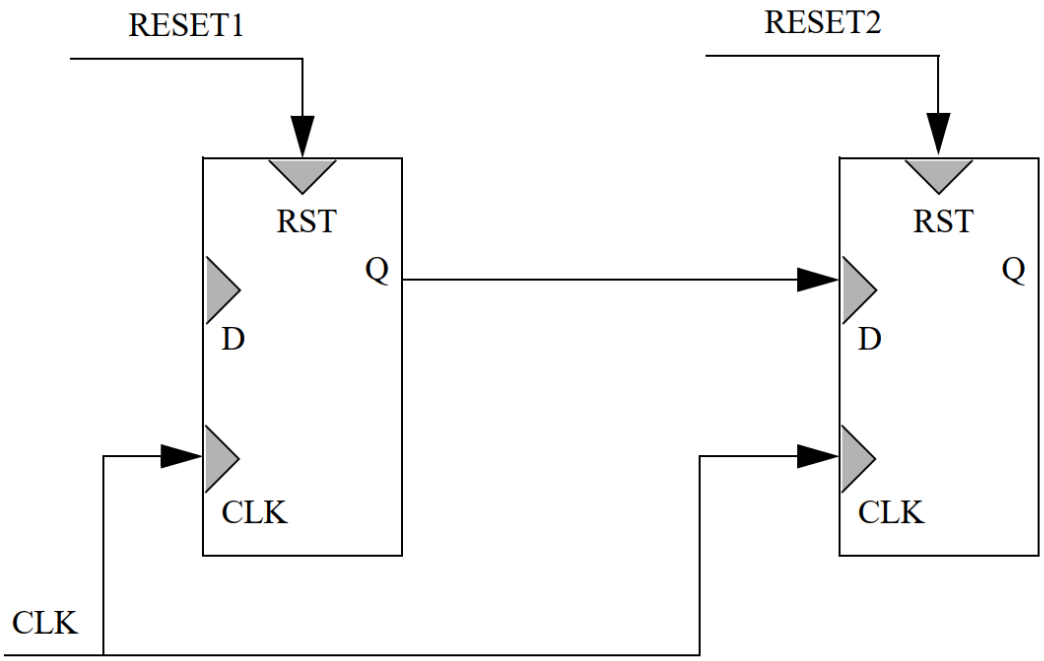
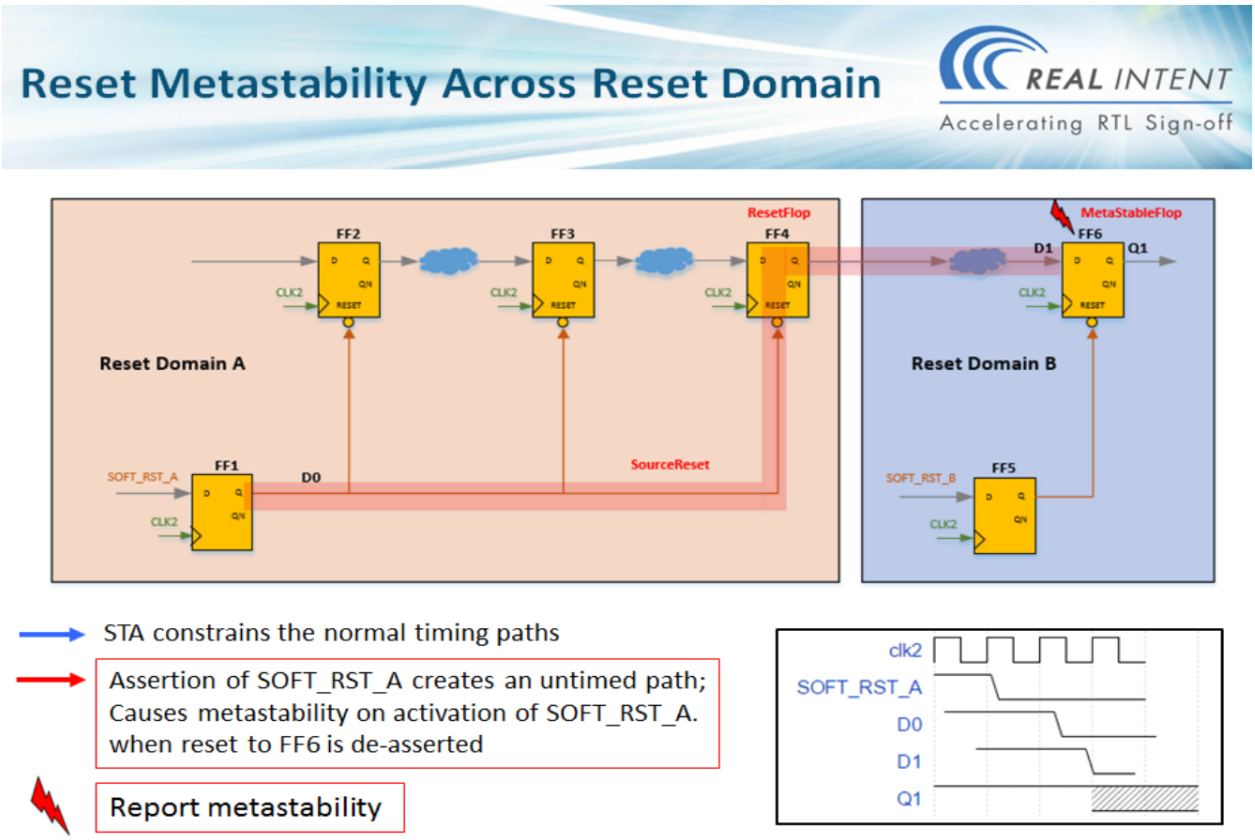
1 Reset Domain Crossing - Basics

When a design is clocked by a single clock but have multiple asynchronous resets, there is a possibility of having reset domain crossing paths which can cause Metastability due to the assertion of asynchronous resets. This is an issue because static timing analysis does not check Reset->Q paths on assertion of reset.

