Functional Description—HPS Memory Controller

4

2013.12.16

emi_rm_017

The hard processor system (HPS) SDRAM controller subsystem provides efficient access to external SDRAM for the ARM® Cortex™-A9 microprocessor unit (MPU) subsystem, the level 3 (L3) interconnect, and the FPGA fabric. The SDRAM controller provides an interface between the FPGA fabric and HPS. The interface accepts Advanced Microcontroller Bus Architecture (AMBA®) Advanced eXtensible Interface (AXI™) and Avalon® Memory-Mapped (Avalon-MM) transactions, converts those commands to the correct commands for the SDRAM, and manages the details of the SDRAM access.

Features of the SDRAM Controller Subsystem

The SDRAM controller subsystem offers the following features:

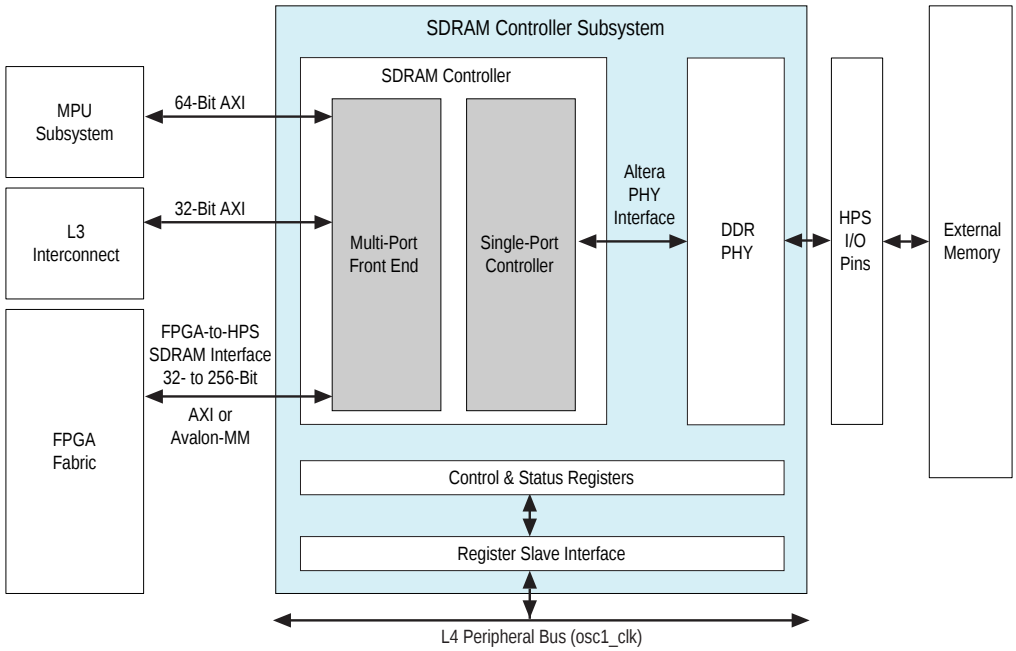
- Support for double data rate 2 (DDR2), DDR3, and low-power DDR2 (LPDDR2) SDRAM
- User-configurable timing parameters
- Up to 4 Gb density parts
- Two chip selects
- Integrated error correction code (ECC), 24- and 40-bit widths
- User-configurable memory width of 8, 16, 16+ECC, 32, 32+ECC
- Command reordering (look-ahead bank management)
- Data reordering (out of order transactions)
- User-controllable bank policy on a per port basis for either closed page or conditional open page accesses
- User-configurable priority support with both absolute and relative priority scheduling
- Flexible FPGA fabric interface configuration with up to 6 ports and data widths up to 256 bits wide using Avalon-MM and AXI interfaces.
- Power management supporting self refresh, partial array self-refresh (PASR), power down, and LPDDR2 deep power down

SDRAM Controller Subsystem Block Diagram

The SDRAM controller subsystem connects to the MPU subsystem, the main switch of the L3 interconnect, and the FPGA fabric. The memory interface consists of the SDRAM controller, the physical layer (PHY), control and status registers (CSRs), and their associated interfaces.

The following figure shows a high-level block diagram of the SDRAM controller subsystem.

Figure 4-1: SDRAAM Controller Subsystem High-Level Block Diagram



SDRAM Controller

The SDRAM controller provides high performance data access and run-time programmability. The controller reorders data to reduce row conflicts and bus turn-around time by grouping read and write transactions together, allowing for efficient traffic patterns and reduced latency.

The SDRAM controller consists of a multiport front end (MPFE) and a single-port controller. The MPFE provides multiple independent interfaces to the single-port controller. The single-port controller communicates with and manages each external memory device.

The MPFE FPGA-to-HPS SDRAM interface port has an asynchronous FIFO buffer followed by a synchronous FIFO buffer. Both the asynchronous and synchronous FIFO buffers have a read and write data FIFO depth of 8, and a command FIFO depth of 4. The MPU sub-system 64-bit AXI and L3 interconnect 32-bit AXI have asynchronous FIFO buffers with read and write data FIFO depth of 8, and command FIFO depth of 4.

For more information, refer to *Memory Controller Architecture*.

DDR PHY

The DDR PHY provides a physical layer interface between the memory controller and memory devices, which performs read and write memory operations. The DDR PHY has dataflow components, control components, and calibration logic that handle the calibration for the SDRAM interface timing.

Related Information

Memory Controller Architecture on page 4-4

SDRAM Controller Subsystem Interfaces

The following sections describe the SDRAM controller subsystem interfaces.

MPU Subsystem Interface

The SDRAM controller is connected to the MPU subsystem with a dedicated 64-bit AXI interface, operating on the `mpu_l2_ram_clk` clock domain.

L3 Interconnect Interface

The SDRAM controller is connected to the L3 interconnect with a dedicated 32-bit AXI interface, operating on the `l3_main_clk` clock domain.

CSR Interface

The CSR interface is connected the level 4 (L4) bus and operates on the `l4_sp_clk` clock domain. The MPU subsystem uses the CSR interface to configure the controller and PHY, for example, setting the memory timing parameter values or placing the memory to a low power state. The CSR interface also provides access to the status registers in the controller and PHY.

FPGA-to-HPS SDRAM Interface

The FPGA-to-HPS SDRAM interface provides masters implemented in the FPGA fabric access to the SDRAM controller subsystem in the HPS. The interface has three ports types that are used to construct the following AXI or Avalon-MM interfaces:

- Command ports—issue read and write commands, and for receive write acknowledge responses
- 64-bit read data ports—receive data returned from a memory read
- 64-bit write data ports—transmit write data

The FPGA-to-HPS SDRAM interface supports six command ports, allowing up to six Avalon-MM interfaces or three AXI interfaces. Each command port can be used to implement either a read or write command port for AXI, or be used as part of an Avalon-MM interface. The AXI and Avalon-MM interfaces can be configured to support 32-, 64-, 128-, and 256-bit data.

The following table lists the FPGA-to-HPS SDRAM controller interface ports connected to the FPGA.

Table 4-1: FPGA-to-HPS SDRAM Controller Port Types

Port Type	Number
Command	6
64-bit read data	4
64-bit write data	4

The FPGA-to-HPS SDRAM controller interface can be configured with the following characteristics:

- Avalon-MM interfaces and AXI interfaces can be mixed and matched as required by the fabric logic, within the bounds of the number of ports provided to the fabric.
- Each Avalon-MM or AXI interface of the FPGA-to-HPS SDRAM interface operates on an independent clock domain.
- The FPGA-to-HPS SDRAM interfaces are configured during FPGA configuration.

The following table shows the number of ports needed to configure different bus protocols, based on type and data width.

Table 4-2: FPGA-to-HPS SDRAM Port Utilization

Bus Protocol	Command	Read Data	Write Data
32- or 64-bit AXI	2 (1)	1	1
128-bit AXI	2 (1)	2 (2)	2 (2)
256-bit AXI	2 (1)	4 (2)	4 (2)
32- or 64-bit Avalon-MM	1	1	1
128-bit Avalon-MM	1	2	2
256-bit Avalon-MM	1	4	4
32- or 64-bit Avalon-MM write-only	1	0	1
128-bit Avalon-MM write-only	1	0	2
256-bit Avalon-MM write-only	1	0	4
32- or 64-bit Avalon-MM read-only	1	1	0
128-bit Avalon-MM read-only	1	2	0
256-bit Avalon-MM read-only	1	4	0

