SONICS™

MemMax® Reference

DRAM Scheduler System IP

MemMax Version: 5.0.1

Sonics Confidential

MemMax® Reference

Document Revision: February 11, 2016

Contents

About This Document

The Sonics MemMax DRAM System consists of a multi-threaded scheduler and DRAM controller that operate together to provide increased efficiency in memory access. The MemMax DRAM System is complex, with a rich set of configurable options that may be used to tune performance. You can use the SonicsStudio® Director, which is part of the SonicsStudio® Development Environment, to quickly lay out and configure system-on-chip designs that use the MemMax DRAM System.

This guide provides a single reference to the operation and use of the Sonics MemMax DRAM System.

This document contains the following sections:

- Chapter 1: Overview
- Chapter 2: Functionality
- Chapter 3: External Interfaces
- Chapter 4: Internal Organization
- Chapter 5: RTL Configuration
- Chapter 7: Tuning Guide
- Appendix A: Using Compiled Memory
- Appendix B: Fall-Through Latency

Companion Documents

Use the *MemMax Reference* in conjunction with the *SoC Integration Guide* and the *SonicsStudio Director User Guide*. These books contain additional information about the SonicsStudio Director, the tools flow, CLI tools, and other SonicsStudio Development Environment information. For information about the MemMaxC model, see the *SystemC Models Guide*.

Sonics Documentation Library

This section outlines the Sonics® documentation library by providing a brief description of each document. The following diagram shows the relationship of documents to various Sonics systems, products, and tools.

<table>
<tr><th colspan="2"></th><th>Sonics® System IP
Reference Information</th><th>SonicsStudio®
Development Environment</th></tr>
<tr><td rowspan="5">On-Chip Network System IP</td><td>SonicsGN®</td><td>SonicsGN Reference</td><td rowspan="8">SonicsStudio Director
Quick Start

SonicsStudio Director
User Guide

SonicsStudio Director
Tcl Reference

Design Flow Guide

SoC Integration Guide

SystemC Models Guide</td></tr>
<tr><td>SonicsSX®</td><td>SonicsSX Reference</td></tr>
<tr><td>SonicsLX®</td><td>SonicsLX Reference</td></tr>
<tr><td>Sonics3220™</td><td>Sonics3220 Reference</td></tr>
<tr><td>SonicsExpress™</td><td>SonicsExpress Reference</td></tr>
<tr><td>Monitor and Trace IP</td><td>SonicsMT™</td><td>SonicsMT Reference</td></tr>
<tr><td rowspan="2">DRAM Systems</td><td>MemMax®</td><td>MemMax Reference</td></tr>
<tr><td>MemDDR</td><td>SoC Integration Guide</td></tr>
<tr><td rowspan="2">Additional IP Products</td><td>SiliconBackplane™</td><td>SiliconBackplane III Reference</td></tr>
<tr><td>Sonics
Bridges</td><td colspan="2">SonicsConnect Reference
(AXI Bridges, AHB Bridges, APB Bridges,
SonicsConnect AXI and AHB Master Layers,
SonicsConnect AHB and APB Slave Branches)</td></tr>
</table>

On-Chip Network System IP

These documents provide reference information about Sonics' on-chip network system IP products:

- *SonicsGN® Reference*

  Describes the SonicsGN on-chip network system (SGN), which links cores in a scalable interconnect system that combines low-latency crossbars with high performance, pipelined *network-on-chip* (NoC) routers. The document provides details on SonicsGN components, mechanisms, configuration options, and implementation.

- *SonicsMT™ Reference*

  Describes the implementation and usage of the Sonics Performance Monitor and Hardware Trace (SonicsMT) IP, which can optionally be included in a

SonicsGN design. SonicsMT provides performance monitoring and trace capabilities to improve debugging.

- *SonicsSX® Reference*

  Describes the components, mechanisms, configuration options, and implementation of SonicsSX on-chip network system IP (SSX). SonicsSX solves interconnect needs for the most complex designs. With support for 2-D block burst and XOR burst types, data widths up to 256 bits, and multi-channel features, SonicsSX is a superset of SonicsLX.

- *SonicsLX® Reference*

  Describes the SonicsLX on-chip network system IP (SLX), which is a limited version of the SonicsSX that provides a full- or partial-crossbar bus structure suitable for mid-range designs. The document provides details on SLX components, mechanisms, configuration options, and implementation.

- *MemMax® Reference*

  Describes the Sonics MemMax DRAM Scheduler, an optimized DRAM subsystem.

- *SonicsExpress™ High Performance Asynchronous Bridge*

  Describes a highly configurable bridge that can be used to create a high-bandwidth AXI or Open Core Protocol (OCP) socket between two clock domains.

- *Sonics3220™ Reference*

  The source for information on the Sonics3220 on-chip network system IP, a non-blocking, peripheral interconnect that provides low latency access to a large number of low bandwidth target cores. The manual contains information on Sonics3220 components, mechanisms, configuration options, and implementation.

- *Asynchronous Bridge*

  Describes how to use the Asynchronous Bridge to connect two OCP ports operating in separate clock domains.

- *SiliconBackplane™ III Reference*

  Describes the SiliconBackplane MicroNetwork and its agents: InitiatorAgent Module, Enhanced Burst Initiator Module, TargetAgent Module, Enhanced Burst Target Module, and ServiceAgent Module.

System Integration Information

The following documents provide information about system integration:

- *SonicsStudio Director Quick Start*

  Provides a quick way to get started using the SonicsStudio Director. This guide give you a broad overview of its capabilities and features.

- *SonicsStudio Director User Guide*

  Describes how to use the features and capabilities of SonicsStudio Director to create SoC designs. Included in SonicsStudio Director are various editors to assist you through the design process. These include a Schematic Editor that provides a graphical representation of your design and a Design

Configuration Editor that provides text editing capabilities for directly modifying design configuration files. Parameter and configuration information is readily accessible through tables provided in a Properties view. SonicsStudio Director also performs continuous validation while you work on your design, making it possible to quickly identify any areas that need attention. Command line interfaces are readily available through Tcl and soccomp Output consoles.

- *SonicsStudio Director Tcl Reference*

  Describes the Tcl commands that you can use in Tcl scripts or issue from the Director Console. These Tcl commands can be used for performance analysis, in the design flow, with the creation and editing of designs, and in test files.

- *Design Flow Guide*

  Describes the design flow process. This process chains together multiple point tools from Sonics and other vendors into a flow that performs complex tasks with minimal user input. The *Design Flow Guide* also contains information about verification setup, SystemC Modeling flow, and how to use the capabilities of SonicsStudio Director for performance analysis.

- *SoC Integration Guide*

  Describes the SoC preparation tools, which are the SonicsStudio® command line interface tools. This document also includes information on design synthesis configuration files; built-in system, clock, and power bundles; AMBA bundles; RTL implementation issues such as tactical cells, RTL coding guidelines, testability and power management issues; and OCP, AXI, and MemDDR behavioral models. The *SoC Integration Guide* should be used with the *SonicsStudio Director User Guide* and appropriate IP reference manuals.

- *SonicsConnect Reference*, previously titled *AMBA Support Reference*

  Describes the SonicsConnect™ collection of AXI, AHB, and APB bridge components used to connect AMBA AXI, AHB, or APB cores to a device with an OCP interface. Also describes Master Layer and Slave Branch components used to connect multiple masters to a single initiator agent and multiple slaves to a single target agent.

- *SystemC Models Guide*

  Describes the Sonics SystemC/C++ modeling environment, which allows you to build a system-level, executable design using Sonics on-chip network system products. It discusses the modeling flow of the Sonics SystemC Environment, SystemC versions of Sonics Q-Models, SonicsGN C-model (SGNC), SonicsSX C-model (SSXC), SonicsLX C-model (SLXC), Sonics3220 C-model (S3220C), SonicsExpress C-model (ExpressC), MemMax Scheduler C-model (MemMaxC), and product libraries. Tools used to create and run the models are also described.

Supplemental OCP Documentation

The following Open Core Protocol (OCP) documents are managed by the Accellera Systems Initiative. They were previously issued by the OCP International Partnership, also known as OCP-IP.

- *Open Core Protocol™ Specification, Release 2.2*

Provides technical details for working with the OCP2 interface.

- *Open Core Protocol™ Specification, Release 3.0*

  Provides technical details for working with the OCP3 interface.

- *A SystemC™ OCP Transaction Level Communication Channel*

  Describes the SystemC model of an OCP channel. This model is meant for the system simulation of cores that use OCP to connect to one another. This document covers OCP specific versions of the SystemC channel: the OCP specific communication channels for transaction level one and transaction level two.

- *A SystemC™ Generic Transaction Level Communication Channel*

  Describes protocol primitives (functions and events) that can be used to build protocol-specific SystemC transaction level communication channels. Also describes how to use the generic channel to build another channel using OCP as an example.

1 Overview

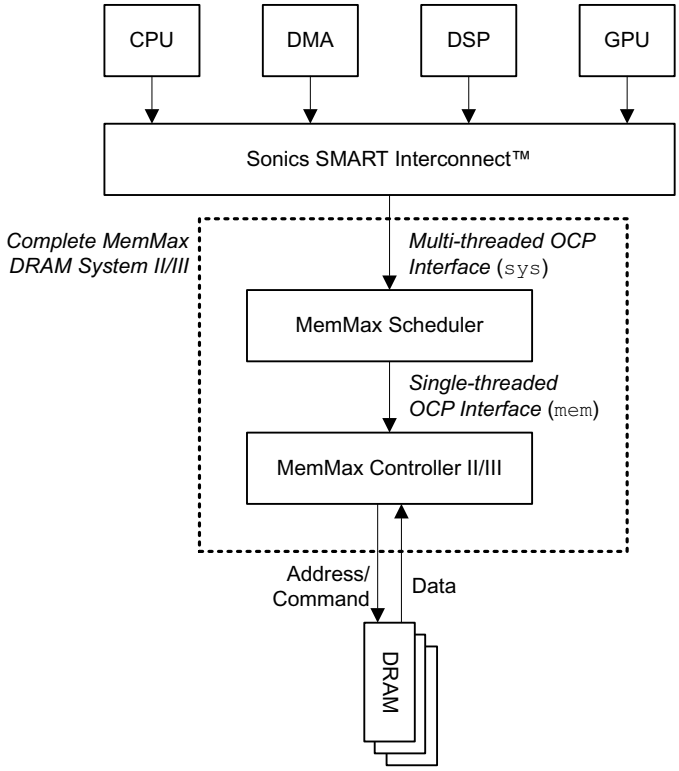
The Sonics MemMax® Scheduler is a key component of the Sonics MemMax DRAM System product line. The MemMax Scheduler prioritizes and schedules DRAM requests to meet the dual goals of maximizing DRAM bandwidth while preserving user-specified quality of service (QoS) requirements. The MemMax Scheduler passes scheduled requests to a DRAM controller—such as the MemMax Controller II—which handles the physical connection to the DRAM devices. A complete MemMax DRAM System is comprised of a MemMax Scheduler, a MemMax Controller, and SDRAM.

Figure 1 shows a typical system-on-chip (SoC) that implements a MemMax DRAM System for an OCP-to-OCP configuration. Four initiator cores—a general-purpose processor core (CPU), a digital signal processor (DSP), a DMA unit, and a graphics accelerator (GPU)—connect to the MemMax DRAM System through a Sonics on-chip network system IP. The MemMax DRAM System consists of a MemMax Scheduler instance, a MemMax Controller (the DRAM controller), and DRAM memory chips.

Features of the OCP-to-OCP MemMax Scheduler include:

- Support for multiple threads and tags
- High DRAM utilization for multiple data flows
- Per-thread quality-of-service scheme selection and per-tag starvation avoidance
- Support for user-configurable address tiling functions
- Support for Open Core Protocol (OCP) INCR, XOR, BLCK, and WRAP bursts
- Independence from DRAM controller technology
- Compatible with Sonics on-chip network system IP products

Figure 1 Typical SoC with shared DRAM subsystem for OCP



Components

The Sonics MemMax DRAM System solution splits the traditional functionality of a DRAM controller into two pieces: the MemMax Scheduler and a DRAM controller. These are shown in Figure 2.

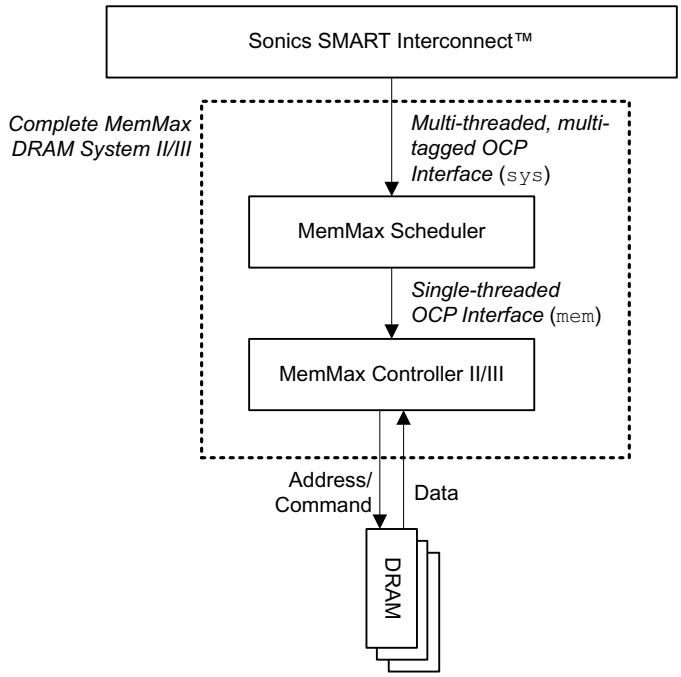
The MemMax Scheduler attempts to increase memory utilization by using knowledge of common DRAM architectural features (such as DRAM type and burst mode support) in the application of scheduling algorithms to minimize page misses and change-of-direction turnaround penalties, which are problems common to all DRAM systems. This allows the operation of the scheduler to be largely independent of the memory controller, which in turn can be relieved of complex thread scheduling functions for multiple thread- or tag-based data flows and instead concentrate on scheduling accesses to the DRAM.

Notes

- The SoC interconnect is connected to the MemMax Scheduler through a multi-threaded, multi-tagged OCP interface named *sys* that is compliant with *Open Core Protocol Specification, Release 2.2* (OCP2) The MemMax Scheduler is in turn connected to the DRAM controller through a single-threaded OCP interface named **mem**, The **mem** interface supports OCP2.
- The MemMax Scheduler supports OCP tags on the *sys* interface. It does not support tags on the *mem* interface.

- The MemMax Scheduler does not allow an AXI Master to be connected directly to the *sys* interface. However, you can use an AXI Master Layer to connect AXI Masters to the MemMax Scheduler.

Figure 2 MemMax Scheduler



Request Processing

The MemMax Scheduler and DRAM controller divide the work of processing DRAM requests as follows:

- The scheduler accepts DRAM requests from one or more initiator threads through its multi-threaded, multi-tagged OCP interface, *sys*.
- The requests are prioritized and (possibly) rescheduled to maximize DRAM bandwidth while preserving user-specified QoS requirements, creating a single stream of DRAM requests. As part of this process, the MemMax scheduler can also perform address translation on DRAM requests through the application of user-configurable address tiling (mapping) functions.
- The scheduler passes the single stream of requests to the DRAM controller through the scheduler's single-threaded and single-tagged OCP *mem* interface.
- The DRAM controller issues requests to the DRAM and returns responses to the scheduler through the *mem i*nterface.
- The scheduler passes the received data to the requesting initiator.

DRAM bandwidth efficiency is maximized when DRAM *turnaround cycles* are minimized. (Turnaround cycles are the number of cycles from request generation to response receipt.) A specific DRAM controller and DRAM devices have a specific minimum number of turnaround cycles. Additional turnaround cycles are incurred by *penalty events*: events that add cycles of latency, such as DRAM

page misses, data bus turnaround cycles, and DRAM rank switching. The scheduler tracks key aspects of the physical DRAM state, allowing it to reschedule requests to minimize penalty events and maximize efficiency.

In addition to tracking the DRAM state, the MemMax Scheduler uses QoS information to determine the best schedule. The MemMax Scheduler provides three QoS levels: *priority*, *allocated bandwidth*, and *best effort*. The use of QoS service levels provide additional flexibility to the designer and increase the effectiveness of the scheduler.

The scheduler performs two-level scheduling: first arbitrating among tags of the same thread and then between different threads to determine the final winner. At each scheduling step a weight is calculated for each candidate request, and the request with maximum weight wins the arbitration.

Prerequisites

Developing a system using the MemMax Scheduler requires familiarity with the material and concepts in the following:

- Open Core Protocol Specification, Release 2.2
- Applicable Sonics on-chip network system IP hardware reference manual,
- The Sonics AXI master layer described in the *SonicsConnect Reference*.

DRAM Controller Requirements

The features required in the DRAM controller are considered common. They can be found in a broad range of existing DRAM controller designs. The DRAM controller:

- Must return requests in-order in compliance with OCP ordering rules. (If this is not possible, a gasket must be implemented to enforce the ordering.)
- Must arrange address bits in the order (LSB to MSB): OCP word, DRAM block, column, bank select, row, chip select.
- Must immediately register all input signals on the controller's OCP interface.
- Must provide all output signals on the OCP interface directly from registers, without any intervening logic.
- Must have a minimum of one cycle latency from the time a request is accepted to the time a response is generated.
- Must handle the BL-unaligned burst access when `dram_block_size` is set to 8 or 16.

2 Functionality

The MemMax Scheduler accepts multiple data flows—each mapped onto an OCP thread—from the interconnect and schedules corresponding DRAM requests to the controller.

The MemMax Scheduler provides the following benefits:

- Different QoS levels can be allocated on a per-thread basis to effectively support system data flow requirements.
- Requests on different threads and tags can be reordered to maximize efficiency.
- User-specified address tiling functions can be used to transform request addresses to improve DRAM bandwidth efficiency.

This chapter also discusses the scheduler implementation of the multi-threaded, multi-tagged OCP interface (*sys*) that is optimized for use with Sonics on-chip network system IP. The *sys* OCP interface allows the scheduler to re-order responses, thus achieving the benefits of OCP non-blocking flow control in the request and response paths.

OCP Tag Support

The MemMax Scheduler supports the use of OCP tags on the *sys* interface (that is, the *sys* interface parameter `tags` can be set greater than 1). OCP tags are not supported on the *mem* interface.

When tags are enabled on the *sys* interface, the scheduler will consider the request's tag value as well as the request's address, burst type (if applicable), and the per-thread QoS settings when scheduling requests.

The MemMax Scheduler maps a request arriving on a thread to one of the *N* internal FIFOs assigned to that thread, where *N* is specified by the `tag_parallelism` parameter. Each FIFO is used to hold requests with the same tag ID. The number of FIFOs is the maximum number of outstanding tags that can be active within the MemMax Scheduler at any given time.

In general, increasing the value of `tag_parallelism` improves scheduling efficiency at the cost of increased scheduler area. Tag-level rescheduling can be disabled on a per-thread basis by setting the thread's `tag_parallelism` parameter to 1.

A registered input (RIN) FIFO is used to break long timing paths from the **sys** interface to command buffer registers. MemMax has per-thread request RIN FIFOs of depth 2. Each request goes through this RIN FIFO before it is mapped to an internal tag ID of the request pool. If the scheduler cannot map the request on the outputs of the RIN FIFO to an internal FIFO and the RIN FIFO becomes full (contains two requests), it will assert *SThreadBusy* for the affected **sys** thread until the request can be mapped and pushed into the request pool. Other conditions, which described below, may also result in *SThreadBusy* being asserted by the scheduler.

Note that when tags are enabled and used on the *sys* interface (that is, the *sys* OCP configuration parameter `tags` is set greater than or equal to 1), the *MTagInOrder* and *MTagID* signals from each request are not passed to the *mem* interface.

In-Order Tags

Any request that has *MTagInOrder* enabled is assigned a unique tag ID by the scheduler. The incoming tag ID of the request is ignored.

If the OCP parameter `taginorder` is enabled, a 32-entry Request2Data queue is instantiated to track requests with *MTagInOrder* asserted. The queue associates a request and data phase based on the *MTagInOrder* value of the request. (This is necessary as the data phase of requests with *MTagInOrder* = 1 do not have a corresponding *MDataTagInOrder* field.)

Scheduler Request Queue and SThreadBusy

If the *sys* interface has `sthreadybusy_pipelined` enabled, *SThreadBusy* is asserted at the *sys* interface when the request queue is almost full and the next request is a write. If the sys interface does not have `sthreadybusy_pipelined` enabled, *SThreadBusy* is asserted at the *sys* interface when the request queue is full.

Response Interleaving

Responses that are part of the same request are kept together, to a limit specified by the OCP parameter `tag_interleave_size`. A non-zero value for `tag_interleave_size` allows blocks of responses from different requests to be interleaved, which may reduce system latency. If `tag_interleave_size` is set to zero, response interleaving is only performed at *sys* burst boundaries.

Note: The MemMax Scheduler supports `tag_interleave_size` values to (0|1|2|4|8|16) with one restriction. The restriction is that `dram_block_size` and `dram_blk_logsize` settings must be configured to a value greater than or equal to the value of the **sys** interface option `tag_interleave_size`. See "DRAM Block Size".

The following restrictions apply to response interleaving:

- For bursts of type WRAP and XOR, `tag_interleave_size` is assumed to be zero, irrespective of the actual value of `tag_interleave_size`.

- If the burst length of the first chopped burst of a *sys* burst is not equal to the DRAM block size (which equals 2^*DRAM_BLK_LOGSIZE*), the entire *sys* burst is scheduled together on that thread, and no tag level interleaving will be performed.
- Further, whenever the address crosses the alignment boundary given by `(data_wdth/8)` < $\log_2$(`tag_interleave_size`), tag interleaving is reset and a new interleaving group is started.
- If `writeresp_enable` is enabled on the *sys* interface, and disabled on the *mem* interface, the MemMax Scheduler will only allow configurations with `tag_interleave_size` = 0.

Requests with Overlapping Addresses

This section refers to OCP compliance rules for transactions with overlapping addresses. Note that as stated in the OCP Specification, tagged transactions to overlapping addresses must always be committed in order. MemMax has implemented address checking that is compliant with this OCP rule.

The MemMax Scheduler provides the `disable_tag_addr_overlap` configuration parameter to allow users to optimize address overlap behavior on a per-thread basis. The setting of `mem_clock_async` affects the use and interpretation of this parameter as described below.

Synchronous Operation

When `disable_tag_addr_overlap` is set to 0 (the default), the MemMax Scheduler implementation will include logic to check each new request for address overlap with a pending request. (The check is performed on a per-thread basis.) If there is an address overlap, the new request will only be scheduled after the pending request has been sent to the memory controller. The address bits used in the address overlap comparison are specified by the `tag_addr_lsb` and `tag_addr_msb` configuration parameters.

When `disable_tag_addr_overlap` is set to 1, the MemMax Scheduler implementation does *not* include logic to check for overlapping request addresses and should, therefore, be avoided.

Asynchronous Operation

When `mem_clock_async` = 1 and `tags` is enabled, the MemMax Scheduler does not check for overlapping addresses that are out of order. Transactions with the same thread ID may be re-ordered if they are on different tag IDs, regardless of address overlapping between the transactions. That is, only the configuration `disable_tag_addr_overlap` = 1 is supported. In such situations, the user should ensure that overlapping addresses on the same thread are temporally separated to ensure consistency.

Furthermore, when `disable_tag_addr_overlap`=1 and a master issues a ReadEx/Write pair, the master should not issue another request on the same thread with an overlapping address until the ReadEx/Write pair has completed its transaction.

Important! Disabling address overlap checking (`disable_tag_addr_overlap` = 1) can significantly reduce the area of a MemMax Scheduler implementation. However, OCP violations may result if requests with overlapping addresses are sent to the MemMax Scheduler.

Quality-of-Service Scheduling

Quality-of-service (QoS) scheduling includes:

- QoS Levels
- Dynamic QoS Level Adjustment
- Dynamic QoS Parameters

QoS Levels

MemMax supports three quality-of-service (QoS) levels that are used during arbitration:

Best-effort
: Provides the lowest level of service, without any service guarantees. Best-effort threads only receive service when no higher-priority thread is in need of service, making use of any bandwidth left over by the priority or allocated-bandwidth threads.

Allocated bandwidth
: Provides a minimum sustained bandwidth on the thread. If the allocated bandwidth is exceeded, the thread's service level drops to best effort. The allocated bandwidth service level is useful for applications such as video decoding that transfer data at a more-or-less constant rate.

Priority
: Provides the highest level of access to a thread, minimizing latency. Priority threads have a specific bandwidth allocation: if the bandwidth allocation is exceeded, the thread's service level drops to best effort. The priority service level is useful for latency-sensitive initiators with burst access patterns, for example, for CPUs that require low-latency cache fill.

The QoS filter lets a thread go through and be a candidate for the next stage if there is no other thread whose current QoS level is higher than this thread. The QoS level of the threads is calculated during the QoS update stage.

Dynamic QoS Level Adjustment

The MemMax Scheduler provides fairness by dynamically adjusting the QoS mode of each thread between its programmed value and best-effort. The adjustment is made based on a per-thread credit value that indicates whether or not a thread is being over- or under-serviced.

By default, the thread's credit is incremented at a rate consistent with the user-specified bandwidth allocation. For example, if the thread has been allocated 40% of the total bandwidth, then the credit value is incremented by two counts every five cycles (on average). If the thread's credit is positive, the thread is under-serviced, and it is dynamically promoted to the programmed QoS level. Higher QoS level threads always win arbitration. In the case of multiple priority

threads, the arbitration mechanism picks the least-recently serviced priority thread.

When a thread wins arbitration, its credit is reduced by the number of words serviced in that transaction. If the credit becomes negative, then the thread has been over-serviced, and the thread is dynamically demoted to the best-effort QoS level. Note that demotion does not prevent the thread from being serviced.

Promoted (or demoted) threads that have a credit of zero are being serviced correctly, and are returned to their programmed QoS level.

Dynamic QoS Parameters

The rate at which threads accumulate credit is specified with the *BANDWIDTH_RATE* register field of the thread's scheduling register. This register field is 16 bits wide and specifies the number of cycles between credit increments. When `bandwidth_high_precision` is set to 1, the default remains 2. The interpretation is 2/256 = 1/128 of the total bandwidth; that is, 0.7%.

The thread's credit saturates at user-specified limits. The MemMax Scheduler supports two QoS modes—*normal* and *fine-grained*. (Mode selection is made using the `fine_grained_qos` parameter described in "Fine-Grained QoS Mode".)

In normal QoS mode, the user can specify a single saturation limit that applies to both credit accumulation and credit reduction. In fine-grained QoS mode, the user can specify different limits for credit accumulation and credit reduction.

The saturation limits are specified using register fields in the primary and extended per-thread scheduling registers. In normal QoS mode, only the primary per-thread scheduling register is available. The register field WORDS_PER_PERIOD specifies the saturation limit that is applied to credit accumulation and credit reduction.

In fine-grained QoS mode, the extended per-thread scheduling register is available. This register provides the register field WORDS_PER_PERIOD_MAX, which is used to set the saturation limit for credit accumulation. In this mode, the register field WORDS_PER_PERIOD is used to set the saturation limit for credit reduction.

Configuration statements to set the values of the register fields are described in "Bandwidth Allocation". The layout of the per-thread scheduling registers are described in "THREADSCH(0-15)".

Scheduling

Improving DRAM efficiency requires arranging requests to avoid events that interrupt a smooth pipelined flow of operations in the DRAM. Events that can impede operation include page misses (particularly page misses to a bank that has recently been opened), data bus turnaround (that is, switching between reads and writes), and switching between physical ranks.

