ECE 590: DIGITAL SYSTEMS DESIGN USING HARDWARE DESCRIPTION LANGUAGE

HOMEWORK-1

ARMSTRONG NUMBER

BHARATH REDDY GODI

SURENDRA MADDULA

NIKHIL MARDA

Table of contents:

1.0 INTRODUCTION

This report is consists of Homework 1 i.e. finding a given number is an Armstrong number or not using digital design and VHDL implementation.

The Implementation is done using FINITE STATE MACHINE DATAPATH. Armstrong number is also called as perfect number or a narcissistic number is a number that is the sum of its own digits each raised to the power of the number of digits.

The definition of a Armstrong number relies on the decimal representation $n = d_k d_{k-1} \ldots d_1$ of a natural number n, i.e.,

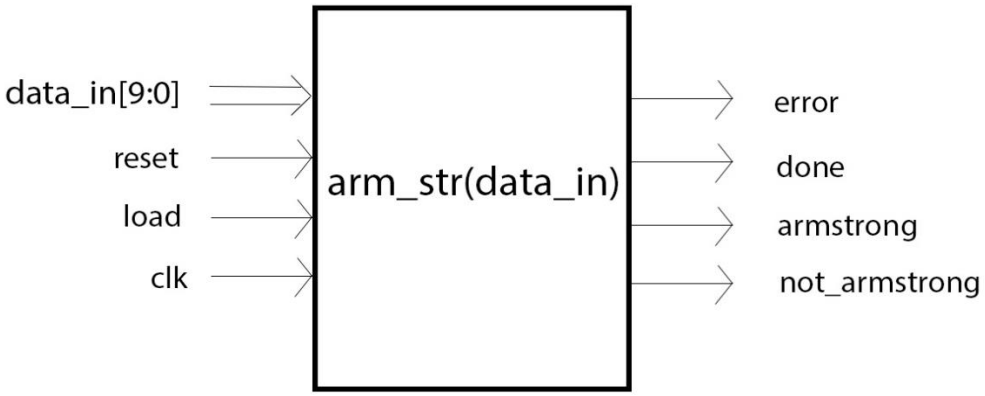
$n = d_k \cdot 10^{k-1} + d_{k-1} \cdot 10^{k-2} + \ldots + d_2 \cdot 10 + d_1$,

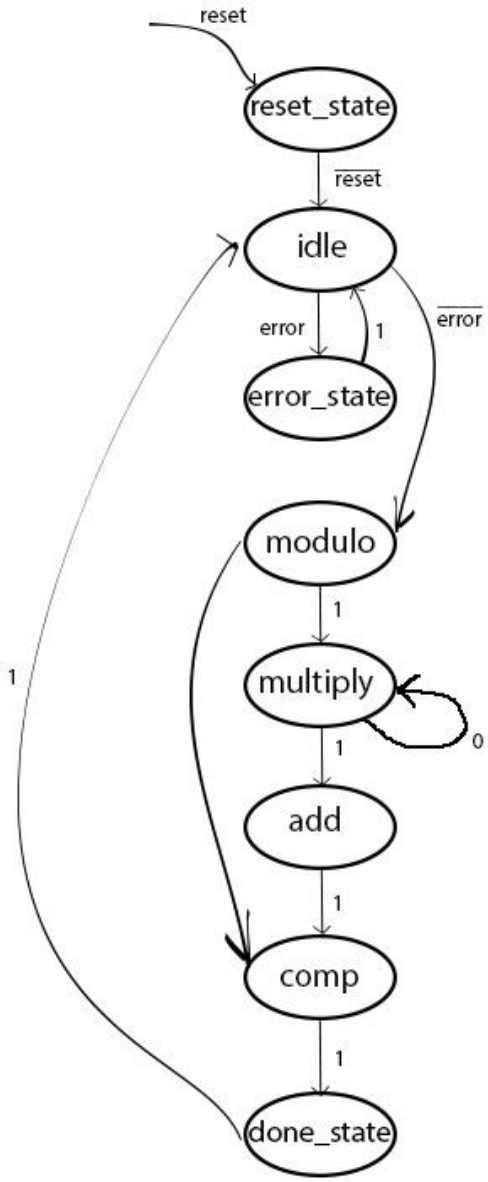
with k digits d_i satisfying $0 \leq d_i \leq 9$. Such a number n is called narcissistic if it satisfies the condition

$n = d_k^k + d_{k-1}^k + \ldots + d_2^k + d_1^k$.

