Part 1: Design of a (parameterizable) ALU

The op-code table, and definition for each flag:

CATEGORY	FUNCTION	OPCODE
Identity:	ALU_OUT <= A	0000
Bit-wise logic:	ALU_OUT <= A AND B	0100
	ALU_OUT <= A OR B	0101
	ALU_OUT <= A XOR B	0110
	ALU_OUT <= NOT A	0111
Arithmetic:	ALU_OUT <= A + 1	1000
	ALU_OUT <= A - 1	1001
	ALU_OUT <= A + B	1010
	ALU_OUT <= A - B	1011
Shift:	ALU_OUT <= Arithmetic shift left A by X bits	1100
	ALU_OUT <= Arithmetic shift right A by X bits	1101
	ALU_OUT <= Rotate left A by X bits	1110
	ALU_OUT <= Rotate right A by X bits	1111

Flags(0): OUT = 0
Flags(1): OUT $\neq$ 0
Flags(2): OUT = 1
Flags(3): OUT < 0
Flags(4): OUT > 0
Flags(5): OUT $\leq$ 0
Flags(6): OUT $\geq$ 0
Flags(7): Overflow

VHDL Code for ALU:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.DigEng.ALL;
use IEEE.NUMERIC_STD.ALL;


-- This module implements the Arithmetic logic unit of the CPU
-- The module is only consisted of combinational logic units
entity ALU is
    generic(data_size : natural := 8);
    Port ( A : in STD_LOGIC_VECTOR (data_size-1 downto 0); -- Input data
           B : in STD_LOGIC_VECTOR (data_size-1 downto 0); -- Input data
           -- This signal determines For How many positions we are shifting
           X : in STD_LOGIC_VECTOR (log2(data_size)-1 downto 0);
           Opcode : in STD_LOGIC_VECTOR(3 downto 0);       -- This control signal determines
                                                           -- which operation is done by ALU
           flags  : out STD_LOGIC_VECTOR(7 downto 0);      -- The output which shows the characteristics
                                                           -- of the output of ALU operation
           ALU_out : out STD_LOGIC_VECTOR (data_size-1 downto 0));  -- output parallel data
end ALU;

architecture Behavioral of ALU is

signal ALU_out_int : SIGNED (data_size-1 downto 0);

begin
-- The data output of the ALU is assigned its respective value based on the opcode
-- It’s implemented using a mux and other combinational gates
```

```
    with opcode select
    ALU_out_int <=  signed(A)                                        when "0000",
                    signed(A and B)                                  when "0100",
                    signed(A or B)                                   when "0101",
                    signed(A xor B)                                  when "0110",
                    signed(not A)                                    when "0111",
                    signed(A) + 1                                    when "1000",
                    signed(A) - 1                                    when "1001",
                    signed(A) + signed(B)                            when "1010",
                    signed(A) - signed(B)                            when "1011",
                    shift_left(signed(A),to_integer(unsigned(X)))    when "1100",
                    shift_right(signed(A),to_integer(unsigned(X)))   when "1101",
                    rotate_left(signed(A),to_integer(unsigned(X)))   when "1110",
                    rotate_right(signed(A),to_integer(unsigned(X))) when "1111",
                    (others => '0')                                  when others;

-- The flags are assigned values based on output of ALU
-- and opcode
-- The overflow condition for addition only happens when inputs have the same sign but the output has a
-- different sign
-- In signed data the MSB indicates the sign of our number
-- For subtraction operation the principle is the same but it should be considered that the sign of second
-- operand is negated
-- In Increment by 1 operation, we can only have an overflow for positive numbers
-- In decrement by 1 operation, we can only have an overflow for negative numbers
    flags(0) <= '1'   when ALU_out_int = 0  else '0';
    flags(1) <= '1'   when ALU_out_int /= 0 else '0';
    flags(2) <= '1'   when ALU_out_int = 1  else '0';
    flags(3) <= '1'   when ALU_out_int < 0  else '0';
    flags(4) <= '1'   when ALU_out_int > 0  else '0';
    flags(5) <= '1'   when ALU_out_int <= 0 else '0';
    flags(6) <= '1'   when ALU_out_int >= 0 else '0';
```

```
flags(7) <= '1'   when (((opcode =  "1010") and (A(data_size-1) = '0') and (B(data_size-1) = '0')  and
                      (ALU_out_int(data_size-1) = '1')) or
                      (( opcode = "1010") and A(data_size-1) = '1' and B(data_size-1) = '1'  and
                      ALU_out_int(data_size-1) = '0') or
                      (( opcode = "1011") and A(data_size-1) = '0' and B(data_size-1) = '1'  and
                      ALU_out_int(data_size-1) = '1') or
                      (( opcode = "1011") and A(data_size-1) = '1' and B(data_size-1) = '0'  and
                      ALU_out_int(data_size-1) = '0') or
                      (( opcode = "1000") and A(data_size-1) = '0' and ALU_out_int(data_size-1) = '1')
                      or
                      (( opcode = "1001") and A(data_size-1) = '1' and ALU_out_int(data_size-1) = '0'))
                      else
                      '0';

ALU_out <= std_logic_vector(ALU_out_int);
end Behavioral;
```

VHDL code for the testbench:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.DigEng.ALL;
use IEEE.NUMERIC_STD.ALL;


entity ALU_tb is

end ALU_tb;

architecture Behavioral of ALU_tb is

constant data_size : natural := 16;

signal A : STD_LOGIC_VECTOR (data_size-1 downto 0);
signal B : STD_LOGIC_VECTOR (data_size-1 downto 0);
signal X : STD_LOGIC_VECTOR (log2(data_size)-1 downto 0);
signal Opcode :  STD_LOGIC_VECTOR(3 downto 0);
signal flags  :  STD_LOGIC_VECTOR(7 downto 0);
signal ALU_out : STD_LOGIC_VECTOR (data_size-1 downto 0);
```

```
type test_vector is record
        A : STD_LOGIC_VECTOR (data_size-1 downto 0);
        B : STD_LOGIC_VECTOR (data_size-1 downto 0);
        X : STD_LOGIC_VECTOR (log2(data_size)-1 downto 0);
        Opcode :  STD_LOGIC_VECTOR(3 downto 0);
        flags  :  STD_LOGIC_VECTOR(7 downto 0);
        ALU_out : STD_LOGIC_VECTOR (data_size-1 downto 0);
end record;

-- For the purpose of testing this module, for each opcode
-- a set of tests were designed, For the first function only 2 different numbers
-- were given as inputs, for AND function the test was designed in a way that
-- every entry in the logic table for AND logic was tested by bitwise operation
-- For example in every nibble of data all 4 rows in the logic table was tested
-- (a more comprehensive test would do the same for every bit insteaad of nibble)
-- The same method was used also done for OR, XOR and NOT, The specifics about each
-- test and what it tests is written in front of it in the vector table
type test_vector_array is array
        (natural range <>) of test_vector;
constant test_vectors : test_vector_array := (
        --A,       B,      X,        Opcode, flags, ALU_out
        -- #0 This test also verifies the flag bits for "non-zero", "less than zero", " less than equal to
        -- zero"
        (x"ABAB", x"0000", "0000",  "0000", x"2A", x"ABAB"),
        (x"3434", x"0000", "0000",  "0000", x"52", x"3434"), -- #1 The new tested flags are "greater than
                                                             -- zero", " greater or equal to zero"
        (x"3A3A", x"3535", "0000",  "0100", x"52", x"3030"), -- #2 Testing the AND function with
                                                             -- aforementioned method
        (x"5A5A", x"3C3C", "0000",  "0101", x"52", x"7E7E"), -- #3 Testing the OR function with
                                                             -- aforementioned method
        (x"AAAA", x"3C3C", "0000",  "0110", x"2A", x"9696"), -- #4 Testing the XOR function with
                                                             -- aforementioned method
        (x"0000", x"0000", "0000",  "0111", x"2A", x"FFFF"), -- #5 Testing if the first entry of logic
                                                             -- table for NOT works for all bits
        -- #6 Testing if the second entry of logic table for NOT works for all bits This also tests the
        -- "equal to zero" flag
        (x"FFFF", x"0000", "0000",  "0111", x"61", x"0000"),
        (x"5A5A", x"0000", "0000",  "0111", x"2A", x"A5A5"), -- #7 Testing a random number for NOT
        (x"0000", x"0000", "0000",  "1000", x"56", x"0001"), -- #8 This also tests the "equal to one" flag
```

```
(x"FF9B", x"0000", "0000",  "1000", x"2A", x"FF9C"), -- #9 testing if a carry can ripple through
                                                     -- the bits
(x"7FFF", x"0000", "0000",  "1000", x"AA", x"8000"), -- #10 testing the OV flag for incrementing
                                                     -- positive numbers
(x"FFFF", x"0000", "0000",  "1000", x"61", x"0000"), -- #11 making sure that ov flag doesn't get
                                                     -- enabled by error
                                                     -- at transition from negative to zero
(x"8000", x"0000", "0000",  "1001", x"D2", x"7FFF"), -- #12 testing the OV flag for decrement by 1
(x"FFFF", x"0000", "0000",  "1001", x"2A", x"FFFE"), -- #13 testing just a random number for
                                                     -- decrement by 1
(x"7000", x"1000", "0000",  "1010", x"AA", x"8000"), -- #14 testing the overflow for addition
                                                     -- function (Both Positive)
(x"1f10", x"5310", "0000",  "1010", x"52", x"7220"), -- #15 testing a random number
(x"8000", x"8000", "0000",  "1010", x"E1", x"0000"), -- #16 testing the overflow for addition
                                                     -- function (Both negative)
(x"1100", x"F800", "0000",  "1010", x"52", x"0900"), -- #17 adding one negative and one positive
                                                     -- number
(x"4000", x"C000", "0000",  "1011", x"AA", x"8000"), -- #18 checking the OV flag for subtracting a
                                                     -- negative from a positive number
(x"8120", x"7EE0", "0000",  "1011", x"D2", x"0240"), -- #19 checking the OV flag for subtracting a
                                                     -- positive from a negative number
(x"0151", x"0023", "0000",  "1011", x"52", x"012E"), -- #20 checking the functionality for
                                                     -- subtracting a positive from a positive
(x"8012", x"FFF8", "0000",  "1011", x"2A", x"801A"), -- #21 checking the functionality for
                                                     -- subtracting a negative from a negative
(x"A54F", x"0000", "0001",  "1100", x"52", x"4A9E"), -- #22 checking the functionality for shift
                                                     -- left with one position
(x"3D5B", x"0000", "0101",  "1100", x"2A", x"AB60"), -- #23 checking the functionality for shift
                                                     -- left with five positions
(x"A4D5", x"0000", "0001",  "1101", x"2A", x"D26A"), -- #24 checking the functionality for shift
                                                     -- right by one position for a negative number
(x"56B5", x"0000", "0011",  "1101", x"52", x"0AD6"), -- #25 checking the functionality for shift
                                                     -- right by three positions for a positive No
(x"A54F", x"0000", "0001",  "1110", x"52", x"4A9F"), -- #26 checking the functionality for rotate
                                                     -- left with one position
(x"3D5B", x"0000", "0100",  "1110", x"2A", x"D5B3"), -- #27 checking the functionality for rotate
                                                     -- left with five positions
(x"A4D5", x"0000", "0001",  "1111", x"2A", x"D26A"), -- #28 checking the functionality for rotate
                                                     -- right by one position for a negative number
(x"56B5", x"0000", "0011",  "1111", x"2A", x"AAD6"));-- #29 checking the functionality for rotate
                                                     -- right by three positions for a positive No
```

```
begin
UUT : entity work.ALU
   Generic map(data_size => data_size)
   port map(
        A => A,
        B => B,
        X => X,
        opcode => opcode,
        flags => flags,
        ALU_out => ALU_out );


test_process : process
begin
    -- wait 100 ns for global reset to finish
    wait for 100ns;

    for i in test_vectors' range loop
      A <= test_vectors(i).a;
      B <= test_vectors(i).b;
      X <= test_vectors(i).x;
      opcode <= test_vectors(i).opcode;
      wait for 20 ns;
      assert ((flags = test_vectors(i).flags) and
              (ALU_out = test_vectors(i).ALU_out))
      report "Test vector " &
             integer'image(i) &
             " failed for inputs a = " &
             integer'image(to_integer(signed(a))) &
             " and b = " &
             integer'image(to_integer(signed(b))) &
             " and x = " &
             integer'image(to_integer(unsigned(x))) &
             " and opcode = " &
             integer'image(to_integer(unsigned(opcode))) &
             ". Expected flags = " &
             integer'image(to_integer(unsigned(test_vectors(i).flags))) &
             " and ALU output =" &
             integer'image(to_integer(signed(test_vectors(i).ALU_out))) &
             "; observed  flags = " &
             integer'image(to_integer(unsigned(flags))) &
```

```
            " and ALU output =" &
            integer'image(to_integer(signed(ALU_out)))
      severity failure;       -- to stop the simulation

      report "The output corresponds to expectation at test vector " &
      integer'image(i)
      severity note;

    end loop;
    wait;

end process;

end Behavioral;
```

Simulation screenshots from console:

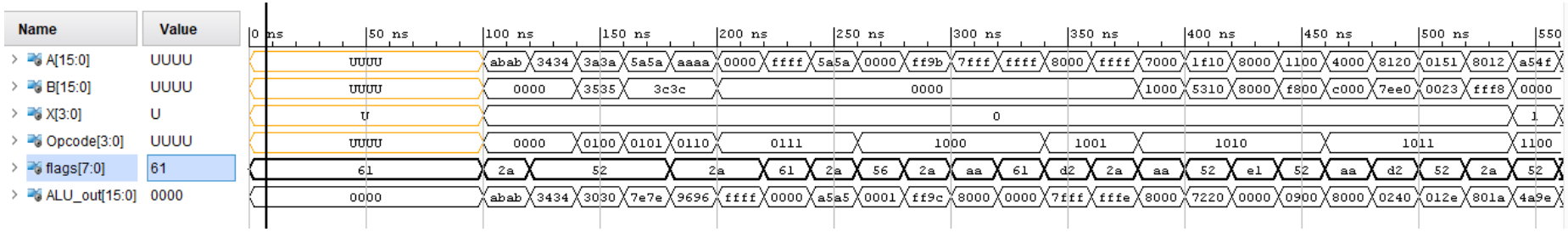
Tcl Console

```
Time resolution is 1 ps
source ALU_tb.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave
#   }
# }
# run 1000ns
Warning: NUMERIC_STD."<=": metavalue detected, returning FALSE
Time: 0 ps  Iteration: 0  Process: /ALU_tb/UUT/line__59  File: E:/York/Digital_Design/VivadoProjects/AI
Warning: NUMERIC_STD."=": metavalue detected, returning FALSE
Time: 0 ps  Iteration: 0  Process: /ALU_tb/UUT/line__56  File: E:/York/Digital_Design/VivadoProjects/AI
Warning: NUMERIC_STD."=": metavalue detected, returning FALSE
Time: 0 ps  Iteration: 0  Process: /ALU_tb/UUT/line__54  File: E:/York/Digital_Design/VivadoProjects/AI
Note: The output corresponds to expectation at test vector 0
Time: 120 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/
Note: The output corresponds to expectation at test vector 1
Time: 140 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/
Note: The output corresponds to expectation at test vector 2
Time: 160 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/
Note: The output corresponds to expectation at test vector 3
Time: 180 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/
Note: The output corresponds to expectation at test vector 4
Time: 200 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/
Note: The output corresponds to expectation at test vector 5
Time: 220 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/
Note: The output corresponds to expectation at test vector 6
Time: 240 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/
Note: The output corresponds to expectation at test vector 7
Time: 260 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/
Note: The output corresponds to expectation at test vector 8
```

Tcl Console

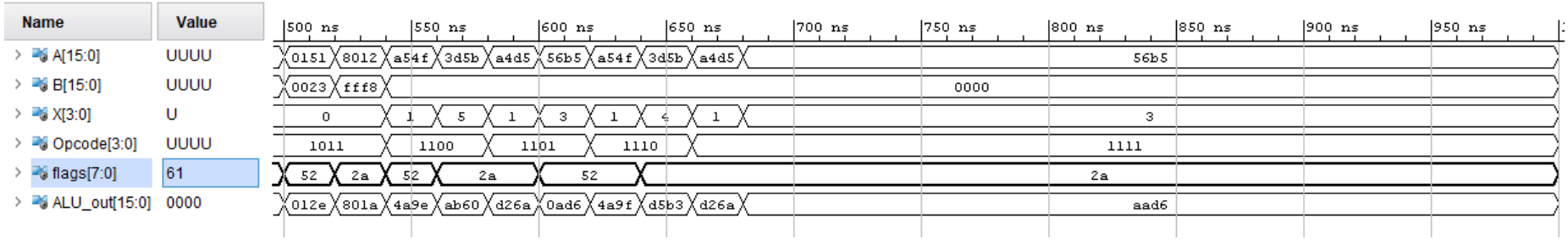
```
Time: 280 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU
Note: The output corresponds to expectation at test vector 9
Time: 300 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU
Note: The output corresponds to expectation at test vector 10
Time: 320 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU
Note: The output corresponds to expectation at test vector 11
Time: 340 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU
Note: The output corresponds to expectation at test vector 12
Time: 360 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU
Note: The output corresponds to expectation at test vector 13
Time: 380 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU
Note: The output corresponds to expectation at test vector 14
Time: 400 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU
Note: The output corresponds to expectation at test vector 15
Time: 420 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU
Note: The output corresponds to expectation at test vector 16
Time: 440 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU
Note: The output corresponds to expectation at test vector 17
Time: 460 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU
Note: The output corresponds to expectation at test vector 18
Time: 480 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU
Note: The output corresponds to expectation at test vector 19
Time: 500 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU
Note: The output corresponds to expectation at test vector 20
Time: 520 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU
Note: The output corresponds to expectation at test vector 21
Time: 540 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU
Note: The output corresponds to expectation at test vector 22
Time: 560 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU
Note: The output corresponds to expectation at test vector 23
Time: 580 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU
Note: The output corresponds to expectation at test vector 24
Time: 600 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU
Note: The output corresponds to expectation at test vector 25
Time: 620 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU
Note: The output corresponds to expectation at test vector 26
Time: 640 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU_standalone/ALU
Note: The output corresponds to expectation at test vector 27
Time: 660 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU_standalone/ALU
Note: The output corresponds to expectation at test vector 28
Time: 680 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU_standalone/ALU
Note: The output corresponds to expectation at test vector 29
Time: 700 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU_standalone/ALU
xsim: Time (s): cpu = 00:00:04 ; elapsed = 00:00:08 . Memory (MB): peak = 717.609 ; gain = 7.789
INFO: [USF-XSim-96] XSim completed. Design snapshot 'ALU_tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:05 ; elapsed = 00:00:45 . Memory (MB): peak = 717.609 ; gain = 7.789
```

Graphical Simulation screenshots:



First opcode is the identity function and the next 4 opcodes are AND, OR, XOR and NOT functions, respectively.

Starting from 1000, the next 4 functions constitute the Arithmetic functions, respectively: increment by 1, decrement by 1, addition and subtraction.



Starting from 1100, the next 4 functions constitute the Shift functions, respectively: arithmetic shift left, arithmetic shift right, rotate left and rotate right.

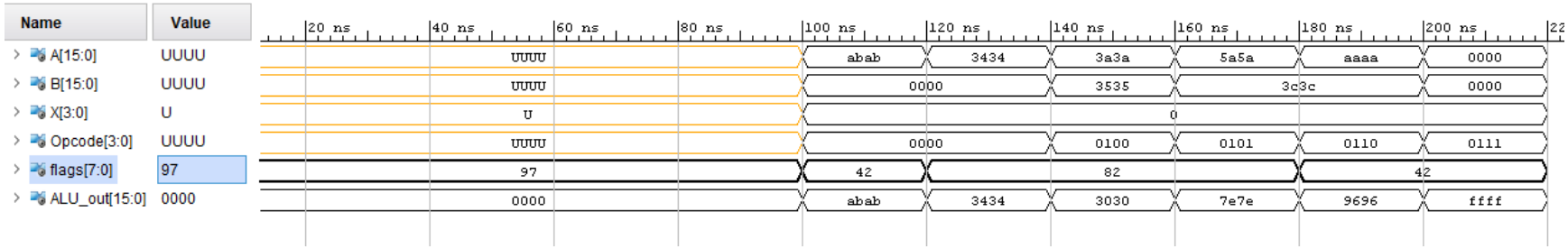
Although it could have been more useful to use signed representation for some arithmetic operations, For the purpose of compactness , hex representation was used for input and output data. From the self-checking testbench and the results that we got from the console we can be more confident about correct function of the module.

