ARC® 700 IP Library

ARC® 700 External Interfaces

Reference

5117-014

ARC® 700 External Interfaces Reference

ARC® International

European Headquarters	North American Headquarters
ARC International,	3590 N. First Street, Suite 200
Verulam Point,	San Jose, CA 95134 USA
Station Way,	Tel. +1 408.437.3400
St Albans, Herts, AL1 5HE, UK	Fax +1 408.437.3401
Tel. +44 (0) 1727 891400	
Fax. +44 (0) 1727 891401	

www.arc.com

5117-014 April-2008

Contents

List of Figures

List of Tables

Chapter 1 — Introduction

An ARC® 700 processor based design supports a number of processor island interfaces. These interfaces encompass memory transactions, host debug access and miscellaneous control. The following sections introduce the interfaces in more detail:

- Overview of Interfaces
- Block Diagram
- Signal Lists

Overview of Interfaces

A given ARC® 700 design offers several processor island interfaces. These interfaces encompass memory transactions, host debug access and miscellaneous control.

Interfacing to the processor is achieved mainly indirectly through the following components.

- JTAG communications module
- Bus Bridge
- Bus Interfaces
- Closely Coupled Memory Direct Memory Interface (DMI)
- XY Memory Direct Memory Interface (DMI)
- Control Signals (Clock, Reset, etc)
- Interrupt Unit
- Memory Management Unit (MMU) – as described in the *ARC 700 MMU Reference*

The processor island is the top-level processor island that should be used for integration into a custom system. For additional information on alternative CPU Island interfaces see *AHB Bus Bridge Reference*, *AXI Bus Bridge Reference*, and *ARC Legacy Bus Bridge Reference*.

For further information on the operation of the processor core see *ARCompact Programmer's Reference*.

For information on the processor module hierarchy see the *ARC 700 System Reference*.

Block Diagram

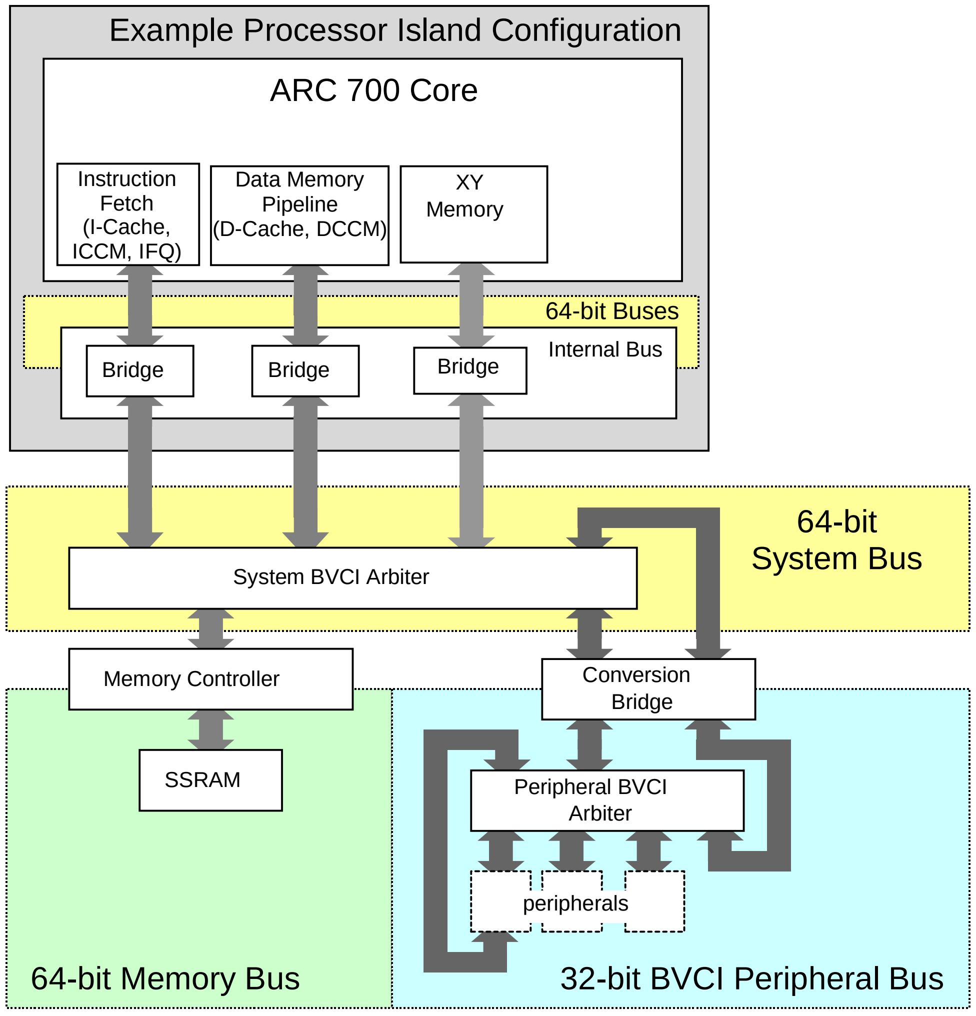


Figure 1 Example External Bus System Architecture

Signal Lists

Various groups of interface signals may appear on the CPU Island. The following signal lists provide more detail on these signal groups:

- JTAG Signal List
- BVCI Signal List
- MWIC Bus Bridge to External Bus System Signal List
- DMP Bus Bridge to External Bus Signal List
- CCM DMI Signal List
- XY DMI Signal List
- Processor Signal List

- Interrupt Signal List
- Test Signal List

Chapter 2 — JTAG (Joint Test Action Group) Communication Module

The JTAG interface has been introduced as a solution for communicating with the standard ARC 700 and ARCangel systems.

The JTAG module draws its interface and protocol from the IEEE STD 1149.1, providing customers with a standard that is universally recognized. The module contains logic for communicating with the ARC 700 processor and its memory system, providing the host with a high level role where transaction parameters are simply specified.

The following subsections outline principles required in order to communicate with the ARC 700 processor and system memory via the JTAG module:

- JTAG Interface
- JTAG Programmer's Model
- JTAG Port
- Setting Up Read/Write Transactions
- JTAG Port Reset
- PC - JTAG Communications

JTAG Interface

The host device communicates with the JTAG module via four interface signals, these four interface signals are required in order to satisfy the IEEE STD 1149.1 standard. These interface signals provide the host with the ability to control and serially pass data in and out of the module. These signals are shown below:

- TCK – Test Clock
- TMS – Test Mode Select
- TDI – Test Data In
- TDO – Test Data Out

An optional JTAG interface signal, Test Reset (`TRST*`) has been provided to allow asynchronous initialization of the JTAG port without supplying a clock. Its use is necessary in simulation, but in actual operation it may be tied high. In addition, there is a chip-level signal not specified by the IEEE standard: RTCK. This is a copy of TCK that has been re-driven in the I/O pad ring. If the JTAG emulator chooses to take advantage of it, by using it to clock in TDO, it can compensate for the cable, board, and I/O pad delays to increase the speed at which TCK may be run. This becomes especially important if many chips are chained together on the board.

The module provides various groups of interface signals.

- **The Memory Arbitrator Interface:** The first group interfaces to the memory arbitrator (refer to the *ARC 700 System Components Reference*) and allows the module to access system memory.
- **ARC 700 Host Interface:** The second group drives the ARC 700 host interface bus, providing essential access to the ARC 700 processor's internal register space.
- **Boundary Scan interface:** The third group of signals have been provided in order to allow the inclusion of a Boundary Scan Register. Refer to The Boundary Scan Register (Instruction Code 0x0 and 0x1) for a detailed explanation. While this capability remains, the ARC 700 JTAG port has been designed to allow on-chip chaining with other TAP controllers. It may provide an easier integration with ATPG flow to use the TAP controller produced by the ATPG software, and chain it with that of the ARC 700.
- **Miscellaneous signals:** The final group of signals are provided for system control:
  - — A processor clock signal is provided allowing the module to carry out read and write transactions to the devices that are synchronized to the processor clock. The JTAG clock, TCK, is designed to run at maximum frequency is 50% of the processor clock. In RTL simulation the ratio of system clock to processor clock may affect the maximum frequency of TCK. In silicon there are additional timing constraints, for example the time from the transition on the input JTAG clock to the output of JTAG TDO must be allowed for.
  - — A system clear signal is included allowing the module to be reset asynchronously with all devices in the system.
  - — An output enable signal, `jtag_tdo_zen_n`, is provided allowing the output `TDO` to go high impedance when inactive. Tri-state is provided for the case of parallel connection of the other driving circuitry to `TDO`. Should `TDO` not be connected to any other driving circuitry, the tri-state output need not be implemented.

— The JTAG busy signal, `jtag_busy`, can be used to provide a helpful indication that there is activity on the debug channel, for example to drive an LED on a development board. If not required, this signal can be left open.

Figure 2 illustrates the signals that make up the JTAG module.

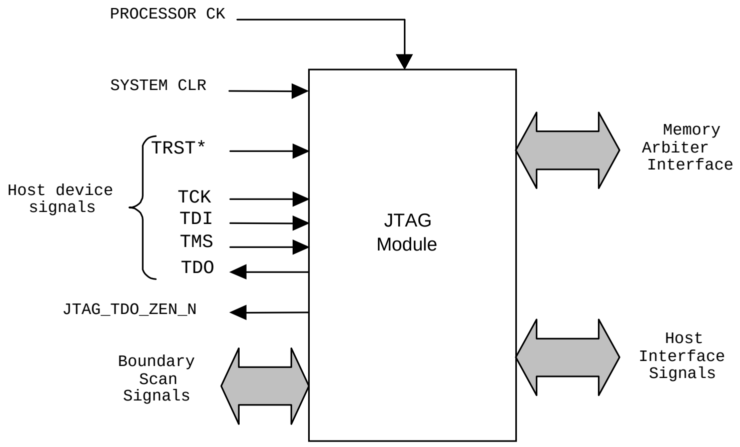


Figure 2 The JTAG Communications Module

The `TMS` input interface signal should be connected to a pull up component as part of the IEEE STD 1149.1 requirement, thus allowing the module to be reset if the input to `TMS` is undefined (for example high impedance) and `TCK` is applied. Pull-ups are not required for `TDI`, `TDO` or `TCK`. The reset mechanism is described in The TAP Controller State Machine.

JTAG Signal List

The following JTAG interface signals may appear on the CPU Island:

Table 1 JTAG Signal List

Signal	Direction	Description
`jtag_tdi`	Input	JTAG data input
`jtag_tms`	Input	JTAG mode select
`jtag_tck`	Input	JTAG clock
`jtag_trst_n`	Input	JTAG reset
`jtag_tdo`	Output	JTAG data output
`jtag_tdo_zen_n`	Output	JTAG TDO output enable signal
`jtag_rtck`	Output	JTAG re-timed clock
`jtag_busy`	Output	JTAG busy signal

JTAG Pin Connector

This is the recommended board connector to attach a debug emulator to the JTAG signals on the board. It is a 20-pin IDC connector, with pins on 0.100” centers, keyed and shrouded.

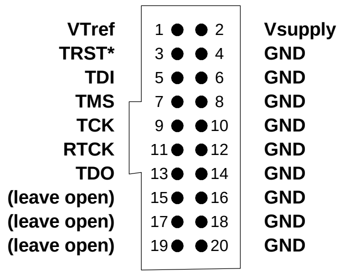


Figure 3 Recommended JTAG Pin Connector, Top View

Table 2 JTAG Pin Connector Descriptions

Signal	Description and Notes
`TCK`	Clock input to debug port. Must be pulled to defined state on board for so as not to clock circuitry when no debug emulator is connected.
`RTCK`	Clock output. If not implemented on chip, drive from TCK on board.
`TMS`	Test Mode Select input. Must be pulled up on board.
`TDI`	Test Data In input. Must be pulled up on board.
`TDO`	Test Data Out output. Must be pulled up on board.
`TRST*`	Test Reset. Must be pulled up on board.
`VTref`	Target Reference Voltage. Should be tied to Vdd of chip.
`Vsupply`	Supply voltage for emulator pod. Should be tied to Vdd of chip.
`GND`	Ground. Tie to Vss of chip.

While it is not necessary, the speed of the JTAG debug connection can be maximized if TDO and RTCK use drivers capable of driving 50-Ohm transmission lines.

JTAG Programmer's Model

The JTAG module includes eight internal registers as shown in Table 3. The host can define a read or write transaction to a memory location or an ARC 700 register through some of these internal registers. Six of the eight registers are collectively referred to as data registers (IEEE STD 1149.1). The remaining registers are the instruction register, which is central in the role of accessing all data registers and the Boundary Scan register.

Table 3 JTAG Registers

Value	Code	JTAG Register	TYPE
N/A	N/A	INSTRUCTION REGISTER*	Instruction
0x8	1000	JTAG STATUS REGISTER	Data
0x9	1001	TRANSACTION COMMAND REGISTER	Data
0xA	1010	ADDRESS REGISTER	Data
0xB	1011	DATA REGISTER	Data
0xC	1100	IDCODE REGISTER	Data
0xF	1111	BYPASS REGISTER*	Data
0x0/0x1	0000/0001	BOUNDARY SCAN REGISTER*	BSR

NOTE * Required as part of IEEE STD 1149.1 specification.

Each of the registers are described in the following sections:

- The Instruction Register
- The JTAG Status Register (Instruction Code 0x8)
- The Transaction Command Register (Instruction Code 0x9)
- The Address Register (Instruction Code 0xA)
- The Data Register (Instruction Code 0xB)
- The IDCODE register (instruction code 0xC)
- The Bypass Register (Instruction Code 0xF)
- The Boundary Scan Register (Instruction Code 0x0 and 0x1)

The Instruction Register

The Instruction register is used to gain access to all data registers. Each data register is addressed by a unique 4-bit instruction code.

31 … 4	3 … 0
Reserved	**Inst Code**

In order to access the required data register, the correct code should be written into the instruction register. Figure 4 illustrates the relationship between the instruction and data registers. The instruction shift register is loaded with 0x1 in Capture-IR, so that external circuitry, by looking for a transition when in Shift-IR, can detect a stuck-at fault in the JTAG chain.

NOTE Because of this behavior, which is specified in IEEE 1149.1, the current contents of the instruction register can not be read by the external circuitry. The register itself is initialized to point to the IDCODE register in Test-Logic-Reset.

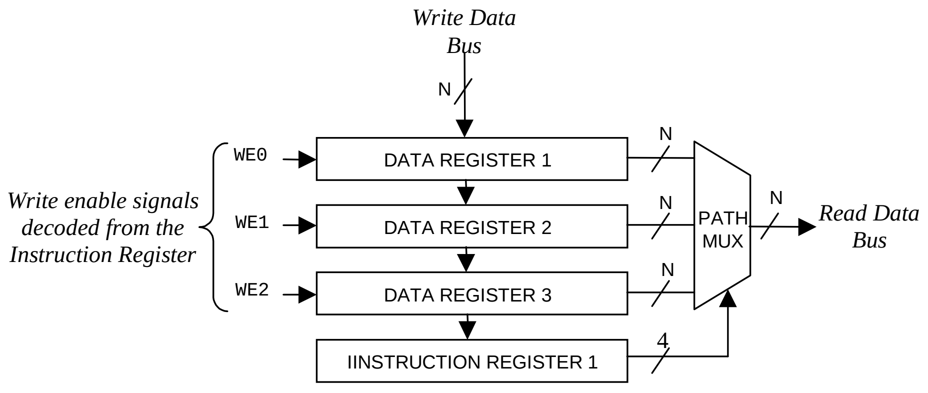
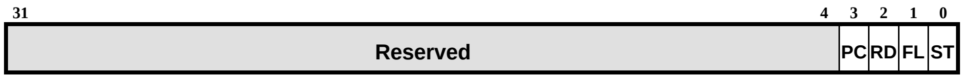


Figure 4 Data Registers Access via the Instruction Register

In addition to accessing the data registers (refer to subsections The JTAG Status Register (Instruction Code 0x8) to The Bypass Register (Instruction Code 0xF)), the instruction register is also used to select a test sequence that should be applied to the device. These test sequences use a special data register known as the Boundary Scan Register (refer to The Boundary Scan Register (Instruction Code 0x0 and 0x1)).

The JTAG Status Register (Instruction Code 0x8)

The JTAG Status Register is read only and is used by the host device to obtain important information on the state of the JTAG module or the result of an ARC 700 processor or memory access. The bits of the register are assigned as follows.



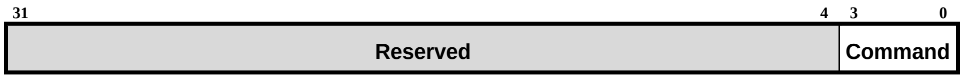
31
24
23
4
3
2
1
0

Each field in the JTAG Status register reflects the following information:

- Bit 0 – Stalled (ST) flag indicates that the current transaction has stalled. This flag is set when the ARC 700 processor asserts the `hold_host` signal to lengthen the duration of a read or write transaction.
- Bit 1 – Failure (FL) flag indicates that a read (or write) has failed when it is true. For example, this flag would be set if an access to a core register is attempted when the processor is running. The failure flag is cleared automatically when a new transaction is started.
- Bit 2 – Ready (RD) flag indicates whether the JTAG module is available to accept another transaction command. This flag is set when a transaction has just completed or when the JTAG module is idle.
- Bit 3 – PC_SEL (PC) flag, is set to the value that is assigned to the `AUX_PCPORT` auxiliary register (refer to the Extension Functions section in the *ARCangel development board manual*). For example, this flag would be set if `1` was written to the `AUX_PCPORT` auxiliary register, and cleared if `0` was written to `AUX_PCPORT.`
- Bits 31 down to 24 – Reserved.

The Transaction Command Register (Instruction Code 0x9)

The Transaction Command Register is used to specify the communication transaction that should be performed.



The JTAG module supports eight different accesses or transactions, which are shown in Table 4 with their associated encoding.

