phase.
MASTER prepares to signal last transfer.

CLOCK, EDGE 5: SLAVE asserts [ACK_I].

SETUP, EDGE 6: SLAVE prepares to latch data on [DAT_O()].
SLAVE expects another transfer and prepares response for new transfer.
MASTER monitors [ACK_I] and prepares to terminate current data phase.

CLOCK, EDGE 6: SLAVE latches data on [DAT_O()].
SLAVE keeps [ACK_I] asserted to indicate that it's ready to latch new data.
MASTER presents new [DAT_O()].
MASTER presents End-Of-Burst on [CTI_O()].

SETUP, EDGE 7: SLAVE prepares to latch last data of burst on [DAT_O()]
MASTER monitors [ACK_I] and prepares to terminate current cycle.

CLOCK, EDGE 7: SLAVE latches data on [DAT_O()].
SLAVE ends burst by negating [ACK_I].
MASTER negates [CYC_O] and [STB_O] ending the burst cycle.

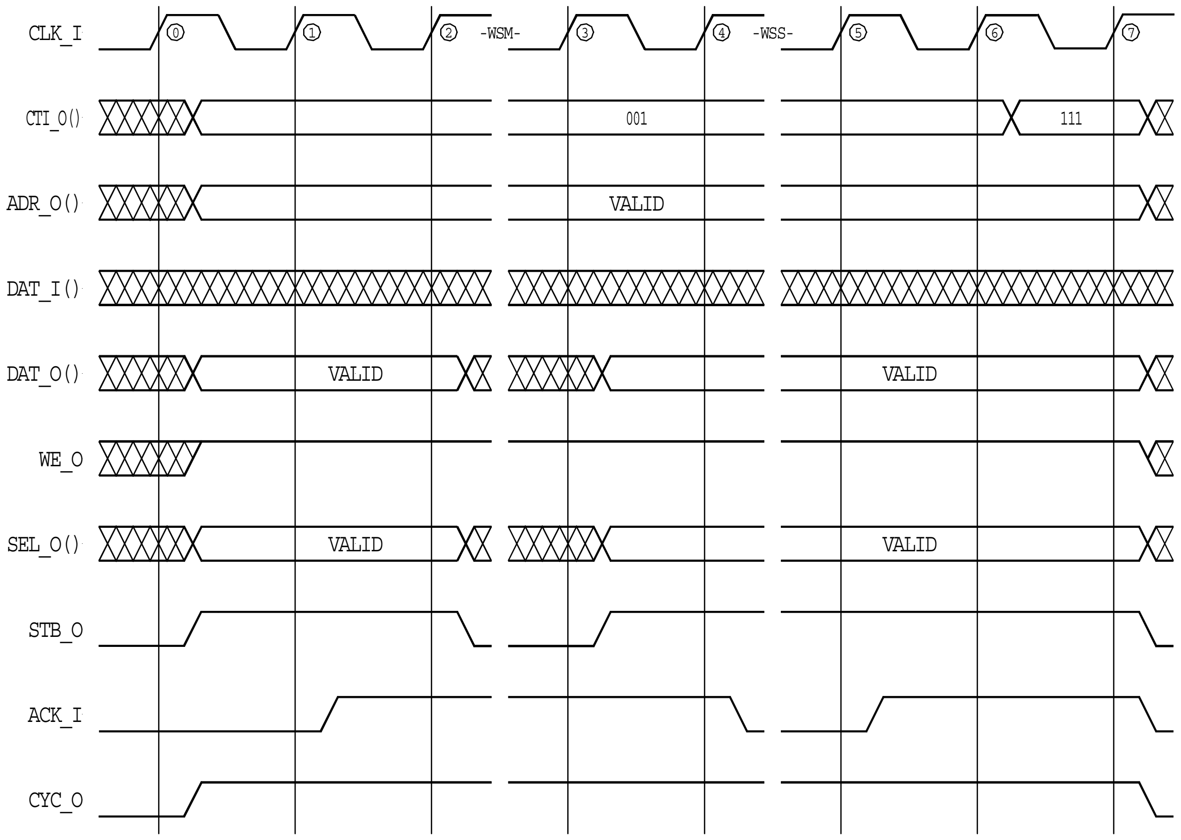


Figure 4-7 Constant address burst

4.2.3 Incrementing Burst Cycle

An incrementing burst is defined as multiple accesses to consecutive addresses. Each transfer the address is incremented. The increment is dependent on the data array [DAT_O()], [DAT_I()] size; for an 8bit data array the increment is 1, for a 16bit data array the increment is 2, for a 32bit data array the increment is 4, etc.

Increments can be linear or wrapped. Linear increments means the next address is one increment more than the current address. Wrapped increments means that the address increments one, but that the addresses' LSBs are modulo the wrap size.

Table 4-3 Wrap Size address increments

Starting address' LSBs	Linear	Wrap-4	Wrap-8
000	0-1-2-3-4-5-6-7	0-1-2-3-4-5-6-7	0-1-2-3-4-5-6-7
001	1-2-3-4-5-6-7-8	1-2-3-0-5-6-7-4	1-2-3-4-5-6-7-0
010	2-3-4-5-6-7-8-9	2-3-0-1-6-7-4-5	2-3-4-5-6-7-0-1
011	3-4-5-6-7-8-9-A	3-0-1-2-7-4-5-6	3-4-5-6-7-0-1-2
100	4-5-6-7-8-9-A-B	4-5-6-7-8-9-A-B	4-5-6-7-0-1-2-3
101	5-6-7-8-9-A-B-C	5-6-7-4-9-A-B-8	5-6-7-0-1-2-3-4
110	6-7-8-9-A-B-C-D	6-7-4-5-A-B-8-9	6-7-0-1-2-3-4-5
111	7-8-9-A-B-C-D-E	7-4-5-6-B-8-9-A	7-0-1-2-3-4-5-6

Example: Processor cache line read

RULE 4.40
A MASTER signaling an incrementing burst MUST initiate another cycle, the next cycle MUST be the same operation (either read or write), the select lines [SEL_O()] MUST have the same value, the address array [ADR_O()] MUST be incremented, and the wrap size MUST be set by the burst type extension [BTE_O()] signals.

PERMISSION 4.45
When the MASTER signals an incrementing burst, the SLAVE MAY assert the termination signal for the next cycle as soon as the current cycle terminates.

Figure 4-8 shows a 4-beat wrapped INCREMENTING BURST read cycle. A total of four transfers are shown. The protocol for this cycle works as follows:

CLOCK EDGE 0: MASTER presents [ADR_O()]
MASTER presents Incrementing Burst on [CTI_O()]
MASTER present 4-beat wrap on [BTE_O()]
MASTER negates [WE_O] to indicate a READ cycle
MASTER presents select [SEL_O()] to indicate where it expects data

PSL Code

```
// Rule 4.40

// [property regarding a given master]

property Incrementing_burst_cycle(boolean mWE_O; bitvector
mCTI,mADR_O,mSEL_O,mBTE_O) =
forall b in boolean:
forall addr[0:MAX_ADDR] in boolean:
forall sel[0:NUM_OF_PORTS] in boolean:
always
 {mCTI[0:2]==3b'010 && mADR_O==addr && mWE_O==b && mSEL_O==sel &&
  mBTE_O[1:0]==2b'00} |=> {mADR==addr+1 && mWE_O==b && mSEL_O==sel}
&&
always
 {mCTI[0:2]==3b'010 && mADR_O==addr && mWE_O==b && mSEL_O==sel &&
  mBTE_O[1:0]==2b'01} |=> {mADR==addr+1%4 && mWE_O==b && mSEL_O==sel}
&&
always
 {mCTI[0:2]==3b'010 && mADR_O==addr && mWE_O==b && mSEL_O==sel &&
  mBTE_O[1:0]==2b'10} |=> {mADR==addr+1%8 && mWE_O==b && mSEL_O==sel}
```

This page is Intentionally Blank

```
&&
always
  {mCTI[0:2]==3b'010 && mADR_O==addr && mWE_O==b && mSEL_O==sel &&
   mBTE_O[1:0]==2b'11} |=>
       {mADR==addr+1%16 && mWE_O==b && mSEL_O==sel};

// [assertion regarding all masters]

assert
forall i in {0:`NUM_OF_MASTERS-1}:
Incrementing_burst_cycle
(mWE_O[i],mCTI_O[i],mADR_O[i],mSEL_O[i],mBTE_O[i]);
```

MASTER asserts [CYC_O] to indicate cycle start
MASTER asserts [STB_O]

SETUP, EDGE 1: SLAVE decodes inputs.
SLAVE recognizes Incrementing Burst and prepares response.
MASTER prepares to latch data on [DAT_I()]
MASTER monitors [ACK_I] and prepares to terminate current data phase.

CLOCK EDGE 1: SLAVE asserts [ACK_I]
SLAVE present data on [DAT_I()]

SETUP, EDGE 2: MASTER prepares to latch data on [DAT_I()]
MASTER monitors [ACK_I] and prepares to terminate current data phase.
SLAVE expects another transfer and prepares response.

CLOCK EDGE 2: MASTER latches data on [DAT_I()]
MASTER presents new address on [ADR_O()]
SLAVE presents new data on [DAT_I()]
SLAVE keeps [ACK_I] asserted to indicate that it's ready to send new data.

SETUP, EDGE 3: MASTER prepares to latch data on [DAT_I()]
MASTER monitors [ACK_I] and prepares to terminate current data phase.
SLAVE expects another transfer and prepares response.

CLOCK, EDGE 3: MASTER latches data on [DAT_I()].
MASTER presents new address on [ADR_O()]
SLAVE presents new data on [DAT_I()].
SLAVE keeps [ACK_I] asserted to indicate that it's ready to send new data.

SETUP, EDGE 4: MASTER prepares to latch data on [DAT_I()]
MASTER monitors [ACK_I] and prepares to terminate current data phase.
SLAVE expects another transfer and prepares response.

CLOCK, EDGE 4: MASTER latches data on [DAT_I()].
MASTER presents new address on [ADR_O()]
MASTER presents End-Of-Burst on [CTI_O()].
SLAVE presents new data on [DAT_I()].
SLAVE keeps [ACK_I] asserted to indicate that it's ready to send new data.

SETUP, EDGE 5: MASTER prepares to latch data on [DAT_I()]
MASTER monitors [ACK_I] and prepares to terminate current data phase.
SLAVE recognizes End-Of-Burst and prepares to terminate burst.

CLOCK, EDGE 5: MASTER latches data on [DAT_I()].
MASTER negates [CYC_O] and [STB_O] ending burst cycle
SLAVE ends burst by negates [ACK_I]

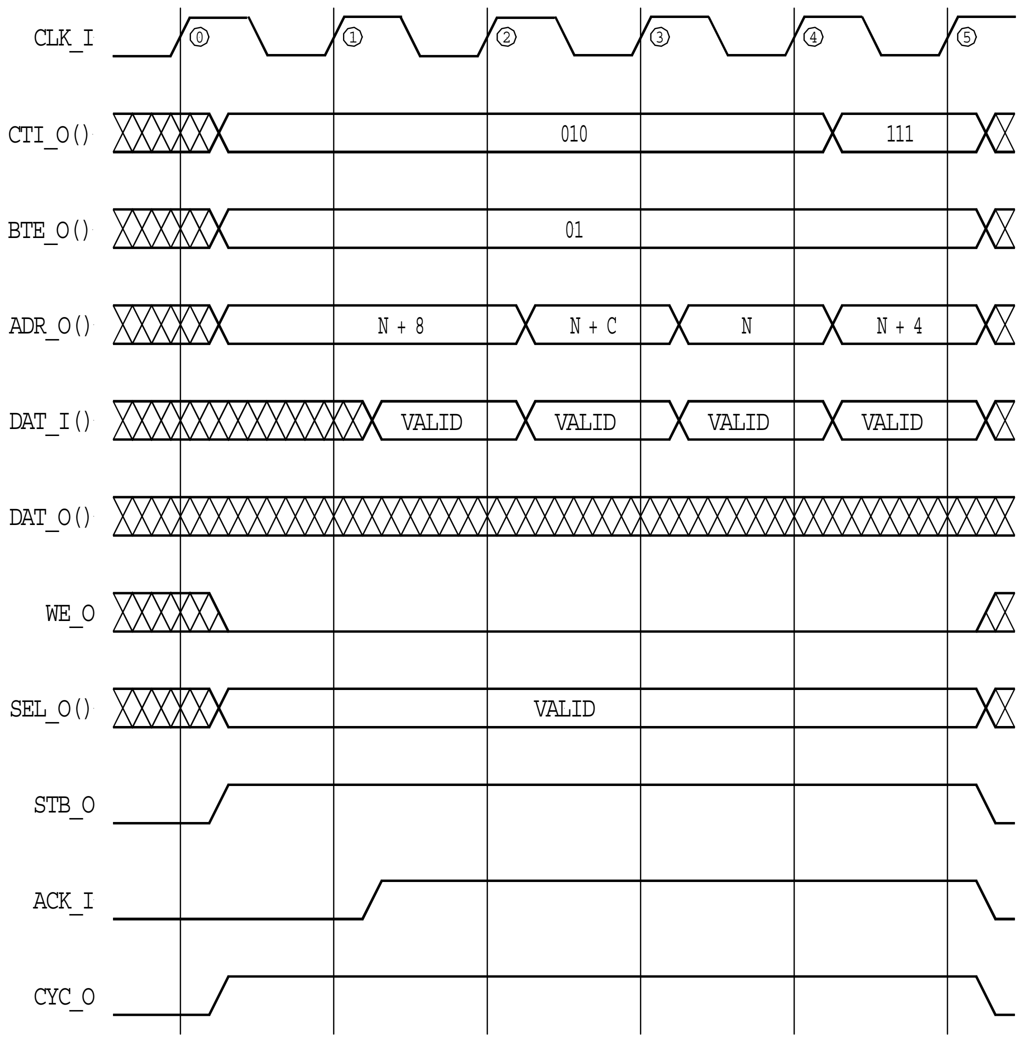


Figure 4-8 4-beat wrapped incrementing burst for a 32bit data array

Chapter 5 – Timing Specification

The WISHBONE specification is designed to provide the end user with very simple timing constraints. Although the application specific circuit(s) will vary in this regard, the interface itself is designed to work without the need for detailed timing specifications. In all cases, the only timing information that is needed by the end user is the maximum clock frequency (for [CLK_I]) that is passed to a place & route tool. The maximum clock frequency is dictated by the time delay between a positive clock edge on [CLK_I] to the setup on a stage further down the logical signal path. This delay is shown graphically in Figure 5-1, and is defined as Tpd,clk-su.

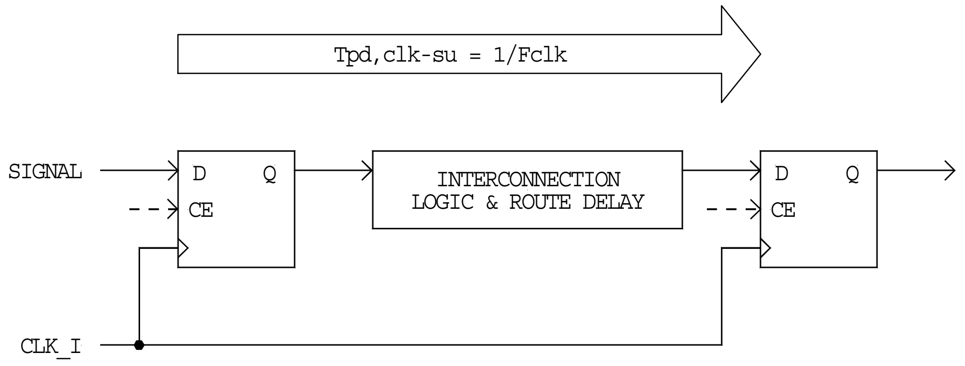


Figure 5-1. Definition for Tpd,clk-su.

RULE 5.00
The clock input [CLK_I] to each IP core MUST coordinate all activities for the internal logic within the WISHBONE interface. All WISHBONE output signals are registered at the rising edge of [CLK_I]. All WISHBONE input signals MUST be stable before the rising edge of [CLK_I].