Simulation result with deliberately introducing an erroneous dataset at test vector 5:

(To verify the assert clause)

Tcl Console

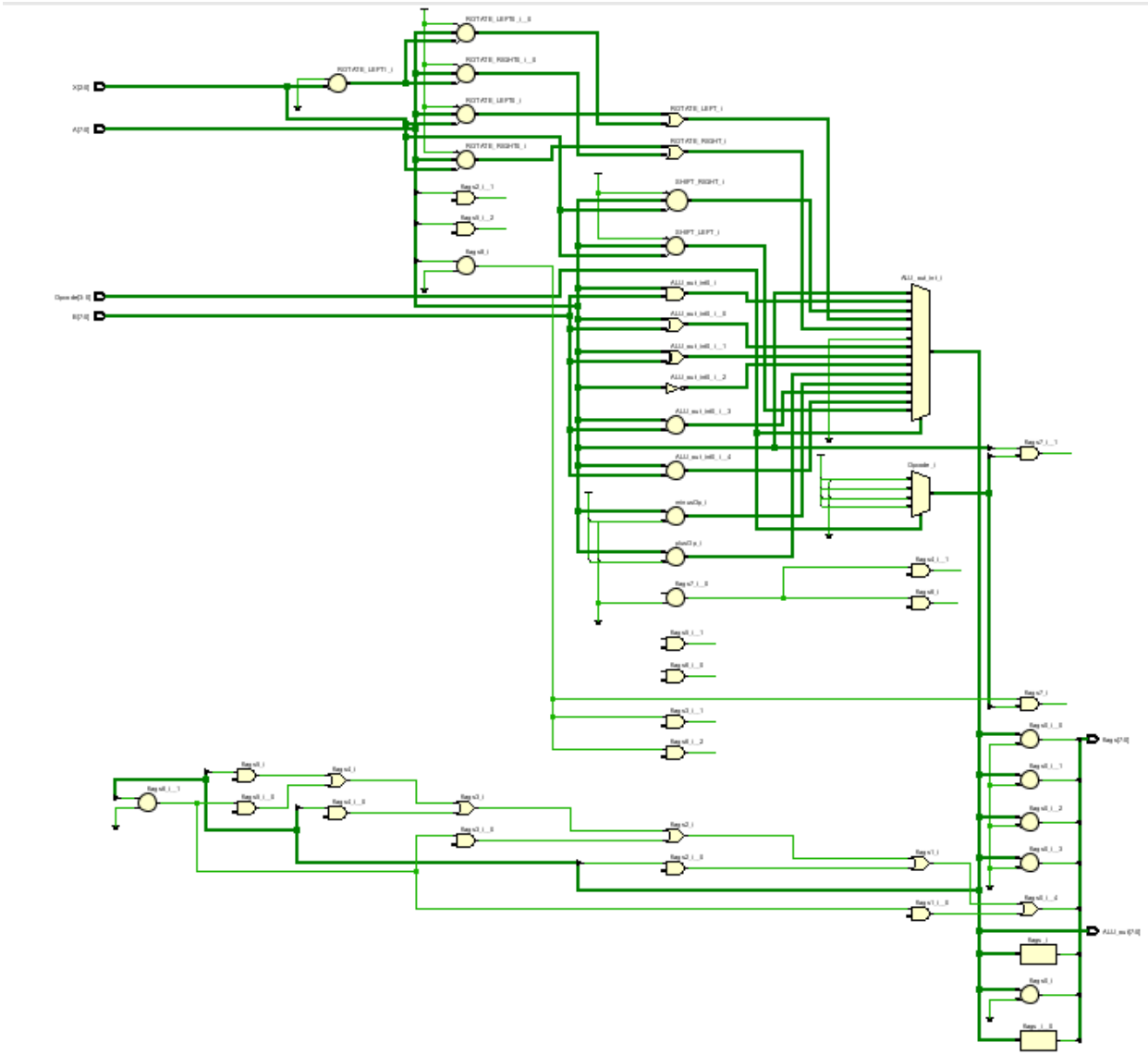
```
Time resolution is 1 ps
source ALU_tb.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window. If you want to open a wave window go to 'File->New Wa
#   }
# }
# run 1000ns
Warning: NUMERIC_STD."<=": metavalue detected, returning FALSE
Time: 0 ps  Iteration: 0  Process: /ALU_tb/UUT/line__59  File: E:/York/Digital_Design/VivadoProjects/ALU_standalone/ALU_standalone.srcs/sources_1/imports/ALU.srcs/s
Warning: NUMERIC_STD."=": metavalue detected, returning FALSE
Time: 0 ps  Iteration: 0  Process: /ALU_tb/UUT/line__56  File: E:/York/Digital_Design/VivadoProjects/ALU_standalone/ALU_standalone.srcs/sources_1/imports/ALU.srcs/s
Warning: NUMERIC_STD."=": metavalue detected, returning FALSE
Time: 0 ps  Iteration: 0  Process: /ALU_tb/UUT/line__54  File: E:/York/Digital_Design/VivadoProjects/ALU_standalone/ALU_standalone.srcs/sources_1/imports/ALU.srcs/s
Note: The output corresponds to expectation at test vector 0
Time: 120 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU_standalone/ALU_standalone.srcs/sources_1/imports/ALU.srcs
Note: The output corresponds to expectation at test vector 1
Time: 140 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU_standalone/ALU_standalone.srcs/sources_1/imports/ALU.srcs
Note: The output corresponds to expectation at test vector 2
Time: 160 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU_standalone/ALU_standalone.srcs/sources_1/imports/ALU.srcs
Note: The output corresponds to expectation at test vector 3
Time: 180 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU_standalone/ALU_standalone.srcs/sources_1/imports/ALU.srcs
Note: The output corresponds to expectation at test vector 4
Time: 200 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU_standalone/ALU_standalone.srcs/sources_1/imports/ALU.srcs
Failure: Test vector 5 failed for inputs a = 0 and b = 0 and x = 0 and opcode = 7. Expected flags = 82 and ALU output =-1; observed  flags = 42 and ALU output =-1
Time: 220 ns  Iteration: 0  Process: /ALU_tb/test_process  File: E:/York/Digital_Design/VivadoProjects/ALU_standalone/ALU_standalone.srcs/sources_1/imports/ALU.srcs
$finish called at time : 220 ns : File "E:/York/Digital_Design/VivadoProjects/ALU_standalone/ALU_standalone.srcs/sources_1/imports/ALU.srcs/sim_1/new/ALU_tb.vhd" Li
INFO: [USF-XSim-96] XSim completed. Design snapshot 'ALU_tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:04 ; elapsed = 00:00:08 . Memory (MB): peak = 758.516 ; gain = 0.000
```


Name
Value
A[15:0]
UUUU
B[15:0]
UUUU
X[3:0]
U
Opcode[3:0]
UUUU
flags[7:0]
97
ALU_out[15:0]
0000
20 ns
40 ns
60 ns
80 ns
100 ns
120 ns
140 ns
160 ns
180 ns
200 ns
UUUU
abab
3434
3a3a
5a5a
aaaa
0000
0000
3535
3c3c
0000
U
0
UUUU
0000
0100
0101
0110
0111
97
42
82
42
0000
abab
3434
3030
7e7e
9696
ffff

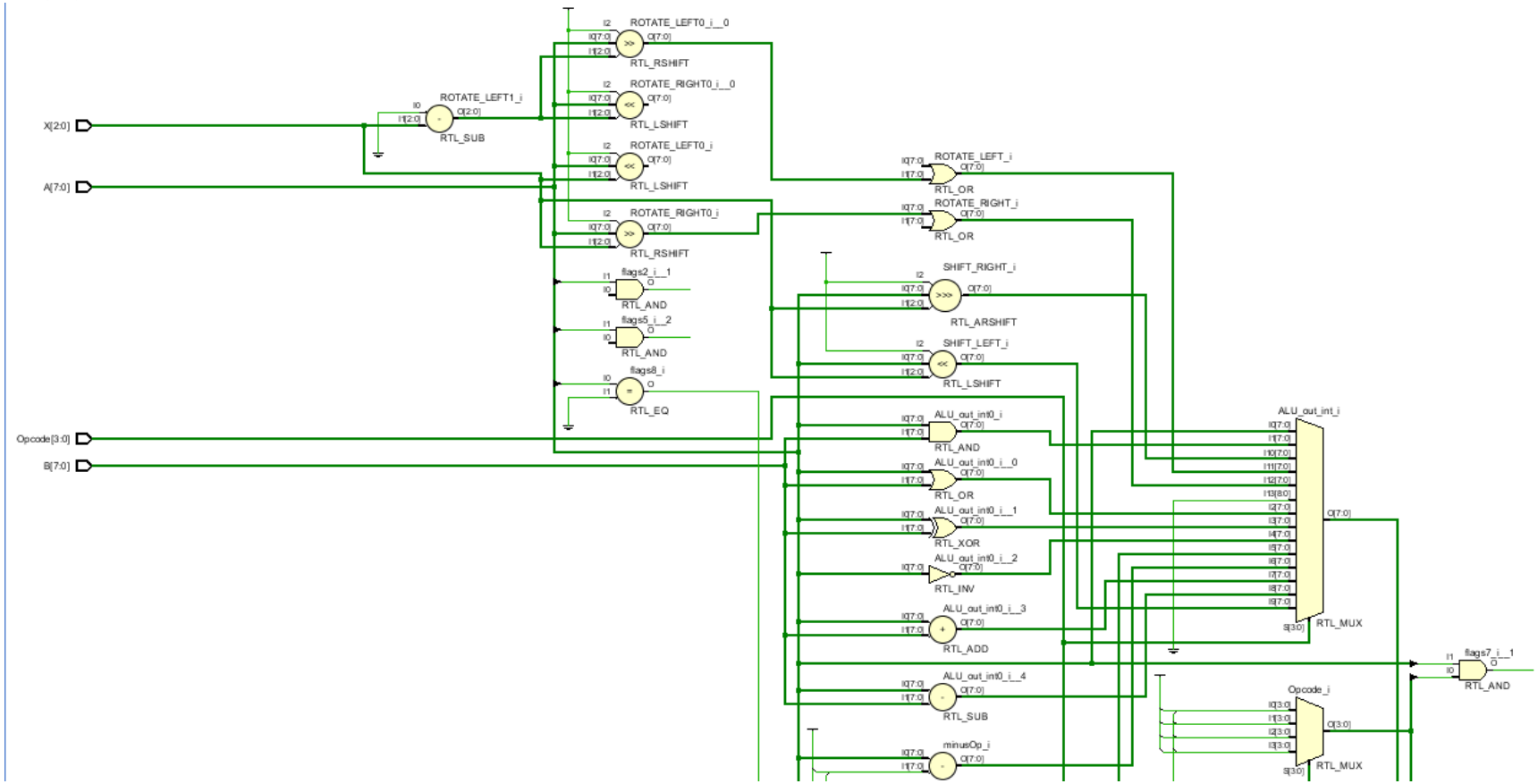
```
---------------------------------------------------------------------------------
Start RTL Component Statistics
---------------------------------------------------------------------------------
Detailed RTL Component Info :
+---Adders :
          2 Input      8 Bit       Adders := 1
          3 Input      8 Bit       Adders := 1
+---XORs :
          2 Input      8 Bit         XORs := 1
+---Muxes :
         14 Input      8 Bit        Muxes := 2
          2 Input      1 Bit        Muxes := 2
---------------------------------------------------------------------------------
Finished RTL Component Statistics

---------------------------------------------------------------------------------
Start RTL Hierarchical Component Statistics
---------------------------------------------------------------------------------
Hierarchical RTL Component report
Module ALU
Detailed RTL Component Info :
+---Adders :
          2 Input      8 Bit       Adders := 1
          3 Input      8 Bit       Adders := 1
+---XORs :
          2 Input      8 Bit         XORs := 1
+---Muxes :
         14 Input      8 Bit        Muxes := 2
          2 Input      1 Bit        Muxes := 2
---------------------------------------------------------------------------------
Finished RTL Hierarchical Component Statistics
```

ALU Schematics:

An overview, the schematics was generated for an 8-bit data size for the sake of compactness:

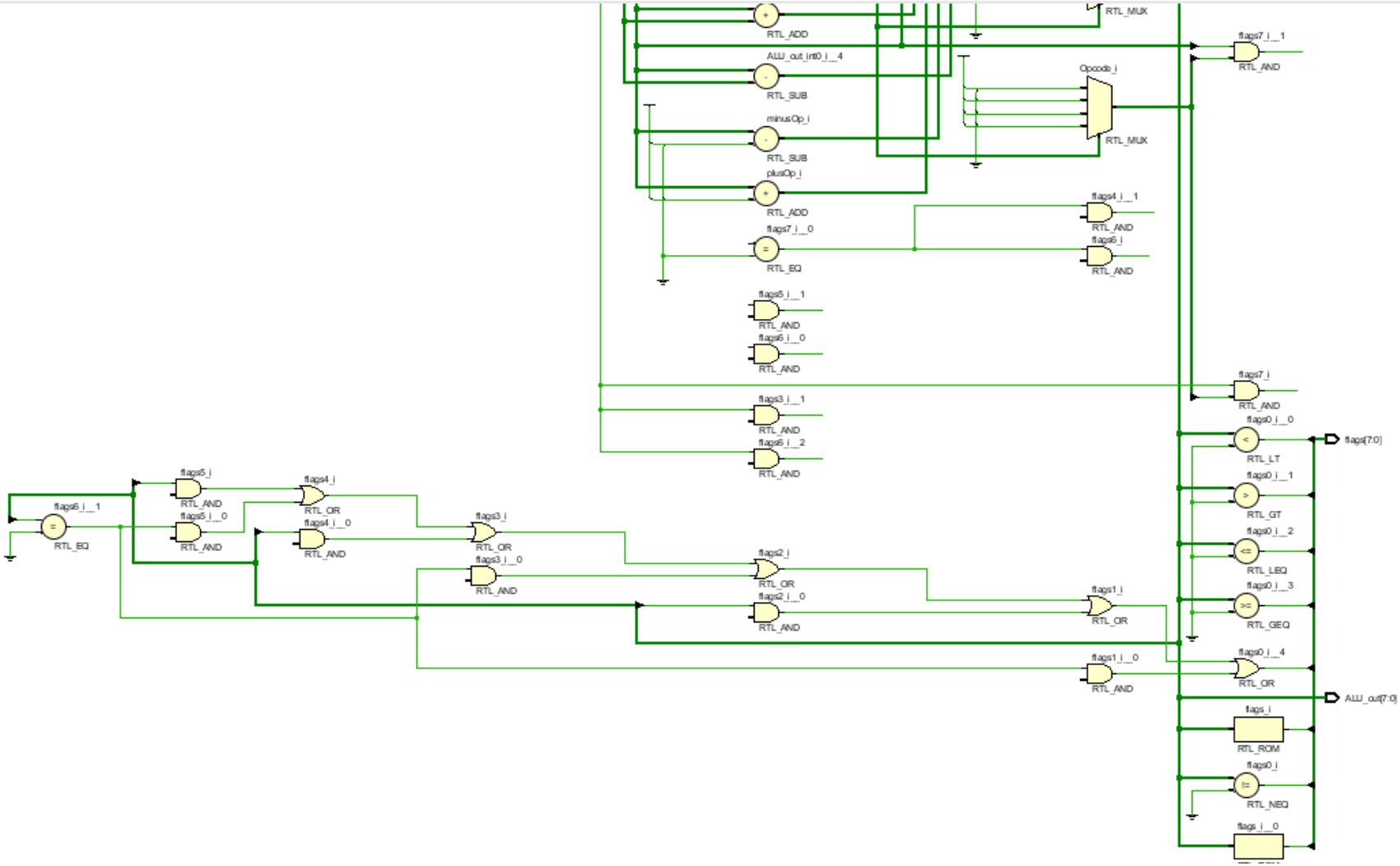


Top part:



This part is mostly consisted of the circuits that implement the first “with-select” in the code. The functions of the ALU are produced with corresponding logic and RTL units and then they get sellected by the large MUX in the right-handside.

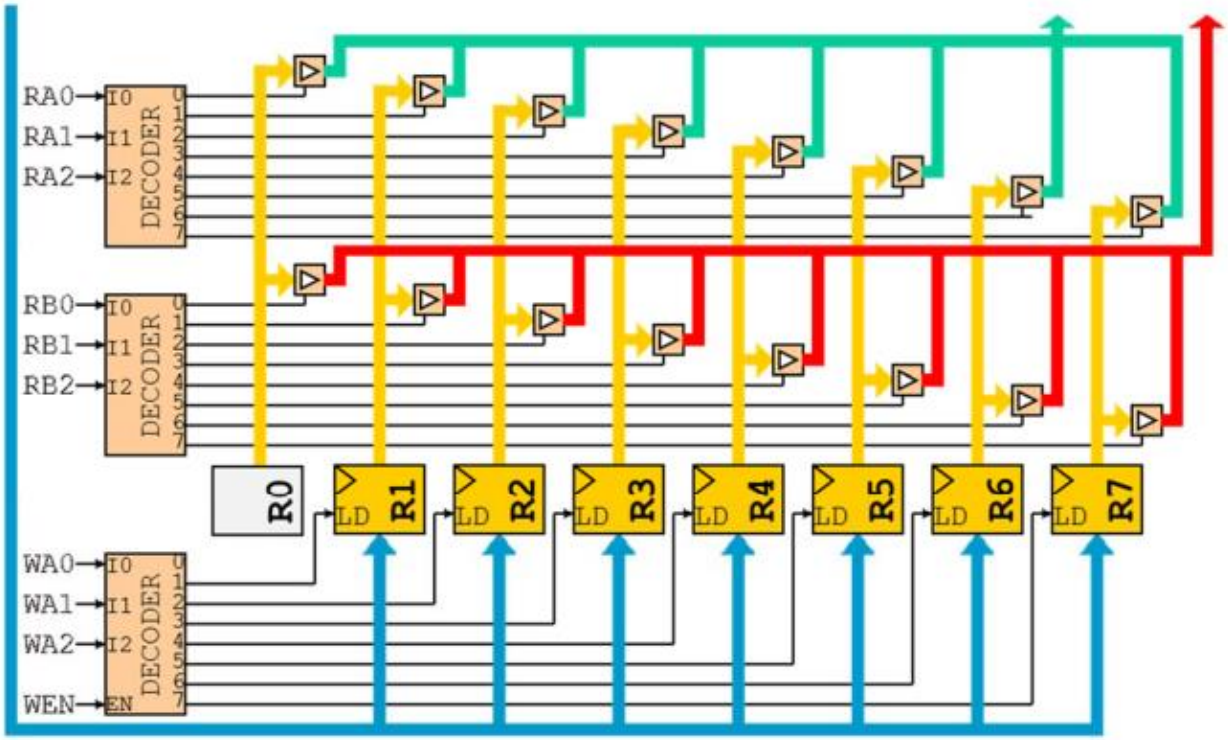
Bottom part:



This part is consisted of the circuits that create flags, as it can be seen the flags are produced based on the ALU output and in some cases the op-codes. The bottom left-handside circuits implement the conditional statements for creating the overflow flag.

Part 2: Design of a (parameterizable) register bank

The objective of this part is to design a register bank containing a parameterizable number of registers of parameterizable size (number of bits). The schematics in below is implemented using VHDL. (In this design, the tri-state buffers are used just for practice as it's not the conventional to use them in datapath of a cpu.)



VHDL code for D-type register:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.DigEng.ALL;
use IEEE.NUMERIC_STD.ALL;

-- This module contains the behavioral description
-- of a D-type register with variable size and synchronous reset
-- It is consisted of a sequential circuit

entity REG_NBIT is
 generic (size : integer := 32);
 port(CLK, RST, WEN : in STD_LOGIC;
      D             : in STD_LOGIC_VECTOR(size-1 downto 0); -- The data input
                                                            -- to the
                                                            -- register
      Q             : out STD_LOGIC_VECTOR(size-1 downto 0) -- The output of
                                                            -- register
      );
end REG_NBIT;

architecture arch of REG_NBIT is
begin
  REG: process (CLK)
  begin
    if (rising_edge(CLK)) then
      if (RST = '1') then
        Q <= (others => '0');
      elsif (WEN = '1') then
        Q <= D;
      end if;
    end if;
  end process REG;
end arch;
```

VHDL code for register bank:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.DigEng.ALL;
use IEEE.NUMERIC_STD.ALL;

-- In this entity the register bank of CPU is defined
-- There are two output buses and only one register can
-- be connected to one bus at a time
-- It is consisted of certain number of registers
-- the control signals determine which register output will be
-- connected to the buses or which register is gonna get written
-- The only sequential part of this circuit is the registers,
-- The rest is consisted of combinational logic
```

```
entity reg_bank is
    generic(data_size : integer := 32;
            num_reg   : integer := 8);
    port( -- the data that goes through the registers
          input_data  : in STD_LOGIC_VECTOR( data_size-1 downto 0);
          -- the signal which is gonna get decoded and select which register
          -- puts its data on the bus A
          RA          : in STD_LOGIC_VECTOR( log2(num_reg)-1 downto 0);
          -- the signal which is gonna get decoded and select which register
          -- puts its data on the bus B
          RB          : in STD_LOGIC_VECTOR( log2(num_reg)-1 downto 0);
          -- the signal which is gonna get decoded and select which register
          -- is gonna get written to
          WA          : in STD_LOGIC_VECTOR( log2(num_reg)-1 downto 0);
          -- the write enable signal for writing into registers, it
          -- determines if we have any writing to do or not
          WEN         : in STD_LOGIC;
          clk         : in STD_LOGIC;
          rst         : in STD_LOGIC;
          -- the output data on the Bus A
          Data_out_A  : out STD_LOGIC_VECTOR( data_size-1 downto 0);
           -- the output data on the Bus B
          Data_out_B  : out STD_LOGIC_VECTOR( data_size-1 downto 0)

         );

end reg_bank;

architecture Behavioral of reg_bank is

type reg_bank_type is array (num_reg-1 downto 0) of
    std_logic_vector(data_size-1 downto 0);

signal  reg_bank_o : reg_bank_type;                        -- Each element of
                                                           -- this array gets
                                                           -- connected to
                                                           -- the output of
                                                           -- corresponding
                                                           -- register
signal  wen_int    : std_logic_vector(num_reg-1 downto 1); -- The internal
                                                           -- write enable
                                                           -- signal
                                                           -- which gets
                                                           -- connected to
                                                           -- registers
                                                           -- directly

begin
-- The for-generate loop creates the registers from 1 to n
-- and also, the necessary decoders and buffers
```

```
 reg_bank: for i in 1 to num_reg-1 generate     -- it is started from 1
                                                 -- because
                                                 -- register 0 is not going to
                                                 -- be synthesized here
    -- This module contains the behavioral description
    -- of a D-type register with variable size
    -- It is consisted of a sequential circuit
     one_reg: entity work.REG_NBIT
        generic map(size => data_size)
        port map ( clk => clk,
                   rst => rst,
                   WEN => wen_int(i),    -- Write enable signal for register
                   Q   => reg_bank_o(i), -- The output of register
                   D   => input_data);   -- The data input to the register

     -- This circuit implements the tri-state buffer which is between
     -- the output of registers and bus A with the required decoders
     Data_out_A <= reg_bank_o(i) when ( unsigned(RA) = i)
                   else (others => 'Z');

     -- This circuit implements the tri-state buffer which is between
     -- the output of registers and bus B with the required decoders
     Data_out_B <= reg_bank_o(i) when ( unsigned(RB) = i)
                   else (others => 'Z');

     -- This circuit implements the decoder for producing the write enable
     -- signals for the registers
     wen_int(i) <= '1' when ( unsigned(WA) = i and WEN = '1')
                   else '0';

    end generate;

    -- The following set of circuits create the R0 which is
    -- simply grounded bus and required tri-state buffers and decoders
    reg_bank_o(0) <= (others => '0');

    Data_out_A <= reg_bank_o(0) when ( unsigned(RA) = 0)
                  else (others => 'Z');

    Data_out_B <= reg_bank_o(0) when ( unsigned(RB) = 0)
                  else (others => 'Z');



end Behavioral;
```

VHDL code for testbench:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.DigEng.ALL;
use IEEE.NUMERIC_STD.ALL;
```

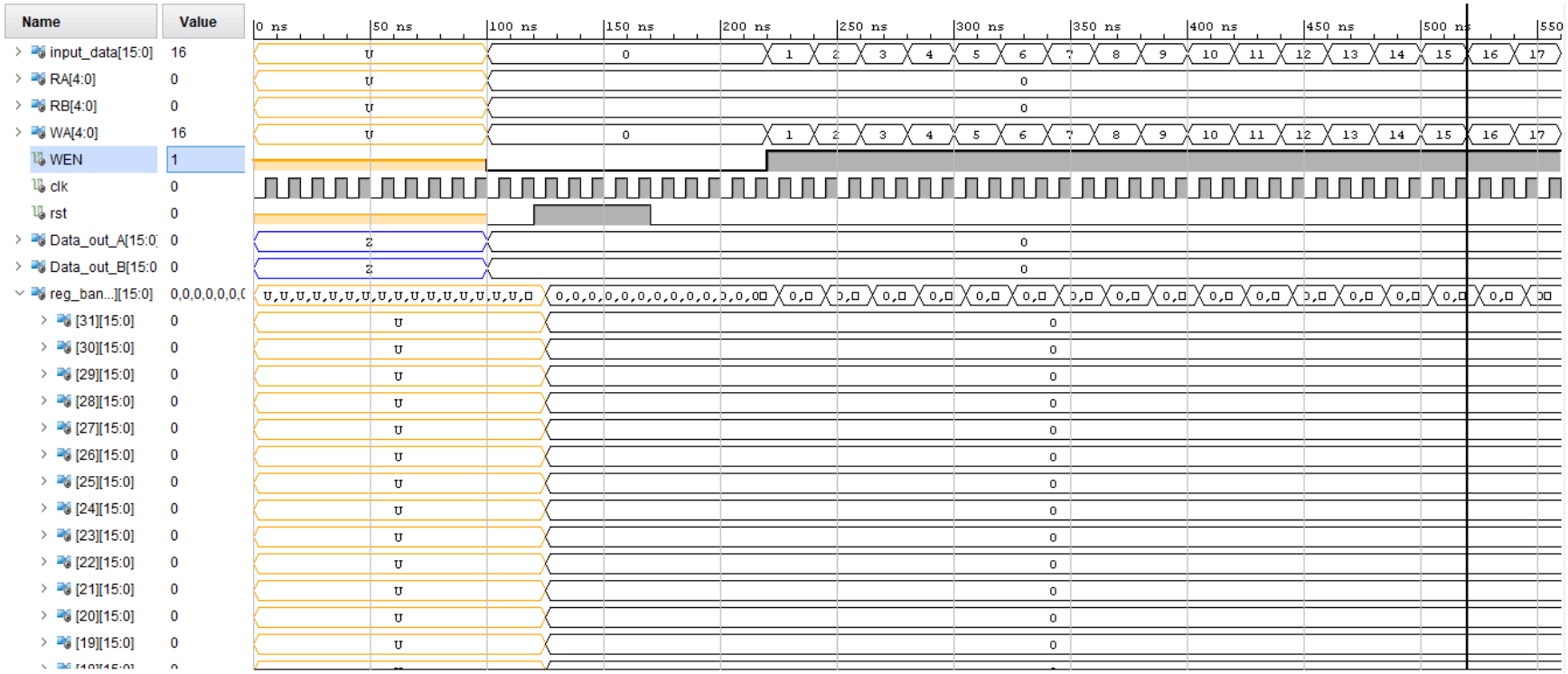
```
entity register_bank_tb is

end register_bank_tb;

architecture Behavioral of register_bank_tb is

constant clk_period : time := 10 ns;
constant data_size : integer := 16;
constant num_reg   : integer := 32;

signal input_data  :  STD_LOGIC_VECTOR( data_size-1 downto 0);
signal RA          :  STD_LOGIC_VECTOR( log2(num_reg)-1 downto 0);
signal RB          :  STD_LOGIC_VECTOR( log2(num_reg)-1 downto 0);
signal WA          :  STD_LOGIC_VECTOR( log2(num_reg)-1 downto 0);
signal WEN         :  STD_LOGIC;
signal clk         :  STD_LOGIC;
signal rst         :  STD_LOGIC;
signal Data_out_A  :  STD_LOGIC_VECTOR( data_size-1 downto 0);
signal Data_out_B  :  STD_LOGIC_VECTOR( data_size-1 downto 0);


begin

UUT : entity work.reg_bank
  generic map( data_size => data_size,
               num_reg   => num_reg)
  port map(
       input_data => input_data,
       RA => RA,
       RB =>  RB,
       WA =>  WA,
       WEN =>  WEN,
       clk =>  clk,
       rst =>  rst,
       Data_out_A =>  Data_out_A,
       Data_out_B =>  Data_out_B);

