Department of Engineering and Information Science

Course of Embedded Systems

NON VOLATILE EMULATOR FOR FPGA

Caronti Luca 192298
Ruffini Simone 188101

Accademic year 2019/2020

Contents

Abstract

The goal of this project was to create a Non Volatile Emulator (NVE), written in VHDL for FPGAs. The theoretical concept was taken from paper *Emulating Intermittent Non-Volatile Computing Systems Using Off-the-shlef FPGAs* of Davide Brunelli and Kasım Sinan Yıldırım.
There are 4 main modules:

- **Power Approximator:** It's basically a counter that counts how many clock cycles each power state is enable.
- **Instant Power Calculator:** Transforms Power Approximator counters into a Power value.
- **Intermittency Emulator:** Emulates the intermittency due to lack of energy.
- **NV Register Emulator:** Emulates a non volatile register as an add-on for volatile registers.

These 4 modules are needed to emulate the behavior of the entity under test. In this case it's a simple **adder** that reads a value from BRAM, adds one to this value, saves it's again in the BRAM, and so on.

Nomenclature

- **`PA`** : power approximator
- **`IE`** : intermittency emulator
- **`IPC`** : instant power calculator
- **`NVR`** : non volatiler register
- **`VR`** : volatile register
- **`NVRE`** : non volatiler register emulator
- **BRAM**: block RAM

Chapter 1

Project Overview

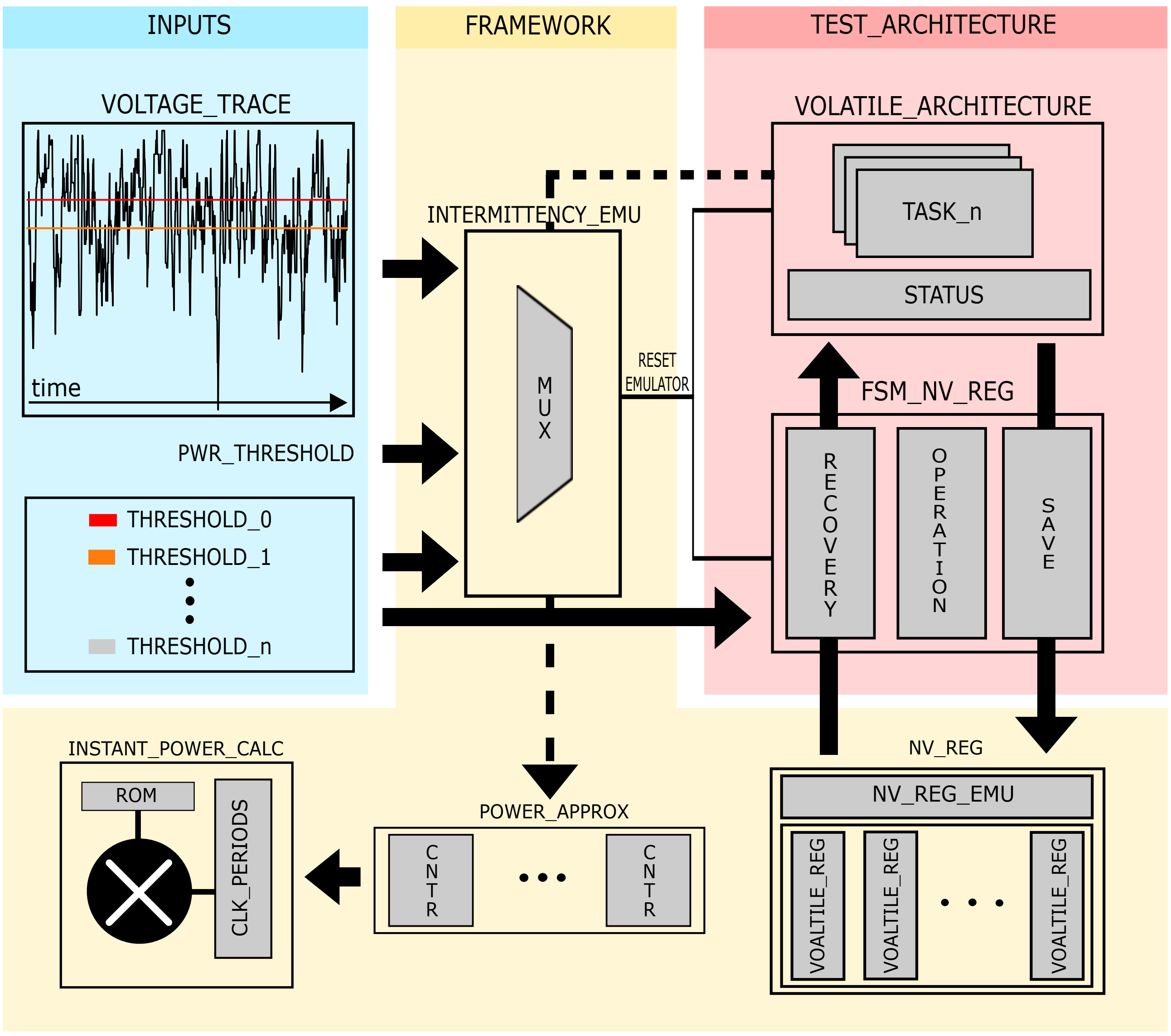


Figure 1.1: General overview of the whole project

The project is divided in two parts: the framework and the test architecture. The main goal has been to develop the framework for the transient computing model, but to test such framework a test architecture was implemented.

1.1 Framework

The framework is composed of four entities:

- **IE** : The intermittency emulator has the task to simulate power shortages in the test architecture. Such power shortages are generated by defining a threshold (**PWR_THRESHOLD**) that is the minimum voltage level required by the architecture to run. An input voltage trace is used to simulate the voltage level available by test architecture, when such voltage gets below **PWR_THRESHOLD** the module triggers the **RESET_EMULATION** signal that will shutdown the test architecture.
  Other threshold values can be defined if the test architecture needs them for power management (the **IE** is responsible to generate signals for such)

- **NVR** : The non volatile register is the memory that allows the transient model to be implemented. Such component is made of two different parts:

  - **NVRE** : The non volatile register emulator is the component that implements the usage protocol of volatile memories. The important aspect of this entity is that
    it does not transforms volatile memories in non-volatile ones but implements the usage protocol to make them appear as non-volatile
    that means that if the protocol is not respected the behaviour of the **NVR** is not logically correct.
  - **VOLATILE_REGISTER** : this is a normal volatile **BRAM** memory that is dependent on the FPGA platform.

  So the **NVR** is a wrapper that encloses the **NVRE** and the **VOLATILE_REGISTER** for ease of use of the end user.
  $\Rightarrow$ Such wrapper deeply relies on the type of volatile memory that the FPGA makes available and this means that different memories require different wrappers. So: for each type of memory used a specific wrapper must be wrote, keeping as reference the one implemented in this framework. $\Leftarrow$

- **PA** : The power approximator calculates the total usage of each power component (in terms of active clock cycles) of test architecture during the test. Such number is then used by the **IPC** to calculate the used energy by each component or by the whole architecture. Each power component corresponds to an entity in the system for which a value E_{clk} (energy per clock cycle) is defined. Typically the test architecture is made of such components and it tells the **PA** when they are active through a status array called **POWER_STATES**.

