# 1. mbist

## 1.1 memory bist

mbist测试由mbist fsm控制。mbist controller/MBISTPG_FSM/STATE表示mbist fsm状态。RUNTEST_EN表示正在跑mbist测试。

mbist结果。MBISTPG_DONE = 1表示mbist测试结束。此时如果MBISTPG_GO为1，表示mbist测试PASS。否则为FAIL。

```
#这里面的step是什么意思
#一个step里多个memory inst
#和分开的step。每个step一个memory inst。有什么区别？
# step的意思是先测试完一个step，再测试下一个step。
#默认工具出来的配置是所有memory一起测，即在同一个step内。

  MemoryBist {
    ijtag_host_interface : Sib(mbist);
    Controller(c1) {
      clock_domain_label : dco_clk;
      Step {
        MemoryInterface(m1) {
          instance_name : DATA_MEM_1;
        }
      }
      Step {
        MemoryInterface(m2) {
          instance_name : PROG_MEM_2;
        }
      }
    }
  }
```

## 1.2 pipeline

mbist controller pipeline: AdvancedOptions/pipeline_controller_outputs : on”

memory interface pipeline: Step/bist_data_in_pipelining and Step/bist_data_out_pipelining to on

**Figure 1. Standard Flow Pipeline Stage Locations (Comparators in MemoryBIST Controller)**

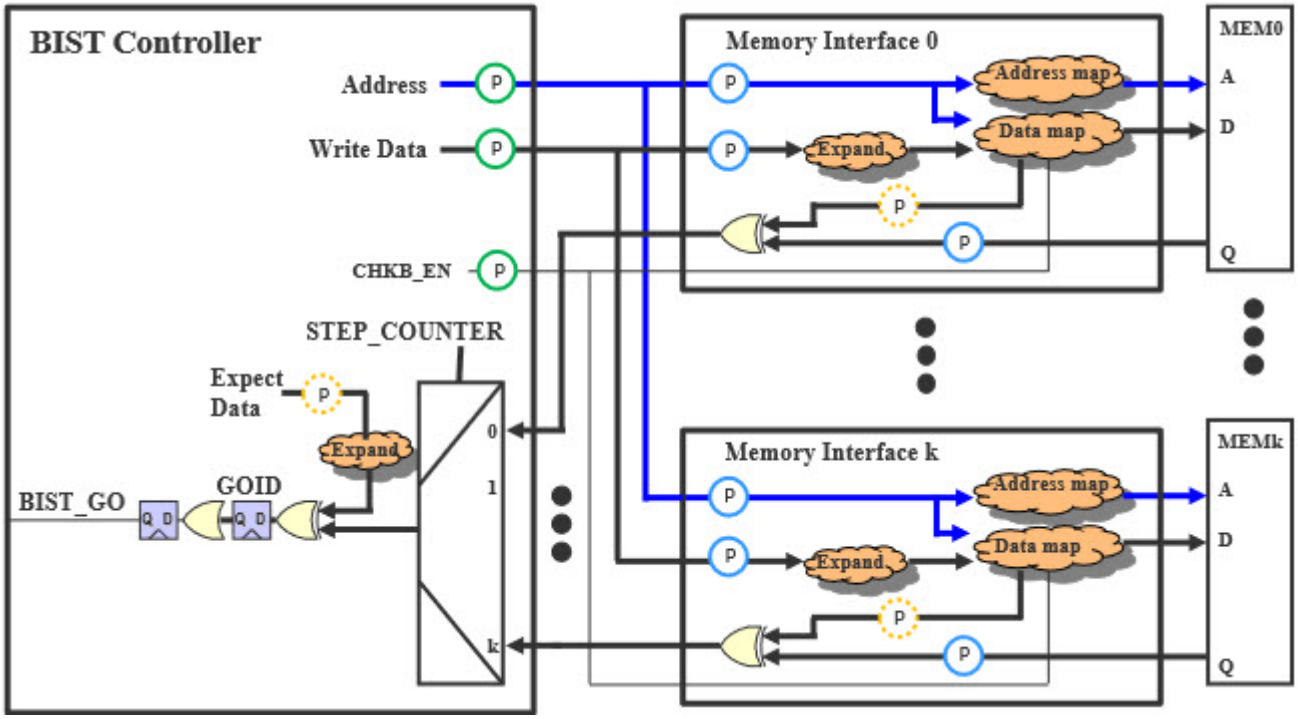



**Figure 2. Standard Flow Pipeline Stage Locations (Local Comparators)**

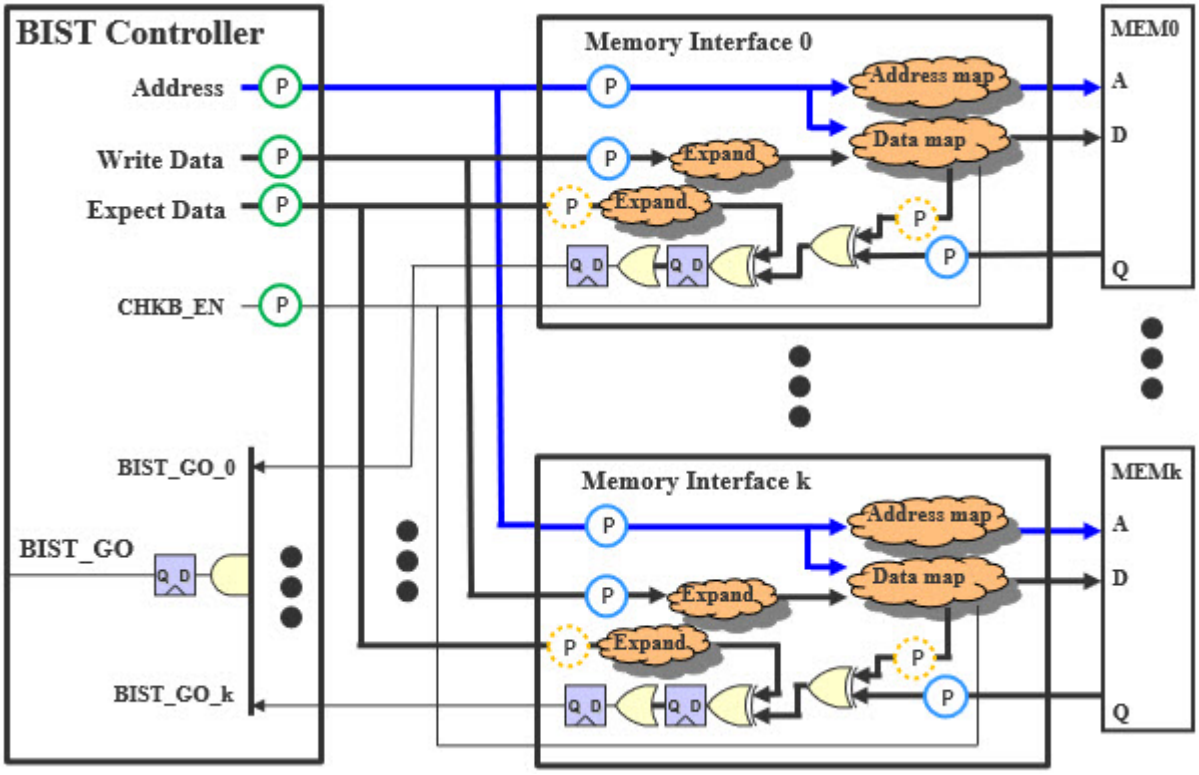
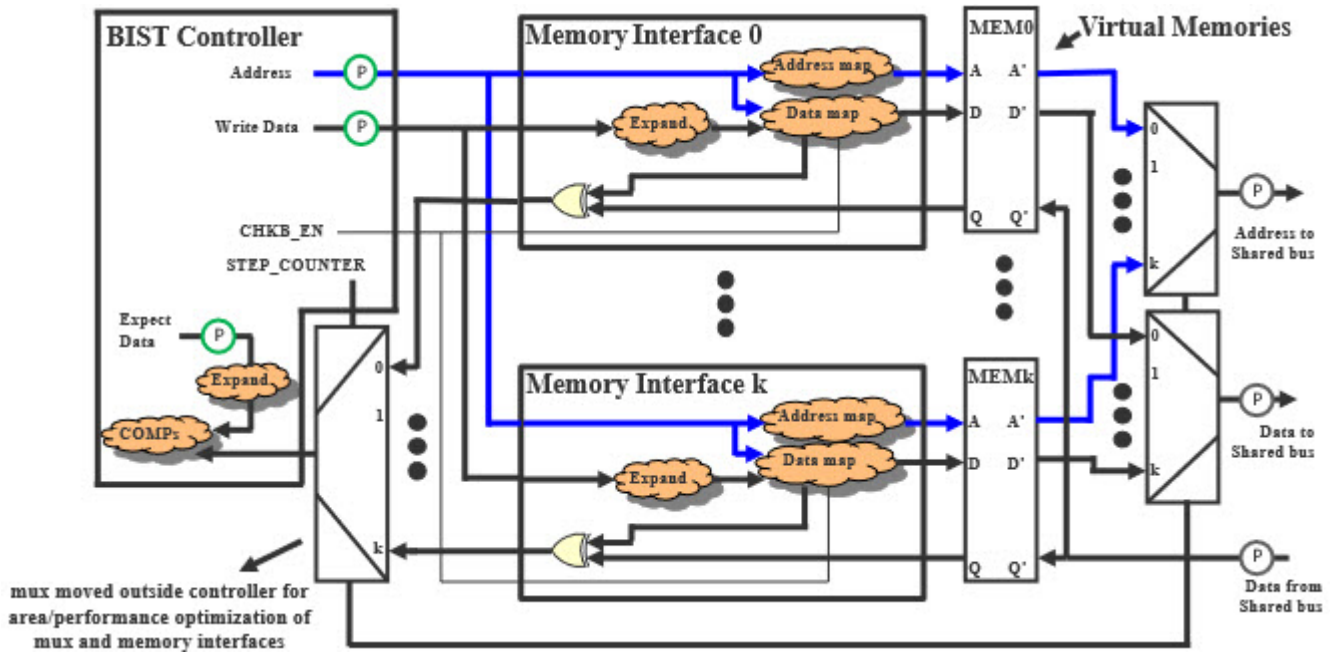
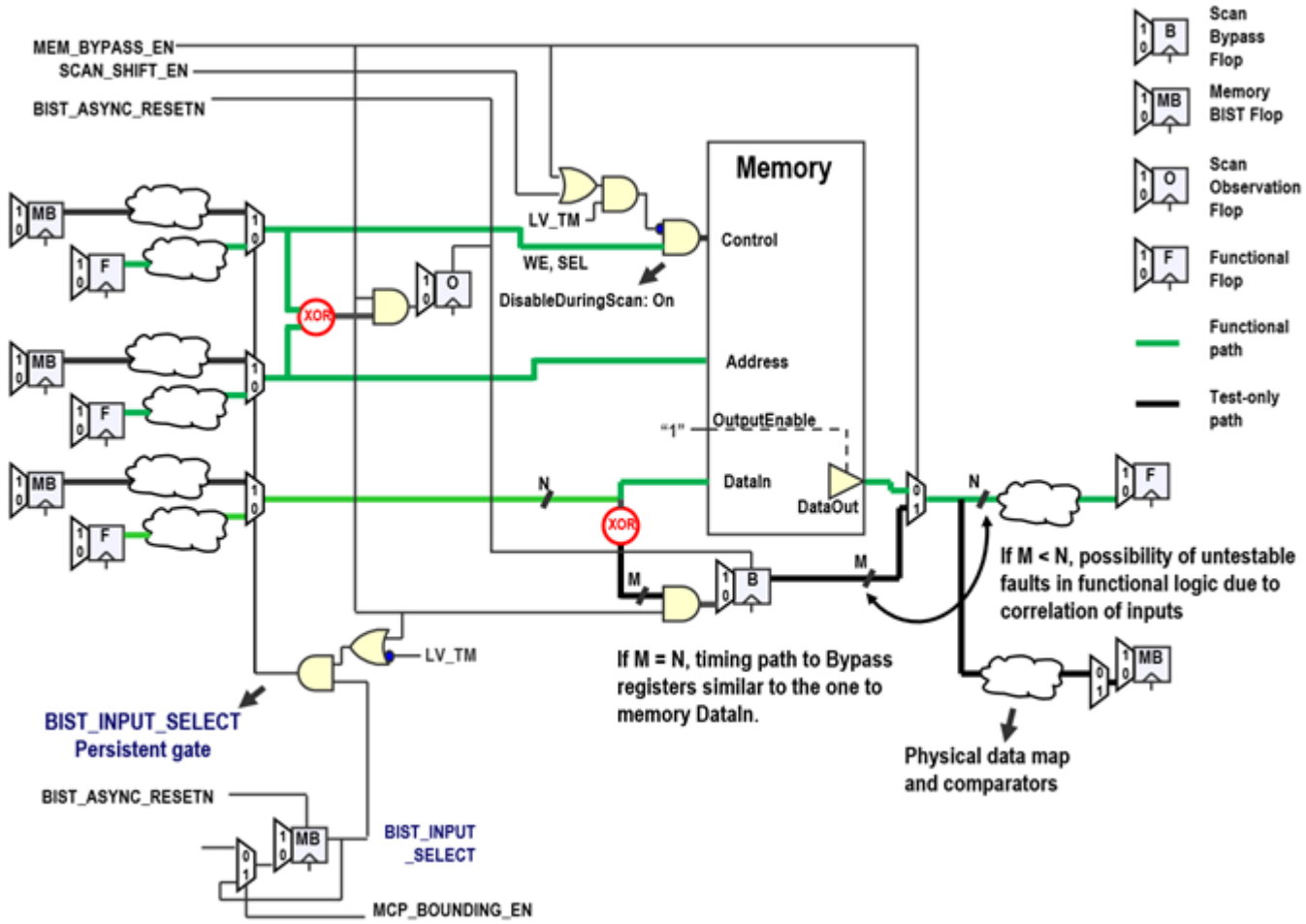