-- Clock process
clk_process :process
begin
     clk <= '0';
     wait for clk_period/2;
     clk <= '1';
     wait for clk_period/2;
end process;

-- For the purpose of testing this module the strategy is to first -
-- initializing the registers
-- to some different values and then selecting different registers on
-- different busses
-- The middle reset needed for testing sequential circuits is also performed
-- and then the same strategy is repeated
-- with this method we can make sure the tri-state buffers and decoders work
-- well and also there is no problem with writing into registers
```

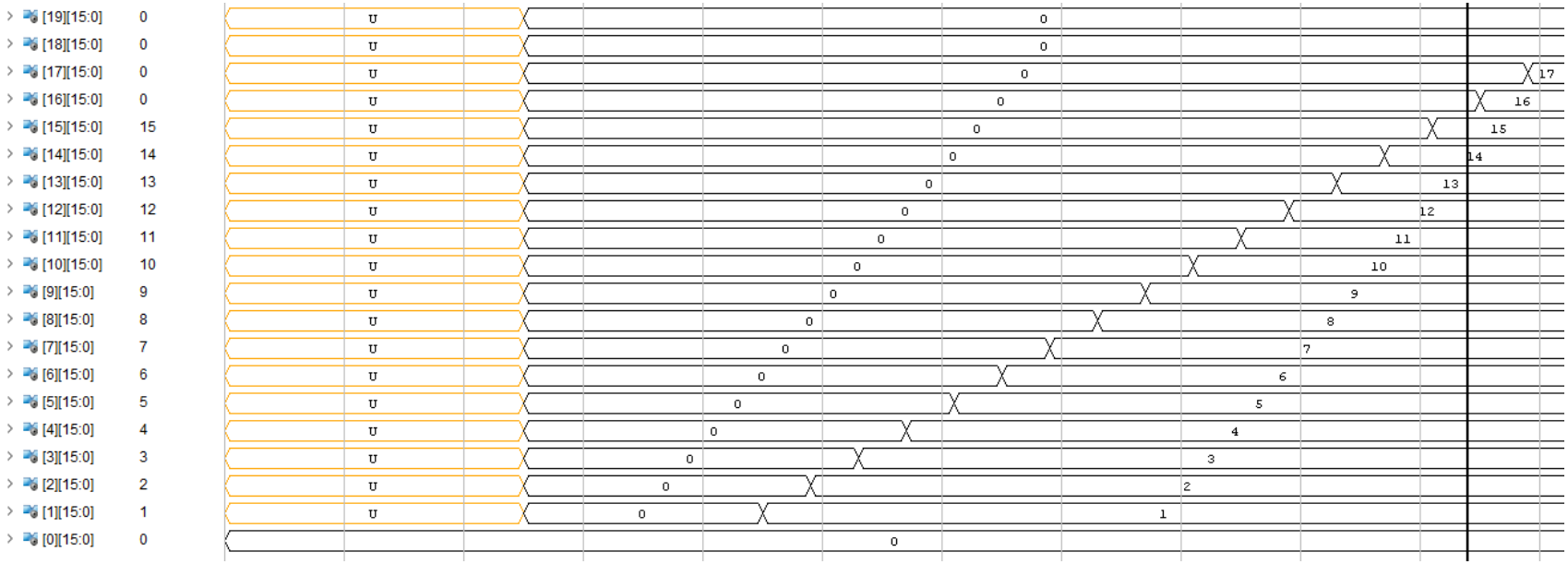
```
test_process : process
begin
    -- wait 100 ns for global reset to finish
    wait for 100ns;

    wait until falling_edge(clk); -- Clock Synchronization

    -- initializing inputs
    input_data <= (others => '0');
    RA  <= (others => '0');
    RB  <= (others => '0');
    WA  <= (others => '0');
    WEN <= '0';
    rst <= '0';
    wait for clk_period*2;

    rst <= '1';                    -- resetting the module
    wait for clk_period*5;

    rst <= '0';
    wait for clk_period*5;

    WEN <= '1';
    initializing_registers : for i in 1 to num_reg-1 loop

        WA <= std_logic_vector(to_unsigned(i, log2(num_reg)));
        input_data <= std_logic_vector(to_signed(i, data_size));
        wait for clk_period*2;

    end loop initializing_registers;

    WEN <= '0';
    display_on_bus : for i in 0 to num_reg-1 loop

        RA <= std_logic_vector(to_unsigned(i, log2(num_reg)));
        RB <= std_logic_vector(to_unsigned((num_reg-i-1), log2(num_reg)));
        wait for clk_period*2;

    end loop display_on_bus;

    rst <= '1';                    -- middle reset
    wait for clk_period*1;

    rst <= '0';
    wait for clk_period*1;

    WEN <= '1';
    initializing_registers_2 : for i in 1 to num_reg-1 loop

         WA <= std_logic_vector(to_unsigned(i, log2(num_reg)));
         input_data <= std_logic_vector(to_signed(i+10, data_size));
         wait for clk_period*2;

     end loop initializing_registers_2;

     WEN <= '0';
```

```
    display_on_bus_2 : for i in 0 to num_reg-1 loop

        RA <= std_logic_vector(to_unsigned((num_reg-i-1), log2(num_reg)));
        RB <= std_logic_vector(to_unsigned(i, log2(num_reg)));
        wait for clk_period*2;

    end loop display_on_bus_2;

    wait;

end process;

end Behavioral;
```

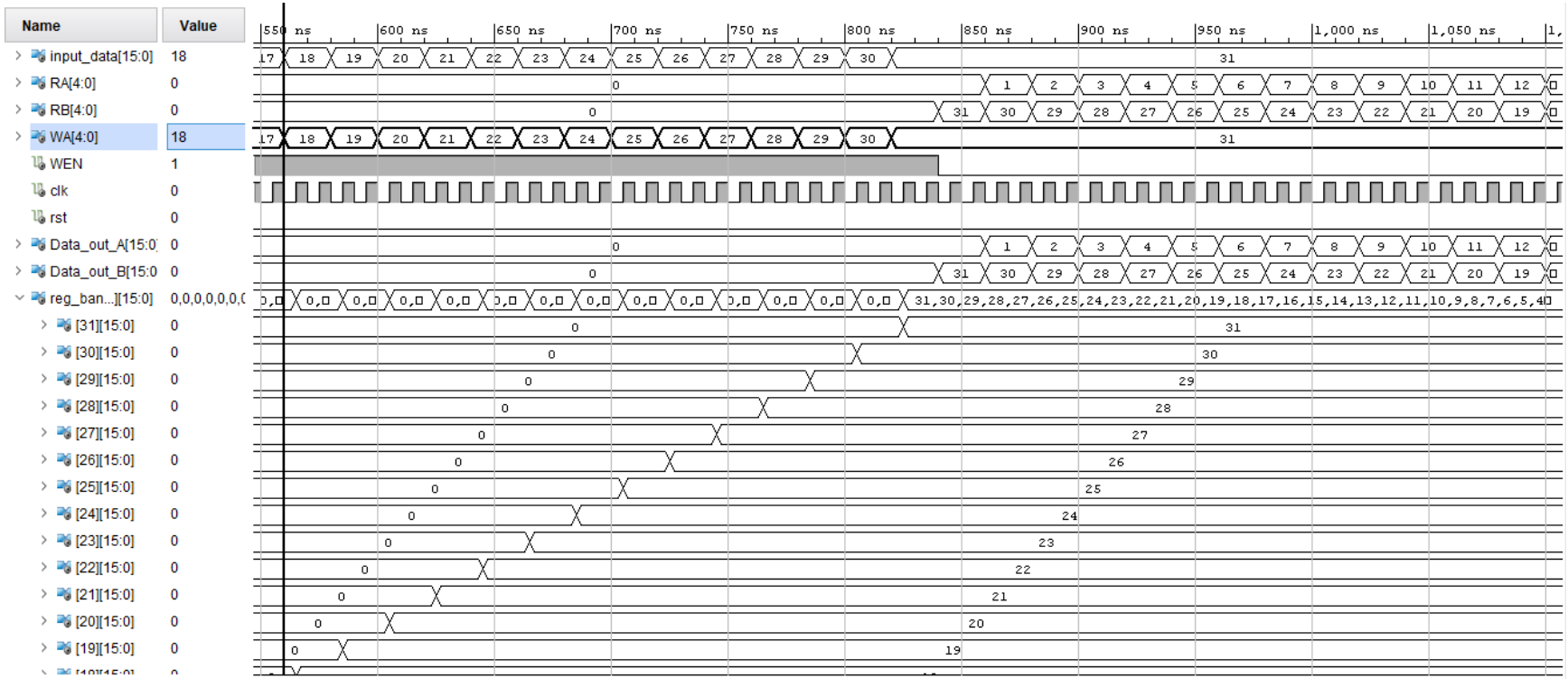
Simulation Screenshots:

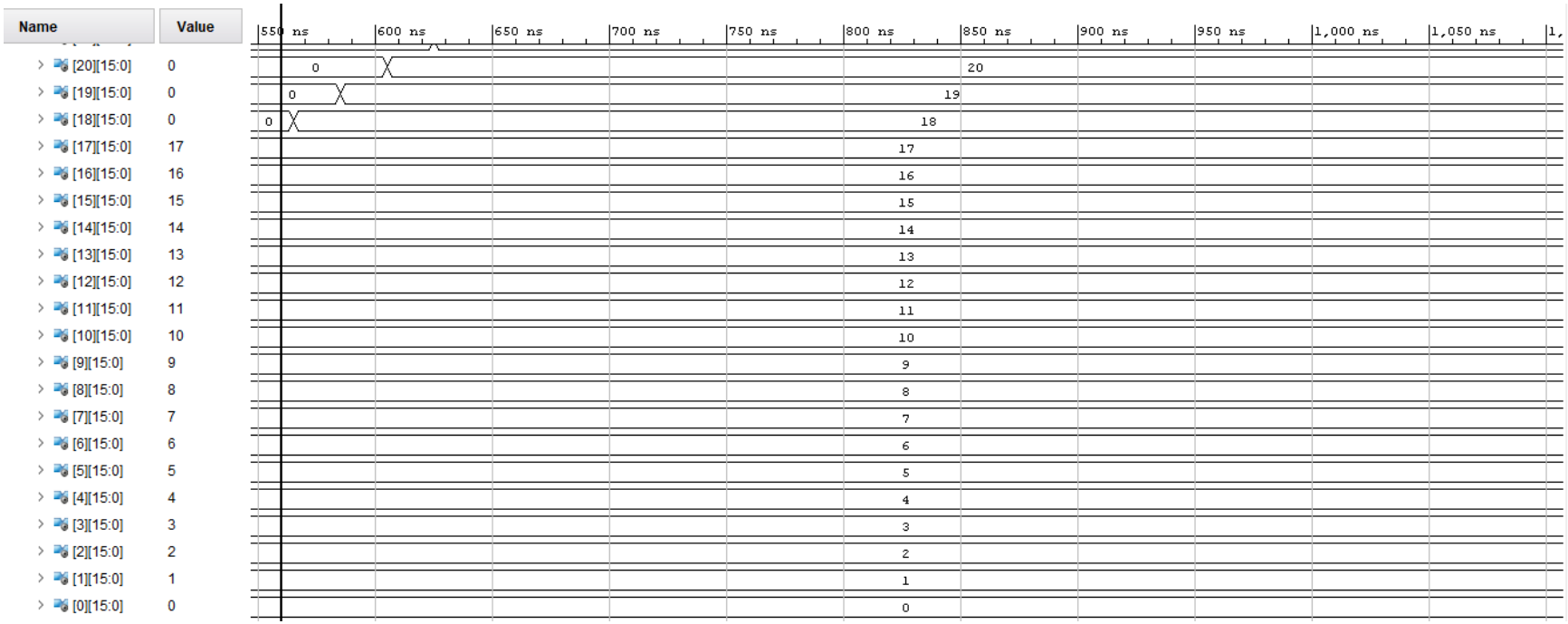
Start of simulation & initializing registers:



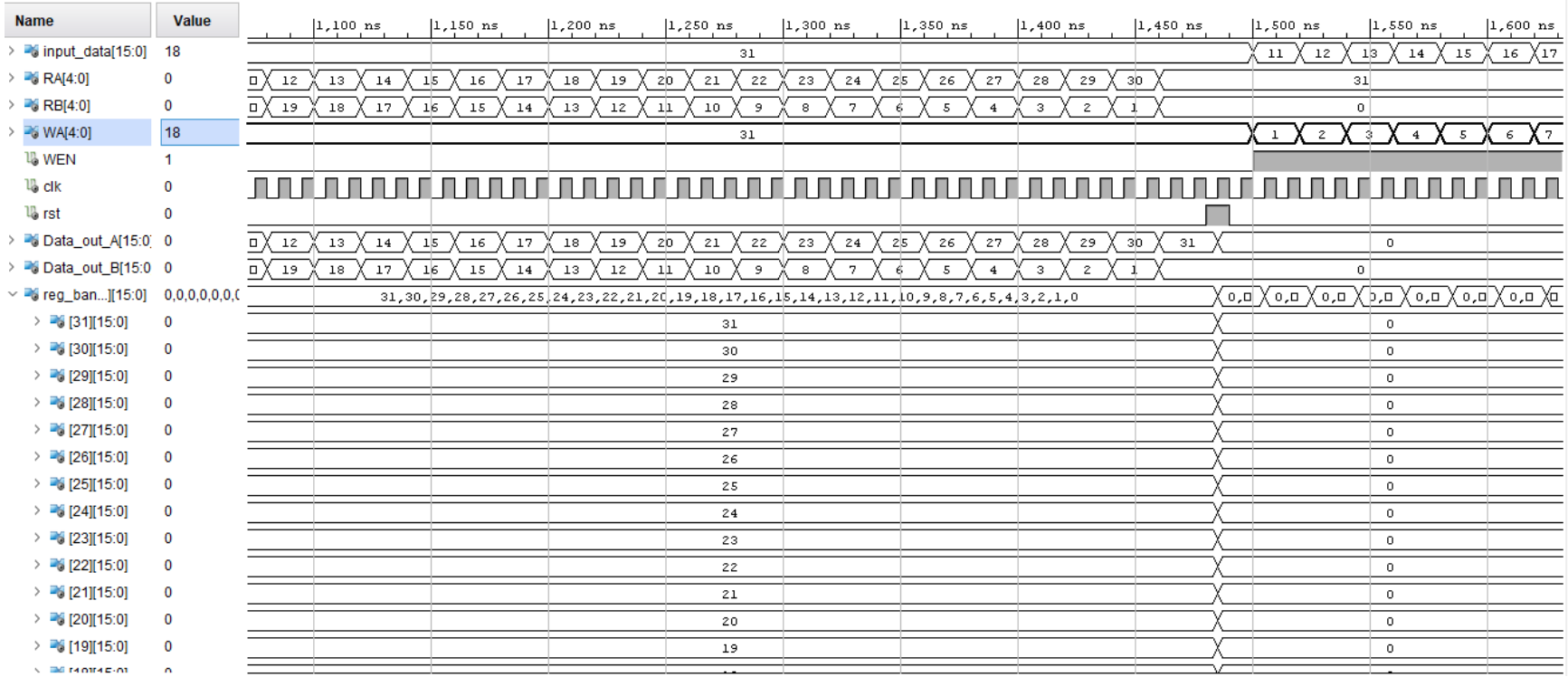

[19][15:0] 0
[18][15:0] 0
[17][15:0] 0
[16][15:0] 0
[15][15:0] 15
[14][15:0] 14
[13][15:0] 13
[12][15:0] 12
[11][15:0] 11
[10][15:0] 10
[9][15:0] 9
[8][15:0] 8
[7][15:0] 7
[6][15:0] 6
[5][15:0] 5
[4][15:0] 4
[3][15:0] 3
[2][15:0] 2
[1][15:0] 1
[0][15:0] 0

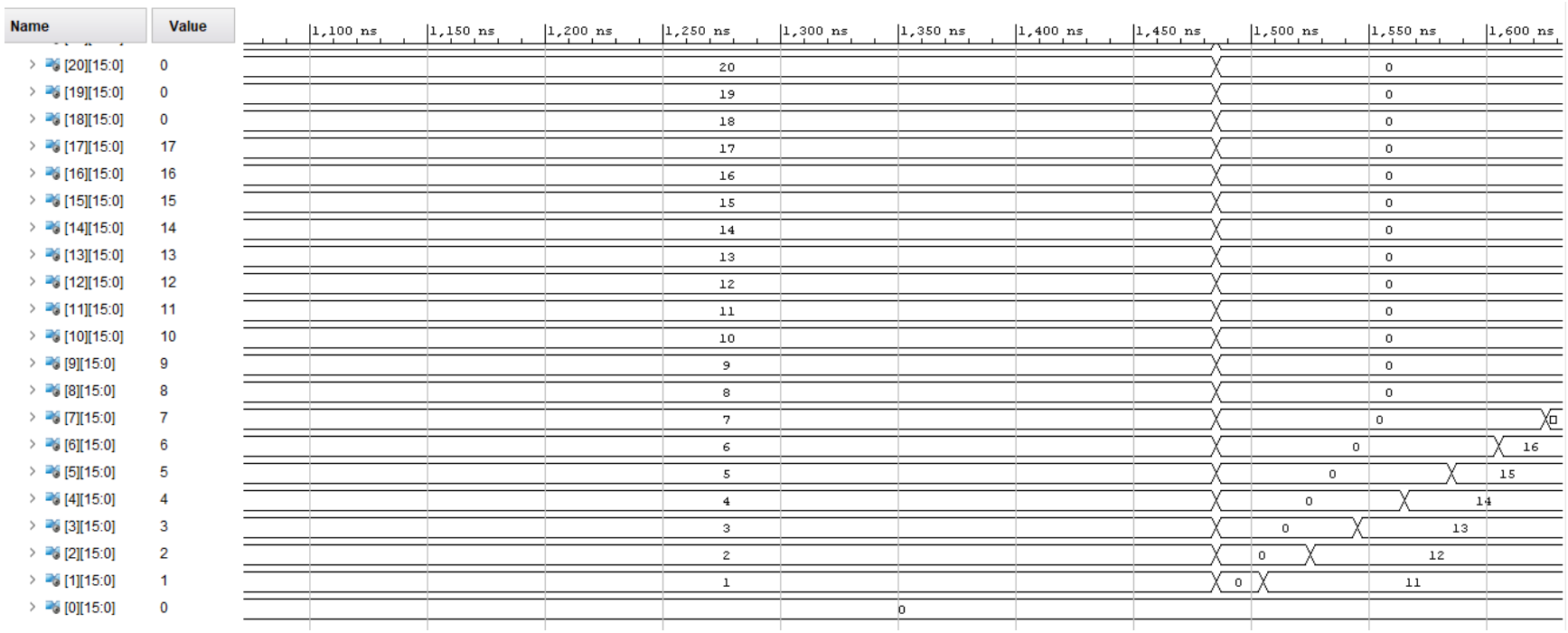
Displaying registers on the buses:



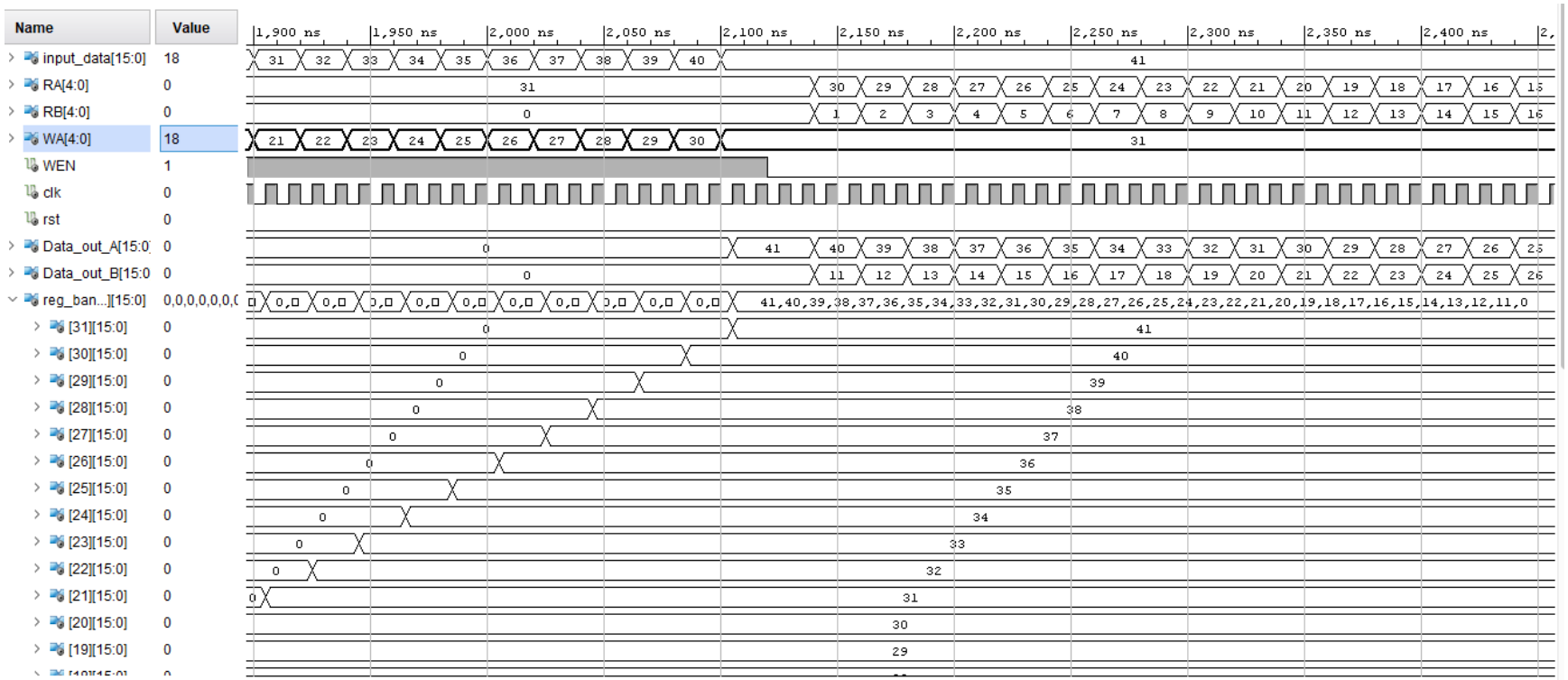

Name
Value
550 ns
600 ns
650 ns
700 ns
750 ns
800 ns
850 ns
900 ns
950 ns
1,000 ns
1,050 ns
[20][15:0] 0
[19][15:0] 0
[18][15:0] 0
[17][15:0] 17
[16][15:0] 16
[15][15:0] 15
[14][15:0] 14
[13][15:0] 13
[12][15:0] 12
[11][15:0] 11
[10][15:0] 10
[9][15:0] 9
[8][15:0] 8
[7][15:0] 7
[6][15:0] 6
[5][15:0] 5
[4][15:0] 4
[3][15:0] 3
[2][15:0] 2
[1][15:0] 1
[0][15:0] 0

Middle reset:

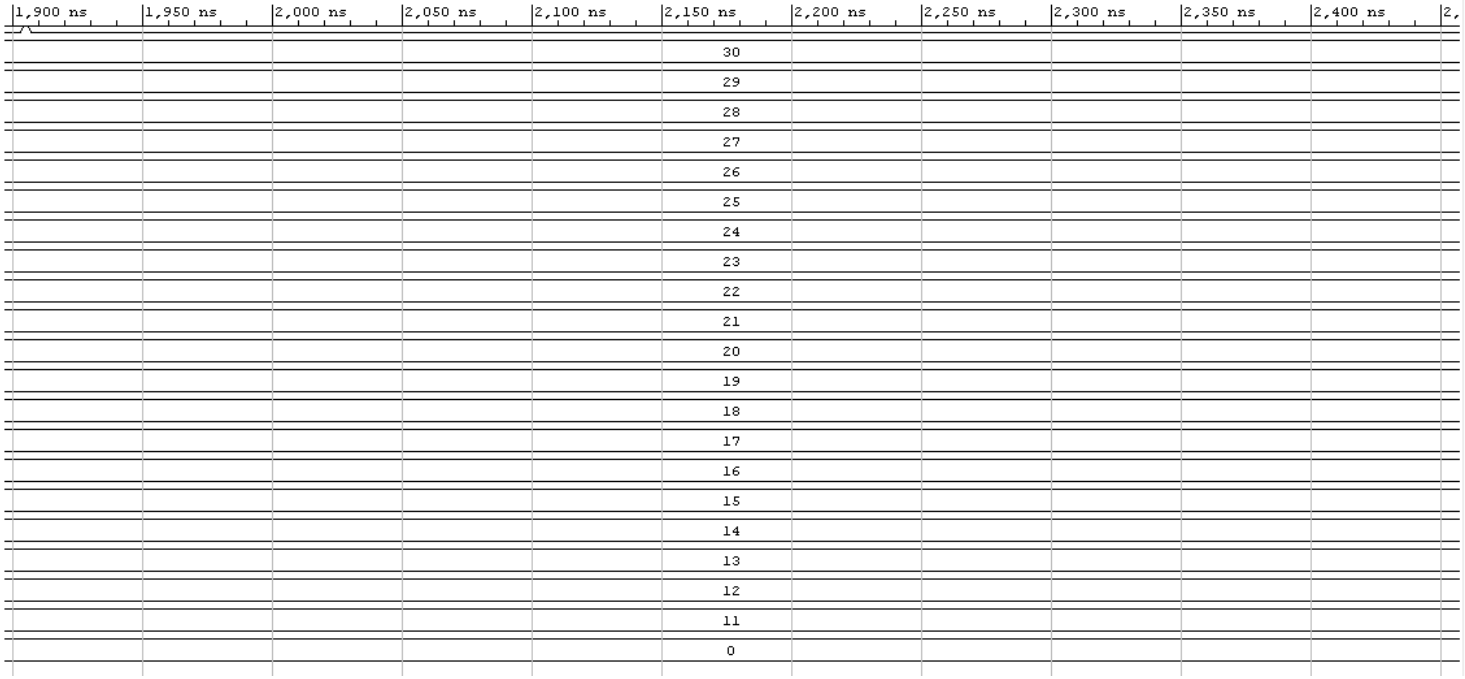



Name
Value
1,100 ns
1,150 ns
1,200 ns
1,250 ns
1,300 ns
1,350 ns
1,400 ns
1,450 ns
1,500 ns
1,550 ns
1,600 ns
[20][15:0] 0
[19][15:0] 0
[18][15:0] 0
[17][15:0] 17
[16][15:0] 16
[15][15:0] 15
[14][15:0] 14
[13][15:0] 13
[12][15:0] 12
[11][15:0] 11
[10][15:0] 10
[9][15:0] 9
[8][15:0] 8
[7][15:0] 7
[6][15:0] 6
[5][15:0] 5
[4][15:0] 4
[3][15:0] 3
[2][15:0] 2
[1][15:0] 1
[0][15:0] 0

The rest of simulation:



Name	Value
[20][15:0]	0
[19][15:0]	0
[18][15:0]	0
[17][15:0]	17
[16][15:0]	16
[15][15:0]	15
[14][15:0]	14
[13][15:0]	13
[12][15:0]	12
[11][15:0]	11
[10][15:0]	10
[9][15:0]	9
[8][15:0]	8
[7][15:0]	7
[6][15:0]	6
[5][15:0]	5
[4][15:0]	4
[3][15:0]	3
[2][15:0]	2
[1][15:0]	1
[0][15:0]	0



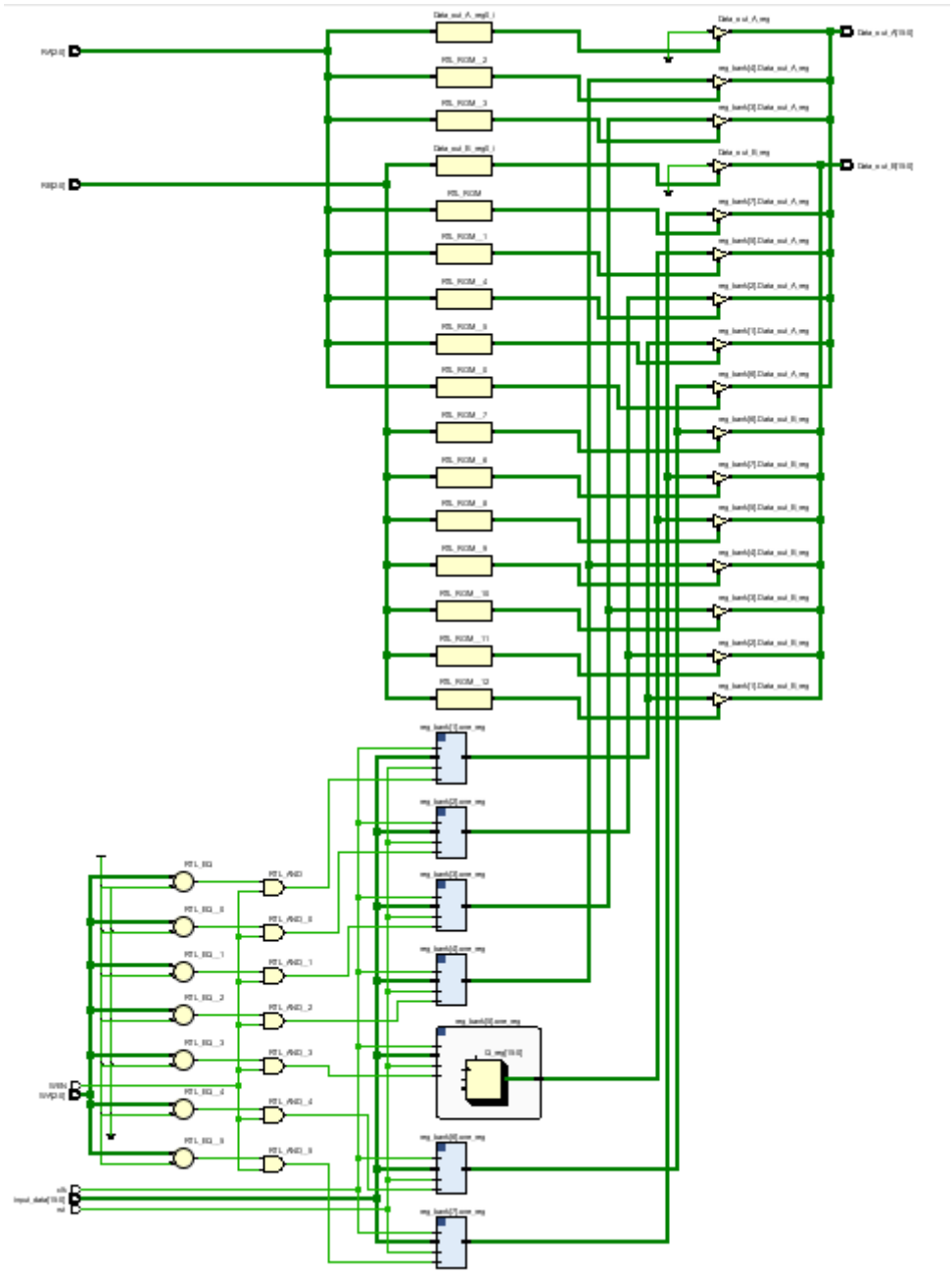
The synthesis was done for 8 registers with 16-bit data size (So it would fit better in screenshot of schematics)

