Chapter 4

SVA FOR DATA INTENSIVE DESIGNS

In any design, there are two areas that need to be verified thoroughly:

a. Is the control logic behaving correctly? – These signals control the flow of data in the design and have complex timing relationships between each other.
b. Is my output data as expected? – This makes sure that the output data of the RTL matches the output of the golden model (usually written in C). This guarantees that the functionality of the optimized hardware algorithms implemented in RTL matches that of the golden model.

In general, assertion based verification is very suited for checking signals that have complex timing relationships or in other words, the control logic. The declarative nature of the language makes it more suitable for temporal checking. While assertions don't add any additional value for data checking, it can still be used for writing efficient self-checking environments.

4.1 A simple multiplier check

SystemVerilog assertions have the advantage of using most data types and operators that are part of the SystemVerilog language. This gives great flexibility in writing simple arithmetic checks.

Example 4.1 A simple multiplier

```
module au (
  input logic [7:0] a, b, c,
  input logic sel,
  output logic [15:0] d
);

logic [15:0] e;
logic [15:0] sel_h, sel_l;

// Resource sharing architecture

always_comb
begin
  if(sel) e = b; else e = c;
  d = multiply(a, e);
end

// Functional sva checker

always@(a, b, c, sel)
begin
sel_h = a*b;
sel_l = a*c;

if(sel)
sel_high : assert (sel_h == d);
if (!sel)
sel_low : assert (sel_l == d);

end
endmodule
```

Example 4.1 shows a simple multiplier. Only one multiplier is used and the multiplication is done based on the input that is selected by the "sel" line. Two simple checkers named "sel_high" and "sel_low" can be written to verify the multiplier. The check "sel_high" is active when the "sel" line is high and the check "sel_low" is active when the "sel" line is low. The user can choose to use any type of multiplier relevant to the user's environment. For example, it can be a shift/add multiplier, a booth multiplier or something else. From a functional verification standpoint, we need to make sure that no matter which type of multiplier algorithm is used, the end output result matches. Figure 4-1 shows the results produced by these two checkers. Note that the checker is active, based on the status of the "sel" signal (immediate assertion).

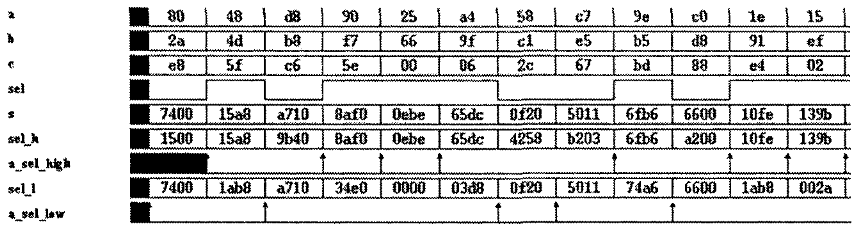

Figure 4-1. Waveform for Multiplier checker

4.2 Sample Design – Arithmetic unit

In this section, verification of an arithmetic unit is discussed (a Walsh-Hadamard Transform (WHT) block). WHT is a common algorithm used in still image compression applications. WHT is used to convert the pixels from time domain to frequency domain before the image is encoded. Typically, these algorithms are tested very easily with C or Matlab programs. But when the algorithms are converted to hardware, it goes through severe optimizations that will make it more hardware efficient. It is very common to provide the same input data to both the golden C model and the RTL model. The RTL is verified by comparing its output with that of the C model. In this section, SVA is used to produce the golden results dynamically during a simulation and compared with the results from RTL.

4.2.1 WHT Algorithm

The WHT algorithm is an 8x8 matrix multiplication. In image compression applications, data is processed one block at a time. Each block is an 8x8 matrix, or in other words, 64 data points. The objective is to perform a matrix multiplication between 2 matrices, each of size 8x8, to produce the end result, an 8x8 matrix. Due to the repetitive nature of the matrix multiplication, this can be achieved one block at a time. The WH matrix is defined as follows. Since the matrix without the scaling factor consists of +1 and –1, the transform operation consists simply of addition and subtraction.

```
WHT [8][8] =
    {
        {1, 1, 1, 1, 1, 1, 1, 1},
        {1, 1, 1, 1,-1,-1,-1,-1},
        {1, 1,-1,-1,-1,-1, 1, 1},
        {1, 1,-1,-1, 1, 1,-1,-1},
        {1,-1,-1, 1, 1,-1,-1, 1},
        {1,-1,-1, 1,-1, 1, 1,-1},
        {1,-1, 1,-1,-1, 1,-1, 1},
        {1,-1, 1,-1, 1,-1, 1,-1}

        };
```

4.2.2 WHT Hardware implementation

In hardware, the WHT algorithm can be optimized to reduce the number of additions and subtractions performed. This is achieved by exploiting the redundant additions and subtractions performed on the same set of data. Each arithmetic block is optimized to perform a 1x8 by 8x8 matrix multiplication. In other words, one row of data (8 data points) is processed at a time. The simplified arithmetic unit performs 3 stages of addition and subtraction to produce one row of output data. Figure 4-2 shows the block diagram of the hardware implementation of the WHT algorithm.

Assume D1, D2, D3, D4, D5, D6, D7 and D8 form a row of data. Now this row of data has to be processed through the WHT matrix. The equations of the three stages of optimized arithmetic operations can be listed as follows.