PERMISSION 5.00
The user's place and route tool MAY be used to enforce RULE 5.00.

OBSERVATION 5.00
Most place and route tools can be easily configured to enforce RULE 5.00. Generally, it only requires a single timing specification for Tpd,clk-su.

RULE 5.05
The WISHBONE interface MUST use synchronous, RTL design methodologies that, given nearly infinitely fast gate delays, will operate over a nearly infinite range of clock frequencies on [CLK_I].

OBSERVATION 5.05
Realistically, the WISHBONE interface will never be expected to operate over a nearly infinite frequency range. However this requirement eliminates the need for non-portable timing constraints (that may work only on certain target devices).

OBSERVATION 5.10
The WISHBONE interface logic assumes that a low-skew clock distribution scheme is used on the target device, and that the clock-skew shall be low enough to permit reliable operation over the environmental conditions.

PERMISSION 5.05
The IP core connected to a WISHBONE interface MAY include application specific timing requirements.

RULE 5.10
The clock input [CLK_I] MUST have a duty cycle that is no less than 40%, and no greater than 60%.

PERMISSION 5.10
The SYSCON module MAY use a variable clock generator. In these cases the clock frequency can be changed by the SYSCON module so long as the clock edges remain clean and monotonic, and if the clock does not violate the duty cycle requirements.

PERMISSION 5.15
The SYSCON module MAY use a gated clock generator. In these cases the clock shall be stopped in the low logic state. When the gated clock is stopped and started the clock edges are required to remain clean and monotonic.

SUGGESTION 5.00
When using a gated clock generator, turn the clock off when the WISHBONE interconnection is not busy. One way of doing this is to create a MASTER interface whose sole purpose is to acquire the WISHBONE interconnection and turn the clock off. This assures that the WISHBONE interconnection is not busy when gating the clock off. When the clock signal is restored the MASTER then releases the WISHBONE interconnection.

OBSERVATION 5.15
This specification does not attempt to govern the design of gated or variable clock generators.

SUGGESTION 5.10
Design an IP core so that all of the circuits (including the WISHBONE interconnect) follow the aforementioned RULEs, as this will make the core portable across a wide range of target devices and technologies.

Chapter 6 – Cited Patent References

This chapter contains a partial list of the patents that have been reviewed by the WISHBONE steward and others. In the opinion of the steward, the WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores does not infringe on any of these patents. However, the possibility exists that an integrated circuit device designed to the WISHBONE specification could infringe on the intellectual property rights of others. The user assumes all responsibility for determining if their WISHBONE design infringes on the rights of others. All of these documents contain information relevant to SoC design and integration. Noteworthy patents are marked with a '(*)'.

This list is maintained for three reasons. First, a public domain specification can't depend on patented ideas (unless permission to use the patent is obtained). This is the list of documents that have been reviewed to see if WISHBONE infringes on any known patents. Second, it provides a starting point for IC designers and other researchers who need to know if a specific SoC design infringes on the rights of others. Third, the patent database is a wonderful place to learn how other people have solved similar SoC problems. Patent documents are very good in this regard, as they must fully describe how to reproduce an invention.

6.1 General Methods Relating to SoC

Hartmann, Alfred C. - US Patent No. 6,096,091
DYNAMICALLY RECONFIGURABLE LOGIC NETWORKS INTERCONNECTED BY FALL-THROUGH FIFOS FOR FLEXIBLE PIPELINE PROCESSING IN A SYSTEM-ON-A-CHIP

Luk et al. - US Patent No. 5,790,839
SYSTEM INTEGRATION OF DRAM MACROS AND LOGIC CORES IN A SINGLE CHIP ARCHITECTURE

Luk et al. - US Patent No. 5,883,814
SYSTEM-ON-CHIP LAYOUT COMPILATION

Wingard et al. - US Patent No. 5,948,089
FULLY-PIPELINED FIXED-LATENCY COMMUNICATIONS SYSTEM WITH A REAL TYPE DYNAMIC BANDWIDTH ALLOCATION.

Wingard et al. - US Patent No. 6,182,183 B1
COMMUNICATION SYSTEM AND METHOD WITH MULTILEVEL CONNECTION IDENTIFICATION

6.2 Methods Relating to SoC Testability

Edwards, et al.- US Patent No. 6,298,394 B1 (*)

SYSTEM AND METHOD FOR CAPTURING INFORMATION ON AN INTERCONNECT IN AN INTEGRATED CIRCUIT

Flynn, David W. - US Patent No. 5,525,971
INTEGRATED CIRCUIT

6.4 Methods Relating to Variable Clock Frequency

Gandhi et al. - US Patent No. 6,185,691 B1
CLOCK GENERATION

Kardach et al. - US Patent No. 5,473,767
METHOD AND APPARATUS FOR ASYNCHRONOUSLY STOPPING THE CLOCK ON A PROCESSOR.

Kardach et al. - US Patent No. 5,918,043
METHOD AND APPARATUS FOR ASYNCHRONOUSLY STOPPING THE CLOCK ON A PROCESSOR.

Maitra, Amit K. - US Patent No. 5,623,647
APPLICATION SPECIFIC CLOCK THROTTLING

Orton et al. - US Patent No. 6,118,306 (*)
CHANGING CLOCK FREQUENCY

Poplingher et al. - US Patent No. 6,173,379 B1
MEMORY DEVICE FOR A MICROPROCESSOR REGISTER FILE HAVING A POWER MANAGEMENT SCHEME AND METHOD FOR COPYING INFORMATION BETWEEN MEMORY SUB-CELLS IN A SINGLE CLOCK CYCLE

Stinson et al. - US Patent No. 6,127,858
METHOD AND APPARATUS FOR VARYING A CLOCK FREQUENCY ON A PHASE BY PHASE BASIS

Thomas, Thomas P. - US Patent No. 6,140,883
TUNABLE, ENERGY EFFICIENT CLOCKING SCHEME

Wong et al. - US Patent No. 5,586,307
METHOD AND APPARATUS SUPPLYING SYNCHRONOUS CLOCK SIGNALS TO CIRCUIT COMPONENTS

Young, Bruce - US Patent No. 6,079,022
METHOD AND APPARATUS FOR DYNAMICALLY ADJUSTING THE CLOCK SPEED OF A BUS DEPENDING ON BUS ACTIVITY

6.5 Methods Relating to Selection of IP Cores

Lee et al. - US Patent No. 6,102,961
METHOD AND APPARATUS FOR SELECTING IP BLOCKS

Methods Relating to Data Flow Architectures
Cismas, Sorin C. - US Patent No. 6,145,073
DATA FLOW INTEGRATED CIRCUIT ARCHITECTURE

6.6 Methods Relating to Crossbar Switch Architectures

Brewer et al. - US Patent No. 5,577,204
PARALLEL PROCESSING COMPUTER SYSTEM INTERCONNECTIONS UTILIZING UNIDIRECTIONAL COMMUNICATION LINKS WITH SEPARATE REQUEST AND RESPONSE LINES FOR DIRECT COMMUNICATION OR USING A CROSSBAR SWITCHING DEVICE

Nelson et al. - US Patent No. 6,138,185
HIGH PERFORMANCE CROSSBAR SWITCH

Van Krevelen et al. - US Patent No. 6,230,229 B1
METHOD AND SYSTEM FOR ARBITRATING PATH CONTENTION IN A CROSSBAR INTERCONNECT NETWORK

Appendix A – WISHBONE Tutorial[4]

By: Wade D. Peterson, Silicore Corporation

The WISHBONE System-on-Chip (SoC) interconnection is a method for connecting digital circuits together to form an integrated circuit 'chip'. This tutorial provides an introduction to the WISHBONE design philosophy and its practical applications.

The WISHBONE architecture solves a very basic problem in integrated circuit design. That is, how to connect circuit functions together in a way that is simple, flexible and portable. The circuit functions are generally provided as 'IP Cores' (Intellectual Property Cores), which system integrators can purchase or make themselves.

Under this topology, IP Cores are the functional building blocks in the system. They are available in a wide variety of functions such as microprocessors, serial ports, disk interfaces, network controllers and so forth. Generally, the IP cores are developed independently from each other and are tied together and tested by a third party system integrator. WISHBONE aides the system integrator by standardizing the IP Core interfaces. This makes it much easier to connect the cores, and therefore much easier to create a custom System-on-Chip.

A.1 An Introduction to WISHBONE

WISHBONE uses a MASTER/SLAVE architecture. That means that functional modules with MASTER interfaces initiate data transactions to participating SLAVE interfaces. As shown in Figure A-1, the MASTERs and SLAVEs communicate through an interconnection interface called the INTERCON. The INTERCON is best thought of as a 'cloud' that contains circuits. These circuits allow MASTERs to communicate with SLAVEs.

The term 'cloud' is borrowed from the telecom community. Oftentimes, telephone systems are modeled as clouds that represent a system of telephone switches and transmission devices. Telephone handsets are connected to the cloud, and are used to make phone calls. The cloud itself represents a network that carries a telephone call from one location to another. The activity going on inside the cloud depends upon where the call is made and where it is going. For example, if a call is made to another office down the hall, then the cloud may represent a local telephone switch located in the same building. However, if the call is made across an ocean, then the cloud may represent a combination of copper, fiber optic and satellite transmission systems.

[4] This tutorial is not part of the WISHBONE specification.

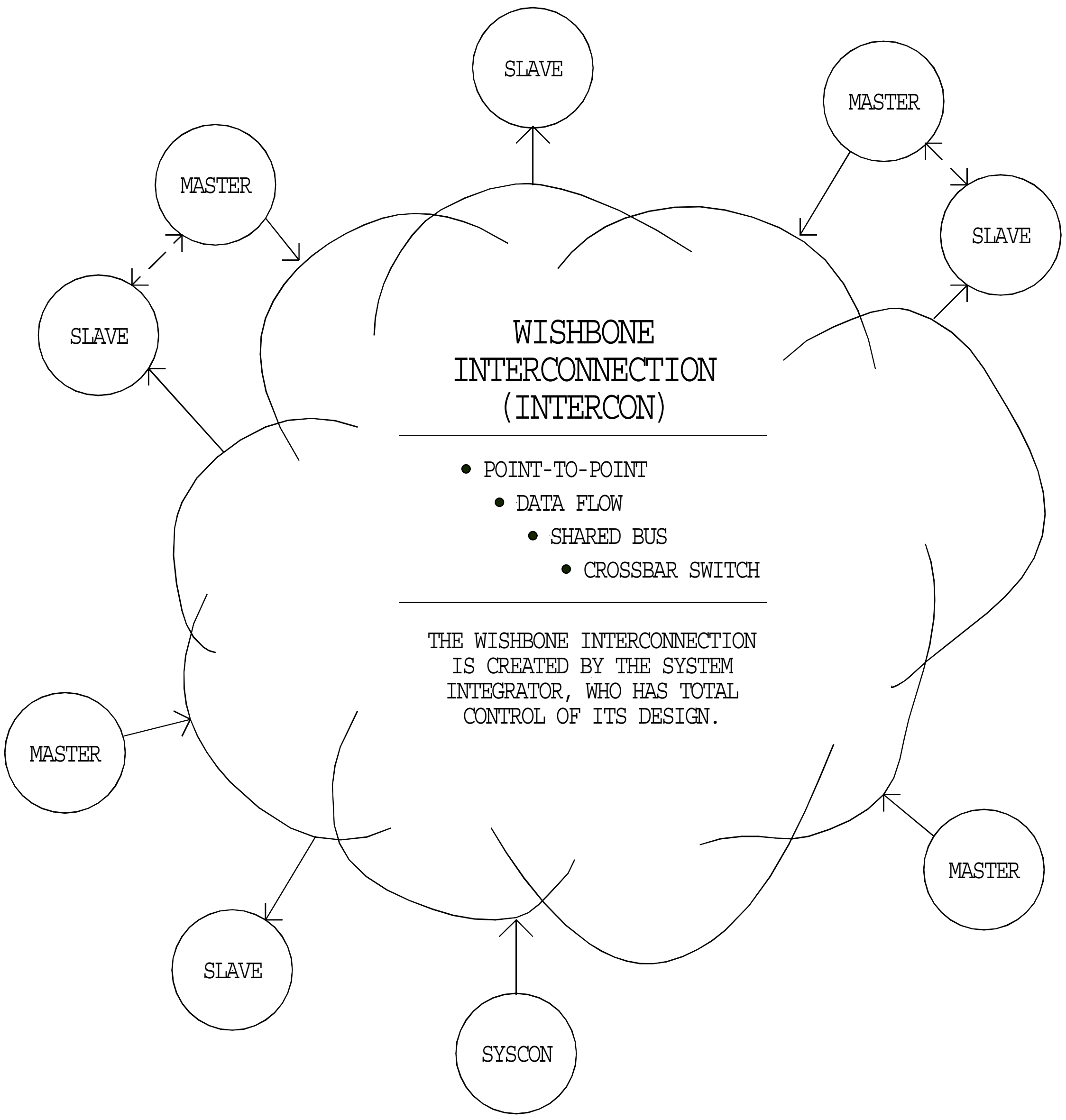

SLAVE
MASTER
MASTER
SLAVE
SLAVE
WISHBONE INTERCONNECTION (INTERCON)
• POINT-TO-POINT
• DATA FLOW
• SHARED BUS
• CROSSBAR SWITCH
THE WISHBONE INTERCONNECTION IS CREATED BY THE SYSTEM INTEGRATOR, WHO HAS TOTAL CONTROL OF ITS DESIGN.
MASTER
MASTER
SLAVE
SYSCON

Figure A-1. The WISHBONE interconnection.

The cloud analogy is used because WISHBONE can be modeled in a similar way. MASTER and SLAVE interfaces (which are analogous to the telephones) communicate thorough an interconnection (which is analogous to the telephone network 'cloud'). The WISHBONE interconnection network can be changed by the system integrator to suit his or her own needs. In WISHBONE terminology this is called a *variable interconnection*.

Variable interconnection allows the system integrator to change the way in which the MASTER and SLAVE interfaces communicate with each other. For example, a pair of MASTER and SLAVE interfaces can communicate with point-to-point, data flow, shared bus or crossbar switch topologies.

The variable interconnection scheme is very different from that used in traditional microcomputer buses such as PCI, cPCI, VMEbus and ISA bus. Those systems use printed circuit backplanes with hardwired connectors. The interfaces on those buses can't be changed, which severely limits how microcomputer boards communicate with each other. WISHBONE overcomes this limitation by allowing the system integrator to change the system interconnection.

This is possible because integrated circuit chips have interconnection paths that can be adjusted. These are very flexible, and take the form of logic gates and routing paths. These can be 'programmed' into the chip using a variety of tools. For example, if the interconnection is described with a hardware description language like VHDL or Verilog®, then the system integrator has the ability to define and adjust the interconnection. Interconnection libraries can also be formed and shared.

The WISHBONE interconnection itself is nothing more than a large, synchronous circuit. It must be designed to *logically* operate over a nearly infinite frequency range. However, every integrated circuit has physical characteristics that limit the maximum frequency of the circuit. In WISHBONE terminology this is called a *variable timing specification*. This means that a WISHBONE compatible circuit will theoretically function normally at any speed, but that it's maximum speed will always be limited by the process technology of the integrated circuit.

At Silicore Corporation we generally define our WISHBONE interconnections using the VHDL hardware description language. These take the form of an ASCII file that contains the VHDL language instructions. This allows us to fully define our interconnections in a way that best fits the application. However, it also allows us to share our interconnections with others, who can adjust them to meet their own needs. It's important to note, though, that there's nothing magic about the interconnection itself. It's just a file, written with off-the-shelf tools, that fully describes the hardware in the interconnection.

A.2 Types of WISHBONE Interconnection

The WISHBONE variable interconnection allows the system integrator to change the way that IP cores connect to each other. There are four defined types of WISHBONE interconnection, and include:

- Point-to-point
- Data flow
- Shared bus
- Crossbar switch

A fifth possible type is the off-chip interconnection. However, off-chip implementations generally fit one of the other four basic types. For example, WISHBONE interfaces on two different integrated circuits can be connected using a point-to-point interconnection.

