EDC1 Exam project 1
Design summary

Michał Szopiński

January 20, 2020

1 Design objectives

The goal of the project was to create a digital system calculating the equation:

$$y = \frac{\sum_{i=1}^{k}(4x_{2i}^2 - \frac{1}{4}x_{2i-1}^2)}{k} \tag{1}$$

where $k \in \{1, 2, 4, 8, 16, 32, 64, 128\}$. The numbers arrive in series over an 8-bit parallel bus and are coded using natural binary code. k arrives first, x_1, x_2, x_3, ... follow.

Handshaking was to be implemented using the following signals:

- `RDY` (ready) – raised by the system to request a number
- `NRDY` (number ready) – raised by the external circuit to signalize that the requested number is waiting at the input
- `RR` (result ready) – raised by the system to signalize that the output contains the calculated result, used simultaneously with `RDY`

Result accuracy and computational speed were critical factors of the design.

2 Operational subsystem design

2.1 Multiplication operands

A crucial observation to the realization of the circuit is that formula (1) can be rewritten as:

$$y = \frac{\sum_{i=1}^{k} (2x_{2i} + \frac{1}{2}x_{2i-1})(2x_{2i} - \frac{1}{2}x_{2i-1})}{k} \tag{2}$$

thus reducing the required amount of multiplication operations to one. Because calculating the operands of the multiplication requires both numbers to be available, x_{2i-1} must be buffered inside a register before x_{2i} is loaded.

Once both numbers are available, they can be supplied to two adders, where they can be added and subtracted. The results are then loaded as the operands of the multiplicator circuit.

2.2 Bus width considerations

In order to preserve accuracy when x_{2i-1} and x_{2i} are divided and multiplied by 2, the buses supplying them must be widened to carry at least 10 bits, nine of which integer and one fractional. In practice, a 16-bit bus is used. Because both numbers are positive, zeros are used as padding on on both sides.

To prevent integer overflow when summing up individual products, a sufficiently wide register must be chosen to store the result. A product of two 10-bit numbers can be up to 20 bits wide, therefore the output width was chosen to be the nearest power of 2, 32, which can store the sum of up to 2^{11} such numbers, positive or negative.

Even though the output of the adders is 16-bit, the shift registers storing the operands must match the width of the result register. This is because when the registers are shifted, the operands may become truncated, resulting in an incorrect sum.

The sum of products, before the final division, is a 32-bit number with 1 sign bit, 29 integer bits and 2 fractional bits, as a natural consequence of the multiplication of two 1-fractional-bit signed numbers.

2.3 Multiplicator circuit

The part of the operational unit responsible for the multiplication is a standard implementation of the "shift and add" algorithm. On each iteration, operand A is shifted to the left and operand B is shifted to the right. If the least significant bit of B is high, A is added to the sum. The multiplication ends when operand B is zero.

The result is cleared when k is loaded to the circuit.

2.4 Dynamic division and output format

One of the requirements of the project is the dynamic division of the final sum by a variable power of two. There are two commonly used methods of performing such division, each resulting in a different output format.

The first way is to split the output bus into an integer sub-bus an a fractional sub-bus, each containing a different amount of significant bits. By default, the output contains two significant fractional bits (2.2§4), but it may contain up to 9 significant fractional bits and as few as 23 significant integer bits (including the sign) after being divided by $2^7 = 128$.

The implementation of this method would require the use of two shift registers, one shifting into the other, and is linear in complexity. As such, it was concluded that it would be inefficient given the requirements of the design.

The second approach is to output the summation result as-is and specify the number of fractional bits separately for the external system to handle. The number of significant bits is given by:

$$n_{frac} = 2 + \log_2 k \tag{3}$$

Given the fact that k is restricted to a set of powers of two, the logarithm-calculating circuit can be constructed as a simple combinatorial circuit of three OR gates. Its output can then be fed into an adder with a constant value supplied as the second operand, and the number of fractional bits is thereby calculated with static complexity. Such an approach was used in the final design.

Because k only appears at the start of the number sequence, it must be buffered to be available at the end.