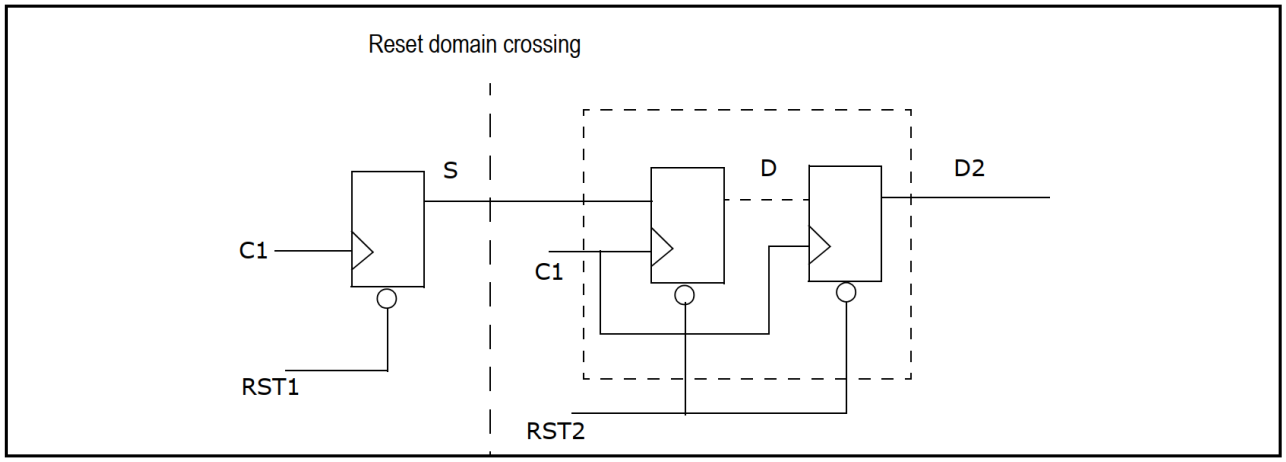


When RESET1 is asserted and RESET2 is in de-asserted state, the right flop can go to metastable state thereby corrupting the downstream logic.

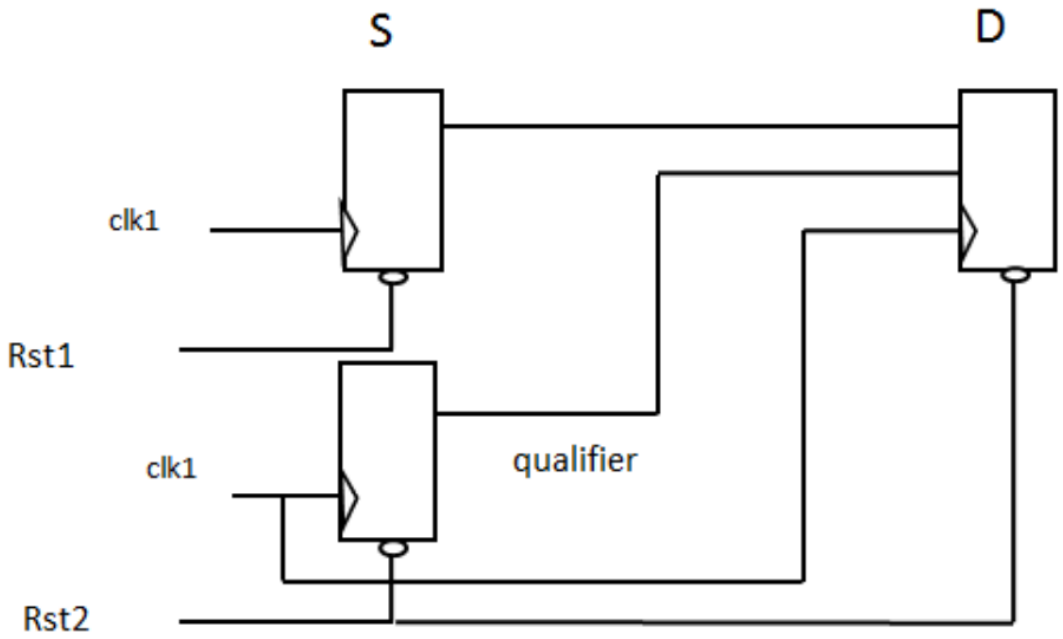


There are couple of ways in which RDC paths can be fixed.

