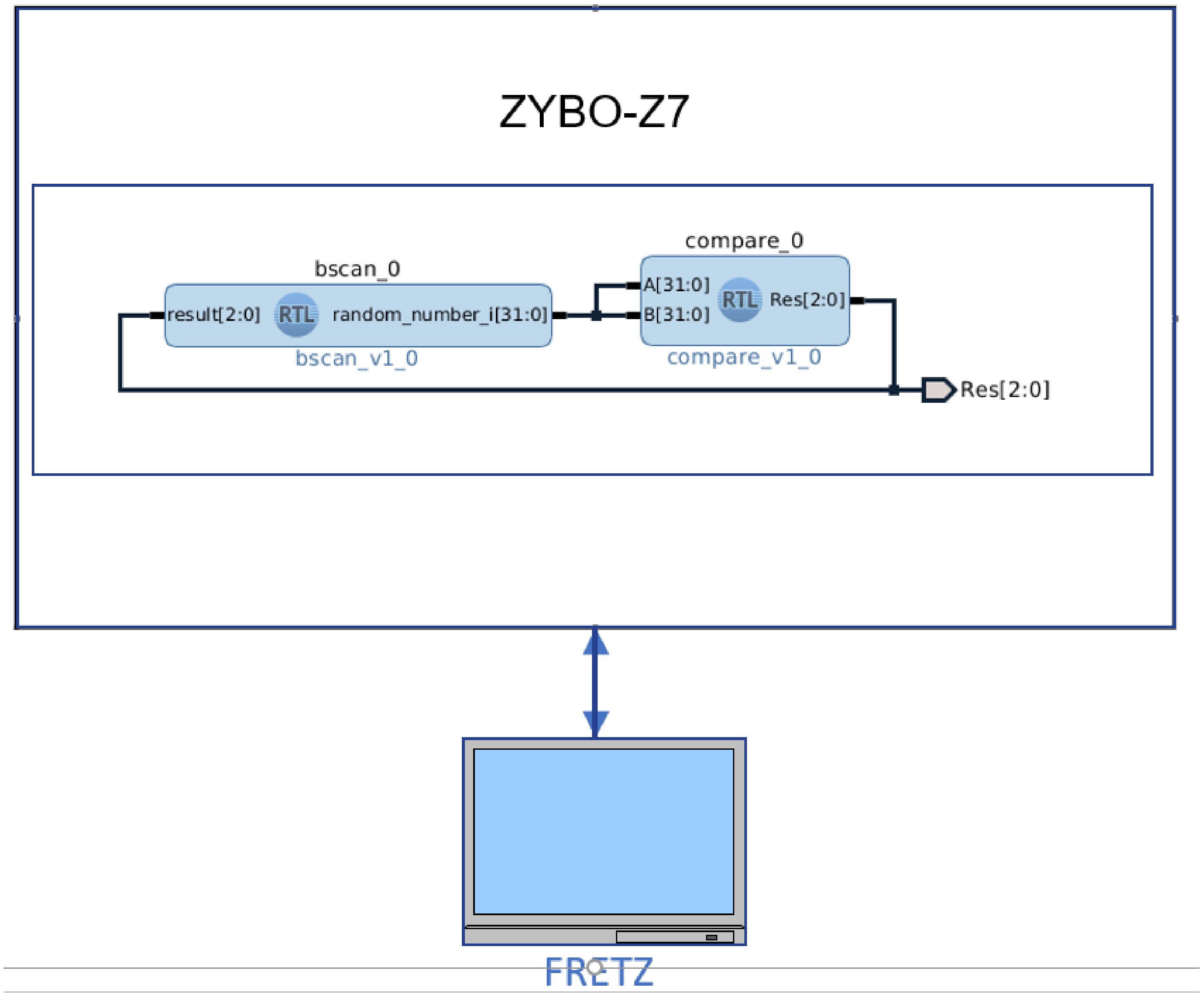
# CDS206 Lab2: Reducing the architectural vulnerability factor of FPGA circuits with TMR

## Introduction

In our first lab, we developed a 32-bit comparator and a 32-bit adder on the FPGA to perform fault injection and find the circuits' Architectural Vulnerability (AVF). We found that some bitflips in the FPGA's configuration memory resulted in a circuit error.

We used JTAG/BSCAN to send input to the circuit and get the result. In FREtZ, we compared the result of the circuit with a golden reference to check if an injected fault resulted in an error.