Stage 1

$Y1 = D1 + D2, Y2 = D3 + D4, Y3 = D5 + D6, Y4 = D7 + D8,$
$Y5 = D1 + D4, Y6 = D5 + D8, Y7 = D2 + D3, Y8 = D6 + D7,$
$Y9 = D1 + D3, Y10 = D6 + D8, Y11 = D2 + D4, Y12 = D5 + D7$

Stage 2

$Z1 = Y1 + Y2, Z2 = Y3 + Y4, Z3 = Y1 + Y4, Z4 = Y2 + Y3,$
$Z5 = Y1 + Y3, Z6 = Y2 + Y4, Z7 = Y5 + Y6, Z8 = Y7 + Y8,$
$Z9 = Y5 + Y8, Z10 = Y7 + Y6, Z11 = Y9 + Y10,$
$Z12 = Y11 + Y12, Z13 = Y9 + Y12, Z14 = Y11 + Y10$

Stage 3

$X1 = Z1 + Z2, X2 = Z1 - Z2, X3 = Z3 - Z4, X4 = Z5 - Z6,$
$X5 = Z7 - Z8, X6 = Z9 - Z10, X7 = Z11 - Z12, X8 = Z13 - Z14$

X1, X2, X3, X4, X5, X6, X7 and X8 form a row of processed output data.

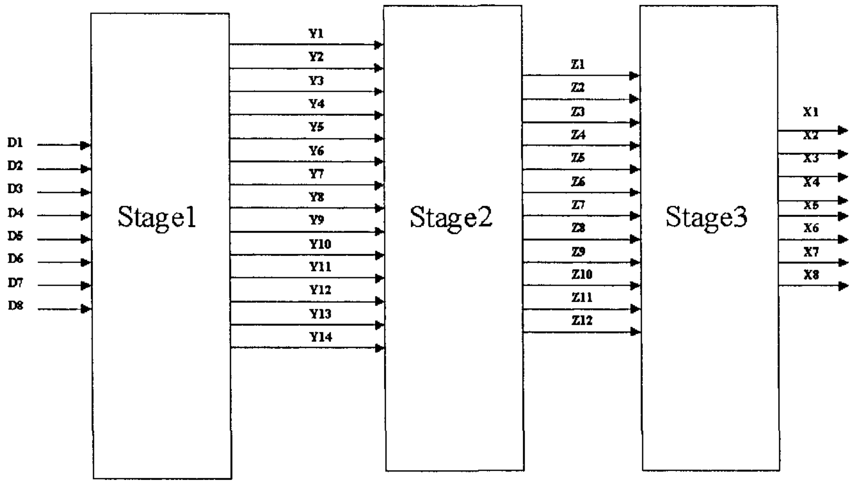

Figure 4-2. WHT hardware block diagram

4.2.3 SVA Checker for WHT block

To functionally verify this block, a checker does not have to know the internal implementation details of this block. A checker should be able to produce the golden result and compare it with the design data. The golden data can be produced within the SVA checker by simply performing a matrix multiplication. This data can then be compared with the output of the WHT block. Figure 4-3 shows a simple checker configuration for the WHT block.

It is very common to register the outputs of such combination blocks to obtain the most stable data. A checker can be written using the enable signal of the register as the trigger. The results produced within the checker should be compared with the original registered output of the WHT arithmetic block. A sample SVA checker is shown in Example 4.2.

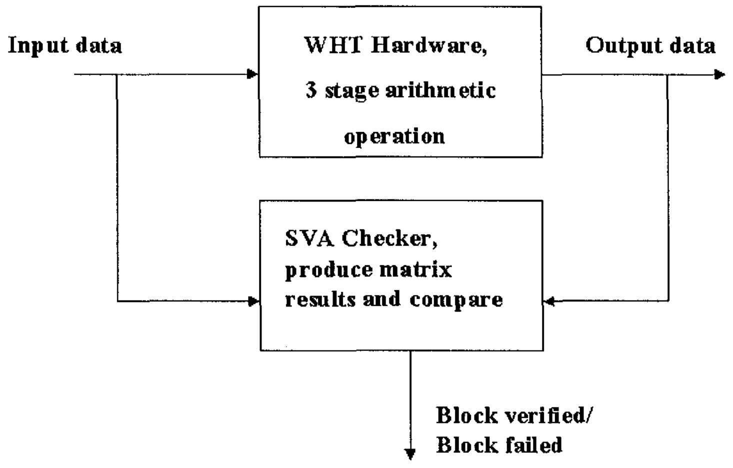
Input data
WHT Hardware, 3 stage arithmetic operation
Output data
SVA Checker, produce matrix results and compare
Block verified/ Block failed