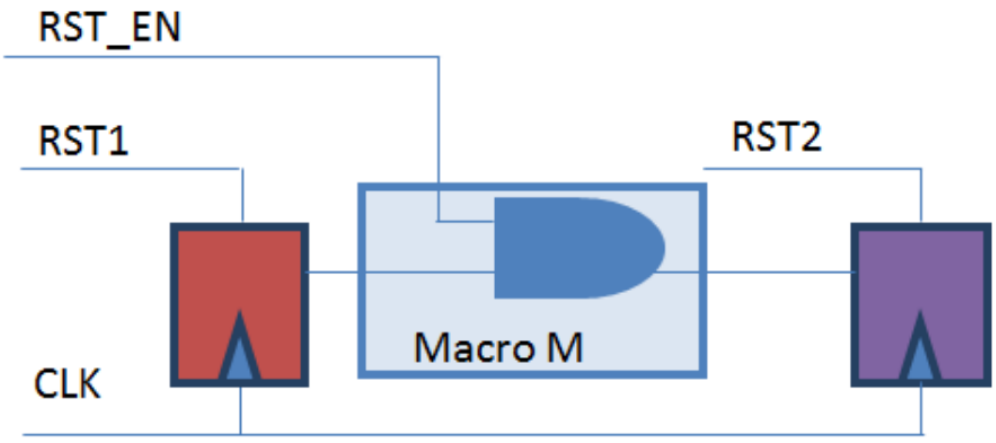
- **Data path Synchronization** - Adding synchronizer in data path



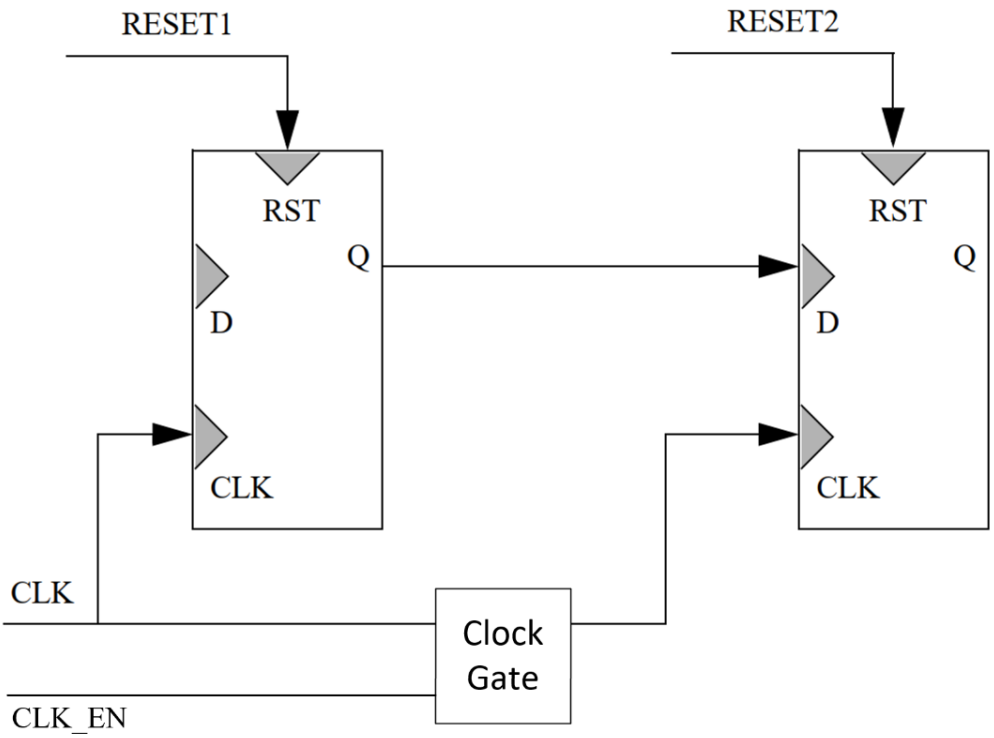
- **Synchronized Enable Synchronization Scheme** - Block the ingress data path of destination flop during the source flop reset assertion window.