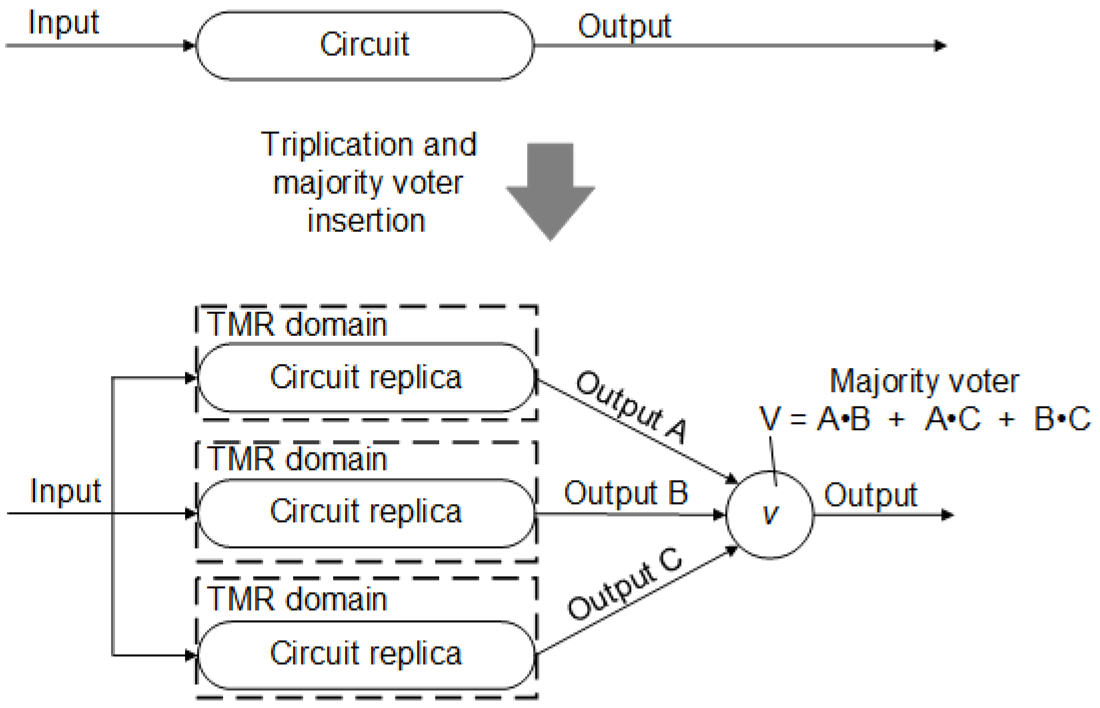


In this lab, we will develop a Triple Modular Redundant (TMR) adder Design Under Test (DUT) and perform fault injection to investigate how TMR improves the circuit's AVF.

Hardware TMR (Triple Modular Redundancy) is a technique that uses three identical hardware modules to improve fault tolerance. In hardware TMR, each module is designed to perform the same function, and each module has its own set of inputs and outputs.

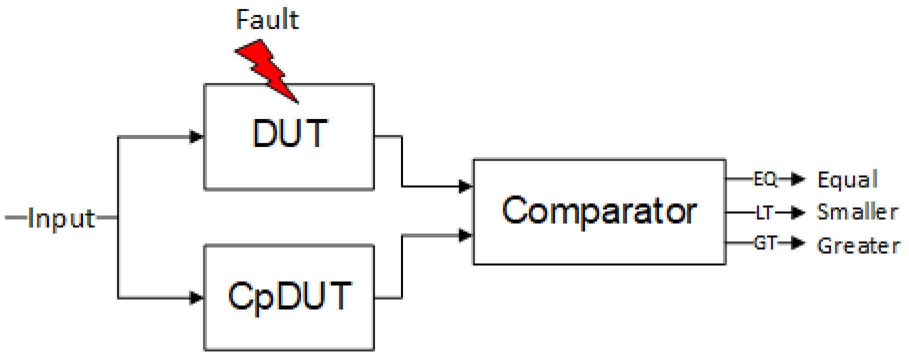
The inputs are usually synchronized so that they are applied to each module at the same time. The outputs of the three modules are then compared to detect any discrepancies. If the outputs of any two modules match, then the output is considered to be correct, and the system continues to operate as expected. If the outputs do not match, the system enters a "voting" phase, where a majority vote is taken to determine which output is correct.

Hardware TMR is commonly used in mission-critical applications where reliability is of utmost importance. Examples of such applications include avionics, military systems, and medical devices. Hardware TMR provides a higher level of redundancy compared to software TMR, which relies on redundant software modules running on a single processor. Hardware TMR is often implemented using field-programmable gate arrays (FPGAs) or application-specific integrated circuits (ASICs).



However, we will perform fault injection through the following Duplication With Comparison (DWC) paradigm.

- Implement two copies of the DUT  DUT and Copy DUT (CpDUT) or otherwise golden DUT.
- Connect the same input to the DUT and CpDUT
- Connect the output of the DUT and CpDUT in a comparator
- Inject a fault ONLY into the DUT
- Provide the same input to the DUT and CpDUT
- Read the output of the comparator
  - If `Equal != 1`, then the fault resulted in an error



But how do we inject faults only into the configuration memory of the DUT?

We do this through Vivado's **Placement Constraints.** Placement constraints are applied to cells to control their locations within the device --> read pages 139-143 in https://docs.xilinx.com/v/u/2016.1-English/ug903-vivado-using-constraints. In simple words, we instruct Vivado to implement the DUT in a particular place (commonly called placement block or in short PBLOCK) in the FPGA and then we inject faults into the configuration frames of the PBLOCK.

