Department of Engineering and Information Science

Course of Embedded Systems

NON VOLATILE MEMORY EMULATION FRAMEWORK FOR FPGAS

Caronti Luca

luca.caronti@studenti.unitn.it

192298

Ruffini Simone

simone.ruffini@tutanota.com

188101

Accademic year 2019/2020

Contents

Abstract

The goal of this project was to develop a non volatile memory emulation framework (**NVMEF**), 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.
The framework itself is composed of 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 series of simple counters (**VCNTR**) that read a values from the volatile register (**VR**), add a constant number to this value and save it again in the volatile register. This cycles till an end value is reached.

Nomenclature

- **PA** : power approximator
- **IE** : intermittency emulator
- **IPC** : instant power calculator
- **NVR** : non volatile register
- **VR** : volatile register
- **NVMEF** : non volatile memory emulation framework
- **NVRE** : non volatile register emulator
- **BRAM** : block RAM
- **VARC** : volatile architecture
- **VCNTR** : volatile counter
- **FSMNVR** : finite state machine for non volatile register

Chapter 1

Project Overview

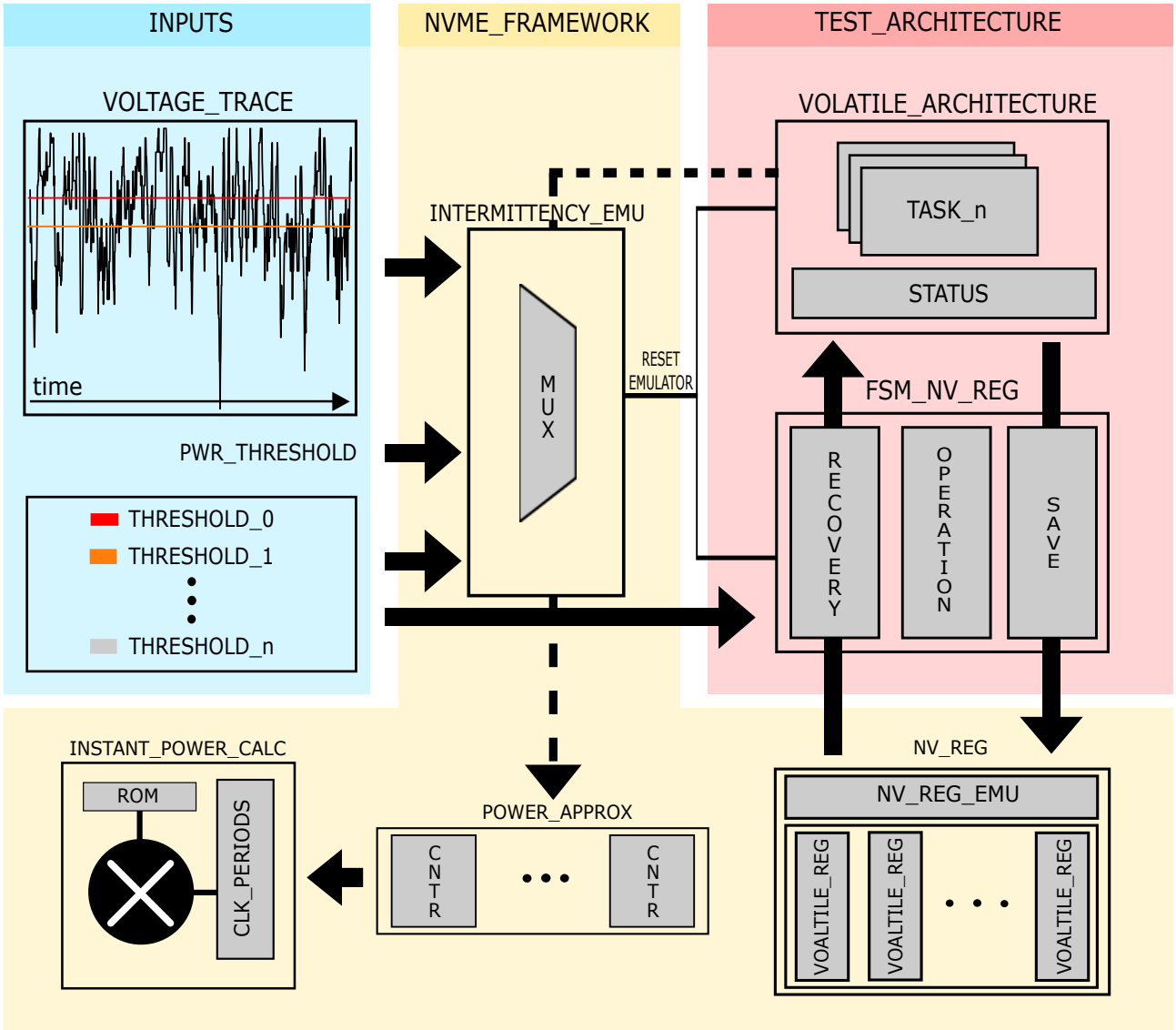


Figure 1.1: General overview of the whole project

The project is divided in two parts: the non volatile memory emulation framework (**NVMEF**) and the test architecture. The main goal has been to develop the **NVMEF** for the transient computing model, but to test such framework a test architecture was implemented.

1.1 Non Volatile Memory Emulation Framework

The **NVMEF** 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_emulator** 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** (**VARC**): in this case a series of simple counters that increment by different values. After each increment the value is saved in a local volatile register (**VR**) and this lasts 2 clock cycles, resulting in increments every 3.
  Such architecture loses all its status (counter value) when the **reset_emulator** triggers (a shutdown event). When the simulated power goes up, the last stored state, in the **NVR** , is restored and the adder continues its operation.

- **FSM_NV_REG** (**FSMNVR**): 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 are user defined. 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 simulation.

Chapter 2

Non Volatile Memory Emulation Framework

2.1 Power Approximator

2.1.1 Description

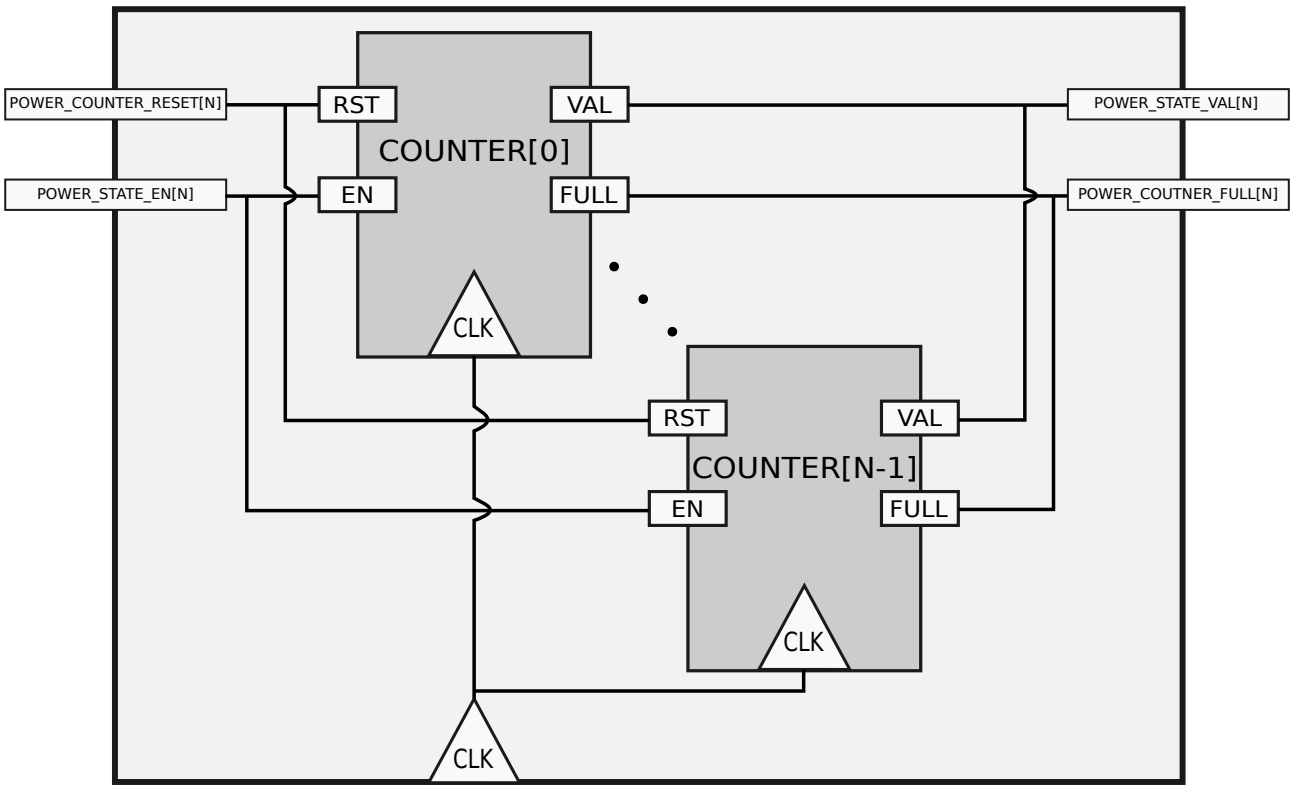


Figure 2.1: Power Approximator Architecture

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:

IN **sys_clk:** Connect to this port the system clock

IN **power_state_en:** It's a vector in which each bit indicates if a power state is enable ('1') or not ('0').

OUT **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.

OUT **power_counter_full:** It's a vector in which each bit indicates a counter full.

IN **power_counter_reset:** It's a vector in which with each bit you can reset a corresponding counter.

2.1.3 Simulation

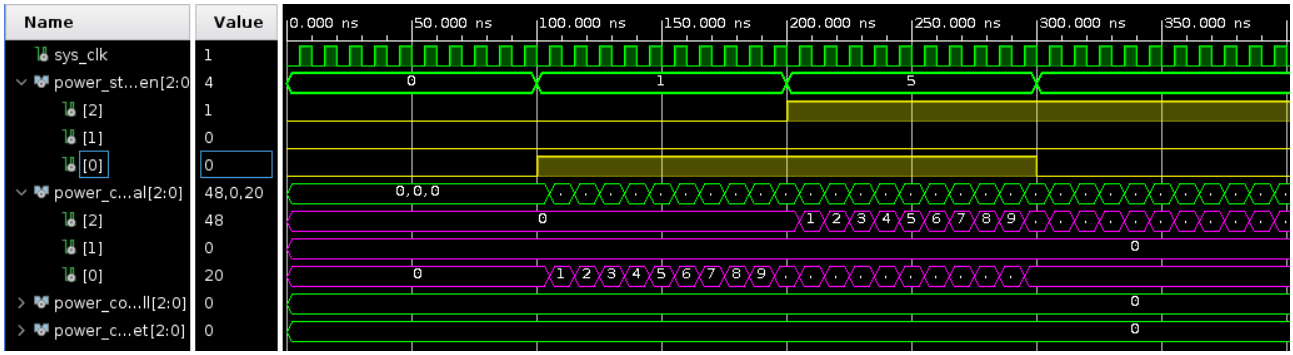


Figure 2.2: PA simulation

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 Description

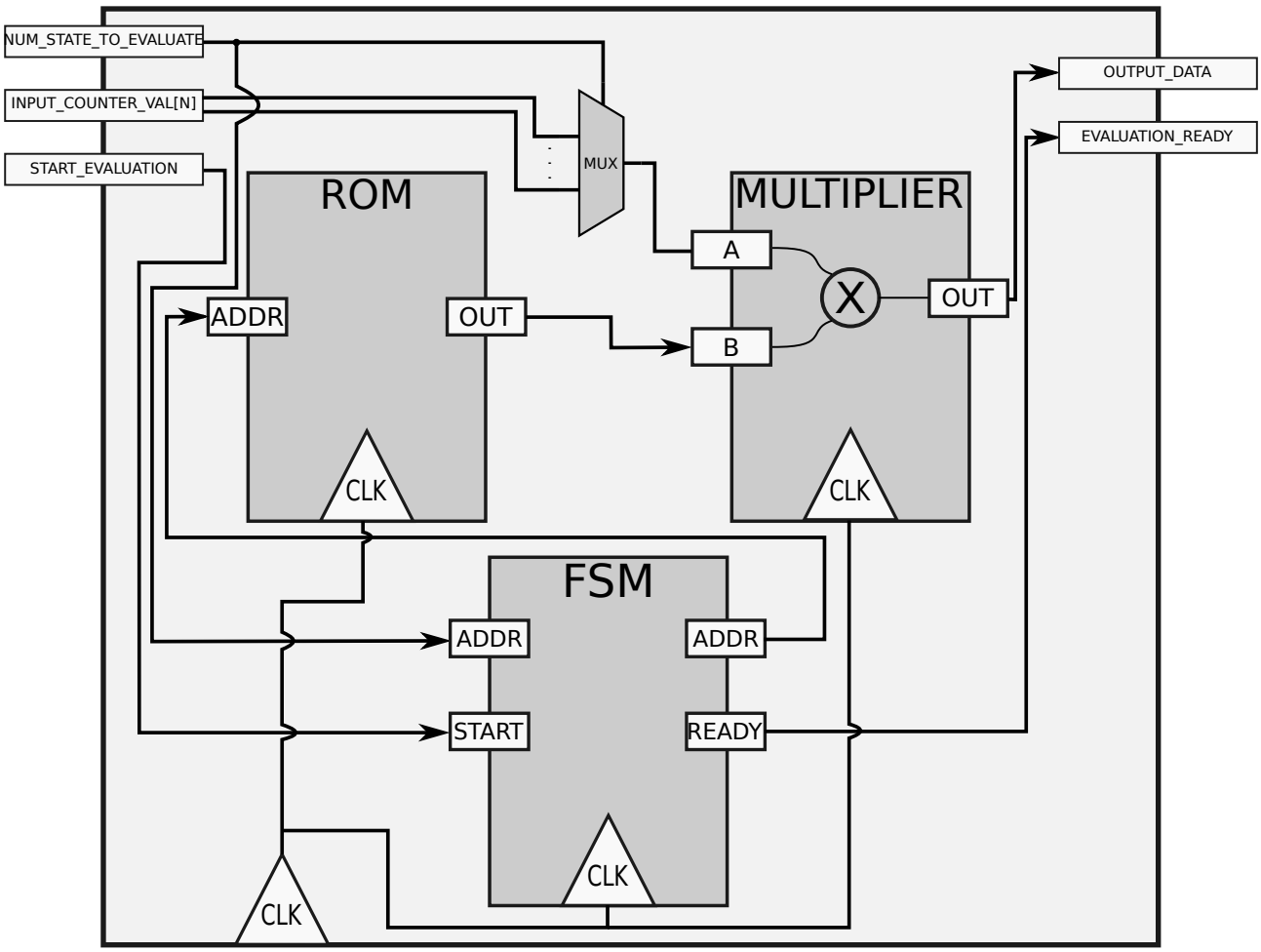


Figure 2.3: Instant power calculator architecture

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).