- **IPC** : The intermittency power calculator takes the number of clocks cycles an entity was active and computes the energy used by such entity multiplying its number with the respective E_{clk}. If the **IPC** is provided with regularly timed active clk cycles deltas, an instant energy value can be calculated.

1.2 Test Architecture

For this project the test architecture is composed of two entities:

- **VOLATILE_ARCHITECTURE**: in this case a simple adder that increments by one its previous value. After each increment the value is saved in a local register (**BRAM**) and this lasts 2 clock cycles, resulting in increments every 3.
  Such architecture loses all its status (adder value) when the **RESET_EMULATOR** triggers (a shutdown event). When power goes up the last stored state, in the **NVR** , is restored and the adder continues its operation.

- **FSM_NV_REG**: The finite state machine of the non volatile register monitors the thresholds of the system and manages the volatile architecture process. It regulates the data recovery, data save and operation time of the volatile architecture by applying some policies that depend on the thresholds. This entity has the task to make the transient computing possible, different policies define the goodness of the system and its data recovery capabilities. Like the volatile architecture this entity loses its state when there is a power shortage.

Chapter 2

Architecture

2.1 Power Approximator

2.1.1 General description

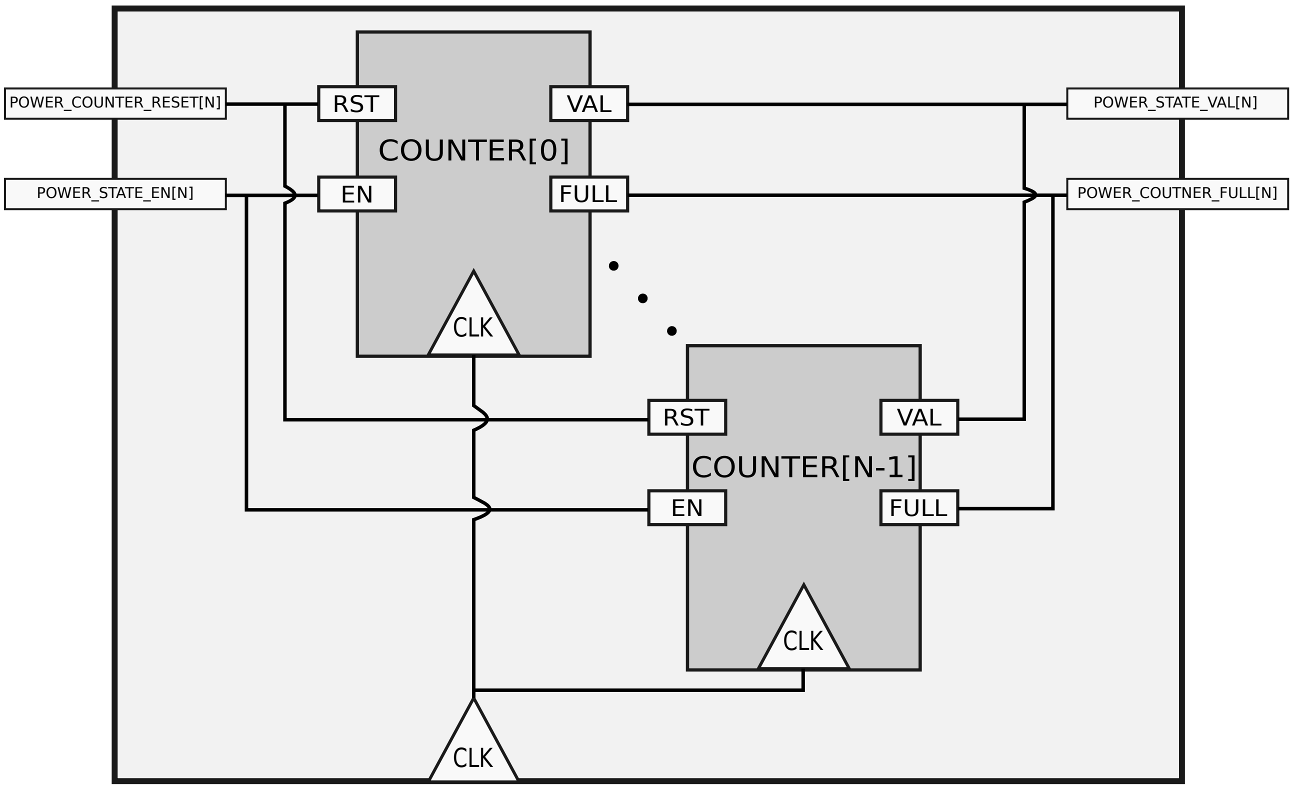


Figure 2.1: Power Approximator Architecture

Power Approximator (**PA**) (Figure 2.1) is used to calculate how many clock cycles each power state is enable. There are N counter, one for each state, that are instantiate automatically as shown below.

```
GEN_COUNTERS : for i in 0 to NUM_PWR_STATE - 1 generate
    COUTER : counter
        generic map(
            MAX          => 2**PWR_APPROX_COUNTER_NUM_BITS-1,
            INIT_VALUE   => 0,
            INCREASE_BY  => 1
        )
        port map(
            clk          => sys_clk,
            INIT         => power_counter_reset(i),
            CE           => power_state_en(i),
            TC           => power_counter_full(i),
            value        => power_counter_val(i)
        );
end generate;
```

It's possible to see that counter generated are equal to constant **NUM_PWR_STATE** that are located in file called `global_settings.vhd`. The max value of counter is set with constant **COUNTER_MAX_NUM_BIT**, always located in the previous cited file, and it's equal to:

$$\text{Max counter val} = 2^{\texttt{COUNTER\_MAX\_NUM\_BIT}}$$

2.1.2 Port description

```
entity POWER_APPROXIMATION is
port(
sys_clk             : in std_logic;
power_state_en      : in std_logic_vector(NUM_PWR_STATES - 1 downto 0);
power_counter_val   : out power_approx_counter_type(NUM_PWR_STATES - 1 downto 0);
power_counter_full  : out std_logic_vector(NUM_PWR_STATES - 1 downto 0);
power_counter_reset : in std_logic_vector(NUM_PWR_STATES - 1 downto 0));
end POWER_APPROXIMATION;
```

Port explanation:

- **`sys_clk:`** Connect to this port the system clock
- **`power_state_en:`** It's a vector in which each bit indicates if a power state is enable ('1') or not ('0').
- **`power_counter_val:`** It's an array with **NUM_PWR_STATES** elements, and each element is an integer that represent the counter value of Power Approximation.
- **`power_counter_full:`** It's a vector in which each bit indicates a counter full.
- **`power_counter_reset:`** It's a vector in which with each bit you can reset a corresponding counter.