qualifier = RST_EN



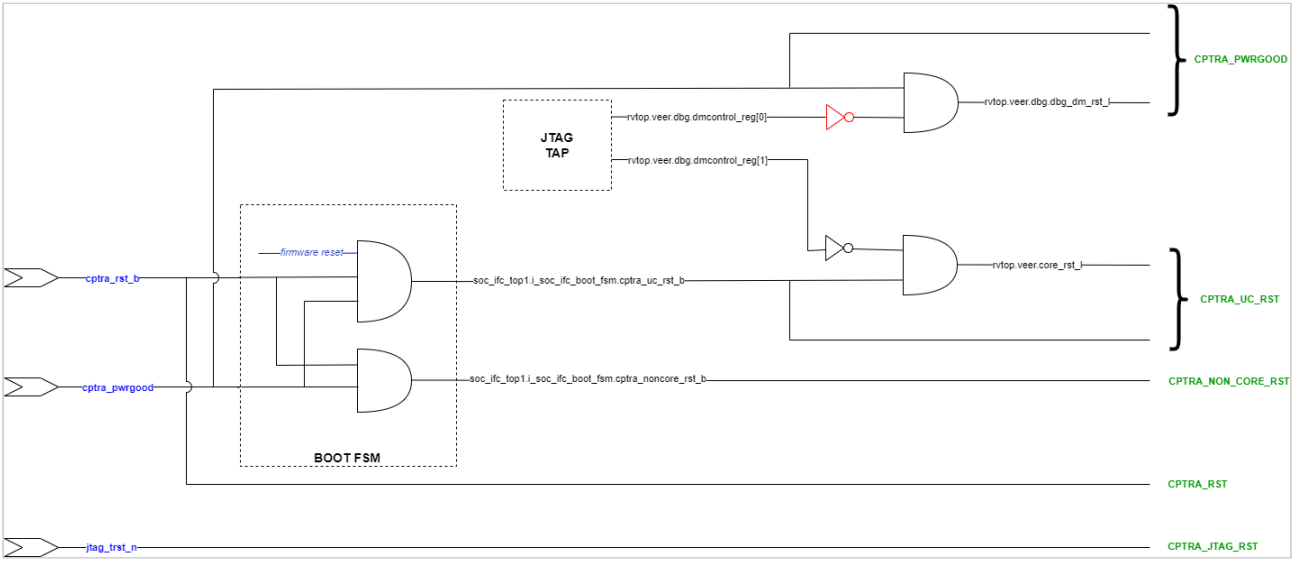
- **Clock-Gating Cell Synchronization Scheme** - Gate the clock of the destination flop when source reset is asserted


RESET1
RESET2
RST
Q
D
CLK
RST
Q
D
CLK
CLK
Clock
Gate
CLK_EN

2 Reset Definitions

Reset Name	Reset Type	Reset Polarity	Definition Point	Reset Generated by
CPTRA_PWRGD	Async	Active Low	cptra_pwrgood	Primary Input
CPTRA_RST	Async	Active Low	cptra_rst_b	Primary Input
CPTRA_UC_RST	Async	Active Low	caliptra_top.soc_ifc_top1.i_soc_ifc_boot_fsm.cptra_uc_rst_b	Generated by Boot FSM
CPTRA_NON_CORE_RST	Async	Active Low	caliptra_top.soc_ifc_top1.i_soc_ifc_boot_fsm.cptra_noncore_rst_b	Generated by Boot FSM
RISCV_VEER_CORE_RST	Async	Active Low	caliptra_top.rvtop.veer.core_rst_l	AND of BOOT_FSM_CPTRA_UC_RST and RISCV_VEER_DBG_CORE_RST *Functionally same as CPTRA_UC_RST*
RISCV_VEER_DBG_DM_RST	Async	Active Low	caliptra_top.rvtop.veer.dbg.dbg_dm_rst_l	AND of CPTRA_PWRGD and a bit controlled from JTAG TAP *Functionally same as CPTRA_PWRGD*
CPTRA_JTAG_RST	Async	Active Low	jtag_trst_n	Primary Input

The reset definitions can be visually represented as shown in the diagram below.


CPTRA_PWRGOOD
rvtop.veer.dbg.dbg_dm_rst_l
JTAG
TAP
rvtop.veer.dbg.dmcontrol_reg[0]
rvtop.veer.dbg.dmcontrol_reg[1]
firmware reset
cptra_rst_b
soc_ifc_top1.i_soc_ifc_boot_fsm.cptra_uc_rst_b
rvtop.veer.core_rst_l
CPTRA_UC_RST
cptra_pwrgood
soc_ifc_top1.i_soc_ifc_boot_fsm.cptra_noncore_rst_b
CPTRA_NON_CORE_RST
BOOT FSM
CPTRA_RST
jtag_trst_n
CPTRA_JTAG_RST

3 RDC false paths

This section needs to be reviewed by Caliptra designers.

Also assuming that the resets controlled from JTAG TAP will not be asserted during functional state.

The below table shows the false paths between various reset groups

from_reset	to_reset	Comment
CPTRA_PWRGD	all other groups	CPTRA_PWRGD is the deepest reset domain, so all RDC paths from CPTRA_PWRGD can be set as false paths
CPTRA_RST	CPTRA_UC_RST	Boot FSM is reset by CPTRA_RST
CPTRA_RST	CPTRA_NON_CORE_RST	Boot FSM is reset by CPTRA_RST
CPTRA_NON_CORE_RST	CPTRA_RST	CPTRA_NON_CORE_RST can be asserted only be asserting CPTRA_RST
CPTRA_NON_CORE_RST	CPTRA_UC_RST	CPTRA_NON_CORE_RST can be asserted only be asserting CPTRA_RST Asserting CPTRA_RST means CPTRA_UC_RST will be asserted

The reset dependencies can be visually represented as explained in Section 5.

4 RDC Crossings Reported

Sl No	from_reset	to_reset	Violation Count
1	CPTRA_UC_RST	CPTRA_NON_CORE_RST	3704
2	CPTRA_UC_RST	CPTRA_PWRGD	2348
4	CPTRA_UC_RST	CPTRA_RST	373
5	CPTRA_RST	CPTRA_PWRGD	673
6	CPTRA_NON_CORE_RST	CPTRA_PWRGD	1899

5 RDC Analysis

5.1 Reset Sequencing Scenarios

The reset defined in Section 1 has the following sequencing phases which are applicable for different reset scenarios - cold boot, cold reset, warm reset and firmware reset. For simplification of our RDC analysis, we are assuming that the following debug registers which are driven from JTAG will not be toggled during functional flow.