2.5 Processed values counter

Lastly, to keep track of the number of processed pairs (the number i), the circuit uses a bi-directional counter permanently configured to decrement its stored value. The counter is loaded simultaneously with the k buffer.

3 Control subsystem algorithm

3.1 I/O operations and handshaking

Considering the specification of the handshaking protocol, inputting numbers into the system can be thought of as "requesting" them from the external circuit, following a double-acknowledgment pattern. All input operations follow the general algorithm:

1. `RDY` is raised to request a new number from the external circuit.
2. `NRDY` is raised to signalize that the number is ready to be read.
3. `RDY` is lowered to request input withdrawal once it's been read.
4. `NRDY` is lowered, enabling further communication.

Output operations are simplified; the system raises `RR` whenever its output contains a valid result. No acknowledgment is taken from the external circuit.

3.2 Main program loop

The main loop starts with a special case of the input loop. It requests a number while simultaneously raising `RR` to signify that the result of the previous calculation is ready. Once the first number k becomes available, `RR` is lowered and the number is read into the memory, specifically the pair counter and the divisor buffer. The result register is also cleared.

When k is withdrawn, the program proceeds into the pair processing loop, after which the main loop reiterates.

3.3 Pair processing loop

To begin, the pair processing loop requests two numbers: x_{2i-1} and x_{2i}. When x_{2i-1} becomes available, it's loaded into a buffer. When x_{2i} becomes available,

the sum and the difference of the two numbers are fed into the multiplicator circuit for further processing.

As an optimization, the loading of x_{2i-1} is also used to decrement the processed pairs counter. A separate step would make the algorithm clearer, but also slower.

Before the multiplication begins, the algorithm checks if it has already finished to account for multiplication by 0. If that is the case, the multiplication step is skipped, otherwise, the program steps into a dedicated loop.

Following the multiplication, the processed values counter is taken into consideration. If there are no more pairs to be processed, the processing loop terminates, otherwise it reiterates.

3.4 Multiplication loop

The multiplication loop is a standard implementation of the "shift and add" algorithm. Addition and shifting occur on separate steps to avoid a race condition.

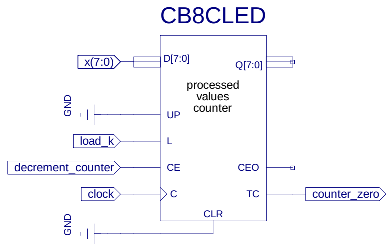
3.5 Algorithm reiteration

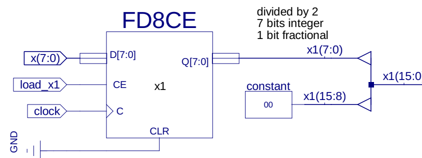
Once the processing loop terminates, the main loop reiterates. No further steps are required at this point. The result of the summation is stored in the memory and is available at the output. The divisor circuit is outputting the correct number of fractional bits thanks to the number k being stored in its memory. The `RR` flag can be safely raised while awaiting input.

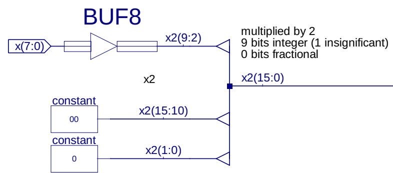
3.6 Two-phase clock

To avoid race conditions between the control and operational subsystems, a two-phase clock is used. This introduces a delay between the control signals being sent and executed, ensuring they are properly received before being processed. In practice, the clock is implemented by negating the clock signal in front of the operational subsystem.

