Chapter 5

SVA FOR MEMORIES

Memory controller protocol

Computers and consumer electronic devices have huge amount of memory to store multimedia application data. In the ASIC world, almost all the chips that are being designed today use embedded memory (DRAM, SRAM, ROM, etc.). As the memory access time is becoming faster and faster, it becomes very essential that the end product work with multiple memory vendors and with different timing requirements. The major bottleneck in the verification of memory controller interface is the timing of the control signals. This can be effectively done using assertions. The assertions written for a particular type of memory device can be re-used with multiple vendors just by modifying the timing parameters. This chapter discusses developing reusable SVA checkers for different types of memory devices.

5.1 Sample System – Memory controller

The sample system has a CPU that interfaces with a memory controller. The CPU can read and write data to the various memories connected to the memory controller. The memory controller can interface to different type of memories such as SDRAM, DDR-SDRAM, SRAM, Flash, ROM, etc. The block diagram for the sample system is shown in Figure 5-1.

5.1.1 CPU - AHB Interface Operation

The CPU is a generic processor that uses the AHB bus interface to interact with the memory controller. The CPU generates the read/write commands. The CPU also generates the chip select signals for selecting the memory with which the read/write operation will be performed. It supports both separate and shared memory address/data busses to SDRAM and static memories. It supports the AHB data width of 32, 64 or 128 bits. It also supports the AHB 32bit wide address bus.

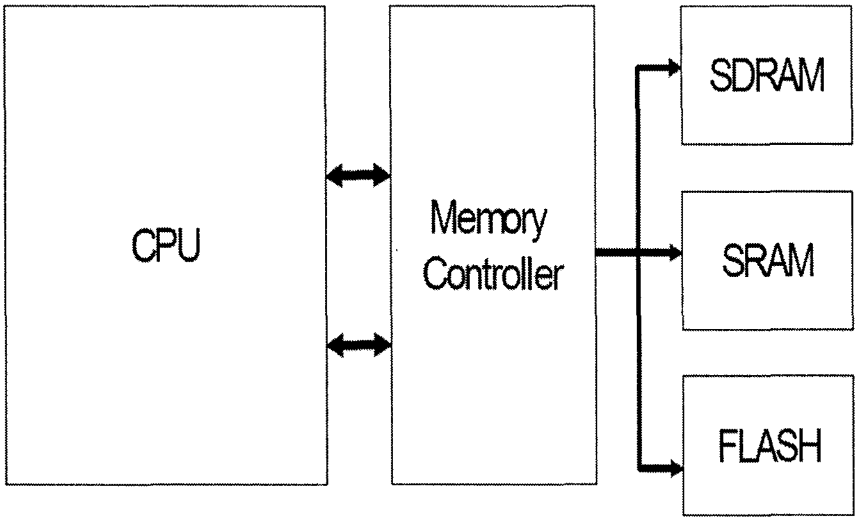

Figure 5-1. System block diagram

Figure 5-2 shows the signal interface between the CPU-AHB bus interface and the memory controller. A brief description of the pins is listed below.

- hclk – clock generated by the CPU
- haddr – read/write address generated by the CPU
- hwdata – write data generated by the CPU
- hrdata – read data to the CPU from the memory
- hready – ready signal from CPU
- hready_resp – memory acknowledge signal for hready
- hsize – configures the size of the data transfer from CPU to memory
  - 00 – 128 bits of data transfer at a time
  - 01 – 64 bits of data transfer
  - 10 – 32 bits of data transfer
- hburst – defines how the memory address is accessed
  - 000 – single – one single memory location
  - 001 – INCR – Increments address 0x40, 0x44 …

- 010 – WRAP4 – Wraps address in 4 word boundaries (0x48, 0x4c, 0x40 0x44)
- 011 – INCR4 – Increments address by 4 words from the current address
- 100 – WRAP8 – Wraps address in 8 word boundaries
- 101 – INCR8 – Increments address in 8 word blocks
- 110 – WRAP16 – Wraps address in 16 word boundaries
- 111 – INCR16 – Increments address in 16 word blocks

- sel_mem, sel_reg – select whether to do transaction with the external memories or the registers

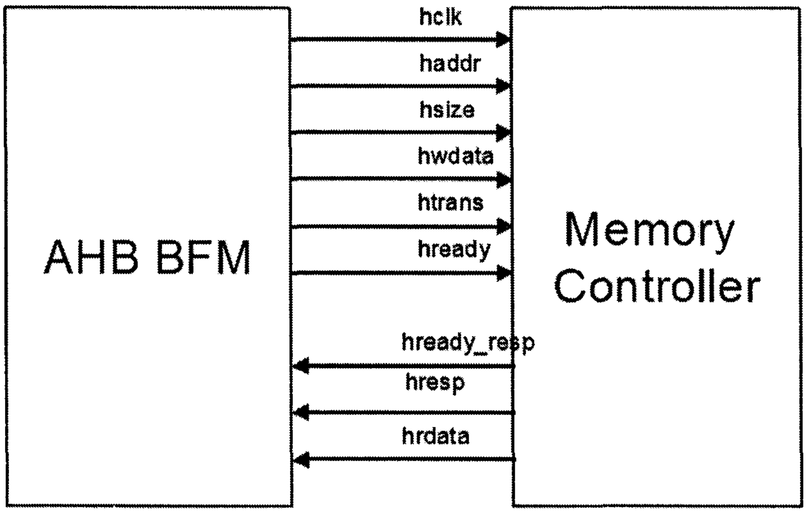

Figure 5-2. CPU block diagram

Figure 5-3 shows the waveform of a CPU write transaction to the memory controller. The CPU initiates a write transaction to the memory controller at marker 1. The size (hsize) is set to "10" and hence the data transaction is 32 bits wide. The burst (hburst) is set to "010" and hence the burst type is WRAP4. Based on the burst type, the address access will be 0x0, 0x4, 0x8 and 0xc. Hence, the write transaction of size 32 bits and of type WRAP4 is initiated at marker 1. The figure shows that the address is incrementing from 80000000 to 8000000c and the data on "hwdata" is being written to the SDRAM.

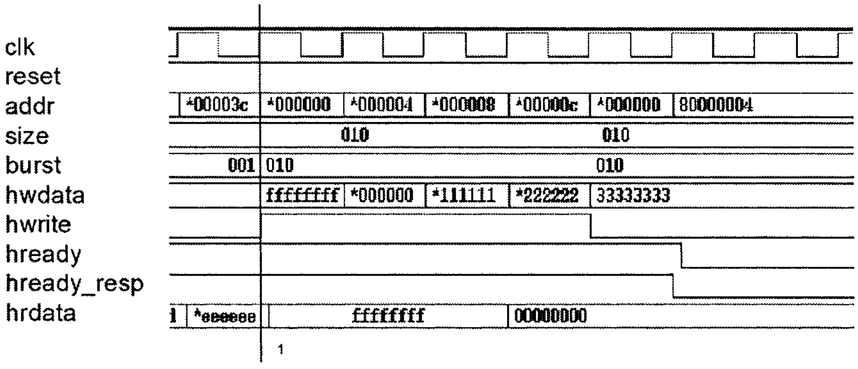

Figure 5-3. CPU-AHB write

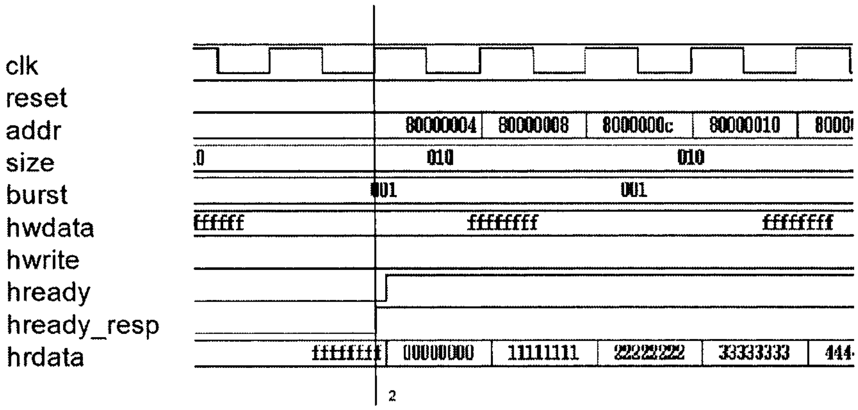

Figure 5-4. CPU-AHB read

Figure 5-4 shows the waveform of a CPU read transaction to the memory controller. A read transaction of size 32 bits and type WRAP4 is initiated at marker 2 when the "ready" and "ready_response" signals are asserted. The figure shows that the same data that was written is being read.

5.1.2 Memory controller operation

The memory controller in the sample system interfaces to the SDRAM, SRAM and Synchronous Flash devices. A block diagram of the connection is shown in Figure 5-5.

SDRAM Interface

The memory controller interface to the SDRAM is generic. The interface is fully synchronous and all signals are registered on the positive edge of the clock. Read and write access to the SDRAM is burst oriented and they start at the address specified by the AHB bus and continue for a programmed number of locations. The connection from the memory controller to the SDRAM is direct and has no glue. It supports 16 SDRAM address bits. It also has programmable row and column address widths. All the SDRAM timing parameters are programmable. It supports auto-refresh with programmable refresh intervals.

