cadence®

SDRAM ABVIP User Guide

Product Version 11.3
May 2024

Cadence Design Systems, Inc., 555 River Oaks Parkway, San Jose, CA 95134, USA

communications. We are also active in the removal and/or replacement of inappropriate language from existing content. This product documentation may however contain material that is no longer considered appropriate but still reflects long-standing industry terminology. Such content will be addressed at a time when the related software can be updated without end-user impact.

Contents

1 5
Chapter 1 _ Overview 5
2 14
Chapter 2 _ Interface 14
3 16
Chapter 3 _ Spec Functionality 16
4 25
Chapter 4 _ Test Plan 25
5 29
Chapter 5 _ Requirements 29
6 41
Chapter 6 _ Strategy 41
7 54
Chapter 7 _ Configure the Proof Kit 54
8 57
Chapter 8 _ Proof Kit RTL and Various Templates 57
9 58
Chapter 9 _ Advanced Debug Files 58
10 59
SDRAM Design Example 59
Setting up the Environment 59
Verifying SDRAM Using Xcelium 59

1

Chapter 1 _ Overview

This section briefly reviews the Single Data Rate (SDR) synchronous DRAM specification. For a detailed overview of the SDR functionality, see the JEDEC SDR SDRAM STANDARD (JEDEC No.21-C).
This section includes the following:

- Creating Your SDRAM Interface Requirements Model
- Identifying SDRAM Basic Functionality
- Modeling Requirements
- End-To-End Requirements
- Arbitration Requirements

1.1 : Creating Your SDRAM Interface Requirements Model

This section offers guidance on how to create your SDRAM Interface high-level requirements model. In general, a specification provides the following information to help you create your requirements model:

- Interface definitions
- Tables or state diagrams defining conceptual states of the SDRAM and environment
- Waveform and descriptions showing control and protocol handshake requirements

1.1.1 : Interface Definitions

For our SDRAM requirements model, we specify the interface input and output signals that should be monitored as part of your high-level requirements in the Interface section. For our discussion in this section, we present the signal description for a 16-pin device as defined in the following table.

Table 1. SDRAM Specification: Pin Functional Description

SYMBOL	TYPE	DESCRIPTION[1]
A(n)	Input	ADDRESS INPUTS:Those inputs that select (address) a particular cell or set of cells within a memory arry for presentation on the device outputs.
BA(n)	Input	BANK ADDRESS:The BANK ADDRESS is used to select any one of the available banks.
CLK	Input	CLOCK:An input that controls the activation of both input and output circuitry, normally storage registers or latches.
CKE	Input	CLOCK ENABLE:In certain synchronous memory devices, a logic level input that enables the clock input and allows it to fulfill its defined function.
$\overline{CS}$	Input	CHIP ENABLE:Input selects the associated memory die.
$\overline{RAS}$	Input	ROW ADDRESS STROBE:A chip enable signal that, on certain dynamic RAMs, actuates only row address oriented internal circuitry.
$\overline{CAS}$	Input	COLUMN ENABLE:An enable signal that on some dynamic RAMs actuates only the column oriented internal circuits and the input/output circuits.
$\overline{WE}$	Input	WRITE ENABLE:The input that causes the data present on the DQ pin(s) to be written into the address cell(s) of the device.
DQM(n)	Input/Output	INPUT/OUTPUT MASK:A control signal used primarily on SDRAMs that acts as mask for reading and writing functions.
DQ(n)	Input/Output	DATA INPUT/OUTPUT:The pins that serve as data output(s) when in the read mode and as data input(s) when in the write mode.

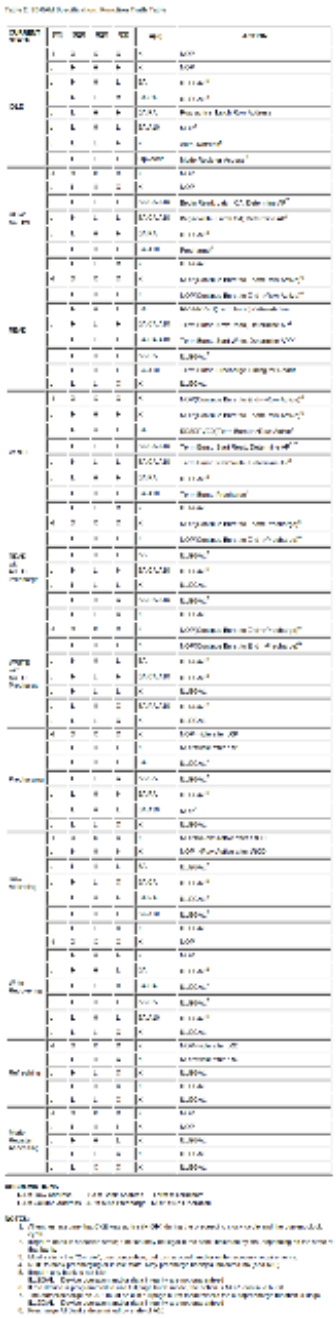
NOTE:
For detailed information about the functionality, see Table 2 below and the functional description in the SDRAM Specification.

1.2 : Identifying SDRAM Basic Functionality

The SDRAM operations (or basic functionality) are listed below:

- Burst Read
- Burst Write
- Multibank ping-pong access
- Burst Read with Autoprecharge
- Burst Write with Autoprecharge
- Burst Read terminated with Precharge
- Burst Write terminated with Precharge
- Burst Read terminated with another Burst Read/Write
- Burst Write terminated with another Burst Write/Read
- Burst Terminate Command(optional)

- DQM# masking
- Precharge All command
- Auto Refresh
- Self Refresh command
- Power Down
- Multibank operation
- Full Page Burst Mode(optional)
- CL=2, 3 and 4(optional)
- Burst Length 1(optional), 2, 4, and 8
  1.2.1 : SDRAM Function Truth Table
  The SDRAM command truth table from the JEDEC SDRAM Specification is reproduced below. Consult this table when creating an SDRAM interface requirements model.



- 1.2.2 : SDRAM Function Truth Table for CKE

  The SDRAM command truth table for CKE from the JEDEC SDRAM Specification is reproduced below. Consult this table when creating an SDRAM interface requirements model.

  Table 3. SDRAM Specification: Function Truth Table for CKE

CURRENT STATE	CKE_{n-1}	CKE_n	$\overline{CS}$	$\overline{RAS}$	$\overline{CAS}$	$\overline{WE}$	A(n)	ACTION
	H	X	X	X	X	X	X	INVALID

Self–Refresh[1]	L	H	H	X	X	X	X	EXIT Self–Refresh=>ABI
	L	H	L	H	H	H	X	EXIT Self–Refresh=>ABI
	L	H	L	H	H	L	X	ILLEGAL
	L	H	L	H	L	X	X	ILLEGAL
	L	H	L	L	X	X	X	ILLEGAL
	L	L	X	X	X	X	X	NOP(Maintain Self–Refresh)
Power–Down	H	X	X	X	X	X	X	INVALID
	L	H	H	X	X	X	X	EXIT Power–Down=>ABI
	L	H	L	H	H	H	X	EXIT Power–Down=>ABI
	L	H	L	H	H	L	X	ILLEGAL
	L	H	L	H	L	X	X	ILLEGAL
	L	H	L	L	X	X	X	ILLEGAL
	L	L	X	X	X	X	X	NOP(Maintain Power–Down)
All Banks Idle[2]	H	H	X	X	X	X	X	Refer to Table 2
	H	L	H	X	X	X	X	Enter Power–Down
	H	L	L	H	H	H	X	Enter Power–Down
	H	L	L	H	H	L	X	ILLEGAL
	H	L	L	H	L	X	X	ILLEGAL
	H	L	L	L	H	X	X	ILLEGAL
	H	L	L	L	L	H	X	Enter Self–Refresh
	H	L	L	L	L	L	X	ILLEGAL
	L	L	X	X	X	X	X	NOP
Any State other than listed above	H	H	X	X	X	X	X	Refer to Table 2
	H	L	X	X	X	X	X	Begin Clock Suspend next cycle[3]
	L	H	X	X	X	X	X	Exit Clock Suspend next cycle[3]
	L	L	X	X	X	X	X	Maintain Clock Suspend

ABBREVIATIONS

ABI = All Bank Idle

NOTES:

1. CKE Low–to–High transition will re–enable CK and other inputs asynchronously. A minimum setup time must be satisfied before any command other than EXIT.
2. Power–Down and Self–Refresh can be entered only from the All Banks Idle State.
3. Must be legal command.

1.2.3 : SDRAM Command List

Table 4 lists the SDR operations (or basic functionality) as defined in the SDR specification.

Table 4. SDR Command List

Command	Symbol	Operation
Deselect	DES	Device deselect Current operation continues.
No Operation	NOP	No operation Current operation continues.
Active	ACT	Open (or activate) a row in a particular bank for a subsequent access.
Read	RD	Start burst read.
Read with Auto Precharge	RDA	Start burst read. After burst read is finished, precharge starts automatically
Write	WR	Start burst write.
Write with Auto Precharge	WRA	Start burst write. After burst write is finished, precharge starts automatically
Burst Terminate	BST	Terminate burst operation.
Precharge	PRE	Precharge selected bank.
Precharge All Banks	PREA	Precharge all banks.
Auto Refresh	REF	Start auto refresh.
Mode Register Set	MRS	Set mode register.
Enter Self Refresh	SRE	Enter self refresh.
Exit Self Refresh	SRX	Exit self refresh.
Enter Power Down	PDE	Enter power-down mode.
Exit Power Down	PDX	Exit power-down mode.
Enter Deep Power Down	DPDE	Enter deep power-down mode.
Exit Deep Power Down	DPDX	Exit deep power-down mode.

1.2.4 : SDRAM Command Truth Table

The SDR command truth table from the SDR specification, which defines SDR interface behavior, is reproduced in Table 5 below. Consult this table when creating a SDR interface requirements model.

In the following table, note the following:

- H: High level
- L: Low level
- X: High or low level (Don't care)

Table 5. SDR Command Truth Table

Command	Symbol	CKE		$\overline{CS}$	$\overline{RAS}$	$\overline{CAS}$	$\overline{WE}$	A(n)
		n-1	n					
Deselect	DES	H	H	H	X	X	X	X
No Operation	NOP	H	H	L	H	H	H	X
Active	ACT	H	H	L	L	H	H	BA, RA
Read	RD	H	H	L	H	L	H	BA, CA, A10(L)
Read with auto precharge	RDA							BA, CA, A10(H)
Write	WR	H	H	L	H	L	L	BA, CA, A10(L)
Write with auto precharge	WRA							BA, CA, A10(H)
Burst Terminate	BST	H	H	L	H	H	L	X
Precharge	PRE	H	H	L	L	H	L	BA, A10(L)
Precharge All Banks	PREA							A10(H)
Auto Refresh	REF	H	H	L	L	L	H	X
Mode Register Set	MRS	H	H	L	L	L	L	Op-Code
Enter Self Refresh	SRE	H	L	L	L	L	H	X
Exit Self Refresh	SRX	L	H	H	X	X	X	X
				L	H	H	H	
Enter Power Down	PDE	H	L	H	X	X	X	X
				L	H	H	H	
Exit Power Down	PDX	L	H	H	X	X	X	X
				L	H	H	H	
Enter Deep Power Down	DPDE	H	L	H	H	L	H	X
Exit Deep Power Down	DPDX	L	H	X	X	X	X	X

ABBREVIATIONS

BA = Bank Address RA = Row Address CA = Column Address

1.3 : Modeling Requirements

There are two types of requirements for this type of interface:

1. Illegal Commands €" for example, when the main FSM is in IDLE state, the memory controller cannot issue a READ command.
2. Illegal Bus-Interface Protocol €" for example, if Trcd (RAS to CAS delay) is set to 3, the CAS cycle should not come before three latency cycles.

To model requirements that check for illegal commands, simply go through each state in the state and identify all illegal transitions. To model requirements that check for illegal bus-interface protocol, simply go through each timing constant and verify each of them.
One technique useful when constructing an SDRAM requirements model is to create additional registers in the requirements module that will keep track of past transactions. For example, when the main FSM is in ACT, it is okay to activate a new row of a different bank as long as the bank-to-bank delay is met. However, to activate a new row within the same bank, the memory controller has to first perform a precharge. Hence we need a bus to keep track of all the banks that have been activated.
To construct your requirements model, review the following sections: Interface and Testplan. An example of the SDRAM interface requirements model is shown in the Requirements section.

1.4 : End-To-End Requirements

Although this section focuses primarily on SDRAM interface requirements, verifying end-to-end requirements for your memory controller is important too. For example, for many SDRAM designs that support burst writes to the memory, data is usually queued inside the memory controller as part of its datapath. For a detailed discussion on creating requirements for datapaths, see the Datapath section of the Formal Testplanner „¢ knowledge base, which demonstrates how to create a high-level requirements model using the Jasper „¢ Datapath Module to capture the data coming from the memory agents to the SDRAM and from SDRAM to memory agents.
For end-to-end requirements, it is important to allow all agents to arbitrarily send and receive data through the memory controller, but to simplify the proof we only capture the sent data from one agent using the Jasper Datapath Module. If the memory agents are symmetrical (all agents go through the same logic within the memory controller), then only one path is needed to verify that all paths are working. Otherwise, we recommend that you verify each path separately instead of attempting to model and verify all paths at once.