The scheduler issues requests to the controller as bursts of a short, configurable DRAM block size that is typically set to match the DRAM burst length. If the transfer of less than a full block of data is requested, the scheduler can issue smaller-sized bursts down to a minimum size of a single OCP word.

Page Policy

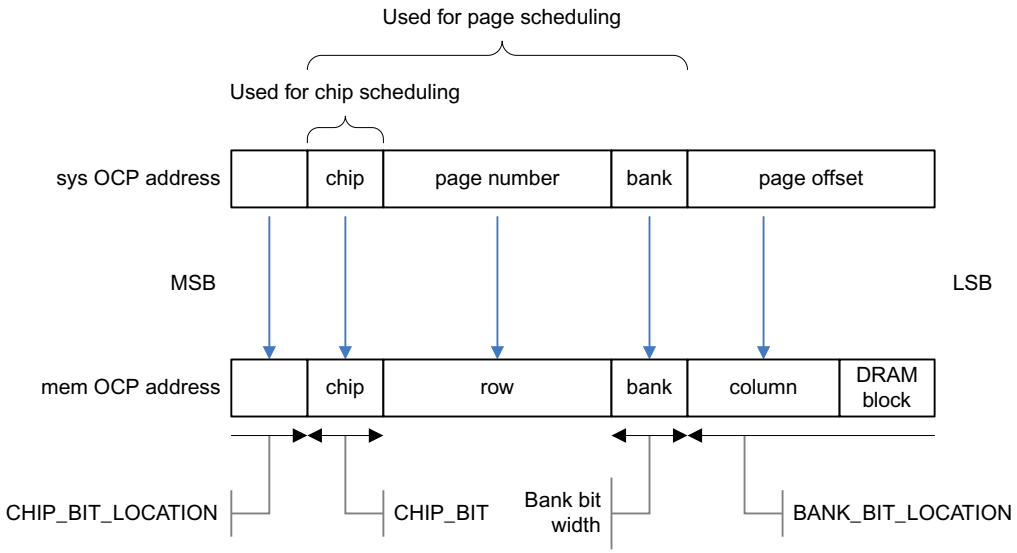
The MemMax Scheduler assumes that once a DRAM page is opened, it stays open until it is explicitly closed by either of the following:

- An access that indicates an auto-precharge is required (this is controllable per-thread).
- An access to a different page in the same bank.

Therefore, the scheduler tries to issue multiple requests to the same open page to minimize the cost associated with closing one page and opening a new one. However, the scheduler may close an open page—even if there are multiple pending requests for that page—and open a new page in order to satisfy quality-of-service requirements.

Figure 3 shows the address bit use and shifting for open pages. The bank bits are placed above the DRAM page size, as configured by the `bank_bit_location` parameter (described in "Bank Bit Location"). The scheduler uses the bank bits and chip bits (if any) to determine the bank number. The bits in between the chip bits and bank bits are treated as a page number. The bank number and page number are used by the scheduler to favor the open page. Any chip bits are used to avoid switching between chips. The maximum number of open pages that can be tracked is 16. Therefore, if the DRAM consists of 8 banks, the scheduler can track at most two ranks. If the DRAM consists of 4 banks, the scheduler can track up to 4 ranks.

Figure 3 Address Map for Open Pages



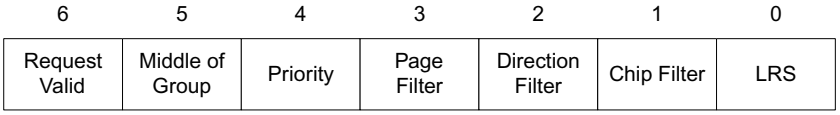
Arbitration

The MemMax Scheduler performs arbitration in two steps: first, requests on the same *sys* thread are arbitrated at the tag level, and then the winning requests from each thread are arbitrated at the thread level.

Per-Tag Arbitration

The scheduler arbitrates among the tagged requests on each thread by assigning a seven-bit weight vector to each request and choosing the request with maximum weight. The values of each bit in the weight vector for each request are updated each cycle, based on the state of the system. The bits of the weight vector for the tag-level arbitration are shown in Figure 4.

Figure 4 Per-Tag Weight Vector Bit Assignments

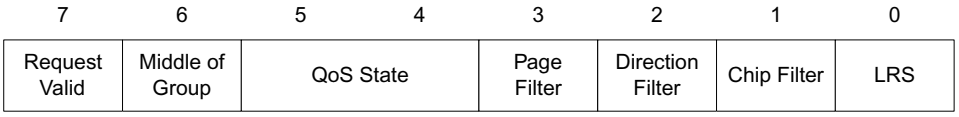


With the exception of the Priority field (bit 4), the fields in the per-tag weight vector are identical to the fields of the per-thread weight vector, and are computed in the same way. (The weight computation filters are discussed in "Weight Vector Bit Computation".)

Per-Thread Arbitration

The MemMax Scheduler arbitrates among multiple threads every clock cycle by assigning a eight-bit weight vector to each thread and choosing the thread with maximum weight. The values of each bit in the weight vector for each thread are updated each cycle, based on the state of the system. The bits of the weight vector for the thread-level arbitration are shown in Figure 5, below:

Figure 5 Per-Thread Weight Vector Bit Assignments



Weight Vector Bit Computation

The computation of the values for each bit are discussed below.

Request Valid

If the thread has a valid request pending, the *Request Valid* bit of the weight vector is set to 1. A request is valid only if all the conditions below are true:

1. There is a request pending on the thread to be sent to the controller.
2. If the request would result in a page miss, the bank busy flag for the corresponding bank is not high.
3. If the request is a write and `reqdata_together` is high for the *mem* OCP interface, the *mem SDataThreadbusy* signal should be de-asserted.
4. If the request is a read, there is enough room in the scheduler's internal read data buffers to hold the response(s).
5. An OCP ReadEx lock is set and the request on the thread tries to access the address. (See also "Mutual Exclusion".) Note that currently, the MemMax Scheduler supports only one outstanding ReadEx lock.

6. If a request is a write, all of the data words for the chopped request are available to be sent.

Middle of Group

The scheduler allows the user to specify a non-zero group size, *Gt*, for each thread, *t*, using the parameter `request_group_size` (described in "Request Group Size"). When a request from a thread that has a non-zero group size is scheduled, the scheduler considers the request to be part of a group. The MemMax Scheduler will attempt to continue to schedule requests from the same thread until one of the following conditions is met:

1. *Gt* requests have been scheduled for the thread.
2. The scheduler does not have a valid request to send on the thread.
3. All requests corresponding to the *sys* transaction on the thread have been scheduled.

When the scheduler is processing a group, requests from any other thread—including priority threads—are deferred until the group has completed.

QoS State

Two bits in the weight vector indicate the thread's dynamic QoS service level, which would be best-effort if the thread has been demoted, or its programmed QoS level if the thread has not been demoted.

Table 1 QoS Service Level Weight Vector Bit Assignments

Weight Vector Bit		Meaning
Bit 5	**Bit 4**	
0	0	Best-effort
0	1	Bandwidth
1	0	Priority
1	1	Reserved

Priority

Each request on the *sys* interface is assigned to an internal FIFO that has an associated *starvation counter*. In each cycle that the request does *not* win tag-level arbitration, the counter is decremented by one until the counter reaches zero (at which point it is no longer decremented). The priority bit in the tag-level weight vector is set to one if the starvation counter is zero, and set to zero otherwise. This *starvation avoidance* mechanism ensures that every internal FIFO can eventually win tag-level arbitration and be processed.

When the request—including all applicable sub-bursts—has been scheduled, the starvation counter is reloaded with the value specified in the requesting thread's STARVATION_COUNTER register field. Configuration of this register field is described in "Starvation Avoidance Counter".

Note: If the starvation counter reload value is set to 0, starvation avoidance is disabled for the requesting thread.

Page Filter

The scheduler applies a predictive page filter algorithm to determine whether a request is a page miss or a page hit. It compares the address of the request with that of the current open page in the bank to which the request is directed, and marks the request as a page hit if the request has identical row and bank addresses as the currently open page. Due to timing considerations, this comparison is performed a cycle before the scheduling arbitration. In the current cycle, the output of the page filter will be overridden to be a page miss if a different thread accesses the same bank as that of the request.

Direction Filter

The MemMax Scheduler acts to prevent frequent changes in data direction that would lower DRAM bandwidth efficiency. If the current request and the last scheduled request have the same direction, the direction filter bit is set to 1, otherwise it is set to 0. For example, if the last scheduled request was a read, the bit is set to 1 if the pending request on the thread is also a read.

Chip Filter

The scheduler also attempts to minimize rank switching. This is accomplished by introducing a rank-switch[1] bit to the arbitration weight vector, which is set to 1 if the last scheduled request and the pending request on the thread belong to the same rank in the DRAM subsystem.

Least-Recently Serviced (LRS)

When all other weight bits are equal, the scheduler uses a least-recently serviced algorithm to determine the winning thread. The LRS algorithm sets the candidate thread that was least-recently serviced as the winning thread. Invalid requests are not considered by the LRS algorithm.

Mutual Exclusion

The MemMax Scheduler can be configured to support mutual exclusion between threads using the OCP ReadEx/Write command sequence. The scheduler implements mutual exclusion using a lock status bit and lock address; only one lock may be active at a time. The locking granularity is a single OCP word.

Address locking is implemented as follows:

- Initially, the lock status bit is reset (for example, after device reset). Then, when a ReadEx request is received and scheduled, it is treated as a Read command and the scheduler also sets the lock status bit and records the request address as the locked address. To clear the lock status bit, the locking thread must follow the ReadEx request with a Write request to the locked address. Any other requests to the locked address and any other ReadEx requests from any thread are ignored and are not scheduled.
- Furthermore, when `disable_tag_addr_overlap`=1 and a master issues a ReadEx/Write pair, the master should not issue another request on the same thread with an overlapping address until the ReadEx/Write pair has completed its transaction.

1 In this document the terms *rank* and *chip* are used interchangeably.

Configuration Register Access

The MemMax Scheduler contains registers for run-time configuration (or reconfiguration). The DRAM controller may also have configuration registers. MemMax Scheduler configuration registers are defined in “RTL Configuration”. See the DRAM Controller documentation for details on any available DRAM Controller configuration registers.

Configuration registers are accessed using the target agent’s *MAddrSpace* signal: when MAddrSpace is zero, the registers are accessed, and when MAddrSpace is non-zero, DRAM is accessed. MemMax supports two modes for accessing configuration registers: *pass-through* and *masked*. Both of these modes are described below. The access mode is set at design time using the parameter `mask_reg_access_msb_addr_bits`, described in “Configuration Register Access Mode”.

Note that independent of the access mode used, the DRAM controller must return responses for register reads or writes in the order that the requests were received. DRAM controller register writes only return responses when the memory socket is configured with `writeresp_enable` enabled (that is, set equal to 1). When `writeresp_enable` is disabled on the *mem* interface and enabled on the *sys* interface, responses to the *sys* OCP interface are internally generated by the MemMax Controller.

Note: Configurations with `writeresp_enable` disabled on the *sys* interface and enabled on the *mem* interface are not supported.

Pass-Through Configuration Register Access Mode

In pass-through mode, shown in Table 2, MemMax Scheduler registers are located at the beginning of the configuration register address space, and DRAM controller registers start at the address specified by the power-of-two given by the REG_BASE_ADDR register field.

For access to registers in the DRAM controller, the MemMax Scheduler forwards the value of *MAddrSpace* at the *sys* OCP interface (which should be 0)[2], and masks off the most-significant bits of the address that are at or higher than the bit position specified by the REG_BASE_ADDR field, which must be a power of 2.

Table 2 Configuration Register Space Memory Map

Address	Register	
0x000	DRAMCONFIG	Global
0x008	DRAM_REGCONFIG	

2 The *MAddrSpace* signal at the *mem* OCP interface is one bit wide.

Address	Register	
0x020	Thread 0 schedule (primary)	Per-Thread Scheduling
0x028	Thread 0 schedule (extended)	
0x030	Thread 1 schedule (primary)	
0x038	Thread 1 schedule (extended)	
...		
0x110	Thread 15 schedule (primary)	
0x118	Thread 15 schedule (extended)	
0x120	TILING_SWAP(0)	Address Tiling Configuration
0x130	TILING_BAOP(0)	
...		
0x1E0	TILING_SWAP(6)	
0x1F0	TILING_BAOP(6)	
...		
reg_base_addr	DRAM controller configuration register space	

Masked Configuration Register Access Mode

In masked mode, shown in Table 3, both MemMax Scheduler configuration registers and DRAM Controller configuration registers are located in parallel address spaces. The REG_BASE_ADDR field in the DRAMRECONFIG register specifies a bit-mask that is applied to the request address when *MAddrSpace* is zero. Then:

- If the masked address is zero, the request is for the Scheduler configuration registers.
- If the masked address is non-zero, the request is for the DRAM Controller configuration registers.

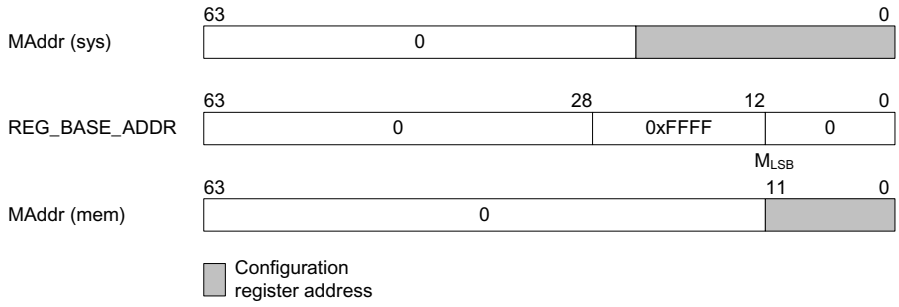
In either case, only bits `MLSB:0` of *MAddr* are used as the address of the selected configuration register, where `MLSB` is the position of the least-significant 1 in REG_BASE_ADDR.

Table 3 Configuration Register Space Memory Map

Address	MemMax Scheduler Register		DRAM Controller
0x000	DRAMCONFIG	Global	DRAM Controller Registers
0x008	DRAM_REGCONFIG		
0x020	Thread 0 schedule (primary)	Per-Thread Scheduling	
0x028	Thread 0 schedule (extended)		
0x030	Thread 1 schedule (primary)		
0x038	Thread 1 schedule (extended)		
...			
0x110	Thread 15 schedule (primary)		
0x118	Thread 15 schedule (extended)		
0x120	TILING_SWAP(0)	Address Tiling Configuration	
0x130	TILING_BAOP(0)		
...			
0x1E0	TILING_SWAP(6)		
0x1F0	TILING_BAOP(6)		
...			
LSB of REG_BASE_ADDR			

Figure 6 illustrates the mask access mode operation for a DRAM Controller configuration register access, where the result of the mask operation is non-zero. In this example, M_{LSB} is equal to 12.

Figure 6 Example DRAM Controller Configuration Register Access, Masked Mode



Address Tiling

The MemMax Scheduler can be configured to apply a user-defined transformation to the address bits in a request in a process known as *address tiling*. The goal of address tiling is to increase the number of page hits to the

underlying physical DRAM and hence improve the efficiency of the DRAM subsystem.

The user may define up to seven distinct address tiling transformations at run-time, each corresponding to a unique non-zero address space (that is, a unique *MAddrSpace* value). Each address tiling function is programmed using two configuration registers within the MemMax Scheduler. These registers are described in "MemMax Register Block".

Address tiling is performed after the input request has been chopped into `dram_block_size`-sized chunks (the `dram_block_size` parameter is described in "DRAM Block Size"). After applying the tiling function, the scheduler sets the *MAddrSpace* value output on the `mem` interface to 1 before the request is passed to the DRAM controller.

Transform Steps

The address tiling operation is performed by applying two address transformations in parallel: a bit-swapping transformation and a bitwise-mask-then-ADD transformation. The bit-swapping operation is used to re-order the address bits of the untiled address. The bitwise-masking-then-ADD operation is used to select a different bank in the event of a page miss, such that page precharge costs are reduced.

Address tiling is specific to each physical DRAM chip. Therefore, all tiling transformations are done "below" (that is. to the bits to the right of) the LSB of the chip-select bits. The bit position of the least-significant bit (LSB) of the chip bits is calculated as follows:

```
IF (chip_bits > 0)
   chip_bit_lsb = addr_wdth - chip_bit_location - chip_bits
ELSE
   chip_bit_lsb = addr_wdth - 1
```

For example, if the value of the `addr_wdth` parameter of the *sys* OCP interface is 32, `chip_bit_location` is equal to 5, and there is one chip bit (`chip_bits` = 1), then `chip_bit_lsb` is equal to 26.

Support for Address Bit-Swapping

The request address provided via the *sys* interface is known as the *un-tiled address*. The address produced by the bit-swapping transformation is known as the *swapped address*. The address produced by the bit-swapping transformation and the bank-address transformation is known as the *tiled address*; it is the tiled address that is output on the *mem* OCP interface *MAddr* signal.

In general, an un-tiled address can be considered to be a concatenation of an ordered set of segments *SM*, each with a corresponding length, *LM*. The number of segments, *M*, is defined to be an integer with $0 \leq M \leq 5$. The un-tiled address is then $\boldsymbol{S} = \{S5, S4, S3, S2, S1, S0\}$, with the LSB of *S0* corresponding to the LSB of the input address, and the most-significant bit (MSB) of *S5* corresponding to the MSB of the input address.

A new address can be formed by re-ordering the segments to make a new address, $\boldsymbol{S'}$. The re-ordering follows the following rules:

- The user can divide the untiled address into at most six segments by specifying the lengths *LM*.

- The segments are contiguous.
- The segments are mapped in a one-to-one fashion to the untiled address.
- Given segmentation of an input untiled address into $N \leq 6$ segments:
- The segment *S0* has an LSB at a fixed bit position (bit location 0), and a length $S0 \geq \log_2$(DRAM block size in bytes).
- For segments *SM*, where $0 < M < N$, the LSB is at bit position LSB_M, where LSB*M* = LSB*M-1* + L*M-1*.
- For the last segment *SN-1*, the position of the LSB and segment length are given by LSB*N-1* = LSB*M-2* + L*N-1* and L*N-1* = `chip_bit_lsb` + LSB*N-1*.

For segments *S1* through *S5*, the location of each segment's LSB in the untiled address is given by the user-specified parameters `ILOC1` through `ILOC5`. (As noted above, the location of the LSB of segment *S0* is fixed.)

The bit-swapped address is then generated using the bit-swapping algorithm shown in Figure 7.

Figure 7 Algorithm for the Address Bit Swapping Operation

```
Swapped_Addr[L0-1:0] = Sys-OCP MAddr[L0-1:0];
LSB0 = 0;
W = MAddr width;
FOREACH integer M, where 0 < M < N:
    LSBM = LSBM-1 + LM-1;
    IF (M equals N - 1): LM = chip_bit_lsb - LSBM;
    Swapped_Addr[LSBM + LM-1: LSBM] =  Sys-OCP MAddr[ILOCM + LM-1:ILOCM];
Swapped_Addr[W-1:chip_bit_lsb] = Sys-OCP MAddr[W-1:chip_bit_lsb];
```

Bank-Address Transformation

The only supported bank address transformation is for bank selection. The MemMax Scheduler provides a flexible *bitwise-masking-add-modulo* (BAM) function that is used to transform the bank selection bits in the request address. A maximum of eight banks are supported (specifically, *`BA0P.width`* can be configured to be 1, 2, or 3 if *`BANK8`* is set to 1, and 1 or 2 if *`BANK8`* is set to 0). The BAM function is defined as follows (where *K* is the position of the least-significant bank address bit in the tiled address):

Figure 8 Algorithm for Bank Address Transformation

```
K = BANK_BIT_LOCATION;

IF (DISABLE_BIT_MOVE == 0)
    Tiled Mem-OCP MAddr[BAOP.low_loc + BAOP.width - 1 : \
    BAOP.low_loc] = Sys-OCP_Addr[K + BAOP.width -1: K]
END-IF

Tiled Mem-OCP MAddr[K+BAOP.width-1:K] =
    LSB BAOP.width bits of  ( (BAOP.high_en & Untiled Sys-OCP \
    MAddr[BAOP.high_loc+BAOP.width-1:BAOP.high_loc]) + \
    (BAOP.low_en  & Untiled Sys-OCP MAddr[BAOP.low_loc +\
    BAOP.width -1 :BAOP.low_loc]))
```

To improve the available design space for exploring tiling functions, you can use the DISABLE_BIT_MOVE register field to prevent the bit move operation during the BAOP transformation of the tiling function. To disable, set DISABLE_BIT_MOVE to 1.

The values *BAOP.high_en* and *BAOP.low_en* are bit enable masks, each *BAOP.width* bits wide. The enable bit masks are bitwise-ANDed with each of two *BAOP.width*-wide bitfields extracted from the untiled address:

- The first extracted bitfield contains high-order untiled address bits, with the position of the least-significant bit of the extracted bitfield given by *BAOP.high_loc*.
- The second bitfield contains low-order untiled address bits, with the position of the least-significant bit of the extracted bitfield given by *BAOP.low_loc*.

The two bit-masked fields are then summed to produce a single bitfield that is clipped to be *BAOP.width* bits wide. This final bitfield result is then written into the swapped address, positioned so that the right-most bit of the result is placed at the bit position given by the value of the `bank_bits_location` parameter (or *BANK_BITS_LOCATION* register field), creating the final tiled address.

An example of the bank address transformation is shown in Figure 9. The example assumes that the tiling configuration parameters are those shown in Table 4.

Configuration for Address Tiling

As noted earlier, the MemMax Scheduler supports seven address tiling functions, one for each non-zero value of the *sys* port OCP signal *MAddrSpace*. (An *MAddrSpace* value of zero is reserved for accessing the configuration registers in either the MemMax Scheduler or the DRAM controller.) Each tiling function is specified using a dedicated pair of scheduler configuration registers: *TILING_SWAP*, used to specify the parameters for address bit-swapping, and *TILING_BAOP*, used to specify the parameters for the bank-address transformation. There are seven pairs of tiling function configuration registers, corresponding to the seven non-zero values of *MAddrSpace*. If *K* tiling functions are configured, they must be configured for contiguous values of *MAddrSpace* starting from 1.

To disable either the address swapping of bits or the bank-address transformation function or both, the corresponding register fields should be programmed with the value zero. If no transformation needs to be done, all the fields of the registers should be programmed to 0, which is the default. It is also important to note that the segments must always start at S0, and be sequential. In other words, {S0, S1, S2} is a valid description, but {S0, S1, S3}, and {S1, S2, S3} are not valid.

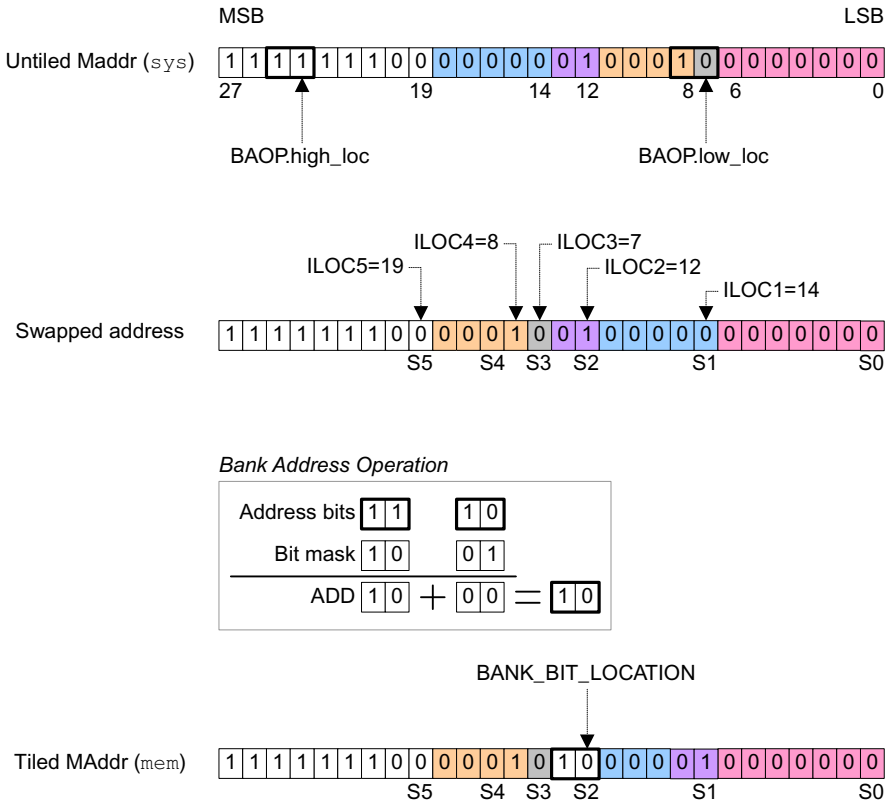
Example Tiling Function

The configuration parameters and operation of an example tiling function is Table 4 and Figure 9. The example assumes that the `bank_bit_location` parameter is set to 12, the `bank8` parameter is set to 0, the `chip_bit_location` parameter is set to 0, and the OCP parameter `addr_wdth` for both *sys* and *mem* interfaces is set to 28. Table 4 lists the values programmed into the *TILING_SWAP* and *TILING_BAOP* registers to create this address tiling function. The operation of the tiling function itself is shown in Figure 9.

Table 4 Example Address Tiling Configuration Settings

Bit	Value	Bit	Value
Bit-Swapping Function Parameters			
Len0	7	ILOC1	14
Len1	5	ILOC2	12
Len2	2	ILOC3	7
Len3	1	ILOC4	8
Len4	4	ILOC5	19
Bank Address Transformation Parameters			
BAOP.low_en	2b'01	BAOP.low_loc	7
BAOP.high_en	2b'10	BAOP.high_loc	24
		BAOP.width	2

Figure 9 Address Tiling Example, Assuming Settings From Table 4



Note: Any write to a tiling configuration register must be done with full data byte enables. That is, the entire register must be modified in a single write. Partial writes into a tiling configuration register are not supported. If the sys data width is less than 64 bits, consecutive writes must be used to write all 64-bits of the tiling register.

As shown in the Figure 9, the input request address is 0xFE01100, and is 28 bits wide. In the first stage, the address bit-swapping transformation and bank address transformation computation are performed in parallel. The final tiled address is obtained by replacing the bank select bits in the swapped address by those obtained from the bank address transformation.

Note that both transformations are specified with respect to the original un-tiled address. That is, the value of *BA0P.high_loc* of 24 represents the bit at bit position 24 in the untiled address, and not bit 24 in the transformed address.

Per Thread Configuration of Address Tiling

The MemMax Scheduler allows you to select a subset of the available threads and perform address tiling only on those threads. You can specify this by setting the `per_thread_tiling` parameter. The parameter is defined as follows:

```
per_thread_tiling{
     addrspace 1 { threads all | threads n, n1, n2,... }
     addrspace 2 { threads all | threads n, n1, n2,... }
     ...
}
```

If the `per_thread_tiling` parameter is not included in the RTL configuration file, all threads apply to all the programed tiling functions on all threads. Otherwise, the MemMax Scheduler uses the MAddrspace versus threads table to denote the threads on which the request with a given address space can appear. For example, assume a configuration with 8 threads. If thread 0 applies the tiling function corresponding to MAddrspace 2, thread 1 applies the tiling function corresponding to MAddrspace 2 and MAddrspace 3, and all threads have a pass through option corresponding to MAddrspace 1. In this case, you define the `per_thread_tiling` parameter as follows:

```
per_thread_tiling {
    addrspace 1 { threads all }
    addrspace 2 { threads 0 1 }
    addrspace 3 { threads 1   }
}
```

When an `addrspace` is not present in the block, the MemMax Schedule applies the corresponding tiling function to all threads.