Figure 4-3. WHT checker configuration

Example 4.2 SVA Checker for WHT block

```
module au_comp_chk (
  input logic clk, reset, enable1, enable2,
  input logic signed [15:0]
        d1, d2, d3, d4, d5, d6, d7, d8,
  input logic signed [15:0]
        o1, o2, o3, o4, o5, o6, o7, o8
);

logic signed [15:0] in_local[0:7];
logic signed [15:0] out_orig[0:7];
logic signed [15:0] out_local[0:7];

integer i, k;

integer wh_local[0:7][0:7] =
    {
        {1, 1, 1, 1, 1, 1, 1, 1},
        {1, 1, 1, 1,-1,-1,-1,-1},
        {1, 1,-1,-1,-1,-1, 1, 1},
        {1, 1,-1,-1, 1, 1,-1,-1},
        {1,-1,-1, 1, 1,-1,-1, 1},
        {1,-1,-1, 1,-1, 1, 1,-1},
        {1,-1, 1,-1,-1, 1,-1, 1},
          {1,-1, 1,-1, 1,-1, 1,-1}

          };

always@(o1, o2, o3, o4, o5, o6, o8)
   begin
     out_orig[0] <= o1;
     out_orig[1] <= o2;
     out_orig[2] <= o3;
     out_orig[3] <= o4;
     out_orig[4] <= o5;
     out_orig[5] <= o6;
     out_orig[6] <= o7;
     out_orig[7] <= o8;
   end

   always@(d1, d2, d3, d4, d5, d6, d7, d8)
   begin

     for(i=0; i<8; i++)

     begin

     out_local[i] <=
    (d1*wh_local[i][0]) + (d2*wh_local[i][1]) +
    (d3*wh_local[i][2])   +   (d4*wh_local[i][3])   +
   (d5*wh_local[i][4]) +   (d6*wh_local[i][5])        +
   (d7*wh_local[i][6]) + (d8*wh_local[i][7]) ;

   end

   end

   genvar j;
   generate

   for(j=0; j<8; j++)
   begin : loop
   a_au_comp_chk_o :
   assert property
   (@(posedge clk) (reset &&enable2) |->
               (out_local[j] == out_orig[j]));
```

```
  end

  endgenerate

  endmodule

bind au_comp au_comp_chk a1
(clk, reset, enable1, enable2, d1, d2, d3, d4,
d5, d6, d7, d8, o1, o2, o3, o4, o5, o6, o7,
o8);
```

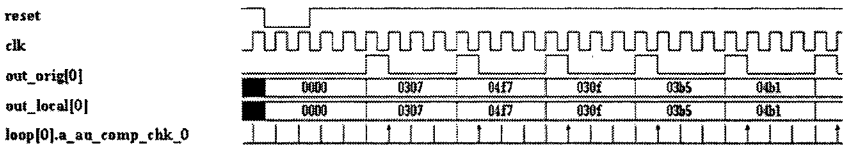

Figure 4-4. Waveform for WHT checker

The checker calculates the expected output locally and puts the results in an array named "out_local." The original output data from the design is also stored in an array named "out_orig." The checker creates an array of properties to verify the 8 data points by using the "generate" statement. The special variable "genvar," allows the use of a "for loop" to create 8 separate properties to verify each one of the data points simultaneously. The property is asserted when the enable signal is high and the design is not in reset. Each property will compare the respective output data, "out_local" and "out_orig." If they are not equal, the assertion fails. Figure 4-4 shows the results from the first data point in the array.

4.3 Sample Design – A JPEG based data-path design

In this section, verification of a sample JPEG model is discussed. The design block is part of a JPEG encoder wherein data is read from memory and transformed using certain arithmetic algorithms. The transformed data is then stored in a memory for package and transmission.

There are three main modules in the JPEG model - the data feeder, the data path and the data control modules. The top level block diagram of the JPEG model is shown in Figure 4-5. Details of each module are provided below:

- The data control module helps with the hand shaking process with the memories and also generates control signals for the data path and data feeder modules to process the data.
- The data feeder module reads a block of data at a time from the memory and provides it to the data path module to process.
- The data path module performs the arithmetic operation on the block of data and stores it in memory.

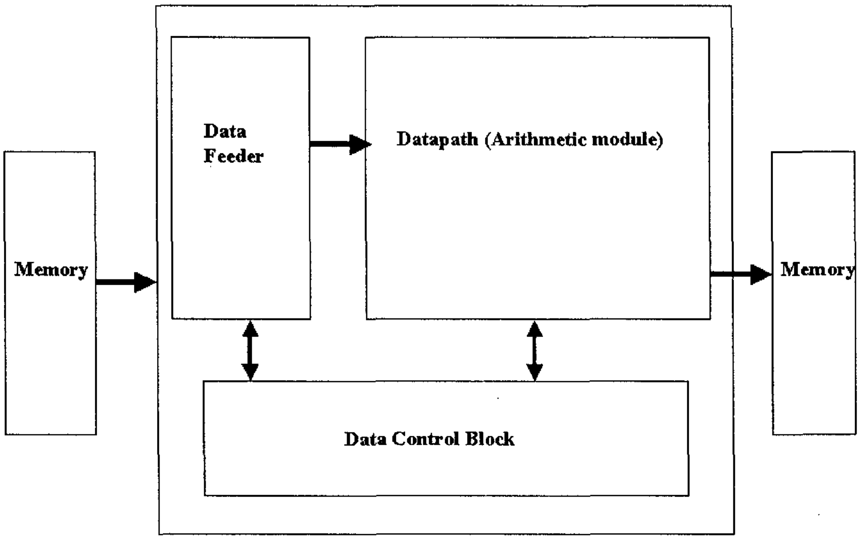

Figure 4-5. Block diagram of JPEG model

4.3.1 A closer look at the individual modules

Data feeder module

Figure 4-6 shows a block diagram of the data feeder module. This module consists of two modules, a serial in parallel out module (SIPO) and a parallel in parallel out module (PIPO). The SIPO reads in one 16-bit data at a time and provides 64 16-bit data in parallel as output.

When enabled (sipo_enable), the SIPO will start pushing the input data into the shift registers. Once we have 64 data samples, the SIPO will be disabled and the PIPO will latch the valid data out. The latched data is used by the datapath module for further processing. The data control block generates the enable signals for the SIPO and PIPO. Figure 4-7 shows a sample waveform that shows the functionality of the data feeder module.

