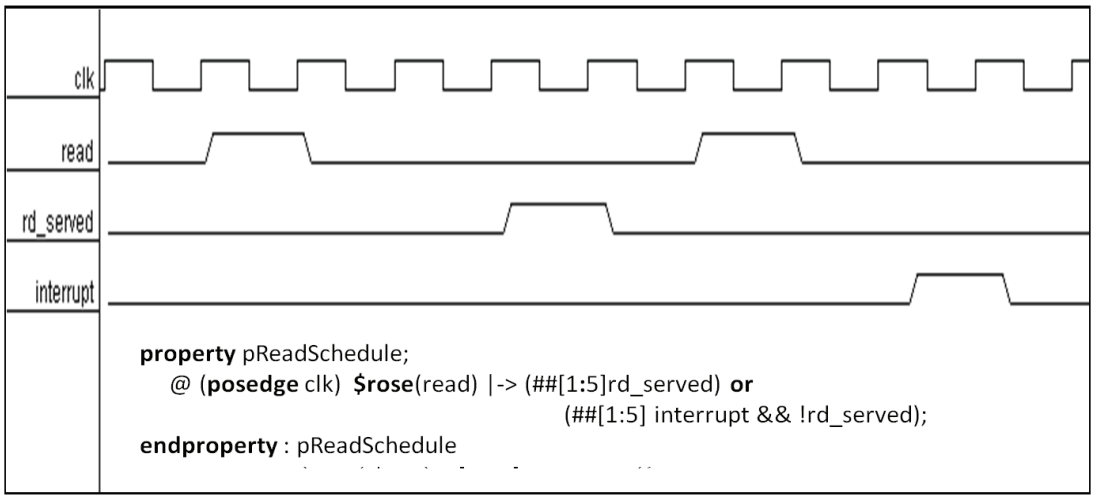


An alternative to the ORing of the properties would be to use the **sync_accept_on.**

```
property pReadSchedule2;
   @ (posedge clk)  sync_accept_on (interrupt)
         $rose(read) |-> (##[1:5] rd_served);
endproperty : pReadSchedule2
```

Assertion suceeds vacuously if *interrupt==1*

```
property pWriteIntrpt ;  // Property passes if interrupt  or wr_served
  @ (posedge clk) sync_accept_on (interrupt)
   ($rose(write)) |-> (##[1:5] (wr_served ));
endproperty : pWriteIntrpt
```

Assertion suceeds vacuously if *interrupt==1*

You can also use the **disable iff** clause to cause the property to be cancelled when an *interrupt* occurs. However, in this case, since an interrupt is a normal operating condition, this may not be desired.

```
property pReadSchedule3;  // Property cancelled  if interrupt, passed if rd_serve
   @ (posedge clk)  disable iff  (interrupt)
         $rose(read) |-> (##[1:5]rd_served);
 endproperty : pReadSchedule3
```

Assertion cancelled if *interrupt==1*

```
property pWriteIntrpt2;  // Property cancelled  if interrupt, passed if wr_served
  @ (posedge clk) disable iff  (interrupt)
   ($rose(write)) |-> (##[1:5] (wr_served ));
endproperty : pWriteIntrpt2
```

Assertion cancelled if *interrupt==1*

The following coverage provides information about the occurrences of sequences (see 4.5.1.4 for more information on the statistics obtained with the cover statement).

```
cq_read_no_intrpt: cover sequence ($rose(read) ##[1:5]rd_served);
cq_read_intrpt: cover sequence ($rose(read) ##[1:5] interrupt && !rd_served);
cq_write_no_intrpt: cover sequence ($rose(write) ##[1:5]wr_served);
cq_write_intrpt: cover sequence ($rose(write) ##[1:5] interrupt && !wr_served);
```

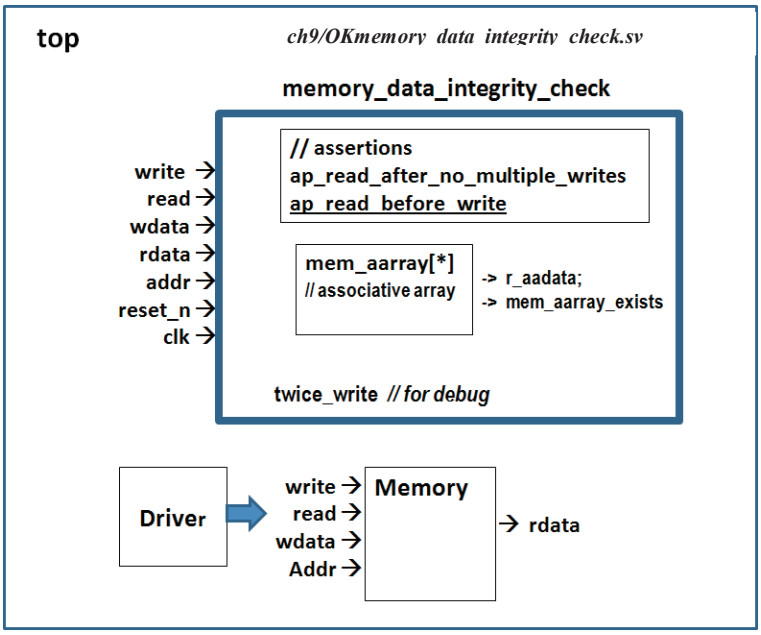
10.15 Data Integrity in memory: data read from memory should be same as what was last written

Given a large memory (or a port), the following properties must be verified:
* Data should never be read before it is first written; thus only valid data must be read.
* Data read from the memory is what was last written into it.