2.1.3 Simulation

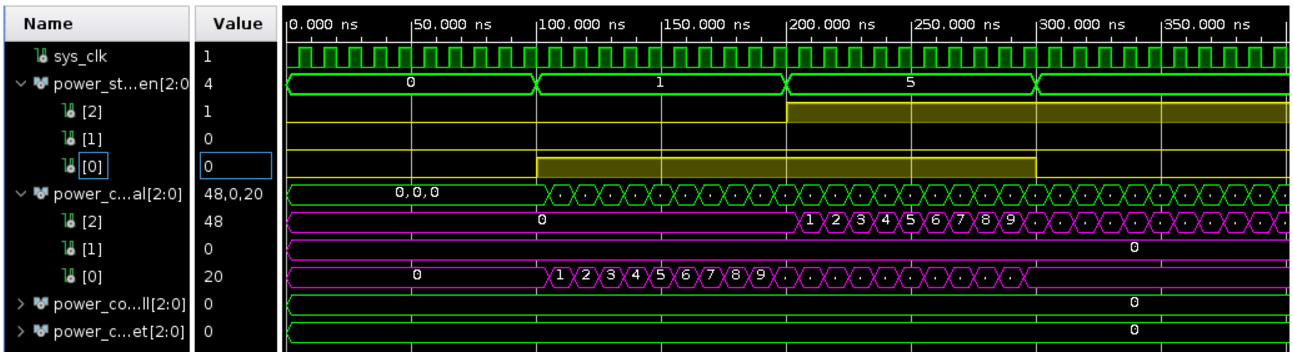


Figure 2.2: PA simulation

In figure 2.2 there is a simulation with *NUM_PWR_STATE = 3*. As you can see, at each clock rising edge, if *power_state_en[n]* is high, the corresponding counter increase by one.

2.2 Instant Power Calculator

2.2.1 General description

Instant Power Calculator (**IPC**) (Figure 2.3) is used to convert the counter value of PA into a power value. It is performed multiplying the counter value with the corresponding power consumption for clock cycle (PCCC).

$$\text{PWR}[W] = \text{COUNTER\_VAL}[CLK] \cdot \Delta\text{PWR\_CONSUMPTION}\left[\frac{W}{CKL}\right]$$

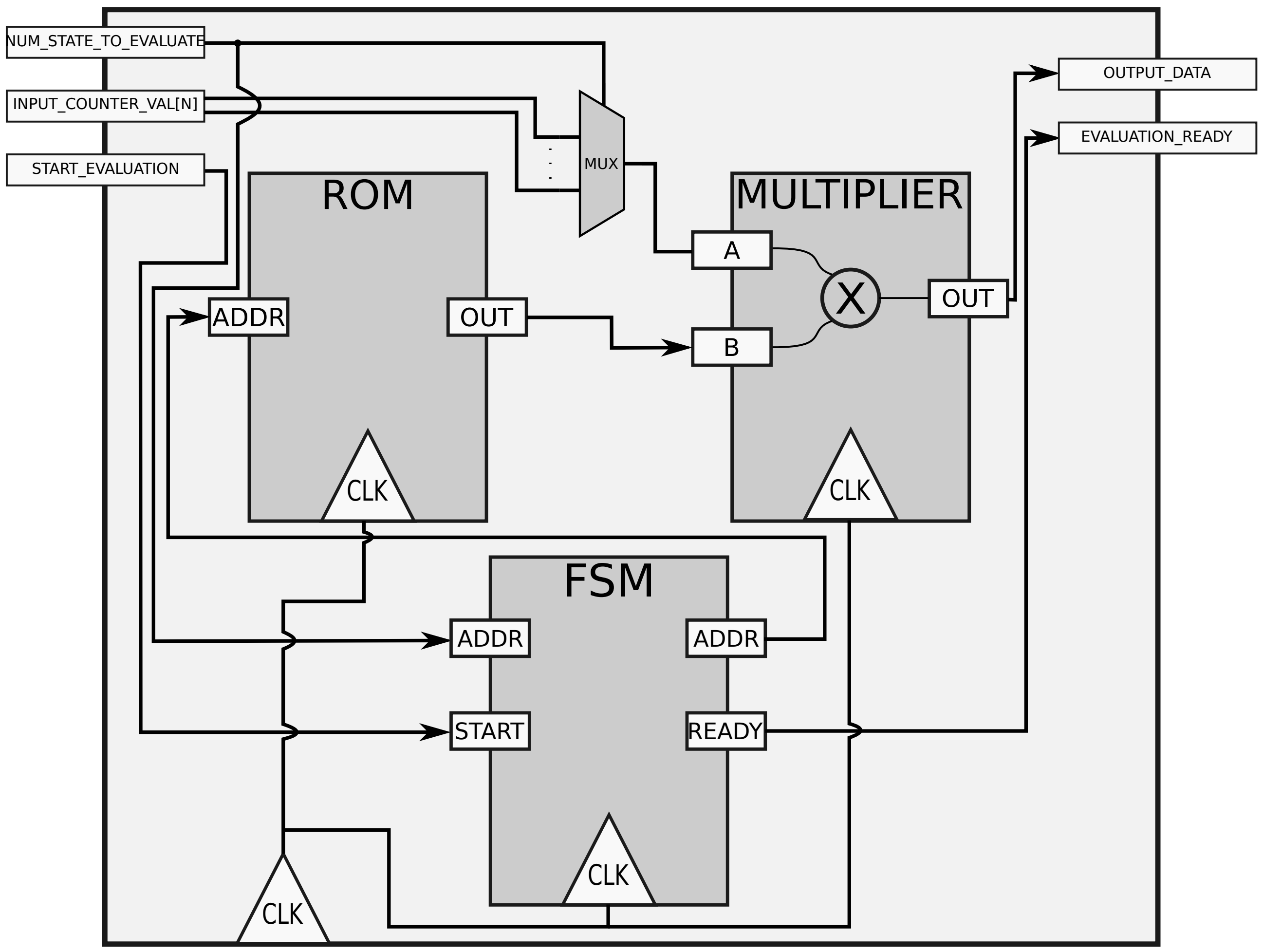


Figure 2.3: Instant power calculator architecture

All PCCC values are stored in a ROM, with the address corresponding to the number of the state. All operations, like reading values from ROM and coordinate the multiplication, are managed by a finite state machine. The latter also sets the *EVALUATION_READY* signal for one clock cycle when the evaluation is performed.