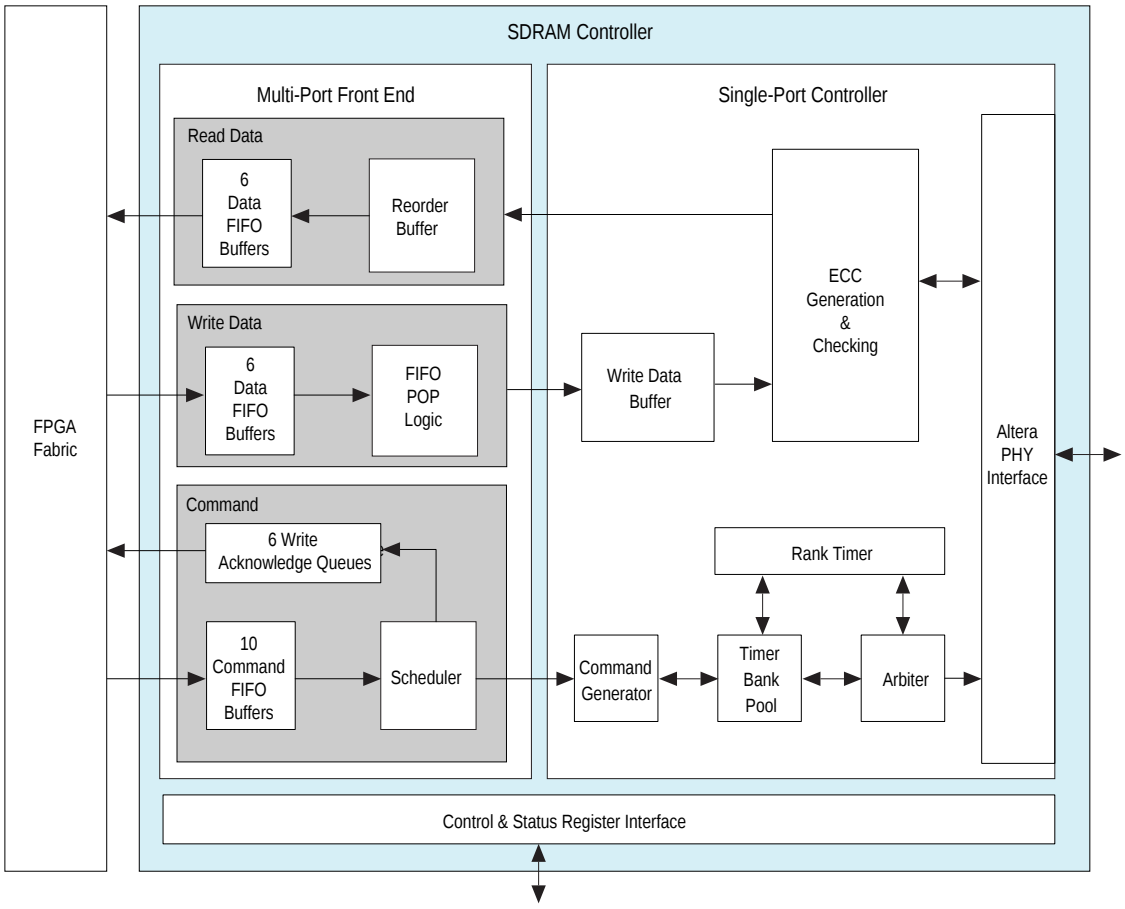
Notes to Table:

1. Because the AXI protocol allows simultaneous read and write commands to be issued, two SDRAM control ports are required to form an AXI interface.
2. Because the native size of the data ports is 64 bits, extra read and write ports are required to form an AXI interface.

Memory Controller Architecture

The SDRAM controller consists of an MPFE, a single-port controller, and an interface to the CSRs. The following figure shows a block diagram of the SDRAM controller portion of the SDRAM controller subsystem.

Figure 4-2: SDRAM Controller Block Diagram



Multi-Port Front End

The Multi-Port Front End (MPFE) is responsible for scheduling pending transactions from the configured interfaces and sending the scheduled memory transactions to the single-port controller. The MPFE handles all functions related to individual ports.

The MPFE consists of three primary sub-blocks, which are described below.

Command Block

The command block accepts read and write transactions from the FPGA fabric and the HPS. When the command FIFO buffer is full, the command block applies backpressure by deasserting the ready signal. For each pending transaction, the command block calculates the next SDRAM burst needed to progress on that transaction. The command block schedules pending SDRAM burst commands based on the user-supplied configuration, available write data, and unallocated read data space.

Write Data Block

The write data block transmits data to the single-port controller. The write data block maintains write data FIFO buffers and clock boundary crossing for the write data. The write data block informs the command block of the amount of pending write data for each transaction so that the command block can calculate eligibility for the next SDRAM write burst.

Read Data Block

The read data block receives data from the single-port controller. Depending on the port state, the read data block either buffers the data in its internal buffer or passes the data straight to the clock boundary crossing FIFO buffer. The read data block reorders out-of-order data for Avalon-MM ports.

In order to prevent the read FIFO buffer from overflowing, the read data block informs the command block of the available buffer area so the command block can pace read transaction dispatch.

SinglePort Controller

The single-port logic is responsible for following actions:

- Queuing the pending SDRAM bursts
- Choosing the most efficient burst to send next
- Keeping the SDRAM pipeline full
- Ensuring all SDRAM timing parameters are met

Transactions passed to the single-port logic for a single page in SDRAM are guaranteed to be executed in order, but transactions can be reordered between pages. Each SDRAM burst read or write is converted to the appropriate Altera PHY interface (AFI) command to open a bank on the correct row for the transaction (if required), execute the read or write command, and precharge the bank (if required).

The single-port logic implements command reordering (looking ahead at the command sequence to see which banks can be put into the correct state to allow a read or write command to be executed) and data reordering (allowing data transactions to be dispatched even if the data transactions are executed in an order different than they were received from the multiport logic).

Command Generator

The command generator accepts commands from the MPFE and from the internal ECC logic, and provides those commands to the timer bank pool.

Timer Bank Pool

The timer bank pool is a parallel queue that operates with the arbiter to enable data reordering. The timer bank pool tracks incoming requests, ensures that all timing requirements are met, and, on receiving write-data-ready notifications from the write data buffer, passes the requests to the arbiter.

Arbiter

The arbiter determines the order in which requests are passed to the memory device. When the arbiter receives a single request, that request is passed immediately. When multiple requests are received, the arbiter uses arbitration rules to determine the order to pass requests to the memory device.

Rank Timer

The rank timer performs the following functions:

- Maintains rank-specific timing information
- Ensures that only four activates occur within a specified timing window
- Manages the read-to-write and write-to-read bus turnaround time
- Manages the time-to-activate delay between different banks

Write Data Buffer

The write data buffer receives write data from the MPFE and passes the data to the PHY, on approval of the write request.

ECC Block

The ECC block consists of an encoder and a decoder-corrector, which can detect and correct single-bit errors, and detect double-bit errors. The ECC block can correct single- bit errors and detect double-bit errors resulting from noise or other impairments during data transmission.

AFI Interface

The AFI interface provides communication between the controller and the PHY.

CSR Interface

The CSR interface is accessible from the L4 bus. The interface allows code executing in the HPS MPU and FPGA fabric to configure and monitor the SDRAM controller.

Functional Description of the SDRAM Controller Subsystem

This section provides a functional description of the SDRAM controller subsystem.

MPFE Operation Ordering

Requests to the same SDRAM page arriving at a given port are executed in the order in which they are received. Requests arriving at different ports have no guaranteed order of service, except when a first transaction has completed before the second arrives.

Operation ordering is defined and enforced within a port, but not between ports. All transactions received on a single port for overlapping addresses execute in order. Transactions received on different ports have no guaranteed order unless the second transaction is presented after the first has completed.

Avalon-MM does not support write acknowledgement. When a port is configured to support Avalon-MM, you should read from the location that was previously written to ensure that the write operation has completed. When a port is configured to support AXI, the master accessing the port can safely issue a read operation to the same address as a write operation as soon as the write has been acknowledged. To keep write latency low, writes are acknowledged as soon as the transaction order is guaranteed—meaning that any operations received on any port to the same address as the write operation are executed after the write operation.

To ensure that the overall latency of traffic is as low as possible, the single port logic can return read data out of order to the multi-port logic which will reorder it when transactions return out of order. A large percentage of traffic reordering will be between ports and transactions only are ordered within a port. For traffic which is reordered between ports but not within a port, no reordering needs to be done. Eliminating unnecessary reordering reduces average latency.

MPFE Multiport Scheduling

Multiport scheduling is governed by two factors, the absolute priority of a request and the weighting of a port.

The evaluation of absolute priority ensures that ports carrying higher-priority traffic are served ahead of ports carrying lower-priority traffic. The scheduler recognizes eight priority levels (0-7), with higher values representing higher priorities. For example, any transaction with priority seven is scheduled before transactions of priority six or lower.

When ports carry traffic of the same absolute priority, relative priority is determined based on port weighting. Port weighting is a five-bit value (0-31), and is determined by a deficit-weighted round robin (DWRR) algorithm, which corrects for past over-servicing or under-servicing of a port. Each port has an associated weight which is updated every cycle, with a user-configured weight added to it and the amount of traffic served subtracted from it. The port with the highest weighting is considered the most eligible.

To ensure that high-priority traffic is served quickly and that long and short bursts are effectively interleaved between ports, incoming transactions longer than a single SDRAM burst are scheduled as a series of SDRAM bursts, with each burst arbitrated separately.

To ensure that lower priority ports do not build up large running weights while higher priority ports monopolize bandwidth, the controller's DWRR weights are updated only when a port matches the scheduled priority. Therefore, if three ports are being accessed, two being priority seven and one being priority four, the weights for both ports at priority seven are updated but the port with priority four remains unchanged.