**Figure 3. Shared Bus Flow Pipeline Stage Locations**




## 1.3 MCP




图中MCP_BOUNDING_EN为1时，SCAN_EN=0时，BIST_INPUT_SELECT值不变
BIST_INPUT_SELECT register上SCAN chain, SCAN shift可控制其值，决定SCAN_en为0时测试funtion -> memory还是mbist -> memory数据通路。

## 1.4 no bypass

scan_bypass_logic: none; 设置没有memory input 到memory output的bypass通路,这一般用于memory macro里面自带有scan测试通路的情况。

```
DftSpecification(module_name,id) {
  MemoryBist {
    Controller(id) {
      Step { // repeatable
        algorithm                      : algo_name ;         // *DefSpec
        operation_set                  : opset_name ;
                                         // default: from_library
        comparator_location            : shared_in_controller |
                                         per_interface ;    // *DefSpec
        bist_data_in_pipelining      : on | off | int ;      // *DefSpec
        bist_data_out_pipelining     : on | off | per_port ; // *DefSpec
        MemoryInterface(id) {                               // repeatable
          generate_external_repair_logic : on | off ;
          instance_name                : inst_name ;
          memory_library_name          : mem_lib_name ;
          repair_analysis_present      : auto | off ;        // *DefSpec
          repair_group_name            : none | group_name ;
          scan_bypass_logic            : async_mux | none | sync_mux |
                                         from_library ; // *DefSpec
          local_comparators_per_go_id  : int | all ;         // 1 *DefSpec
          rom_content_file             : file_path;
          output_enable_control        : always_on | system ;
          observation_xor_size         : auto | off ;
          data_bits_per_bypass_signal  : 1..MaxPosInt | all ;// 1 *DefSpec
        }
        ReusedMemoryInterface(id) {      // repeatable *DefSpec
          instance_name                : inst_name ;
          reused_interface_id          : [ctrl_id:]mem_interface_id ;
          repair_group_name            : none | group_name ;
        }
      }
    }
  }
}
```

## 1.5 function debug

tessent支持Functional Debug Memory Access功能，意思是添加自定义算法，可通过mbist接口读写memory数据。

一般对debug有用的是读数据功能，因为是使用mbist接口来操作，使用上会有以下限制：

- memory接口需要一直有时钟

- 如果是想业务过程中读取的话，会影响正常业务功能

**相关文章**

有文章说通过SCAN chain的方式将memory的控制接口串在一起，然后通过控制这个scan chain来读写memory。
而且这条chain是单独的一条chain,其chain clock也是单独的
需要和其它的scan chain单独区分开，否则在其它scan chain dump时，memory可能会被误操作，这样就失去debug的意义了。

串scan chain时：

- 把memory 接口的cell 单独串chain
- 比如可以考虑给memory加一圈wrapper cell,这些wrapper cell可单独串一条chain,需要加个MUX控制
- 而且在dump的时候这个wrapper chain时候应该也是不能动
- 然后在dump的时候将其从整个chain上bypass掉，这样可保证memory接口信号不变。

以上只是臆想，未经过验证。

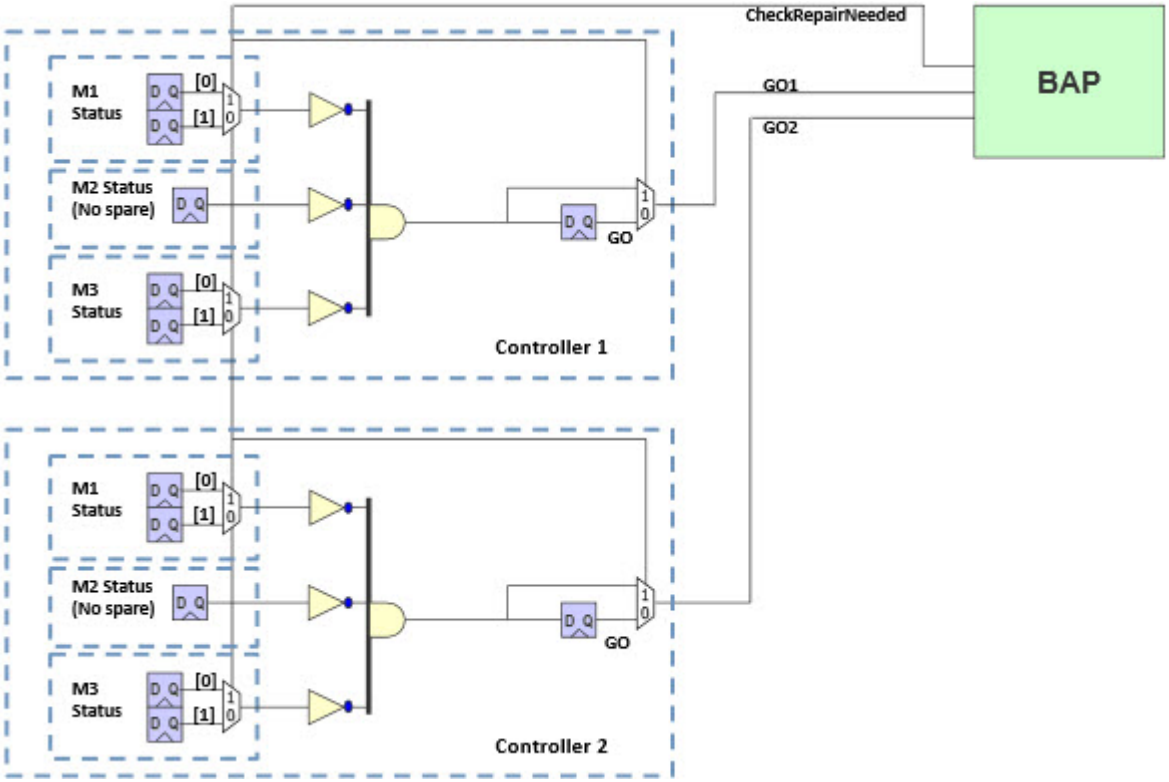
## 1.6 BIRA Repair Status

参考：http://vmcc.vicp.net:9090/tessent_v2023.1_doc/htmldocs/mgchelp.htm#context=tshell_mbist_user&id=203&tag=id3ea04440-17f9-477f-abc9-d5768e2d593c