```
---------------------------------------------------------------------------------
Start RTL Component Statistics
---------------------------------------------------------------------------------
Detailed RTL Component Info :
+---Registers :
                      16 Bit    Registers := 7
+---Muxes :
          2 Input     16 Bit        Muxes := 16
---------------------------------------------------------------------------------
Finished RTL Component Statistics
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
Start RTL Hierarchical Component Statistics
---------------------------------------------------------------------------------
Hierarchical RTL Component report
Module reg_bank
Detailed RTL Component Info :
+---Muxes :
          2 Input     16 Bit        Muxes := 16
Module REG_NBIT
Detailed RTL Component Info :
+---Registers :
                      16 Bit    Registers := 1
---------------------------------------------------------------------------------
Finished RTL Hierarchical Component Statistics
---------------------------------------------------------------------------------
```

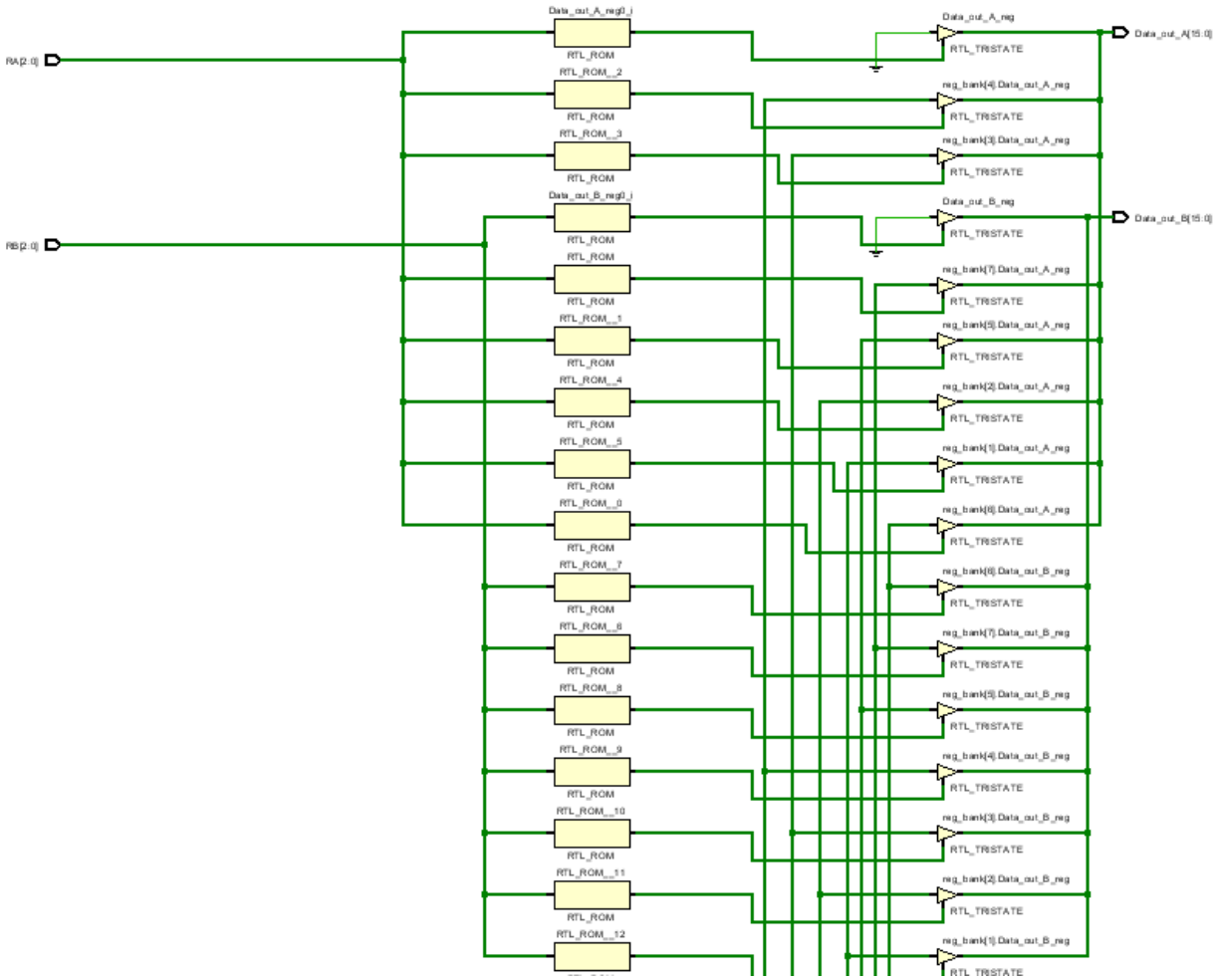
No tri-state buffer can be seen in the report.

Schematics for top-module:

The schematics was generated for 8 registers with 32-bit data for the sake of compact screenshots, Overview of the cell:

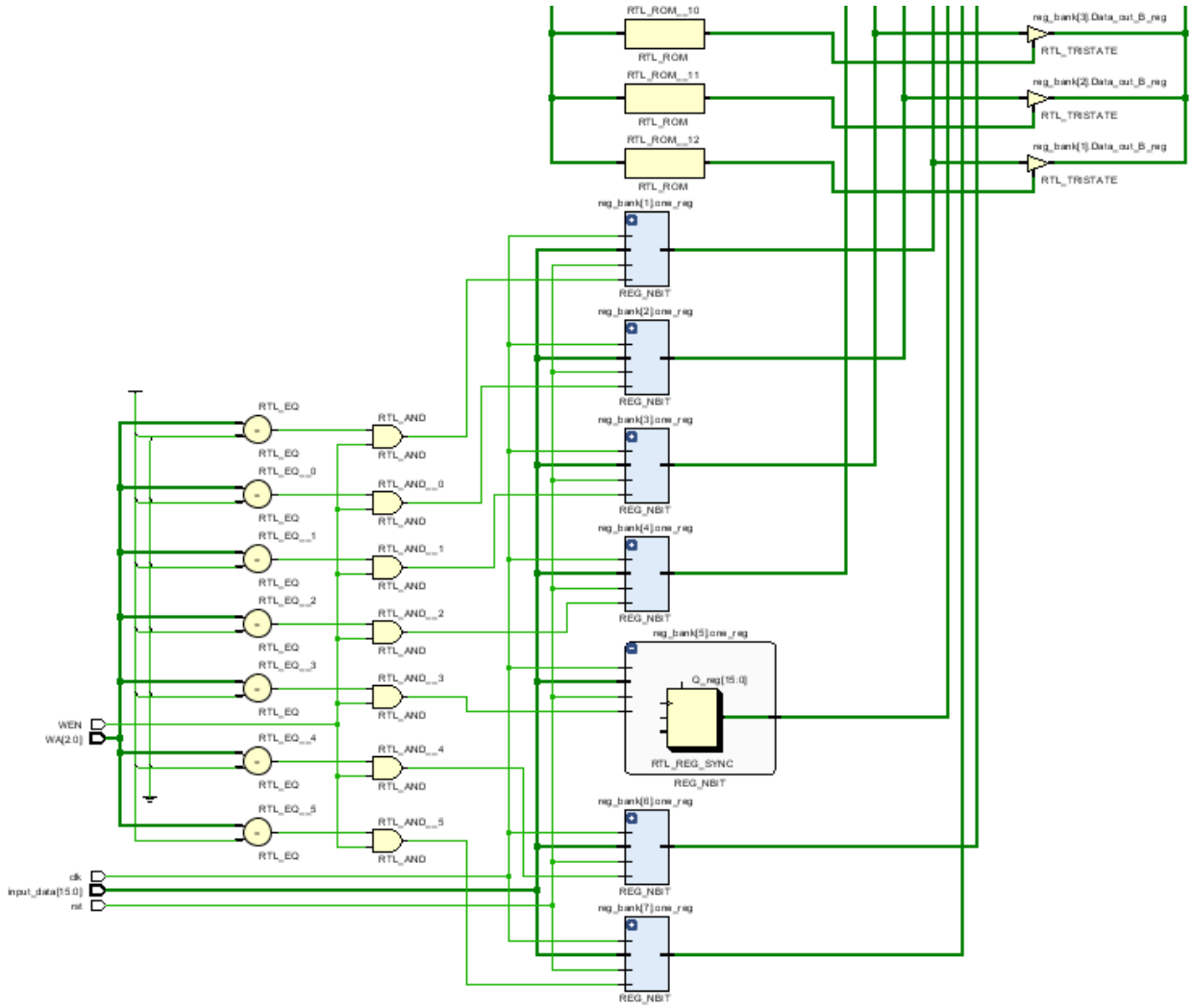


Top part:



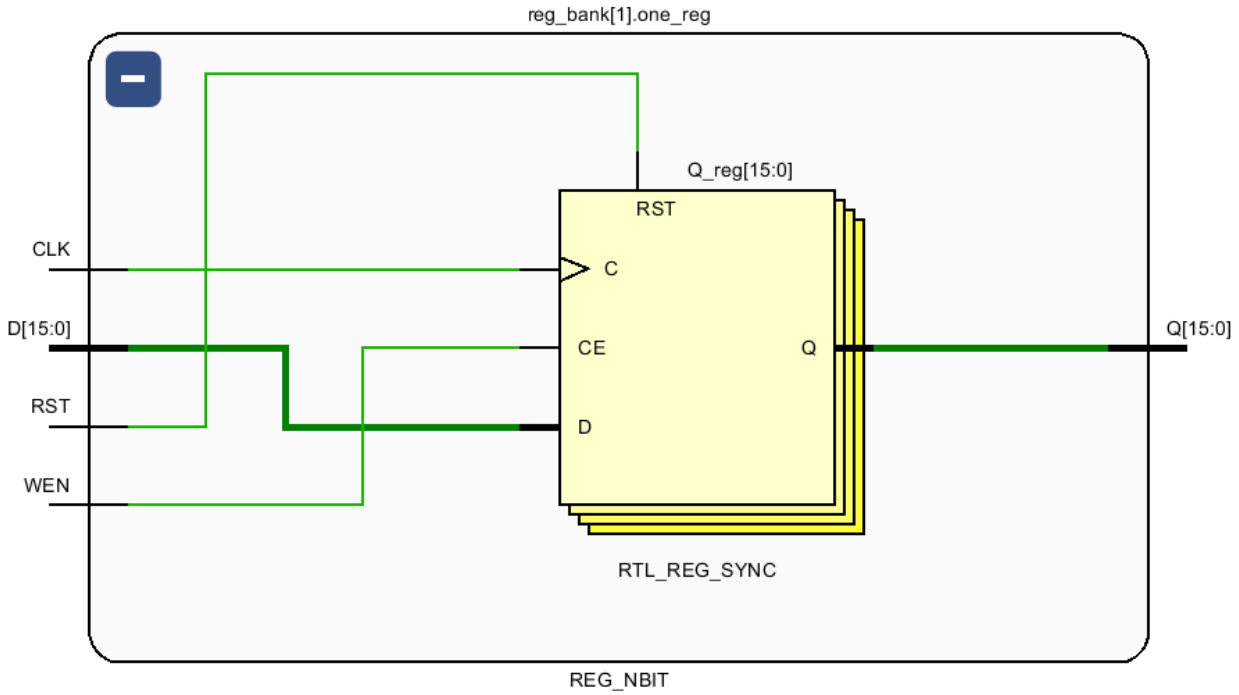
In the top part, the tri-state buffers that get connected to buses are placed. Instead of one wide decoder several smaller ROMs are used to generate each enable signal for tri-states.

Bottom part:



In the bottom part, the registers and the circuitry that generates the write enable signal for them are placed.

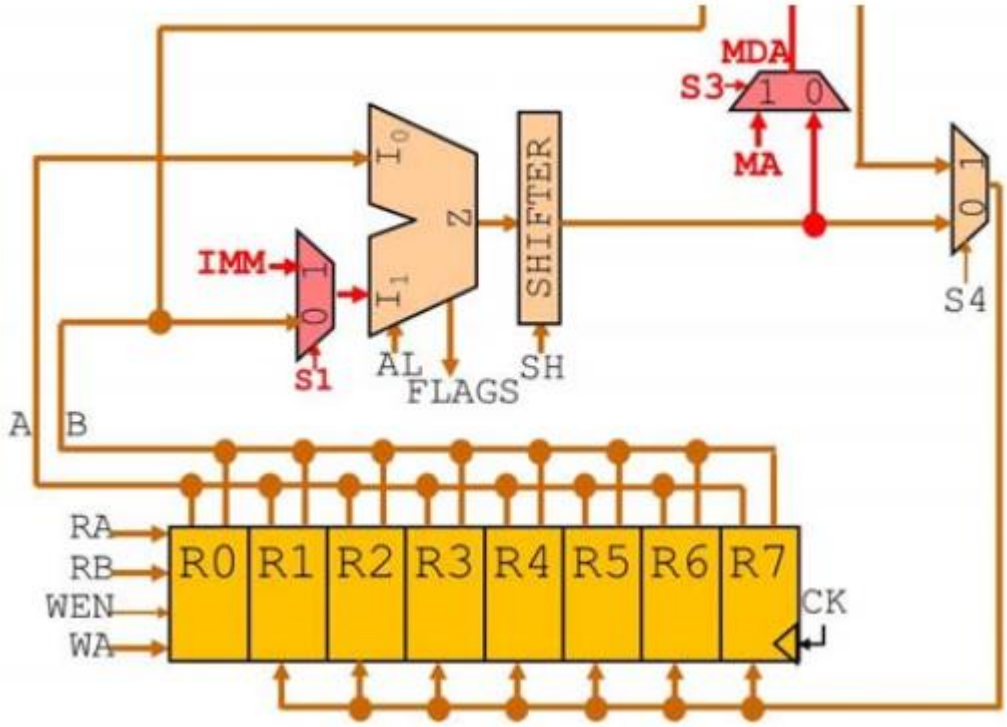
Schematics for D-type register:



The description that is written for D-type register is correctly inferred by the tools as a D-type register with synchronous reset.

Part 3: Design of a (parameterizable) single-cycle datapath

The objective of this part is to design the datapath for a single-cycle processor implementation. The datapath uses the ALU and the register bank implemented in the previous parts. The intended architecture is as depicted below (Except from how the shifter is placed):



The VHDL code for datapath:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.DigEng.ALL;
use IEEE.NUMERIC_STD.ALL;

-- This module implements the data path part for the CPU
-- It is consisted of ALU, register banks and few muxes
-- The register bank is consisted of sequential units
-- but the rest of module is just combinational circuits
entity data_path is
    generic(data_size : integer := 32;
            num_reg   : integer := 8);
    port(AL          : in STD_LOGIC_VECTOR(3 downto 0);-- This control signal
                                                       -- determines which
                                                       -- operation is done
                                                       -- by ALU
         -- This signal determines For How many positions we are shifting
         SH          : in STD_LOGIC_VECTOR (log2(data_size)-1 downto 0);
         clk         : in STD_LOGIC;
         rst         : in STD_LOGIC;
         s           : in STD_LOGIC_VECTOR (2 downto 0);-- The control logic
                                                        -- for  muxes
         -- The immediate value which can get selected for the input of one
         -- of ALU's input
         Imm         : in STD_LOGIC_VECTOR (data_size-1 downto 0);
         -- The immediate value which can get selected as an input address
         -- for memory
         MA          : in STD_LOGIC_VECTOR (data_size-1 downto 0);
         -- the signal which is gonna get decoded and select which register
         -- puts its data on the bus A
         RA          : in STD_LOGIC_VECTOR( log2(num_reg)-1 downto 0);
         -- the signal which is gonna get decoded and select which register
         -- puts its data on the bus B
         RB          : in STD_LOGIC_VECTOR( log2(num_reg)-1 downto 0);
         -- the signal which is gonna get decoded and select which register
         -- is gonna get written to
         WA          : in STD_LOGIC_VECTOR( log2(num_reg)-1 downto 0);
         WEN         : in STD_LOGIC;                     -- the write enable
                                                         -- signal for writing
                                                         -- into registers
                                                         -- It determines if
                                                         -- we have any
                                                         -- writing to do or
                                                         -- not
          -- The data coming from the memory as an input to a mux which its
          -- output gets connected to registers
         Data_mem_i  : in STD_LOGIC_VECTOR (data_size-1 downto 0);

         flags       : out STD_LOGIC_VECTOR(7 downto 0);-- The flags that
                                                        -- result from ALU
                                                        -- operations
```

```
        -- The address for memory
        MDA         : out STD_LOGIC_VECTOR(data_size-1 downto 0);
        -- The data from bus B of register banks which is an output data to
        -- memory
        Data_mem_o  : out STD_LOGIC_VECTOR (data_size-1 downto 0)
    );

end data_path;

architecture Behavioral of data_path is
-- The outputs from register banks' buses
signal A       : STD_LOGIC_VECTOR( data_size-1 downto 0);
signal B       : STD_LOGIC_VECTOR( data_size-1 downto 0);
Signal I1         : STD_LOGIC_VECTOR( data_size-1 downto 0);    -- The output
                                                                -- going to
                                                                -- I1 port of
                                                                -- the ALU
signal ALU_output : STD_LOGIC_VECTOR( data_size-1 downto 0);
signal reg_input_data : STD_LOGIC_VECTOR( data_size-1 downto 0);-- The input
                                                                -- data for
                                                                -- register
                                                                -- banks

begin
-- This module implements the Arithmatic logic unit of the CPU
-- The module is only consisted of combinational logic units
int_ALU : entity work.ALU
            Generic map(data_size => data_size)
            PORT MAP (
               A => A,                -- Input data
               B => I1,               -- Input data
               -- This signal determines For How many positions we are
               -- shifting
               X => SH,
               Opcode => AL,          -- This control signal determines
                                      -- which operation is done by ALU
               flags  => flags,       -- The output which shows the
                                      -- characteristics
                                      -- of the output of ALU operation
               ALU_out => ALU_output -- output parallel data
               );

-- In this entity the register bank of CPU is defined
-- There are two output buses and only one register can
-- be connected to one bus at a time
-- It is consisted of certain number of registers
-- the control signals determine which register output will be
-- connected to the buses or which register is gonna get written
-- The only sequential part of this circuit is the registers,
-- The rest is consisted of combinational logic
```

```
REG_BANK : entity work.reg_bank
                Generic map(data_size => data_size,
                            num_reg   => num_reg)
                PORT MAP(
                    input_data => reg_input_data,-- the data that goes
                                                 -- through the registers
                    RA         => RA,            -- the signal which is gonna
                                                 -- get decoded and
                                                 -- select which register
                                                 -- puts its data on the bus
                                                 -- A
                    RB         => RB,            -- the signal which is gonna
                                                 -- get decoded and select
                                                 -- which register puts its
                                                 -- data on the bus B
                    WA         => WA,            -- the signal which is gonna
                                                 -- get decoded and select
                                                 -- which register is gonna
                                                 -- get written to
                    WEN        => WEN,           -- the write enable signal
                                                 -- for writing into
                                                 -- registers, It determines
                                                 -- if we have any writing to
                                                 -- do or not
                    clk        => clk,
                    rst        => rst,
                    Data_out_A => A,             -- the output data on the
                                                 -- Bus A
                    Data_out_B => B              -- the output data on the
                                                 -- Bus B
                    );


Data_mem_o <= B;

-- Creating the mux before ALU input
I1 <= B   when ( S(0) = '0') else
      Imm when ( S(0) = '1') else
      (others => 'U') ;

-- Creating the mux for memory address bus
MDA <= ALU_output when ( S(1) = '0') else
       MA         when ( S(1) = '1') else
      (others => 'U')  ;

-- Creating the mux for register inputs
reg_input_data <= ALU_output  when ( S(2) = '0') else
                  Data_mem_i  when ( S(2) = '1') else
                  (others => 'U')  ;


end Behavioral;
```