- caliptra_top.rvtop.veer.dbg.dmcontrol_reg[1] = 0
- caliptra_top.rvtop.veer.dbg.dmcontrol_reg[0] = 0

In addition to the above, we have set caliptra_top.scan_mode = 0 for our analysis.

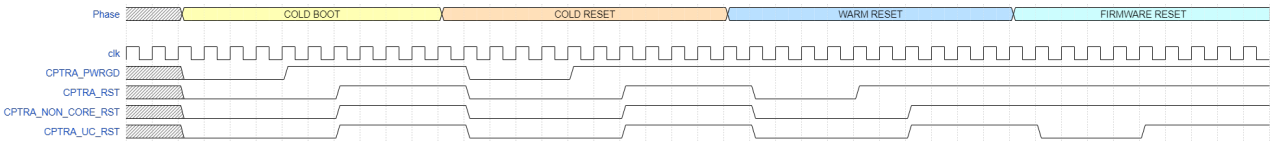
There is an issue in design related to the above JTAG registers. Please see Section 3 and review.

The reset sequencing looks like below.

Reset Sequencing

```
{signal: [
  {name: "Phase",              wave: "x.3.........4.........5.........6.........",
data: ["COLD BOOT", "COLD RESET", "WARM RESET", "FIRMWARE RESET"] },
  {},
  {name: 'clk',                wave:
'p.....................................'},
  {name: 'CPTRA_PWRGD',        wave:
'x.0...1......0...1.........................'},
  {name: 'CPTRA_RST',          wave:
'x.0.....1....0.....1....0...1.............'},
  {name: 'CPTRA_NON_CORE_RST', wave:
'x.0.....1....0.....1....0.....1...........'},
  {name: 'CPTRA_UC_RST',       wave: 'x.0.....1....0.....1....0.....1....0...1....'}
  ],


}
```



5.2 Reset Ordering

The table below defines the order in which reset can get asserted. A ">>" in a cell at row X and column Y indicates that if the reset in row X is asserted, the reset is row Y will also get asserted. For rest of the cells (in which symbol ">>" is not there) the above assumption doesn't hold good and hence the paths between those resets are potential RDC violations. The black cells can be ignored as they are between same resets.

	CPTRA_PWRGOOD	**CPTRA_RST**	**CPTRA_NON_CORE_RST**	**CPTRA_UC_RST**
CPTRA_PWRGOOD		>>	>>	>>
CPTRA_RST			>>	>>
CPTRA_NON_CORE_RST		>>		>>
CPTRA_UC_RST				

There are mainly two groups of reset crossings as identified in the design

1. During WARM_RESET scenario, when CPTRA_RST is asserted, we will have crossings from flops on CPTRA_RST, CPTRA_NON_CORE_RST and CPTRA_UC_RST to flops on CPTRA_PWRGOOD
2. During FIRMWARE_RESET scenario, when CPTRA_UC_RST is asserted, we will have crossings from flops on CPTRA_UC_RST to flops on CPTRA_PWRGOOD, CPTRA_RST and CPTRA_NON_CORE_RST

In order to fix these issues, we can try to gate the clock of the metastable flop when the reset of the source flop is getting asserted.

5.3 RDC crossing examples

Below table captures some examples of RDC crossings which are being reported in the current design.

Sl No	**Source Reset**	**Destination Reset**	**Source Flop**	**Destination Flop**	**Schematic**
1	CPTRA_UC_RST	CPTRA_PWRGOOD	caliptra_top.rvtop.veer.dbg.dbg_state_reg.genblock.dffs.dout[3:0]	caliptra_top.rvtop.veer.Gen_AXI_To_AHB.sb_axi4_to_ahb.buf_writeff.genblock.dffs.genblock.dffs.dout[0]	

Sl No	Source Reset	Destination Reset	Source Flop	Destination Flop	Schematic
2	CPTRA_UC_RST	CPTRA_PWRGOOD	caliptra_top.rvtop.veer.Gen_AXI_To_AHB.lsu_axi4_to_ahb.buf_alignedff.genblock.dffs.genblock.dffs.dout[0]	caliptra_top.soc_ifc_top1.i_soc_ifc_reg.field_storage.CPTRA_HW_ERROR_FATAL.error_code.value[0]	[redacted]
3	CPTRA_UC_RST	CPTRA_RST	caliptra_top.rvtop.veer.Gen_AXI_To_AHB.lsu_axi4_to_ahb.buf_alignedff.genblock.dffs.genblock.dffs.dout[0]	caliptra_top.soc_ifc_top1.i_soc_ifc_reg.field_storage.intr_block_rf.error_intr_en_r.error_bad_fuse_en.value	[redacted]
4	CPTRA_UC_RST	NON_CORE_RST	caliptra_top.rvtop.veer.Gen_AXI_To_AHB.lsu_axi4_to_ahb.buf_alignedff.genblock.dffs.genblock.dffs.dout[0]	caliptra_top.soc_ifc_top1.i_sha512_acc_top.i_sha512_acc_csr.field_storage.LOCK.LOCK.value	[redacted]
5	NON_CORE_RST	CPTRA_PWRGOOD	caliptra_top.soc_ifc_top1.i_ahb_slv_sif_soc_ifc.addr[0]	caliptra_top.soc_ifc_top1.i_soc_ifc_reg.field_storage.CPTRA_FW_EXTENDED_ERROR_INFO[2].error_info.value[31:0]	[redacted]
6	NON_CORE_RST	CPTRA_PWRGOOD	caliptra_top.data_vault1.dv_ahb_slv1.addr[1:0]	caliptra_top.data_vault1.dv_reg1.field_storage.STICKY_DATA_VAULT_ENTRY[0][8].data.value[31:0]	[redacted]