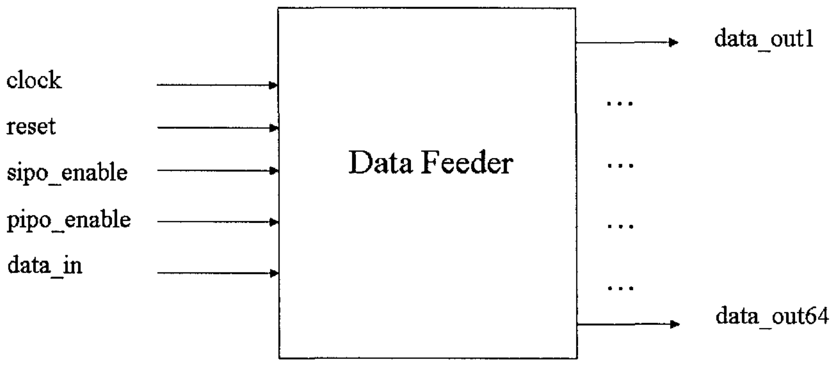

Figure 4-6. Data feeder block diagram

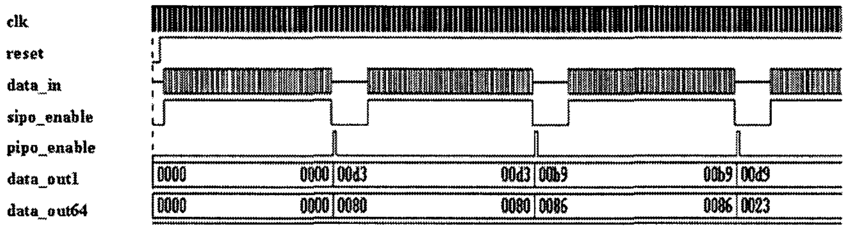

Figure 4-7. Waveform for Data feeder module

Data path module

The data path module takes in 64 16-bit data at a time and performs certain arithmetic operations on them. The process extends over multiple cycles to accommodate the completion of all operations. A multi-level pipeline is used to accomplish this task. Figure 4-8 shows a block diagram of the pipeline used to perform the arithmetic operations.

The data path is a simple latch based pipeline design. The data goes through four stages of processing, transform1, transpose, transform2 and quantization. After each stage of processing, the stable values are latched to the next stage by using a PIPO. The PIPO is a latch that is controlled by the enable signals generated by the data control module. Each module gets 2 clock cycles to complete their process. In other words, the data control module generates 4 enable signals at an interval of 2 clock cycles that are used to latch the stable data from the output of each stage. Figure 4-9 shows the relationship between the control signals of the pipeline.

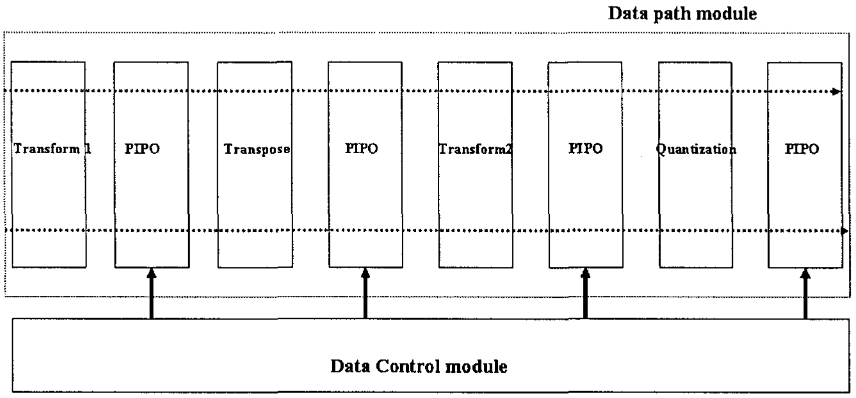

Figure 4-8. Block diagram showing details of the pipeline

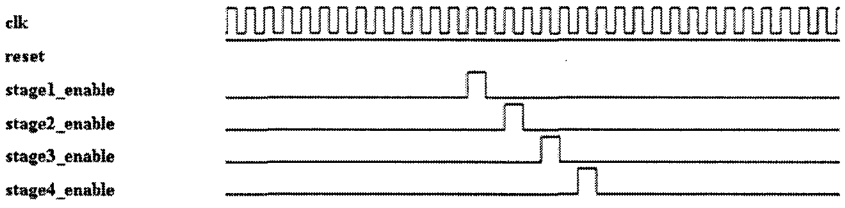

Figure 4-9. Waveform for pipeline control

Data control module

The data control module is a simple finite state machine (FSM). It produces the control signals required to keep the data moving along the pipeline smoothly. Figure 4-10 is a sample block diagram of the data control module and Figure 4-11 is a waveform showing the generation of the control signals.

The data control module generates the control signals for the data feeder and the data path modules. The state machine starts operating once it gets a "get_data" signal from outside. This kicks off the counters to generate the "read" signal for the memory and also the "read_address." It helps read 64 valid data every time.