For example the 3-digit decimal number 153 is a narcissistic number because $153 = 1^3 + 5^3 + 3^3$

2.0 ENTITY

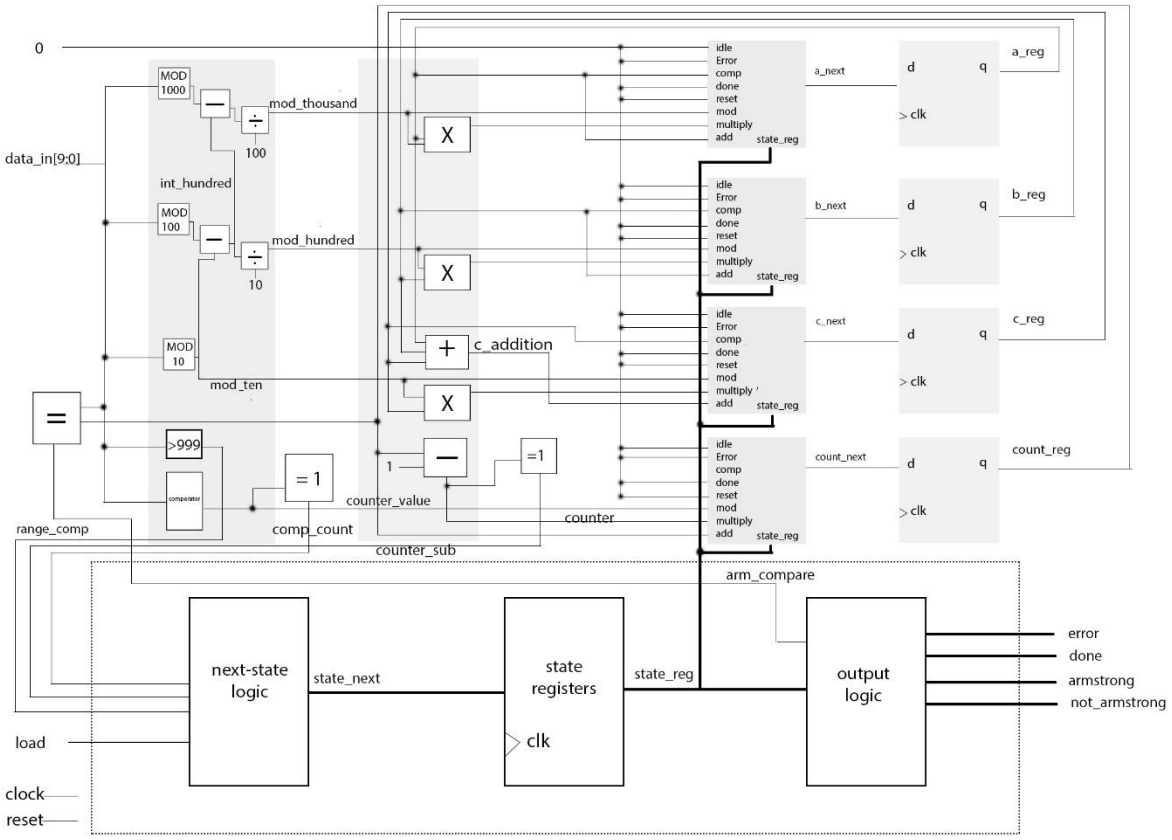




3.0 LOGIC DESCRIPTION

The circuit implements the following algorithm for the computing a number is ARMSTRONG number or not.

```
while (n1!=0)
{
   rem=n1%10;
   num+=rem*rem*rem;
   n1/=10;
}
if(num==n) // is an Armstrong number.
else // not an Armstrong number.
```



3.1 DATAPATH

The data path consists of 4 registers, 5 comparators, 5 Multiplexers. As the data enters the data path, we try to divide the number in to single digits and use registers to store those digits. Registers are also used in different states like modulus, multiply and addition and comparison states. Comparator have been used to check whether the number is 4 digit number or 3 digit or 2 digit number. Another comparator is used to check the counter value. Multiplexers are used to choose the FSM states and to load the count value.