5.4 Solution Space

There are two ways to resolve RDC crossings

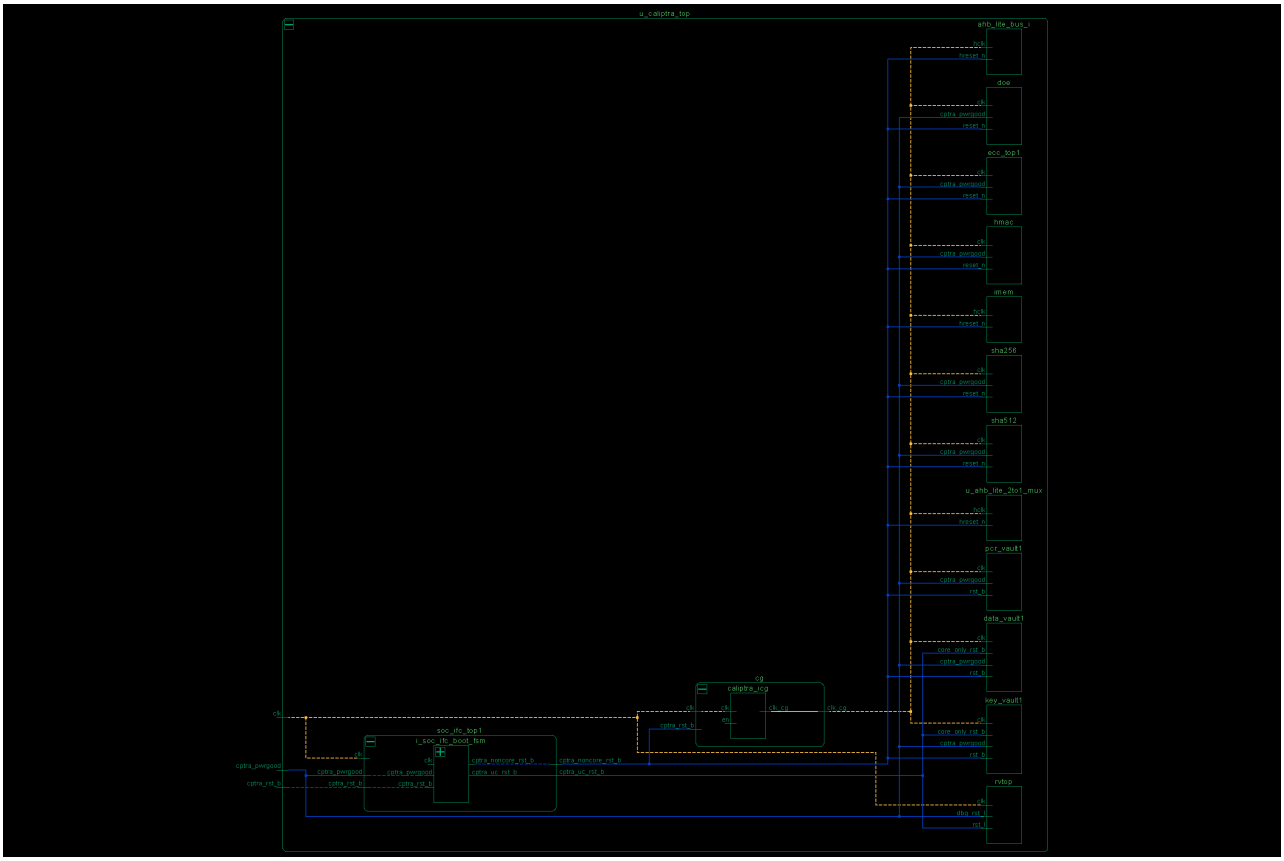
1. Manually verify that all potential crossings that are reported by the tool are ok to waive because either
    a. The crossing is gated by an enable signal which gets functionally activated at the time of source reset assertion, OR
    b. The destination flop going to metastable state will not cause a functional issue
2. Add synchronization in the failing paths so that the destination flops don't go into metastable state. This can be hard as there are a huge number of crossings in the design.
3. Ensure that the clock of the destination flops are gated during the source flop reset assertion window. This will ensure that when source reset is asserted, the destination flop doesn't have any clock edges active around this time thereby eliminating the chance of going to metastable state.

I will explore the solution space mentioned in point 3 above.

Both CPTRA_UC_RST and CPTRA_RST can get asserted while other resets will be in their de-asserted state as mentioned in the sequence in Section 7.1.

