WISHBONE
COMPATIBLE

*Specification for the:*

# *WISHBONE System-On-Chip (SoC) Interconnection Architecture for Portable IP Cores*

*Revision: B*

Preliminary

***Silicore Corporation***
*6310 Butterworth Lane – Corcoran, MN 55340*
*TEL: (763) 478-3567 FAX: (763) 478-3568*
*www.silicore.net*

***Silicore***

***Electronic Design***
***Sensors ● IP Cores***

**This Page is Intentionally Blank**

## Stewardship

Stewardship for this specification is maintained by Silicore Corporation. Questions, comments and suggestions about this document are welcome and should be directed to:

Wade D. Peterson, Silicore Corporation
6310 Butterworth Lane – Corcoran, MN 55340
TEL: (763) 478-3567; FAX: (763) 478-3568
E-MAIL: wadep@silicore.net URL: www.silicore.net

Silicore Corporation provides this document as a service to its customers and to the IP core industry as a whole. The intent of this specification is to improve the quality of Silicore products, as well as to foster cooperation among the users and suppliers of IP cores.

## Copyright Release / Royalty Release / Patent and Trademark 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.

## Revision History

Revision A (preliminary) - June 16, 1999
Original specification release.

Revision A.1 (preliminary) - Updated July 27, 1999
Correct typographical errors.

Revision B (preliminary) – Updated January 5, 2001
Incorporate feedback from users. Clarify use of acknowledge and TAG signals. Add [TAGN_O] signal to SLAVE and [TAGN_I] to MASTER. Correct typographical errors. Change name from ‘WISHBONE Interconnection Architecture For Portable IP Cores’ to ‘WISHBONE System-On-Chip (SoC) Interconnection Architecture for Portable IP Cores’. Add ‘WISHBONE COMPATIBLE’ logo. Standardize printing and covers, Change standard font to Times New Roman 12 pt. Change ‘WISHBONE’ to all capital letters. Change steward address. Add appendcies with application notes. Move the glossary to chapter 1. Add an index. Reformat under Adobe Acrobat with bookmarks.

# Table of Contents

**CHAPTER 1 - INTRODUCTION** ........ **6**

1.1 WISHBONE FEATURES ........ 7
1.2 WISHBONE OBJECTIVES ........ 9
1.3 SPECIFICATION TERMINOLOGY ........ 11
1.4 USE OF TIMING DIAGRAMS ........ 12
1.5 SIGNAL NAMING CONVENTIONS ........ 14
1.6 WISHBONE LOGO ........ 14
1.7 GLOSSARY OF TERMS ........ 14

**CHAPTER 2 – INTERFACE SPECIFICATION** ........ **21**

2.1 REQUIRED DOCUMENTATION FOR IP CORES ........ 21
2.2 WISHBONE SIGNAL DESCRIPTION ........ 22

**CHAPTER 3 – BUS INTERFACE** ........ **27**

3.1 GENERAL OPERATION ........ 27
3.2 SINGLE READ / WRITE CYCLES ........ 32
3.3 BLOCK READ / WRITE CYCLES ........ 37
3.4 RMW CYCLE ........ 44
3.5 DATA ORGANIZATION ........ 47

**CHAPTER 4 – TIMING SPECIFICATION** ........ **56**

**CHAPTER 5 – APPLICATION INTERFACE** ........ **58**

5.1 INTERCONNECTION METHODS ........ 58
5.2 THREE-STATE INTERCONNECTIONS ........ 61
5.3 ENDIAN CONVERSION ........ 62

**REFERENCES** ........ **63**

**APPENDIX A – WISHBONE DESIGN PHILOSOPHY** ........ **64**

**APPENDIX B - WISHBONE INTERFACE FOR SLAVE I/O PORTS** ........ **76**

**APPENDIX C - WISHBONE INTERFACE FOR MEMORY ELEMENTS** ........ **83**

**INDEX** ........ **86**

# 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 multiplexors 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.

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 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
    - EVENT cycle

- Data bus widths[2] and operand sizes up to 64-bits.
- Supports both BIG ENDIAN and LITTLE ENDIAN data ordering.
- Variable core interconnection method supports memory mapped, FIFO memory and crossbar interconnections.
- Handshaking protocol allows each core to throttle data transfer speed.
- Supports single clock data transfers.
- Supports normal cycle termination, retry termination and termination due to error.

[2] Specifications are given for data port and operand sizes up to 64-bits. However, the basic architecture can theoretically support any data width (e.g. 128-bit, 256-bit etc.). Also, zero bit data bus accesses are permissible (for event cycles).

- Address widths[3] up to 64-bits.

- 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 tag support. This is useful for identifying data transfers such as:

  - Data transfers
  - 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, including:

  - Single MASTER / single SLAVE
  - Multiple MASTER / single SLAVE
  - Single MASTER / multiple SLAVE
  - Multiple MASTER / multiple SLAVE
  - Crossbar switches

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

- Supports various IP core interconnection means, including:

  - Unidirectional bus
  - Bi-directional bus
  - Multiplexor based interconnections
  - Three-state based interconnections
  - Off chip I/O

- Synchronous design assures portability, simple design and ease of test.

- Very simple, variable timing specification.

- Documentation requirements allow the end user to quickly evaluate interface needs.

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

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

[3] Specifications are given for address widths between zero and 64-bits. However, the basic architecture can theoretically support any address width.

- 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 objective of the specification is to create a flexible interconnection means for use with semiconductor IP cores. This allows various IP core modules to be connected together to form a system-on-chip.

A further objective of the specification is to enforce good compatibility between IP core modules. This enhances design reuse.

A further objective of the specification is to create a robust standard, but one that does not unduly constrain the creativity of the core developer or the end user.

A further objective of the specification is to make it easy to understand by both the core developer and the end user.

A further objective of the specification is 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.

A further objective of the specification is create a portable interface that is independent of the underlying semiconductor technology. For example, the interconnect must be capable of working with both FPGA and ASIC hardware target devices.

A further objective of the specification is to make the interface independent of logic signaling levels.

A further objective of the specification is 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.

A further objective of the specification is 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.

A further objective of the specification is to require a minimum standard for documentation. This allows IP core users to quickly evaluate and integrate new cores.

A further objective of the specification is 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 interface.

A further objective is 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 un-thought-of, requirements.

A further objective is 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.

A further objective is 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.

A further objective is to create an architecture with variable address and data path widths to meet a wide variety of system requirements.

A further objective is 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.

A further objective is to create an architecture that allows data to be tagged. This allows the purpose for each bus cycle to be identified by a SLAVE. For example, in microprocessor based systems it is often necessary to discriminate between data transfer, interrupt acknowledge and caching operations.

A further objective is 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.

A further objective is to create an architecture that supports crossbar switches.

A further objective is 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.

A further objective is 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.

A further objective is to create a synchronous protocol that provides a simple timing specification. This makes the interface very easy to integrate.

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

A further objective is to create a synchronous protocol where data may be transferred through memory mapped, FIFO memory or crossbar switch interconnections.

A further objective is 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.

## 1.3 Specification Terminology

To avoid confusion, and to clarify the requirements for compliance, this specification makes use of five keywords to define the operation of the WISHBONE interconnect. The keywords 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 uppercase 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 single MASTER / single SLAVE configuration, 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.

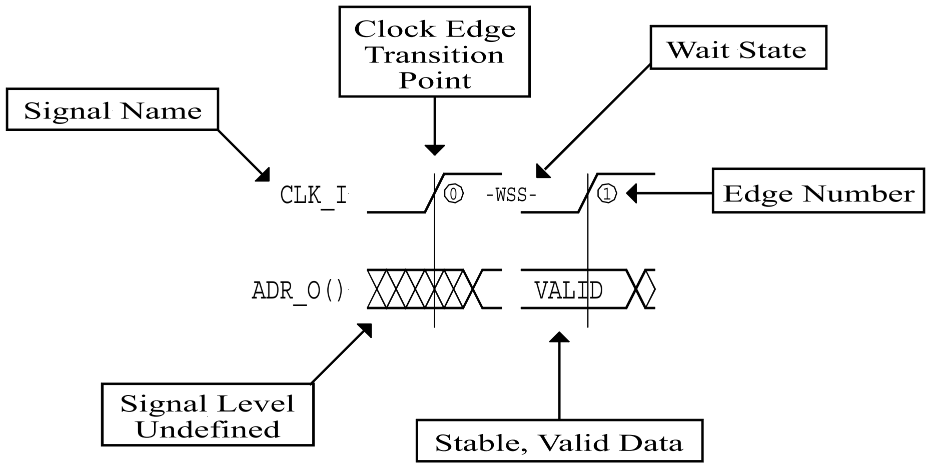


Figure 1-1. Use of timing diagrams.

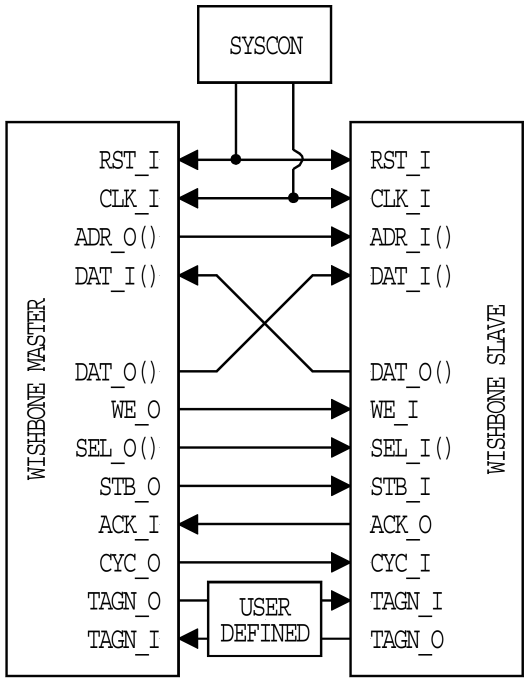


Figure 1-2. Standard connection for timing diagrams.

Individual signals may or may not be present on an specific interface. That's because most 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.

Undefined signal levels are indicated by a hatched region. In MASTER interfaces, this region indicates that the signal level is undefined, and may take any state. In SLAVE interfaces, this region 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'. Non-array signals show a steady high or low state.

## 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.

In some cases, the input and output characters 'I' and 'O' may be omitted and replaced by an 'X'. For example: [TAG3_X]. This is not an actual signal name, but rather a shorthand form to indicate both the [TAG3_I] and [TAG3_O] signal.

Signal arrays are identified by a name followed by the array boundaries in parenthesis. For example, [DAT_I(63..0)] is a signal array with upper array boundary number sixty-three, and lower array boundary number zero. Furthermore, the array boundaries indicate the full range of the permissible array size. 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.

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 1.10**
Documents describing a WISHBONE compatible SoC component that are 100% compliant with this standard, MAY use the WISHBONE logo.

## 1.7 Glossary of Terms

**0x (prefix)**
The '0x' prefix indicates a hexadecimal number. It is the same nomenclature as commonly used in the UNIX operating system.

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

**ASIC**
Acronym for: Application Specific Integrated Circuit. 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**
A verb indicating that a logic state as switched from the inactive state to the active state. When active high logic is used it means that a signal has switched from a low logic level to a high logic level.

**Bus**
A common group of paths over which input and output signals are routed.

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

**Crossbar Interconnect (Switch)**
Crossbar switches are mechanisms that allow IP cores 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.

**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 (see references).

**ENDIAN**
See the definition under 'Data Organization'.

**FIFO Memory**
Acronym for: First In First Out. A type of memory used to transfer data between ports on two devices. The first data loaded into the FIFO by an output port on one device is the first data read by an input port on another device. The FIFO memory is very useful for interconnecting cores of differing speeds.

**Fixed Interconnection**
A microcomputer bus interconnection 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 microcomputer bus 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.

**Foundry**
See silicon foundry.

**FPGA**
Acronym for: Field Programmable Gate Array. Generally 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 module decodes all thirty-two address bits. This technique is used on standard microcomputer buses like PCI and VMEbus. Also see *partial address decoding*.

**Glue Logic**
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.

**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*.

**HDL**
Acronym for: Hardware Description Language. Examples include VHDL and Verilog®.

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

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

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

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

**Multiplexor Logic Interconnection**
A microcomputer bus interconnection that uses multiplexors 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 as switched from the active state to the inactive state. When active high logic is used it means that a signal has switched from a high logic level to a low logic level. Also see *asserted*.

**Operand Size**
The operand size is the largest single unit of data transfer that will be moved through the 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*.

**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 has only four registers, then it decodes only two address bits. This technique is used on SoC microcomputer buses, and has the advantages of: less redundant logic in the system, it supports variable address buses, it supports variable interconnection buses and is relatively fast. Also see *partial address decoding*.

**PCI**
Acronym for: Peripheral Component Interconnect. Generally used as an nterconnection scheme between chips. While this specification is very flexible, it isn't practical for system-on-chip applications because if it's large size and slow speed.

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

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

**RTL Design Methodology**
A design methodology that uses register-transfer-logic (RTL) concepts. This design methodology 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 which means that each register is clocked with a common clock.

**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.

**SoC**
Acronym for System-on chip. See also System-on-chip.

**Structured Design**
A popular design practice used by 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 complex projects.

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

**System-on-chip**
The ability to create whole systems on a single integrated circuit. In many cases, this requires the use of IP cores which have been designed by various IP core providers and individuals. 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.

**Target Device**
The semiconductor type (or technology) onto which the IP core design is impressed. Typical examples include FPGA and ASIC 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. Three-state buffers are sometimes called Tri-State® buffers. Tri-State® is a registered trademark of National Instruments Corporation. Also see *multiplexor logic interconnection*.

**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 microcomputer bus timing specification that is not fixed, and can vary between implementations. Used in SoC buses like WISHBONE. When used in SoC applications, the timing specifications are dictated by the system integrator, and are enforced by integration software such as place-and-route tools.

**Verilog®**
A textual based computer language intended for use in circuit design. The VHDL 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 and elsewhere.

**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 and elsewhere.

**VMEbus**
Acronym for: Versa Module Eurocard bus. A popular microcomputer (board) bus. While this specification is very flexible, it isn't practical for system-on-chip applications because if it's large size and slow speed.

**WISHBONE DATASHEET**
Documentation must be provided for each IP core with a WISHBONE interconnect. This helps the end user understand the operation of the core, and how to connect it to other cores. The documentation takes the form of a WISHBONE DATASHEET. This can be included in a technical reference manual for the IP core.