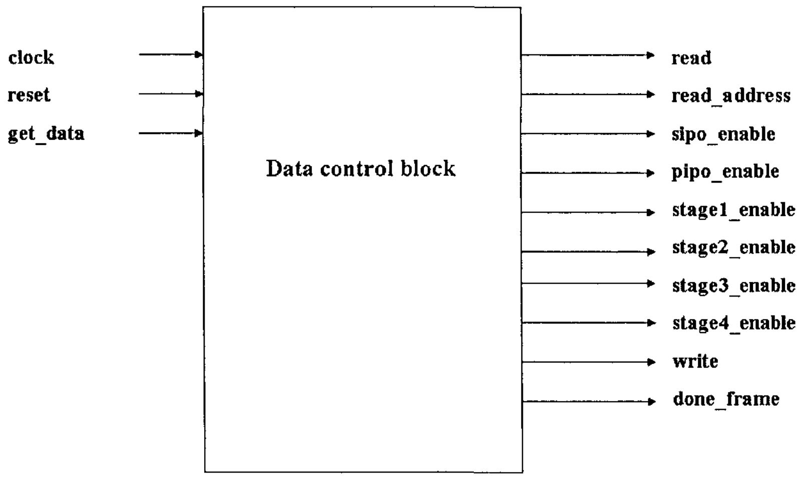

Figure 4-10. Block diagram for data control block

Once 64 valid data is read, the "read" is disabled and the enable signals for the pipeline are generated sequentially. After the data is processed through the 4 stages of pipeline, the "write" signal is generated to store the processed data into a memory model. After the "write," a fresh set of 64 data is read from the memory. This process continues until all the data is read from the memory. In the sample JPEG design used, the memory can hold 262144 bytes (equivalent to a 512 X 512 image). This means that the control signal generation is repeated 4096 (262144/64) times to finish the processing of all data points. After completing all blocks, the control block asserts the "done_frame" signal and immediately the "get_data" signal is de-asserted.

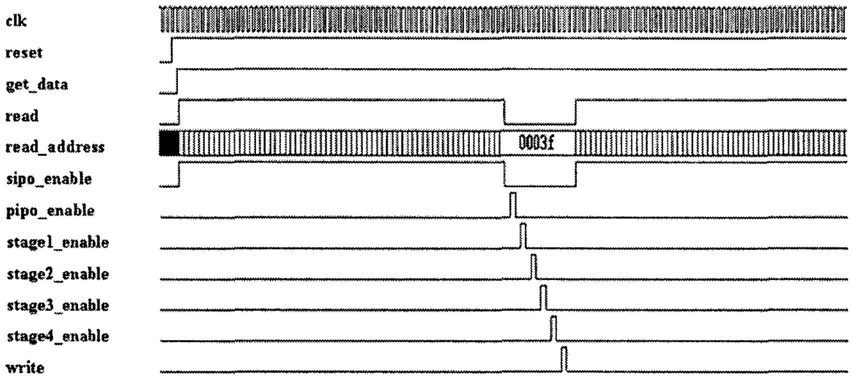

Figure 4-11. Waveform for control block

What needs to be verified?

- The validity of the data flow control signals. Are they generated correctly according to the timing requirements?
- Is the data path working correctly, is it producing valid output data to be stored in the memory?

4.3.2 SVA Checkers for the JPEG design

Based on the description of the design, the following list of checkers needs to be written to verify the design thoroughly.

JPEG_chk1: "get_data" and "done_frame" signals are mutually exclusive.

The design starts reading data from the memory when the "get_data" signal is asserted. While acquiring and processing data, "done_frame" signal should be held low. When all data has been processed, the design asserts the "done_frame" signal and de-asserts the "get_data" signal. Hence, these two signals can never be asserted at the same time.

```
property p_mutex;
 @(posedge clk) ((reset_) |->
              not (done_frame && get_data));
endproperty
a_mutex: assert property(p_mutex);
```

This is a mutually exclusive condition and can be written easily with a "**not**" operator. The "**not**" operator states that the test expression can never be true. The checker kicks off on every positive edge of the clock. The result of the checker "a_mutex" is shown in Figure 4-12.

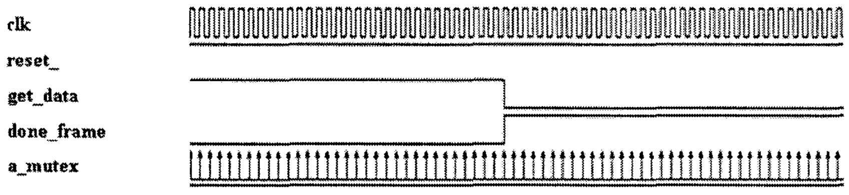

Figure 4-12. Waveform for JPEG_chk1

JPEG_chk2: The "read" signal is held high for 64 cycles continuously and during this period, the "read_address' is incremented by one in every clock cycle.

The sample design processes 64 data points at a time, which means that, a burst read is done for 64 cycles to get all the data that need to be processed. By verifying the above statement, we prove that we read a unique data on each of the 64 clock cycles.

```
sequence s_read;
  (rd_addr == $past (rd_addr)+1) [*0:$] ##1
                 $fell (rd);
endsequence

property p_read;
  @(posedge clk)
 (($rose (rd) && reset_) |->
                           s_read);
endproperty

a_read: assert property(p_read);
```

The read address is checked for the increment by using the **$past** operator. The value of "rd_addr" in the current clock cycle should be the value in the previous clock cycle incremented by 1. This checking is done from the rising edge of the read signal (**$rose** (rd)) until the falling edge of the read signal (**$fell** (rd)). The "**repeat until** [*0:$]" operator is used to check the validity of the address (until the "rd" signal is de-asserted). The result of the check "a_read" is shown in Figure 4-13. Every time there is a rising edge on the read signal, a valid match on the property is shown. The property itself will be active for several clock cycles but the match is indicated only once in the results and this is the point where the property begins to become active.