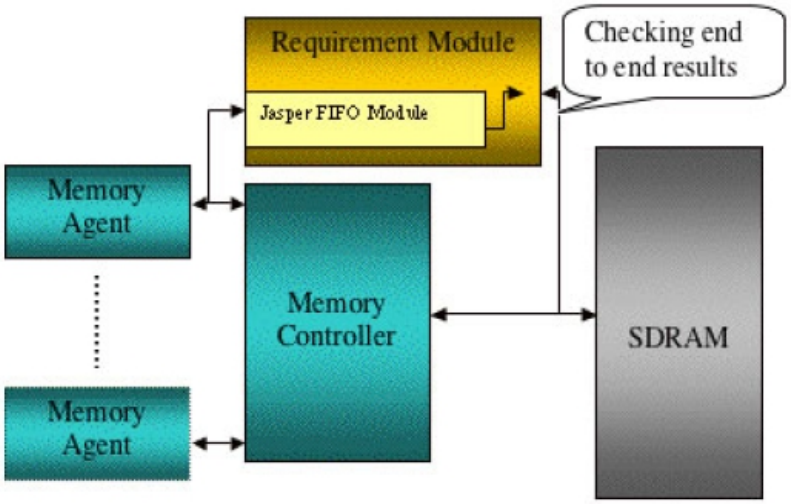


Figure 1. SDRAM End-to-End Requirements Model

1.5 : Arbitration Requirements

There are typically two types of arbitrations: fair arbitration and prioritized arbitration. Fair arbitration simply means that all agents should have an equal chance to access the memory. The requirements for an 8-agent memory controller are as follows:

- Reset a counter to eight after agent x sends a request to the memory controller. The counter counts down by one every time a grant that does not belong to agent x is issued. The counter will reset to eight and be disabled after a grant for agent x is issued.
- If the counter counts down to 0, there is a fairness violation because there were more than eight grants in an 8-agent system issued to other agents.
  For prioritized arbitration, perform the same checks among high-priority ports and among low-priority ports. The arbitration between a high-priority port and a low-priority port is usually well-defined; thus, it is usually possible to provide a requirement for it.

2

Chapter 2 _ Interface

This section includes the following topics:

- SDRAM Interface Description
- INOUT (Tri-State) Considerations

2.1 : SDRAM Interface Description

This section defines the list of signal connections (that is, interface) we have chosen to monitor as part of our high-level requirements model (JASPER_SDRAM).

In many cases, the proofkit will have signals that are not used by the DUV (they are not input or output of the DUV). For example, if the DUV is the slave, arbiter-output signals do not go to the slave. Or if the DUV does not support a certain mode (like split), it does not have the mode-related signals.

For most cases, users need to leave these signals unconnected or free. To do this, comment out the lines containing these signals in the bind or instantiation statements in the wrapper file.

If such signals are tied to constants, they can cause over-constraint problems in the proofkit.

In the rare cases that having certain signals unconnected or free causes a false CEX, users can tie them to 0 or 1 as needed.

NOTE: Signals ending with N are active low.

clk

Primary clock input

cke

Clock enable

csN

Chip select

rasN

Row address strobe

casN

Column address

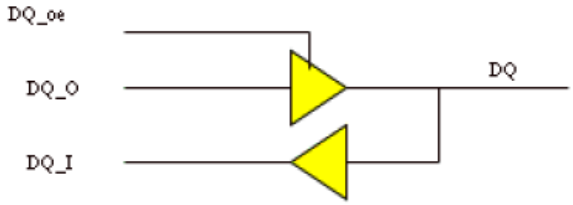
weN

Write enable

BA

Bank address
A
Address bus
DQ
Data bus
DQM
Data mask
pk_resetn
reset for ProofKit
2.2 : INOUT (Tri-State) Considerations
The following signals are not needed for memory interface verification but will be needed for end-to-end verification:

- DQ_IDQ_O
- DQ_oe
  DQ (data) is actually separated into three signals for each bit of data (that is, the signals before the bidirectional buffers).
  By separating the inout into three signals, we can write more precise requirements for read and write transactions. Furthermore, we can include requirements that verify that the memory controller is driving the data bus when it is supposed to and not driving the bus when it is not supposed to.



3

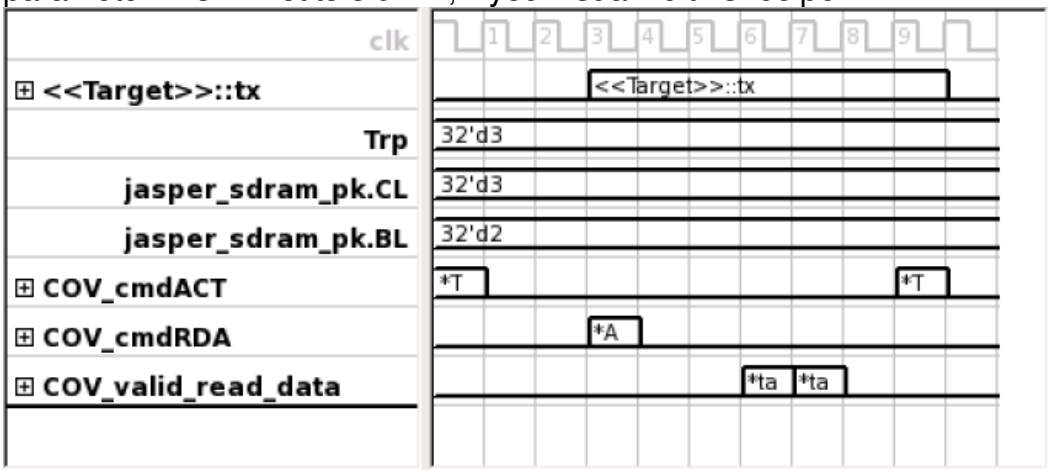
Chapter 3 _ Spec Functionality

3.1 : Auto Precharge

The user may specify that the bank currently being accessed precharge itself as soon as the burst is completed. This is done using address bit AP during the column address cycle.
The user must wait until the precharge is completed before issuing another command to the device. Timing for auto precharge is required to be the same as or less than the minimum requirement of external precharge.

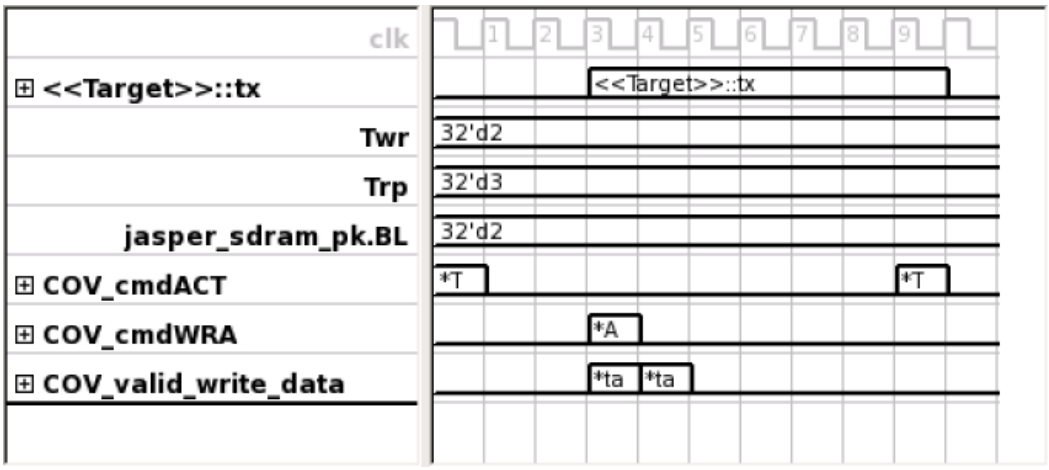
Read with Auto Precharge

This is a recipe visualizing the behavior of burst read with auto precharge. You need to set the parameter mrsBL not to 3'b111, if you visualize this recipe.



Write with Auto Precharge

This is a recipe visualizing the behavior of burst write with auto precharge. You need to set the parameter mrsBL not to 3'b111, if you visualize this recipe.

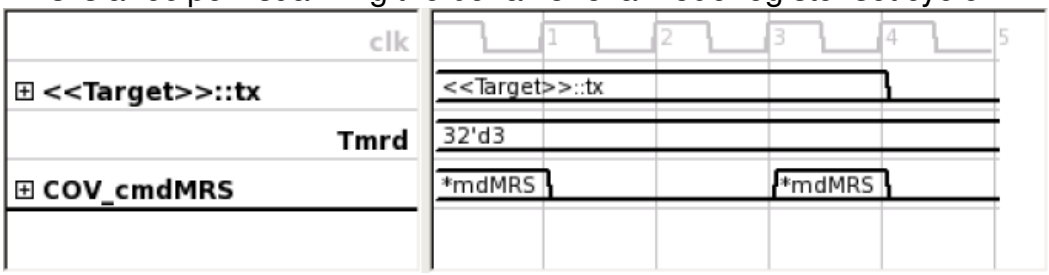


3.2 : Mode Register Write Timing

A mode register set cycle can be followed by a new command in no less than 3 clock cycles. A mode register set cycle is illegal if any bank is not idle.

Mode Register Set

This is a recipe visualizing the behavior of a mode register set cycle.

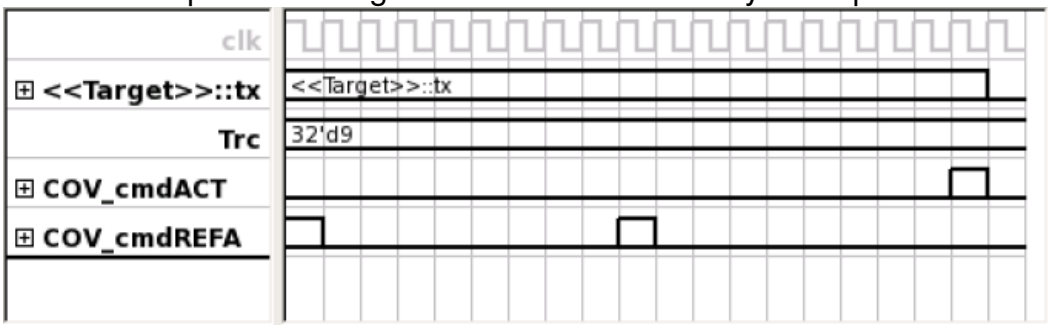


3.3 : Auto Refresh

Auto refresh is an operation that initiates a single refresh cycle for an SDRAM, but that once initiated, is completed by internal control in the device with the refresh address being supplied by an internal register in the device. Before performing an Auto refresh, all banks of the device must be precharged. All banks will automatically precharge at the end of the refresh cycle. Additional commands must not be supplied to the device during the minimum refresh time specified.

Refresh Cycle

This is a recipe visualizing the behavior of refresh cycle requirements.



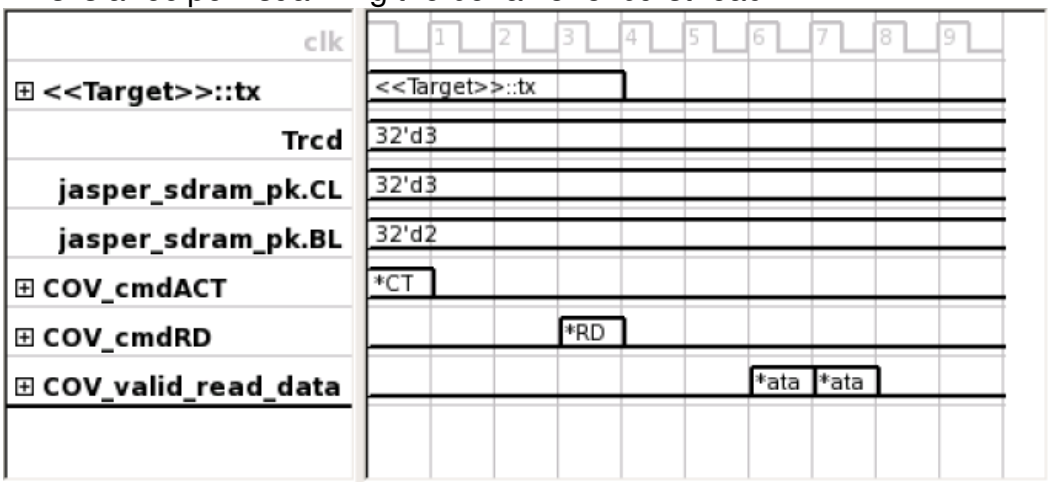
3.4 : Read/Write Latency

Read/Write Latency for SDR Synchronous DRAM shall be as defined as the clock cycle difference between the clock where read/write command and collumn address are asserted and the clock

where first data to be read/written is asserted.
The mode register is used to define a CAS latency for read operation. Write latency is 0 clock.

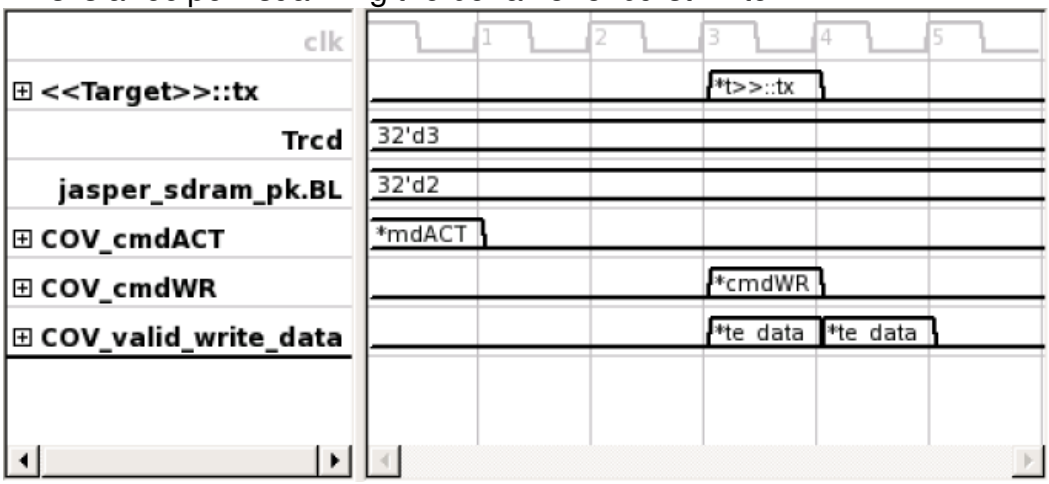
Single Read

This is a recipe visualizing the behavior of burst read.