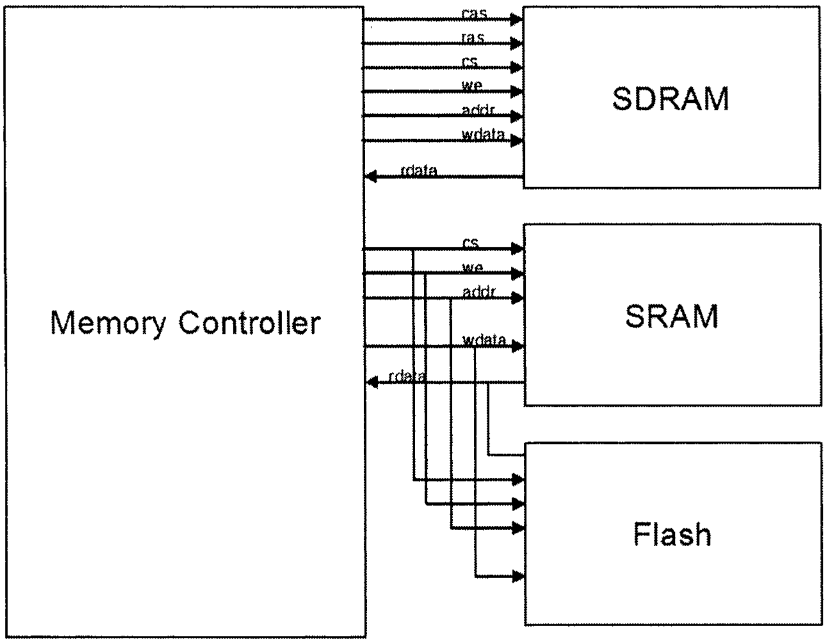

Figure 5-5. Memory Controller Block Diagram

A brief description of the pin connections between the SDRAM and the memory controller is listed below.

- clk – clock input to SDRAM
- ras_ – selects the row address

- cas_ – selects the column address
- we_ – when asserted write signal, when de-asserted read signal
- sel_ – when asserted SDRAM is selected
- data – bi-directional data bus for both reads and writes
- addr – read/write address
- bank_sel – selects a particular bank of SDRAM

A sample waveform for a write command issued by the memory controller is shown in Figure 5-6. A burst write of length four is written and read back. An active command (ras and chip select are asserted) is issued and then a write command (cas, we, and sel are asserted) is issued to the address 0x0000. Figure 5-6 shows that the data is written with a burst length of 4 to the memory at marker 1. Figure 5-7 shows a burst read command issued by the memory controller. The data is read out of the memory (marker 1) after a "cas" latency of two clock cycles.

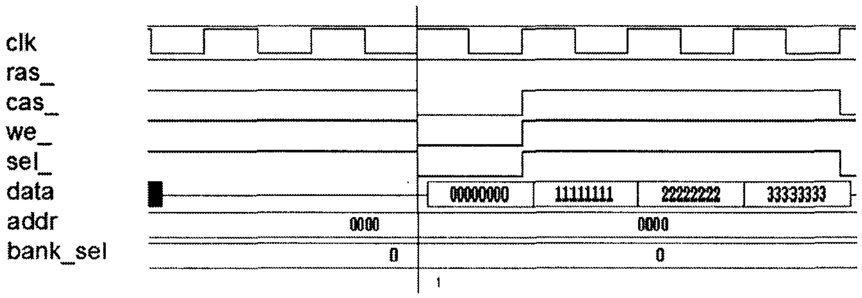

Figure 5-6. SDRAM write operation

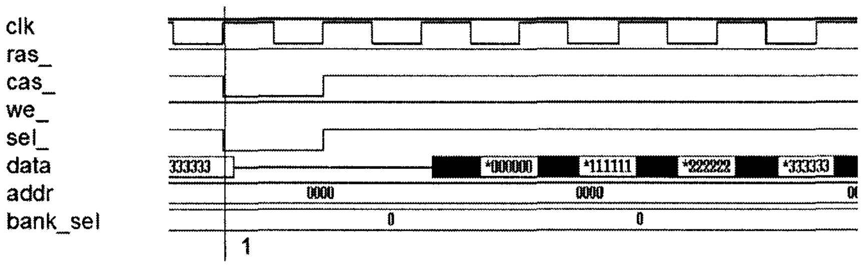

Figure 5-7. SDRAM read operation

SRAM/FLASH Interface

The SRAM/FLASH memories are static memories and the interfaces are similar. The memory controller supports asynchronous SRAMs, page-mode flashes and ROMs. The address width can be configured up to 32 bits. It also has the "ready" handshake signal to support non-SRAM type devices. The memory data width can be configured to 8 bits, 16 bits, 32 bits, 64 bits or 128 bits. The static memory data width can be a minimum of 8 bits instead of the 16 bits standard requirement. The flash memory used in the sample system is write protected, so that the important system information is protected in the boot block.

A brief description of the pin connections between the SRAM and the memory controller is listed below:

- addr – address pins to the static memories from memory controller
- data – data to/from the static memories to the memory controller
- sel_ – chip select pins to select the corresponding static memory
- we_ – write when asserted
- oe_ – output enable asserted during read
- bs_ – byte control pins to enable different data widths

The interface for flash is similar to SRAM except that it has two more signals:

- wp_ – write protect pin
- rp_ – reset power down pin

Figure 5-8 shows a sample waveform for the interface between the memory controller and the SRAM. When signals "we_" and "sel_" are asserted (marker 1), a write is done to the SRAM. Similarly, when signals "sel_" and "oe_" are asserted (marker 2) and signal "we_" is de-asserted, a read is done from the SRAM. Figure 5-9 shows a sample waveform for the interface between the memory controller and the flash memory. The figure shows a burst read from the flash device. When signals "sel_" and "oe_" are asserted, a read operation from the address specified in the address bus (addr) is performed.

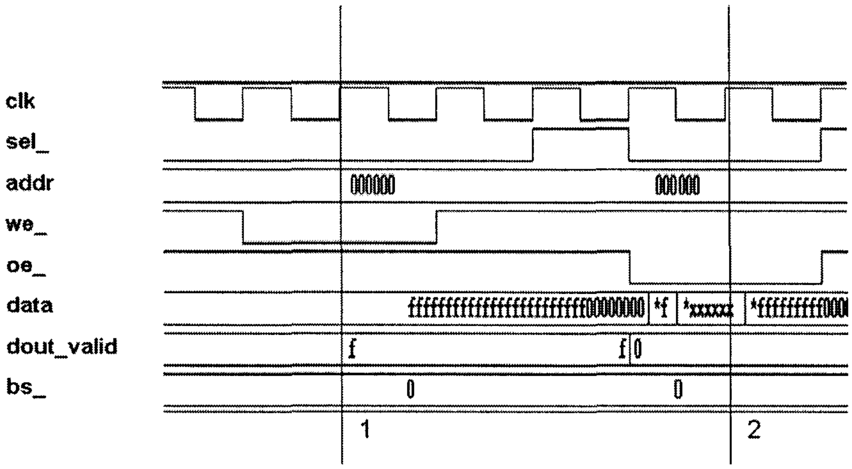

Figure 5-8. SRAM interface signals

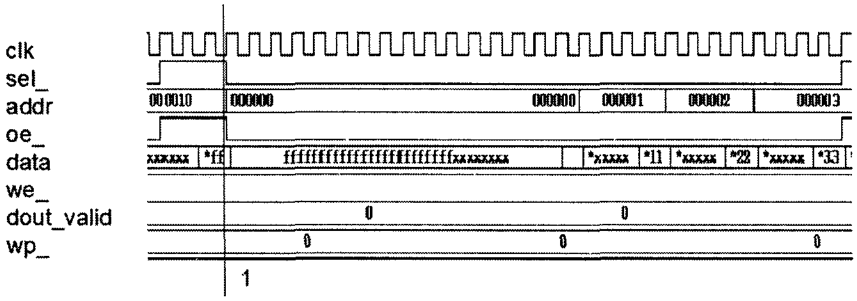

Figure 5-9. Flash interface signals

5.2 SDRAM Verification

This section will discuss how to verify the SDRAM control signals. There are a lot of timing parameters for SDRAM device and assertion based verification can be used effectively to verify that these timing requirements are not violated. The sample system uses the following SDRAM configuration.

512Mb SDRAM - 8M X 16bit X 4 bank

The 512Mb SDRAM under verification is a quad bank SDRAM and includes a synchronous interface. All signals are registered on the positive edge of the clock. Each of the 4 banks is organized as 8192 rows X 1024 columns X 16 bits. Read and write access to the SDRAM is burst oriented. The access starts at a selected location and continues for a programmed number of locations. Read/Write access always begins with an active command followed by a read/write command. The address bit corresponding to the active command denotes the row address and the bank that is selected. A0-A11 denotes the address and BA[1:0] denotes the bank that is being accessed. The address bits corresponding to the read/write command denote the starting column address (denoted by A0-A7).

The different combinations of the SDRAM interface signals sel_, ras_, cas_ and we_ constitute the different commands. All the SDRAM commands are summarized in Table 5-1. The "Command Inhibit" condition prevents the SDRAM from executing the new commands, regardless of whether the clock signal is enabled or not. Operations already in progress will not get affected (sel_ = 1, cas_, ras_, we_ = x).

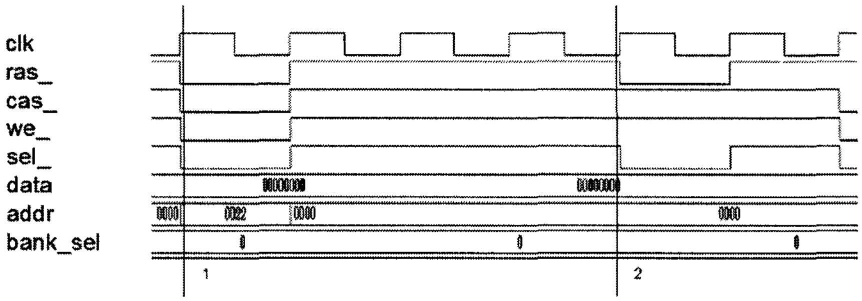

Figure 5-10. Load Mode Register/Active command

- **No Operation**: This prevents unwanted commands from being registered in idle/wait state (sel_ = 0, cas_, ras_, we_ = 1).

- **Load Mode Register**: The register is loaded through the address bus (A0-A11). The Load mode register is issued only when all the banks are idle (sel_, cas_, ras_, we_ = 0). Figure 5-10 shows a "load mode register" operation at marker 1 and an "active" operation at marker 2.

- **Active**: This command is issued to activate/open a row for access. The value on the address bus is the value of the row and the value on the bank_address bus specifies the bank (sel_, ras_ = 0; cas_, we_ = 1).