**WISHBONE Signal**
A signal that is defined as part of the WISHBONE interface. 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.

# Chapter 2 – Interface Specification

This chapter describes the signaling method between MASTER and SLAVE modules. 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 must 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.

**RULE 2.10**
Each WISHBONE compatible IP core MUST include a WISHBONE DATASHEET as part of the IP core documentation.

**RULE 2.11**
The WISHBONE DATASHEET MUST include the revision level of the WISHBONE specification to which it was designed.

**RULE 2.20**
The WISHBONE DATASHEET MUST include the signal names that are defined for a WISHBONE SoC component. If a signal name is different than defined in this specification then it MUST be cross-referenced to the corresponding signal name which is used in this specification.

**PERMISSION 2.10**
Any signal name MAY be used to describe the WISHBONE signals.

**RULE 2.30**
Signal names MUST adhere to the rules of the native tool in which the IP core is designed.

**OBSERVATION 2.10**
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.

**RECOMENDATION 2.10**
It is recommended that the interface use the signal names that are defined in this document.

**OBSERVATION 2.20**
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.

**RULE 2.40**
All WISHBONE interface signals MUST use active high logic.

**PERMISSION 2.11**
Non-WISHBONE signals MAY be used with IP cores.

**OBSERVATION 2.21**
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 of the disk interface signals. However, it does require that the operation of these other signals be documented in the WISHBONE DATASHEET.

**OBSERVATION 2.22**
[TAGN_I] and [TAGN_O] are user defined signals that must adhere to the timing specifications given in this document.

## 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 Signals

#### CLK_O
The clock output [CLK_O] coordinates all activities for the internal logic within the WISHBONE interconnect. It is connected to the [CLK_I] input on MASTER and SLAVE modules.

**RST_O**
The reset output [RST_O] forces the WISHBONE interface to restart. Furthermore, all internal self-starting state machines will be forced into an initial state. It is connected to [RST_I] on MASTER and SLAVE modules.

### 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 must be stable before the rising edge of [CLK_I].

**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.

**TAGN_I**
The tag input(s) [TAGN_I] are user defined, and are used with either MASTER or SLAVE interfaces. 'N' in this signal name refers to a tag number because multiple tags may be used (e.g. [TAG3_I]). Tag inputs are used whenever an IP core needs specific information from the interconnection. For example, a MASTER can be designed to monitor the state of a FIFO.

**TAGN_O**
The tag output(s) [TAGN_O] are user defined, and are used with either MASTER or SLAVE interfaces. For example, the tag output(s) can be used to indicate the type of data transfer in progress. Furthermore, 'N' in this signal name refers to a tag number because multiple tags may be used. For example, [TAG1_O] may indicate a valid data transfer cycle, [TAG2_O] may indicate an interrupt acknowledge cycle and so on. The exact meaning of each tag is defined by the IP core provider in the WISHBONE DATASHEET.

### 2.2.3 MASTER Signals

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

**ADR_O(63..0)**
The address output array [ADR_O(63..0)] is used to pass a binary address, with the most significant address bit at the higher numbered end of the signal array. The lower array boundary is specific to the data port size. The higher array boundary is core-specific. 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. Once the arbiter grants the bus to the MASTER, it is held until [CYC_O] is negated.

**DAT_I(63..0)**
The data input array [DAT_I(63..0)] is used to pass binary data. The array boundaries are determined by the port size. Also see the [DAT_O(63..0)] and [SEL_O(7..0)] signal descriptions.

**DAT_O(63..0)**
The data output array [DAT_O(63..0)] is used to pass binary data. The array boundaries are determined by the port size. Also see the [DAT_I(63..0)] and [SEL_O(7..0)] signal descriptions.

**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 in the WISHBONE DATASHEET. Also see the [ACK_I] and [RTY_I] signal descriptions.

**RTY_I**
The retry input [RTY_I] 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 in the WISHBONE DATASHEET. Also see the [ERR_I] and [RTY_I] signal descriptions.

**SEL_O(7..0)**
The select output array [SEL_O(7..0)] indicates where valid data is expected on the [DAT_I(63..0)] signal array during READ cycles, and where it is placed on the [DAT_O(63..0)] signal array during WRITE cycles. Also see the [DAT_I(63..0)], [DAT_O(63..0)] 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(7..0)]. The SLAVE must assert either the [ACK_I], [ERR_I] or [RTY_I] signals in response to every assertion of the [STB_O] signal.

**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(63..0)**
The address input array [ADR_I(63..0)] is used to pass a binary address, with the most significant address bit at the higher numbered end of the signal array. The lower array boundary is specific to the data port size. The higher array boundary is core-specific. 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.

**DAT_I(63..0)**
The data input array [DAT_I(63..0)] is used to pass binary data. The array boundaries are determined by the port size. Also see the [DAT_O(63..0)] and [SEL_O(7..0)] signal descriptions.

**DAT_O(63..0)**
The data output array [DAT_O(63..0)] is used to pass binary data. The array boundaries are determined by the port size. Also see the [DAT_I(63..0)] and [SEL_O(7..0)] signal descriptions.

**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 in the WISHBONE DATASHEET. Also see the [ACK_O] and [RTY_O] signal descriptions.

**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 in the WISHBONE DATASHEET. Also see the [ERR_O] and [RTY_O] signal descriptions.

**SEL_I(7..0)**
The select input array [SEL_I(7..0)] indicates where valid data is placed on the [DAT_I(63..0)] signal array during WRITE cycles, and where it should be present on the [DAT_O(63..0)] signal array during READ cycles. Also see the [DAT_I(63..0)], [DAT_O(63..0)] and [STB_I] signal descriptions.

**STB_I**
The strobe input [STB_I] indicates a valid data transfer cycle. It is used to qualify various other signals on the interface such as [SEL_I(7..0)]. The SLAVE must assert either the [ACK_O], [ERR_O] or [RTY_O] signals in response to every assertion of the [STB_I] signal.

**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 – Bus Interface

The bus interface is described in terms of its general operation, reset operation, handshaking protocol, bus cycles and the data organization during transfers. Additional requirements for the bus interface (especially those relating to [CLK_I]) can be found in the timing specifications in Chapter 4.

## 3.1 General Operation

Each MASTER and SLAVE are interconnected with a set of signals that permit them to exchange data. For descriptive purposes this interconnection is called a *bus*. Address, data and other information is impressed upon this bus in the form of *bus cycles*.

**RULE 3.05**
The WISHBONE DATASHEET MUST indicate whether it is a MASTER, SLAVE or SYSCON interface. Furthermore, it MUST indicate the types of bus cycles it supports.

### 3.1.1 Reset Operation

All hardware must be initialized to a pre-defined state. This is accomplished with the reset signal [RST_O]. This signal can be asserted at anytime. It is also used for test simulation purposes by initializing all self-starting state machines and counters which may be used in the interface. The reset signal [RST_O] is driven by the SYSCON functional module. It is connected to the [RST_I] on all MASTER and SLAVE modules.

**RULE 3.10**
MASTER and SLAVE interfaces MUST initialize themselves after the assertion of [RST_I].

**RULE 3.20**
[RST_I] MUST be asserted for at least one complete clock cycle on all MASTER and SLAVE interfaces.

**RULE 3.30**
All MASTER and SLAVE modules MUST be capable of reacting to [RST_I] at any time.

**RULE 3.40**
Self-starting state machines and counters on MASTER and SLAVE modules MUST initialize themselves to a pre-defined state after the assertion of [RST_I].

**OBSERVATION 3.10**
In general, a self-starting state machine does 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. The initialization rule prevents this problem by forcing the state machine to a pre-defined state.

**RULE 3.50**
The following MASTER signals MUST be negated after the assertion of [RST_I]: [STB_O], [CYC_O]. The state of all other MASTER signals are undefined.

**OBSERVATION 3.11**
SLAVES will automatically negate [ACK_O], [ERR_O] and [RTY_O] when all MASTERs negate [STB_O].

**RECOMENDATION 3.11**
Design reset generators to assert [RST_O] after a power-up condition.

### 3.1.2 Handshaking Protocol

All bus cycles use a handshaking protocol between the MASTER and SLAVE interfaces. As shown in Figure 3-1, 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. Both sides of the interface can then completely control the rate at which data is transferred. If the SLAVE always operates at the maximum speed of the core, and if the [ERR_I] and [RTY_I] signals are not used, then the [ACK_I] signal may be tied 'high'. The interface will function normally under these circumstances.

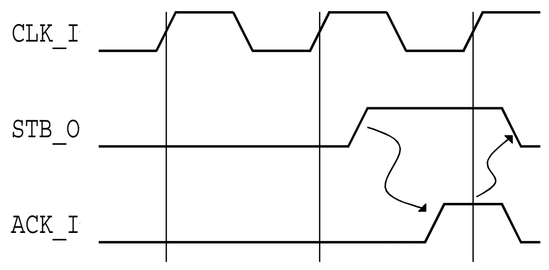


Figure 3-1. Local bus handshaking protocol.

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 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 would assert [RTY_I] if the local resource is busy. This specification does not dictate when or how the MASTER will respond to [RTY_I].

The simplest form of the WISHBONE interconnect is the EVENT cycle. In this case only the handshaking signals are present. To indicate an event the MASTER asserts [STB_O], and the slave reacts by asserting [ACK_O].

**RULE 3.60**
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] and [RST_I]. All other signals are optional.

**PERMISSION 3.10**
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].

**RULE 3.70**
If a MASTER supports the [ERR_I] signal, then the WISHBONE DATASHEET MUST describe how it reacts in response to the signal. If a SLAVE supports the [ERR_O] signal, then the WISHBONE DATASHEET MUST describe the conditions under which the signal is generated.

**PERMISSION 3.20**
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 3.80**
If a MASTER supports the [RTY_I] signal, then the WISHBONE DATASHEET MUST describe how it reacts in response to the signal.

**RULE 3.90**
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].

**RULE 3.100**
MASTER and SLAVE interfaces MUST be designed so that there are no intermediate logic gates between a registered flip-flop and the signal outputs on: [STB_O] and [CYC_O].

**OBSERVATION 3.20**
The WISHBONE interface can be designed so that there are no intermediate logic gates between a registered flip-flop and the signal output. Delay timing for [STB_O] and [CYC_O] are very often the most critical paths in the system. This rule prevents sloppy design practices from slowing down the interconnect because of added delays on these two signals.

**RULE 3.110**
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]. Furthermore, this activity MUST occur asynchronous to the [CLK_I] signal (i.e. there is a combinatorial logic path between [STB_I] and [ACK_O], etc.).

**OBSERVATION 3.30**
The asynchronous logic requirement assures that the interface can accomplish one data transfer per clock cycle. Furthermore, it simplifies the design of arbiters in multi-MASTER applications.

**PERMISSION 3.30**
Under certain circumstances SLAVE interfaces MAY be designed to hold [ACK_O] in the asserted state. This situation occurs when there is a single SLAVE on the interface, and that SLAVE always operates without wait states. In this case, the MASTER will assert the [STB_O] signal for one clock cycle.

**RULE 3.130**
MASTER interfaces MUST be designed to operate normally when SLAVE interface holds [ACK_I] in the asserted state.

### 3.1.3 Use of [STB_O]

**RULE 3.140**
MASTER interfaces MUST qualify the following signals with [STB_O]: [ADR_O], [DAT_O()], [SEL_O()], [WE_O], [SEL_O] and [TAGN_O].

**RULE 3.150**
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].

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

**RULE 3.160**
SLAVE interfaces MUST qualify the following signals with [ACK_O], [ERR_O] or [RTY_O]: [DAT_O()].

### 3.1.5 Use of [TAGN_I] and [TAGN_O] Signals

The TAG signals [TAGN_I] and [TAGN_O] are used by both MASTER and SLAVE modules. They are used for three purposes: (a) to tag data with information such as parity or time stamps, (b) to identify specialty bus cycles (like interrupts or cache control operations) and (c) to communicate with the bus interconnection. These signals are user defined.

For example, the designer of a MASTER module may wish to add parity check bits to its bus cycle. In this case a [TAGN_O] signal is defined by the IP core designer, and logic would be created to generate the bit. Furthermore, the signal would be described in the WISHBONE DATASHEET.

In another example, the designer of a SLAVE module may wish to notify the bus interconnection logic with the size of it's data interface. In this case a [TAGN_O] signal is defined by the IP core designer, and logic would be created to reflect the bus size. The signal would also be described in the WISHBONE DATASHEET.

**RULE 3.180**
All interfaces that support the [TAGN_I] or [TAGN_O] signal(s) MUST describe their use in the WISHBONE DATASHEET.

**RULE 3.181**
The [TAGN_I] and [TAGN_O] signals MUST adhere to the timing specifications given in this document.

## 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.

**RULE 3.190**
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 3.40**
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 3-2 shows a SINGLE READ cycle. The bus protocol works as follows:

CLOCK EDGE 0: MASTER presents [ADR_O()] and [TAGN_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] to indicate the start of the cycle.
MASTER asserts [STB_O] to qualify [ADR_O()], [SEL_O()] and [WE_O].

SETUP, EDGE 1: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE presents valid data on [DAT_I()].
SLAVE asserts [ACK_I] in response to [STB_O] to indicate valid data.
SLAVE presents [TAGN_O].
MASTER monitors [TAGN_I].
MASTER monitors [ACK_I], and prepares to latch data on [DAT_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()].
MASTER latches [TAGN_I].
MASTER negates [STB_O] and [CYC_O] to indicate the end of the cycle.

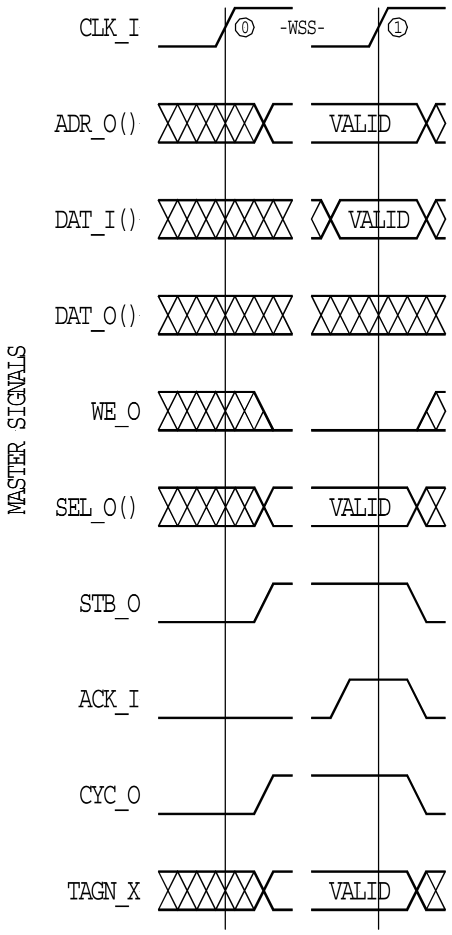


Figure 3-2. SINGLE READ cycle.