Multiport scheduling is performed between all of the ports connected to the FPGA fabric and internally in the HPS to determine which transaction is serviced next. Arbitration is performed on a SDRAM burst basis to ensure that a long transaction does not lock other transactions or cause latency to significantly increase for high-priority ports.

Arbitration supports both absolute and relative priority. Absolute priority is intended for applications where one master should always get priority above or below others. Relative priority is supported through a programmable weight field which controls scheduling between ports at the same priority.

The scheduler is work-conserving. Write operations can only be scheduled when enough data for the SDRAM burst has been received. Read operations can only be scheduled when sufficient internal memory is free and the port is not occupying too much of the read buffer.

The multiport scheduling configuration can be updated while traffic is flowing. Both priority and weight for a port can be updated without interrupting traffic on a port. Updates are used in scheduling decisions within 10 memory clock cycles of being updated, so priority can be updated frequently if needed.

Read Data Handling

The MPFE contains a read buffer shared by all ports. If a port is capable of receiving returned data then the read buffer is bypassed. If the size of a read transaction is smaller than twice the memory interface width, the buffer RAM cannot be bypassed.

MPFE SDRAM Burst Scheduling

SDRAM burst scheduling recognizes addresses that access the same row/bank combination, known as open page accesses. Operations to a page are served in the order in which they are received by the single-port controller. Selection of SDRAM operations is a two-stage process. First, each pending transaction must wait for its timers to be eligible for execution. Next, the transaction arbitrates against other transactions that are also eligible for execution.

The following rules govern transaction arbitration:

- High-priority operations take precedence over lower-priority operations
- If multiple operations are in arbitration, read operations have precedence over write operations
- If multiple operations still exist, the oldest is served first

A high-priority transaction in the SDRAM burst scheduler wins arbitration for that bank immediately if the bank is idle and the high-priority transaction's chip select, row, or column fields of the address do not match an address already in the single-port controller. If the bank is not idle, other operations to that bank yield until the high-priority operation is finished. If the chip select, row, and column fields match an earlier transaction, the high-priority transaction yields until the earlier transaction is completed.

Clocking

The FPGA fabric ports of the MPFE can be clocked at different frequencies. Synchronization is maintained by clock-domain crossing logic in the MPFE. Command ports can operate on different clock domains, but the data ports associated with a given command port must be attached to the same clock as that command port. For example, a command port paired with a read and write port to form an Avalon-MM interface must operate at the same clock frequency as the data ports associated with it.

SinglePort SDRAM Controller Operational Behavior

This section describes the operational behavior of the single-port controller.

SDRAM Interface

The SDRAM interface is up to 40 bits wide and can accommodate 8-bit, 16-bit, 16-bit plus ECC, 32-bit, or 32-bit plus ECC configurations. The SDRAM interface supports LPDDR2, DDR2, and DDR3 memory protocols.

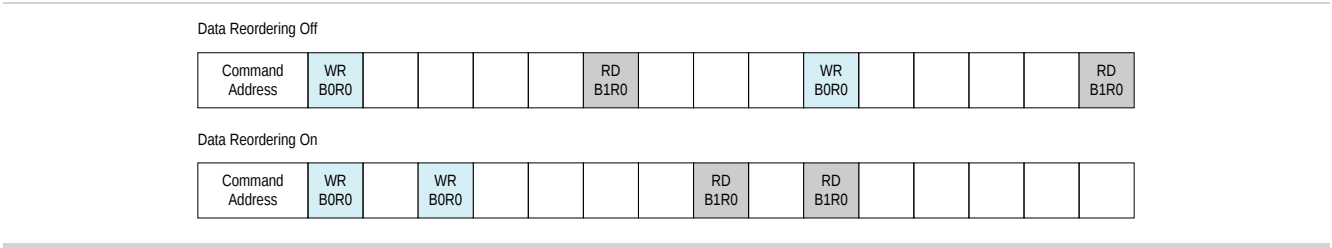
Command and Data Reordering

The heart of the SDRAM controller is a command and data reordering engine. Command reordering allows banks for future transactions to be opened before the current transaction finishes.

Data reordering allows transactions to be serviced in a different order than they were received when that new order allows for improved utilization of the SDRAM bandwidth. Operations to the same bank and row are performed in order to ensure that operations which impact the same address preserve the data integrity.

The following figure shows the relative timing for a write/read/write/read command sequence performed in order and then the same command sequence performed with data reordering. Data reordering allows the write and read operations to occur in bursts, without bus turnaround timing delay or bank reassignment.

Figure 4-3: Data Reordering Effect



The SDRAM controller schedules among all pending row and column commands every clock cycle.

Bank Policy

The bank policy of the SDRAM controller allows users to request that a transaction's bank remain open after an operation has finished so that future accesses do not delay in activating the same bank and row combination. The controller supports only eight simultaneously-opened banks, so an open bank might get closed if the bank resource is needed for other operations.

Open bank resources are allocated dynamically as SDRAM burst transactions are scheduled. Bank allocation is requested automatically by the controller when an incoming transaction spans multiple SDRAM bursts or by the extended command interface. When a bank must be reallocated, the least-recently-used open bank is used as the replacement.

If the controller determines that the next pending command will cause the bank request to not be honored, the bank might be held open or closed depending on the pending operation. A request to close a bank with a pending operation in the timer bank pool to the same row address causes the bank to remain open. A request to leave a bank open with a pending command to the same bank but a different row address causes a precharge operation to occur.

Write Combining

The SDRAM controller combines write operations from successive bursts on a port where the starting address of the second burst is one greater than the ending address of the first burst and the resulting burst length does not overflow the 11-bit burst-length counters.

Write combining does not occur if the previous bus command has finished execution before the new command has been received.

Burst Length Support

The controller supports burst lengths of 2, 4, 8, and 16, and data widths of 8, 16, and 32 bits for non-ECC operation, and widths of 24 and 40 operations with ECC enabled. The following table shows the type of SDRAM for each burst length.

Table 4-3: SDRAM Burst Lengths

Burst Length	SDRAM
4	LPDDR2, DDR2
8	DDR2, DDR3, LPDDR2
16	LPDDR2

Width Matching

The SDRAM controller automatically performs data width conversion.

ECC

The single-port controller supports memory ECC calculated by the controller. The controller ECC employs standard Hamming logic to detect and correct single-bit errors and detect double-bit errors. The controller ECC is available for 16-bit and 32-bit widths, each requiring an additional 8 bits of memory, resulting in an actual memory width of 24-bits and 40-bits, respectively.

Controller ECC provides the following features:

- Byte writes—The memory controller performs a read-modify-write operation to ensure that the ECC data remains valid when a subset of the bits of a word is being written. If an entire word is being written (but less than a full burst) and the DM pins are connected, no read is necessary and only that word is updated. If controller ECC is disabled, byte-writes have no performance impact.
- ECC write backs—When a read operation detects a correctable error, the memory location is scheduled for a read-modify-write operation to correct the single-bit error. ECC write backs are enabled and disabled through the `cfg_enable_ecc_code_overwrites` field in the `ctrlcfg` register.
- Notification of ECC errors—The memory controller provides interrupts for single-bit and double-bit errors. The status of interrupts and errors are recorded in status registers, as follows:
  - The `dramsts` register records interrupt status.
  - The `dramintr` register records interrupt masks.
  - The `sbecount` register records the single-bit error count.
  - The `dbecount` register records the double-bit error count.
  - The `erraddr` register records the address of the most recent error.

Byte Writes

Byte writes with ECC enabled are executed as a read-modify-write. Typical operations only use a single entry in the timer bank pool. Controller ECC enabled sub-word writes use two entries. The first operation is a read and the second operation is a write. These two operations are transferred to the timer bank pool with an address dependency so that the write cannot be performed until the read data has returned. This approach ensures that any subsequent operations to the same address (from the same port) are executed after the write operation, because they are ordered on the row list after the write operation.

If an entire word is being written (but less than a full burst), then no read is necessary and only that word is updated.

ECC Write Backs

If the controller ECC is enabled and a read operation results in a correctable ECC error, the controller corrects the location in memory, if write backs are enabled. The correction results in scheduling a new read-modify-write. A new read is performed at the location to ensure that a write operation modifying the location is not overwritten. The actual ECC correction operation is performed as a read-modify-write operation.

User Notification of ECC Errors

The following methods notify you of an ECC error:

For the MPU subsystem, an interrupt signal provides notification and the ECC error information is stored in the status registers.

For more information, refer to the *Cortex A9 Microprocessor Unit Subsystem* chapter in your device handbook.