Table 5-1. SDRAM Commands

Command	Ras	Cas	We	Sel
No Operation	H	H	H	L
Active	L	H	H	L
Read	H	L	H	L
Write	H	L	L	L
Burst Terminate	H	H	L	L
Load Mode Register	L	L	L	L
Precharge	L	H	L	L
Auto-Refresh	L	L	H	L

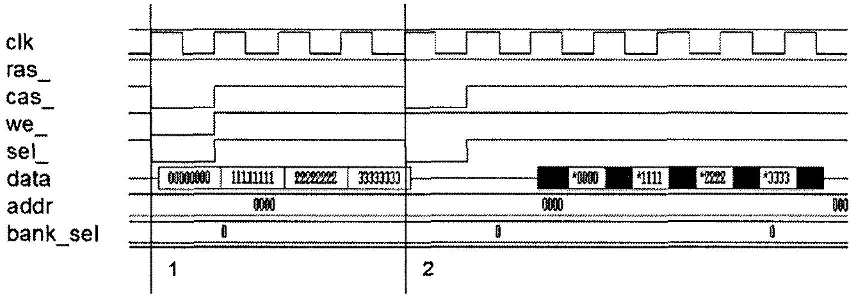

Figure 5-11. SDRAM read/write

- **Read**: This command is issued to do a burst read to an active row. The address provided on the bus "addr" provides the starting column address (sel_, cas_ = 0, ras_ we_ = 1).

- **Write**: This command is issued to initiate a burst write access to an open row. The address on the bus "addr" provides the starting column address (sel_, cas_, we_ = 0, ras_=1). Figure 5-11 shows a simple SDRAM read/write operation. A burst write is performed with a burst size of 4 at marker 1. A burst read to the same address location is done at marker 2.

- **Precharge**: Precharge is used to de-activate the rows (sel_, ras_, we_ = 0, cas_=1). If during precharge the addr[10] bit is set to 1, then all the rows in the banks are de-activated.

- **Auto-refresh**: This command is issued in the normal operation of the SDRAM. This command must be issued every time a refresh is required. All active banks must be precharged prior to issuing an auto-refresh.

- **Burst Terminate**: A burst terminate command is used to terminate a burst read or a burst write command.

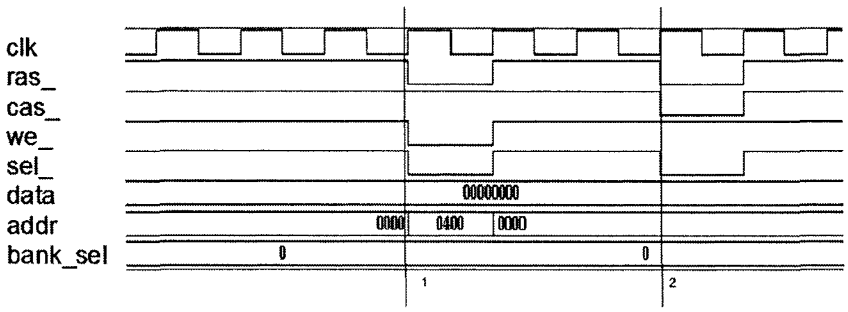

Figure 5-12. Precharge / Auto-refresh

Figure 5-12 shows the precharge command at marker 1 and auto-refresh command at marker 2.

A read/write operation to a SDRAM can be performed once the steps summarized in Figure 5-13 are completed. The description of the steps is as follows:

1. Initialization – once power is applied, SDRAM requires ~100us to initialize before any command can be issued.
2. Once the initialization is completed, one NOP/COMMAND INHIBIT is applied.
3. Then a precharge command is issued and all the rows are de-activated.
4. A Refresh command is issued after precharge.

5. Mode register is loaded (set the “cas” latency, burst size and other configurations).
6. Active command is issued (to activate the rows).
7. Read/Write command is issued.

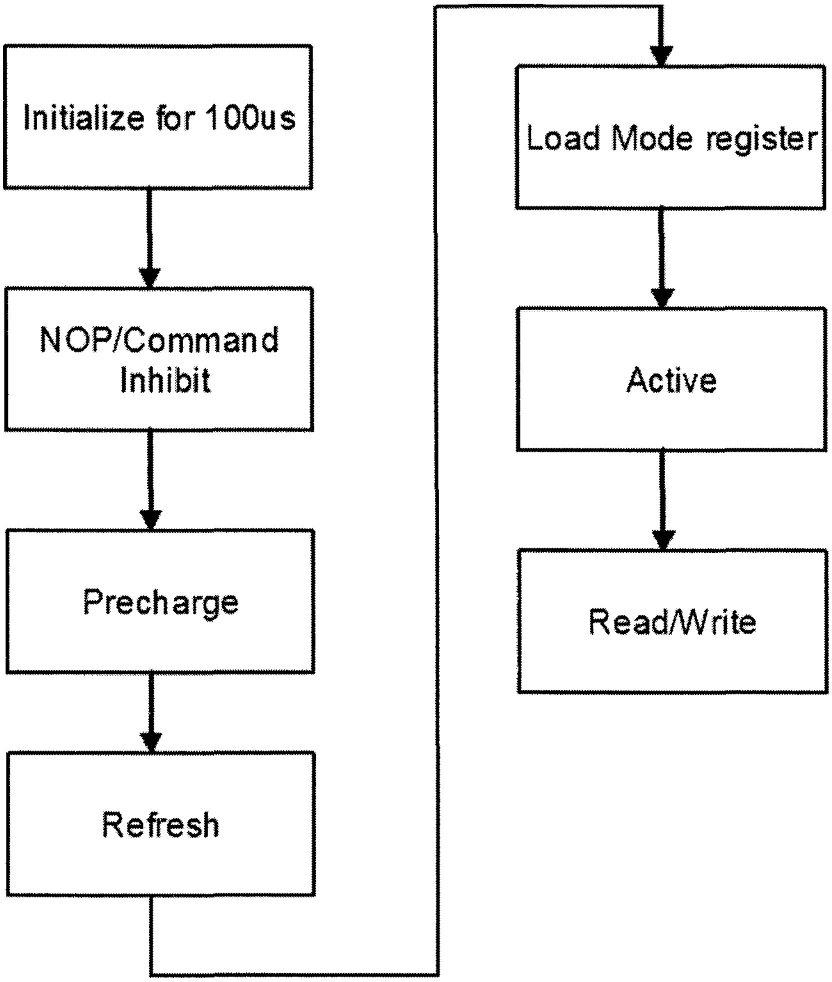

Figure 5-13. SDRAM operation flow chart

5.2.1 SDRAM Assertions

All the SDRAM commands like read, write, burst terminate, active, precharge, load mode register, etc. are derived from the four signals ras_, cas_, sel_ and write enable. Hence, all of these signals should be defined using `defines. These definitions can be re-used in the SVA checkers wherever necessary.

```
`define s_precharge
   (!ras_n && !sel_n[0] && !we_n && cas_n)

`define s_read
  (ras_n  &&  !sel_n[0]  &&  we_n  &&  !cas_n  &&
  (burst == 3'b000))

`define s_burst_read
   (ras_n  &&  !sel_n[0]  &&  we_n  &&  !cas_n  &&
   (burst != 3'b000))

`define s_write
   (ras_n && !sel_n[0] && !we_n && !cas_n)

`define s_autorefresh
   (!ras_n && !cas_n && !sel_n[0] && we_n)

`define s_loadmoderegister
   (!ras_n && !cas_n && !sel_n[0] && !we_n)

`define s_active
   (!ras_n && !sel_n[0] && cas_n && we_n)

`define s_write
    (!cas_n  &&  !we_n  &&  !sel_n[0]  &&  ras_n  &&
    (burst == 3'b000))

`define s_burst_write
   (!cas_n && !we_n && !sel_n[0] && ras_n &&
   (burst != 3'b000))
```

Some of the possible SVA checkers extracted based on the functionality of the SDRAM are shown below. The timing parameters used in these checkers specific to the SDRAM under consideration is listed in Table 5-2. Some of the timing parameters are specified in clock cycles and others in nanoseconds (ns). For the values specified in nanoseconds, the number of clock cycles is dependent on the clock cycle period (tCK). In this sample design the value of tCK is 10ns. Hence, the number of clock cycles is derived based on the value of the clock period and the value of the timing parameter provided in the specification of the SDRAM as shown below.

For example, tRCD=18ns

tRCD/tCK = 1.8 clock cycles

Hence, the timing window between an active command and a read/write command should be at least 2 clock cycles.

Table 5-2. Timing parameters for SDRAM

Parameter	**Symbol**	**Min**	**Max**
Load mode register to active	tMRD	4 cycles	4 cycles
Active to Active Command period	tRC	6 cycles (60ns)	-
Active to Read/Write	tRCD	2 cycles (18ns)	-
Read latency	tCAS	2 cycles	-
Auto Refresh period	tRFC	6 cycles (60ns)	-
Precharge Command period	tRP	2 cycles (18ns)	-
Active Bank a to Active Bank b	tRRD	2 cycles (12ns)	-
Active to Precharge command	tRAS	5 cycles (42ns)	12000 cycles (120000ns)

SDRAM_chk1: Load mode register to active command (tMRD).

The load mode register is used to load the mode register of the SDRAM with information on how the device is configured. Once the SDRAM is configured an active command should arrive in "tMRD" (4 clock cycles). Figure 5-14 shows that the load mode register command is sampled at marker 1 and four clock cycles later active command is sampled (marker 2), as expected. Hence, the check a_tMRD succeeds.

```
property p_tMRD;
     @(posedge clk)
     `s_loadmoderegister |->
                    ##[tMRD] `s_active;
endproperty
```

```
a_tMRD: assert property(p_tMRD);
c_tMRD: cover property(p_tMRD);
```

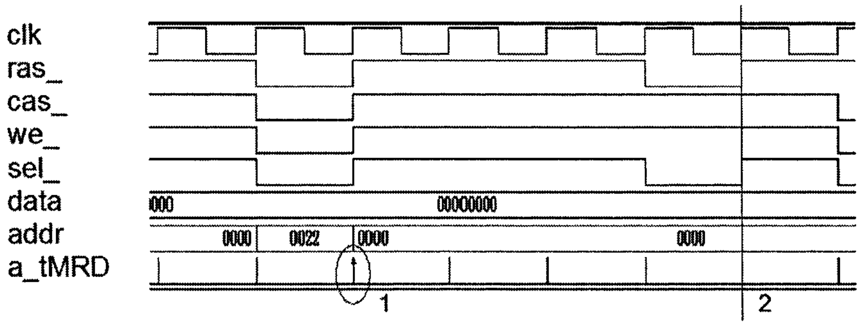

Figure 5-14. Load mode register to Active command, tMRD

SDRAM_chk2: Check the value of load mode register (022).