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

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(
```

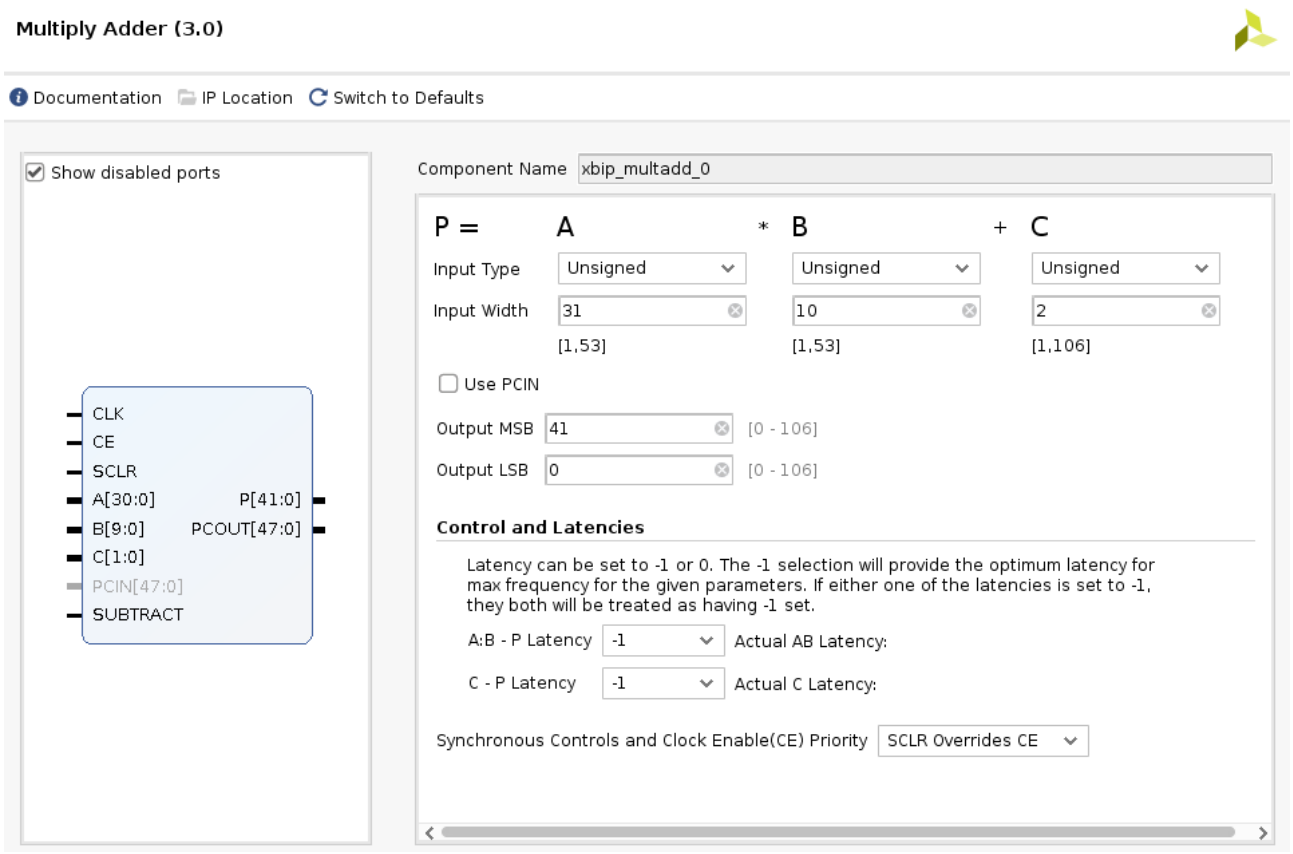


Figure 2.4: Multiplier IP

```
                                PWR_APPROX_COUNTER_NUM_BITS +
                                PWR_CONSUMPTION_ROM_BITS downto 0)
);
end instant_pwr_calc;
```

Port explanation:

IN **sys_clk**: Connect to this port the system clock

IN **start_evaluation**: This signal is used to trigger the fsm which starts the calculation procedure.

OUT **evaluation_ready**: This signal becomes high when evaluated data is ready

IN **num_state_to_evaluate**: It's a integer that indicates which state is to evaluate.

IN **input_counter_val**: Connect this port to **power_counter_val** of **PA** .

OUT **output_data**: It's the output data port

2.2.3 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.

2.2.4 Power consumption value ROM

```
entity pwr_consumption_val_ROM is
    generic(
        NUM_ELEMENTS_ROM : integer;
```

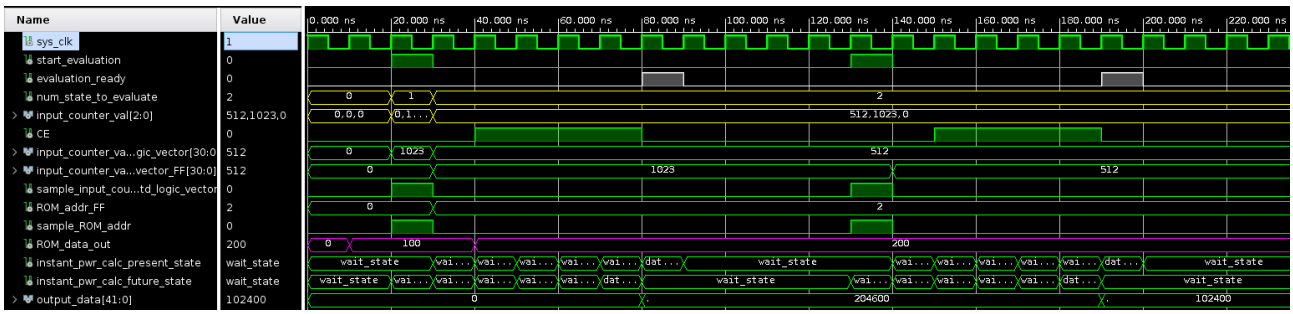


Figure 2.5: IPC simulation

```
        MAX_VAL         : integer
    );
    port(
        clk       : in  std_logic;
        addr      : in  integer range 0 to NUM_ELEMENTS_ROM - 1;
        data_out  : out integer range 0 to MAX_VAL - 1
    );
end pwr_consumption_val_ROM;

architecture Behavioral of pwr_consumption_val_ROM is
    type rom_type is array (0 to NUM_ELEMENTS_ROM - 1) of integer range 0 to  MAX_VAL -
    1;
    signal ROM: rom_type := (
        100,
        50,
        200
    );
begin
    get_data:process (clk) is begin
        if rising_edge(clk) then
            data_out <= ROM(addr);
        end if;
  end process;

end Behavioral;
```

This entity contains **`NUM_ELEMENTS_ROM`** elements, and each of those represent a energy consumption for clock cycle $\left[\frac{J}{clk}\right]$. When you want to get the real value of energy spent, you should only multiply those value with the corresponding **`power_counter_value`**, so you will have:

$$\mathrm{W} = \textbf{pwr\_consumption\_val\_ROM[n]} \left[\frac{J}{clk}\right] \cdot \textbf{power\_counter\_value[n]}[clk] = \mathrm{Energy}[J]$$

2.2.5 Extensibility

If you want to change the multiplier IP, you must:

- Change multiplier component declaration and its instance.

```
--- multiplier component declaration ---
COMPONENT xbip_multadd_0
    PORT (
        CLK      : IN STD_LOGIC;
        CE       : IN STD_LOGIC;
        SCLR     : IN STD_LOGIC;
        A        : IN STD_LOGIC_VECTOR(PWR_APPROX_COUNTER_NUM_BITS - 1 DOWNTO 0);
        B        : IN STD_LOGIC_VECTOR(PWR_CONSUMPTION_ROM_BITS - 1 DOWNTO 0);
        C        : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        SUBTRACT : IN STD_LOGIC;
        P        : OUT STD_LOGIC_VECTOR(PWR_APPROX_COUNTER_NUM_BITS +
    PWR_CONSUMPTION_ROM_BITS DOWNTO 0);
        PCOUT    : OUT STD_LOGIC_VECTOR(47 DOWNTO 0)
```

```
    );
END COMPONENT;
```

```
--- MULTIPLIER INSTANCE ---
MULTIPLIER_0 : xbip_multadd_0
    port map(
        CLK => sys_clk,
        CE => CE,
        SCLR => SCLR,
        A => input_counter_val_std_logic_vector_FF,
        B => ROM_data_out_std_logic_vector,
        C => (others => '0'),
        SUBTRACT => '0',
        P => P,
        PCOUT => PCOUT
    );
```

- Change the fsm that manage all the multiplication process

```
--- SEQUANTIAL FSM ---
fsm_seq : process(sys_clk) begin
  ...
end process;

--- COMBINATORY FSM --
fsm_comb : process(instant_pwr_calc_present_state, start_evaluation) begin
    ...
end process;

--- Flip Flop PROCESS ---
FF_proc : process(sys_clk) begin
    ...
end process;
```

2.3 Intermittency Emulator

2.3.1 Description

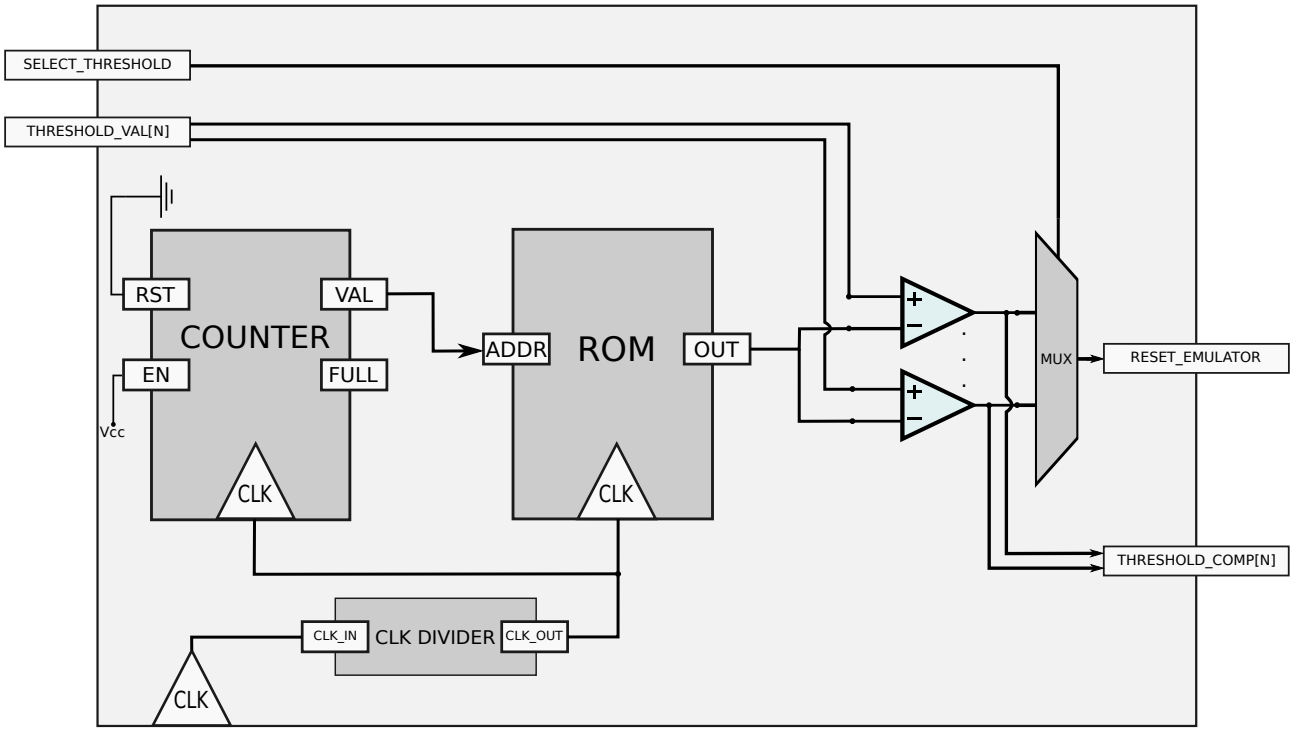


Figure 2.6: Intermittency Emulator Architecture

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
```

```
10 );
11 end intermittency_emulator;
```

Port explanation:

IN **sys_clk**: Connect to this port the system clock

OUT **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.

IN **threshold_value**: It's an array of integer in which each element represents a threshold.

OUT **threshold_compared**: It's a vector in which each bit indicates if voltage value is higher ('0') or lower ('1') then corresponding threshold.

IN **select_threshold**: It's a integer used to select which threshold should be connected with reset_emulator signal.

2.3.3 Simulation

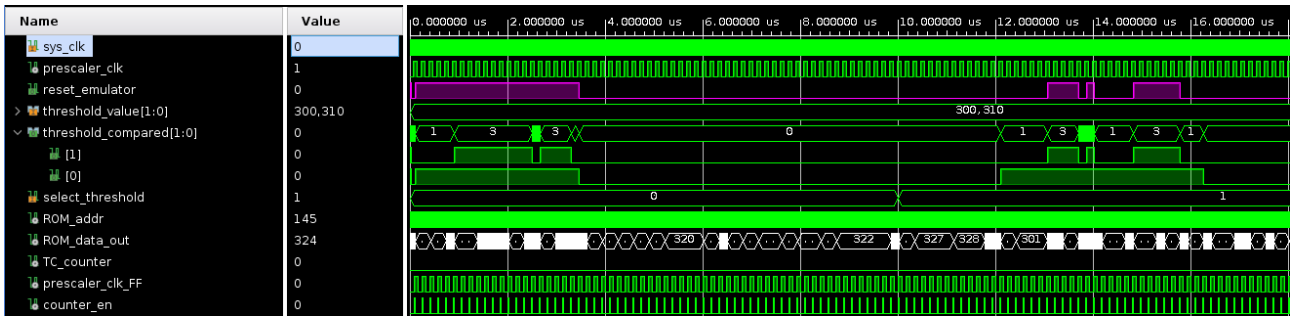


Figure 2.7: Intermittency Emulator simulation

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 Description

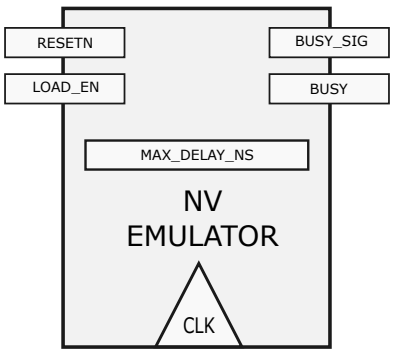


Figure 2.8: NV register emulator

The non volatile register emulator (**NVRE**) is the entity that implements the protocol which allows to treat volatile registers as non volatile ones. This entity must be used in every **NVR** as part of it.

2.4.2 Emulation Protocol

1. *The clock of the **NVR** is the same as the system one*
2. *An operation time or delay time t_{delay} must be defined, i.e. the memory time used to perform a task*
3. *A signal must be used to tell the **NVR** when it will be accessed*
4. *Once the **NVR** is accessed the operation can not be halted and must complete before requesting a new one*
5. *The access time is equal to t_{delay}*
6. *After the operation is accepted by the **NVR** a signal **BUSY** will turn on*
7. *Until the signal **BUSY** is on all the memory related input ports of the **NVR** must remain constant*
8. *The **BUSY** signal will stay on for $(t_{delay} - clk\_period)$ seconds after the **NVR** was accessed then it shuts off*
9. *The operation that the **NVR** will accrpt and performe is the one presented in the first clk cyle where **BUSY** is on*
10. *The output data of the **NVR** can be captured when **BUSY** is sampled off*

The concept of the protocol is to regulate the access of the registers used in the **NVR** . It is important to remember that such registers behave normally during every clock cycle when they are enabled. This means that if the protocol is not respected (especially during write operations) the contents of the memory can be easily corrupted.

Hardware constraints

- The registers used for the **NVR** must be non erasable by a reset signal
- Have normal access ports (address, data in, data out, enable, write enable)
- Are configured with an additional output register after the the main latch. This means that when the data is requested from the register on the rising edge, its value will be available after the next rising edge.

2.4.3 Port Description

```
-- file src/nv_reg_emu.vhd

entity nv_reg_emu is
    Generic(
        MAX_DELAY_NS: INTEGER
    );
    Port (
        clk     : IN STD_LOGIC;
        resetN  : IN STD_LOGIC;
        load_en : IN STD_LOGIC;
        busy_sig: OUT STD_LOGIC;
        busy    : OUT STD_LOGIC
    );
end nv_reg_emu;
```

GENERIC **MAX_DELAY_NS**: this generic value (non modifiable at runtime) defines the t_{delay} of the to be emulated **NVR** . This is an integer value expressing time in nanoseconds.

IN **CLK**: system clock

IN **RESETN**: system reset (active low)

IN **LOAD_EN**: this is the access signal of the protocol (point 3), used to tell the **NVR** when it will be accessed. This signal is interpreted by the entity in a synchronous way (captured on the rising edge of the clock). When the signal is captured the entity will operate according to the protocol specifics (point 5,6,8).

OUT **BUSY**: This signal works according to the protocol (point 8). When the busy signal shuts off does not mean that the t_{delay} time is elapsed, in fact this signal stops one clock cycle before (point 8) because it could be used by synchronous processes to understand when to capture **NVR** output.

OUT **BUSY_SIG**: This signal is the same as **BUSY** but anticipates its value by one clock cycle. This signal can be used by synchronous processes to update the input ports of the **NVR** so for continuous request no clock cycles are missed.