Related Information

- **Cortex-A9 Microprocessor Unit Subsystem (Arria V)**
- **Cortex-A9 Microprocessor Unit Subsystem (Cyclone V)**

Interleaving Options

The controller supports the following address-interleaving options:

- Noninterleaved
- Bank interleave without chip select interleave
- Bank interleave with chip select interleave

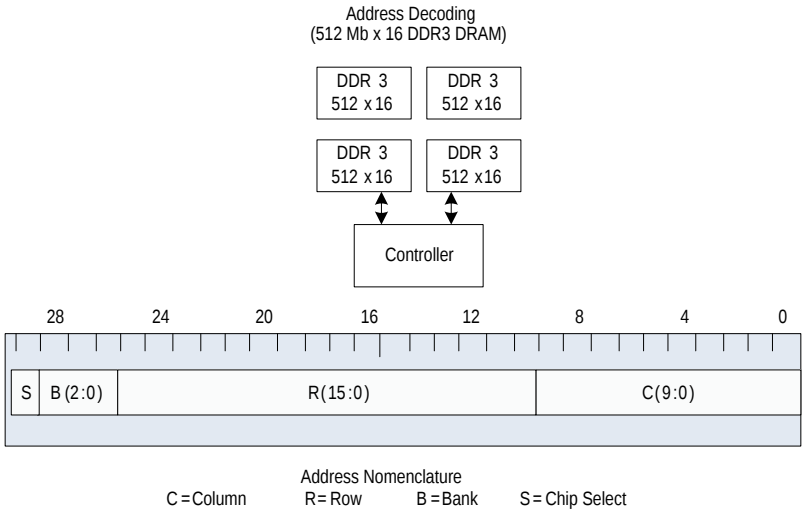
All of the interleaving examples use 512 megabits (Mb) x 16 DDR3 chips and are documented as byte addresses. For RAMs with smaller address fields, the order of the fields stays the same but the widths may change.

Noninterleaved

RAM mapping is noninterleaved.

The following figure shows noninterleaved address decoding.

Figure 4-4: Noninterleaved Address Decoding

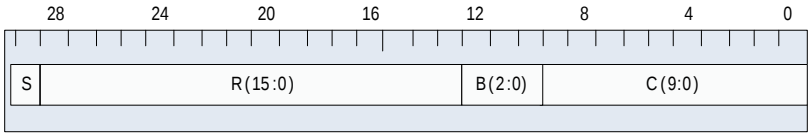


Bank Interleave Without Chip Select Interleave

Bank interleave without chip select interleave swaps row and bank from the noninterleaved address mapping. This interleaving allows smaller data structures to spread across all banks in a chip.

The following figure shows bank interleave without chip select interleave address decoding.

Figure 4-5: Bank Interleave Without Chip Select Interleave Address Decoding

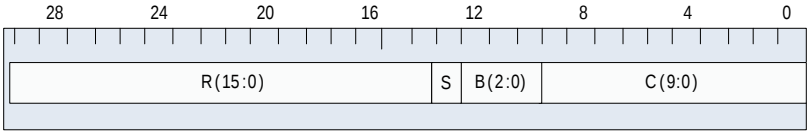


Bank Interleave with Chip Select Interleave

Bank interleave with chip select interleave moves the row address to the top, followed by chip select, then bank, and finally column address. This interleaving allows smaller data structures to spread across multiple banks and chips (giving access to 16 total banks for multithreaded access to blocks of memory). Memory timing is degraded when switching between chips.

The following figure shows bank interleave with chip select interleave address decoding.

Figure 4-6: Bank Interleave With Chip Select Interleave Address Decoding



AXI-Exclusive Support

The single-port controller supports AXI-exclusive operations. The controller implements a table shared across all masters, which can store up to 16 pending writes. Table entries are allocated on an exclusive read and table entries are deallocated on a successful write to the same address by any master.

Any exclusive write operation that is not present in the table returns an exclusive fail as acknowledgement to the operation. If the table is full when the exclusive read is performed, the table replaces a random entry.

Note: When using AXI-exclusive operations, accessing the same location from Avalon-MM interfaces can result in unpredictable results.

Memory Protection

The single-port controller has address protection to allow the software to configure basic protection of memory from all masters in the system. If the system has been designed exclusively with AXI masters, TrustZone® is supported. Ports that use Avalon-MM can be configured for port level protection.

For information about TrustZone®, refer to the ARM website (www.arm.com).

Memory protection is based on physical addresses in memory. You can set rules to allow or disallow accesses to a range of memory, or to enable only secure accesses to a range of memory (or a combination of the two).

Secure and non-secure regions are specified by rules containing a starting address and ending address with 1 MB boundaries for both addresses. You can override the port defaults and allow or disallow all transactions.

The memory protection table, which is an internal table addressed through the CSR interface, contains rules to permit or deny memory access. You can configure up to a maximum of twenty rules to control memory access. The following table lists the fields that you can specify for each rule.

Table 4-4: Fields for Rules in Memory Protection Table

Field	Width	Description
Valid	1	Set to 1 to activate the rule. Set to 0 to deactivate the rule.
Port Mask *(1)*	10	Specifies the set of ports to which the rule applies, with one bit representing each port, as follows: bits 0 to 5 correspond to FPGA fabric ports 0 to 5, bit 6 corresponds to AXI L3 switch read, bit 7 is the CPU read, bit 8 is L3 switch write, and bit 9 is the CPU write.
TID_low *(1)*	12	Low transfer ID of the rules to which this rule applies. Incoming transactions match if they are greater than or equal to this value. Ports with smaller TIDs have the TID shifted to the lower bits and zero padded at the top.
TID_high *(1)*	12	High transfer ID of the rules to which this rule applies. Incoming transactions match if they are less than or equal to this value.

Field	Width	Description
Address_low	12	Points to a 1MB block and is the lower address. Incoming addresses match if they are greater than or equal to this value.
Address_high	12	Upper limit of address. Incoming addresses match if they are less than or equal to this value.
Protection	2	A value of 00 indicates that the protection bit is not set; a value of 01 sets the protection bit. Systems that do not set AXI protection to a known value should program this for either protection value.
Fail/allow	1	Set this value to 1 to force the operation to fail or succeed.

Note to Table:

1. Although TID and Port Mask could be redundant, including both in the table allows possible compression of rules. If masters connected to a port do not have contiguous TIDs, a port-based rule might be more efficient than a TID-based rule, in terms of the number of rules needed.

A port has a default access status of either allow or fail, nd rules with the opposite allow/fail value can override the default. The system evaluates each transaction against every rule in the memory protection table. A transaction received on a port which by default allows access, would fail only if a rule with the fail bit matches the transaction. Conversely, a port which by default prevents access, would allow access only if a rule allows that transaction to pass.

Exclusive transactions are security checked on the read operation only. A write operation can occur only if a valid read is marked in the internal exclusive table. Consequently, a master performing an exclusive read followed by a write, can write to memory only if the exclusive read was successful.

Related Information

www.arm.com

Example of Configuration for TrustZone

For a TrustZone configuration, memory is divided into a range of memory accessible by secure masters and a range of memory accessible by nonsecure masters. The two memory address ranges may have a range of memory that overlaps.

This example implements the following memory configuration:

- 2 GB total RAM size
- 0—512 MB dedicated secure area
- 513—576 MB shared area
- 577—2048 MB dedicated nonsecure area

In this example, each port is configured by default to disallow all accesses. The following table shows the two rules programmed into the memory protection table.

Table 4-5: Rules in Memory Protection Table for Example Configuration

Rule #	Port Mask	TID Low	TID High	Address Low	Address High	Prot	Fail/Allow
1	0'b1111111111	0	4095	0	576	b01	allow
2	0'b1111111111	0	4095	512	2047	b00	allow

The port mask value, TID Low, and TID High, apply to all ports and all transfers within those ports. Each access request is evaluated against the memory protection table, and will fail unless a rule matches allowing a transaction to complete successfully.

The following table shows the result for a sample set of transactions.

Table 4-6: Result for a Sample Set of Transactions

Operation	Source	Address	Prot	Result	Comments
Read	CPU	4096	1	Allow	Matches rule 1.
Write	CPU	536, 870, 912 (512 MB)	1	Allow	Matches rule 1.
Write	L3 attached masters	605, 028, 350 (577 MB)	1	Fail	Does not match rule 1 (out of range of the address field), does not match rule 2 (protection bit incorrect).
Read	L3 attached masters	4096	0	Fail	Does not match rule 1 (prot value wrong), does not match rule 2 (not in address range) .
Write	CPU	536, 870, 912 (512 MB)	0	Allow	Matches rule 2.
Write	L3 attached masters	605, 028, 350 (577 MB)	0	Allow	Matches rule 2.

If you did not want any overlap between the memory blocks, you could specify the address ranges in the two rules of the Memory Protection Table to be mutually exclusive. Depending on your desired TrustZone configuration, you can add rules to the memory protection table to create multiple blocks of protected or unprotected space.

SDRAM Power Management

The SDRAM controller subsystem supports the following power saving features in the SDRAM:

- Partial array self-refresh (PASR)
- Power down
- Deep power down for LPDDR2

Power-saving mode initiates either due to a user command or from inactivity.

Power-down mode is initiated by writing to the appropriate control register. It forces the SDRAM burst-scheduling bank-management logic to close all banks and issue the power down command. You can program the controller to enable power-down when the SDRAM burst-scheduling queue is empty for a specified number of clock cycles. The SDRAM automatically reactivates when an active SDRAM command is received.

Other power-down modes are performed only under user control.

DDR PHY

The DDR PHY connects the memory controller and external memory devices in the speed critical command path.