Michal Szopinski
EDC exam problem 1
Data path



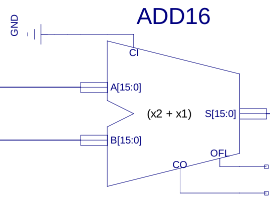


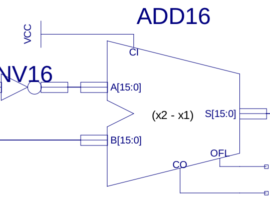


x1(15:0)

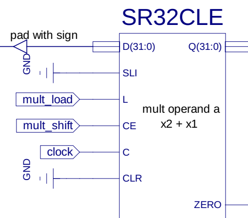
x2(15:0)

input to adders
15 bit int (incl. sign)
1 bit frac

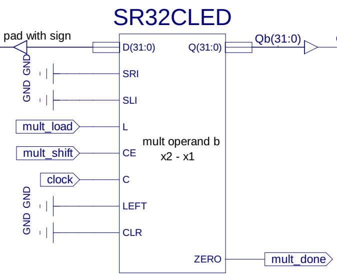




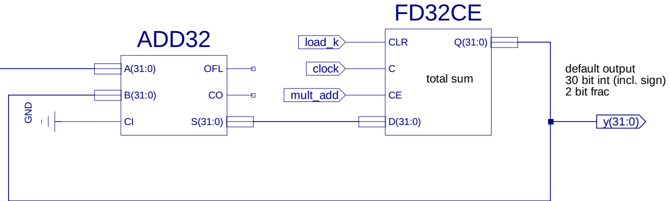
S1(15:0)
pad with sign



S2(15:0)
pad with sign

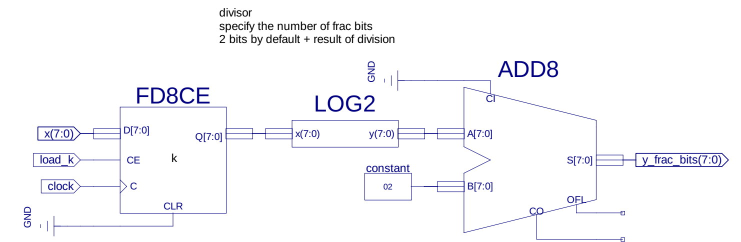


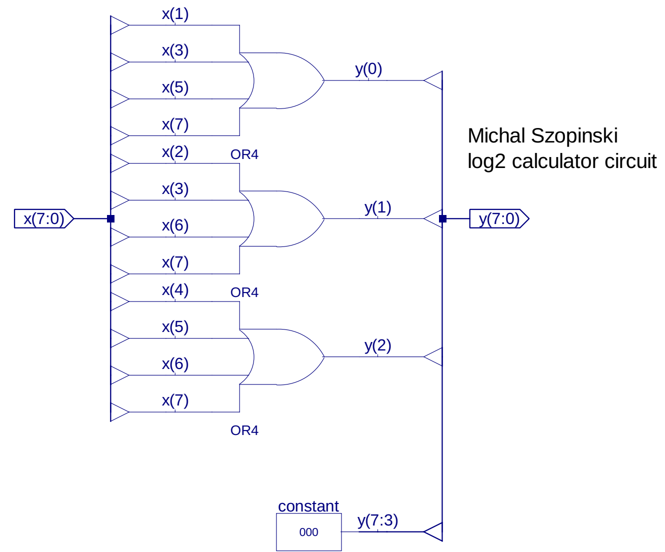
Qb(31:0)
Qb(0)
BUF
mult_lsb



default output
30 bit int (incl. sign)
2 bit frac
y(31:0)