2.4.4 Behaviour

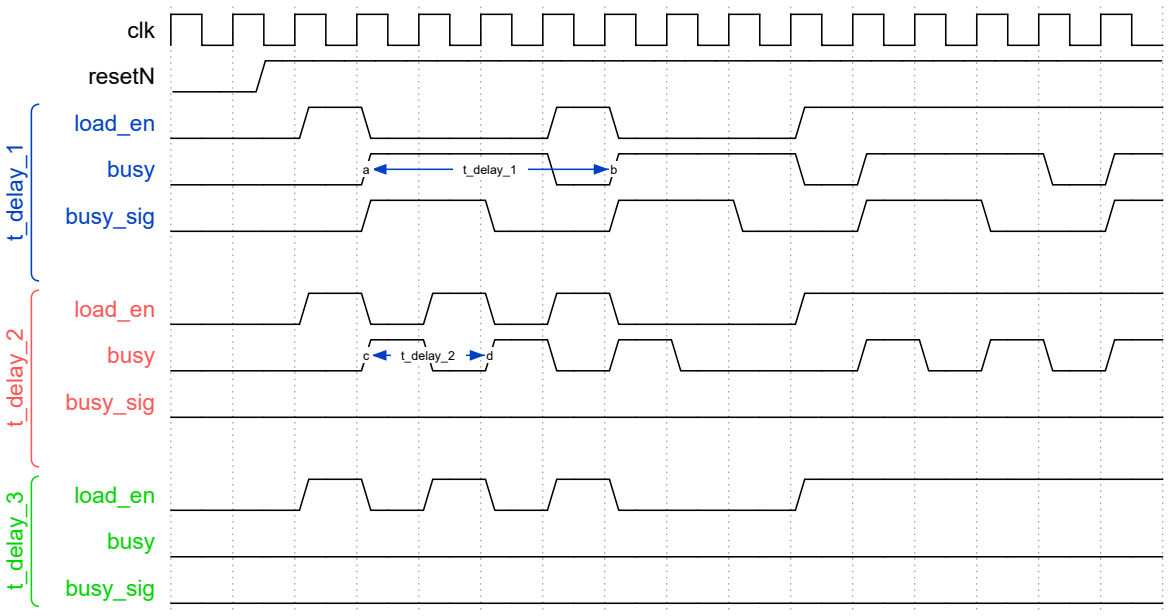


Figure 2.9: **NVRE** behaviour with **MAX_DELAY_NS** equal to $t\_delay\_1, t\_delay\_2, t\_delay\_3$

In figure 2.9 we can see the behaviour of the **NVRE** (protocol) for different t_{delay} times respectively equal to: $4*$**SYSTEM_CLK_PERIOD**, $2*$**SYSTEM_CLK_PERIOD** and **SYSTEM_CLK_PERIOD**. The last case is the most interesting one because the **NVRE** assumes that the **NVR** will be as fast as the speed of the system. In such case the **NVR** will behave exactly like a normal **BRAM** that means that it will never be busy by the protocol rules.
*(N.B. the access policies of the **NVR** will be the same as the **BRAM** ones! so refer to its documentation).*

Another aspect visible in figure is the 4th rule of the protocol: when the **NVR** is accessed by asserting **load_en** the operation will last until the t_{busy} is elapsed, even if the access signal is deasserted during the operation.

Notes

A particular note on how the t_{delay} value is used must be expressed. The delay can not be exactly reproduced inside the architecture because time is quantized, this means that the actual value is slightly different.
The internal delay (t'_{delay}) is mathematically computed by the following equation:

$$\forall\, t_{delay} |\quad t_{delay} \geq sys\_clk\_period > 0$$
$$t'_{delay} = t_{delay} + [sys\_clk\_period - (t_{delay}\ \textbf{mod}\ sys\_clk\_period)]$$

Such expression is implemented in the code considering that: t_{delay} is obtained through a counter that increments by periods of sys_clk_period; the end value of such counter is obtained by the function `get_busy_counter_end_value` in the file **packages.vhd**

2.5 Non Volatile Register

2.5.1 Description

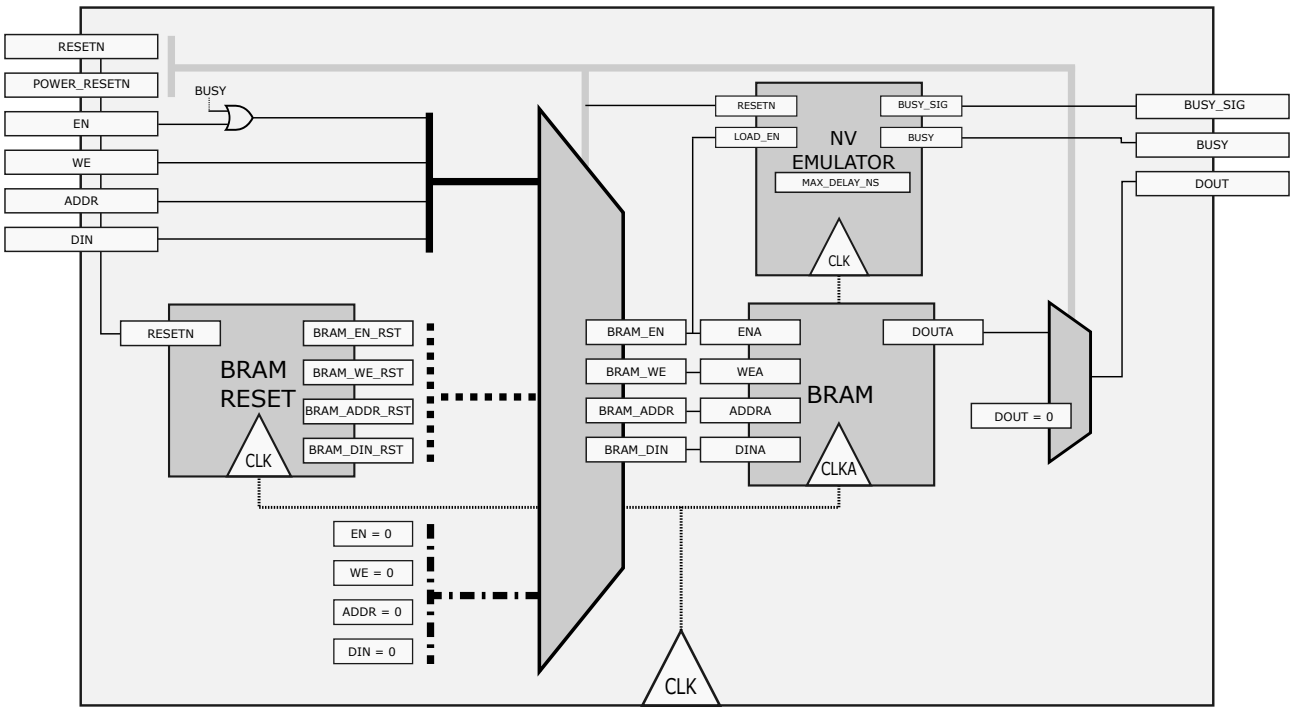


Figure 2.10: NV register Architecture

The non volatile register (**NVR**) is the memory used by the test architecture to implement the transient model. This entity is practically only a wrapper that combines the ports of the **NVRE** and a **BRAM** memory for the ease of use of end users.

The **NVR** memory presented in this report is based on the *single port RAM* insatiable by the Block Memory Generator LogiCORE™ IP by Xilinx® with the relative characteristics documented in Figure 2.11 and Figure 2.12. Although it is possible to use different types of **BRAM** (like a *simple dual port* or a *true dual port RAM*) changes in the entity code will be necessary. Those changes will be documented in the next sections.
⇒**N.B.**: Memories that are not generated by the Xilinx® IP, can use the general code model of the **NVR** but they will be not documented.⇐

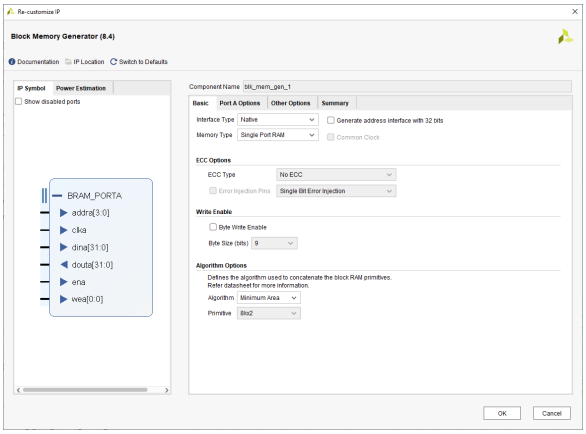


Figure 2.11:
Single port RAM block ram generator page1

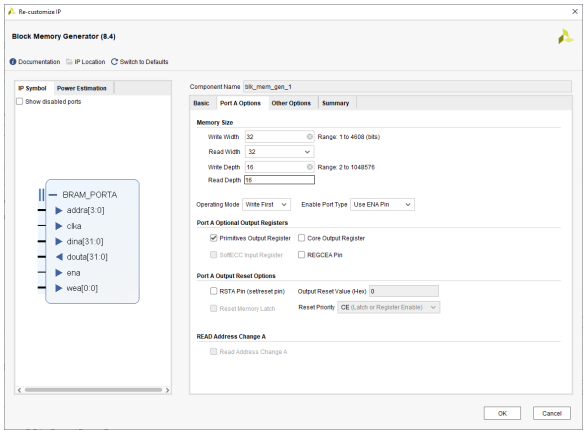


Figure 2.12:
Single port RAM block ram generator page2

The main feature to enable in the bram generation is the optional output register

2.5.2 Working model

As said before, the **NVR** is a wrapper for his inner components and in a general way it combines the ports of **NVRE** and **BRAM** into one single entity. Nonetheless some additional procedures and features are present to emulate better a non volatile register:

- RST_BRAM: this process is used to wipe the content of the **BRAM** . Block memories are not erasable at runtime by a reset signal, only by powering down the FPGA. Even after such procedure, at runtime, is possible that values previous stored in registers can be found (if enough time has passed after power off this is not true).
  This procedure makes possible to erase the **BRAM** when the whole system is under reset status.
  N.B. The process writes zeros in each register of the **BRAM** while the main reset is on. To reset the **BRAM** completely, the reset signal must be on for at least NV_REG_WIDTH $* sys\_clk\_period$.

  This feature can be deleted by the **NVR** without consequences on the global behaviour.

- MUX: this is an asynchronous combinatorial process that selects which inputs present to the **BRAM** . It multiplexes over the two possible reset signals of the system: the system\global reset common to the whole system and the power reset (**reset_emulator**) generated by the **IE** . When the main reset is active the input ports of the **BRAM** are the outputs of RST_RAM process while when the power reset signal is active, all ports are pulled down to 0 (even the output of the **NVR**). Otherwise if none of the reset signals is active the input ports of the **NVR** are directly connected to the **BRAM** except the enable signal (see below).

- ENABLE_HOLD: The enable signal under normal conditions (no reset active) is not entirely directly connected to the **BRAM** . A summ (logical OR) against the busy signal of the **NVRE** is performed before reaching the **BRAM** enable port.
  The reason behind this is the 4th rule of the Emulation Protocol. Since the enable signal controls the status of the **BRAM** we cannot disable it while an operation is ongoing.

  This is not properly a feature but an extension of the Emulation protocol when the **BRAM** has an internal enable signal. (It is possible to generate **BRAM** s whitout that signal, in this case the 4th property is always respected).

⇒ All features are observable in the code by searching its relative name.

2.5.3 Port Description

```
--file src/nv_reg.vhd

entity nv_reg is
    Generic(
        MAX_DELAY_NS: INTEGER;
        NV_REG_WIDTH: INTEGER
    );
    Port (
        clk             : in STD_LOGIC;
        resetN          : in STD_LOGIC;
        power_resetN    : in STD_LOGIC;
        -------------chage from here--------------
        busy            : out STD_LOGIC;
        busy_sig        : out STD_LOGIC;
        en              : in STD_LOGIC;
        we              : in STD_LOGIC_VECTOR(0 DOWNTO 0);
        addr            : in STD_LOGIC_VECTOR(integer(ceil(log2(real(NV_REG_WIDTH))))-1
    DOWNTO 0);
        din             : in STD_LOGIC_VECTOR(31 DOWNTO 0);
        dout            : out STD_LOGIC_VECTOR(31 DOWNTO 0)
        -------------chage to here----------------
    );
end nv_reg;
```

GENERIC **MAX_DELAY_NS**: See MAX_DELAY_NS **NVRE**

GENERIC **NV_REG_WIDTH**: The number of memory cells of the generated primitive.
This parameter will not change the size of the already generated BRAM memory, it's only used internally to define the length of arrays.
The size of the BRAM must be changed via the LogiCORE IP

IN **clk**: system clock

IN **resetN**: system reset (the main signal that will reset both the test architecture and the **NVMEF**). Active low

IN **power_resetN**: The reset generated by the **IE** (**reset_emulator**), this reset simulates a power failure (the contents of the **NVR** will persist). Active low.

OUT **busy**: See busy **NVRE**

OUT **busy_sig**: See busy_sig **NVRE**

IN \OUT **BRAM specific ports**: The primitive ports of the selected **BRAM** , typically: enable, address, write enable, data input and data output. Those ports will change with respect to the chosen memory primitive.

2.5.4 Behaviour

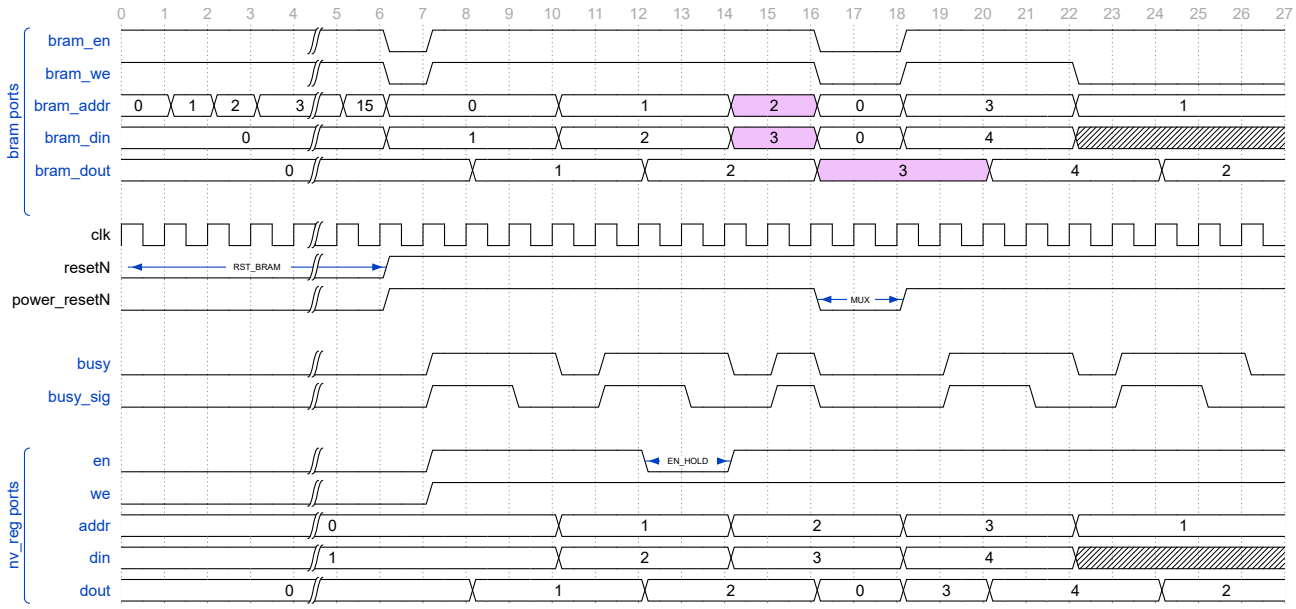


Figure 2.13: NV register Behaviour and **BRAM** internal ports behaviour
The used **BRAM** is the one instantiated in figure 2.11 and 2.12.

periods	Figure 2.13 description
$[0, 6]$	Until **resetN** is off, RST_RAM is active and wipes the **BRAM** .
$[7, 10]$	In this period the input ports (addr, din, we) are hold constant because the **NVR** is in busy mode.
$[12, 14]$	Here we can observe the ENABLE_HOLD *"feature"* that keeps the operation ongoing even if the en signal was deasserted during the busy time. In fact at the **BRAM** level the en signal remains asserted.
$[15, 17]$	The protocol is violated. In this case the event was not avoidable because the asynchronous **power_resetN** signal interrupted the operation. We can see that din was still captured, nevertheless, at architecture level, we cannot rely on such probability and events like this should be avoided by good policies.
$[16, 18]$	MUX in action. **power_resetN** is active but **NVR** ports are not null (intentionally wrong) instead the **BRAM** ports are switched to a default value (this applies to dout too that is defaulted to 0).
$[18, 22]$	As described in the 10^{th} rule of the protocol the output value of the **NVR** cannot be captured during busy time because such value can change, instead the appropriate time would be between $[23, 24]$.