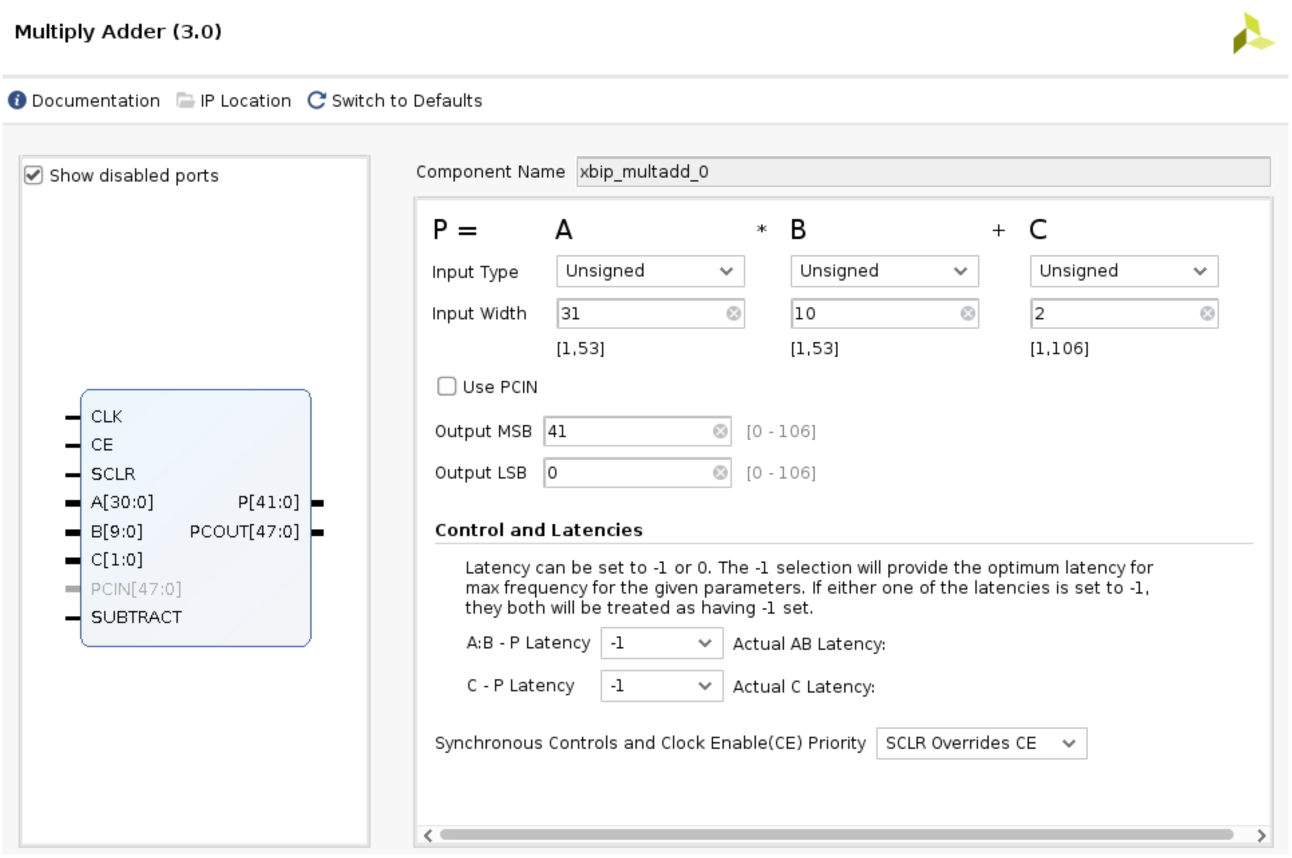
The multiplier IP is declared as follow in figure 2.4

2.2.2 Port description

```
entity INSTANT_PWR_CALC is
port (
sys_clk                  : in std_logic;
start_evaluation         : in std_logic;
evaluation_ready         : out std_logic;
num_state_to_evaluate    : in integer range 0 to NUM_PWR_STATES;
input_counter_val        : in power_approx_counter_type(
                                 NUM_PWR_STATES -1 downto 0);
output_data              : out std_logic_vector(
                                 PWR_APPROX_COUNTER_NUM_BITS +
                                 PWR_CONSUMPTION_ROM_BITS downto 0)
);
end INSTANT_PWR_CALC;
```

Port explanation:

- **`sys_clk`:** Connect to this port the system clock
- **`start_evaluation`:** This signal is used to trigger the fsm which starts the calculation procedure.
- **`evaluation_ready`:** This signal becomes high when evaluated data is ready


Multiply Adder (3.0)

Documentation IP Location Switch to Defaults

☑ Show disabled ports

Component Name: xbip_multadd_0

P =	A	* B	+ C
Input Type	Unsigned	Unsigned	Unsigned
Input Width	31	10	2
	[1,53]	[1,53]	[1,106]

☐ Use PCIN

Output MSB: 41 [0 - 106]

Output LSB: 0 [0 - 106]

Control and Latencies

Latency can be set to -1 or 0. The -1 selection will provide the optimum latency for max frequency for the given parameters. If either one of the latencies is set to -1, they both will be treated as having -1 set.

A:B - P Latency: -1 Actual AB Latency:

C - P Latency: -1 Actual C Latency:

Synchronous Controls and Clock Enable(CE) Priority: SCLR Overrides CE

Figure 2.4: Multiplier IP

- **`num_state_to_evaluate`:** It's a integer that indicates which state is to evaluate.
- **`input_counter_val`:** Connect this port to **`power_counter_val`** of **`PA`** .
- **`output_data`:** It's the output data port

2.2.3 Simulation

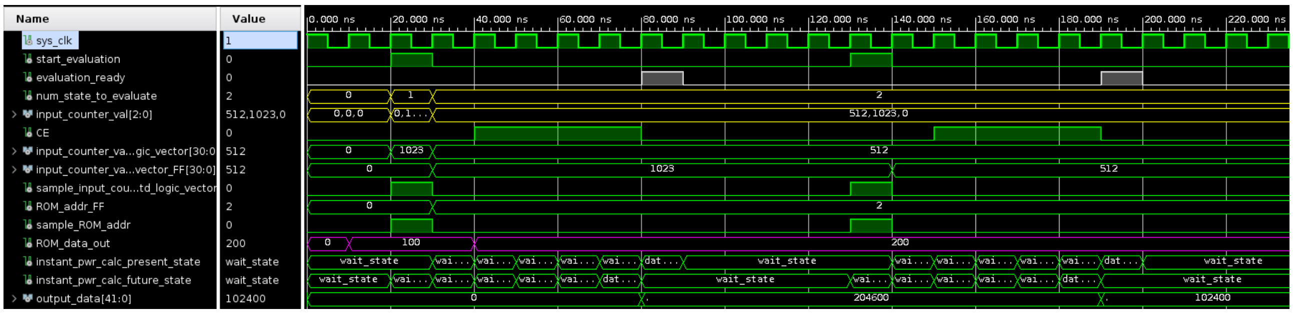

Name	Value
sys_clk	1
start_evaluation	0
evaluation_ready	0
num_state_to_evaluate	2
input_counter_val[2:0]	512,1023,0
CE	0
input_counter_va...gic_vector[30:0	512
input_counter_va...vector_FF[30:0]	512
sample_input_cou...td_logic_vector	0
ROM_addr_FF	2
sample_ROM_addr	0
ROM_data_out	200
instant_pwr_calc_present_state	wait_state
instant_pwr_calc_future_state	wait_state
output_data[41:0]	102400

Figure 2.5: IPC simulation

In the figure 2.5 it's possible to see the module behavior. When there is a clock rising edge and *start_evaluation* is high a sampling is done of counter values and the number of state to evaluate. In this way the user don't need to keep them the same for all the procedure. Now the FSM goes to take the corresponding value from ROM and starts the multiplication. After few clock cycle, when there is a clock rising edge and the signal *evaluation_ready* is high is it possible to sample and take out output data.