Table 4 JTAG Read/Write Transactions

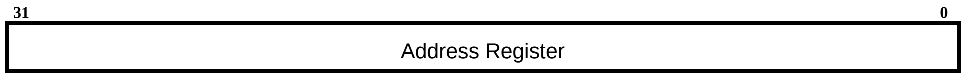
Value	Code	Communication Transaction
0x0	0000	Write to a memory location
0x1	0001	Write to a ARC 700 core register
0x2	0010	Write to a ARC 700 auxiliary register
0x3	0011	NOP, The register is initialized to this value
0x4	0100	Read from a memory location
0x5	0101	Read from a ARC 700 core register
0x6	0110	Read from a ARC 700 auxiliary register

Value	Code	Communication Transaction
0x7	0111	Write to a MADI* register
0x8	1111	Read from a MADI* register

NOTE The MADI register is only available where the debugging of multiple ARC 700 processor systems is required. The MADI system is no longer the recommended way of debugging multiple cores on a chip. ARC now recommends that each processor have its own JTAG port, and that these be chained together by distributing TCK, TMS, and TRST* in parallel, and connecting the TDO from one processor to the TDI of the next.

The Address Register (Instruction Code 0xA)

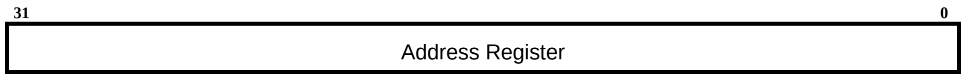
The Address Register is used to supply the address for read and write transactions to the ARC 700 registers and system memory.



Accesses to memory must always be given in bytes. Access to the ARC 700 internal registers is specified by their register numbers. The value contained in this register is automatically incremented by four (a memory access) or one (an ARC 700 register access) when a read or write transaction has completed. This feature is used to save valuable cycle time when downloading / uploading a stream of data, hence the register does not need to be rewritten with the next address value.

The Data Register (Instruction Code 0xB)

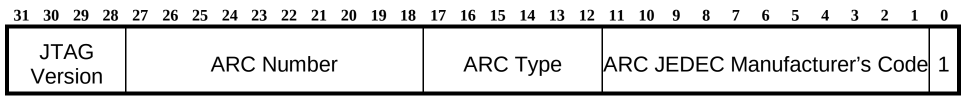
The data register performstwo functions. When data is written to this register, it is placed into a write buffer that drives two write data buses, one for the ARC 700 host interface and other for the memory arbitrator interface. The bus is used to specify the data contents that should be written when performing a write transaction.



When reading this register a read buffer is selected. The read buffer is used to store data retrieved from the target device during a read transaction. The appropriate read data bus (arbitrator or host interface) is selected according to which device the host is accessing.

The IDCODE register (instruction code 0xC)

The IDCODE register is used by the JTAG emulator to identify the core as an ARC 700 core.



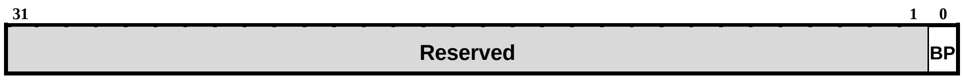
Each field in the IDCODE register reflects the following information:

- Bits 31 down to 28 — These bits define the version of the JTAG module. Currently set to the value 0x1.
- Bits 27 down to 18 — This field will be set by the designer to the number of ARC processors swithin the system. It will have the same value as the corresponding field in the IDENTITY register in auxiliary space, if there are fewer than 256 ARC processors in the system.

- Bits 17 down to 12 — This field will be set to 0x03 to identify the processor type as an ARC 700 type.
- Bits 11 down to 1 — This field contains the code assigned to ARC International by JEDEC, encoded as specified in the IEEE 1149.1-2001 standard. ARC has been assigned the manufacturer's code 0x58 in group five, so this field encodes to 0100 101 1000.
- Bit 0 — This field is fixed at 1, as specified in the IEEE 1149.1-2001 standard. This is used, along with the previous field, to allow automatic discovery when the chain is initialized. JEDEC will never assign the manufacturer's code 0x7F in group 0. The JTAG emulator, therefore, can shift the 32-bit IDCODE 0x000000FF into TDI at the beginning of the chain after reset. The standard specifies that upon receiving a TCK when in the Reset-Test-Logic state, the instruction register will be initialized to point to the IDCODE register if it exists, and to the BYPASS register otherwise. In Capture-DR, the shift register is loaded with this 32-bit code if IDCODE is in the instruction register, and with a single bit of 0 if BYPASS is. Thus the external circuitry can examine TDO in Shift-DR, and know if it's zero that this TAP controller has only a 1-bit bypass register, and if it's one that this TAP controller has a 32-bit IDCODE register. By shifting through looking for these until the 0x000000FF appears, the emulator can uniquely identify the number and kind of devices in the chain.

The Bypass Register (Instruction Code 0xF)

The Bypass Register is required as part of the IEEE STD 1149.1 standard.



When this register is selected, the serial data input (`TDI`) is connected to the serial data output (`TDO`) through this register. The data on `TDI` is passed to `TDO` on the rising edge of the JTAG clock `TCK` when in Shift-DR, and the register is initialized to 0 in Capture-DR. In all other states, the data in this register is held.. The Bypass register is automatically selected when a reset is applied to the JTAG module allowing the data on `TDI` to bypass the core logic to `TDO`.

The Boundary Scan Register (Instruction Code 0x0 and 0x1)

The Boundary Scan register is selected when the four-bit code `0000` or `0001` is written into the Instruction register. This register is used to retrieve the logic state of a device and control data on its input and output pins. The register does not exist within the module and must be provided externally (it is connected to the JTAG module via the boundary scan interface). The codes `0000` and `0001` relate to the `EXTEST` and `SAMPLE/PRELOAD` instructions as shown in Table 5. These boundary scan instructions are necessary as part of the IEEE STD 1149.1. Refer to the Application Note 'Interfacing the JTAG Module to a Boundary Scan Register' for more detail.

NOTE Since the ARC 700 TAP controller may be chained on the chip with other TAP controllers, use of a separate TAP controller known to be compatible with the user's test software is recommended.

Table 5 Instructions that Employ the Boundary Scan Register

Value	Code	Instruction
0x0	0000	EXTEST
0x1	0001	SAMPLE/PRELOAD

The instructions contained in Table 6 have also been defined in the IEEE STD 1149.1. These are optional instructions and are not supported with version 1.0 of the JTAG module.

Table 6 Non Implemented Instructions

Instruction	**Description**
INTEST	Performs an internal test (uses the boundary scan register)
RUNBIST	Runs an internal core logic test (additional logic)
USERCODE	Captures user defined information about a device (additional logic)

JTAG Port

The block diagram shown in Figure 5 shows how the JTAG communications port integrates within an ARC 700 system. It is linked to the host interface of the ARC 700 processor in addition to system memory via the Data Memory Pipeline (DMP).

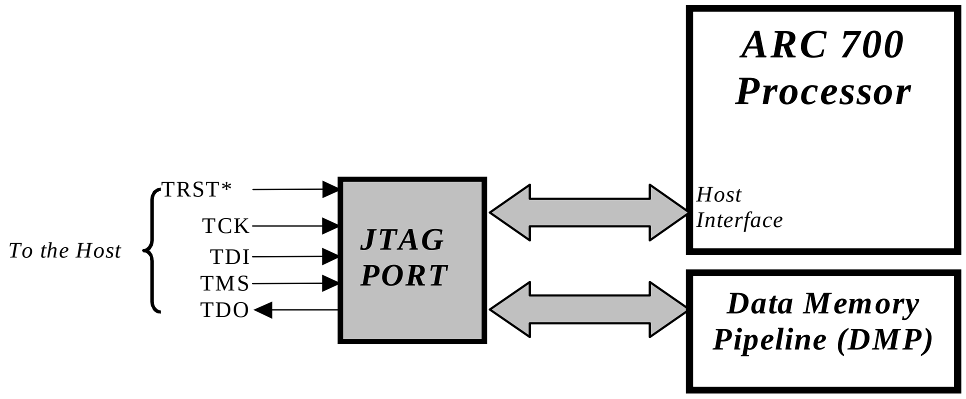


Figure 5 A JTAG Port with an ARC 700 Processor

- The TAP Controller
- The TAP Controller State Machine
- The Debug Port
- The Host Interface to BVCI Target

The TAP Controller

The Test Access Port (TAP) controller is central to the operation of the JTAG module as shown in Figure 6. All internal register accesses are performed serially using the TAP controller. An accompanying block, the Debug Port, serves as the workhorse, performs the majority of internal (accessing internal JTAG registers) and external (performing BVCI transactions) tasks. The Debug Port and the TAP Controller are clocked off of TCK, and a separate module synchronizes the BVCI initiator signals to the system clock. On the other side of the BVCI Debug Interface, there is a module that contains the address, data, command, and status registers, and handles the host interface and DMP transactions: the Host Interface to BVCI Target module. This allows the user, if desired, to replace the JTAG port, either by a custom interface to an external debugger, or to another processor in a master/slave configuration..

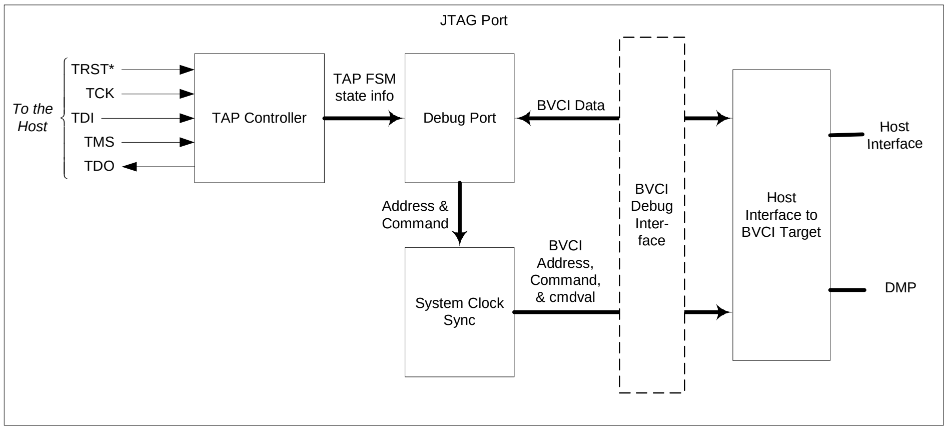


Figure 6 Internal Structure of the JTAG Port

The TAP controller is an internal state machine that is controlled entirely by the host using the `TMS` and `TCK` interface signals. The controller is used to indirectly initiate communication transactions and access the internal JTAG registers. The state machine consists of 16 states that are connected together as shown in Figure 7.

The TAP Controller State Machine

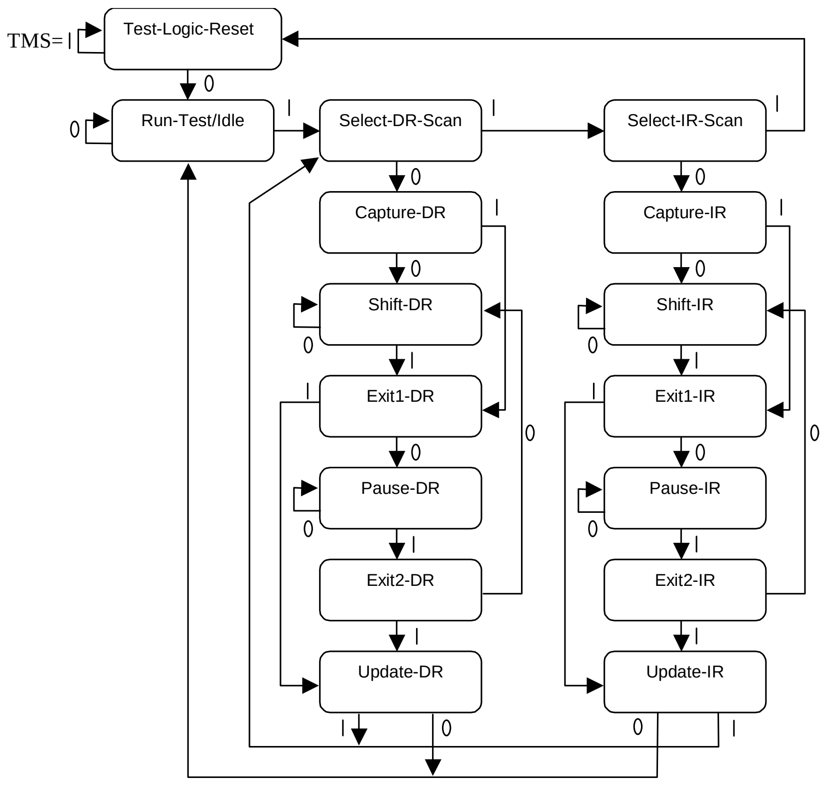


Figure 7 TAP Controller State Diagram

Each state contains at least one entry point with two possible exit paths. A state transition is performed on the rising edge of `TCK`. The decision to determine the exit path is made according to the logic level of `TMS`.

The `Test-Logic-Reset` state is used to initialize all internal JTAG registers and control signals to default contents and inactive logic levels respectively. The state is entered immediately when TRST* is asserted. In addition, this state can also be entered (regardless of the current state) at any time during operation by holding `TMS` high and applying a maximum of five clock pulses on `TCK`.

The `Run-Test/Idle` state always precedes the `Test-Logic-Reset`, `Update-DR` and `Update-IR` states on the rising edge of `TCK` when `TMS` is low. This state is employed to initiate a read/write access or place the JTAG module in the idle state. The read/write access defined by the address, data and command registers only occurs once on entry to `Run-Test/Idle.`

The remaining section of the state diagram (in Figure 7) contains two state sequence structures that are used to access all internal JTAG registers. Registers can be written to or read from serially using the `TDI` and `TDO` signals along with the aforementioned `TCK` and `TMS` signals. Both structures are identical, however, as denoted by the mnemonics IR and DR, one structure is used to access the instruction register and the other dedicated solely to accessing all data registers.

An internal JTAG register is accessed by placing the TAP controller into the appropriate scan structure (`Select-DR-Scan or Select-IR-Scan`). The data contents of the selected register are loaded into a shift register in the `Capture-xR` state. The state is then entered from the `Select-xR-Scan` state by pulling `TMS` low and applying a clock pulse on `TCK`. Capture occurs when the `Capture-xR` state is exited.

The shift register is used to shift data into the chain from `TDI` (write phase) simultaneously shifting data out of the chain at `TDO` (read phase).

By holding `TMS` low and applying a second clock pulse on `TCK` the TAP controller goes into the `Shift-xR` state. The Tap Controller remains in the `Shift-xR` state when `TMS` is held low. This state allows data to be loaded serially (least significant bit first) into the shift register. The `TDI` signal is always sampled on the rising of edge of `TCK`, starting on the second entry into the `Shift-xR` state. The data shifted out is placed on `TDO` on the falling edge of `TCK` starting on the first entry into the `Shift-xR` state. The last sample of `TDI` is always performed when exiting the `Shift-xR` state. For instance, when the instruction register contains the BYPASS instruction, a 0 is loaded into the 1-bit bypass register on the clock that exits `Capture-DR` and enters `Shift-DR`. At this point, the 0 will appear on TDO. On the next clock, TDI will appear on TDO, and data will continue to be shifted through until the final TDI is shifted to TDO on the clock which exits `Shift-DR` and enters `Exit1-DR`.

When the data is finally shifted in or out of the shift register, the selected JTAG register is updated with the shift register contents when the TAP controller is placed into the `Update-xR` state. Updating occurs on the falling edge of `TCK` after the state is entered.

The remaining states `Pause-xR` and `Exit2-xR` are used to stall the shift process if the data to be shifted in cannot be presented in time for the next rising edge of `TCK` (assuming a continuous frequency on `TCK`).

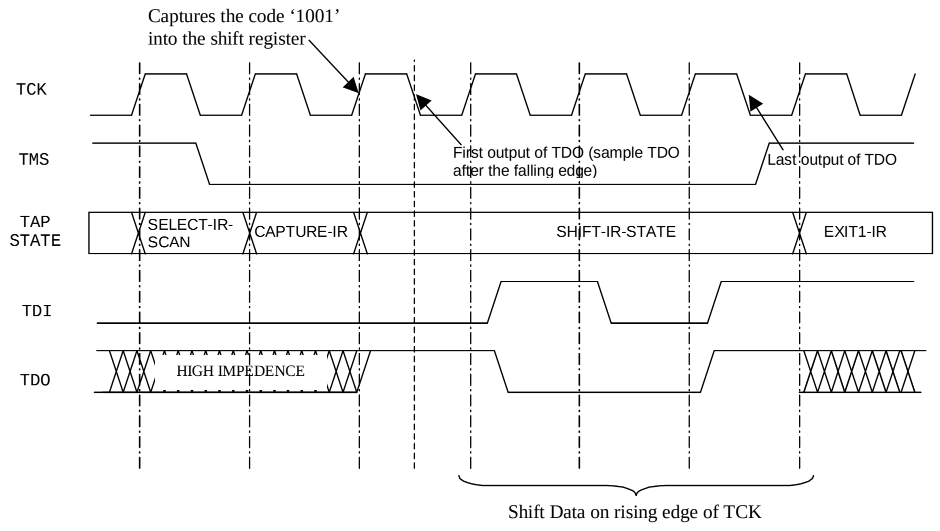

Captures the code '1001' into the shift register
TCK
TMS
TAP STATE
TDI
TDO
First output of TDO (sample TDO after the falling edge)
Last output of TDO
SELECT-IR-SCAN
CAPTURE-IR
SHIFT-IR-STATE
EXIT1-IR
HIGH IMPEDENCE
Shift Data on rising edge of TCK

Figure 8 Loading Data into the Shift Register

The timing diagram in Figure 8 illustrates the concept of shifting data into the shift register using `TCK`, `TMS` & `TDI`. The diagram illustrates the host device writing to the instruction register with the four-bit value `1010`, thus selecting the Address register. In the capture stage the four-bit value `0001` is loaded into the shift register ready to be shifted out. The instruction register is not updated (and the Address register is not selected) with the shift register contents until the `Update-IR` state is entered. Following the `Update-IR` state the `Select-DR-Scan` state structure is entered to access the Address register.

During the `Capture-IR` phase the four-bit value `0001` is always loaded into the shift register, regardless of the instruction register contents. The first two least significant bits aid in diagnosing faults along the IEEE 1149.1-1990 bus.