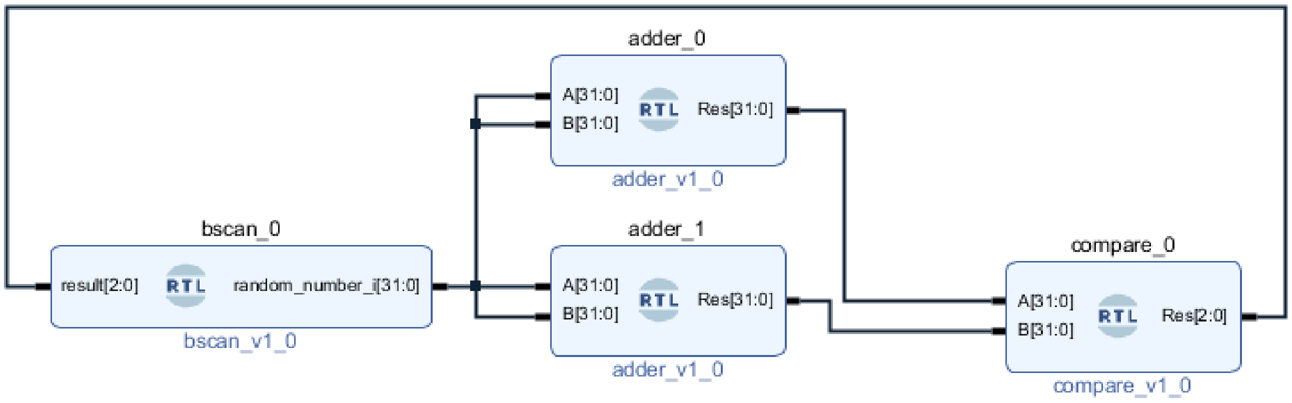
## Lab exercise

### AVF of the simplex adder

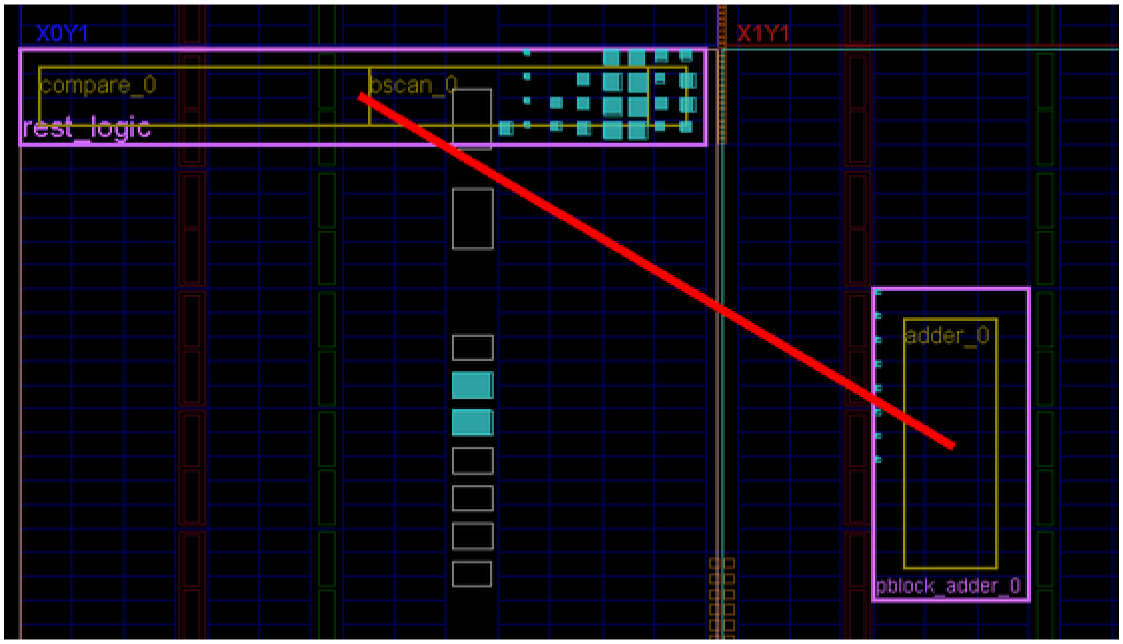
Initially, we will implement the simplex adder (not TMR) in a DWC structure and perform fault injection as described in the introduction.

The DUT is adder_0, and the cpDUT is adder_1.

Please use the source code and tutorial from lab_1 to develop the circuit and perform fault injection with FREtZ to evaluate the DUT's AVF.



As mentioned, one important step for the DWC structure is to place the DUT (i.e., adder_0) into a PBLOCK (namely pblock_adder_0) and the cpDUT(i.e., adder_1), compare_0 and bcan_0 into another PBLOCK(namely rest_logic)



This can be done by adding the following constraints in the `constraints.xdc` file:

```
set_property BITSTREAM.SEU.ESSENTIALBITS yes [current_design]
set_property BITSTREAM.GENERAL.PERFRAMECRC YES [current_design]
set_property BITSTREAM.CONFIG.INITSIGNALSERROR DISABLE [current_design]

create_pblock pblock_adder_0
add_cells_to_pblock [get_pblocks pblock_adder_0] [get_cells -quiet
[list design_1_i/adder_0]]
resize_pblock [get_pblocks pblock_adder_0] -add {SLICE_X26Y27:
SLICE_X31Y39}

create_pblock rest_logic
add_cells_to_pblock [get_pblocks rest_logic] [get_cells -quiet [list
design_1_i/adder_1]]
add_cells_to_pblock [get_pblocks rest_logic] [get_cells -quiet [list
design_1_i/bscan_0]]
add_cells_to_pblock [get_pblocks rest_logic] [get_cells -quiet [list
design_1_i/compare_0]]
resize_pblock [get_pblocks rest_logic] -add {SLICE_X0Y46:SLICE_X21Y49}
```