The MemMax Scheduler also allows you to specify the bounds of address tiling bits within the incoming address by specifying the `ADDR_TILING_ST_BIT` and `ADDR_TILING_END_BIT` bits. These bits are set to 0 and address width - 1, respectively. You can change these bits to limit the range within which address tiling is done. For example, if all address tiling bits are within the bits 21 and 8 respectively, you could set `ADDR_TILING_ST_BIT` to 8, and `ADDR_TILING_END_BIT` to 21. In this case, all bits above 21 and below 8 never participate in address tiling. Specifying these bits helps reduce the gate count of the design. However, note that these are design time parameters, and you cannot change them at runtime. Therefore, you should take care to ensure that the address tiling bits are always within the range these parameters specify. The following is an example of per thread tiling with the start bit set to 5 and the end bit set to 57.

```
per_thread_tiling {
    addrspace 1 { threads 2 4 5 6 8       }
    addrspace 2 { threads 0 2 4 5 6 7 8 9 }
```

```
    addrspace 3 { threads 1 2 34 5 6 9    }
}
param addr_tiling_st_bit 5
param addr_tiling_end_bit 57
```

3 External Interfaces

The MemMax Scheduler connects to the system primarily through two interfaces, sys and `mem`.

- The sys interface is a multi-threaded interface that connects the scheduler to the interconnect.
- The `mem` interface is single-threaded and connects the scheduler to the DRAM controller.

Both interfaces support the *Open Core Protocol Specification, Release 2.2* (OCP2).

Supported OCP Configurations

The OCP interfaces of the MemMax Scheduler support a subset of OCP configurations that are summarized in this section. All OCP configuration parameters follow OCP2 dependency and default tie-off rules.

Table 5 Basic Signals

OCP Signal Name	Enable Parameter(s)	Allowed Values for Ports		Comment
		sys	mem	
Clk	Required	1	1	OCP clock, fixed width
MAddr	addr addr_wdth	1 16 to 64	1 16 to 64	Transfer address
MCmd	None			Always present with a fixed width of 3 bits
MData	mdata data_wdth	1 16, 32, 64, 128, 256	1 16, 32, 64, 128, 256	Write data

OCP Signal Name	Enable Parameter(s)	Allowed Values for Ports		Comment
		sys	mem	
MDataValid	datahandshake	1	1	Write data valid Must allow datahandshake phases for writes
SData	sdata data_wdth	1 16, 32, 64, 128, 256	1 16, 32, 64, 128, 256	Read data
SResp	resp Fixed width	1 2	1 2	Transfer response

Table 6 Simple OCP Extensions

OCP Signal Name	Enable Parameter(s)	Allowed Values		Comment
		sys Port	mem Port	
MAddrSpace	addrspace addrspace_wdth	1 1–3	1 1	Address space Always used
MByteEn	byteen Derived width	0,1 data_wdth/8	0	Request Phase Byte Enable can be optionally enabled at sys interface. It is used only for supporting ReadEx when there is width conversion in the interconnect. The MemMax Scheduler does not use it internally. Data width must be a multiple of 8 Note that MByteEn is **not** supported at the mem OCP interface.
MDataByteEn	mdatabyteen Derived width	1 data_wdth/8	1 data_wdth/8	Datahandshake phase write byte enables are always used for writes
MReqInfo	reqinfo reqinfo_wdth	0,1 0–32	0,1 0–33[1]	mem and sys reqinfo configuration must be identical, except when page hit/miss information is used[1]

1 The value of `reqinfo_wdth` for the `mem` interface must be greater than the value of `reqinfo_wdth` for the `sys` interface when page hit/miss information is exported to the DRAM controller (as described in “Sending Page Hit/Miss Information”).

Table 7 OCP Burst Extensions

OCP Signal Name	Enabling Parameter(s)	Allowed Values for Port		Comment
		sys	mem	
MBlockHeight	mblockheight_wdth	2–6 (inclusive) ≤ addr_wdth	NA	Height of sys BLCK burst. Blockheight needs to be a power of 2.
MBlockStride	mblockstride_wdth	6–21 (inclusive) ≤ addr_wdth	NA	Stride of sys BLCK burst
MBurstLength	burstlength burstlength_width	1 2–6	1 1–5	Burst length
MBurstPrecise	burstprecise	0	0	Given burst length is precise
MBurstSeq	burstseq Fixed width	0,1	0,1	Burst address sequence. 0 if only INCR burst is supported. 1 if other burst sequences are also supported (refer to Table 13 for supported burst sequences).
MBurstSingleReq	burstsinglereq	0,1	1 Only SRMD bursts are sent	Single request/multiple data and precise multiple request/multiple data protocols
MDataLast	datalast	0,1	0,1	Last write data in burst
MReqLast	reqlast	0,1	0	Last request in burst
SRespLast	resplast	0,1	0,1	Last response in burst

Table 8 OCP Thread and Tag Extensions

OCP Signal Name	Enabling Parameter(s)	Allowed Values for Port		Comment
		sys	mem	
MDataTagID	Always available	1–tags	Locked to 1	Tags are **not** supported on the mem interface
MDataThreadID	threads>1 & datahandshaking Derived width	 $\log_2$(threads)	False NA	Write data thread identifier
MTagID	Always available	1–tags	Locked to 1	Tags are **not** supported on the mem interface

OCP Signal Name	Enabling Parameter(s)	Allowed Values for Port		Comment
		sys	mem	
MTagInOrder	taginorder	0,1	NA	Tags are **not** supported on the mem interface
MThreadBusy	mthreadbusy mthreadbusy_exact mthreadbusy_pipelined Derived width	1 1 0,1 thread	0 0 0 NA	Master thread busy exact on sys socket only
MThreadID	threads × tags > 1 Derived width	1–16 $\log_2$(threads)	False NA	Request thread identifier
SDataThreadBusy	sdatathreadbusy sdatathreadbusy_exact sdatathreadbusy_ pipelined Derived width	1 1 0 threads	1 1 0 threads	Slave write data thread busy exact protocol is used
STagID	Always available	1–tags	Locked to 1	Tags are **not** supported on the mem interface
STagInOrder	taginorder	0,1	NA	Tags are **not** supported on the mem interface
SThreadBusy	sthreadbusy sthreadbusy_exact sthreadbusy_pipelined Derived width	1 1 0 threads	1 1 0 threads	Slave request thread busy exact protocol is used
SThreadID	threads × tags > 1 Derived width	1–16 $\log_2$(threads)	Locked to 1 NA	Response thread identifier

Table 9 OCP Sideband Signals

OCP Signal Name	Enabling Parameter(s)	Allowed Values for Port		Comment
		sys	mem	
Control	control control_wdth	0,1 Up to 16	0,1 Up to 16	OCP core control information (sys), OCP system control information (mem)
EnableClk	enablclk	0,1	0,1[1]	Enable divided clock
MReset_n	mreset	0,1	0,1	Synchronous master reset
SError	serror	0	0	Not supported

OCP Signal Name	Enabling Parameter(s)	Allowed Values for Port		Comment
		sys	mem	
SFlag	sflag sflag_wdth[2]	0 NA	0,1 $2^{chip\_bits+2+k}$	Bank busy flags. k=value of bank8 when bank8 is RO. k=1 when bank8 is RW.
SReset_n	sreset	0,1	0,1	Synchronous slave reset
Status	status status_wdth	0,1 Up to 16	0,1 Up to 16	OCP core status information

1 When the parameter `mem_clock_async` is set to 0, the `enablclk` parameter must be identical on both `sys` and `mem` interfaces. When `mem_clock_async` is set to 1, `enableclk` for the `mem` interface must be set to 0.
2 The maximum value of `sflag_wdth` is 16.

Table 10 OCP Configuration Parameters

OCP Parameter Name	Allowed Values for Port		Comment
	sys	mem	
datahandshake	burstsinglereq	burstsinglereq	Enables decoupling between request and data phases.
endian	little, big, neutral, both	little, big, neutral, both	All endian types are supported. Because there is no width conversion between the sys and mem sockets, endianness has no effect. However, the configuration register accesses are always little-endian.
reqdata_together	0,1	0,1	Request and data phases of the first request together.
tag_interleave_size	0–16		Default value is 1. Note that dram_block_size and dram_blk_logsize settings must be configured to a value greater than or equal to the value of tag_interleave_size. See “DRAM Block Size”.
taginorder	0,1		Default value is 0.
tags	1–4096		Default value is 1.
writeresp_enable	0,1	0,1	Enables write responses. If enabled on mem socket must also be enabled on sys socket.

Bank Busy Flags

The MemMax Scheduler interprets the *SFlag* signal as a set of bank busy bits. The total width of *SFlag* is $2^{chip\_bits+2+k}$, where `chip_bits` is the number of chip bits

and *k* is equal to the value of the parameter `bank8`. That is, *SFlag* is interpreted as a set of 2`chip_bits` bitfields, with each bitfield containing either four or eight bits.

The 4-bit (or 8-bit) bitfield containing the bank busy flags for each chip is arranged in ascending order by bank number, with the bank busy bit for bank 0 as the right-most bit and the bank busy bit for bank 3 (or bank 7) as the left-most bit.

Within *SFlag*, the per-chip bank busy bits are arranged in ascending order by chip number, with the bank busy bits for chip 0 at the right-most end of SFlag and the bank busy bits for chip *N* at the left-most end.

Sending Page Hit/Miss Information

During scheduling, the MemMax Scheduler computes whether a request is targeted to a page miss or page hit. This information can be passed to the DRAM controller, which may allow the use of a smaller DRAM controller (through the elimination of the hardware required to perform the same computation at the controller).

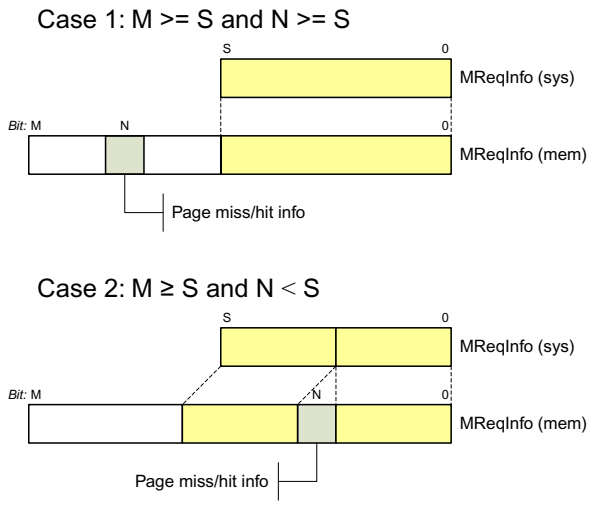
To enable passing the page hit/miss information to the DRAM controller, the `mem` interface must be configured with a single-bit subnet named `SEND_PAGE_MISS_INFO`. For example, adding this subnet statement to the `mem` interface configuration would cause bit 3 to be used for the page hit/miss information:

```
subnet MReqInfo 3 SEND_PAGE_MISS_INFO
```

The width of the *MReqInfo* signal (*M*) of the `mem` interface must be at least one greater than the width of the *MReqInfo* signal (*S*) of the sys interface and must be larger than the chosen page hit/miss information bit position, *N*. Figure 10 illustrates how the scheduler inserts the page hit/miss information bit into the *MReqInfo* signal of the `mem` interfaces. Note that the scheduler does not support the case where *S* is greater than *M*.

Important! The subnet statement and the additional bit of *MReqInfo* must be configured manually in the RTL configuration file prior to running soccomp because subnet configuration is not yet supported by SonicsStudio Director.

Figure 10 Position of Page Hit/Miss Information Bit in MReqInfo



Sending Close Page Information

The MemMax Scheduler can indicate on the output interface whether the memory controller should keep the page open, or close it after the access is performed. The memory controller uses this information to cause an access with auto-precharge.

To enable passing the page-close information to the DRAM controller, you must configure the *PAGEMODE* register. For more information, refer to the pagemode parameter and register definition in the “Page Mode” and “Bank Busy Support” sections of Chapter 5: RTL Configuration. In addition to the *PAGEMODE* register, you must configure the mem interface with a single-bit subnet named *SEND_CLOSE_PAGE_INFO*.

For example, adding this subnet statement to the mem interface configuration causes bit 4 to be used for the close-page information:

```
subnet MReqInfo 4 SEND_CLOSE_PAGE_INFO
```

Important! The subnet statement and the additional bit of *MReqInfo* must be configured manually in the RTL configuration file prior to running soccomp because subnet configuration is not yet supported by SonicsStudio Director.

OCP Command Support

The MemMax Scheduler supports a subset of the OCP 2.2 commands, summarized in Table 11, below.

Table 11 Supported OCP Release 2.2 Commands

OCP Command	Enable Parameter	Allowed Values	Request Type
Idle			(none)
Write	write_enable	1	write
Read	read_enable	1	read
ReadEx	readex_enable[1]	0,1	read
ReadLinked	rdlwrc_enable	0	Not supported
WriteNonPost	writenonpost_enable	0	
WriteConditional	rdlwrc_enable	0	
Broadcast	broadcast_enable	0	

1 Only supported on the sys OCP interface.

Burst Handling

The MemMax Scheduler supports the OCP burst types *Precise INCR*, *WRAP*, *BLCK*, and *XOR* as shown in Table 12. Besides the XOR burst, all other burst types are chopped into dram_block_size aligned addresses (2^*DRAM_BLK_LOGSIZE* * data width) on the mem interface side of the MemMax Scheduler. The data width is measured in bytes.

Table 12 OCP Burst Types

Burst Types			Socket Parameters		Socket Support	
Imprecise	**Precise**	**SRMD**	**burstprecise**	**burstsinglereq**	**sys**	**mem**
No	Yes	No	Tie-off 1	Tie-off 0	No	No
No	Yes	Yes	Tie-off 0	1	SRMD, precise MRMD	SRMD
Yes	No	No	Tie-off 0	Tie-off 0	NA	NA
Yes	No	Yes	Tie-off 0	Illegal	NA	NA
Yes	Yes	No	1	Tie-off 0	No	No
Yes	Yes	Yes	1	1	No	No

The scheduler supports the OCP burst sequences shown in Table 13.

Table 13 OCP Burst Sequences

OCP Burst Type	Enable Parameter	Allowed Values		Note
		sys	**mem**	
INCR	burstseq_incr_enable	1	1	The sys interface supports SRMD, precise MRMD. The mem interface supports only SRMD
DFLT1	burstseq_dflt1_enable	0	0	Not supported
WRAP	burstseq_wrap_enable	0,1	0,1	By default only read WRAP bursts are supported. To support write WRAP burst, set the internal parameter wrap_wr_support to 1.
DFLT2	burstseq_dflt2_enable	0	0	Not supported
XOR[1]	burstseq_xor_enable	0,1	0,1	The mem interface supports only SRMD.
STRM	burstseq_strm_enable	0	0	Not supported
UNKN	burstseq_unkn_enable	0	0	Not supported
BLCK	burstseq_blck_enable	0,1	0	The MemMax Scheduler only supports power of 2 blockheight BLCK burst.

1 XOR in the sys interface is supported only if the mem interface also supports XOR.

Burst alignment, described in Table 14, is supported at the sys or mem OCP sockets. The sockets must each support at least one kind of burst sequence.

The MemMax Scheduler translates MRMD/SRMD bursts of type INCR, WRAP, XOR, or BLCK that are input on the sys port by chopping each burst into an optimal sequence of SRMD sub-bursts of type INCR, WRAP, or XOR that are then output on the mem port. For burst requests that have been chopped, the MemMax

Scheduler performs all necessary transformations on the mem port OCP response stream before forwarding the response on the sys port.

Table 14 Burst Alignment

OCP Parameter	Allowed Values		Function
	sys	mem	
burst_aligned	0,1	0,1	When set to 1, INCR bursts must have a burst length that is a power of two, and a starting address that is aligned to the total burst size.

Burst translation is determined using a combination of run-time and design-time parameters.

- Run-time configuration parameters must be programmed to exactly reflect the configuration of the downstream SDRAM and consist of the following:

  dram_burst_type
  Value corresponds to either sequential or interleaved

  dram_type
  Value corresponds to either DDR1, DDR2, or DDR3

  dram_blk_logsize
  The value N in this field corresponds to a DRAM block size of 2^N.

- Design-time configuration parameters used for burst translation are the mem port OCP burst enable parameters listed in Table 13 (burstseq_incr_enable, burstseq_wrap_enable, and burstseq_xor_enable), and the burst_aligned parameter of the mem port.

For a descriptions of the MemMax Scheduler configuration parameters, see Chapter 5: RTL Configuration.

Table 15, Table 16, and Table 17 summarize the burst translation algorithms used by the MemMax Scheduler for each valid combination of burst type. Each algorithm is described after the tables.

Table 15 MemMax Burst Conversion (Write)

Sys Burst Type	DRAM Parameters	Mem OCP Burst Sequence Support							
					Aligned				
	Block Size[1] Burst Type DRAM Type	INCR & XOR	INCR & WRAP (& XOR)[2]	INCR	INCR	INCR & XOR	INCR & WRAP (& XOR)[4]	WRAP (& XOR)[3]	XOR
INCR	2 Any Any	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned WRAP	aligned XOR
INCR	4 Sequential DDR1	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned WRAP	aligned XOR

Sys Burst Type	DRAM Parameters	Mem OCP Burst Sequence Support							
					Aligned				
	Block Size[1] Burst Type DRAM Type	INCR & XOR	INCR & WRAP (& XOR)[2]	INCR	INCR	INCR & XOR	INCR & WRAP (& XOR)[4]	WRAP (& XOR)[3]	XOR
INCR	4 Interleaved DDR1	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned WRAP	aligned XOR
INCR	4 ANY DDR2/DDR3	aligned INCR	aligned INCR	aligned NCR	aligned INCR	aligned INCR	aligned INCR	aligned WRAP	aligned XOR
INCR	8 or 16 ANY ANY	INCR	INCR	INCR	Not supported	Not supported	Not supported	Not supported	Not supported
WRAP	2 ANY ANY	aligned INCR[5]	aligned INCR[5]	aligned INCR[5]	aligned INCR[5]	aligned INCR[5]	aligned INCR[5]	N/A	N/A
WRAP	4[6] ANY ANY	aligned INCR[5]	aligned INCR[5]	aligned INCR[5]	aligned INCR[5]	aligned INCR[5]	aligned INCR[5]	N/A	N/A
WRAP	4[6] Sequential DDR1	aligned INCR[5]	aligned INCR[5]	aligned INCR[5]	aligned INCR[5]	aligned INCR[5]	aligned INCR[5]	N/A	N/A
WRAP	4 Interleaved DDR1	aligned INCR[5]	aligned INCR[5]	aligned INCR[5]	Only-0 INCR[5]	Only-0 INCR[5]	Only-0 INCR[5]	N/A	N/A
WRAP	4 ANY DDR2	aligned INCR[5]	aligned INCR[5]	aligned INCR[5]	aligned INCR[5]	aligned INCR[5]	aligned INCR[5]	N/A	N/A
WRAP	4 ANY DDR3	aligned INCR[5]	aligned INCR[5]	aligned INCR[5]	aligned INCR[5]	aligned INCR[5]	aligned INCR[5]	N/A	N/A
WRAP	8 or 16 ANY ANY	INCR[5]	INCR[5]	INCR[5]	Not supported	Not supported	Not supported	Not supported	Not supported
XOR	2 ANY ANY	XOR	XOR	Not supported	Not supported	XOR	XOR	XOR	XOR
XOR	4 ANY DDR2/DDR3	XOR	XOR	Not supported	Not supported	XOR	XOR	XOR	XOR
XOR	4 Sequential DDR1	XOR2INCR	XOR2WRAP	Not supported	Not supported	XOR2INCR	XOR2WRAP	XOR2WRAP	XOR2INCR

1 OCP words
2 For INCR & WRAP & XOR, the burst translation algorithms are the same as INCR & WRAP.
3 For WRAP & XOR, the burst translation algorithms are the same as WRAP.

4 For Aligned INCR, WRAP, and XOR, the burst translation algorithms are the same as Aligned INCR and WRAP.
5 Wrap_wr_support needs to be enabled.
6 For *MBurstLength* equal to 2.

Table 16 MemMax Burst Conversion (Read, optimize_wrap_rd_for_timing_and_area=0)

Sys Burst Type	DRAM Parameters	Mem OCP Burst Sequence Support							
					Aligned				
	Block Size[1] Burst Type DRAM Type	INCR & XOR	INCR & WRAP (& XOR)[2]	INCR	INCR	INCR & XOR	INCR & WRAP (& XOR)[4]	WRAP (& XOR)[3]	XOR
INCR	2 Any Any	INCR	INCR	INCR	aligned INCR	aligned INCR	aligned INCR	aligned WRAP	aligned XOR
INCR	4 Sequential DDR1	INCR	INCR	INCR	aligned INCR	aligned INCR	aligned INCR	aligned WRAP	aligned XOR
INCR	4 Interleaved DDR1	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned WRAP	aligned XOR
INCR	4 ANY DDR2/DDR3	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned WRAP	aligned XOR
INCR	8 or 16 ANY ANY	INCR	INCR	NCR	Not Supported	Not Supported	Not Supported	Not Supported	Not Supported
WRAP	2 ANY ANY	XOR	WRAP	Only-0 INCR	Only-0 INCR	XOR	WRAP	WRAP	XOR
WRAP	4, MBurstLength =2 ANY ANY	Even XOR	Even WRAP	Even INCR	Even INCR	Even XOR	Even WRAP	Even WRAP	Even XOR
WRAP	4[5] Sequential DDR1	Even XOR	Even WRAP	Even INCR	Even INCR	Even XOR	Even WRAP	Even WRAP	Even XOR
WRAP	4 Interleaved DDR1	Even XOR	Even WRAP	Even INCR	Even INCR	Even XOR	Even WRAP	Even WRAP	Even XOR
WRAP	4 ANY DDR2	Even XOR	Even WRAP	Even INCR	Even INCR	Even XOR	Even WRAP	Even WRAP	Even XOR
WRAP	4 ANY DDR3	Even XOR	Even WRAP	Only-0 INCR	Only-0 INCR	Even XOR	Even WRAP	Even WRAP	Even XOR

Sys Burst Type	DRAM Parameters	Mem OCP Burst Sequence Support							
					Aligned				
	Block Size[1] Burst Type DRAM Type	INCR & XOR	INCR & WRAP (& XOR)[2]	INCR	INCR	INCR & XOR	INCR & WRAP (& XOR)[4]	WRAP (& XOR)[3]	XOR
WRAP	8 or 16 ANY ANY	Even XOR	WRAP	Only-0 INCR	Not supported	Not supported	Not supported	Not supported	Not supported
XOR	2 ANY ANY	XOR	XOR	Not supported	Not supported	XOR	XOR	XOR	XOR
XOR	4 ANY DDR2/DDR3	XOR	XOR	Not supported	Not supported	XOR	XOR	XOR	XOR
XOR	4 Sequential DDR1	XOR2INCR	XOR2WRAP	Not supported	Not supported	XOR2INCR	XOR2WRAP	XOR2WRAP	XOR2INCR
XOR	4 or 16 ANY ANY	Not supported	Not supported	Not supported	Not supported	Not supported	Not supported	Not supported	Not supported

1 OCP words
2 For INCR &WRAP & XOR the burst translation algorithms are the same as INCR &WRAP.
3 For WRAP & XOR the burst translation algorithms are the same as WRAP.
4 For Aligned INCR, WRAP, and XOR the burst translation algorithms are the same as Aligned INCR and WRAP.
5 For *MBurstLength* equal to 2.

Table 17 MemMax Burst Conversion (Read, optimize_wrap_rd_for_timing_and_area=1)

Sys Burst Type	DRAM Parameters	Mem OCP Burst Sequence Support							
					Aligned				
	Block Size[1] Burst Type DRAM Type	INCR & XOR	INCR & WRAP (& XOR)[2]	INCR	INCR	INCR & XOR	INCR & WRAP (& XOR)[4]	WRAP (& XOR)[3]	XOR
INCR	2 Any Any	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	Not supported	Not supported
INCR	4 Sequential DDR1	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	Not supported	Not supported
INCR	4 Interleaved DDR1	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	Not supported	Not supported

Sys Burst Type	DRAM Parameters	Mem OCP Burst Sequence Support							
					Aligned				
	Block Size[1] Burst Type DRAM Type	INCR & XOR	INCR & WRAP (& XOR)[2]	INCR	INCR	INCR & XOR	INCR & WRAP (& XOR)[4]	WRAP (& XOR)[3]	XOR
INCR	4 ANY DDR2/DDR3	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	Not supported	Not supported
INCR	8 or 16 ANY ANY	INCR	INCR	NCR	Not supported	Not supported	Not supported	Not supported	Not supported
WRAP	2 ANY ANY	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	Not supported	Not supported
WRAP	4, MBurstLength =2 ANY ANY	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	Not supported	Not supported
WRAP	4[5] Sequential DDR1	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	Not supported	Not supported
WRAP	4 Interleaved DDR1	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	Not supported	Not supported
WRAP	4 ANY DDR2	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	Not supported	Not supported
WRAP	4 ANY DDR3	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	aligned INCR	Not supported	Not supported
WRAP	8 or 16 ANY ANY	INCR	WRAP	INCR	Not supported	Not supported	Not supported	Not supported	Not supported
XOR	2 ANY ANY	Not supported	Not supported	Not supported	Not supported	Not supported	Not supported	Not supported	Not supported
XOR	4 ANY DDR2/DDR3	Not supported	Not supported	Not supported	Not supported	Not supported	Not supported	Not supported	Not supported
XOR	4 Sequential DDR1	Not supported	Not supported	Not supported	Not supported	Not supported	Not supported	Not supported	Not supported
XOR	8 or 16 ANY ANY	Not supported	Not supported	Not supported	Not supported	Not supported	Not supported	Not supported	Not supported

1 OCP words

2 For INCR &WRAP & XOR the burst translation algorithms are the same as INCR &WRAP.
3 For WRAP & XOR the burst translation algorithms are the same as WRAP.
4 For Aligned INCR, WRAP, and XOR the burst translation algorithms are the same as Aligned INCR and WRAP.
5 For *MBurstLength* equal to 2.

Table 18 Legend for Table 15, Table 16, and Table 17

Straight	MRMD/SRMD request bursts are chopped at DRAM block boundaries and forwarded to the DRAM controller as SRMD bursts of the specified type. The MemMax Scheduler forwards the corresponding response stream from the mem port to the sys port without further transformation.
Aligned INCR	MRMD/SRMD request bursts are chopped at DRAM block boundaries. The scheduler creates a series of aligned SRMD INCR sub-bursts that are forwarded to the DRAM controller. Each aligned INCR burst has a burst length that is a power-of-two and has a starting address that is aligned to the burst length. The scheduler adjusts the starting address and the burst length of the first and last SRMD INCR sub-bursts. For read requests, the scheduler drops any extra response words that are not from the original read request from the response returned via the sys OCP port. For transformed write requests, the scheduler provides extra data words (with appropriate byte enable settings) to the DRAM controller.
Aligned WRAP/XOR	Aligned WRAP/XOR burst requests are transformed in the same manner as aligned INCR bursts. The scheduler uses WRAP or XOR bursts in the SRMD sub-bursts instead of INCR bursts, as appropriate.
Only-0 INCR	The scheduler aligns the starting address of the requested WRAP read burst to the next lowest address aligned to the DRAM block size, then generates a series of SRMD INCR type sub-bursts aligned to DRAM block size boundaries. The scheduler stores any response words that are received first (and hence correspond to data from addresses below the starting address of the original request), and forwards them to the requestor after forwarding the response data corresponding to the addresses of the original request.
Even	Before chopping the sys WRAP burst at DRAM block boundaries, the scheduler aligns the starting address to the next lowest address with an even offset within the DRAM block boundary. If the first and last sub-bursts are shorter than the intermediate sub-bursts, they would be accessing the same DRAM block; the first sub-burst accessing the second half of the DRAM block and the last sub-burst accessing the first half of it. • If *even WRAP* or *even XOR* translation is used, the scheduler combines the first and last sub-bursts into a single WRAP or XOR SRMD burst with a length equal to the DRAM block size. • If *even INCR* translation is used, the scheduler sends the first and last sub-bursts back-to-back as INCR SRMD bursts, each with a length that is half of the DRAM block size. All remaining sub-bursts, with a length equal to the DRAM block size are sent to the mem OCP interface with the appropriate burst sequence type. The scheduler stores any response words that are below the starting address and are received first, and forwards them with the sys OCP after forwarding all response words that are at or above the starting address.