This assertion is used to check the value written into the mode register. This value is important as it determines the burst size and “cas” latency. When the load mode register command is issued, the value on the address bus is written into the load mode register. Bits [0:2] specify the burst size and the burst size is set to 4 (100). The “cas” latency value is set to 2. To set these parameters, the register has to be written with 0x0022.

Figure 5-14 shows a load mode register command being issued by the memory controller at marker 1. At this point, the address bus has a value of 0x0022. Hence, the check a_loadmoderegister succeeds.

```
property p_loadmoderegister;
        @(posedge clk)
        (`s_loadmoderegister) |->
                     (addr == 16'h0022);
endproperty
a_loadmoderegister:
        assert property(p_loadmoderegister);
c_loadmoderegister:
        cover property(p_loadmoderegister);
```

SDRAM_chk3: tCAS, read data is available with a latency of tCAS after the read command is issued.

In the sample SDRAM memory, whenever a read command is issued, the data is available after the "cas" latency (Column Address Select Latency). This is programmed in the mode register based on the memory vendor. Figure 5-15 shows that a read command is sampled at marker 1. After tCAS cycles, the data is valid as shown by marker 2. To verify this property, implication construct and **$isunknown** construct are used.

```
property p_read;
        @(posedge clk)
         (`s_read || `s_burst_read) |->
                 ##tCAS ($isunknowndata) == 0);
endproperty

a_read: assert property(p_read);
c_read: cover property(p_read);
```

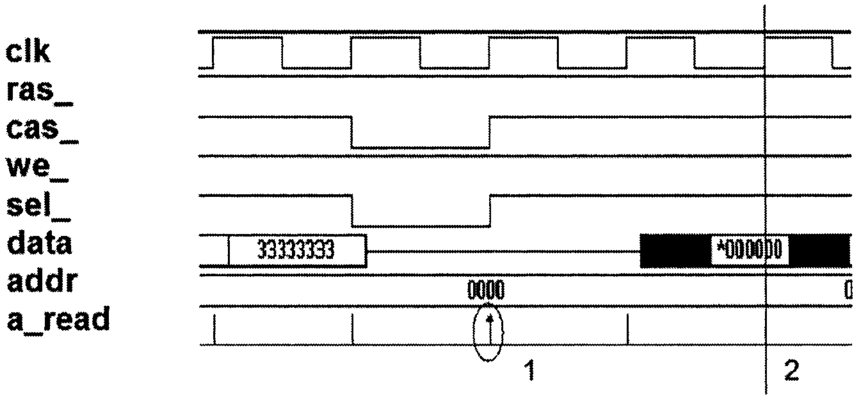

Figure 5-15. SDRAM read with tCAS latency

SDRAM_chk4: tRCD, after an active command, read/write can occur only after tRCD.

If the memory controller has issued an active command, then the read or write command cannot be issued within "tRCD" cycles. In the sample system used, once an active command is issued, a read/write command should be issued within 10 clock cycles. There are two specific conditions that need to be tested:

1. Once the active command is issued a read/write command does not occur within "tRCD" (this is a forbidden property).
2. Once the active command is issued, the read/write command must be issued within 10 clock cycles.

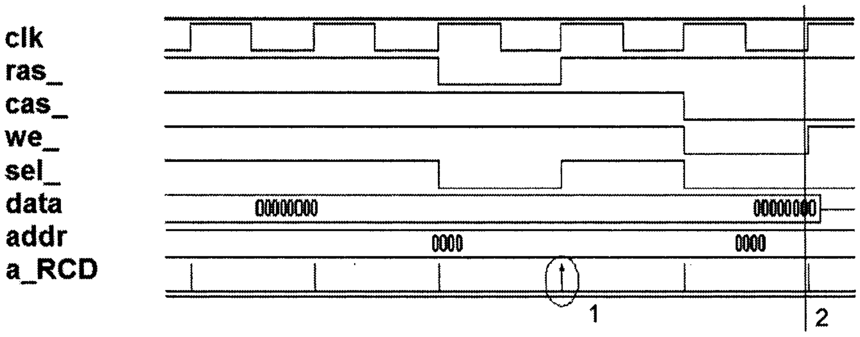

Figure 5-16. Active to Read/Write command, tRCD

```
property p_tRCD_not;
   @(posedge clk)
   `s_active |-> not ##[0: (tRCD - 1)]
        (`s_read || `s_write || `s_burst_read
        || `s_burst_write);
endproperty

property p_tRCD;
   @(posedge clk)
  `s_active |->
          ##[tRCD:10]  (`s_read  ||  `s_write  ||
          `s_burst_read || `s_burst_write);
endproperty

a_tRCD_not: assert property (p_tRCD);
a_tRCD: assert property (p_tRCD);

c_tRCD_not: cover property (p_tRCD);
c_tRCD: cover property (p_tRCD);
```

Figure 5-16 shows that the active command is sampled at marker 1. The write command is sampled 2 cycles after the active command at marker 2. Hence, the check a_tRCD is successful.

SDRAM_chk5: tRC, active to active command cannot come within tRC.

If an active command is issued, the controller cannot issue another active command within "tRC" (6 clock cycles). In the sample system used, if an active command is issued, then the next active command should be issued within 12000 clock cycles.

```
property p_tRC_not;
        @(posedge clk)
         `s_active |->
              not ##[1: (tRC - 1)] `s_active;
endproperty

property p_tRC;
        @(posedge clk)
         `s_active |->
                ##[tRC:12000] `s_active;
endproperty

a_tRC_not: assert property (p_tRC_not);
a_tRC: assert property (p_tRC);
c_tRC_not: cover property (p_tRC_not);
c_tRC: cover property (p_tRC);
```

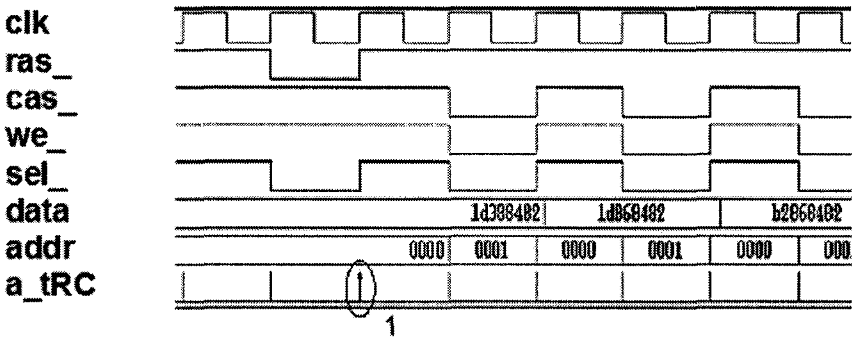

Figure 5-17. Active to Active command, tRC

Figure 5-17 shows the first active command with marker 1. The next active command arrives after 11,625 clock cycles (not shown in the figure) and hence the check a_tRC is successful.

SDRAM_chk6: tRFC, auto-refresh to auto-refresh cannot come within tRFC.

This property is similar to the previous property, in that the window between consecutive auto-refresh commands should be greater than tRFC. In the sample system used, if an auto-refresh command is issued then the next auto-refresh command should be issued within 12000 clock cycles.

```
property p_tRFC_not;
        @(posedge clk)
         `s_autorefresh |->
              not ##[1: (tRFC-1)] `s_autorefresh;
endproperty

property p_tRFC;
        @(posedge clk)
         `s_autorefresh |->
              ##[tRFC:12000] `s_autorefresh;
endproperty

a_tRFC_not: assert property (p_tRFC_not);
a_tRFC: assert property (p_tRFC);

c_tRFC_not: cover property (p_tRFC_not);
c_tRFC: cover property (p_tRFC);
```

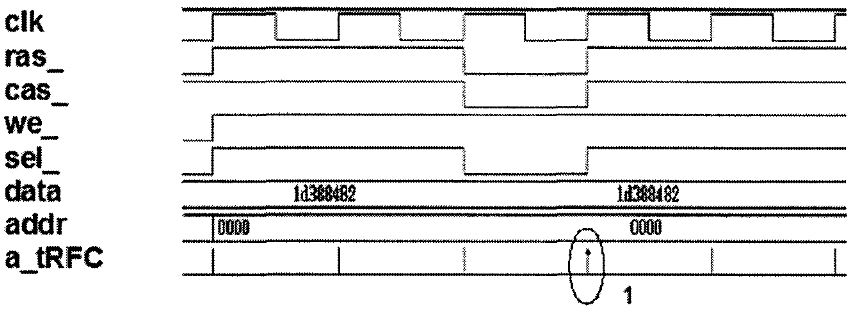

Figure 5-18. Auto-refresh to Auto-refresh command, tRFC

Figure 5-18 shows an auto-refresh command at marker 1. Another auto-refresh command arrives after 9 cycles (not shown in the figure). As this window is greater than "tRFC" as required by the memory specification, the assertion is successful.

SDRAM_chk7: Write command can follow a read command only after tCAS.

A write command cannot follow a read command immediately. By definition, a read command has a "cas" latency of 2 cycles. So the write can follow the read only after the "cas" latency window is satisfied.

```
property p_rd_wr;
        @(posedge clk)
         `s_read |->
               not ##[0:tCAS] `s_write;
endproperty

a_rd_wr: assert property (p_rd_wr);
c_rd_wr: cover property (p_rd_wr);
```

SDRAM_chk8: tRP, precharge to active command cannot be issued until "tRP" is met.