BIRA_en==0, 跑mbist是得到memory bist GO 读写测试结果。pass=1(没错)。 pass=0(有错)

BIRA_en==1。然后再跑mbist。得到repair status[1:0]。以及的repair 信息 其中CheckRepairNeed将repair status[1:0]分成高低bit输出到go
GO = CheckRepairNeed ? ~repair_status[0] : ~repair_status[1]。

图示如下：



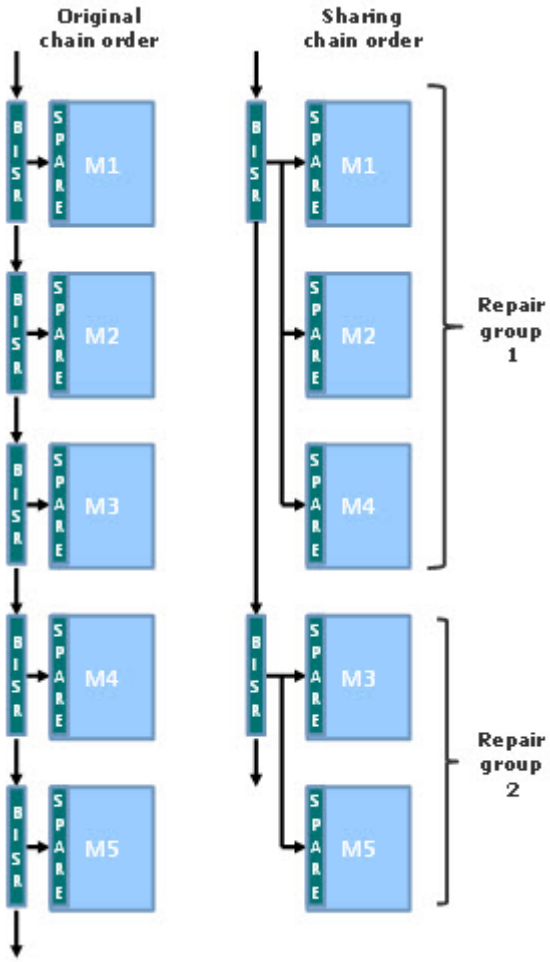

repair_status[1:0]定义如下：

```
Bit1 Bit0     Repair Status
00            No Repair Required
01            Repair Required
1x            Not Repairable
```

## 1.7 repair share

为了节省repair chain的长度，从而降低fuse空间占用。将同类型memory repair信息进行共享。
使用限制：

- memory repair必须是并行接口
- 要share repair的memory必须是相同repair type Row/word-only, Column/IO-only
- 只会使用一个repair segment, 如果memory有多个spare element的，多余的部分不会使用
- memory spare size必须一样
- repair group与spec一致
- physical address mapping要兼容 segment size要兼容




### 1.7.1 tcl

```
set_defaults_value DftSpecification/MemoryBist/RepairOptions/repair_sharing
on;
set_memory_instance_options [get_memory_instances] -repair_sharing on
```

### 1.7.2 spec

```
DftSpecification(top,rtl) {
  MemoryBist {
    ijtag_host_interface : Sib(mbist);
    Controller(c1) {
      clock_domain_label : clka;
      Step {
        comparator_location : shared_in_controller; # // comparator必须要
在mbist controller里面
        MemoryInterface(m2) {
          instance_name : core_inst1/blockA_clka_i1/mem4;
          repair_group_name : bira_g1;
        }
        MemoryInterface(m3) {
          instance_name : core_inst1/blockA_clka_i1/mem5;
```

```
          repair_group_name : bira_g2;
        }
        MemoryInterface(m4) {
          instance_name : core_inst2/blockA_clka_i1/mem5;
          repair_group_name : bira_g1;
        }
      }
    }
  }
}
```

生成出来的order file样式

```
BisrSegmentOrderSpecification {
  PowerDomainGroup(-) {
    // bisr_si --> {OrderedElement ...} --> bisr_so
    OrderedElements {
      mem_m3; // RepairGroup:c1_RG2 BISRLength:18    -- RG2有两块memory共享18bit的repair信息。
      mem_m5; // RepairGroup:c1_RG2
      mem_m1; // RepairGroup:c1_RG1 BISRLength:18
      mem_m2; // RepairGroup:c1_RG1
      mem_m4; // RepairGroup:c1_RG1
    }
  }
}
```

## 1.8 mbist test clock

对于多PORT memory(比如RF2)测试，选port的最高CLK freq来测试，即使某个memory port在function业务时工作在较低频率。

因为这是走mbist <=> memory之间的path, 只是这段path用高频，实际业务跑低频也没有关系，不会产生影响。

一般来说memory compiler生成出来的memory每个port的操作频率都是一样，所以memory本身应该也能够承受高频的测试。

## 1.9 使用TCK时钟来测试memory

可能是为了慢速调试，不使用高频时钟，而使用tck时钟去做memory bist读写测试。

出pattern方式：

方式一：直接配置OCC的inject_tck为1，TCK直接注入到时钟通路，跑mbist测试。

方式二，Pattern(MemoryBist_P1)/tck_clock_only为on

注：不能用于ATPG测试。

## 1.10 mbist memory分组

**方法一:** 推荐方式

```
# 根据memory hier name 关键字将memory分成多个组
set_memory_instance_options [get_memory_instances -filter "name=~*ab*"] -
partitioning_group ab
set_memory_instance_options [get_memory_instances -filter "name=~*cd*"] -
partitioning_group cd
```

**方法二:**

```
# 根据def信息来进行memory bist controller分组。
read_def xxx.def
set_memory_instance_option -physical_cluster_size_ratio 30

# 这个有点不太好用，不能精确控制分组。
```

**方法三:**

手动修改spec进行手动分组。

## 1.11 时钟定义

memory要做mbist测试， 那连到memory port的clock需要先在tcl用 add_clocks 定义

# 2. mbisr

## 2.1 repair mode

| opcode[2:0] | Run Mode | Description | ========== |
|---|---|---|---|

| opcode[2:0] | Run Mode | Description | ========== |
|---|---|---|---|
| 3'b000 | Functional Power-Up | Extracts the repair data from the eFuse and loads it inside the BISR chains. This mode also calculates the BISR chain length during the chain loading process. | |
| 3'b001 | BISR Chain Length Calculation | Performs an asynchronous reset of the BISR chains, followed by a serial loading of 0s and calculation of the BISR chain length. | |
| 3'b010 | Self Fuse Box Program | Compresses the content of the BISR chains and programs it into the eFuse. A BISR chain rotation is performed during this process. | |
| 3'b100 | Verify Fuse Box | Performs a BISR chain rotation and compares the content against the compressed data inside the eFuse. The BISR controller GO output is high if the content matches, otherwise GO stays low. | |