The WISHBONE specification does not dictate how any of these are implemented by the system integrator. That's because the interconnection itself is a type of IP Core interface called the INTERCON. The system integrator can use or modify an off-the-shelf INTERCON, or create their own.

A.2.1 Point-to-point Interconnection

The point-to-point interconnection is the simplest way to connect two WISHBONE IP cores together. As shown in Figure A-2, the point-to-point interconnection allows a single MASTER interface to connect to a single SLAVE interface. For example, the MASTER interface could be on a microprocessor IP core, and the SLAVE interface could be on a serial I/O port.

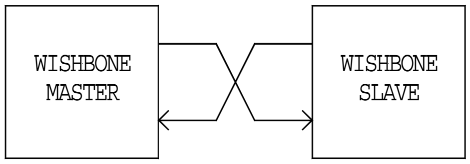


Figure A-2. The point-to-point interconnection.

A.2.2 Data Flow Interconnection

The data flow interconnection is used when data is processed in a sequential manner. As shown in Figure A-3, each IP core in the data flow architecture has both a MASTER and a SLAVE interface. Data flows from core-to-core. Sometimes this process is called *pipelining*.

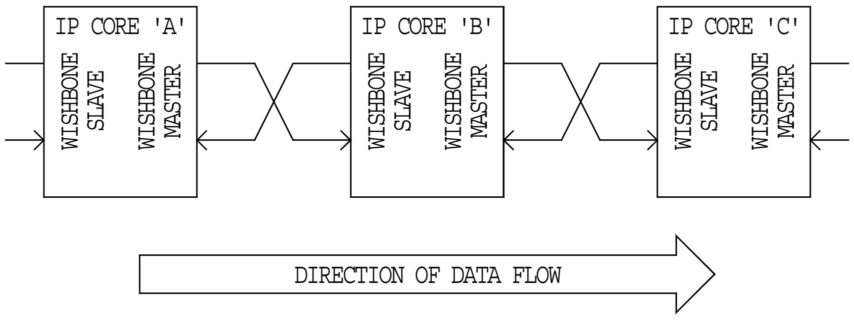


Figure A-3. The data flow interconnection.

The data flow architecture exploits parallelism, thereby speeding up execution time. For example, if each of the IP cores in the Figure represents a floating point processor, then the system has three times the number crunching potential of a single unit. This assumes, of course, that each IP Core takes an equal amount of time to solve its problem, and that the problem can be solved in a sequential manner. In actual practice this may or may not be true, but it does illustrate how the data flow architecture can provide a high degree of parallelism when solving problems.

A.2.3 Shared Bus Interconnection

The shared bus interconnection is useful for connecting two or more MASTERs with one or more SLAVEs. A block diagram is shown in Figure A-4. In this topology a MASTER initiates a bus cycle to a target SLAVE. The target SLAVE then participates in one or more bus cycles with the MASTER.

An arbiter (not shown in the Figure) determines when a MASTER may gain access to the shared bus. The arbiter acts like a 'traffic cop' to determine when and how each MASTER accesses the shared resource. Also, the type of arbiter is completely defined by the system integrator. For example, the shared bus can use a priority or a round robin arbiter. These grant the shared bus on a priority or equal basis, respectively.

The main advantage to this technique is that shared interconnection systems are relatively compact. Generally, it requires fewer logic gates and routing resources than other configurations, especially the crossbar switch. Its main disadvantage is that MASTERs may have to wait before gaining access to the interconnection. This degrades the overall speed at which a MASTER may transfer data.

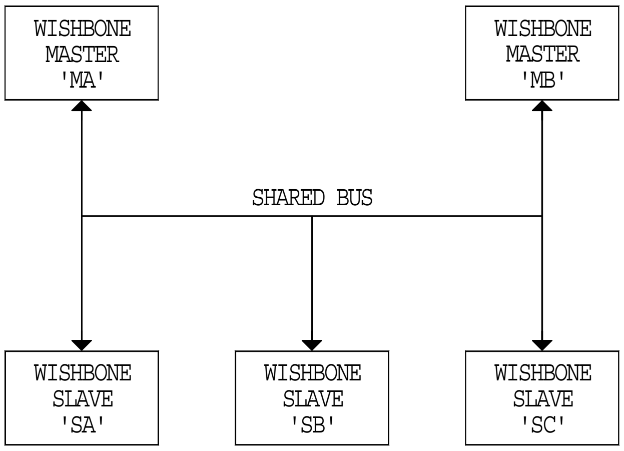


Figure A-4. Shared bus interconnection.

The WISHBONE specification does not dictate how the shared bus is implemented. Later on, we'll see that it can be made either with multiplexer or three-state buses. This gives the system integrator additional flexibility, as some logic chips work better with multiplexor logic, and some work better with three-state buses.

The shared bus is typically found in standard buses like PCI and VMEbus. There, a MASTER interface arbitrates for the common shared bus, and then communicates with a SLAVE. In both cases this is done with three-state buses.

A.2.4 Crossbar Switch Interconnection

The crossbar switch interconnection is used when connecting two or more WISHBONE MASTERs together so that each can access two or more SLAVEs. A block diagram is shown in Figure A-5. In the crossbar interconnection, a MASTER initiates an addressable bus cycle to a target SLAVE. An arbiter (not shown in the diagram) determines when each MASTER may gain access to the indicated SLAVE. Unlike the shared bus interconnection, the crossbar switch allows more than one MASTER to use the interconnection (as long as two MASTERs don't access the same SLAVE at the same time).

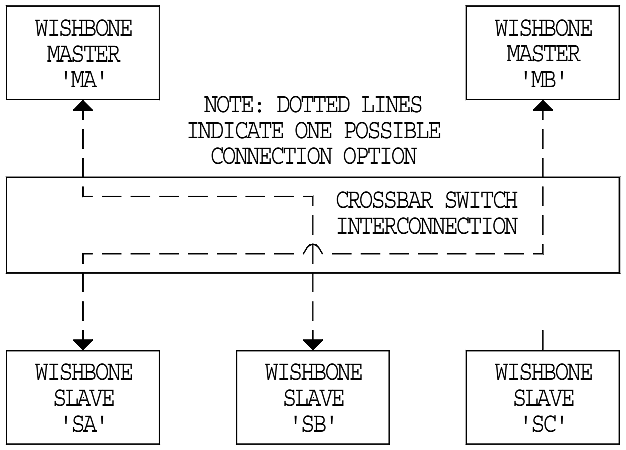


Figure A-5. Crossbar switch interconnection.

Under this method, each master arbitrates for a ‘channel’ on the switch. Once this is established, data is transferred between the MASTER and the SLAVE over a private communication link. The Figure shows two possible channels that may appear on the switch. The first connects MASTER ‘MA’ to SLAVE ‘SB’. The second connects MASTER ‘MB’ to SLAVE ‘SA’.

The overall data transfer rate of the crossbar switch is higher than shared bus mechanisms. For example, the figure shows two MASTER/SLAVE pairs interconnected at the same time. If each communication channel supports a data rate of 100 Mbyte/sec, then the two data pairs would operate in parallel at 200 Mbyte/sec. This scheme can be expanded to support extremely high data transfer rates.

One disadvantage of the crossbar switch is that it requires more interconnection logic and routing resources than shared bus systems. As a rule-of-thumb, a crossbar switch with two MASTERs and two SLAVEs takes twice as much interconnection logic as a similar shared bus system (with two MASTERs and two SLAVEs).

The crossbar interconnection is a typical configuration that one might find in microcomputer buses like[5] RACEway, SKY Channel or Myrinet.

[5] Raceway: ANSI/VITA 5-1994. SKYchannel: ANSI/VITA 10-1995. Myrinet: ANSI/VITA 26-1998. For more information about these standards see www.vita.com.

A.3 The WISHBONE Interface Signals

WISHBONE MASTER and SLAVE interfaces can be connected together in a number of ways. This requires that WISHBONE interface signals and bus cycles be designed in a very flexible and reusable manner. The signals were defined with the following requirements:

- The signals allow MASTER and SLAVE interfaces to support point-to-point, data flow, shared bus and crossbar switch interconnections.

- The signals allow three basic types of bus cycle. These include SINGLE READ/WRITE, BLOCK READ/WRITE and RMW (read-modify-write) bus cycles. The operation of these bus cycles are described below.

- A handshaking mechanism is used so that either the MASTER or the participating SLAVE interface can adjust the data transfer rate during a bus cycle. This allows the speed of each bus cycle (or phase) to be adjusted by either the MASTER or the SLAVE interface. This means that all WISHBONE bus cycles run at the speed of the slowest interface.

- The handshaking mechanism allows a participating SLAVE to accept a data transfer, reject a data transfer with an error or ask the MASTER to retry a bus cycle. The SLAVE does this by generating the [ACK_O], [ERR_O] or [RTY_O] signals respectively. Every interface must support the [ACK_O] signal, but the error and retry acknowledgement signals are optional.

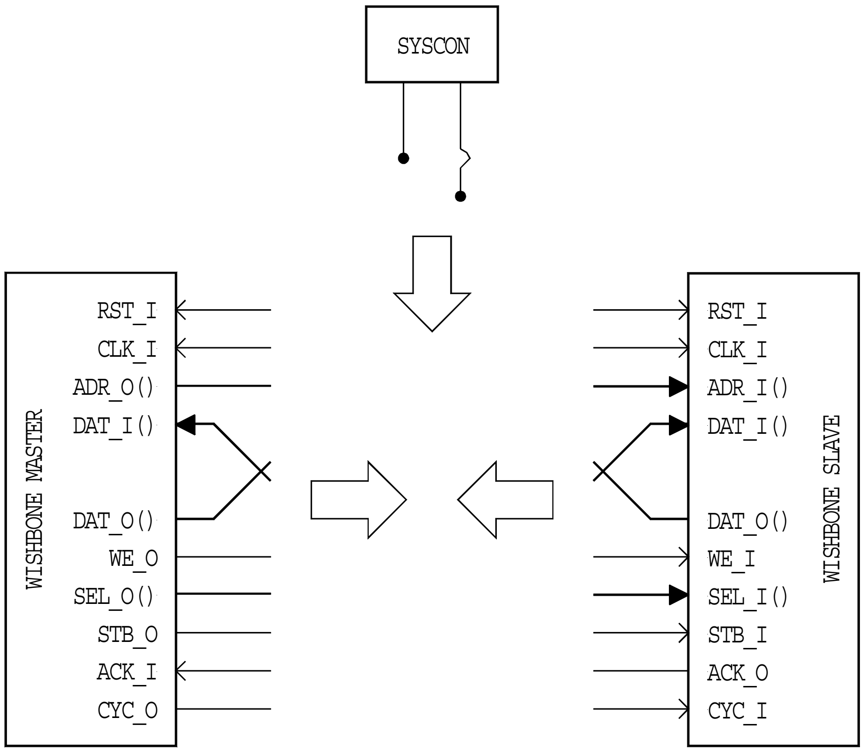
- All signals on MASTER and SLAVE interfaces are either inputs or outputs, but are never bi-directional (i.e. three-state). This is because some FPGA and ASIC devices do not support bi-directional signals. However, it is permissible (and sometimes advantageous) to use bi-directional signals in the interconnection logic if the target device supports it.

- Address and data bus widths can be altered to fit the application. 8, 16, 32 and 64-bit data buses, and 0-64 bit address buses are defined.

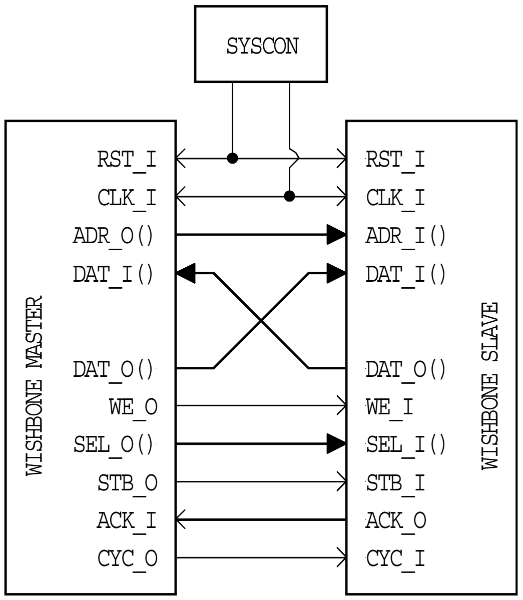
- As shown in Figure A-6, all signals are arranged so that MASTER and SLAVE interfaces can be connected directly together to form a simple point-to-point interface. This allows very compact and efficient WISHBONE interfaces to be built. For example, WISHBONE could be used as the external system bus on a microprocessor IP Core. However, it's efficient enough so that it can be used for internal buses inside of the microprocessor.

- User defined signals in the form of 'tags' are allowed. This allows the system integrator to add special purpose signals to each WISHBONE interface. For example, the system integrator could add a parity bit to the address or data buses.

A comprehensive list of the WISHBONE signals and their descriptions is given in the specification.



(A) FORMING A POINT-TO-POINT INTERCONNECTION.



(B) POINT-TO-POINT INTERCONNECTION.

Figure A-6. The WISHBONE signals are selected to permit MASTER and SLAVE interfaces to be directly connected, thereby forming a simple point-to-point interface.

A.4 The WISHBONE Bus Cycles

There are three types of defined WISHBONE bus cycles. They include:

- SINGLE READ/WRITE
- BLOCK READ/WRITE
- READ MODIFY WRITE (RMW)

A.4.1 SINGLE READ/WRITE Cycle

The SINGLE READ/WRITE is the most basic WISHBONE bus cycle. As the name implies, it is used to transfer a single data operand. Figure A-7 shows a typical SINGLE READ cycle.

The WISHBONE specification shows all bus cycle timing diagrams as if the MASTER and SLAVE interfaces were connected in a point-to-point configuration. They also show all of the signals on the MASTER side of the interface. This provides a standard way of describing the interface without having to describe the whole system. For example, the Figure shows a signal called [ACK_I], which is an input to a MASTER interface. In this configuration it is directly connected to [ACK_O], which is driven by the SLAVE. If the timing diagram were shown from the perspective of the SLAVE, then the [ACK_O] signal would have been shown. The SINGLE READ cycle operates thusly:

1. In response to clock edge 0, the MASTER interface asserts [ADR_O()], [WE_O], [SEL_O], [STB_O] and [CYC_O].

2. The SLAVE decodes the bus cycle by monitoring its [STB_I] and address inputs, and presents valid data on its [DAT_O()] lines. Because the system is in a point-to-point configuration, the SLAVE [DAT_O()] signals are connected to the MASTER [DAT_I()] signals.

3. The SLAVE indicates that it has placed valid data on the data bus by asserting the MASTER's [ACK_I] acknowledge signal. Also note that the SLAVE may delay its response by inserting one or more wait states. In this case, the SLAVE does not assert the acknowledge line. The possibility of a wait state in the timing diagrams is indicated by '-WSS-'.

4. The MASTER monitors the state of its [ACK_I] line, and determines that the SLAVE has acknowledged the transfer at clock edge 1.

5. The MASTER latches [DAT_I()] and negates its [STB_O] signal in response to [ACK_I].

The SINGLE WRITE cycle operates in a similar manner, except that the MASTER asserts [WE_O] and places data on [DAT_O]. In this case the SLAVE asserts [ACK_O] when it has latched the data.