The Debug Port

The debug port module contains all the registers specific to the JTAG interface. This includes the instruction and data shift registers, the instruction register itself, the bypass register, and the IDCODE register. It also contains a restricted BVCI initiator, which, in conjunction with the system clock synchronization module, is responsible for access to the address, data, transaction command, and status registers, and for initiation of read and write accesses. It must be stressed that the debug BVCI interface is completely separate from the memory BVCI interfaces. The address space of the debug BVCI interface contains only six valid addresses in its stock configuration: the addresses of the address, data, transaction command, and status registers, along with two special addresses which, when written with any data, cause the read/write access to be initiated and the address, data, and command registers to be reset, respectively.

The Host Interface to BVCI Target

The address, data, status, and transaction command registers are to be found in the host interface to BVCI target module (as shown in Figure 9). It contains a state machine, which performs all read and write bus transactions that are supported by the JTAG module. This is providing there is a valid code in the Transaction Command register; an access is initiated by placing the TAP controller into the `Run-Test/Idle` state, which causes the debug port to write the do_cmd address on the BVCI interface. This request is then fed to the state machine. The scheduler is responsible for verifying transaction requests and providing the mechanism that allows the host device and the JTAG module to maintain a strict synchronizing relationship.

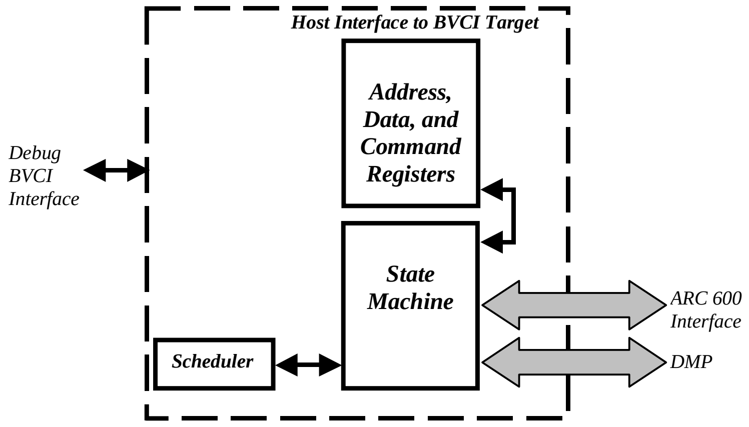

Host Interface to BVCI Target
Address, Data, and Command Registers
Debug BVCI Interface
State Machine
ARC 600 Interface
DMP
Scheduler

Figure 9 Internal Structure of the Host Interface to BVCI Target Module

The scheduler verifies a transaction request from the host device by checking the value contained in the Transaction Command register. The transaction request signal is asserted only when the Transaction Command register contains a valid code and a write to the do_cmd address occurs on the debug BVCI interface. The scheduler asserts a transaction request signal to start the defined bus transaction. The BVCI target always responds to all commands in a single cycle. It is the responsibility of the debugger to maintain synchronization by polling the status register for the READY bit after an access has been started.

Setting Up Read/Write Transactions

A guide through the stages of defining and initiating read and write accesses:

- Setting up a Write Access to the ARC 700 Processor or Memory
- Accessing the Status Register
- Setting up a Read Access from the ARC 700 Processor or Memory

Setting up a Write Access to the ARC 700 Processor or Memory

A write access requires placing the TAP controller into the `Test-Logic-Reset` state. This should reset the JTAG module. This initializes all the internal JTAG registers to default values and all interface control signals to inactive logic levels. This initialization process is performed by asserting TRST* or by holding `TMS` high and applying a maximum of five clock pulses on `TCK`. This will ensure that when the `Run-Test/Idle` state is entered a bus transaction is not triggered from a valid code already contained in the Transaction Command register.

The reset procedure is not required for every read and write access, and is performed only when there has been a system reset. The next stage is to set-up the transaction parameter registers. These include the Address register, Data register and the Transaction Command register.

The contents of the Address and Data registers are loaded with the appropriate values so that the write access can be performed on the ARC 700 processor or to memory. Firstly, the instruction register is loaded with the code for accessing the Data register. This is accomplished by entering the `Select-IR-Scan` state and updating the Instruction register. Then the code `1011` for the Data register is serially loaded in the `SHIFT-IR` state. The timing diagram for writing the instruction register with the code `1011` which selects the Data register is shown in Figure 10.

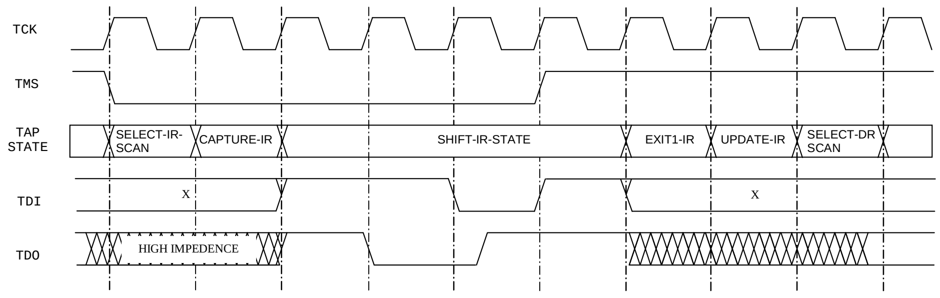


Figure 10 Loading the Instruction Register

X =Don't care, Z = high impedance

The `Select-DR-Scan` state is then selected to serially load in the data to be employed by the write data bus when write access is performed. This is shown in the timing diagram in Figure 11.

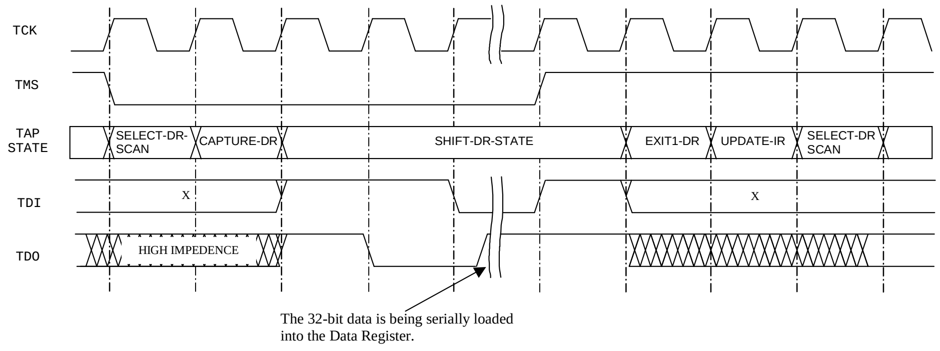


Figure 11 Loading the Data Register

The Address register is now accessed by loading the code `1010` into the instruction register. We then enter the `Select-DR-Scan` state structure and serially load in the data to be used on the address bus.

The last stage involves writing the Transaction Command register with the code instructing the JTAG module to perform a write transaction to either the memory or the ARC 700 registers.

Once all the transaction parameters have been setup, the write transaction is started by placing the TAP controller into the `Run-Test/Idle` state.

To obtain information about the transaction, the JTAG Status register is accessed. Since this is a read only register the signal supplied on `TDI` is ignored when the register contents are shifted out through `TDO`. The appropriate bit fields are then checked to verify a write transaction was performed.

Accessing the Status Register

The JTAG Status register is accessed in the following way, the first bit in the Status register (refer to The JTAG Status Register (Instruction Code 0x8)) is shifted out to determine whether the JTAG module has been stalled. If the stalled bit is set, then the `Select-DR-Scan` state structure is exited and returned to later. This happens if a requested transaction is already underway.

If the JTAG module is not stalled, then the second bit is shifted out of the Status register to determine whether that transaction has failed. If the failed bit is set then the `Select-DR-Scan` state structure is exited.

If the failed bit is not set then the ready bit is shifted out to determine whether the transaction has completed or not. If the ready bit is set, then the transaction has finished and a new transaction can be started. If it has not been set, then the `Select-DR-Scan` state structure is exited and the above procedure is repeated.

The `PC_SEL` is an optional bit that does not need to be shifted out.

Setting up a Read Access from the ARC 700 Processor or Memory

Setting up a read transaction follows almost the same procedure as setting up a write transaction.

The Address register is accessed by writing the 4 bit code `1010` into the instruction register. `Select-IR-Scan` state structure is selected and the code `1010` is serially loaded. When the `Update-IR` state is entered the instruction register is updated and the Address register is selected.

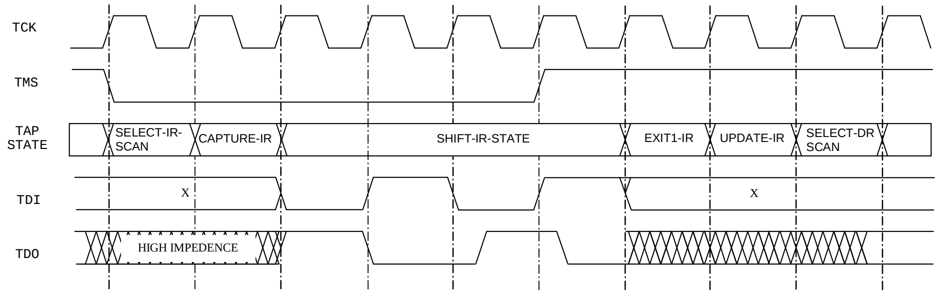


Figure 12 Loading the Instruction Register (Select Address Register)

The data register does not need to be accessed at this stage as a read transaction is being performed.

The last stage of setting up the transaction parameter registers involves writing to the Transaction Command register with the required read transaction.

When all the transaction parameters have been set up the access is started by placing the TAP controller into the `Run-Test/Idle` state.

To obtain information about the transaction, the JTAG status register is interrogated. Since this is a read only register the signal supplied on `TDI` is ignored when the register contents are shifted out through `TDO`. The appropriate bit fields are then checked to verify the read transaction. Refer to Accessing the Status Register for a standard routine on how to decode the JTAG Status register.

When the transaction has completed successfully the data register is selected. `Select-IR-Scan` state structure is selected and the code `1011` is serially loaded into the instruction register. The `Select-DR-Scan` state is then entered and data is shifted out from the selected device.

JTAG Port Reset

When implementing a system with a JTAG port, pin SS1 must be connected to logic on the board such that, when the PC sets signal SS1 low (active), signal xclr will be driven low to reset the ARC 700 processor.

PC - JTAG Communications

The ARCangel3 (AA3) Development board features an interface to a bi-directional PS/2 parallel port. This port allows the following functions to be performed:

- start, stop and single step the ARC 700 processor
- read/write all core registers
- read/write all auxiliary registers
- read/write external memory
- perform system reset
- generate a ARC 700 interrupt

With the exception of resets, all functions are performed by downloading a 32-bit address/control word, followed by 32 bits of send or receive data. The 32 bit values are sent serially, least significant bit first.

Table 7 shows a summary of the signals in use on the parallel port connector.

Table 7 JTAG Port Signals

Pin	Driver	Signal	Function
1	pc	TCK	Test Clock - Used to control data flow. Data from the PC latched on rising edge.
2			Not used
3			
4			
5			
6			
7			
8	pc	TMS	Test Mode Select – used to select the TAP controller states
9	pc	TDI	Test Data In - Serial Data input
10			Not used
11	aa	busy	Valid for AA3. Set high whenever AA3 JTAG port is in non-idle state. Drives the right-hand LED on AA3.
12			Not used
13	aa	TDO	Test Data Out – serial data out
14			Not used
15			Not connected
16	pc	ss0	Valid for AA3. See following page.
17	pc	ss1	Valid for AA3. See following page.
18		0v	
19		0v	

Pin	Driver	Signal	Function
20		0v	
21		0v	
22		0v	
23		0v	
24		0v	
25		0v	

As a requirement for the SeeCode DLL an external reset signal should be provided in the JTAG interface version of the ARC 700 processor allowing the chip to be reset. There is a soft reset mechanism in the JTAG module that is used to reset JTAG module alone.

ss0 and ss1 are used as follows when a JTAG comms port is implemented on the AA3 development board. At the D25 connector to the AA3 board, the functionality of ss0 and ss1 is:

ss0	ss1	
0	0	Reset the ARC 700 system on the FPGA
1	0	FPGA Configuration download
0	1	ARC 700 system in normal operation
1	1	ARC 700 system in normal

Chapter 3 — Bus Bridge

A bus bridge provides several useful architectural functions. The bridge provided performs the following functions.

- Bus protocol translation
- Bus timing registering
- Clock crossing

Bus protocol translation enables the ARC 700 design to interface to any bus topology that uses a specific bus protocol. The ARC 700 processor uses native BVCI (mandatory signaling subset as defined by pages 27, and 29 - 30 of the VSI Alliance Virtual Component Interface Standard Version 2 (OCB 2 2.0)). In order to interface to a non-BVCI memory system, an appropriate bus bridge is needed that can perform BVCI to non-BVCI bus protocol standards conversion. The default bridge that is included in an ARC 700 processor build performs no bus protocol translation, therefore the default memory interfacing standard it BVCI.

Bus timing isolation is used to register signals that are late arriving (relative to the rising edge of the CPU clock) from the various processor components, such as the instruction and data caches. The output of the bridge interface ensures that a transaction starts on the rising edge of the clock, providing a complete cycle for any proprietary memory subsystem.

Clock crossing provides the ability to support a different clock speed on the system bus to that of the processor internal bus (BVCI).

The following sections describe the bus bridges in more detail:

- Bus Bridge Block Diagram
- BVCI Protocol
- Bus Bridge Block Diagram
- Clock Synchronization Unit
- Clock Crossing BVCI Bridge

For additional information on alternative CPU Island interfaces see *AHB Bus Bridge Reference*, *AXI Bus Bridge Reference*, and *ARC Legacy Bus Bridge Reference*.

Bus Bridge Block Diagram

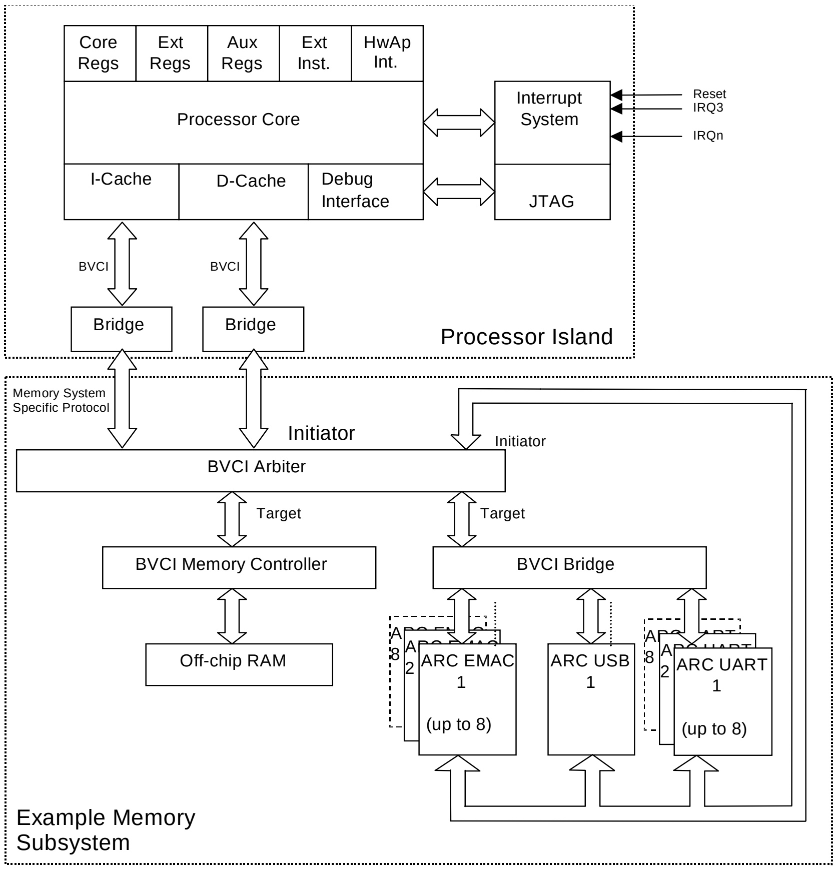


Figure 13 Example of a Typical ARC 700 System

BVCI Protocol

Basic Virtual Component Interface (BVCI), is a sub-set of the Virtual Component Interface (VCI) standard, and it is a protocol standard resulting from the work of the On-Chip Bus Development Working Group of the Virtual Socket Interface Alliance (VISA). This open standard was written to provide a general interface specification for Virtual Components (hardware Intellectual Property) so that they can be easily used to built System-on-Chip (SoC). Designed primarily as a point-to-point on-chip protocol, it is technology independent but has the advantage of being a powerful protocol, and yet inherently efficient and simple to implement. The VCI standard defines three levels of Complexity, and they are, in the order of complexity, Peripheral VCI (PVCI), Basic VCI (BVCI) and Advance VCI (AVCI). PVCI protocol is a sub-set of BVCI and AVCI adds onto BVCI by specifying optional signals that can be added.

The internal processor bus protocol implements BVCI. BVCI defines an inherently split protocol. It allows multiple access commands to be sent from an initiator interface to a target interface before data or responses of the commands are returned. Its flexibility also makes it suitable for most applications, and makes it relatively easier to translate to other protocols.

With the BVCI protocol, individual access commands are sent as cells over a synchronous command bus. The read data and access response are then returned as cells over a separate response bus. These cells are combined to form packets (burst transfers). Handshaking between initiator and target interface at cell level is performed using simple cell valid and acknowledgement signals. One or more cells are packaged into packets, using additional signals to denote the last cell of a packet and define the addressing scheme.

Detailed information on the BVCI protocol may be obtained from:

- VSI Alliance - www.vsi.org: *Virtual Component Interface Standard (v2.0 - OCB 2 2.0)*

BVCI Signal List

The following BVCI interface signals may appear on the CPU Island, where * is the particular BVCI signal group:

Table 8 BVCI Signal List

Signal	Direction	Description
`*_address[31:0]`	Output	*Physical Byte Address*. It needs to be updated on every cycle during a burst.
`*_be[7:0]`	Output	*Byte Enables*. They can be byte, word, long word or double long words sized transactions.
`*_cmd[1:0]`	Output	*Command*. The types that can be issued are: • Read – 0x1 • Write – 0x2 • Locked Read – 0x3
`*_cmdval`	Output	*Command Cell Validate*. The values on the command bus are valid when this signal is true.
`*_eop`	Output	*End of Packet*. This signal is asserted on the last burst cycle to indicate the end of that burst request.
`*_plen[5:0]`	Output	*Packet Length*. Describes the packet length in bytes: • Byte – 0x1 • Word (16 bits) – 0x2 • Longword (32 bits) – 0x4 • Burst of 32 bytes (4 cells packet) – 0x20
`*_wdata[63:0]`	Output	*Write Data*. This is the data from a write.
`*_cmdack`	Input	*Command Acknowledge*. Acknowledges the valid command cell.
`*_rdata[63:0]`	Input	*Read Data*. This is the returning data from a read request.
`*_reop`	Input	*End of packet* on the response bus.
`*_rspval`	Input	*Response Valid*.
`*_rerror`	Input	*Response Error*. Bus errors are transferred directly to the ARC 700 core via the BVCI interface.