3.2 CONTROL PATH

The control logic does eight things.

- *RESET:* The machine will be entering this state by default, unless it is set to zero.
- *IDLE:* In this it will wait for the load signal to be asserted. If it see it asserted it may go to error state or modulo state based on range_comp signal. If range comp is asserted it goes to error state else modulo state.
- *ERROR:* Here it asserts the error signal and moves back to idle state.
- *MODULO:* It checks the data input is a single digit number or not. If it is a single digit number it goes to comparison state or else it goes to multiply state.
- *MULTIPLY:* It does the cube of individual digits and stores in them in the registers. And it goes to next state i.e. addition.

- *ADD:* In this state, it does the addition of the cube of individual digits. And goes to comparison state.
- *COMP:* This is comparison state, which compares the result of the addition state with the data input that we received. And update the signal.
- *DONE:* This state asserts the done flag and indicates the operation is done. It also sets the registers to zero.

4.0 SUBCIRCUITS

4.1 MODULUS

This component is used to separate each digits from the given number and store them in the registers. The logic used for this purpose is shown below

```
--modulo
mod_ten <= unsigned(data_in) mod ten;
int_hundred <= (unsigned(data_in) mod hundred);
mod_hundred <= ((int_hundred - mod_ten)/ten);
mod_thousand <= ((unsigned(data_in) mod thousand) - int_hundred)/hundred;
counter_value <= one when unsigned(data_in) < ten else
```

4.2 MULTIPLICATION

This component performs the multiplication of individual digits to the power of number of digits in the given number. The code used for this is shown below

```
when multiply =>
        a_next <= a_mul(9 downto 0);
        b_next <= b_mul(9 downto 0);
        c_next <= c_mul(9 downto 0);
        count_next <= counter;

        a_mul <= a_reg * mod_thousand;
        b_mul <= b_reg * mod_hundred;
        c_mul <= c_reg * mod_ten;
```

4.3 ADDITION

This component perform the addition of individual multiplied digits together and stores in the register.

```
when add =>
        a_next <= a_reg;
        b_next <= b_reg;
        c_next <= c_addition(9 downto 0);
```

```
                count_next <= count_reg;
                c_addition <= a_reg + b_reg + c_reg;
```

4.4 COMPARISION

This component compares the result obtained by adding the registers in the addition state with the input value and asserts the armstrong or not_armstrong signals respectively.

```
                when comp =>
                a_next <= a_reg;
                b_next <= b_reg;
                c_next <= c_reg;
                count_next <= count_reg;
arm_compare <= '1' when unsigned(data_in) = c_reg else '0';
```

4.5 CONTROLLER

This component traverses through the states depending upon the inputs. The code for this is below

```
process(state_reg, load, range_comp, comp_count, counter_sub)
        begin
                case state_reg is
                        when idle =>
                                if (load = '1') then
                                        if (range_comp = '1') then      --
Compares whether the given input is above working range (>999)
                                                state_next <= error_state;
                                        else
                                                state_next <= modulo;
                                        end if;

                                else
                                        state_next <= idle;
                                end if;

                        when error_state =>
                                state_next <= idle;

                        when modulo =>
                                if (comp_count = '1') then
                                        state_next <= comp;
                                else
                                        state_next <= multiply;
                                end if;
```

```
                when multiply =>
                        if (counter_sub) = '1' then
                                state_next <= add;

                        else
                                state_next <= multiply;

                        end if;

                when add =>
                        state_next <= comp;

                when comp =>
                        state_next <= done_state;

                when done_state =>
                        state_next <= idle;
                when reset_state =>
                        state_next <= idle;

        end case;
end process;
```