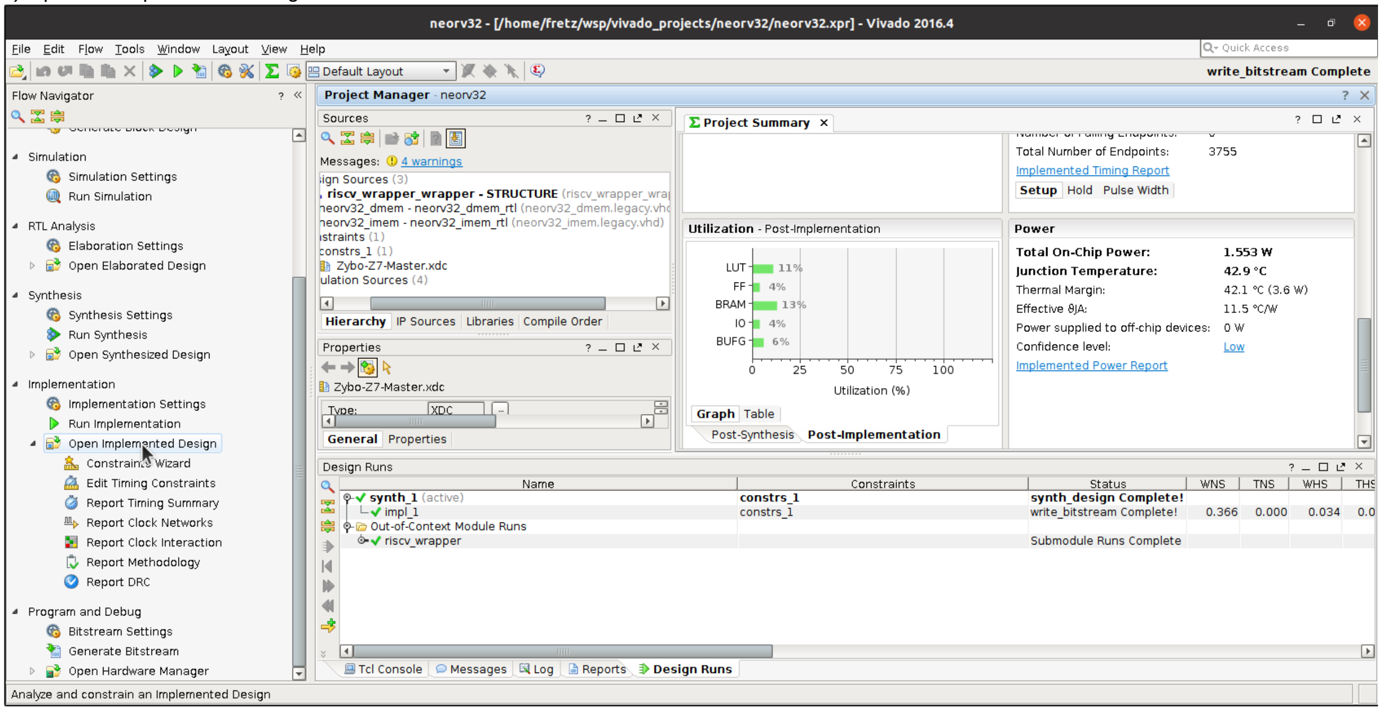
**AVF of the NEORV32's multiplier unit**

In this exercise, we will implement the NEORV32 and then find the AVF of its multiplier unit through fault injection.

Follow the tutorial `NEORV-32-tutorial.pdf` from CDS105 to implement the NEORV RISC-V.

When you implement the design, please follow the steps below to place the ALU of NEORV32 into a PBLOCK to perform targeted fault injection.

1) Open the implemented design



Messages: 4 warnings
ign Sources (3)
riscv_wrapper_wrapper - STRUCTURE (riscv_wrapper_wra
neorv32_dmem - neorv32_dmem_rtl (neorv32_dmem.legacy.vhd
neorv32_imem - neorv32_imem_rtl (neorv32_imem.legacy.vhd)
straints (1)
constrs_1 (1)
Zybo-Z7-Master.xdc
ulation Sources (4)

Zybo-Z7-Master.xdc
Type: XDC

Total Number of Endpoints: 3755
Implemented Timing Report

**Power**

| | |
|---|---|
| **Total On-Chip Power:** | **1.553 W** |
| **Junction Temperature:** | **42.9 °C** |
| Thermal Margin: | 42.1 °C (3.6 W) |
| Effective ϑJA: | 11.5 °C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