Bus Bridge Block Diagram

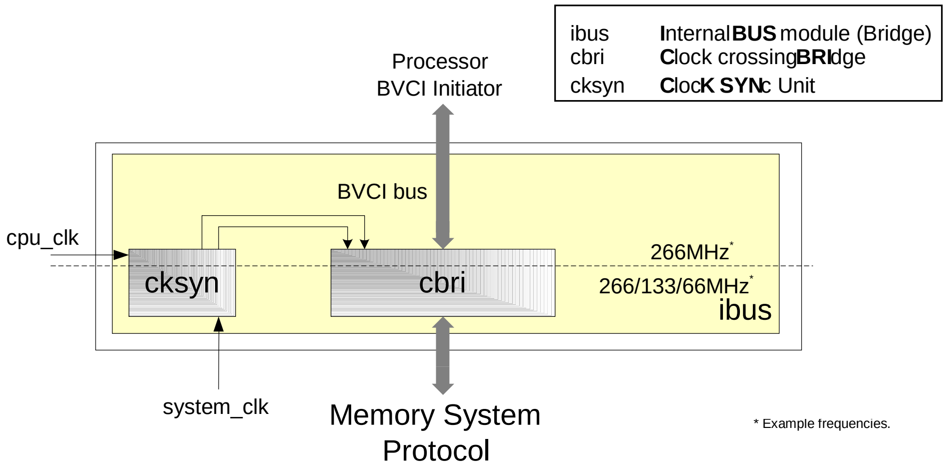


Figure 14 Bus Bridge Structure (IBUS)

Logic that is contained within the bus bridge is referred to as the *IBUS*. The name *Internal Bus* is derived from the concept of the processor island. A processor island denotes a collection of processor specific components that operate from a single processor clock. Processor Island components natively make use the BVCI protocol; therefore the IBUS protocol is BVCI.

The bridge is made up of two sub-modules and they are:

- Clock Crossing Bridge
- Clock Synchronization Unit

The Clock Crossing Bridge (CBRI) is responsible for converting accesses from a processor island component (such as the instruction cache), which is on the CPU clock domain (`clk_cpu`) onto the memory system clock domain (`clk_sys`). The supported frequency ratios of `clk_cpu` to `clk_sys` are 1:1, 2:1, 3:1 and 4:1, and both clocks have to be phased locked and clock tree balanced. This module is also responsible for isolating the timing within the internal bus from that of the memory system bus.

The Clock Synchronization Unit (CKSYN) is responsible for keeping track of the phase relationship between `clk_cpu`, used within the processor island, and `clk_sys`. This unit sends out synchronization signals to the bridge to help it latch and transfer data correctly.

All modules in the bridge module, except CKSYN, supports clock gating by generating a busy signal whenever there are outstanding accesses being handled.

Clock Synchronization Unit

The Clock Synchronization Unit, called CKSYN, is used to provide synchronization signals to the bridge so as to ensure that data on the BVCI bus can be transferred correctly across the clock domain. This module generates a synchronization signal to support processor clock to system clock frequency ratios of 1:1, 2:1, 3:1 and 4:1. Both clocks must be phased locked.

Figure 15 shows the design of the clock synchronization unit.

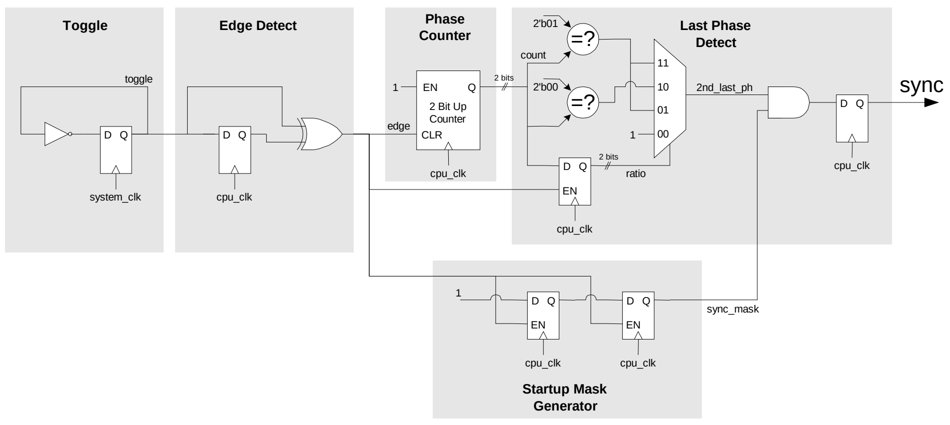


Figure 15 Design of the CKSYN

The CKSYN unit is used to generate a pulse on the synchronization signal (`sync`) that correspond to the last main clock period that resides in the system clock domain. It does this without having to resort to the use of the clock signals themselves as input signals, and without the need for the user to configure the hardware via software programmable registers or via configurations during RTL generation. This allows the module to be placed and routed without special consideration, allowing the user to avoid complicated issues often related to using clocks as signals, and easing the task of choosing or changing the system bus frequency throughout the SoC development cycle.

The CKSYN uses a number of logic units to generate the synchronization signal.

- Toggle Unit
- Edge Detection
- Phase Detection
- Mask Generator
- Last Phase Detect

Toggle Unit

The Toggle Unit generates a toggle signal (`toggle`) that changes at every rising edge of the system clock (`clk_sys`).

Edge Detection

The Edge Detect unit detects the rising edge of the toggle signal on the CPU clock domain. This edge signal (`edge`) will always go high at the first CPU clock (`clk_cpu`) period within a system clock period (`clk_sys`).

Phase Detection

The Phase Counter Unit counts up continuously and gets cleared whenever the edge signal is high. Hence the counter always counts up to one less than the number of CPU clock cycles that fit in the system clock cycle. Therefore, for example, for a CPU clock to system clock frequency ratio of 4:1, the counter counts continuously from 0 to 3. The range of clock ratios supported is dependent on the width of the counter, with the largest ratio supported at 2^N:1 for a N bit counter. For the bridge to support up to 4:1, the counter only needs to be two bits wide.

Mask Generator

The Startup Mask Generator unit is basically a 2-bit shift register that turns the synchronization mask off (`sync_mask` = 1) after two detected edges of the toggle signal. This mask is used to block incorrect synchronization signal that can be generated during the start-up of this module after global reset.

Last Phase Detect

The Last Phase Detect unit is used to generate a pulse at the last CPU clock cycle within the system clock cycle. Since it only needs to support up to 4:1 clock ratios, this module has been simplified. A 2-bit register is used to capture the count value at every toggle edge and it (ratio) represents the clock ratio of the two different clocks, with “00” representing 1:1, “01” representing 2:1, “10” representing 3:1 and “11” representing 4:1. This value is then used to select from four different pulses that are generated, (each pulse refers to a specific clock ratio). The first pulse type is for ratio 1:1 where the pulse is always on. The second pulse type is for ratio 2:1, where the signal is set to high every time the count value is at “01”. The third pulse type is for ratio 3:1, where the signal is set to high when the count value is “00”. Finally the fourth pulse type is for ratio 4:1, where the signal is set to high when the count value is “01”. The selected pulse signal is then combined with the mask signal through the AND gate, and registered to become the synchronization (sync) output.

NOTE During global reset, all flops, registers and the counter are set to zero.

Figure 16, Figure 17, Figure 18 and Figure 19 show four example timing diagrams when dealing with 4:1, 3:1, 2:1 and 1:1 clock ratios respectively.

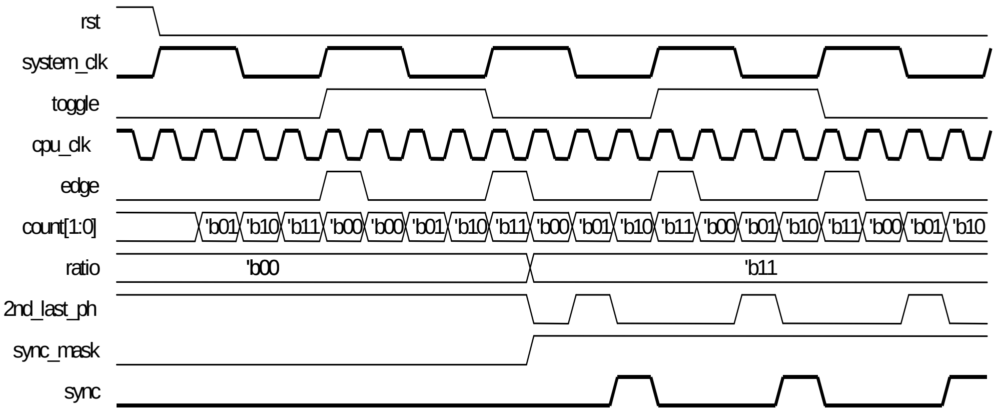

rst
system_clk
toggle
cpu_clk
edge
count[1:0]
'b01 'b10 'b11 'b00 'b00 'b01 'b10 'b11 'b00 'b01 'b10 'b11 'b00 'b01 'b10 'b11 'b00 'b01 'b10
ratio
'b00
'b11
2nd_last_ph
sync_mask
sync

Figure 16 CKSYN Timing Diagram at 4:1 Clock Ratio

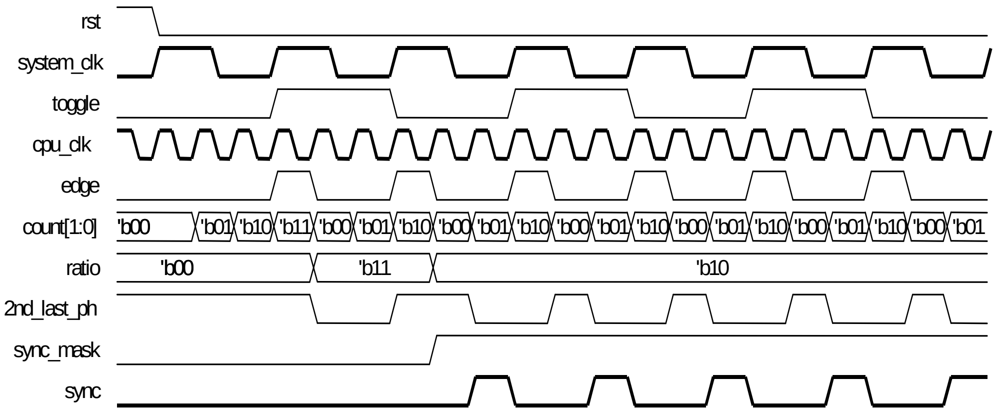

rst
system_clk
toggle
cpu_clk
edge
count[1:0]
'b00 'b01 'b10 'b11 'b00 'b01 'b10 'b00 'b01 'b10 'b00 'b01 'b10 'b00 'b01 'b10 'b00 'b01 'b10 'b00 'b01
ratio
'b00
'b11
'b10
2nd_last_ph
sync_mask
sync

Figure 17 CKSYN Timing Diagram at 3:1 Clock Ratio

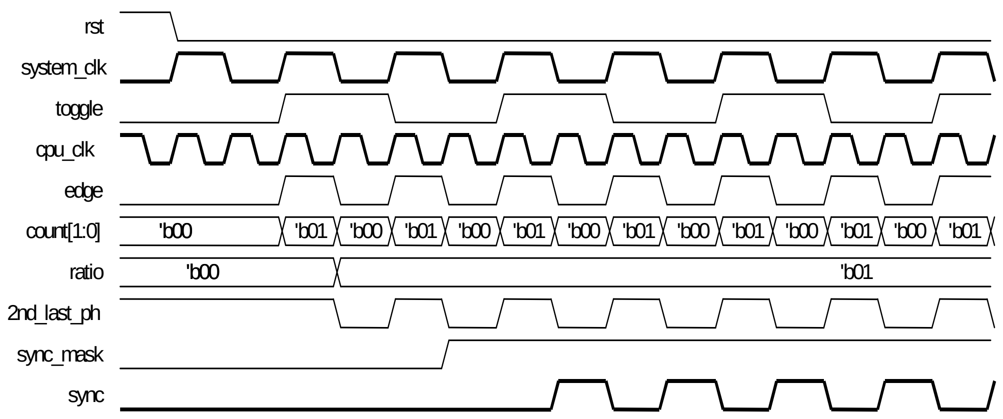


Figure 18 CKSYN Timing Diagram at 2:1 Clock Ratio

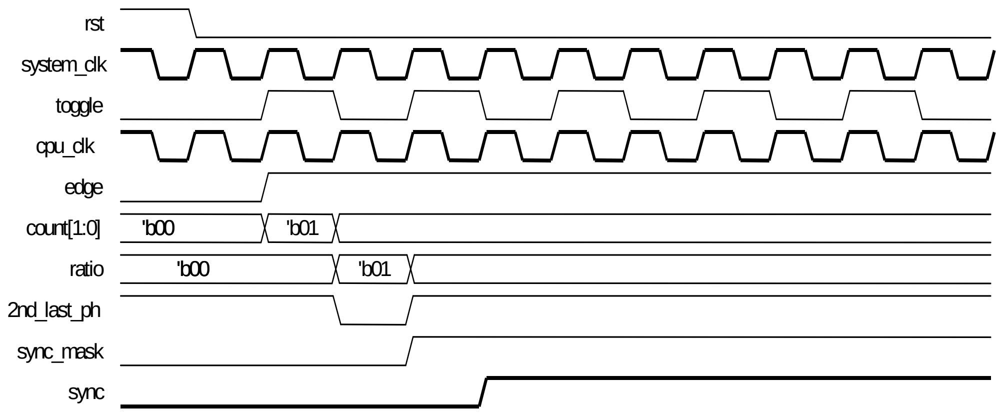


Figure 19 CKSYN Timing Diagram at 1:1 Clock Ratio

Clock Crossing BVCI Bridge

The CBRI is used to convert accesses between the two clock domains (`clk_cpu` and `clk_sys`). The supported frequency ratios of `clk_cpu` to `clk_sys` are 1:1, 2:1, 3:1 and 4:1, in the bridge module. However, this sub-module can support any ratios as long as the CPU clock is the same or higher frequency than system clock, and where it is higher, it is multiple times the frequency of the system clock. Both clocks have to be phased locked and clock tree balanced. This module is also responsible for isolating the timing of the internal bus and the BVCI System Bus from each other.

The CBRI cannot perform data packing and unpacking when converting access from one clock domain to another, and therefore, is essentially a bus repeater. It is capable of handling the crossing clock domain with the help of the CKSYN unit.

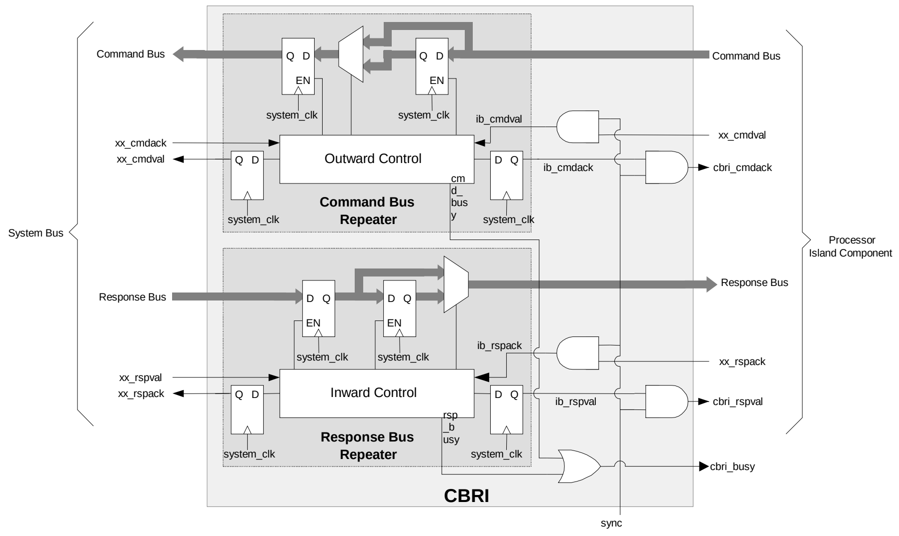


Figure 20 Design of the CBRI - Resets Not Shown

The CBRI is made up of a number of logic units:

- A command bus repeater module, used to deal with the command bus timing isolation.
- A response bus repeater module, used to deal with the response bus timing isolation
- Handshaking gating logic, used to ensure correct handshaking when crossing the clock domain
- An OR gate to combine the busy signals from the two repeaters.

Chapter 4 — Bus Interfaces

This section summarizes the signal naming convention for the various processor island components that attach to the bus bridges, as well as the bus bridge interface to the external memory system for the following modules:

- Instruction Cache (MWIC) to Memory Bus System (via Bus Bridge)
- DMP to Memory Bus System (via Bus Bridge)

For additional information on alternative CPU Island interfaces see *AHB Bus Bridge Reference*, *AXI Bus Bridge Reference*, and *ARC Legacy Bus Bridge Reference*.