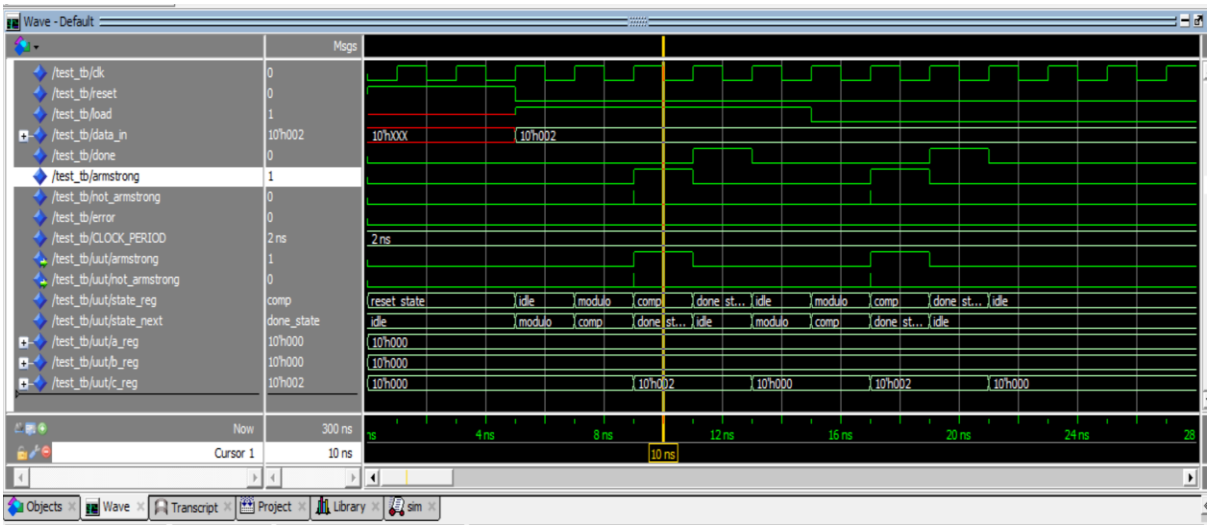
5.0 VALIDATION

Validation is done using the test bench. The below cases are covered in it.

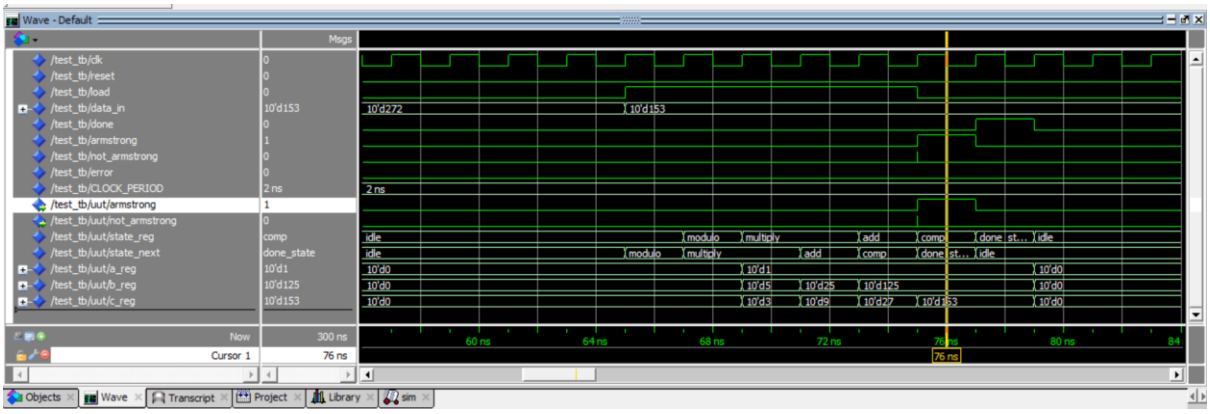
1. Validated the transition of FSM states.
2. Validated the data path with different possible inputs.
3. Tested each sub modules
4. Overall design and implementation is verified.