2.3 Intermittency Emulator

2.3.1 General description

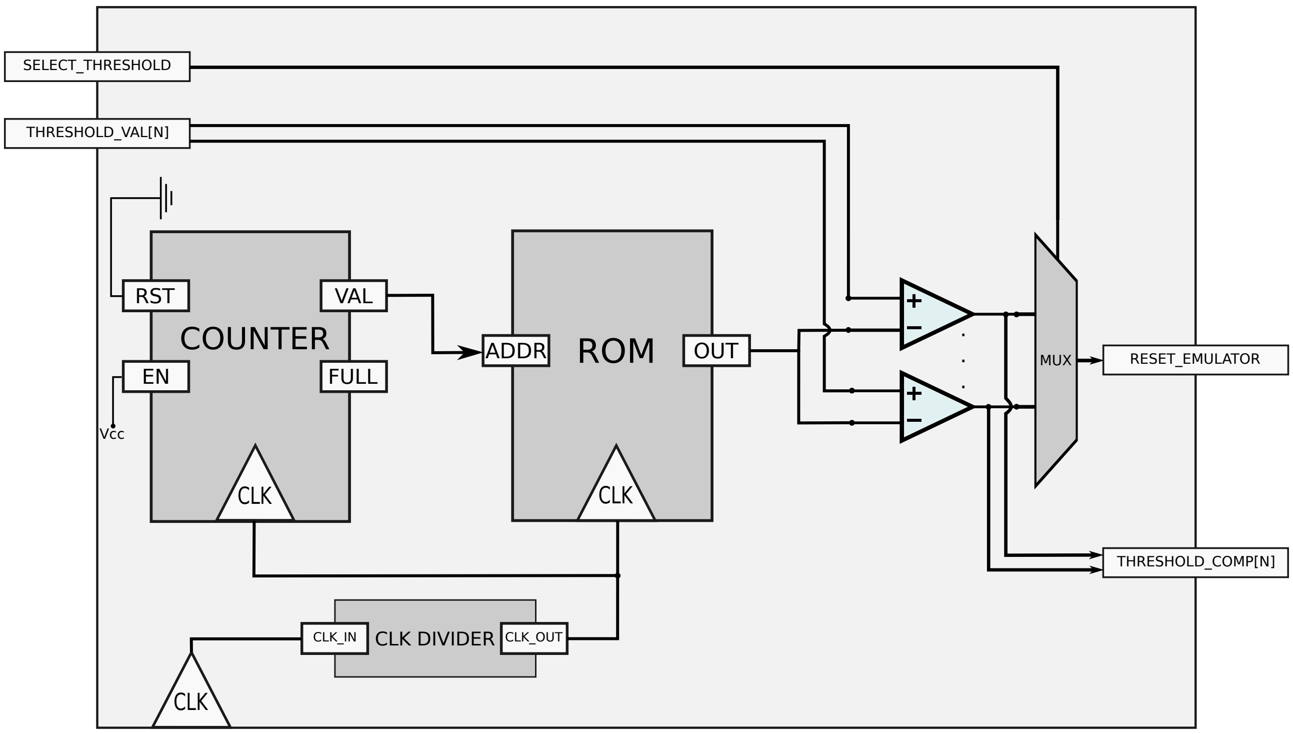


Figure 2.6: Intermittency Emulator Architecture

Itermittency Emulator (**IE**) (figure 2.6) is used to emulate a real case of lack of energy, in fact there is a ROM that contains a feigning voltage trace. User can set a variable number of threshold to compare with the voltage trace. One of those can be used as reset for the entity under test. The voltage trace values change each clock cycle (divided by a prescaler) thanks to a counter that is always-on. The process that menages the counter prescaled is:

```
clk_sync : process(sys_clk) begin
    if rising_edge(sys_clk) then
        if (prescaler_clk /= prescaler_clk_FF) then
            prescaler_clk_FF <= prescaler_clk;
            if prescaler_clk = '1' then
                counter_en <= '1';
            end if;
        else
            counter_en <= '0';
        end if;
    end if;
end process;
```

Due to this, the counter value has a fixed delay from system clock.

2.3.2 Port description

```
entity INTERMITTENCY_EMULATOR is
port(
sys_clk             : in std_logic;
reset_emulator      : out std_logic;
threshold_value     : in intermittency_arr_int_type(
                                INTERMITTENCY_NUM_THRESHOLDS - 1 downto 0);
threshold_compared  : out std_logic_vector(
                                INTERMITTENCY_NUM_THRESHOLDS - 1 downto 0);
select_threshold    : in integer range 0 to INTERMITTENCY_NUM_THRESHOLDS -1
);
end INTERMITTENCY_EMULATOR;
```

Port explanation:

- **sys_clk:** Connect to this port the system clock
- **reset_emulator:** This is the signal that emulates the reset, so it is to connect to your main reset module that you want to emulate.
- **threshold_value:** It's an array of integer in which each element represents a threshold.
- **threshold_compared:** It's a vector in which each bit indicates if voltage value is higher ('0') or lower ('1') then corresponding threshold.
- **select_threshold:** It's a integer used to select which threshold should be connected with reset_emulator signal.

2.3.3 Simulation

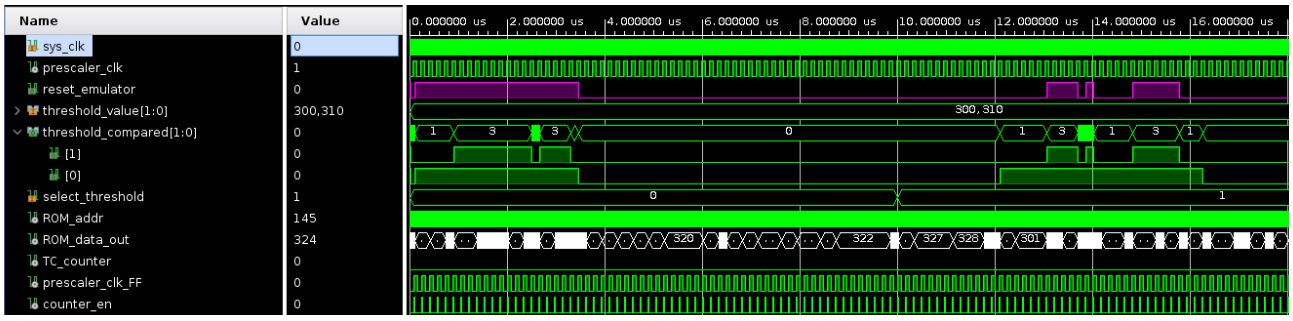


Figure 2.7: Intermittency Emulator simulation

In the figure 2.7 it's possible to see the module behavior. In our case voltage trace goes from 0 to 330 (mean 3.3V) and there are two threshold, one at 310 and one at 300. As you can see *threshold_compared[n]* indicates when the voltage trace is lower than the corresponding threshold, an it that case it goes high. You can use *select_threshold* port to select which signal should be the *reset_emulator*.