Instruction Cache (MWIC) to Memory Bus System (via Bus Bridge)

The instruction cache (MWIC) does not connect directly to the memory bus systems. An intermediate bus bridge is used as shown in the following sections:

- MWIC and Bus Bridge Block Diagram
- MWIC to Bus Bridge Signal List
- MWIC Bus Bridge to External Bus System Signal List
- MWIC Unimplemented Signal List
- Big-Endian Configuration
- Interface Timing

MWIC and Bus Bridge Block Diagram

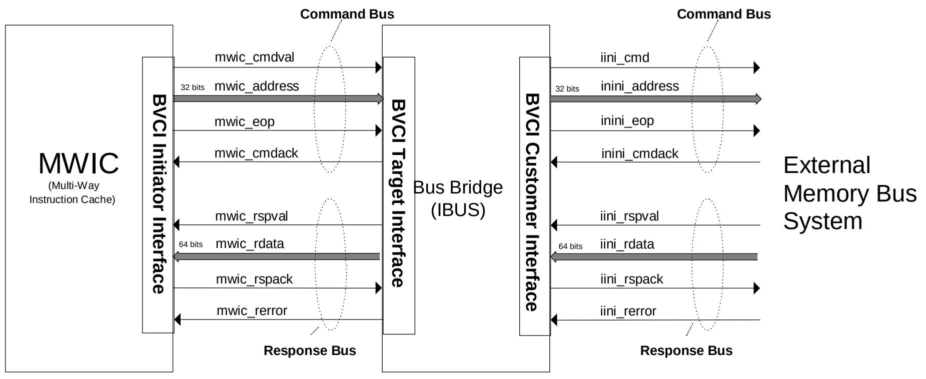


Figure 21 Instruction Cache to IBUS

MWIC to Bus Bridge Signal List

The following MWIC Bus Bridge interface signals are internal to the processor island and will *not* appear on the CPU Island:

Table 9 MWIC to IBUS Interface

Name	Direction	Width	Description
`mwic_rspack`	Input	1	Acknowledgement from the MWIC to say that it has received a valid 64-bit data item. Active High
`mwic_cmdval`	Input	1	Validates the command cell. Active High
`mwic_eop`	Input	1	Signifies end of packet and only asserted on the fourth and final command cell. Active High
`mwic_address`	Input	32	Physical byte address of instruction word to be fetched
`mwic_cmdack`	Output	1	This is the signal from the bus bridge that acknowledges the receipt of a valid command
`mwic_rdata`	Output	64	This is the returning 64-bit instruction word returning from the bus bridge.

Name	Direction	Width	Description
`mwic_rspval`	Output	1	Validates ‘mwic_rdata’
`mwic_rerror`	Output	1	Response error. Bus errors are transferred directly to the ARC 700 core via the BVCI interface.

MWIC Bus Bridge to External Bus System Signal List

The following MWIC Bus Bridge interface signals may appear on the CPU Island:

Table 10 Bus Bridge to External Memory System

Name	Direction	Width	Description
`iini_rspack`	Input	1	Acknowledgement from the bus bridge to say that it has received a valid 64-bit data item. Active High
`iini_cmdval`	Input	1	Validates the command cell. Active High
`iini_eop`	Input	1	Signifies end of packet and only asserted on the fourth and final command cell. Active High
`iini_address`	Input	32	Physical byte address of instruction word to be fetched
`iini_cmdack`	Output	1	This the signal from the memory bus system that acknowledges the receipt of a valid command
`iini_rdata`	Output	64	This is the 64-bit instruction word returning from the memory bus system
`iini_rspval`	Output	1	Validates ‘iini_rdata’
`iini_rerror`	Output	1	Response error. Bus errors are transferred directly to the ARC 700 core via the BVCI interface.

MWIC Unimplemented Signal List

Both the MWIC BVCI Target interface and bus bridge omit some signals from the BVCI protocol. These signals are omitted because they are always constant. If required, a tie value can be used as show in Table 11.

Table 11 Unimplemented BVCI Interface Signals on the MWIC to IBUS Interface or IBUS to External Memory

Name	Direction	Width	Tied Value	Description
`mwic_cmd\iini_cmd`	Input	2	2’b01	Command, which for this interface is always a read from the MWIC.
`mwic_contig\iini_contig`	Input	1	1	States that the addresses provided are contiguous. Always true from the MWIC.
`mwic_wrap\iini_wrap`	Input	1	1	States that the addresses provided are critical word first, wrap around format. This is always true from the

Name	Direction	Width	Tied Value	Description
				MWIC.
`mwic_constant\iini_constant`	Input	1	0	Implies a constant address. Never true.
`mwic_plen\iini_plen`	Input	6	32	Total number of bytes that is required. Always 32.
`mwic_be\iini_be`	Input	8	8'bff	Byte enable signal. Always has value 0xFF in this case.
`mwic_reop\iini_reop`	Output	1	-	End of Response Packet. Not used by MWIC.

Big-Endian Configuration

When the ARC 700 processor is configured as a big-endian system, the 32-bit local data is appropriately aligned within the 64-bit data in the system memory.

Interface Timing

The MWIC module only performs read accesses, with a packet size of 32 bytes (burst of four 64 bits cells). The interaction between the MWIC BVCI Initiator interface and the bus bridge interface is described, however the transactions between the bus bridge and memory bus system are identical.

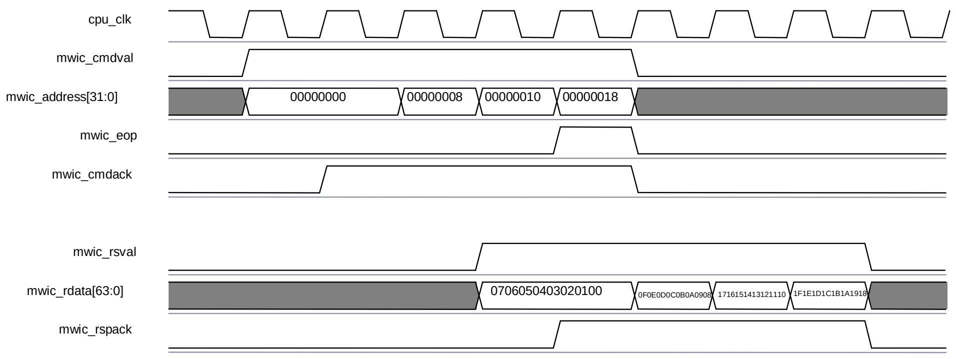


Figure 22 MWIC Target Interface Read Access

1. In the diagram the MWIC initiates the start of an access by asserting the command valid signal (`mawic_cmdval` = 1) and presents the first address of the wrap around style burst transfer on the address bus (`mwic_address`).
2. This first command cell is then accepted by the IBUS target interface by asserting the command acknowledgement signal (`mwic_cmdack` = 1) either in the same clock cycle (known as default acknowledgement), or after one or more clock cycles later (Figure 22).
3. With each new command acknowledgement, the MWIC initiator present the next address, qualifying it with the command valid signal each time. At the last address cell, the End Of Packet signal is asserted as well.
4. When the data of this access is available one or more cycles after the beginning of the access, the IBUS module presents it on the data bus (`mwic_rdata`) and qualifies it with the response valid signal (`mwic_rspval` = 1). Each data cell is acknowledged by the initiator interface using the response acknowledgement signal (`mwic_rspack`). Wait cycles can be inserted between each data cell by de-asserting either the response valid and/or response acknowledgement signal.

DMP to Memory Bus System (via Bus Bridge)

The DMP unit does not connect directly to the memory bus systems. An intermediate bus bridge is used as shown in the following sections:

- DMP and Bus Bridge Block Diagram
- DMP to Bus Bridge Signal List
- DMP Bus Bridge to External Bus Signal List
- DMP Unimplemented Signal List
- Big-Endian Configuration
- Interface Timing

DMP and Bus Bridge Block Diagram

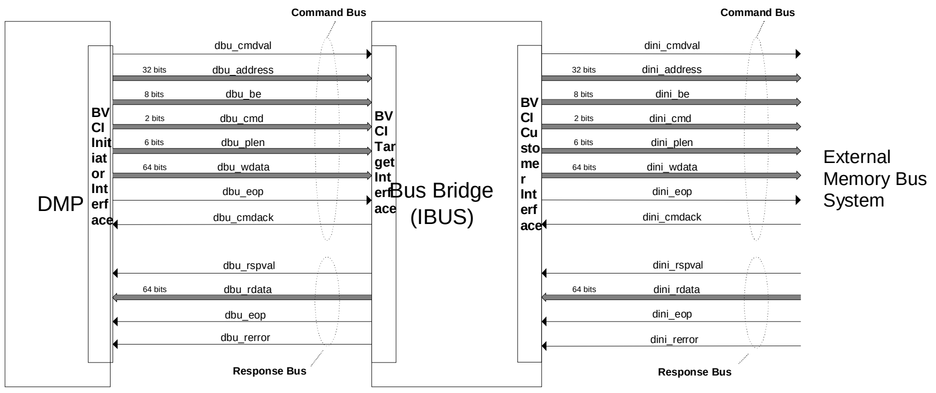


Figure 23 DMP to IBUS Interface

DMP to Bus Bridge Signal List

The following DMP Bus Bridge interface signals are internal to the processor island and will *not* appear on the CPU Island:

Table 12 DMP to IBUS Interface

Signal	Direction	Bus Width	Description
`dbu_address`	Input	32	Physical byte address. It must be updated on every cycle during a burst.
`dbu_be`	Input	8	Byte Enables. They can be byte, word, long word or double long words sized transactions.
`dbu_cmd`	Input	2	Command. The types that can be issued by the DMP are: Read – 0x1 Write – 0x2 Locked Read – 0x3
`dbu_cmdval`	Input	1	Validates the command cell. The values on the command bus are valid when this signal is true.
`dbu_eop`	Input	1	End of Packet. This signal is asserted on the last burst cycle to indicate the end of that burst request.

Signal	Direction	Bus Width	Description
dbu_plen	Input	6	Describes the packet length in bytes: Byte – 0x1 Word (16 bits) – 0x2 Longword (32 bits) – 0x4 Burst of 32 bytes (4 cells packet) – 0x20
dbu_wdata	Input	64	Write Data. This is the data from a write.
dbu_cmdack	Output	1	Command Acknowledge. Acknowledges the valid command cell.
dbu_rdata	Output	64	Read Data. This is the returning data from a read request.
dbu_reop	Output	1	End of packet on the response bus.
dbu_rspval	Output	1	Response Valid.
dbu_rerror	Output	1	Response error. Bus errors are transferred directly to the ARC 700 core via the BVCI interface.

DMP Bus Bridge to External Bus Signal List

The following DMP Bus Bridge interface signals may appear on the CPU Island:

Table 13 Bus Bridge to Memory Bus System

Signal	Direction	Bus Width	Description
dini_address	Input	32	Physical byte address. It needs to be updated on every cycle during a burst.
dini_be	Input	8	Byte Enables. They can be byte, word, long word or double long words sized transactions.
dini_cmd	Input	2	Command. The types that can be issued by the DMP are: Read – 0x1 Write – 0x2 Locked Read – 0x3
dini_cmdval	Input	1	Validates the command cell. The values on the command bus are valid when this signal is true.
dini_eop	Input	1	End of Packet. This signal is asserted on the last burst cycle to indicate the end of that burst request.
dini_plen	Input	6	Describes the packet length in bytes: Byte – 0x1 Word (16 bits) – 0x2 Longword (32 bits) – 0x4 Burst of 32 bytes (4 cells packet) – 0x20
dini_wdata	Input	64	Write Data. This is the data from a write.
dini_cmdack	Output	1	Command Acknowledge. Acknowledges the valid command cell.
dini_rdata	Output	64	Read Data. This is the returning data from a read request.

Signal	Direction	Bus Width	Description
dini_reop	Output	1	End of packet on the response bus.
dini_rspval	Output	1	Response Valid.
dini_rerror	Output	1	Response error. Bus errors are transferred directly to the ARC 700 core via the BVCI interface.

DMP Unimplemented Signal List

The DMP BVCI Target interface omits some signals from the BVCI protocol. If required, a tie value can be used as show in Table 14.

Table 14 Unimplemented BVCI Interface Signals

Signal	Direction	Bus Width	Tied Value	Description
dbu_contig\dini_contig	Input	1	1	Contiguous Operation. This is tied high.
dbu_rspack\dini_rspack	Input	1	1	Response Acknowledge. Indicates that the received valid data has been acknowledged. This signal is always asserted high.
dbu_wrap\dini_wrap	Input	1	1	Burst Wrap Around. Asserted on the end of cache line boundary to wrap around thus achieving critical word first requests. This is tied high.
dbu_constant\dini_constant	Input	1	0	Implies a constant address. Never true.

Big-Endian Configuration

When the ARC 700 processor is configured as a big-endian system, the 32-bit local data is appropriately aligned within the 64-bit data in the system memory.

Interface Timing

The DMP BVCI interface is used to perform direct load\stores and data cache refills\ data write backs to and from main memory. The interaction between the DMP BVCI Initiator interface and the bus bridge interface is described, however the transactions between the bus bridge and memory bus system are identical.

The transactions can be classed into several types of operations:

- Read Type I – A contiguous cache line fill where the requested data is in the first 64-bit cell
- Read Type II – A cache line fill where the requested data is not in the first 64-bit cell and the burst request wraps around to complete the burst read.
- Read Type III – A single byte, word, or longword access when all the ways are locked and there is a cache miss or an access is made to an uncached location.
- Write Type I – A contiguous cache line writeback is performed to physical memory.

- Write Type II – A single byte, word, or longword writeback is performed to physical memory.
- Locked Read – A longword, word or byte read access that locks the memory controller so that the next access is serviced from the DMP interface. All other interfaces are ignored until this happens.

Each of the operations use standard BVCI transactions, performed using either a single cell packet transfer or a burst of four cell transfer.

Single Cell Read Accesses

The DMP uses single cell read accesses to perform read type III operations.Figure 24 shows an example timing diagram.

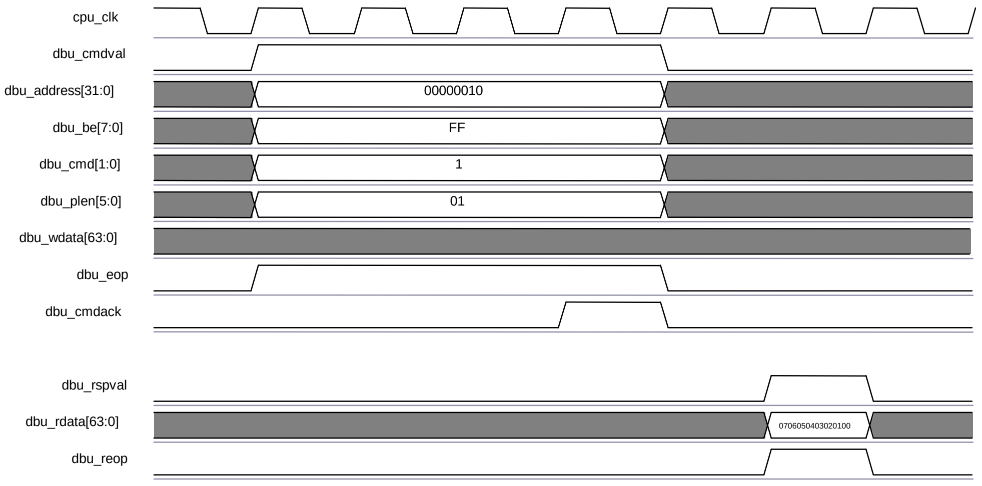


Figure 24 Single Cell Packet Read Access

In the diagram the DMP initiates the start of an access by asserting the command valid signal (`dbu_cmdval` = 1) and presents the address (`dbu_address`), packet length (`dbu_plen`), end of packet (`dbu_eop`) and access command (`dbu_cmd`) of the access on the command bus.

This first command cell is then accepted by the IBUS target interface by asserting the command acknowledgement signal (`dbu_cmdack` = 1) either in the same clock cycle (known as default acknowledgement), or after one or more clock cycles later.

When the data of this access is available one or more cycles after the beginning of the access, the IBUS module presents it on the data bus (`dbu_rdata`) along with the end of packet signal (`dbu_eop`), qualified using the response valid signal (`dbu_rspval` = 1).

Single Cell Write Accesses

The DMP uses single cell write accesses to perform write type II operations. Figure 25 shows an example timing diagram.

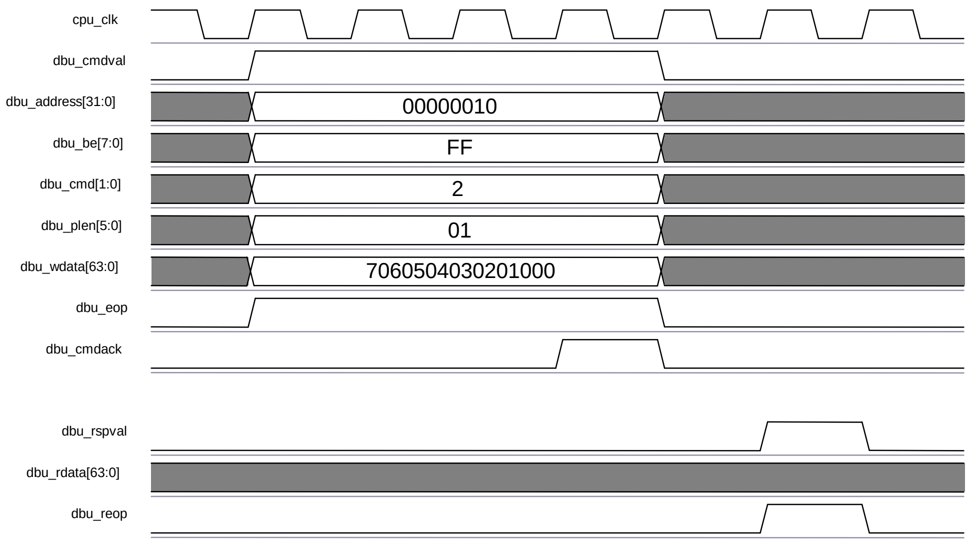


Figure 25 Single Cell Packet Write Access on BVCI Target Interface between DMP and IBUS

In the diagram the DMP initiates the start of an access by asserting the command valid signal (`dbu_cmdval` = 1) and presents the address (`dbu_address`), packet length (`dbu_plen`), end of packet (`dbu_eop` = 1), access command (`dbu_cmd`) and the write data (`dbu_wdata`) of the access on the command bus.

This first command cell is then accepted by the IBUS target interface by asserting the command acknowledgement signal (`dbu_cmdack` = 1) either in the same clock cycle (known as default acknowledgement), or after one or more clock cycles later.

When the data of this access is written one or more cycles after the beginning of the access, the IBUS module respond by asserting the end of packet signal (`dbu_eop` = 1) and the response valid signal (`dbu_rspval` = 1).

Burst of 4 Cell Read Accesses

The DMP uses burst of four cell read accesses to perform Read Type I and Type II operations. Figure 26 shows an example timing diagram of such an access.