6.0 SIMULATION RESULTS

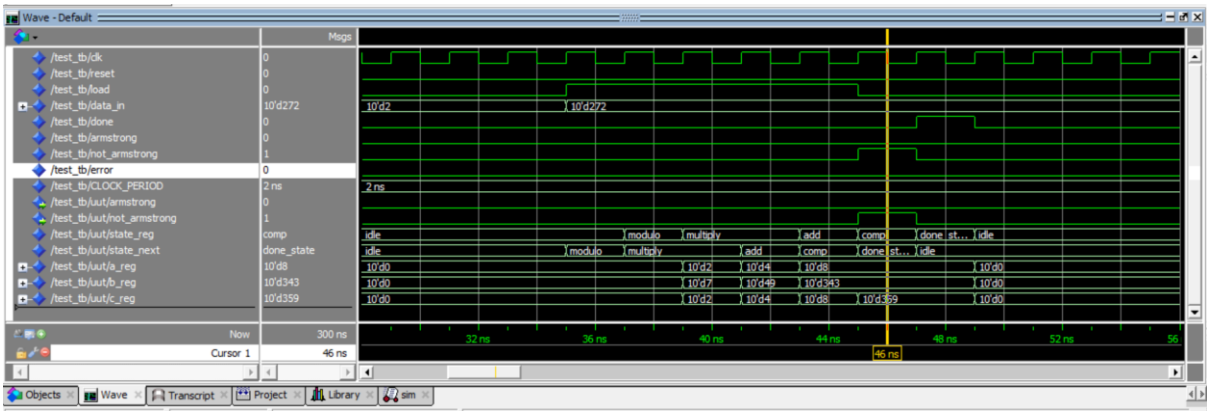
Input: 2



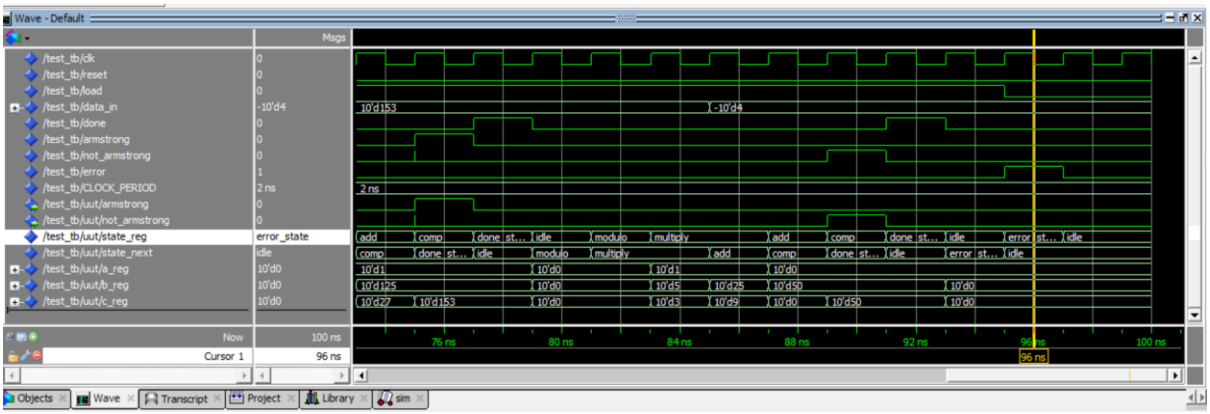
Input: 272



Input: 153



Input: 1020



7.0 CODE

```
--------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------
-- Filename:    armstrong.vhd
-- Created by:  Bharath Reddy Godi, Surendra Maddula, Nikhil Marda
-- Date:        Apr 19, 2016
-- ECE 590: Digital systems design using hardware description language (VHDL).
-- Assignment 1
-- This is an FSMD based Armstrong number generator, which takes inputs clk, reset, load, data_in
--  and gives output done when operation is finished and gives ouput whther given number is armstrong
--  number or not by asserting outputs armstrong or not_armstrong.
--------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity arm_str is port(
        clk, reset: in std_logic;
        load: in std_logic;
        data_in: in std_logic_vector(9 downto 0);
        done: out std_logic;
        armstrong: out std_logic;
        not_armstrong: out std_logic;
        error: out std_logic);
end arm_str;
```

```
architecture behavioral of arm_str is

	constant three: unsigned(2 downto 0) := "011";
	constant one: unsigned(2 downto 0) := "001";
	constant two: unsigned(2 downto 0) := "010";
	constant zero: unsigned(2 downto 0) := "000";
	constant range_value: integer:= 1000;
	constant ten: integer:= 10;
	constant hundred: integer:= 100;
	constant thousand: integer:= 1000;
	type state_type is (idle, error_state, modulo, multiply, add, comp, done_state, reset_state);
	signal state_reg, state_next: state_type;
	--signal a_is_0, b_is_0, count_0: std_logic;
	signal a_reg, a_next: unsigned(9 downto 0);
	signal b_reg, b_next: unsigned(9 downto 0);
	signal c_reg, c_next: unsigned(9 downto 0);
	signal c_addition: unsigned(9 downto 0);
	signal a_mul, b_mul, c_mul: unsigned(19 downto 0);
	signal count_reg, count_next, counter, counter_value: unsigned(2 downto 0);
	signal comp_count, arm_compare, counter_sub: std_logic;
	signal range_comp: std_logic;
	signal mod_ten, mod_hundred, mod_thousand,int_hundred: unsigned(9 downto 0);
	--signal adder_out: unsigned(2*WIDTH-1 downto 0);
	--signal sub_out: unsigned(WIDTH-1 downto 0);

begin

	-- control path: state register
	process(clk, reset)
	begin
		if (reset = '1') then
			state_reg <= reset_state;
		elsif (clk'event and clk = '1') then
			state_reg <= state_next;
		end if;
	end process;

	-- control path: next-state/output logic
	process(state_reg, load, range_comp, comp_count, counter_sub)
	begin
		case state_reg is
```

```
                        when idle =>
                                if (load = '1') then
                                        if (range_comp = '1') then      --      Compares
whether the given input is above working range (>999)
                                                state_next <= error_state;
                                        else
                                                state_next <= modulo;
                                        end if;

                                else
                                        state_next <= idle;
                                end if;

                        when error_state =>
                                state_next <= idle;

                        when modulo =>
                                if (comp_count = '1') then
                                        state_next <= comp;
                                else
                                        state_next <= multiply;
                                end if;

                        when multiply =>
                                if (counter_sub) = '1' then
                                        state_next <= add;