2.4 Non Volatile Register Emulator

2.4.1 General description

2.4.2 Port Description

2.5 Non Volatile Register

2.5.1 General description

2.5.2 Port Description

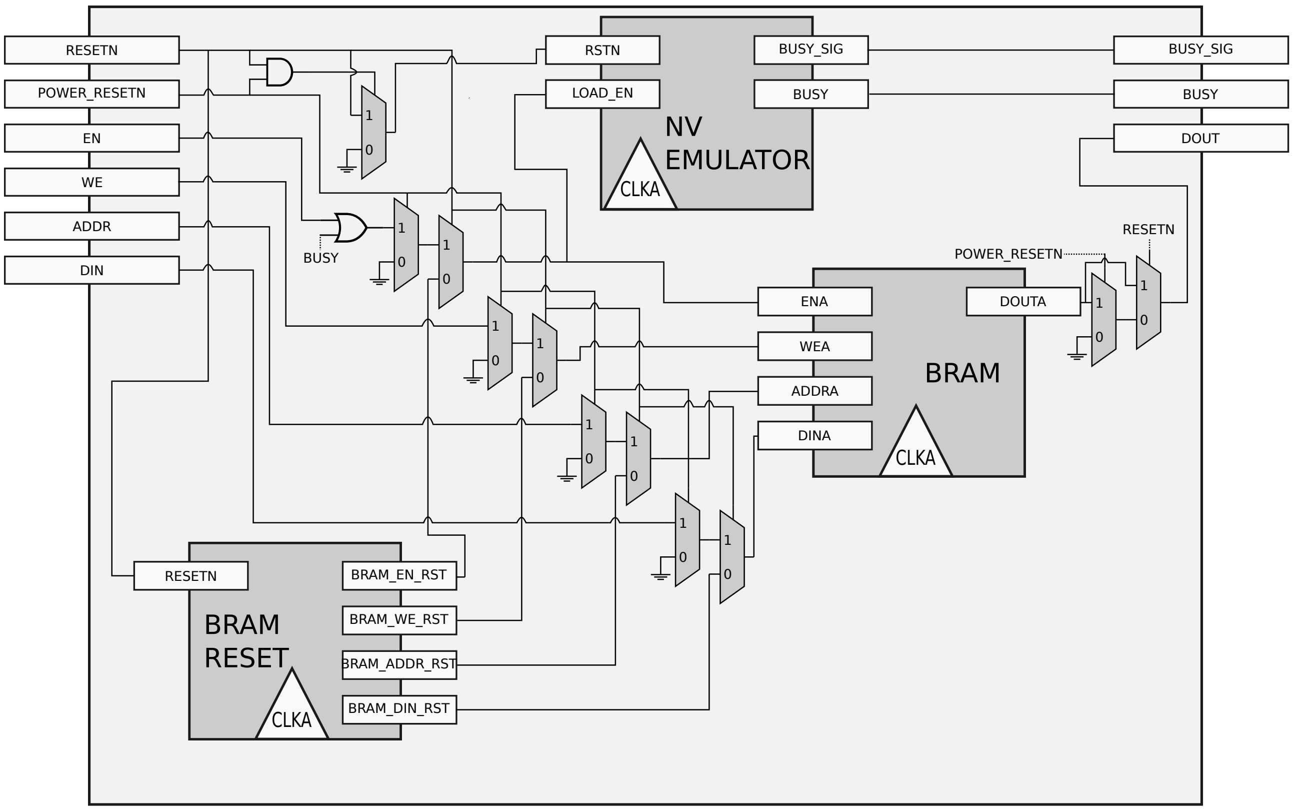


Figure 2.8: NV register Architecture

Chapter 3

Test Architecture

3.1 Fsm NV Reg

3.1.1 General description

Fsm NV Reg (fsm_nv_reg.vhd) illustrated in the figure 3.1 is the entity that takes care about coordination of the various operating states of architecture.

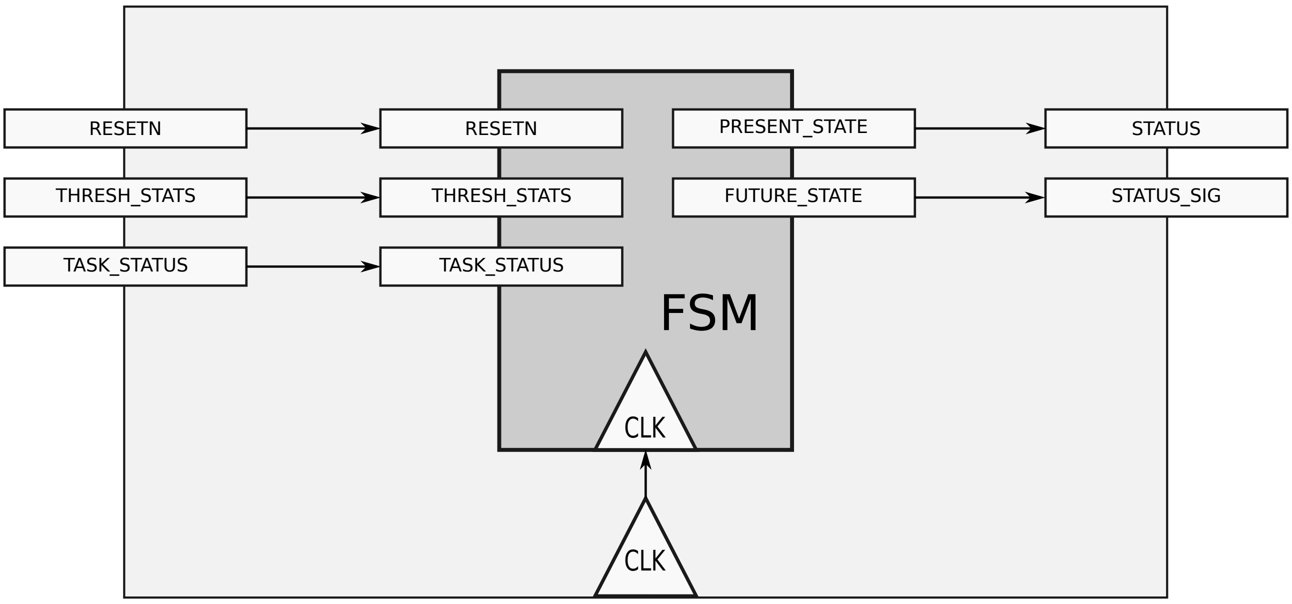


Figure 3.1: fsm_nv_reg Architecture

Inside **TEST_MODULE_PACKAGE.VHD** are declared the following types.

```
type fsm_nv_reg_state_t is(
    shutdown_s,
    init_s,
    start_data_recovery_s,
    recovery_s,
    data_recovered_s,
    do_operation_s,
    start_data_save_s,
    data_save_s,
    data_saved_s
    );

type threshold_t is(
    hazard,
    waring,
    nothing
);
```

First indicates the fsm states. In particular:

- **shutdown_s:**
- **init_s:**
- **start_data_recovery_s:**
- **recovery_s:**
- **data_recovered_s:**
- **do_operation_s:**
- **start_data_save_s:**
- **data_save_s:**
- **data_saved_s:**

Second indicates the threshold type. In particular:

- **hazard:**
- **waring:**
- **nothing:**

3.1.2 Port Description

- **sys_clk:** Connect to this port the system clock
- **resetN:**
- **thresh_stats:**
- **task_status:**
- **status:**
- **status_sig:**

3.2 Adder (entity under test)

3.2.1 General description

In the figure 3.2 it's possible to see the module architecture. There is true dual port BRAM, where are stored the counter values. The FSM reads these counter, adds one and than saves them again into BRAM.
There is a signal called *fsm_status* that indicates the status in which it is operating. So based on the state, FSM decide which operations to do (backup, recovery or normal operation).