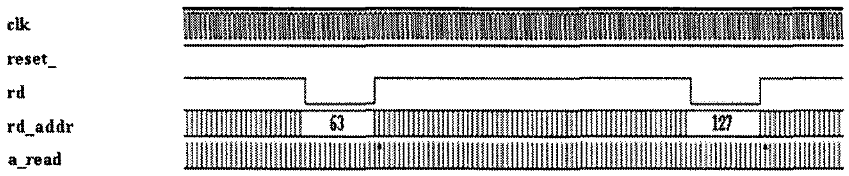

Figure 4-13. Waveform for JPEG_chk2

JPEG_chk3: The "sipo_en" is held high for 64 cycles during the read cycle and then disabled, 2 cycles later the "pipo_en" signal is asserted to latch the data that will be processed by the datapath module.

The data feeder module depends on the "sipo_en" and the "pipo_en" signals to provide the valid data to the datapath module. This checker verifies the functionality of the data feeder module.

```
sequence s_datafeeder;
    sipo_en[*64] ##1 $fell (sipo_en) ##1
        latch_en ##1 !latch_en;
endsequence

property p_datafeeder;
  @(posedge clk) ($rose (sipo_en) && reset_) |->
        s_datafeeder;
endproperty

a_datafeeder: assert property(p_datafeeder);
```

A simple **repeat operator** (*) is used to monitor whether the "sipo_en" signal is held high for 64 clock cycles. Once the "sipo_en" signal goes down, a "latch_en" pulse is asserted. The result of the check "a_datafeeder" is shown in Figure 4-14.

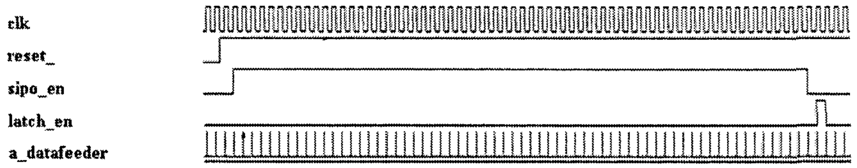

Figure 4-14. Waveform for JPEG_chk3

The sequence "s_datafeeder" can also be written as follows.

```
sequence s_datafeeder;
   ##64 $fell (sipo_en) ##2
   latch_en ##1 latch_en;
endsequence
```

In this description of "s_datafeeder," the falling edge of "sipo_en" is checked after 64 clock cycles and it does not guarantee that "sipo_en" was held high during these 64 clock cycles.

JPEG_chk4: In the datapath module, each stage is enabled with a gap of 2 clock cycles.

Every stage in the data path has 2 clock cycles to provide the stable value for the next stage. The signals "dp1_enable," "dp2_enable," "dp3_enable" and "dp4_enable" help latch the stable data at each stage and they are asserted in a sequence of 2 clock cycle gaps. This makes sure that the data flow is happening correctly.

```
sequence s_control;
  dp1_en ##1 !dp1_en ##1 dp2_en ##1 !dp2_en ##1
  dp3_en ##1 !dp3_en ##1 dp4_en ##1 !dp4_en ##1
  wr ##1 !wr;
endsequence

property p_control;
  @(posedge clk) $fell (latch_en) |=> s_control;
endproperty
a_control: assert property(p_control);
```

Each enable signal for the PIPO is a pulse of one clock cycle and they are generated 2 clock cycles apart. The sequence "s_control" monitors the rise and fall of each one of these control signals in a simple sequential concatenation method. The sequence starts when the data is loaded into the data path module (latch_en). An **implication operator** (|->) is used to indicate that the falling edge of the signal "latch_en" is the gating condition for the rest of the sequence to be tested. The sequence ends when the data has passed through the data path and the processed data has been written into the memory (wr).

JPEG_chk5: From the rising edge of "get_data" to the falling edge of "get_data," sequences "s_datafeeder" and "s_control" are repeated 4096 times unless the design is reset.

This guarantees that all data has been processed in the correct order. This is also used as a functional coverage check to make sure that all data from the memory has been processed.

```
property p_control_all;
  @(posedge clk) ($rose (sipo_en) && reset_) |->
          s_datafeeder ##1 s_control;
endproperty

property p_block;
 @(posedge clk)
  $fell (get_data) && $rose(done_frame) |->
                              (block == 4095);
endproperty

a_control_all: assert property(p_control_all);
c_control_all:
     cover property(p_control_all)block++;

a_block: assert property(p_block);
```

To make sure that the entire sequence repeats 4096 times to process all the data points, the checks JPEG_chk3 and JPEG_chk4 should be concatenated. The property p_control_all will start when the data is read from the memory (sipo_en) and will end when the processed data has been written to the output memory (wr). A "cover" statement can be declared for the property p_control_all that will provide information on how many times the property really succeeded and how many times it succeeded vacuously. If there is a real success, the variable "block" is incremented by one each time. The number of real successes should equal 4095. The property a_block uses this variable to verify that all blocks of data have been verified. If the "done_frame" signal has a rising edge and on the same clock cycle if the "get_data" signal has a falling edge, that indicates that the last block of data is being processed and at this point the variable "block" should indicate a value of 4095. The result of the check "a_control" is shown in Figure 4-15. The result of the check "a_block" is shown in Figure 4-16.