                                else
                                        state_next <= multiply;
                                end if;

                        when add =>
                                state_next <= comp;

                        when comp =>
                                state_next <= done_state;

                        when done_state =>
                                state_next <= idle;
                        when reset_state =>
                                state_next <= idle;
                end case;
        end process;

        -- control path: output logic
```

```
done <= '1'  when state_reg=done_state else '0';
armstrong <= arm_compare when state_reg=comp else '0';
not_armstrong <= not(arm_compare) when state_reg=comp else '0';
error <= '1' when state_reg=error_state else '0';

-- data path: data register
process(clk, reset)
begin
	if	(reset = '1') then
		a_reg <= (others=>'0');
		b_reg <= (others=>'0');
		c_reg <= (others=>'0');
		count_reg <= (others=>'0');
	elsif (clk'event and clk= '1') then
		a_reg <= a_next;
		b_reg <= b_next;
		c_reg <= c_next;
		count_reg <= count_next;
	end if;
end process;

-- data path: routing multiplexer

	process(state_reg, mod_ten, mod_hundred, mod_thousand, counter_value,
a_mul, b_mul, c_mul, counter, c_addition)
	begin
		case state_reg is
			when idle =>
				a_next <= (others=>'0');
				b_next <= (others=>'0');
				c_next <= (others=>'0');
				count_next <= (others=>'0');

			when error_state =>
				a_next <= (others=>'0');
				b_next <= (others=>'0');
				c_next <= (others=>'0');
				count_next <= (others=>'0');

			when modulo =>
				a_next <= mod_thousand;
				b_next <= mod_hundred;
				c_next <= mod_ten;
				count_next <= counter_value;
```

```
                when multiply =>
                        a_next <= a_mul(9 downto 0);
                        b_next <= b_mul(9 downto 0);
                        c_next <= c_mul(9 downto 0);
                        count_next <= counter;

                when add =>
                        a_next <= a_reg;
                        b_next <= b_reg;
                        c_next <= c_addition(9 downto 0);
                        count_next <= count_reg;

                when comp =>
                        a_next <= a_reg;
                        b_next <= b_reg;
                        c_next <= c_reg;
                        count_next <= count_reg;

                when done_state =>
                        a_next <= (others=>'0');
                        b_next <= (others=>'0');
                        c_next <= (others=>'0');
                        count_next <= (others=>'0');

                when reset_state =>
                        a_next <= (others=>'0');
                        b_next <= (others=>'0');
                        c_next <= (others=>'0');
                        count_next <= (others=>'0');

            end case;
        end process;

    range_comp <= '1' when unsigned(data_in) > "1111100111" else '0';

    --modulo
    mod_ten <= unsigned(data_in) mod ten;
    int_hundred <= (unsigned(data_in) mod hundred);
    mod_hundred <= ((int_hundred - mod_ten)/ten);
    mod_thousand <= ((unsigned(data_in) mod thousand) - int_hundred)/hundred;
    counter_value <= one when unsigned(data_in) < ten else
                     two when unsigned(data_in) < hundred else three;
    --
```

```
comp_count <= '1' when unsigned(data_in) < ten  else '0';
counter_sub <= '1' when counter = "001" else '0';
counter <= count_reg - "001";
a_mul <= a_reg * mod_thousand;
b_mul <= b_reg * mod_hundred;
c_mul <= c_reg * mod_ten;
c_addition <= a_reg + b_reg + c_reg;
arm_compare <= '1' when unsigned(data_in) = c_reg else '0';


end behavioral;
```