2.5.5 Extensibility

Like stated beforehand the primitive register **NVR** can be changed with other types of **BRAM** . The changes to the code are documented inside the source code but an example will be made. (It is recommended to make new entities based on **NVR** instead of changing **NVR**).
For example if we want to use a *Ture dual port RAM* **BRAM** the changes that we will make are the following ones:

```
--file src/nv_reg_2.vhd
...
    Port (
        ...
        -------------chage from here--------------
        busya            : out STD_LOGIC;
        busya_sig        : out STD_LOGIC;
        ena              : in STD_LOGIC;
        wea              : in STD_LOGIC_VECTOR(0 DOWNTO 0);
        addra            : in STD_LOGIC_VECTOR(integer(ceil(log2(real(NV_REG_WIDTH))))-1
    DOWNTO 0);
        dina             : in STD_LOGIC_VECTOR(31 DOWNTO 0);
        douta            : out STD_LOGIC_VECTOR(31 DOWNTO 0)
        busyb            : out STD_LOGIC;
        busyb_sig        : out STD_LOGIC;
        enb              : in STD_LOGIC;
        web              : in STD_LOGIC_VECTOR(0 DOWNTO 0);
        addrb            : in STD_LOGIC_VECTOR(integer(ceil(log2(real(NV_REG_WIDTH))))-1
    DOWNTO 0);
        dinb             : in STD_LOGIC_VECTOR(31 DOWNTO 0);
        doutb            : out STD_LOGIC_VECTOR(31 DOWNTO 0)
        -------------chage to here----------------
    );
    architecture Behavioral of nv_reg_2 is
    ...
    --to use a different bram memory as primitive for the nv_reg
    --------------------place here new memory component--------------------
    COMPONENT blk_mem_gen_1 IS
    ... --true dual port ram ports (use same clock)
    END COMPONENT blk_mem_gen_1;
    ------------------------------------------------------------------------
    ....
    begin
    ....
```

```
    --since we have "logically" two brams we need two nv_reg_emu
    EMU1: nv_reg_emu
        port map(...)
    EMU2: nv_reg_emu
        port map(...)
    -------------------------MUX------------------------------
    ...
    ------------------place new logic here--------------------
    bram_enb <= bram_en_rst when resetN = '0' else
            '0' when power_resetN = '0' else
            (enb or busyb_internal);              --ENABLE HOLD
    bram_web <= bram_we_rst when resetN = '0' else
            (OTHERS => '0') when power_resetN = '0' else
            web;
    bram_addrb <= bram_addr_rst when resetN = '0' else
            (OTHERS => '0') when power_resetN = '0' else
            addrb;
    bram_dinb <= bram_din_rst when resetN = '0' else
            (OTHERS => '0') when power_resetN = '0' else
            dinb;
    doutb <= bram_dout when resetN = '0' else
            (OTHERS => '0') when power_resetN = '0' else
            bram_doutb;
    ----------------------END MUX-----------------------------

    --the reset process remains the same (the mem. is shared), just make sure that the
    --bram_rst_enb is disabled so no collision will happen during erasing

    bram_rst_enb <= '0';
    ...
end nv_reg_2;
```

Chapter 3

Test Architecture

The test architecture is a combination of volatile modules that loose their state after each reset. The assertion of an internal signal routed to the reset ports of such architecture emulates a shutdown of the system/FPGA but localized only for the test architecture.

The components of the non volatile memory emulation framework (**NVMEF**) are the building blocks that enable non volatile memory storage emulation on FPGAs. On the other hand the test architecture was developed to demonstrate the correctness and capabilities of **NVMEF** and also to propose a simple but effective model design for transient based computing.

The test architecture for this paper consist of:

- **Volatile architecture**: The content itself. A set of modules, entities, processes etc... that loose their state after a reset. The volatile architecture must be compatible with the transient model computing. This holds if modules follow the directives of the **FSMNVR** .
  In this paper the **VARC** is:
  - **Volatile counter** (**VCNTR**): A set of simple counters that increase by different constant values. The value of each counter is stored in volatile memory/registers (**VR**) so before each count step the current value must be requested and then modified.
- **Finite state machine for non volatile register** (**FSMNVR**): This module supervises the interactions between the volatile architecture (**VARC**) and the **NVMEF** , specifically the **NVR** . With this entity the **VARC** is able to compute in a transiently-powered system. Different implementation of this module define different policies of backup. In this paper we explore three different types of policies:
  - **Task end backup** (**TB** or **FSMNVR_TB**): No backup are done during the task, only once the task is complete data are stored.
  - **Constant intermittent backup** (**CB** or **FSMNVR_CB**): Backup during task are done at a prefixed time.
  - **Dynamic intermittent backup** (**DB** or **FSMNVR_DB**): Backup are done if certain runtime conditions are met, i.e. by checking the threshold signals of the voltage trace.

This simple **VCNTR** was chosen as a **VARC** to demonstrate how a fundamental component of all controllers/ICs can be implemented with transient based computing (using the paper's framework+test architecture implementation), as to make the precedent that will lead to implementation of more complex components and eventually a full microprocessor implementation (ex: RISC-V ISA).

A feature of this test architecture is its re-usability: by keeping a finite state machine that controls interactions between volatile and non volatile memory and a volatile architecture that follows the directions of such finite state machine, a large variety of architectures can be implemented even by using the current implementations of the fsm.

3.1 Fsm NV Reg

3.1.1 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.

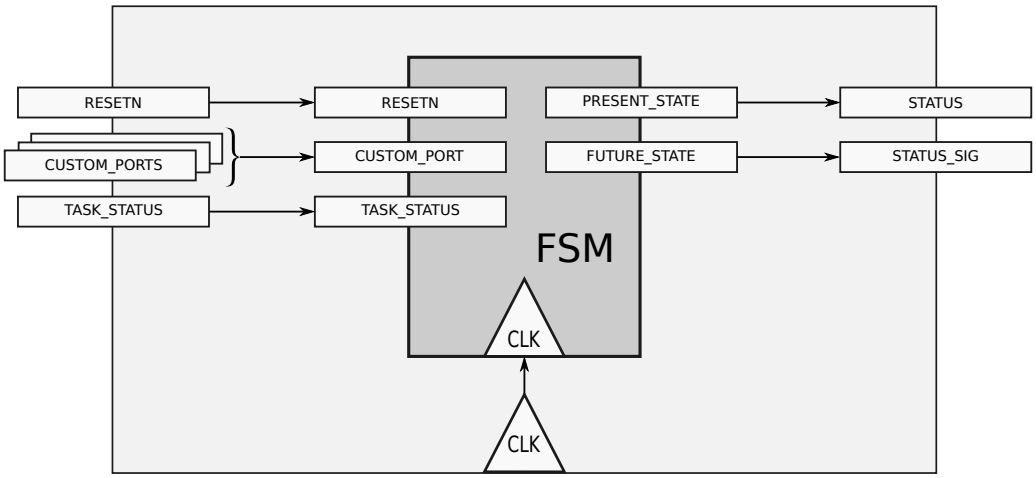


Figure 3.1: fsm_nv_reg Architecture

***CUSTOM_PORT:** Custom port it's different for each type of architecture (described below):

- **TB**: There are 2 ports, one is connected to counter_1 value, the other one is connected to task_complete_val_counter and indicates the value that counter_1 should to reach to consider the task completed.
- **DB**: it's connected to threshold_status value.
- **CB**: it's a signal that indicates the number of clock cycles of backup period.

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,
    warning,
    nothing
);
```

First indicates the fsm states. In particular:

- **shutdown_s**: It's the shutdown state (when there is the reset signal). In this state all is off.
- **init_s**: This state takes one clock cycle. It's the initialization state.
- **start_data_recovery_s**: When fsm is in this state it means that the recovery is starting.
- **recovery_s**: State during the recovery operations.
- **data_recovered_s**: Indicates that the recovery finished.

- **do_operation_s**: It's the normal state where all the modules can do their operations.
- **start_data_save_s**: When fsm is in this state it means that the threshold state is in hazard mode, so the data saving operations should start.
- **data_save_s**: State in which saving operations are done.
- **data_saved_s**: Saving operation finished.

3.1.2 FSM DB

DM (Dynamic Backup) means that the backup is done only when there is a warning signal, that in our case become active when the voltage trace falls below a certain threshold.

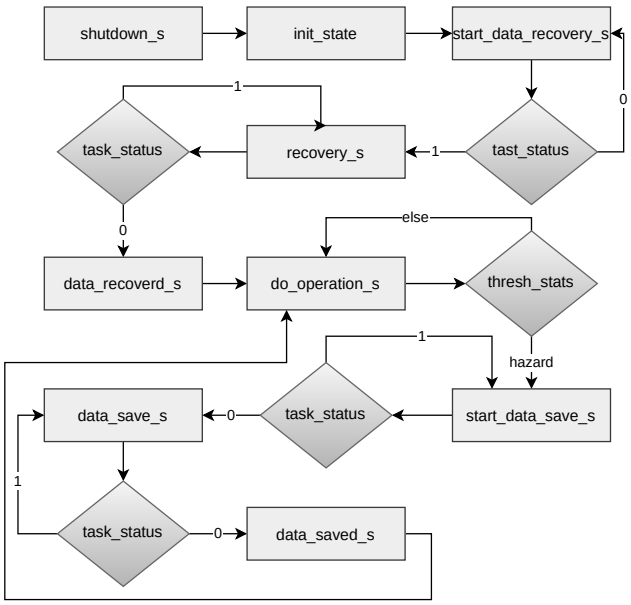


Figure 3.2: db_fsm

In our case, we set also that when the FSM arrives into the **do_operation_s** state, it should remain in this state for at least 10 clock cycles. Otherwise if the warning status were always active there would be no time to update the counters.

Backup condition

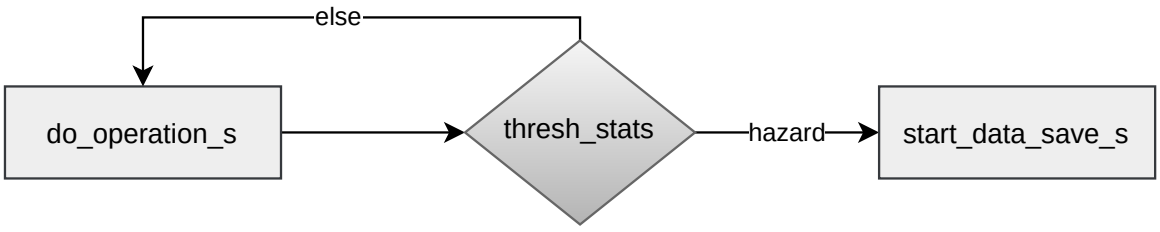


Figure 3.3: Backup condition DB

Like shown in figure 3.3, the condition to start a backup procedure is that signal **thresh_stats** become equal to "hazard".

3.1.3 FSM CB

CB (Constant Backup) means that the backup is done with a prefixed period. The fsm is illustrated in figure 3.4. It's practically the same of DB, with the difference that you start the backup (**start_data_save_s** state)

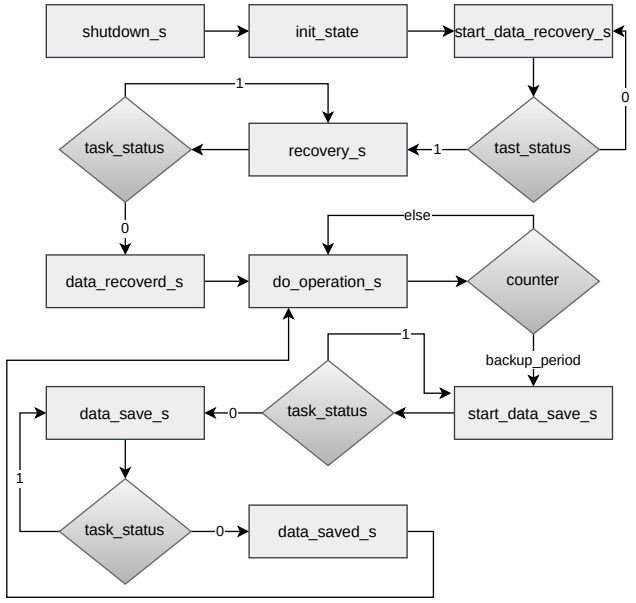


Figure 3.4: cb_fsm

only when a internal counter reach a prefixed value (backup period).

Backup condition

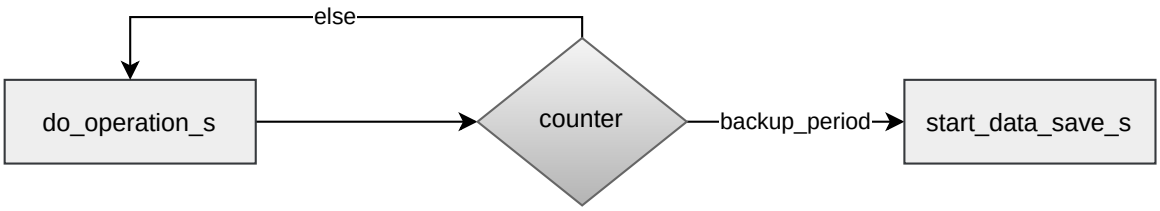


Figure 3.5: Backup condition CB

Like shown in figure 3.5, the condition to start a backup procedure is that a internal entity **counter** reach the value specified in the port called **period_backup_clks**.

3.1.4 FSM TB

TB (Task Backup) means that the backup is done only when counter (or in general the architecture) reach a prefixed value.

Backup condition

Like shown in figure 3.7, the condition to start a backup procedure is that **volatile_counter_1** reach the value specified in the signal called **task_complete_val_counter** (or its multiple).

