Chapter 4

Implementation of a Test Circuit

We use a simplified cost model (which is the number of transistors) to evaluate the performance of our BIST design methods. Although the simplified cost model allows us to evaluate our methods with reasonable complexity, the actual performance may deviate from the model. The model does not account for the area for routing and additional delays incurred by the test circuitry. Hence, we propose an implementation of a DSP algorithm that incorporates our BIST method. The implementation enables us to accurately measure the performance of our BIST methods and to improve our cost model. It would also help us to identify areas to be improved in our BIST methods. Thus we choose a DCT (Discrete Cosine Transform) algorithm for the implementation [43][44].

We implemented an 8×8 two-dimensional DCT (2-D DCT) processor. The main part of the circuit was described in a high level language called Silage and high-level synthesis tools called HYPER and our high-level BIST synthesis tool, ADVBIST_h (which implements our region-wise heuristic ILP method) were used to generate data paths with BIST. The rest of the circuit was described in VHDL. The circuit was laid out based on our ASIC design flow. We implemented two versions of the circuit, one with incorporation of our BIST method and the other one without BIST.

In the following section, we describe the ASIC design flow used for the implementation and a library development first and then implementation of a DCT circuit including basic operation of the DCT algorithm, a fast algorithm, and details on operation with a timing diagram. We also describe the architecture for the data path, the controller and the BIST circuit embedded in the DCT circuit.

4.1 ASIC Design Flow

To design and fabricate a chip involves a variety of CAD tools and is a complex task as shown in the design flow. CAD tools are used for high level synthesis and simulation, logic synthesis and simulation, circuit simulation, and layout verification. The design flow adopted at Virginia Tech to design ASIC chips is shown in Fig. 4.1.

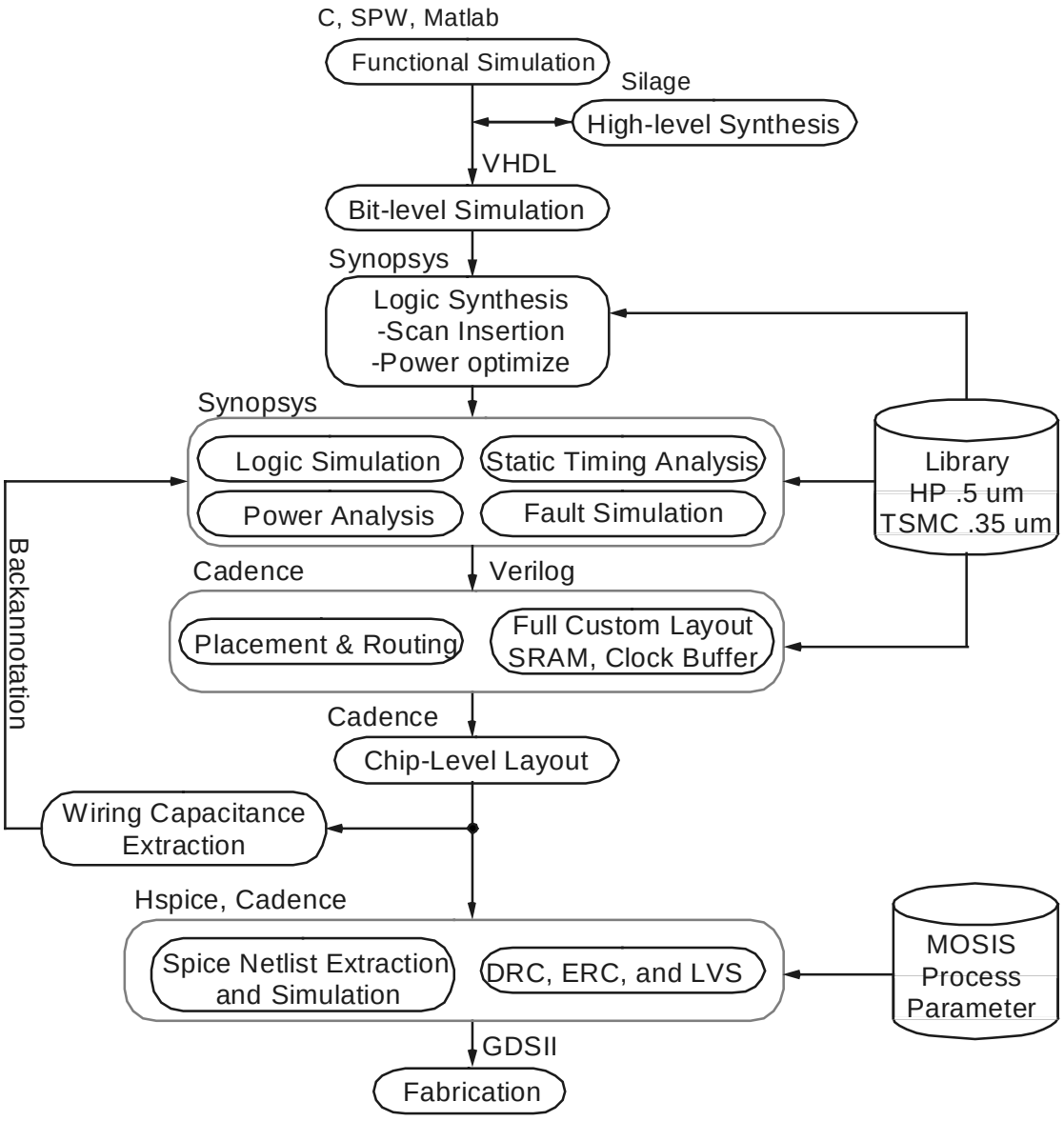


Figure 4.1 Design flow

4.1.1 Behavior and Logic Design

The design process starts from functional simulation using a programming language (C or C++) or commercial system-level tools such as Matlab or SPW of Cadence. Data path intensive

circuits such as FIR/IIR filters or various DSP applications may be designed using a high-level synthesis tool. (The test chip design in the high-level is quite different from the main design flow and is described in Section 4.2.) A synopsys logic synthesis tool is applied to synthesize a gate-level circuit from a VHDL model. The VHDL model can contain RT-level data paths, and controllers, glue logic, and handcrafted design blocks. During the logic synthesis, scan insertion or power optimization technique may be applied for better testability or lower power consumption.

Upon completion of the logic synthesis, logic simulation is performed to the synthesized gate-level circuits. Static timing analysis is applied to check if there is any path with potential timing problems (such as setup and hold time violations) for the circuit. In case there is a setup time violation, timing optimization is applied to the problematic critical paths to fix the problem. Insertion of buffers in the data path between flip-flops is applied for a hold-time violation. In addition, power analysis, fault simulation, and various design-for-testability techniques may be applied for the synthesized circuit.

4.1.2 Circuit and Layout Design

Before placement and routing, pads for primary input and output are inserted into the netlist. Pads for power and ground (that are not connected to the core logic) are also inserted. Silicon Ensemble of Cadence is used to place and route a chip in a semicustom design style. Silicon Ensemble accepts library cells in LEF (Layout Exchange Format). The connection between pads and internal cells are taken care of by the place and route tool. A chip-level layout is necessary to connect semicustom and full custom logics (such as SRAMs). A spice netlist is extracted from the layout and is simulated for the detailed verification and rule checking.

4.2 High-Level BIST Design

Fig. 4.2 shows the overall flow of the high-level BIST synthesis which was used to design a test circuit. The test circuit was described in Silage language and was synthesized using a high-level synthesizer, HYPER [68]. Input parse, scheduling, and module assignment were performed by HYPER. In our environment, all the assignments were guided for our target architecture, which is based on multiplexers and registers as described in Chapter 2.

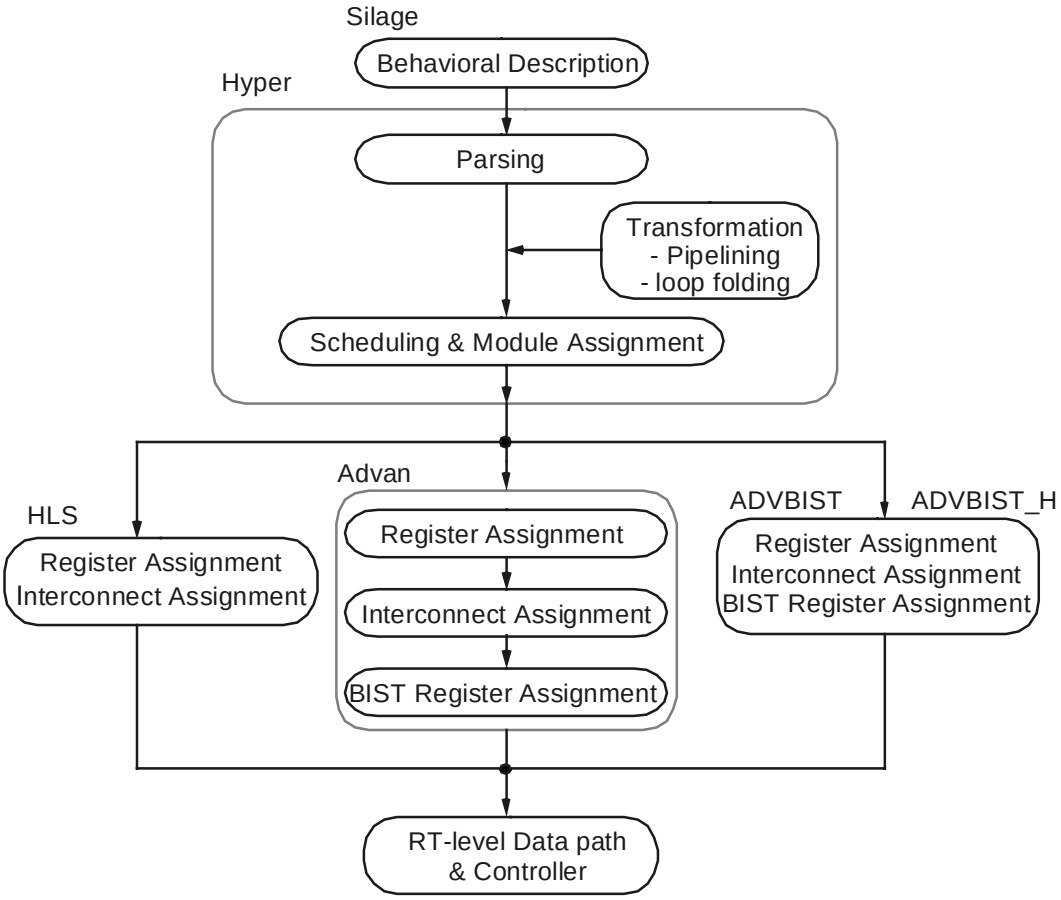


Figure 4.2 High-level BIST synthesis system

After parsing, the clock period is set by the user, which serves as a constraint for scheduling. A scheduling and a module assignment are performed concurrently in HYPER. The number of necessary modules is calculated to meet the time requirement. The number of control steps needed to complete the circuit operation is also reported. It is possible to change the number of control steps and of modules by changing the clock period.

The result of HYPER is described in AFL format that is a scheduled and module assigned data flow graph. The AFL file is then converted to various formats for our tools such as HLS, ADVAN, ADVBIST, and ADVBIST_H. Detailed descriptions on the use of the tools are covered in the Appendix A.

4.3 Library Development

One of the key elements in ASIC design flow is library. Library development requires four tasks as shown in Fig. 4.3: logic design, circuit design, layout design, and timing parameter extraction. We describe our efforts to develop an ASIC library in the following.

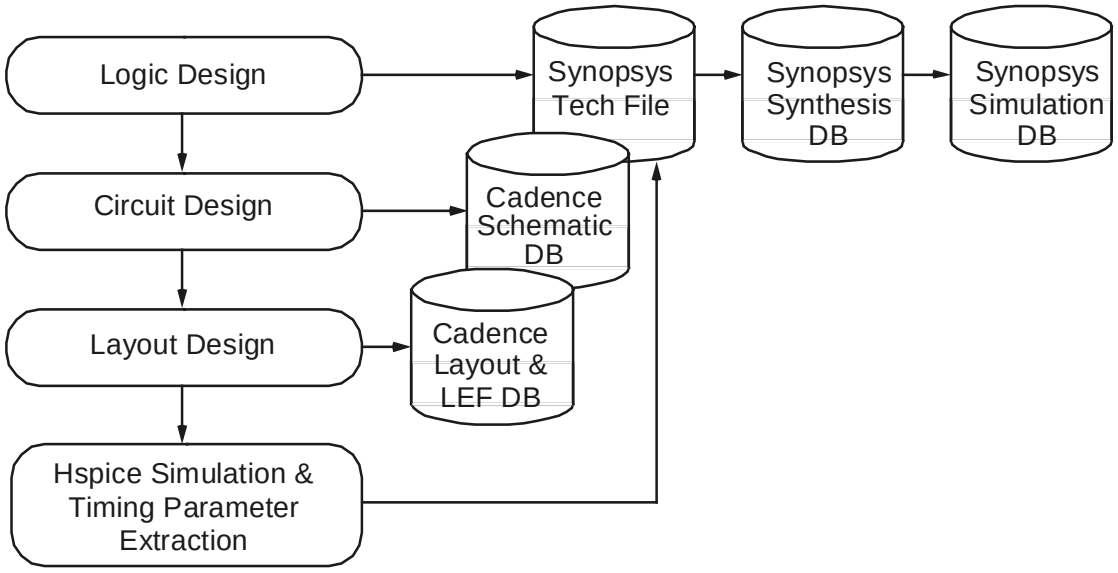


Figure 4.3 Library development process

Characteristics of each library cell are described in the Synopsys technology file in logic design phase. Descriptions of a cell include the logic function, input/output pins, input capacitance, area, and pin-to-pin timing. The capacitance of an input pin is obtained from SPICE simulation of a SPICE netlist extracted from the layout. The area of a cell is represented in a real number in which the unit area is the area of a two-input NAND gate.

There are several timing models to represent pin-to-pin timing. Synopsys supports timing models such as linear, piecewise linear, and a two-dimensional table model. The linear timing model is based on the assumption that a delay is proportional to a loading capacitance. It is simple but less accurate than the other models. The two-dimensional table model is most widely used in industry but complex in extracting timing parameters. In this model, a delay is dependent

both on a loading capacitance and an input slope. There are two issues regarding the model. The first one is the number of sample points. For example, a 3×3 table model requires 9 sample points. An adequate number of points depends on the target process and on the cell. A 4×4 table model which is used widely in industry, seems a good compromise between complexity and accuracy. The other issue is the sample points. Sample points should be picked to represent widely used local capacitances and input slopes, which depend on circuits and processes.

Based on our experiments, we adopted the linear model since it is simple, but reasonably accurate for test circuits. The types of library cells developed for our ASIC library include inverter, n-input nand/and, n-input nor/or, D flip-flip/latch, resettable D flip-flop/latch, and two-input multiplexer. The other cells such as AOI (and-or-inverter), OAI (or-and-inverter), n-input multiplexer, decoders, and cells with different characteristics (in area, delay and driving capability) are desirable for better performance of overall chip, and are left for future taks.

After a Synopsys technology file is prepared, it is compiled to generate a binary .db file, which can be used for synthesis. A VHDL file for each library cell can be generated from the .db file, and simulation library can be constructed by compiling VHDL files. It is important to note that the VHDL file has information on only the intrinsic delay of the cell, but not on the loading capacitance delay. Thus, SDF (Standard Delay Format) file that contains all the instances' pin-to-pin timing dependent on loading capacitance should be generated from the Synopsys database for the circuit under simulation.

In the circuit design phase, schematic and simulation properties such as transistor size are annotated to circuit elements. Although all the information can be retrieved from the layout of the cell, a schematic is more abstract and easier to analyze than the layout. In the layout design phase, layouts of all the cells are drawn not only for area, speed, and power consumption, but also for easy placement and routing. For example, if a pin of the cell is located on the routing grid, then the routing layer can reach easily to the pin. Also if the height of all the cells are the same or similar, then the routing area can be utilized effectively.

4.4 Discrete Cosine Transform (DCT)

JPEG is a lossy compression method for still images. The lossy compression algorithm operates in three successive stages, the 2-D (2-Dimensional) DCT (Discrete Cosine Transform), quantization, and lossless entropy encoding, as shown in Fig. 4.4. Implementation of VLSI chips for JPEG are reported as [59]-[61]. The image is partitioned into 8×8 blocks, and the 2-D DCT is applied to each block. Then the transformed image is quantized and encoded. The key component in the compression process is a mathematical transformation known as the DCT algorithm.

Figure 4.4 JPEG system with DCT

Since DCT is a separable transform, the 2-D DCT can be performed by two separate 1-D DCT processes, one 1-D DCT in row direction and another 1-D DCT in column direction. Several implementations adopted this method for its simple structure and reduced hardware complexity [62]-[65]. However, this method requires a transposition memory to store and to rearrange the sequence of data for the second 1-D DCT, which incurs degradation of the performance. A different 2-D DCT implementation in [66] eliminates the need for the transposition memory to result in higher performance, but it increases the system complexity.

The former approach has been used in our implementation for the 2-D DCT, which performs in three steps, 1-D DCT, a memory transposition followed by another 1-D DCT. The formula for DCT is defined as follows:

$$X(0) = \sqrt{\frac{1}{N}} \cdot \sum_{n=0}^{N-1} x(n) \tag{4.1}$$

$$X(k) = \sqrt{\frac{2}{N}} \cdot \sum_{n=0}^{N-1} x(n) \cdot \cos\left(\frac{(2n+1)k\pi}{2N} \right) \text{ for } k = 1,2, \ldots, N, \tag{4.2}$$

where $x(n)$ represents each pixel value, $X(k)$ represents transformed value, and N is 8 for the 8×8 pixel size. The 2-D DCT for an 8×8 image with is described in the following VHDL code:

```
 for i in 0 to 7 loop
   for j in 0 to 7 loop
         temp := (others => '0');
         for k in 0 to 7 loop
             temp := temp + (c(8 * i + k) * x(8 * k + j));
          end loop;
          t(8 * i + j) := temp(32 downto 16);
   end loop;
end loop;

for i in 0 to 7 loop
  for j in 0 to 7 loop
        temp := (others => '0');
        for k in 0 to 7 loop
             temp := temp + (c(8 * i + k) * t(8 * k + j));
        end loop;
        o(8 * i + j) := temp(32 downto 16);
   end loop;
end loop;
```

We used the above behavioral-level VHDL description for the verification of the function. The simulation results were compared with those of a code written in C. After the high-level verification was done, the DCT algorithm was described in Silage and high-level synthesis tools, HYPER and ADVBIST_h, were applied to generate a RT-level circuit netlist.

4.4.1 Fast DCT Algorithm

It is possible to improve the original DCT algorithm described above, so that it has fewer operations to increase the speed and to reduce the circuit complexity [46]-[48]. We describe the method proposed by Chen et al. in this subsection [46]. Chen et al.'s method is known to be one of fastest DCT algorithms and is partly considered in our implementation. An N-point 1-D DCT algorithm involves $N \times N$ matrix calculations. By inspection, the $N \times N$ matrix calculations in Eqs. (4.1) and (4.2) can be decomposed into following two equations for N = 8 [46][64]. The decomposition reduces the number of multiplications from N^2 (=64) to $N^2/2$ (=32). It is important to note that the decomposition produces the same result as the original formula defined in (4.1) and (4.2).

$$\begin{bmatrix} X(0) \\ X(2) \\ X(4) \\ X(6) \end{bmatrix} = \begin{bmatrix} a & a & a & a \\ c & f & -f & -c \\ a & -a & -a & a \\ f & -c & c & -f \end{bmatrix} \begin{bmatrix} x(0)+x(7) \\ x(1)+x(6) \\ x(2)+x(5) \\ x(3)+x(4) \end{bmatrix} \tag{4.3}$$

$$\begin{bmatrix} X(1) \\ X(3) \\ X(5) \\ X(7) \end{bmatrix} = \begin{bmatrix} b & d & e & g \\ d & -g & -b & -e \\ e & -b & g & d \\ g & -e & d & -b \end{bmatrix} \begin{bmatrix} x(0)-x(7) \\ x(1)-x(6) \\ x(2)-x(5) \\ x(3)-x(4) \end{bmatrix} \tag{4.4}$$

$$\text{where } \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \end{bmatrix} = \sqrt{\frac{2}{N}} \begin{bmatrix} \cos\frac{\pi}{4} \\ \cos\frac{\pi}{16} \\ \cos\frac{\pi}{8} \\ \cos\frac{3\pi}{16} \\ \cos\frac{5\pi}{16} \\ \cos\frac{3\pi}{8} \\ \cos\frac{7\pi}{16} \end{bmatrix} = \begin{bmatrix} 0.35355339 \\ 0.49039264 \\ 0.46193976 \\ 0.41573480 \\ 0.27778511 \\ 0.19134171 \\ 0.09754516 \end{bmatrix}.$$

The number of multiplications required for the above equations is 32 for the 8-point DCT. If the two-level multiply-add structure is converted to a multiple-level one by rearranging operations, the number of multiplications is further reduced to 16.

However, the above fast DCT algorithm requires a matched inverse DCT (IDCT) algorithm. That is, if data is compressed using the general 1-D DCT algorithm defined in Eq. (4.1) and (4.2), the fast IDCT algorithms cannot be used for decompression. For this reason, the number of operations is minimized in our implementation as long as it produces the same transformed sequence of the original DCT as specified in Eq. (4.1) and (4.2). We proposed the data flow graph shown in Fig 4.5, which shows the DCT algorithm employed in our implementation. It has the same structure as that for Chen's algorithm only for even-indexed terms that requires eight multiplications. However, it does not reduce the number of multiplications for odd-indexed terms that is 16 as specified in Eq. (4.4). The overall number of multiplications for necessary for our implementation is 24, whereas it is 16 for Chen's algorithm.

We described the algorithm (whose data flow graph shown in Fig 4.5), in Silage and then applied HYPER for scheduling and module assignment. Our tool, ADVBIST_h (which implements the proposed region-wise ILP approach), is applied to the resultant data flow graph to assign BIST register. A data path and a controller were described in VHDL based on the BIST design.

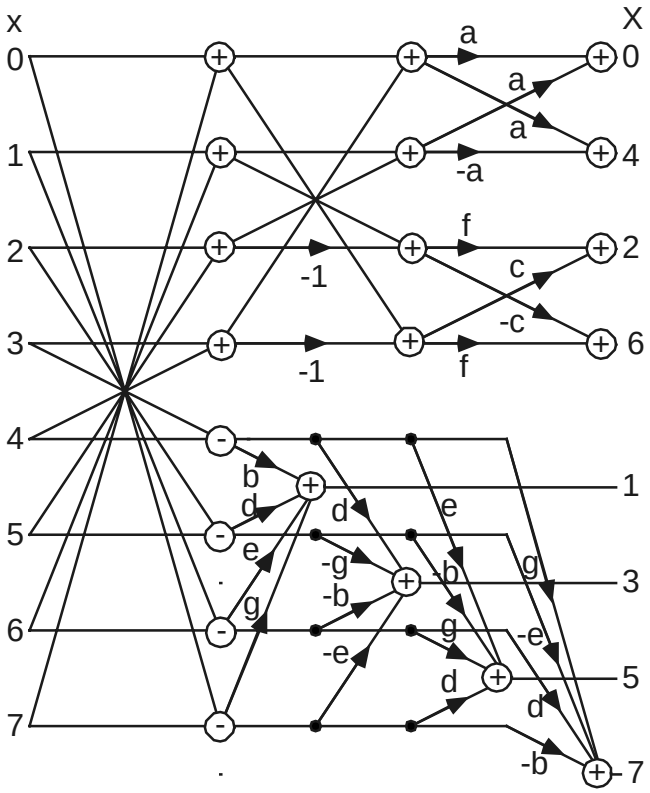


Fig. 4.5 Data flow graph for 8-point DCT

4.4.2 Wordlength and Number Representation

The bit width of a signal (in VHDL) affects the chip area, so the word length should be chosen carefully to balance accuracy and cost. As is for other implementations such as [63], the wordlength of 16 bits is chosen for the data path of our 1-D DCT circuit. When a multiplication is performed, the output bit width is double the input bit width. Thus, after a multiplication is performed, the 32-bit output is truncated to 16 bits in our implementation.

As common in other DSP applications, DCT requires computation of real numbers. We adopted the 2's complement number system, and used predefined data type, SIGNED. An IEEE package "IEEE.std_logic_arith.all" offers the data type and arithmetic operations such as +, -, *, and /. The following VHDL code shows arithmetic operations in real numbers.

```
add80out <= mux18out + mux13out;
sub80out <= mux11out - mux14out;
constant cosa: SIGNEDV := CONV_SIGNED(23170, 16); -- 0.35355339
constant cosb: SIGNEDV := CONV_SIGNED(32139, 16); -- 0.49039264
```

The conversion function CONV_SIGNED converts some types of numbers such as integer and std_logic_vector to SIGNEDV type.

The DCT also involves multiplication of coefficient constants in the range of 1 and -1. The coefficients are scaled up by multiplying 2^{15} and are converted into intergers before multiplications are performed. The results are truncated to 16-bits as mentioned earlier and scaled down by 2^{15}.

4.5 Architecture and Operation of the 2-D DCT Circuit

Fig. 4.6 shows the architecture for the data path and the controller. The multiplexer-based architecture is used for the data path, and a finite state machine (FSM) is used for the controller. The system is clocked by one external system clock.

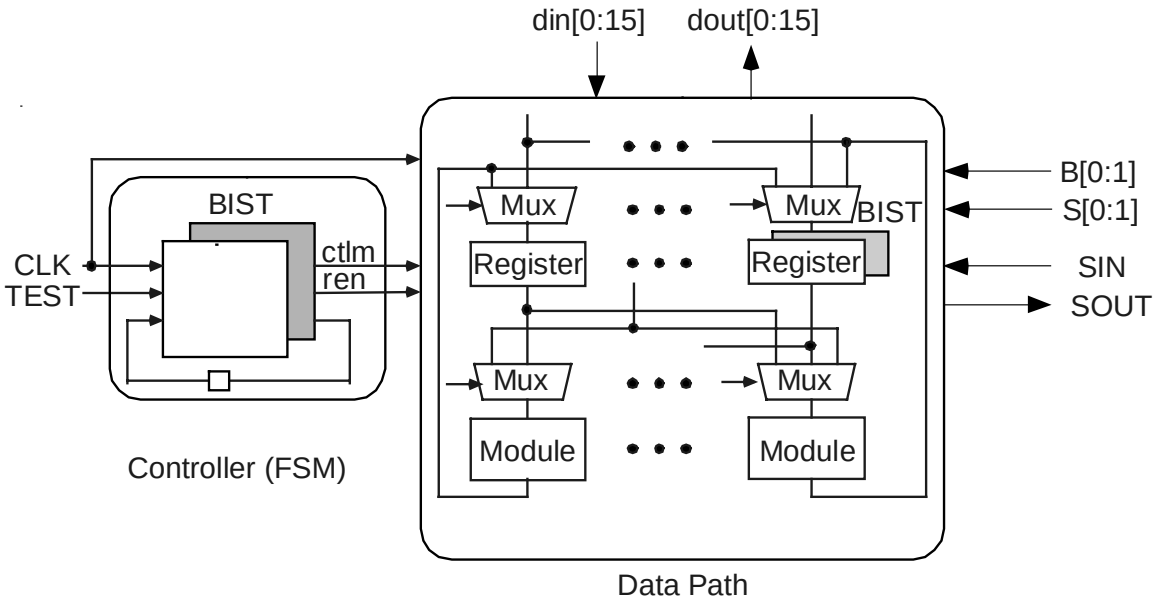


Figure 4.6 Target data data/controller architecture for 1-D DCT

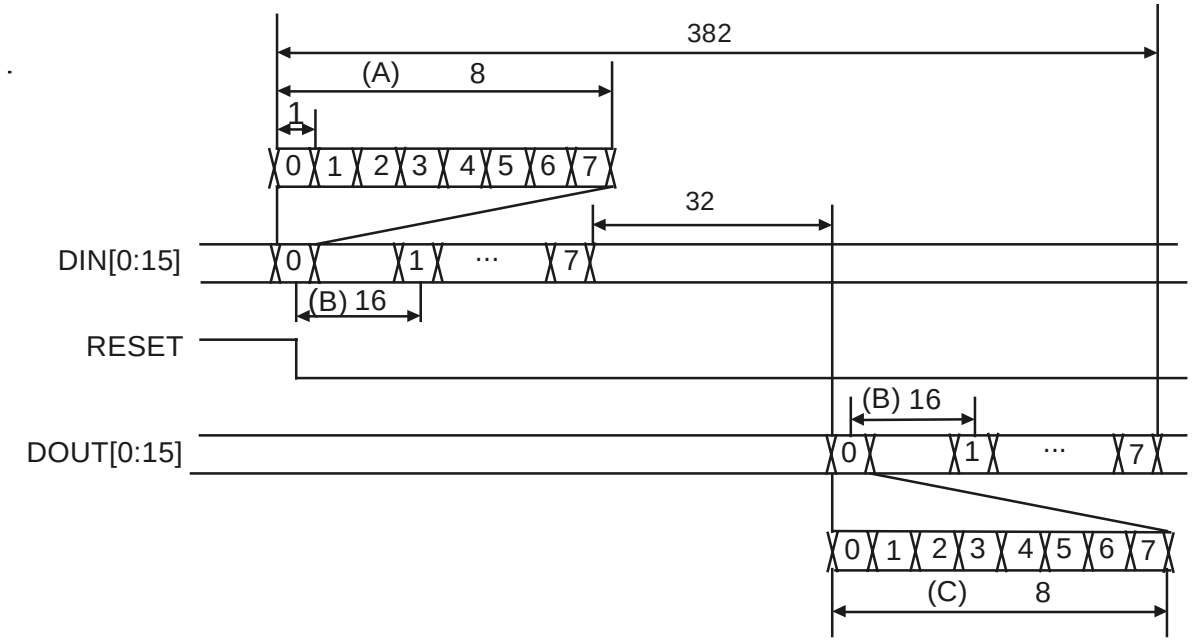
Table 4.1 lists in/out port of our 8×8 2-D DCT circuit.

Table 4.1. In/out port description

Name	Direction	Function
CLK	Input	System clock
RESET	Input	Initialize system
DIN[0:15]	Input	Data input
TEST	Input	Used when testing and scan in/out
B[0:1]	Input	To specify operation for normal, scan, BIST
S[0:1]	Input	To specify sub-test session
SI	Input	Scan in
SO	Output	Scan out
DOUT[0:15]	Output	Data output

4.5.1 Overall Operations

When the reset signal becomes inactive, the system starts its operation. Each row of an 8×8 image block is loaded serially through DIN port. A pixel of 16-bit long is loaded on every clock, and it takes 8 clocks to load 8 pixels. A serial-to-parallel converter makes the data available to the 1-D DCT circuit in parallel. It takes 16 clock cycles (which comprised of 15 control steps in the data flow graph and one additional clock cycle) to perform the 1-D DCT operation on a row. After completion of processing eight rows in 128 clocks, the 1-D DCT operation is performed on columns. The result of the 1-D DCT operation on each column is outputted serially to DOUT port after the parallel-to-serial conversion. The first output data is available after 210 clock cycles and the last output data after 392 clock cycles. The timing diagram of external ports is shown in Fig. 4.7.



(A): serial-to-parallel conversion for input
(B): 1-D DCT operation
(C): parallel-to-serial conversion for output

* the number with an arrow denotes the number of clock cycles

Figure 4.7. Timing diagram for external in/out port

A schematic diagram of our 2-D DCT is shown in Fig. 4.8. The FSM (finite state machine) controller generates six control signals described in the Table 4.2. The transposition memory transposes rows and columns, whose operation is explained in the following sections. The roles of the other blocks are self-explanatory.

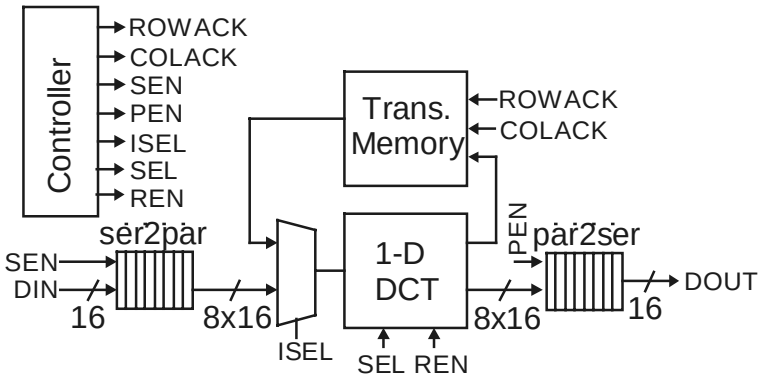


Figure 4.8. Schematic of 2-D DCT

Table 4.2. Functionality of internally generated control signals

Signal Name	Functionality
sen	Enables serial-to-parallel module
pen	Enables parallel-to-serial module
rowack	Enables the transposition memory to move the data vertically
colack	Enables the transposition memory to move the data horizontally
isel	Selects input signal of 1-D DCT module
sel	Selects input signal of multiplexers in the data path
ren	Enables registers in the data path module

4.5.2 Row 1-D DCT Operation

Fig. 4.9 shows a timing diagram for row 1-D DCT operation. After the reset is inactive, the first row of eight data is filled into the serial-to-parallel module in 8 clocks. The 1-D DCT module loads the eight data, and performs the 1-D DCT operations in 15 clocks. (Refer to “CTL

STATE” in Fig. 4.9.) The result is latched into the transposition memory on the following clock under ROWACK = 1. It repeats the same procedure for the remaining seven rows.

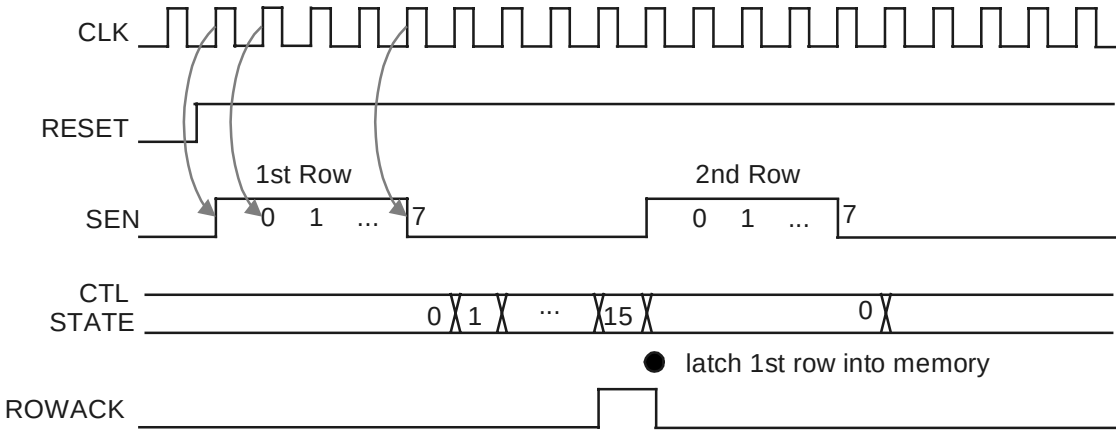


Figure 4.9. Timing diagram of row 1-D DCT

The transposition memory is an 8×8 register array with the data width of 16 bits and is shown in Fig. 4.10. Two control signals, ROWACK and COLACK, determine the direction of data movement. When ROWACK is active, a register (i,j) copies from the register (i-1, j). The data move vertically toward downward. When COLACK is active, a register (i,j) copies from the register (i, j-1). Hence, the data move horizontally from left to right.

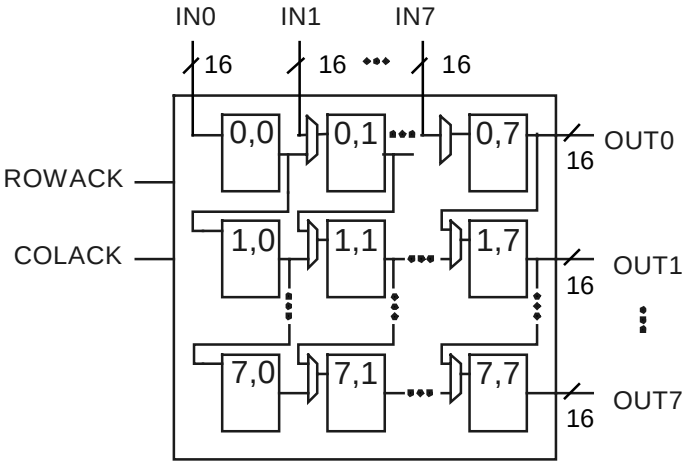


Figure 4.10. Schematic of the transposition memory

4.5.3 Column 1-D DCT Operation

Fig. 4.11 shows timing diagram for the late stage of the row 1-D DCT operation and the early stage of the column 1-D DCT operation. After performing the 1-D DCT on the last row, the results are stored at the transposition memory under ROWACK signal is high. The transposition memory is filled up now, and the 1-D DCT is performed on each column of the memory. The 1-D DCT reads column 7 (the rightmost column) of the memory under ISEL (Input Select) = 1, and performs the 1-D DCT operation on the column. The results are loaded into the parallel-to-serial converter and shifted out serially in eight clocks under active PEN (Parallel enable) signal. Meanwhile, COLACK is active for one clock, and the data on the next column (column 6) are available on the memory output port. The timing diagram of the last part of the column 1-D DCT is shown in Fig. 4.12.

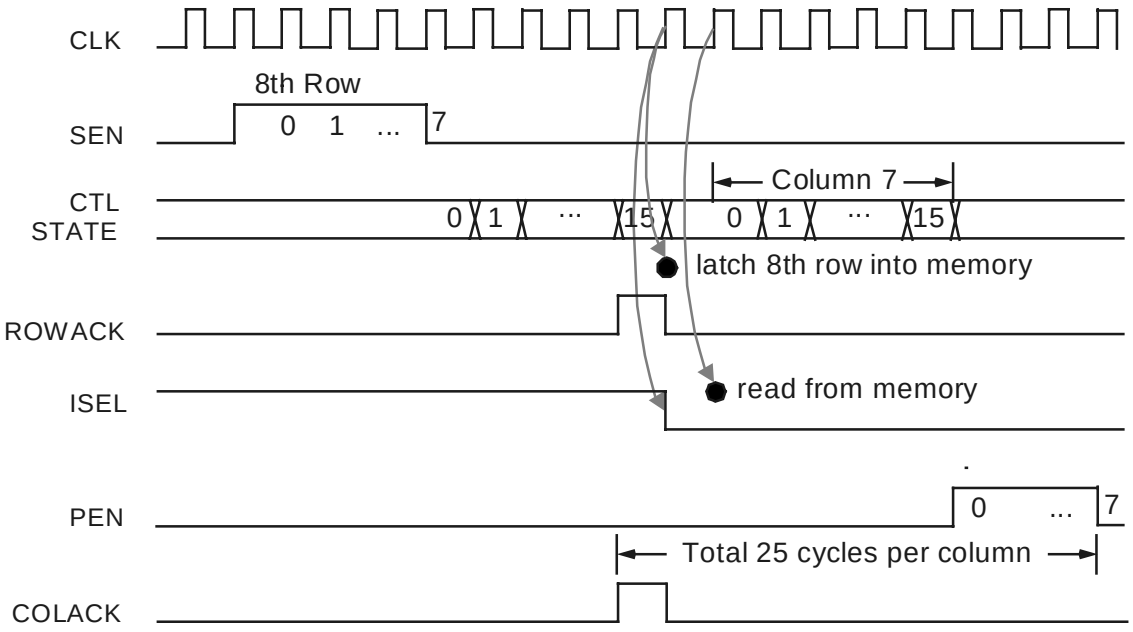


Figure 4.11. Timing diagram between row and column 1-D DCT

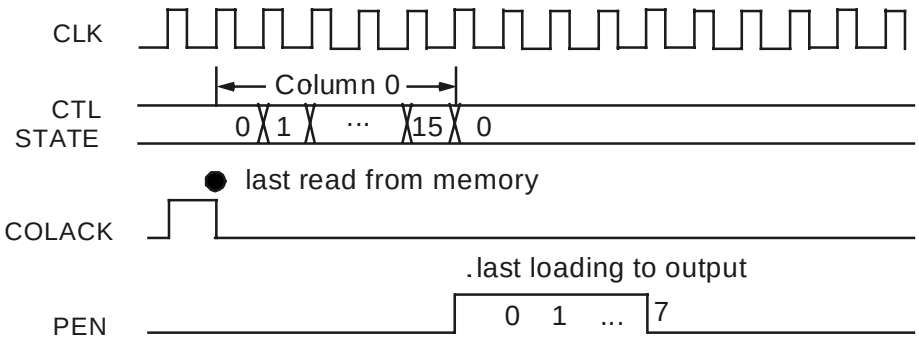


Figure 4.12. Timing diagram of the last part of column 1-D DCT

4.6 BIST Structure

The wordlength of the data path is 16 bits, and hence that for TPGs and SRs. A 16-bit TPG and a 16-bit multiple input signature register (short for signature register) with a connection polynomial, $P(x) = x^{16} + x^5 + x^3 + x^2 + 1$, are shown in Fig. 4.13 and Fig. 4.14, respectively.

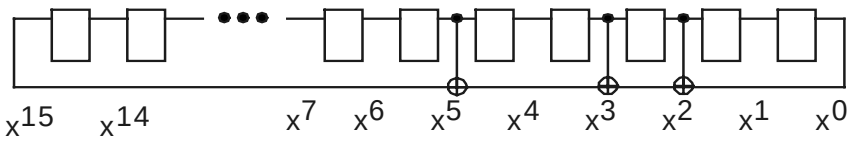


Figure 4.13 A schematic for 16-bit TPG with $P(x) = x^{16} + x^5 + x^3 + x^2 + 1$

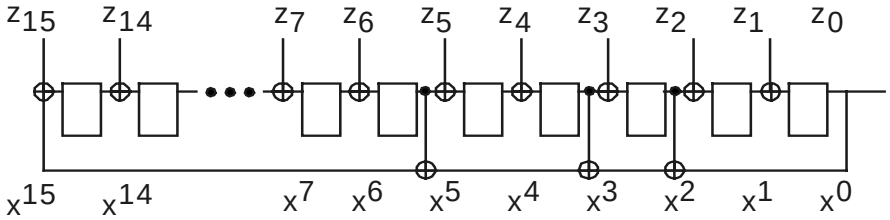


Figure 4.14 A schematic for 16-bit SR with $P(x) = x^{16} + x^5 + x^3 + x^2 + 1$

As existing registers are reconfigured as TPGs and SRs in our implementation, TPGs and SRs should behave as ordinary registers during normal operation. Further, scan operation is also necessary for TPGs and SRs for initialization and/or observation of signatures. Hence we used BILBOs in our implementation.

An original BILBO [28] and its four operating modes are given in Fig. 4.15 and Table 4.3, respectively. If two control signals B_0B_1 are 10, the BILBO can behave as a TPG or an SR. When it is to be used as a TPG, the inputs, Z_{15}-Z_0 , should be held at 0. To ensure it, we modified the original BILBO as shown in Fig. 4.16. The operation modes of the modified BILBO are given in Table 4.4. In summary, we use the original BILBO in Fig. 4.15 as an SR and the modified BILBO in Fig. 4.16 as TPG.