XOR2WRAP	XOR2WRP translation is employed when the mem OCP interface supports WRAP bursts.
XOR2INCR	XOR2INCT translation is the same as XOR2Even XOR.

Table 19 Supported Burst Configurations

				DRAM Controller Burst Support	
Burst Request Type	**DRAM Block Size (OCP Words)**	**DRAM Burst Type**	**DRAM Type**	**INCR**	**Aligned INCR**
INCR	4	ANY	ANY	INCR	Aligned INCR
WRAP	4	ANY	ANY	Only-0 INCR	Only-0 INCR

Timing

When the MemMax Scheduler is configured for synchronous operation, the scheduler runs at the frequency provided to it by the clock signal of the sys OCP interface. When configured to use asynchronous buffers, the sys and `mem` OCP interfaces can run at independent frequencies. The maximum frequency varies with the chip process technology and physical layout.

The sys OCP interface conforms to the Level2 timing guidelines described in the *Open Core Protocol Specification, Release 2.2*.

For the `mem` OCP interface linking the scheduler to the controller, all of the cycle time is allocated to the scheduler. All signals into the controller must be registered immediately and all signals from the controller must come straight out of registers without any intervening logic.

When `resp_bypass_enable` is set, read data is passed directly from the `mem` OCP interface to the sys OCP interface or is read from SRAM, bypassing the internal read data buffer and reducing the response latency. Write responses may also be bypassed. However, read data buffer bypassing can only occur when all read data buffers are empty, that is, when the response does not need to be arbitrated against responses for other threads. Also, the read data buffer cannot be bypassed if *MThreadbusy* signal on the sys interface is asserted.

Note: Response bypassing is not supported when `mem_clock_async=1`, or when the `mem` interface has `writeresp_enable=0` and the sys interface has `writeresp_enable=1`.

The SRAMs inside the read and write buffers are assumed to have a register to latch the address and write data and produce read data sequentially in the next cycle. The upper limit on SRAM access time is determined by the response group timing for OCP Level2 timing, which is 60% of the cycle time. For example, given a scheduler operating frequency of 333 MHz, the worst-case SRAM access time must be no more than 1.8 ns.

System Reset

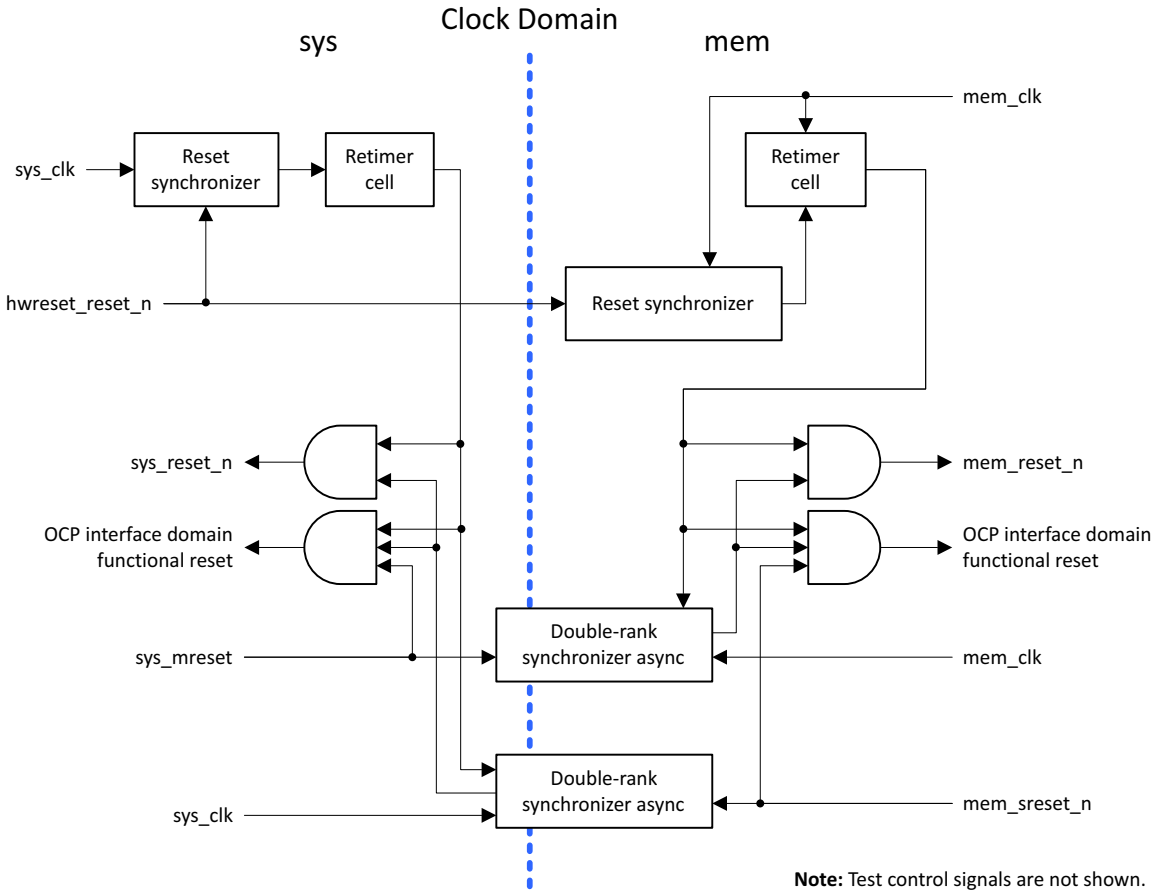
The MemMax Scheduler takes a single hardware reset *hwreset_reset_n* on its system interface. The parameter `mem_clock_async` determines how the hardware reset is conditioned.

- When `mem_clock_async` is set, the hardware reset is synchronized to the sys and `mem` clocks inside the MemMax instance as described in “Internal Reset Synchronization”.
- If `mem_clock_async` is not set, the synchronization of the hardware reset signal must be external to the MemMax instance as described in “External Reset Synchronization”.

Internal Reset Synchronization

Figure 11 shows the reset structure for the scheduler when the `mem` OCP interface is in a separate clock domain to the sys OCP interface.

Figure 11 Reset Synchronization with retimed_reset=1 and mem_clock_async=1



When the MemMax Scheduler has an internal asynchronous clock domain crossing as shown in Figure 11, the scheduler must create a version of the *hwreset_reset_n* signal for use by the logic that operates on the `mem` OCP clock. Once the *hwreset_reset_n* signal is available to the `mem` OCP clock domain, it is

used in the generation of the *mem_reset_n* signal, if enabled, and for the hardware and functional resets needed by internal registers in the `mem` clock domain.

Once the *hwreset_reset_n* signal is a available to the sys OCP clock domain, it is used in the generation of the *sys_reset_n* signal, if enabled, and for the hardware and functional resets needed by internal registers in the sys clock domain.

The style of reset synchronizer tactical cells instantiated automatically inside the scheduler depends on the `asyncreset_enable` parameter. If `asyncreset_enable` is set, then `sonics_reset_synchornizer_async` tactical cells will be used. If `asyncreset_enable` is not set, the `sonics_reset_synchronizer_sync` tactical cell will be instantiated.

External Reset Synchronization

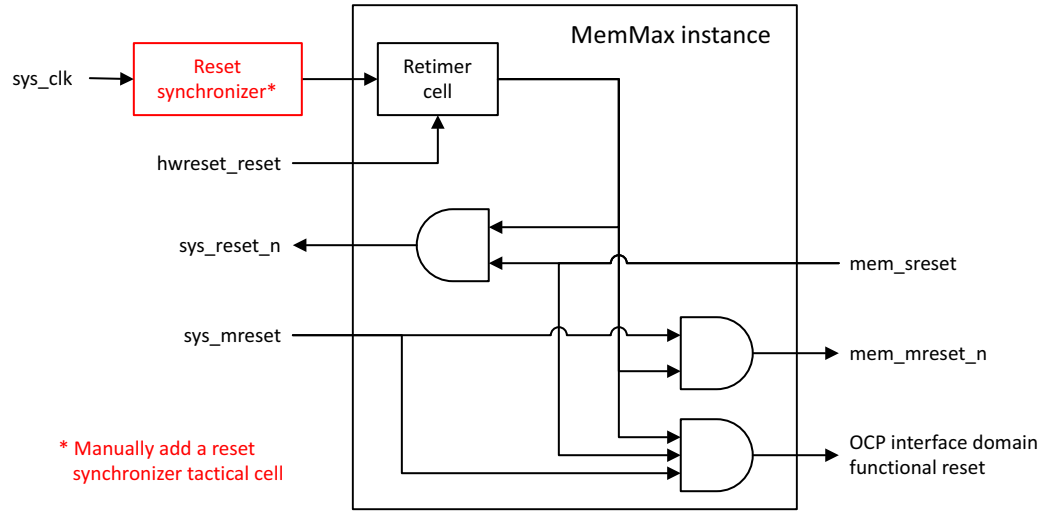
Figure 12 shows the reset structure for the scheduler when the `mem` and sys OCP interfaces are in the same clock domain.

Important! When `mem_clock_async=0` (the sys clock and the `mem` clock are synchronous), the logic is simplified. However, be sure to manually add a reset synchronizer as shown in Figure 12.

Examine the `asyncreset_enable` parameter to determine what style of reset synchronizer needs to be instantiated manually. If `asyncreset_enable` is set, the `sonics_reset_synchronizer_async` tactical cell should be used. If asyncreset_enable is not set, the `sonics_reset_synchronizer_sync` tactical cell should be used. See the *SoC Integration Guide* for more information about reset synchronizer tactical cells. For further description of the `asyncreset_enable` parameter, see “Asynchronous Reset”.

The *hwreset_reset_n* signal is used in the generation of the *mem_reset_n* signal, if enabled, and for the hardware and functional resets needed by internal registers on the `mem` side. The *hwreset_reset_n* signal is also used in the generation of the *sys_reset_n* signal, if enabled, and for the hardware and functional resets needed by the sys side.

Figure 12 Reset Synchronization with retimed_reset=1 and mem_clock_async=0



Clock Gating

MemMax uses the same approach to clock gating that is used in other Sonics hardware. This uses two levels of clock gating latch, one for fine-grained, leaf-node clock gating and one for coarse-grained clock gating. These tactical cells and their implications on interactions with test flow are described in the *SoC Integration Guide*.

Divided Clocks

The MemMax Scheduler supports clock division at its sys and `mem` interface by means of the OCP *EnableClk* signal (defined in the *Open Core Protocol Specification, Release 2.2*). Clock division at the two interfaces is performed as follows:

Case 1

When the `mem_clock_async` parameter is set to 1, the MemMax Scheduler performs asynchronous clock boundary crossing between its sys and `mem` interfaces. Under this condition, *enableClk* at the `mem` interface must be 0, and *enableClk* at the sys interface can be 0 or 1.

Case 2:

When the `mem_clock_async` parameter is set to 0, the MemMax Scheduler uses a single clock, which is provided by the clock signal of the sys interface. In this case, the values of `enableclk` at the sys and `mem` OCP interfaces must be identical.

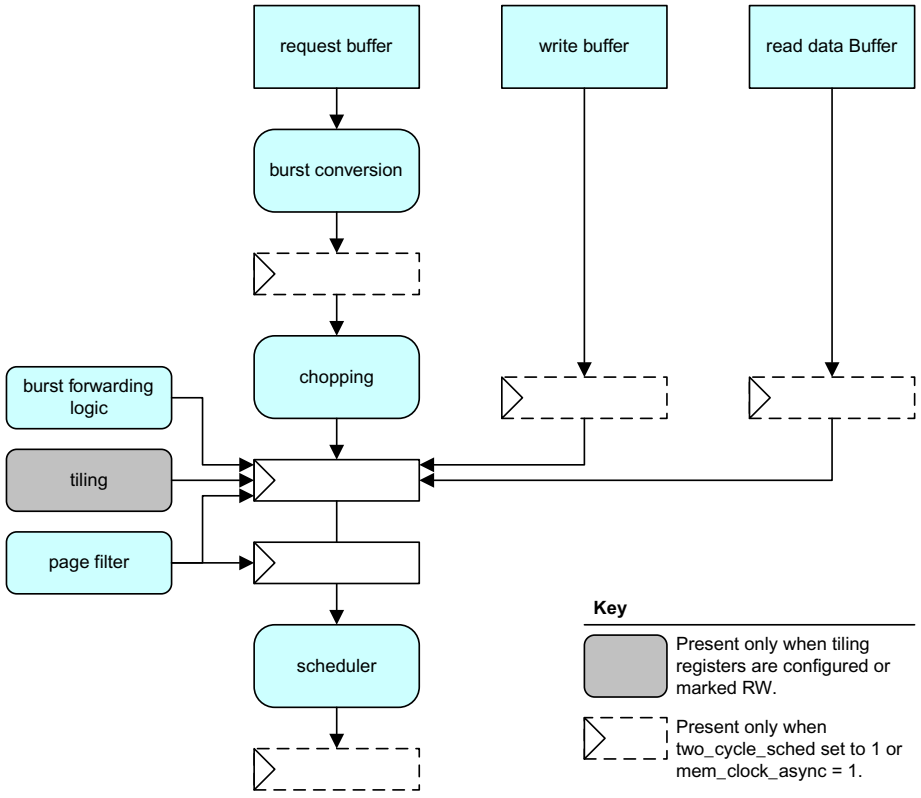
4 Internal Organization

This chapter discusses the internal organization of the MemMax Scheduler. Topics include the organization of the scheduler, the data processing flow, and thread handling mechanisms.

Block Organization

Figure 13 shows a block diagram of the MemMax Scheduler.

Figure 13 MemMax Scheduler Block Diagram



The major blocks and their functions are: as follows

Request Buffer
: Stores incoming OCP commands and chops them into short OCP bursts.

Write Buffer
: Holds write data on its way from the interconnect to the DRAM controller.

Read Data Buffer
: Holds read data and write responses in transit from the DRAM controller to the interconnect.

Address Tiler
: Transforms the request address pattern to generate a DRAM address pattern that is more efficient.

Tag and Thread Scheduler
: Arbitrates among requests presented by different request buffer threads. Schedules for DRAM efficiency and thread quality of service.

Data Flow

This section describes the flow of data through the scheduler.

Note: The implementation and effect on scheduling due to factors such as address overlap, a request's *MTagInOrder* field, QoS settings, and the value of `tag_interleave_size` has been omitted from the discussion below. Each of these factors—and others not listed here—are considered by the scheduler in each scheduling cycle, but do not change the generalized description of data flow presented here.

Read Requests

The progress of a read through the scheduler is as follows:

1. A read request arrives at the *sys* interface and is placed in the request buffer along with the thread number and tag ID.
2. A request at the head of the per-thread command queue is chopped into sub-burst requests as described in "Burst Handling".
3. Storage in the read data buffer is pre-allocated to ensure that storage is available when read data is returned. When the preallocation succeeds, the SRMD read burst is presented to the thread scheduler.
4. The tag and thread scheduler decides the order in which chopped bursts from different tags and threads are presented to the DRAM controller via the *mem* OCP interface.
5. When the DRAM controller has fetched the read data, it is either passed directly to the *sys* interface (see "Response Bypass"), or placed into the pre-allocated entries in the read data buffer.
6. The MemMax Scheduler performs any necessary reordering on the stored read data before forwarding them in order to the requestor via the *sys* OCP interface.

Write Requests

The progress of a write request through the scheduler is as follows:

1. A write request arrives via the *sys* OCP interface, and the command is placed in the request buffer.
2. At the same time, space is allocated in the write buffer to hold the write data and byte enables.
3. As write data arrives from the interconnect, it is placed into the write buffer. Once all of the write data has arrived, the write buffer informs the request buffer to allow the request to be scheduled.
4. When a write request arrives at the head of the request buffer queue, it is chopped into a series of sub-burst write requests as described in “Burst Handling”.
5. If the *sys* socket is configured with `writeresp_enable`, smaller requests pre-allocate any required resources in the read data buffer to ensure that the correct number of write responses are returned.
6. When there is sufficient data in the write buffer for the write burst and the read data buffer can handle any corresponding write responses, the write burst is transferred to the thread scheduler.
7. The tag and thread scheduler decides the order in which chopped bursts from different tags and threads are presented to the DRAM controller.
8. The write responses for all write requests associated with the chopped bursts are passed to the read data buffer. Each response includes information on how many times it must be reproduced at the *sys* socket.
   - If the *mem* interface has `writeresp_enable` set, only the write response of the *last* write request associated with a *sys* SRMD write burst is passed to the read data buffer.
   - If the *sys* interface is configured with `writeresp_enable` and the *mem* interface is not, then there will be no write responses available from the *mem* interface to be passed to the read data buffer. Instead the MemMax Scheduler response logic internally generates the write responses at the *sys* socket as smaller write requests are accepted at the *mem* socket
9. When the *sys* socket is configured with `writeresp_enable`, the responses associated with a write burst are returned on the *sys* interface once responses for any preceding request on the same thread are returned.
10. The scheduler ignores write responses associated with extra data words that it supplies to the DRAM controller when following the burst translation algorithms discussed in “Burst Handling”.

Request and Data Buffers

The amount of scheduler buffering is based on the number of tags and threads, the user-specified buffer depths, and the OCP data width. There are three buffers:

- Request Buffer
- Write Data Buffer

- Read Data Buffer

The buffers allow traffic to be queued, which provides decoupling between the *sys* and *mem* interfaces. Larger buffers allow the queueing of more traffic. When more traffic is queued, the scheduler has a wider range of choices, which can potentially result in improved performance.

Request Buffer

The request buffer receives commands from the interconnect and processes them for presentation to the thread scheduler. There is a separate request buffer structure for each individual incoming OCP thread. The request buffer is essentially an in-order queue. OCP commands of the request buffer's thread enter at the tail of the command queue and exit to the thread scheduler at the head of the command queue.

Write Data Buffer

The write data buffer holds write data from the time it appears on the *sys* OCP interface until it is passed to the DRAM controller. There is a single shared embedded SRAM for all threads, but the control structures are separated on a per-thread basis. For RAM compilation, the write buffer SRAM appears as a dual-ported, directly-addressed structure that is the width of the controller's data word plus the number of data byte enables when `data_wdth` > 32 bits. The width is the same as the controller's data word for `data_wdth` < 32 bits. The width is calculated from the OCP `data_wdth` parameter.

Read Data Buffer

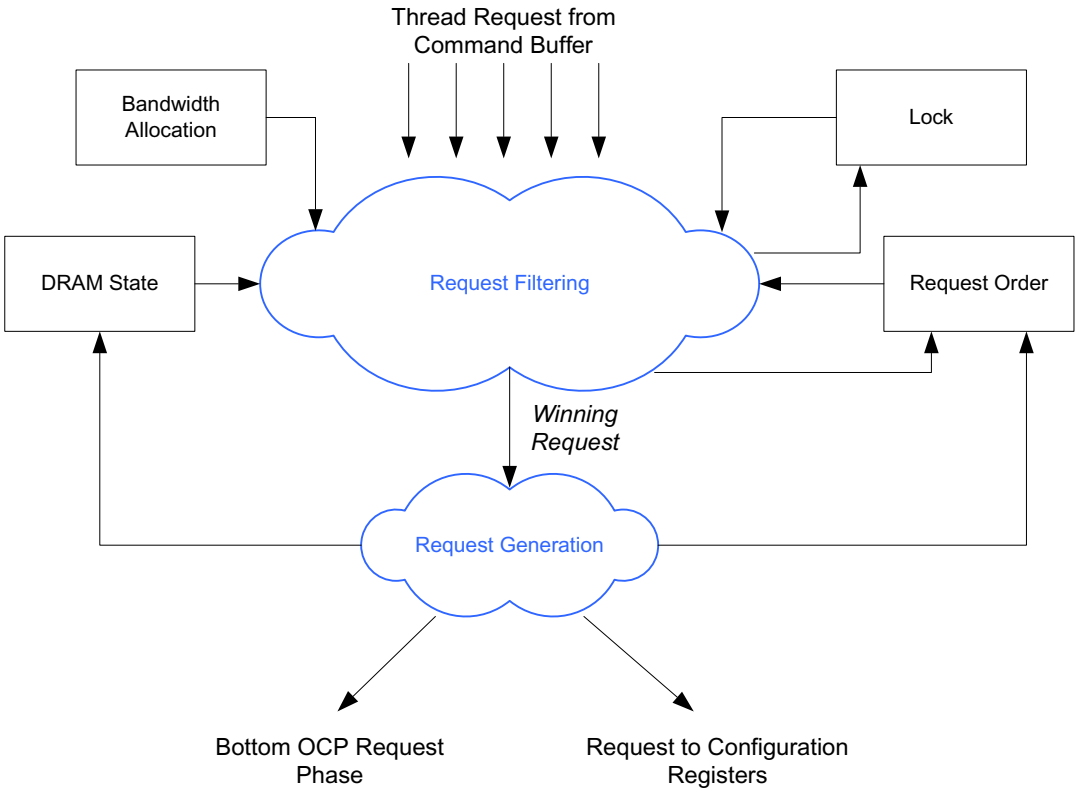
The read data buffer holds either read data or write response information between the time it arrives at the *mem* OCP interface until it is delivered to the *sys* OCP interface to the interconnect. There is a single shared embedded SRAM for all threads, but the control structures are separated on a per-thread basis. For the purposes of RAM compilation the read data buffer SRAM looks like a dual-ported, directly addressed structure that is the width of the controller's data word. The width is indicated by the OCP `data_wdth` parameter.

Tag and Thread Scheduling

Tag and thread scheduling maximizes overall DRAM efficiency and provides for different quality of service levels to satisfy the demands of the different threads.

The tag and thread scheduler block is responsible for performing arbitration of the multiple thread requests (which may also be tagged), arriving from the request buffer. The tag and thread scheduler block generates a single-threaded OCP request stream to be sent to the DRAM controller. A block diagram showing the organization of the tag and thread scheduler block is shown in Figure 14.

Figure 14 Scheduler Block Diagram



The thread scheduler block functions are:

- Request filtering block

  Performs arbitration for the *mem* OCP interface to the controller

- DRAM state function

  Tracks the state of the DRAM banks

- Bandwidth allocation block

  Tracks whether a thread is within its allocated bandwidth

- Request order block

  Tracks the order in which requests are presented to the scheduler

- Request generation block

  Generates the OCP request that is presented to the *mem* OCP interface or to the internal configuration registers

- Lock/unlock block

  The lock address issued by a ReadEx request is remembered and prevents any other threads from accessing the locked address. A write request following a ReadEx request unlocks the address. When an address is locked by a ReadEx request, any new ReadEx requests on other threads are also blocked.

5 RTL Configuration

Configuration parameters affect the instantiation and operation of the MemMax Scheduler. Design-time, or *instance*, parameters are specified using statements in an RTL configuration file and affect the instantiation of the MemMax Scheduler, for example, controlling the number of threads supported.

Run-time configuration parameters—such as the QoS level for a specific thread—are specified using bitfields in software-accessible configuration registers. Run-time parameters are a subset of the design-time parameters: not all design-time parameters may be modified at run-time, and the initial (that is, reset) values of the bitfields are taken from the specified values (or defaults) of the corresponding design-time parameters.

This chapter first presents the syntax used to specify instance parameters in the section “RTL Configuration File Syntax”. Each instance parameter is discussed in “Instance Parameters for the MemMax Scheduler”. Instance parameters are presented first for the scheduler and then for the DRAM controller. The set of registers used for run-time configuration are presented in “MemMax Registers”.

RTL Configuration File Syntax

Within a design RTL configuration file, connections and external interfaces are *always* placed ahead of all other constructs. Instances of other modules generally reflect the order in which they were created in SonicsStudio Director. The ordering of statements *within* an instance, interface, or connection is, however, determined by SonicsStudio Director, and this order will be preserved for a given revision of the tool.

The design RTL configuration file is written using standard Tcl syntax. Indentation is not important and statements are delimited by line endings. Unlike some languages, a semi-colon is not used to delimit statements. Conventions should be read outside-in and left to right.

Syntax is described using the following conventions:

Syntax	Meaning
[ ]	Square brackets indicate an optional argument or statement.
< >	Less than and greater than symbols delimit an argument or argument list with the default underlined.
( )	Parentheses are used to indicate grouping.
|	An or symbol marks an alternative, a logical *OR* operation.
*	An asterisk symbol represents *zero* or more repetitions.
+	The plus symbol represents *one* or more repetitions.
?	A question mark indicates zero instances or one instance.
\	A backslash at the end of a line indicates line continuation.
{ }	Curly brackets indicate where something starts and stops, are part of the format, and are required.
#	The hash symbol at the beginning of a line indicates a comment.

Example 1

Here is an example specification for the `reg_depth` statement that uses some of the syntax components defined above.

```
[ reg_depth (request|response) {(uniform <depth>|per_thread <depth>+)} ]?
```

The syntax specifies the following:

- The entire statement is optional, as indicated by the enclosing square brackets, and no more than one unique instance should be given as indicated by the use of the final question mark (`?`).
- One of the keywords `request` or `response` must be given, as indicated by the grouping parentheses and the vertical bar.
- One of the keywords `uniform` or `per_thread` must be given, and they differ in their arguments.
- The `uniform` keyword takes one argument, `depth`.
- The `per_thread` keyword is followed by an argument list. The use of a plus sign (`+`) indicates the argument list must have at least one element. Argument lists are space separated and must fit on one line.

The following examples are legal according to the syntax definition above:

```
reg_depth response per_thread 2 2 10 8
reg_depth request  uniform 3
```

Two data types are supported: integers and strings. Integers may be written in decimal or hexadecimal; for hexadecimal numbers use an 0x prefix, for example, 0x400. Strings may be written in upper or lower case and may contain decimal digits, underscores, and periods—no other punctuation is allowed. Strings may optionally be surrounded by quotes. For example, the following are legal integer and string data types:

```
0x1000              16384
RT.addr_map         SMS_reg
```

Example 2

Here is an example specification for the SonicsGN `regblock` statement that uses some of the syntax components defined earlier. SonicsExpress users can skip this example because SonicsExpress does not contain registers.

```
[ regblock <reg_block_type> {<register>*} ]
```

Where:

```
<reg_block_type> :: ia|ta|pm|rt

<register> :: register <reg_name> [{<regfield>*}]

  <reg_name> :: <name> | <name>(<index>)

  <regfield> :: regfield <regfield_name> [<param_value>] [{<regfield_body>*}]

    <regfield_body> :: access (ro|rw) | export (int_const|ext_const|ext_prog)
```

The syntax specifies that:

- The entire statement is optional, as indicated by the square brackets.
- One of the keywords `ia`, `ta`, `pm`, or `rt` must follow the word `regblock`.
- After the keyword, use braces to enclose the register definitions.
- One or more register definitions can be given, as indicated by the asterisk.
- Each register definition begins with the word `register` followed by a register name (for example, `agent_status`) or a register name with an index (for example, `addr_match(5)`).
- After the register name, the `regfield` statement is optional, as indicated by the square brackets. If present, they must be surrounded by curly braces.
- If present, multiple `regfield` statements can be included inside the braces, as indicated by the asterisk next to `<regfield>`.
- Each `regfield` starts with the key word `regfield` and is followed by a required name, and two optional parts: an (initial) value for the regfield, and the regfield body.
- When present, the `param_value` indicates the value held by the register.
- When present, the `regfield_body` must be surrounded by braces, and one or more regfield_body commands can be enclosed.
- Each `regfield_body` must start with one of two key words: `access`, or `export`.
  - When the keyword `access` is used, it must be followed by either `ro` or `rw`.
  - When the keyword `export` is used, it must be followed by one of `int_const`, `ext_const`, or `ext_prog`.