Single Write

This is a recipe visualizing the behavior of burst write.

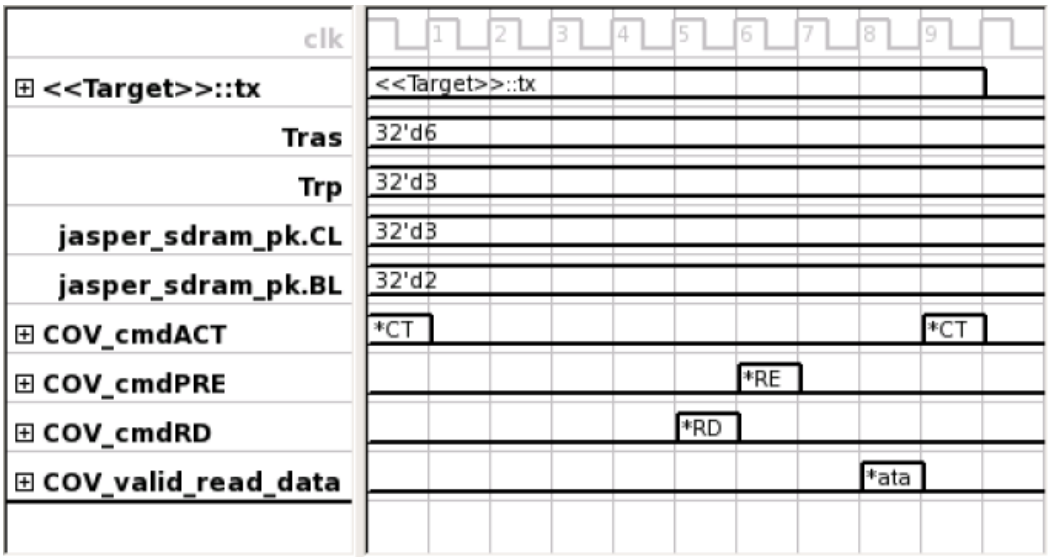


3.5 : Precharge Timing for Reads

The assertion of the precharge command has a direct relationship to the timing of data out for a read cycle. For a CAS latency of 1, the minimum requirements is that the precharge command will be allowed to coincide with output of the last data from a burst regardless of burst length. For a CAS latency greater than 1, theminimum requirement is that the precharge command will be allowed to coincide with output of the next-to-last data from a burst, regardless of burst length, without interrupting burst data.

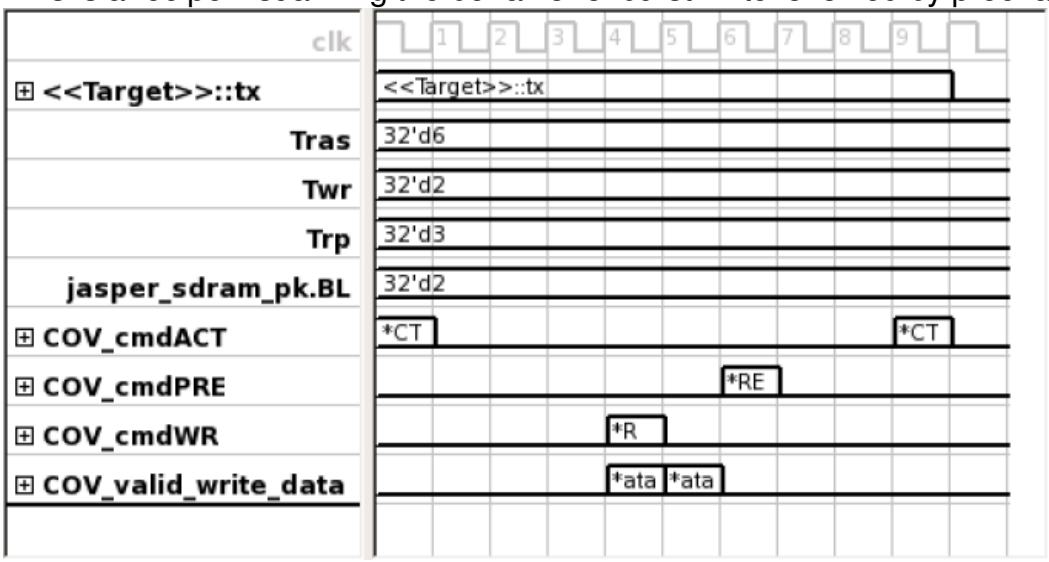
Read to Precharge

This is a recipe visualizing the behavior of burst read followed by precharge command.



Write to Precharge

This is a recipe visualizing the behavior of burst write followed by precharge command.



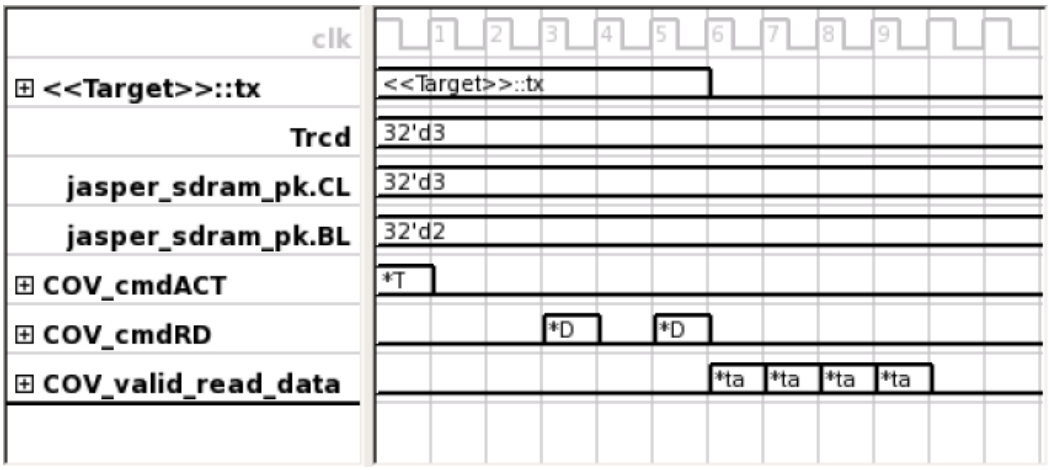
3.6 : Column Address to Column Address Delay

The minimum column-address-to-column-address delay time is two clock cycles, independent of operating frequency.

For interrupted bursts, column addresses must, at aminimum, follow the €2n rule € while a read or write burst is in progress. 2n rule: after the initial read or write command, a new column address can be presented to the device every other clock cycle. That is, if the initial read ~~or~~ write command occured on an odd clock cycle, the new column addresses must presented on an odd clock cycle while the burst is in progress.

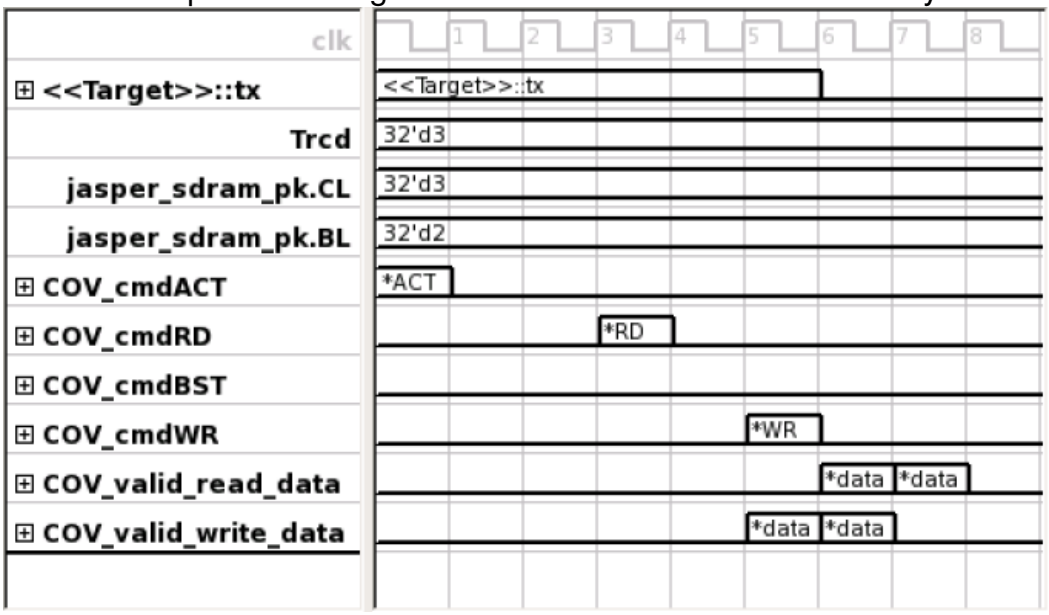
Read to Read

This is a recipe visualizing the behavior of burst read followed by burst read.



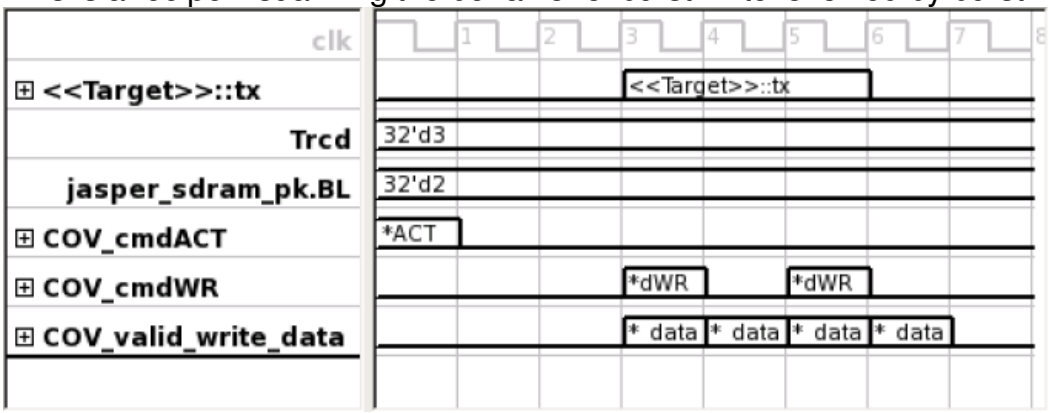
Read to Write

This is a recipe visualizing the behavior of burst read followed by burst write.



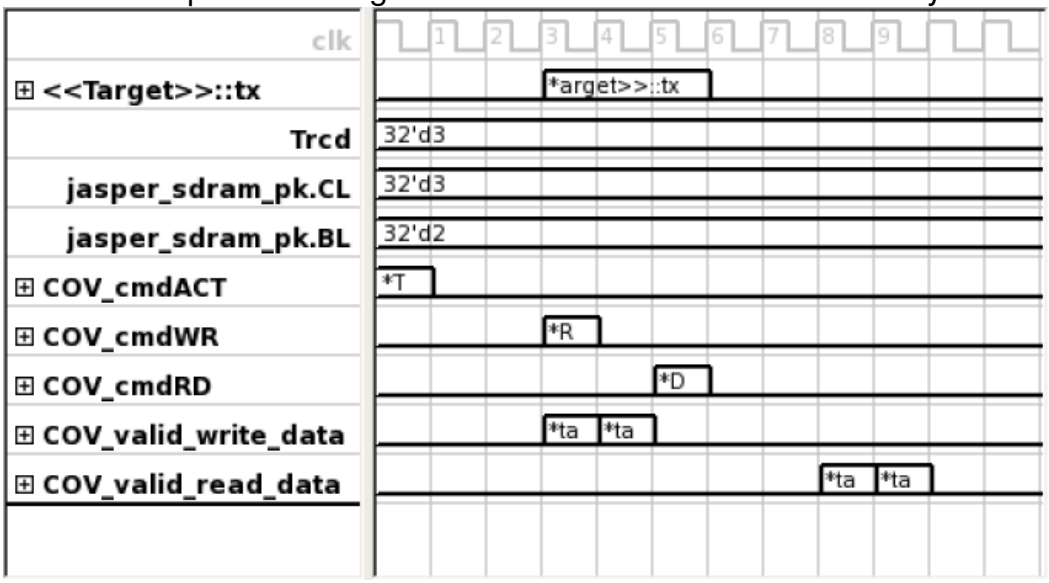
Write to Write

This is a recipe visualizing the behavior of burst write followed by burst write.



Write to Read

This is a recipe visualizing the behavior of burst write followed by burst read.