## 3.2.2 SINGLE WRITE Cycle

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

| | |
|---|---|
| CLOCK EDGE 0: | MASTER presents [ADR_O()] and [TAGN_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] to indicate the start of the cycle. |
| | MASTER asserts [STB_O] to qualify [ADR_O()], [SEL_O()] and [WE_O]. |
| SETUP, EDGE 1: | SLAVE decides inputs, and responds by asserting [ACK_I]. |
| | SLAVE presents prepares to latch data on [DAT_O()]. |
| | SLAVE asserts [ACK_I] in response to [STB_O] to indicate latched data. |
| | SLAVE presents [TAGN_O]. |
| | MASTER monitors [TAGN_I]. |
| | 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()]. |
| | MASTER latches [TAGN_I]. |
| | MASTER negates [STB_O] and [CYC_O] to indicate the end of the cycle. |

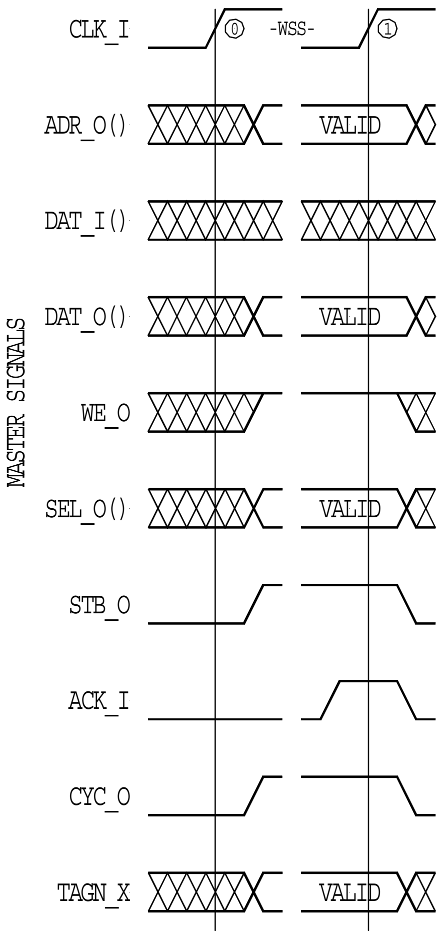


Figure 3-3.  SINGLE WRITE cycle.

## 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 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 3-4, the [CYC_O] signal is asserted for the duration of a BLOCK cycle. This signal can be used to request permission to access from a shared resource from a local arbiter, and hold the access until the end of the current cycle. During each of the data transfers (within the block transfer), the normal handshaking protocol between [STB_O] and [ACK_I] is maintained.

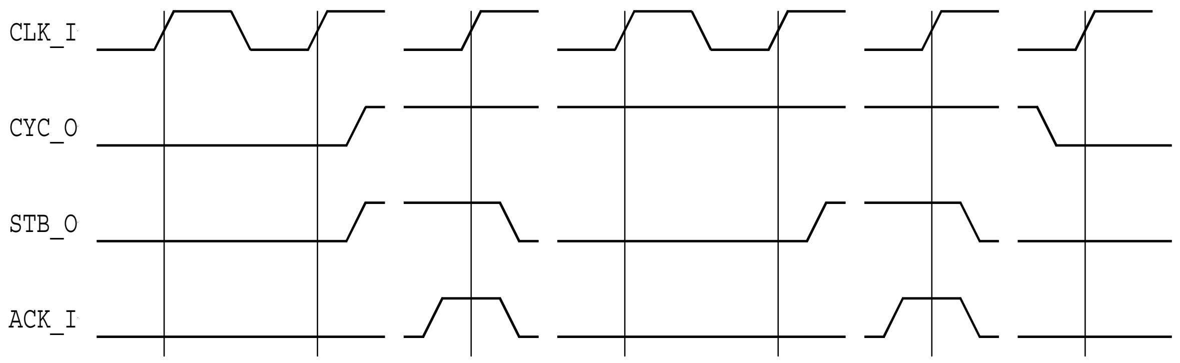


Figure 3-4. Use of [CYC_O] signal during BLOCK cycles.

It should be noted that the [CYC_O] signal does not necessarily rise and fall at the same time as [STB_O]. [CYC_O] may be asserted at the same time as [CYC_O], or one or more [CLK_I] edges before [CYC_O]. Similarly, [CYC_O] may be negated at the same time as [STB_O], or after an indeterminate number of clock cycles.

**RULE 3.200**
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 3.50**
MASTER and SLAVE interfaces MAY be designed so that they do not support the BLOCK cycles.

### 3.3.1 BLOCK READ Cycle

Figure 3-5 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 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 [TAGN_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] to indicate the start of the cycle.
MASTER asserts [STB_O].

Note: the MASTER must assert [CYC_O] and/or [TAGN_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].
SLAVE presents [TAGN_O].
MASTER monitors [TAGN_I].
MASTER monitors [ACK_I], and prepares to latch data on [DAT_I()].

CLOCK EDGE 1: MASTER latches data on [DAT_I()].
MASTER latches [TAGN_I].
MASTER presents new [ADR_O()] and [TAGN_O].
MASTER presents new bank select [SEL_O()] to indicate where data is.

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

CLOCK EDGE 2: MASTER latches data on [DAT_I()].
MASTER latches [TAGN_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 at this point.

CLOCK EDGE 3: MASTER presents new [ADR_O()] and [TAGN_O].
MASTER presents new bank select [SEL_O()].

MASTER asserts [STB_O].

SETUP, EDGE 4: SLAVE decodes inputs, and responds by asserting [ACK_I].
SLAVE presents valid data on [DAT_I].
SLAVE presents [TAGN_O].
MASTER monitors [TAGN_I].
MASTER monitors [ACK_I], and prepares to latch data on [DAT_I()].

CLOCK EDGE 4: MASTER latches data on [DAT_I()].
MASTER presents [ADR_O()] and [TAGN_O].
MASTER latches [TAGN_I].
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].
SLAVE presents [TAGN_O].
MASTER monitors [TAGN_I].
MASTER monitors [ACK_I], and prepares to latch data on [DAT_I()].

CLOCK EDGE 5: MASTER latches data on [DAT_I()].
MASTER latches [TAGN_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].
MASTER monitors [ACK_I], and prepares to latch data on [DAT_I()].

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

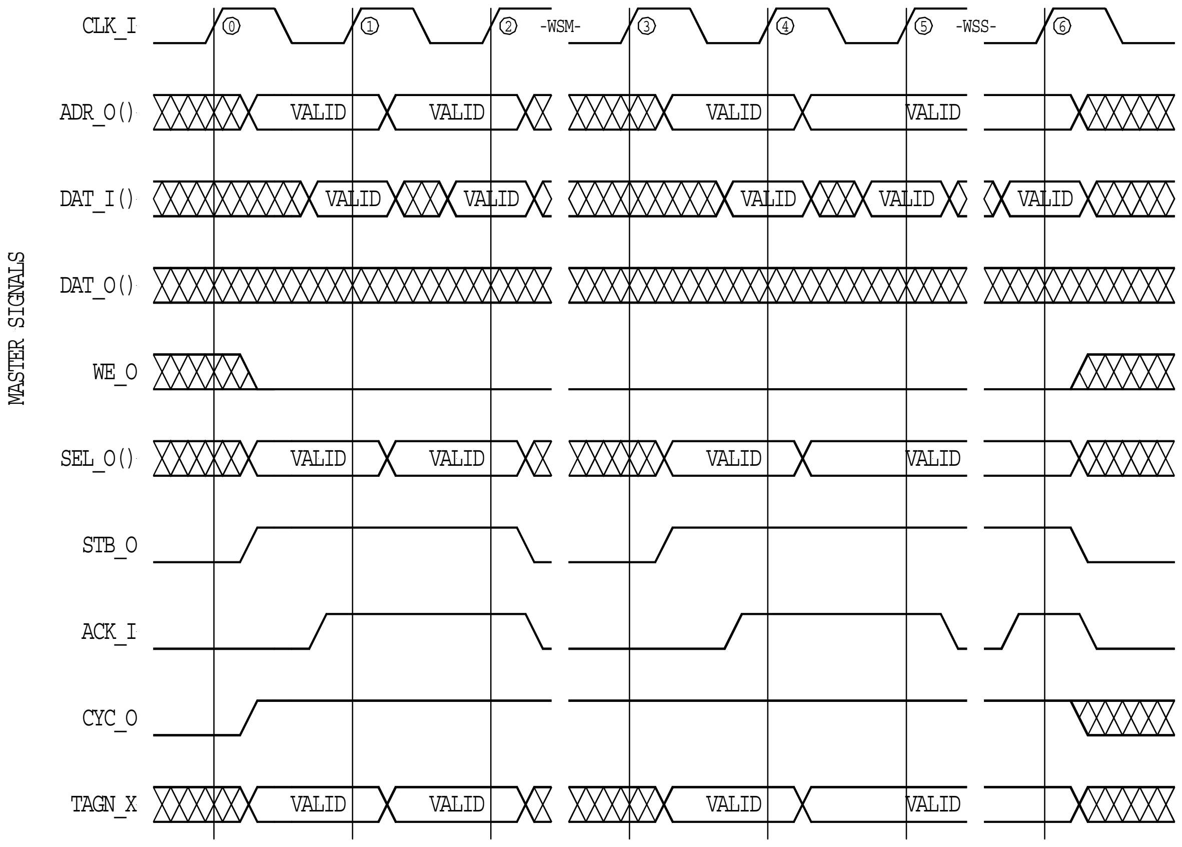


Figure 3-5. BLOCK READ cycle.

### 3.3.2 BLOCK WRITE Cycle

Figure 3-6 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 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 [TAGN_O].
MASTER asserts [WE_O] to indicate a WRITE cycle.
MASTER presents bank select [SEL_O()] to indicate where it expects data.
MASTER asserts [CYC_O] and [TAGN_O] to indicate cycle start.
MASTER asserts [STB_O].

Note: the MASTER must assert [CYC_O] and/or [TAGN_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 prepares to latch data on [DAT_O].
SLAVE presents [TAGN_O].
MASTER monitors [TAGN_I].
MASTER monitors [ACK_I], and prepares to terminate current data phase.

CLOCK EDGE 1: SLAVE latches data on [DAT_O()].
MASTER latches [TAGN_I].
MASTER presents [ADR_O()] and [TAGN_O].
MASTER presents new bank select [SEL_O()].

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

CLOCK EDGE 2: SLAVE latches data on [DAT_O()].
MASTER latches [TAGN_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 at this point.

CLOCK EDGE 3: MASTER presents [ADR_O()] and [TAGN_O].
MASTER presents 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 prepares to latch data on [DAT_O].
SLAVE presents [TAGN_O].
MASTER monitors [TAGN_I].
MASTER monitors [ACK_I], and prepares to terminate data phase.

CLOCK EDGE 4: SLAVE latches data on [DAT_O()].
MASTER latches [TAGN_I].
MASTER presents [ADR_O()] and [TAGN_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 prepares to latch data on [DAT_O].
SLAVE presents [TAGN_O].
MASTER monitors [TAGN_I].
MASTER monitors [ACK_I], and prepares to terminate data phase.

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

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].
MASTER monitors [ACK_I], and prepares to terminate data phase.

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

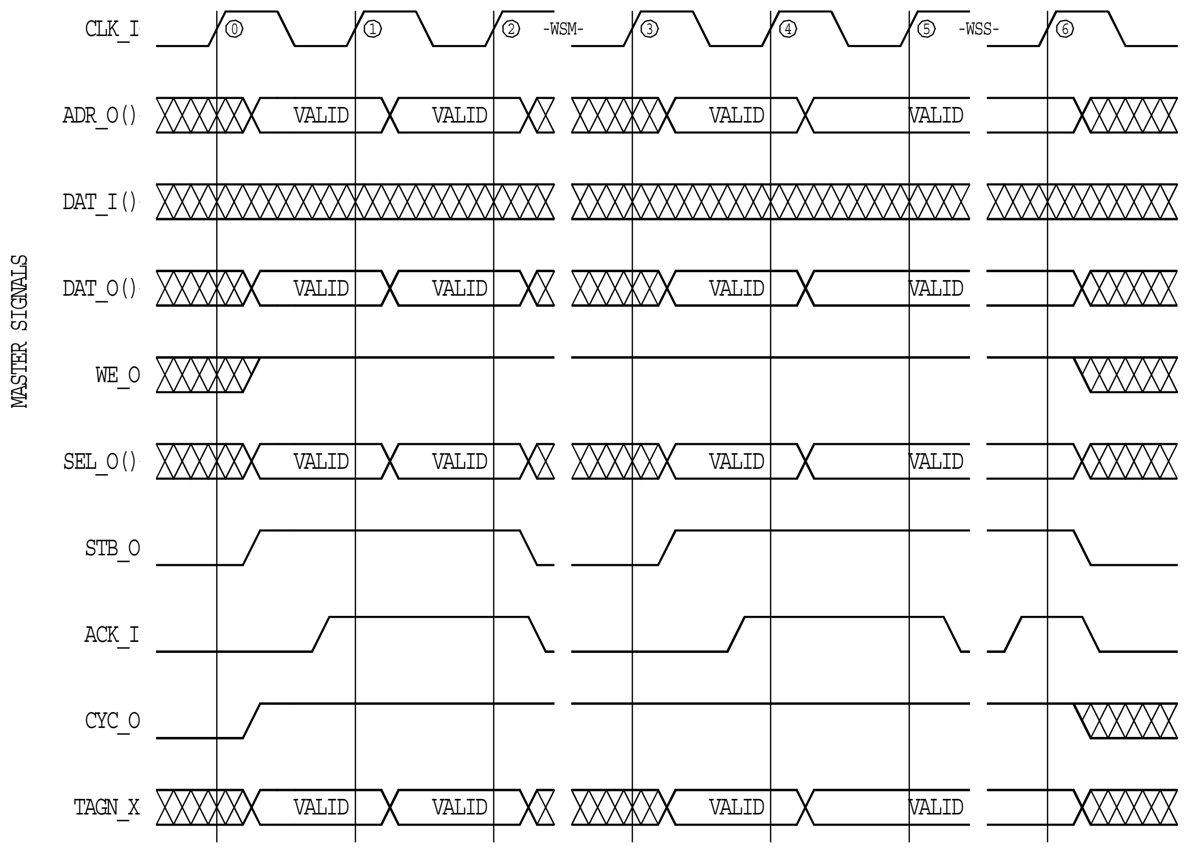


Figure 3-6. BLOCK WRITE cycle.