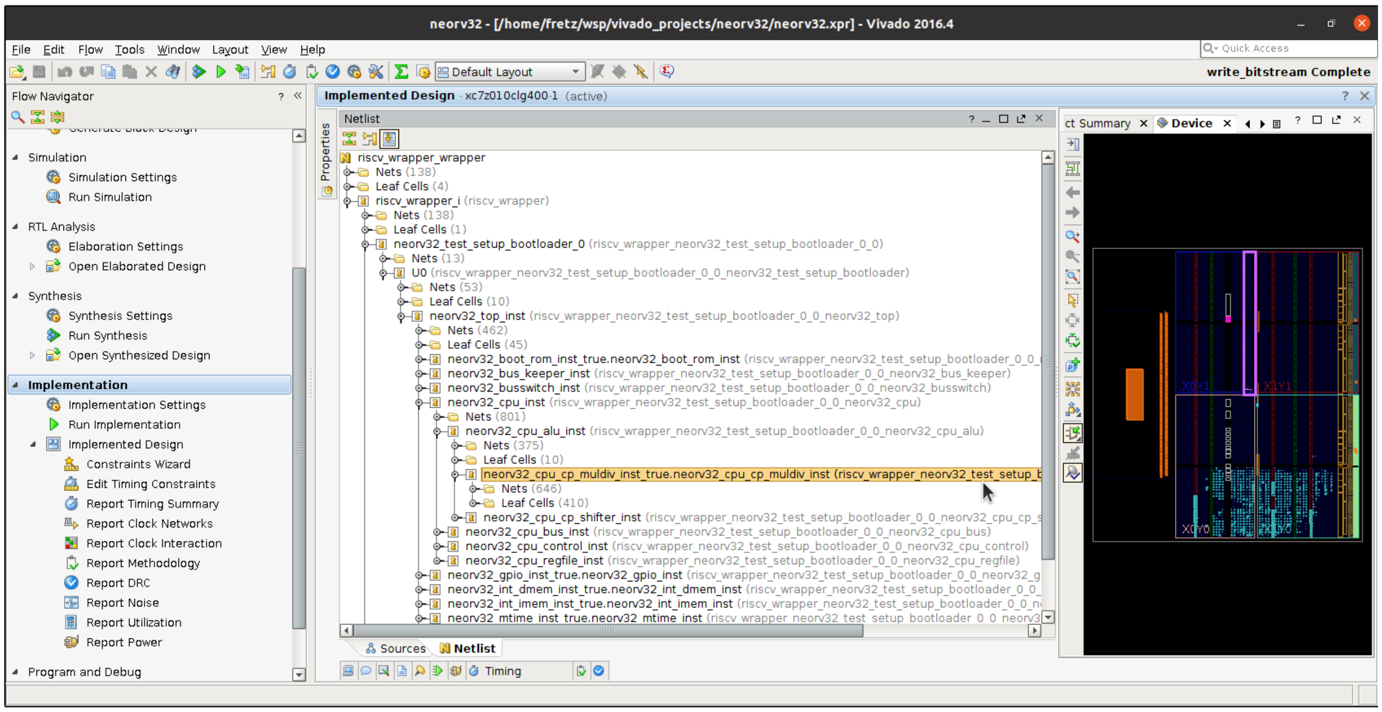
Implemented Power Report

Design Runs

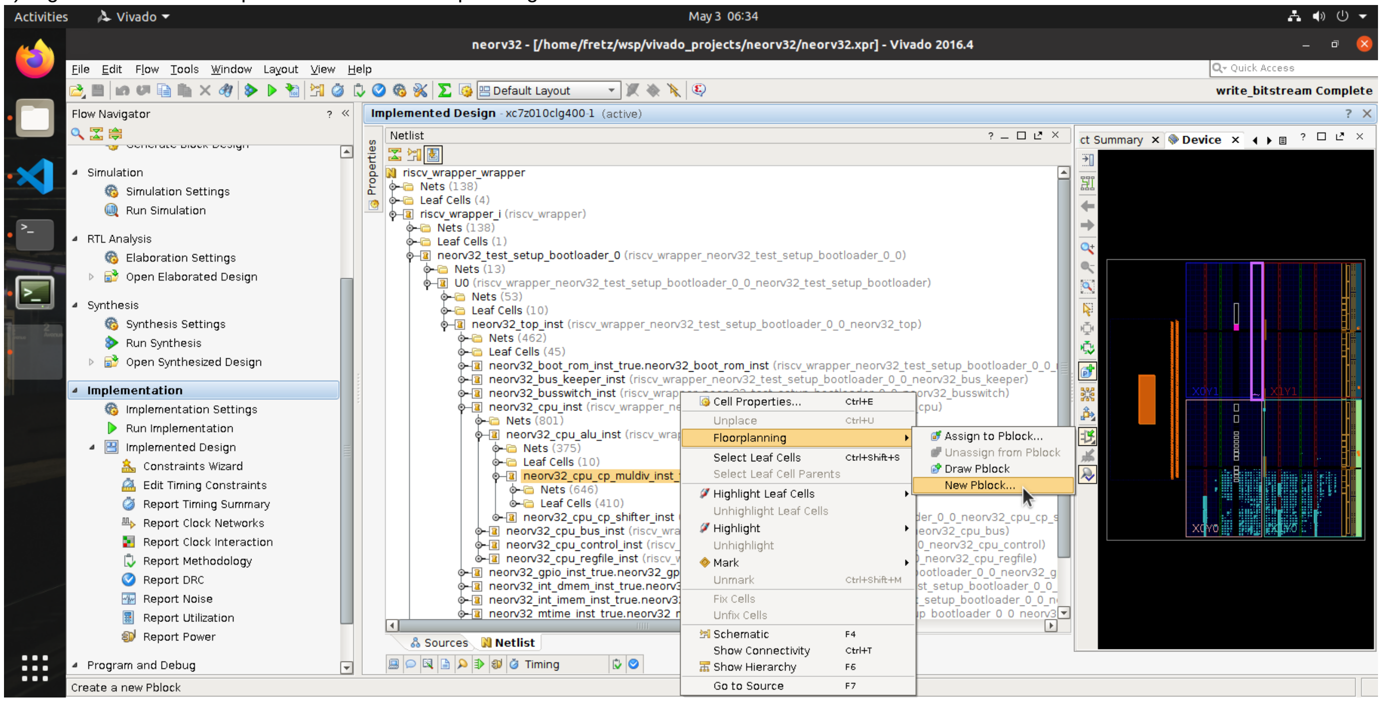
| Name | Constraints | Status | WNS | TNS | WHS | THS |
|---|---|---|---|---|---|---|
| **synth_1** (active) | **constrs_1** | **synth_design Complete!** | | | | |
| impl_1 | constrs_1 | write_bitstream Complete! | 0.366 | 0.000 | 0.034 | 0.0 |
| Out-of-Context Module Runs | | | | | | |
| riscv_wrapper | | Submodule Runs Complete | | | | |

Analyze and constrain an Implemented Design

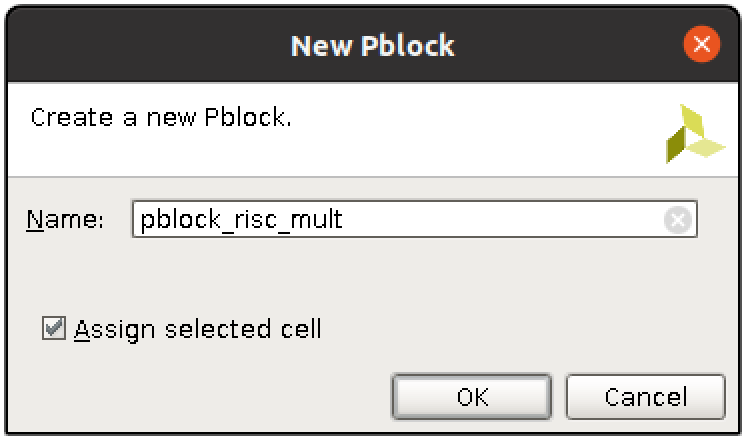
2) Find from the netlist the multiplier unit.


neorv32 - [/home/fretz/wsp/vivado_projects/neorv32/neorv32.xpr] - Vivado 2016.4
File Edit Flow Tools Window Layout View Help
write_bitstream Complete
Flow Navigator
Implemented Design - xc7z010clg400-1 (active)
Netlist
riscv_wrapper_wrapper
neorv32_cpu_cp_muldiv_inst_true.neorv32_cpu_cp_muldiv_inst (riscv_wrapper_neorv32_test_setup_b
Sources
Netlist
Device