Data integrity can easily be checked using a scoreboard that emulates the behavior of the DUT's memory, and then compare the read data results of both memories. However, because the DUT memory is very large, using a memory for the scoreboard can be expensive in terms of resources

used by the simulator. In this solution, an associative array is used to maintain the scoreboarding because it is more efficient. This model also brings up some interesting issues in the construction of properties.

File *ch10/memory_data_integrity_check.sv* provides the complete model and the testbench. The figure below represents the architecture of the design, verification model, and the testbench.




```
module memory_data_integrity_check (   // /ch10/10.15/memory_data_integrity_check.sv
    input bit write, // memory write
    input bit read,  // memory read
    input bit[31:0] wdata, // data written to memory
    input bit [31:0] rdata,  // data read from memory, next cycle as read
    input bit[31:0]  addr,  // memory address -- small for simulation
    input bit reset_n, // active low reset
    input bit clk);    // clock

    timeunit 1ns;   timeprecision 100ps;
    default clocking cb_clk @ (posedge clk);  endclocking
    int mem_aarray[*]; // associative array (AA), stores address
    bit [31:0] r_aadata, r_aadata_dly;  // data read from memory
    bit  mem_aarray_exists;  // exists at specified address

    assign mem_aarray_exists  = mem_aarray.exists(addr);
    always_comb
        if(mem_aarray_exists)
            r_aadata  = mem_aarray[addr];  //  debug

    always@ (posedge clk)begin
       if (reset_n==1'b0)    mem_aarray.delete; // Clears AA elements
       else if (write)       mem_aarray[addr]  = wdata; // store data
       r_aadata_dly <= r_aadata;
    end
```

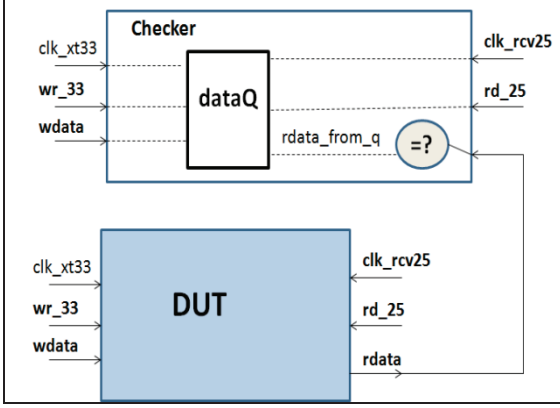
scoreboard, supports assertions

```
    property p_read_after_writes;
        (read && mem_aarray_exists) |=>
           rdata==r_aadata_dly;
    endproperty : p_read_after_writes
    ap_read_after_writes : assert property (p_read_after_writes);

    // never a read on an non-written address
    ap_read_before_write : assert property (not (read && !mem_aarray_exists));
endmodule : memory_data_integrity_check
```

10.16 Data integrity in queues. interface data written must be properly transferred to the receiving hardware

The data received from an interface (with `wr_33` control) must be properly transferred to the receiving hardware (with `rd_25` control). The data is sourced at a 33 MHz rate and is extracted at a 25 MHz rate. The data extracted by the receiver (`rdata`) is in the same order that it was transmitted. The following figure shows a block diagram of the verification environment and the timing diagram for the interface.



1. DUT has internal asynchronous FIFO to provide this data transfer.
2. Checker uses a SystemVerilog queue instead.
3. Checkers verifies for illegal conditions, such as a *read* with no data in the queue or *write* on a full queue.
4. Checker assumes a queue size of 1024
5. checker verifies that data read from the queue (i.e., the expected result) is same as output data from DUT.

For this problem, it is important to verify that the data inserted into the FIFO at the 33 MHz rate is correctly read from the memory at the 25 MHz rate. It is also important to verify that there is no data overrun on the write of data (i.e., the FIFO data exchange does not exceed the size of the FIFO – capacity exceeded, more data written than read).

Key notes about this model are addressed below. The simplest solution to express this verification is to create a verification module or checker, which can be instantiated or bound to a verification module. In this model, a SystemVerilog queue is used to store data at the 33 MHz rate upon a `wr_33` signal. The declaration **`int`** `dataQ [`**`$`**`]` declares an unbounded queue. Data is stored into the queue with the **`push_front`** method. A `rd_25` signal, synchronous to the 25 MHz clock, causes data to be popped into a variable `rdata_from_q` from the queue, using the **`pop_back`** method. In this model, it is understood that the data is extracted by the receiver (`rdata`) in the cycle following the `rd_25` control. With the use of the queue, and queue management code, the verification logic needs not be concerned with the synchronization between the two clocks. That simplifies the definition of the verification properties. In fact, one property is needed to verify the data integrity: