COMMUNICATION CHIP - PROJECT DESCRIPTION

Design consists of CoreB block inside which there are 4 instances of CoreA blocks

Functional Clocks:
add_clocks clka -period 10ns
add_clocks clka -period 13.2ns
add_clocks clka -period 14ns
functional clock frequency: 400MHz
Scan Clock frequency: 25MHz

CoreA:
Int mode chains = 15
Length = 45 flops per chain
Ext mode chains = 5
Length = 27 flops per chain
Compression Ratio = 8x

<u>Using scan chain report and scan cell report:-</u>
Number of int mode chains = 15 (core logic flip flops of CoreA + wrapper flip flops of CoreA)
Number of flip flops in each chain = 45

Total Flops in Core A = 45 x 15 = 675 flops

There are 4 instances of Core A = 675 x 4 = 2700 flops

CoreB:
Int mode chains = 15
Length = 68 flops per chain
Ext mode chains = 5
Length = 28 flops per chain
Compression Ratio = 10x

<u>Using scan chain report and scan cell report:-</u>

No of int mode chains = 15 (CoreB core logic flops + CoreB wrapper flops + CoreA wrapper flops)
Number of flip flops in each chain = 68

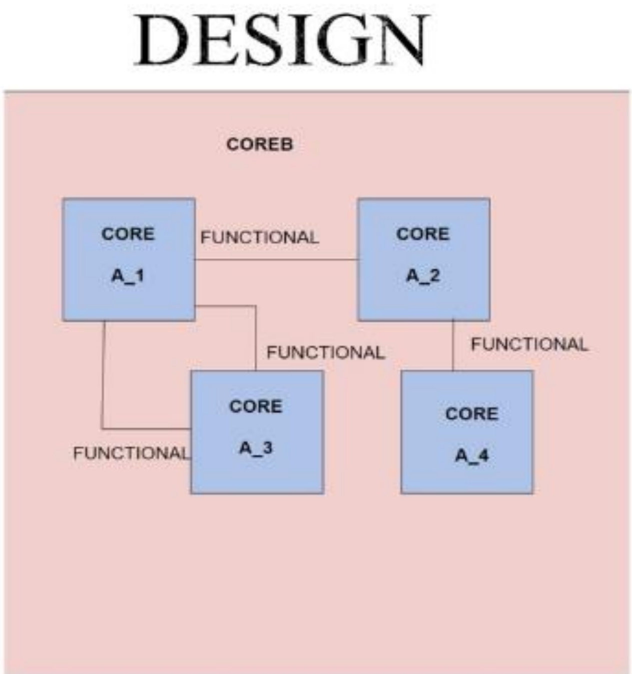
Total flops = 68 x 15 = 1020 (wrapper flops of CoreA)

Therefore, the wrapper flops of CoreA are already included in the above 2700 flops.

= 1020 **–** [4 **x** (No. of CoreA ext mode **x** Length of each chain)]
 ………. In ext mode, we have only wrappers
= 1020 – [4 x (5 x 27)]
= 1020 – [4 x 135]
= 480

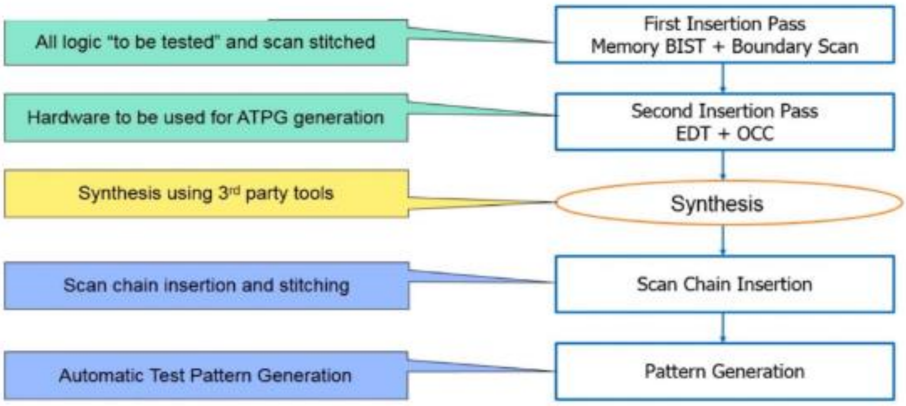
Total Flip Flops = 480 + 2700 = 3,180 flip flops

Tools:
MBIST: Tessent MBIST
Scan Insertion: Tessent
Scan EDT: Tessent TestKompress A
TPG: Tessent TestKompress
Simulation: QuestaSim



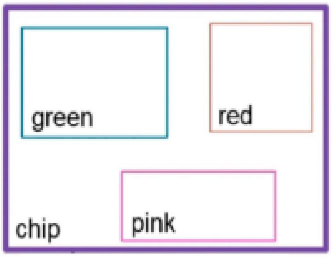
Pre-Scan DFT Insertion

DFT insertion is performed on pre-scan inserted netlist.



Pre Scan Insertion Flow for RTL

Flat vs Hierarchical

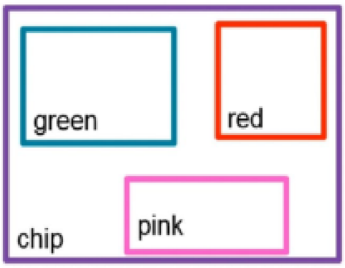


Flat

DFT insertion is done once at chip level.

Hierarchical

DFT insertion is performed bottom up for each of the green, red and pink hierarchical layout regions, and then DFT insertion is performed at the chip level.

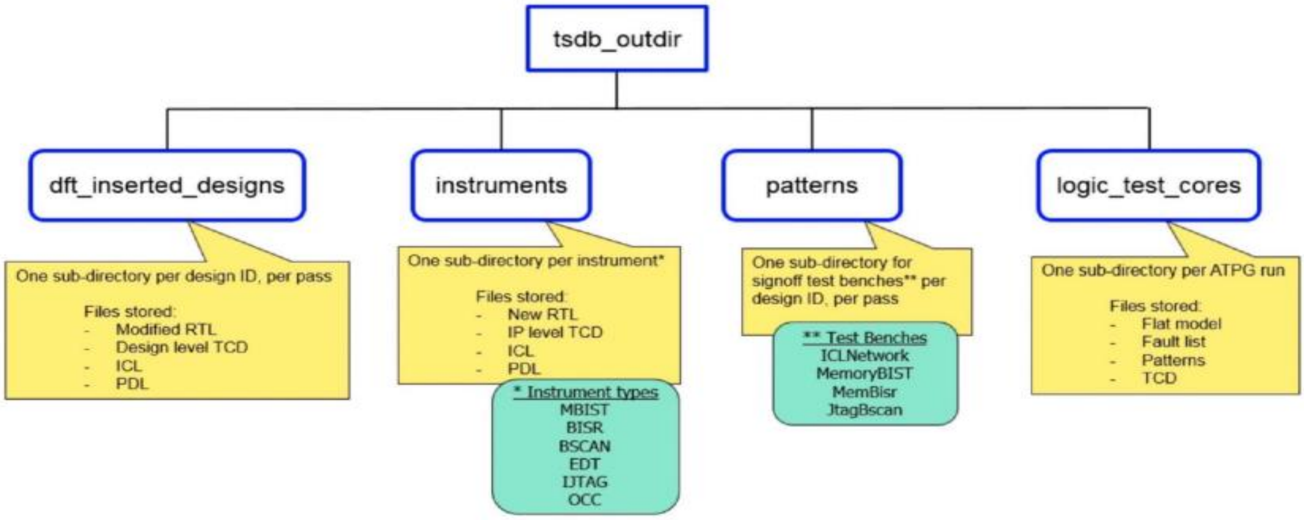


Tessent Shell DataBase (TSDB)

- It is a highly structured and efficient database.
- It includes data about everything that was created during the runs (modified RTL files, patterns, DFT Specifications, generated ICL and PDL files etc)

Default name : tsdb_outdir
Default location : current directory
The location of the tsdb_outdir can be changed by using the command ***set_tsdb_output_directory***.

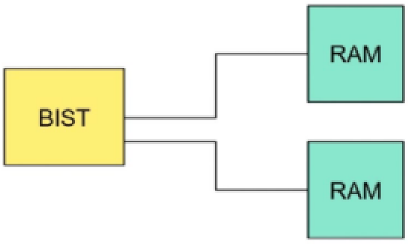
Structure of tsdb_outdir



TESSENT MBIST

Tessent Core Description file (TCD file)

- When testing memories with Tessent MBIST, we have a set of configurations (memory types, testing algorithms, port descriptions) and memory descriptions which we have to pass to the tool.



- In tessent shell, the description of main elements like memory library are presented to the tool in the form of TCD file.
- Extension : .tcd_mem_lib

.tcd_mem_lib file

```
MemoryTemplate (SYNC_1R1W_16x8) { // {{{

    MemoryType          : SRAM;  // PG mandatory
    CellName            : SYNC_1R1W_16x8;
    Algorithm           : SMarchCHKBvcd;
    BitGrouping         : 1;
    //ShadowWrite         : On;
    //ShadowWriteOK       : On;
    //ShadowRead          : On;
    //ReadOutofRangeOK    : On;
    TransparentMode     : SyncMux;
    //DataOutStage        : None;
    OperationSet        : Sync;
    LogicalPorts        : 1R1W;
```

Memory property section describes the physical characteristics of the memory

.tcd_mem_lib file (cont..)

```
Port (CLKR) {
    Function: Clock;
    Polarity: ActiveHigh;
    LogicalPort: R;
}
Port (CLKW) {
    Function: Clock;
    Polarity: ActiveHigh;
    LogicalPort: W;
}
Port (AR[3:0]) {
    Function: Address;
    LogicalPort: R;
}
Port (AW[3:0]) {
    Function: Address;
    LogicalPort: W;
}
Port (D[7:0]) {
    Function: Data;
    Direction: Input;
    LogicalPort: W;
}
```

Memory port section describes the ports of the memory and properties about the ports

.tcd_mem_lib file (cont..)

```
AddressCounter {
    Function ( ColumnAddress ) {
        LogicalAddressMap { ColumnAddress[2:0]:Address[2:0]; }
        CountRange[0:7];  // 8 col
    }
    Function ( RowAddress ) {
        LogicalAddressMap { RowAddress[0:0]:Address[3:3]; }
        CountRange[0:1];  // 2 row
    }
}
```

Specify the row and column bits segmentation within address bits

- There are 4 Address bit. The first 3 address bit [2:0] are mapped to columns. And the third address bit [3] is mapped to row.
- 3 column bits → 2**3 = 8 columns
- 1 row bit → 2**1 = 2 rows

- The functionality of the memory is available in the Verilog file (.v file)

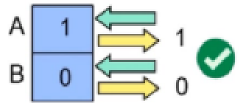
```
module SYNC_1R1W_16x8 (
    CLKR, CLKW,
    AR, AW,
    D,
    Q,
    RE, WE, GWE,
    OE
);

input         CLKR, CLKW;
input  [3:0] AR, AW;
input  [7:0] D;
output [7:0] Q;
input [7:0] GWE;
input         RE, WE;
input         OE;

reg [7:0] MEM [0:15];
reg [7:0] Q_REG;

assign Q = (OE) ? Q_REG : 8'bz;

always @ (posedge CLKW) begin
    if (WE) begin
      MEM[AW] <= (GWE & D) | (~GWE & MEM[AW]);
    end
end
wire [7:0] MEM_0, MEM_8;

always @ (posedge CLKR) begin
    if (RE) begin
      Q_REG  <= MEM[AR];
    end
```

Snapshot of SYNC_1R1W_16x8.v file

Test Algorithms

Memory test algorithms are very specific sequence of writing to and reading from memory cells.

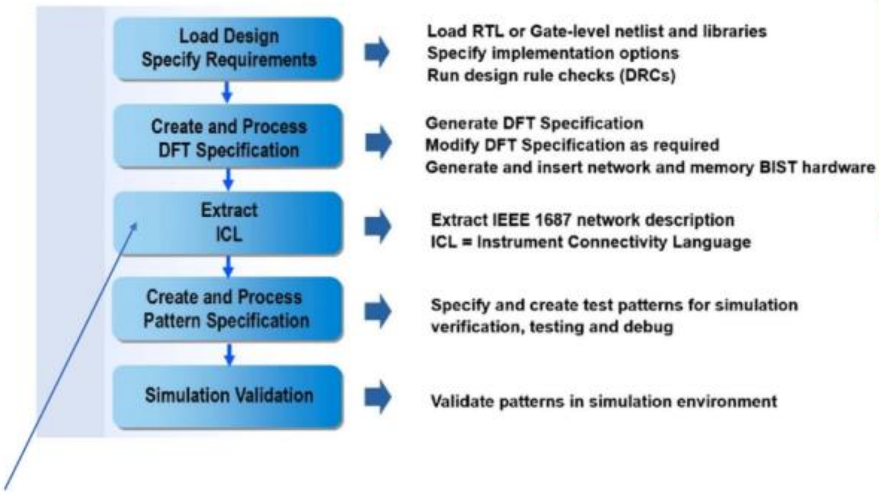
Example :



Writing 1 onto cell A and 0 onto cell B, if everything is OK, then we have to read 1 from cell A and 0 from cell B

In our Labs, SMarchCHKBvcd algorithm is used.

Tessent MBIST Flow



NOTE :
ICL is a language in which the IJTAG describes the interface for an instrument and the connections of these instruments.

Extract ICL step verifies the proper connectivity of the modules that were inserted in the previous step. With no DRCs detected, the top level ICL is extracted.

Inputs for MBIST

- Memory files (.v , .tcd_mem_lib)
- Library files (.tcelllib)
- Design file (.v)

Setting Context and Giving the inputs

```
### context "dft rtl" tell the tool to enter into RTL insertion mode, design_id tells to create a seperate directory in tsdb for this particular below insertion steps. -design_id <your custom
set_context dft -rtl -design_id first_insertion

## specify the output directory where you want to dump the outputs of mbist insertion
set_tsdb_output_directory ../tsdb_outdir

## provide the design files and library files
read_cell_library ../../library/adk.tcelllib

read_verilog ../../library/mems/SYNC_1R1W_16x8.v -exclude_from_file_dictionary -verbose

read_verilog ../design/corea.v -verbose

## Elaborate the design
set_current_design corea

# set the design level to either chip, physical_block or sub_block
set_design_level physical_block
```

Where DFT is Inserted?

COMMAND :
set_design_level

Physical Block :

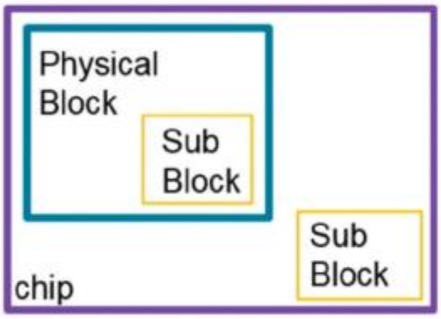
It is a layout region
Layout of this region is done separately and the module boundary will exists after the layout is done.

Sub Block :

The hierarchy of the sub block netlist may not exist after layout and gets merged with its parent.

Chip :

It is a layout region
Required ports for DFT need to exist and be connected to IO pads.



NOTE :
The design level will be decided by Design Team, DFT Team and the PD team together.

- Define all the clocks that are used by the memories.

add_clocks 0 clka -period 10ns
add_clocks 0 clkb -period 13.2ns

add_dft_signals

- add_dft_signal <signal_name> : It will insert TDR logic for specified signal.

> Note : TDR can be inserted only for static signals (Example TE)

Example :

tck_occ_en : A global DFT control signal that is used to enable the mini-OCC present inside the Sib(sti) node.

ltest_en : A logic test control signal that is used to enable the logic test mode. This signal is force high during all logic test modes.

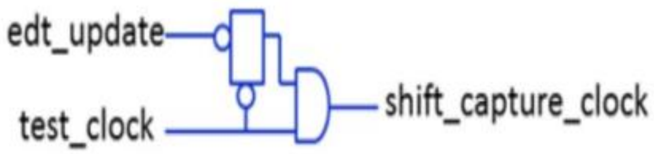
memory_bypass_en: To bypass memories. This signal is set to 1 by default during logic test

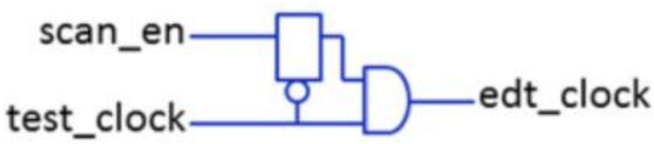
add_dft_signal <signal name> -source node <top_level_port name>

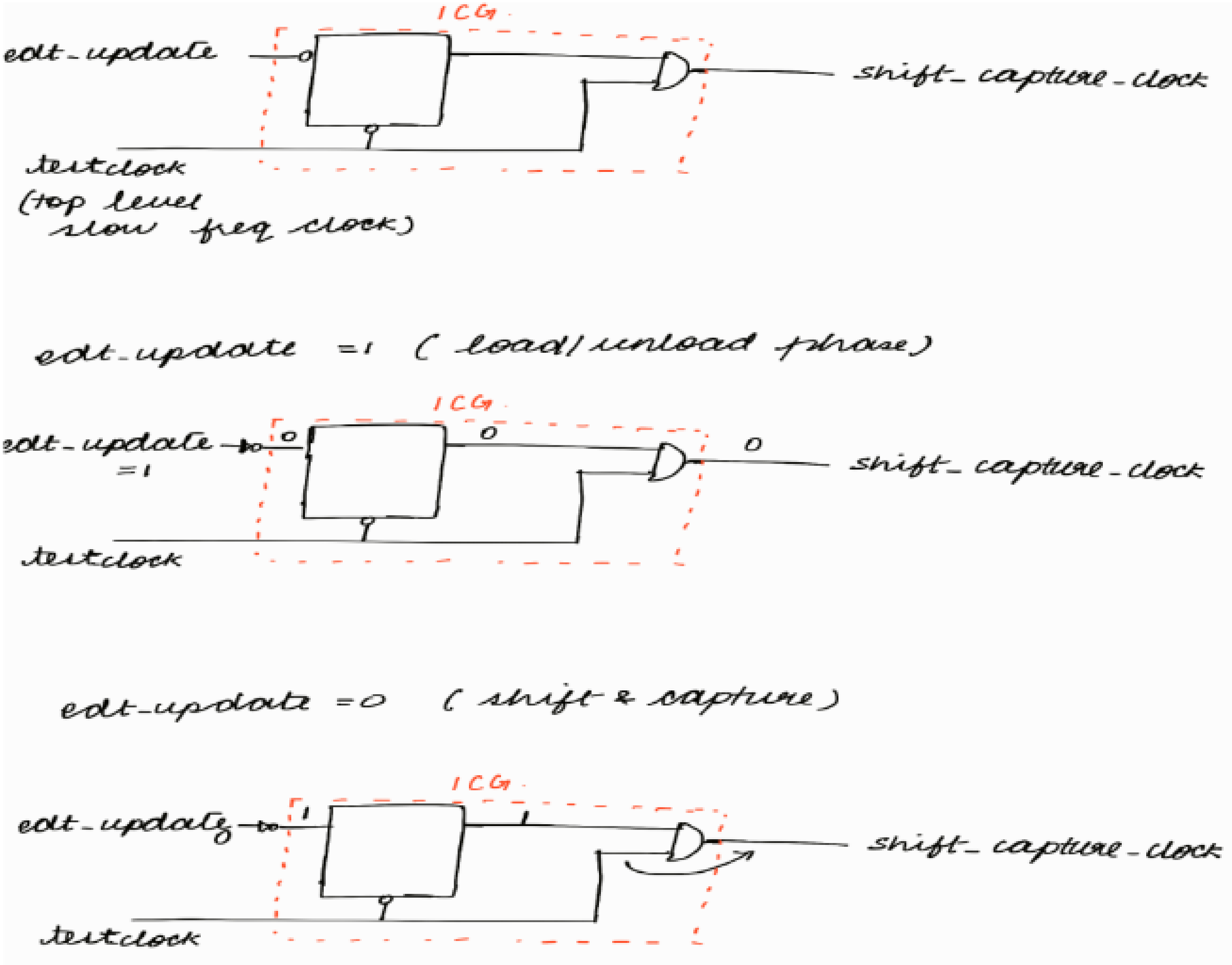
> For dynamic signals, we can't insert TDR. So it has to controlled from the top level.

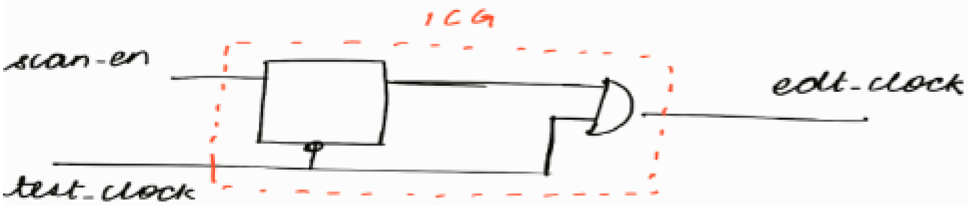
Command :

add_dft_signal edt_clock shift_capture_clock -create_from_other_signals

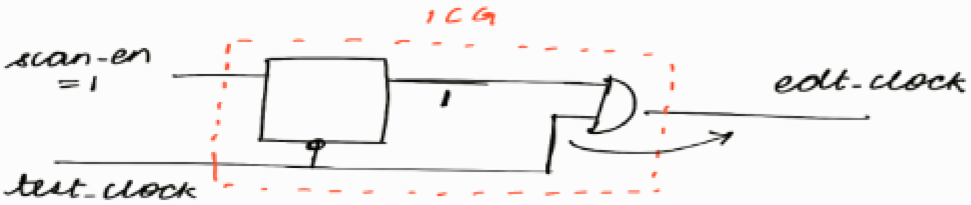




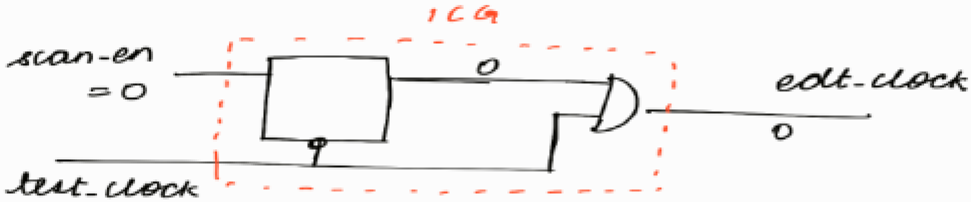

ICG
edt_update
shift_capture_clock
testclock
(top level slow freq clock)
edt_update =1 (load/unload phase)
ICG
edt_update =1
0
0
0
shift_capture_clock
testclock
edt_update =0 (shift & capture)
ICG
edt_update 0
1
1
shift_capture_clock
testclock

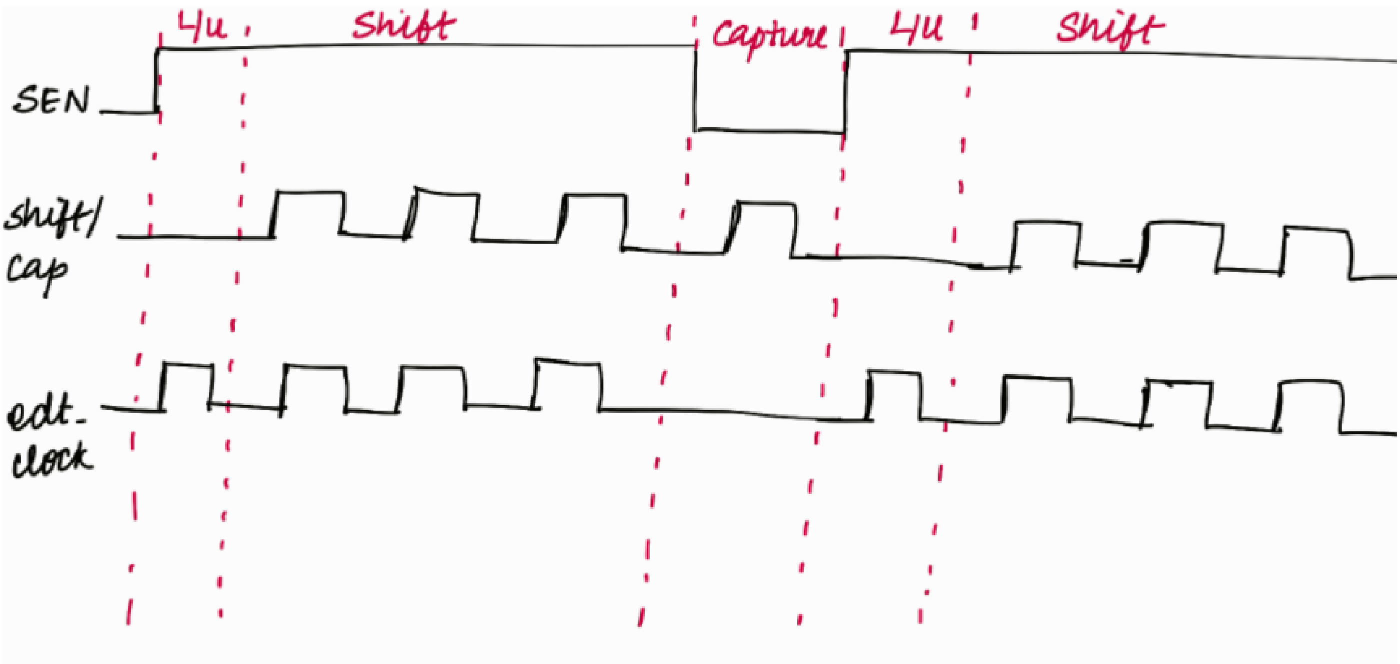


scan_en = 1 (during load/unload & shift phase)



scan_en = 0 (capture)





Specifying DFT Requirement

- Specifies the requirements to be checked during ***check_design_rules***.

Command : ***set_dft_specification_requirements -memory_test on***

- "-memory_test on" option is needed when implementing memory test.

- When memory_test on, memory_bist, memory_bisr_chains, and memory_bisr_controller default to auto; otherwise they are default to off.

- When memory_bist option is set to auto, if the current design contains memories, the MBIST pre-DFT DRC rules will be checked during *check_design_rules* command. The required specifications will also be included to the DFT specification when the command *create_dft_specification is executed*.

DRCs Check

- Once everything is defined check_design_rules command will run a DRC check on the current design.

DFT Specification

- The DFT Specification describes the test hardware that will be added to your design.
    - o IJTAG Network configuration
    - o Memory BIST partitioning/configuration

      Memory BIST partitioning :
        - Listing of Memory BIST controllers to be generated.
        - Clock domain associated with each memory BIST controller.
        - Memories assigned to each controller.

Creating a New DFT Specification

- A new DFT Specification can be created using the command ***create_dft_specification***
- This command uses information from prior settings :
    - set_design_level
    - Design netlist etc

Viewing a DFT Specification

Command : ***report_config_data***

```
DftSpecification(corea,first_insertion) {
  IjtagNetwork {
    HostScanInterface(ijtag) {
      Sib(sri) {
        Attributes {
          tessent_dft_function : scan_resource_instrument_host;
        }
        Tdr(sri_ctrl) {
          Attributes {
            tessent_dft_function : scan_resource_instrument_dft_control;
          }
        }
      }
      Sib(sti) {
        Attributes {
          tessent_dft_function : scan_tested_instrument_host;
        }
        Sib(mbist) {
        }
      }
    }
  }
```

IJTAG Network

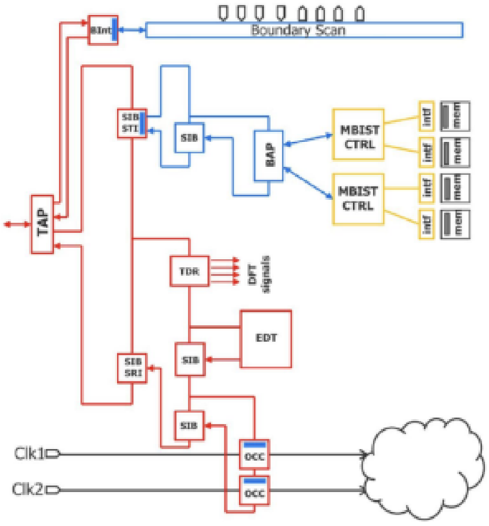
Viewing a DFT Specification (Cont…)

```
MemoryBist {
  ijtag_host_interface : Sib(mbist);
  Controller(c1) {
    clock_domain_label : clka;
    Step {
      MemoryInterface(m1) {
        instance_name : ram1;
      }
    }
  }
  Controller(c2) {
    clock_domain_label : clkb;
    Step {
      MemoryInterface(m1) {
        instance_name : ram2;
      }
    }
  }
}
}
```

Memory BIST

Here the clocks of the 2 memories are different.
So they are controlled by 2 different controllers.

Diagrammatic Representation of DFT Specification



- A SIB is a special node in IJTAG that acts as a switch.
- Instruments that needs to be active during scan (EDT OCC) are inserted under 1 SIB and the ones that are scan tested such as MBIST controller are inserted under another SIB.

Types of SIBS

<u>Scan Tested Instrument (STI)</u> : The SIB STI provides access to the IJTAG network for MBIST controller (in this fig).

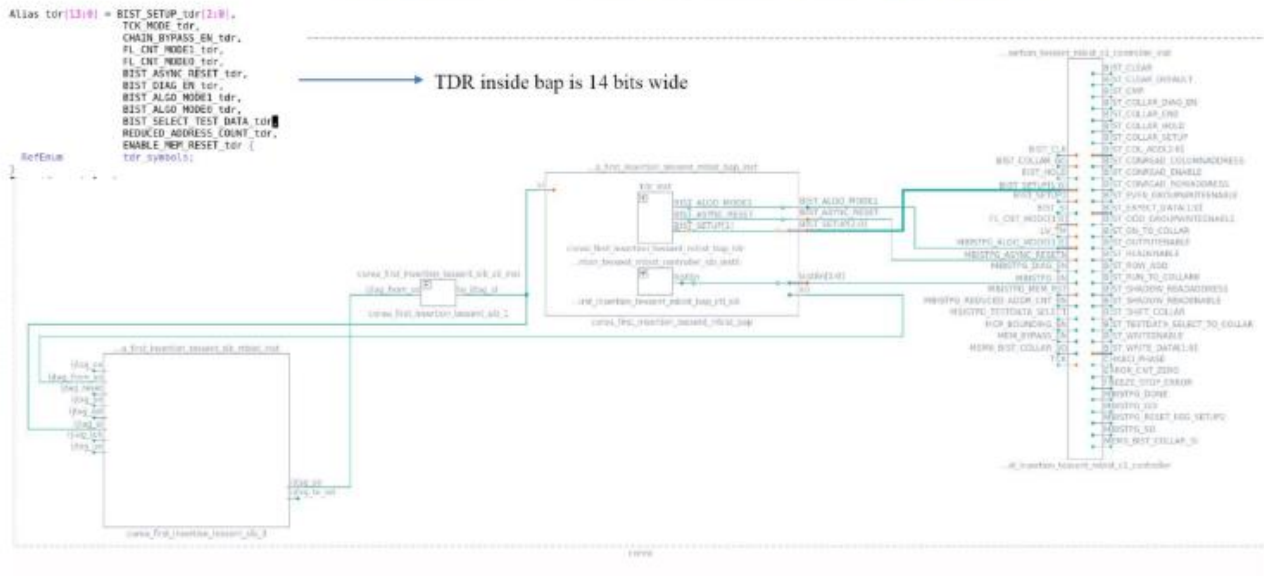
<u>Scan Resource Instrument (SRI)</u> : The SIB SRI provides access to the IJTAG network for logic instruments (EDT, OCC)

Generating and Inserting the Hardware

Command : ***process_dft_specification***

- Validates the DFT Specification
- Generates hardware : MBIST related controllers, memory interfaces, IJTAG network
    - RTL and ICL descriptions
- Edits design and inserts generated hardware
- Generates SDC constraints
- Generated files are written into TSDB directory

IJTAG Connections in MBIST



Extract ICL

- Generates the ICL file for the specified design.

ICL Extraction Process :

- Automated generation of the interconnection information of the various IJTAG building blocks (SIBs, TDRs etc)
- Tessent Instrument ICL files are created during ***process_dft_specification***
- Extract ICL process verifies the proper connectivity of the ICL modules that were inserted during the process_dft_specification command.
- ICL extraction must pass with no violations in order to generate the test patterns.

Pattern Specification

- ***create_patterns_specification*** is used to generate a patterns specification.
- Validation and processing of the specification is done using ***process_patterns_specification***.

Syntax of create_pattern_specification

create_pattern_specification [usage] [-replace]

Usage :

- Specifies the intended usage of the patterns to be either signoff or manufacturing
  - **Signoff :** Indicates that simulation signoff patterns are requested.
  - **Manufacturing :** Indicates the manufacturing patterns are requested.

- **Signoff patterns** : to verify each instrument in the design using a Verilog simulation testbench. They are used for simulation and verification of the design.
- **Manufacturing patterns :** patterns generated for ATE

Replace :

Allows the PatternSpecification Wrapper to be replaced by new one.

The default patterns specification is created and stored in a TCL variable called spec1

Command : ***set spec1 [create_patterns_spec]***

Using the command ***report_config_data $spec1***, the pattern spec is displayed in the session window.

Pattern Specification

A pattern specification for a MBIST-only design typically consists of :

One ICL verification pattern :

This pattern verifies the ICL description is correct.

One or multiple MBIST patterns : which exercise implemented MBIST controllers by clocking the design with appropriate clocks and instruct every MBIST controller to launch a memory test.

```
Patterns(ICLNetwork) {
  ICLNetworkVerify(corea) {
  }
}
```

```
Patterns(MemoryBist_P1) {
  ClockPeriods {
    clkb : 13.2ns;
    clka : 10.0ns;
  }
  TestStep(run_time_prog) {
    MemoryBist {
      run_mode : run_time_prog;
      reduced_address_count : on;
      Controller(corea_first_insertion_tessent_mbist_c1_controller_inst) {
        DiagnosisOptions {
          compare_go : on;
          compare_go_id : on;
        }
      }
      Controller(corea_first_insertion_tessent_mbist_c2_controller_inst) {
        DiagnosisOptions {
          compare_go : on;
          compare_go_id : on;
        }
      }
    }
  }
}
```

Pattern Specification (Contd…)

reduced_address_count : on ; → Enables the MBIST controller to run on 4 corners of the common memory address space.

This is useful to check the proper functionality of the BIST controller without having to simulate the test for the entire memory space.

This is reduce the run time.

```
TestStep(run_time_prog) {
  MemoryBist {
    run_mode : run_time_prog;
    reduced_address_count : on;
    Controller(corea_first_insertion_tessent_mbist_c1_controller_inst) {
      DiagnosisOptions {
        compare_go : on;
        compare_go_id : on;
      }
    }
```

Pattern Specification (Contd..)

GO bit indicates whether the controller has passed/failed the simulation.
It indicates the comparator's pass/fail status.

When the test starts, the GO bit starts high and falls when a comparator fails (BIST controller received erroneous response from the memory) and stays low till the end of the test.

At the end of the test,

if GO is 1'b1, it indicates no failing memories
if GO is 1'b0, it indicates detection of failing memories.

```
TestStep(run_time_prog) {
  MemoryBist {
    run_mode : run_time_prog;
    reduced_address_count : on;
    Controller(corea_first_insertion_tessent_mbist_c1_controller_inst) {
      DiagnosisOptions {
        compare_go : on;
        compare_go_id : on;
      }
    }
```

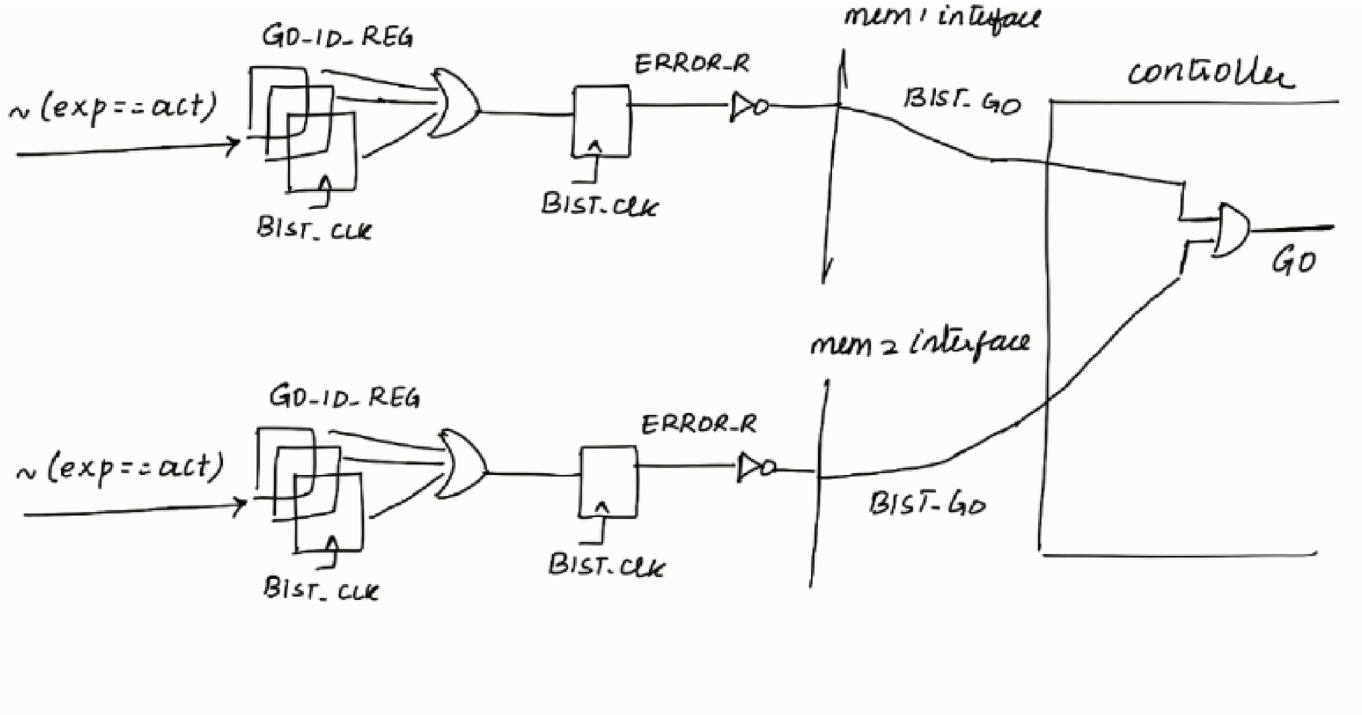
If compare_go is on, then it will check the GO bits.

Pattern Specification (Contd..)

When compare_go_id is set to on, we can directly get from the simulation logfile the failing controllers, the ICL and design instance of the memory interface and go_id_reg of the failing memory.

When compare_go_id is set to off, the only information which we get from the simulation logfile is the failing controller.

```
TestStep(run_time_prog) {
  MemoryBist {
    run_mode : run_time_prog;
    reduced_address_count : on;
    Controller(corea_first_insertion_tessent_mbist_c1_controller_inst) {
      DiagnosisOptions {
        compare_go : on;
        compare_go_id : on;
      }
    }
```



Running Testbench Simulations

Command : ***run_testbench_simulations***

- This command compiles and simulates the testbenches located in the TSDB which was generated by the command ***process_pattern_specification***.
- The information on how to load the design into the simulator is contained in **.design_source_dictionary file**.
- We have to specify the location of the simulation library modules using the command ***set_simulation_library_sources***

  Example : ***set_simulation_library_sources -v ../../library/adk.v -y ../../library/mems -extension v***

RETENTION TEST

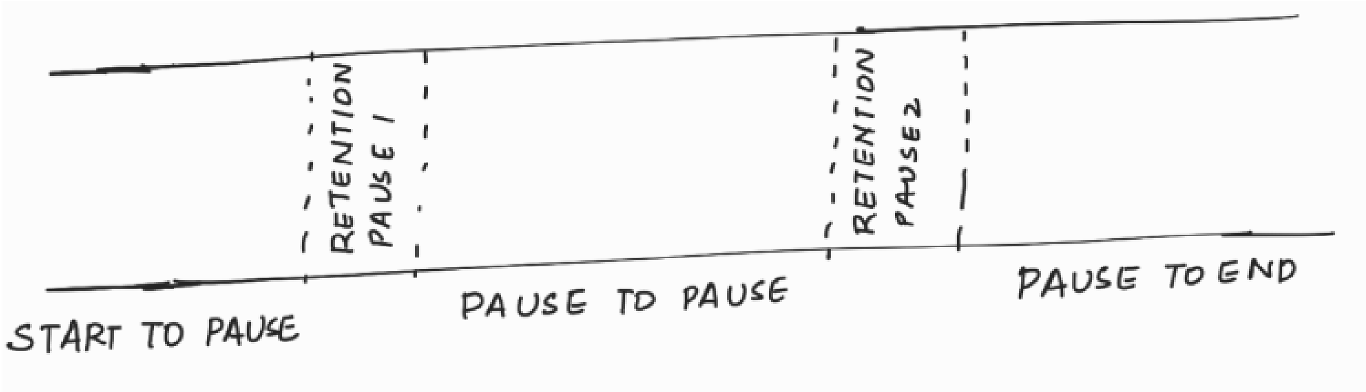
Retention Test is done to check if the memory cells are able to retain the value for some time.
So instead of reading immediately after a cell is written, we pause it for some time and then read the memory.

Step	Event	Library Algorithm Phase Execution
1	Load checkerboard background	start_to_pause
2	Apply first retention pause	n/a
3	Read checkerboard background. Load inverse checkerboard background.	pause_to_pause
4	Apply second retention pause	n/a
5	Read inverse checkerboard background	pause_to_end

```
Patterns (prt) {
  ClockPeriods {
    clka : 10ns;
  }
  TestStep(write_ckb) {
    MemoryBist {
      access_protocol : parallel;
      run_mode : hw_default;
      AdvancedOptions {
        retention_test_phase : start_to_pause; //sys_retention_test_phase= 01
      }
      Controller(chip_rtl_tessent_mbist_c1_controller_inst) {
        DiagnosisOptions {
          compare_go : on;
        }
        AdvancedOptions {
          apply_algorithm : SMarchCHKBvcd;
        }
      }
    }
  }
```

```
ProcedureStep(pause1) {
  wait_time : 500ns;
}
TestStep(read_ckb_write_inv_ckb) {
  MemoryBist {
    access_protocol : parallel;
    run_mode : hw_default;
    AdvancedOptions {
      retention_test_phase : pause_to_pause; //sys_retention_test_phase= 10
    }
    Controller(chip_rtl_tessent_mbist_c1_controller_inst) {
      DiagnosisOptions {
        compare_go : on;
      }
      AdvancedOptions {
        apply_algorithm : SMarchCHKBvcd;
      }
    }
  }
```

```
ProcedureStep(pause2) {
  wait_time : 500ns;
}
TestStep(read_inv_ckb) {
  MemoryBist {
    access_protocol : parallel;
    run_mode : hw_default;
    AdvancedOptions {
      retention_test_phase : pause_to_end; //sys_retention_test_phase= 11
    }
    Controller(chip_rtl_tessent_mbist_c1_controller_inst) {
      DiagnosisOptions {
        compare_go : on;
      }
      AdvancedOptions {
        apply_algorithm : SMarchCHKBvcd;
      }
    }
  }
}
}
```



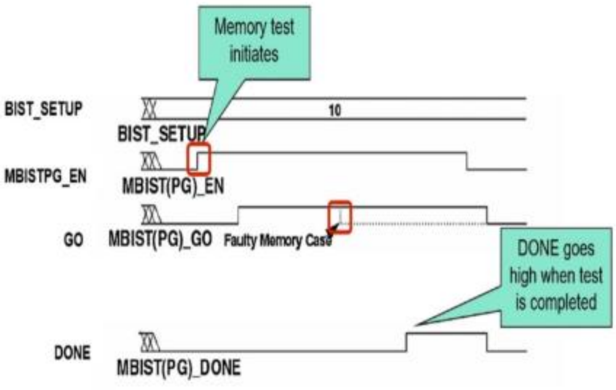
Instrument Connectivity Language (ICL)

- Language used to describe test instruments and their access network.
- ICL is not a complete netlist. It only consists of abstract information required to navigate the instrument access network.

Procedural Description Language (PDL)

- PDL provides a way to define procedures to operate an instrument.
- PDL is a sequence of commands that are a set of stimuli and expected responses that are applied on the interface of an instrument.
  - **iWrite :** Stimuli to the instrument are specified by iwrite.
    - Ex: iWrite Block1.MyTdr.R 8b00000101
      - In this example, the register R, an object in ICL instance MyTdr that is instantiated in instance Block1, is loaded with a value of 8b00000101
  - **iRead :** Expected responses are specified with the command iRead.
    - Ex: iRead Block1.MyTdr.R 0b0101
      - In this example, if R is an 8 bit wide register, then the compare value is zero padded to 0b00000101

TESSENT MBIST OPERATING PROTOCOL



- Memory Test is initiated when MBISTPG_EN signal is high and 2 bits of BIST_SETUP is 10.
- If an error is found during test, the GO signal goes low and stays low during the remaining test.
- When the test is completed, the DONE signal goes high.

DONE	GO	
1	1	The memory BIST test ran to completion and passed
1	0	The memory BIST controller completed the test, but one of the memories being tested failed

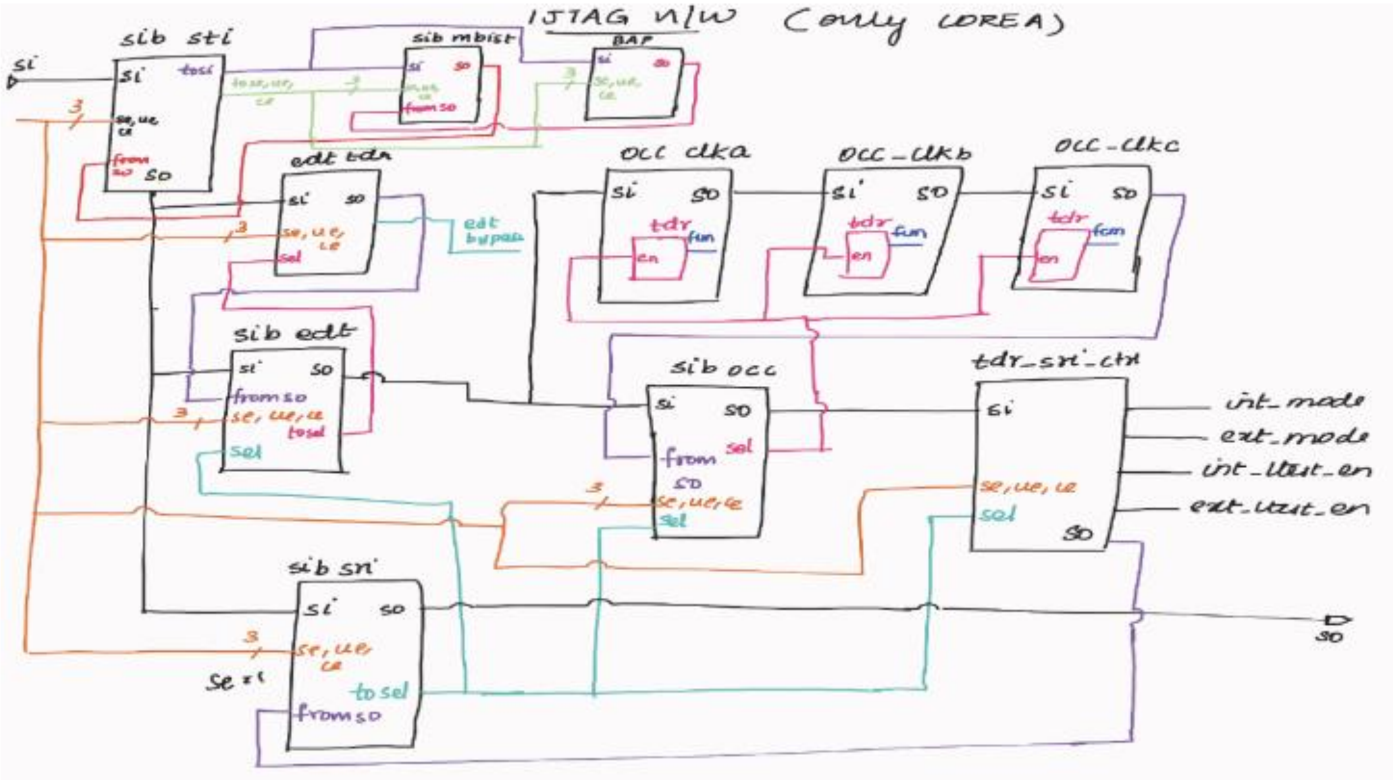
NOTE :

Suppose if a particular controller is controlling 50 memories, then the tool will automatically divide it into steps

ie in step 0 → 25 mem
in step 1 → 25 mem

In our lab, we are having 2 controllers
ie controller c1 → ram1 (1mem)
controller c2 → ram2 (1mem)

ram1 is controlled in step 0 of c1
ram2 is controlled in step 0 of c2.



NOTE: Actually se, ue, ce are separate signals. But as the diagram will become clumsy, I am representing it as 3 bit signal.

SYNTHESIS SCRIPT FILE EXPLANATION

set_size_only command ⇒ tool can change only the size of instances (drive strength
↓
how much load it can drive)

-filter {is_hierarchical == false}
↳ it can change only the size of leaf cells.

get_cells tessent_persistent_cell_*
↳ whichever instances are having 'tessent_persistent_cell' in its name, for only those instances, the size can be changed.