Upon analyzing the RDC violations reports, we found that the following modules/instances receive CPTRA_PWRGOOD, CPTRA_RST and NON_CORE_RST and the flops on these resets inside these blocks are destination flops in RDC crossings.

Module	**CPTRA_PWRGOOD**	**CPTRA_RST**	**CPTRA_NON_CORE_RST**
ahb_lite_bus_i			x
data_vault1	x		x
doe	x		x
ecc_top1	x		x
hmac	x		x
key_vault1	x		x
pcr_vault1	x		x
sha256	x		x
sha512	x		x
soc_ifc_top1.i_ahb_slv_sif_soc_ifc			x
soc_ifc_top1.i_mbox			x
soc_ifc_top1.i_sha512_acc_top	x		x
soc_ifc_top1.i_soc_ifc_reg	x	x	
u_ahb_lite_2to1_mux			x
imem			x
rvtop	x		



Two new signals are added in the design to gate the clocks of the destination modules:-

- ***uc_rst_assert_window*** - This signal is driven by Boot FSM just around the firmware reset assertion window as shown below

Boot FSM

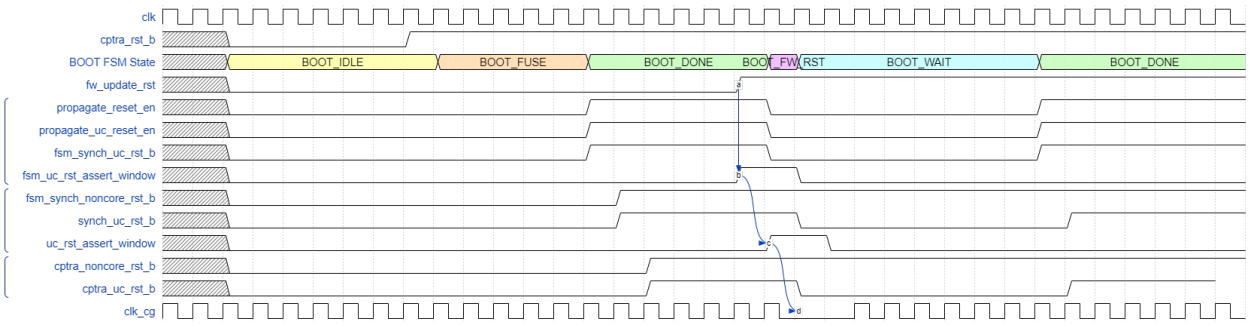
```
{signal: [
  {name: 'clk',                       wave: 'p.....................................'}
,
  {name: 'cptra_rst_b',               wave: 'x.0.....1.............................'}
,
  {name: "BOOT FSM State",            wave: "x.3......4....7.....86.......7......",
data: ["BOOT_IDLE", "BOOT_FUSE", "BOOT_DONE", "BOOT_FW_RST", "BOOT_WAIT", "BOOT_DONE"]
 },
  {name: 'fw_update_rst',             wave: 'x.0................1................',
node: '..................a', },
  ['',
```

```
  {name: 'propagate_reset_en',          wave: 'x.0..........1.....0........1......'}
,
  {name: 'propagate_uc_reset_en',       wave: 'x.0..........1.....0........1......'}
,
  {name: 'fsm_synch_uc_rst_b',          wave: 'x.0..........1.....0........1......'}
,
  {name: 'fsm_uc_rst_assert_window',    wave: 'x.0................1.0.............',
node: '..................b',},
  ],
  ['',
  {name: 'fsm_synch_noncore_rst_b',     wave: 'x.0............1...................'}
,
  {name: 'synch_uc_rst_b',              wave: 'x.0............1.....0........1.....'}
,
  {name: 'uc_rst_assert_window',        wave: 'x.0.................1.0.............',
node: '...................c',},
  ],
  ['',
  {name: 'cptra_noncore_rst_b',         wave: 'x.0.............1..................'}
,
  {name: 'cptra_uc_rst_b',              wave: 'x.0.............1....0........1....'},
  ],

  {name: 'clk_cg',                      wave: 'p...................l.p...........',
node: '.....................d',},
  ],

edge: [
    'a~>b', 'b~>c', 'c~>d',
  ]

}
```



To achieve the above timing, the following changes are done to BOOT FSM (This needs to be reviewed by Caliptra designers)