The precharge command (de-activates the rows) to the active command (enables the rows) cannot happen within the "tRP" (2 cycles) window. In the sample system used, if a precharge command is issued, then an active command should be issued within 12000 clock cycles.

```
property p_tRP_not;
        @(posedge clk)
         `s_precharge |->
                 not ##[0:(tRP - 1)] `s_active;
endproperty

property p_tRP;
        @(posedge clk)
         `s_precharge |->
                 ##[tRP:12000] `s_active;
endproperty

a_tRP_not: assert property (p_tRP_not);
a_tRP: assert property (p_tRP);

c_trp_not: assert property (p_tRP_not);
```

```
c_trp: assert property (p_tRP);
```

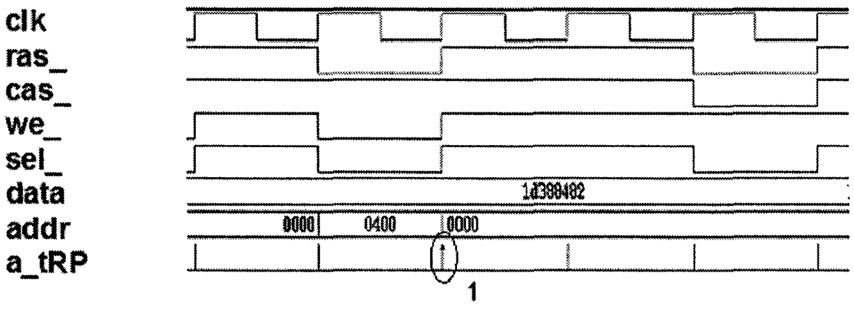

Figure 5-19. Precharge to Active command, tRP

Figure 5-19 shows a precharge occurring at marker 1. An active command is issued within 12000 cycles (not shown in figure) and hence the assertion is successful at marker 1.

SDRAM_chk9: tRAS, active to precharge must occur between tRASmin (5 clock cycles) to tRASmax (12000 clock cycles).

The active command (enables the rows) to the precharge command (de-activates the rows) cannot happen within the "tRASmin" cycles and should happen within "tRASmax" cycles.

```
property p_tRAS_not;
       @(posedge clk)
       `s_active |->
          not ##[0: (tRAS_min - 1)] `s_precharge;
endproperty

property p_Tras;
        @(posedge clk)
         `s_active |->
              ##[tRAS_min:tRAS_max] `s_precharge;
endproperty

a_tRAS_not: assert property (p_tRAS_not);
a_tRAS: assert property (p_tRAS);
c_tRAS: cover property (p_tRAS);
```

```
c_tRAS_not: cover property (p_tRAS_not);
```

SDRAM_chk10: Back to back writes are not allowed.

```
property p_wr_wr;
        @(posedge clk)
         `s_write |->
                     not ##1 `s_write;
endproperty

a_wr_wr: assert property (p_wr_wr);
c_wr_wr: cover property (p_wr_wr);
```

SDRAM_chk11: Check if auto-precharge is disabled during read/write operations.

Most of the SDRAM today can be precharged automatically by setting the addr[10] bit to a high during read/write operations. This assertion is written using implications and logical operation on the command definitions.

```
property p_disable_autoprecharge;
@(posedge clk)
(`s_write || `s_burst_write ||
`s_read || `s_burst_read) |->
                    addr[10] == 0;
endproperty

a_disable_autoprecharge:
       assert property(p_disable_autoprecharge);
```

Figure 5-20 shows the waveform for disabling auto-precharge during read/write commands.

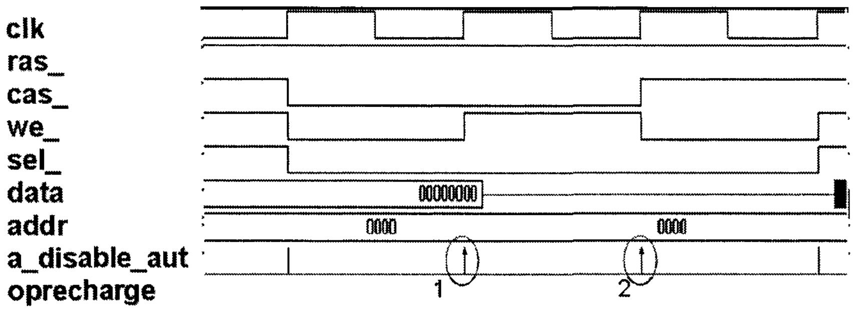

Figure 5-20. Disabling Auto-precharge

A write command is sampled at marker 1 and corresponding to the write, the addr[10] is 0. Similarly, when a read command is in progress (marker 2), the addr[10] is set to 0.

SDRAM_chk12: tRRD, minimum time interval between active commands to different banks is defined by tRRD (2 cycles).

Usually there are multiple banks in the SDRAM. There is a minimum time interval that is required between issuing active commands to different banks. The current system under verification has four banks.

```
property p_tRRD;
     @(posedge clk)
     (`s_active && bank_addr[1:0] == 0) |->
                not ##[0: tRRD]  (`s_active  &&
                bank_addr[1:0] != 0));
endproperty

a_tRRD: assert property(p_tRRD);
c_tRRD: cover property (p_tRRD);
```

This check verifies that if an active command is issued to bank 0, then an active command cannot be issued to other banks (1, 2, 3) within "tRRD." The same check has to be repeated for banks 1, 2 and 3 respectively. This can be done easily with a generate statement and a "for" loop as shown below.

```
genvar j;
generate
```

```
for (j=0; j<4; j++)
begin:loop
a_generate: assert property(@(posedge clk)
  (`s_active && bank_addr[1:0] == j)
  |-> not ##[1: tRRD] (`s_active &&
                 (bank_addr[1:0] != j)));
c_generate: cover property(@(posedge clk)
 (`s_active && bank_addr[1:0] == j)
   |-> not ##[1: tRRD] (`s_active &&
                 (bank_addr[1:0] != j)));
end
endgenerate
```

SDRAM_chk13: If "data_size" is 128, then check the mask operation.

The CPU-AHB bus can define the data size and write to the memory in 128 bits, 64 bits or 32 bits. The most commonly used data size is 32 bits. But when 128/64 bits are used, the mask bits are used to write the data in 32-bit chunks to the same address.

```
property p_xfer128;
@(posedge clk)
((size == 0) && ((dqm[0] == 0 && (`s_write
||`s_burst_write))) |->
  ##2 ($fell (dqm[1]) && addr == $past (addr, 2)
&&
   (`s_write ||`s_burst_write))
  ##1 $rose (dqm[1])
  ##1 ($fell (dqm[2]) && addr == $past (addr, 2)
&&   (`s_write ||`s_burst_write))
  ##1 $rose (dqm[2])
  ##1 ($fell (dqm[3]) && addr == $past (addr, 2)
 &&  (`s_write ||`s_burst_write))
  ##1 $rose (dqm[3]));
endproperty

a_xfer128: assert property(p_xfer128);
c_xfer128: cover property(p_xfer128);
```

Figure 5-21 shows the 128-bit data transfer. Data is written in 4 chunks of 32 bits to the same address location. Marker 1 shows the first 32 bits of data being written to address 0x0021 and marker 2 shows the fourth chunk of 32 bits of data being written to address 0x0021. Mask Bits dqm[3:0] are used to control the data that is being written to the memory.

- When the data transfer size is 32, all the bits of the vector dqm[3:0] are set to 0.
- If the data transfer size is 64, data is transferred in two chunks of 32 bits. When the first chunk of 32-bit data is transferred, dqm[1:0] is set to 0. When the second chunk of 32-bit data is transferred, dqm[3:2] is set to 0.
- For 128-bit transfers, when dqm[0] is set to 0, the first 32 bits of data is written to the memory and when dqm[1] is asserted, the second 32 bits of data is written to the memory. Similarly, when dqm[2] and dqm[3] are set to 0, the third and fourth chunks of 32 bits of data are written to the memory respectively.

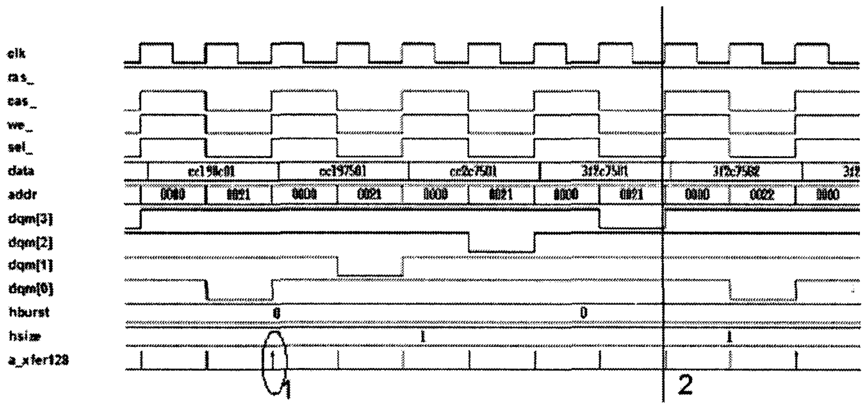

Figure 5-21. 128-bit data transfer

SDRAM_chk14: If "data_size" is 64, each read/write operation takes 2 cycles.

This property is similar to the previous one. The data is written in two chunks of 32 bits. When dqm[0] == 0 and dqm[1] == 0, the first chunk of 32 bits of data is written. When dqm[2] == 0 and dqm[3] == 0, the second chunk of 32 bits of data is written.

Figure 5-22 shows the 64-bit data transfer. The first 32 bits of data is written to address 0x0103 and mask signals 0 and 1 are set to 0 (marker 1).