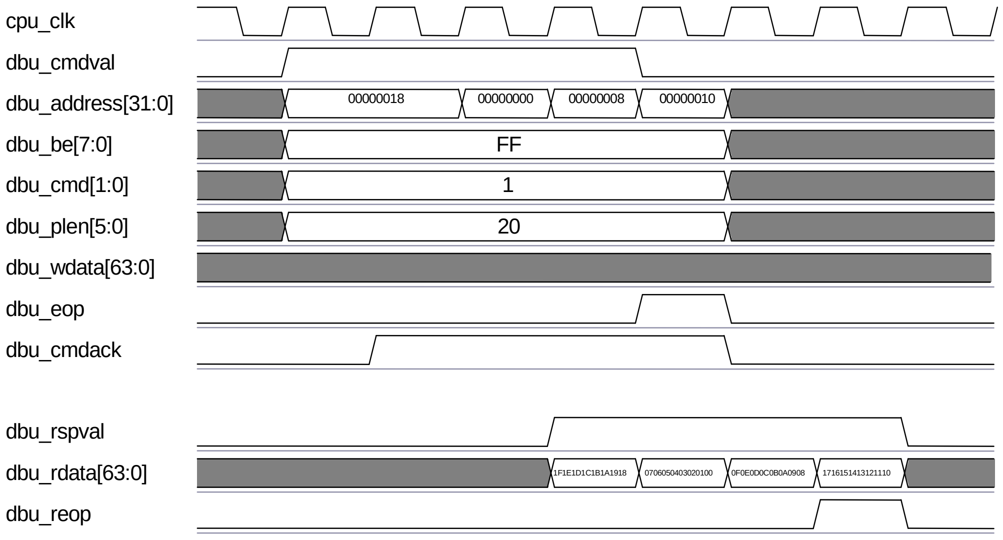


Figure 26 Burst of 4 Cell Packet Read Access on BVCI Target Interface between DMP and IBUS

In the diagram the DMP initiates the start of an access by asserting the command valid signal (`dbu_cmdval` = 1) and presents the address (`dbu_address`), packet length (`dbu_plen`), end of packet (`dbu_eop` = 0) and access command (`dbu_cmd` = 1 = Read) of the access on the command bus.

This first command cell is then accepted by the IBUS target interface by asserting the command acknowledgement signal (`dbu_cmdack` = 1) either in the same clock cycle (known as default acknowledgement), or after one (as in the example above) or more clock cycles later.

With each new command acknowledgement, the DMP initiator present the next address, qualifying it with the command valid signal each time. The address is normally incremented by 8 bytes, however, when the address crosses the boundary of the packet length, the address wraps around to the first byte address of the packet boundary, which in the case of the example above, is address 0 in the second command cell. At the last address cell, the End Of Packet signal is asserted (`dbu_eop` = 1) as well to denote that the command packet is completed.

When the data of this access is available one or more cycles after the beginning of the access, the IBUS module presents it on the data bus (`dbu_rdata`) and qualifies it with the response valid signal (`dbu_rspval` = 1). Each data cell is acknowledged by the initiator interface by default. Wait cycles can be inserted between each data cell by de-asserting the response valid.

Burst of 4 Cell Write Accesses

The DMP uses burst of four cell write accesses to perform Write Type I operations. Figure 27 shows an example timing diagram of such an access. In the diagram the DMP initiates the start of an access by asserting the command valid signal (`dbu_cmdval` = 1) and presents the address (`dbu_address`), packet length (`dbu_plen`), end of packet (`dbu_eop` = 0), write data (`dbu_wdata`) and access command (`dbu_cmd` = 2 = Write) of the access on the command bus.

This first command cell is then accepted by the IBUS target interface by asserting the command acknowledgement signal (`dbu_cmdack` = 1) either in the same clock cycle (known as default acknowledgement), or after one (as in the example above) or more clock cycles later. With each new command acknowledgement, the DMP initiator presents the next address and data, qualifying it with the command valid signal each time. The address is normally incremented by 8 bytes, however, when the address crosses the boundary of the packet length, the address wraps around to the first byte address of the packet boundary, which in the case of the Figure 26, is address 0 in the second command cell. At the last command cell, the End Of Packet signal is asserted (`dbu_eop` = 1) as well to denote that the command packet is completed.

When the data of this access is written one or more cycles after the beginning of the access, the IBUS module responds using the valid signal (`dbu_rspval` = 1). Each data cell is acknowledged by the initiator interface by default. Wait cycles can be inserted between each response cell by de-asserting the response valid signal.

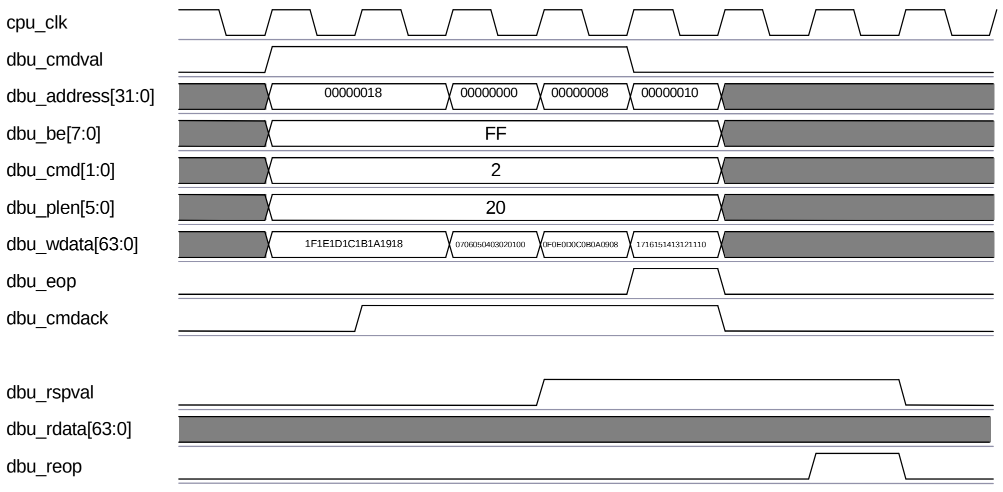


Figure 27 Burst of 4 Cell Packet Write Access on BVCI Target Interface between DMP and IBUS

Single Cell locked Read Accesses

The DMP uses single cell locked read accesses to perform Locked Read operations. Figure 28 shows an example timing diagram of such an access.

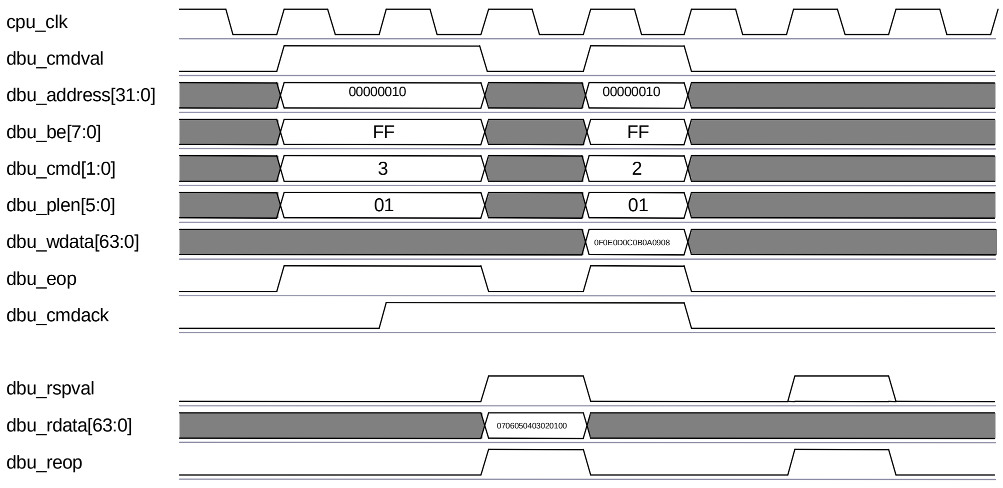


Figure 28 Locked Read Access followed by Single Write Access on BVCI Target Interface between DMP and IBUS

1. The first read access proceeds in a very similar way to a single read access except that the command is a Locked Read (`dbu_cmd` = 0x3).
2. Once the command cell is acknowledged in the third clock cycle, the external arbiter design must lock the bus to the same initiator (DMP-bus bridge), giving it default grant until it receives a single or burst write access from the same DMP-bus bridge initiator interface.

Chapter 5 — Closely Coupled Memories (CCM)

The following subsections cover the direct memory interfaces that are available on the Instruction Closely Coupled Memory (ICCM) and the Data Closely Coupled Memory (DCCM):

- Closely Coupled Memories
- CCM DMI Interfaces

Closely Coupled Memories

CCMs are used to complement or replace traditional instruction and data cache memories. Unlike standard cache architectures, CCMs are passive memories that attach to the instruction and data fetch interfaces of the processor, and provide fast data and program code access. It is the responsibility of the programmer to ensure that valid program data exists in the ICCM and valid data in the DCCM.

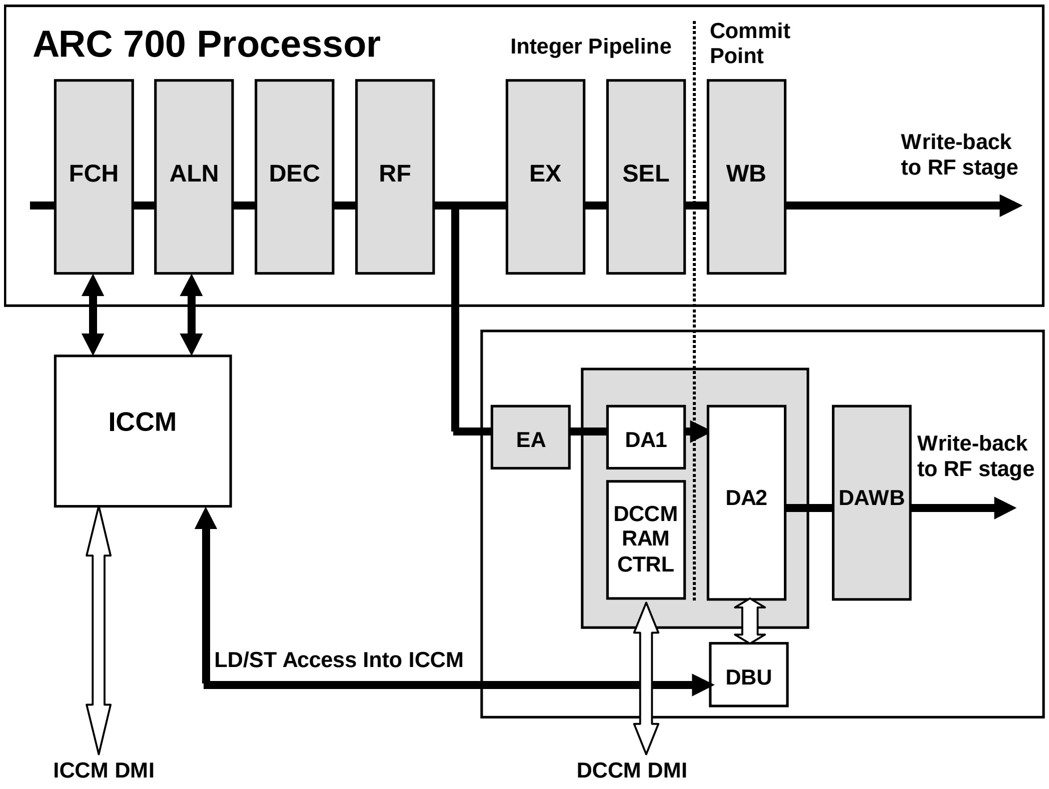


Figure 29 ICCM and DCCM Configuration Example

CCM DMI Interfaces

The ICCM and DCCM memories support a direct memory interface into the RAMs (DMI). The purpose of the DMI is to allow an external client, such as a DMA engine, to initialize the contents of the RAMs prior to processor execution. It is also possible to modify the contents of the RAMs whilst the CPU is in a 'run' state, however data coherency issues must be considered.

CCM DMI Signal List

The following CCM DMI BVCI interface signals may appear on the CPU Island.

Table 15 DCCM Direct Memory Interface (DMI)

ICCM Signals	DCCM Signals	Direction	Bus Width	Description
`iccm_dmi_address`	`dccm_address`	Input	32	Byte Address. The ccm_address is updated on every cycle during a burst.
`iccm_dmi_be`	`dccm_be`	Input	4	Byte Enables. The requests to this interface can be of the size byte, word or longword (32-bit), therefore this signal should be set depending upon the size of the requested cell.
`iccm_dmi_cmd`	`dccm_cmd`	Input	2	Command. The type of command to be performed is specified by this bus: Read – 0x1 Write – 0x2 This bus is qualified with dccm_cmdval.
`iccm_dmi_cmdval`	`dccm_cmdval`	Input	1	Command is Valid. The values on ccm_cmd, ccm_address, are valid when this signal is true.
`iccm_dmi_contig`	`dccm_contig`	Input	1	Unused
`iccm_dmi_eop`	`dccm_eop`	Input	1	End of Packet. The signal dccm_eop is asserted on the last burst cycle to indicate the end of that burst request.
`iccm_dmi_plen`	`dccm_plen`	Input	6	Unused
`iccm_dmi_rspack`	`dccm_rspack`	Input	1	Response Acknowledge. This signal tells the CCM that the received valid data has been acknowledged.
`iccm_dmi_wdata`	`dccm_wdata`	Input	32	Write Data. This is the data from a write request (dccm_cmd = 0x2) and is qualified when dccm_cmdval is true.
`iccm_dmi_wrap`	`dccm_wrap`	Input	1	Unused

ICCM Signals	DCCM Signals	Direction	Bus Width	Description
`iccm_dmi_cmdack`	`dccm_cmdack`	Output	1	Command Acknowledge. This signal acknowledges every cell during an operation.
`iccm_dmi_rdata`	`dccm_rdata`	Output	32	Read Data. This is the returning data from a read request (dccm_cmd = “01”) and is qualified when dccm_rspval is true.
`iccm_dmi_reop`	`dccm_reop`	Output	1	End of packet.
`iccm_dmi_rerror`	`dccm_rerror`	Output	1	Unused
`iccm_dmi_rspval`	`dccm_rspval`	Output	1	Response Valid. The dccm_rspval acknowledges both read and write data.

Interface Reset State

Upon a global reset all signals on this interface are set to zero. This is also expected to be the state of the interface at time zero for simulation purposes.

CCM DMI Behavior

The CCM DMI interfaces supports all the command modes provided by the BVCI protocol (Refer to the Virtual Component Interface Standard), and the capabilities of both the ICCM and DCCM are identical.

The CCM’s support the following types of operation:

- Read Type I – A burst read operation
- Read Type II – A single cell read operation of a byte, word, or longword (32-bits) size
- Write Type I – A burst write operation
- Write Type II – A single cell write operation of a byte, word, or longword (32-bits) size

Read Type I Timing Behavior

The memory requesting device issues a read burst requests to the CCM controller. A cycle-by-cycle description:

1. **Time = 10ns**. The address is set up via *ccm_address* = ADDR0 when a read (*ccm_cmd* = 0x1) is performed. This access is valid (*ccm_cmdval* = ‘1’) and this access has a burst length of 32 bytes, i.e. *ccm_plen* = 0x20. The write data (*ccm_wdata*) is ignored. Also ccm_contig and ccm_wrap are ignored, because the CCM control module does not service the request differently if any of these are set. All bytes are returned during read operations, so ccm_be is actually ignored by the CCM.
2. **Time = 20ns**. The CCM control module acknowledges the read request (*ccm_cmdack* = ‘1’). The address, access type, and qualifier signals are maintained, i.e. *ccm_address* = ADDR0, *ccm_cmd* = 0x1 and *ccm_cmdval* = ‘1’ respectively.
3. **Time = 30ns**. Data (*ccm_rdata* = DATA0) is returned to the memory requesting device and it is valid (*ccm_rspval* = ‘1’). The memory requesting device has the response acknowledgement default set in this example (*ccm_rspack* = ‘1’), which means that the device immediately acknowledges the returning data. Also, the address is set up for the next

access via *ccm_address* = ADDR1 for the read (*ccm_cmd* = 0x1) is performed. This access is valid (*ccm_cmdval* = ‘1’). This is a contiguous access (*ccm_contig* = ‘1’) and all bytes are to be written back to the CCM (*ccm_be* = 0xFF). The write data (*ccm_wdata*) is ignored. The CCM control module acknowledges the read request (*ccm_cmdack* = ‘1’) made on this cycle.

4. **Time = 40ns**. Data (*ccm_rdata* = DATA1) is returned to the CCM control module and it is valid (*ccm_rspval* = ‘1’). The memory requesting device also acknowledges receipt of the received data (*ccm_rspack* = ‘1’). The address is set up for the next access (*ccm_address* = ADDR2) in the burst sequence. The CCM control module acknowledges the read request (*ccm_cmdack* = ‘1’) made on this cycle.
5. **Time = 50ns**. Data (*ccm_rdata* = DATA2) is returned to the CCM control module and it is valid (*ccm_rspval* = ‘1’). The requesting device also acknowledges receipt of the received data (*ccm_rspack* = ‘1’). The address is set up for the next access (*ccm_address* = ADDR3) in the burst sequence. This is the last request in the burst sequence, which is indicated by the end of packet being set (*ccm_eop* = ‘1’).
6. **Time = 60ns**. Data (*ccm_rdata* = DATA3) is returned to the CCM control module and it is valid (*ccm_rspval* = ‘1’). The CCM control module confirms that this data is last in the burst (*ccm_reop* = ‘1’) and the requesting device acknowledges receipt of the received data (*ccm_rspack* = ‘1’). The requesting device has no more valid requests to make to the CCM (*ccm_cmdval* = ‘0’).
7. **Time = 70ns**. There are no further requests by the requesting device (*ccm_cmdval* = ‘0’).