The example below is legal according to the above syntax definition:

```
regblock pm {
    register control {
        regfield ERROR_PRIMARY_REP 1 {access rw}
```

```
        }
        register addr_match(2) {
            regfield BASE_ADDR 0 {access rw}
            regfield LEVEL 1 {access rw}
            regfield SIZE 6 {access rw}
        }
    }
```

Parameter Summary

This sections provides a summary of the configuration parameters that control the behavior of the scheduler. Some parameters are derived from the MemMax Controller. These parameters are summarized “Parameters Derived from the DRAM Controller”. All other parameters are summarized in “Instance Parameters”

Instance Parameters

This sections lists the RTL configuration parameters used to control the behavior of the scheduler. Some configuration parameters have a corresponding register field. Other parameters, you can only specify in the RTL configuration file and have no equivalent register field.

RTL configuration parameters that have a corresponding configuration register field are listed in Table 20. Table 21 lists parameters that do **not** have a corresponding register field. For a description of these parameters, see “Instance Parameters for the MemMax Scheduler”.

Table 20 MemMax Run-Time Configuration Parameters/Register Fields

Run-Time Parameter/ Register Field	Description
bandwidth_rate<n>	Bandwidth allocation rate (per thread <n>)
bankbusy_tread	The number of **mem** clock cycles after a read during which a miss will not be issued.
bankbusy_twrite	The number of **mem** clock cycles after a write during which a miss will not be issued.
clock_gate_disable	Disables fine-grained clock gating
disable_bit_move_f<n>	Unused, or not user-visible?
dram_blk_logsize	Log_2 of DRAM block size
high_en_f<n>	Bit mask with BAOP.width significant bits
high_loc_f<n>	Bit position of the LSB of the high OP bits
iloc<n>_f<m>	Bit position of <n>th segment LSB
len<n>_f<m>	Number of bits in <n>th segment
low_en_f<n>	Bit mask with BAOP.width significant bits.
low_loc_f<n>	Bit position of the LSB of the low OP bits.

Run-Time Parameter/ Register Field	Description
page_policy	DRAM page scheduling policy
pagemode<n>	Operation in open (0) or closed (1) page mode per thread<n>.
qos_mode<n>	Per-thread quality of service mode: priority, allocated-bandwidth, or best-effort.
request_group_size<n>	Number of requests that are grouped together (per thread <n>).
starvation_counter<n>	Tag level arbitration wait count for starvation avoidance. (See “Starvation Avoidance Counter” for more information.)
threads	Number of threads configured in the sys interface of the scheduler instance
words_per_period<n>	Bandwidth allocation in OCP words per scheduling period (per thread <n>).

Table 21 MemMax Design Parameters without a Corresponding Register Field

Parameter Name	Description
addr_tiling_end_bit	The end bit for address tiling. (For more information about address tiling, see “Per Thread Configuration of Address Tiling”.)
addr_tiling_st_bit	The start bit for address tiling. (For more information about address tiling, see “Per Thread Configuration of Address Tiling”.)
asyncreset_enable	Sets reset mode
bandwidth_rate_high_precision	When set, allows higher precision (1/256) in bandwidth allocation per thread.
compiled_mem	Allocate compiled memory
disable_mrmd	Disable MRMD support in MemMax Scheduler
disable_tag_addr_overlap	Disable address overlapping checking
dram_block_size	Maximum number of OCP words that can be issued as a burst to the controller
eight_multiples_only	Width of compiled SRAM is a multiple of 8
export_initializers	Presents interface signals allowing the initialization values for configuration registers to be supplied from outside the module through tie-offs instead of hard-coded in the RTL.
fine_grained_qos	Specifies QoS operating mode
mask_reg_access_msb_ addr_bits	Determines interpretation of the DRAM Regconfig register’s REG_BASE_ADDR field.
mem_clock_async	Implement an asynchronous buffer
optimize_wrap_rd_for_timing	Simplifies the handling of WRAP read bursts
read_data_buffers	Number of read data buffers
readex_enable	Support for ReadEx command on sys OCP

Parameter Name	Description
reg_base_addr	Base address of scheduler register space
request_buffers	Number of request buffers
set_interlock_depth	Set the size of the request to write-data cross-over queue
tag_addr_lsb	LSB of the address comparison bit for tags
tag_addr_msb	MSB of the address comparison bit for tags
tag_parallelism	Maximum number of tags allowed per thread
two_cycle_sched	Schedule once every second cycle
width_f<n>	Width of the low and high OP bits.
wrap_wr_support	Instantiates logic to support wrap write bursts
write_data_buffers	Number of write buffers

Parameters Derived from the DRAM Controller

Table 22 lists the configuration parameters derived from the DRAM controller. For a description of these parameters, see “DRAM Controller Parameters”. With the exception of the `resp_bypass_enable` and `max_trans` parameters, all parameters derived form the DRAM controller have a corresponding register.

Table 22 Parameters Derived from DRAM Controller

Parameter/Field Name	Description
resp_bypass_enable	Creates a **mem** OCP response bypass
bank8	Specifies the number of open banks per DRAM chip
bank_bit_location	Lowest bit number of bank bits within the **mem** OCP address, also used to define the DRAM page size
chip_bit_location	Lowest bit number of chip bits within the **mem** OCP address
chip_bits	Number of chip bits within the **mem** OCP address
dram_burst_type	Burst type used by DRAM controller to access the attached DRAM subsystem
dram_type	Type of DRAM used in the memory subsystem
max_trans	The maximum number of outstanding requests that MemMax is allowed to issue on the DRAM controller.

RTL Configuration Parameters

Configuration parameters are specified in the RTL configuration file using the syntax described above. Many RTL configuration parameters have corresponding configuration register bitfields—described in “MemMax Registers”—that may be configured with read-only or read-write access. The initial or reset value of a configuration register bitfield is the value of the

corresponding RTL configuration parameter (or the default value for the parameter, if the parameter is optional and omitted). Only fields marked read-write can be modified at run-time.

Instance Parameters for the MemMax Scheduler

Instance parameters are used to affect both the design-time configuration of the MemMax Scheduler instance (for example, the depth of specific data buffers) and the run-time operation of the scheduler (for example, the quality of service level for a particular thread).

Asynchronous Operation

The parameter `mem_clock_async` enables the use of separate clock domains on the *sys* and **mem** interfaces. The syntax of the parameter is:

```
[ param mem_clock_async (0|1) ]
```

When `mem_clock_async` is set to 1, the sys and `mem` OCP interfaces may be in separate clock domains and the scheduler instance includes logic for synchronization between the *sys* and **mem** clock domains.

Note: The **mem** OCP interface does not support asynchronous operation with configurations that have (or require) `reqdata_together` set equal to 1. The `soccomp` tool will generate an error if `mem_clock_async` is set to 1 in this case.

When `mem_clock_async` is enabled and `tags` = 1, the MemMax Scheduler uses FIFO-based synchronization between the *sys* and **mem** clock domains. This requires that the depths of the request buffer, read data buffer, and the write data buffer be even (or one).

When `mem_clock_async` is enabled and `tags` > 1, the MemMax Scheduler uses a synchronization bit in each active entry in the command buffer pool.

When `mem_clock_async` is set to 0 (the default), both *sys* and **mem** interfaces must reside in the same clock domain. See also “Divided Clocks” in Chapter 3.

Asynchronous Reset

The parameter `asyncreset_enable` defines whether the standard cells will have an asynchronous (direct set and clear) or synchronous reset. The syntax of the parameter is:

```
[ param asyncreset_enable (0|1) ]
```

When set to 1, the generated scheduler RTL contains reset retimer and synchronization tactical cells that use *asynchronous* reset. When set to 0, the generated scheduler RTL contains reset retimer and synchronization tactical cells that use *synchronous* reset. If omitted, a default value of 1 (asynchronous) is assumed. See the *SoC Integration Guide* for additional information about Sonics tactical cells.

Bandwidth Allocation

The `bandwidth_rate<thread>`, `words_per_period<thread>`, and `words_per_period_max<thread>` register fields specify the bandwidth to allocate

for priority and allocated-bandwidth threads, respectively. Another parameter bandwidth_rate_high_precision allows you to set the allocation of the bandwidth with a precision of 1/256.

The values for bandwidth allocation are specified on a per-thread basis using the following syntax:

```
[ regfield bandwidth_rate<thread> <rate> { access (RO|RW) } ]
[ regfield words_per_period_max<thread> (<limit>)  { access (RO|RW) } ]
[ regfield words_per_period<thread> (<limit>)  { access (RO|RW) } ]
```

Where:

```
<thread>:: 0|1|2|...|15
<rate>:: 1|2|3|...|65535
<limit>:: 1|2|3|...|65535
```

Note: The words_per_period_max<thread> register fields are only available when fine-grained QoS is enabled.

When bandwidth_rate_high_precision is not set, the parameter bandwidth_rate<thread> <rate> specifies the number of cycles after which the thread accumulates one credit, and may range between 2 and 65,535 (inclusive). If omitted, a default value of 2 is assumed. The value of bandwidth_rate<thread> is inversely proportional to the bandwidth served on the thread as a proportion of the total available bandwidth. For example, if the maximum available bandwidth is 2 GBs and a thread requires 200 MBs bandwidth, then the bandwidth_rate parameter should be set to 10.

When the bandwidth_rate_high_precision parameter is set (the default), the bandwidth for the thread is calculated as bandwidth_rate/256 instead of 1/bandwidth_rate. For example, a bandwidth rate of 225 (0xFF) gives a bandwidth allocation of 99.6%, while a bandwidth rate of 191 (0xBF) gives a bandwidth allocation of 74.6%. Also, when bandwidth_rate_high_precision is set, only the lower 8 bits of the BANDWIDTH_RATE register are use for the QoS metronome. The upper 8 bits are ignored.

Note: When importing designs created prior to MemMax DRAM System version 3.6, the parameter bandwidth_rate_high_precision is not set. For new designs, this parameters is set by default.

A thread's credit is stored and interpreted as a signed integer value, and changes (accumulations or reductions) in the thread's credit value are saturated to user-specified positive and negative limits. The positive and negative saturation limits are specified with one or both of the values words_per_period<thread> and words_per_period_max<thread>, depending on the QoS operating mode.

In normal QoS mode, the words_per_period value specifies the positive and negative saturation limits for the thread's credit. In fine-grained QoS mode, the positive limit is specified with words_per_period_max and the negative limit is specified with words_per_period.

The bandwidth_rate<thread>, words_per_period<thread>, and words_per_period_max<thread> parameters can also be set at run time—provided read-write access has been specified and depending on the QoS operating mode—by writing to the corresponding register fields (*BANDWIDTH_RATE*, *WORDS_PER_PERIOD*, and *WORDS_PER_PERIOD_MAX*, respectively). See THREADSCH(0-15).

Bank Busy Support

```
[ regfield pagemode<thread> ( 0 | 1 ) ]
[ regfield bankbusy_tread (0..255) ]
[ regfield bankbusy_twrite (0..255) ]
```

Where:

```
<thread>:: 0|1|2|...|15
```

The first of these sets the bit per thread in the *PAGEMODE* register, while the others set the number of **mem** interface clock cycles for which the bankbusy signal will be asserted after issue of the read or write. For more information about the *PAGEMODE* register, see “Page Mode”.

Bankbusy Prediction

To prevent issuing sequences where a miss would cause a stall of the memory-controller pipeline, thus degrading utilization, the MemMax Scheduler can use information from the memory-controller supplied on the *bankbusy* signals. Alternatively, the MemMax Scheduler can predict the bankbusy state using its own internal tracking capabilities. Two register fields are used to configure this feature:

```
[regfield bankbusy_twrite (0..255)]
[regfield bankbusy_tread  (0..255)]
```

These register fields control the number of **mem** clock cycles after a write or read, respectively. During the cycles, a miss will not be issued to the same memory bank.

Compiled Memory

The MemMax Scheduler may be configured to use either a flop-based or compiled memory module for the read data and write data buffers using the parameter `compiled_mem`. The syntax for the parameter is:

```
[ param compiled_mem (0|1) ]
```

When this parameter is set to 0, the scheduler is configured to use flop-based memory for the response and write data buffers. When set to 1, a compiled memory module is used. If omitted, a default value of 0 is assumed.

Notes

When tags > 1 and compiled memory is set:

- The parameter `reqdata_together` in the **mem** interface cannot be set to 1.
- The MemMax Scheduler uses additional logic for writing and reading to and from compiled memory. As a result, the behavior of the MemMax Scheduler with compiled SRAM in the data handshake path needs two extra cycles in the non-synchronous boundary configuration. The MemMax Scheduler needs one extra cycle in the data handshake path than the synchronous boundary needs.

For example, when `tags>1`, `compiled_mem=1`, and `mem_clock_async=1` (that is, the boundary is non-synchronous), the Datahandshake phase shows *two* additional

clock latencies compared to when `tags>1`, `compiled_mem=0`, and `mem_clock_async=1`.

When `tags>1`, `complied_mem=1`, and `mem_clock_async=0` (that is, the boundary is synchronous), the Datahandshake phase shows *one* additional clock latency compared to `tags>1`, `compiled_mem=0`, and `mem_clock_async=0`.

Compiled SRAM Width

The parameter `eight_multiples_only` is used to indicate whether the compiled SRAM can only support widths that are a multiple of 8. The syntax for the parameter is:

```
[ param eight_multiples_only (0|1) ]
```

If the parameter is disabled (the default), the MemMax Scheduler assumes that SRAMs with non-multiple of 8 widths can be supported, and generates SRAM to store all bits—that is, data and byte enable bits—for each of the read, write, and residue buffers.

When the parameter is enabled, the MemMax Scheduler generates SRAM only for widths that are a multiple of 8. In this case, the MemMax Scheduler stores data byte enables separately within the scheduler when the OCP data width is 16 or 32 bits. When the OCP data width is 64, 128, or 256 bits, the scheduler generates SRAM to store all bits for each of the read, write, and residue buffers.

Configuration Register Access Mode

The Boolean parameter `mask_reg_access_msb_addr_bits` controls the interpretation of the ***REG_BASE_ADDR*** field that is used to control access to the MemMax Scheduler or configuration registers of the memory controller. This parameter is provided for backwards-compatibility with earlier implementations of the MemMax Scheduler. The syntax for this parameter is:

```
[ param mask_reg_access_msb_addr_bits (0|1) ]
```

When this parameter is disabled (the default), the ***DRAMREGCONFIG*** register is read-only. The value programmed into the ***DRAMREGCONFIG*** register specifies the next highest address above the Scheduler's register address space. Requests with an address lower than the specified value access the MemMax Scheduler's register address space. Requests with an address greater than or equal to the programmed value access the controller's address space. When `mask_reg_access_msb_addr_bits` is disabled, the MemMax Scheduler does not mask the incoming *sys* OCP address.

When `mask_reg_access_msb_addr_bits` is enabled, the ***DRAMREGCONFIG*** register is writable. The MemMax Scheduler uses the value programmed in the register to determine the most-significant address bit for register accesses. It also uses the value to determine whether a register access request should be sent to the MemMax Scheduler register address space or the controller's register address space.

Disable MRMD Support

The parameter `disable_mrmd` is used to disable MRMD support. The syntax of the parameter is:

```
[ param disable_mrmd (0|1) ]
```

When enabled (that is, set to 1), MemMax Scheduler assumes that MRMD requests are never be presented at its *sys* interface and hence no hardware is instantiated to handle MRMD requests. If omitted, a default value of 0 is assumed, which implies that MRMD requests may be presented at the *sys* interface.

DRAM Block Size

The parameter `dram_block_size` is used to derive the *range boundary* for the *`DRAM_BLK_LOGSIZE`* register field, which is in turn used to define the maximum burst length of requests forwarded on the **mem** interface. The syntax of this parameter is:

```
[ param dram_block_size (1|2|4|8|16)]
```

The value of `dram_block_size` is specified in OCP words. The DRAM block size is chosen so that the product of DRAM block size and OCP data width is equal to the product of DRAM burst length and DRAM data width; that is:

`dram_block_size` = `DDR_BL x DDR_DATA_WIDTH / OCP_DATA_WIDTH`

For example:

- The **mem** interface data width is 32 bits, and the operating frequency is 533 MHz.
- The DRAM burst length is 8, data width (in bits) is 16, and the bus frequency is 533 MHz

In this case, the `dram_block_size` should be set to 4. If the `dram_block_size` parameter is omitted, a default value of 2 is assumed.

The scheduler may send bursts shorter than the maximum size in accordance with the burst conversion rules, or when the requesting burst length is smaller than the maximum size specified.

The maximum length of bursts can be adjusted at run-time by modifying the *`DRAM_BLK_LOGSIZE`* field in the *`DRAMCONFIG`* register, which is described in “DRAMCONFIG”.

Important! The internal configuration of the MemMax Scheduler is dependent on the initial value of `dram_block_size`. For correct operation of the scheduler, ensure that the following three conditions are always met:

- Condition 1: the *`DRAM_BLK_LOGSIZE`* register field is set to a value N such that 2^N is less than or equal to the value of `dram_block_size`.
- Condition 2: the *`DRAM_BLK_LOGSIZE`* register field is set to a value N such that 2^N is greater than or equal to the value of the `tag_interleave_size` parameter on the **sys** interface.
- Condition 3: the `dram_block_size` parameter is set to a value N such that N is greater than or equal to the value of the `tag_interleave_size` parameter on the **sys** interface.

DRAM Controller Register Base Address

When `mask_reg_access_msb_addr_bits` is disabled, the parameter `reg_base_addr` sets the base address for the DRAM controller's configuration registers. Accesses to addresses below this base address are directed to the MemMax Scheduler configuration registers. The syntax for this parameter is:

```
[ regfield reg_base_addr (2^8|2^9|2^10|...|2^15) { access (RO|RW) } ]
```

The value must be set to the size of the DRAM controller's register space rounded up to the nearest power-of-two between 2^8 and 2^{15}, inclusive. If addressing tiling is used, or if the scheduler is configured to operate with 14 or 15 threads, this parameter must be set to 0x200 or higher. If omitted, a default value of 0x100 (256) is assumed.

When `mask_reg_access_msb_addr_bits` is enabled, the parameter `reg_base_addr` specifies a bit pattern that is used as an address mask to indicate the address range to use for configuration register access, as described below. The syntax for this parameter is:

```
[ regfield reg_base_addr <bitpattern> { access (RO|RW) } ]
```

The value `bitpattern` must contain no more than one contiguous set of '1' bits. For example, `0x00000FF0` is a legal bit pattern, and `0x00000000` and `0x30FF07EFF` are not legal.

For a given legal bit pattern, the position of the most-significant 1 is denoted M_{MSB}, and the position of the least-significant 1 is M_{LSB}. Requests on **sys** that target the configuration registers (that is, *MAddrSpace* is set to zero) are masked with the bit pattern. Requests with masked addresses lower than $2\hat{}M_{LSB}$ access the MemMax Scheduler registers. Requests with masked addresses greater than or equal to $2\hat{}M_{LSB}$ is sent to the memory controller. In both cases the masked address is used as the register address.

Transactions with a non-zero value for *MAddrSpace* are always treated as memory transactions.

Fine-Grained QoS Mode

The `fine_grained_qos` parameter selects between the normal and fine-grained QoS modes described in "Dynamic QoS Parameters". The syntax is:

```
[ param fine_grained_qos (0|1) ]
```

When fine-grained QoS mode is disabled, only the primary per-thread QoS registers are available, and each thread's positive and negative credit limit is specified with a single, per-thread value.

When fine-grained QoS mode is enabled, the extended per-thread QoS registers are available and each thread's positive and negative credit limits can be individually set.

Note: When `bandwidth_rate_high_precision` is set, `fine_grained_qos` must also be set.

Request and Data Buffer Depth

The buffers allow traffic to be queued and provide decoupling between the *sys* and **mem** interfaces. Larger buffers allow the queuing of more traffic and the scheduler a wider range of choices, which can potentially improve performance.

You must specify the depth (in terms of number of entries) for each of the request, read data, and write data buffers using the `request_buffers`, `read_data_buffers`, and `write_data_buffers` statements, respectively. None of these parameters have default values. The ranges for these buffer parameters are listed in the following table.

Parameter	Range
request_buffers	1–32
read_data_buffers	1–256
write_data_buffers	1–256

Note: All integers in the ranges are supported. They are not restricted only to power-of-two values.

The syntax of these buffer statements is:

```
request_buffers { <req_buffer_depth> }
write_data_buffers { <data_buffer_depth> }
read_data_buffers { <data_buffer_depth> }
```

Where:

```
<req_buffer_depth>::  uniform 1|2|...|32  | { per_thread (1|2|...|32)+ }
<data_buffer_depth>:: uniform 1|2|...|256 | { per_thread (1|2|...|256)+ }
```

When the keyword `uniform` is used, the supplied value is applied to each thread. When the keyword `per_thread` is used, the size of the buffer for each thread is taken from the space-separated list of values provided: the first value is used for thread 0, the second value to thread 1, and so on. A value must be provided for each thread.

The request buffer, read and write data buffer must have a depth of at least one, subject to the following constraints:

- For any single thread, either the read buffer or the write buffer may not have zero depth.
- It is illegal to specify a uniform (that is, for all threads) depth of zero for any of the request, read data, or write data buffers.
- If `mem_clock_async=1`, do not set `reqdata_together`, `resp_bypass_enable=1`, and `req_bypass_enable=1`.
- For all three buffers. you cannot set `buffer_depth` to an odd number (other than 1) `if mem_clock_async=1`.
- The `command_buffer` depth in each thread cannot be 0.
- The `write_buffer` and `read_buffer` depth for each thread must be larger than `dram_block_size`.

Quality of Service

The qos_mode<thread> statement configures the quality-of-service level for each thread on a per-thread basis and configures read-write access to the corresponding QoS level field in the per-thread scheduling register. The syntax of the qos_mode<thread> statement is:

```
[ regfield qos_mode<thread> (0|1|2) { access (RO|RW) } ]
```

The QoS level is specified as an integer that maps to the desired QoS level as follows:

QoS Level	Meaning
0	Best-effort
1	Allocated bandwidth
2	Priority

When the qos_mode statement specifies access RW (read-write access) to the corresponding register field, the QoS level for each thread can be set at run-time, as described in THREADSCH(0-15).

If omitted, a default value of 0 (best-effort) is assumed, and the corresponding register field is assigned read-only access.

ReadEx Support

The parameter readex_enable is used to configure the MemMax Scheduler to support the OCP ReadEx command on the *sys* OCP interface. The syntax of the parameter is:

```
[ param readex_enable (0|1) ]
```

When set to 0 (the default), the scheduler does not support the ReadEx command. When set to 1, the MemMax Scheduler is configured to support the ReadEx command on the *sys* interface. There can be a single active lock at any time, as discussed in “Mutual Exclusion”. Mutual exclusion is provided on a per-OCP word basis.

Request Group Size

The scheduler parameter request_group_size specifies the number of consecutive requests (on the same thread) that will be considered to be part of the same group. (The concept of the request group and the impact of a group on scheduling is discussed in “Middle of Group”.) The syntax of the request_group_size parameter is:

```
[ regfield request_group_size<thread> (0|1|2|...|2^8-1) {access (RO|RW) }
```

When request_group_size for a thread is set to the value *N*, the scheduler schedules *N*+1 consecutive requests from the thread as a single group, provided that valid requests continue to be available on that thread. A new group is started when:

1. In the previous cycle, the request from this thread was invalid *or* no memory requests were available for scheduling *and* a valid request is available in this cycle.
2. There is a new transaction on the *sys* interface.

If not specified, a default value of 0 is assumed, which causes the scheduler to treat each request on a thread as a new group.

The group size may also be specified by writing to the *REQUEST_GROUP_SIZE* field in the per-thread scheduling configuration register for the thread, as described in THREADSCH(0-15).

Starvation Avoidance Counter

Each request in each scheduler-internal thread (there can be up to `tag_parallelism` requests per thread on the *sys* interface) has an associated starvation counter used to ensure that the request is eventually serviced, by boosting the request's priority when the starvation counter expires. (The use of the starvation counter is described in "Weight Vector Bit Computation".) The initial value of the starvation counter is specified on a per-requesting thread basis using the following syntax:

```
[regfield starvation_counter<thread> <starvation_value> {access(RO|RW)}]
```

Where:

```
<thread>:: 0|1|2|...|15
<starvation_value>:: 0|1|2|...|255
```

The value 0 has special meaning: if a value of 0 is specified, starvation avoidance is disabled for this thread.

If the `starvation_counter` statement is not specified for a particular thread, a default value of 0 and a default access of R is assumed. When read-write access is enabled, the starvation counter initialization value can be set at run-time by writing to the STARVATION_COUNTER field in the thread's global scheduling register.

Tag Address Comparison

When tag address overlap checking is enabled, the parameters `tag_addr_lsb` and `tag_addr_msb` specify the least-significant and most-significant bits of the address to use when comparing addresses for overlap. The syntax for these parameters is:

```
[ param tag_addr_lsb (0|1|...|(addr_wdth-1)) ]
[ param tag_addr_msb (0|1|...|(addr_wdth-1)) ]
```

For example, the following parameters specify the use of address bits 9–23, inclusive, for comparison:

```
param tag_addr_lsb 9
param tag_addr_msb 23
```

If not specified, default values are used. The default value of `tag_addr_lsb` is $\log_2(B \times W)$, where B is the DRAM block size, in bytes, and W is the OCP data width, in bytes. The default value of `tag_addr_msb` is `address_width` - 1.

The tag_addr_lsb and tag_addr_msb have options that allow you to specify the bits per thread. For example, in the following command, each number represents the tag_addr_msb for each thread:

```
param tag_addr_msb { per_thread 35 34 29 }
```

In the following command, every thread uses the same number of `tag_addr_msb`:

```
param tag_addr_msb { uniform 29 }
```

Tag Address Overlap Check Disable

The parameter `disable_tag_addr_overlap` is used to enable or disable address overlap comparison logic. The syntax for the parameter is:

```
[ param disable_tag_addr_overlap (0|1) ]
```

When `disable_tag_addr_overlap` is disabled (set to 0, the default), the MemMax Scheduler implementation will include logic to check each new request for address overlap with a pending request. If the addresses overlap, the new request is blocked pending completion of the pending request. The address overlap comparison depends on the values specified for `tag_addr_lsb` and `tag_addr_msb`, described above.

When `disable_tag_addr_overlap` is enabled (set to 1), the MemMax Scheduler implementation does not include logic to check for overlapping request addresses. New requests may be scheduled ahead of pending requests.

Important! Disabling the address overlap check can reduce the area of a MemMax Scheduler implementation. However, OCP violations may result if requests with overlapping addresses are sent to the MemMax Scheduler.

Tag Parallelism

The `tag_parallelism` statement specifies the number of tagged requests allowed per thread on the *sys* interface. The syntax for this parameter is:

```
[ tag_parallelism { <parallelism> } ]
```

Where:

```
<parallelism>:: uniform 1|2|...|16 | { per_thread (1|2|...|16)+ }
```

A value can be specified for all threads using the `uniform` keyword. If the `tag_parallelism` statement is omitted, a default value of 1 is assumed for all threads (which disables tag-level rescheduling).