The second chunk of data is written to the same address 0x0103 and mask signals 2 and 3 are set to 0 (marker 2).

```
property p_xfer64;
@(posedge clk) ((size == 1) && ((dqm[1:0] == 0
&&(`s_write ||`s_burst_write))) |->
##2 ($fell (dqm[2] &&  dqm[3]) && addr == $past
(addr, 2) && (`s_write ||`s_burst_write))
##1 $rose (dqm[3] && dqm[2]));
Endproperty

a_xfer64: assert property(p_xfer64);
c_xfer64: cover  property(p_xfer64);
```

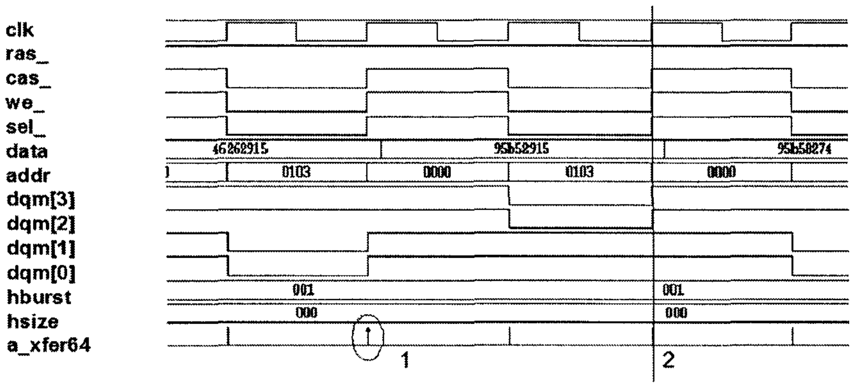

Figure 5-22. 64-bit data transfer

SDRAM_chk15: Read/write terminated by a burst terminate.

A burst terminate command is used to terminate a burst read/write command. So, if a burst terminate command is issued, the previous cycle must be a burst read/write operation.

```
property p_wr_rd_bursttermіnate;
@(posedge clk) (s_burstterminate) |->
$past ((`s_burst_write || `s_write || `s_read ||
`s_burst_read), 1);
endproperty
```

```
p_wr_rd_burstterminate:
        assert property(p_wr_rd_burstterminate);
c_wr_rd_burstterminate:
        cover property(p_wr_rd_burstterminate);
```

SDRAM_cover_chk1: Write terminated by a burst terminate.

There are some scenarios and properties that should be covered as part of the verification. For example, in the property (p_wr_rd_burstterminate), if a "burst terminate" command is issued, the previous command should be a "burst write" or a "burst read" command. But in the result of the check there is no classification on which specific command (read/write) was terminated by the "burst terminate" command, since all possible legal conditions have been combined with the logical OR operator. In order to obtain this kind of scenario information, the property is split and cover statements are written.

A separate property is written to check if the "burst write" was terminated using burst terminate. If this property is asserted, there might be failures because "burst terminate" command can be issued for terminating "burst read" also. Hence, for collecting coverage information on scenarios, there is no need to declare assert statements.

```
property p_wr_burstterminate;
@(posedge clk)
(s_burstterminate) |->
        $past ((`s_burst_write || `s_write), 1);
endproperty

c_wr_burstterminate:
          cover property(p_wr_burstterminate);
```

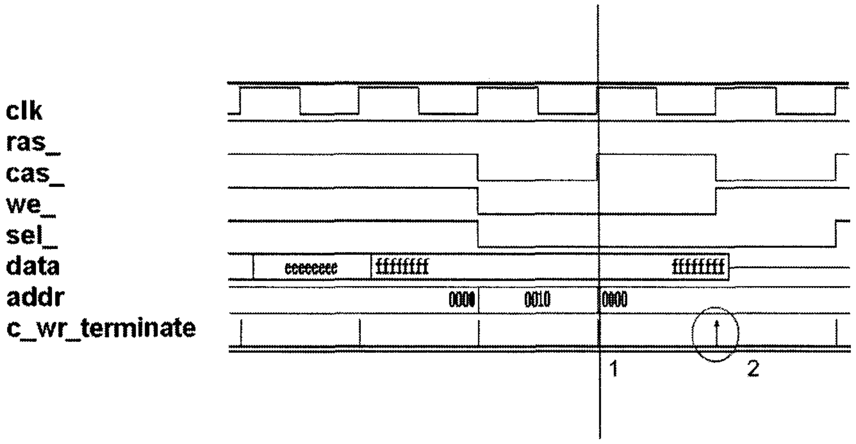

Figure 5-23. Burst write to Burst terminate command

Figure 5-23 shows a write command sampled at marker 1 and a burst terminate command sampled at marker 2. The cover statement is successful since the previous cycle of the burst terminate was a burst write command.

SDRAM_cover_chk2: Read terminated by a burst terminate.

The read command can be terminated by the "burst terminate" command similar to the previous check. This assertion checks that if in the current cycle a burst terminate command is issued than the previous cycles is a read/burst read.

Figure 5-24 shows a read command (marker 1) terminated by the "burst terminate" command (marker 2). But since there is "cas" latency to the read command, the data for the terminated read will be available two cycles later from when the read command was issued, as shown by marker 3. Until the data is available, no other command can be issued.

```
property p_rd_burstterminate;
        @(posedge clk) (s_burstterminate) |->
           $past  ((`s_burst_read  ||  `s_read),
        1);
endproperty

c_rd_burstterminate:
           cover property(p_rd_burstterminate);
```

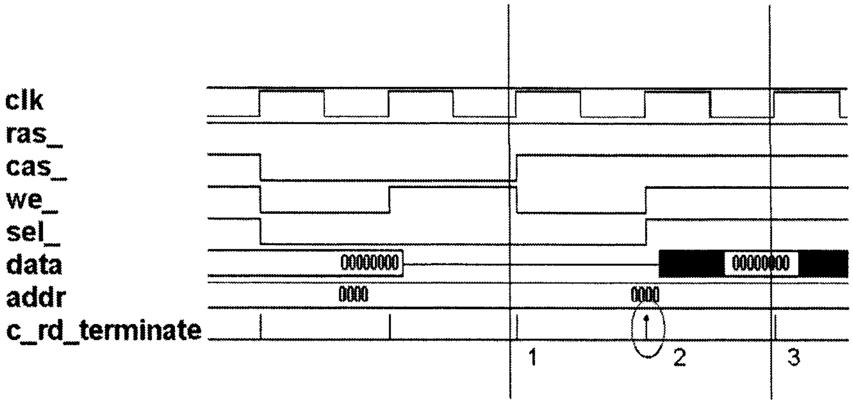

Figure 5-24. Read to Burst terminate

SDRAM_cover_chk3: Write terminated by a read.

The write command can be terminated by a read command. If a write command is in progress and a read command is issued, the write is immediately aborted. Once again, there is no need to assert this property. The write command can either be terminated by other ways or can be followed by any other command. Hence, asserting this property might produce unnecessary failures.

Figure 5-25 shows that a write command (marker 1) is being terminated by the read command (marker 2) and the read command is terminated by the burst terminate command. The burst size in this example is 4 and both the read/write operations are being terminated after just one write/read transfer.

```
property p_wr_rdterminate;
 @(posedge clk) (`s_write ||
  `s_burst_write) ##1 (`s_read || `s_burst_read);
endproperty

c_wr_rdterminate :
          cover property(p_wr_rdterminate);
```

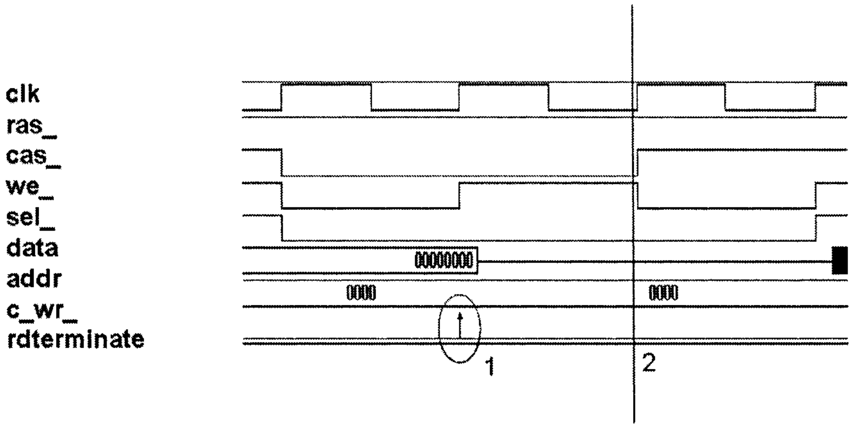

Figure 5-25. Write terminated by a Read command

5.3 SRAM/FLASH Verification

- SRAM (static RAM) is a type of memory that holds data without external refresh as long as it is powered.
- SRAMs are a lot faster than SDRAMs.
- SRAMs are expensive and take more space/area.

The verification of an SRAM/FLASH is very simple as there are no complex refresh mechanisms. When the "write" and "chip select" signals are asserted, the data is written in the memory starting from the location specified in the address bus. Similarly, when chip select is asserted, write enable is de-asserted and output enable is asserted, the data is read out of the memory. In the sample system, data cannot be written to the flash, since write protect signal is always asserted. The sample system uses these static memories.

SRAM : 256K X 16bit high speed Static RAM

FLASH: 128Mbit flash (16Mbytes)

The timing parameters of the SRAM used in the sample system are shown in Table 5-3. The timing parameters of the Flash used in the sample system are shown in Table 5-4.

Table 5-3. Timing parameters for SRAM

Parameter	Symbol	Min	Max
Write Cycle Time	tWC	1 cycle (10ns)	-
Write Pulse Width	tWP	2 cycle (20ns)	-
Read Cycle Time	tRC	1 cycle (10ns)	-
Chip Select to output	tCO	1 cycle (10ns)	-
Address Access time	tAA	1 cycle (10ns)	-

Table 5-4. Timing parameters for Flash memory

Parameter	Symbol	Min	Max
Read/Write Cycle time	tAVAV	15 cycles (150ns)	-
Chip select to Output Delay	tELQV	-	15 cycles (150ns)
Page Address Access time	tAPA	-	3 cycles (25ns)

5.3.1 SRAM/FLASH Assertions

SRAM_chk1: Write cycle time, tWC.

The SRAM write cycle time should be greater than the "tWC" mentioned in the specification. The write cycle time is the time in which the address is stable and in which the chip select and write enable signals are asserted.