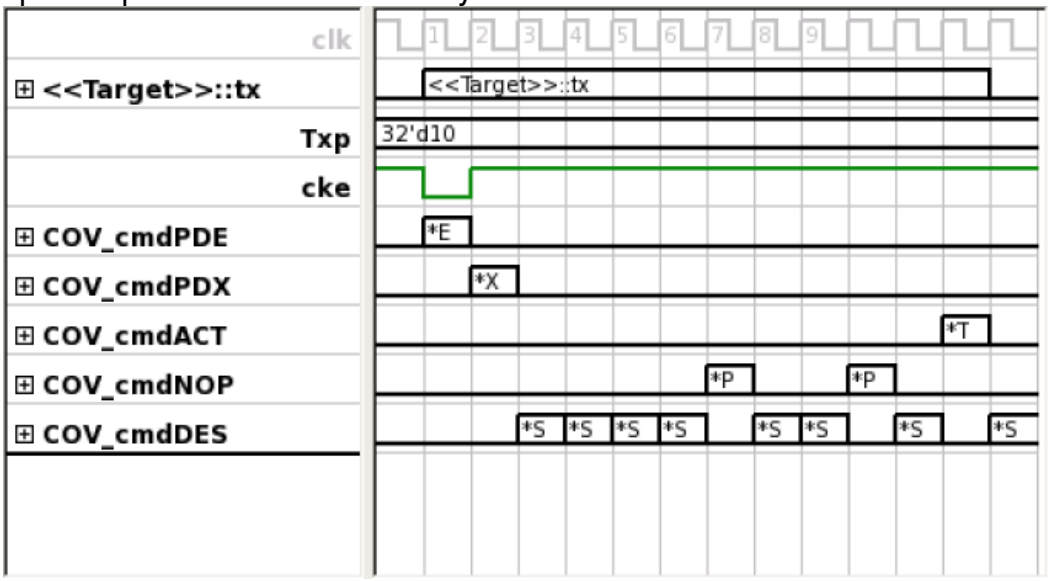


3.7 : CKE Timing for Power Down

Using CKE to enter the low power state can only be performed while all internal banks of the device are precharged. During power down the device is not refreshed. Therefore the minimum refresh specification still applies during power down.

Power-down

This is a recipe visualizing the behavior of power-down entry and exit. Note that Txp is the supplier specific power-down exit latency.

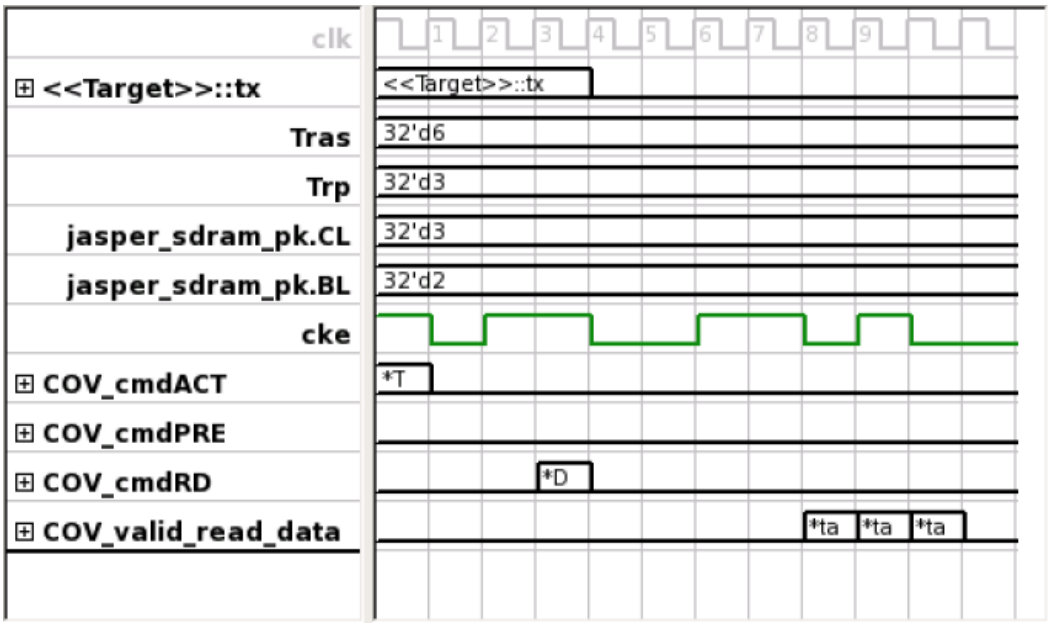


3.8 : CKE Timing for Clock Suspend

The clock may be suspended for one or more clock cycles.

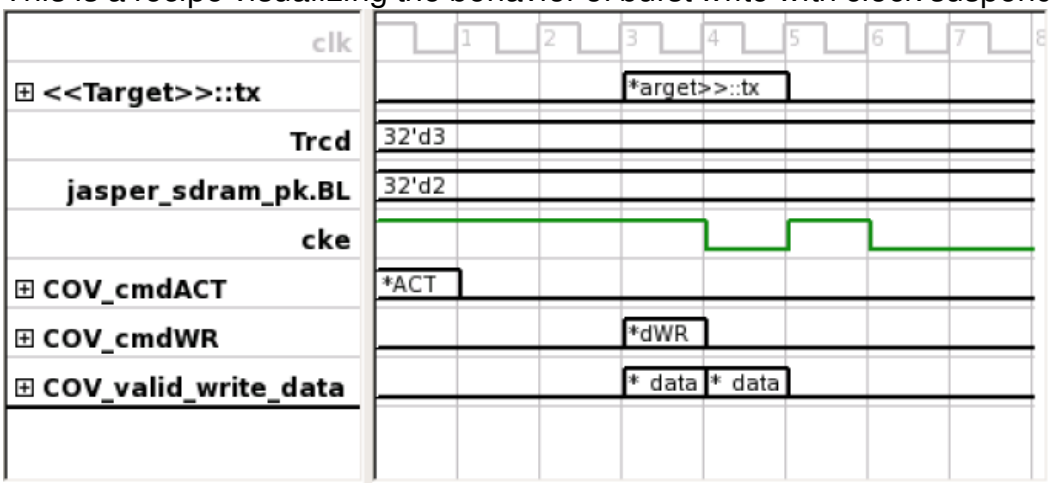
Read with Clock Suspend

This is a recipe visualizing the behavior of burst read with clock suspend.



Write with Clock Suspend

This is a recipe visualizing the behavior of burst write with clock suspend.

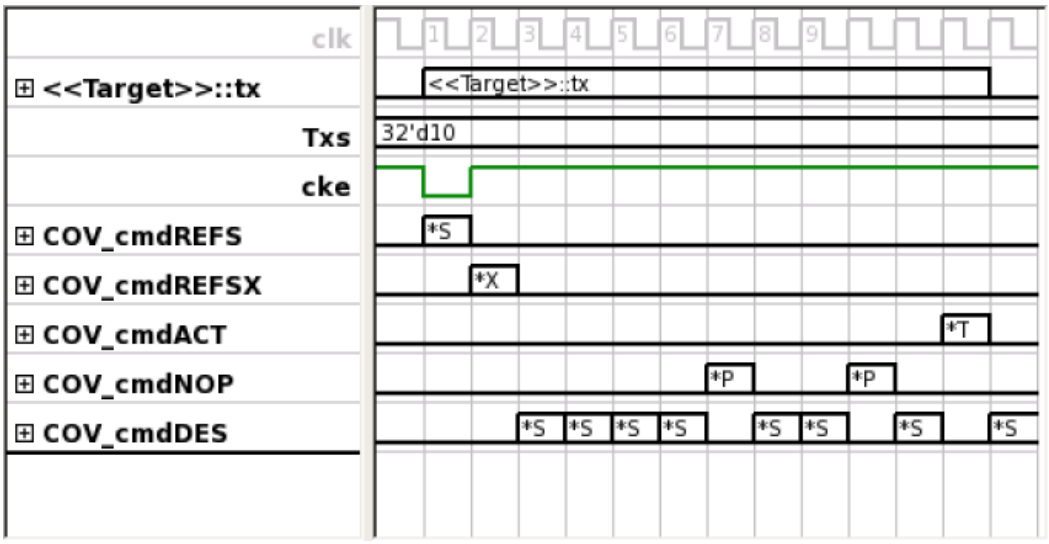


3.9 : Self Refresh Entry and Exit

In self refresh mode, the device refreshes itself without outside intervention. Self-refresh mode is entered by precharging all banks and then inserting an auto-refresh command with CKE low. Exit from self-refresh mode is accomplished by starting the clock and then asserting CKE. NOP commands must be asserted for a supplier-specified minimum period, which must include 3 clocks, to allow the device to return to the IDLE state.

Self Refresh

This is a recipe visualizing the behavior of self refresh entry and exit. Note that Txs is the supplier specific self refresh exit latency.



3.10 : Burst Terminate Command for SDRAM

The Burst Terminate command (BST) is an optional feature for SDRAM. If the Burst Terminate command is included in an SDRAM, the following functionality is required:

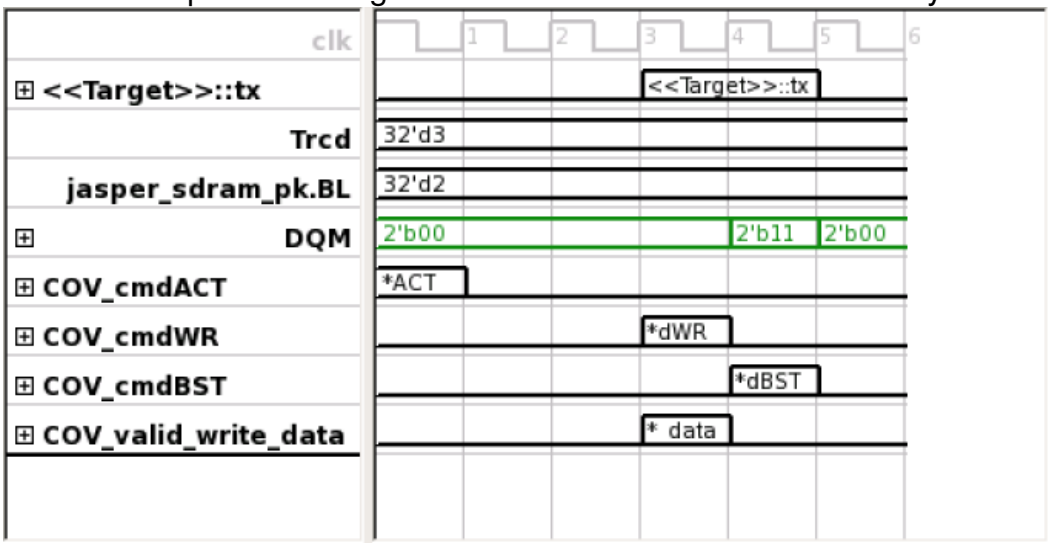
1. BST applies to all burst lengths.
2. BST is not a valid command during read or write with autoprecharge
3. When terminating a burst read command, the BST command must be issued n clock cycles before
the clock edge at which the last desired data word is valid, where n equals the CAS latency for read operations minus 1.
4. When terminating a burst write command, the BST command must be issued one clock after the clock edge which samples the last word of data that is required to be written into the memory. The DQM input(s) must be high for the clock edge that is coincident with the BST command.

Read Burst Terminate

This is a recipe visualizing the behavior of burst read followed by burst terminate command.

Write Burst Terminate

This is a recipe visualizing the behavior of burst write followed by burst terminate command.

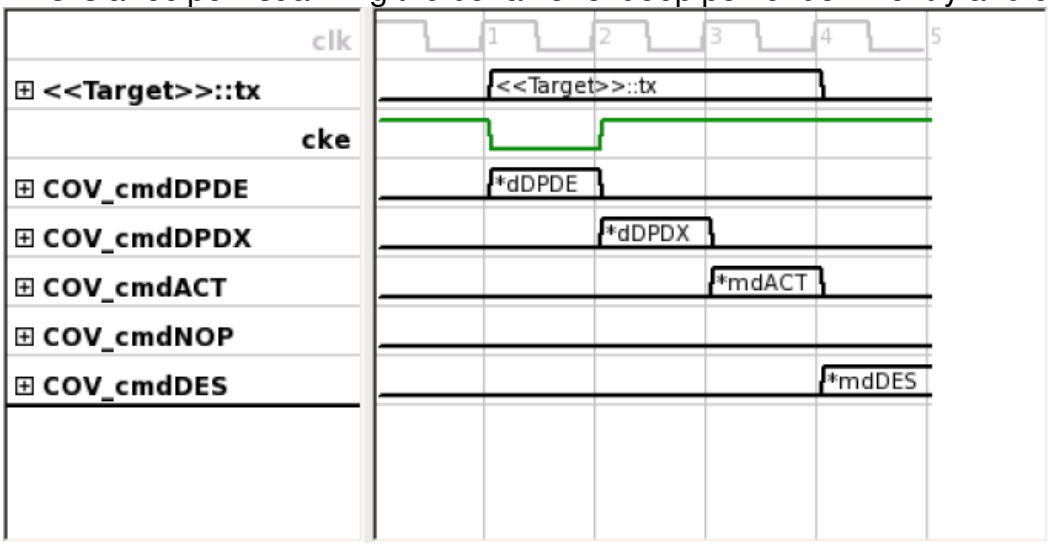


3.11 : Deep Power Down Mode

Deep Power Down Mode is an operating mode that is used to achieve maximum power reduction by cutting the power of the whole memory array of the devices. Data will not be retained once the device has entered into Deep Power Down Mode. Full initialization is required when the device exits from Deep Power Down Mode.

Deep Power Down

This is a recipe visualizing the behavior of deep power down entry and exit.



4

Chapter 4 _ Test Plan

4.1 : Executed state of SDR Commands
4.1.1 : ACT
Activate.
According to Table 3.11.5.1.1 on p.3 in the specs, this command should be issued only from the IDLE state.
4.1.2 : RD
Burst Read.
According to Table 3.11.5.1.1 on p.3 in the specs, this command should be issued only when the corresponding bank is activated.
4.1.3 : WR
Burst Write.
According to Table 3.11.5.1.1 on p.3 in the specs, this command should be issued only when the corresponding bank is activated.
4.1.4 : BST
Burst Terminate.
According to Table 3.11.5.1.1 on p.3 in the specs, this command should be issued only from the READ and WRITE state.
4.1.5 : PRE
Precharge per bank.
According to Table 3.11.5.1.1 on p.3 in the specs, this command should be issued only from the IDLE state or when the corresponding bank is activated.
4.1.6 : PREA
Precharge to all bank.
According to Table 3.11.5.1.1 on p.3 in the specs, this command should be issued only from the IDLE state or when the corresponding bank is activated.
4.1.7 : REF
Refresh.
According to Table 3.11.5.1.1 on p.3 in the specs, this command should be issued only from the IDLE.
4.1.8 : MRS
Mode Register Set.
According to Table 3.11.5.1.1 on p.3 in the specs, this command should be issued only from the IDLE.
4.1.9 : SRE
Self Refresh Entry.