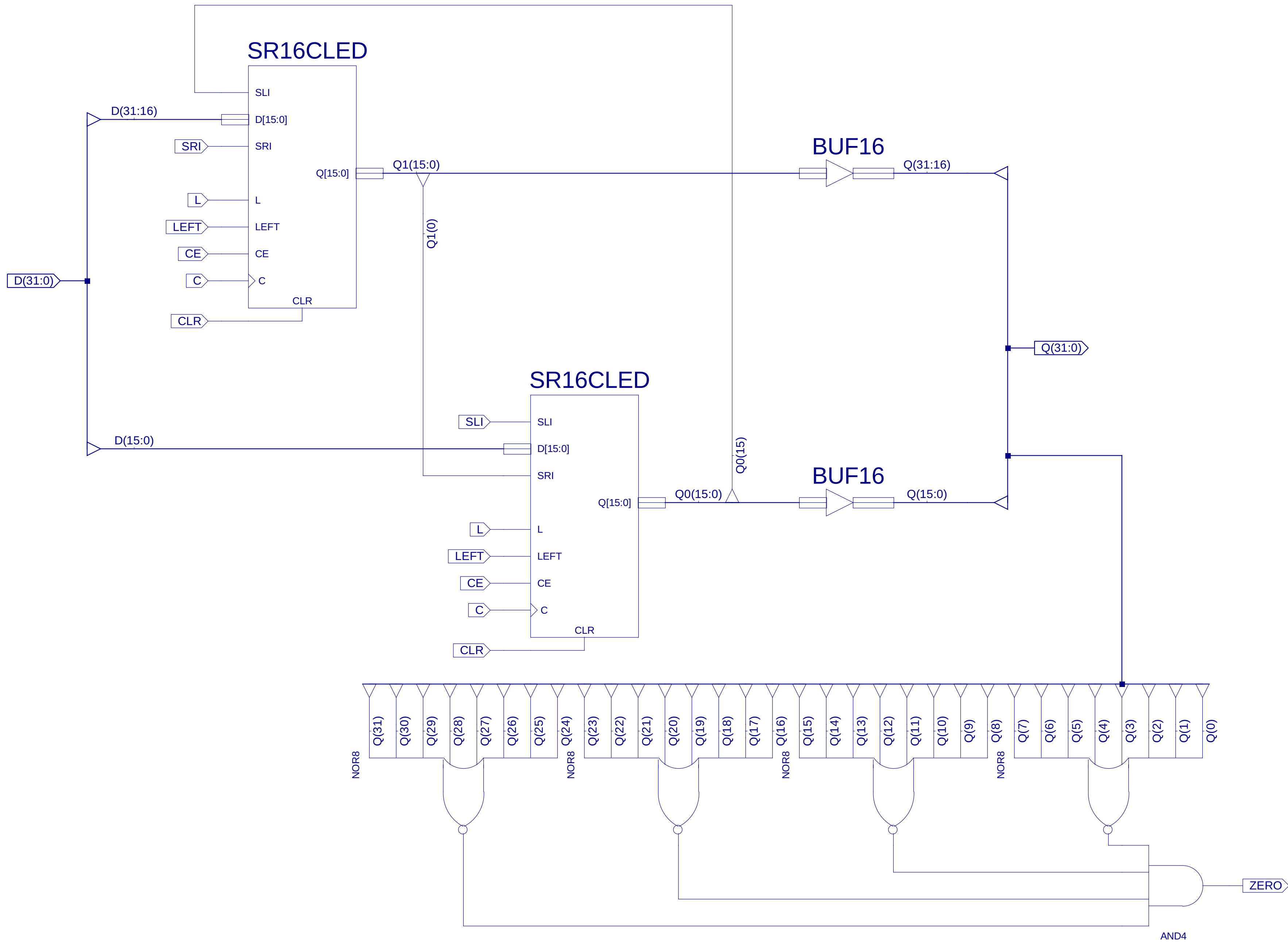
divisor
specify the number of frac bits
2 bits by default + result of division