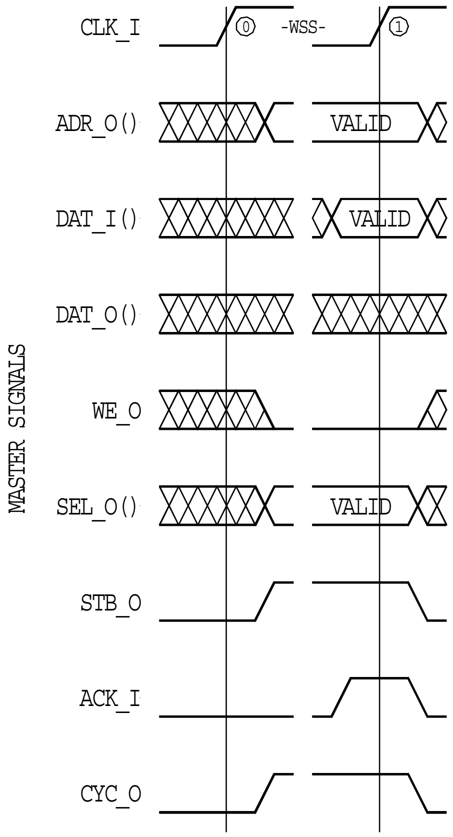


Figure A-7. SINGLE READ cycle.

A.4.2 BLOCK READ/WRITE Cycle

The BLOCK READ/WRITE cycles are very similar to the SINGLE READ/WRITE cycles. The BLOCK cycles can be thought of as two or more back-to-back SINGLE cycles strung together. In WISHBONE terminology the term *cycle* refers to the whole BLOCK cycle. The small, individual data transfers that make up the BLOCK cycle are called *phases*.

The starting and stopping point of the BLOCK cycles are identified by the assertion and negation of the MASTER [CYC_O] signal (respectively). The [CYC_O] signal is also used in shared bus and crossbar interconnections because it informs system logic that the MASTER wishes to use the bus. It also informs the interconnection when it is through with its bus cycle.

A.4.3 READ-MODIFY-WRITE (RMW) Cycle

The READ-MODIFY-WRITE cycle is used in multiprocessor and multitasking systems. This special cycle allows multiple software processes to share common resources by using semaphores. This is commonly done on interfaces for disk controllers, serial ports and memory. As the name implies, the READ-MODIFY-WRITE cycle reads and writes data to a memory location in a single bus cycle. It prevents the allocation of a common resource to two or more processes. The READ-MODIFY-WRITE cycle can also be thought of as an *indivisible cycle*.

The read portion of the cycle is called the *read phase*, and the write portion is called *the write phase*. When looking at the timing diagram of this bus cycle, it can be thought of as a two phase BLOCK cycle where the first phase is a read and the second phase is a write.

The READ-MODIFY-WRITE cycle is also known as an indivisible cycle because it is designed for multiprocessor systems. WISHBONE shared bus interconnections must be designed so that once an arbiter grants the bus to a first MASTER, it will not grant the bus to a second MASTER until the first MASTER gives up the bus. This allows a single MASTER (such as a microprocessor) to read some data, modify it and then write it back…all in a single, contiguous bus cycle. If the arbiter were allowed to change MASTERs in the middle of the cycle, then the two processors could incorrectly interpret the semaphore. The arbiter does this by monitoring the [CYC_O] cycle from each MASTER on the interconnection. The problem is averted because the [CYC_O] signal is always asserted for the duration of the RMW cycle.

To illustrate this point, consider a two processor system with a single disk controller. In this case each processor has a MASTER interface, and the disk controller has a SLAVE interface. Oftentimes, these systems require that only one processor access the disk at any given time[6]. To satisfy this requirement, a semaphore bit somewhere in memory is assigned to act as a 'traffic cop' between the two processors. If the bit is cleared, then the disk is available for use. If it's set, then the disk controller is busy.

[6] This is a common requirement to prevent one form of disk 'thrashing'. In this case, if both processors were allowed to access the disk during the same time interval, then one processor could request data from one sector of the disk while the other requested data from another sector. This could cause a situation where the disk head is constantly moved between the two locations, thereby degrading its performance or causing it to fail altogether.

Now consider a system where the two processors both need to use the disk. We'll call them processor #0 and processor #1. In order for processor #0 to acquire the disk it first reads and stores the state of the semaphore bit, and then sets the bit by writing back to memory. The reading and setting of the bit takes place inside of a single RMW cycle.

Once the processor is done with the semaphore operation, it checks the state of the bit it read during the first phase of the RMW cycle. If the bit is clear it goes ahead and uses the disk controller. If the other processor attempts to use the disk controller at this time, it reads a '1' from the semaphore, thereby preventing it from accessing the disk controller. When the first processor (#0) is done with the disk controller, it simply clears the semaphore bit by writing a '0' to it. This allows the other processor to gain access to the controller the next time it checks the semaphore.

Now consider the same situation, except where the semaphore is set and cleared using a SINGLE READ cycle followed by a SINGLE WRITE cycle. In this case it is possible for both processors to gain access to the disk controller at the same time…a situation that would crash the system. That's because the arbiter can grant the bus in the following order:

- Processor #0 reads '0' from the semaphore bit.
- Processor #1 reads '0' from the semaphore bit.
- Processor #0 writes '1' to the semaphore bit.
- Processor #1 writes '1' to the semaphore bit.

This leads to a system crash because both processors read a '0' from the semaphore bit, thereby causing both to access the disk controller.

It is important to note that a processor (or other device connected to the MASTER interface) must support the RMW cycle in order for this to be effective. This is generally done with special instructions that force a RMW bus cycle. Not all processors do this. A good example is the 680XX family of microprocessors. These use special compare-and-set (CAS) and test-and-set (TAS) instructions to generate RMW cycles, and to do the semaphore operations.

A.5 Endian

The WISHBONE specification regulates the ordering of data. This is because data can be presented in two different ways. In the first way, the most significant byte of an operand is placed at the higher (bigger) address. In the second way, the most significant byte of an operand can be placed at the lower (smaller) address. These are called BIG ENDIAN and LITTLE ENDIAN data operands, respectively. WISHBONE supports both types.

ENDIAN becomes an issue when the granularity of a WISHBONE port is smaller than the operand size. For example, a 32-bit port can have an 8-bit (BYTE wide) granularity. This results in a fairly ambiguous situation where the most significant byte of the 32-bit operand could be placed at the higher or lower byte address of the port. However, ENDIAN is not an issue when the granularity and port size are the same.

The system integrator may wish to connect a BIG ENDIAN interface to a LITTLE ENDIAN inteface. In many cases the conversion is quite straightforward, and does not require any exotic conversion logic. Furthermore, the conversion does not create any speed degradation in the interface. In general, the ENDIAN conversion takes place by renaming the data and select I/O signals at a MASTER or SLAVE interface.

Figure A-8 shows a simple example where a 32-bit BIG ENDIAN MASTER output (CORE 'A') is connected to a 32-bit LITTLE ENDIAN SLAVE input (CORE 'B'). Both interfaces have 32-bit operand sizes and 8-bit granularities. As can be seen in the diagram, the ENDIAN conversion is accomplished by cross coupling the data and select signal arrays. This is quite simple since the conversion is accomplished at the interconnection level, or using a wrapper. This is especially simple in soft IP cores using VHDL or Verilog® hardware description languages, as it only requires the renaming of signals.

In some cases the address lines may also need to be modified between the two cores. For example, if 64-bit operands are transferred between two cores with 8-bit port sizes, then the address lines may need to be modified as well.

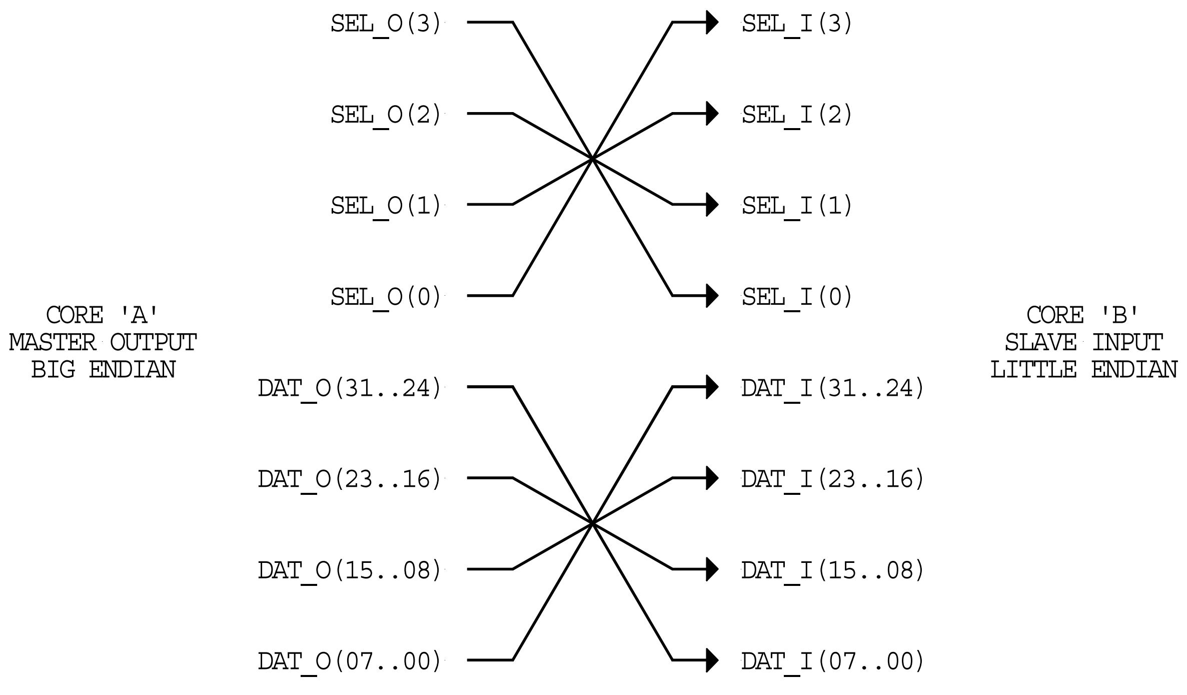


Figure A-8. BIG ENDIAN to LITTLE ENDIAN conversion example.

A.6 SLAVE I/O Port Examples

In this section we'll investigate several examples of WISHBONE interface for SLAVE I/O ports. Our purpose is to:

- Show some simple examples of how the WISHBONE interface operates.
- Demonstrate how simple interfaces work in conjunction with standard logic primitives on FPGA and ASIC devices. This also means that very little logic is needed to implement the WISHBONE interface.
- Demonstrate the concept of *granularity*.
- Provide some portable design examples.
- Give examples of the WISHBONE DATASHEET.
- Show VHDL implementation examples.

A.6.1 Simple 8-bit SLAVE Output Port

Figure A-9 shows a simple 8-bit WISHBONE SLAVE output port. The entire interface is implemented with a standard 8-bit 'D-type' flip-flop register (with synchronous reset) and a single AND gate. During write cycles, data is presented at the data input bus [DAT_I(7..0)], and is latched at the rising edge of [CLK_I] when [STB_I] and [WE_I] are both asserted.

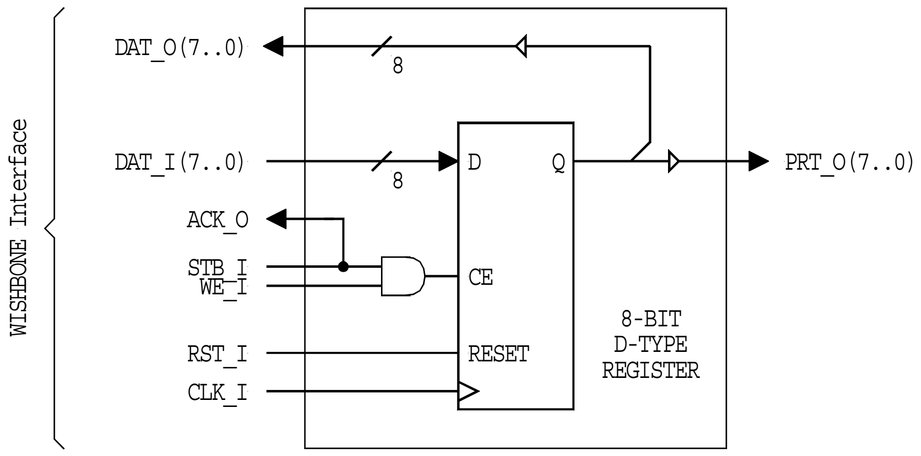


Figure A-9. Simple 8-bit WISHBONE SLAVE output port.

The state of the output port can be monitored by a MASTER by routing the output data lines back to [DAT_O(7..0)]. During read cycles the AND gate prevents erroneous data from being latched into the register.

This circuit is highly portable, as all FPGA and ASIC target devices support D-type flip-flops with clock enable and synchronous reset inputs.

The circuit also demonstrates how the WISHBONE interface requires little or no logic overhead. In this case, the WISHBONE interface does not require any extra logic gates whatsoever. This is because WISHBONE is designed to work in conjunction with standard, synchronous and combinatorial logic primitives that are available on most FPGA and ASIC devices.

The WISHBONE specification requires that the interface be documented. This is done in the form of the WISHBONE DATASHEET. The standard does not specify the form of the datasheet. For example, it can be part of a comment field in a VHDL or Verilog® source file or part of a technical reference manual for the IP core. Table A-1 shows one form of the WISHBONE DATASHEET for the 8-bit output port circuit.

The purpose of the WISHBONE DATASHEET is to promote design reuse. It forces the originator of the IP core to document how the interface operates. This makes it easier for another person to re-use the core.

Table A-1. WISHBONE DATASHEET for the 8-bit output port example.

Description	Specification	
General description:	8-bit SLAVE output port.	
Supported cycles:	SLAVE, READ/WRITE SLAVE, BLOCK READ/WRITE SLAVE, RMW	
Data port, size: Data port, granularity: Data port, maximum operand size: Data transfer ordering: Data transfer sequencing:	8-bit 8-bit 8-bit Big endian and/or little endian Undefined	
Supported signal list and cross reference to equivalent WISHBONE signals:	Signal Name	WISHBONE Equiv.
	ACK_O	ACK_O
	CLK_I	CLK_I
	DAT_I(7..0)	DAT_I()
	DAT_O(7..0)	DAT_O()
	RST_I	RST_I
	STB_I	STB_I
	WE_I	WE_I

Figure A-10 shows a VHDL implementation of same circuit. The WBOPRT08 entity implements the all of the functions shown in the schematic diagram of Figure A-9.

```
library ieee;
use ieee.std_logic_1164.all;

entity WBOPRT08 is
port(
      -- WISHBONE SLAVE interface:

      ACK_O:      out   std_logic;
      CLK_I:      in    std_logic;
      DAT_I:      in    std_logic_vector( 7 downto 0 );
      DAT_O:      out   std_logic_vector( 7 downto 0 );
      RST_I:      in    std_logic;
      STB_I:      in    std_logic;
      WE_I:       in    std_logic;

      -- Output port (non-WISHBONE signals):

      PRT_O:      out   std_logic_vector( 7 downto 0 )

    );
end entity WBOPRT08;

architecture WBOPRT081 of WBOPRT08 is

    signal  Q: std_logic_vector( 7 downto 0 );

begin

    REG: process( CLK_I )
    begin

        if( rising_edge( CLK_I ) ) then

            if( RST_I = '1' ) then
                Q <= B"00000000";
            elsif( (STB_I and WE_I) = '1' ) then
                Q <= DAT_I( 7 downto 0 );
            else
                Q <= Q;
            end if;

        end if;

    end process REG;

    ACK_O <= STB_I;
    DAT_O <= Q;
    PRT_O <= Q;

end architecture WBOPRT081;
```

Figure A-10. VHDL implementation of the 8-bit output port interface.

A.6.2 Simple 16-bit SLAVE Output Port With 16-bit Granularity

Figure A-11 shows a simple 16-bit WISHBONE SLAVE output port. Table A-2 shows the WISHBONE DATASHEET for this design. It is identical to the 8-bit port shown earlier, except that the data bus is wider. Also, this port has 16-bit granularity. In the next section, it will be compared to a 16-bit port with 8-bit granularity.