According to Table 3.11.5.1.2 on p.5 in the specs, this command should be issued only from the IDLE.

4.1.10 : SRX

Self Refresh Exit.

According to Table 3.11.5.1.2 on p.5 in the specs, this command should be issued only from the SELF Refreshing state.

4.1.11 : PDE

Power Down Entry.

According to Table 3.11.5.1.2 on p.5 in the specs, this command should be issued only from the IDLE.

4.1.12 : PDX

Power Down Exit.

According to Table 3.11.5.1.2 on p.5 in the specs, this command should be issued only from the Power Down state.

4.1.13 : DPDE

Deep Power Down Entry.

According to Table 3.11.5.1.2 on p.5 in the specs, this command should be issued only from the IDLE.

4.1.14 : DPDX

Deep Power Down Exit.

According to Table 3.11.5.1.2 on p.5 in the specs, this command should be issued only from the Deep Power Down state.

4.1.15 : RDA

Read with Auto Precharge.

According to Table 3.11.5.1.1 on p.3 in the specs, this command should be issued only when the corresponding bank is activated.

4.1.16 : WRA

Write with Auto Precharge.

According to Table 3.11.5.1.1 on p.3 in the specs, this command should be issued only when the corresponding bank is activated.

4.2 : AC timings for SDR commands

4.2.1 : ACT

4.2.1.1 : tRC

According to Table 3.11.6.3 on P.4 in "3_11_06R9.pdf", the minimum time interval between two successive ACTIVE commands on the same bank is defined by tRC.

4.2.1.2 : tRRD

According to Table 3.11.6.3 on P.4 in "3_11_06R9.pdf", the minimum time interval between two successive ACTIVE commands on different banks is defined by tRRD.

4.2.1.3 : tRCD

According to Section 3.11.5.1.12 on P.13 in the specs, once a row is open, a READ or WRITE command could be issued to that row after tRCD cycles.

The corresponding commands are RD, RDA, WR, WRA.

4.2.1.4 : tRAS
According to Table 3.11.6.3 on P.4 in "3_11_06R9.pdf", tRAS is the minimum time between Activate command and Precharge command to the same bank.
The corresponding commands are PRE, PREA.
4.2.2 : RDA
4.2.2.1 : After auto precharge command
According to Section 3.11.5.1.5 on P.7 in the specs, another command to the same bank cannot be issued until the precharging time (tRP) is completed ater the Read with Auto Precharge command.
The corresponding commands are ACT, REF, MRS, SRE, PDE, DPDE.
4.2.3 : WR
4.2.3.1 : tWR
According to Sec 3.11.5.1.9 on P.10 in the specs, in order for a Precharge to follow a WRITE without truncating the WRITE burst, the Precharge must not be issued until after tWR cycles.
The corresponding commands are PRE, PREA.
4.2.4 : WRA
4.2.4.1 : After auto precharge command
According to Sec 3.11.5.1.5 on P.7 in the specs, another command to the same bank cannot be issued until the precharging time (tRP) is completed ater the Write with Auto Precharge command.
The corresponding commands are ACT, REF, MRS, SRE, PDE, DPDE.
4.2.5 : PRE
4.2.5.1 : tRP
According to Sec 3.11.5.1.1 on P.3 in the specs, after the Precharge command is issued, the bank(s) will be available for a subsequent row access for tRP cycles.
The corresponding commands are ACT, REF, MRS, SRE, PDE, DPDE.
4.2.6 : PREA
4.2.6.1 : tRP
According to Sec 3.11.5.1.1 on P.3 in the specs, after the Precharge all command is issued, the bank(s) will be available for a subsequent row access for tRP cycles.
The corresponding commands are ACT, REF, MRS, SRE, PDE, DPDE.
4.2.7 : REF
4.2.7.1 : tRC
According to Sec 3.11.5.1.8 on P.9 in the specs, another command to the same bank cannot be issued until tRC is completed ater the Auto Refresh command.
The corresponding commands are ACT, PRE, PREA, REF, MRS, SRE, PDE, DPDE.
4.2.8 : MRS
4.2.8.1 : tMRD
According to Sec 3.11.5.1.7 on P.8 in the specs, susbsequent executable commands cannot be issued until 3 clock cycles (which is defined as tMRD in the proof kit) after the MODE REGISTER SET command is issued.
The corresponding commands are ACT, PRE, PREA, REF, MRS, SRE, PDE, DPDE.
4.2.9 : SRX
4.2.9.1 : tXS

According to Sec 3.11.5.1.15 on P.16 in the specs, a supplier specific delay (which is defined as tXS in the proof kit) must be satisfied before next valid command can be issued to the device to allow for completion of any internal refresh in progress.
The corresponding commands are ACT, PRE, PREA, REF, MRS, SRE, PDE, DPDE.

4.2.10 : PDX

4.2.10.1 : tXP

According to Sec 3.11.5.1.13 on P.14 in the specs, a supplier specific delay (which is defined as tXP in the proof kit) must be satisfied before next valid command can be issued to the device.
The corresponding commands are ACT, PRE, PREA, REF, MRS, SRE, PDE, DPDE.

4.3 : Miscellaneous

4.3.1 : Illegal command

According to Sec 3.11.5.1.1 on P.3 in the specs, all states and sequences not shown are illegal or reserved.

4.3.2 : Illegal command in Full Page Burst Mode

According to Sec 3.11.5.1.17 on P.18 in the specs, autoprecharge command is not a legal command in Full Page Burst Mode.

4.3.3 : DQM for write burst terminate

According to Sec 3.11.5.1.16 on P.17 in the specs, when terminating a burst write command with the BST, DQM should be high.

4.3.4 : Clock suspend

According to Sec 3.11.5.1.14 on P.15 in the specs, Read data should be stable after CKE is low.

5

Chapter 5 _ Requirements

5.1 : Requirement Naming Convention
All requirements are named according to the following format:
assert_NAME_X
Where:

- assert : represents an assertion
- NAME : is the property name.

5.2 : Requirements Set 1: SDR Command Requirements
assert_not_cmdACT
Cross-referenced at: ACT
When the 'ACT' command is issued, current state should be IDLE.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Table 3.11.5.1.1
assert_not_cmdRD
Cross-referenced at: RD
When the 'RD' command is issued, current state should be Bank active, READING, WRITING.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Table 3.11.5.1.1
assert_not_cmdWR
Cross-referenced at: WR
When the 'WR' command is issued, current state should be Bank active, WRITING, READING.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Table 3.11.5.1.1
assert_not_cmdBST
Cross-referenced at: BST
When the 'BST' command is issued, current state should be WRITING, READING.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Table 3.11.5.1.1
assert_not_cmdPRE
Cross-referenced at: PRE
When the 'PRE' command is issued, current state should be IDLE, Bank active, READING, WRITING, Precharging.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Table 3.11.5.1.1
assert_not_cmdPREA
Cross-referenced at: PREA
When the 'PREA' command is issued, current state should be IDLE, Bank active, READING, WRITING, Precharging.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Table 3.11.5.1.1

assert_not_cmdREF
Cross-referenced at: REF
When the 'REF' command is issued, current state should be IDLE.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Table 3.11.5.1.1
assert_not_cmdMRS
Cross-referenced at: MRS
When the 'MRS' command is issued, current state should be IDLE.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Table 3.11.5.1.1
assert_not_cmdSRE
Cross-referenced at: SRE
When the 'SRE' command is issued, current state should be IDLE.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.5 Table 3.11.5.1.2
assert_not_cmdSRX
Cross-referenced at: SRX
When the 'SRX' command is issued, current state should be Self Refreshing.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.5 Table 3.11.5.1.2
assert_not_cmdPDE
Cross-referenced at: PDE
When the 'PDE' command is issued, current state should be IDLE.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.5 Table 3.11.5.1.2
assert_not_cmdPDX
Cross-referenced at: PDX
When the 'PDX' command is issued, current state should be Power Down.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.5 Table 3.11.5.1.2
assert_not_cmdDPDE
Cross-referenced at: DPDE
When the 'DPDE' command is issued, current state should be IDLE.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.5 Table 3.11.5.1.2
assert_not_cmdDPDX
Cross-referenced at: DPDX
When the 'DPDX' command is issued, current state should be Deep Power Down.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.5 Table 3.11.5.1.2
assert_not_cmdRDA
Cross-referenced at: RDA
When the 'RDA' command is issued, current state should be Bank active, READING, WRITING.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Table 3.11.5.1.1
assert_not_cmdWRA
Cross-referenced at: WRA
When the 'WRA' command is issued, current state should be Bank active, WRITING, READING.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Table 3.11.5.1.1

5.3 : Requirements Set 2: SDR Interface Requirements
assert_cmdACT2cmdACT_same

Cross-referenced at: tRC
tRC is the minimum cycles between the 'ACT' command and the subsequent 'ACT' command to the same bank.
JEDEC Standard No.21-C(3_11_06R9.pdf) P.4 Table 3.11.6.3
assert_cmdACT2cmdACT_diff
Cross-referenced at: tRRD
tRRD is the minimum cycles between the 'ACT' command and the subsequent 'ACT' command to a different bank.
JEDEC Standard No.21-C(3_11_06R9.pdf) P.4 Table 3.11.6.3
assert_cmdACT2cmdRD
Cross-referenced at: tRCD
The minimum cycles between the 'ACT' command and the subsequent 'RD' command is tRCD.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.13 Sec 3.11.5.1.12
assert_cmdACT2cmdWR
Cross-referenced at: tRCD
The minimum cycles between the 'ACT' command and the subsequent 'WR' command is tRCD.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.13 Sec 3.11.5.1.12
assert_cmdACT2cmdPRE
Cross-referenced at: tRAS
The minimum cycles between the 'ACT' command and the subsequent 'PRE' command is tRAS.
JEDEC Standard No.21-C(3_11_06R9.pdf) P.4 Table 3.11.6.3
assert_cmdACT2cmdPREA
Cross-referenced at: tRAS
The minimum cycles between the 'ACT' command and the subsequent 'PREA' command is tRAS.
JEDEC Standard No.21-C(3_11_06R9.pdf) P.4 Table 3.11.6.3
assert_cmdACT2cmdRDA
Cross-referenced at: tRCD
The minimum cycles between the 'ACT' command and the subsequent 'RDA' command is tRCD.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.13 Sec 3.11.5.1.12
assert_cmdACT2cmdWRA
Cross-referenced at: tRCD
The minimum cycles between the 'ACT' command and the subsequent 'WRA' command is tRCD.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.13 Sec 3.11.5.1.12
assert_cmdRDA2cmdACT
Cross-referenced at: After auto precharge command
The minimum cycles between the 'RDA' command and the subsequent 'ACT' command is CL+BL-2+tRP.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.7 Sec 3.11.5.1.5
assert_cmdRDA2cmdREF
Cross-referenced at: After auto precharge command
The minimum cycles between the 'RDA' command and the subsequent 'REF' command is CL+BL-2+tRP.

JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.7 Sec 3.11.5.1.5
assert_cmdRDA2cmdMRS
Cross-referenced at: After auto precharge command
The minimum cycles between the 'RDA' command and the subsequent 'MRS' command is CL+BL-2+tRP.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.7 Sec 3.11.5.1.5
assert_cmdRDA2cmdSRE
Cross-referenced at: After auto precharge command
The minimum cycles between the 'RDA' command and the subsequent 'SRE' command is CL+BL-2+tRP.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.7 Sec 3.11.5.1.5
assert_cmdRDA2cmdPDE
Cross-referenced at: After auto precharge command
The minimum cycles between the 'RDA' command and the subsequent 'PDE' command is CL+BL-2+tRP.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.7 Sec 3.11.5.1.5
assert_cmdRDA2cmdDPDE
Cross-referenced at: After auto precharge command
The minimum cycles between the 'RDA' command and the subsequent 'DPDE' command is CL+BL-2+tRP.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.7 Sec 3.11.5.1.5
assert_cmdWR2cmdPRE
Cross-referenced at: tWR
The minimum cycles between the 'WR' command and the subsequent 'PRE' command is tWR.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.10 Sec 3.11.5.1.9
assert_cmdWR2cmdPREA
Cross-referenced at: tWR
The minimum cycles between the 'WR' command and the subsequent 'PREA' command is tWR.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.10 Sec 3.11.5.1.9
assert_cmdWRA2cmdACT
Cross-referenced at: After auto precharge command
The minimum cycles between the 'WRA' command and the subsequent 'ACT' command is BL-1+tWR+tRP.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.7 Sec 3.11.5.1.5
assert_cmdWRA2cmdREF
Cross-referenced at: After auto precharge command
The minimum cycles between the 'WRA' command and the subsequent 'REF' command is BL-1+tWR+tRP.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.7 Sec 3.11.5.1.5
assert_cmdWRA2cmdMRS
Cross-referenced at: After auto precharge command
The minimum cycles between the 'WRA' command and the subsequent 'MRS' command is BL-