The DDR PHY implements the following functions:

- Calibration—the DDR PHY supports the JEDEC-specified steps to synchronize the memory timing between the controller and the SDRAM chips. The calibration algorithm is implemented in software.
- Memory device initialization—the DDR PHY performs the mode register write operations to initialize the devices. The DDR PHY handles re-initialization after a deep power down.
- Single-data-rate to double-data-rate conversion.

Clocks

All clocks are assumed to be asynchronous with respect to the `ddr_dqs_clk` memory clock. All transactions are synchronized to memory clock domain.

The following table shows the SDRAM controller subsystem clock domains.

Table 4-7: SDRAM Controller Subsystem Clock Domains

Clock Name	Description
`ddr_dq_clk`	Clock for PHY
`ddr_dqs_clk`	Clock for MPFE, single-port controller, CSR access, and PHY
`ddr_2x_dqs_clk`	Clock for PHY
`l4_sp_clk`	Clock for CSR interface
`mpu_l2_ram_clk`	Clock for MPU interface
`l3_main_clk`	Clock for L3 interface
`f2h_sdram_clk[5:0]`	Six separate clocks used for the FPGA-to-HPS SDRAM ports to the FPGA fabric

In terms of clock relationships, the FPGA fabric connects the appropriate clocks to write data, read data, and command ports for the constructed ports.

For more information, refer to the *Clock Manager* chapter in volume 3 of your device handbook.

Related Information

- **Clock Manager (Arria V)**

- Clock Manager (Cyclone V)

Resets

The SDRAM controller subsystem supports a full reset (cold reset) and a warm reset, which may or may not preserve the contents of memory.

To preserve memory contents, the reset manager can request that the single-port controller place the SDRAM in self-refresh mode prior to issuing the warm reset. If memory contents are preserved, the PHY and the memory timing logic is not reset, but the rest of the controller is reset.

For more information, refer to the *Reset Manager* chapter in volume 3 of your device handbook.

Related Information

- Reset Manager (Arria V)
- Reset Manager (Cyclone V)

Initialization

The SDRAM controller subsystem has control and status registers (CSRs) which control the operation of the controller including DRAM type, DRAM timing parameters and relative port priorities. It also has a small set of bits which depend on the FPGA fabric to configure ports between the memory controller and the FPGA fabric; these bits are set for you when you configure your implementation using the HPS GUI in Qsys.

The CSRs are configured using a dedicated slave interface, which provides accesses to registers. This region controls all SDRAM operation, MPFE scheduler configuration, and PHY calibration.

The FPGA fabric interface configuration is programmed into the FPGA fabric and the values of these register bits can be read by software. The ports can be configured without software developers needing to know how the FPGA-to-HPS SDRAM interface has been configured.

Protocol Details

The following topics summarize signals for the Avalon-MM Bidirectional port, Avalon-MM Write Port, Avalon-MM Read Port, and AXI port.

Avalon-MM Bidirectional Port

The Avalon-MM bidirectional ports are standard Avalon-MM ports used to dispatch read and write operations. Each configured Avalon-MM bidirectional port consists of the signals listed in the following table.

Table 4-8: Avalon-MM Bidirectional Port Signals

Name	Bits	Direction	Function
clk	1	In	Clock for the Avalon-MM interface
read	1	In	Indicates read transaction
write	1	In	Indicates write transaction
address	32	In	Address of the transaction
readdata	32, 64, 128, or 256	Out	Read data return
readdatavalid	1	Out	Valid cycle flag for read data return
writedata	32, 64, 128, or 256	In	Write data for a transaction
byteenable	4 (32-bit data), 8(64-bit data), 16(128-bit data), 32(256-bit data)	In	Byte enables for each write byte
waitrequest	1	Out	Indicates need for additional cycles to complete a transaction
burstcount	11	In	Transaction burst length

The read and write interfaces are configured to the same size. The byte-enable size scales with the data bus size.

For information about the Avalon-MM protocol, refer to the *Avalon Interface Specifications*.

Related Information

Avalon Interface Specifications

Avalon-MM Write Port

The Avalon-MM write ports are standard Avalon-MM ports used only to dispatch write operations. Each configured Avalon-MM write port consists of the signals listed in the following table.

Table 4-9: Avalon-MM Write Port Signals

Name	Bits	Direction	Function
reset	1	In	Reset
clk	1	In	Clock
write	1	In	Indicates write transaction
address	32	In	Address of the transaction
writedata	32, 64, 128, or 256	In	Write data for a transaction
byteenable	4 (32-bit data), 8(64-bit data), 16(128-bit data), 32(256-bit data)	In	Byte enables for each write byte
waitrequest	1	Out	Indicates need for additional cycles to complete a transaction
burstcount	11	In	Transaction burst length

For information about the Avalon-MM protocol, refer to the *Avalon Interface Specifications*.

Related Information

Avalon Interface Specifications

Avalon-MM Read Port

The Avalon-MM read ports are standard Avalon-MM ports used only to dispatch read operations. Each configured Avalon-MM read port consists of the signals listed in the following table.

Table 4-10: Avalon-MM Read Port Signals

Name	Bits	Direction	Function
reset	1	In	Reset
clk	1	In	Clock
read	1	In	Indicates read transaction
address	32	In	Address of the transaction
readdata	32, 64, 128, or 256	Out	Read data return
readdatavalid	1	Out	Flags valid cycles for read data return
waitrequest	1	Out	Indicates the need for additional cycles to complete a transaction. Needed for read operations when delay is needed to accept the read command.
burstcount	11	In	Transaction burst length

For information about the Avalon-MM protocol, refer to the *Avalon Interface Specifications*.

Related Information

Avalon Interface Specifications

AXI Port

The AXI port uses an AXI-3 interface. Each configured AXI port consists of the signals listed in the following table. Each AXI interface signal is independent of the other interfaces for all signals, including clock and reset.

Table 4-11: AXI Port Signals

Name	Bits	Direction	Function
ARESETn	1	In	Reset
ACLK	1	In	Clock

Name	Bits	Direction	Function
Write Address Channel Signals			
AWID	4	In	Write identification tag
AWADDR	32	In	Write address
AWLEN	4	In	Write burst length
AWSIZE	3	In	Width of the transfer size
AWBURST	2	In	Burst type
AWREADY	1	Out	Indicates ready for a write command
AWVALID	1	In	Indicates valid write command.
Write Data Channel Signals			
WID	4	In	Write data transfer ID
WDATA	32, 64, 128 or 256	In	Write data
WSTRB	4, 8, 16, 32	In	Byte-based write data strobe. Each bit width corresponds to 8 bit wide transfer for 32-bit wide to 256-bit wide transfer.
WLAST	1	In	Last transfer in a burst
WVALID	1	In	Indicates write data+strobes are valid
WREADY	1	Out	Indicates ready for write data and strobes

Name	Bits	Direction	Function
Write Response Channel Signals			
BID	4	Out	Write response transfer ID
BRESP	2	Out	Write response status
BVALID	1	Out	Write response valid signal
BREADY	1	In	Write response ready signal
Read Address Channel Signals			
ARID	4	In	Read identification tag
ARADDR	32	In	Read address
ARLEN	4	In	Read burst length
ARSIZE	3	In	Width of the transfer size
ARBURST	2	In	Burst type
ARREADY	1	Out	Indicates ready for a read command
ARVALID	1	In	Indicates valid read command

Name	Bits	Direction	Function
Read Data Channel Signals			
RID	4	Out	Read data transfer ID
RDATA	32, 64, 128 or 256	Out	Read data
RRESP	2	Out	Read response status
RLAST	1	Out	Last transfer in a burs
RVALID	1	Out	Indicates read data is valid
RREADY	1	In	Read data channel ready signal

For information about the AXI-3 interface, refer to the AMBA Open Specifications on the ARM website (www.arm.com).

For information about the AXI interface ports in the high-performance II controller (HPC II), refer to the *Functional Description—HPC II Controller* chapter, in the *External Memory Interface Handbook*.

Related Information

- **www.arm.com**
- **Functional Description—HPC II Controller**

SDRAM Controller Subsystem Programming Model

SDRAM controller configuration occurs through software programming of the configuration registers using the CSR interface.

Initialization

Initialization of the SDRAM controller has two separate regions with different controls.

Timing Parameters

The SDRAM controller supports a complete set of timing parameters, configurable at run time.

SDRAM Controller Address Map and Register Definitions

The address map and register definitions reside in the **hps.html** file that accompanies this handbook volume. Click the link at the bottom of this topic to open the file.

To view the module description and base address, scroll to and click the link for the following module instance:

- **sdr**

To view the register and field descriptions, scroll to and click the register names. The register addresses are offsets relative to the base address of each module instance.

The base addresses of all modules are also listed in the *Introduction to the Hard Processor System* chapter in volume 3 of your device handbook.

Related Information

- **hps.html (Arria V)**
- **hps.html (Cyclone V)**
- **Introduction to the Hard Processor System (Arria V)**
- **Introduction to the Hard Processor System (Cyclone V)**

HPS Memory Interface Architecture

The configuration and initialization of the memory interface by the ARM processor is a significant difference compared to the FPGA memory interfaces, and results in several key differences in the way the HPS memory interface is defined and configured.