The VHDL code for testbench:

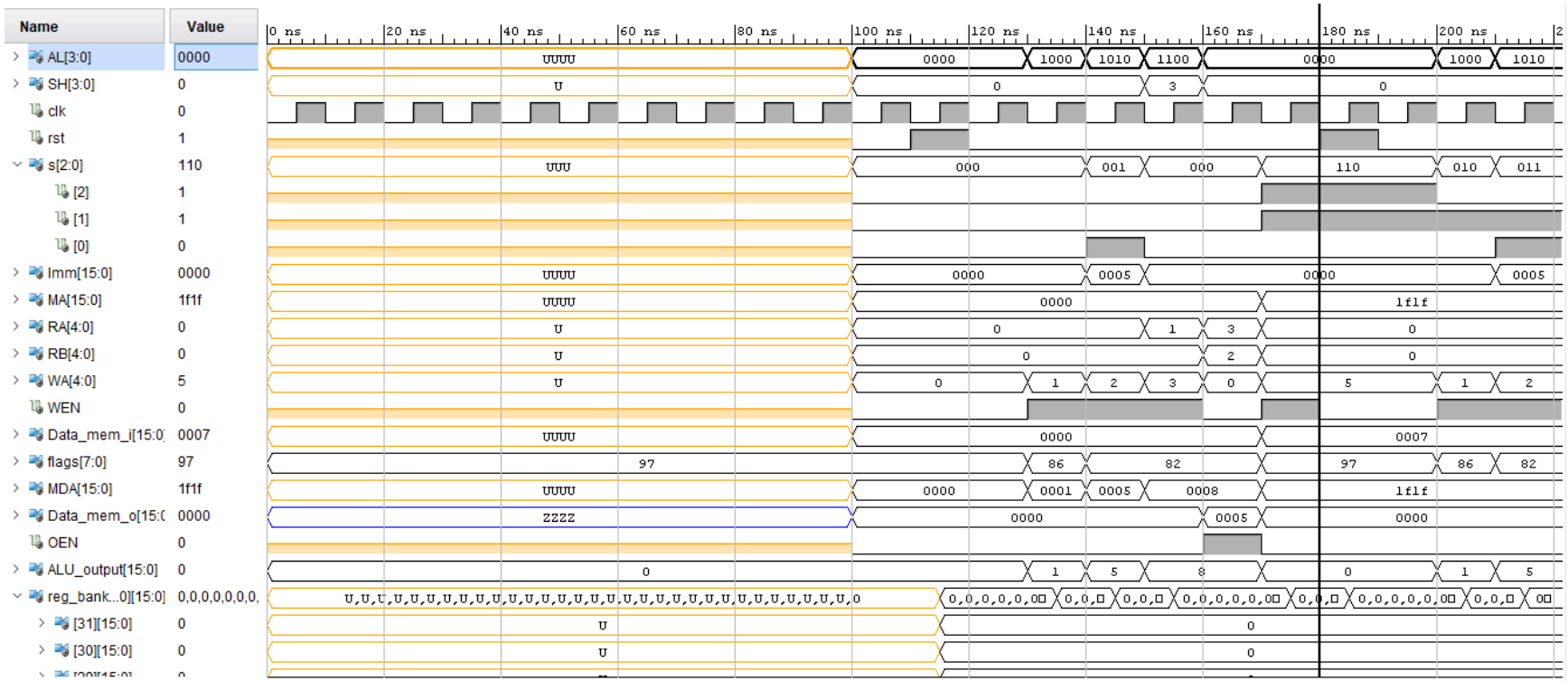
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.DigEng.ALL;
use IEEE.NUMERIC_STD.ALL;


entity datapath_tb is

end datapath_tb;

architecture Behavioral of datapath_tb is

constant clk_period : time := 10 ns;
constant data_size : integer := 16;
constant num_reg   : integer := 32;

signal AL          : STD_LOGIC_VECTOR(3 downto 0);
signal SH          : STD_LOGIC_VECTOR (log2(data_size)-1 downto 0);
signal clk         : STD_LOGIC;
signal rst         : STD_LOGIC;
signal s           : STD_LOGIC_VECTOR (2 downto 0);
signal Imm         : STD_LOGIC_VECTOR (data_size-1 downto 0);
signal MA          : STD_LOGIC_VECTOR (data_size-1 downto 0);
signal RA          : STD_LOGIC_VECTOR( log2(num_reg)-1 downto 0);
signal RB          : STD_LOGIC_VECTOR( log2(num_reg)-1 downto 0);
signal WA          : STD_LOGIC_VECTOR( log2(num_reg)-1 downto 0);
signal WEN         : STD_LOGIC;
signal Data_mem_i  : STD_LOGIC_VECTOR (data_size-1 downto 0);
signal flags       : STD_LOGIC_VECTOR(7 downto 0);
signal MDA         : STD_LOGIC_VECTOR(data_size-1 downto 0);
signal Data_mem_o  : STD_LOGIC_VECTOR (data_size-1 downto 0);
signal OEN         : STD_LOGIC;

begin

UUT : entity work.data_path
  generic map(data_size => data_size,
              num_reg => num_reg)
  port map(
   AL => AL,
   SH => SH,
   clk => clk,
   rst => rst,
   s => s,
   Imm => Imm,
   MA  => MA,
   RA  => RA ,
```

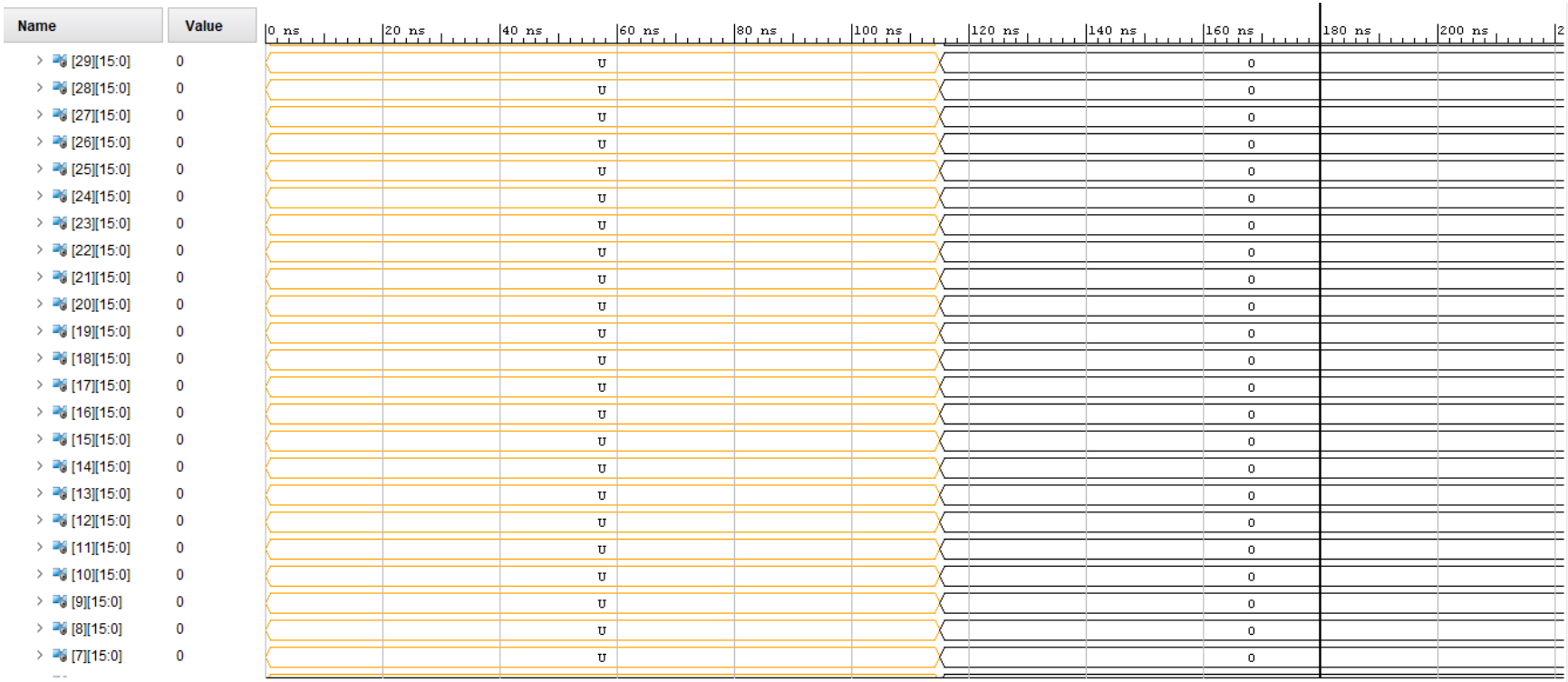
```
   RB  => RB,
   WA  => WA,
   WEN => WEN,
   Data_mem_i => Data_mem_i,
   flags => flags,
   MDA  =>  MDA,
   Data_mem_o => Data_mem_o
   );

-- Clock process
clk_process :process
begin
     clk <= '0';
     wait for clk_period/2;
     clk <= '1';
     wait for clk_period/2;
end process;

-- This module is gonna get tested with a sequence of instructions and
-- verifying if it can execute them properly,
-- for each instruction the corresponding
-- control signals will be generated
-- in the testbench, here are the instruction:
-- inc R1, R0;
-- addi R2, R0, 005;
-- shl R3, R1, 3;
-- storr R2, R3;
-- loadi R5, 1f1f;
-- After this sequence the reset signal is tested and then first two
-- instructions are repeated again

test_process : process
begin
   -- wait 100 ns for global reset to finish
   wait for 100ns;

   wait until falling_edge(clk); -- Clock Synchronization

   -- initializing inputs
   AL  <= (others => '0');
   SH  <= (others => '0');
   rst <= '0';
   s   <= (others => '0');
   Imm <= (others => '0');
   MA  <= (others => '0');
   RA  <= (others => '0');
   RB  <= (others => '0');
   WA  <= (others => '0');
   WEN <= '0';
   Data_mem_i <= (others => '0');
   OEN <= '0';
   wait for clk_period*1;
```

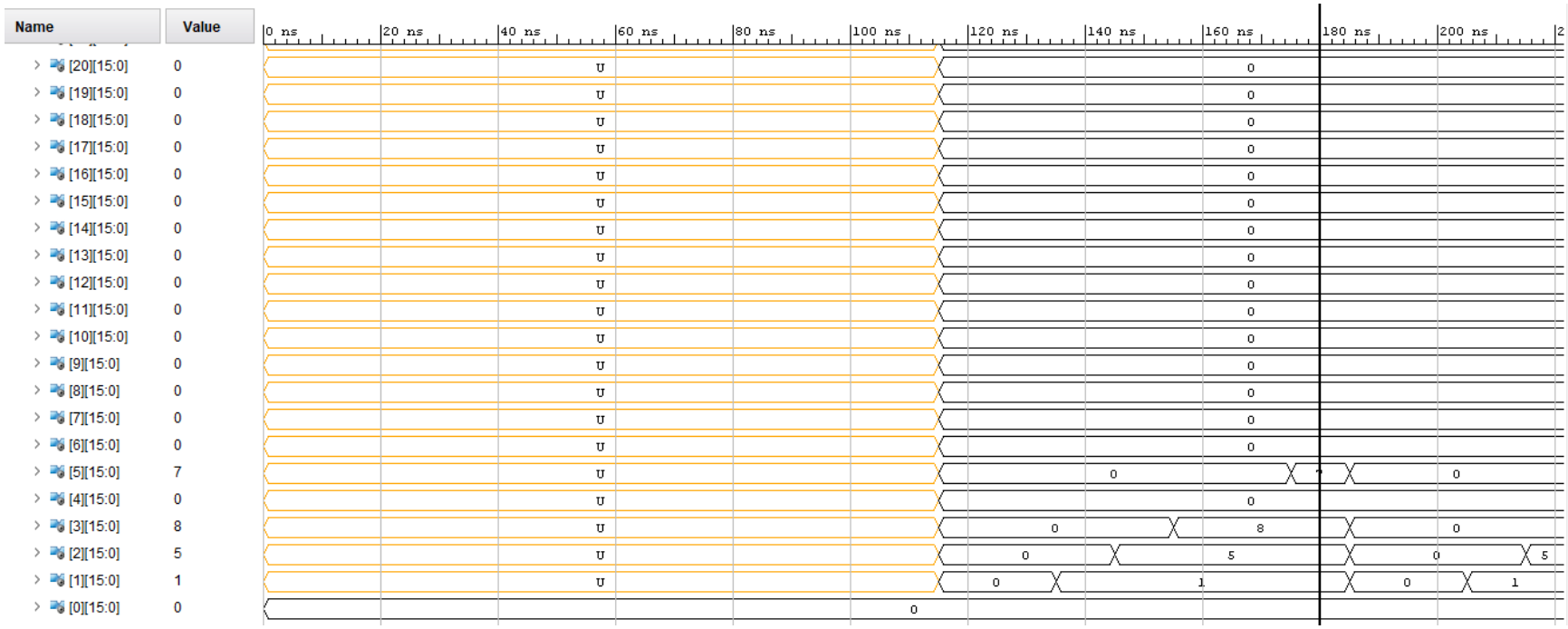
```
rst <= '1';                    -- resetting the module
wait for clk_period*1;

rst <= '0';
wait for clk_period*1;

-- Simulating the control signals for inc R1, R0
RA  <= (others => '0');
AL  <= "1000";
s(2)<= '0';
WEN <= '1';
WA  <= std_logic_vector(to_unsigned(1, log2(num_reg)));
wait for clk_period*1;

-- Simulating the control signals for addi R2, R0, 005
RA   <= (others => '0');
s(0) <= '1';
Imm  <= std_logic_vector(to_signed(5, data_size));
AL   <= "1010";
WEN  <= '1';
WA   <= std_logic_vector(to_unsigned(2, log2(num_reg)));
wait for clk_period*1;

-- Simulating the control signals for shl R3, R1, 3
RA   <= std_logic_vector(to_unsigned(1, log2(num_reg)));
s(0) <= '0';
Imm  <= std_logic_vector(to_signed(0, data_size));
AL   <= "1100";
SH   <= std_logic_vector(to_unsigned(3, size(data_size-1)));
WEN  <= '1';
WA   <= std_logic_vector(to_unsigned(3, log2(num_reg)));
wait for clk_period*1;

-- Simulating the control signals for storr R2, R3
RA   <= std_logic_vector(to_unsigned(3, log2(num_reg)));
RB   <= std_logic_vector(to_unsigned(2, log2(num_reg)));
OEN  <= '1';
AL   <= "0000";
s(1) <= '0';
SH   <= (others => '0');
WEN  <= '0';
WA   <= (others => '0');
wait for clk_period*1;

-- Simulating the control signals for loadi R5, 1f1f
RA   <= (others => '0');
RB   <= (others => '0');
OEN  <= '0';
MA   <= x"1F1F";
s(1) <= '1';
s(2) <= '1';
Data_mem_i <= std_logic_vector(to_signed(7, data_size));
WEN  <= '1';
WA   <= std_logic_vector(to_unsigned(5, log2(num_reg)));
wait for clk_period*1;
```

```
    -- resetting the module, when this module gets integrated
    -- to the final top module and the reset signal is set the
    -- values for WEN and OEN will also get set to zero by the
    -- control unit, here this assignment is also performed to
    -- simulate the reality
    rst <= '1';
    OEN <= '0';
    WEN <= '0';
    wait for clk_period*1;

    rst <= '0';
    wait for clk_period*1;

    -- Simulating the control signals for inc R1, R0
    RA  <= (others => '0');
    AL  <= "1000";
    s(2)<= '0';
    WEN <= '1';
    WA  <= std_logic_vector(to_unsigned(1, log2(num_reg)));
    wait for clk_period*1;