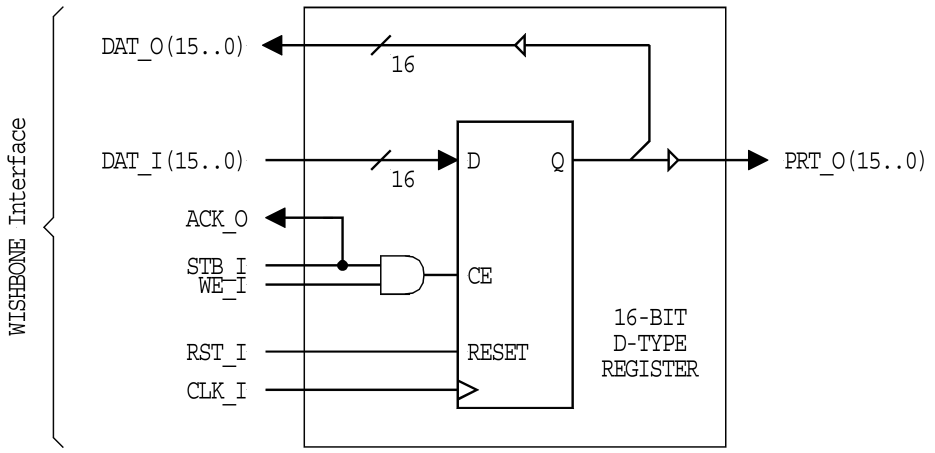


Figure A-11. Simple 16-bit WISHBONE SLAVE output port with 16-bit granularity

Table A-2. WISHBONE DATASHEET for the 16-bit output port with 16-bit granularity.	
Description	Specification
General description:	16-bit SLAVE output port.
Supported cycles:	SLAVE, READ/WRITE SLAVE, BLOCK READ/WRITE SLAVE, RMW
Data port, size: Data port, granularity: Data port, maximum operend size: Data transfer ordering: Data transfer sequencing:	16-bit 16-bit 16-bit Big endian and/or little endian Undefined
Supported signal list and cross reference to equivalent WISHBONE signals:	Signal Name — WISHBONE Equiv. ACK_O — ACK_O CLK_I — CLK_I DAT_I(15..0) — DAT_I() DAT_O(15..0) — DAT_O() RST_I — RST_I STB_I — STB_I WE_I — WE_I

A.6.3 Simple 16-bit SLAVE Output Port With 8-bit Granularity

Figure A-12 shows a simple 16-bit WISHBONE SLAVE output port. This port has 8-bit granularity, which means that data can be transferred 8 or 16-bits at a time.

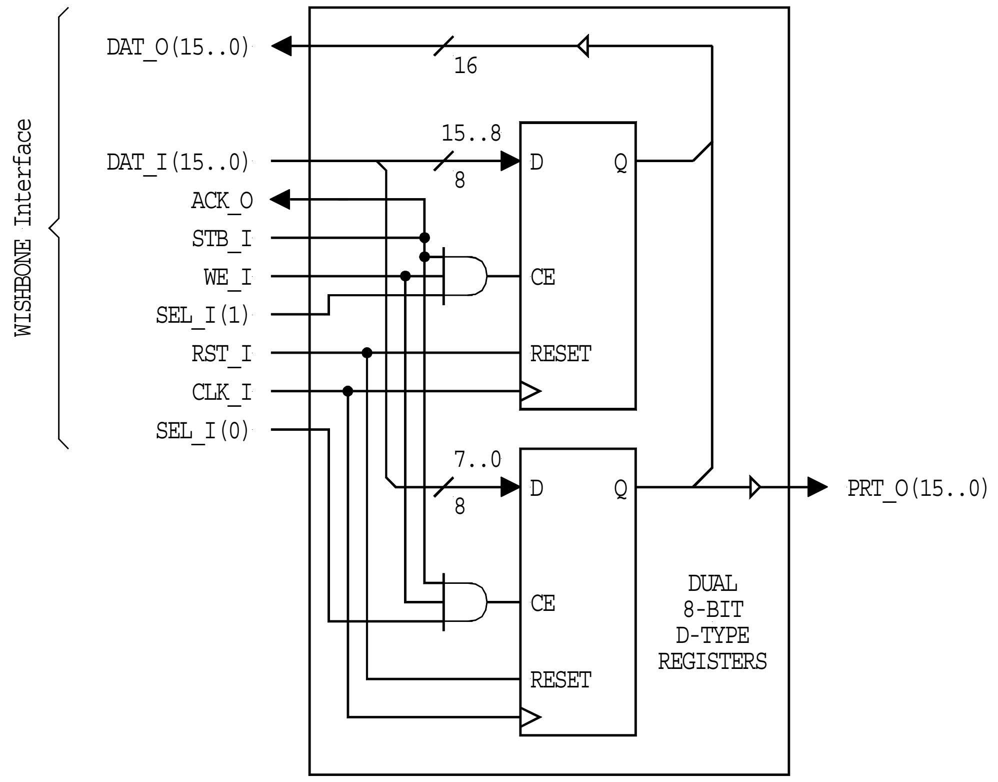


Figure A-12. Simple 16-bit WISHBONE SLAVE output port with 8-bit granularity.

This circuit differs from the aforementioned 16-bit port because it has 8-bit granularity. This means that the 16-bit register can be accessed with either 8 or 16-bit bus cycles. This is accomplished by selecting the high or low byte of data with the select lines [SEL_I(1..0)]. When [SEL_I(0)] is asserted, the low byte is accessed. When [SEL_I(1)] is asserted, the high byte is accessed. When both are asserted, the entire 16-bit word is accessed.

The circuit also demonstrates the proper use of the [STB_I] and [SEL_I()] lines for slave devices. The [STB_I] signal operates much like a chip select signal, where the interface is selected when [STB_I] is asserted. The [SEL_I()] lines are only used to determine where data is placed by the MASTER or SLAVE during read and write cycles.

In general, the [SEL_I()] signals should never be used by the SLAVE to determine when the IP core is being accessed by a master. They should only be used to determine where data is placed on the data input and output buses. Stated another way, the MASTER will assert the select signals [SEL_O()] during every bus cycle, but a particular slave is only accessed when it monitors that its [STB_I] input is asserted. Stated another way, the [STB_I] signal is generated by address decode logic within the WISHBONE interconnect, but the [SEL_I()] signals may be broadcasted to all SLAVE devices.

Table A-3 shows the WISHBONE DATASHEET for this IP core. This is very similar to the 16-bit data port with 16-bit granularity, except that the granularity has been changed to 8-bits.

It should also be noted that the datasheet specifies that the circuit will work with READ/WRITE, BLOCK READ/WRITE and RMW cycles. This means that the circuit will operate normally when presented with these cycles. It is left as an exercise for the user to verify that the circuit will indeed work with all three of these cycles.

Table A-3. WISHBONE DATASHEET for the 16-bit output port with 8-bit granularity.

Description	Specification	
General description:	16-bit SLAVE output port with 8-bit granularity.	
Supported cycles:	SLAVE, READ/WRITE SLAVE, BLOCK READ/WRITE SLAVE, RMW	
Data port, size: Data port, granularity: Data port, maximum operand size: Data transfer ordering: Data transfer sequencing:	16-bit 8-bit 16-bit Big endian and/or little endian Undefined	
Supported signal list and cross reference to equivalent WISHBONE signals:	Signal Name	WISHBONE Equiv.
	ACK_O	ACK_O
	CLK_I	CLK_I
	DAT_I(15..0)	DAT_I()
	DAT_O(15..0)	DAT_O()
	RST_I	RST_I
	STB_I	STB_I
	WE_I	WE_I

Figure A-13 shows a VHDL implementation of same circuit. The WBOPRT16 entity implements the all of the functions shown in the schematic diagram of Figure A-12.

```
entity WBOPRT16 is
port(
      -- WISHBONE SLAVE interface:
      ACK_O:      out   std_logic;
      CLK_I:      in    std_logic;
      DAT_I:      in    std_logic_vector( 15 downto 0 );
      DAT_O:      out   std_logic_vector( 15 downto 0 );
      RST_I:      in    std_logic;
      SEL_I:      in    std_logic_vector(  1 downto 0 );
      STB_I:      in    std_logic;
      WE_I:       in    std_logic;

      -- Output port (non-WISHBONE signals):
      PRT_O:      out   std_logic_vector( 15 downto 0 )

    );
end entity WBOPRT16;

architecture WBOPRT161 of WBOPRT16 is
    signal  QH: std_logic_vector( 7 downto 0 );
    signal  QL: std_logic_vector( 7 downto 0 );
begin

    REG: process( CLK_I )
    begin
        if( rising_edge( CLK_I ) ) then
            if( RST_I = '1' ) then
                QH <= B"00000000";
            elsif( (STB_I and WE_I and SEL_I(1)) = '1' ) then
                QH <= DAT_I( 15 downto 8 );
            else
                QH <= QH;
            end if;
        end if;

        if( rising_edge( CLK_I ) ) then
            if( RST_I = '1' ) then
                QL <= B"00000000";
            elsif( (STB_I and WE_I and SEL_I(0)) = '1' ) then
                QL <= DAT_I( 7 downto 0 );
            else
                QL <= QL;
            end if;
        end if;

    end process REG;

    ACK_O <= STB_I;
    DAT_O( 15 downto 8 ) <= QH;
    DAT_O(  7 downto 0 ) <= QL;
    PRT_O( 15 downto 8 ) <= QH;
    PRT_O(  7 downto 0 ) <= QL;

end architecture WBOPRT161;
```

Figure A-13. VHDL implementation of the 16-bit output port with 8-bit granularity.

A.7 WISHBONE Memory Interfacing

In this section we'll examine WISHBONE memory interfacing and present some examples. The purpose of this section is to:

- Introduce the FASM synchronous RAM and ROM models.
- Show a simple example of how the WISHBONE interface operates.
- Demonstrate how simple interfaces work in conjunction with standard logic primitives on FPGA and ASIC devices. This also means that very little logic (if any) is needed to implement the WISHBONE interface.
- Present a WISHBONE DATASHEET example for a memory element.
- Describe portability issues with regard to FPGA and ASIC memory elements.

A.7.1 FASM Synchronous RAM and ROM Model

The WISHBONE interface can be connected to any type of RAM or ROM memory. However, some types will be faster and more efficient than others. If the memory interface closely resembles the WISHBONE interface, then everything will run very fast. If the memory is significantly different than WISHBONE, then everything will slow down. This is such a fundamental and important issue that both the WISHBONE interface and its bus cycles were designed around the most common memory interface that could be found.

This was very problematic in the original WISHBONE concept. That's because there are very few portable RAM and ROM types used across all both FPGA and ASIC devices. In fact, the most common memory type that could be found are what we call 'FASM', or the FPGA and ASIC Subset Model[7].

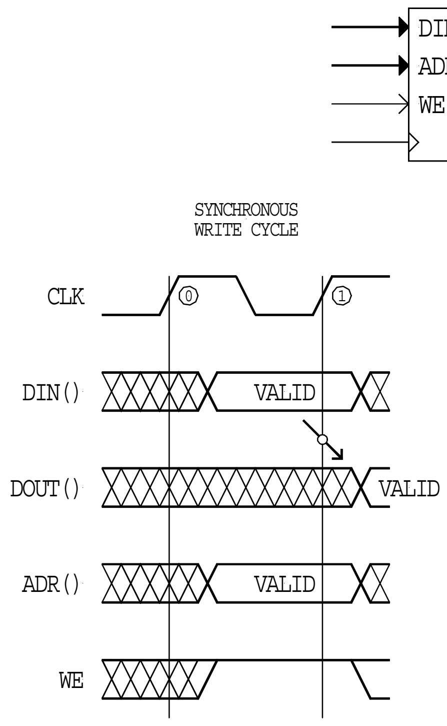
The FASM synchronous RAM model conforms to the connection and timing diagram shown in Figure A-14. The WISHBONE bus cycles all are designed to interface directly to this type of RAM. During write cycles, FASM Synchronous RAM stores input data at the indicated address whenever: (a) the write enable (WE) input is asserted, and (b) there is a rising clock edge.

During read cycles, FASM Synchronous RAM works like an asynchronous ROM. Data is fetched from the address indicated by the ADR() inputs, and is presented at the data output (DOUT). The clock input is ignored. However, during write cycles, the output data is updated immediately during a write cycle.

A good exercise for the user is to compare the FASM Synchronous RAM cycles to the WISHBONE SINGLE READ/WRITE, BLOCK READ/WRITE and READ-MODIFY-WRITE cycles. You will find that these three bus cycles operate in an identical fashion to the FASM Synchronous RAM model. They are so close, in fact, that FASM RAMs can be interfaced to WISHBONE with as little as one NAND gate.

[7] The original FASM model actually encompasses many type of devices, but in this tutorial we'll focus only on the FASM synchronous RAM and ROM models.

While most FPGA and ASIC devices will provide RAM that follows the FASM guidelines, you will probably find that most devices also support other types of memories as well. For example, in some brands of FPGA you will find block memories that use a different interface. Some of these will still interface very smoothly to WISHBONE, while others will introduce a wait-state. In all cases that we found, all FPGAs and most ASICs did support at least one style of FASM memory.



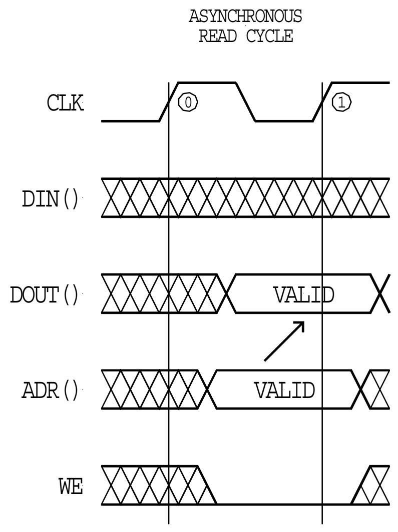


Figure A-14. Generic FASM synchronous RAM connection and timing diagram.

The FASM ROM closely resembles the FASM RAM during its read cycle. FASM ROM conforms to the connection and timing diagram shown in Figure A-15.

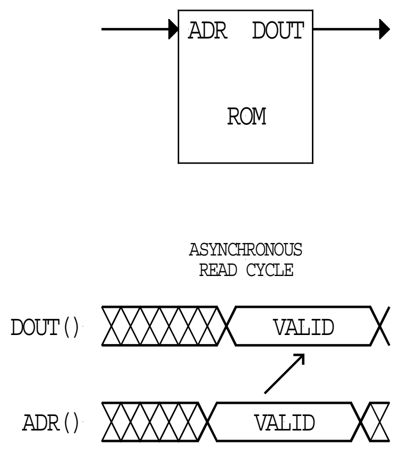


Figure A-15. FASM asynchronous ROM connection and timing diagram.

A.7.2 Simple 16 x 8-bit SLAVE Memory Interface

Figure A-16 shows a simple 8-bit WISHBONE memory. The 16 x 8-bit memory is formed from two 16 x 4-bit FASM compatible synchronous memories. Besides the memory elements, the entire interface is implemented with a single AND gate. During write cycles, data is presented at the data input bus [DAT_I(7..0)], and is latched at the rising edge of [CLK_I] when [STB_I] and [WE_I] are both asserted. During read cycles, the memory output data (DO) is made available at the data output port [DAT_O(7..0)].