Boot-up configuration of the HPS memory interface is handled by the initial software boot code, not by the FPGA programmer, as is the case for the FPGA memory interfaces. The Quartus II software is involved in defining the configuration of I/O ports which is used by the boot-up code, as well as timing analysis of the memory interface. Therefore, the memory interface must be configured with the correct PHY-level timing information. Although configuration of the memory interface in Qsys is still necessary, it is limited to PHY- and board-level settings.

HPS Memory Interface Configuration

To configure the external memory interface components of the HPS, open the HPS interface by selecting the Hard Processor System component in Qsys. Within the HPS interface, select the EMIF tab to open the EMIF parameter editor.

The EMIF parameter editor contains four additional tabs: PHY Settings, Memory Parameters, Memory Timing, and Board Settings. The parameters available on these tabs are similar to those available in the parameter editors for non-SoC device families.

There are significant differences between the EMIF parameter editor for the Hard Processor System and the parameter editors for non-SoC devices, as follows:

- Because the HPS memory controller is not configurable through the Quartus II software, the Controller and Diagnostic tabs, which exist for non-SoC devices, are not present in the EMIF parameter editor for the hard processor system.
- Unlike the protocol-specific parameter editors for non-SoC devices, the EMIF parameter editor for the Hard Processor System supports multiple protocols, therefore there is an SDRAM Protocol parameter, where you can specify your external memory interface protocol.By default, the EMIF parameter editor assumes the DDR3 protocol, and other parameters are automatically populated with DDR3-appropriate values. If you select a protocol other than DDR3, you will have to change other associated parameter values appropriately.
- Unlike the memory interface clocks in the FPGA, the memory interface clocks for the HPS are initialized by the boot-up code using values provided by the configuration process. You may accept the values provided by UniPHY, or you may use your own PLL settings. If you choose to specify your own PLL settings, you must indicate that the clock frequency that UniPHY should use is the requested clock frequency, and not the achieved clock frequency calculated by UniPHY.

Note: The HPS does not support EMIF synthesis generation, compilation, or timing analysis.The HPS hard memory controller cannot be bonded with another hard memory controller on the FPGA portion of the device.

HPS Memory Interface Simulation

The HPS component provides a complete simulation model of the HPS memory interface controller and PHY, providing cycle-level accuracy, comparable to the simulation models for the FPGA memory interface.

The simulation model supports only the skip-cal simulation mode; quick-cal and full-cal are not supported. An example design is not provided, however you can create a test design by adding the traffic generator component to your design using Qsys. Also, the HPS simulation model does not use external memory pins to connect to the DDR memory model; instead, the memory model is incorporated directly into the HPS SDRAM interface simulation modules.

Simulation of the FPGA-to-SDRAM interfaces requires that you first bring the interfaces out of reset, otherwise transactions cannot occur. You should add a stage to your testbench to assert and deassert the H2F reset in the HPS. Appropriate Verilog code is shown below:

```
initial
begin
   // Assert reset
   <base name>.hps.fpga_interfaces.h2f_reset_inst.reset_assert();
   // Delay
   #1
   // Deassert reset
   <base name>.hps.fpga_interfaces.h2f_reset_inst.reset_deassert();
end
```

Generating a Preloader Image for HPS with EMIF

To generate a Preloader image for an HPS-based external memory interface, you must complete the following tasks:

- Create a Qsys project.
- Create a top-level file and add constraints.
- Create a Preloader BSP file.
- Create a Preloader image.

The following topics provide procedures for each of the above tasks.

Creating a Qsys Project in Preparation for Generating a Preloader Image

This topic describes creating a Qsys project in preparation for generating a Preloader image.

1. On the **Tools** menu in the Quartus II software, click **Qsys**.
2. Under **Component library**, expand **Embedded Processor System**, select **Hard Processor System** and click **Add**.
3. Specify parameters for the **FPGA Interfaces**, **Peripheral Pin Multiplexing**, and **HPS Clocks**, based on your design requirements.
4. On the **SDRAM** tab, select the SDRAM protocol for your interface.
5. Populate the necessary parameter fields on the **PHY Settings**, **Memory Parameters**, **Memory Timing**, and **Board Settings** tabs.
6. Add other Qsys components in your Qsys design and make the appropriate bus connections.
7. Save the Qsys project.
8. Click **Generate** on the **Generation** tab, to generate the Qsys design.

Creating a Top-Level File and Adding Constraints

This topic describes adding your Qsys system to your top-level design and adding constraints to your design.

1. Add your Qsys system to your top-level design.
2. Add the Quartus II IP files (**.qip**) generated in step 2, to your Quartus II project.
3. Perform analysis and synthesis on your design.
4. Constrain your EMIF design by running the <*variation_name*>**_p0_pin_assignments.tcl** pin constraints script.
5. Add other necessary constraints—such as timing constraints, location assignments, and pin I/O standard assignments—for your design.
6. Compile your design to generate an SRAM object file (**.sof**) and the hardware handoff files necessary for creating a preloader image.

   Note: You must regenerate the hardware handoff files whenever the HPS configuration changes; for example, due to changes in Peripheral Pin Multiplexing or I/O standard for HPS pins.

Creating a Preloader BSP File

The following steps explain how to create a Qys project in preparation for generating a preloader image for HPS with external memory interface.

1. At a command prompt, navigate to <*SoC EDS installation*>*Quartus II version***embedded**\.
2. Start the embedded command shell, as follows:
   a. On Windows systems, run the batch file <*SoC EDS installation Folder*>**embedded****Embedded_Command_Shell.bat**.

b. On Linux systems, run the shell script <*SoC EDS installation Folder*>**embedded****embedded_command_shell.sh**

3. At the command shell, type `bsp-editor` to launch the Preloader Support Package Generator.
4. To create a new BSP project, select **New BSP** on the **File**menu.
5. Change the **Preloader settings directory** to the handoff files directory, and click **OK**.
6. Set up the **.bsp** file based on your system's requirements.
7. Click **Generate** to create the **.bsp** file.
8. Close the BSP Editor after you generate the **.bsp** file.

Creating a Preloader Image

This topic describes the creation of a Preloader image.

1. At the command shell, change the directory to <*design path*>**software****spl_bsp**.
2. To compile the Preloader sources and generate a Preloader image, type `make all`. The Preloader image (preloader-mkpimage.bin) is created in the <*design path*>**software****spl_bsp** folder after the compliation.

   Note: You must regenerate the Preloader image if any of the source files (**.bsp**, **.c**, or **.h**) files change.

Debugging HPS SDRAM in the Preloader

The following tools are available at the preloader stage, to assist in debugging your design:

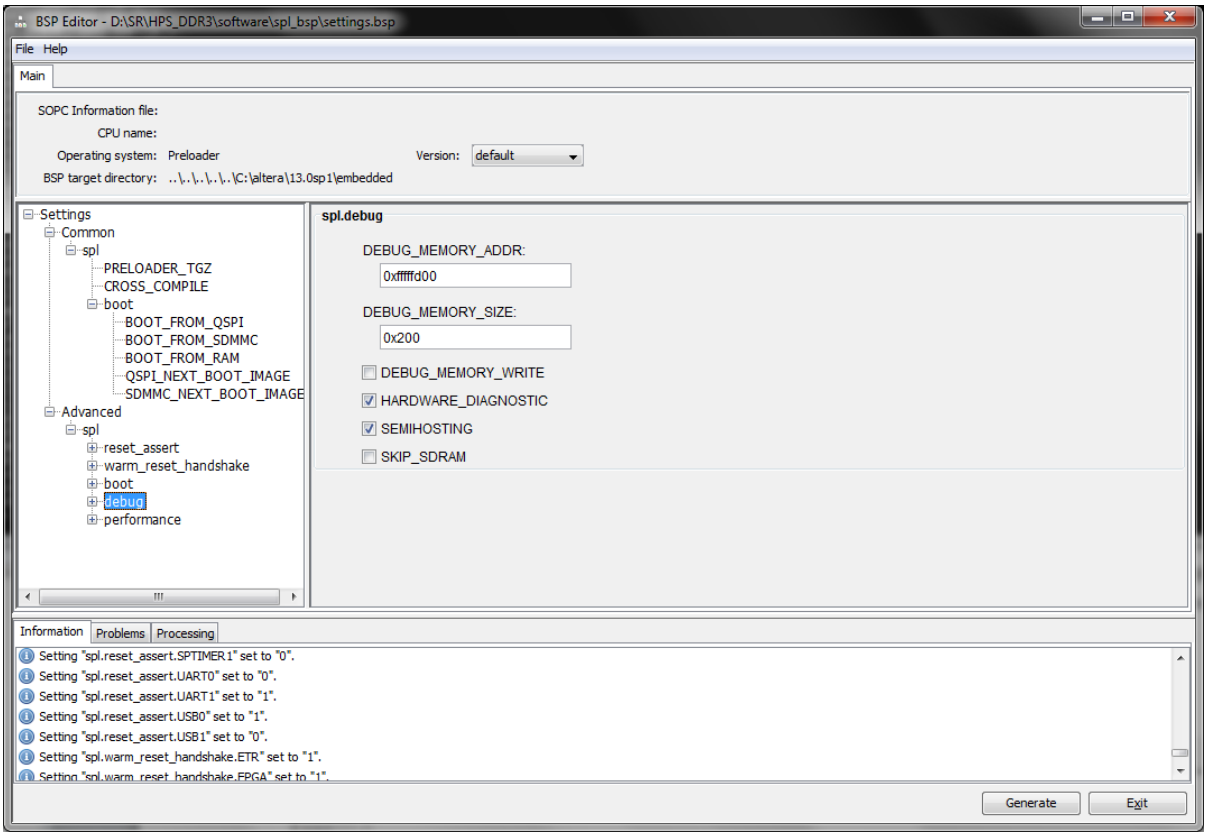
- UART or semihosting printout
- Simple memory test
- Debug report
- Predefined data patterns

The following topics provide procedures for implementing each of the above tools.