```
1 if ((to_integer(unsigned(volatile_counter_val)) mod task_complete_val_counter) = 0) then
```

As show in a code above, the "if" condition has operator "mod" inside, and this can be only simulated and not synthesized. So if you want to implement in hardware the TB architecture you must take care about the necessary change of this condition.

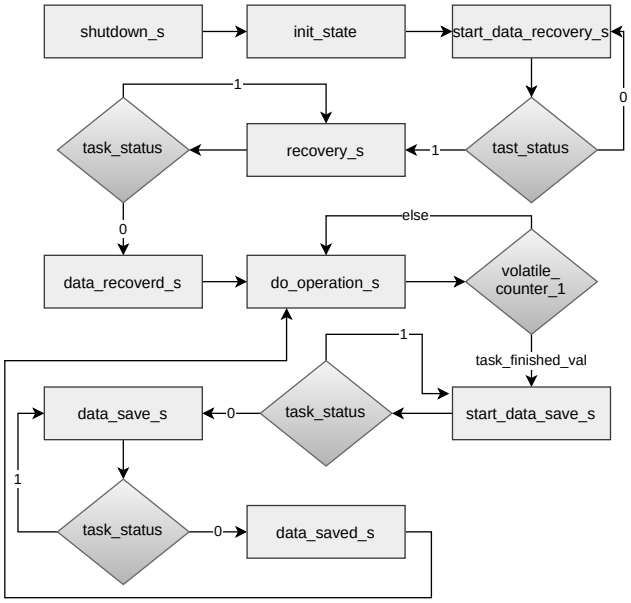


Figure 3.6: tb_fsm

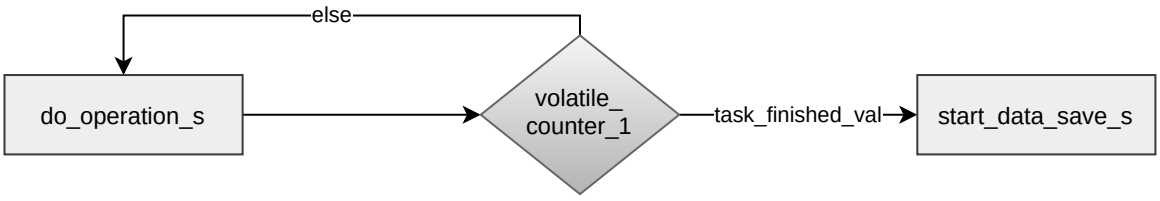


Figure 3.7: Backup condition TB

3.2 Volatile Architecture

3.2.1 Description

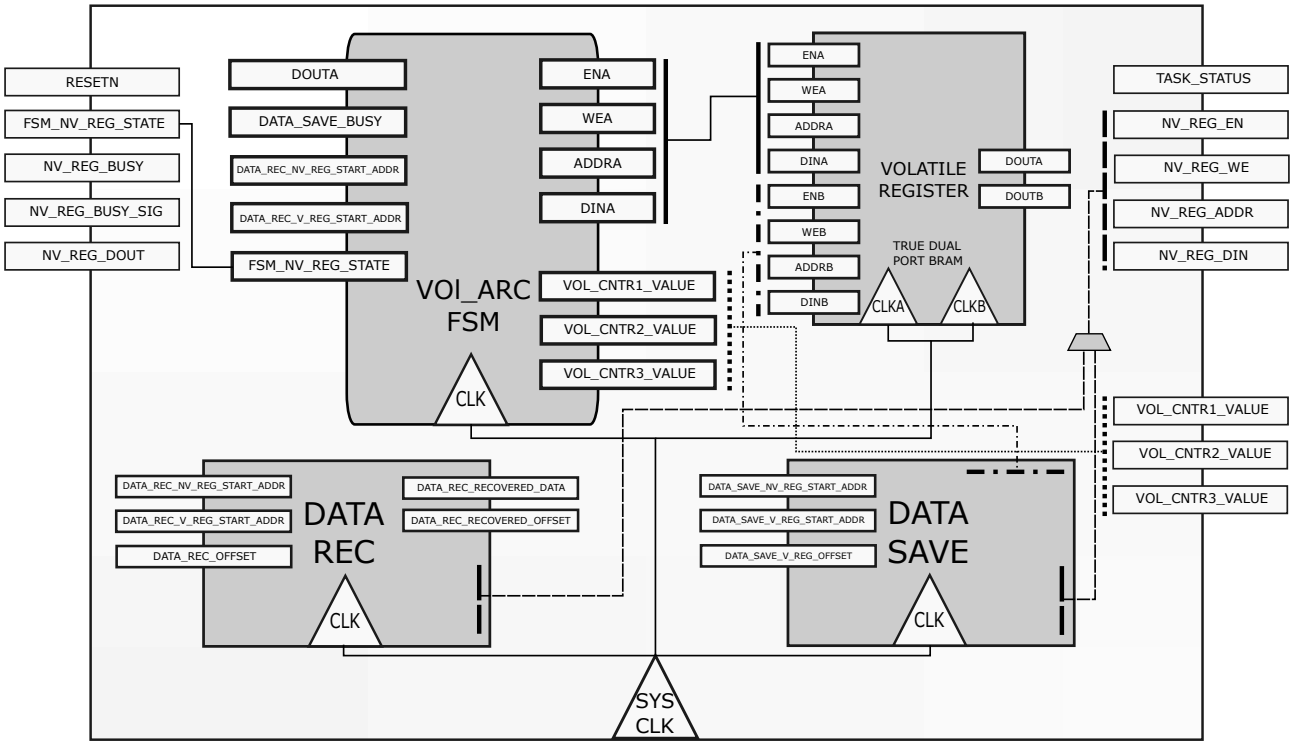


Figure 3.8: Volatile Architecture representation

the full entity is not representable because of its magnitude so a simpler and concise version is shown

The volatile architecture (**VARC**) is one of the components of the test architecture.
This entity is the user code that eventually will be written by the end user using the non volatile memory emulation framework (**NVMEF**). In this instance a simple set of counters was chosen to represent the unit that uses the **NVMEF** .
By itself the counters are maybe four lines of VHDL code but the **VARC** is much more extensive since it must be compatible with the transient based computing. To achieve this task the **VARC** listens and behaves following the imperative orders of a **FSMNVR** implementation.
While the **FSMNVR** tells only what to do, the real implementation of such operations are inside the **VARC** thus the magnitude of the entity. Such task are: the process that recovers the previously saved data inside **NVR** (**DATA_REC**), the process that stores the data inside **NVR** (**DATA_SAVE**) and the final state machine that reacts to **FSMNVR** (**VOL_ARC_FSM**) and controls the volatile register (**VARC**) where the counter values are stored.
In the next paragraphs a brief explanation of the inner components will be presented.

> While the volatile counter implemented in the volatile architecture is the true test module that has nothing to offer then just the proof of concept, the processes surrounding such architecture (**DATA_SAVE** and **DATA_REC**) are reusable for future test architectures or at least their concept designs.

DATA_REC

The data recovery process is responsible of recovering data from **NVR** when the **FSMNVR** tells the **VARC** to enter such state (**fsm_nv_reg_state** = **start_data_recovery_s**), typically happens after a simulated power shortage.
This process is configurable in how much data is restored from **NVR** , where the data will be fetched from **NVR** and where data will be stored in **VR** .
N.B. only contiguous memory space is recoverable after the process starts. To recover sparse and non contiguous memory spaces from **NVR** , multiple consecutive instances of this process must be called.
The signals responsible to configure the data recovery process behaviour are:

- **data_rec_nv_reg_start_addr**: the **NVR** memory address from which **DATA_REC** process will start fetching data
- **data_rec_offset**: the offset used to calculate the last **NVR** memory address from which **DATA_REC** will recover data. This offset will be summed to the previous signal so it defines how much data will be fetched from **NVR** and saved to **VR**
- **data_rec_v_reg_start_addr**: the **VR** memory address where **DATA_REC** wil save the recovered data

$\Rightarrow$ the values assigned to such signals must be in range with the maximum size of the relative memories:

$$\textbf{data\_rec\_nv\_reg\_start\_addr} + \textbf{data\_rec\_offset} \leq \texttt{NV\_REG\_WIDTH}$$

$$\textbf{data\_rec\_v\_reg\_start\_addr} + \textbf{data\_rec\_offset} \leq \texttt{V\_REG\_WIDTH}$$

During the process operation **DATA_REC** provides the recovered value through **data_rec_recovered_data** and its position inside **NVR** using **data_rec_recovered_offset**. Those signals are then used by sister processes of **DATA_REC** that manage the **VR** during recovery.

Those configuration signals combined with the ones of the **DATA_SAVE** process enable the volatile architecture to *travel in time* or to define special behaviours. An example will be provided in the next section.

DATA_SAVE

The data save process is responsible of saving data into **NVR** from **VR** when the **FSMNVR** tell the **VARC** to do so (**fsm_nv_reg_state** = **start_data_save_s**), this is purely dependant on the type of backup policies implemented inside **FSMNVR** .
This process can be configured with respect to which data will be stored inside **NVR** from **VR** and where it will be stored in **NVR** .
The signals responsible to configure the data recovery process behaviour are:

- **data_save_nv_reg_start_addr**: the start **NVR** memory address where **DATA_SAVE** process will store data fetched from **VR**
- **data_save_v_reg_start_addr**: the start **VR** memory address from where **DATA_SAVE** will fetch data that will be saved inside **NVR** .
- **data_save_v_reg_offset**: the offset used to calculate the last **VR** memory address from which **DATA_SAVE** will fetch data. This offset will be summed to the previous signal so it defines how much data will be fetched from **VR** and saved to **NVR**

This configuration signals are a feature, they enable the volatile architecture to manage the **NVR** in multiple ways. An example is proposed: suppose a micro-controller capable of transient computing that implements the **NVMEF** is used. The process in execution on the micro-controller can be summarized with the contents of its process address space (**PAS**) that are stored in volatile memory/register. Since the micro uses the framework and the structure of the test architecture of this paper is safe to assume that it saves its status or **PAS** in **NVR** when the **FSMNVR** tells to do so. With such hypotheses the **VARC** can perform some careful and wise decision (implemented by the developer) on how to store **PSA** in **NVR** . A possible decision could be to save the current

PSA on a new memory space inside **NVR** instead of overwriting the previous one:
⇒ *this is achieved by changing the value of* ***data_save_nv_reg_start_addr****!*

With such option a way to remember the position of each different **PSA** must be found hence the introduction of the *transient program counter* (**TPC**) stack. Each **TPC** points to his relative **PSA** and every time a new instance of the **PSA** is made a new **TPC** entry in the stack is placed.
This leaves the developer an option to choose where to restart the process after a power shortage and effectively branching the normal process flow. The **NVR** could be seen like a version controlled folder explorable by with commits and branches.

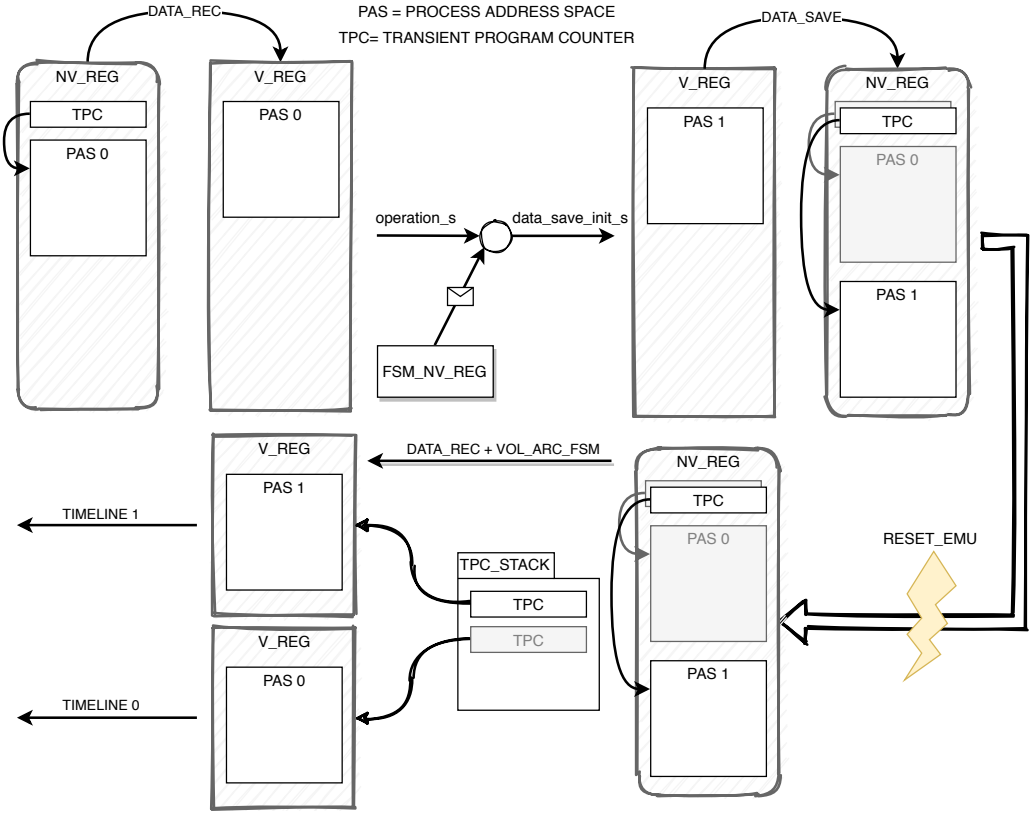


Figure 3.9: Use of the configuration signals of DATA_REC/SAVE in a transient computing micro-controller

VOL_ARC_FSM

Fig 3.10 represents the finite state machine that reacts to **FSMNVR** orders and that manages the volatile counters i.e. fetching, waiting and storing of **vol_cntr[X]_val** from **VR** .
The darker dotted lines are the main escape points from which the fsm could changes operation mode if so told by **FSMNVR** . This occurs when **FSMNVR** orders to perform anything other than **do_operation_s** when in such state or to perform **do_operation_s** when in **data_save_s**. The curved arrow is the cyclic behaviour of the fsm when in normal operation (no interruptions from **FSMNVR**), i.e. reading **vol_cntr1**, increasing its value, than do this for **vol_cntr2** and **vol_cntr3** and restart from the beginning.

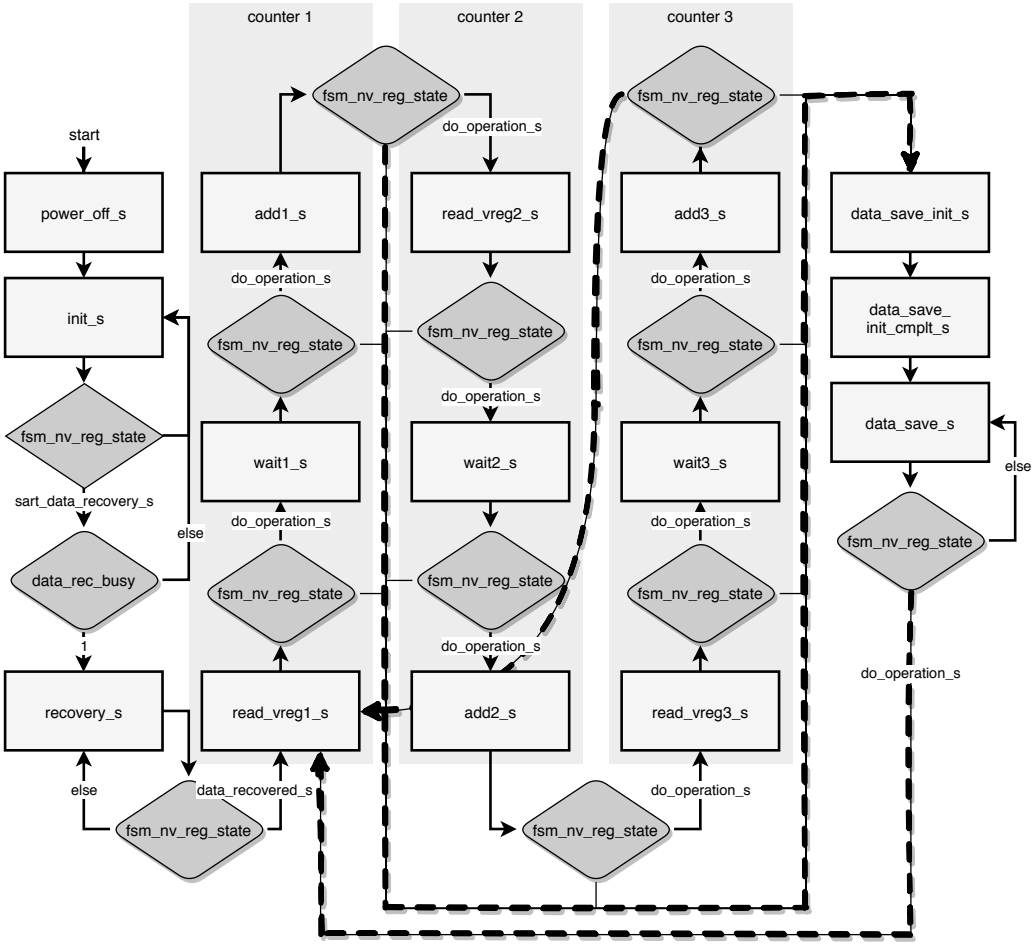


Figure 3.10: Volatile architecture FSM

3.2.2 Port Description

```
entity vol_arc is
    port(
        sys_clk             : in  STD_LOGIC;
        resetN              : in  STD_LOGIC;
        fsm_nv_reg_state    : in  fsm_nv_reg_state_t;
        task_status         : out STD_LOGIC;
        nv_reg_en           : out STD_LOGIC;
        nv_reg_busy         : in  STD_LOGIC;
        nv_reg_busy_sig     : in  STD_LOGIC;
        nv_reg_we           : out STD_LOGIC_VECTOR(0 DOWNTO 0);
        nv_reg_addr         : out STD_LOGIC_VECTOR(nv_reg_addr_width_bit-1 DOWNTO 0);
        nv_reg_din          : out STD_LOGIC_VECTOR(31 DOWNTO 0);
        nv_reg_dout         : in  STD_LOGIC_VECTOR(31 DOWNTO 0);
        vol_cntr1_value     : out STD_LOGIC_VECTOR(31 DOWNTO 0);
        vol_cntr2_value     : out STD_LOGIC_VECTOR(31 DOWNTO 0);
        vol_cntr3_value     : out STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
end vol_arc;
```

IN **sys_clk**: the clock source of the architecture

IN **resetN**: the global reset for the whole system, active low, routed to **MAIN_RESETN**

IN **fsm_nv_reg_state**: current state of the **FSMNVR**

OUT **task_status**: this signal tells the **FSMNVR** the status of the current ongoing operation inside the **VARC** . The signal is used for synchronization purposes

IN,OUT **nv_reg_X**: **NVR** signals

OUT **vol_cntr1_value**: the value of the first volatile counter

OUT **vol_cntr2_value**: the value of the second volatile counter

OUT **vol_cntr3_value**: the value of the third volatile counter

3.2.3 Behaviour

The **VARC** obeys orders dispatched from **FSMNVR** but during normal operation (**fsm_nv_reg_sate** = **do_operation_s**) the values of **vol_cntr[X]_val** change in this way:

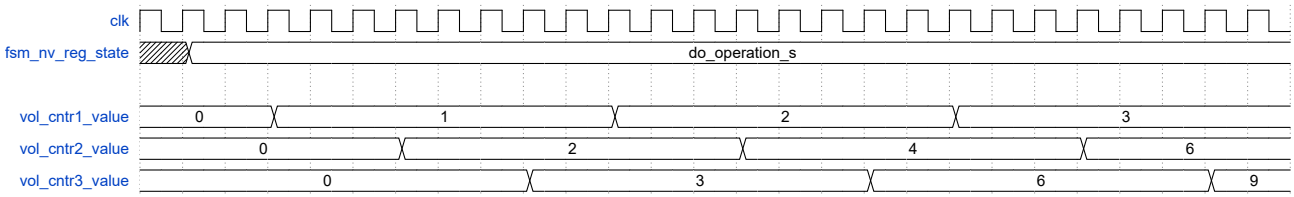


Figure 3.11: Behaviour of volatile architecture under normal conditions

Chapter 4

Project code and usage

All the necessary components needed to implement a **NVMEF** and its test architecture are described in the previous chapters. To simulate, manage and describe the whole project additional files are necessary. This chapter describes the most important components used to run the project's code.

4.1 Project Directory