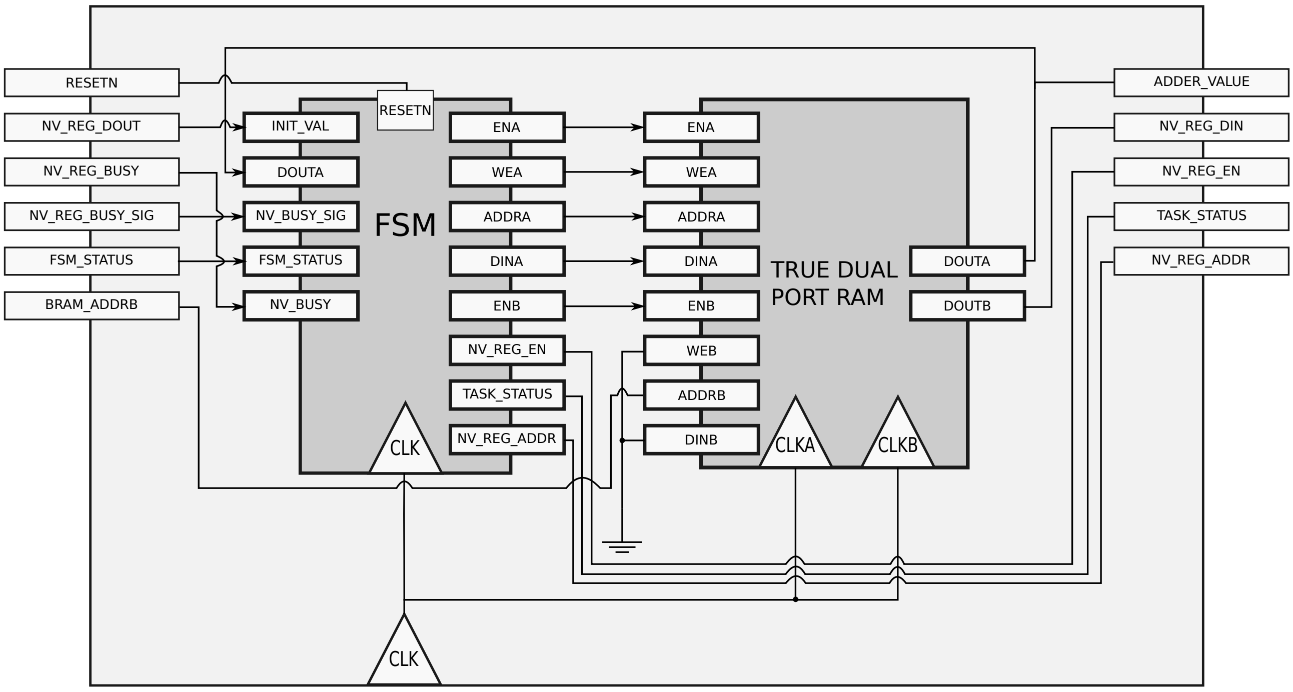


Figure 3.2: Adder Architecture

Chapter 4

Code usage

All the code was written in VHDL and the project was developed on Vivado 2020, so if someone would use an other version or an other software, should take care about conversion on IP modules.
The external library used are:

```
    IEEE.STD_LOGIC_1164.ALL;
    IEEE.NUMERIC_STD.ALL;
```

4.1 Global settings

```
package GLOBAL_SETTINGS is
    constant NUM_PWR_STATES                    : integer := 3;
    constant PWR_CONSUMPTION_ROM_BITS          : integer := 10;
    constant PWR_APPROX_COUNTER_NUM_BITS       : integer := 31;
    constant INTERMITTENCY_NUM_ELEMNTS_ROM     : integer := 1000;
    constant INTERMITTENCY_MAX_VAL_ROM_TRACE   : integer := 330;
    constant INTERMITTENCY_PRESCALER           : integer := 16;
    constant INTERMITTENCY_NUM_THRESHOLDS      : integer := 2;
    constant MASTER_CLK_SPEED                  : INTEGER := 100000000;
    constant MASTER_CLK_PERIOD_NS              : INTEGER :=
                                                  (1e9/MASTER_CLK_SPEED);
end package GLOBAL_SETTINGS;
```

Global settings is a package where are stored all the constant of the architecture, and it is stored in **global_settings.vhd**.
The constants are:

- **NUM_PWR_STATES:** Indicates the number of power states.
- **PWR_CONSUMPTION_ROM_BITS:** Indicates how many bits can have the values inside the Power Consumption ROM.
- **PWR_APPROX_COUNTER_NUM_BITS:** Indicates how many bits can have the counters of Power Approximator. The counters will go from 0 to $(2^{\text{PWR\_APPROX\_COUNTER\_NUM\_BITS}} - 1)$.
- **INTERMITTENCY_NUM_ELEMNTS_ROM:** Indicates the number of element inside the Intermittency ROM (practically how many voltage values there are).
- **INTERMITTENCY_MAX_VAL_ROM_TRACE:** Indicates how big can be the values stored into the Intermittency ROM. The possible values of voltage will be from 0 to INTERMITTENCY_MAX_VAL_ROM_TRACE - 1.
- **INTERMITTENCY_PRESCALER:** It's a prescaler for INTERMITTENCY EMULATOR clock. This value must be a multiple of 2.
- **INTERMITTENCY_NUM_THRESHOLDS:** Indicates the number of thresholds that can exist.
- **MASTER_CLK_SPEED:** Indicates the master clock speed (MHz).
- **MASTER_CLK_PREIOD_NS:** Indicates the master clock period (ns).

4.2 Data type declaration

In the file **packages.vhd** there are all data types declaration.

```vhdl
package POWER_APPROXIMATION_PKG is
    type power_state_en_type is array (integer range <>) of std_logic;
    type power_state_out_type is array (integer range <>) of integer
                            range 0 to 2**PWR_APPROX_COUNTER_NUM_BITS - 1;
    type power_counter_full_type is array (integer range <>) of std_logic;
    type power_counter_resetN_type is array (integer range <>) of std_logic;
end package POWER_APPROXIMATION_PKG;

package INTERMITTENCY_PKG is
    type intermittency_arr_int_type is array (integer range <>) of integer;
end package INTERMITTENCY_PKG;
package COMMON_PACKAGE is
        type PWR_STATES_ARR_TYPE is array(natural range <>) of INTEGER;
        pure function get_busy_counter_end_value(
            input_clk_period : INTEGER;  --in nannoseconds
            max_delay_time: INTEGER  --in nannoseconds
        ) return INTEGER;
end package;

library IEEE;
use ieee.math_real.all;
package body COMMON_PACKAGE is
    pure function get_busy_counter_end_value(
            input_clk_period : INTEGER;
            max_delay_time: INTEGER
        ) return INTEGER is
    begin
        assert (max_delay_time > 0)
            report "nv_reg dealy time = 0" severity Failure;
        --if the fram is as fast as the clk or faster
        if(max_delay_time <= input_clk_period) then
            return 0; --then nothing to do
        else
            --return the upper bound if the delay is greater then the clk period
            return  integer(ceil(real(max_delay_time)/real(input_clk_period))) - 1;

        end if;
    end function;
end package body COMMON_PACKAGE;
```

4.3 IP modification

To avoid problems, make sure that is you modify the IPs all the signals connected to them have the same width.

Chapter 5

Results

NOTE: THIS PART IS STILL TO CHECK

First let's introduce some notes. The goal of our project is to demonstrate how our architecture with non-volatile registers can optimize operations in case there is an unstable power supply. So we compare three different cases:

- **No backup during the task (NB):** No backup are done during the task, only once it finishes the data are stored.
- **Constant intermittent backup (CB):** Backup during task are done at a prefixed time.
- **Dynamic intermittent backup (DB):** It's like our architecture works, backup are done if there is warning.