## 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 3.210**
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 3-7 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 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 [TAGN_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 [TAGN_O] to indicate the start of cycle.
MASTER asserts [STB_O].

Note: the MASTER must assert [CYC_O] and/or [TAGN_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].
SLAVE presents [TAGN_O].
MASTER monitors [TAGN_I].
MASTER monitors [ACK_I], and prepares to latch data on [DAT_I()].

CLOCK EDGE 1: MASTER latches data on [DAT_I()].
MASTER latches [TAGN_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 the same [ADR_O()] and [TAGN_O] as was on clock 1.
MASTER presents WRITE data on [DAT_O()].
MASTER presents new bank select [SEL_O()].
MASTER asserts [STB_O].

SETUP, EDGE 3: SLAVE decodes inputs, and responds by asserting [ACK_I] (when ready).
SLAVE presents valid data on [DAT_I].
SLAVE presents [TAGN_O].
MASTER monitors [TAGN_I].
MASTER monitors [ACK_I], and prepares to latch data on [DAT_I()].

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

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

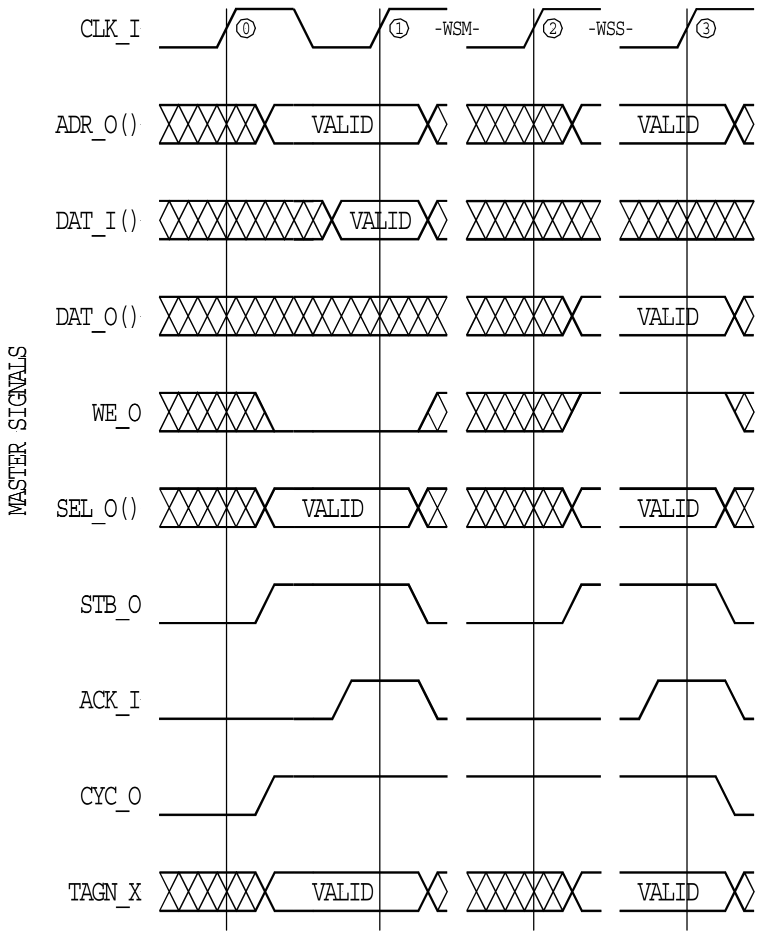


Figure 3-7. RMW cycle.

## 3.5 Data Organization

Data organization refers to the ordering of data during transfers. There are two general types of ordering which 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 3-8 shows the operand locations for input and output data ports.

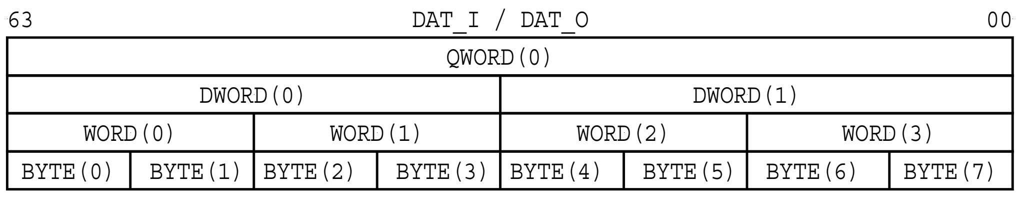
Table 3-1. Data transfer nomenclature.

| 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'. |

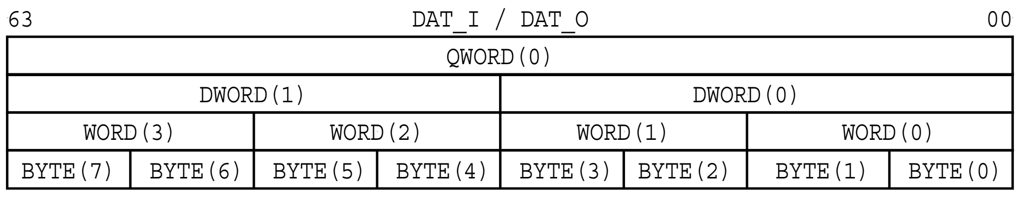
The table also defines the granularity of the interface. This indicates the minimum unit of data transfer that is supported by the interface. For example, the smallest operand that can be passed through a port with 16-bit granularity is a 16-bit WORD. In this case, an 8-bit operand cannot be transferred.

Figure 3-9 shows an example of how the 64-bit value of 0x0123456789ABC 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 / DAT_O(63). The least significant bit is at DAT_I / 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.

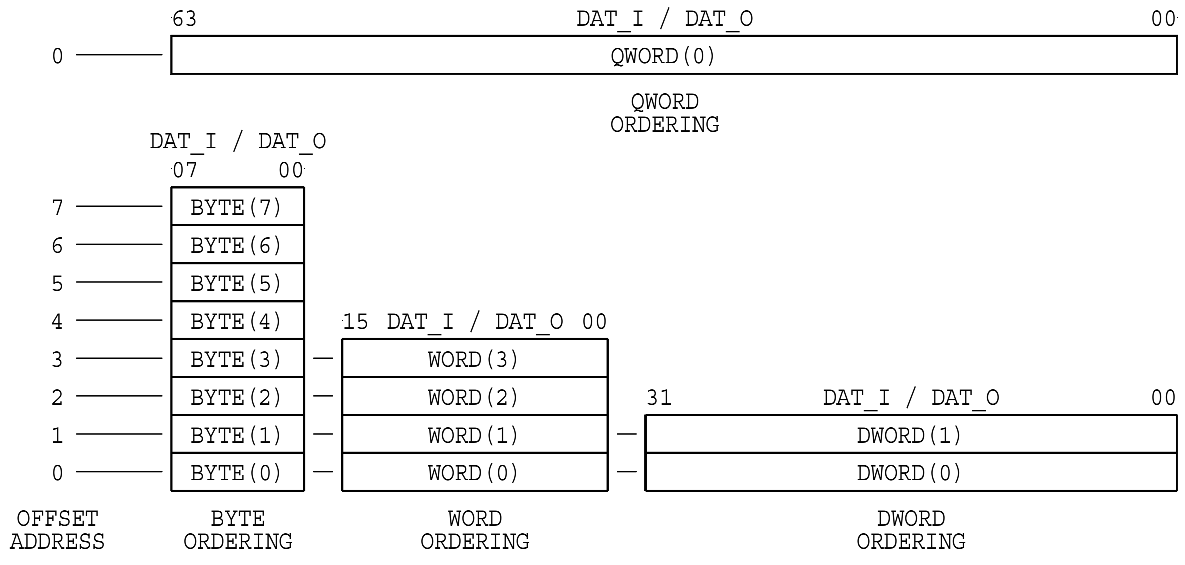
Figure 3-10 shows an example of how the 64-bit value of 0x0123456789ABC is transferred through BYTE, WORD, DWORD and QWORD ports using LITTLE ENDIAN data organization. Through the 64-bit QWORD port the number is directly transferred with the most significant bit at DAT_I / DAT_O(63). The least significant bit is at DAT_I / 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.



(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 3-8. Operand locations for input and output data ports.

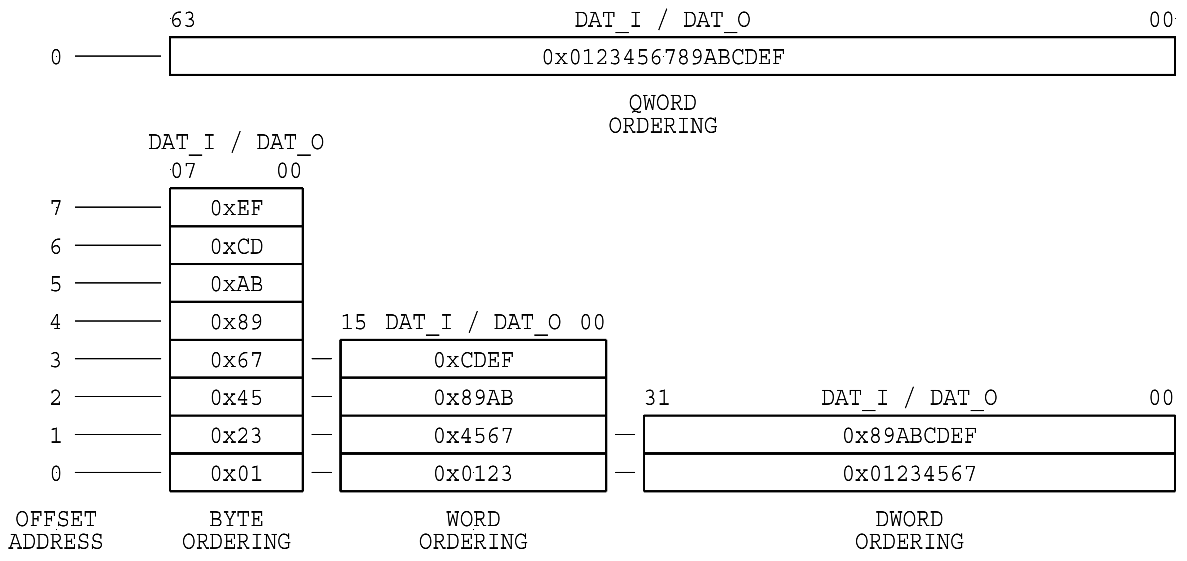


Figure 3-9. Example showing a variety of BIG ENDIAN transfers over various port sizes.

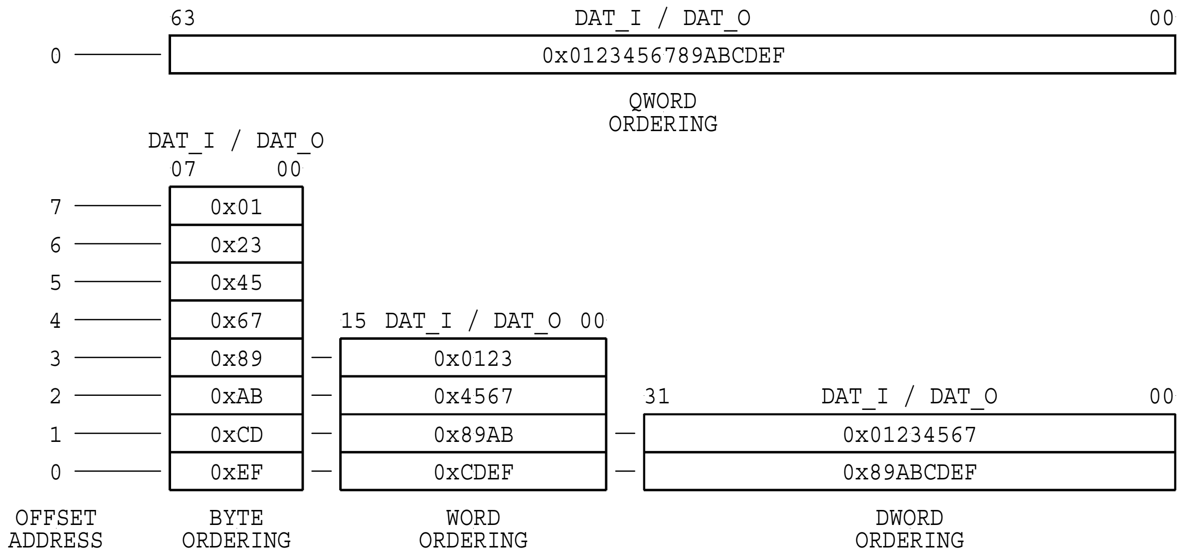


Figure 3-10. Example showing a variety of LITTLE ENDIAN transfers over various port sizes.

**RULE 3.220**
Data organization MUST conform to the ordering indicated in Figure 3-8.

**RULE 3.230**
The WISHBONE DATASHEET MUST indicate the port size. The port size MUST be indicated as: 8-bit, 16-bit, 32-bit or 64-bit.

**RULE 3.235**
The WISHBONE DATASHEET MUST indicate the port granularity. The granularity MUST be indicated as: 8-bit, 16-bit, 32-bit or 64-bit.

**RULE 3.237**
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.

**PERMISSION 3.35**
In some cases the maximum operand size is unknown. In those cases, the maximum operand size shall be the same as the granularity.

**RULE 3.240**
The WISHBONE DATASHEET MUST indicate the data transfer ordering. The ordering MUST be indicated as BIG ENDIAN or LITTLE ENDIAN.

**PERMISSION 3.70**
When the port size equals the granularity, then the interface may be specified as BIG ENDIAN and/or LITTLE ENDIAN.

**OBSERVATION 3.40**
When the port size equals the granularity, then BIG ENDIAN and LITTLE ENDIAN transfers are identical.

### 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 3.05**
Design interfaces so that data is transferred sequentially from lower addresses to a higher addresses.