```
NVMEF-FPGA/
├── doc/ ..... Documentation of the project
│   └── resources/ ..... Images and pdfs used by the documentation
├── scripts/ ..... Scripts used for documentation and project management
│   ├── characterization/ ..... Characterization scripts
│   │   ├── results/ ..... Results of characterization scripts
│   │   ├── tcl/ ..... Tcl batches produced by characterization scripts
│   │   └── generate_[XX]_results.py ..... Characterization python script
│   ├── voltage_trace_gen/ ..... Scripts for voltage trace generation
│   └── generate_plots.py ..... Script that generates graphs for the given results
├── src/ ..... Source files
│   ├── [entity].vhd
│   ├── top_level.vhd ..... Top level entity
│   ├── COMMON_PACKAGE.vhd ..... Package with project constants and global variables
│   ├── TEST_ARCHITECTURE_PACKAGE.vhd ..... Package with test architecture constants
│   └── NVME_FRAMEWORK_PACKAGE.vhd ..... Package with NVMEF constants
├── test/ ..... Testbench files and waveconfig properties
│   ├── [entity]_testbench.vhd
│   ├── [entity]_testbench.wfcg
│   └── characterization_testbench.vhd ..... Characterization testbench
└── vivado/ ..... Vivado project directories
```

4.2 Project Details

The code was written in VHDL and the project was developed on Vivado 2020.1. Porting this code to another toolchain is possible by reading **Extensibility** chapters of every entity described in this paper. Particular care must be taken for IPs.
External libraries used in this project:

```
1    IEEE.STD_LOGIC_1164.ALL;
2    IEEE.NUMERIC_STD.ALL; --for unsigned
3    IEEE.MATH_REAL.ALL; --for log2 and ceil
```

4.3 Top Level

The top level entity implements the model represented in 1.1 by effectively connecting all pieces. The ports of such entity will be connected directly to FPGA's internal or external pins in accordance to the constraints file. Such ports are user definable, but a useful example used in this paper is:

IN **sys_clk**: FPGA's clock

IN **global_resetN**: FPGA's reset (typically routed to a user button)

OUT **IPC_start_evaluation**: port used to start evaluation by the **IPC**

OUT **IPC_evaluation_ready**: **IPC** calculation status

IN **IPC_num_state_to_evaluate**: which power state to calculate power consumption upon to

OUT **IPC_output_data**: the computed value by **IPC**

OUT **val1**: **VCNTR** 1 value

OUT **val2**: **VCNTR** 2 value

OUT **val3**: **VCNTR** 3 value

The backup behaviour depends on the type of **FSMNVR** used, as such all 3 types of finite state machines are included in the code but commented out. When a different type of backup policy wants to be used it is sufficient to un-comment the proper instance and comment the previous one. All signals are already routed properly.

In the last lines of code at the end of **top_level.vhd**, some behavioral features of the whole system can be set:

```
    -- IPC trigger: represents power consuption of the vol_cntr
    power_state_en(0) <= '1' when fsm_nv_reg_state_internal = do_operation_s else '0';

    -- IPC trigger: represents power consuption of the framework
    power_state_en(1) <= '1' when nv_reg_en = '1' else '0';

    -- IPC trigger: represents power consuption of the data_save process
    --> (utilization of test_architecture(v_regs) + fsm_nv_reg_X + framework(nv_reg))
    power_state_en(2) <= '1' when fsm_nv_reg_state_internal = data_save_s else '0';


    -- sets reset_emulator threshold
    threshold_value(0) <= RST_EMU_THRESH;
    -- sets the value for the hazard threshold, used by fsm_nv_reg_db
    threshold_value(1) <=hazard_threshold;
    --additional threshold if set in package, used by fsm_nv_reg_db
--    threshold_value(2) <= 210;

    -- index where reset_emulator threshold must be selected from thesehold_value()
    select_threshold <= 0;