| opcode[2:0] | Run Mode | Description | ========== |
|---|---|---|---|
| 3'b101 | Rotate BISR chain (No capture) | Rotates the content of the BISR chain. Use this mode to transfer the BISR chain content into internal repair registers of memories with serial repair interfaces. | MSEL=1, 将memory BISR chain内容转一圈，移入到BISR chain中，然后可用于jtag读取 |
| 3'b110 | Capture BIRA into BISR without chain rotation | Performs a capture of the redundancy analysis values into the BISR chain. This mode is typically used after running redundancy analysis, and before the Self Fuse Box Program mode. This mode is also used when performing soft repair in designs that contain memories with parallel repair interfaces. | 产生一个bisr clk bisrSE=0, 将BIRA的值保存到BISR chain寄存器，然后可用jtag读取 |

| opcode[2:0] | Run Mode | Description | ========== |
|---|---|---|---|
| 3'b111 | Capture BIRA into BISR chain with chain rotation | Performs a capture of the redundancy analysis values into the BISR chain, followed by a BISR chain rotation. This mode is also used when performing soft repair in designs that contain memories with serial repair interfaces. | MSEL=0将BIRA值加载到BISR chain然后转一圈将BISR值移入到memory中，实现repair |

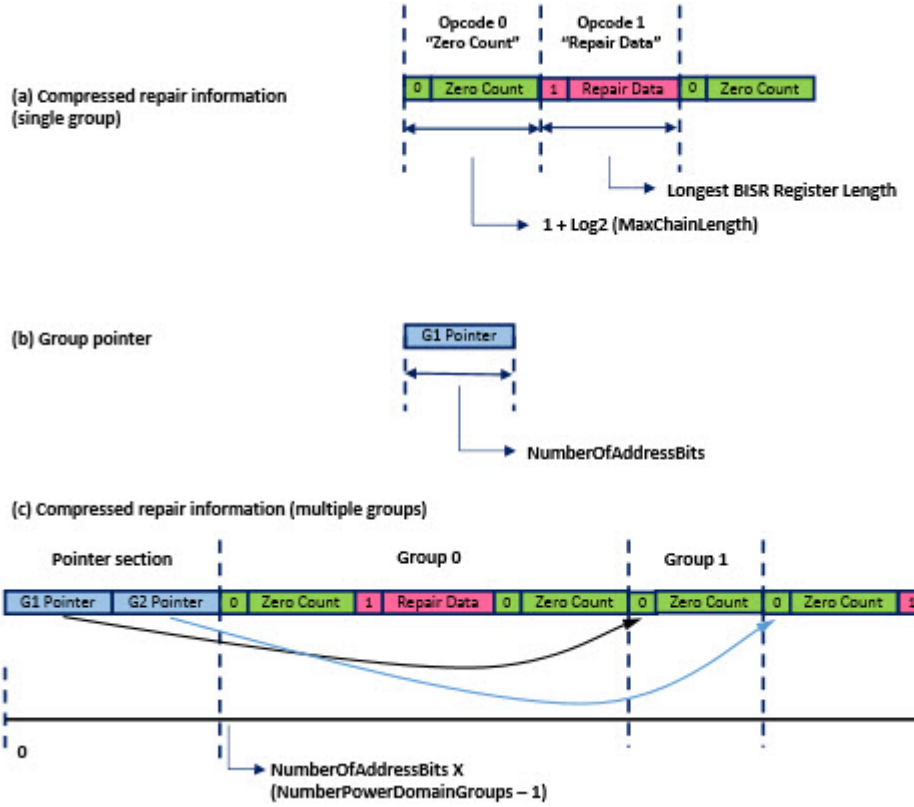
## 2.2 Incremental Repair

原生结构支持多少fuse信息烧写,每次烧写数据通过指针索引。
将max_fuse_box_programming_sessions配置为大于1的整数，就表示支持最大多少次数的fuse烧写。

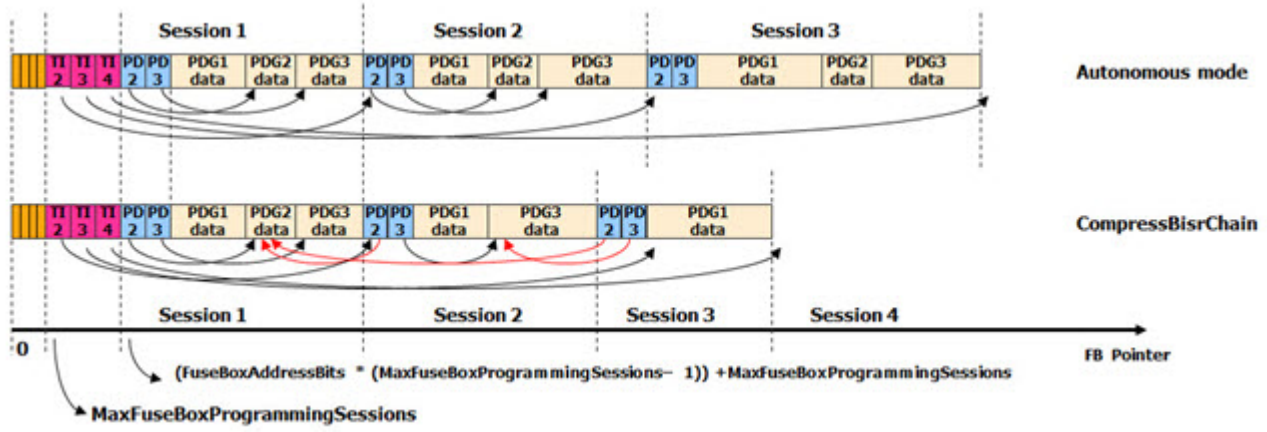
```
DftSpecification {
  MemoryBisr {
    Controller {
      ...
      max_fuse_box_programming_sessions : <int> ;
      ...
    }
  }
}
```

## 2.3 fuse box组织结构

只支持单次fuse烧写的情况




支持多次fuse烧写的情况




# 2.4 soft repair flow

## 2.4.1 原生结构

原生架构，只能所有memory一起进行soft repair，不能部分做soft repair。

### 上电自检

1. load fuse to bisr chain, 这个信息同时也进入memory，memory完成hard repair
2. do mbist, check mbist status, pass=1/0

### soft repair

1. 将mbisr BISR chain清零（bisr_rst_n = 0, repair_mode[2:0] = 0）, 此步骤可删除，因为将清零后

的BIRA加载到memory一样也实现BISR chain清零
2. 将mbist controller BIRA清零（mbist controller test_start=0, test_init=0）
3. 将清零后的BIRA加载到memory（repair_mode[2:0]=7, bisr_rst_n产生一个上升沿)
4. 进入mbist BIRA, (mbist controller test_start=1, bira_en=1, test_init=1)
5. 将新的BIRA加载到memory（repair_mode[2:0]=7, bisr_rst_n产生一个上升沿）