**OBSERVATION 3.50**
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.

**RULE 2.45**
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 that it is undefined.

### 3.5.3 Data Organization for 64-bit Ports

**RULE 3.250**
Data organization on 64-bit ports MUST conform to Figure 3-11.

| 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 3-11. Data organization for 64-bit ports.

## 3.5.4 Data Organization for 32-bit Ports

**RULE 3.260**
Data organization on 32-bit ports MUST conform to Figure 3-12.

| 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 3-12. Data organization for 32-bit ports.

### 3.5.5 Data Organization for 16-bit Ports

**RULE 3.270**
Data organization on 16-bit ports MUST conform to Figure 3-13.

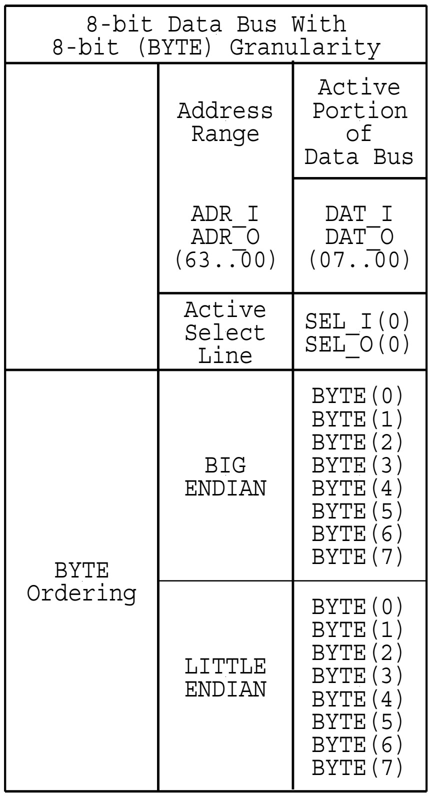
| 16-bit Data Bus With<br>8-bit (BYTE) Granularity | | | |
|---|---|---|---|
| | Address<br>Range | Active Portion<br>of Data Bus | |
| | ADR_I<br>ADR_O<br>(63..01) | DAT_I<br>DAT_O<br>(15..08) | DAT_I<br>DAT_O<br>(07..00) |
| | Active<br>Select<br>Line | SEL_I(1)<br>SEL_O(1) | SEL_I(0)<br>SEL_O(0) |
| BYTE<br>Ordering | BIG<br>ENDIAN | BYTE(0)<br>BYTE(2)<br>BYTE(4)<br>BYTE(6) | BYTE(1)<br>BYTE(3)<br>BYTE(5)<br>BYTE(7) |
| | LITTLE<br>ENDIAN | BYTE(1)<br>BYTE(3)<br>BYTE(5)<br>BYTE(7) | BYTE(0)<br>BYTE(2)<br>BYTE(4)<br>BYTE(6) |

| 16-bit Data Bus With<br>16-bit (WORD) Granularity | | |
|---|---|---|
| | Address<br>Range | Active Portion<br>of Data Bus |
| | ADR_I<br>ADR_O<br>(63..00) | DAT_I<br>DAT_O<br>(15..00) |
| | Active<br>Select<br>Line | SEL_I(0)<br>SEL_O(0) |
| WORD<br>Ordering | BIG<br>ENDIAN | WORD(0)<br>WORD(1)<br>WORD(2)<br>WORD(3) |
| | LITTLE<br>ENDIAN | WORD(0)<br>WORD(1)<br>WORD(2)<br>WORD(3) |

Figure 3-13. Data organization for 16-bit ports.

## 3.5.6 Data Organization for 8-bit Ports

**RULE 3.280**
Data organization on 8-bit ports MUST conform to Figure 3-14.



| 8-bit Data Bus With 8-bit (BYTE) Granularity | | |
|---|---|---|
| | Address Range | Active Portion of Data Bus |
| | ADR_I<br>ADR_O<br>(63..00) | DAT_I<br>DAT_O<br>(07..00) |
| | Active Select Line | SEL_I(0)<br>SEL_O(0) |
| BYTE Ordering | BIG ENDIAN | BYTE(0)<br>BYTE(1)<br>BYTE(2)<br>BYTE(3)<br>BYTE(4)<br>BYTE(5)<br>BYTE(6)<br>BYTE(7) |
| | LITTLE ENDIAN | BYTE(0)<br>BYTE(1)<br>BYTE(2)<br>BYTE(3)<br>BYTE(4)<br>BYTE(5)<br>BYTE(6)<br>BYTE(7) |

Figure 3-14. Data organization for 8-bit ports.

# Chapter 4 – 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 4-1, and is defined as Tpd,clk-su.

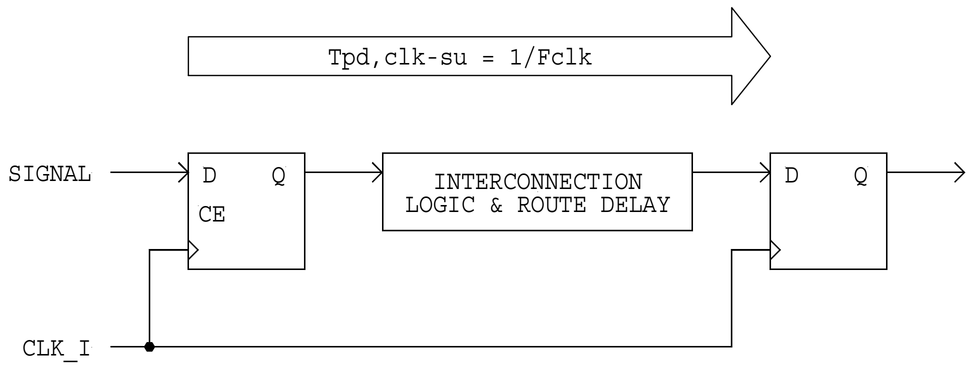


Figure 4-1. Definition for Tpd,clk-su.

**RULE 4.10**
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 4.10**
The user's place and route tool MAY be used to enforce this rule.

**OBSERVATION 4.10**
Most place and route tools can be easily configured to enforce this rule. Generally, it only requires a single timing specification for Tpd,clk-su.

**RULE 4.20**
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 4.20**
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 4.30**
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 4.20**
The IP core connected to a WISHBONE interface MAY include application specific timing requirements.

**RULE 4.30**
The clock input [CLK_I] MUST have a duty cycle that is no less than 40%, and no greater than 60%.

**SUGGESTION 4.1**
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.

**RULE 4.40**
The WISHBONE DATASHEET MUST indicate if the SoC component has any constraints on its [CLK_I] or [CLK_O] signal.

# Chapter 5 – Application Interface

This chapter describes various methods and issues for WISHBONE interconnection. In some cases, the interconnection requires no glue logic. In more sophisticated systems, the user may need to add this logic. However, the common interface between the system-on-chip components will make this task much simpler.

## 5.1 Interconnection Methods

There are four general ways to interconnect MASTER and SLAVE IP cores. These include:

- Single MASTER / Single SLAVE interconnection
- Single MASTER / Multiple SLAVE interconnection
- Multiple MASTER interconnection
- Crossbar interconnection

### 5.1.1 Single MASTER / Single SLAVE Interconnection

The application interface may be operated with a single MASTER and a single SLAVE. This interconnection is generally the simplest and the fastest way to integrate IP cores. It is simplest because (in many cases) the MASTER and SLAVE can simply be connected together. Figure 5-1 shows an example of this interconnection. It should be noted that this example assumes that the port widths, port granularities, operand sizes and address widths are equal.

### 5.1.2 Single MASTER / Multiple SLAVE Interconnection

The WISHBONE application interface may be operated with a single MASTER and multiple SLAVEs. Refer to the multiple MASTER configuration (below) for more information.

### 5.1.3 Multiple MASTER Interconnection

The WISHBONE application interface may be operated with multiple MASTER interconnections. Figure 5-2 shows a sample application interface with two WISHBONE MASTERs, and two WISHBONE SLAVEs. For purposes of clarity, only the [CYC_O], [STB_O] and [ACK_I] signals are shown. Figure 5-3 shows a sample timing diagram for the same circuit.

At the beginning of a bus cycle a MASTER asserts its [CYC_O] signal. The [CYC_O] signals from the two MASTERs are routed to an arbiter that determines which gets possession of the interconnection. The winning MASTER is identified by the arbiter by the assertion of [GNT1] or [GNT2], which correspond to the first and second masters respectively. The grant signals are then used determine which of the MASTERs will drive the common cycle [COMCYC] and strobe [COMSTB] signals.

The acknowledge signals from the SLAVEs are 'or'ed together to form a common acknowledge signal [COMACK]. In this case, each slave will decode the common address (not shown), and will respond if it is the participating slave in a bus cycle.

The other interconnect signals (e.g. [DAT_I] and [DAT_O]) are not shown in this example. These can be interconnected using multiplexors or three-state buses.

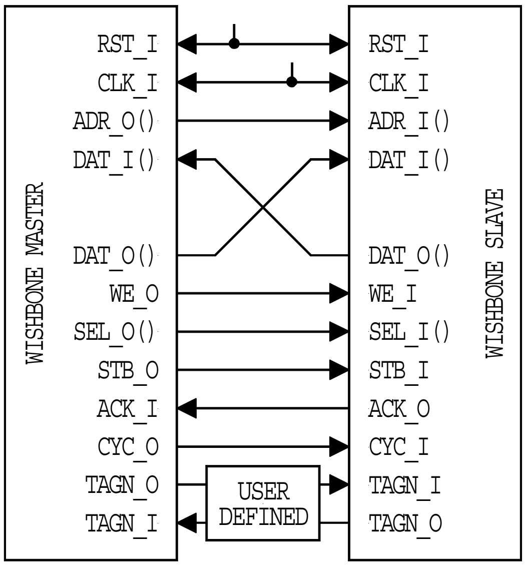


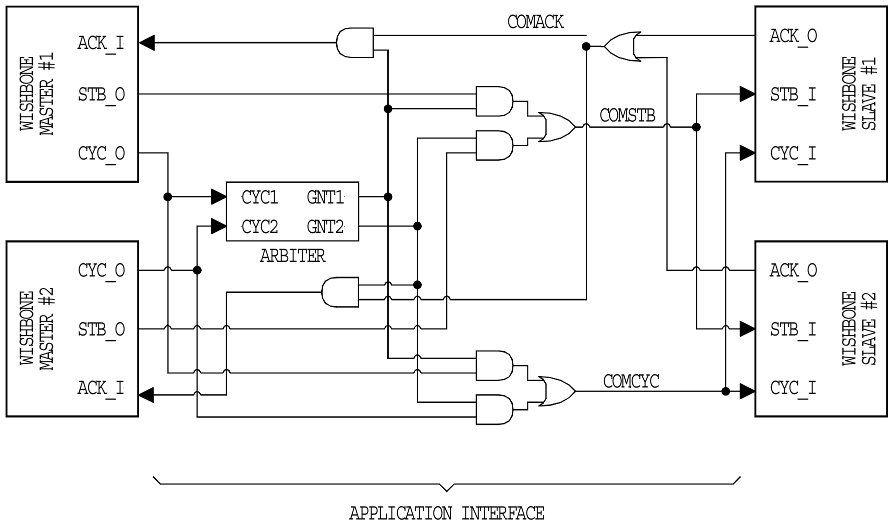
Figure 5-1. Single MASTER / Single SLAVE interconnection example.

### 5.1.4 Crossbar Interconnection

The WISHBONE MASTERs and SLAVEs may be interconnected with a crossbar switch. Crossbar switches are systems that usually have multiple MASTERs and multiple SLAVEs.

Crossbar switches are mechanisms that allow individual pairs of MASTERs and SLAVEs 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.

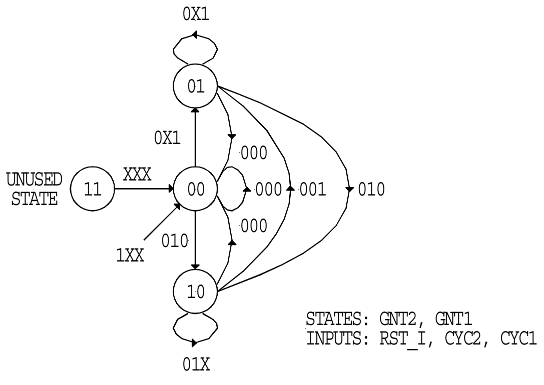
Crossbar routing mechanisms generally support dynamic configurability. This essentially creates a reconfigurable and reliable network system. Most crossbar architectures are also scalable, meaning that families of crossbars can be added as the needs arise.



```
ARBITER EQUATIONS:

GNT1 <= ( not(RST_I) and not(GNT2)                and CYC1 )
     or ( not(RST_I) and not(GNT1) and not(CYC2) and CYC1 );

GNT2 <= ( not(RST_I) and not(GNT2)                and CYC2 and not(CYC1) )
     or ( not(RST_I) and     GNT2  and not(GNT1) and CYC2               );
```



ARBITER STATE DIAGRAM

Figure 5-2. Sample application with two WISHBONE MASTERs, and two WISHBONE SLAVEs.

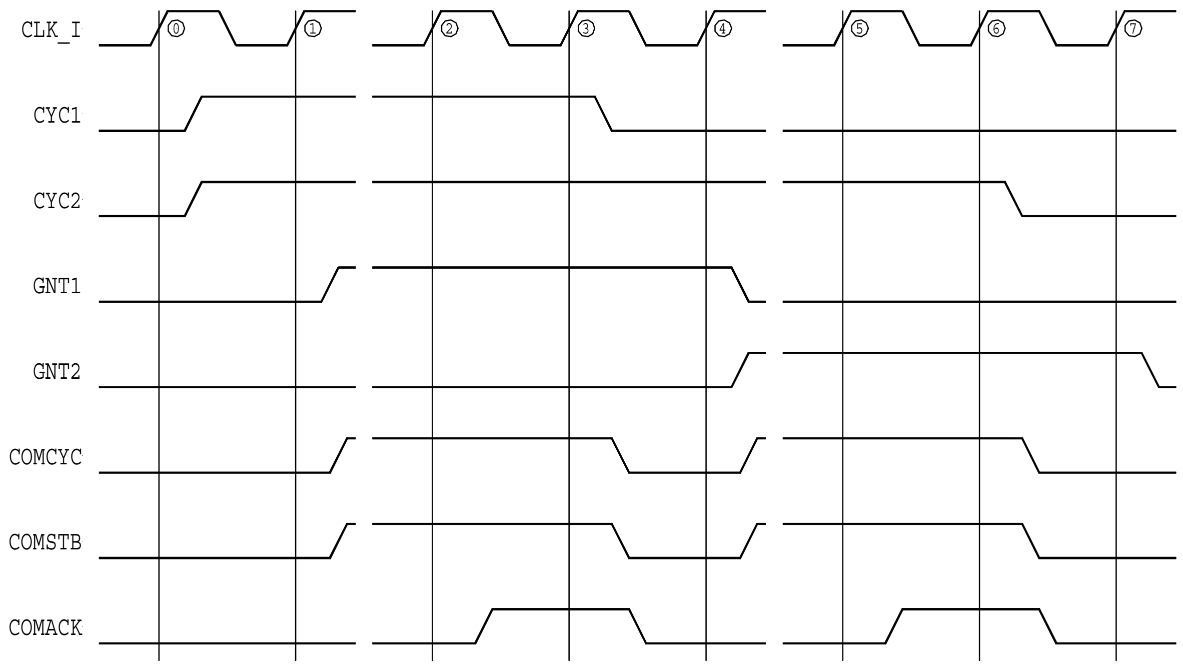


Figure 5-3. Timing diagram for multiple MASTER / multiple SLAVE example.