Two-Cycle Scheduling

The parameter `two_cycle_sched` can be used to configure the MemMax Scheduler to use an internal register to hold the arbitration winner prior to writing the request to the **mem** interface. The syntax of the `two_cycle_sched` parameter is:

```
[ param two_cycle_sched (0|1) ]
```

When the parameter is set to 0 (the default, if the parameter is not set), scheduling takes place every cycle. At each cycle, the winning request is forwarded to the **mem** interface.

When the parameter is set to 1, scheduling takes place once every two cycles. Arbitration is done in the first cycle, and the winning request is registered. In the next cycle, this request is forwarded to the **mem** interface provided the request is not invalidated. The request can be invalidated if one of the following conditions are true:

- The *SThreadBusy* signal on the **mem** interface is asserted, or,
- The request is a page-miss, and the *bank_busy* flag for the thread is asserted.

The `two_cycle_sched` parameter replaces the `sched_repeat_delay` parameter used in previous releases of the MemMax Scheduler. The `sched_repeat_delay` parameter is deprecated: RTL configurations that use `sched_repeat_delay` will be converted to use `two_cycle_sched` and a warning will be displayed.

Write Request Interlock Depth

The parameter `set_interlock_depth` specifies the depth of the scheduled write request buffer. (Scheduled write requests are write requests received by the MemMax Scheduler that have been arbitrated and scheduled.) The syntax of the parameter is:

```
[ param set_interlock_depth (0|1|...|127) ]
```

When `set_interlock_depth` is set to a non-zero value, *N*, the scheduler can queue up to *N* scheduled write requests to the DRAM controller before further scheduled write requests are interlocked. Interlocking causes the scheduler to hold the next write request until the last data word of the current write transaction is sent to the DRAM controller.

Notes

- You must have the **mem** OCP interface parameter `reqdata_together` set to `0` to avoid a performance penalty. (The default value is `0`.) If `reqdata_together` is set to 1, the `set_interlock_depth` has no impact on performance.
- On setting the `set_interlock_depth` parameter to N (0; N < 127), the WTID buffer will have a depth of N + 1.
- To avoid a performance impact, the internal cross-over queue must be deep enough to allow the issue of sufficient write requests to hide the latency from a write-request being issued by MemMax until the DRAM controller accepts the corresponding final data-transfer.

  Assuming that the controller de-queues a command once the column command (RD/WR) has issued, given maximum memory frequency fMEM and minimum DRAM block size BL, calculate the interlock depth as follows. Note that `ceil` rounds up to the nearest integer:

  ```
  ceil((ceil[tRP*fMEM]+ceil[tRCD*fMEM]+ceil[CWL*tCK*fMEM])/BL)
  ```

- The default setting is expected to be adequate in most systems, provided `max_trans` is set correctly, and is calculated as follows.
  - If `reqdata_together=1`, set `set_interlock_depth=0`.

- Otherwise, set `set_interlock_depth<=(max_trans-1)`.

Arbitration Interleave Style

You can use the arbitration interleaving style of the MemMax Scheduler instance to control response behavior. To set the interleave style, use the `resp_transaction_interleave_enable` parameter, which sets the arbitration interleave style for the response. When the parameter is set to a value of 0, the multi-threaded arbiters of the MemMax response buffer is configured to grant access to the arbitrating thread that has least recently completed a transfer. When the parameter is set to a value of 1, the multi-threaded arbiters of the Memmax response buffer are configured to grant access to the arbitrating thread that has least recently completed a transaction.

Wrap Write Support

The `wrap_wr_support` parameter allows the MemMax Scheduler to instantiate logic to support WRAP WR bursts. Aligned WRAP WR bursts are passed to the **mem** interface by treating them as INCR bursts. Unaligned WRAP bursts are chopped into two INCR bursts. The first burst starts from the start address and ends at the WRAP boundary. The second burst starts at the WRAP base address and ends at the starting address. Also note that all WR bursts are issued at the **mem** OCP interface such that the starting address is aligned to the DRAM block size (see “DRAM Block Size”). This parameter is not set by default.

Optimization of Wrap Read for Timing

When the `optimize_wrap_rd_for_timing` parameter is enabled, handling WRAP RD bursts is simplified. Aligned WRAP WR bursts are passed to the **mem** interface by treating them as INCR bursts. Unaligned WRAP bursts are chopped into two INCR bursts. The first bursts starts from the start address and end at the WRAP boundary. The second burst starts at the WRAP base address and ends at the starting address. When this option is selected, the MemMax Scheduler does not require any re-order buffers in the response phase, and its handling of the WRAP burst is generally simplified. This results in a lower area/higher frequency design. However, this may involve some penalties regarding performance.

For optimization of wrap read for timing, you should also be aware of the following:

- If `wrap_write_support` is set to 1, you should set `burstseq_incr_enable` to 1 in the **mem** interface.
- If `optimize_wrap_rd_for_timing_and_area` is set to 1, you cannot set `burstseq_xor_enable` in the sys interface.
- If `optimize_wrap_rd_for_timing_and_area` is set to 1, you should set `burstseq_incr_enable` to 1 in the **mem** interface.
- If `optimize_wrap_rd_for_timing_and_area` is set to 1, you must set `dram_block_size` larger than 1.

`ocp_register_slice`
: There are two parameters in the MemMax Scheduler to help timing at the I/O boundary. Setting `sys_ocp_reg_slice_enable` to 1 inserts one register slice

at the sys interface. Setting `mem_ocp_reg_slice_enable` to 1 inserts one register slice at the **mem** interface.

`tag_arbitration_pipelined`
This parameter causes the MemMax Scheduler to insert one pipeline stage after tag arbitration and add one clock latency.

Page Mode

The commands issued on the `mem` interface can be tagged using a bit within the *MReqInfo* field to indicate whether the accessed memory page should be closed or kept open after the access. The value issued can be controlled on a per-thread basis, using the per-thread *PAGEMODE* regfield bit.

```
[ regfield pagemode<thread> (0|1) {access (RO|RW)} ]
```

The *PAGEMODE* register field defaults to the value `0` with default access type `RO`. You can set *PAGEMODE* register fields to read-write (RW) but they default to read-only (RO) unless specifically configured to RW.

- A value of `0` means open-page mode, with the issued commands signalling to keep the memory page open after the access. MemMax operates in open-page mode but assumes a bank is immediately available after a miss (unless predicted or real bankbusy signals indicate otherwise.
- A value of 1 means closed-page mode, with the issued commands signalling to close the memory page after the access.

MemMax gives you the ability to select open page or close page modes for each thread of the OCP sys interface, with close page signaling provided on the memory interface to indicate when the memory controller should close a page and begin a precharge.

DRAM Controller Parameters

The RTL configuration parameters described in this section are used to describe the attributes of the DRAM controller and attached DRAM system.

Number of Banks Per DRAM Chip

The parameter `bank8` is used to specify the number of open banks per DRAM chip. The syntax of the parameter is:

```
[ regfield bank8 (0|1) { access (RO|RW) } ]
```

When `bank8` is set to 1, the scheduler assumes that each DRAM chip has eight banks. When `bank8` is set to 0 (the default), the scheduler assumes four banks per chip.

The number of banks in use by the memory subsystem and the number of bank bits used in the address field is:

bank8 value	Banks Per-chip	Address Field Bank Bits	Banks Memory Subsystem
1	8	3	8 << CHIP_BITS
0	4	2	4 << CHIP_BITS

When read-write access to the corresponding register field is specified, the number of banks per chip can be set at run-time by writing to the BANKS field in the DRAMCONFIG register.

Bank Bit Location

The parameter `bank_bit_location` specifies the location of the least-significant bit of the bank selection bitfield in the address written on the **mem** OCP interface. The syntax of this parameter is:

```
[ regfield bank_bit_location (8|9|...|15) { access (RO|RW) } ]
```

If the parameter is omitted, a value of 11 is assumed.

The bank selection bitfield width is determined by the value of the `bank8` parameter. When `bank8` is set to 0, a width of two bits is assumed; when `bank8` is set to 1, a width of three bits is assumed. For example, if `bank_bit_location` is set to 11 and `bank8` is set to 0, the controller reads the two bank bits from *MAddr[12:11]*.

The `bank_bit_location` parameter can be set at run time by writing to the BANK_BIT_LOCATION field in the DRAMCONFIG register.

For information about the interpretation of address bits in the MemMax Scheduler, see Figure 9.

Number of Chip Bits

The parameter `chip_bits` specifies the number of chip bits that are present in the address written on the `mem` OCP interface. The syntax of this parameter is:

```
[ regfield chip_bits (0|1|2) { access (RO|RW) } ]
```

For example, if the DRAM controller requires two chip bits, set `chip_bits` to 2.

If the `chip_bits` parameter is omitted, a default value of 0 is assumed (that is, no chip bits). The legal values of `chip_bits` depend on the value of the `bank8` parameter: if `bank8` is set to1, `chip_bits` may only be set to 0 or 1.

The `chip_bits` parameter can also be set at run time by writing to the *`CHIP_BITS`* field in the DRAMCONFIG register.

Chip Selection Bitfield Location

The parameter `chip_bit_location` specifies the location of the most-significant bit in the chip-selection bitfield as an offset from the most-significant OCP address bit. The syntax of this parameter is:

```
[ regfield chip_bit_location (0|1|2|...|7) { access (RO|RW) } ]
```

For example, if the DRAM controller expects two chip-selection bits in *MAddr[25:24]*, and the OCP address width parameter, `addr_wdth`, is set to 27, set `chip_bit_location` to 1. If omitted, a default value of 0 is assumed. If the `chip_bits` parameter is set to 0, the address is assumed to contain no chip-selection bits and the value of `chip_bit_location` is ignored.

The chip-selection bitfield offset can be set at run-time by writing to the register field *`CHIP_BIT_LOCATION`* field in the *`DRAMCONFIG`* register (summarized in

Table 26). The *CHIP_BIT_LOCATION* register field is interpreted as an unsigned integer in the range 0 to 7, inclusive. The register field is initialized with 0 or the value assigned to `chip_bit_location`, if given.

DRAM Burst Type

The register field `dram_burst_type` specifies the burst type used in the attached DRAM subsystem. The MemMax Scheduler supports DRAMs in either interleaved or sequential burst access modes. The syntax is:

```
[ regfield dram_burst_type (0|1) { access (RO|RW) } ]
```

When `dram_burst_type` is set to 0, the burst access mode is sequential. When `dram_burst_type` is set to 1 (the default), the burst access mode is interleaved.

The burst type must match the type that is programmed into the attached DRAM devices.

DRAM Type

The `dram_type` register field specifies the type of DRAM used in the memory subsystem—the MemMax Scheduler supports DDR1, DDR2, and DDR3 DRAM types. The syntax is:

```
[ regfield dram_type (1|2|3) { access (RO|RW) } ]
```

The DRAM_TYPE register field is interpreted as an unsigned two-bit integer. Values of 1, 2, and 3, indicate DDR1, DDR2, and DDR3, respectively. The value 0 is reserved, and should not be used. The default value of this field is 2 (DDR2).

Response Bypass

The parameter `resp_bypass_enable` creates a bypass that allows a response arriving on the **mem** OCP interface to be immediately returned to the *sys* OCP interface, reducing request-to-response latency by one cycle. The syntax is:

```
[ param resp_bypass_enable (0|1) ]
```

When `resp_bypass_enable` is enabled (that is, set to 1), the response can be bypassed only when there are no pending responses in any of the read data buffers in the MemMax Scheduler. When this parameter is disabled (that is, set to 0), any response from the **mem** OCP is delayed at least 1 cycle. If omitted, a default value of enabled is assumed.

Note: If `writeresp_enable` is enabled on the *sys* OCP interface and `writeresp_enable` is disabled on the **mem** OCP interface, the MemMax Scheduler does not bypass the response from **mem** to *sys*, and setting `resp_bypass_enable` is not allowed.

Maximum Transactions

This parameter limits the number of transactions MemMax can have outstanding on the **mem** OCP interface. An outstanding transaction is one where the request was issued but the response has not yet been received. The `max_trans` value should be set large enough to keep the memory-controller/DRAM pipeline full, but no larger because that costs extra area within MemMax.

The optimal value can be calculated using the following formula:

```
max_trans = CEIL((tFE + tRP + tRCD + CL + tBL + tBE)/tBL)
```

Where:

- `CEIL` rounds the result up to the nearest integer.
- `tFE` models the delay from command acceptance from MemMax **MEM** interface until the DRAM PHY sends the precharge (“Front End”).
- `tRP` is the DRAM precharge-to-activate command delay.
- `tRCD` is the DRAM activate-to-read command delay.
- `CL` is the delay from read command to first read data beat.
- `tBL` is the time required to send the read burst (8 beats at 1600Mbps for DDR3-1600).
- `tBE` models the delay from read data entering the PHY until it is returned to MemMax **MEM** interface (“Back End”).

MemMax Scheduler and Master Layer

When connecting a MemMax Scheduler to a SonicsConnect AXI Master Layer (AXIML), you must configure the AXI ports of the AXIML component as follows:

- Set all ports to the same protocol, either AXI3 or AXI4.
- Set `read_enable` and `write_enable`.
- Set `id_width` > 0.
- Set at least one AXI port as follows so that the address space information is put into the least significant bit (LSB) of *AxUSER* delivered to MAddrSpace on the outgoing OCP interface:
  - `aruser_width` > 0
  - `awuser_width` > 0
  - `userinfo_to_maddrespace_len > 0`
- Set all AXI3 ports as follows:
  - `len_width` = 4
  - `lock_width` = 2
- Set all AXI4 ports as follows:
  - `len_width` = 8
  - `lock_width` = 1
- Attach the AXI Master Layer (AXIML) to the OCP interface of the MemMax Scheduler. You cannot use it as a standalone component.

Note: If you set the `addr_width` to different values on each port, the maximum value of `addr_width` is used on the outgoing OCP interface.

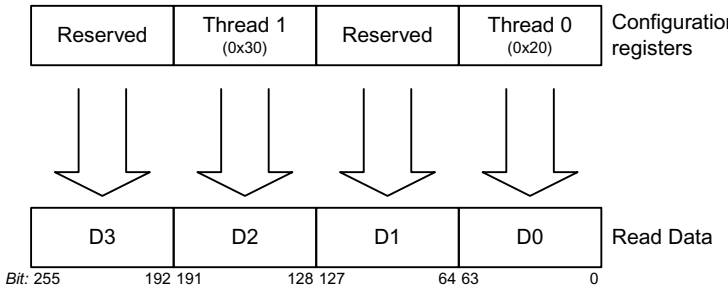
For more information about SonicsConnect and the AXI Master Layer, see the *SonicsConnect Reference*.

6 MemMax Registers

All MemMax Scheduler configuration registers are 64 bits wide. Configuration registers can be read using single transfers of 16, 32, 64, 128, or 256 bits. When a 16- or 32-bit read is performed, only the least-significant (right-most) 2 or 4 bytes of the addressed configuration register are returned. When a 128-bit (or 256-bit) read is performed, the contents of 2 (or 4) contiguous registers are returned.

Figure 15 shows an example 256-bit read from address 0x20. The scheduler returns the values of the four configuration registers at addresses 0x20 through 0x38, inclusive.

Figure 15 Example of 256-bit read from address 0x20



The MemMax Scheduler only supports writes to configuration registers of up to 64 bits, using the *MDataByteEn* signal to select the bits to write from an input OCP data word of 16, 32, 64, 128, or 256 bits. The *MDataByteEn* signal is used to select the bytes from the OCP data word that is copied to the destination register.

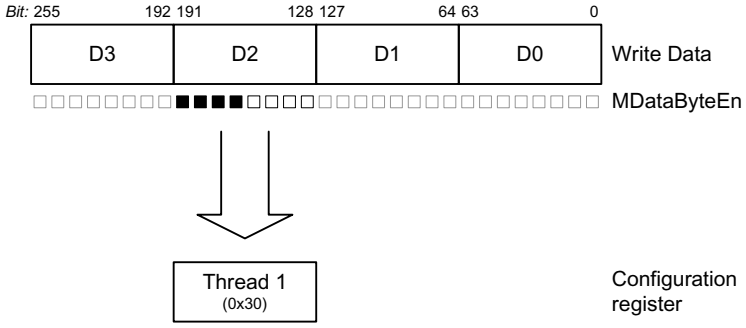
When the OCP data word is 16- or 32-bits wide, multiple transfers are needed to overwrite all of the bytes in the configuration register. For example, to modify all of the bytes in the configuration register at address 0x00 using 16-bit transfers, four consecutive writes to addresses 0x00, 0x02, 0x04, and 0x06 are required. (Only bytes for which the *MDataByteEn* signal bits are asserted are overwritten.)

When the OCP data word is 128 (or 256) bits wide, the *MDataByteEn* signal is interpreted as 2 (or 4) groups of 8 byte-enable bits. Each byte-enable group corresponds to a set of 64 data bits within the OCP data word. Only one of the

byte-enable groups may be non-zero. The bytes from the OCP data word corresponding to the non-zero bits of this byte-enable group are copied to the target register.

Figure 16 shows an example 256-bit write to the per-thread configuration register at address 0x20. Here, the value of *MDataByteEn* is 0x00F00000, causing the four most-significant bytes of the 64-bit word D2 to be written to the four most-significant bytes of the per-thread configuration register at address 0x20.

Figure 16 Example of 256-bit write to address 0x20



Bits specified as reserved are read as 0. System software is advised to write zeros to these bits to maintain future compatibility.

There are configuration registers in both the MemMax Scheduler and the DRAM controller. Configuration registers in either device are accessed by setting the *MAddrSpace* signal in the *sys* OCP interface to zero. Requests with non-zero values of *MAddrSpace* are interpreted as accesses to the physical DRAM memory. The *REG_BASE_ADDR* field in the DRAM controller configuration register determines the base address for the configuration registers for both the MemMax Scheduler and DRAM controller. Address matchers within the scheduler determine whether the request targets the configuration registers in the scheduler or in the DRAM controller.

Impact on MemMax Scheduler

Before modifying configuration registers, the scheduler should be allowed to complete all pending requests and brought to a quiescent state.

MemMax Register Block

Configuration registers with global scope are located at 8-byte offsets between 0x00 and 0x18, inclusive. Per-thread control registers are arranged at 16-byte offsets between 0x20 (for thread 0) and 0x110 (for thread 15). The seven pairs of address tiling configuration registers are located at 32-byte offsets between 0x120 (tiling function 0) and 0x1E0 (tiling function 6).

Note: Only the 9 least-significant bits of the address are used to access the configuration registers of the MemMax Scheduler. Thus, the configuration registers are aliased every 0x200 bytes. For example, the *DRAMCONFIG* register is accessible at addresses 0x0000, 0x200, 0x400, and so on.

Table 23 summarizes the configuration registers in the MemMax Scheduler and their addresses in the configuration address space.

Table 23 Register Summary for the MemMax Scheduler

MEMMAX__MEMMAX

Address Offset	Register	Description
0x00	DRAMCONFIG	DRAM configuration
0x08	DRAMREGCONFIG	DRAM regconfig
0x10	GLOBALSCHEDULING	Global thread scheduling
0x18	CONTROL	Reserved register
0x20, 0x30, 0x40, 0x50, ..., 0x110	THREADSCH(0-15)	Per-thread scheduling
0x28, 0x38, 0x48, 0x58, ..., 0x118	THREADSCHEXT(0-15)	Per-thread scheduling
0x120, 0x140, 0x160, ..., 0x1e0	TILING_SWAP(0-6)	Base address of the ***n***th tiling bit-swap control register.
0x130, 0x150, 0x170, ..., 0x1f0	TILING_BAOP(0-6)	Base address of the ***n***th tiling bit-mask-and-add control register.

Each register is summarized as a table and the column headings for each register are explained in the following table.

Table 24 Register Field Information

REG-FIELD-COLUMN-HEADING-INFO

Column Heading	Description
Bits	The bit range of the field. Bit positions are numbered from 0 to x, where 0 is the least-significant bit and x is the most-significant bit.
Field	The name of the field.
Range	The permissible set of values for a register field. When all encodings that can be expressed by the bit field width are allowed, the word *full* is used.
Default	The initial value assigned to this field. The term *N/A* is used when a default is not applicable, either because the user has no control over the value or because a specific value must always be provided.
R/W	The read/write attribute associated with a register field. Register fields can be assigned read/write attributes. If any field of a register in a register block is set to WC or RW, the register block must be made accessible by including it in the address map.
Description	Describes the purpose of the field.

Table 25 Register Field Read/Write Attributes

READ-WRITE-ATTRIBUTES-TABLE

Attribute	Acronym	Description
read-only	RO	Register field can only be read.
read/write	RW	Register field can be read and written, but defaults to RO unless configured as RW.

DRAMCONFIG

The DRAM configuration register, ***DRAMCONFIG***, contains register fields used to configure the scheduler to make it consistent with the DRAM controller. Changing any of the fields in this register will usually require a concomitant change in the DRAM controller configuration registers (and vice-versa).

When operating in open page mode, the MemMax Scheduler selectively updates its page history registers so that a page miss request is not issued if the bank_busy flag is high. If one or more fields of the ***DRAMCONFIG*** are writable, the scheduler resets its page history registers when writing to the ***DRAMCONFIG*** register, and treats the next scheduled request as a page miss. The page history registers are not affected by reads from the ***DRAMCONFIG*** register or any accesses (read or write) to other scheduler configuration registers.

Table 26 DRAMCONFIG Register Fields

MEMMAX__MEMMAX__DRAMCONFIG

Bits	Field	Range	Default	R/W	Description
63:30	Reserved				Reserved
29:28	CHIP_BITS	0-2	0	RO	Number of bits in chip selection bitfield in the mem OCP address
27	Reserved				Reserved
26:24	CHIP_BIT_LOCATION	0-7	0	RO	Offset of most-significant chip-selection bit from the most-significant bit in the mem OCP address
23:20	Reserved				Reserved
19:16	BANK_BIT_LOCATION	8-15	11	RO	Offset of least-significant bit of bank-selection bitfield from the least-significant bit in the mem OCP address
15:10	Reserved				Reserved
9:8	DRAM_BLK_LOGSIZE	0,1,2, 3, 4	1	RO	Log_2 of DRAM block size: 0 = 1 OCP word 1 = 2 OCP words 2 = 4 OCP words 3 = 8 OCP words 4 = 16 OCP words Must be less than or equal to the dram_block_size parameter specified at design time. For multi-tag configurations with tag_parallelism >1, this field must also be configured to a value greater or equal to the value of tag_interleave_size. See “DRAM Block Size”.
7:6	Reserved				Reserved
5:4	DRAM_TYPE	1-3	2	RO	Type of DRAM used in the memory subsystem 1 is DDR1 2 is DDR2 3 is DDR3 0 is not supported
3	Reserved				Reserved

MEMMAX__MEMMAX__DRAMCONFIG

Bits	Field	Range	Default	R/W	Description
2	DRAM_BURST_TYPE	0,1	0	RO	Burst type used by DRAM controller to access the attached DRAM subsystem 0 is Sequential 1 is Interleaved
1	Reserved				Reserved
0	BANK8	0,1	0	RO	Number of banks per-chip 0 = 4 banks 1 = 8 banks

DRAMREGCONFIG

The *DRAMREGCONFIG* register is located at address 0x08 and contains the *REG_BASE_ADDR* field, which can be set using the `reg_base_addr` parameter. If the `reg_base_addr` parameter is not specified, the default value of this field is 0x400. Note that the syntax of the `reg_base_addr` parameter depends on the setting of the `mask_reg_access_msb_addr_bits` parameter.

The interpretation and purpose of the REG_BASE_ADDR field is described in "Configuration Register Access Mode".

Table 27 DRAMREGCONFIG Fields

MEMMAX__MEMMAX__DRAMREGCONFIG

Bits	Field	Range	Default	R/W	Description
63:0	REG_BASE_ADDR	Full	0x400	RW	Specifies the base address and range of valid addresses for configuration register access.

GLOBALSCHEDULING

The global thread scheduling register contains a single read-only field, *THREADS*, that reflects the number of threads configured on the **sys** interface for this instance of the MemMax Scheduler. The number of threads supported is specified with the *sys* interface's `threads` parameter. This register is located at address 0x10 in the scheduler's configuration address space.

Table 28 Global Thread Scheduling Register Fields

MEMMAX__MEMMAX__GLOBALSCHEDULING

Bits	Field	Range	Default	R/W	Description
63:48	Reserved				Reserved
47:40	BANKBUSY_TWRITE	0-0xff	0	RW	For bankbusy internal prediction. Configures the number of MemMax memory-interface clock cycles that a bank is considered busy immediately after an access. During this busy period the page-miss requests are not scheduled to that same bank.

MEMMAX__MEMMAX__GLOBALSCHEDULING

Bits	Field	Range	Default	R/W	Description
39:32	BANKBUSY_TREAD	0-0xff	0	RW	For bankbusy internal prediction. Configures the number of MemMax memory-interface clock cycles that a bank is considered busy immediately after an access. During this busy period the page-miss requests are not scheduled to that same bank.
31:13	Reserved				Reserved
12:8	THREADS	0x01–0x10		RO	Number of threads configured on the sys interface for this MemMax instance
7:5	Reserved				Reserved
4	PAGE_POLICY			RO	This field has been deprecated and is locked to 1 for backward compatibility
4:0	Reserved				Reserved

CONTROL

The CONTROL register has a single field, *`CLOCK_GATE_DISABLE`*, that was previously used to control fine-grained clock gating but is now deprecated. All other bits of the register are reserved. This register is located at address 0x18 in the scheduler's configuration address space. See the *SoC Integration Guide* for current information about clock gating.

Table 29 CONTROL Register Fields

MEMMAX__MEMMAX__CONTROL

Bits	Field	Range	Default	R/W	Description
63:57	Reserved				Reserved
56	CLOCK_GATE_DISABLE	0,1	0	RO	Overrides fine-grained hardware clock gating (deprecated)
55:0	Reserved				Reserved

THREADSCH(0-15)

Each thread is allocated two scheduling configuration registers: one primary and one extended. The fields in these registers are used to set parameters related to QoS arbitration, as described in Quality-of-Service Scheduling.

The MemMax Scheduler supports two modes of QoS operation: normal QoS and fine-grained QoS. The mode is selected at design-time using the parameter `fine_grained_qos` (described in Fine-Grained QoS Mode). When `fine_grained_qos` is set to 1, the primary and extended per-thread scheduling registers are active. When `fine_grained_qos` is set to 0, only the primary per-thread scheduling register is available.

Changes to any of the fields in the per-thread scheduling registers affect only that thread. However, the change can affect pending requests and hence the system should be in a quiescent state before the registers are modified.

The primary per-thread scheduling registers are located at 16-byte offsets starting at address 0x20 (for thread 0) through 0x110 (for thread 15) in the scheduler's configuration address space. The extended per-thread scheduling registers are located at 16-byte offsets starting at address 0x28 (for thread 0) through 0x118 (for thread 15) in the scheduler's configuration address space.