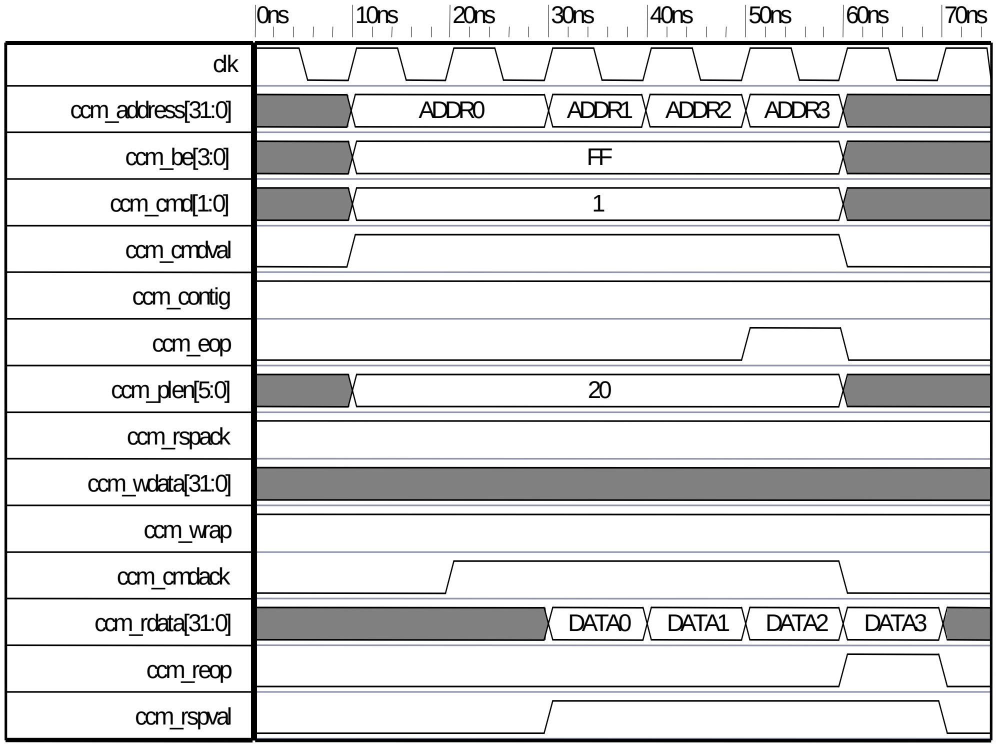


Figure 30 Read Burst on the CCM Burst Interface

Read Type II Timing Behavior

The memory requesting device issues a single cell read request to the CCM. A cycle-by-cycle description:

1. **Time = 10ns**. The address is set up via *ccm_address* = ADDR. A read (*ccm_cmd* = 0x1) is performed. This access is valid (*ccm_cmdval* = ‘1’) and this access has a burst length of 4 bytes, i.e. *ccm_plen*= 0x04. The minimum amount of data that can be read on a BVCI interface is a cell, which in this case is 4 bytes (32-bits). All 4 bytes will be sent back, because during read operations the level of granularity is one cell. It is up to the memory requestor to extract the relevant bytes, when it receives the requested cells. As it is a read operation both the write data (*ccm_wdata*) and the byte enables (*ccm_be*) are ignored.
2. **Time = 20ns**. The CCM acknowledges the read request (*ccm_cmdack* = ‘1’). The address, access type, and qualifier signals are maintained, i.e. *ccm_address* = ADDR, *ccm_cmd* = 0x1 and *ccm_cmdval* = ‘1’ respectively.
3. **Time = 30ns**. There are no further requests by the CCM control (*ccm_cmdval* = ‘0’). Data (*ccm_rdata* = DATA) is returned to the CCM control module and it is valid (*ccm_rspval* = ‘1’). The memory requestor also acknowledges receipt of the received data (*ccm_rspack* = ‘1’). The write data (*ccm_wdata*) is ignored. There are no further requests by the memory requestor (*dccm_cmdval* = ‘0’).

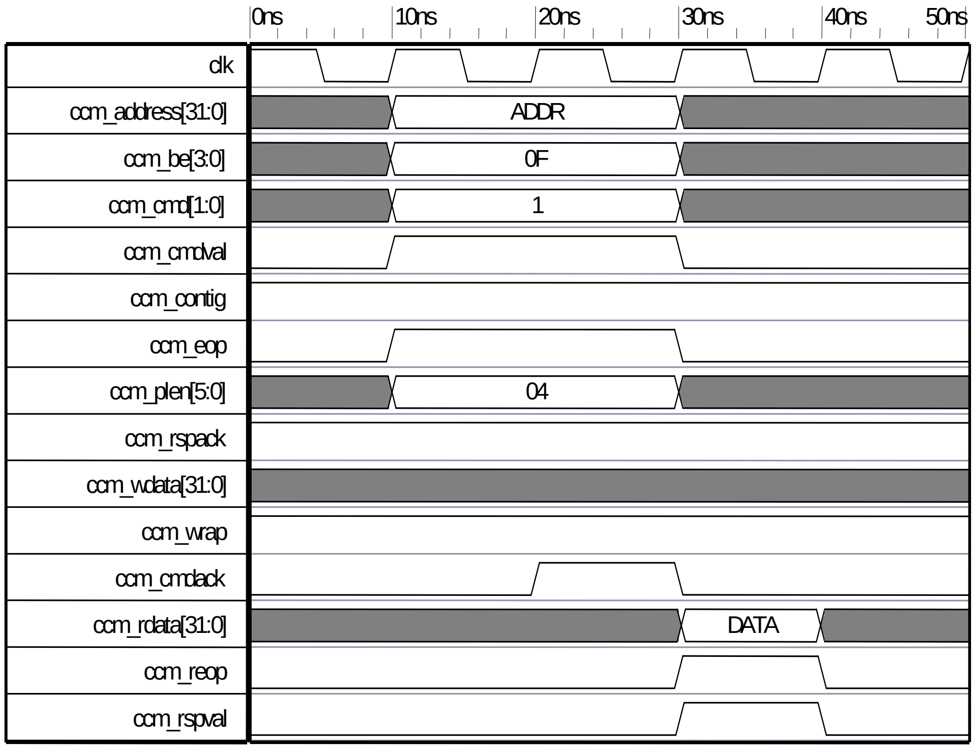


Figure 31 Single Cell Read Operation on the CCM DMI

Write Type I Timing Behavior

The memory requestor issues a burst write request to the CCM. A cycle-by-cycle description:

1. **Time = 10ns**. The address is set up via *ccm_address* = ADDR0 when a write (*ccm_cmd* = 0x2) is performed. This access is valid (*ccm_cmdval* = ‘1’) and this access has a burst length of 32 bytes, i.e. *ccm_plen* = 0x20. This is a contiguous access (*ccm_contig* = ‘1’) and all bytes are to be written to the CCM (*ccm_be* = 0xFF). The write data (*ccm_wdata* = DATA0) is valid.

2. **Time = 20ns**. The CCM acknowledges the write request (*ccm_cmdack* = ‘1’). The address, data, access type, and qualifier signals are maintained, i.e. *ccm_address* = ADDR0, *ccm_wdata* = DATA0, *ccm_cmd* = 0x2 and *ccm_cmdval* = ‘1’ respectively.
3. **Time = 30ns**. The write operation has completed successfully (*ccm_rspval* = ‘1’). The memory requestor acknowledges receipt of the written data (*ccm_rspack* = ‘1’). The address is set up for the next access via *ccm_address* = ADDR1 and write data ccm_wdata = DATA1. The next access is immediately acknowledged (ccm_cmdack = ‘1’).
4. **Time = 40ns**. The write operation has completed successfully (*ccm_rspval* = ‘1’). The memory arbitrator acknowledges receipt of the written data (*ccm_rspack* = ‘1’). The address is set up for the next access via *ccm_address* = ADDR2 and write data *ccm_wdata* = DATA2. The next access is immediately acknowledged (*ccm_cmdack* = ‘1’).
5. **Time = 50ns**. The write operation has completed successfully (*ccm_rspval* = ‘1’). The memory requestor acknowledges receipt of the written data (*ccm_rspack* = ‘1’). The address is set up for the next access via *ccm_address* = ADDR3 and write data *ccm_wdata* = DATA3. This is the last request in the burst (*ccm_eop* = ‘1’). The last access is immediately acknowledged (*ccm_cmdack* = ‘1’).
6. **Time = 60ns**. The write operation has completed successfully (*ccm_rspval* = ‘1’). The CCM control module confirms that this data is last in the burst (*ccm_reop* = ‘1’) and the memory requestor acknowledges receipt of the received data (*ccm_rspack* = ‘1’). The CCM control module has no more valid requests to make to the CCM (*ccm_cmdval* = ‘0’). The address, write data and access type can be ignored.
7. **Time = 70ns**. There are no further requests by the memory requestor (*ccm_cmdval* = ‘0’).

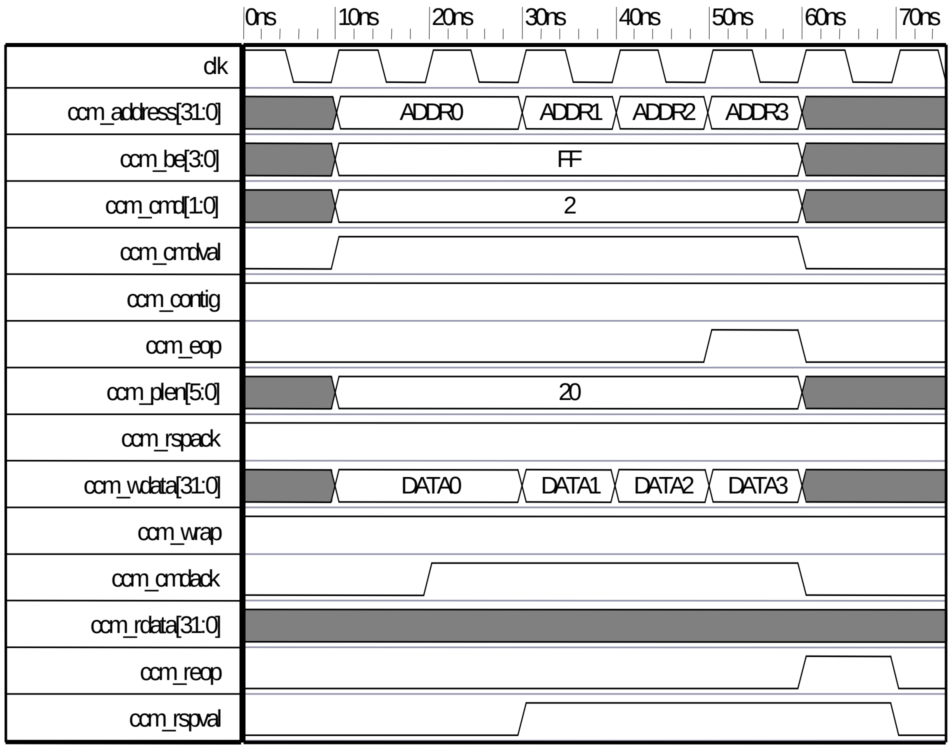


Figure 32 Write Burst Operation to the CCM DMI

Write Type II Timing Behavior

The memory requestor issues a single cell write request to the CCM. A cycle-by-cycle description:

1. **Time = 10ns**. The address is set up via *ccm_address* = ADDR when a write (*ccm_cmd* = 0x2) is performed. This access is valid (*ccm_cmdval* = ‘1’) and this access has a burst length of 4 bytes, i.e. *ccm_plen* = 0x4. The lower 4 bytes are to be written to the CCM (*ccm_be* = 0x0F). The write data (*ccm_wdata* = DATA) is valid.
2. **Time = 20ns**. The CCM control module acknowledges the write request (*ccm_cmdack* = ‘1’). The address, data, access type, and qualifier signals are maintained, i.e. *ccm_address* = ADDR, *ccm_wdata* = DATA, *ccm_cmd* = 0x2 and *ccm_cmdval* = ‘1’ respectively.
3. **Time = 30ns**. The write operation has completed successfully (*ccm_rspval* = ‘1’). The CCM control module confirms that this data is last in the burst (*ccm_reop* = ‘1’) and the memory requestor acknowledges receipt of the received data (*ccm_rspack* = ‘1’). There are no further requests by the memory requestor (*ccm_cmdval* = ‘0’).
4. **Time = 40ns**. The memory requestor has no more valid requests to make to the CCM (*ccm_cmdval* = ‘0’). The address, write data and access type can be ignored.

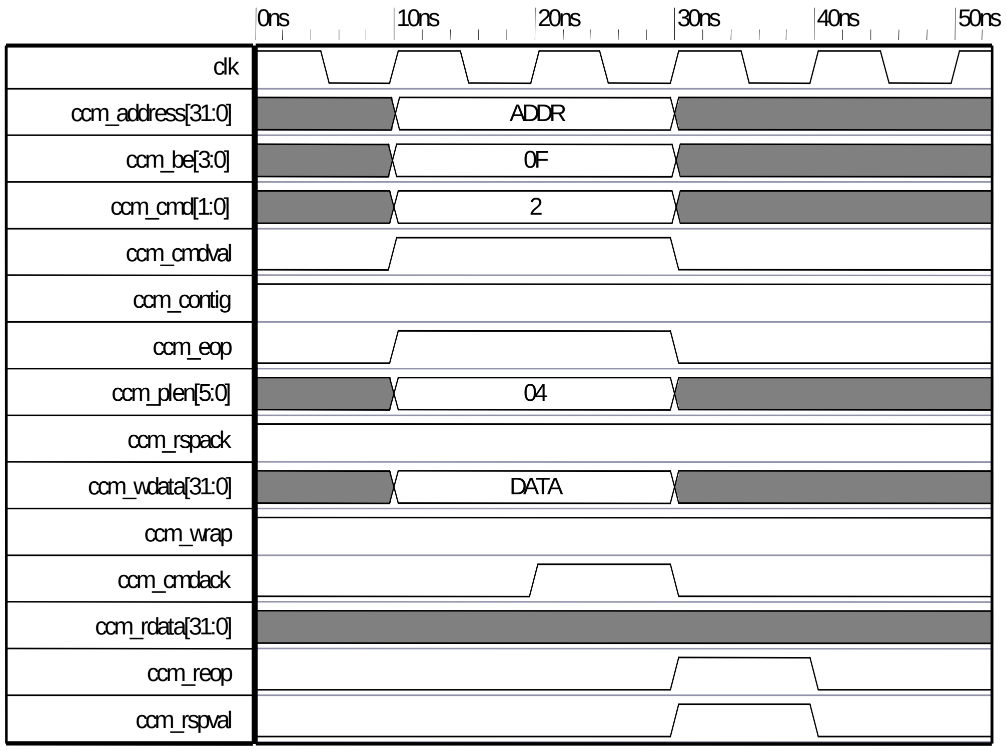


Figure 33 Single Cell Write Operation on the CCM DMI

Chapter 6 — XY Memory

The following subsections cover the direct memory interfaces that are available on the XY Memory Module:

- XY Memory
- XY DMI interface

XY Memory

The XY Memory module is an optional DSP extension to the ARC 700 processor core. This extension provides a high data throughput closely coupled memory, accessible via pointer, that can be automatically updated. The XY memory extension contains two memory regions of equal size, each configurable at build time from 4K up to 32K each. Also configurable at build time is a direct memory interface (DMI). The DMI enables an external client, such as a DMA engine, to perform the following:

- Initialize the contents of the RAMs prior to processor execution.
- Modify or upload the contents of the RAMs whilst the CPU is running.
- Read or offload the contents of the RAMs whilst the CPU is running.

Since no coherence protection is provided with the XY memory DMI port, the user has to be aware that other software or hardware mechanisms may be required to deal with data coherency.

For more details on the XY memory module, please refer to the *ARC 700 DSP Options Reference*.

XY DMI interface

The XY DMI interface is an optional DMI interface. The XY DMI Signal List section lists the signals on the DMI interface.

XY DMI Signal List

The following XY DMI BVCI interface signals may appear on the CPU Island:

Table 16 XY DMI Interface Signals

Signals	**Direction**	**Bus Width**	**Description**
`xydmi_address`	Input	N	Byte Address. It is updated on every cycle during a burst. N varies with the size of each memory region: • 4k per region, N = 13 • 8k per region, N = 14 • 16k per region, N = 15 • 32k per region, N = 16
`xydmi_be`	Input	8	Byte Enables. The requests to this interface can be of the size byte, word , longword (32-bit) or double longword (64bits). Therefore this signal should be set depending upon the size of the requested cell.
`xydmi_cmd`	Input	2	Command. The type of command to be performed is specified by this bus: Read – 0x1 Write – 0x2 This bus is qualified with xydmi_cmdval.

Signals	Direction	Bus Width	Description
`xydmi_cmdval`	Input	1	Command is Valid. The values on xydmi_cmd, xydmi_address, xydmi_be and xydmi_eop are valid when this signal is true.
`xydmi_eop`	Input	1	End of Packet. The signal dccm_eop is asserted on the last burst cycle to indicate the end of that burst request.
`xydmi_rspack`	Input	1	Response Acknowledge. This signal tells the XY memory module that the received valid data has been acknowledged.
`xydmi_wdata`	Input	64	Write Data. This is the data from a write request (xydmi_cmd = 0x2) and is qualified when xydmi_cmdval is true.
`xydmi_cmdack`	Output	1	Command Acknowledge. This signal acknowledges every cell during an operation.
`xydmi_rdata`	Output	64	Read Data. This is the returning data from a read request (xydmi_cmd = “01”) and is qualified when xydmi_rspval is true.
`xydmi_reop`	Output	1	End of packet.
`xydmi_rspval`	Output	1	Response Valid. The xydmi_rspval acknowledges both read and write data.

The X and the Y memory regions are mapped onto the address space with the X region occupying the lower half of the memory area and Y region occupying the upper half.

Interface Reset State

Upon a global reset all signals on this interface are set to zero. This is also expected to be the state of the interface at time zero for simulation purposes.

XY DMI Behavior

The XY DMI interfaces supports all the command modes provided by the BVCI protocol (Refer to the Virtual Component Interface Standard). The following types of operation are supported:

- Read Type I – A burst read operation
- Read Type II – A single cell read operation of a byte, word, longword (32 bits) or double longword (64 bits) size.
- Write Type I – A burst write operation
- Write Type II – A single cell write operation of a byte, word, longword (32bits) or double longword (64 bits) size.