    -- hazard state set, used by fsm_nv_reg_db
    fsm_nv_reg_thresh_stats <= hazard when threshold_compared(1) = '1' else nothing;
```

Some of those features can be directly set in the code of **top_level.vhd** like **threshold_value(X)** or when to trigger the **IPC** . The size of **power_state_en** (that is how many power states the sistem has), instead, can be set by modifying constant values in the proper package. This will be described in next chapters.

4.4 Packages

This project has 3 packages that contain constants,functions, states and ad-hoc data structures.
Some of this constants change the behaviour of the whole system so is advised to read the documentation inside these packages.

- **COMMON_PACKAGE**: This package contains functions and constants common to the whole system.
- **NVMEF_PACKAGE**: This package contains constants and data structures relative to **NVMEF** .
- **TEST_ARCHITECTURE_PACKAGE**: This package contains constants and data structures relative to the test architecture.

4.5 TRACE_ROM

The **TRACE_ROM**is a read only memory where the voltage trace is stored. This rom is accessed by the **IE** to compare thresholds and to simulate a shutdown event.
The code for this rom is stored inside **trace_ROM.vhd** and what it needs to run is a set of values to add inside the body of the component. The value of the rom can be any positive 32 bit integer number, for this project values where voltages in the order of mV. The maximum value reached by the trace and the quantity of such values must be coherent with **INTERMITTENCY_NUM_ELEMNTS_ROM** and **INTERMITTENCY_MAX_VAL_ROM_TRACE** relatively.
The rom is accessed at a speed of **MASTER_CLK_SPEED_HZ/INTERMITTENCY_PRESCALER**.

4.6 IP modification

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

Chapter 5

Results

First let's introduce some notes. One of the reasons why the test architecture has been implemented was to demonstrate how **NVMEF** can optimize **VARC** operations in case the power source is unstable. As we know by there previous chapters this accomplished by the use of a **FSMNVR** .
This project has 3 types of **FSMNVR** with their relative backup polices, each of those will be tested and results on their effectiveness will be presented in this chapter:
3 backup policies, 2 types of tests = 6 simulations. For each simulation multiple runs will be executed with different constraints to show how the policy performs.
Backup Policies:

- **Task backup (TB):** **FSMNVR_TB** as **FSMNVR** is used. For each run a different task value will be used (constraint). This will show the best and worst results on the voltage trace by the selected test.
- **Constant intermittent backup (CB):** **FSMNVR_CB** as **FSMNVR** is used. For each run a different backup period will be used (constraint). This will show the best and worst results on the voltage trace by the selected test.
- **Dynamic intermittent backup (DB):** **FSMNVR_DB** as **FSMNVR** is used. For each run the hazard state voltage threshold state will be changed (constraint). This will show the best and worst results on the voltage trace by the selected test.

Test types:

- **Fixed time test:** The goal is to run for a fixed time the test architecture (a reachable value). Once such goal is reached the state of **VARC** will be captured and stored.
- **Fixed value test:** The goal is to run the test architecture until a prefixed state is reached by **VARC** (in our case **VCNTR** should reach a fixed value). **If** the goal is reached the state of **VARC** will be captured.

It is important to notice that all found results deeply depend on the voltage trace in use. Different voltage traces will lead to different results. For example if the voltage trace is constant and above **reset_emulator** threshold for the whole simulation time, there will be no differences between backup policies.
Technical note on the voltage trace: such trace is implemented as a ROM that is queried constantly at **MASTER_CLK_SPEED****INTERMITTENCY_PRESCALER** Hz, obviously this ROM has an end. In such case the voltage trace/ROM value, restart from beginning, in this way voltage trace/ROM lasts as much as needed but with repetitive values after a certain time.

5.0.1 About the voltage trace

To simulate a real case scenario, with regards to power usage, this data was adopted: BatterylessSim.
Between all available traces, `2.txt` was the chosen one for its rapid changing behaviour.

Trace Data Processing

To reduce the size and memory footprint of the synthesized ROM the data set was compressed in a non lossless way:

- voltage values were cut to the third decimal point (ROM word size reduction)
- number of samples cut from 39232 to 1570 by taking a mean value out of every 25 voltage values

This compression maintains a reasonable enough voltage trace. **`INTERMITTENCY_PRESCALER`** is used to regulate the time it takes to evaluate entirely the voltage trace in simulation. So in this simulations $f_{clk} = 100$ MHz $\rightarrow f_{trace} = 12,5$ MHz making the voltage trace full evaluation time equal to $125,6\ \mu s$.

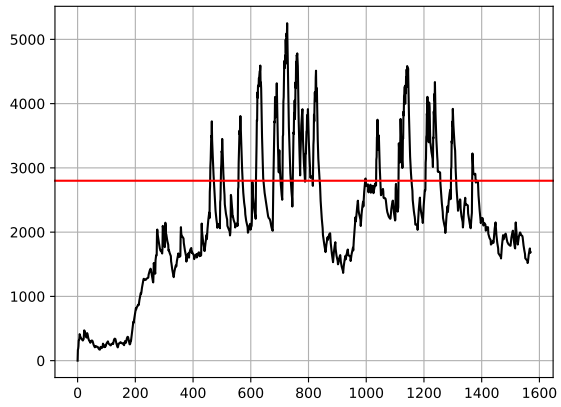


Figure 5.1: Voltage trace

5.1 Dynamic backup policy results

5.1.1 Simulation's predefined constant values

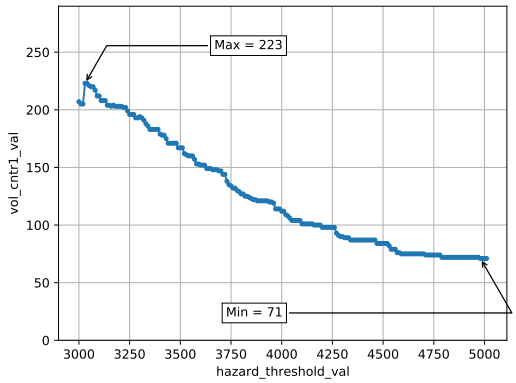
In this simulations the shutdown threshold is equal and fixed to 2800, i.e. the voltage at which the test architecture will shut down is at 2.8 V. (red line in figure 5.1). 75% Of the trace is under the shutdown threshold.

5.1.2 Simulation's predefined variable values

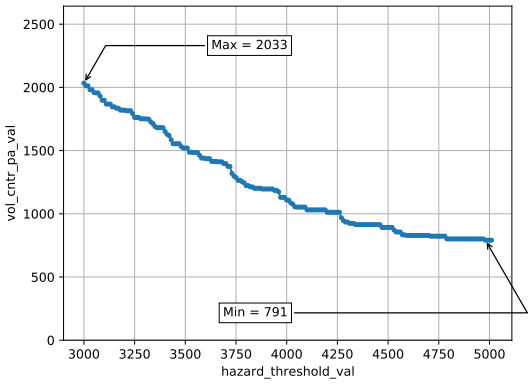
Each run of the simulation differs by the value of the hazard threshold (**`threshold_value[1]`**): from 3000 mV to 5000 mV with each run increasing by 20 mV.

5.1.3 Fixed time results

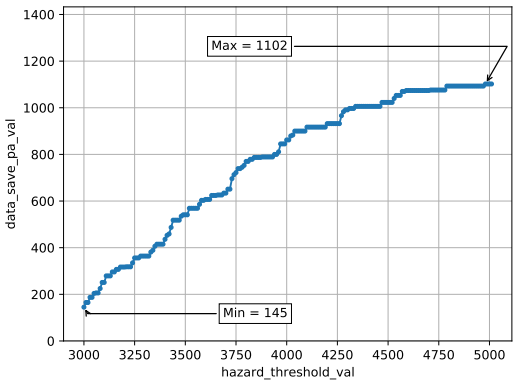
In this type of test the test architecture is run for multiple times at different **`hazard_threshold_val`** values and stopped every time at 100 μs hence fixed time. What we want to know is the reached value of **`VCNTR`** 1.



In this figure we can see the behaviour of **val_cntr1_val** (counter 1 value) during each run of the simulation. The proposed graph is globally decreasing, this is in fact what we expect: if the threshold increases the reached value after $100\ \mu s$ will decrease because **FSMNVR_DB** will not be able to backup. This concept is understandable best by looking at the voltage trace and understand when **FSMNVR** will backup.
The optimal[1] threshold value is equal to $3040\ mV$ because in such run the greatest value for **val_cntr1_val** is reached.

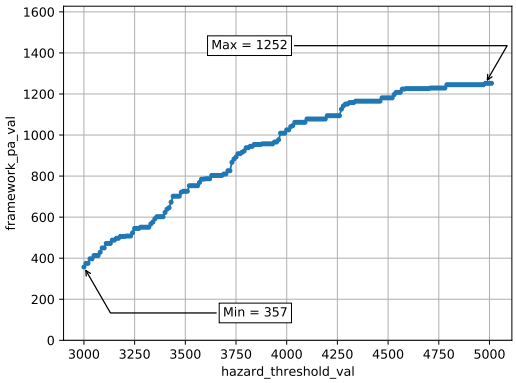


This figure represents the power consumption of **VCNTR** as the number of clk cycles it was active. This is in fact the value of **val_cntr_pa_val**. The plot graphs this value for each run of the simulation that differs by the value of hazard threshold **hazard_threshold_val**.
The profile of this plot resembles the previous one: in this graph too the more the threshold increases the less time **VCNTR** will be operational and most time will be spent on data save (look at the voltage trace).
The max value for **vol_cntr_pa_val** is $2033\,clks$ for **hazard_threshold_val** equal to $3000\,mV$, which is less then the optimal[1] threshold value. This is exact. Intuitively this happens because most of the time is spent on computing than performing saves: the voltage trace is above hazard threshold more that it is between hazard and shutdown.
Instead for **hazard_threshold_val** equal to $3030\,mV$ (the optimal[1] threshold value), **val_cntr_pa_val** equals to $1981\,clks$.



The power consumption of the data save process (**data_save_pa_val**) for each run is the sum of the power used by **VR** , **FSMNVR** and **NVR** . This curve explains exactly what was told previously. For small values of **hazard_threshold_val** **FSMNVR_DB** spends most time in computing mode hence the small values of **data_save_pa_val**. While if the threshold is ridiculously high the system is only able to do saves of the current state and perform practically no operation time.

[1] To this particular voltage trace



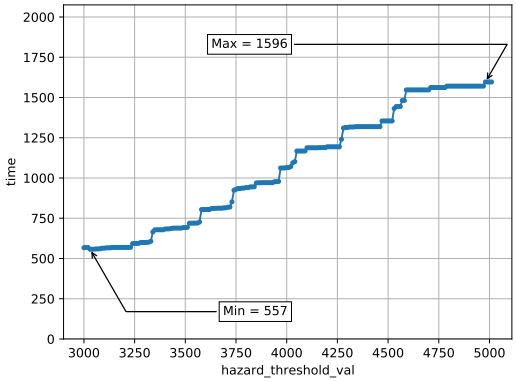
In this figure we can see the framework power approximator value (**framework_pa_val**) at the end of each run (always with respect to **hazard_threshold_value**changes). In this case the **PA** approximates the usage of framework components, in particular **NVR** . This graph is a direct analogy of the previous one, but with the addition that **NVR** are used during backup too (after **reset_emulator** goes off), so for each run additional utilization time is consumed (at threshold equal to 5000, **data_save_pa_val** = 1102 while **framework_pa_val** = 1252).

For each run such values were found constant after simulation:

- **clk_counter** $= 2513$: each run performs 2513 clock cycles.
- **shtdwn_counter** $= 12$: each run has 12 shutdowns (the voltage level goes under the **reset_emulator** threshold) and **FSMNVR_DB** has to do 12 recoveries.
- **trace_rom_addr** $= 1250$: each run consumes 1250 values of the voltage trace. This is useful to understand how the **FSMNVR** works in each run by knowing what part of the voltage trace will be used.

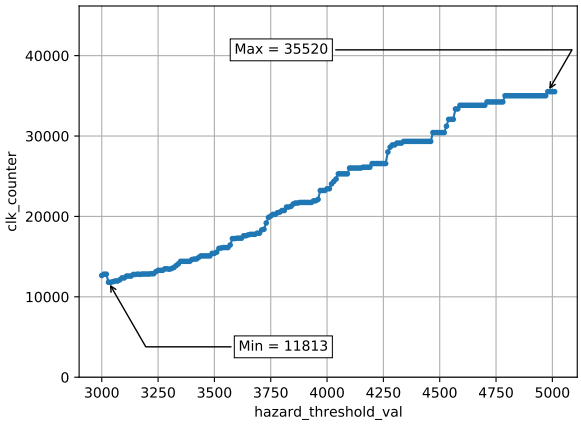
5.1.4 Fixed value results

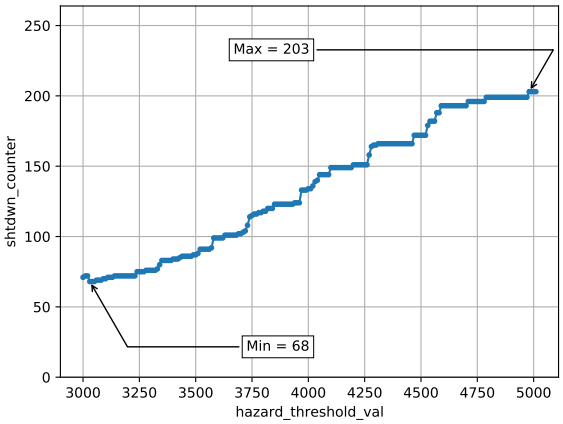
In this type of test the test architecture is run for multiple times at different **hazard_threshold_val** values and stopped every time **val_cntr1_val** reaches 1000, hence fixed value. What we want to know is how much time does it take.



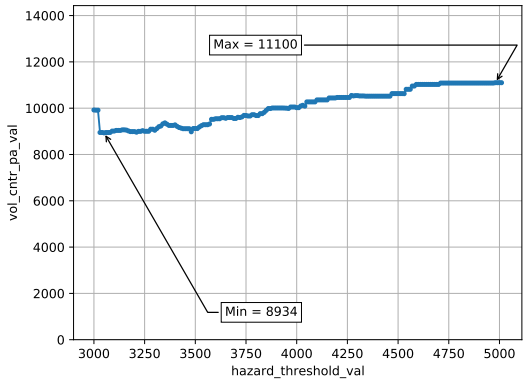
This graphs plots the time (μs) it takes to perform each run while **hazard_threshold_val** changes.
As expected this plot is globally monotonous increasing. While the threshold gets bigger less time is spent on operation time (increasing **VCNTR**) while only data save are performed (the voltage threshold fails to exceed the hazard one). This results in small losses after a shutdown but with less progress (this will be visible in the others graphs). On the same note if the threshold value is too near the shutdown threshold, **FSMNVR_DB** will perform less data save resulting in losses after a shutdown. This is visible by the fact that the optimal [1] threshold is not the smallest value ($3000\,mV$) but $3030\,mV$ with a time of $557395\,ns$ (means 4+ times a full voltage trace run). This corresponds with the value found in the previous test!

[1] To this particular voltage trace

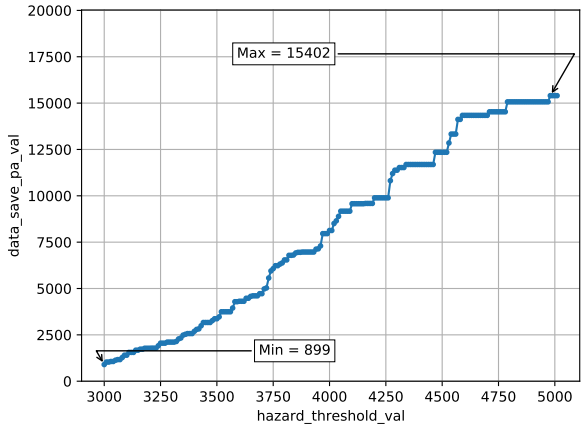


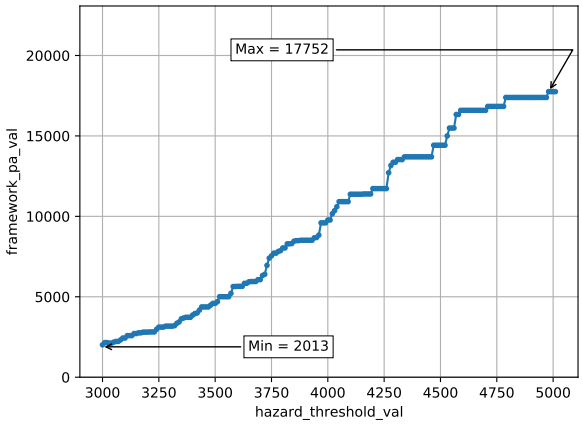


This two graphs tell the same thing but from different points of view. The first one plots the behaviour of **`clk_counter`**: the number of useful clock cycles where test architecture is on (**`reset_emulator`** is off). The second graph, on the other hand, plots the behavior of **`shutdown_counter`**: the number of of clock cycles where **`reset_emulator`** is on and the test architecture is consequently off, i.e. the time spent in shutdown.
From this graphs it is clear that the more the threshold increases the more time it takes to perform the fixed task (**`clk_counter`** in fact gets bigger) and so more shutdown are experienced by the test architecture.
As usual near $3000mV$ a change in steepness is found. This happens because if **`hazard_threshold`** is near **`shutdown_threshold`** less time is spent in data save (see how **`FSMNVR_DB`** works and see fig 5.1), this means that from time to time data is lost because backups where not to quick thus more time spent to complete the fixed task.



This graphs plots the power utilization (as number of clock cycles of activity) of the **`VCNTR`** . In theory this graph should be constant from $3030\,mV$ or at least globally monotone decreasing. The total utilization time (therefore **`vol_cntr_pa_val`**), of **`VCNTR`** should remain the same after the point of no data loss (**`hazard_threshold`** $> 3030mV$) because **`VR`**s are saved more often. This clearly is not what is shown. After code examination a bug was found: extra false positives are captured by **`PA`** (see subsubsection 6.0.1). This bug decreases precision of this **`PA`** instance the longer the test is run hence the increasing behaviour.
Nonetheless a reasonable constant behaviour can be seen. While the full graph is mostly incorrect, its first part (3000 to $3200\,mV$) is correct. In particular it is possible to see the region ($< 3030\,mV$) where the data save process has not enough time to complete and values are lost, i.e. more **`VCNTR`** consumption, and where a proper value is used ($\geq 3030\,mV$), i.e. sudden drop by **`PA`** .





The last 2 graphs of this test have similar commentary to the ones of fixed time simulation (see previous chapter).

5.2 Constant backup policy results

5.2.1 Simulation's predefined constant values

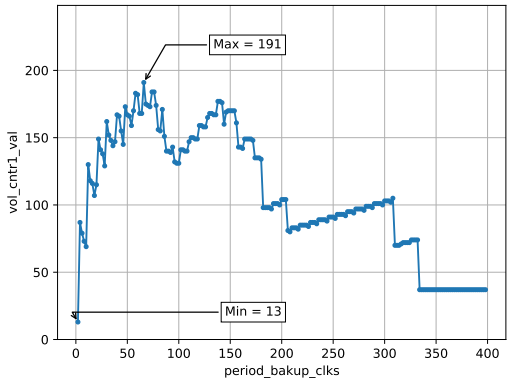
In this simulations the shutdown threshold is equal and fixed to 2800, i.e. the voltage at which the test architecture will shut down is at 2.8 V. (red line in figure 5.1). 75% Of the trace is under the shutdown threshold.

5.2.2 Simulation's predefined variable values

Each run of the simulation differs by the duration at wich **FSMNVR_CB** backups, i.e. **period_backup_clks**. Values range from periods of 0 clks to periods of 400 clks, with each run increasing by 2 clks.

5.2.3 Fixed time results

In this type of test the test architecture is run for multiple times at different **period_backup_clks** values and stopped every time at 100 μs hence fixed time. What we want to know is the reached value of **VCNTR** 1.

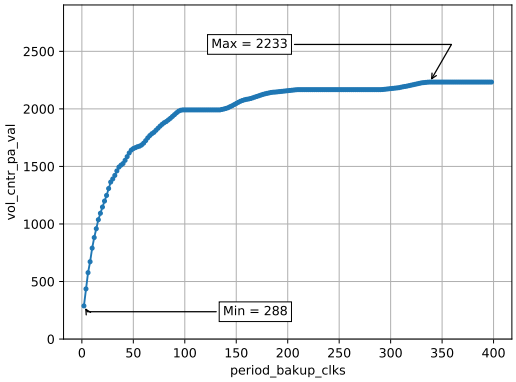


The behaviour of this graph is what is to be expected: for unusable backup periods the reached value by **VCNTR** after 100 μs is small (too much time spent on backup than on operations) and increases together with the period. Once the optimal[2]value is reached (**period_backup_clks** = 66 clks, **val_cntr1_val** = 191) the systems starts to loose information. During a period shutdowns occur prior to a backup. This continues until a maximum acceptable value is reached (**period_backup_clks** = 334, **val_cntr1_val** = 37) and from that point the system just can't keep up with shutdowns, therefore **VCNTR** increases but without frequent enough saves.

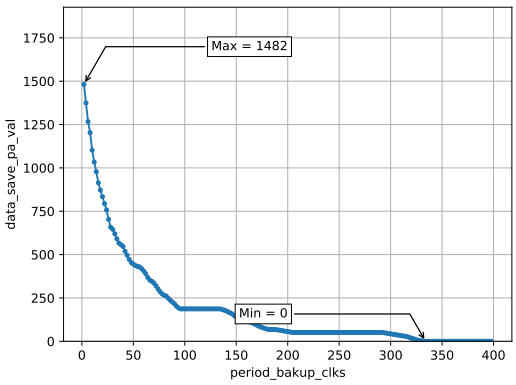
The step behaviour of this plot (and of the consecutive simulations too) is explainable in this way: if there are 82 clks of good operational time (voltage trace over **shutdown_threshold**), **period_backup_clks** = 66 and a data save lasts 15 clks then in real terms we have 66 clk cycles where **VCNTR** increments then 15 clks where it is saved and then shutdown. While if **period_backup_clks** = 68 then 68+15 = 83 that is more than the available 82 clks of good operational time, hence all 68 useful clks are lost because backup didn't complete.

[2]For this particular voltage trace

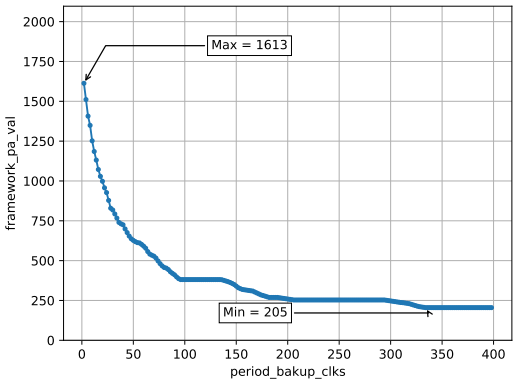
This type of backup policy is really sensitive on the available operational time the voltage trace has and how the processes uses it. It can be though as a game of tetris where the length of the bottom row changes frequently.