    -- Simulating the control signals for addi R2, R0, 005
    RA   <= (others => '0');
    s(0) <= '1';
    Imm  <= std_logic_vector(to_signed(5, data_size));
    AL   <= "1010";
    WEN  <= '1';
    WA   <= std_logic_vector(to_unsigned(2, log2(num_reg)));
    wait for clk_period*1;

    wait;

end process;

end Behavioral;
```

Simulation screenshots:




Name
Value
0 ns
20 ns
40 ns
60 ns
80 ns
100 ns
120 ns
140 ns
160 ns
180 ns
200 ns
[29][15:0] 0
[28][15:0] 0
[27][15:0] 0
[26][15:0] 0
[25][15:0] 0
[24][15:0] 0
[23][15:0] 0
[22][15:0] 0
[21][15:0] 0
[20][15:0] 0
[19][15:0] 0
[18][15:0] 0
[17][15:0] 0
[16][15:0] 0
[15][15:0] 0
[14][15:0] 0
[13][15:0] 0
[12][15:0] 0
[11][15:0] 0
[10][15:0] 0
[9][15:0] 0
[8][15:0] 0
[7][15:0] 0
U
0

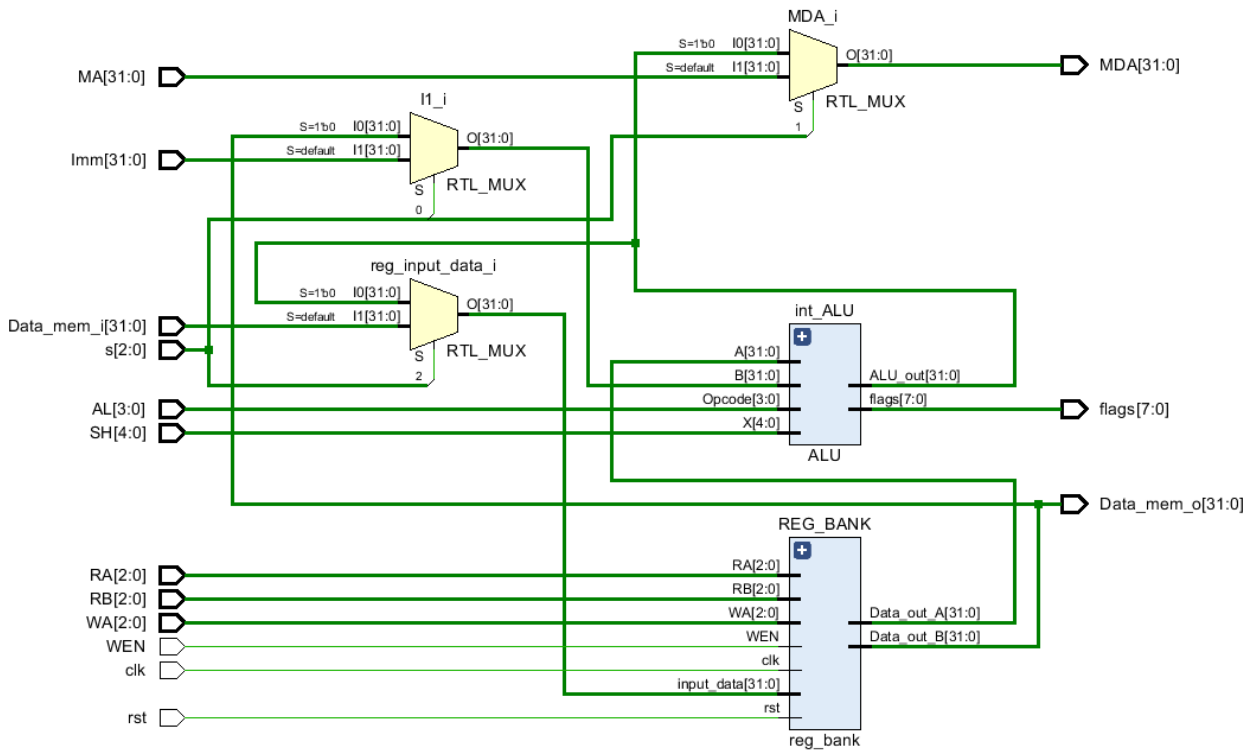

Name
Value
0 ns
20 ns
40 ns
60 ns
80 ns
100 ns
120 ns
140 ns
160 ns
180 ns
200 ns
[20][15:0] 0
[19][15:0] 0
[18][15:0] 0
[17][15:0] 0
[16][15:0] 0
[15][15:0] 0
[14][15:0] 0
[13][15:0] 0
[12][15:0] 0
[11][15:0] 0
[10][15:0] 0
[9][15:0] 0
[8][15:0] 0
[7][15:0] 0
[6][15:0] 0
[5][15:0] 7
[4][15:0] 0
[3][15:0] 8
[2][15:0] 5
[1][15:0] 1
[0][15:0] 0

```
---------------------------------------------------------------------------------
Start RTL Component Statistics
---------------------------------------------------------------------------------
Detailed RTL Component Info :
+---Adders :
          3 Input     32 Bit       Adders := 1
+---XORs :
          2 Input     32 Bit         XORs := 1
+---Registers :
                      32 Bit    Registers := 7
+---Muxes :
         14 Input     32 Bit        Muxes := 2
          2 Input     32 Bit        Muxes := 19
          2 Input      1 Bit        Muxes := 2
---------------------------------------------------------------------------------
Finished RTL Component Statistics
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
Start RTL Hierarchical Component Statistics
---------------------------------------------------------------------------------
Hierarchical RTL Component report
Module data_path
Detailed RTL Component Info :
+---Muxes :
          2 Input     32 Bit        Muxes := 3
Module ALU
Detailed RTL Component Info :
+---Adders :
          3 Input     32 Bit       Adders := 1
+---XORs :
          2 Input     32 Bit         XORs := 1
+---Muxes :
         14 Input     32 Bit        Muxes := 2
          2 Input      1 Bit        Muxes := 2
Module REG_NBIT
Detailed RTL Component Info :
+---Registers :
                      32 Bit    Registers := 1
Module reg_bank
Detailed RTL Component Info :
+---Muxes :
          2 Input     32 Bit        Muxes := 16
---------------------------------------------------------------------------------
Finished RTL Hierarchical Component Statistics
---------------------------------------------------------------------------------
```

The components conform to what we expected, no latches or other unexpected type of components were spotted

Schematics for top-level module:

The schematics was generated for 8 registers with 32-bit data size



The components are instantiated and connected according to expectations.

Part 4: Control Logic design

In this part the instruction set is defined and the value of the control signals for all the instructions in the instruction set is determined, then based on that the control unit of CPU is designed.

This is a processor with 32-bit instructions and 16-bit data. The processor specifications are:

- A dual-read, single-write bank of 32 registers of 16 bits each.
- The offsets encoded in instruction set are able to fully cover:
  - a word-addressable data address space of 512 16-bit words.
  - a word-addressable instruction address space of 256 instructions (i.e. an 8-bit PC).

The list of instructions:

Category	Instruction	Operation	Opcode
No-operation	`nop`		00 0000
Arithmetic	`add rt, ra, rb`	rt <= ra + rb	00 0100
	`sub rt, ra, rb`	rt <= ra – rb	00 0101
	`addi rt, ra, imm`	rt <= ra + immediate value	00 0110
	`subi rt, ra, imm`	rt <= ra – immediate value	00 0111
	`inc rt, ra`	rt <= ra + 1	00 1000
	`dec rt, ra`	rt <= ra – 1	00 1001
Logic	`not rt, ra`	rt <= NOT ra	01 0000
	`and rt, ra, rb`	rt <= ra AND rb	01 0001
	`or rt, ra, rb`	rt <= ra OR rb	01 0010
	`xor rt, ra, rb`	rt <= ra XOR rb	01 0011
	`andi rt, ra, imm`	rt <= ra AND immediate value	01 0100
	`ori rt, ra, imm`	rt <= ra OR immediate value	01 0101
	`xori rt, ra, imm`	rt <= ra XOR immediate value	01 0110
	`shl rt, ra, n`	rt <= ra shifted left by n bits	01 1000
	`shr rt, ra, n`	rt <= ra shifted right by n bits	01 1001
	`rol rt, ra, n`	rt <= ra rotated left by n bits	01 1010
	`ror rt, ra, n`	rt <= ra rotated right by n bits	01 1011
Transfer	`move rt, ra`	rt <= ra	10 0000
	`loadi rt, imm`	rt <= DMEM[imm] {direct addressing}	10 0001
	`loadr rt, ra`	rt <= DMEM[ra] {register indirect addressing}	10 0010
	`loado rt, ra, off`	rt <= DMEM[ra+off] {base plus offset addressing}	10 0011
	`stori rb, imm`	DMEM[imm] <= rb {direct addressing}	10 0101
	`storr rb, ra`	DMEM[ra] <= rb {register indirect addressing}	10 0110
	`storo rb, ra, off`	DMEM[ra+off] <= rb {base plus offset addressing}	10 0111
Control	`jmp off`	Jump to IMEM[PC+off]	11 0111
	`brc ra, cond, off`	If condition is true, then jump to IMEM[PC+off], else continue.	
		Conditions: ra = 0	11 0000
		ra ≠ 0	11 0001
		ra = 1	11 0010
		ra < 0	11 0011
		ra > 0	11 0100
		ra ≤ 0	11 0101
		ra ≥ 0	11 0110

Similar instructions use similar opcodes so the decode logic for issuing control signals would get simplified.

The instruction coding is the following:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
add rt, ra, rb; sub rt, ra, rb; and rt, ra, rb; or rt, ra, rb; xor rt, ra, rt																															
OPCODE																	Rb					Ra					Rt				
inc rt, ra; dec rt, ra; not rt, ra; move rt, ra; loadr rt, ra; storr rb, ra																															
OPCODE																						Ra					Rt/b				
addi rt, ra, imm; subi rt, ra, imm; andi rt, ra, imm; ori rt, ra, imm; xori rt, ra, imm																															
OPCODE	IMMEDIATE[15:0]																					Ra					Rt				
shl rt, ra, n; shr rt, ra, n; rol rt, ra, n; ror rt, ra, n																															
OPCODE																		n				Ra					Rt				
loadi rt, imm; stori rb, imm																															
OPCODE	IMMEDIATE[15:0]																										Rt/b				
loado rt, ra, off; storo rb, ra, off																															
OPCODE	0	0	0	0	0	0	OFFSET[9:0]															Ra					Rt/b				
brc ra, cond, off																															
OPCODE								OFFSET[8:0]														Ra									
jmp off																															
OPCODE								OFFSET[8:0]																							

The number of instructions determine how many bits would be needed.

$$\text{size of opcode } = \max(\log_2(number\ of\ instructions))$$

The number of registers that we have determine the size of register fields.

$$\text{size of register field } = \max(\log_2(number\ of\ registers))$$

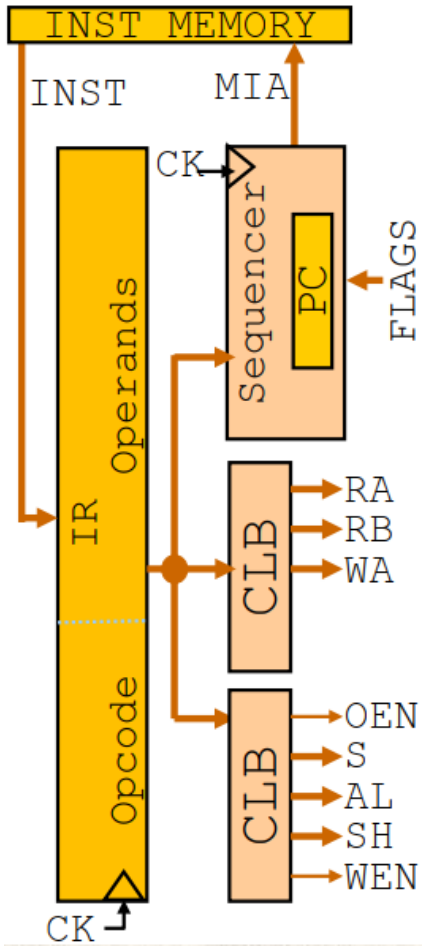
Size of registers determine of immediate field and its equal to it.

Size of registers determine the size of n field for shift instructions.

$$\text{size of n field} = \max(\log_2(Size\ of\ registers\ ))$$

The lodo/storo instructions need to navigate the data addressable space and for doing that the offset should be able to represent 512 places in signed notation which requires 10 bits. The brc/jmp use instruction addressable space and there will be needed to represent 256 instructions in signed notation which requires 9 bits.

The overview intended architecture for control unit:



The intended architecture for sequencer:

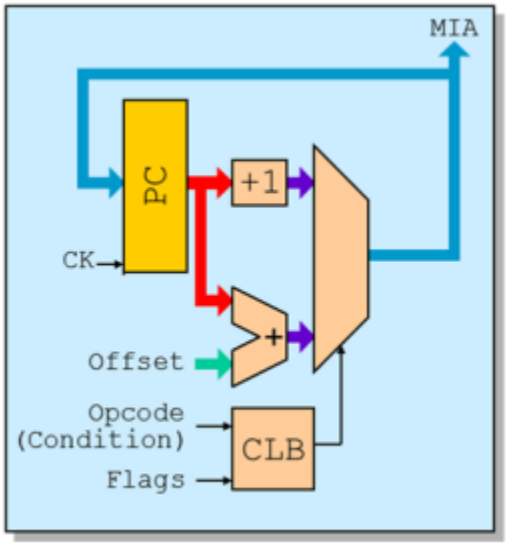


Table of control signals:

	S[1,3,4]	AL	OEN	WEN
add rt, ra, rb	[0, ϕ, 0]	1010	0	1
nop	[ϕ, ϕ, ϕ]	0000	0	0
sub rt, ra, rb	[0, ϕ, 0]	1011	0	1
addi rt, ra, imm	[1, ϕ, 0]	1010	0	1
subi rt, ra, imm	[1, ϕ, 0]	1011	0	1
inc rt, ra	[ϕ, ϕ, 0]	1000	0	1
dec rt, ra	[ϕ, ϕ, 0]	1001	0	1
not rt, ra	[ϕ, ϕ, 0]	0111	0	1
and rt, ra, rb	[0, ϕ, 0]	0100	0	1
or rt, ra, rb	[0, ϕ, 0]	0101	0	1
xor rt, ra, rb	[0, ϕ, 0]	0110	0	1
andi rt, ra, imm	[1, ϕ, 0]	0100	0	1
ori rt, ra, imm	[1, ϕ, 0]	0101	0	1
xori rt, ra, imm	[1, ϕ, 0]	0110	0	1
shl rt, ra, n	[ϕ, ϕ, 0]	1100	0	1
shr rt, ra, n	[ϕ, ϕ, 0]	1101	0	1
rol rt, ra, n	[ϕ, ϕ, 0]	1110	0	1
ror rt, ra, n	[ϕ, ϕ, 0]	1111	0	1
move rt, ra	[ϕ, ϕ, 0]	0000	0	1
loadi rt, imm	[ϕ, 1, 1]	ϕϕϕϕ	0	1
loadr rt, ra	[ϕ, 0, 1]	0000	0	1
loado rt, ra, off	[1, 0, 1]	1010	0	1
stori rb, imm	[ϕ, 1, ϕ]	ϕϕϕϕ	1	0
storr rb, ra	[ϕ, 0, ϕ]	0000	1	0
storo rb, ra, off	[1, 0, ϕ]	1010	1	0
jmp off	[ϕ, ϕ, ϕ]	ϕϕϕϕ	0	0
brc ra, cond, off	[0, ϕ, ϕ]	0101	0	0

VHDL Code for control unit:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.DigEng.ALL;
use IEEE.NUMERIC_STD.ALL;

-- In this module the control logic for CPU is defined
-- The control logic fetches the instruction from memory and puts
-- it in Instruction Register, then based on the instruction
-- it issues the control signals needed for data path unit,
-- It also contains a unit called sequencer, which calculates the address of
-- next instruction based on the current instruction and Flags from datapath

entity control_logic is
    port(clk          : in STD_LOGIC;
         rst          : in STD_LOGIC;
         flags        : in STD_LOGIC_VECTOR(6 downto 0); -- The flags that
                                                         -- result from ALU
                                                         -- operations
         -- The instruction that comes from memory
         INST         : in STD_LOGIC_VECTOR (31 downto 0);
         AL           : out STD_LOGIC_VECTOR(3 downto 0);-- This signal
                                                         -- determines which
                                                         -- operation is done
                                                         -- by ALU
         -- This signal determines For How many positions we are shifting
         SH           : out STD_LOGIC_VECTOR (3 downto 0);
         s            : out STD_LOGIC_VECTOR (2 downto 0);-- The control logic
                                                          -- for muxes
         -- The immediate value which can get selected for the input of one
         -- of ALU's input
         Imm          : out STD_LOGIC_VECTOR (15 downto 0);
         -- The immediate value which can get selected as an input address
         -- for memory
         MA           : out STD_LOGIC_VECTOR (15 downto 0);
         -- the signal which is gonna get decoded and select which register
         -- puts its data on the bus A
         RA           : out STD_LOGIC_VECTOR(4 downto 0);
         -- the signal which is gonna get decoded and select which register
         -- puts its data on the bus B
         RB           : out STD_LOGIC_VECTOR(4 downto 0);
         -- the signal which is gonna get decoded and select which register
         -- is gonna get written to
         WA           : out STD_LOGIC_VECTOR(4 downto 0);
         WEN          : out STD_LOGIC;                   -- the write enable
                                                         -- signal for writing
                                                         -- into registers
                                                         -- It determines if
                                                         -- we have any
                                                         -- writing to do or
                                                         -- not
```

```
        OEN         : out STD_LOGIC;                    -- The enable signal
                                                        -- that enables
                                                        -- writing data to
                                                        -- memory/peripherals
                                                        -- from datapath

        -- The address of next instruction going to memory
        MIA         : out STD_LOGIC_VECTOR(7 downto 0)

        );

end control_logic;

architecture Behavioral of control_logic is

signal  IR_o  : STD_LOGIC_VECTOR( 31 downto 0); -- The output of
                                                 -- instruction register
signal  PC_o  : STD_LOGIC_VECTOR( 7 downto 0); -- The output of
                                                -- PC register
signal  MIA_int : STD_LOGIC_VECTOR(7 downto 0);

signal  PC_o_sum : STD_LOGIC_VECTOR(8 downto 0);-- The 9-bit result of
                                                 -- PC and offset sum

begin

-- This module contains the behavioral description
-- of a D-type register with variable size
-- It is consisted of a sequential circuit
-- This instance is used as instruction register
    instruction_reg: entity work.REG_NBIT
        generic map(size => 32)
        port map ( clk => clk,
                   rst => rst,
                   WEN => '1',      -- Write enable signal for register
                   Q   => IR_o,     -- The output of register
                   D   => INST);    -- The data input to the register

-- This module contains the behavioral description
-- of a D-type register with variable size
-- It is consisted of a sequential circuit
-- This instance is used as program counter
    program_counter: entity work.REG_NBIT
       generic map(size => 8)
       port map ( clk => clk,
                  rst => rst,
                  WEN => '1',      -- Write enable signal for register
                  Q   => PC_o,     -- The output of register
                  D   => MIA_INT); -- The data input to the register
```

```
-- Here the control signals which only depend on opcode are generated
-- The muxes are designed based on the table of control signals in
-- document
    WEN <= '0' when ((IR_o(31 downto 26) = "000000") or
                     (IR_o(31 downto 26) = "100101") or
                     (IR_o(31 downto 26) = "100110") or
                     (IR_o(31 downto 26) = "100111") or
                     (IR_o(31 downto 30) = "11"))   else
           '1';

    OEN <= '1' when ((IR_o(31 downto 26) = "100101") or
                     (IR_o(31 downto 26) = "100110") or
                     (IR_o(31 downto 26) = "100111"))else
           '0';

    with IR_o(31 downto 26) select
    AL <= "1010"    when "000100",
          "1010"    when "000110",
          "1010"    when "100011",
          "1010"    when "100111",
          "1011"    when "000101",
          "1011"    when "000111",
          "1000"    when "001000",
          "1001"    when "001001",
          "0111"    when "010000",
          "0100"    when "010001",
          "0100"    when "010100",
          "0110"    when "010011",
          "1100"    when "011000",
          "1101"    when "011001",
          "1110"    when "011010",
          "1111"    when "011011",
          "0000"    when "000000",
          "0000"    when "100000",
          "0000"    when "100010",
          "0000"    when "100110",
          "0101"    when others;

    s(0) <= '1'    when ((IR_o(31 downto 26) = "000110") or
                         (IR_o(31 downto 26) = "000111") or
                         (IR_o(31 downto 26) = "010100") or
                         (IR_o(31 downto 26) = "010101") or
                         (IR_o(31 downto 26) = "010110") or
                         (IR_o(31 downto 26) = "100011") or
                         (IR_o(31 downto 26) = "100111")) else
            '0';

    s(1) <= '1'     when ((IR_o(31 downto 26) = "100001") or
                          (IR_o(31 downto 26) = "100101"))else
            '0';

    s(2) <= '1'     when ((IR_o(31 downto 26) = "100001") or
                          (IR_o(31 downto 26) = "100010") or
                          (IR_o(31 downto 26) = "100011"))else
            '0';
```

```
-- Here the control signals which only depend on operand and opcode are
-- generated,
-- these signals are generated based on the instruction coding for each
-- instruction
    RA  <= IR_o(9 downto 5);

    RB  <= IR_o(4 downto 0) when (IR_o(31)= '1') else
           IR_o(14 downto 10);

    WA  <= IR_o(4 downto 0);

    SH  <= IR_o(13 downto 10);

    Imm <= IR_o(25 downto 10);

    MA  <= IR_o(25 downto 10);

-- Here the sequencer is defined, the address of next instruction
-- is calculated based on the opcodes and flags, The only sequential
-- part is the PC register

    -- The content of PC is an unsigned number which indicates a memory
    -- address but for being able to do jumps/branched we
    -- sometimes need to decrement the Pc so here a sign bit is concatenated
    -- to PC output and summation is done in signed number notation.

PC_o_sum <= std_logic_vector((signed('0'&PC_o)+signed(IR_o(18 downto 10))));

    MIA_int <= PC_o_sum(7 downto 0) when
                (((IR_o(31 downto 26) = "110000") and (flags(0) = '1'))or
                ((IR_o(31 downto 26) = "110001") and (flags(1) = '1'))or
                ((IR_o(31 downto 26) = "110010") and (flags(2) = '1'))or
                ((IR_o(31 downto 26) = "110011") and (flags(3) = '1'))or
                ((IR_o(31 downto 26) = "110100") and (flags(4) = '1'))or
                ((IR_o(31 downto 26) = "110101") and (flags(5) = '1'))or
                ((IR_o(31 downto 26) = "110110") and (flags(6) = '1'))or
                ((IR_o(31 downto 26) = "110111"))) else
            std_logic_vector(unsigned(PC_o) + 1);

MIA <= MIA_int;


end Behavioral;
```

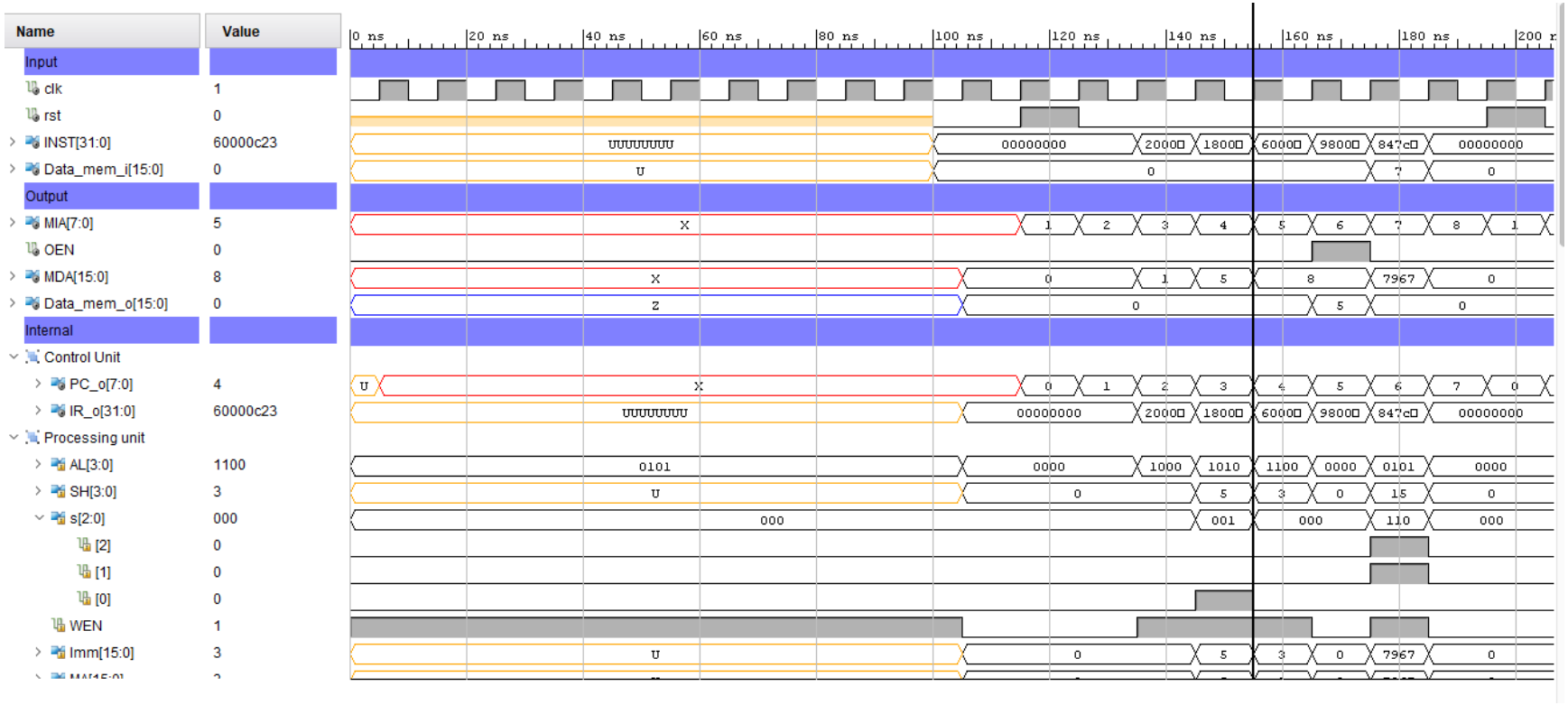
VHDL Code for Testbench:

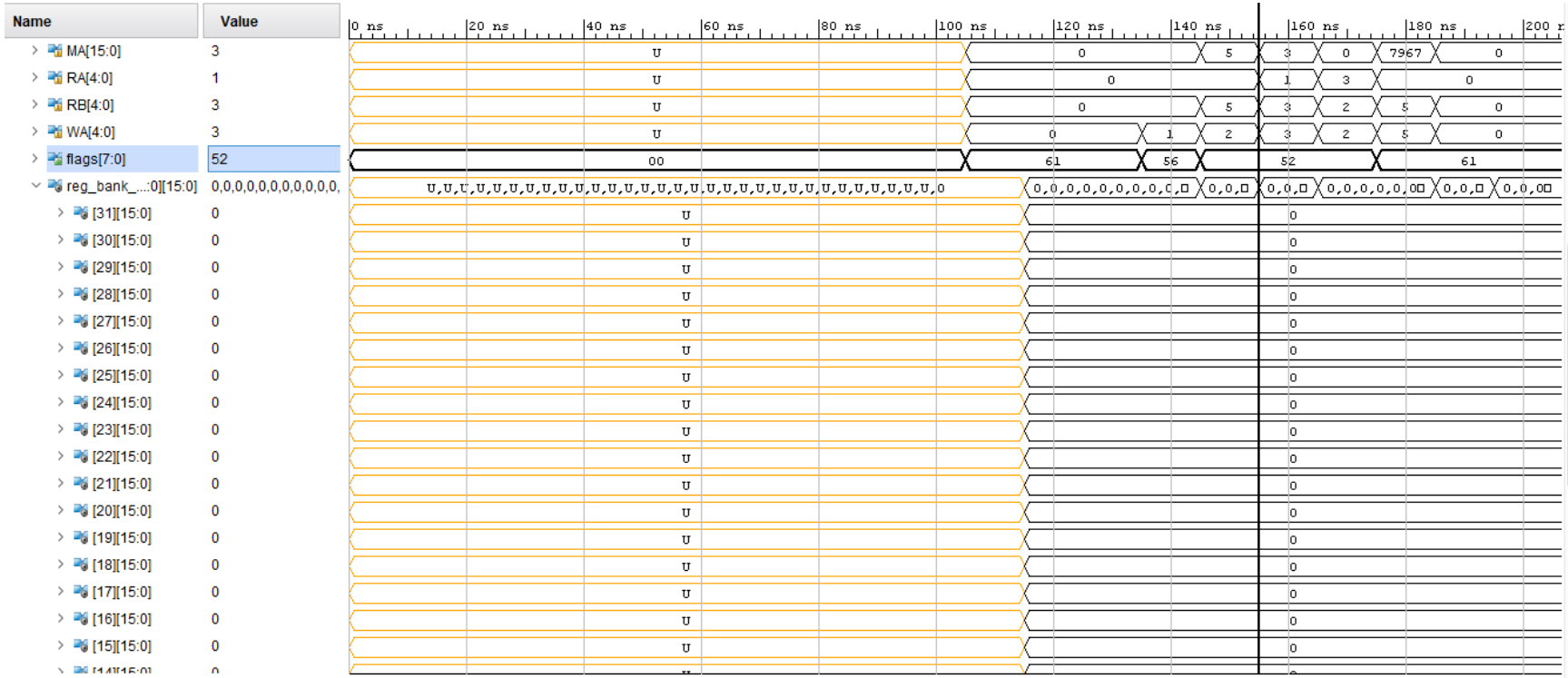
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.DigEng.ALL;
use IEEE.NUMERIC_STD.ALL;


entity control_datapath_tb is

end control_datapath_tb;

architecture Behavioral of control_datapath_tb is

constant clk_period : time := 10 ns;

signal clk          :  std_logic;
signal rst          :  std_logic;
signal INST         :  std_logic_vector(31 downto 0);
signal Data_mem_i   :  STD_LOGIC_VECTOR(15 downto 0);
signal Data_mem_o   :  STD_LOGIC_VECTOR(15 downto 0);
signal MDA          :  STD_LOGIC_VECTOR(15 downto 0);
signal OEN          :  STD_LOGIC;
signal MIA          :  STD_LOGIC_VECTOR(7 downto 0);

type test_vector is record
        rst : STD_LOGIC;
        INST :  STD_LOGIC_VECTOR(31 downto 0);
        Data_mem_i   :  STD_LOGIC_VECTOR(15 downto 0);
        Data_mem_o   :  STD_LOGIC_VECTOR(15 downto 0);
        MDA          :  STD_LOGIC_VECTOR(15 downto 0);
        OEN          :  STD_LOGIC;
        MIA          :  STD_LOGIC_VECTOR(7 downto 0);
end record;

type test_vector_array is array
        (natural range <>) of test_vector;

-- For the partial verification of the current module, few instructions were
-- encoded and given to cpu as the input and then the output was monitored
-- The output is checked one clock cycle after the input was given to module
constant test_vectors : test_vector_array := (
     -- rst, INST,      Data_mem_i, Data_mem_o, MDA, OEN, MIA
     -- reseting the module
     ('1', x"00000000", x"0000",    x"0000", x"0000", '0', x"01"),--#0
     ('0', x"00000000", x"0000",    x"0000", x"0000", '0', x"02"),--#1
     -- inc R1, R0;
     ('0', x"20000001", x"0000",    x"0000", x"0001", '0', x"03"),--#2
     -- addi R2, R0, 005
     ('0', x"18001402", x"0000",    x"0000", x"0005", '0', x"04"),--#3
     -- shl R3, R1, 3,
     ('0', x"60000c23", x"0000",    x"0000", x"0008", '0', x"05"),--#4
     -- storr R2, R3
     ('0', x"98000062", x"0000",    x"0005", x"0008", '1', x"06"),--#5
     -- loadi R5, 1f1f
     ('0', x"847C7C05", x"0007",    x"0000", x"1f1f", '0', x"07"),--#6
```

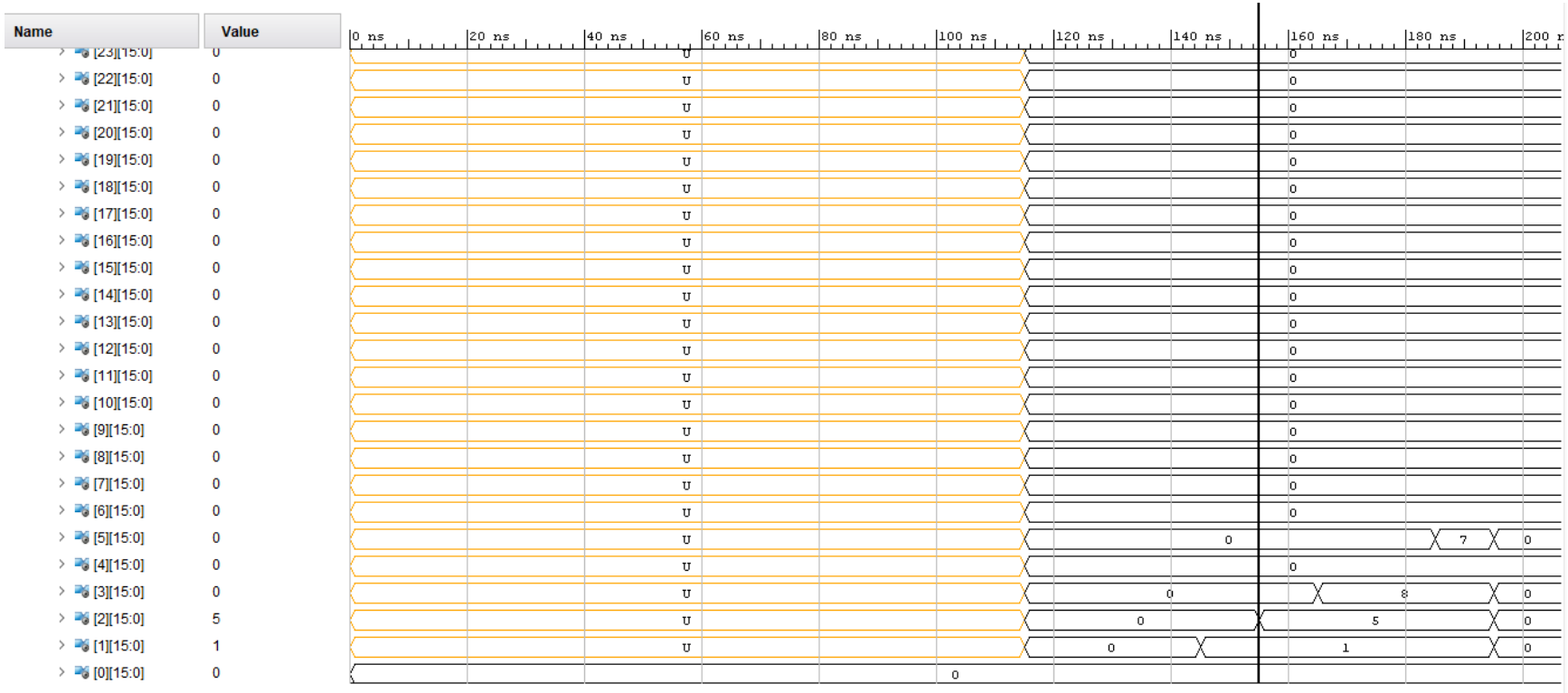
```
     -- NOP
     ('0', x"00000000", x"0000",    x"0000", x"0000", '0', x"08"),--#7
     -- Middle reset
     ('1', x"00000000", x"0000",    x"0000", x"0000", '0', x"01"),--#8
     ('0', x"00000000", x"0000",    x"0000", x"0000", '0', x"02"),--#9
     -- inc R1, R0;
     ('0', x"20000001", x"0000",    x"0000", x"0001", '0', x"03"),--#10
     -- addi R2, R0, 005
     ('0', x"18001402", x"0000",    x"0000", x"0005", '0', x"04"));--#11


begin
UUT: entity work.control_datapath
    port map(clk => clk,
             rst => rst,
             INST => INST,
             Data_mem_i => Data_mem_i,
             Data_mem_o => Data_mem_o,
             MDA => MDA,
             OEN => OEN,
             MIA => MIA
             );

-- Clock process
clk_process :process
begin
  clk <= '0';
  wait for clk_period/2;
  clk <= '1';
  wait for clk_period/2;
end process;

test_process: process
begin

    -- wait 100 ns for global reset to finish
    wait for 100ns;

    wait until falling_edge(clk); -- Clock Synchronization

    -- initializing inputs
    rst <= '0';
    INST <= x"00000000";
    Data_mem_i <= x"0000";
    wait for 1*clk_period;

    for i in test_vectors' range loop
       rst <= test_vectors(i).rst;
       INST <= test_vectors(i).INST;
       Data_mem_i <= test_vectors(i).Data_mem_i;

       wait for 1*clk_period;
       assert ((Data_mem_o = test_vectors(i).Data_mem_o) and
                      (MDA = test_vectors(i).MDA) and
                      (OEN = test_vectors(i).OEN) and
                      (MIA = test_vectors(i).MIA))
```

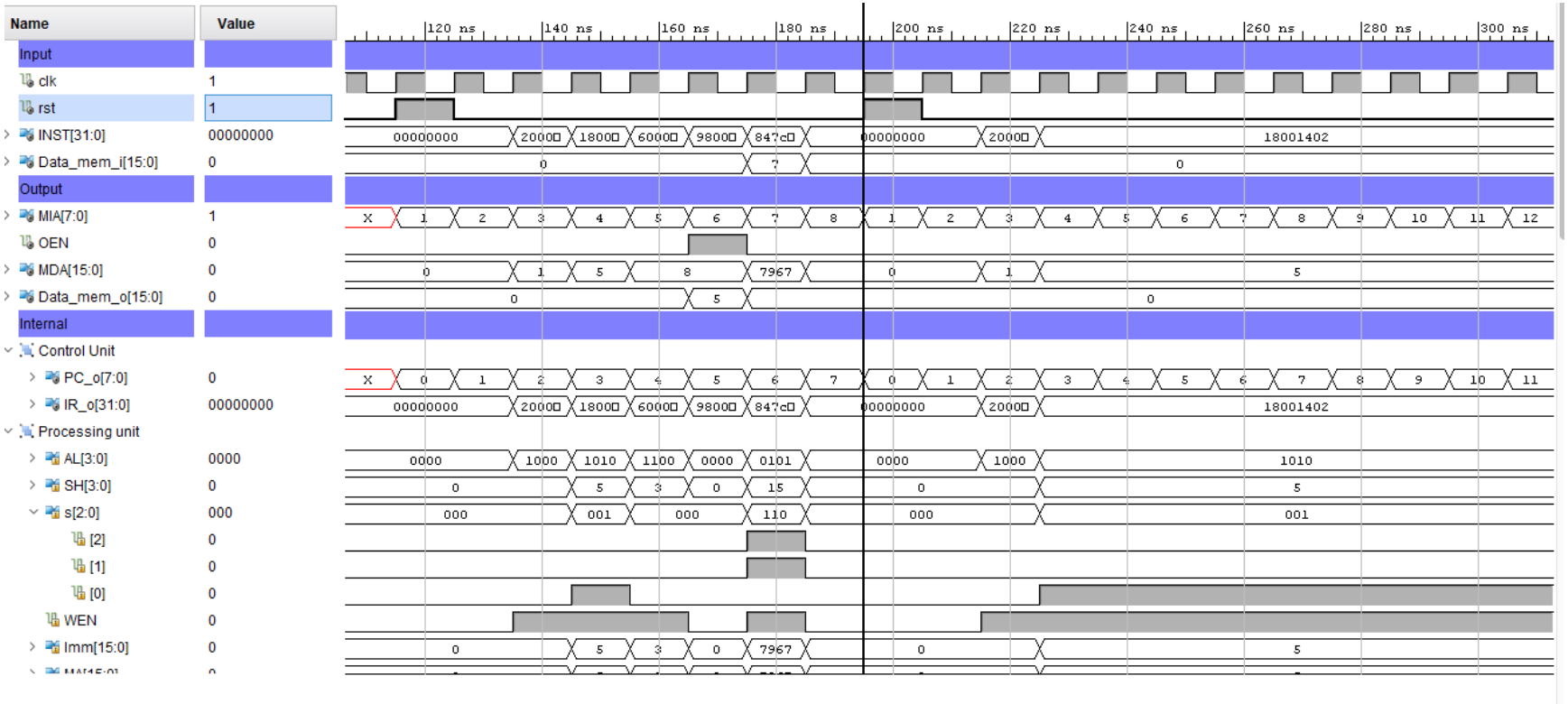
```
        report "Test vector " &
              integer'image(i) &
              " failed for inputs rst = " &
              std_logic'image(rst) &
              " and INST = " &
              integer'image(to_integer(unsigned(INST))) &
              " and Data_mem_i = " &
              integer'image(to_integer(unsigned(Data_mem_i))) &
              ". Expected Data_mem_o = " &
              integer'image(to_integer(unsigned(test_vectors(i).Data_mem_o)))&
              " and MDA =" &
              integer'image(to_integer(unsigned(test_vectors(i).MDA))) &
              " and OEN =" &
              std_logic'image(test_vectors(i).OEN) &
              " and MIA =" &
              integer'image(to_integer(unsigned(test_vectors(i).MIA))) &
              "; observed Data_mem_o = " &
              integer'image(to_integer(unsigned(Data_mem_o))) &
              " and MDA =" &
              integer'image(to_integer(unsigned(MDA))) &
              " and OEN =" &
              std_logic'image(OEN) &
              " and MIA =" &
              integer'image(to_integer(unsigned(MIA)))
        severity error; -- to stop the simulation

        report "The output corresponds to expectation at test vector " &
               integer'image(i)
        severity note;


    end loop;
    wait;
end process;