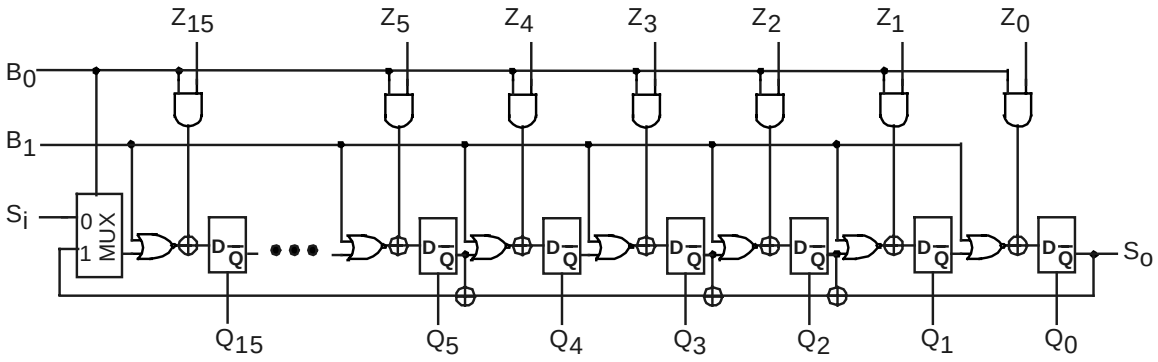


Figure 4.15 A schematic for the original 16-bit BILBO

Table 4.3 Operation modes of an original BILBO

B0	B1	Modes
1	1	Normal function
0	0	Shift register
1	0	TPG & SR
0	1	Reset

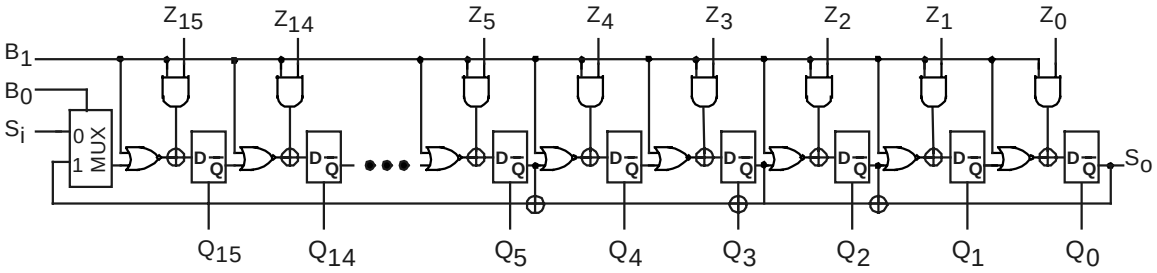


Figure 4.16 A schematic for a modified 16-bit BILBO

Table 4.4 Operation modes of the modified BILBO

B0	B1	Modes
1	1	Normal function
0	0	Shift register
1	0	TPG

Fig. 4.17 shows the BIST configurations for the testing of the DCT data paths. Only resources related to BIST are shown in the figure. All the BIST registers (four TPGs and two SRs) are connected through a scan chain to initialize LFSRs and to scan out signatures. Four test sessions are used for testing of six modules, and details about the testing are described in Chapter 5. For each test session, initial test patterns are scanned in, target module(s) are tested, and captured signatures are scanned out.

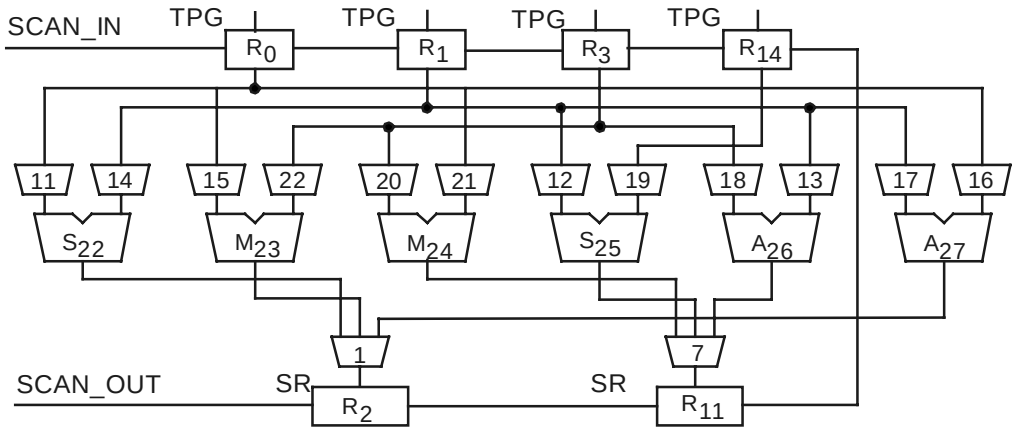


Figure 4.17 BIST configuration for testing of DCT data paths

The controller circuit consists of a 9-bit counter and associated combinational logic. The combinational logic receives the content of the counter and generates 66 control signals. The counter serves as a TPG (Test Pattern Generator) in our BIST circuit. Test responses on the control signals are compacted in space to 32 outputs and then compressed in time using two existing SRs in the data path module. Fig. 4.18 shows the structure of the controller after incorporation of BIST. When the circuit is in the controller testing mode, the counter counts up starting from 0. The CLB outputs, after compressed using XOR gates, are applied to the two SRs through multiplexers attached on the inputs of the two SRs.

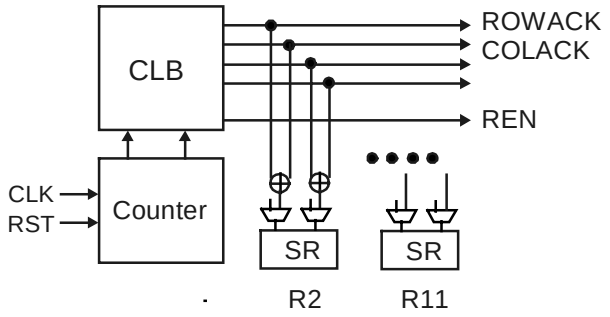


Figure 4.18. BIST structure of the controller

In the testing of the transposition memory, we intend to test stuck-at faults in the registers. The approach is to march through all-0-patterns to detect stuck-at 1 faults and then all-1-patterns to detect stuck-at 0 faults. In this testing, a register (i, j) copies from the register (i, j-1) on every clock and the registers on the rightmost column are copied into the *output registers*, which are scan registers in the data path module. The operation of the testing process is as follows. First, the test pattern 0000 (hex) is scanned into the eight registers on the first column of the transposition memory. Note that only registers in the first column of the memory are scannable. Next, eight clocks are applied to the module in the memory testing mode. The all-0-patterns march through the array and are captured in the *output registers*. After switching to the scan mode, the contents of the *output registers* are scanned out for examination. If the contents of all the *output registers* are all 0's, there is no stuck-at 1 fault in the array. The same process repeats for the test pattern FFFF (hex).

To test the serial-to-parallel (ser2par) module, we apply the test pattern 0000 (hex) to primary inputs INS(0:15) and apply eight clocks in the testing mode. Then the all-0-pattern marches through the eight 16-bit registers in the converter and is latched into a scan register in the data path module. The content of the register is scanned out for examination. Next, we apply the test pattern FFFF (hex) to INS and repeat the same process.

Testing of the parallel-to-serial (par2ser) module is similar. We scan in the test pattern 0000 (hex) into a scan register in the par2sel module and apply 7 clocks in the testing mode. If test responses on primary outputs OUTS(0:15) are all 0's, the module does not have any stuck-at 1 fault. Next, we scan in the test pattern FFFF (hex) to detect stuck-at 0 faults.

The necessary control signals for testing of each module are summarized in the Table 4.5. The testing is performed in the order shown in the table such that the data path (which has four sub-test sessions), the controller, the memory, the serial-to-parallel module, and the parallel-to-serial module.

Table 4.5. Test modes and control signals

TEST	B0	B1	B2	S0	S1	Description
0	1	1	0	-	-	Normal operation
1	0	0	0	-	-	Scan operation
1	1	0	0	0	0	Sub-test session 0
1	1	0	0	0	1	Sub-test session 1
1	1	0	0	1	0	Sub-test session 2
1	1	0	0	1	1	Sub-test session 3
1	1	0	1	0	0	Controller testing
1	1	1	1	0	1	Memory testing
1	1	1	1	1	0	Ser2par testing
1	1	1	1	1	1	Par2ser testing