3) Right-click on the multiplier and choose  Floorplanning  New Pblock



4) Provide the following name to the PBLOCK

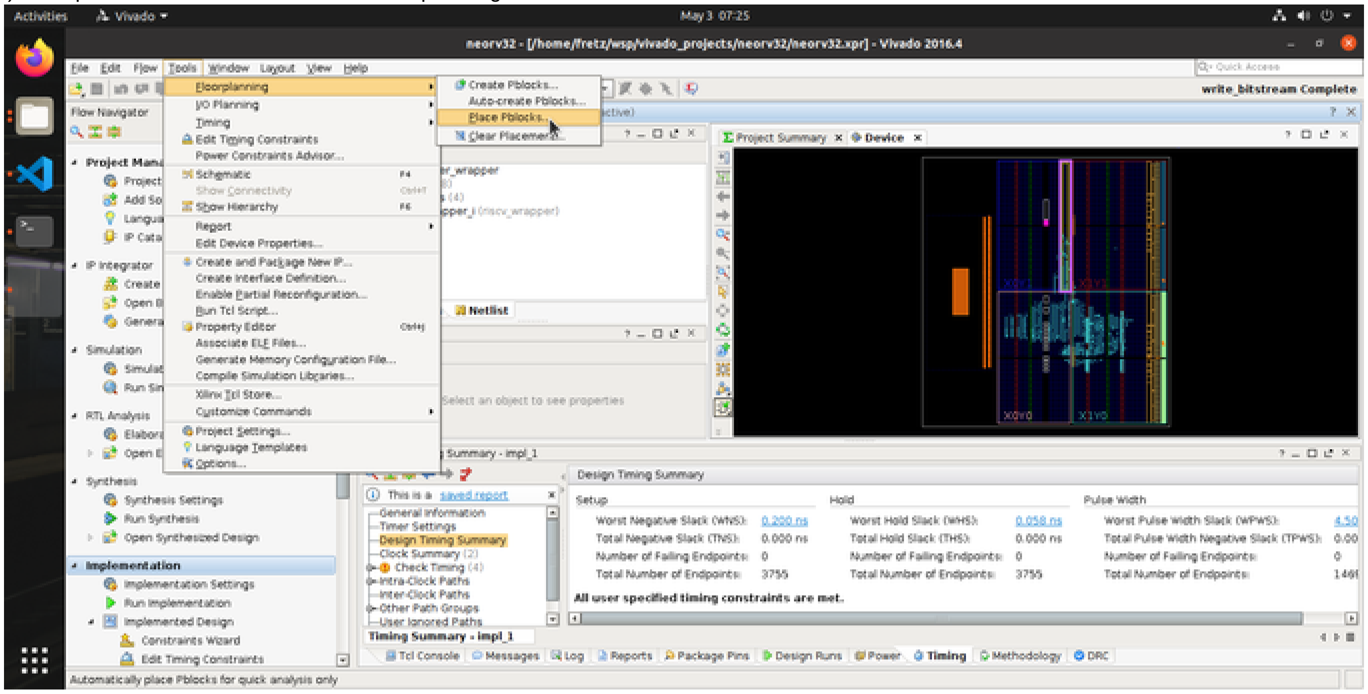

New Pblock

Create a new Pblock.

Name: pblock_risc_mult

☑ Assign selected cell

5) Then press on Vivado's menu Tools  Floorplanning  Place Pblocks…


neorv32 - [/home/fretz/wsp/vivado_projects/neorv32/neorv32.xpr] - Vivado 2016.4

write_bitstream Complete

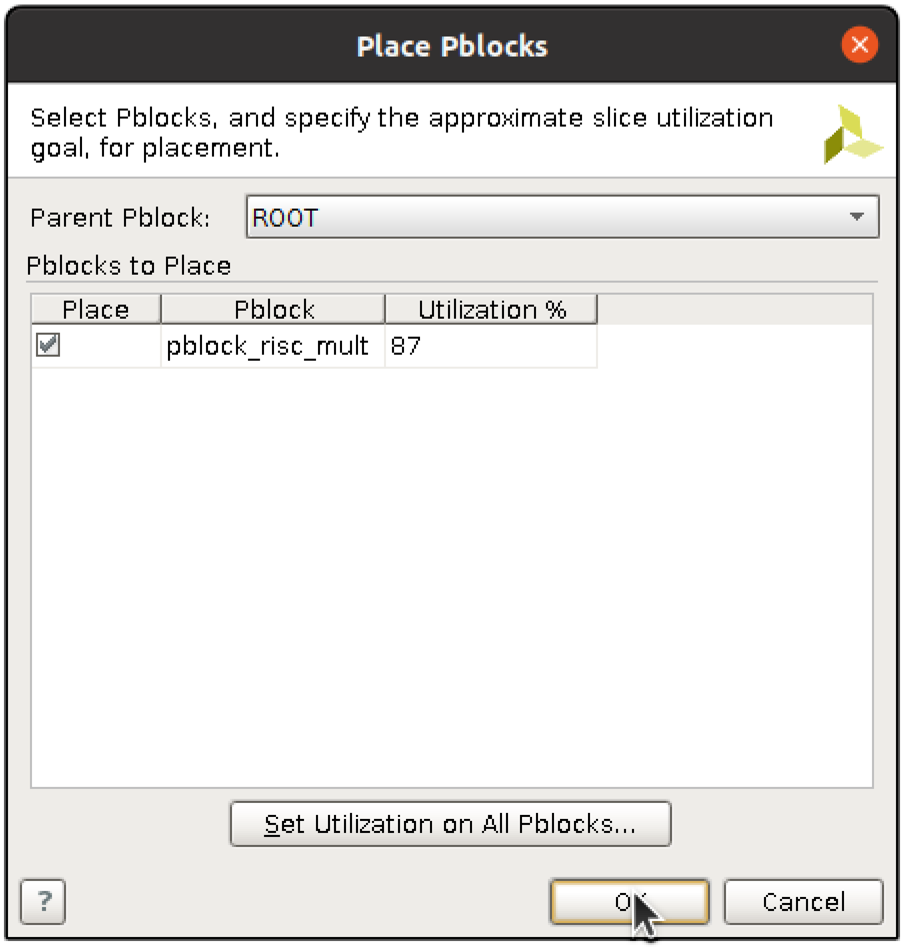
Design Timing Summary

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 0.200 ns | Worst Hold Slack (WHS): | 0.058 ns | Worst Pulse Width Slack (WPWS): | 4.50 |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.00 |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 3755 | Total Number of Endpoints: | 3755 | Total Number of Endpoints: | 146 |

All user specified timing constraints are met.