注：对于多pd group的场景，需要单独注意，即在使用bisr 加载bira到memory之前需要先做一个bisr复位操作。
具体序列为（ repair_mode[2:0]=0, bisr_rst_n=0, 不要产生bisr_rst_n的上升沿

**重新自检**

1. 开始进行mbist测试（mbist controller test_start=1, test_init=0, bira_en=0）

### 2.4.2 改进结构

注：此改进仅针对于memory repair 接口是串行接口，而不是并行接口。
并行接口的话，需要另外加逻辑, 因为memory reapir直接从bist_inst的ShiftReg上取数据（bisr chain移动的时候这个值会变
所以在repair期间memory不能工作,除非加个类似锁存的功能，将Q值锁存住。

问题：

1. 因为repair始终是需要capture BIRA（所以如果某些memory不重做BIRA的话，这个值从哪来？
2. 或者不需要soft repair的memory可不可以不走capture BIRA这一步？

方案1： 修改点在mbist interface RA分析模块

1. mbist添加一个单独信息从BISR里面把fuse里面的BIRA信息load到本地BIRA register（
2. 这些mbist pass的memory就可以不用再做BIRA分析了，以及再之后的mbist
3. 但必须得有BIST_CLK（BIRA信息才能load到本地，这样看起来还不如重新跑下BIRA呢。

方案2：修改点在bisr_inst

1. bisr 模块加一个单独控制，不capture BIRA信息
2. 这样不做soft repair的memory就不用去capture BIRA,只要保证不能发清BISR chain的序列(bisr_rst=0， repair_mode[2:0] == 3'h0 / 3'h4)（就不会有问题。
3. 但是在其它memory repair过程中，因为有bisr chain shift, shift 过程中memory不能进行正常读写。

方案3：

1. 让送给memory的bist clk & bisr reset控制一下，不需要重新repair的memory连bisr_clk都没有，这样repair信息一直保持原值，其它memory repair时也可以正常工作。
2. BISR chain register不需要做特殊控制，清零也没有关系（memory内部的repair信息是保持之前的值不受影响。

所以可以考虑方案3, 已仿真确认方案3可行。

## 2.5 spec

### 2.5.1 在BISR chain上添加pipeline

#### 在每个Meory inst的BISR前插入一个pipeline

```
MemoryBisr {
  BisrElement(*) {
    Pipeline(before) {
      leaf_instance_name : %s_%m_bisr_pipeline_inst ;
    }
  }
  bisr_segment_order_file      : filename ;
  AdvancedOptions {
  }
  Interface {
  }
  Controller {
  }
}
```

#### 在ram0和ram2之前插入pipeline

```
MemoryBisr {
  BisrElement(ram_inst0) {
    Pipeline(before) {
      leaf_instance_name : %s_%m_bisr_pipeline_inst ;
    }
  }
  BisrElement(ram_inst2) {
    Pipeline(before) {
      leaf_instance_name : %s_%m_bisr_pipeline_inst ;
    }
  }
}
```

memory repair, 本质上就是把repair信息重新写入到memory中，memory内部完成col 或 row的替换。

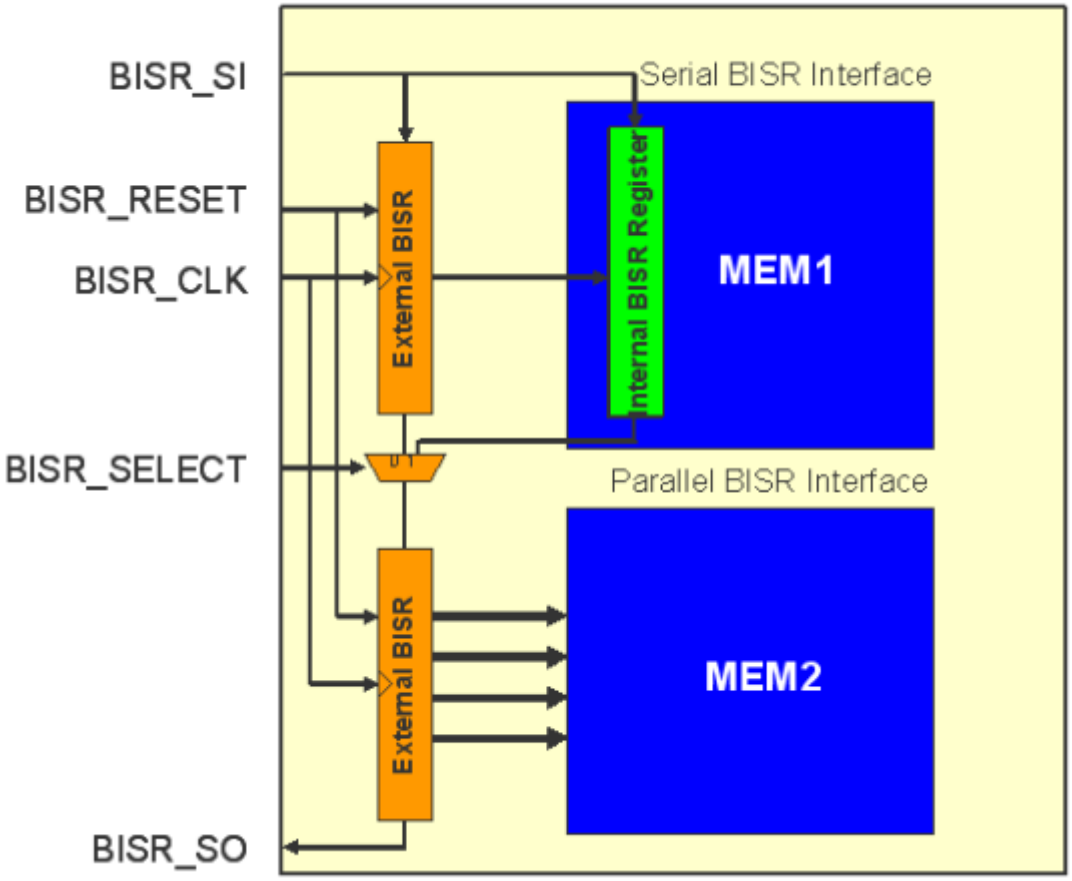
## 2.6 推pattern

bisr相关的pattern完全是根据icl文件描述来进行的,

所以还是有可能通过手动修改ICL的方式，在顶层添加一个register pipe逻辑在repair chain上。

## 2.7 memory serial repair interface

**Figure 5-2. Memories With Serial and Parallel Self-Repair Interfaces**




每一个有repair能力的memory外部都有一条bisr chain, 与memory internal的repair chain相对应，它们的bisr_si都是一样的，但最终输出可以选择从外部bisr chain或者memory内部的repair chain输出。

bisr rotate功能：

就是在整个bisr chain上，将bisr_so与bisr_si相连shift一圈。主要目的是实现外部bisr chain的内容刷到memory内部repair chain中。比如在rotate转圈的时候BISR_SELECT选择external, 即相当于外部bisr chain转了一圈，内容没有变，但转圈的同时memory内部的chain也被刷为跟外部bisr chain的内容一样。这样就是repair信息导入到memory中了，下次做memory bist测试的时候，测试就能跑过。