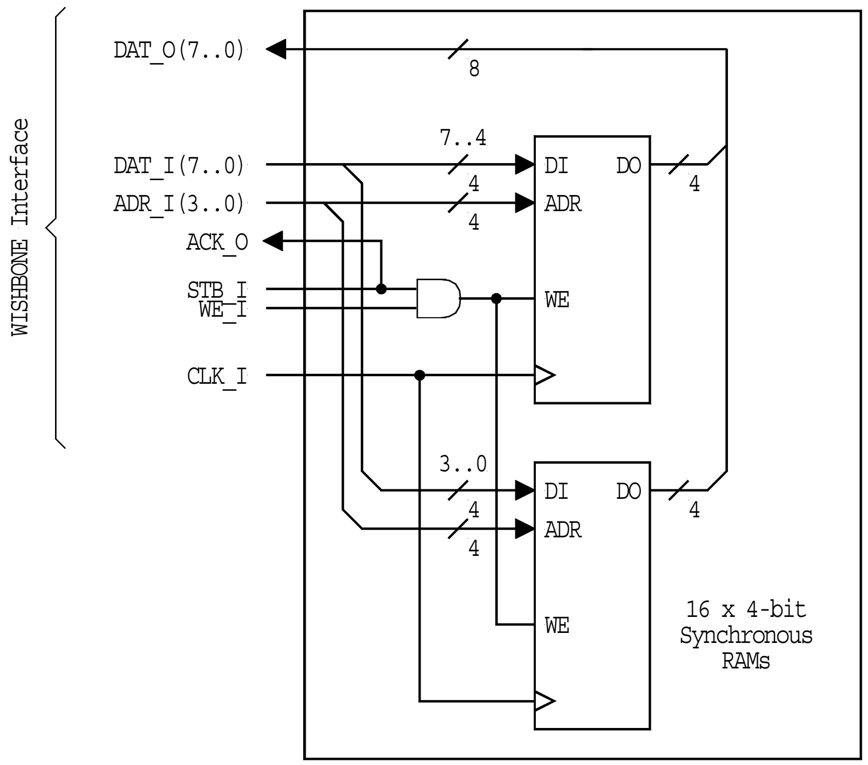


Figure A-16. Simple 16 x 8-bit SLAVE memory.

The memory circuit does not have a reset input. That's because most RAM memories do not have a reset capability.

The circuit also demonstrates how the WISHBONE interface requires little or no logic overhead. In this case, the WISHBONE interface requires a single AND gate. This is because WISHBONE is designed to work in conjunction with standard, synchronous and combinatorial logic primitives that are available on most FPGA and ASIC devices.

The WISHBONE specification requires that the interface be documented. This is done in the form of the WISHBONE DATASHEET. The standard does not specify the form of the datasheet. For example, it can be part of a comment field in a VHDL or Verilog® source file or part of a technical reference manual for the IP core. Table A-4 shows one form of the WISHBONE DATASHEET for the 16 x 8-bit memory IP core.

The purpose of the WISHBONE DATASHEET is to promote design reuse. It forces the originator of the IP core to document how the interface operates. This makes it easier for another person to re-use the core.

Table A-4. WISHBONE DATASHEET for the 16 x 8-bit SLAVE memory.

Description	Specification	
General description:	16 x 8-bit memory IP core.	
Supported cycles:	SLAVE, READ/WRITE SLAVE, BLOCK READ/WRITE SLAVE, RMW	
Data port, size: Data port, granularity: Data port, maximum operand size: Data transfer ordering: Data transfer sequencing:	8-bit 8-bit 8-bit Big endian and/or little endian Undefined	
Clock frequency constraints:	NONE (determined by memory primitive)	
Supported signal list and cross reference to equivalent WISHBONE signals:	Signal Name	WISHBONE Equiv.
	ACK_O	ACK_O
	ADR_I(3..0)	ADR_I()
	CLK_I	CLK_I
	DAT_I(7..0)	DAT_I()
	DAT_O(7..0)	DAT_O()
	STB_I	STB_I
	WE_I	WE_I
Special requirements:	Circuit assumes the use of synchronous RAM primitives with asynchronous read capability.	

A.7.3 Memory Primitives and the [ACK_O] Signal

Memory primitives, specific to the FPGA or ASIC target device, are usually used for the RAM storage elements. That's because most high-level languages (such as VHDL and Verilog®) don't synthesize these very efficiently. For this reason, the user should verify that the memory primitives are available for the target device.

The memory circuits shown above are highly portable, but do assume that FASM compatible memories are available. During *write* cycles, most synchronous RAM primitives latch the input data when at the rising clock edge when the write enable input is asserted. However, during *read* cycles the RAM primitives may behave in different ways.

There are two types of RAM primitives that are generally found on FPGA and ASIC devices: (a) those that synchronously present data at the output after the rising edge of the clock input, and (b) those that asynchronously present data at the output after the address is presented to the RAM element.

The circuit assumes that the RAM primitive is the FASM, asynchronous read type. That's because during read cycles the WISHBONE interface assumes that output data is valid at the rising

[CLK_I] edge following the assertion of the [ACK_O] output. Since the circuit ties the [STB_I] signal back to the [ACK_O] signal, the asynchronous read RAM is needed on the circuit shown here.

For this reason, if *synchronous* read type RAM primitives are used, then the circuit must be modified to insert a single wait-state during all read cycles. This is quite simple to do, and only requires an additional flip-flop and gate in the [ACK_O] circuit.

Furthermore, it can be seen that the circuit operates faster if the asynchronous read type RAM primitives are used. That's because the [ACK_O] signal can be asserted immediately after the assertion of [STB_I]. If the synchronous read types are used, then a single-clock wait-state must be added.

A.8 Customization with Tags and TAG TYPEs

One fundamental problem with traditional microcomputer buses is that they can't be customized very easily. That's because they rely on fixed printed circuit boards and connectors. Customization of these components are costly (in terms of time and money) and very often result in system components that aren't compatible.

SoC interconnections like WISHBONE are much easier to customize. That's because their interconnections are programmable in FPGA and ASIC target devices. Furthermore, they are supported by a rich set of development tools such as the VHDL and Verilog® hardware description languages, as well as a wide variety of routing tools. These make it possible to quickly change the interface. However, this can also lead to incompatible interfaces.

WISHBONE allows MASTER and SLAVE interfaces to be customized, while still retaining a highly compatible interface. This is accomplished with a technique known as a *tagged architecture*. This technique is not new to the microcomputer bus industry. The IEEE Authoritative Dictionary defines it as: a computer architecture in which each word is 'tagged' as either an instruction or a unit of data. A similar concept is used by WISHBONE.

Custom tags can be added to the WISHBONE interface as long as they conform to something called a TAG TYPE. There are three general TAG TYPEs defined by WISHBONE. They include an address tag, a set of data tags and a cycle tag. When a custom signal is added to a WISHBONE interface it is assigned a TAG TYPE. This indicates the exact timing that the signal must adhere to. Table A-5 shows some examples for each of the TAG TYPEs.

Table A-5. TAG TYPE examples.

General Type of tag (MASTER):	Address Tag	Data Tag	Cycle Tag
Assigned TAG TYPE:	TGA_O()	TGD_I() & TGD_O()	TGC_O()
Examples:	• Address width (e.g. 16-bit, 24-bit etc.) • Memory management (e.g. user vs. protected address)	• Parity bits • Error correction codes (ECC) • Time stamps • Endian type • Size of data accepted by SLAVE	• Cycle type (e.g. SINGLE, BLOCK and RMW cycles) • Cache control • Interrupt acknowledge cycles • Instruction vs. data cycles

For example, consider a MASTER interface that is adapted to generate a parity bit called [PAR_O]. Since this signal modifies the output data bus, it would be assigned a data TAG TYPE of: TGD_O(). If the MASTER's input bus were similarly modified, it would be assigned a TAG TYPE of: TGD_O(). Assignment of the TAG TYPE does two things: (a) it links the parity bit [PAR_O] to the data bus and (b) it defines the exact timing during SINGLE, BLOCK and RMW bus cycles. However, the creator of the MASTER interface would still need to define the operation of [PAR_O] in the WISHBONE DATASHEET.

Also note that the [TGD_O()] TAG TYPE is an array (i.e. it includes a parenthesis), while the [PAR_O] tag is not arrayed. This is allowed under the WISHBONE specification.

A.9 Point-to-point Interconnection Example

Now that we've reviewed some of the WISHBONE basics, it's time to try them out with a simple example. In this section we'll create a complete WISHBONE system with a point-to-point interconnection. The system includes a 32-bit MASTER interface to a DMA[8] unit, and a 32-bit SLAVE interface to a memory. In this example the DMA transfers data to and from the memory using block transfer cycles.

The purpose of this system is to create a portable benchmarking device. Although the system is very simple, it does allow the user to determine the typical maximum speeds and minimum sizes on any given FPGA or ASIC target device[9].

[8] DMA: Direct Memory Access.

[9] Benchmarking can be a difficult thing to do. On this system the philosophy was to create a 'real-world' SoC that estimates 'typical maximum' speeds and 'typical minimum' size. It's akin to the 'flight envelope' of an airplane. A flight envelope is a graph that shows the altitude vs. the speed of the aircraft. It's one 'benchmark' for the airplane. While the graph may show that the airplane is capable of flying at MACH 2.3 at an altitude of 28,000 meters, it may

Source code for this example can be found in the WISHBONE Public Domain Library for VHDL (under 'EXAMPLE1' in the EXAMPLES folder). The library also has detailed documentation for the library modules, including detailed circuit descriptions and timing diagrams for the INTERCON, SYSCON, DMA and memory modules. The reader is encouraged to review and experiment with all of these files.

Figure A-17 shows the system. The WISHBONE interconnection system (INTERCON) can be found in file ICN0001a. That system connects a simple DMA MASTER (DMA0001a) to an 8 x 32-bit register based memory SLAVE (MEM0002a). The reset and clock signals are generated by system controller SYSCON (SYC0001a).

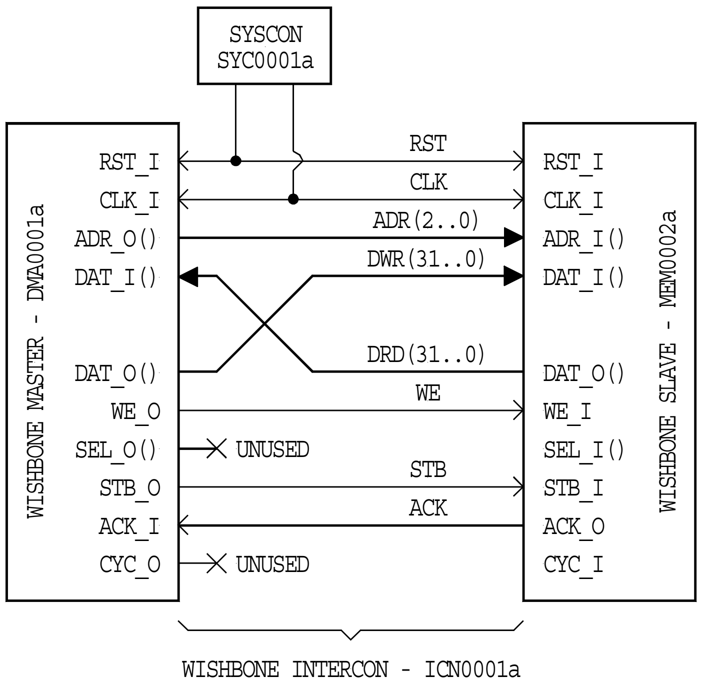


Figure A-17. Point-to-point interconnection example.

This system was synthesized and routed on two styles of Xilinx[10] FPGA: the Spartan 2 and the Virtex 2. For benchmarking purposes the memories were altered so that they used Xilinx distributed RAMs instead of the register RAMs in MEM0002a. A memory interface for the Xilinx RAMs can be found in MEM0001a, which is substituted for MEM0002a.

never actually fly in that situation during its lifetime. The graph is simply a tool for quickly understanding the engineering limits of the design. The same is true for the WISHBONE benchmarks given in this tutorial. However, having said this it is important to remember that the benchmarks are real systems, and do include all of the logic and routing resources needed to implement the design.

[10] Xilinx is a registered trademark of Xilinx, Inc.

It should be noted that the Xilinx distributed RAMs are quite efficient on the WISHBONE interface. As can be seen in the source code, only a single 'AND' gate was needed to interface the RAM to WISHBONE.

The system for the Xilinx Spartan 2 was synthesized and operated on a Silicore evaluation board. This was a 'reality check' that verified that things actually routed and worked as expected. Some of the common signals were brought out to test points on the evaluation board. These were monitored with an HP54620a logic analyzer to verify the operation. Figure A-18 shows an example trace from the logic analyzer. Address lines, data write lines and several control signals were viewed. That Figure shows a write cycle with eight phases followed by a read cycle with eight phases. The data lines always show 0x67 because that's the data transferred by the DMA in this example.

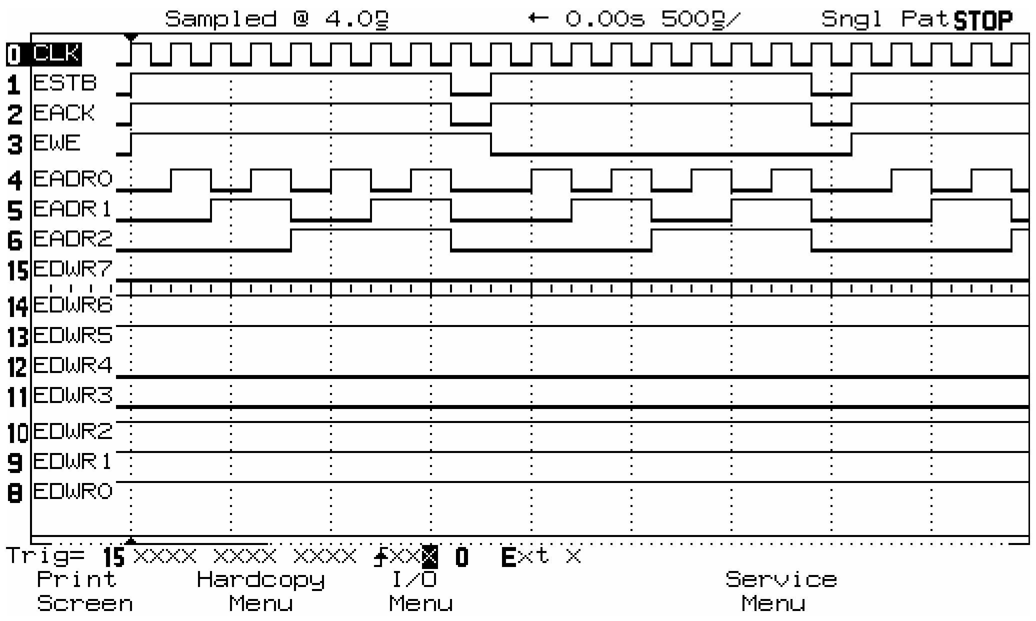


Figure A-18. Logic analyzer trace on the Spartan 2 evaluation board[11].

Table A-6 shows the speed of the system after synthesis and routing. The Spartan 2 benchmarked at about 428 Mbyte/sec, and was tested in hardware (HW TEST). The Virtex 2 part was synthesized and routed, but was only tested under software (SW TEST).

Table A-6. 32-bit Point-to-point Interconnection Benchmark Results

MFG & Type	Part Number	HW TEST	SW TEST	Size	Timing Constraint (MIN)	Actual Speed (MAX)	Data Transfer Rate (MAX)

[11] The logic analyzer samples at 500 Mhz, so the SoC was slowed down to make the traces look better. This trace was taken with a SoC clock speed of 5 MHz. Slowing the clock down is also a good way to verify that the speed of the WISHBONE interface can be 'throttled' up and down.

Xilinx Spartan 2 (FPGA)	XC2S50-5-PQ208C	√		53 SLICE	100 MHz	107 MHz	428 Mbyte/sec
Xilinx Virtex 2 (FPGA)	XC2V40-4-FG256C		√	53 SLICE	200 MHz	203 MHz	812 Mbyte/sec

Notes:
VHDL synthesis tool: Altium Accolade PeakFPGA 5.30a
Router: Xilinx Alliance 3.3.06I_V2_SE2
Hardware evaluation board: Silicore 170101-00 Rev 1.0
Listed size was reported by the router.
Spartan 2 test used '-5' speed grade part (slower than the faster '-6' part).

A.10 Shared Bus Example

Now that we've built a WISHBONE point-to-point interconnection, it's time to look at a more complex SoC design. In this example, we'll create a 32-bit shared bus system with four MASTERs and four SLAVEs. Furthermore, we'll re-use the same DMA, memory and SYSCON modules that we used in the point-to-point interconnection example above. This will demonstrate how WISHBONE interfaces can be adapted to many different system topologies.