Table 30 THREADSCH(n) Primary Per-Thread Scheduling Register Fields

MEMMAX__MEMMAX__THREADSCH(0-15)

Bits	Field	Range	Default	R/W	Description
63:48	WORDS_PER_PERIOD	1-((2^16)-1)	1	RW	Bandwidth allocation: OCP words per scheduling period, minimum
47:32	BANDWIDTH_RATE	1-((2^16)-1)	2	RW	Bandwidth allocation: cycles per OCP request
31:24	REQUEST_GROUP_SIZE	0-((2^8)-1)	0	RW	Number of requests to be grouped together
15:8	STARVATION_COUNTER	0-((2^8)-1)	0	RW	Starvation counter reset value
7	PAGEMODE	0,1			Page Mode 0 = open 1 = closed
6:2	Reserved				Reserved
1:0	QOS_MODE	0,1,2	0	RW	Quality of service level 0 = best effort 1 = allocated bandwidth 2 = priority

THREADSCHEXT(0-15)

The primary per-thread scheduling registers are located at 16-byte offsets starting at address 0x20 (for thread 0) through 0x110 (for thread 15) in the scheduler's configuration address space. The extended per-thread scheduling registers are located at 16-byte offsets starting at address 0x28 (for thread 0) through 0x118 (for thread 15) in the scheduler's configuration address space.

Table 31 THREADSCHEXT(n) Extended Per-Thread Scheduling Register Fields

MEMMAX__MEMMAX__THREADSCHEXT(0-15)

Bits	Field	Range	Default	R/W	Description
63:16	Reserved				Reserved
15:0	WORDS_PER_PERIOD_MAX	1-((2^16)-1)	1	RW	Bandwidth allocation: OCP words per scheduling period (maximum)

TILING_SWAP(0-6)

Seven pairs of registers are used to enable and define the address tiling functions. The register pairs are located on 32-byte boundaries starting at address 0x120. Within each pair, the *TILING_SWAP(n)* register controls the bit-

swapping function, and the *TILING_BAOP(n)* register controls the bank address transformation function.

Note: Any write to a tiling configuration register must be done with full data byte enables. That is, the entire register must be modified in a single write. Partial writes into a tiling configuration register are not supported. If the sys data width is less than 64 bits, consecutive writes must be used to write all 64-bits of the tiling register.

The purpose and use of each of the register fields are described in further detail in the “Address Tiling” section.

Table 32 TILING_SWAP(n) Fields

MEMMAX__MEMMAX__TILING_SWAP(0-6)

Bits	Field	Range	Default	R/W	Description
63:62	Reserved	N/A	0	NA	Reserved
61:56	ILoc5	0 or (D < ILoc5 < A), where D is the base-2 logarithm of the DRAM block size minus 1, and A is the bit location of the LSB of the DRAM chip bits.	0	RW	Bit position of 5th segment LSB
55:50	Len4	0–A, where A is the bit location of the LSB of the DRAM chip bits.	0	RW	Number of bits in 4th segment
49:44	ILoc4	0 or (D < ILoc4 < A), where D is the base-2 logarithm of the DRAM block size minus 1, and A is the bit location of the LSB of the DRAM chip bits.	0	RW	Bit position of 4th segment LSB
43:38	Len3	0–A, where A is the bit location of the LSB of the DRAM chip bits.	0	RW	Number of bits in 3rd segment
37:32	ILoc3	0 or (D < ILoc3 < A), where D is the base-2 logarithm of the DRAM block size minus 1, and A is the bit location of the LSB of the DRAM chip bits.	0	RW	Bit position of 3rd segment LSB
31:30	Reserved	NA	0	NA	Reserved
29:24	Len2	0–A, where A is the bit location of the LSB of the DRAM chip bits.	0	RW	Number of bits in 2nd segment
23:18	ILoc2	0 or (D < ILoc2 < A), where D is the base-2 logarithm of the DRAM block size minus 1, and A is the bit location of the LSB of the DRAM chip bits.	0	RW	Bit position of 2nd segment LSB
17:12	Len1	0–A, where A is the bit location of the LSB of the DRAM chip bits.	0	RW	Number of bits in 1st segment
11:6	ILoc1	0 or (D< ILoc1 < A), where D is the base-2 logarithm of the DRAM block size minus 1, and A is the bit location of the LSB of the DRAM chip bits.	0	RW	Bit position of 1st segment LSB
5:00	Len0	0–A, where A is the bit location of the LSB of the DRAM chip bits.	0	RW	Number of bits in 0th segment

TILING_BAOP(0-6)

Seven pairs of registers are used to enable and define the address tiling functions. The register pairs are located on 32-byte boundaries starting at address 0x120. Within each pair, the *TILING_SWAP(n)* register controls the bit-swapping function, and the *TILING_BAOP(n)* register controls the bank address transformation function.

Note: Any write to a tiling configuration register must be done with full data byte enables. That is, the entire register must be modified in a single write. Partial writes into a tiling configuration register are not supported. If the sys data width is less than 64 bits, consecutive writes must be used to write all 64-bits of the tiling register.

The purpose and use of each of the register fields are described in further detail in the “Address Tiling” section.

Table 33 TILING_BAOP(n)

MEMMAX__MEMMAX__TILING_BAOP(0-6)

Bits	Field	Range	Default	R/W	Description
63:23	Reserved	N/A	0	N/A	Reserved
22:22	disable_bit_move	0/1	0	RW	Disables bit move during BAOP operations when set to 1
21:19	high_en	Full	0	RW	Bit mask with BAOP.width significant bits. A bit of this vector is set to 1 if the corresponding bit in horizontal tiling should be part of the BAOP function
18	Reserved	N/A	0	N/A	Reserved
17:15	low_en	Full	0	RW	Bit mask with BAOP.width significant bits. A bit of this vector is set to 1 if the corresponding bit in vertical tiling should be part of the BAOP function
14	Reserved	N/A	0	N/A	Reserved
13:8	high_loc	0, or (D < L_0 < A), where D is the base-2 logarithm of the DRAM block size minus 1, and A is the bit location of the LSB of the DRAM chip bits.	0	RW	Bit position of the LSB of the high OP bits.
7:2	low_loc	0, or (D < L_0 < A), where D is the base-2 logarithm of the DRAM block size minus 1, and A is the bit location of the LSB of the DRAM chip bits.	0	RW	Bit position of the LSB of the low OP bits.
1:0	width	Full	0	RW	Width of the low and high OP bits.

7 Tuning Guide

The MemMax Scheduler is an intelligent DRAM scheduler that provides a considerable degree of flexibility to the system designer. This section presents strategies for tuning the configuration of the scheduler to optimize performance and to assist the designer to quickly achieve the desired timing and area closure.

This section first describes aspects of MemMax DRAM System configuration that affect memory efficiency in “Memory Efficiency Optimization”. Factors that affect the frequency of operation and timing closure are discussed in “Frequency Optimization”. Configuration choices that affect area are described in “Area Optimization”. A step-by-step guide to tuning the MemMax Scheduler for specific goals is provided in “Step-By-Step Tuning Guide”.

The appendixes provide additional material useful for MemMax performance tuning as follows:

- Appendix A: Using Compiled Memory discusses how to add compiled memory to the MemMax Scheduler.
- Appendix B: Fall-Through Latency summarizes the fall-through latencies for the various supported configurations of the MemMax Scheduler.

Memory Efficiency Optimization

The scheduler follows two general strategies to improve DRAM memory efficiency: it attempts to minimize page miss penalties and direction turnarounds. Page misses and changes in request direction are costly in terms of DRAM cycles and in terms of additional latency reflected back to the initiators. In this section we discuss approaches to improving the scheduler’s ability to follow through on these strategies.

Minimizing Page Misses

The MemMax Scheduler assumes that when a page in a DRAM bank is opened, it stays open until another transaction to a different page in the same bank is issued, at which point the open page is closed and a new page opened. The

scheduler maintains an internal estimate of the current page hit/miss status for each bank. The internal estimate is updated as each request is issued, but may differ from the actual page hit/miss status at the DRAM controller due to one or more of the following events:

- DRAM page refresh, which requires open pages to be closed. As the scheduler is not responsible for DRAM refresh, it cannot know when a previously-open page has been closed for refresh.
- A write to the *DRAM Config* register, which causes the scheduler's internal page status tracking registers to be reset. The first request to each bank following a *DRAM Config* register update is therefore assumed to be a page miss, irrespective of the actual page status at the DRAM.
- Out-of-order request processing by the DRAM controller. The scheduler assumes that requests sent to the DRAM controller are processed in-order. If the controller processes requests out of order, the scheduler's estimate of page status may be invalid.

The MemMax Scheduler attempts to estimate open pages conservatively. That is, while there might be cases where the MemMax Scheduler assumes a page is closed when it is actually open, as long as the refresh command does not open a new page, there will not be cases where the MemMax Scheduler assumes that a page is open when it is actually closed and a different page is open.

Typically, DRAM controllers maintain a deep command queue, which allows the controller to "scan ahead" and issue precharge/activate commands before the DRAM request arrives at the head of the command queue. Hence, by the time a request makes it to the head of the command queue, the precharge and activate latency overhead is paid for.

The MemMax Scheduler will not send command requests to the controller while *SThreadbusy* is asserted. To maintain a full command queue at the DRAM controller, the controller should de-assert the *SThreadbusy* signal as soon as the Refresh command is issued.

Open Page Mode and Close Page Mode

MemMax can expect the memory controller to operate in two modes: open page mode and close page mode.

- The open page mode is where once a page is opened in a bank, it is kept open until a page-miss occurs in that bank, at which time the precharge operation is performed and the bank becomes busy for some duration. This is desirable when you expect both temporal and spatial locality within the accesses to a bank.
- However, not all traffic flows exhibit such behavior, a good example being traffic from CPUs. While a cache-reload from CPU would use a burst to load a cache-line of data from the DRAM, a subsequent miss likely accesses a different page in the memory. In this situation it is more efficient for the DRAM system to operate in a close page mode for such accesses. In this case, the bank is automatically precharged (the page is closed) after each access. This potentially reduces the average latency encountered in memory accesses because when a later request arrives at the bank to open a new page, it does not have to first wait for a precharge activity. Furthermore, it allows a reduced utilization of the memory command bus, potentially avoiding pipeline stalls while a new precharge command is issued.

Bank Busy Tracking

Activating a new page within an active bank is more expensive than opening a new page in an inactive bank. The *SFlag* signal can be used to pass bank-busy information from the DRAM controller to the scheduler, allowing the scheduler to track bank status. The scheduler will not issue a page miss request to a bank that is marked busy in *SFlag*, although it may send page hit requests to that bank.

To use the bank busy bits, ensure that *SFlag* is enabled on the *mem* OCP interface and has the correct width and ensure that the four bankbusy timing configuration register values are all zero. Table 9 presents the parameters used to configure *SFlag*. (This assumes that the controller provides bank busy bits in the correct format for use by the scheduler, as discussed in “Bank Busy Flags”.)

When used correctly, the *SFlag* signals can help in avoiding back-to-back page misses to the same bank. In general, when the controller receives a request, it should assert the *SFlag* bit corresponding to the bank that contains the target address. The bit should be asserted for at least the number of cycles required to precharge and activate the bank, and deasserted when there is no pending request to the bank in the command queue of the controller.

The use of *bankbusy* signals is optional. They can be configured to not exist, and if present can be disabled at run-time. The MemMax Scheduler has the ability to internally predict when banks are busy. Values programmed into the configuration registers indicate the number of MemMax “memory-interface” clock cycles that a bank is considered “busy” immediately after an access. If a non-zero value is programmed into any of the registers, the internal prediction mechanism is used.

If all registers are programmed with a zero value, the bankbusy status is used from the *SFlags* signals, if present. The registers are `bankbusy_twrite` and `bankbusy_tread`.

In order to achieve appropriate bankbusy prediction, registers must be programmed with the number of cycles after an access that the bank should be considered busy and unable to immediately service a miss. The values to program into these registers are as follows:

- `bankbusy_twrite = tRP + tRCD + CL + BL/2 + tWR -1`
  This is equivalent to the row-cycle time for writes - 1.
- `bankbusy_tread = tRC -1`
  This is equivalent to the row-cycle time for reads - 1.

In these formulas, the units of measurement are MemMax memory-interface clock cycles. The abbreviations in the equations above are defined as follows:

`tRP`	Row precharge time, required between the issuing of the precharge command and opening of the next row.
`tRCD`	RAS (Row Address Strobe) to CAS (Column Address Strobe) is the delay time required between the opening of a row of memory and accessing columns within it.
`CL`	CAS (Column Address Strobe) Latency is the time between sending a column address to the memory and the beginning of the data in response. This is the time it takes to read the first bit of memory from a DRAM with the correct row already open.
`BL`	Burst Length

`tWR` WRITE recovery time is the time that must elapse between the last write command to a row and precharging it.

`tRC` Row cycle time (`tRAS` + `tRP`).

`tRAS` Row Active Time is the number of clock cycles required between a bank active command and issuing the precharge command. This is the time needed to internally refresh the row and overlaps with `tRCD`.

Address Tiling

The user can define up to seven address tiling functions that re-map request addresses such that a sequence of request addresses will hit the same page, rather than a sequence of different pages, which in turn minimizes page misses. The address tiling algorithm, and the parameters that control it, are discussed in detail in “Address Tiling”.

In general, address tiling is particularly useful for BLCK bursts where the stride between rows is greater than one page. A properly-written address tiling function will allow the BLCK request addresses to be transformed so that multiple rows fall in the same bank and page. For address tiling to work well, it is also important that the stride between rows (*MBlockStride*) be a power of two.

For example, given a stride of 0x1000, a good tiling function would be:

```
TiledAddr[ 7:0]  = MAddr[ 7: 0]
TiledAddr[11:8]  = MAddr[14:11]
TiledAddr[14:12] = MAddr[17:15]
TiledAddr[17:15] = MAddr[10: 8]
TiledAddr[AddrWidth - 1: 18] = MAddr[AddrWidth - 1 : 18]
```

The values of ILOC corresponding to this function are shown in the table below.

Table 34 Example Address Tiling Configuration Settings

Bit	Value	Bit	Value
Bit-Swapping Function Parameters			
Len0	8	ILOC1	11
Len1	4	ILOC2	15
Len2	3	ILOC3	8
Len3	3	ILOC4	18

Thread Behavior

In general, two different threads accessing the same bank will cause a high page miss rate. Therefore, improving the spatial locality within a thread is an important factor in improving the efficiency of the system.

As a first step, the designer should try to ensure that the page hit rate for each thread is maximized and the direction turnaround is minimized. Initiators such as video decoders and display drivers typically generate such correlated traffic. However, initiators such as CPUs generally do not, which will eventually cause some degradation in the throughput.

Thread Merging

If multiple initiator threads are merged into the same target thread, different traffic profiles will be mixed. Ensuring a high burst length (for example, burst lengths of 16–32) will offset the performance loss.

BLCK Bursts

BLCK write bursts within a Sonics interconnect may be chopped into rows. Hence, if the BLCK write burst is merged with another thread, it can cause the locality of the traffic within the BLCK burst to be lost. Avoid merging threads with BLCK write bursts with other threads.

Bank-Level Parallelism

In general, the higher the bank-level parallelism, the higher DRAM efficiency.

- If consecutive bursts in a single thread are not directed to the same page, they should be directed to a different bank. This minimizes the cost of activating a new page in the same bank, which is more expensive than activating a new page in a different bank.
- Across threads, it is preferable to minimize requests to the same bank.

There is a performance relationship between thread-level parallelism and bank-level parallelism. In general, increasing the number of threads increases the effectiveness of bank-level parallelism, and consequently, leads to better performance. However, performance can be reduced by the following:

- If the number of threads exceeds the number of available banks, performance will decrease. (Increasing the number of DRAM chips increases the number of banks, but the scheduler is limited to a maximum of 16 threads.)
- Traffic that is not evenly distributed among the multiple threads causes a reduction in efficiency from the use of bank-level parallelism.

Buffer Depth

The user can configure the depth of the internal buffers used to hold requests and read and write data. Each buffer can be sized individually; increasing buffer depth generally improves performance at the cost of increased area.

Typically, the request buffer is small, with a depth of 2–8 entries per thread, and is implemented using flops. The write and read data buffers are generally larger, and can be implemented using either flops or compiled memory.

Request Buffer Depth Sizing

The request buffer holds *sys* OCP bursts in single request, multiple data (SRMD) sequence. Even if the input burst is multiple request, multiple data (MRMD), the scheduler converts it to SRMD before storing it in the request buffer. The scheduler performs burst translation, which causes a single burst input on the *sys* interface to be translated into multiple requests on the *mem* interface.

For optimum DRAM performance, the scheduler should keep the command queue of the DRAM controller as full as possible. Therefore, if the burst sizes are

large (each *sys* burst is multiple DRAM burst lengths long), the request buffer need only be small (for example, two entries). However, if the expected burst size is small (only one or two DRAM burst lengths long), a deeper request buffer will enable it to accumulate more bursts.

Larger request buffer depths reduce the maximum operating frequency of the MemMax Scheduler, as they translate into larger muxes feeding the burst-conversion logic.

The request buffer depth can be configured on a per-thread basis, allowing the depth to be matched to the burst sizes of the requests arriving on the *sys* interface for each thread.

Write Buffer Depth

The write buffers accumulate the write data for each request. When all of the write data for a request has arrived, the write request can be scheduled and passed to the controller. The write buffers tend to fill quickly due to the non-deterministic behavior of the underlying DRAM. Thus, larger write buffers may be needed to ensure the desired write throughput.

The size of the write buffer can be configured on a per-thread basis. Assigning the maximum depth (256) will allow the scheduler to issue consecutive requests from the same thread, if this is the best scheduling decision in a given cycle.

Smaller write buffer depths can be allocated if it is determined that this does not affect performance.

Read Buffer Depth

A thread's read buffer must at least be `dram_block_size` entries deep, and should be as large as the largest *sys* burst arriving on that thread. The scheduler reserves slots in the read buffers before sending the read requests to the DRAM controller. To allow read requests from a thread to be sent back-to-back, the read buffer should be deep enough to hold at least two maximally sized (for the thread) *sys* bursts.

Smaller read buffer depths can be allocated if it is determined that this does not affect performance.

Interlock Depth

The `set_interlock_depth` parameter enables the scheduler to send a new write request to the controller before the last data phase of the previous write request has been completed. The interlock depth specifies the number of write requests that can be sent before the interlock is activated. When the interlock is activated, a new write request cannot be sent until all of the write data for the preceding request has been sent.

For information about setting parameter values for this feature, see "Write Request Interlock Depth".

Quality of Service Support

The MemMax Scheduler provides support for three quality-of-service (QoS) levels: priority, allocated bandwidth, and best effort. These QoS levels are

intended to provide the best match to three commonly occurring types of data traffic: traffic that has low latency requirements, traffic that requires a certain sustained bandwidth with low jitter, and traffic that does not have any QoS requirements, respectively. Selecting and setting the QoS level for each of the threads in a system should be done carefully—in general, use of priority or allocated bandwidth QoS levels automatically reduces memory efficiency.

The three configuration parameters that affect the QoS of each thread and hence the overall memory system efficiency are:

- The QoS mode of the thread
- The value of `bandwidth_rate`
- The value of `words_per_period`

The configuration syntax for these parameters is presented in “Bandwidth Allocation”.

The scheduler ensures maximum memory utilization when the QoS modes of all threads are set to best-effort (turned off). If all threads are best-effort, the value of request group size per thread should be set to 0.

When the QoS level must be set to either priority or allocated bandwidth, the following factors should be considered:

1. The value of `bandwidth_rate` (per thread) should be inversely proportional to the ratio of the bandwidth on the thread to the total bandwidth requested by the system on the DRAM channel. For example, if the peak bandwidth of the system is 2 GB/s, and thread *T* requires 200 MB/s, then `bandwidth_rate<T>` should be set to 10 when `bandwidth_rate_high_precision==0`. If `bandwidth_rate_high_precision==1`, this would be set to (200/2000)*256
2. The `words_per_period` parameter should be set by choosing the minimum scheduling period over which each thread requests data transfers within its bandwidth requirement. For example, if each thread requests bandwidth according to its bandwidth rate in a period of 1000 cycles, the `words_per_period` value of each thread should be set at 1000/ `bandwidth_rate` of the thread.
3. The request group size indicates the number of requests serviced consecutively on a thread. The choice of the request group size per thread is dependent on the burst length of the *sys* burst on the thread and the traffic profile. As a first approximation, it should be set to the number of requests it takes to hide a page miss in the memory controller. For example,
   - Assume that *SFlag* is enabled and used in accordance to the suggestions above.
   - Assume a precharge latency of 10 ns, and activate latency of 10 ns. (These values are chosen for simplicity. Actual values will depend on the type of physical memory.)
   - Assume that the DRAM controller operates at 266 MHz, and issues a read or write command every two cycles.

   To hide the page miss, the precharge and activate latencies must be hidden, which is a total of 20 ns. A request group size of 2 (that is, 3 requests) will ensure that the page miss latency is hidden.

Sys OCP Burst Sizes

The *sys* OCP burst sizes are a significant factor in MemMax Scheduler performance and area trade-offs. As a rule of thumb, larger *sys* OCP burst sizes are better for the scheduler as they are more likely to target the same page of the DRAM, increasing the page hit-to-miss ratio. Larger *sys* bursts also reduce the required depth of the request buffer, reducing area.

On the other hand, if a *sys* burst is shorter than the DRAM burst length, it will significantly affect the DRAM efficiency, as idle cycles will be introduced due to the access granularity requirements of the DRAM.

Frequency Optimization

In this section, we describe the factors that have the most significant effect on timing.

Within the MemMax Scheduler the most significant sources of timing sensitivity are:

1. The request scheduler path,
2. The burst-conversion logic, and
3. The address tiling and page filter logic.

The request scheduler path is sensitive to the number of threads and per-thread QoS settings. The burst-conversion logic is sensitive to the scheduler's configuration settings. The tiling and page-filter logic is sensitive to the use of the BAOP operation in the tiling function. Factors affecting timing sensitivity for each group are discussed in greater detail in the following sections.

Request Scheduler Path

The request scheduler path consists of thread scheduling, followed by updating of internal state, and sending the request to the memory controller. The thread scheduling part is influenced by the following:

Once-in-Two-Cycle Scheduling

The MemMax Scheduler supports a once-in-two-cycle scheduling feature to improve timing. When enabled (by setting the parameter `two_cycle_sched` equal to 1), a flop is introduced into the internal scheduler paths between the arbitration and the update phases.

Use of this feature is the easiest way to improve the timing of the scheduler path. However, if there are significant number of *mem* OCP bursts whose burst length (*MBurstLength*) is less than 2^*DRAM_BLK_LOGSIZE*, the efficiency of the DRAM will be negatively affected.

Number of Threads

Reducing the number of threads in the scheduler allows a higher operating frequency to be achieved. As discussed above, avoid merging threads that require different QoS levels.

The best way to reduce the number of threads is to map all best-effort initiator threads to a single MemMax Scheduler thread, and map initiator threads that require controlled bandwidth and priority QoS levels to separate threads.

Reducing the number of threads can lead to a potential loss in performance in DRAM system efficiency. Therefore, the choice of the number of threads must be made carefully.

Burst Conversion Path

The burst conversion path is enabled whenever there is a translation according to the conditions summarized in Table 15. Burst conversions corresponding to a *DRAM_BLK_LOGSIZE* of two require more complex logic than conversions corresponding to a *DRAM_BLK_LOGSIZE* of one, resulting in a lower maximum frequency limit.

To remove burst conversion from the critical timing path, make the registers read-only and configure the system such that *DRAM_BLK_LOGSIZE* is less than two.

If alternate-cycle scheduling is enabled, the burst conversion path does not present any timing issues and frequencies of up to 533 MHz can be achieved.

Page Filter and Address Tiling

The chopping logic consists of the address tiling and page filter calculation. If these components fall on the critical timing path, the designer must disable the BAOP operation associated with address tiling.

To disable BAOP operation for a specific address tiling function, set the *BAOP* register fields to zero and make them read only.

Area Optimization

MemMax Scheduler features and configuration settings that have the largest impact on area are: read buffer depth, write buffer depth, interlock depth, WRAP and XOR burst support, the number of threads, and request buffer depth.

Typically, the request buffers are small, and, even though implemented using flops, will not contribute greatly to the area. Each of the remaining significant contributors to area are discussed separately below.

Read and Write Buffers

Tuning of the read and write buffers based on performance was discussed earlier. By default, the MemMax Scheduler instantiates flop-based memories for these buffers. Use of compiled memories will significantly reduce the area required by these buffers compared to flop-based designs.

Interlock Depth

The setting of the `set_interlock_depth` parameter affects the size of the internal write buffer. The effect of modifying the interlock depth is discussed in “Interlock Depth”.

WRAP and XOR Burst Support

If WRAP bursts are supported, a burst conversion logic instantiated along with additional buffers in the response path increases the area overhead. Similarly, if XOR bursts are supported, and the DRAM type is DDR1 or RW (the register field can be modified at boot time or run time), residue buffers are instantiated, which contribute to area.

Number of Threads

Minimizing the number of threads can greatly reduce the area of the scheduler: the number of threads in the design has a quadratic effect on the number of comparators needed in request arbitration, and each thread requires a separate request, read, and write buffer.

The number of threads can be kept low if the length of incoming *sys* bursts are large. In this case, priority traffic can be assigned separate threads, while the allocated bandwidth and best-effort traffic can be merged at different levels of granularity. The advantage of large burst sizes (here, “large” means about eight times the DRAM burst length) is that MemMax Scheduler can issue multiple requests to the same bank and page, thus maximizing page hits, and then move on to a different bank which, even if it is a page miss, will not cause a significant drop in performance.

Run-Time Flexibility Versus Timing and Area

The MemMax Scheduler offers run-time flexibility by providing configurable registers that can be set at design-time and modified at run-time. However, run-time configuration requires additional logic (particularly muxes) and hence an increase in area.

Avoid superfluous run-time configurable parameters: set the access for all but essential parameters to read-only.

Table 35 presents the summary of area overhead associated with the addition of different MemMax Scheduler features.

Table 35 Area Estimates By Functionality

Feature	Area Overhead	Width	Compiled Memory
Number of threads = N	Approximately N^2 comparators	NA	No
Bursts with only straight translation	No extra buffers	NA	NA
Aligned INCR	No extra buffers	OCP data width	
Only-0 INCR	For actual values, see Appendix A: Using Compiled Memory	OCP data width	Yes
Even XOR and Even WRAP or only 0 INCR translation	For actual values, see Appendix A: Using Compiled Memory	OCP data width	Yes

Feature	Area Overhead	Width	Compiled Memory
DRAM Type = DDR-2 or DDR-3 and Read Only	No extra buffers	NA	NA
XOR bursts with DRAM Type = DDR-1, DRAM BURST TYPE = Sequential, DRAM_BLK_LOGSIZE = 2	For actual values, see Appendix A: Using Compiled Memory	OCP data width	Yes
Interlock Depth	N + 1 entries, where N is the interlock depth	Log_2 (Number of threads) + DRAM_BLK_LOGSIZE	No
Tiling SWAP Operation	Area overhead is K M-bit muxes, where: M = min(Address width-1, Chip bit location-1) when chip_bits=1) K = (Number of tiling functions enabled)* (Number of threads) If any bit in the SWAP register is writable, 6M flops are added.	NA	No
Tiling BAOP Operation	6 M-bit muxes	NA	No
Two cycle scheduling	Two buffers per thread	sys OCP addr width	No
Two cycle scheduling	Two buffers per thread	4 bits	No
Pipeline Threadbusy	One extra buffer if pipelined MThreadbusy is asserted	OCP data width	No

Step-By-Step Tuning Guide

This section contains a step-by-step guide to tuning the MemMax DRAM System that distills the various trade-offs discussed previous section. Following this guide should enable you to rapidly tune the MemMax Scheduler to meet the desired goals of performance, frequency, and area.

Step 1: Configuration Checks

The following basic checks help to ensure that the requests issued by the MemMax Scheduler are optimal for the attached DRAM controller.