## 5.2 Three-State Interconnections

The interconnection can take the form of a three-state bus. Figure 5-4 shows the connection of a MASTER or SLAVE data input and output buses to a three-state data bus. Also note that the resistor/current source listed in the figure can also be a 'pull-down' resistor or current source.

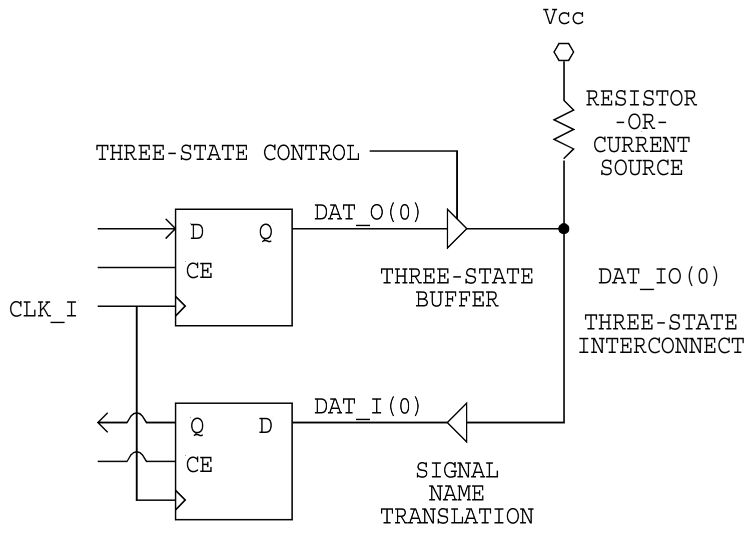


Figure 5-4. Connection of data bus to a three-state interconnection.

## 5.3 Endian Conversion

In some cases the user may wish to connect a BIG ENDIAN IP core to a LITTLE ENDIAN IP core. 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 of the interface. In general, the ENDIAN conversion takes place by renaming the data and select I/O signals at the source or destination IP core.

Figure 5-5 shows a simple example where a 32-bit BIG ENDIAN core output (CORE 'A') is connected to a 32-bit LITTLE ENDIAN core input (CORE 'B'). Both cores have 32-bit operand sizes and 8-bit granularity. 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 by the interconnection wiring between the cores. 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.

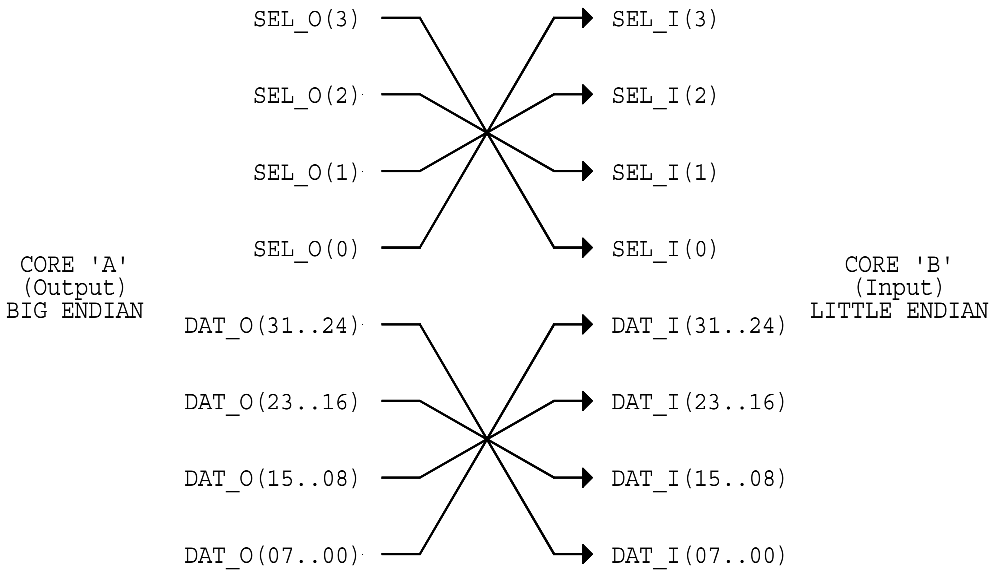


Figure 5-5. Converting a BIG ENDIAN output to a LITTLE ENDIAN input.

# References

[1] 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.]

# Appendix A – WISHBONE Design Philosophy

(This appendix is not part of the WISHBONE specification).

The design philosophy behind the WISHBONE system-on-chip (SoC) architecture is very similar to that found in standard microcomputer buses like PCI and VMEbus. However, there are some important differences between these two approaches. This application note describes some of the fundamental characteristics of the WISHBONE SoC architecture by comparing and contrasting it to standard bus technologies.

This method for describing WISHBONE is useful because SoC often requires new ways of thinking about microcomputer bus architectures. For the most part, SoC integration is no different than traditional approaches. However, the design strategies on SoC can be radically different because they are integrated on a single semiconductor device. This is especially true if the SoC interconnection takes full advantage of the technology.

This application note is also useful if the reader is approaching SoC from a background of traditional microcomputer buses like PCI or VMEbus. In this case the reader may find that traditional concepts, such as three-state buses and fixed timing requirements, are not required in SoC architectures like WISHBONE. In some cases, the reader may discover that he or she must learn a 'new way of thinking' about the microcomputer bus problem

This application note attempts to:

- Identify the important differences between standard and SoC bus architectures.
- Discuss where (and why) multiplexed address and data paths are used.
- Discuss where (and why) three-state interconnections are used.
- Discuss the difference between fixed and variable interconnection systems.
- Describe the methodologies for creating timing parameters.
- Present some of the design trade-offs between standard and SoC architectures.
- Discuss testability issues.
- Present some of the solutions afforded by the WISHBONE SoC architecture.

## General Differences in Design Philosophies

There are several distinctions between standard and SoC bus architectures. These include:

- Standard microcomputer buses are always ‘pin limited’, both in terms of connector and IC package pins. This often requires strategies such as three-state logic and multiplexed address and data lines. For the most part, these requirements do not exist on SoC buses like WISHBONE because they contain a rich set of interconnection (i.e. routing) resources.

- Standard microcomputer buses use a fixed interconnection scheme. That’s because they are usually routed across a standard backplane. This requirement does not exist on SoC buses like WISHBONE because it is very easy to change the interconnection resources. This allows a variable interconnection scheme which can support parallel, crossbar switch and other bus structures.

- Standard microcomputer bus slaves fully decode a fixed address. This requirement does not exist in SoC buses like WISHBONE because the system integrator has the ability to create address decoders that are tailored to his or her application. This reduces redundant logic and also speeds up the system.

- Standard microcomputer buses have fixed timing requirements. That’s because they are both tested as sub-assemblies, and have highly capacitive and inductive loads. Furthermore, they are designed for the worst-case operating conditions when unknown bus modules are connected together. SoC buses like WISHBONE also have timing requirements, but they are defined so that they can be adjusted by the system integrator for best performance. This is accomplished with a variable timing specification that can be enforced with place & route tools.

- Standard microcomputer bus modules are tested at the card level. That’s because each bus module is designed and tested independently. SoC buses like WISHBONE require a higher level of testability because they are tested as a system. This problem is somewhat simplified by the fact that they are interconnected at the more abstract ‘tool’ level. The interconnections can be described either by source code (like VHDL or Verilog® hardware description languages), or with physical layout tools. Both of these solutions require a compatible set of development tools.

## The Pin-limited Design Approach

Standard microcomputer buses like PCI and VMEbus are generally constrained by the number of connector and IC package pins that are available. This constraint leads to two design methodologies:

- Multiplexed address and data buses
- Three-state I/O buffers.

## Multiplexed Address and Data Buses

Multiplexed address and data buses reduce pin count by routing different types of signals over the same set of pins. Figure A-1 shows a common technique where address and data lines share a common set of pins. In this case, a 32-bit address and 32-bit data bus can be combined to form a 32-bit common address/data bus. This reduces the number of signal pins from 64 to 32.

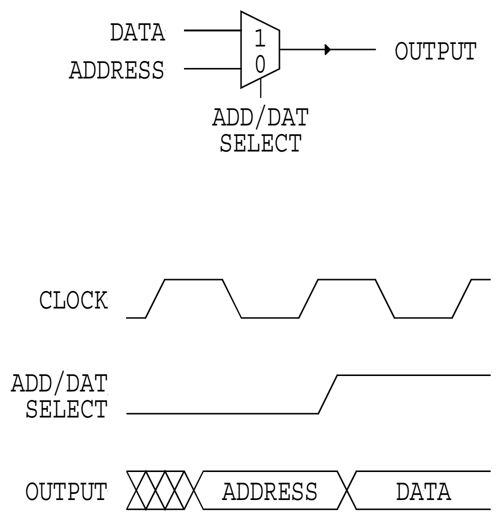


Figure A-1. Circuit and timing diagram for a multiplexed address/data bus.

The major disadvantage of this topology is that it takes twice as long to move the information. Non-multiplexed, synchronous buses 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.

The WISHBONE SoC architecture uses non-multiplexed schemes.

## Three-State I/O vs. Multiplexor Logic

Three-state I/O buffers can also be used to reduce the number of pins required on the interface, and have long been used in the microcomputer board industry. For example, in master-slave architectures, the master that 'owns' the microcomputer bus turns its buffers 'on', and the other master(s) turn their buffers 'off'. [Here, 'on' and 'off' refer to the three-state and non three-state conditions, respectively]. 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, the slave enables its output buffers during read cycles when it has been selected.

Three-state buffers are sometimes called Tri-State® buffers. Tri-State® is a registered trademark of National Semiconductor Corporation.

A simple system that uses three-state I/O buffers is shown in the block diagram of Figure A-2(a). There, the data buses on two master and two slave modules are interconnected with three-state logic. This situation is required in standard microcomputer buses because there generally aren't enough IC package or connector pins to do it any other way.

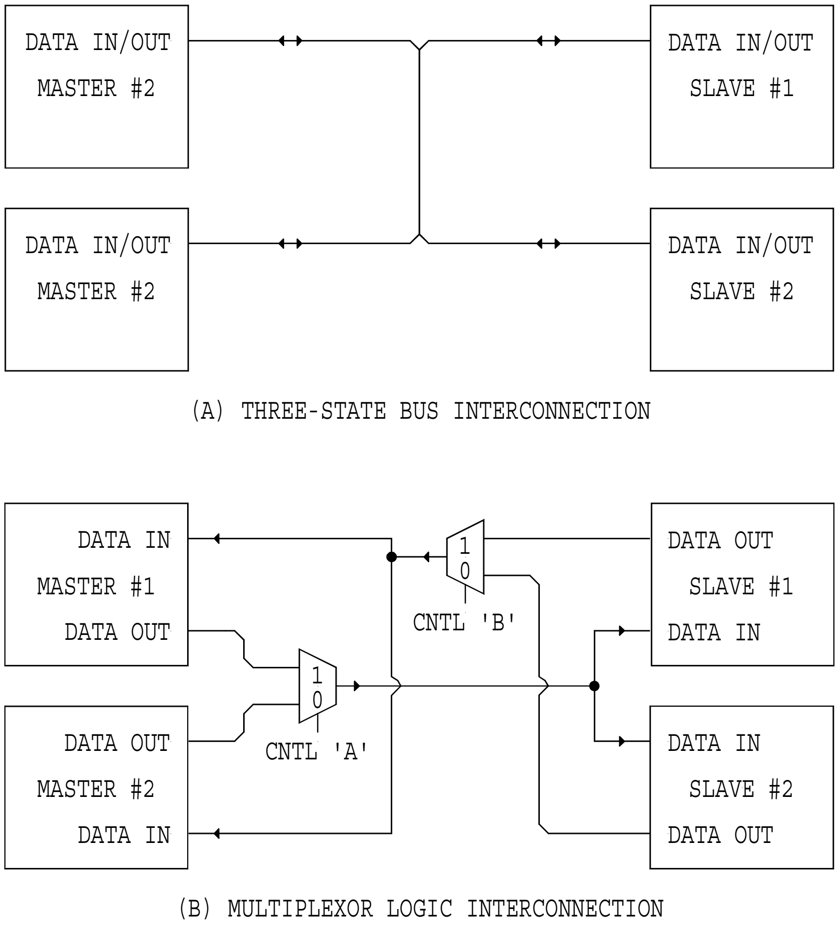


Figure A-2. Three-state bus interconnection vs. multiplexor logic interconnection.

This approach can also be used in the SoC buses. However, the three-state I/O interconnections have two major drawbacks. First, they are inherently slower than direct interconnections (because there are always minimum timing parameters that must be met to turn buffers on-and-off). Second, many IC devices do not have three-state internal routing resources available to them, or they are very restrictive in terms of location or quantity of interconnects.

As shown in Figure A-2(b), the SoC internal bus can use multiplexor logic interconnection to achieve the same goal. The main advantage of this approach is that it does not require the limited three-state routing resources which are available on FPGA and ASIC devices.

The main disadvantage of the multiplexor logic interconnection is that it requires a larger number of routed interconnects and multiplexor logic (which is 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 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 place & route 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.