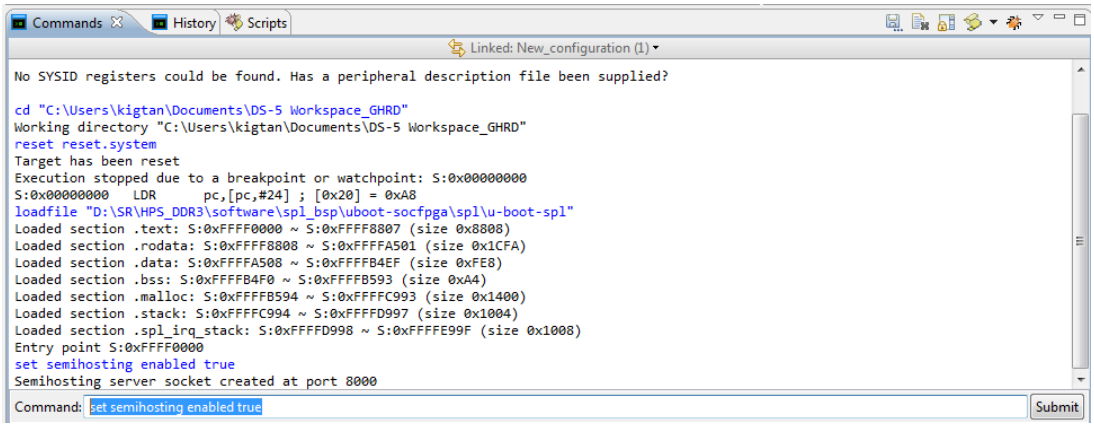
Enabling UART or Semihosting Printout

UART printout is enabled by default. If UART is not available on your system, you can use semihosting together with the debugger tool. To enable semihosting in the Preloader, follow these steps:

1. When you create the **.bsp** file in the BSP Editor, turn on **SEMIHOSTING** in the **spl.debug** window to enable semihosting.



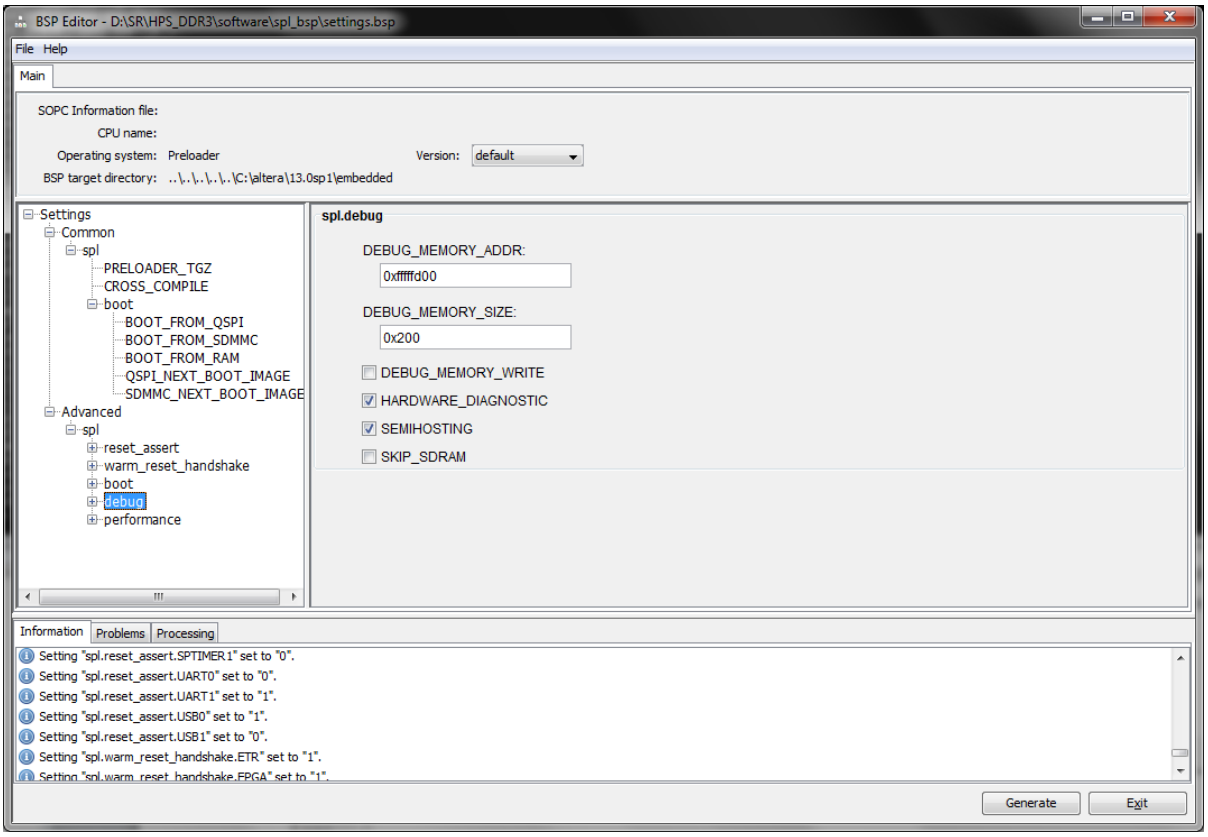
2. You must also enable semihosting in the debugger, by typing `set semihosting enabled true` at the command line in the debugger.



Enabling Simple Memory Test

To enable the simple memory test, follow these steps:

1. When you create the **.bsp** file in the BSP Editor, turn on **HARDWARE_DIAGNOSTIC** in the **spl.debug** window to enable the simple memory test.



2. The simple memory test assumes SDRAM with a memory size of 1 GB. If your board contains a different SDRAM memory size, open the file *<design folder>***spl_bsp\uboot-socfpga\include\configs\socfpga_cyclone5.h** in a text editor, and change the PHYS_SDRAM_1_SIZE parameter at line 292 to specify your actual memory size in bytes.

socfpga_cyclone5.h

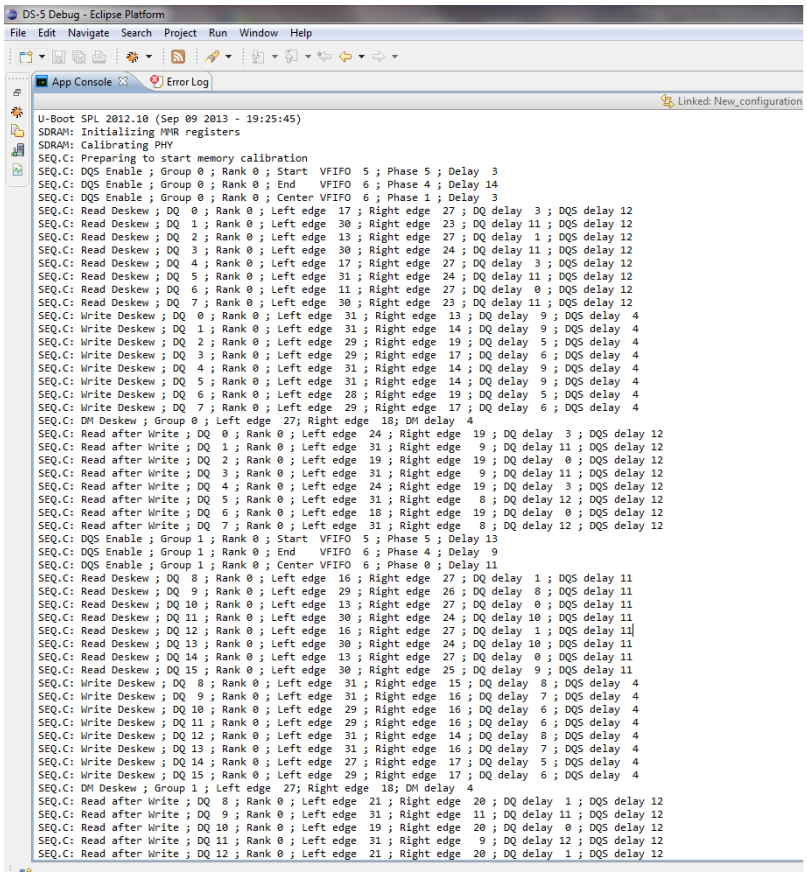
```
278   * Hardware drivers
279   */
280
281  /*
282   * SDRAM Memory Map
283   */
284  /* We have 1 bank of DRAM */
285  #define CONFIG_NR_DRAM_BANKS        1
286  /* SDRAM Bank #1 */
287  #define CONFIG_SYS_SDRAM_BASE       0x00000000
288  /* SDRAM memory size */
289  #ifdef CONFIG_SOCFPGA_VIRTUAL_TARGET
290  #define PHYS_SDRAM_1_SIZE       0x80000000
291  #else
292  #define PHYS_SDRAM_1_SIZE       0x40000000
293  #endif
294  /* SDRAM Bank #1 base address */
295  #define PHYS_SDRAM_1            CONFIG_SYS_SDRAM_BASE
296  /* memtest setup */
297  /* Begin and end addresses of the area used by the simple memory test.c */
298  #define CONFIG_SYS_MEMTEST_START    0x00000000
299  #define CONFIG_SYS_MEMTEST_END      PHYS_SDRAM_1_SIZE
300
```

Enabling the Debug Report

You can enable the SDRAM calibration sequencer to produce a debug report on the UART printout or semihosting output. To enable the debug report, follow these steps:

1. After you have enabled the UART or semihosting, open the file <*project directory*>**hps_isw_handoff\sequencer_defines.h**in a text editor.
2. Locate the line `#define RUNTIME_CAL_REPORT 0` and change it to `#define RUNTIME_CAL_REPORT 1`.