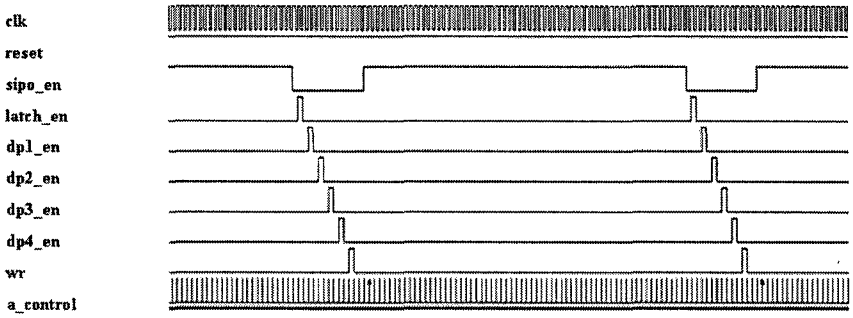

Figure 4-15. Waveform for JPEG_chk5

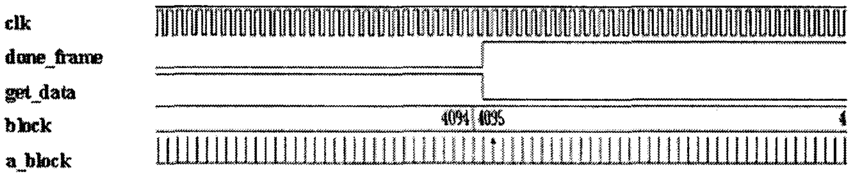

Figure 4-16. Waveform for check a_block

4.3.3 Data checking for the JPEG model

In Section 4.3.2, SVA checkers were written to verify that all the control signals are generated correctly. This guarantees that the data is moving along the pipeline smoothly. This does not check for data integrity. Each block in the pipeline performs some transformation to the data and this needs to be verified. A very common method used to verify the data is by dumping the output to a file. This output file is later compared with the result produced by the golden model as a post-process. While this method could work, it has a few disadvantages:

1. The simulation has to finish in order to compare the output with the expected results. If the output is wrong, then a lot of simulation cycles have been wasted.

2. This method is not very debug friendly since it does not say at which stage of the pipeline the output data started failing. One simple way to overcome this would be to dump the output of each and every pipeline stage and do the comparisons. While this will help debug, it will still waste simulation cycles as mentioned before..

A more efficient way to do the data checking will be to do the comparison dynamically. This can be accomplished in several ways and each user has to decide which one is good for his or her simulation environment. Since the data checking is a repetitive process, the dynamic comparisons can be shut down after a few data packets have been verified. In other words, a goal can be set to gain confidence on the dynamic data checking process and once the goal is attained, these checkers can be shut down, hence improving simulation throughput.

Possible steps for dynamic data checking of JPEG model:

- Simulate the golden C model that will produce results for each pipeline stage as shown in Figure 4-17. While it is not always easy to match the RTL pipeline stages with that of the golden C model, it is also not impossible.
- Generate the following output data files from the golden C model of the JPEG design before simulating the actual RTL - Wh1.dat (output of transform1), Xpose.dat (output of transpose), Wh2.dat (output of transform1), Quantize.dat (output of quantization).
- Use the same input data file on the RTL to perform data checking.
- Create a generic checker that can load the golden results into the simulation environment and then compare them dynamically as the RTL simulates, as shown in Figure 4-18. As the simulation proceeds, at the relevant trigger points, the checker will compare the golden results with the design results and report any failures. A sample data checker is shown in Example 4.3.

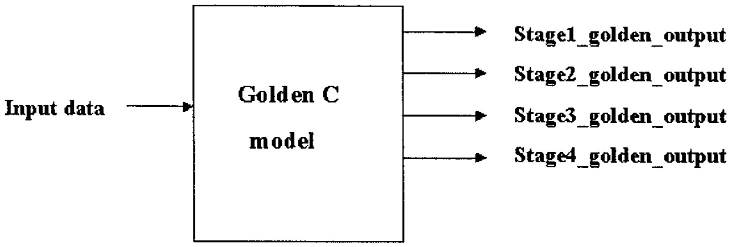

Figure 4-17. Golden output from C model

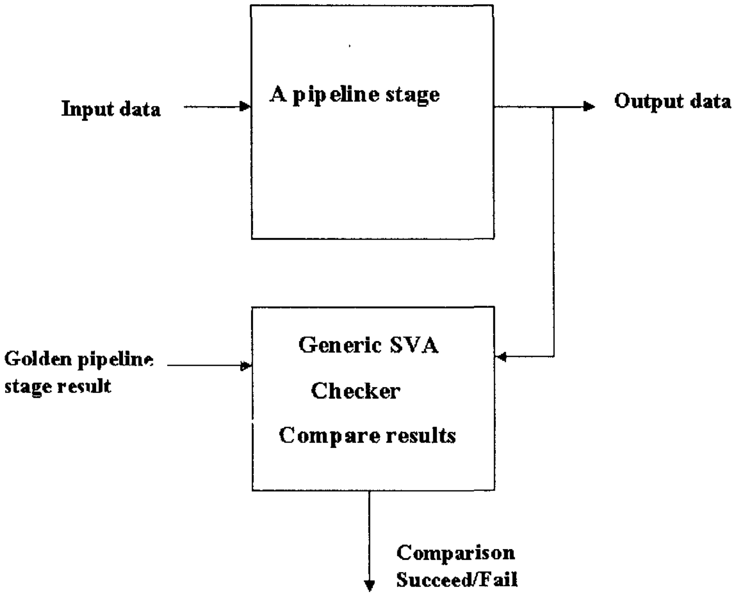
Input data
A pipeline stage
Output data
Golden pipeline stage result
Generic SVA Checker Compare results
Comparison Succeed/Fail