在上面的shift转圈过程中，如果BISR_SELECT选择internal, 即memory内部chain在转圈后内容保持不变，但外部bisr chain的内容刷为到memory 内部chain一样。

## 2.8 BISR CE DISABLE

在memory repair分发过程中，会产生bisr ce disable信号出来，表示在此期间不能进行memory功能读写操作。

- 这个信号只在有串行的memory repair接口存在时才会使用，
- 如果memory是并行的repair接口，则不需要这个BISR_MEM_DISABLE信号。

# 2.9 fuse value

BISR控制器会从fuse里读出repair数据，这里有两种场景：

- fuse还没有烧
- fuse已经烧写

对于fuse还未烧写的情况，需要确保从fuse里读出的数据必须为全0，这样BISR状态机可以很快是产生bisr DONE信号，这样方便走后续BOOT流程
此时是不会有bisr clock送给memory, 只是在INIT_BISR的时候会产生一个cycle的bisr时钟给memory

对于fuse已烧写的情况，需要烧真实工具产生出来的fuse数据

## 2.9.1 多次memory bist repair测试结果积累

比如有的memory带2个repair列，第一次测试时可能只错第1列， 第2次测试的时候错第2列，那其实最终应该是把错2列当时结果，来计算最终的repair value

方式：
测试第一次时，将fuse ram内容清空，写fuse

测试第二次，运用第一次测试得出来的fuse进行上电分发fuse值不要烧到fuse,即bisr chain上还是保留的是带repair的信息，这时直接进行repair analysis
得结果后，先把fuse ram内容清空，写fuse.

此时最终的fuse值就得出来了。

清空很重要。一定要清空

| **fuse初始值** | **不造错** | **造L列错** | **造R列错** | **造L列R列错** |
|---|---|---|---|---|
| 全0 | PASS, fuse值不变 | PASS, fuse值带L列修复 | PASS, fuse值带R列修复 | PASS, fuse值带L列R列修复 |
| 带L列修复 | PASS, fuse值不变 | PASS, fuse值不变 | **PASS, fuse值跟带L列R列修复值一样** | **PASS, fuse值跟带L列R列修复值一样** |
| 带R列修复 | PASS, fuse值不变 | **PASS, fuse值跟带L列R列修复值一样** | PASS, fuse值不变 | **PASS, fuse值跟带L列R列修复值一样** |
| 带L列R列修复 | PASS, fuse值不变 | PASS, fuse值不变 | PASS, fuse值不变 | PASS, fuse值不变 |

# 2.10 fusebox interface

## 2.10.1 external fusebox

通过修改fusebox.v或者是它的tcd文件，可以让mbisr controller内部访问fusebox的地址进行适配，从而不会限制在默认10bit地址。

这样就可以支持更大的fusebox地址段。

```
# tcd格式
Core(module_name) {
  FuseBoxInterface {

    Interface {
      // inputs
      bisr_en                : port_name;
      clock                  : port_name;
      select                 : port_name;
      reset                  : port_name;
      access_en              : port_name;
      write_en               : port_name;
      address                : port_name;  // n-bit
      write_buffer_transfer : port_name;
      read_buffer_select     : port_name;
      programming_voltage     : port_name;

      write_duration_count   : port_name;  // n-bit

      logictest_en           : port_name;

      // outputs
      done                   : port_name;
      read_data              : port_name; // 1-bit
      read_buffer_output     : port_name; // n-bit
    }
  }
}



# dft spec
DftSpecification(module_name,id) {
  MemoryBisr {
    Controller {
      ExternalFuseBoxOptions {
        design_instance          : inst_name ;
        multiplexing             : on | off | auto ;
        ConnectionOverrides {
          // Inputs
          bisr_en                : pin_name,... ;
          clock                  : pin_name ;
          select                 : pin_name ;
          reset                  : pin_name ;
          access_en              : pin_name ;
          write_en               : pin_name ;
          write_duration_count   : pin_name ;
          read_buffer_select     : pin_name ;
          write_buffer_transfer : pin_name ;
          address                : pin_name ;
          // Outputs
```

```
            done                    : pin_name ;
            read_data               : pin_name ; // 1-bit
            read_buffer_output      : pin_name ; // n-bit
          }
        }
      }
    }
  }
```

### 2.10.2 internal fusebox

可根据tessent自动生成出来的文件，创建并读取xx_fusebox_interface.v文件
通过配置spec
MemoryBisr.Controller.fuse_box_interface_module : xx_fusebox_interface;
MemoryBisr.Controller.fuse_box_location:internal;

读xx_fusebox_interafce.tcd文件

这样bisr可以支持更大地址范围的fusebox

## 2.11 bisr_segment_order

这个文件会在check_design_rules的时候自动生成

或者是

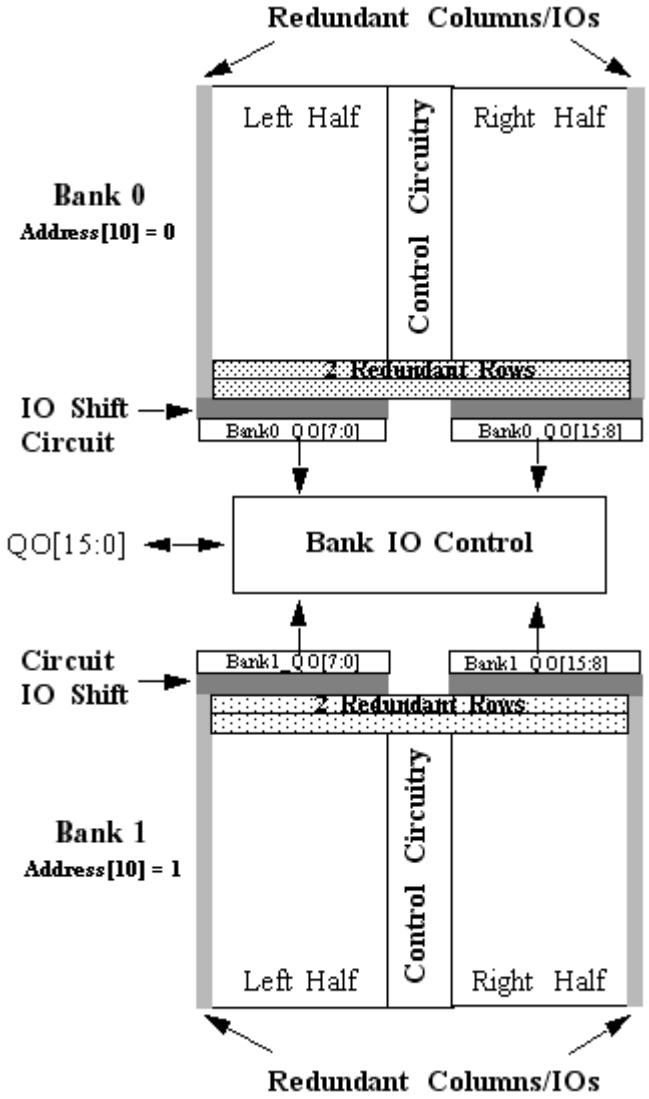
set_dft_specification_requirements -bisr_segment_order_file command was used, in which case the specified file is not generated but validated.