8.0 TESTBENCH

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;

ENTITY test_tb is
END test_tb;

ARCHITECTURE behavior OF test_tb IS

component arm_str port(
        clk, reset: in std_logic;
        load: in std_logic;
        data_in: in std_logic_vector(9 downto 0);
        done: out std_logic;
        armstrong: out std_logic;
        not_armstrong: out std_logic;
        error: out std_logic);
end component;

signal clk : std_logic;
signal reset: std_logic;
signal load: std_logic;
signal data_in : std_logic_vector(9 downto 0);
signal done, armstrong, not_armstrong, error: std_logic;
constant CLOCK_PERIOD : time := 2 ns;

BEGIN

uut: arm_str port map(
```

```
        clk => clk,
        reset => reset,
        load => load,
        data_in => data_in,
        done => done,
        armstrong => armstrong,
        not_armstrong => not_armstrong,
        error => error
);

clocked_process: process
begin
   clk <= '0';
   wait for CLOCK_PERIOD/2 ;
   clk <= '1';
   wait for CLOCK_PERIOD/2;
end process;

--stimulus Process
Stimuli: process
begin
        reset <= '1';
   wait for 5 ns;
        reset <= '0';
        load <=  '1';
        data_in <= "0000000010";
   wait for 10 ns;
        load <=  '0';

   wait for 20 ns;
        reset <= '0';
        load <=  '1';
        data_in <= "0100010000";
   wait for 10 ns;
        load <=  '0';

   wait for 20 ns;
        reset <= '0';
        load <=  '1';
        data_in <= "0010011001";
   wait for 20 ns;
        reset <= '0';
        load <=  '1';
        data_in <= "1111111100";
```

```
    wait for 10 ns;
        load <=  '0';
    wait for 100 ns;

end process;
end behavior;
```

9.0 CONTRIBUTION

Bharath Reddy Godi – Design & Control path

Surendra Maddula – Data path & Documentation

Nikhil Marda – Test bench & Documentation

10.0 REFERENCES

1. https://en.wikipedia.org/wiki/Narcissistic_number
2. http://web.cecs.pdx.edu/~mperkows/CLASS_VHDL/index.html