Michal Szopinski
log2 calculator circuit

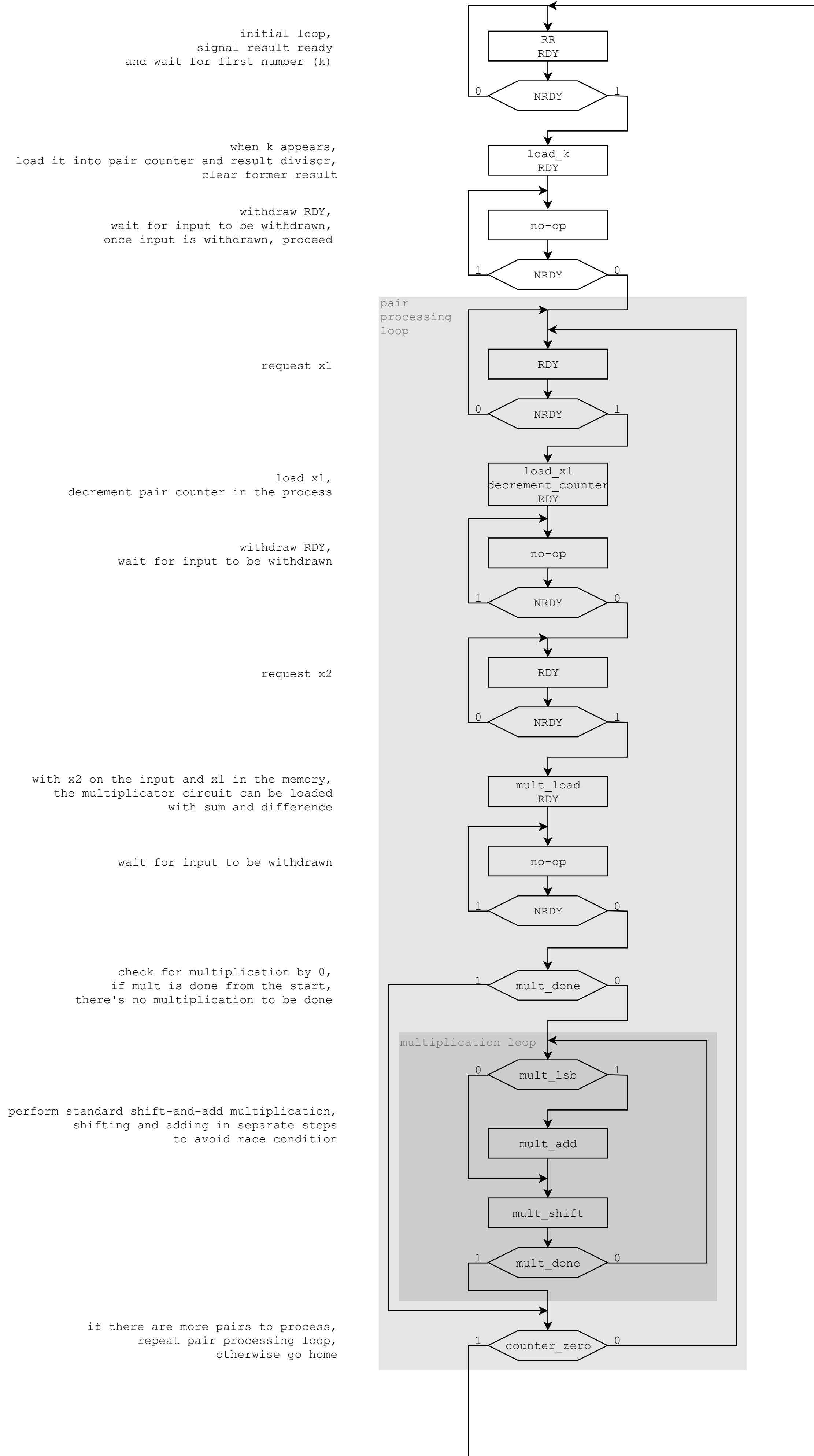
Michal Szopinski
32-bit async-clearable sync-loaded bidirectional shift register