This example will require the introduction of some new concepts. As the system integrator, we'll need to make some decisions about how we want our system to work. After that, we'll need to create the various parts of the design in order to finish the job. Some of the decisions and tasks include:

- Choosing between multiplexed and non-multiplexed bus topology.
- Choosing between three-state and multiplexor based interconnection logic.
- Creating the interconnection topology.
- Creating an address map (using variable address decoding).
- Creating a four level round-robin arbiter.
- Creating and benchmarking the system.

A.10.1 Choosing Between Multiplexed and Non-multiplexed Bus Topology

The first step in designing a shared bus is to determine how we'll move our data around the system. In this section we'll explore multiplexed and non-multiplexed buses, and explore some of the trade-offs between them.

The big advantage of multiplexed buses is that they reduce the number of interconnections by routing different types of data over the same set of signal lines. The most common form of multiplexed bus is where address and data lines share a common set of signals. A multiplexed bus is shown in Figure A-19. For example, a 32-bit address bus and 32-bit data bus can be combined to form a 32-bit common address/data bus. This reduces the number of routed signals from 64 to 32.

The major disadvantage of the multiplexed bus is that it takes twice as long to move the information. In this case a non-multiplexed, synchronous bus can generally move address and data information in as little as one clock cycle. Multiplexed address and data buses require at least two clock cycles to move the same information.

Since we're creating a benchmarking system that is optimized for speed, we'll use the non-multiplexed scheme for this example. This will allow us to move one data operand during every clock cycle.

It should be noted that multiplexed buses have long been used in the electronics industry. In semiconductor chips the technique is used to reduced the number of pins on a chip. In the microcomputer bus industry the technique is often used to reduce the number of backplane connector pins.

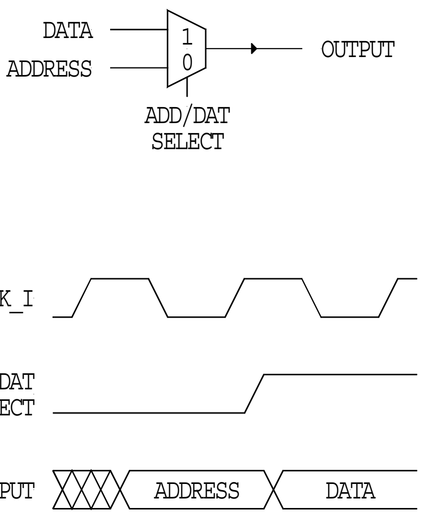


Figure A-19. Circuit and timing diagram for a multiplexed address/data bus.

A.10.2 Choosing Between Three-State and Multiplexor Interconnection Logic

WISHBONE interconnections can use three-state[12] or multiplexor logic to move data around a SoC. The choice depends on what makes sense in the application, and what's available on the integrated circuit.

Three-state I/O buffers have long been used in the semiconductor and microcomputer bus industries. These reduce the number of signal pins on an interface. In microcomputer buses with master-slave architectures, the master that 'owns' the bus turns its buffers 'on', while the other master(s) turn their buffers 'off'[13]. This prevents more than one bus master from driving any signal line at any given time. A similar situation also occurs at the slave end. There, a slave that participates in a bus cycle enables its output buffers during read cycles.

In WISHBONE, all IP core interfaces have 'in' and 'out' signals on the address, data and other internal buses. This allows the interface to be adapted to point-to-point, multiplexed and three-state I/O interconnections. Figure A-20 shows how the 'in' and 'out' signals can be connected to a three-state I/O bus[14].

[12] Three-state buffers are sometimes called Tri-State® buffers. Tri-State® is a registered trademark of National Semiconductor Corporation.
[13] Here, 'on' and 'off' refer to the three-state and non three-state conditions, respectively.
[14] Also note that the resistor/current source shown in the figure can also be a pull-down resistor or a bus terminator, or eliminated altogether.

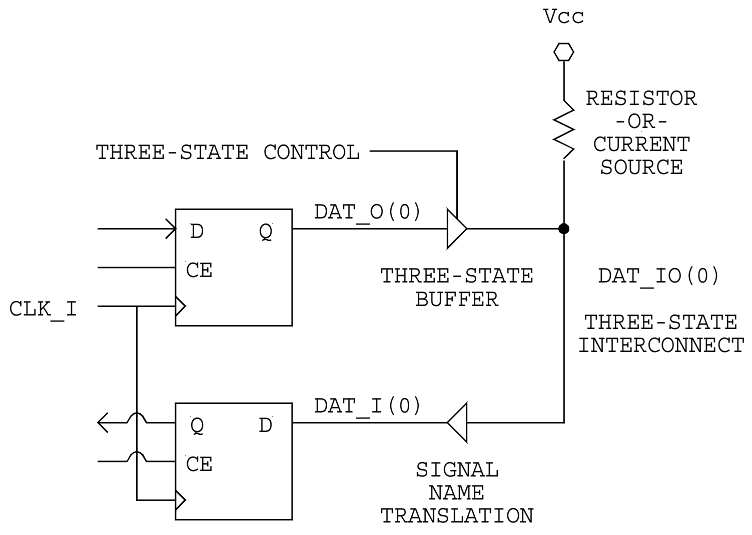


Figure A-20. Connection of a data bus bit to a three-state interconnection.

A simple SoC interconnection that uses three-state I/O buffers is shown in the block diagram of Figure A-21(a). There, the data buses on two master and two slave modules are interconnected with three-state logic. However, this approach has two major drawbacks. First, it is inherently slower than direct interconnections. That's because there are always minimum timing parameters that must be met to turn buffers on-and-off. Second, many IC devices do not have any internal three-state routing resources available to them, or they are very restrictive in terms of location or quantity of these interconnects.

As shown in Figure A-21(b), the SoC bus can be adapted to use multiplexor logic to achieve the same goal. The main advantage of this approach is that it does not use the three-state routing resources which are not available on many FPGA and ASIC devices.

The main disadvantage of the multiplexor logic interconnection is that it requires a larger number of routed interconnects and logic gates (which are not required with the three-state bus approach).

However, there is also a growing body of evidence that suggests that this type of interconnection is easier to route in both FPGA and ASIC devices. Although this is very difficult to quantify, the author has found that the multiplexor logic interconnection is quite easily handled by standard FPGA and ASIC routers. This is because:

- Three-state interconnections force router software to organize the SoC around the fixed three-state bus locations. In many cases, this constraint results in poorly optimized and/or slow circuits.

- Very often, 'bit locations' within a design are grouped together. In many applications, the multiplexor logic interconnection is easier to handle for place & route tools.

- Pre-defined, external I/O pin locations are easier to achieve with multiplexor logic interconnections. This is especially true with FPGA devices.

For the shared bus example we will use multiplexor logic for the interconnection. That's because multiplexor logic interconnections are more portable than three-state logic designs. The shared bus design in this example is intended to be used on many brands of FPGA and ASIC devices.

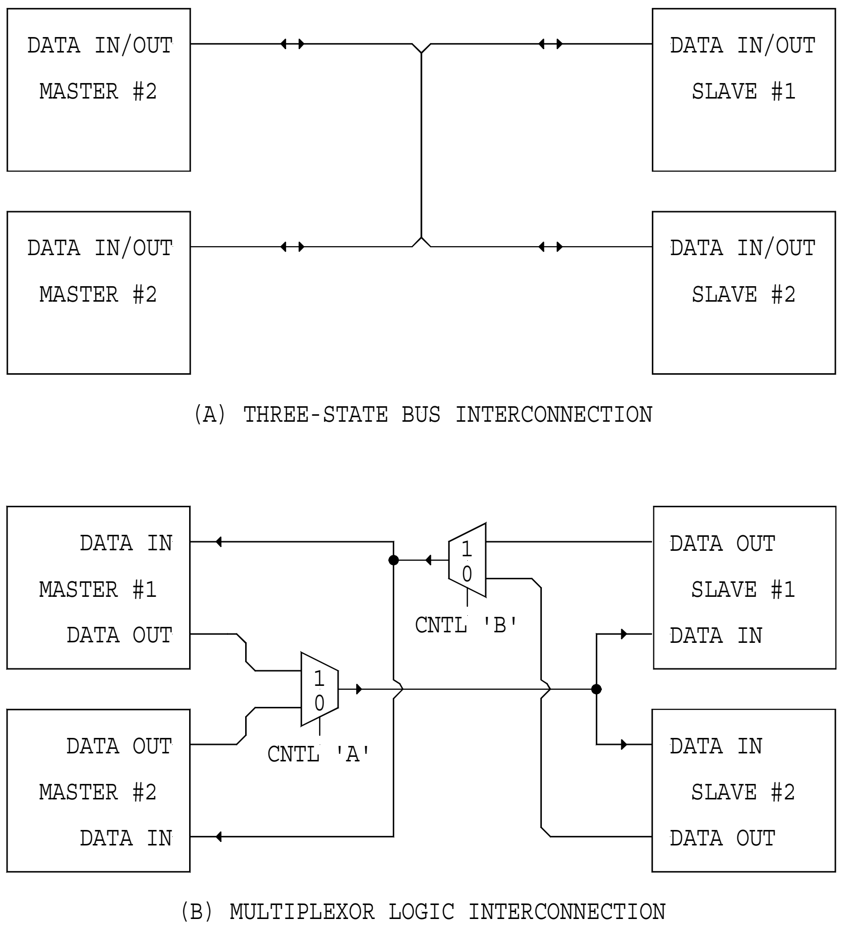


Figure A-21. Three-state bus interconnection vs. multiplexor logic interconnection.

This page is Intentionally Blank

A.10.3 Creating the Interconnection Topology

In the previous two sections it was decided to use multiplexor interconnections with non-multiplexed address and data buses. In this section we'll refine those concepts into a broad interconnection topology for our system. However, we'll save the details for later. For now, we're just interested in looking at some of the general issues.

In WISHBONE nomenclature, the interconnection is also called the INTERCON. The INTERCON is an IP Core that connects all of the MASTER and SLAVE interfaces together.

Figure A-22 shows the generic topology of an INTERCON that supports multiplexor interconnections with multiplexed address and data buses. By 'generic', we mean that there are 'N' MASTERs and SLAVEs shown in the diagram. The actual number of MASTER and SLAVE interfaces can be adjusted up or down, depending upon what's needed in the system. In the shared bus example we'll use four MASTERs and four SLAVEs. However, for now we'll think in more general terms.

A module called the SYSCON provides the system with a stable clock [CLK_O] and reset signal [RST_O]. For now, we'll assume that the clock comes from off-chip, and that the reset signal is synchronized from some global system reset.

After the initial system reset, one or more MASTERs request the interconnection, which we'll call a 'bus' for now. MASTERs do this by asserting their [CYC_O] signal. An arbiter, which we'll discuss shortly, determines which MASTER can use the bus. One clock edge after the assertion of a [CYC_O] signal the arbiter grants the bus to one of the MASTERs that requested it. It grants the bus by asserting grant lines GNT0 – GNTN and GNT(N..0). This informs both the INTERCON as to which MASTER can own the bus.

Once the bus is arbitrated, the output signals from the selected MASTER are routed, via multiplexors, onto the shared buses. For example, if MASTER #0 obtains the bus, then the address lines [ADR_O()] from MASTER #0 are routed to shared bus [ADR()]. The same thing happens to the data out [DAT_O()], select out [SEL_O()], write enable [WE_O] and strobe [STB_O] signals. The shared bus output signals are routed to the inputs on the SLAVE interfaces.

The arbiter grant lines are also used to enable the terminating signals [ACK_I], [RTY_I] and [ERR_I]. These are enabled at the MASTER that acquired the bus. For example, if MASTER #0 is granted the bus by the arbiter, then the [ACK_I], [RTY_I] and [ERR_I] are enabled at MASTER #0. Other MASTERs, who may also be requesting the bus, never receive a terminating signal, and therefore will wait until they are granted the bus.

During this interval the common address bus [ADR()] is driven with the address lines from the MASTER. The address lines are decoded by the address comparator, which splits the address space into 'N' sections. The decoded output from the comparator is used to select the slave by way of its strobe input [STB_I]. A SLAVE may only respond to a bus cycle when its [STB_I] is asserted. This is also a wonderful illustration of the partial address decoding technique used by WISHBONE, which we'll discuss in depth below.

PSL Code

```
// High level properties

// [property regarding all masters and the arbiter]

property arbitrating0 =
%for i in 0:`NUM_OF_MASTERS-1 do
      %for j in 0:`NUM_OF_MASTERS-1 do
            always !(i=j) -> !( GNT%{i} && GNT%{j});
      %end
%end

// [property regarding all masters and the arbiter]

property arbitrating1 =
always onehot(GNT[0:`NUM_OF_MASTERS-1]);

// [property regarding all masters and the arbiter]

property arbitrating2 =
%for i in 0:`NUM_OF_MASTERS-1 do
      always GNT[i] <-> GNT%{i};
%end

// [property regarding all masters and the arbiter]

property arbitrating =  arbitrating0 && arbitrating1 && arbitrating2;

// [assertion regarding all masters and the arbiter]

assert arbitrating;
```

For example, consider a system with an addressing range of sixteen bits. If the addressing range were evenly split between all of the SLAVEs, then each SLAVE would be allocated 16 Kbytes of address space. This is shown in the address map of Figure A-23. In this case, the address comparator would decode bits [ADR(15..14)]. In actual practice the system integrator can alter the address map at his or her discretion.

Once a SLAVE is selected, it participates in the current bus cycle generated by the MASTER. In response to the cycle, the SLAVE must assert either its [ACK_O], [RTY_O] or [ERR_O] output. These signals are collected with an 'or' gate, and routed to the current MASTER.

Also note that during read cycles, the SLAVE places data on its [DAT_O()] bus. These are routed from the participating SLAVE to the current MASTER by way of a multiplexor. In this case, the multiplexor source is selected by some address lines which have been appropriately selected to switch the multiplexor.

Once the MASTER owning the bus has received an asserted terminating signal, it terminates the bus cycle by negating its strobe output [STB_O]. If the MASTER is in the middle of a block transfer cycle, it will generate another phase of the block transfer. If it's performing a SINGLE cycle, or if its at the end of a BLOCK cycle, the MASTER terminates the cycle by negating its [CYC_O] signal. This informs the MASTER that it's done with the bus, and that it can re-arbitrate it.