As expected the power usage approximation for **VCNTR** increases with the backup period.
A part from the correct behaviour of this graph 3 intervals stand out as they maintain the same value over time **period_backup_clks**: A:[98,134],B:[210,288] and C:[338,398]. This is strange because in theory by increasing **period_backup_clks**, **vol_cntr_pa_val** should increase too since more time is spent in computational time. While this is not the case **vol_cntr1_val**, in the same periods,increase too. Lets break down what happens. For period C the reason is simple: all periods are not compatible with the voltage trace, in the previous graph that region sees no increase in **vol_cntr1_val** meaning that the architecture is not able to save its state. On the other hand, periods A and B are similar. The solution to such behaviour is visible in the next graph.



In regions A and B, this graph too, has the same constant behaviour. What this tells us is: for each **period_backup_clks** in interval A/B, the number of clks spent in save operations are the same. If the usable time of the run (where voltage trace is over shutdown threshold) is X and the total time spent in save operations and recovery is Y than the operational time is $X - Y = Z$. By knowing that X is constant by definition ($100\,\mu s$) and Y by looking at the graph, Z must consequently be constant. So the real question is why the time spent in save operation the same. The simple answer is: the voltage trace. The voltage trace in our hands lets the architecture to save it's state the same amount of time for each period value that resides in A/B. So this means that (like said in the first graph of this test) the period of backup is strictly tied to the voltage trace resulting in strange properties where some periods "resonate" with the trace.
A part from this exceptions the behaviour of this graph is in line with the theory: for smaller backup periods more saves are performed so more power is drawn by the save process. The bigger the period gets the less time is spent on saving the architecture state thus lees power used.



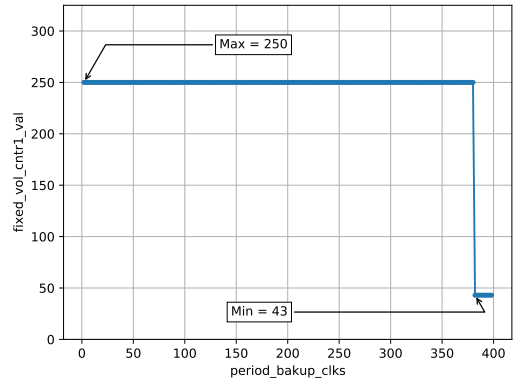
This graph is the dual counterpart of the one above just for the framework power approximation. The only difference is that a constant minimum value of power is used, this being the times when the framework has **NVR** used for recovery.

For each run such values were found constant after simulation:

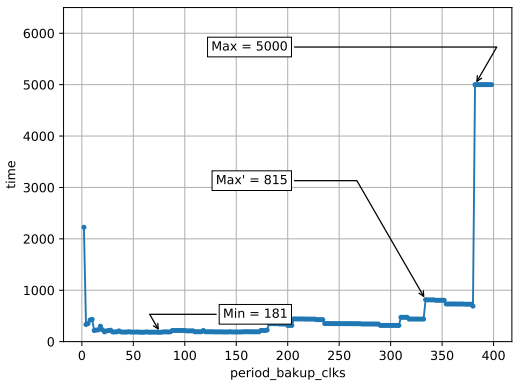
- **clk_counter** $= 2513$: each run performs 2513 clock cycles.
- **shtdwn_counter** $= 12$: each run has 12 shutdowns (the voltage level goes under the **reset_emulator** threshold) and **FSMNVR_CB** has to do 12 recoveries.
- **trace_rom_addr** $= 1250$: each run consumes 1250 values of the voltage trace. This is useful to understand how the **FSMNVR** works in each run by knowing what part of the voltage trace will be used.

5.2.4 Fixed value results

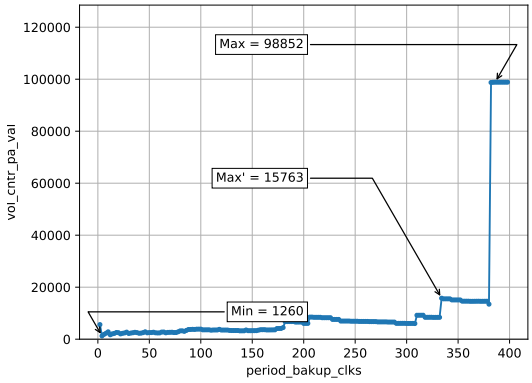
In this type of test the test architecture is run for multiple times at different **period_backup_clks** values and stopped every time **val_cntr1_val** reaches 250, hence fixed value. What we want to know is how much time does it take.



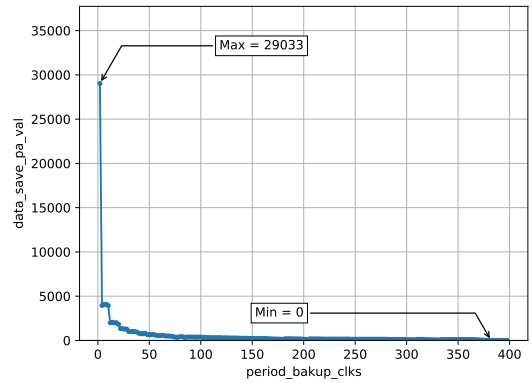
In theory all tests of same type should have been similar as to be comparable afterwards. In this instance it was not possible to reach 1000 on every run of the simulation since it would have taken too much time to run all tests and because just for a period of backup grater then 50 this value seemed to be unreachable in proportional times. So a more appropriate value of 250 for **val_cntr1_val** was chosen.
As clearly visible, for a **period_backup_clk** of over 382, such fixed value can't be reached. This happens because the voltage trace does not have periods of usable time grater then 382 clks where operation, recovery and save could be performed.



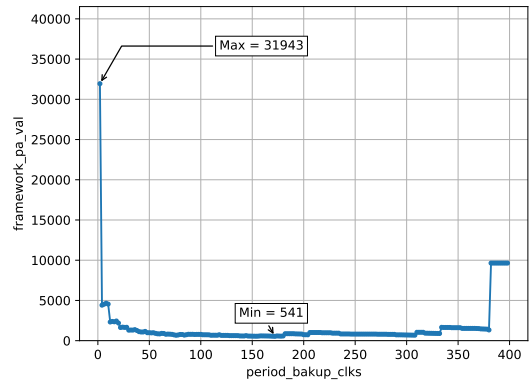
As expectable, for **period_backup_clks** too small the necessary time to complete the task skyrockets.
While the value becomes grater a local time minima can be found for **period_backup_clks** = 76 and **time** = $181.685\,\mu s$. The worst times are: **period_backup_clks** = 2 **time**= $2.224645\,ms$ and **period_backup_clks** = 364 clks with a total **time** = $815.935\,\mu s$, relatively worst time for too small period and worst time for inadequately big period.



The approximated power use of **VCNTR** with the optimal[3]backup period (**period_backup_clks=** 76) is **data_save_pa_val** = 2567 clks, but the best[3] approximated power consumption of **VCNTR** is at **period_backup_clks** = 4 with a total power of 1260 clks (x2 factor decrease).
What this tells is that if the most power hungry component of the architecture is the **VCNTR** then is best to spend extra time in saving the status thus not loosing a lot of data after shutdowns (**period_backup_clks** = 4) while if the architecture as a whole is power hungry then is best to complete the task as fast as possible (**period_backup_clks** = 76).



This graph represents the monotonic decreasing behaviour of the data save power approximation simulation. This behaviour is in line with the theory and in the period [382,398] is equal to 0 since the architecture is unable to backup.



The approximated power usage of the **NVMEF** has the same trend of the data save process one (previous graph). But since **NVR** are used in recovery and data save process too this graph can be thought as a sum of the previous one plus the power use of **NVR** that scale in the same way as the timing graph (first one).

5.3 Task backup policy results

5.3.1 Simulation's predefined constant values

In this simulations the shutdown threshold is equal and fixed to 2800, i.e. the voltage at which the test architecture will shut down is at 2.8 V. (red line in figure 5.1). 75% Of the trace is under the shutdown threshold.

[3]For this particular voltage trace

5.3.2 Simulation's predefined variable values

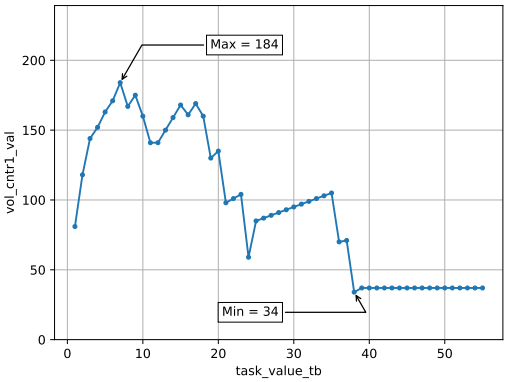
Each run of the simulation differs by when **FSMNVR_TB** backups, i.e. when:

$$(\textbf{vol\_cntr1\_val}\ modulo\ \textbf{task\_value\_tb}\ ) = 0$$

Values range from 0 to 55, with each run increasing by 1.

5.3.3 Fixed time results

In this type of test the test architecture is run for multiple times at different **task_value_tb** values and stopped every time at 100 μs hence fixed time. What we want to know is the reached value of **VCNTR** 1.



The general behaviour of this plot is well in line with what expected. As in the constant backup policy the backup is still time driven but in a different way: the bigger the task the more time it needs to reach it. The difference is that **VARC** is not composed of only one **VCNTR** but three of them. This policy backups on the value of **VCNTR** 1 that increases first after a shutdown and then after **VCNTR** 2/3 have too.
This means that for smaller values of **task_value_tb** the majority of the available operation time is spent on data saves (growing behaviour in graph). Then an optimal[4]spot is reached (**task_value_tb** = 7, **vol_cntr1_val** = 184) and from such point forward the operation time is dominated by computation (**VCNTR** s are incremented) then on saving the status (hence the descending behaviour). At last, a point where the architecture will never be able to backup is reached (**task_value_tb** > 39) and so its state is always lost.
The remaining question is why the curve presents sudden drops followed by gains, this is explained by the next figure.

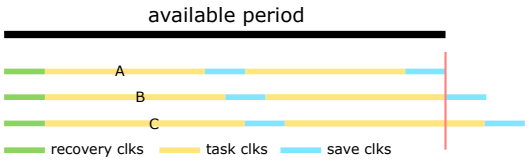


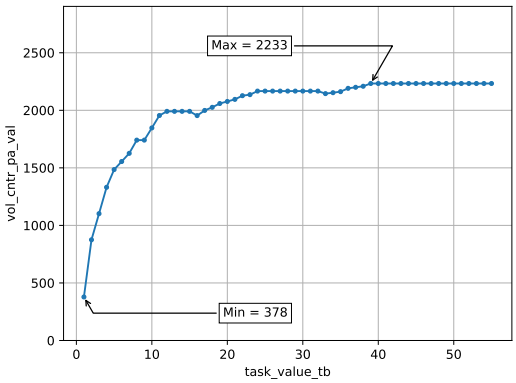
Figure 5.2: period usage

Suppose during a run the system is halted right before the voltage trace starts to rise over shutdown threshold. Now, from this point on, an available period of computable time (*available period* in figure) is available. This time period lasts X clks and starts every time with a recovery (green bar in figure) so the effective computable time is shorter. Now let's examine 3 different scenarios: scenario 1 where **task_value_tb** = A, 2 where **task_value_tb** =B and scenario 3 with **task_value_tb** = C, $A < B < C$ and $B = A + 1, C = B + 1$.
In scenario 1 in the *available period* **VARC** performs the task ans saves it (light blue line in figure) two times, this means that at the end of the period and after a shutdown **VCNTR** = **NVR** = $2A$. In scenario 2, instead, **VARC** (after recovery) performs the task, saves the state, performs another task and looses all the work done because it was not able to save its state. So at the end of the *available period* **VCNTR** = $2B$ and **NVR** = $B << 2A$. The same applies for scenario 3 with **VCNTR** = $C + 0.9C$ and **NVR** = $C << 2A$.

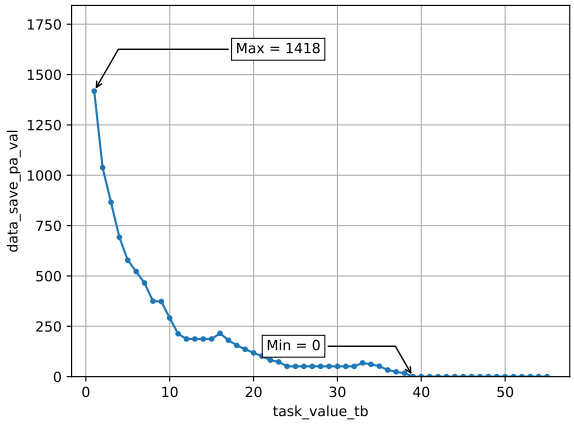
In practical terms this means that after a run with **task_value_tb** = A a sudden **VCNTR** drop is to be

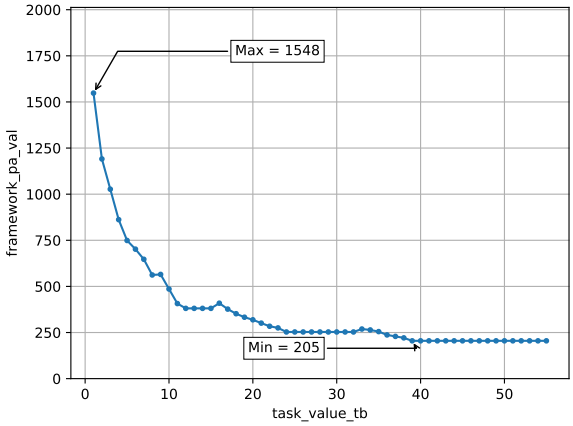
[4]For this particular voltage trace

expected and no further value for **task_value_tb** can increase more then $2A$ **VCNTR** in *available period*.



As usual the **PA** graph for **VCNTR** increases the more the policy gives time to the computation.
The approximated power usage for the optimal[5] **task_value_tb** =7 is 1626 clks.





This two graphs plot the **PA** for the data save process and of the whole framework during each run. The relative values for the optimal[5] are 465 clks and 647 relatively. As we can see **data_save_pa_val** decrease with the growing of **task_value_tb**, this because the architecture will save less times.
PA for the framework is slightly bigger in value because it evaluates the time **NVR** is used: i.e. during data saves and recovery too.

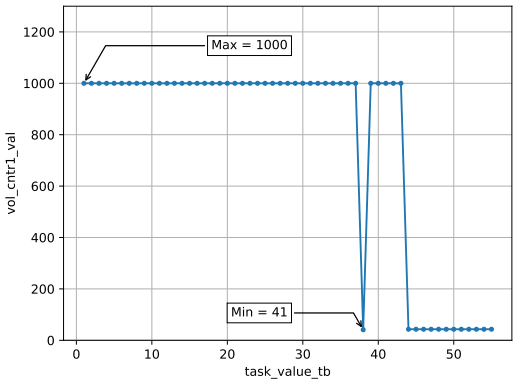
For each run such values were found constant after simulation:

- **clk_counter** = 2513: each run performs 2513 clock cycles.
- **shtdwn_counter** = 12: each run has 12 shutdowns (the voltage level goes under the **reset_emulator** threshold) and **FSMNVR_CB** has to do 12 recoveries.
- **trace_rom_addr** = 1250: each run consumes 1250 values of the voltage trace. This is useful to understand how the **FSMNVR** works in each run by knowing what part of the voltage trace will be used.

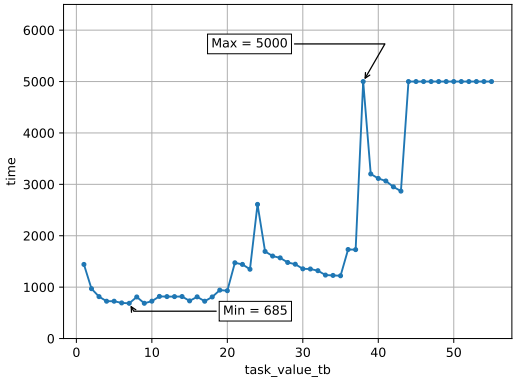
5.3.4 Fixed value results

In this type of test the test architecture is run for multiple times at different **task_value_tb** values and stopped every time **val_cntr1_val** reaches 1000, hence fixed value. What we want to know is how much time does it take.

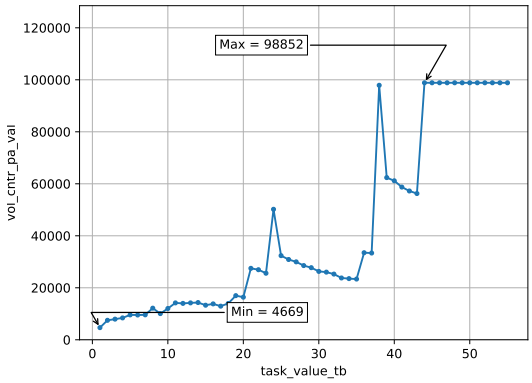
[5] For this particular voltage trace



This graph shows that each simulation reaches the fixed value with the exception of **task_value_tb** =38 where this behaviour is not totally expected (see 6.0.2). From **task_value_tb** $>$ 44 the architecture is no more able to perform it's task since **task_value_tb** is too big for the specific voltage trace.

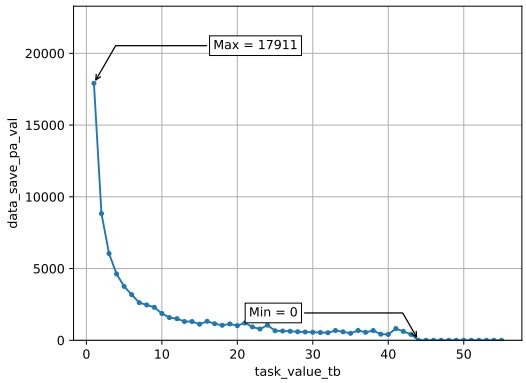


Like in the previous test the optimal[6]value for this type of backup policy is for **task_value_tb** = 7 and time = 685 μs since is the one that takes less time to complete.
A relatively constant behaviour can be seen in range [5,18] for **task_value_tb** this because the majority of computable periods in the voltage trace are used. After that point some periods become useless.
The decreasing behaviour in region [23,35] can be explained with 5.2 on scenarios 2 and 3.
As expected the time increases with **task_value_tb** until a maximum for the trace is reached where even after $5ms$ of time the architecture is unable to complete its task.

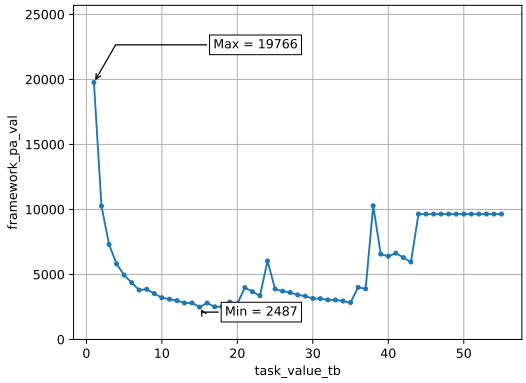


This graph plots the behaviour of the approximted power usage of **VCNTR** 1 in relation to **task_value_tb** . This graph too exibits the typical behaviour of it's type (see same one in other simulations). For small values of **task_value_tb** the utilisation time of **VCNTR** 1 is poor hence it's minimum value directly at the start.
val_cntr_pa_val for the optimal[6] time is equal to 9597 clks.

[6]For this particular voltage trace



The usual consideration for this type of graph are made, look the ones previously. The power usage of the data save process is note in direct relation with the total time spent by the run, as one could think. In fact for longer runs this the total power decreases, this because **task_value_tb** increases and less saves are made.



In contrary to the previous **PA** value the framework one is in relation with the time spent for the run and the value of **task_value_tb** . **framework_pa_val** assumes bigger values for small **task_value_tb** and relatively short runs as well for bigger **task_value_tb** and longer runs.

Chapter 6

Bugs

6.0.1 Bug on `vol_cntr_pa` for `FSMNVR_DB`

FSMNVR_DB has a bug on the way it recovers after a data save process that affects the way **PA** calculates value for all **VCNTR** .
Let's say **FSMNVR_DB** is in **do_operation_s** and the voltage trace goes below the hazard threshold voltage, **FSMNVR_DB** will consequently go to **start_data_save_s** to start the data save process. After such process, when **FSMNVR_DB** reaches **data_saved_s** the machine goes directly to **do_operation_s** state where the voltage trace is checked again if it is below the hazard threshold. This direct movement is the bug that has to be fixed. Since **PA** (for **VCNTR** 1) uses as trigger the state of **FSMNVR_DB** where it is **do_operation_s** this direct transition triggers **PA** even if the voltage trace is below hazard threshold.
This bug gets worse the more the simulation runs while for simulations that last the same and use the same voltage trace such bug only adds a constant value to the end result.

Possible Solution

Make a new state that checks the the voltage trace before doing anything and depending on the state of the trace and the available thresholds decide which operation to do. Then at the end of each operation go back to such state.

6.0.2 Bug 38

FSMNVR_TB has a bug where for **task_value_tb** = 38 the architecture is unable to perform the task in any amount of time. The nature of such bug is unknown. Maybe this behaviour is correct and happens only for this specific voltage trace. Further investigation is required.

Chapter 7

Final Notes

This project's tasks were to implement the framework non volatile architecture and to demonstrate a use case for such. The **VARC** for the test architecture are 3 **VCNTR** on which different backup policies where tested. The best policy was the dynamic one since it listens to changes in the voltage trace. In addition to that **FSMNVR_DB** used only one threshold reference while by using more of them, thus having more voltage states, an even better behaviour can be achieved.
The project still lacks a bit in code comments and needs a little bit of refactoring to make it more open source friendly.
Other improvements could be removing all Xilinx™proprietary IP's and use open source versions compatible on most FPGA's.
The project took approximately 5 months where most of the technical code was written in 2.