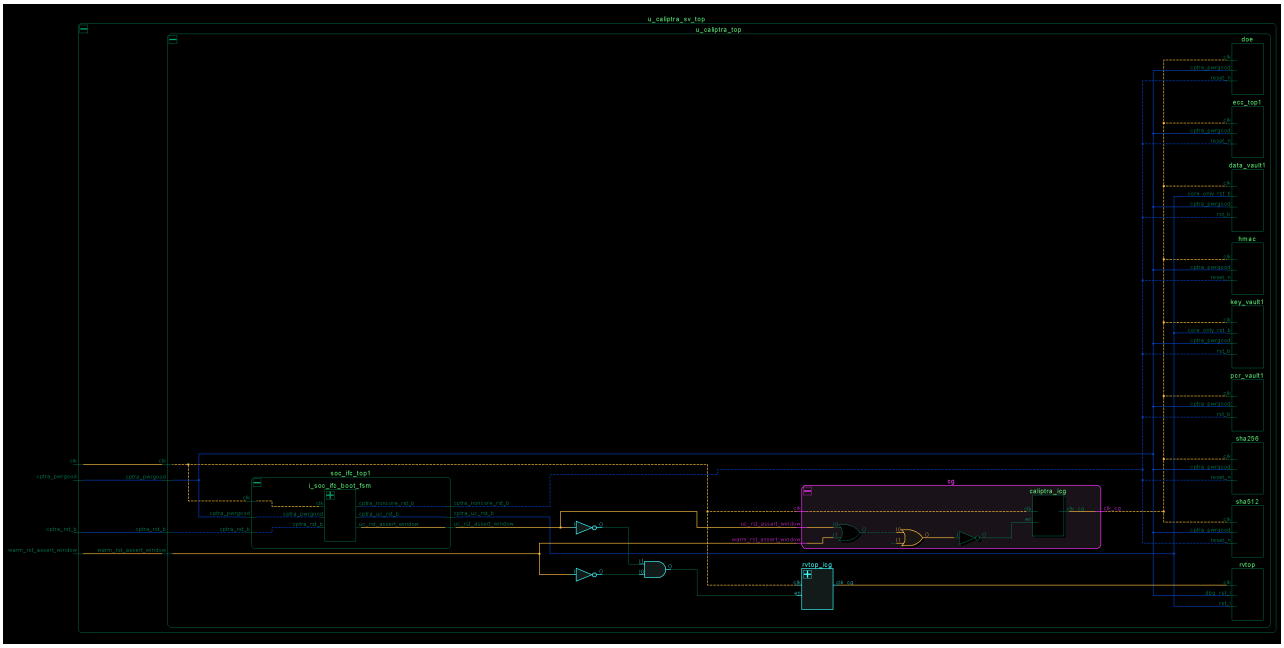
- uc_rst_assert_window is set to 1 in BOOT_DONE state if arc_BOOT_DONE_BOOT_FWRST is set
- uc_rst_assert_window is set to 1 in BOOT_FW_RST state

~~*In addition to above, a new state is required to be introduced after BOOT_FW_RST so that uc_rst_assert_window is kept at 1 for at-least two cycles after UC reset is asserted. This is not true as of today and need FSM update.*~~

- ***warm_rst_assert_window*** - During warm reset assertion, Caliptra IP has no control on its assertion timing since this is externally driven. In order to prevent the RDC crossings during warm reset assertion, we have added a new top level port called *warm_rst_assert_window* and needs to be driven from SOC. This signal needs to be asserted (set to 1) at-least 1 cycle before warm reset (cptra_rst_b) is asserted and needs to held low (at 0) for at-least 2 cycles after the warm reset assertion edge.

Using the above two signals, we have modified the clock gating structure as shown below. Specifically the changes are as follows:-

- [Highlighted in pink] The clock enable logic at *caliptra_top.cg.caliptra_icg* has been modified to take into account the above two signals. When any of the assert_window signals are 1, the clock is gated.
- [Highlighted in blue] The clock feeding to rv_top is gated when either of the assert_window signals is high



Whether the temporary gating of destination clocks is functionally ok needs to be confirmed by Caliptra IP team. More specifically, we have the below assumption baked in while

- During firmware reset, SW has to ensure that there no pending transactions from RISCV side so that clock gating gated for couple of cycles should not cause in-flight transactions to get missed.
- SOC has to drive warm_rst_assert_window before warm reset is asserted. Whether every SOC can do this is another question to be looked after. For SW initiated SOC warm reset, this might be possible but for hardware event generated warm resets, the HW logic driving them should be able to generate the same. I am not sure whether every SOC design will be able to accommodate this.

RDC violations full log - /home/scratch.amullick_mobile/psc/caliptra_ss_tree/tot_may10/hw/nvip/outdir/ip/caliptra/1.0/syn/caliptra_top/t264/rdc_run1/fv/rtl_rdc/NV_caliptra_top.rdc_struct_filt.report

6 Issues Seen in Design

1. ~~Even though CPTRA_PWRGD is defined to be the deepest reset in the design, the Boot FSM is NOT consuming CPTRA_PWRGD to reset the entire design. In current implementation, asserting CPTRA_PWRGD (setting it to 0) will not assert BOOT_FSM_CPTRA_UC_RST and BOOT_FSM_NON_CORE_RST which doesn't seem to be correct.~~
2. ~~Primary input reset pin *cptra_pwrgood* needs to be synchronized with *clk* before using in the design~~
3. ~~Primary input reset pin *cptra_rst_b* needs to be synchronized with *clk* before using in the design~~
4. ~~Primary input reset pin *jtag_trst_n* needs to be synchronized with *jtag_tck* before using in the design~~

Sl no	Issue
1	In *caliptra_top.rvtop.veer.dbg* module, we have the below assign statement **assign dbg_dm_rst_l = dbg_rst_l & (dmcontrol_reg[0] | scan_mode);** The reset value of *dmcontrol_reg[0]* is 0 and hence the above line is blocking the propagation of *dbg_rst_l* to *dbg_dm_rst_l* causing ASYNC flops to never get its reset.