Automatically place Pblocks for quick analysis only

6) Press OK. This will automatically place the pblock_risc_mult in an optimal position in the FPGA.


Place Pblocks
Select Pblocks, and specify the approximate slice utilization goal, for placement.
Parent Pblock: ROOT
Pblocks to Place
Set Utilization on All Pblocks...
?
OK
Cancel

|Place|Pblock|Utilization %|
|---|---|---|
|✓|pblock_risc_mult|87|

7) Open the constraints file and adjust the position block as follows:

```
resize_pblock [get_pblocks pblock_risc_mult] -add {SLICE_X18Y50:
SLICE_X21Y99}
```

8) Next, we must write software on the RISCV that runs multiplications. After injecting a fault, we will send two numbers to be multiplied through UART from FREtZ and check if the result is correct. In this way, we will calculate the AVF of RISC-V

The source code of the mmult application can be found in

```
lab2_c/srcs/risc_v_mmult_app/mmult
```

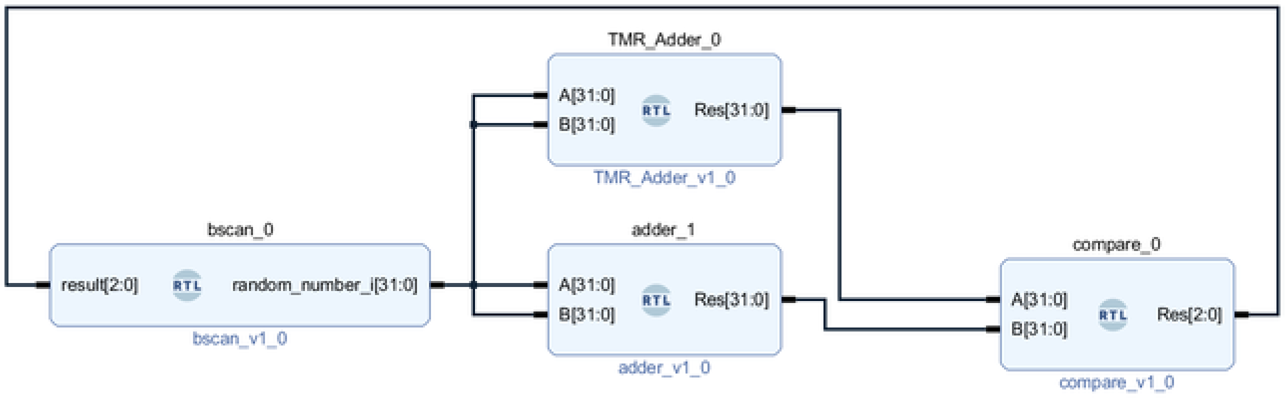
The source code for FREtZ can be found

```
lab2_c\srcs\fretz_1
```

**Homework: AVF of the TMR adder**

Please find with fault injection the AVF of adder_0 when it is Radiation Hardened By Design (RHBD) with TMR.

The design is identical with that of the simplex adder (Fig.1). However, the DUT (i.e., adder_0) is TMR. The TMR_Adder_0 should follow a hierarchical implementation.



Please complete the code of the voter

```
-- This a voter that will be used in the TMR_Adder_0
-- The voter circuit votes upon the inputs from three copies of the
simpex circuit
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity voter is
  Port (    Res1: IN UNSIGNED(31 DOWNTO 0);
            Res2: IN UNSIGNED(31 DOWNTO 0);
            Res3: IN UNSIGNED(31 DOWNTO 0);
            Res : OUT UNSIGNED(31 DOWNTO 0)
        );
end voter;

architecture Behavioral of voter is

signal #### ADD CODE HERE ####;

begin

Res<= #### ADD CODE HERE ####;

end Behavioral;
```

Please complete the code of the hierarchical design

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

entity TMR_Adder is
  Port ( A: in UNSIGNED(31 DOWNTO 0);
        B: IN UNSIGNED(31 DOWNTO 0);
        Res: OUT UNSIGNED(31 DOWNTO 0));
end TMR_Adder;

architecture Behavioral of TMR_Adder is
-- We will use three copies of the simplex adder to implement the TMR
adder
component adder is
  Port (A: in UNSIGNED(31 DOWNTO 0);
        B: IN UNSIGNED(31 DOWNTO 0);
        Res: OUT UNSIGNED(31 DOWNTO 0)
         );
end component;

component voter is
  Port (    Res1: IN UNSIGNED(31 DOWNTO 0);
            Res2: IN UNSIGNED(31 DOWNTO 0);
            Res3: IN UNSIGNED(31 DOWNTO 0);
            Res : OUT UNSIGNED(31 DOWNTO 0)
        );
end component;
attribute dont_touch : string;
signal A0, A1, A2:UNSIGNED(31 DOWNTO 0);
signal B0, B1, B2:UNSIGNED(31 DOWNTO 0);
signal R0, R1, R2:UNSIGNED(31 DOWNTO 0);
-- We use the attribute dont_touch to instruct Vivado to NOT remove
redundant logic
attribute dont_touch of A0: signal is "true";
attribute dont_touch of A1: signal is "true";
attribute dont_touch of A2: signal is "true";
attribute dont_touch of B0: signal is "true";
attribute dont_touch of B1: signal is "true";
attribute dont_touch of B2: signal is "true";
attribute dont_touch of R0: signal is "true";
attribute dont_touch of R1: signal is "true";
attribute dont_touch of R2: signal is "true";