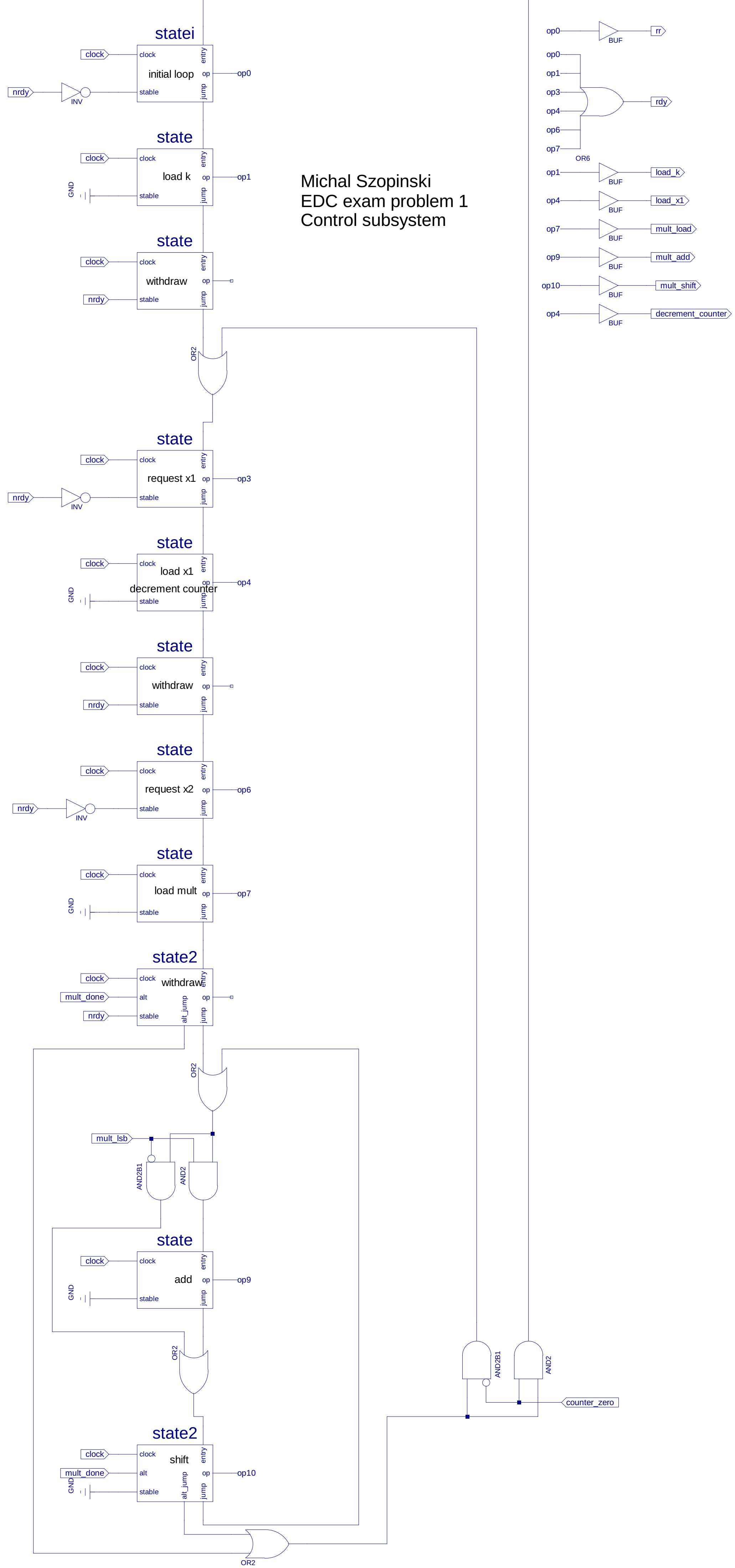


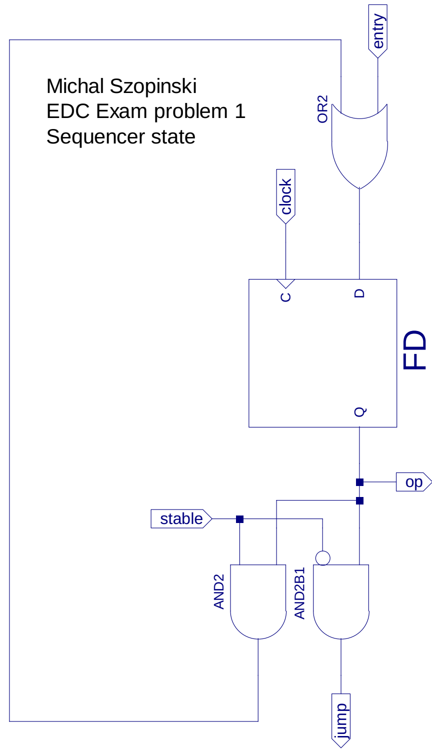
Michał Szopiński

Exam problem 1

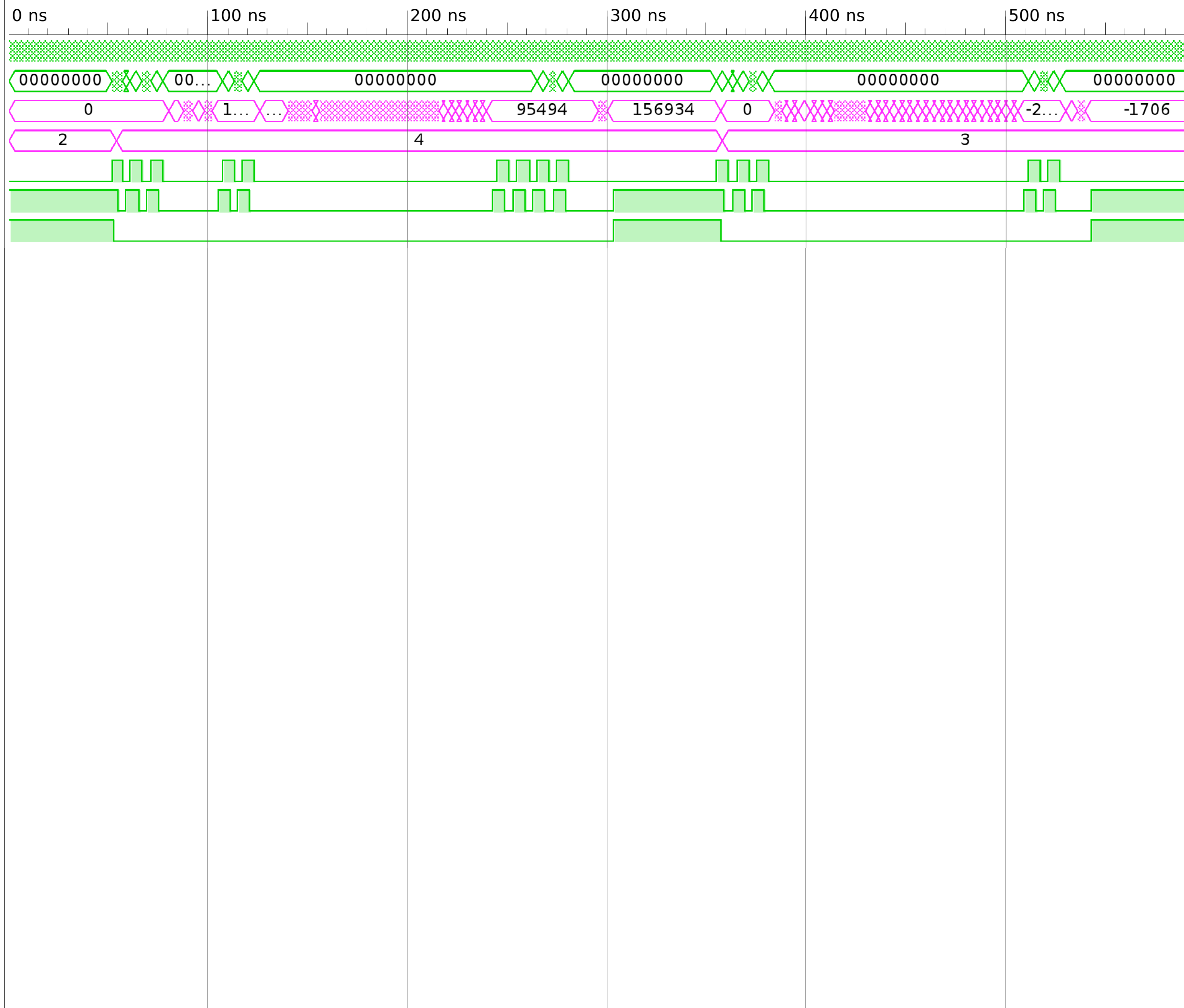
Flowchart




Michal Szopinski
EDC exam problem 1
Control subsystem
statei
initial loop
clock
nrdy
INV
stable
entry
op
jump
op0
state
load k
GND
op1
state
withdraw
OR2
request x1
op3
load x1
decrement counter
op4
withdraw
request x2
op6
load mult
op7
state2
withdraw
mult_done
alt
alt_jump
mult_lsb
AND2B1
AND2
add
op9
shift
op10
counter_zero
op0
BUF
rr
op1
op3
op4
op6
op7
OR6
rdy
load_k
load_x1
mult_load
mult_add
mult_shift
decrement_counter


Michal Szopinski
EDC Exam problem 1
Sequencer state
entry
OR2
clock
C
D
FD
Q
op
stable
AND2
AND2B1
jump

```vhdl
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.numeric_std.ALL;
4  LIBRARY UNISIM;
5  USE UNISIM.Vcomponents.ALL;
6  ENTITY system_system_sch_tb IS
7  END system_system_sch_tb;
8  ARCHITECTURE behavioral OF system_system_sch_tb IS
9
10    COMPONENT system
11    PORT( clock  :   IN  STD_LOGIC;
12           x :   IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
13           y :   OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