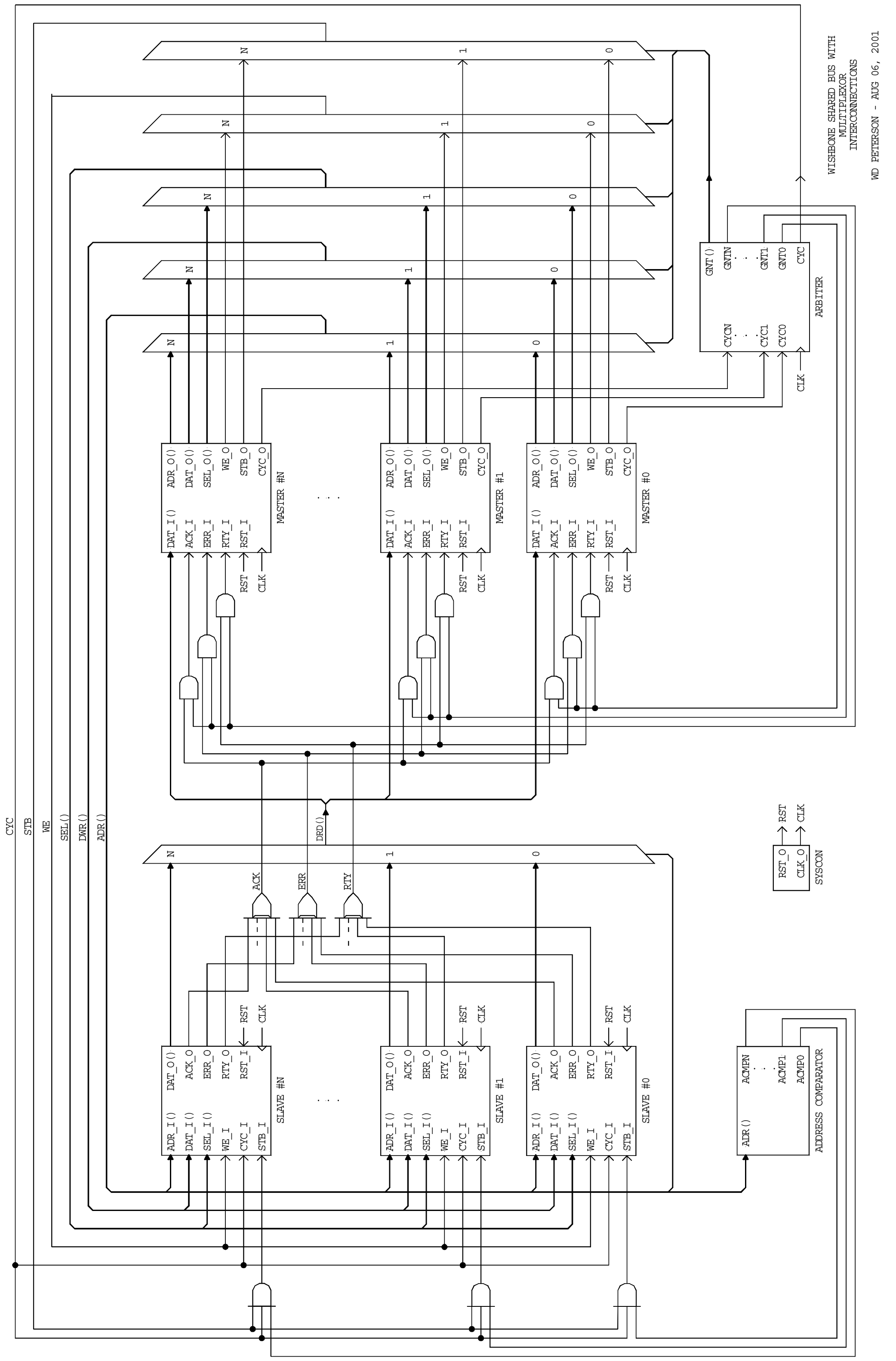


Figure A-22. WISHBONE shared bus with multiplexor interconnections.

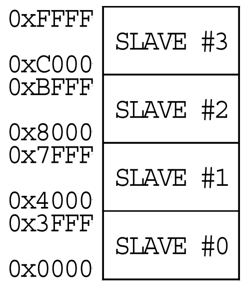


Figure A-23. Address map example.

A.10.4 Full vs. Partial Address Decoding

The address comparator in our INTERCON example is a good way to explain the concept of WISHBONE partial address decoding.

Many systems, including standard microcomputer buses like PCI and VMEbus, use *full address decoding*. Under that method, each slave module decodes the full address bus. For example, if a 32-bit address bus is used, then each slave decodes all thirty-two address bits (or at least a large portion of them).

SoC buses like WISHBONE use *partial address decoding* on slave modules. Under this method, each slave decodes only the range of addresses that it uses. For example, if the slave has only four registers, then the WISHBONE interface uses only two address bits. This technique has the following advantages:

- It facilitates high speed address decoders.
- It uses less redundant address decoding logic (i.e. fewer gates).
- It supports variable address sizing (between zero and 64-bits).
- It supports the variable interconnection scheme.
- It gives the system integrator a lot of flexibility in defining the address map.

For example, consider the serial I/O port (IP core) with the internal register set shown in Figure A-24(a). If *full address decoding* is used, then the IP core must include an address decoder to select the module. In this case, the decoder requires: 32 bits – 2 bits = 30 bits. In addition, the IP core would also contain logic to decode the lower two bits which are used to determine which I/O registers are selected.

If *partial address decoding* is used, then the IP core need only decode the two lower address bits ($2^2 = 4$). The upper thirty bits are decoded by logic outside of the IP core. In this case the decoder logic is shown in Figure A-24(b).

Standard microcomputer buses always use the full address decoding technique. That's because the interconnection method does not allow the creation of any new signals on the interface. However, in WISHBONE this limitation does not exist. WISHBONE allows the system integrator to modify the interconnection logic and signal paths.

One advantage of the partial address decoding technique is that the size of the address decoder (on the IP core) is minimized. This speeds up the interface, as decoder logic can be relatively slow. For example, a 30-bit full address decoder often requires at least 30 XOR gates, and a 30-input AND gate.

Another advantage of the partial address decoding technique is that less decoder logic is required. In many cases, only one 'coarse' address decoder is required. If full address decoding is used, then each IP core must include a redundant set of address decoders.

Another advantage of the partial address decoding technique is that it supports variable address sizing. For example, on WISHBONE the address path can be any size between zero and 64-bits. Slave modules are designed to utilize only the block of addresses that are required. In this case, the full address decoding technique cannot be used because the IP core designer is unaware of the size of the system address path.

Another advantage of the partial address decoding technique is that it supports the variable interconnection scheme. There, the type of interconnection logic is unknown to the IP core designer. The interconnection scheme must adapt to the types of slave IP cores that are used.

The major disadvantage of the partial address decoding technique is that the SoC integrator must define part of the address decoder logic for each IP core. This increases the effort to integrate the IP cores into the final SoC.

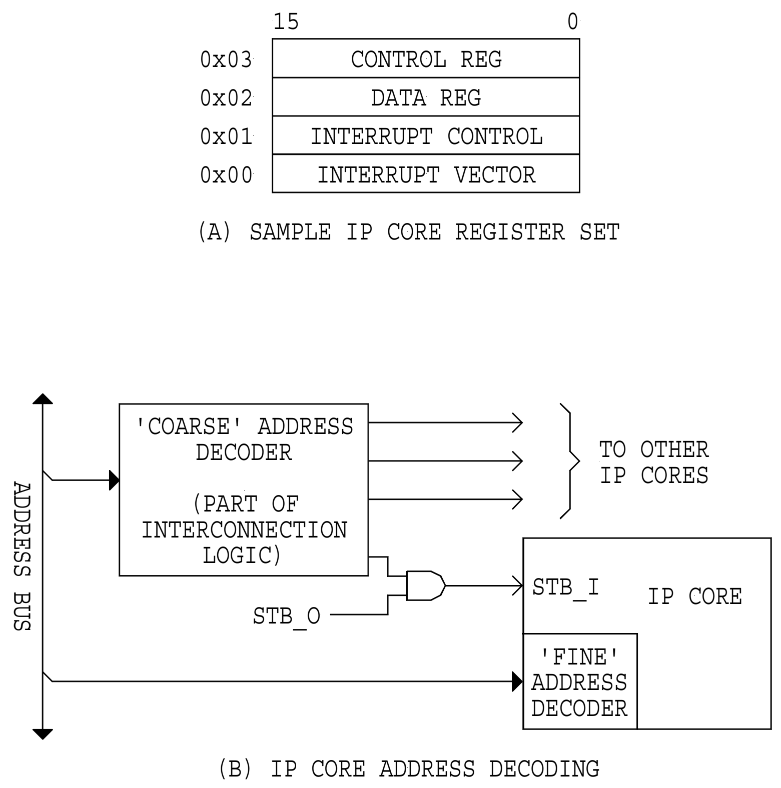


Figure A-24. WISHBONE partial address decoding technique.

A.10.5 The System Arbiter

The system arbiter determines which MASTER can use the shared bus. The WISHBONE specification allows a variety of arbiters to be used. However, in this example a four level round-robin arbiter is used.

Round-robin arbiters give equal priority to all of the MASTERs. These arbiters grant the bus on a rotating basis much like the four position rotary switch shown in Figure A-25. When a MASTER relinquishes the bus (by negating its [CYC_O] signal), the switch is turned to the next position, and the bus is granted to the MASTER on that level. If a request is not pending on a certain level, the arbiter skips over that level and continues onto the next one. In this way all of the MASTERs are granted the bus on an equal basis.

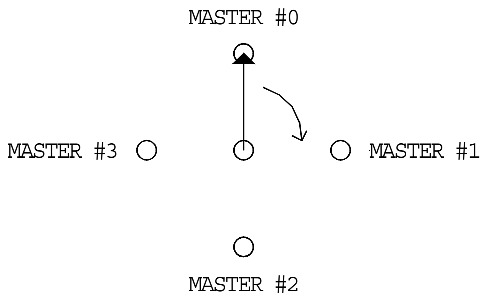


Figure A-25. Round-robin arbiters grant the bus on a rotating basis much like a rotary switch.

Round-robin arbiters are popular in data acquisition systems where data is collected and placed into shared memory. Often these peripherals must off-load data to memory on an equal basis. Priority arbiters (where each MASTER is assigned a higher or lower level of priority) do not work well in these applications because some peripherals would receive more bus bandwidth than others, thereby causing data 'gridlock'.

The arbiter used in this example can be found in the WISHBONE Public Domain Library for VHDL. ARB0001a is used for the example.

A.10.6 Creating and Benchmarking the System

The final task in our shared bus system example is to create and benchmark the entire system. The INTERCON in our example system is based on the generic shared bus topology that was described above. However, that system is fine tuned to give the exact features that we will need.

The final system supports four DMA0001a MASTERs, four MEM0002a memories (SLAVEs), a 32-bit data bus, a five bit address bus, a single SYC0001a system controller and a ARB0001a four level round-robin arbiter. The resulting VHDL file can be found under ICN0002a in the WISHBONE Public Domain Library for VHDL.

In this application, the round-arbiter was chosen because all of the MASTERs are DMA controllers. That means that all four MASTERs continuously vie for the bus. If a priority arbiter were used, then only the one or two highest priority MASTERs would ever get the bus.

As we'll see shortly, the error and retry signals [ERR_I] and [RTY_I] are not supported on the MASTER and SLAVE interfaces on our example system. That's perfectly okay because these signals are optional in the WISHBONE specification. We could have added these signals in there, but they would have been removed by synthesis and router logic minimization tools.

Since all of the MASTERs and SLAVEs on this system have identical port sizes and granularities, the select [SEL] interconnection has been omitted. This could have been added, but it wasn't needed.

The INTERCON system includes a partial address decoder for the SLAVEs. This decoder creates the system address space shown in Figure A-26. The final address map is shown in Table A-7.

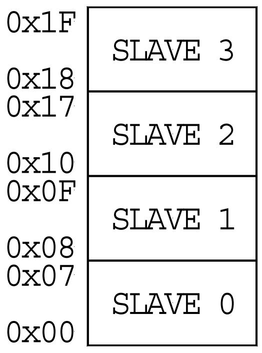


Figure A-26. Address map used by the INTERCON example.

Table A-7. Address spaces used by INTERCON.

DMA Master:	DMA's To:	At Addresses	Using Cycles
MASTER #0	SLAVE #0	0x00 – 0x07	BLOCK READ/WRITE
MASTER #1	SLAVE #1	0x08 – 0x0F	BLOCK READ/WRITE
MASTER #2	SLAVE #2	0x10 – 0x17	BLOCK READ/WRITE
MASTER #3	SLAVE #3	0x18 – 0x1F	SINGLE READ/WRITE

Source code for this example can be found in the WISHBONE Public Domain Library for VHDL (in the EXAMPLES folder). The library also has detailed documentation for the library modules, including detailed circuit descriptions and timing diagrams. The reader is encouraged to review and experiment with all of these files.

This system was synthesized and routed on two styles of Xilinx[15] FPGA: the Spartan 2 and the Virtex 2. For benchmarking purposes the memories were altered so that they used Xilinx distributed RAMs instead of the register RAMs in MEM0002a. A memory interface for the Xilinx RAMs can be found in MEM0001a, which is substituted for MEM0002a.

It should be noted that the Xilinx distributed RAMs are quite efficient on the WISHBONE interface. As can be seen in the source code, only a single 'AND' gate was needed to interface the RAM to WISHBONE.

The system for the Xilinx Spartan 2 was synthesized and operated on a Silicore evaluation board. In order to verify that the system actually does run correctly, an HP54620a logic analyzer was connected to test pins on the board, and some of the signals were viewed. Figure A-27 shows the trace. Address lines, data write lines and several control signals are shown.

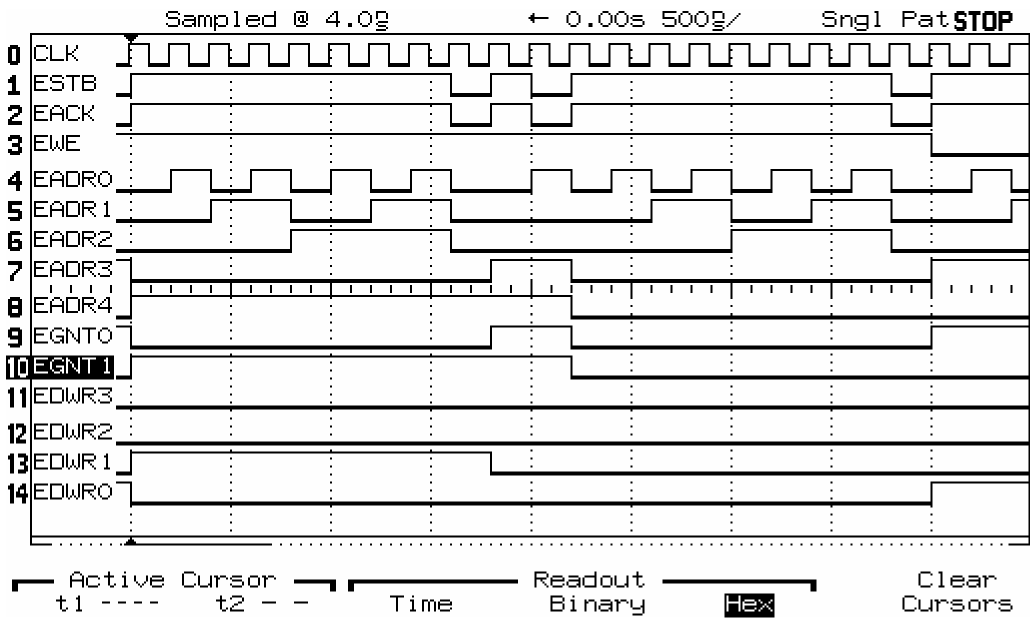


Figure A-27. Logic analyzer trace on the Spartan 2 evaluation board[16].

Table A-8 shows the speed of the system after synthesis and routing. The Spartan 2 benchmarked at about 220 Mbyte/sec, and was tested in hardware (HW TEST). The Virtex 2 part was only synthesized and routed, and showed a maximum speed of about 404 Mbyte/sec (SW TEST).

[15] Xilinx is a registered trademark of Xilinx, Inc.
[16] The logic analyzer samples at 500 Mhz, so the SoC was slowed down to make the traces look better. This trace was taken with a SoC clock speed of 5 MHz.

Table A-8. 32-bit Shared Bus Interconnection Benchmark Results

MFG & Type	Part Number	HW TEST	SW TEST	Size	Timing Constraint (MIN)	Actual Speed (MAX)	Data Transfer Rate (MAX)
Xilinx Spartan 2 (FPGA)	XC2S50-5-PQ208C	√		356 SLICE	55 MHz	55MHz	220 Mbyte/sec
Xilinx Virtex 2 (FPGA)	XC2V250-5-FG256C		√	355 SLICE	99 MHz	101 MHz	404 Mbyte/sec

Notes:
VHDL synthesis tool: Altium Accolade PeakFPGA 5.30a
Router: Xilinx Alliance 3.3.06I_V2_SE2
Hardware evaluation board: Silicore 170101-00 Rev 1.0
Listed size was reported by the router.
Spartan 2 test used '-5' speed grade part (slower than the faster '-6' part).

A.10.7 Other Benchmarks

Other benchmarks have also been run on the shared bus interconnection. At present, the fastest implementation of the 4 MASTER / 4 SLAVE shared bus system has been with 64-bit interfaces. This has resulted in data transfer rates in excess of 800 Mbyte/sec.

A.11 References

Di Giacomo, Joseph. Digital Bus Handbook. McGraw-Hill 1990. ISBN 0-07-016923-3.

Peterson, Wade D. The VMEbus Handbook, 4th Edition. VITA 1997. ISBN 1-885731-08-6

INDEX

Ω