1+tWR+tRP.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.7 Sec 3.11.5.1.5
assert_cmdWRA2cmdSRE
Cross-referenced at: After auto precharge command
The minimum cycles between the 'WRA' command and the subsequent 'SRE' command is BL-1+tWR+tRP.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.7 Sec 3.11.5.1.5
assert_cmdWRA2cmdPDE
Cross-referenced at: After auto precharge command
The minimum cycles between the 'WRA' command and the subsequent 'PDE' command is BL-1+tWR+tRP.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.7 Sec 3.11.5.1.5
assert_cmdWRA2cmdDPDE
Cross-referenced at: After auto precharge command
The minimum cycles between the 'WRA' command and the subsequent 'DPDE' command is BL-1+tWR+tRP.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.7 Sec 3.11.5.1.5
assert_cmdPRE2cmdACT
Cross-referenced at: tRP
tRP is the minimum cycles between the 'PRE' command and the subsequent 'ACT' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Table 3.11.5.1.1
assert_cmdPRE2cmdREF
Cross-referenced at: tRP
tRP is the minimum cycles between the 'PRE' command and the subsequent 'REF' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Table 3.11.5.1.1
assert_cmdPRE2cmdMRS
Cross-referenced at: tRP
tRP is the minimum cycles between the 'PRE' command and the subsequent 'MRS' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Table 3.11.5.1.1
assert_cmdPRE2cmdSRE
Cross-referenced at: tRP
tRP is the minimum cycles between the 'PRE' command and the subsequent 'SRE' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Table 3.11.5.1.1
assert_cmdPRE2cmdPDE
Cross-referenced at: tRP
tRP is the minimum cycles between the 'PRE' command and the subsequent 'PDE' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Table 3.11.5.1.1
assert_cmdPRE2cmdDPDE
Cross-referenced at: tRP
tRP is the minimum cycles between the 'PRE' command and the subsequent 'DPDE' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Table 3.11.5.1.1
assert_cmdPREA2cmdACT

Cross-referenced at: tRP
tRP is the minimum cycles between the 'PREA' command and the subsequent 'ACT' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Table 3.11.5.1.1
assert_cmdPREA2cmdREF
Cross-referenced at: tRP
tRP is the minimum cycles between the 'PREA' command and the subsequent 'REF' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Table 3.11.5.1.1
assert_cmdPREA2cmdMRS
Cross-referenced at: tRP
tRP is the minimum cycles between the 'PREA' command and the subsequent 'MRS' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Table 3.11.5.1.1
assert_cmdPREA2cmdSRE
Cross-referenced at: tRP
tRP is the minimum cycles between the 'PREA' command and the subsequent 'SRE' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Table 3.11.5.1.1
assert_cmdPREA2cmdPDE
Cross-referenced at: tRP
tRP is the minimum cycles between the 'PREA' command and the subsequent 'PDE' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Table 3.11.5.1.1
assert_cmdPREA2cmdDPDE
Cross-referenced at: tRP
tRP is the minimum cycles between the 'PREA' command and the subsequent 'DPDE' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Table 3.11.5.1.1
assert_cmdREF2cmdACT
Cross-referenced at: tRC
tRC is the minimum cycles between the 'REF' command and the subsequent 'ACT' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.9 Sec 3.11.5.1.8
assert_cmdREF2cmdPRE
Cross-referenced at: tRC
tRC is the minimum cycles between the 'REF' command and the subsequent 'PRE' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.9 Sec 3.11.5.1.8
assert_cmdREF2cmdPREA
Cross-referenced at: tRC
tRC is the minimum cycles between the 'REF' command and the subsequent 'PREA' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.9 Sec 3.11.5.1.8
assert_cmdREF2cmdREF
Cross-referenced at: tRC
tRC is the minimum cycles between two 'REF' commands.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.9 Sec 3.11.5.1.8
assert_cmdREF2cmdMRS
Cross-referenced at: tRC
tRC is the minimum cycles between the 'REF' command and the subsequent 'MRS' command.

JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.9 Sec 3.11.5.1.8
assert_cmdREF2cmdSRE
Cross-referenced at: tRC
tRC is the minimum cycles between the 'REF' command and the subsequent 'SRE' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.9 Sec 3.11.5.1.8
assert_cmdREF2cmdPDE
Cross-referenced at: tRC
tRC is the minimum cycles between the 'REF' command and the subsequent 'PDE' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.9 Sec 3.11.5.1.8
assert_cmdREF2cmdDPDE
Cross-referenced at: tRC
tRC is the minimum cycles between the 'REF' command and the subsequent 'DPDE' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.9 Sec 3.11.5.1.8
assert_cmdMRS2cmdACT
Cross-referenced at: tMRD
tMRD is the minimum cycles between the 'MRS' command and the subsequent 'ACT' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.8 Sec 3.11.5.1.7
assert_cmdMRS2cmdPRE
Cross-referenced at: tMRD
tMRD is the minimum cycles between the 'MRS' command and the subsequent 'PRE' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.8 Sec 3.11.5.1.7
assert_cmdMRS2cmdPREA
Cross-referenced at: tMRD
tMRD is the minimum cycles between the 'MRS' command and the subsequent 'PREA' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.8 Sec 3.11.5.1.7
assert_cmdMRS2cmdREF
Cross-referenced at: tMRD
tMRD is the minimum cycles between the 'MRS' command and the subsequent 'REF' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.8 Sec 3.11.5.1.7
assert_cmdMRS2cmdMRS
Cross-referenced at: tMRD
tMRD is the minimum cycles between the 'MRS' command and the subsequent 'MRS' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.8 Sec 3.11.5.1.7
assert_cmdMRS2cmdSRE
Cross-referenced at: tMRD
tMRD is the minimum cycles between the 'MRS' command and the subsequent 'SRE' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.8 Sec 3.11.5.1.7
assert_cmdMRS2cmdPDE
Cross-referenced at: tMRD
tMRD is the minimum cycles between the 'MRS' command and the subsequent 'PDE' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.8 Sec 3.11.5.1.7
assert_cmdMRS2cmdDPDE

Cross-referenced at: tMRD
tMRD is the minimum cycles between the 'MRS' command and the subsequent 'DPDE' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.8 Sec 3.11.5.1.7
assert_cmdSRX2cmdACT
Cross-referenced at: tXS
tXS is the minimum cycles between the 'SRX' command and the subsequent 'ACT' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.16 Sec 3.11.5.1.15
assert_cmdSRX2cmdPRE
Cross-referenced at: tXS
tXS is the minimum cycles between the 'SRX' command and the subsequent 'PRE' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.16 Sec 3.11.5.1.15
assert_cmdSRX2cmdPREA
Cross-referenced at: tXS
tXS is the minimum cycles between the 'SRX' command and the subsequent 'PREA' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.16 Sec 3.11.5.1.15
assert_cmdSRX2cmdREF
Cross-referenced at: tXS
tXS is the minimum cycles between the 'SRX' command and the subsequent 'REF' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.16 Sec 3.11.5.1.15
assert_cmdSRX2cmdMRS
Cross-referenced at: tXS
tXS is the minimum cycles between the 'SRX' command and the subsequent 'MRS' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.16 Sec 3.11.5.1.15
assert_cmdSRX2cmdSRE
Cross-referenced at: tXS
tXS is the minimum cycles between the 'SRX' command and the subsequent 'SRE' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.16 Sec 3.11.5.1.15
assert_cmdSRX2cmdPDE
Cross-referenced at: tXS
tXS is the minimum cycles between the 'SRX' command and the subsequent 'PDE' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.16 Sec 3.11.5.1.15
assert_cmdSRX2cmdDPDE
Cross-referenced at: tXS
tXS is the minimum cycles between the 'SRX' command and the subsequent 'DPDE' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.16 Sec 3.11.5.1.15
assert_cmdPDX2cmdACT
Cross-referenced at: tXP
tXP is the minimum cycles between the 'PDX' command and the subsequent 'ACT' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.14 Sec 3.11.5.1.13
assert_cmdPDX2cmdPRE
Cross-referenced at: tXP
tXP is the minimum cycles between the 'PDX' command and the subsequent 'PRE' command.

JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.14 Sec 3.11.5.1.13
assert_cmdPDX2cmdPREA
Cross-referenced at: tXP
tXP is the minimum cycles between the 'PDX' command and the subsequent 'PREA' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.14 Sec 3.11.5.1.13
assert_cmdPDX2cmdREF
Cross-referenced at: tXP
tXP is the minimum cycles between the 'PDX' command and the subsequent 'REF' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.14 Sec 3.11.5.1.13
assert_cmdPDX2cmdMRS
Cross-referenced at: tXP
tXP is the minimum cycles between the 'PDX' command and the subsequent 'MRS' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.14 Sec 3.11.5.1.13
assert_cmdPDX2cmdSRE
Cross-referenced at: tXP
tXP is the minimum cycles between the 'PDX' command and the subsequent 'SRE' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.14 Sec 3.11.5.1.13
assert_cmdPDX2cmdPDE
Cross-referenced at: tXP
tXP is the minimum cycles between the 'PDX' command and the subsequent 'PDE' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.14 Sec 3.11.5.1.13
assert_cmdPDX2cmdDPDE
Cross-referenced at: tXP
tXP is the minimum cycles between the 'PDX' command and the subsequent 'DPDE' command.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.14 Sec 3.11.5.1.13

5.4 : Requirements Set 3: Miscellaneous SDR Requirements
param_mrsBL
Check parmeter for "Burst Length".
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.6 Sec 3.11.5.1.3
param_mrsCL
Check parmeter for "CAS Latency".
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.6 Sec 3.11.5.1.3
assert_illegal_command
Cross-referenced at: Illegal command
Illegal command check.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.3 Sec 3.11.5.1.1
assert_DQM_for_BST
Cross-referenced at: DQM for write burst terminate
When terminating a burst write command with the BST, DQM should be high.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.17 Sec 3.11.5.1.16
assert_read_data_stable
Cross-referenced at: Clock suspend

CKE timing for Clock suspend.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.15 Sec 3.11.5.1.14
assert_illegal_ap_command
Cross-referenced at: Illegal command in Full Page Burst Mode
Illegal command check in the Full Page Burst Mode.
JEDEC Standard No.21-C(3_11_05_01R12.pdf) P.18 Sec 3.11.5.1.17

5.5 : Requirements Set 4: SDR Arbiter Requirements
When multiple agents issue transfers to the same memory device, proper arbitration is required in the controller to avoid data collision (or loss of data). Common arbitration requirements include:

- When an agent requests memory access but no other agent requests memory access, the next grant should belong to the requesting agent.
- When an agent requests memory access, it should receive a grant within a number of transactions equal to the number of agents or according to its priority.

5.6 : Requirements Set 5: SDR End-to-End Datapath Requirements
A DDR controller can be seen as a bridge that transfers address, control, and data information from requesting agents to memory devices. Formally verifying that there is no corruption, duplication, or droppage of the information is critical to ensure proper functioning. Common end-to-end datapath requirements include:

- Address (row and column) should propagate correctly to the memory.
- Control (read/write) should propagate correctly to the memory.
- Data masking information should propagate correctly to the memory.
- Data should propagate to the memory without corruption, duplication, or droppage.

5.7 : Behaviors
cover_valid_write_data
Indicate valid data cycle for write command.
cover_valid_read_data
Indicate valid data cycle for read command.
cover_cmdDES
Issued a command for "Device Deselect"
cover_cmdNOP
Issued a command for "No Operation"
cover_cmdACT
Issued a command for "Activate"
cover_cmdRD
Issued a command for "Burst Read"
cover_cmdWR
Issued a command for "Burst Write"

cover_cmdBST
Issued a command for "Burst Terminate"
cover_cmdPRE
Issued a command for "Precharge per bank"
cover_cmdPREA
Issued a command for "Precharge all bank"
cover_cmdREF
Issued a command for "Refresh"
cover_cmdMRS
Issued a command for "Mode Register Set"
cover_cmdSRE
Issued a command for "Self Refresh Entry"
cover_cmdSRX
Issued a command for "Self Refresh Exit"
cover_cmdPDE
Issued a command for "Power Down Entry"
cover_cmdPDX
Issued a command for "Power Down Exit"
cover_cmdDPDE
Issued a command for "Deep Power Down Entry"
cover_cmdDPDX
Issued a command for "Deep Power Down Exit"
cover_cmdRDA
Issued a command for "Burst Read with Auto Precharge"
cover_cmdWRA
Issued a command for "Burst Write with Auto Precharge"
cover_state_IDLE
Covered state for "IDLE"
cover_state_DEEP_POWER_DOWN
Covered state for "Deep Power Down"
cover_state_SELF_REFRESH
Covered state for "Self Refreshing"
cover_state_POWER_DOWN
Covered state for "Power Down"
cover_state_BANK_ACTIVE
Covered state for "Bank Active"
cover_state_WRITING
Covered state for "WRITING"
cover_state_READING
Covered state for "READING"
cover_state_PRECHARGING
Covered state for "Precharging"

cover_state_WRITING_AUTO
Covered state for "WRITING with Autoprecharge"
cover_state_READING_AUTO
Covered state for "READING with Autoprecharge"

> IPK style of property naming convention can be enabled with define macro `IPK_STYLE_SUPPORT`.