14           y_frac_bits   :   OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
15           nrdy  :   IN  STD_LOGIC;
16           rdy   :   OUT STD_LOGIC;
17           rr    :   OUT STD_LOGIC);
18    END COMPONENT;
19
20    SIGNAL clock :   STD_LOGIC := '0';
21    SIGNAL x :   STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
22    SIGNAL y :   STD_LOGIC_VECTOR (31 DOWNTO 0);
23    SIGNAL y_frac_bits   :   STD_LOGIC_VECTOR (7 DOWNTO 0);
24    SIGNAL nrdy  :   STD_LOGIC := '0';
25    SIGNAL rdy   :   STD_LOGIC;
26    SIGNAL rr    :   STD_LOGIC;
27
28 BEGIN
29
30    UUT: system PORT MAP(
31         clock => clock,
32         x => x,
33         y => y,
34         y_frac_bits => y_frac_bits,
35         nrdy => nrdy,
36         rdy => rdy,
37         rr => rr
38    );
39
40     clock <= not clock after 1 ns;
41
42 -- *** Test Bench - User Defined Section ***
43    tb : PROCESS
44
45     procedure Feed_number(
46         constant num : in integer) is
47     begin
48         if rdy = '0' then wait until rdy = '1'; end if;
49         wait for 2 ns; -- comment out wait for faster operation
50
51         x <= std_logic_vector(to_unsigned(num, 8));
52         nrdy <= '1';
53