To implement this assertion, the **$stable** system function and the implication operator are used. The **$stable** function makes sure that the value of the address in the current clock cycle is the same as the previous cycle.

```
property p_tWC;
@(posedge clk)
($fell (we_n) && !sel_n[2]) |=>
              $stable(addr[22:0]);
endproperty
a_tWC: assert property(p_tWC);
c_tWC: cover property(p_tWC);
```

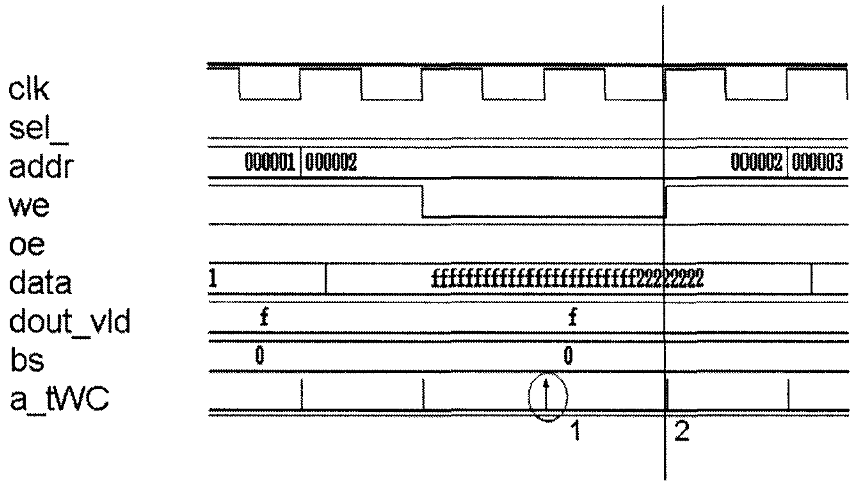

Figure 5-26. Write wycle time, tWC

Figure 5-26 shows that a write command is sampled at marker 1 and the assertion succeeds at marker 2, since the address is stable for at least one clock cycle from when the write was issued.

SRAM_chk2: Write enable pulse width, tWP.

This check verifies that the write pulse width is always greater than the minimum specified in the specification (2 cycles). Figure 5-27 shows that the falling edge of write (marker 1) and the rising edge of write (marker 2) are sampled 2 cycles apart.

```
property p_tWP;
@(posedge clk)
$fell (we_n) |->
          ##tWP $rose (we_n);
endproperty

a_tWP: assert property(p_tWP);
c_tWP: cover property(p_tWP);
```

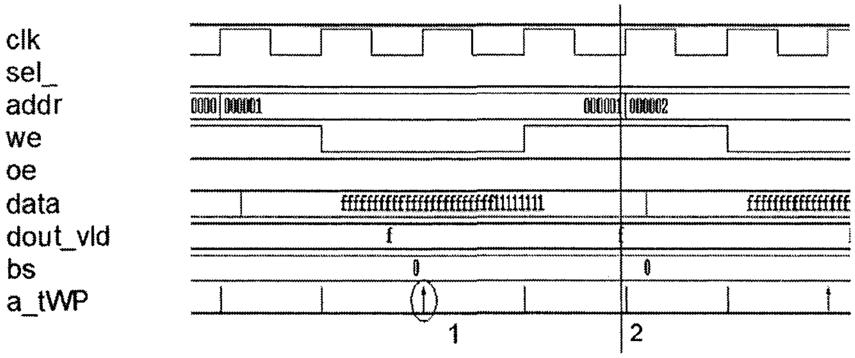

Figure 5-27. Write pulse width, tWP

SRAM_chk3: tRC - read cycle time.

This is similar to the write cycle time check. The read cycle time is the time in which the address is stable and in which the chip select and output enable are asserted. Figure 5-28 shows that chip select and output enable are asserted at marker 1. The address value is the same at both marker 1 and marker 2.

```
property p_tRC;
@(posedge clk)
(!sel_n[2] && we_n && !oe_n) |=>
                           ($stable (addr));
endproperty

a_tRC: assert property(p_tRC);
c_tRC: cover property(p_tRC);
```

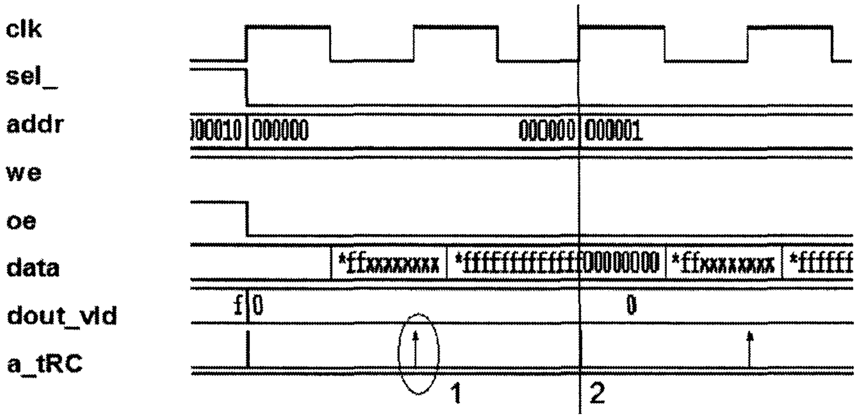

Figure 5-28. Read cycle time, tRC

SRAM_chk4: tCO - chip select to output data valid.

The parameter "tCO" is the minimum time that chip select has to be asserted before data becomes valid. Figure 5-29 shows that the chip select and output enable are asserted at marker 1. In the same clock cycle, the data value is "x." One cycle later, the data value is valid (marker 2).

```
property p_tCO;
@(posedge clk)
(!sel_n[2] && we_n && !oe_n &&
($isunknown (data))) |=>

        ($isunknown (data))==0;
endproperty

a_tCO: assert property(p_tCO);
c_tCO: cover property(p_tCO);
```

SRAM_chk5: tAA - Valid address to valid data.

The parameter "tAA" is the minimum time for which address has to be valid before data becomes valid. Figure 5-30 shows that a read command is sampled at marker 1. The address in this clock cycle should be stable in the next clock cycle and the data should be valid as shown by marker 2.

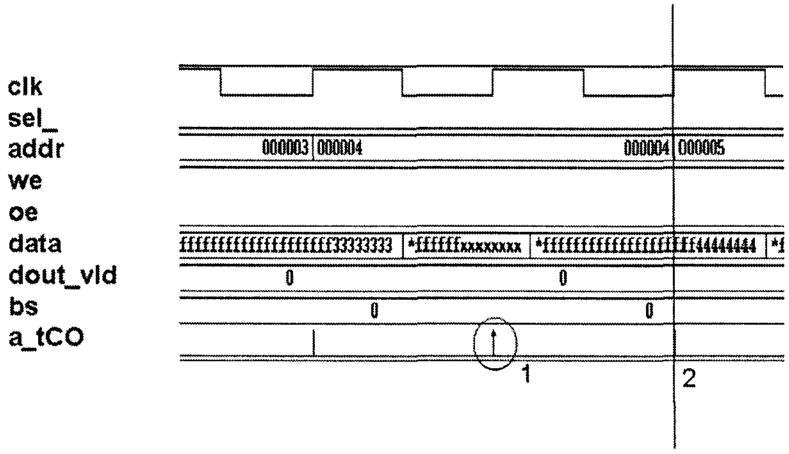

Figure 5-29. Chip Select to valid data, tCO

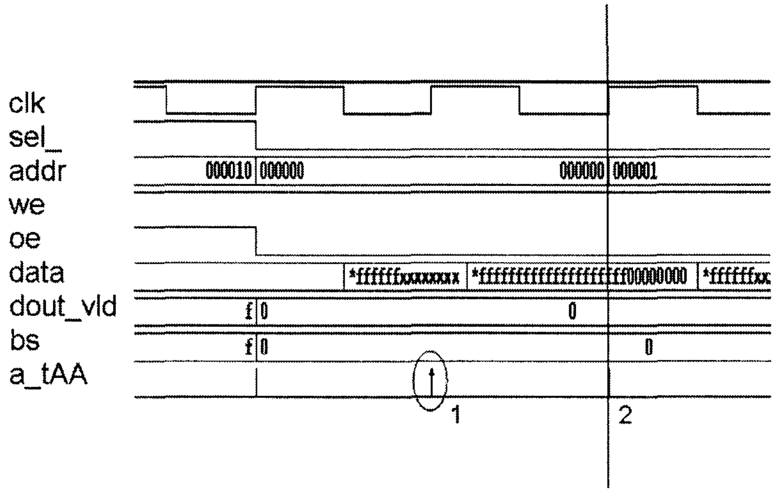

Figure 5-30. Valid address to Valid data, tAA

```
property p_tAA;
@(posedge clk) (!sel_n[2] && we_n && !oe_n) |=>
           ((addr == $past (addr,1))  ##0
           ($isunknown (data))==0);
endproperty
```

```
a_tAA: assert property(p_tAA);
c_tAA: cover property(p_tAA);
```

FLASH_Chk1: Flash is write protected.

The flash memory used in the sample system is write protected. It is necessary to make sure that the write protect signal (wp_) is always asserted when the chip select for flash is enabled.

```
property p_write_protect;
 @(posedge clk)
(!sel_n[3]) |->
            wp_n == 0;
endproperty

a_write_protect:
       assert property (p_write_protect);
c_write_protect:
       cover property (p_write_protect);
```

FLASH_chk2: Complete read cycle time (tAVAV).

The minimum read cycle time as mentioned in the Table 5-4 is tAVAV(15 clock cycles). In the sample system used, the read cycle time cannot be more than 900 clock cycles. Hence, two checks are written to verify both the minimum and maximum timing requirements.

```
property p_tAVAV_not;
 @(posedge clk)
(!sel_n[3] && $fall(oe_n)) |->
                    not ##[0:15] $rose (oe_n);
endproperty

property p_tAVAV;
 @(posedge clk)
 (!sel_n[3] && $fell (oe_n))   |->
                   ##[16:900] $rose (oe_n);
endproperty

a_tAVAV: assert property(p_tAVAV);
a_tAVAV_not: assert property(p_tAVAV_not);

c_tAVAV: cover property(p_tAVAV);
```

```
c_tAVAV_ not: cover property(p_tAVAV_not);
```

FLASH_chk3: CS/ADDR to valid data is tELQV.