如果需要某些mem_init不做mbisr 或 mbist的时候，某设置需要在check_design_rules之前设置。

# 3. memory lib

http://vmcc.vicp.net:9090/tessent_v2023.1_doc/htmldocs/mgchelp.htm#context=ETAssembleReference&id=187&tag=ide3d3723f-7a5d-4677-a2cc-327ec2b0c7d7

## 3.1 memory lib




```
RedundancyAnalysis {
   RowSegmentRange {
      SegmentAddress[0]: AddressPort(Address[10]);
   }
   RowSegment (Bank0){
      NumberOfSpareElements: 2;
      RowSegmentCountRange [1'b0:1'b0]; // Bank 0
      FuseSet {
         Fuse[3]: AddressPort(Address[9]);
         Fuse[2]: AddressPort(Address[8]);
         Fuse[1]: AddressPort(Address[7]);
         Fuse[0]: AddressPort(Address[0]);
      }
   }
   RowSegment (Bank1){
      NumberOfSpareElements: 2;
      RowSegmentCountRange [1'b1:1'b1]; // Bank 1
      FuseSet {
         Fuse[3]: AddressPort(Address[9]);
         Fuse[2]: AddressPort(Address[8]);
         Fuse[1]: AddressPort(Address[7]);
```

```
         Fuse[0]: AddressPort(Address[0]);
      }
   }
   ColumnSegment (Bank0_Left){
      RowSegmentCountRange [1'b0:1'b0]; // Bank0
      ShiftedIORange: QO[7:0]; // Left
      FuseSet {
         FuseMap[3:0]{
            ShiftedIO(QO[0]): 4'b0001;
            ShiftedIO(QO[1]): 4'b0010;
            ShiftedIO(QO[2]): 4'b0011;
            ShiftedIO(QO[3]): 4'b0100;
            ShiftedIO(QO[4]): 4'b0101;
            ShiftedIO(QO[5]): 4'b0110;
            ShiftedIO(QO[6]): 4'b0111;
            ShiftedIO(QO[7]): 4'b1000;
         }
      }
   }
   ColumnSegment (Bank0_Right){
      RowSegmentCountRange [1'b0:1'b0]; // Bank 0
      ShiftedIORange: QO[15:8]; // Right
      FuseSet {
         FuseMap[3:0]{
            ShiftedIO(QO[8]): 4'b0001;
            ShiftedIO(QO[9]): 4'b0010;
            ShiftedIO(QO[10]): 4'b0011;
            ShiftedIO(QO[11]): 4'b0100;
            ShiftedIO(QO[12]): 4'b0101;
            ShiftedIO(QO[13]): 4'b0110;
            ShiftedIO(QO[14]): 4'b0111;
            ShiftedIO(QO[15]): 4'b1000;
         }
      }
   }
   ColumnSegment (Bank1_Left){
      RowSegmentCountRange [1'b1:1'b1]; // Bank 1
      ShiftedIORange: QO[7:0]; // Left
      FuseSet {
         FuseMap[3:0]{
            ShiftedIO(QO[0]): 4'b0001;
            ShiftedIO(QO[1]): 4'b0010;
            ShiftedIO(QO[2]): 4'b0011;
            ShiftedIO(QO[3]): 4'b0100;
            ShiftedIO(QO[4]): 4'b0101;
            ShiftedIO(QO[5]): 4'b0110;
            ShiftedIO(QO[6]): 4'b0111;
            ShiftedIO(QO[7]): 4'b1000;
         }
      }
   }
```

```
   ColumnSegment (Bank1_Right){
      RowSegmentCountRange [1'b1:1'b1]; // Bank 1
      ShiftedIORange: QO[15:8]; // Right
      FuseSet {
         FuseMap[3:0]{
            ShiftedIO(QO[8]): 4'b0001;
            ShiftedIO(QO[9]): 4'b0010;
            ShiftedIO(QO[10]): 4'b0011;
            ShiftedIO(QO[11]): 4'b0100;
            ShiftedIO(QO[12]): 4'b0101;
            ShiftedIO(QO[13]): 4'b0110;
            ShiftedIO(QO[14]): 4'b0111;
            ShiftedIO(QO[15]): 4'b1000;
         }
      }
   }
}
```

# 4. 其它

## 4.1 memory不做bist, 不做bisr

```
#不做bisr, 不上bisr chain
set_memory_instance_options memory_instances -
use_in_memory_bisr_dft_specification off

#不做bist
set_memory_instance_options memory_instances -
use_in_memory_bist_dft_specification off

#这些命令要在check_design_rules命令之前
```

## 4.2 get tcd_memory pin

http://vmcc.vicp.net:9090/tessent_v2023.1_doc/htmldocs/mgchelp.htm#context=tshell_ref&id=1595&tag=idc5e2f6dc-0772-436d-8ebd-b63f42af3e2e

```
get_pins -of_instances [get_memory_instance] -filter {tcd_memory_function =~
select*}
get_attribute_value_list data_mem_1/CEN -name tcd_memory_function
#高有效返回select,  低有效返回select_inv
```

```
#获取BISR相关PIN
get_pins -of_instances [get_memory_instance] -filter {tcd_memory_function =~
bisr*}
```

```
foreach_in_collection pin [get_pins xxx_mem/* -filter
"tcd_memory_function=~bisr*"] {
   set name [get_attribute_value_list $pin -name name]
   set att  [get_attribute_value_list $pin -name tcd_memory_function]
   puts "name = $name, att = $att"
}
name = xxx_mem/RSCOUT, att = bisr_serial_data
name = xxx_mem/RSCIN,  att = bisr_serial_data
name = xxx_mem/RSCEN,  att = bisr_scan_enable
name = xxx_mem/RSCRST, att = bisr_reset_inv
name = xxx_mem/RSCLK,  att = bisr_clock

如果port极性是ActiveLow,则会在funtion后面跟一个_inv,比如返回bisr_reset_inv
如果port极性是ActiveHigh,则不会在在function后跟_inv,比如返回bisr_reset


# 获取TCD相关PIN
get_pins -of_instances [get_memory_instance] -filter {tcd_memory_function !~
none}
```