WISHBONE supports both three-state and multiplexed interconnections. IP cores with WISHBONE interfaces use unidirectional 'in' and 'out' signals, which can easily be converted to three-state I/O by the system integrator. This is because (as we'll see shortly) WISHBONE uses a variable interconnection methodology that allows either approach to be used.

## Interconnection (Routing) Resources

System-on-chip (SoC) architectures have a rich set of interconnections available to them. These are sometimes called *routing resources*. That's because they don't have the connector pin limitations that exists on standard microcomputer bus architectures. For this reason, the SoC approach does not require multiplexed address and data buses, nor three-state I/O buffers.

It can be argued that this approach requires more routing resources in the FPGA and ASIC device. However, practical experience has shown that this is not a serious problem. Furthermore, this approach fosters portability of IP cores. This is because the SoC can be configured over a greater range of FPGA and ASIC target devices.

## Fixed vs. Variable Interconnections

Standard microcomputer buses like PCI and VMEbus use fixed interconnection resources. These generally take the form of connector pin assignments and their related bus cycles and timing parameters. To insure that all bus modules are compatible with each other, the interconnection resource is fixed (on the backplane). The end user cannot change how the bus modules are interconnected.

SoC buses like WISHBONE can use variable interconnection methods. This is because the interconnection bus is defined by the system integrator at the 'tool level'. For example, if the SoC is described with a hardware description language like VHDL or Verilog®, then the end user has the ability to define the interconnection.

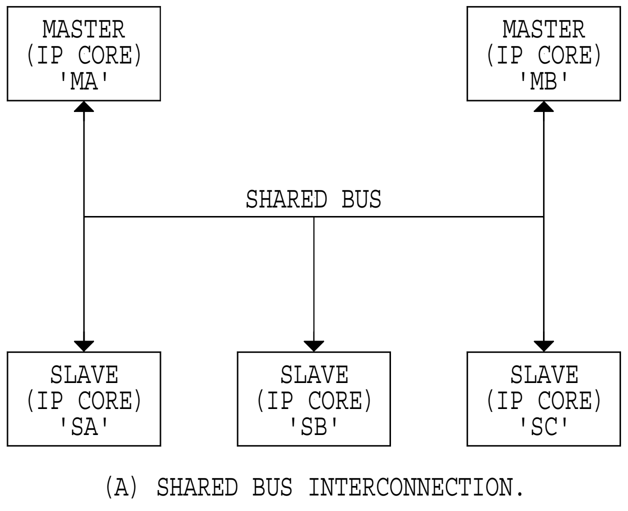
For example, Figure A-3 shows two ways that IP cores can be interconnected. In Figure A-3(a) they are interconnected by a shared bus. This is a typical configuration that one might find in standard buses like PCI or VMEbus. In this configuration each master module arbitrates for the common, shared bus and then communicate with a slave.

Figure A-3(b) shows a radically different approach. This configuration uses a crossbar switch matrix. This is a typical configuration that one might find in microcomputer buses like[4] RACEway, SKY Channel or Myrinet.

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 main advantage of the crossbar switch is that multiple communication paths can operate at the same time. This means that the overall data transfer rate of the system is increased dramatically over the shared bus mechanism. For example, a single communication channel may support 100 Mbyte/sec. However, two parallel communication channels operating simultaneously would support 200 Mbyte/sec. This scheme can be expanded dramatically to support extremely high data transfer rates.

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



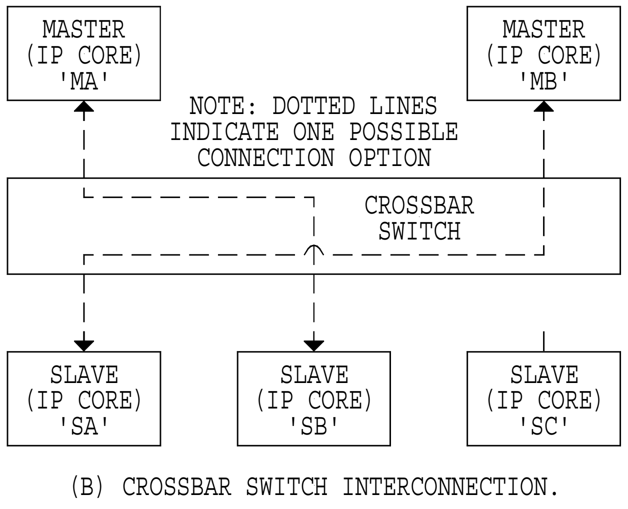


Figure A-3. Two possible methods for IP core interconnection.

A good analogy of the shared and crossbar switch mechanisms can be drawn from a telephone system. The shared bus architecture is much like a 'party line' telephone system, where multiple homes or offices share a single telephone wire. In these systems, only one conversation can happen at any given time.

The crossbar switch operates much like a modern telephone system, where a 'dialing' party can connect to a telephone anywhere else on the system, and over a private communication path. Many such conversations can happen at the same time.

The variable interconnection scheme can support a plethora of other options as well. For example, earlier it was discussed how the interconnection could take the form of a three-state or multiplexor logic interconnection. This choice depends upon the end application and the users' design philosophy.

The variable interconnection scheme also dictates where and how the interconnection bus is specified. Stated another way, the specification determines where to put the 'zippers' in the design. In the case of WISHBONE, the 'zippers' are placed at the IP core interface. The interconnection scheme itself is not inherently part of the specification.

## Full vs. Partial Address Decoding

Standard microcomputer buses like PCI and VMEbus use *full address decoding* on slave modules. 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.

SoC buses like WISHBONE can 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.

For example, consider the serial I/O port (IP core) with the internal register set shown in Figure A-4(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-4(b).

Standard microcomputer buses usually 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 SoC buses this limitation does not exist. SoC buses allow 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.

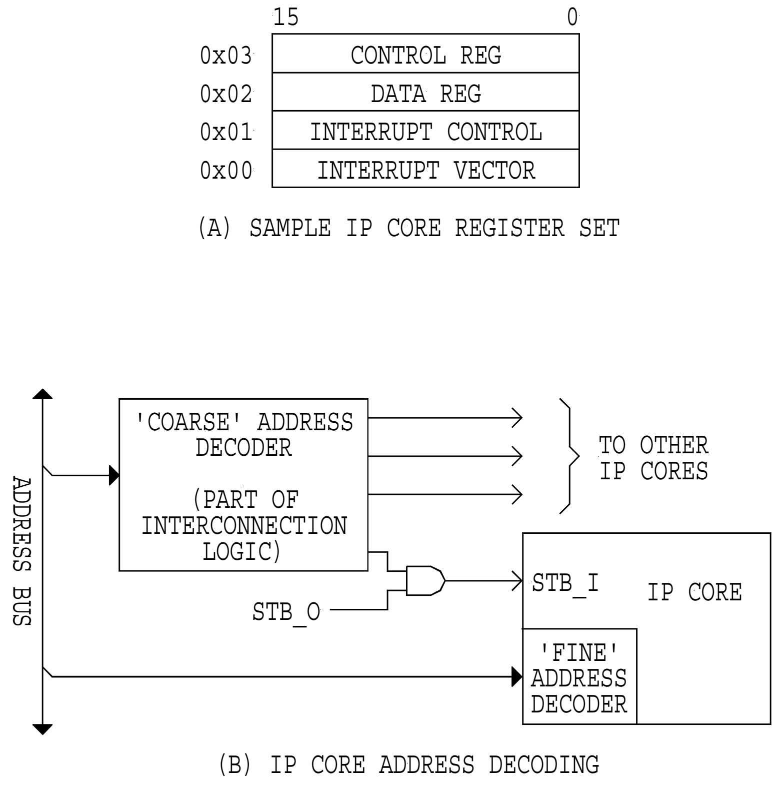


Figure A-4. Partial address decoding technique.

## Timing Specifications

Standard microcomputer buses like PCI or VMEbus use a *fixed timing specification*. Under that method, the exact timing requirements are rigidly enforced by the bus specification.

SoC buses like WISHBONE can use a *variable timing specification*. That means that there are no pre-defined (fixed) timing parameters used on the interconnection. The advantages of this technique are:

- Very simple timing constraints can be enforced with place-and-route tools.
- The SoC integrator can specify the timing requirements of the system.
- The theoretical maximum speed of the system is unlimited.
- The maximum speed is limited only by the target device technology.

The WISHBONE SoC architecture assumes that all logic between the IP cores use an RTL (Register Transfer Logic) design approach. Each RTL section assumes the general configuration shown in Figure A-5. This allows a simple and straightforward set of timing rules to be used. In the case of WISHBONE, the only timing information that is needed 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 defined as Tpd,clk-su.

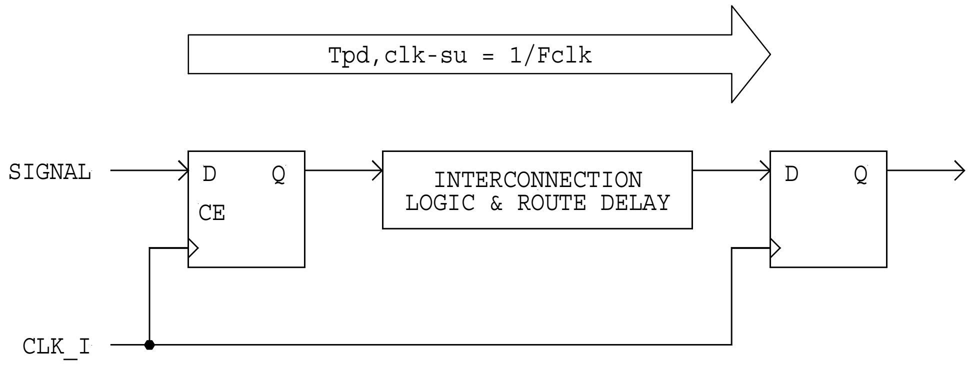


Figure A-5. Definition for Tpd,clk-su.

The WISHBONE specification also requires that the following conditions must be met by all of the IP cores and interconnection logic:

- A central clock [CLK_I] coordinates all activity on the interconnection.
- All output signals are registered at the rising edge of [CLK_I].
- All input signals must be stable before the rising edge of [CLK_I].
- The central clock [CLK_I] must be distributed on a low-skew interconnection.

If these general guidelines are followed, then the variable timing specification can be used.

One advantage to this technique is that very sophisticated systems can be created with only a single timing specification. In the case of WISHBONE, only 'Tpd,clk-su' need be passed to the place-and-route tool. In these cases, the place-and-route tool will enforce the entire system interconnection timing. Another advantage to the technique is that the SoC integrator can specify the maximum speed of the system. This is very useful, as the SoC integrator has a large degree of control over the final performance and cost of the system. Furthermore, as faster technologies appear in the future, the system can be easy scaled to the higher speed devices. These advantages would not be possible if a fixed timing specification were used.

## Testability

Standard microcomputer bus modules like PCI and VMEbus are tested at the card level. That's because each bus module is designed and tested independently. Each card is tested to the common set of standards to insure that they will operate correctly when placed into the final system.

SoC architectures like WISHBONE are generally tested at the system level. That is to say, the system is integrated and then tested as a whole.

These two test approaches are radically different. However, each has its own set of advantages and disadvantages.

SoC buses like WISHBONE require a higher degree of testability because they are tested as a system. This problem is somewhat simplified by the fact that they are interconnected at the more abstract 'tool' level. The interconnections can be described either by source code (like VHDL or Verilog® hardware description languages), or with physical layout tools. Both of these solutions require a compatible set of development tools.

The test approach also varies depending upon the target device technology. For example, ASIC parts generally require a very robust test to insure that the devices operate properly. This is because IC fabrication is a chemical process, and the final test insures that the transistors and other components are all fabricated correctly. Generally, this means supplying test vectors that completely test all of the logic sections within the device. This is a very timing consuming and detailed process.

FPGA parts, on the other hand, require a much lower degree of testability. That's because the individual components within the device can be 100% tested before the circuit is impressed upon them. This means that a lot of shortcuts can be made in the test process without compromising the quality of the final circuit.

The WISHBONE SoC architecture does not specify any testability requirements. That's because test strategies vary considerably throughout the industry.

The author has found that FPGA devices are much more conducive to SoC integration than ASIC devices. That's because the system components (IP cores) can be tested independently of each other. After the system components are integrated, only a rudimentary test bench is required to insure that everything is connected properly. This results in a test that is sufficiently robust to insure high quality devices. Furthermore, the test labor requirement is much lower, thereby improving time-to-market and lowering engineering costs.

## References

Di Giacomo, Joseph. Digital Bus Handbook. McGraw-Hill 1990. ISBN 0-07-016923-3.

# Appendix B - WISHBONE Interface for SLAVE I/O Ports

(This appendix is not part of the WISHBONE specification).

This application note provides several examples of WISHBONE interface for SLAVE I/O ports. The purpose of this application note 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 (if any) 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.

**Simple 8-bit SLAVE Output Port**

Figure B-1 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.

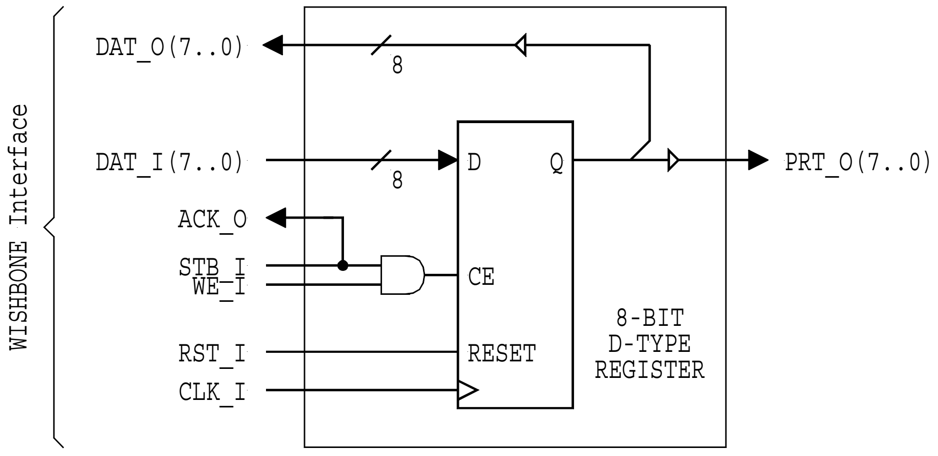


Figure B-1. Simple 8-bit WISHBONE SLAVE output port.