Figure 4-7: Semihosting Printout With Debug Support Enbled



Analysis of Debug Report

The following analysis will help you interpret the debug report.

- The Read Deskew and Write Deskew results shown in the debug report are before calibration. (Before calibration results are actually from the window seen *during* calibration, and are most useful for debugging.)
- For each DQ group, the Write Deskew, Read Deskew, DM Deskew, and Read after Write results map to the before-calibration margins reported in the EMIF Debug Toolkit.

  Note: The Write Deskew, Read Deskew, DM Deskew, and Read after Write results are reported in delay steps (nominally 25ps, in Arria V and Cyclone V devices), not in picoseconds.

  For more information about calibration, refer to *Calibration Stages* in the *Functional Description—UniPHY* chapter, in the *External Memory Interface Handbook*.

- DQS Enable calibration is reported as a VFIFO setting (in one clock period steps), a phase tap (in one-eighth clock period steps), and a delay chain step (in 25ps steps).

  ```
  SEQ.C: DQS Enable ; Group 0 ; Rank 0 ; Start  VFIFO  5 ; Phase 6 ;
  Delay  4
  SEQ.C: DQS Enable ; Group 0 ; Rank 0 ; End    VFIFO  6 ; Phase 5 ;
  Delay  9
  SEQ.C: DQS Enable ; Group 0 ; Rank 0 ; Center VFIFO  6 ; Phase 2 ;
  Delay  1
  ```

  Analysis of DQS Enable results: A VFIFO tap is 1 clock period, a phase is 1/8 clock period (45 degrees) and delay is nominally 25ps per tap. The DQSen window is the difference between the start and end—for the above example, assuming a frequency of 400 MHz (2500ps), that calculates as follows: `start is 5*2500 + 6*2500/8 +4*25 = 14475ps`. By the same calculation, the end is 16788ps. Consequently, the DQSen window is 2313ps.
- The size of a read window or write window is equal to `(left edge + right edge) * delay chain step size`. Both the left edge and the right edge can be negative or positive.:

  ```
  SEQ.C: Read Deskew  ; DQ  0 ; Rank 0 ; Left edge  18 ; Right edge
  27 ; DQ delay  0 ; DQS delay  8
  SEQ.C: Write Deskew ; DQ  0 ; Rank 0 ; Left edge  30 ; Right edge
  17 ; DQ delay  6 ; DQS delay  4
  ```

  Analysis of DQ and DQS delay results: The DQ and DQS output delay (write) is the D5 delay chain. The DQ input delay (read) is the D1 delay chain, the DQS input delay (read) is the D4 delay chain.

- Consider the following example of latency results:

```
SEQ.C: LFIFO Calibration ; Latency 10
```

  Analysis of latency results: This is the calibrated PHY read latency. The EMIF Debug Toolkit does not report this figure. This latency is reported in clock cycles.
- Consider the following example of FOM results:

```
SEQ.C: FOM IN  = 83
SEQ.C: FOM OUT = 91
```

  Analysis of FOM results: The FOM IN value is a measure of the health of the read interface; it is calculated as the sum over all groups of the minimum margin on DQ plus the margin on DQS, divided by 2. The FOM OUT is a measure of the health of the write interface; it is calculated as the sum over all groups of the minimum margin on DQ plus the margin on DQS, divided by 2. You may refer to these values as indicators of improvement when you are experimenting with various termination schemes, assuming there are no individual misbehaving DQ pins.
- The debug report does not provide delay chain step size values. The delay chain step size varies with device speed grade. Refer to your device data sheet for exact incremental delay values for delay chains.

Related Information

Functional Description–UniPHY

Writing a Predefined Data Pattern to SDRAM in the Preloader

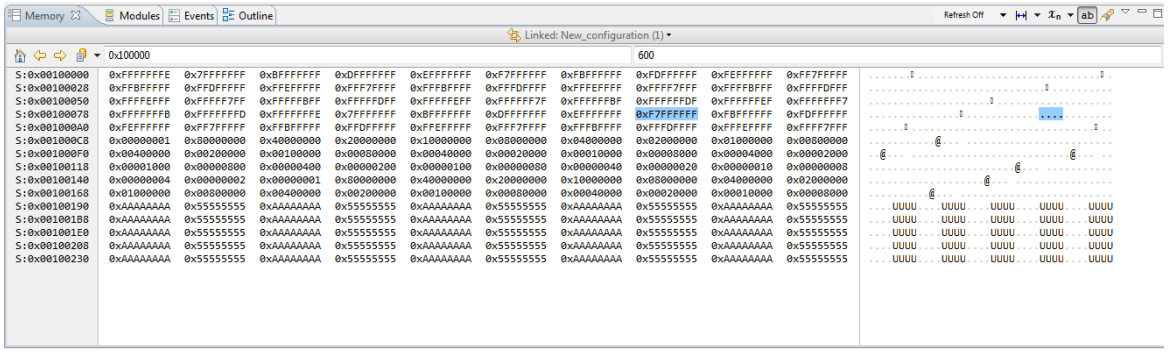
You can include your own code to write a predefined data pattern to the SDRAM in the preloader for debugging purposes.

1. Include your code in the following file: <*project_folder*>**\software\spl_bsp\uboot-socfpga\arch\arm\cpu\armv7\socfpga\spl.c** .
   For example, adding the following code to the **spl.c** file would cause the controller to write walking 1s and walking 0s, repeated five times, to the SDRAM.

```
/*added for demo, place after the last #define statement in spl.c */
#define ROTATE_RIGHT(X) ( (X>>1) | (X&1?0X80000000:0) )
/*added for demo, place after the calibration code */
 test_data_walk0((long *)0x100000,PHYS_SDRAM_1_SIZE);
int test_data_walk0(long *base, long maxsize)
{
 volatile long *addr;
 long          cnt;
 ulong        data_temp[3];
        ulong        expected_data[3];
 ulong        read_data;
       int           i = 0;  //counter to loop different data pattern

        int        num_address;

       num_address=50;

       data_temp[0]=0XFFFFFFFE; //initial data for walking 0 pattern

       data_temp[1]=0X00000001; //initial data for walking 1 pattern
```

```
       data_temp[2]=0XAAAAAAAA; //initial data for A->5 switching

       expected_data[0]=0XFFFFFFFE; //initial data for walking 0
pattern
       expected_data[1]=0X00000001; //initial data for walking 1
pattern
      expected_data[2]=0XAAAAAAAA; //initial data for A->5 switching


       for (i=0;i<3;i++) {

 printf("\nSTARTED %08X DATA PATTERN !!!!\n",data_temp[i]);
 /*write*/
 for (cnt = (0+i*num_address); cnt < ((i+1)*num_address) ;  cnt++ )
{
  addr = base + cnt; /* pointer arith! */
  sync ();
         *addr = data_temp[i];
  data_temp[i]=ROTATE_RIGHT(data_temp[i]);

  }

 /*read*/
 for (cnt = (0+i*num_address); cnt < ((i+1)*num_address) ; cnt = cnt++
 ) {
  addr = base + cnt; /* pointer arith! */
  sync ();
  read_data=*addr;
       printf("Address:%X  Expected: %08X    Read:%08X \n",addr,
expected_data[i],read_data);
           if (expected_data[i] !=read_data) {
               puts("!!!!!!FAILED!!!!!!\n\n");
  hang();
   }
               expected_data[i]=ROTATE_RIGHT(expected_data[i]);
}
}
}
====//End Of Code//=====
```

Figure 4-8: Memory Contents After Executing Example Code



Document Revision History

Date	Version	Changes
December 2013	2013.12.16	• Added *Generating a Preloader Image for HPS with EMIF* section. • Added *Debugging HPS SDRAM in the Preloader* section. • Enhanced *Simulation* section.
November 2012	1.0	• Initial release. • Moved *Using EMI-Related HPS Features in SoC Devices* from *Hard Memory Interface* chapter to this chapter.