Let T_{tot} be the execution time of a task T, P_{sd} the probability that a shut down happen during the task, P_{wrn} the probability that a warning state happen during the task, T_{cb} the constant backup period and T_{bk} the time required for a single backup operation.
In general, the average time to complete a task is:

$$\overline{T_{exec}} = \frac{T_{tot}}{1 - P_{err}} \tag{5.1}$$

Where P_{err} is the probability that execution fails and should restart.
In NB architecture we have that $P_{err} = P_{sd}$, so:

$$\overline{T_{exec_{NB}}} = \frac{T_{tot}}{1 - P_{sd}} \tag{5.2}$$

In CB architecture, instead, the probability to not complete the task is broken down by segments delimited by backups, so $P_{err} = P_{sd} \cdot \frac{T_{cb}}{T_{tot}}$. In addiction there is also the delay caused by the backup saving. So:

$$\overline{T_{exec_{CB}}} = \frac{T_{tot}}{1 - P_{sd}\frac{T_{cb}}{T_{tot}}} + \frac{T_{bk}(\frac{T}{T_{cb}} - 1)}{1 - P_{sd}\frac{T_{cb}}{T_{tot}}} = \frac{T_{tot} + T_{bk}(\frac{T}{T_{cb}} - 1)}{1 - P_{sd}\frac{T_{cb}}{T_{tot}}} \tag{5.3}$$

In DB architecture the $P_{err} = 0$, because the task can't fail as data is always saved. So:

$$\overline{T_{exec_{DB}}} = T_{tot} + \frac{T_{bk}}{1 - P_{wrn}} \tag{5.4}$$

NB: execution dime does not take into account the shut down time.

5.1 Example

Now we take under consideration our simulation in figure 5.1. Let suppose that the task takes 100us, like you

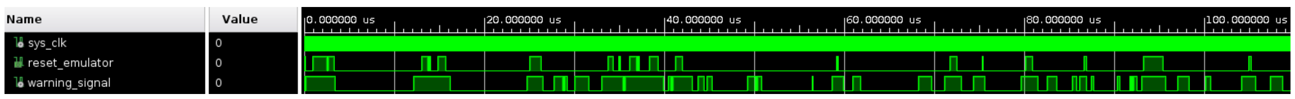


Figure 5.1: Example simulation

can see *reset_emualtor* happens 17 times in the considered period.
Total clock cycles in the period are $(N_{tot} = 10000)$, *reset_emualtor* was activated for $N_{sd} = 1329$ $(13,29\%)$, and *warning_signal* for $N_{wrn} = 4018$ $(40,81\%)$. In our cases T_{up} (update time) is equal to 30ns (3 clock cycles $N_{up} = 3$) and T_{bk} = 140ns (14 clock cycles $N_{bk} = 14$). If warning signal is activated in a continuous way, between one backup and the other must pass 3 data update $(N_{bku} = 3)$ (9 clock cycle). So:

$$End\_val_{DB} = \frac{N_{tot} - N_{wrn}}{N_{up}} + \frac{N_{wrn} - N_{sd}}{\frac{N_{bku}N_{up}+N_{bk}}{N_{bku}}} \simeq 2331 \tag{5.5}$$

In reality, the correct number is 2242 because some time is lost during recovery data or in some cases the saving is not possible because the voltage drop too fast. In that case the efficiency is

$$E_{ff} = \frac{2242}{End\_val_{DB}} = \frac{2242}{2331} \simeq 96,18\% \tag{5.6}$$

In the ideal that the backup procedure takes 0 clock cycle $(N_{bk} = 0)$, the end value would be:

$$End\_val'_{DB} = \frac{N_{tot} - N_{sd}}{N_{up}} \simeq 2890 \tag{5.7}$$

So, considering that case, the real efficiency of the architecture is:

$$E'_{ff} = \frac{2242}{End\_val'_{DB}} = \frac{2242}{2890} \simeq 77,58\% \tag{5.8}$$

If no shut down event had happened, the end value would be $\frac{N_{tot}}{N_{up}} = 3333$.
If we had an architecture NB, the efficiency would have been 0, because the final value would be 0.
With the CB, instead, the end value would have been:

$$End\_val_{CB} = \frac{N_{tot} - N_{sd}}{\frac{N_{bkp}N_{up}}{N_{bkp}-N_{bk}}} \cdot (1 - P_{err}) \tag{5.9}$$

Where N_{bkp} is the constant number of clock cycles between backups and P_{err} is the probability that a shut down event occurs between backups. In order to understand which value of N_{bkp} makes the algorithm CB more effective we have to put that:

$$End\_val_{CB} > End\_val_{DB} \tag{5.10}$$

At this point, for example, we can assume a random value of P_{err}. Of course this assumption in a real case can't be done because P_{err} is correlated with the period of N_{bkp}, if they are correlated. In figure 5.2 it's possible to see how End_val_{CB} varies in relation to different probability of P_{err}. The figure follow that equation:

$$End\_val_{CB} = \frac{N_{tot} - N_{sd}}{\frac{N_{bkp}N_{up}}{N_{bkp}-N_{bk}}} \cdot (1 - P_{err}) = \frac{10000 - 1329}{\frac{3 \cdot N_{bkp}}{N_{bkp}-14}} \cdot (1 - P_{err}) \tag{5.11}$$

With $P_{err} \in]0, 1[$ and $N_{bkp} \in]20, 10000[$. The minimum acceptable value of N_{bkp} is $N_{bk}+N_{up} = 17$, otherwise no addition is possible between backups. The maximum value is, of course, equal to N_{tot}, but in that case the behavior is like NB architecture. From the figure it's also possible to see that if $P_{err} >\sim 0.2$ CB algorithm can't get the DB efficiency. On the contrary, if $P_{err} <\sim 0.2$ and N_{bkp} is small there is a possibility that CB is better than DB.

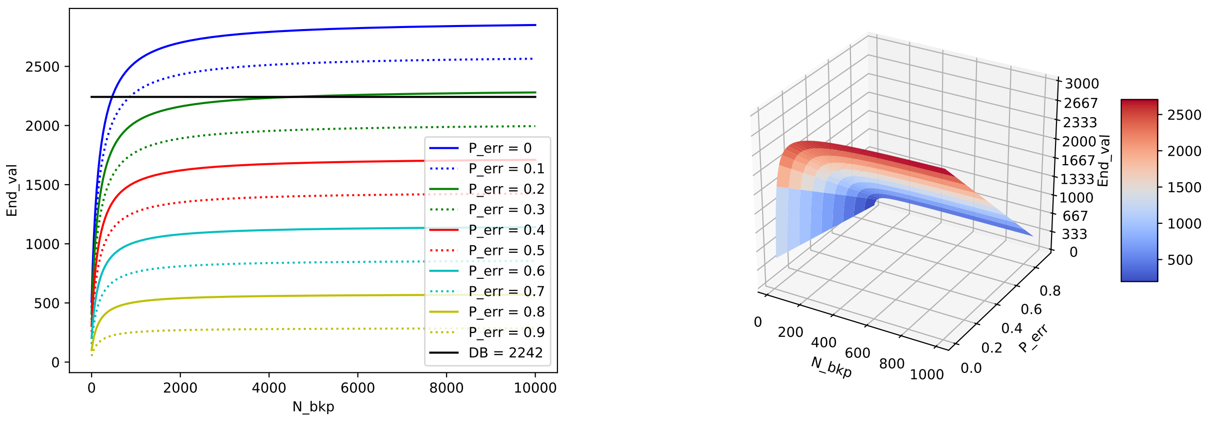

End_val
N_bkp
0
2000
4000
6000
8000
10000
0
500
1000
1500
2000
2500
P_err = 0
P_err = 0.1
P_err = 0.2
P_err = 0.3
P_err = 0.4
P_err = 0.5
P_err = 0.6
P_err = 0.7
P_err = 0.8
P_err = 0.9
DB = 2242
N_bkp
0
200
400
600
800
1000
P_err
0.0
0.2
0.4
0.6
0.8
End_val
0
333
667
1000
1333
1667
2000
2333
2667
3000
500
1000
1500
2000
2500

Figure 5.2