6

Chapter 6 _ Strategy

Use this section to understand techniques and strategies to effectively use your SDR high-level requirements model during a proof.
This section includes:

- Verification Plan
- Proof Speed Strategy
- Alternate Strategies
  6.1 : Verification Plan
  The Verification Plan contains a set of restriction definitions and the recommended verification steps. The combination of restrictions and steps forms an effective methodology.
  This Verification Plan covers the following:
- Restriction Definitions
- Verification Steps
  6.1.1 : Restriction Definitions
  Depending on the maturity of the design, you may choose to skip the four-phase restriction sequence listed below until you are ready for full verification. Restrictions let you verify basic functionality and mainstream problems before complicated corner-case bugs. They also let you verify a design before it is complete.
  1. Phase 1: Maximum restriction - Burst size 1, no refresh, single bank, no data masking, no power-down, one memory agent, cache latency set to 3, no mode register access
- WRITE only
- READ only
- WRITE and READ
  2. Phase 2: Relax burst size to all possibilities
- WRITE only
- READ only
- WRITE and READ
  3. Phase 3: Relax number of banks with multiple memory agents

- WRITE only
- READ only
- WRITE and READ
  4. Phase 4: No restrictions
- WRITE only
- READ only
- WRITE and READ
  6.1.2 : Verification Steps
  The following lists the recommended steps for your proof.
  1. Prove Requirement Set 1 with Phase 1 restrictions.
  2. Prove Requirement Set 1 with Phase 2 restrictions.
  At this point the DUV is considered alive.
  3. Prove Requirement Set 5 with Phase 1 restrictions.
  4. Prove Requirement Set 5 with Phase 2 restrictions.
  At this point the main stream bugs should be flushed out.
  5. Prove Requirement Set 1 with Phase 3 restrictions.
  6. Prove Requirement Set 2 with Phase 3 restrictions.
  7. Prove Requirement Set 5 with Phase 3 restrictions.
  8. Prove Requirement Set 1 with Phase 4 restrictions.
  9. Prove Requirement Set 2 with Phase 4 restrictions.
  10. Prove Requirement Set 5 with Phase 4 restrictions.
  At this point the critical functions for the DUV are verified.
  11. Prove Requirement Set 4 with Phase 4 restrictions.
  12. Prove Requirement Set 3 with Phase 4 restrictions.
  Verification completed.
  6.1.3 : Steps for Memory Controllers that Support Multiple Configuration Sets
  For memory controllers that support multiple sets of configurations, it is important to verify all configurations. However, it is often beneficial to first verify one configuration set before proceeding to all configurations. This approach not only helps debug the requirement; it also enables you to set up the analysis region more quickly. Here is one alternative plan to the previously described option:
- Restriction Set 1: Fix (that is, constrain) all parameters, allow commands from one bank
- Restriction Set 2: Fix (that is, constrain) all parameters, allow commands from all banks
- Restriction Set 3: Parameters take on all legal values, allow commands from one bank

- Restriction Set 4: Parameters take on all legal values, allow commands from all banks

6.2 : Techniques for Speeding Up Your Proof

In some cases, your proof can benefit from an advanced methodology technique. For example, various abstraction techniques are required when the number of arbiter clients is high. Abstraction is one technique that can simplify many complex structures without compromising the integrity of the proof. In this section, we introduce a few advanced techniques that can be useful for speeding up your proof.

6.2.1 : Taking Advantage of Design Symmetry

Many types of designs (such as pointer-based, round-robin arbiters) have a symmetrical implementation. That is, the logic associated with each port has the exact same RTL structure. Generally, the only exception to a symmetrical arbiter implementation is the initial priority value for each port (that is, you will give one port the highest priority).

Symmetry allows us to efficiently prove requirements on a pair of symmetrical paths (for example, ports of an arbiter) while giving us confidence that the requirement is true on all ports due to the symmetrical implementation. If you are confident about the symmetry among ports in the implementation of your design, then we recommend you take advantage of this symmetry as part of your proof.

Arbiter example:

Consider the following Verilog ® code fragment for an arbiter that has a symmetrical implementation per port with the exception of the initial priority value:

```
// pointer to determine round-robin
reg [3:0] priority_pointer;
// priority port
reg [7:0] grant;
always @(posedge clk) begin
if (reset) begin
priority_pointer <= 4 €™d0;
grant <= 8 €™d0;
end
else if (grant !=8 €™d0) grant <= 8 €™d0;
else if (request[priority_pointer]) begin
priority_pointer <= priority_pointer + 5 €™d1;
grant[priority_pointer] <= 1 €™b1;
end
else if (request[priority_pointer+5 €™d1]) begin
priority_pointer<=priority_pointer + 5 €™d2;
grant[priority_pointer + 5 €™d1] <= 1 €™b1;
```

```
end
else if (request[priority_pointer+5 €™d2]) begin
priority_pointer<=priority_pointer + 5 €™d3;
grant[priority_pointer + 5 €™d2] <= 1 €™b1;
end
.
.
.
end
```

In the previous example, you can see that the logic associated with each port is symmetrical since each n-bit variable assignment follows the same RTL structural path. The only part of this implementation that is not symmetrical is the reset value of priority_pointer, which points to port0 after reset (that is, it has the highest priority after reset). To allow true symmetry for our proof, we can use the following technique to ignore the initial value.

- Technique: Use the justify -initValue option.
  justify -initValue priority_pointer false
  This option permits all possible initialization values for the signal priority_pointer to be explored, which restores the true symmetry to the circuit. In the induction section (below), you will see that this option is useful for proving other aspects of an arbiter.

  6.2.2 : Multiplexer Abstraction

  Multiplexers are frequently used to implement arbitration schemes that involve multiple ports. For the case where the port count is high, multiplexers increase the complexity of the proof and present a problem. In "What Ports Should You Monitor as Part of Your Requirements?" (refer to any arbiter strategy section, for example, Fairness), we introduced the technique of specifying a few arbiter requirements by using a fixed (but arbitrary) port p1 as part of our specification. In fact, most arbiter requirements can be represented in terms of p1 and/or p2 instead of specifying all ports or a fixed port number directly. Since our objective is to identify a bug in the control logic for our arbiter, we can take advantage of the technique we present in this section to reduce the complexity of the multiplexer requirements model (as well as the design itself during the proof).

  Consider the example:

  In Figure 1, we illustrate the concept of reducing the complexity of the multiplexer. For this example, we have taken an N-to-1 multiplexer (shown on the left) and reduced it to a 3-to-1 multiplexer (shown on the right). Our abstraction is safe since, for our proof, we only need to distinguish two arbitrary ports p1 and p2 (represented as €œPort P1 € and €œPort P2 € in the figure). Since we are not distinguishing the behavior of all other ports as part of our

requirement, we can represent the behavior of all other ports using a single port we have labeled €œPort Other € (refer to Figure 1). Jasper ® Verification System explores the behavior of the ports we must distinguish (that is, p1 and p2) in the context of the behavior for all the other ports that we do not need to distinguish.
For example, consider the original model:

```
always @(port[0] or port[1] or port[2] or port[3] or port[4] or port[5] or
port[6] or port[7] or round_robin_ptr) begin
case(round_robin_ptr)
0: port_out = port[0];
1: port_out = port[1];
.
.
.
15: port_out = port[15];
endcase
end
```

There are two abstraction techniques we can use to simplify this code when proving the requirements in the context of two arbitrary ports, p1 and p2.

- Technique 1: RTL Code Version of Abstracted Multiplier Model

```
wire [2:0] port_other;
always @(port[0] or port[1] or port[2] or port[3] or port[4] or port[5] or
port[6] or port[7] or round_robin_ptr or p1 or p2) begin
if (round_robin_ptr==p1) port_out=port[p1];
else if (round_robin_ptr==p2) port_out=port[p2];
else port_out = port_other;
end
```

Using the above model, you should add the following Tcl commands to your script:

```
assume {port_other!=port[p1]}
assume {port_other!=port[p2]}
```

- Technique 2: Tcl Script Version of Abstracted Multiplexer Model

```
stopat {port_out}
assume {round_robin==p1 => port_out==port[p1]}
assume {round_robin!=p1 => port_out!=port[p1]}
assume {round_robin==p2 => port_out==port[p2]}
assume {round_robin!=p2 => port_out!=port[p2]}
```

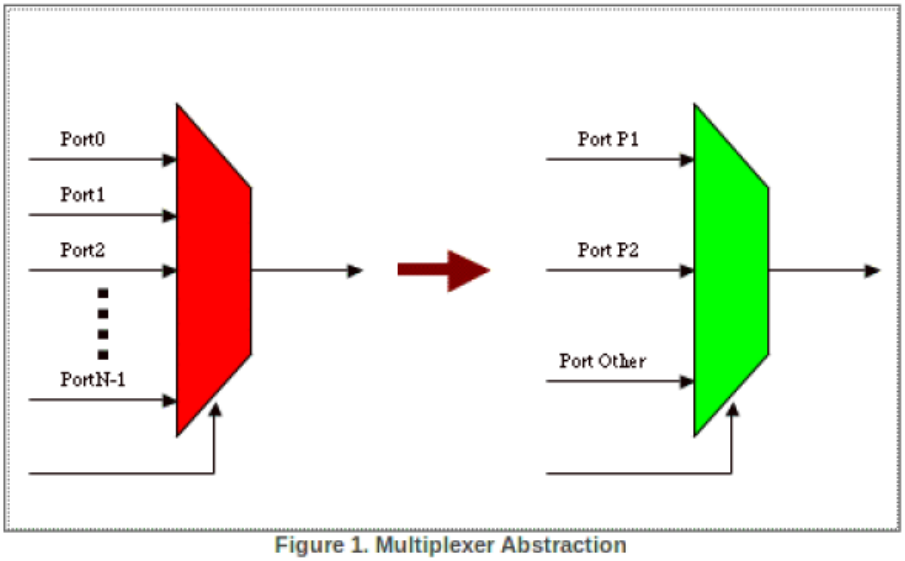


Figure 1. Multiplexer Abstraction

6.2.3 : FIFO Abstraction

Although in general, FIFOs are more likely to be found in the datapath portion of a design rather than the actual arbiter, there are some arbiter implementations that take advantage of a FIFO structure. Consider a round-robin arbiter that uses registers to store the priority for each port. Every time a grant is provided, the priority changes, which affects at least one of the priority registers, and potentially affects all of the priority registers. This priority scheme using multiple registers, in a way, behaves similar to a FIFO (except it may not always behave in a first-in first-out fashion). Since the set of priority registers can be large, we might need to replace this set of large registers with an abstracted model. The following example demonstrates this technique.

Example:

Consider a simple 32-location, circular FIFO-type structure that is keeping track of some type of information (for example, its priority) related to each port of our arbiter. Information enters our FIFO at location 31 and is shifted as new information arrives until it exits at location 0. Essentially, for an arbiter design, the information indicating high priority at location 0 will move to location 31 using this circular FIFO structure. By using the basic ideas related to p1 and p2, which we previously discussed in What Ports Should You Monitor as Part of Your Requirements? (refer to any arbiter strategy section), we are able to dramatically simplify the FIFO and reduce the complexity of the design for our proof. That is, instead of keeping track of the content for every FIFO location, we only need to keep track of the information for p1 and p2, along with their locations within the FIFO. You will see that this reduces our large 32-location FIFO down to four smaller registers and enables us to prove complex properties related to the FIFO controller.

The following Verilog RTL code fragment demonstrates the original RTL code for the FIFO.

```
reg [15:0] fifo0;
reg [15:0] fifo1;
reg [15:0] fifo2;
.
.
.
reg [15:0] fifo31;
always @ (posedge clk) begin
if (reset) begin
fifo0 <= 16 €™h0;
fifo1 <= 16 €™h0;
.
.
.
fifo31 <= 16 €™h0;
end
else if (fifo_en) begin
fifo31 <= new_info;
fifo30 <= fifo31;
fifo29 <= fifo30;
.
.
.
fifo0 <= fifo1;
info_out <= fifo0;
end
end
```

The following Verilog RTL code fragment demonstrates our FIFO abstraction technique.

```
reg [15:0] p1_info;
reg [15:0] p2_info;
reg [4:0] p1_loc;
reg [4:0] p2_loc;
always @(posedge clk) begin
if (reset) begin
p1_info <= 16 €™h0;
p2_info <= 16 €™h0;
//initial position of p1 within the fifo
```

```
p1_loc <= p1_init;
//initial position of p2 within the fifo
p2_loc <= p2_init;
end
else if (fifo_en) begin
p1_loc <= p1_loc  €" 5 €™h1;
p2_loc <= p2_loc = 5 €™h1;
if (p1_loc <= 5 €™h0) begin
p1_info <= new_info;
info_out <= p1_info;
end
else if (p2_loc <= 5 €™h0) begin
p2_info <= new_info;
info_out <= p2_info;
end
end
end
```

6.2.4 : Induction

One of the main advantages of using Jasper Verification System is that it provides the trace that demonstrates the requirement violations with a minimum number of clock cycles. This feature makes the full verification a lot faster and a lot easier to debug. However, certain structures can unnecessarily increase the number of cycles required to demonstrate the bug. These situations are usually caused by structures where the function being evaluated follows this type of relationship:

f(x+1) = g[f]

where f[0] is constant. One of the most common examples is a counter. In the fair arbiter Overview there are examples of functions that follow this relationship.

For example, the logic that generates the priority pointer values for the round-robin arbiter follows this relationship. Similarly, in the FIFO abstraction example shown above, the logic that generates location pointer values for p1 and p2 follows this relationship. If we can ignore the initial value for these pointer examples, thus allowing all possible initialization conditions, we can shorten the number of cycles required to verify the requirement by using induction.

Induction Example

The round-robin pointers are initialized to zero. To prove the requirements, the design must go through the states containing every pointer value. If the pointer goes from 0 to 31, it will take at least 32 transactions to complete the transfer. However, if we free up the initial value of the pointers to allow any initial values, it only takes one transaction to reach all pointer values

from 0 to 31. Consequently, the number of cycles needed to prove the requirements is significantly smaller.

Being able to prove the requirements without needing the initial value also implies that the design is symmetrical. That is, if the design is truly symmetrical, we only need to prove the interface requirements by monitoring one port and fairness requirements by monitoring one pair of ports.

The next section demonstrates how to use induction to simplify a proof that involves a large counter.