The minimum time that the chip select and address should be stable before data is valid is specified by "tELQV" (15 clock cycles). Figure 5-31 shows that the signal "sel" is asserted at marker 1. After 15 cycles, the first data is valid as denoted by marker 2.

```
property p_tELQV;
@(posedge clk)
(!oe_n && $fell (sel_n[3])) |->
##14 $isunknown(data) ##1 ($isunknown (data)==0);
endproperty

a_tELQV: assert property(p_tELQV);
c_tELQV: cover property(p_tELQV);
```

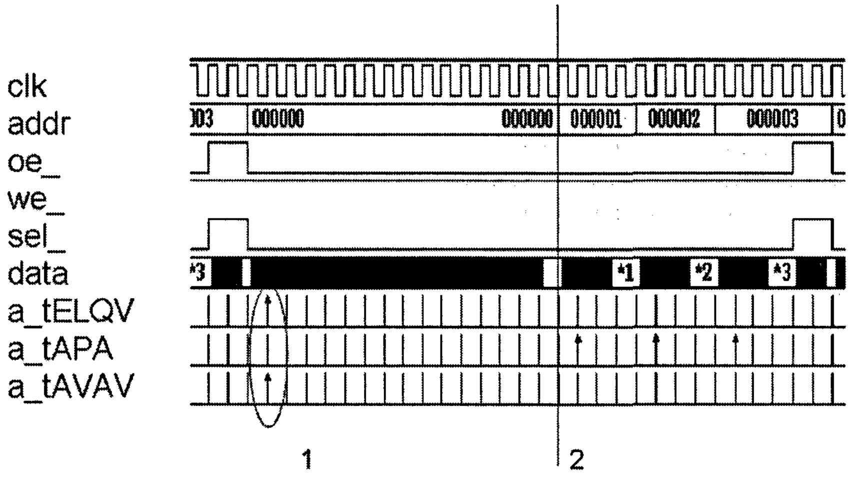

Figure 5-31. Flash waveform for tELQV, tAPA, tAVAV

FLASH_chk4: ADDR to valid data, tAPA.

The parameter tAPA (3 cycles) is the minimum time that the address is required to be stable before data is valid. This is true for only the subsequent reads of the burst read and not the first read in a burst.

Figure 5-32 shows a burst read command. In a burst read, when the address changes, a new data should be read within tAPA. The change in address from "000000" to "000001" is sampled at marker 1. At this point, the data is unknown and 3 clock cycles later a valid data is sampled, as shown by marker 2.

```
sequence s_data_trans;
(!sel_n[3] && !oe_n && ($stable (addr)==0) &&
$stable (oe_n)) ##0 $isunknown (data)
##3 $isunknown (data)==0;
endsequence

property p_tAPA;
@(posedge clk)
s_data_trans |->
          $stable(addr);
endproperty

a_tAPA: assert property(p_tAPA);
c_tAPA: cover property(p_tAPA);
```

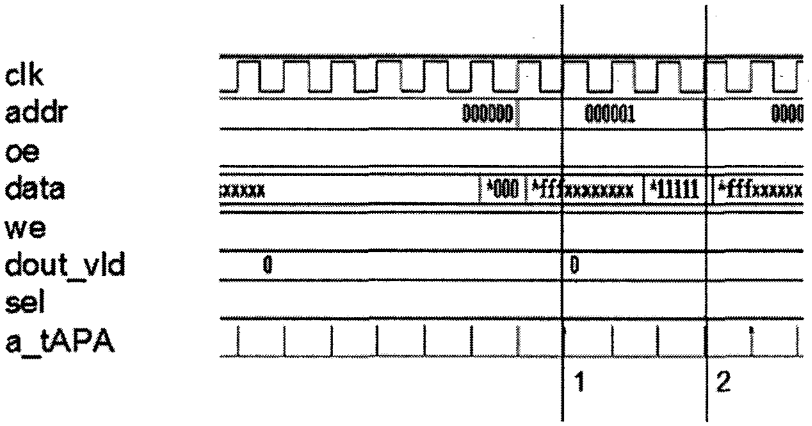

Figure 5-32. Flash waveform for tAPA

5.4 DDR-SDRAM Verification

Double Data Rate Synchronous Dynamic Random Access Memory (DDR-SDRAM) is a type of memory that is similar to Synchronous DRAM but has a higher bandwidth. Data is written and read at both the rising and falling edge of the clock, doubling the speed. The operations are similar to that of the SDRAM. The DDR-SDRAM used in the sample system has the following configuration.

DDR-SDRAM: 4Mword x 16bit x 4bank

The read and burst read operation in the DDR-SDRAM is the same as that of the SDRAM. The burst read command is issued by asserting "sel_" and "cas_" while holding "ras_" and "we_" high. The address inputs determine the starting address of the burst. The first data is available after the "cas" latency after the read command (which is 2 cycles, based on the DDR-SDRAM specifications), and the subsequent data are presented on the rising and falling edges of the signal "dqs" (data strobe).

The burst write command is issued by asserting "sel_," "cas_," "we_" and de-asserting "ras_" on the rising edge of the clock (clk). There is a latency of 1 clock cycle for the signal "dqs" to arrive. There is no latency relative to the signal "dqs" for a write command.

5.4.1 DDR-SDRAM Assertions

DDR_Chk1: Burst Read operation for DDR memories.

In the DDR memory there are multiple clocks. Data transfer and read are done on clock "clk2x" which samples data on both edges. Most of the checks written for SDRAM can be reused for a DDR-SDRAM. New checks have to be written wherever the control signals are crossing clock domains.

The keyword **matched** is used to synchronize signals across multiple clock domains in SVA. In this assertion, the signals cas_, ras_, we_ and sel_ are being generated by clock "clk." The data is read at the negative edge of clock "clk2x." In this case, we have to use the **matched** construct to synchronize the read sequence from one clock domain to another.

```
sequence s_read;
     @(posedge clk)
     (ras_n && !sel_n[0] && we_n && !cas_n);
```

```
endsequence

property p_read;
@(negedge clk2x) s_read.matched |->
##3 ($isunknown (data))
##1 ($isunknown (data) == 0);
endproperty

a_read: assert property(p_read);
c_read: cover property(p_read);
```

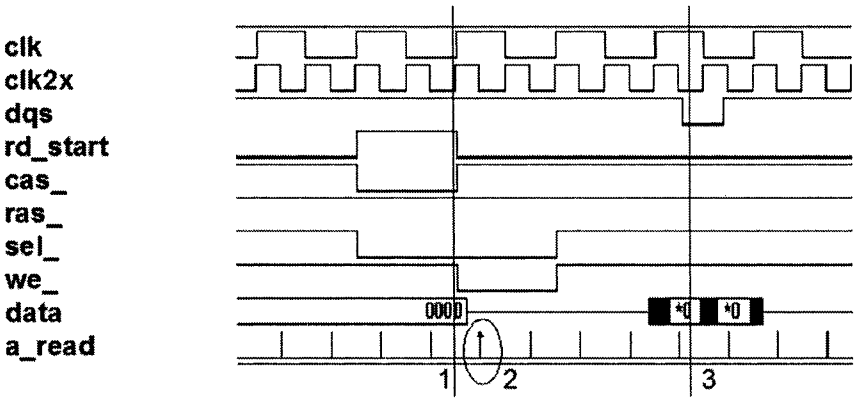

Figure 5-33. DDR-SDRAM Burst read operation

Figure 5-33 shows that a read command is sampled at marker 1 (s_read) based on the clock "clk." The matched value of this sequence is sampled in the next nearest negative edge of clock "clk2x," as shown by marker 2. Data is then read out with a CAS latency of 4 clock cycles (clk2x) denoted by marker 3. A valid data is read on both edges of the signal "dqs" and the signal "dqs" is generated based on clock (clk2x). Hence, the negative edge of the clock (clk2x) is used to sample the data.

DDR_Chk2: Burst write operation on DDR memories.

```
sequence s_write;
 @(posedge clk)
   (ras_n && !sel_n[0] && !we_n && !cas_n);
endsequence
```

```
property p_write;
  @(posedge clk2x) s_write.matched
      |->   ##1 ($isunknown(data) == 0)
            ##1 ($isunknown(data) == 0);
endproperty

a_write: assert property(p_write);
c_write: cover property(p_write);
```

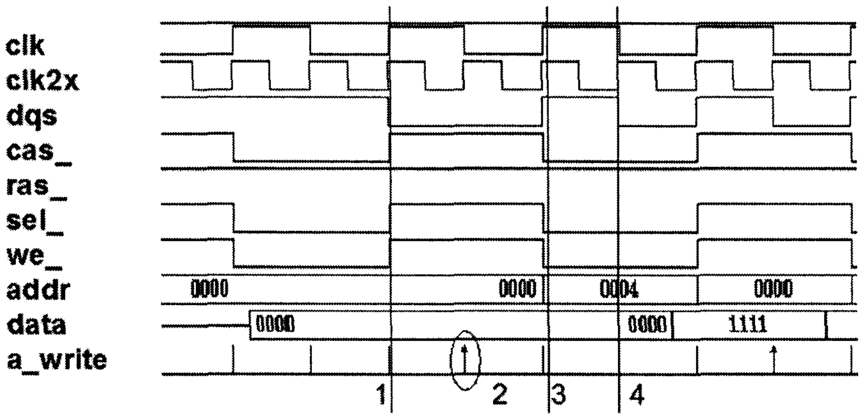

Figure 5-34. DDR-SDRAM Burst write operation

Figure 5-34 shows a write command at marker 1 (based on clock (clk)). The write command is synchronized to the positive edge of clk2x at marker 2. The positive edge of the clock (clk2x) is used for sampling in a write command because the signal "dqs" is generated based on the clock (clk2x). The assertion is successful at marker 2 since data is being written into the memory on both edges of the signal "dqs" (data strobe) as shown by marker 3 and marker 4.

5.5 Summary on SVA for Memories

- Assertions can be used effectively to verify the timing requirements of memory devices.
- All timing information relevant to the memory device should be parameterized. This way, the assertions developed for a particular type of memory can be reused with similar memory device from any other vendor.

- The assertions written for memories provide information on specific **scenario coverage**. For example, was a write terminated by a read/burst terminate, was a read terminated by a burst terminate, was a back to back write performed, did a write command follow a read command, did the tests cover different data widths - 128/64/32 bits, etc. This helps increase the verification confidence and also provides a measure for verification completeness.