1. Check that the value of the parameter `bank8` is consistent with the number of banks supported by the DRAM. Set `bank8` to 1 if the DRAM supports eight banks, and to 0 otherwise.
2. Enable *SFlags* in the *mem* OCP interface and set the width of *SFlags* to 8*C* (`bank8` is set to 1) or 4*C* (`bank8` is set to 0), where *C* is the number of chips.

3. Check that the value of the parameter `bank_bit_location` is consistent with the position of the bank bits in the DRAM. (If the corresponding register field, *`BANK_BIT_LOCATION`*, is used and marked read-write, ensure that the value of the register field is also consistent.)
4. Check that the `dram_block_size` parameter and *`DRAM_BLK_LOGSIZE`* register field are consistent with the DRAM burst length. As a general rule, the product of DRAM block size, OCP data width, and **mem** OCP frequency must equal the product of DRAM burst length, DRAM data width, and DRAM bus frequency.
5. Check that the *MAddrSpace* signal is set properly at the *sys* interface. The *sys* burst arriving at the MemMax Scheduler should have the signal *MAddrSpace* set to 0 if the request is going to a MemMax DRAM System configuration register, 1 if the request targets the DRAM, and a number *N* (with $1 < N < 8$) if the request targets the DRAM and a MemMax Scheduler tiling function is to be invoked.
6. Check that `dram_block_size` and `2^`*`DRAM_BLK_LOGSIZE`* are set to values greater than or equal to the value of the `tag_interleave_size` parameter on the **sys** interface.
   This check is for multi-tag configurations with tag_parallelism > 1.

Step 2: Performance Tuning

The flowchart in Figure 17 presents the steps to be taken for performance tuning. The approach is to first tune for bandwidth requirements of the initiators, followed by latency minimization of priority traffic, followed by jitter minimization of controller bandwidth traffic, and then finally area minimization.

There are three pertinent algorithms referred in the flowchart:

1. Thread merging algorithm,
2. QoS setting algorithm, and
3. Buffer depth optimization algorithm.

In addition, bandwidth should be calculated on a per-thread basis.

Latency should also be measured, including not just network latency, but the latency from when the request was ready to be sent (for example, the request time specified in the stimulus file) to the time when the last response was received. It may be necessary to instrument the initiators to properly measure latency.

Thread Merging Algorithm

As a rule, two threads can be merged only if their programmed QoS level is the same. That is, a priority thread should only be merged with another priority thread, and so on.

Threads can then be merged as follows:

1. In the current configuration, identify two best effort threads, and merge them. Merge the threads in the following order:
   a. Never merge a thread producing BLCK write bursts with any other thread.

b. Merge two threads with large average burst size.

c. Merge two threads where one thread has large burst size and the other thread has small burst size.

d. Merge two threads where the burst sizes on both threads are small.

2. If there are no best effort threads that can be merged, merge two allocated bandwidth threads, using the same four steps (a)–(d) described above.

3. If there are no controlled bandwidth threads to be merged, merge two priority threads using the four steps (a)–(d) described above.

QoS Setting algorithm

Pick a thread with programmed QoS level equal to 0 (best-effort) and where the average latency requirement has been exceeded most often. Set the QoS level for this thread to 2 (priority) if it is a priority thread or 1 (allocated bandwidth) if it is a allocated bandwidth-thread. Set the parameters `words_per_period`, `bandwidth_rate`, and request group size as described in the QoS analysis section.

Buffer Depth Optimization Algorithm

There are three types of buffers whose per-thread depths should be optimized: the request buffer, the write data buffer, and the read data buffer. The Sonics Performance Analysis (SPA) environment can be used to optimize the depth of each buffer as follows:

1. Run the `soccomp` tool with the database option enabled (using the options **-trid -scv**)

2. Launch the SPA shell (these commands assume a C-style shell):

```
scmain --perfmon=1 --dbtype=snxsdb
perf_shell%
perf_shell% set db [read_db mydesign_uc0_sysc.sdb]
```

   It will return a value, say 0.

3. Now execute a selection command on the database:

```
perf_shell% execute_p 0 "select * from generator"
```

   Now, note the rows that have `memmax command buffer` with the desired thread ID (Analysis should be done on a per-thread basis). Note the generator ID of these rows, which is the second column.

4. For each generatorID of interest (for example, N), perform the following command:

```
perf_shell% execute_p  0 \
   "select * from Sonics_QueueMonitor where generatorID = N"
```

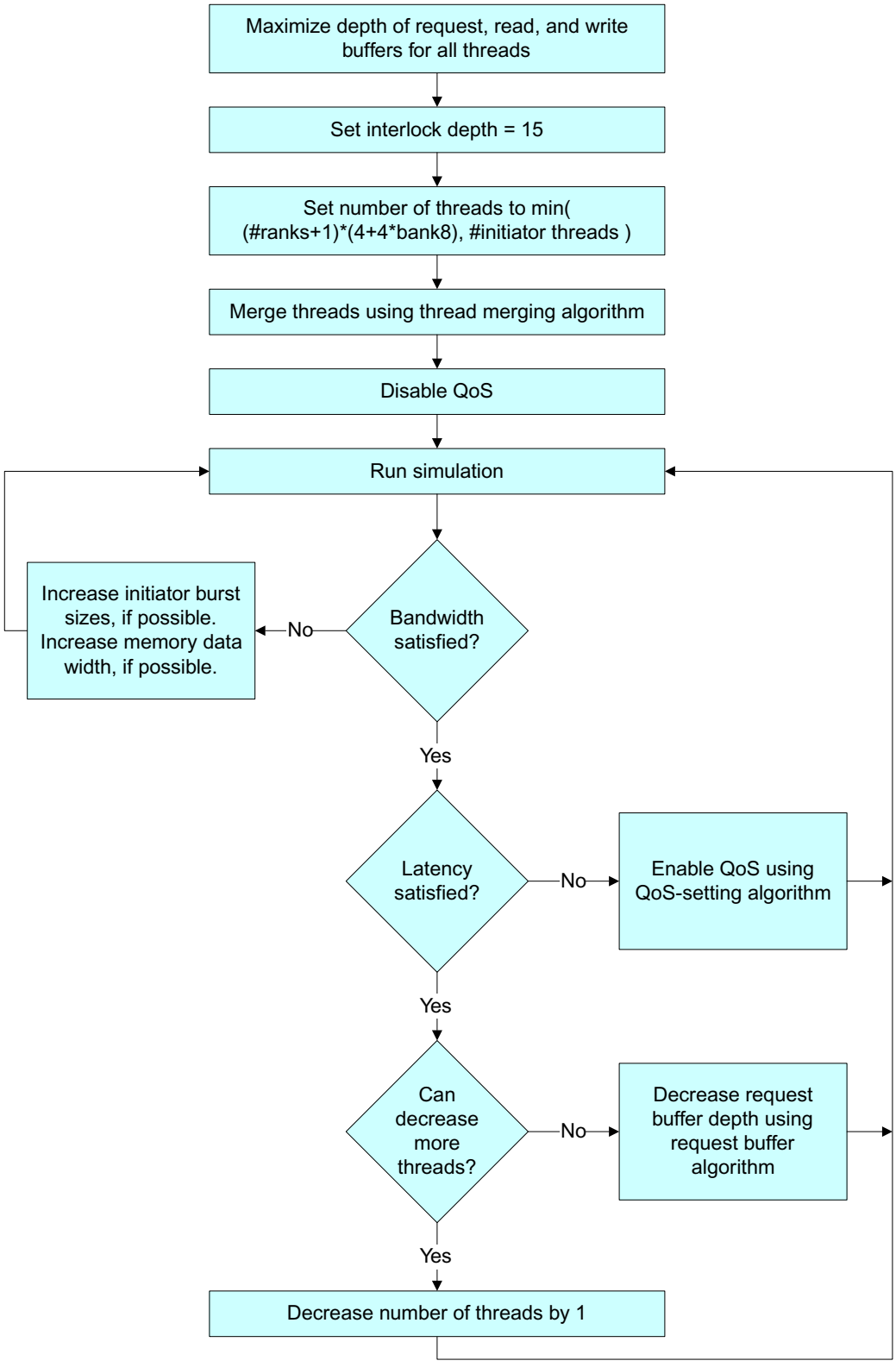
   In the result set, check if either maximum value of the `begin-value` field or the maximum value of `end-value` is equal to the size of the per-thread request buffer. If it is equal to the per-thread request buffer, do not change the buffer depth. Otherwise, set the buffer depth to the maximum value plus one.

Repeat the above for the `write_data_buffer` as well.

For more information about using the Sonics Performance Analysis environment, see the *SPA User Manual*.

For the read data buffers, it is not only important to measure the occupancy, but to have a notion of the number outstanding responses as well. The scheduler reserves space in the buffers before issuing requests. Therefore, when a *sys* request is sent to the scheduler, it can be assumed that the number of slots corresponding to the *MBurstLength* of the *sys* request will be filled.

Figure 17 Performance Tuning Flowchart



Step 3: Timing Closure

For timing closure, consider the following questions:

Question:
Is the external DRAM going to change for the same chip for multiple derivatives?

Answer: Yes.

Implication:
DRAM_BLK_LOGSIZE and *DRAM_BURST_TYPE* register fields should be read-write. This will affect timing if `two_cycle_sched` is not enabled. However, if *DRAM_BLK_LOGSIZE* = 2 and is read-only, it will still affect the timing.
If DDR1 is likely to be used and if WRAP burst sequence is not enabled in the *sys* OCP interface, the DRAM_TYPE should be read-write. In order to attain better timing closure, once-in-two-cycle scheduling should be enabled, if possible.

Answer: No

Implication:
All *DRAMCONFIG* registers fields should be set read-only.

Question: Will the QoS registers be programmed at runtime?

Implication:
Keep only those registers that will be changed during runtime, specifically read-write.

Question:
How many tiling functions are used, and are they runtime configurable

Implication:
Keep the number of tiling functions small and configurable, rather than large and read-only. Those which are not used should be read-only. If all tiling functions are programmed read-write, this will result in lot of area overhead. Even if all tiling registers are enabled and read-only, it will incur area and timing overhead. Typically, three tiling functions (*MAddrSpace* width of 2) should suffice.

Step 4: Area Minimization

The following guidelines should be kept in mind for area minimization:

1. If possible, compiled memory should be used for read and write buffers, as they are typically large.
2. The request buffer is usually small, and its size should be kept small (not more than 8 entries). If the *sys* burst sizes are large, reducing the size of request buffer will not have a significant effect on performance. Hence, if the thread sees a majority of bursts whose burst length is 32, the request buffer depth can be kept small, at 1–2 entries.
3. There will be area overhead incurred when WRAP is enabled in the *sys* interface.
4. Enabling `two_cycle_sched` will incur area overhead due to the addition of per-thread flops to pipeline the design.

5. If XOR burst is enabled in the *sys* interface, and DRAM type is DDR1 or RW, residue buffers will be used which will incur extra registers.

A Using Compiled Memory

This appendix provides proves an overview about the support for compiled memory, the configuration of memory, and the steps necessary for using your own complied memory.

Compiled Memory Support

The MemMax Scheduler may be configured to use compiled, third-party memory or flip-flops for internal data buffers. Incorporating compiled memory for data buffers in the scheduler is an efficient method of saving area and power.

Compiled memory may be made from either dual-port synchronous SRAM or a two-port synchronous register file. In either case, the compiled memory must satisfy the following requirements to be used with the MemMax Scheduler:

- Must provide dual ported behavior, with dedicated read and write ports
- Must provide a read latency of one clock cycle (read data appears one cycle after the read request)
- Must provide a write-to-read latency that is less than or equal to one cycle

The amount of scheduler buffering for each of the data buffers depends upon the configured number of threads, the buffer depths, and the OCP data width.

Memory Depth

The depth of the compiled memory for the write data buffers, in OCP words, is:

Figure 18

$$\text{memory depth} = \sum_{0}^{\text{threads}-1} \text{write buffer depth for thread i} + 2K$$

The depth of the compiled memory for the read data buffers, in OCP words, is:

Figure 19

$$\text{memory depth} = \sum_{0}^{\text{threads}-1} \text{buffer depth for thread i}$$

Depending on the configuration of the MemMax Scheduler, a residue buffer may also be instantiated to hold data as a result of burst conversion. The residue buffer depth is non-zero if either of the following are true:

- XOR bursts are enabled on the *sys* OCP interface and the DRAM block size is four and the DRAM type is DDR1. In this case, *K*=1 in equation Figure 18; in other cases, *K*=0.
- WRAP bursts are enabled and either Even Translation or Only-0 INCR translation are used as specified in Table 15.

The depth of the residue buffer in OCP words is then given by:

Figure 20

$$\text{residue memory depth} = \sum_{i=0}^{\text{threads}-1} R_i$$

Where:

Figure 21

$$R_i = \text{ceil}\left(\max\left(2, \left(\frac{\text{read buffer depth}_i}{\text{DRAM block size}}\right) \times (\text{DRAM block size} - 1)\right)\right)$$

When the parameter `mem_clock_async` is enabled, the MemMax Scheduler uses FIFO-based synchronization of the *sys* and *mem* clock domains, which requires the depth of the buffers to be even. Therefore, if the residue buffer depth evaluates to an odd number, an extra entry is added to make the depth even.

Memory Width

The required widths of the compiled memory for the write and read buffers is summarized in Table 36. Some compiled memories require the width to be a multiple of 8 bits. For this reason, byte enables for writes are stored separately within the scheduler when the OCP data width is 16 or 32 bits. When the OCP data width is 64, 128, or 256 bits, the scheduler stores the byte enables with the write data in the compiled memory.

Table 36 Compiled Memory Width Requirements EIGHT_MULTIPLES_ONLY enabled

OCP Data Width	Write Buffer Width	Read Data Buffer Width	Write Buffer MDataByteen Handling
16	16	16	MDataByteen values stored separately from write data, because width is less than 8 bits.
32	32	32	

OCP Data Width	Write Buffer Width	Read Data Buffer Width	Write Buffer MDataByteen Handling
64	72	64	MDataByteen values stored with write data, because width is multiple of 8 bits.
128	144	128	
256	288	256	

The width of the buffers inside the MemMax Scheduler is configuration dependent. the MemMax Scheduler uses the parameter `eight_multiples_only` to generate SRAM instantiations that are only multiples of 8. If the parameter is disabled, the MemMax Scheduler assumes that non-multiple of 8 SRAMs can be supported, and instantiates the SRAM for all bits in the Read and Write buffers.

The MemMax Scheduler stores data byte enables separately within the scheduler when the OCP data width is 16 or 32 bits and `eight_multiples_only` is enabled (set to 1). When the OCP data width is 64, 128, or 256 bits, the scheduler stores the data byte enables with the write data in the compiled memory.

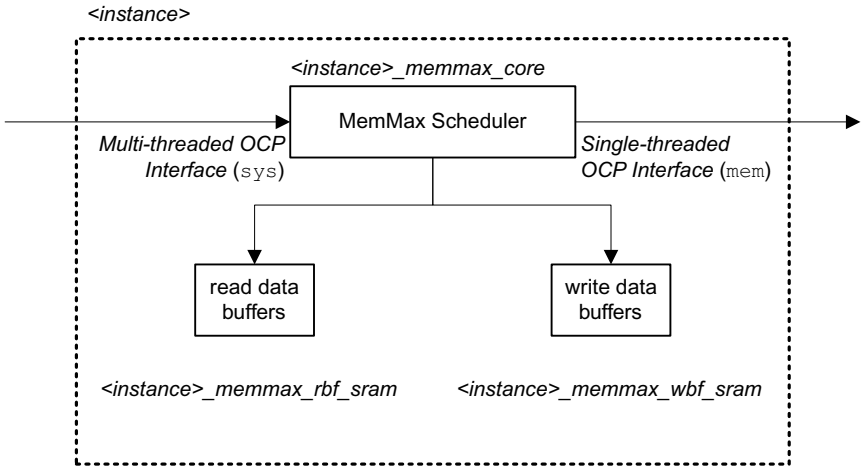
Configuring Memory

The configuration parameter `compiled_mem`, described in “Compiled Memory”, is used to specify whether a flip-flop memory or a compiled memory is to be used for the read and write data buffers. When `compiled_mem` is set to 1, the scheduler is configured to use compiled memory and the memory module must be provided by the user for simulation and synthesis.

Note: When `tags > 1` and `compiled_mem` is set, the MemMax Scheduler uses additional logic for writing and reading to and from compiled memory. As a result, the behavior of the MemMax Scheduler with and without compiled memory may not be identical.

The top level of the scheduler RTL instantiates the memory for the buffers as shown in Figure 22. The user-supplied modules must be named as shown.

Figure 22 Compiled Memory Module Naming



The user-supplied module must have the interface signal names and widths specified in Table 37. If the compiled memory being used does not employ these signal names, the compiled memory must be wrapped to convert the names. The wrapper can also be used to implement any adjustments that may be required to interface to the generated module, such as converting between interface signals with a different assertion level.

Table 37 Compiled Memory Interface

Signal	Description	Width (bits)
clk_i[1]	Clock	1
read_addr_i	Read address	ceil($\log_2$(buffer depth))
read_cmd_i	Read enable	1
read_data_o	Read data	OCP data width
write_addr_i	Write address	ceil($\log_2$(buffer depth))
write_cmd_i	Write enable	1
write_data_i	Write data	OCP data width

1 If the parameter `mem_clock_async` is set, ***clk*** will be split into two ports for read and write clocks, named ***rclk_i*** and ***wclk_i***, respectively.

By default, when compiled memory is used it is assumed to be SRAM, not register based, and hence no memory reset signal is assumed to be necessary[1]. Although the timing diagram in Figure 23 indicates a data delay, the delay is in the compiled memory module, and is not necessarily done with a register (that is, a latch).

If the parameter `compiled_mem` is set to 0, a simple NCSRAM register file is used that has the same functional timing required of the compiled memory. The signals used to interface to this memory are listed in Table 38. When the NCSRAM receives a read request, the contents of the indexed memory register array are loaded into a holding data register.

Because the non-compiled memory version is implemented using registers, these registers must observe the rules governing MemMax Scheduler registers and are required to include a reset input.

In the synchronous case (that is, the default setting of `mem_clock_async` set equal to 0), two reset signals (***wreset*** and ***rreset***) are used to drive the non-compiled memory.

When `mem_clock_async` is set to one, the read/read index and the holding register will be in the read-side clock domain and this domain will be different than that of write/write index. In this asynchronous case, a ***wreset_ni*** port must be a reset synchronous to the clock generating write/write index. All the array registers will

1 If registers are used instead of SRAM, the RTL files that comprise the MemMax Scheduler must be hand-edited to route the appropriate reset signals to the registers.

be reset by this signal. Use RTL to implement SRAM. It will generate a 2D flip-flop.

The *rreset_n* must be a reset synchronized to the read/read index and only the output holding register will be reset by this signal.

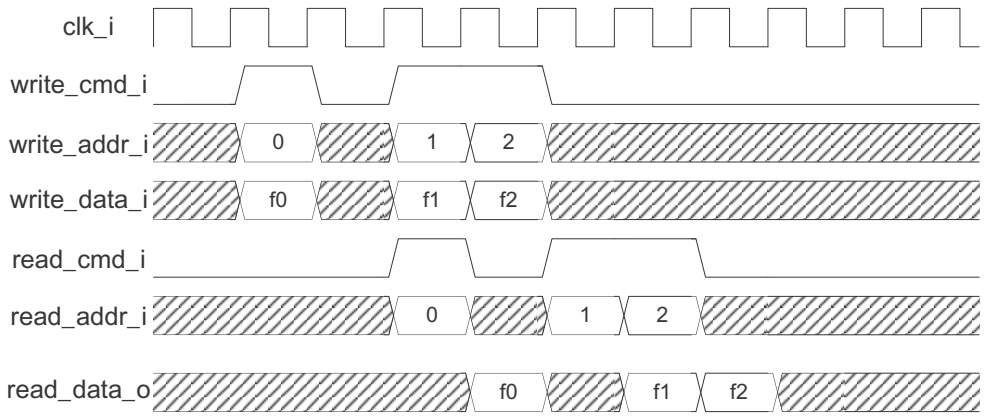
Table 38 Non-Compiled Memory Interface

Signal	Description	Width (bits)
wclk_i	Clock	1
rclck_i	Clock	1
read_addr_i	Read address	ceil($\log_2$(buffer depth))
read_cmd_i	Read enable	1
read_data_o	Read data	OCP data width plus some internal control signal width
rreset_ni	Synchronous active low reset for read	1
wreset_ni	Synchronous active low reset for write	1
write_addr_i	Write address	ceil($\log_2$(buffer depth))
write_cmd_i	Write enable	1
write_data_i	Write data	OCP data width plus some internal control signal width

Memory Interface Timing

Figure 23 shows an example of the signal timing of the memory interface.

Figure 23 Memory Timing Signals



Accessing Memory Test Features

Many embedded memory structures incorporate a user-supplied dedicated test block to assist with production testing. Test signals that support the memory

core are the responsibility of the user and would typically connect from a higher level of the chip hierarchy into the top level of the MemMax Scheduler. The `soccomp` tool does not provide any direct support for this—the user must edit the generated RTL file by hand to add any required test signals.

The design hierarchy is structured as:

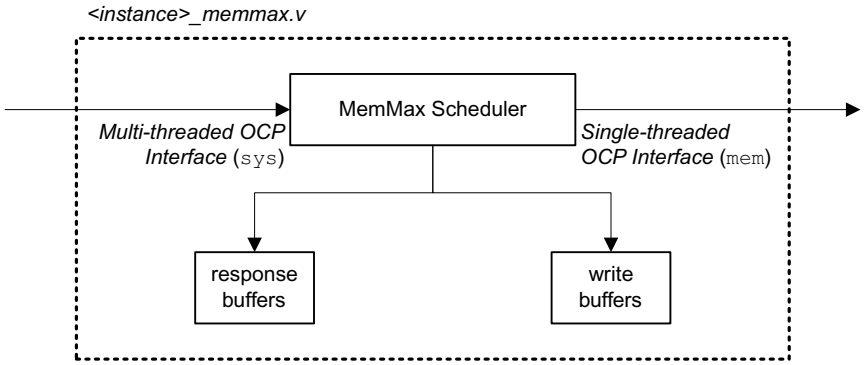
```
<deisgn_name>.v
   <instance name>_memmax.v (memory scheduler)
   <memory controller>.v (memory controller)
   <instance name>_smx.v
```

The MemMax Scheduler instance is located at:

```
<instance name>_memmax/<instance name>_memmax.v.
```

Figure 24 shows the scope of this file.

Figure 24 Accessing MemMax Scheduler Memory Test Features



To add test interfaces to the memory of the scheduler, edit the `<instance>_memmax.v` file, adding any test interfaces and updating the instantiation of the read data and write buffers compiled memory blocks. This update should take place after all configuration is complete to avoid being overwritten when the netlist is created.

Using Compiled Memory

To use your compiled memory, perform the following steps.

Steps to use compiled memory

1. Search for `rbf_sstorage` and `wbf_sstorage` in the MemMax RTL to find the compiled memory size that you will need.

   The lines where `rbf_sstorage` and `wbf_sstorage` appear inside the MemMax RTL show the memory sizes to use with your memory compiler. The following code shows an example.

```
//RBF storage
//rbf_sstorage: width = 64, depth = 32

memmax_memmax_rbf_sram #(64) rbf_sstorage
   (
    .wclk_i           ( rbf_gated_writeclk     ),
    .rclk_i           ( rbf_gated_readclk      ),
```

```
    .write_cmd_i       ( rbf_ssram_write        ),
    .write_addr_i      ( rbf_ssram_waddr        ),
    .write_data_i      ( rbf_ssram_wrdata[63:0] ),
    .read_cmd_i        ( rbf_ssram_read         ),
    .read_addr_i       ( rbf_ssram_raddr        ),
    .read_data_o       ( rbf_ssram_rddata[63:0] )
);

//WBF storage
//wbf_sstorage: width = 72, depth = 6
memmax_memmax_wbf_sram #(72) wbf_sstorage
   (
   .wclk_i            ( wbf_gated_writeclk ),
   .rclk_i            ( wbf_gated_readclk  ),
   .write_cmd_i       ( wbf_ssram_write    ),
   .write_addr_i      ( wbf_ssram_waddr    ),
   .write_data_i      ( wbf_ssram_wrdata   ),
   .read_cmd_i        ( wbf_ssram_read     ),
   .read_addr_i       ( wbf_ssram_raddr    ),
   .read_data_o        wbf_ssram_rddata    )
);

//surefire lint_off
```

2. Compile the memory with your memory compiler.
3. In the RTL output from **soccomp** look for the modules described in step 1 and replace the instantiation of modules `memmax_memmax_rbf_compmem` and `memmax_memmax_wbf_compmem` with the instantiation of your compiled memory from step 2.

   The following shows an example of a wrapper that contains the module `memmax_memmax_rbf_compmem`.

```
module memmax_memmax_rbf_sram (
                               wclk_i,
                               rclk_i,
                               write_cmd_i,
                               write_addr_i,
                               write_data_i,
                               read_cmd_i,
                               read_addr_i,
                               read_data_o);

parameter DATA_WDTH = 64;

input                  wclk_i;
input                  rclk_i;
input                  write_cmd_i;
input  [4:0]           write_addr_i;
input  [DATA_WDTH-1:0] write_data_i;
input                  read_cmd_i;
input  4:0]            read_addr_i;
output [DATA_WDTH-1:0] read_data_o;

wire   [4:0] memwrite_addr;
wire   [4:0] memread_addr;
assign memwrite_addr = write_addr_i;
assign memread_addr  = read_addr_i;
```

```
memmax_memmax_rbf_compmem mem (
    .CLKW     (wclk_i),
    .WEB      (!write_cmd_i),
    .AA       (memwrite_addr),
    .D        (write_data_i[63:0]),
    .CLKR     (rclk_i),
    .REB      (!read_cmd_i),
    .AB       (memread_addr),
    .Q        (read_data_o[63:0]),
    .WCT      (2'b00),
    .RCT      (2'b00),
    .KP       (3'b101)
```

B Fall-Through Latency

Table 39 and Table 40 present the fall through latency of the MemMax Scheduler for single-cycle and two-cycle configurations.

Table 39 Fall-Through Latency for Single-Cycle Scheduling

Feature	Fall through + latency
Request Path	
No Asynchronous boundary crossing	3 cycles (Use Request Buffer + R1 + R0)
With Asynchronous boundary crossing	5 cycles (2 cells for asynchronous boundary crossing)
Response Path—no asynchronous boundary crossing	
Response bypass disabled	1 cycle
Response bypass enabled	0 cycle
Request Path—with asynchronous boundary crossing	
Response bypass disabled	3 cycles
Response bypass enabled	Not allowed
Data_handshake Path	
Compile_sram=1	The data_handshake path needs two more cycles than the request path needs.

Table 40 Fall-Through Latency for Two-Cycle Scheduling

Feature	Fall through + latency
Request Path	
No Asynchronous boundary crossing	5 cycles (CMD Buffer + R1 + R0
With Asynchronous boundary crossing	7 cycles (2 cells for asynchronous boundary crossing)
Response Path—no asynchronous boundary crossing	
Response bypass disabled	1 cycle
Response bypass enabled	0 cycle
Request Path—with asynchronous boundary crossing	
Response bypass disabled	3 (2 cycles for async + 1 mem clock latency)
Response bypass enabled	Not allowed
Data_handshake Path	
compile_sram=1	The data_handshake path needs two more cycles than the request path needs

This document was designed for double-sided printing.
This page has been left blank intentionally so the back cover
of this document appears on a left-facing page.

670 N. McCarthy Blvd, Suite 100
Milpitas, CA 95035-5119
USA

Phone: +1 408 457 2800
Fax: +1 408 457 2899

Customer Support and Feedback
support@sonicsinc.com

www.sonicsinc.com