6.2.5 : Counter Abstraction

Counters are one of the most common components used in a design. If a counter is large, it can be a challenge for any formal verification tool because it can cause the number of evaluation iterations to be very high.

In an arbiter, there are several places that require counters. Dynamic arbiters, for example, typically use a counter to keep track of the arbitration change events. Fortunately, it is usually possible to abstract these counters to reduce the reachable states without compromising the proof.

There are two main counter abstractions: Counter Induction and Counter Reduction.

Counter Induction: The first type uses induction to abstract the counter. For example, consider a dynamic priority arbiter that requires us to prove that the credit/weight is incremented, reset, and decremented correctly. If we follow the actual logic where the credit is reset to a fixed value, we will need to go through many cycles to explore all possible credit values.

Alternatively, we can partition the requirement into two parts:

1. The credit register is initialized correctly.
2. The credit register is incremented and decremented correctly.

The first requirement can be modeled as follows:

```
// credit_init is the expected initialization value of credit and prev_rst is the
// registered version of reset, hence indicate the value of credit during the
// very first cycle
wire err_credit_reset = prev_rst & (credit!=credit_init);
```

The second requirement can be modeled as follows:

```
wire credit_increment = credit_inc & (credit!=prev_credit+1);
wire credit_decrement = credit_dec & (credit!=prev_credit-1);
```

In addition, for the second requirement you must let the initial credit value be free (that is, it is uninitialized and allowed to take on any value). Add the following command to your Jasper Tcl script to let the initial credit value be free:

```
justify  €"initValue {credit} false
```

Counter Reduction: Use the second type of counter abstraction when a specific counter value

triggers a specific event (for example, a timeout counter). Since most of the counting values are uninteresting, and are only used as a sequence to generate the next interesting counter value, we can perform an abstraction to reduce the number of states the formal engine needs to evaluate.

For example, assume our design has a counter that calculates the elapsed time before the design must refill its credit for a credit-based arbiter. For this type of design, there are only two counter values that are interesting (that is, zero and the timeout count to generate the credit refill). All other counter values have no influence or impact on the rest of the design. Hence, we can use an abstraction technique that will fast forward the counter values that are uninteresting and thus reduce the state-space needed to prove the design's high-level requirements.

Original counter model:

```
reg [15:0] count;
always @(posedge clk) begin
if (rst) count <= 16 €™h0;
else if (count == credit_refresh_cnt) count <= 16 €™h0;
else count <= count + 16 €™h1;
end
```

Abstracted counter model:

```
reg abstract_cnt;
reg [15:0] nxt_count;
always @(posedge clk) begin
if (rst) abstract_cnt <= 1 €™b0;
else if (count==credit_refresh_cnt) abstract_cnt <= 1 €™b0;
else abstract_cnt <= 1 €™b1;
end
```

For the abstracted counter, add the following to your Jasper Tcl script:

```
stopat count
```

1. count can assume any value less than or equal to credit_refresh_cnt
   ```
   assume {abstract_cnt => (count<=credit_refresh_cnt)}
   ```
2. count will be set back to zero
   ```
   assume {~abstract_cnt => (count == 16 €™h0)}
   ```
   In our previous example, during reset, the value of abstract_cnt is set to zero, which in turn sets count to zero with the Tcl assumption. After that, abstract_cnt is set to one, which allows count to assume any (and all) values between zero and the credit_refresh_cnt with the Tcl assumption. If there are no meaningful states for Jasper Verification System to analyze

between zero and credit_refresh_cnt, then count will not arbitrarily assume a value that is less than credit_refresh_cnt for any time longer than necessary. Hence, count will assume credit_refresh_cnt (an interesting state) sooner without having to sequence through all the counter values. When count assumes the value of credit_refresh_cnt, abstract_cnt resets back to zero, which in turn resets count back to zero.

One advantage for this type of counter abstraction is that if we miss some meaningful counts (counts that impact the rest of the design) when we are creating our abstract model, our proof returns a false with a counter example. Thus, this form of abstraction might give you a false negative, but it does not give you a false positive. If there are more counter values needed in our abstracted model, we can then declare more states and map these states onto states in our abstracted model. This type of counter abstraction is also useful when counting the number of remaining credits in the credit-based dynamic arbiter.

6.2.6 : Separating Decoding Logic

Separating the decoding logic from the logic used to ensure fairness is especially useful for arbiters that involve dynamic prioritizations and fixed priority arbiters. The essential idea is that for many arbiters there is decoding logic that is responsible for generating the various priority levels based on multiple factors.

For example, assume we are designing an arbiter where the priority of a specific port is increased from normal to high whenever the input FIFO for that port reaches the high watermark. The priority increases further from high to highest when the FIFO is full. All ports with the same priority levels (normal, high, and highest) are arbitrated using a round-robin scheme. However, each priority level has strict priority over the lower priority levels.

To write a high-level requirement for this combined round-robin and priority scheme can be complicated. In addition, proving this combined scheme can be complicated for a formal tool. However, if we separate the verification problem into two parts, we can simplify the coding of our high-level requirement and simplify the proof. The requirement can be partitioned as follows:

1. Verify that the arbitration requirement is met.
2. Verify that the generation of the priority is correct.

For example, to separate the logic that generates the priority from the logic that implements the arbitration scheme, add the following command to your Jasper Tcl script when you prove that the arbitration requirement is correct:

```
stopat {priority[15:0]}
```

Then, in your requirements model, code the following abstract priority model:

```
// If there is at least one port with high priority, the effective
// requests are the ports with request and high priority.
// Otherwise, it is the same as a request
```

```
// if there is no high priority request.
reg [15:0] priority_req = (priority==16 €™b0) ? (priority&req) : req;
// Use priority_req in place of regular request for any
// requirements that are applicable
```

There are also a few assumptions that might be needed to model the behavior of the abstracted priority model correctly. For example, whenever a priority is high, then the priority should not change to low priority prior to receiving a grant. This assumption requires additional modeling in the requirement as shown:

```
// registered previous value of priority
reg [15:0] prev_priority;
// registered previous value of grant
reg [15:0] prev_gnt;
always @(posedge clk) begin
if (rst) begin
prev_priority <= `NUM_PORT €™h0;
prev_gnt <= `NUM_PORT €™h0;
end
else begin
prev_priority <= priority;
prev_gnt <= gnt;
end
end
```

Then add the following assumptions to your Tcl script:

```
assume {(~prev_gnt[0] & prev_priority[0]) => (priority[0]==1'b1)}
assume {(~prev_gnt[1] & prev_priority[1]) => (priority[1]==1'b1)}
.
.
.
```

Using Various Jasper Engines for Improved Performance

One of the main steps involved in proving an arbiter with Jasper Verification System is the Design Tunneling „¢ process. During the initial phase of Design Tunneling, if either the prove or trace commands are slow, you might want to select the Jasper engineB to enlarge the analysis region during the initial Design Tunneling. You can select engineB by keying in the following Tcl command:

```
set_engine_mode engineB
```

Continue using engineB until you reach a good number of iteration cycles. Since each design is different, what is considered a good number of iterations may be different. However, when a

single Design Tunneling step increases the number of iterations by several cycles, it might indicate the analysis region has reached an efficient size. At this point, stop the current proof and use the following command to switch back to the default engine and continue the proof:
set_engine_mode default

7

Chapter 7 _ Configure the Proof Kit

This chapter captures the information on how to configure and customize the Proof Kit for your design. Once you have captured the target configuration, run the tcl script "generateCustomProofKit.tcl" to generate the custom verilog wrapper and the tcl script for running the proof kit in Jasper's products.
Note that you can see an overview of these configuration parameters by going to the "Configuration" report in the "report" tab. And you can get an overview of the useful scripts by going to the "Data file" report in the "report" tab.

7.1 : Parameter List

7.1.1 : BA_ADDR_WDTH

Bank address width.
default is 2.

7.1.2 : ADDR_WDTH

Address width.
default is 14.

7.1.3 : DQ_WIDTH

Data width.
default is 16.

7.1.4 : DQM_WIDTH

Data mask width.
default is DQ_WIDTH/8.

7.1.5 : mrsBL

Burst length : A2, A1, A0
A2 | A1 | A0 | BL
0 0 0 1(optional)
0 0 1 2
0 1 0 4
0 1 1 8
1 0 0 Reserved
1 0 1 Reserved
1 1 0 Reserved
1 1 1 Full Page Burst(optional)
default is 3'b001.

7.1.6 : mrsCL

CAS latency : A6, A5, A4
A6 | A5 | A4 | CL

0 0 0 Reserved
0 0 1 Reserved
0 1 0 2
0 1 1 3
1 0 0 4
1 0 1 Reserved
1 1 0 Reserved
1 1 1 Reserved
default is 3'b011.
7.1.7 : Trc
ACT to ACT or REF command period.
default is 9.
7.1.8 : Trrd
Active to Active commnad period.
default is 2.
7.1.9 : Trcd
RAS 2 CAS delay time.
default is 3.
7.1.10 : Tras
ACT to PRE command period.
default is 6.
7.1.11 : Trp
PRE command period for a single bank.
default is 3.
7.1.12 : Twr
WRITE Recovery time.
default is 2.
7.1.13 : Tmrd
Mode register set command cycle time.
This parameter is defined in this proof kit.
default is 3.
7.1.14 : Txs
Exit self refresh to next command.
This parameter is defined in this proof kit.
default is 10.
7.1.15 : Txp
Exit power down to next command.
This parameter is defined in this proof kit.
default is 10.

7.1.15 : COVERAGE_ON

All the functional cover properties can be enabled/disabled with the proof kit parameter,

COVERAGE_ON. 0: disables all functional cover properties 1: enables all functional cover properties.

The default value is set to 1.

7.2 : Run Script Setup
This section captures how you want to set up the design hierarchy with the proof kit.
7.2.1 : Design Files
The optional "Proof Kit / Design files" attribute of this element can be configured to include your custom design files.
7.2.2 : Elaboration Top
The optional "Proof Kit Parameter" attribute of this element can be configured to use a custom top module for analysis.
7.2.3 : Bind File
The optional "Proof Kit Parameter" attribute of this element can be configured to bind the proof kit to a specific module by specifying the bind file which contains the bind statement for the custom design and the proof kit. The bind statement can be as basic as "bind <top-level design module> <proof kit module> <proof kit instance> (.*);" This parameter is mandatory when you are using a custom top module.
7.2.4 : Clock Name
This specifies the Clock name.
7.2.5 : Reset Command
This specifies the reset name.
7.3 : Useful Scripts
7.3.1 : Reset Configuration Proof Kit value
This will reset the values of the proof kit parameters to their default values.
Read-only Data file : resetConfigureProofKitValue.tcl
7.3.2 : Generate Custom Proof Kit
This will just generate the custom wrapper and tcl files (as well as output the SVA source code file) based on the inputs given in the "Configure Proof Kit" GUI with the intent to hook it up to a design and verify it.
Read-only Data file : generateCustomProofKit.tcl
7.3.3 : Launch Analysis Session for use in IPK mode only
This will launch a new session in Jasper Apps where the proof kit is configured for use in IPK only, like for visualize recipes. This means the proof kit hasn't been configured to be hooked up to a design.
NOTE: This is the script used by default when the IPK is first loaded.
Read-only Data file : launchSessionWithIPK.tcl
7.3.4 : Launch Analysis Session with Custom Proof Kit
This will launch a new session in Jasper Apps, using the custom wrapper and tcl files based on the inputs given in the "Configure Proof Kit GUI", with the intent to hook it up to a design and verify it.
Read-only Data file : launchSessionWithCustom.tcl

8

Chapter 8 _ Proof Kit RTL and Various Templates

8.1 : Proof Kit Verilog Code
This datafile captures the proof kit source code.
Read-only Data file : cdn_abvip_sdram.sv

Read-only Data file : cdn_abvip_sdram_monitor.sv
8.2 : Proof Kit Encrypted Verilog model for Jasper
This is the encrypted verilog model of the SDRAM RTL for Jasper
Read-only Data file : cdn_abvip_sdram_core.svp
8.3 : Proof Kit Standard Wrapper Template
This datafile captures the standard wrapper, which is automatically customable using the scripts in this database.
Read-only Data file : cdn_abvip_sdram_wrapper.sv
8.4 : Proof Kit Tcl Template
This datafile captures the standard tcl commands to elaborate the design with the proof kit, for analysis with Jasper products. It is also automatically customable using the scripts in this database.
Read-only Data file : jg_run.tcl

9

Chapter 9 _ Advanced Debug Files

9.1 : SDRAM Signal File
Read-only Data file : jasper_sdram_list.sig

10

SDRAM Design Example

- Setting up the Environment
- Verifying SDRAM Using Xcelium

The SDRAM ABVIP package contains a set of ABVIP usage examples in the following directory: ${*VIPCAT_INST_DIR*}/tools.platform/abvip/sdram/examples.

Setting up the Environment

To run examples, setup the following environment variables:

1. Set the ABVIP_INST_DIR environment variable to the abvip folder in vipcat install directory as:
   setenv ABVIP_INST_DIR <*VIPCAT_INSTALL_DIR*>/tools/abvip
2. Change to a working directory.

Verifying SDRAM Using Xcelium

1. Copy the sdram examples directory as: `cp -r $ABVIP_INST_DIR/sdram/examples .`
2. Go to examples directory and set the environment variable `example_dir` as:
   ```
   $ setenv example_dir `pwd`
   ```
3. Go to example directory *examples/xcelium*
4. Clean the example - `./CLEAN`
5. To run the example in simulation, specify the following command:
   ```
   xrun -gui -f xrun.f
   ```
6. To run the example in formal verification, specify the following command:
   ```
   xrun -jg -gui -f xrun_jg.f
   ```