Figure 4-18. Dynamic Pipeline checker

Example 4.3 SVA checker for data-path verification

```
module dp_chk(
input logic reset, clk, enable,
input logic [15:0]
      d1, d2, d3, …, d61, d62, d63, d64);
parameter data_file = "";
parameter identity = "";
```

```
integer i=0, j=0;
integer blk=0;
integer fd, fdl;

logic [31:0] pix_in_temp;
logic [15:0] local_array[0:63];
logic [15:0] pix_in [0:262143];

// use $fopen construct to open the golden
// results file

initial
begin
fd = $fopen(data_file, "r");
end

// copy design data to a local array

always@(*)
begin
local_array[0] <= d1;
local_array[1] <= d2;
local_array[2] <= d3;
..
..
local_array[62] <= d63;
local_array[63] <= d64;
end

// load actual results

always@(negedge enable)
begin
  if(reset)
  $display
  ("\nDATA CHECKING: Block number %0d\n", blk);
  for(j=0; j<64; j++)
  begin
  fdl = $fscanf(fd, " %x", pix_in[j]);
  end
  blk++;
end
```

```
// compare results

genvar k;
generate
for(k=0; k<64; k++)
begin: dchk
a_dp_chk: assert property(
  @(posedge clk) (reset && $fell(enable)) |=>
  (pix_in[k] == local_array[k])) else $fatal;
end
endgenerate

endmodule

// check that data is put into blocks of 64
correctly

bind data_feeder dp_chk
#(.data_file("input_image.dat"),
.identity("INPUT")) dpchk1
(reset_, clk, latch_en, q0, q1, ….. ,
q61, q62, q63);

// check that the output of first wh transform is
// correct

bind datapath dp_chk
#(.data_file("wh1.dat"), .identity("WH1"))
dpchk2
reset, clk, dp_enable1,
dwl1,dwl2,….,dwl61,dwl62,dwl63,dwl64);

// check that the transposed data is correct

bind datapath dp_chk
#(.data_file("xposed.dat"),
.identity("TRANSPOSE")) dpchk3
  (reset, clk, dp_enable2, dwlt1, dwlt2, ….,
 dwlt61, dwlt62, dwlt63, dwlt64);

// check that the output of the second wh
// transform is correct
```

```
bind datapath dp_chk
#(.data_file("wh2.dat"), .identity("WH2"))
dpchk4
  (reset, clk, dp_enable3, dwltwl1, dwltwl2, ....,
 dwltwl63, dwltwl64);

// check that the output of quantization is
// correct

bind datapath dp_chk
#(.data_file("quantized.dat"),
.identity("QUANTIZATION")) dpchk5
(reset, clk, dp_enable4,
do1,do2,....do62,do63,do64);
```

Example 4.3 shows a generic SVA datapath checker and how it is bound to the various stages of the pipeline design.

- The checker defines 2 parameters that help identify the checker to be a unique one. The parameter "data_file" defines which golden file should be used by a specific instance of the checker. The parameter "identity" defines which section of the data path the checker is bound to.
- The golden data is stored in a file. This data file is opened for reading purpose using the $fopen construct.
- On trigger (the enable signal), the actual design outputs are stored in the checker locally (local_array). Note that the datapath processes 64 data points at a time and hence, only 64 data points should be read from the golden file on a trigger. A variable is incremented by 1 on each trigger to document which block of the image is being verified currently.
- Using a "generate" statement, 64 checkers are created, one for each data point. The "for" loop helps loop around and check all 64 data points on every trigger.
- The action block of the assert statement uses a $fatal construct. This instructs the simulator to exit the simulation if there is a violation. This prevents running the simulation unnecessarily after finding mismatches.
- The checker can be connected to specific points of the data path by using the "bind" construct. By defining the parameters relevant to each point, each checker becomes a unique instance.

A sample simulation log is shown below.

```
DATA CHECKING: Block number 1
"adv_datapath.sva", 118:
 tb.jpeg_int.datapath_inst.dpchk5.dchk[0].a_dp_ch
 k: started at 795s failed at 805s
  Offending '(pix_in[0] == local_array[0])'
"adv_datapath.sva", 118:
Fatal: "adv_datapath.sva", 118:
tb.jpeg_int.datapath_inst.dpchk5.dchk[0].a_dp_chk
: at time 805
```

Note that the failure is coming from the instance of the checker attached to the "QUANTIZATION" module (instance dpchk5) of the data path. The failure clearly points out the time of failure and which data of the block failed. For example, in the above log, data point 0 of Block 1 failed.

While this is one way to perform dynamic data checking, this might not be suitable for all designs. Each design is different and they have different specifications and requirements. This method can be used as a model to derive a methodology suitable for a specific design.

4.4 Summary for data intensive designs

- SVA provides the capability to perform arithmetic operations and is capable of using most SystemVerilog data types.
- By using Verilog tasks and functions, data checking can be automated and functional coverage information on the design can be obtained.
- Dynamic SVA checkers for data path uses the simulation cycles wisely and does not wait until the end of the simulation to find about design problems. The checkers also make debugging easy by pointing to the exact area of failure.