Read Type I Timing Behavior

The memory requesting device issues a read burst request to the XY memory DMI. A cycle-by-cycle description of an example follows:

- **Time = 10ns**. The burst access starts, with the address, byte enable, and read command placed on the bus on *xydmi_address*, *xydmi_be* and *xydmi_cmd* respectively with *xydmi_cmdval* set to high. The write data (*xydmi_wdata*) is ignored.
- **Time = 20ns**. The XY DMI default acknowledges the read request (*xydmi_cmdack* = ‘1’) and sets the command acknowledge signal to low so that the next cell in the packet is not acknowledged

immediately. With the cell acknowledged, the address (*xydmi_address)* is incremented for the next cell transfer.

- **Time = 30ns**. The request made to XY memory by the XY DMI is granted in this cycle. Wait cycle on the BVCI command bus
- **Time = 40ns**. The data from XY memory is returned for the request made by XY DMI and is registered and presented onto the read data bus (*xydmi_rdata*), with the valid signal (*xydmi_rspval*) set to high.
- **Time** = 50ns. The response is acknowledged by *xydmi_rspack* being high. This sets the command acknowledgment signal (*xydmi_cmdack*) to go high, ready to accept the next command cell, and the response valid (*xydmi_rspval*) signal to go low.
- **Time = 60ns to 130ns**, repeats the same process between 10ns to 50ns, but for the second and third command cell.
- **Time = 140ns,** The last command cell is acknowledged, which has *xydmi_eop* set to high, and set the command acknowledge signal to low.
- **Time = 160ns**. Read data becomes available on the response bus.
- **Time = 170ns**. Since *xydmi_rspack* is low, the response cell has not been acknowledged and the XY DMI interface keeps the response cell for another cycle on bus.
- **Time = 180ns**. With *xydmi_rspack* at high, the response cell has been acknowledged. This completes the burst (packet) transfer.

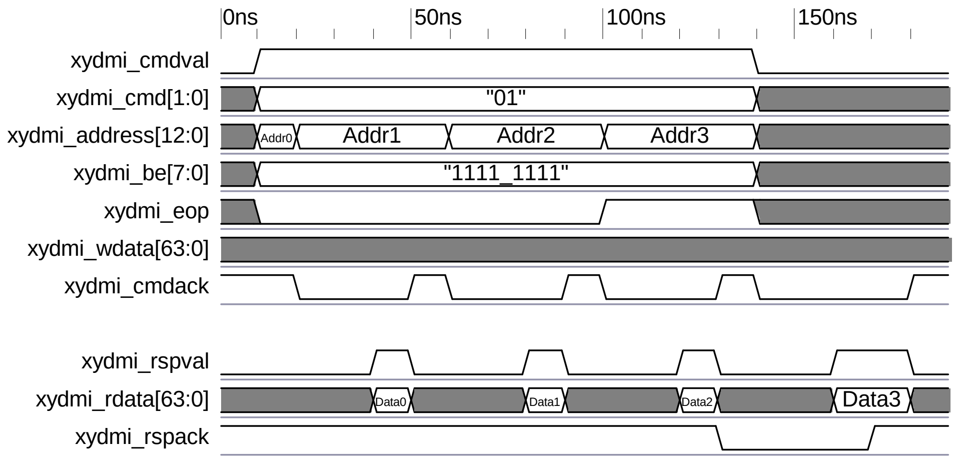


Figure 34 Read Burst Access on XY Memory DMI.

Read Type II Timing Behavior

The memory requesting device issues a single read request to the XY memory DMI. A cycle-by-cycle description of an example follows:

- **Time = 10ns**. The access starts, with the address, byte enable, and read command placed on the bus on *xydmi_address*, *xydmi_be* and *xydmi_cmd* respectively with *xydmi_cmdval* set to high. The end of packet signal (*xydmi_eop*) is also set to high to indicate that it is a single access. The write data (*xydmi_wdata*) is ignored.
- **Time = 20ns**. The XY DMI default acknowledges the read request (*xydmi_cmdack* = ‘1’) and sets the command acknowledge signal to low so that the next command cell is not acknowledged

immediately. With the cell acknowledged, the command valid signal (*xydmi_cmdval)* is de-asserted.

- **Time = 30ns**. The request made to XY memory by the XY DMI is granted in this cycle. Wait cycle on the BVCI command bus
- **Time = 40ns**. The data from XY memory is returned for the request made by XY DMI and is registered and presented onto the read data bus (*xydmi_rdata*), with the valid signal (*xydmi_rspval*) set to high.
- **Time** = 50ns. The response is acknowledged by *xydmi_rspack* being high. This sets the command acknowledgment signal (*xydmi_cmdack*) to go high, ready to accept the next command cell, and the response valid (*xydmi_rspval*) signal to goes low.

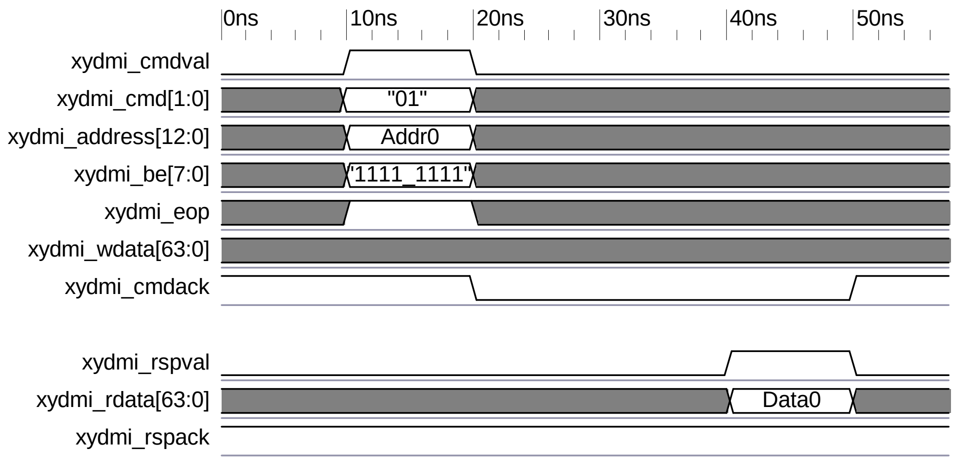


Figure 35 Single Read Access on XY Memory DMI.

Write Type I Timing Behavior

The memory requesting device issues a write burst request to the XY memory DMI. A cycle-by-cycle description of an example follows:

- **Time = 10ns**. The burst access starts, with the address, byte enable, write data and write command placed on the bus on *xydmi_address*, *xydmi_be, xydmi_wdata* and *xydmi_cmd* respectively with *xydmi_cmdval* set to high.
- **Time = 20ns**. The XY DMI default acknowledges the write request (*xydmi_cmdack* = ‘1’) and sets the command acknowledge signal to low so that the next cell in the packet is not acknowledged immediately. With the cell acknowledged, the address (*xydmi_address)* is incremented for the next cell transfer.
- **Time = 30ns**. The XY DMI made a request to XY memory and is granted in this cycle. Hence XY DMI sets the response valid signal (*xydmi_rspval*) to high
- **Time = 40ns**. The response is acknowledged by *xydmi_rspack* being high. This sets the command acknowledgment signal (*xydmi_cmdack*) to go high, ready to accept the next command cell, and the response valid (*xydmi_rspval*) signal to go low.
- **Time = 50ns to 100ns**, repeats the same process between 10ns to 50ns, but for the second and third command cell.

- **Time = 110ns,** The last command cell is acknowledged, which has `xydmi_eop` set to high, and set the command acknowledge signal to low.
- **Time = 120ns**. Response valid (`xydmi_rspval`) go high.
- **Time = 130ns**. Since `xydmi_rspack` is low, the response cell has not been acknowledged and the XY DMI interface keeps the response cell for another cycle on bus.
- **Time = 140ns**. With `xydmi_rspack` at high, the response cell has been acknowledged. This completes the burst (packet) transfer.

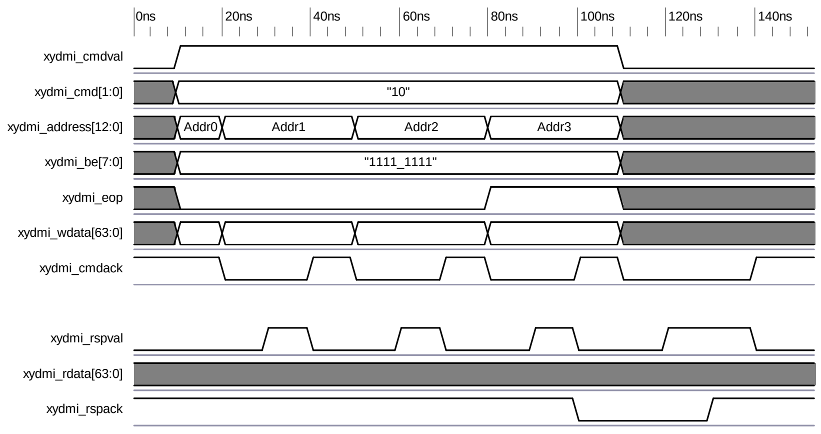


Figure 36 Burst Write Access on XY DMI

Write Type II Timing Behavior

The memory requesting device issues a single burst write requests to the XY memory DMI. A cycle-by-cycle description of an example follows:

- **Time = 10ns**. The access starts, with the address, byte enable, write data and write command placed on the bus on `xydmi_address`, `xydmi_be, xydmi_wdata` and `xydmi_cmd` respectively with `xydmi_cmdval` set to high. The end of packet signal (`xydmi_eop`) is also set to high to indicate that it is a single access.
- **Time = 20ns**. The XY DMI default acknowledges the write request (`xydmi_cmdack` = ‘1’) and set the command acknowledge signal to low so that the next cell in the packet is not acknowledged immediately. With the cell acknowledged, the command valid signal (`xydmi_cmdval)` is de-asserted.
- **Time = 30ns**. The XY DMI made a request to XY memory and is granted in this cycle. And hence XY DMI sets the response valid signal (`xydmi_rspval`) to high.
- **Time** = 40ns. The response is acknowledged by `xydmi_rspack` being high. This sets the command acknowledgment signal (`xydmi_cmdack`) to go high, ready to accept the next command cell, and the response valid (`xydmi_rspval`) signal to goes low.

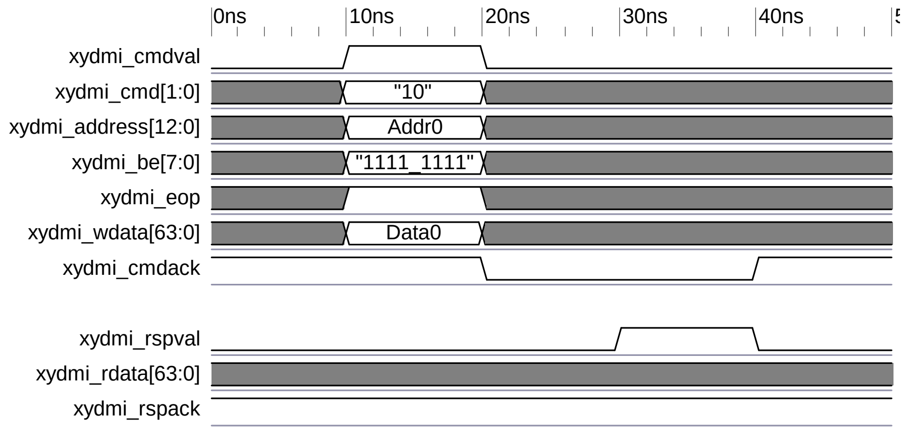


Figure 37 Single Write Access on XY DMI

Chapter 7 — Processor Signals

The processor signals are used to run, clock and interrupt the processor core.

The processor signals are covered in the following subsections:

- Processor Control Interface
- Interrupt Unit
- Test

Processor Control Interface

The processor interface signals are used to run and clock the processor core. The following sections cover the processor control interface in more detail:

- Processor Signal List
- Clocks
- Reset
- Start
- Run

Processor Signal List

The following processor control interface signals may appear on the CPU Island:

Table 17 Processor Control Signal List

Signal	Direction	Description
`clk_cpu`	Input	*Processor Core Clock.*
`clk_sys`	Input	*External Memory System Clock.*
`rst_a`	Input	*Reset* - Asynchronous, and active high.
`ctrl_cpu_start_r`	Input	*Start* - Depends on configuration.
`ctrl_arch_status32_h_r`	Output	*Run* - Set high when processor is halted.

Clocks

The ARC 700 processor is a fully static design, and uses two positive edge clocks `clk_cpu` and `clk_sys`.

`clk_cpu` is the ARC 700 processor core clock. `clk_sys` is the external memory system clock.

These clock nets are not buffered in the design, since it is intended that clock tree synthesis technique will be used.

The supported clock frequency ratios are dependent on the particular CPU Island interfaces in the design. For example see the BVCI Bus Bridge Clock Synchronization Unit.

For additional information on alternative CPU Island interfaces see *AHB Bus Bridge Reference*, *AXI Bus Bridge Reference*, and *ARC Legacy Bus Bridge Reference*.

Reset

The reset net `rst_a` is asynchronous, and active high. ARC International recommends that the reset signal be arranged to be asynchronously applied and synchronously removed. Reset should be applied for a minimum of four clock cycles. The synthesis tool should be allowed to buffer the `rst_a` net.

Start

The start signal, `ctrl_cpu_start_r`, is used to start the processor with particular configurations that are set to halt-on-reset.

Run

The run signal, `ctrl_arch_status32_h_r` is an output signal that is set high when processor is halted.

Interrupt Unit

The ARC 700 system features a configurable interrupt unit that allows selection of 8, 16, or 32 interrupt inputs. The interrupt unit generates interrupt requests (IRQs) to the CPU and has the ability to bring the CPU out of sleep mode when a valid interrupt request is present.

All interrupts can either be pulse or level triggered as well as having individual mask bits and priority levels.

The number of user interrupts lines is dependant upon the number of interrupts that are configured in ARChitect configuration tool.

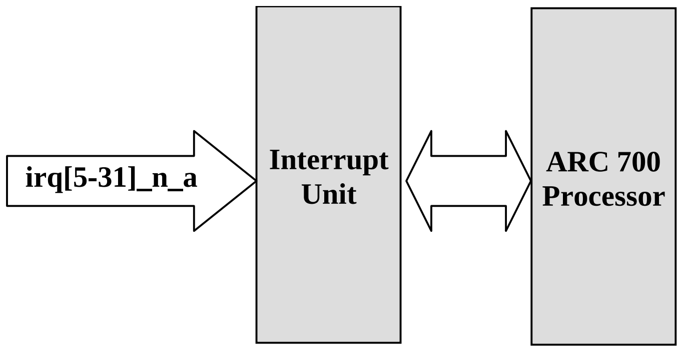


Figure 38 Interrupt Interface

The following sections cover the interrupt interface in more detail:

- Feature List
- Interrupt Signal List
- Incoming Request Interface Timing

Feature List

- Maximum of 26 user-definable IRQs (5 to 31)
- Programmable interrupt type on all IRQs (pulse, level)
- The lowest interrupt number has the highest interrupt priority
- Programmable mask bit on all IRQs
- Programmable priority level (level 1 = low, level 2 = high) on all IRQs
- Software controlled triggers for all IRQs

Interrupt Signal List

The following interrupt interface signals may appear on the CPU Island:

Table 18 Configurable Interrupt Lines

Signal Name	Total Number of User Interrupt Lines	ARChitect Selection	Direction	Purpose
`irq[5:7]_n_a`	3	8 Interrupts	In	Interrupt Request signal
`irq[8:15]_n_a`	11	16 Interrupts	In	Interrupt Request signal
`irq[16:31]_n_a`	27	32 Interrupts	In	Interrupt Request signal

The `irqxx_n_a` signal can be level or pulse type and is asynchronously applied. If pulse type interrupts are used, then the minimum width of the signal should be twice that of the interrupt unit clock. When the `irqxx_n_a` signal is asserted (all interrupts are active low) it raises an interrupt request to the interrupt unit. The interrupt unit will then decide if the signal is legal based on the enable and mask bits. If the interrupt signal type is level, then it is up to the signal source to remove it once the interrupt has been accepted by the CPU (this should be done by the interrupt service routine). If the interrupt type is pulse, then the interrupt unit will register the signal and it is up to the interrupt service routine to clear the interrupt by using the AUX_IRQ_PULSE_CANCEL register.

Incoming Request Interface Timing

Figure 39 shows a level type interrupt request. At time 10ns `irqx_n_a` is asserted (goes low). Interrupt X is a level type interrupt, therefore it remains asserted until the issuing device removes it.

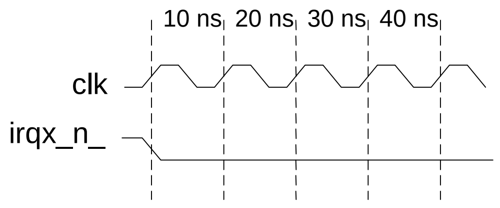


Figure 39 Example Level Type Interrupt

Figure 40 shows a pulse type interrupt request. At time 10ns `irqx_n_a` is asserted for a minimum period of x2 the clock period. `irqx` is a pulse type interrupt, therefore the processor registers the interrupt request, and once serviced, it is the responsibility of the interrupt service routine to clear the interrupt within the interrupt unit (using AUX_IRQ_PULSE_CANCEL).

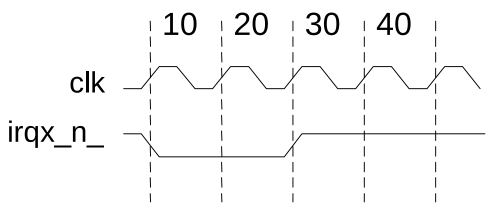


Figure 40 Example Pulse Type Interrupt

Test

The input signal `xtest_mode_atpg` sets the processor in a mode which is optimized for good fault coverage. Only present if the configuration requires some modification in test mode. Not all configurations have this signal.

The input signal `xtest_mode_rambist` allows the built in self test (BIST) control unit, provided by customers, to gain access to the RAMs. Only present if the ARChitect option **-bist_muxes** has been selected.

The test interface signals are summarized in Table 19.

Test Signal List

The following test interface signals may appear on the CPU Island:

Table 19 Test Signal List

Signal	Direction	Description
`xtest_mode_atpg`	Input	*ATPG Test Mode.*
`xtest_mode_rambist`	Input	*RAM BIST Test Mode.*