begin
A0<=### ADD CODE HERE ###;
A1<=### ADD CODE HERE ###;
A2<=### ADD CODE HERE ###;
```

```
B0<=### ADD CODE HERE ###;
B1<=### ADD CODE HERE ###;
B2<=### ADD CODE HERE ###;

adder0:adder Port MAP (
        ### ADD CODE HERE ###
        );

adder1:adder Port MAP (
       ### ADD CODE HERE ###
        );

adder2:adder Port MAP (
       ### ADD CODE HERE ###
        );
voter_inst:voter Port Map(
     ### ADD CODE HERE ###
);
end Behavioral;
```

As mentioned we need to place the TMR adder into a PBLOCK in order to perform targetted fault injection. Otherwise we will inject faults into the components that are not part of the DUT.

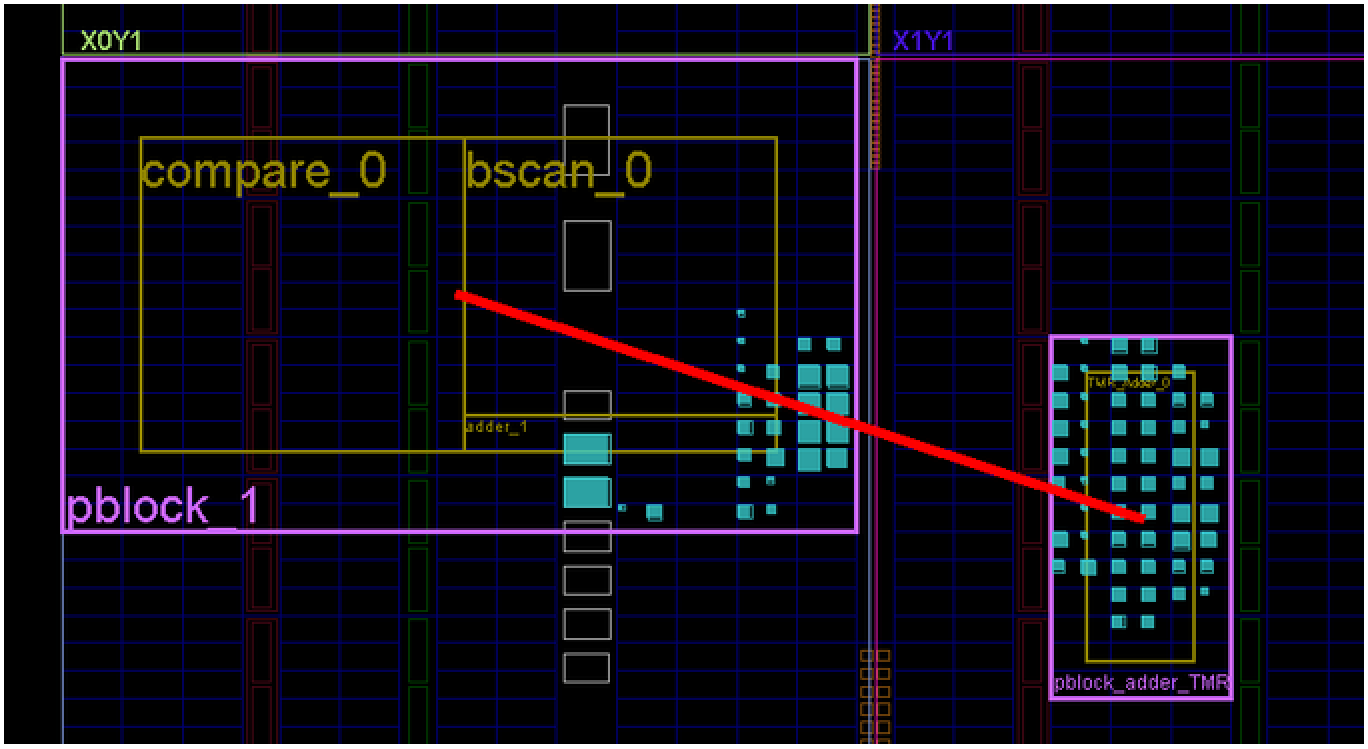
Please use the following constraints for placement.

```
set_property BITSTREAM.SEU.ESSENTIALBITS yes [current_design]
set_property BITSTREAM.GENERAL.PERFRAMECRC YES [current_design]
set_property BITSTREAM.CONFIG.INITSIGNALSERROR DISABLE [current_design]

create_pblock pblock_adder_TMR
add_cells_to_pblock [get_pblocks pblock_adder_TMR] [get_cells -quiet
[list design_1_i/TMR_Adder_0]]
resize_pblock [get_pblocks pblock_adder_TMR] -add {SLICE_X26Y27:
SLICE_X31Y39}

create_pblock pblock_1
add_cells_to_pblock [get_pblocks pblock_1] [get_cells -quiet [list
design_1_i/adder_1]]
add_cells_to_pblock [get_pblocks pblock_1] [get_cells -quiet [list
design_1_i/bscan_0]]
add_cells_to_pblock [get_pblocks pblock_1] [get_cells -quiet [list
design_1_i/compare_0]]
resize_pblock [get_pblocks pblock_1] -add {SLICE_X0Y33:SLICE_X21Y49}
```

You should get the following floorplanning/placement


X0Y1
X1Y1
compare_0
bscan_0
adder_1
pblock_1
TMR_Adder_0
pblock_adder_TMR