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

| **WISHBONE DATASHEET** | | |
|---|---|---|
| Description | Specification | |
| General description: | 8-bit SLAVE output port. | |
| Supported cycles: | SLAVE, READ/WRITE<br>SLAVE, BLOCK READ/WRITE<br>SLAVE, RMW | |
| Data port, size:<br>Data port, granularity:<br>Data port, maximum operand size:<br>Data transfer ordering:<br>Data transfer sequencing: | 8-bit<br>8-bit<br>8-bit<br>Big endian and/or little endian<br>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 B-2 shows a VHDL implementation of same circuit. The WBOPRT08 entity implements the all of the functions shown in the schematic diagram of Figure B-1.

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

entity WBOPRT08 is
port(
      -- WISHBONE 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:

      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 B-2.  VHDL implementation of the 8-bit output port interface.

**Simple 16-bit SLAVE Output Port With 16-bit Granularity**

Figure B-3 shows a simple 16-bit WISHBONE SLAVE output port. Table B-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.

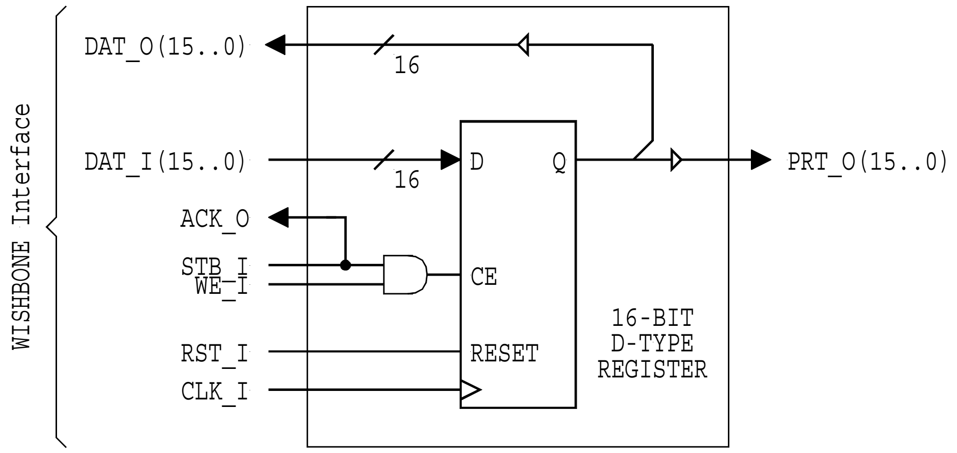


Figure B-3. Simple 16-bit WISHBONE SLAVE output port with 16-bit granularity

Table B-2. WISHBONE DATASHEET for the 16-bit output port with 16-bit granularity.

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

**Simple 16-bit SLAVE Output Port With 8-bit Granularity**

Figure B-4 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.

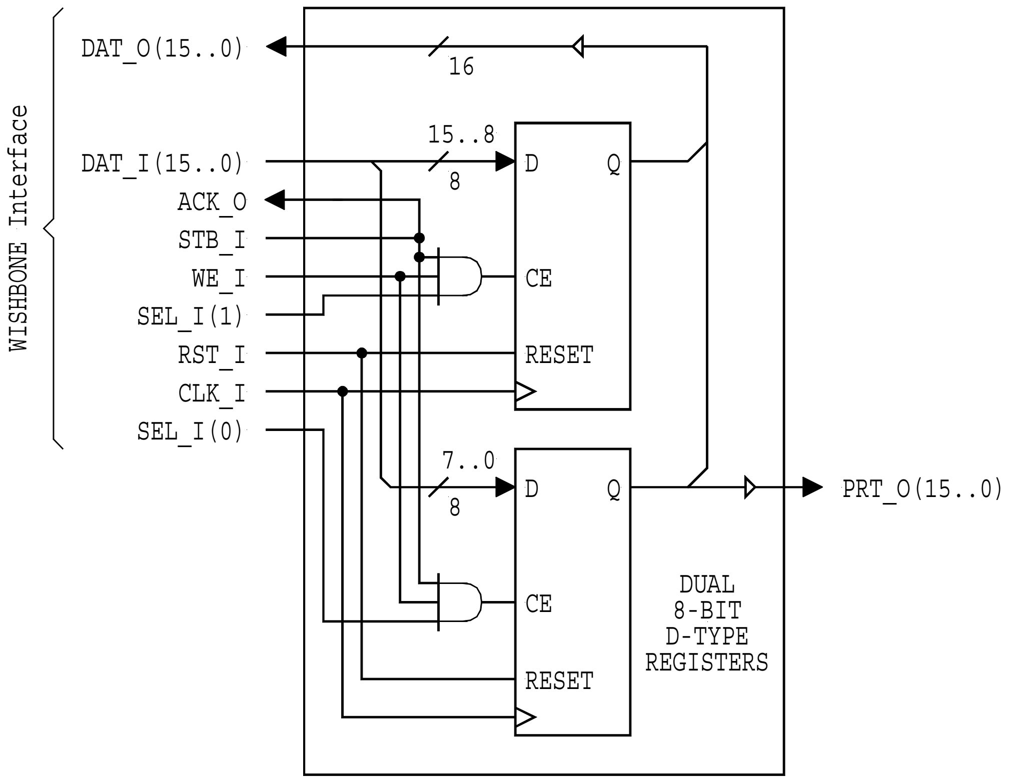


Figure B-4. Simple 16-bit WISHBONE SLAVE output port with 8-bit granularity.

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

| WISHBONE DATASHEET | | |
|---|---|---|
| Description | Specification | |
| General description: | 16-bit SLAVE output port with 8-bit granularity. | |
| Supported cycles: | SLAVE, READ/WRITE<br>SLAVE, BLOCK READ/WRITE<br>SLAVE, RMW | |
| Data port, size:<br>Data port, granularity:<br>Data port, maximum operand size:<br>Data transfer ordering:<br>Data transfer sequencing: | 16-bit<br>8-bit<br>16-bit<br>Big endian and/or little endian<br>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 B-5 shows a VHDL implementation of same circuit. The WBOPRT16 entity implements the all of the functions shown in the schematic diagram of Figure B-4.

```
entity WBOPRT16 is
port(
      -- WISHBONE 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:
      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 B-5.  VHDL implementation of the 16-bit output port with 8-bit granularity.

# Appendix C - WISHBONE Interface for Memory Elements

(This appendix is not part of the WISHBONE specification).

This application note provides an example of a WISHBONE interface for a memory element. The purpose of this application note is to:

- 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.

## Simple 16 x 8-bit SLAVE Memory

Figure C-1 shows a simple 8-bit WISHBONE memory. The 16 x 8-bit memory is formed from two 16 x 4-bit synchronous memories. Besides the memory elements, the entire interface is implemented with a standard 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)].

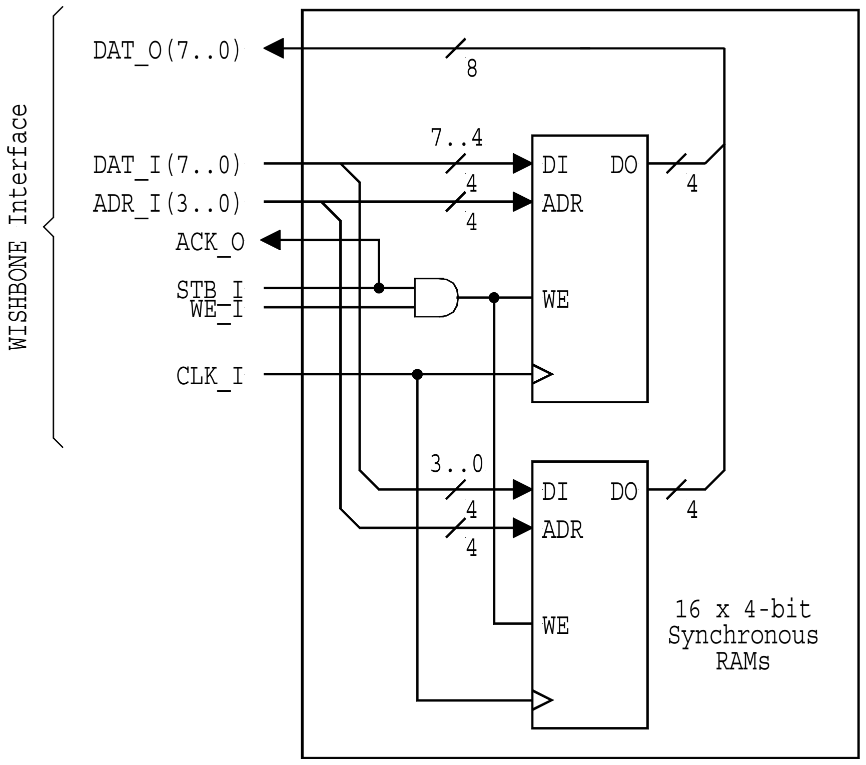


Figure C-1. Simple 16 x 8-bit SLAVE memory.

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 did 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 C-1 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 C-1. WISHBONE DATASHEET for the 8-bit output port example.

| **WISHBONE DATASHEET** | | |
|---|---|---|
| Description | Specification | |
| General description: | 16 x 8-bit memory IP core. | |
| Supported cycles: | SLAVE, READ/WRITE<br>SLAVE, BLOCK READ/WRITE<br>SLAVE, RMW | |
| Data port, size:<br>Data port, granularity:<br>Data port, maximum operand size:<br>Data transfer ordering:<br>Data transfer sequencing: | 8-bit<br>8-bit<br>8-bit<br>Big endian and/or little endian<br>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. | |

## 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 or 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 circuit is highly portable, but does assume that the FPGA or ASIC target device: (a) has synchronous memory elements available and (b) that the memory element has an asynchronous read function. The requirement for synchronous memories is obvious, as WISHBONE is a synchronous interface. However, the asynchronous read function may not be as obvious.

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 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.

In modern FPGA and ASIC devices, the asynchronous read function (in synchronous RAMs) is becoming very popular. For example, the Lucent Technologies RCF16X4 memory primitives (in the ORCA Macro Library) support both synchronous and asynchronous read operations. In this case, a pin is tied high or low to enable or disable the function. Similar memories are also available from other manufacturers.

# INDEX

0x (prefix), 14
ACK_I signal, 23, 29
ACK_O signal, 25, 31
active high logic state, 15
active low logic state, 15
addressing
  FIFO, 16
  memory mapped, 17
  partial, 17
ADR_I(63..0) signal array, 25
ADR_O(63..0) signal array, 24
application interface, 58
ASIC, 15
asserted signal, 15
big endian. *See* endian
BLOCK cycle, 37
BLOCK READ cycle, 38
BLOCK WRITE cycle, 41
bus, 15
bus cycles
  BLOCK, 37
  BLOCK READ, 38
  BLOCK WRITE, 41
  READ/WRITE, 32
  RMW, 44
  SINGLE READ, 33
  SINGLE WRITE, 35
bus interface, 15
  general operation, 27
  handshaking protocol, 28
  reset operation, 27
BYTE(N), 47
CLK_I signal, 23
CLK_O signal, 22
clock edge number, 13
clock transition, 13
compatibility
  IP core, 9
crossbar switch, 10, 11, 15, 59
CYC_I signal, 25
CYC_O signal, 24, 30, 31
DAT_I(63..0) signal array, 24, 25
DAT_O(63..0) signal array, 24, 25
data
  organization, 15
data organization, 47
documentation standard, 9, 21
DWORD(N), 47
endian, 10, 15, 47, 62
ERR_I signal, 24, 29
ERR_O signal, 25, 31
FIFO memory, 11, 16
fixed interconnection, 16
fixed timing specification, 16
foundry. *See* silicon foundry
FPGA, 16
full address decoding, 16
glue logic, 10, 16
granularity, 16, 47
handshaking protocol, 28
HDL, 16
I/O routing, 11
interconnection
  crossbar switch, 11
  FIFO, 11
  fixed, 16
  memory mapped, 11
  multiplexor logic, 17
  off-chip, 11
  three-state, 18
  variable, 19
interconnection methods, 58
IP core, 17
little endian. *See* endian
logic state
  active high, 15
  active low, 15
  signal name, 22
logo. *See* WISHBONE logo
MASTER, 17
MASTER signals, 23
MASTER/SLAVE topology, 10
memory mapped, 11, 17
module, 17
multiple MASTER, 58
multiplexor logic interconnection, 17
negated signal, 17
OBSERVATION, 12
off-chip I/O routing, 11
operand size, 17
partial address decoding, 17
PCI, 17
PERMISSION, 12
port
  64-bit, 51
port size, 17
portable interface, 9
ports
  16-bit, 54
  32-bit, 53
  8-bit, 55
power-up reset, 28
QWORD(N), 47
RECOMMENDATION, 11

reset operation, 27
revision level. *See* WISHBONE revision level
RMW cycle, 44
router, 17
RST_I signal, 23
RST_O signal, 23
RTL design methodology, 18
RTY_I signal, 24, 29
RTY_O signal, 26, 31
RULE, 11
SEL_I(7..0) signal array, 26
SEL_O(7..0) signal array, 24
signal
  asserted state, 15
  description, 22
  input, 14
  logic levels, 9
  name. *See* signal name
  naming conventions, 14, 21
  negated, 17
  non-WISHBONE, 22
  output, 14
  three-state, 61
silicon foundry, 18
SINGLE READ Cycle, 33
SINGLE WRITE cycle, 35
SLAVE, 18
SoC, 18
STB_I signal, 26
STB_O signal, 25, 30, 31
structured design, 9, 18
SUGGESTION, 12
synchronous protocol, 10
SYSCON, 18, 22
system controller. *See* SYSCON
system on chip (SoC), 18
tag, 10
TAGN_I signal, 22, 23, 31
TAGN_O signal, 22, 23, 31
target device, 18
three-state bus interconnection, 18
three-state interconnection, 61
timing delay, 30
timing diagrams, 12
timing specification, 19, 56
transition path, 10
variable address path, 10
variable data path, 10
variable interconnection, 19
variable timing specification, 19
Verilog, 19
VHDL, 19
VMEbus, 19
WE_I signal, 26
WE_O signal, 25
WISHBONE
  copyright release, 3
  DATASHEET, 19, 21, 22, 27, 29, 30, 31, 32, 50, 51, 57
  disclaimer, 4
  documentation standard, 21
  logo, 3, 14, 20
  revision history, 4
  revision level, 21
  signal, 19
  steward, 3
WORD(N), 47
WSM (wait state MASTER), 13
WSS (wait state SLAVE), 13

Ω