54         if rdy = '1' then wait until rdy = '0'; end if;
55         wait for 2 ns;
56
57         x <= "00000000";
58         nrdy <= '0';
59     end procedure;
60
61    BEGIN
62       -- feed k
63         wait for 50 ns;
64         Feed_number(4);
65
66         -- feed 4 pairs
67         Feed_number(123);
68         Feed_number(85);
69         Feed_number(71);
70         Feed_number(2);
71         Feed_number(0);
72         Feed_number(0);
73         Feed_number(64);
74         Feed_number(64);
75         -- expected result: 9808.375 = 156934 / (2^4)
76
77         -- feed k
78         wait until rr = '1';
79         wait for 50 ns;
80         Feed_number(2);
81
82         -- feed 2 pairs
83         Feed_number(69);
84         Feed_number(13);
85         Feed_number(7);
86         Feed_number(5);
87         -- expected result: -213.25 = -1706 / (2^3)
88
89         WAIT; -- will wait forever
90    END PROCESS;
91 -- *** End Test Bench - User Defined Section ***
92
93 END;
94
```



0 ns
100 ns
200 ns
300 ns
400 ns
500 ns
clock
x[7:0]
y[31:0]
y_frac_bits[7:0]
nrdy
rdy
rr
00000000
00...
00000000
00000000
00000000
00000000
0
1...
...
95494
156934
0
-2...
-1706
2
4
3