end Behavioral;
```

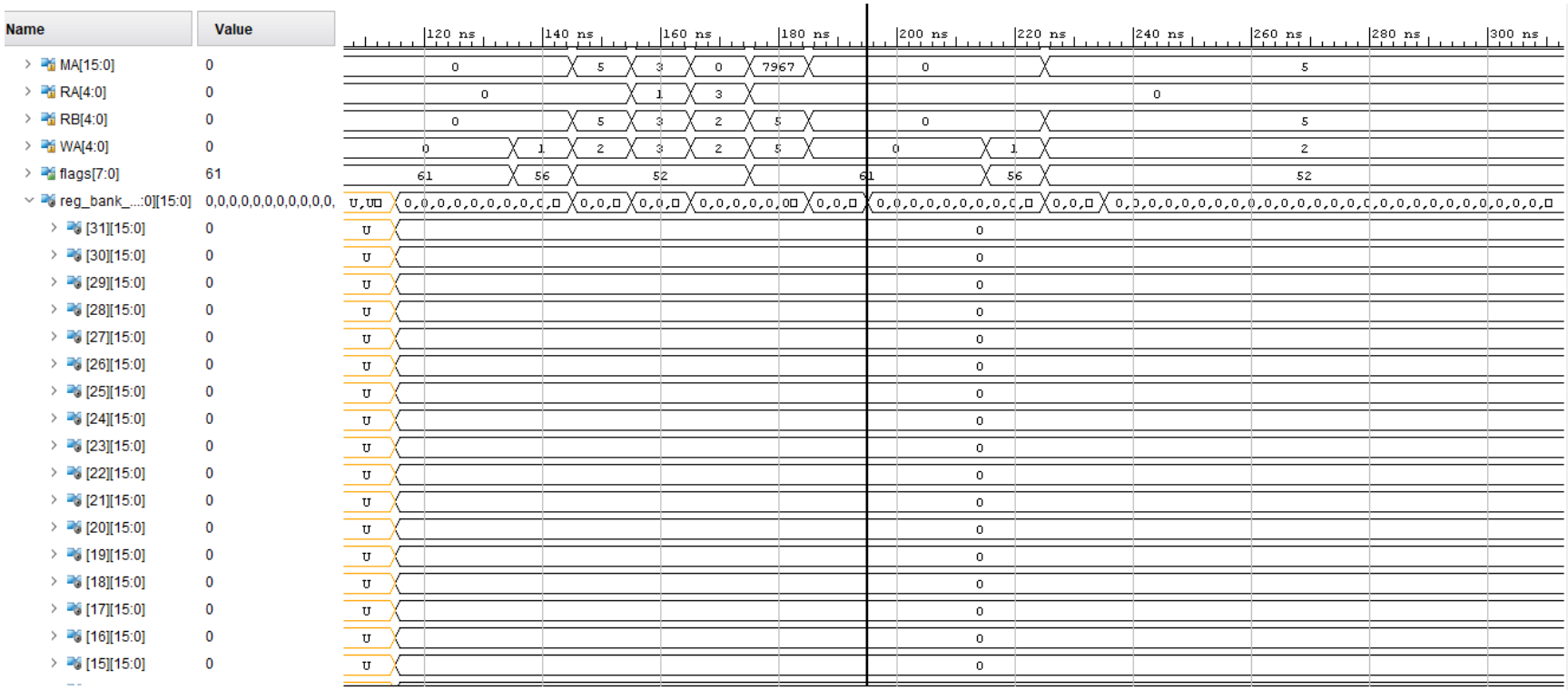
Simulation screenshots:

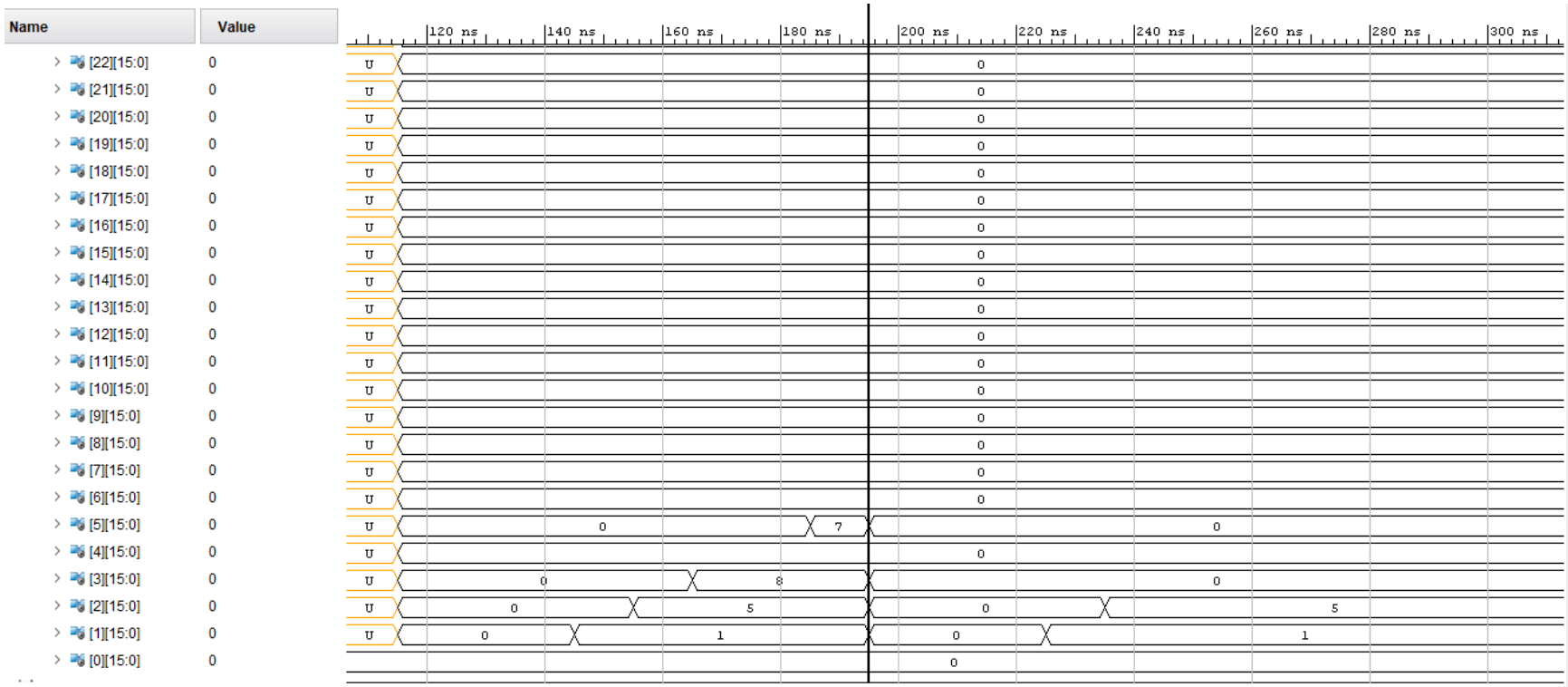



Name
Value
MA[15:0] 3
RA[4:0] 1
RB[4:0] 3
WA[4:0] 3
flags[7:0] 52
reg_bank_...:0][15:0] 0,0,0,0,0,0,0,0,0,0,
[31][15:0] 0
[30][15:0] 0
[29][15:0] 0
[28][15:0] 0
[27][15:0] 0
[26][15:0] 0
[25][15:0] 0
[24][15:0] 0
[23][15:0] 0
[22][15:0] 0
[21][15:0] 0
[20][15:0] 0
[19][15:0] 0
[18][15:0] 0
[17][15:0] 0
[16][15:0] 0
[15][15:0] 0
0 ns
20 ns
40 ns
60 ns
80 ns
100 ns
120 ns
140 ns
160 ns
180 ns


Name
Value
0 ns
20 ns
40 ns
60 ns
80 ns
100 ns
120 ns
140 ns
160 ns
180 ns
[23][15:0] 0
[22][15:0] 0
[21][15:0] 0
[20][15:0] 0
[19][15:0] 0
[18][15:0] 0
[17][15:0] 0
[16][15:0] 0
[15][15:0] 0
[14][15:0] 0
[13][15:0] 0
[12][15:0] 0
[11][15:0] 0
[10][15:0] 0
[9][15:0] 0
[8][15:0] 0
[7][15:0] 0
[6][15:0] 0
[5][15:0] 0
[4][15:0] 0
[3][15:0] 0
[2][15:0] 5
[1][15:0] 1
[0][15:0] 0


Name
Value
120 ns
140 ns
160 ns
180 ns
200 ns
220 ns
240 ns
260 ns
280 ns
300 ns
Input
clk 1
rst 1
INST[31:0] 00000000
Data_mem_i[15:0] 0
Output
MIA[7:0] 1
OEN 0
MDA[15:0] 0
Data_mem_o[15:0] 0
Internal
Control Unit
PC_o[7:0] 0
IR_o[31:0] 00000000
Processing unit
AL[3:0] 0000
SH[3:0] 0
s[2:0] 000
[2] 0
[1] 0
[0] 0
WEN 0
Imm[15:0] 0


Name
Value
MA[15:0] 0
RA[4:0] 0
RB[4:0] 0
WA[4:0] 0
flags[7:0] 61
reg_bank_...:0][15:0] 0,0,0,0,0,0,0,0,0,0,0,
[31][15:0] 0
[30][15:0] 0
[29][15:0] 0
[28][15:0] 0
[27][15:0] 0
[26][15:0] 0
[25][15:0] 0
[24][15:0] 0
[23][15:0] 0
[22][15:0] 0
[21][15:0] 0
[20][15:0] 0
[19][15:0] 0
[18][15:0] 0
[17][15:0] 0
[16][15:0] 0
[15][15:0] 0
120 ns
140 ns
160 ns
180 ns
200 ns
220 ns
240 ns
260 ns
280 ns
300 ns


Name
Value
120 ns
140 ns
160 ns
180 ns
200 ns
220 ns
240 ns
260 ns
280 ns
300 ns
[22][15:0] 0
[21][15:0] 0
[20][15:0] 0
[19][15:0] 0
[18][15:0] 0
[17][15:0] 0
[16][15:0] 0
[15][15:0] 0
[14][15:0] 0
[13][15:0] 0
[12][15:0] 0
[11][15:0] 0
[10][15:0] 0
[9][15:0] 0
[8][15:0] 0
[7][15:0] 0
[6][15:0] 0
[5][15:0] 0
[4][15:0] 0
[3][15:0] 0
[2][15:0] 0
[1][15:0] 0
[0][15:0] 0

Simulation screenshots from console:

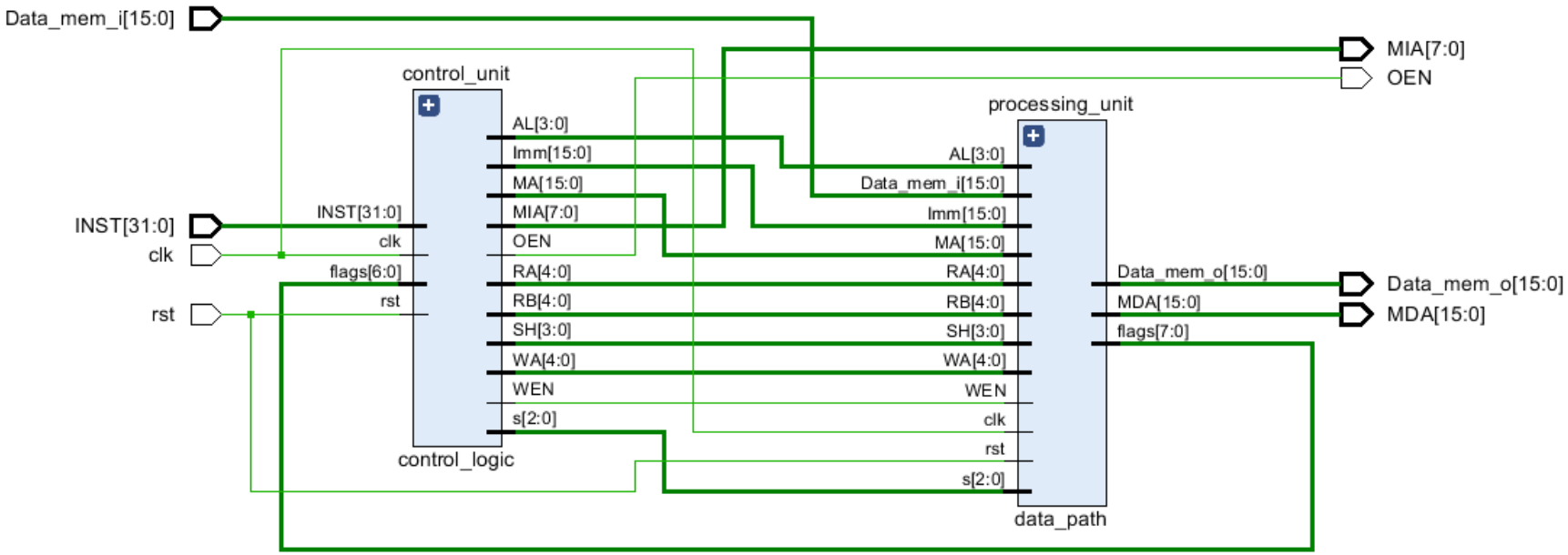
```
Note: The output corresponds to expectation at test vector 0
Time: 125 ns  Iteration: 0  Process: /control_datapath_tb/test_process  File: E:/York/Digital_Design/Documents/Github/datapath_standalone/datapath_standalone.srcs/sim_1/new/control_da
Note: The output corresponds to expectation at test vector 1
Time: 135 ns  Iteration: 0  Process: /control_datapath_tb/test_process  File: E:/York/Digital_Design/Documents/Github/datapath_standalone/datapath_standalone.srcs/sim_1/new/control_da
Note: The output corresponds to expectation at test vector 2
Time: 145 ns  Iteration: 0  Process: /control_datapath_tb/test_process  File: E:/York/Digital_Design/Documents/Github/datapath_standalone/datapath_standalone.srcs/sim_1/new/control_da
Note: The output corresponds to expectation at test vector 3
Time: 155 ns  Iteration: 0  Process: /control_datapath_tb/test_process  File: E:/York/Digital_Design/Documents/Github/datapath_standalone/datapath_standalone.srcs/sim_1/new/control_da
Note: The output corresponds to expectation at test vector 4
Time: 165 ns  Iteration: 0  Process: /control_datapath_tb/test_process  File: E:/York/Digital_Design/Documents/Github/datapath_standalone/datapath_standalone.srcs/sim_1/new/control_da
Note: The output corresponds to expectation at test vector 5
Time: 175 ns  Iteration: 0  Process: /control_datapath_tb/test_process  File: E:/York/Digital_Design/Documents/Github/datapath_standalone/datapath_standalone.srcs/sim_1/new/control_da
Note: The output corresponds to expectation at test vector 6
Time: 185 ns  Iteration: 0  Process: /control_datapath_tb/test_process  File: E:/York/Digital_Design/Documents/Github/datapath_standalone/datapath_standalone.srcs/sim_1/new/control_da
Note: The output corresponds to expectation at test vector 7
Time: 195 ns  Iteration: 0  Process: /control_datapath_tb/test_process  File: E:/York/Digital_Design/Documents/Github/datapath_standalone/datapath_standalone.srcs/sim_1/new/control_da
Note: The output corresponds to expectation at test vector 8
Time: 205 ns  Iteration: 0  Process: /control_datapath_tb/test_process  File: E:/York/Digital_Design/Documents/Github/datapath_standalone/datapath_standalone.srcs/sim_1/new/control_da
Note: The output corresponds to expectation at test vector 9
Time: 215 ns  Iteration: 0  Process: /control_datapath_tb/test_process  File: E:/York/Digital_Design/Documents/Github/datapath_standalone/datapath_standalone.srcs/sim_1/new/control_da
Note: The output corresponds to expectation at test vector 10
Time: 225 ns  Iteration: 0  Process: /control_datapath_tb/test_process  File: E:/York/Digital_Design/Documents/Github/datapath_standalone/datapath_standalone.srcs/sim_1/new/control_da
Note: The output corresponds to expectation at test vector 11
Time: 235 ns  Iteration: 0  Process: /control_datapath_tb/test_process  File: E:/York/Digital_Design/Documents/Github/datapath_standalone/datapath_standalone.srcs/sim_1/new/control_da
relaunch_sim: Time (s): cpu = 00:00:02 ; elapsed = 00:00:06 . Memory (MB): peak = 734.383 ; gain = 10.426
save_wave_config {E:/York/Digital_Design/Documents/Github/datapath_standalone/datapath_tb_behav.wcfg}
save_wave_config {E:/York/Digital_Design/Documents/Github/datapath_standalone/datapath_tb_behav.wcfg}
```

```
---------------------------------------------------------------------------------
Start RTL Component Statistics
---------------------------------------------------------------------------------
Detailed RTL Component Info :
+---Adders :
          2 Input     16 Bit       Adders := 1
          3 Input     16 Bit       Adders := 1
          2 Input      8 Bit       Adders := 1
+---XORs :
          2 Input     16 Bit         XORs := 1
+---Registers :
                      32 Bit    Registers := 1
                      16 Bit    Registers := 31
                       8 Bit    Registers := 1
+---Muxes :
         14 Input     16 Bit        Muxes := 2
          2 Input     16 Bit        Muxes := 67
          2 Input      8 Bit        Muxes := 1
          2 Input      5 Bit        Muxes := 1
         21 Input      4 Bit        Muxes := 1
          2 Input      1 Bit        Muxes := 2
---------------------------------------------------------------------------------
Finished RTL Component Statistics
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
Start RTL Hierarchical Component Statistics
---------------------------------------------------------------------------------
Hierarchical RTL Component report
Module REG_NBIT
Detailed RTL Component Info :
+---Registers :
                      32 Bit    Registers := 1
Module REG_NBIT__parameterized0
Detailed RTL Component Info :
+---Registers :
                       8 Bit    Registers := 1
Module control_logic
Detailed RTL Component Info :
+---Adders :
          2 Input      8 Bit       Adders := 1
+---Muxes :
          2 Input      8 Bit        Muxes := 1
          2 Input      5 Bit        Muxes := 1
         21 Input      4 Bit        Muxes := 1
Module ALU
Detailed RTL Component Info :
+---Adders :
          2 Input     16 Bit       Adders := 1
          3 Input     16 Bit       Adders := 1
+---XORs :
          2 Input     16 Bit         XORs := 1
+---Muxes :
         14 Input     16 Bit        Muxes := 2
          2 Input      1 Bit        Muxes := 2
Module REG_NBIT__parameterized1
Detailed RTL Component Info :
+---Registers :
                      16 Bit    Registers := 1
Module reg_bank
Detailed RTL Component Info :
+---Muxes :
          2 Input     16 Bit        Muxes := 64
```

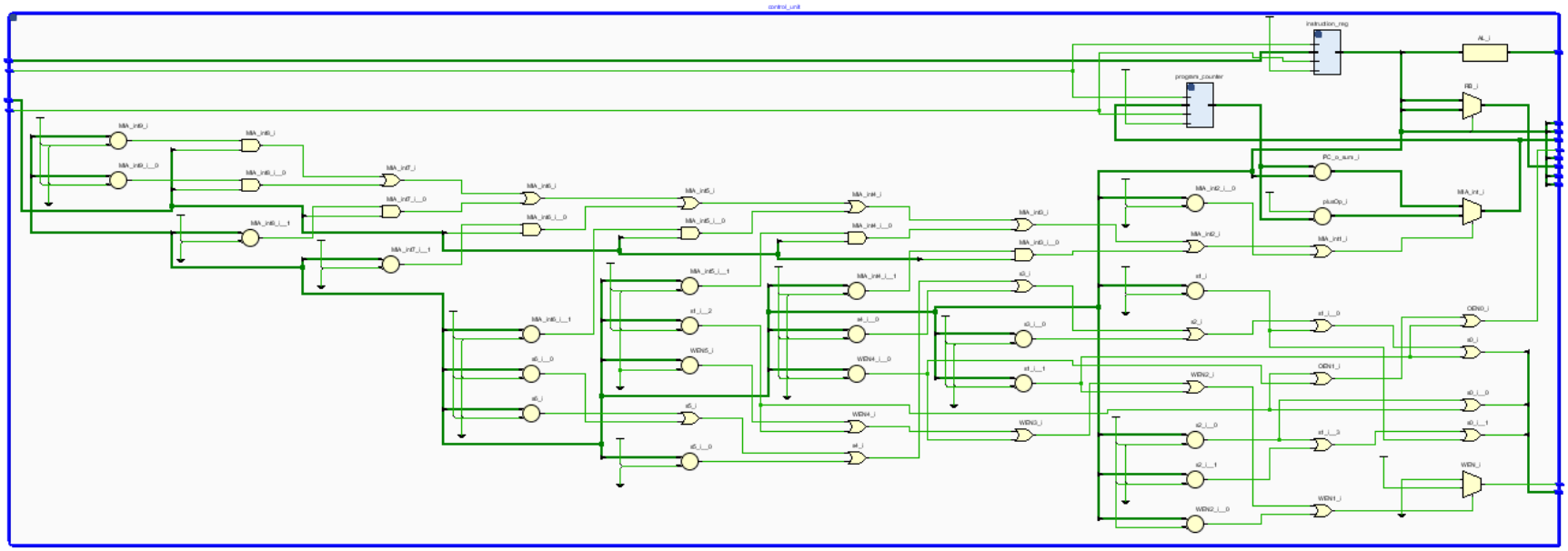
```
Module data_path
Detailed RTL Component Info :
+---Muxes :
          2 Input     16 Bit        Muxes := 3
--------------------------------------------------------------------------------
Finished RTL Hierarchical Component Statistics
--------------------------------------------------------------------------------
```

Schematics screenshots:

Top Module:



Control Unit:



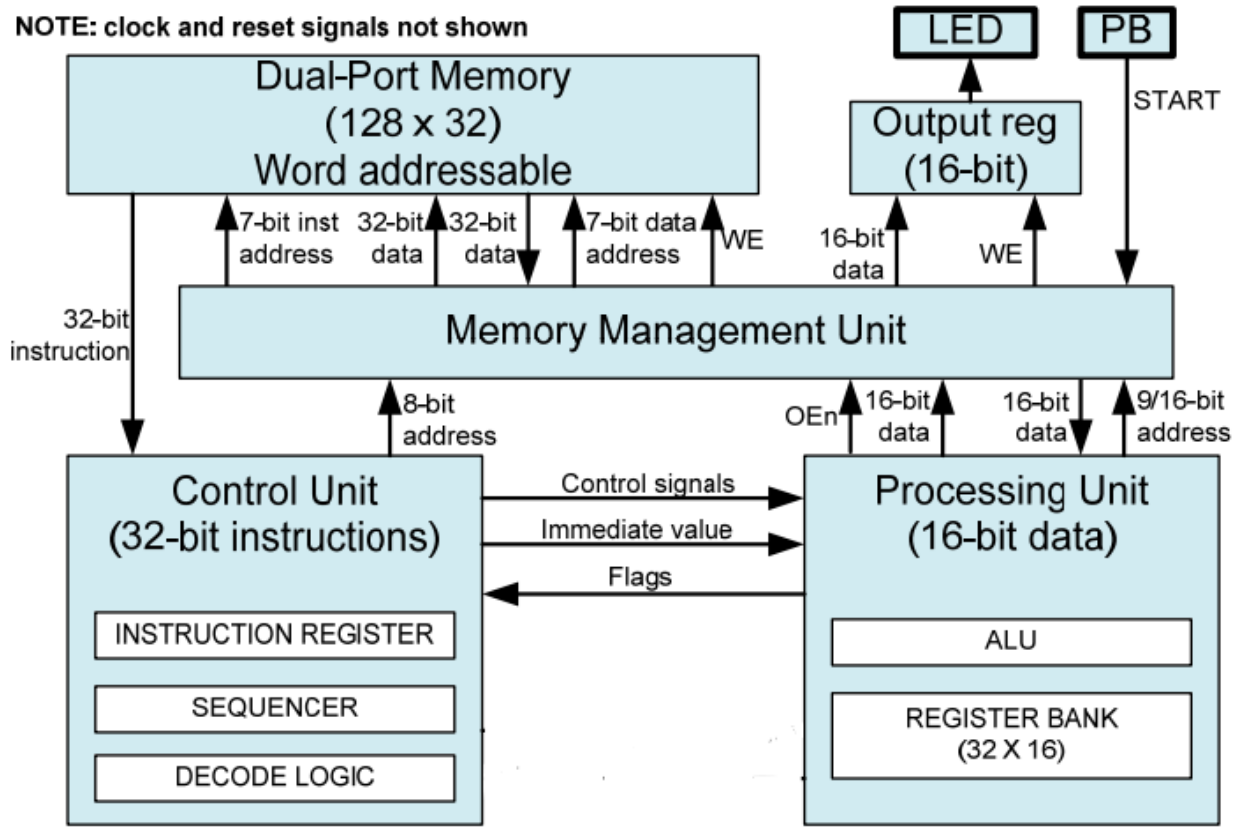
At the top right, the sequencer is placed, using summation in HDL has translated to two adders, beside from PC and IR registers the rest is just combinational logic as intended.

Part V: Memory subsystem and full Integration

The processor sees the memory subsystem as two separate elements: a 256x32 read-only memory for the instructions and a 512x16 read/write memory for the data (both word-addressable). The memory management unit (MMU) will allow the processor to work with a single-write/dual-read RAM.

The MMU will also allow the processor to interface with the external world using two *memorymapped I/O devices* (a pushbutton and an array of LEDs).

It is assumed that the program will not access memory locations outside the memory capacity (except the peripherals), addresses 0 to 63 will be used for the program (instruction memory), while addresses 64 to 127 will be used to store the data (data memory).



In the top module all of the components in architecture will be integrated and the following program will be loaded in the RAM for the purpose of testing the module partially:

MEM	Label	Instruction	Machine Language Binary	Hex
00		nop	00000000000000000000000000000000	00000000
01		loadi r15, h01F0	10000100000001111100000000001111	8407C00F
02		br r15,=0,-1	11000000000001111111110111100000	C007FDE0
03		addi r31, r0, h001F	00011000000000000111110000011111	18007C1F
04		add r1, r0, r0	00010000000000000000000000000001	10000001
05		inc r1, r1	00100000000000000000000000100001	20000021
06	L1	br r31, <0, +5 (L2)	11001100000000000001011111100000	CC0017E0
07		storr r1, r31	10011000000000000000001111100001	980003E1
08		inc r1, r1	00100000000000000000000000100001	20000021
09		dec r31, r31	00100100000000000000001111111111	240003FF
10		jmp -4 (L1)	11011100000001111111000000000000	DC07F000
11	L2	loadi r2, h0010	10000100000000000100000000000010	84004002
12		ori r30, r0, h0004	01010100000000000001000000011110	5400101E
13		loadr r3, r30	10001000000000000000001111000011	880003C3
14		loado r4, r30, 3	10001100000000000000111111000100	8C000FC4
15		xori r29, r0, hFFFF	01011011111111111111110000011101	5BFFFC1D
16		move r7, r0	10000000000000000000000000000111	80000007
17		andi r5, r29, h0010	01010000000000000100001110100101	500043A5
18	L3	andi r6, r4, h0001	01010000000000000000010010000110	50000486
19		br r6, =0, +2 (L4)	11000000000000000000100011000000	C00008C0
20		add r7, r3, r7	00010000000000000001110001100111	10001C67
21	L4	shr r4, r4, 1	01100100000000000000010010000100	64000484
22		shl r3, r3, 1	01100000000000000000010001100011	60000463
23		dec r5, r5	00100100000000000000000010100101	240000A5
24		br r5, ≠0, -6 (L3)	11000100000001111110100010100000	C407E8A0
25		stori r7, 0	10010100000000000000000000000111	94000007
26		rol r7, r7, 9	01101000000000000010010011100111	680024E7
27		ror r7, r7, 3	01101100000000000000110011100111	6C000CE7
28		not r8, r7	01000000000000000000000011101000	400000E8
29		xor r8, r29, r8	01001100000000000010001110101000	4C0023A8
30		sub r9, r8, r7	00010100000000000001110100001001	14001D09
31		subi r10, r9, h0001	00011100000000000000010100101010	1C00052A
32		br r10, >0, +9 (Lx)	11010000000000000010010101000000	D0002540
33		br r10, ≥0, +8 (Lx)	11011000000000000010000101000000	D8002140
34		addi r11, r10, h0002	00011000000000000000100101001011	1800094B
35		br r11, ≤0, +6 (Lx)	11010100000000000001100101100000	D4001960
36		or r12, r7, r11	01001000000000000010110011101100	48002CEC
37		storo r12, r0, 1	10011100000000000000010000001100	9C00040C
38		and r12, r12, r11	01000100000000000010110110001100	44002D8C
39		br r12, =1, +3 (Lok)	11001000000000000000110110000000	C8000D80
40		jmp +0	11011100000000000000000000000000	DC000000
41	Lx	jmp +0	11011100000000000000000000000000	DC000000
42	Lok	stori r7,h01F8	10010100000001111110000000000111	9407E007
43		jmp +0	11011100000000000000000000000000	DC000000

VHDL Code for memory subsystem:

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


-- This module implements the Memory management unit for the CPU, it serves
-- as an interface between the processor and memory/IO, The virtual
-- addresses coming from control and processing unit get transformed to
-- physical addresses suitable for Memory, The required signals for
-- peripherals and memory are also generated
entity MMU is
    Port ( OEn : in STD_LOGIC; -- The enable signal
                               -- that enables the
                               -- data to be written to
                               -- memory/IO from datapath
           IO_Data_i : in STD_LOGIC; -- The input data coming from IO, in
                                     -- this case Its a pushbutton
           MIA_i : in STD_LOGIC_VECTOR (7 downto 0); -- the virtual
                                                     -- instruction
                                                     -- address, coming from
                                                     -- control unit
           MDA_i : in STD_LOGIC_VECTOR (15 downto 0);-- The virtual data
                                                     -- address coming
                                                     -- from processing
                                                     -- unit
           PU_Data_i : in STD_LOGIC_VECTOR (15 downto 0); -- the data which
                                                          -- comes from
                                                          -- processing unit
           Mem_Data_i : in STD_LOGIC_VECTOR (31 downto 0);-- the data which
                                                          -- comes
                                                          -- from memory
           PU_Data_o : out STD_LOGIC_VECTOR (15 downto 0);-- the data which
                                                          -- goes to
                                                          -- processing unit
           MIA_o : out STD_LOGIC_VECTOR (6 downto 0);-- the physical
                                                     -- instruction
                                                     -- address, going to
                                                     -- memory
           MDA_o : out STD_LOGIC_VECTOR (6 downto 0);-- the physical data
                                                     -- address, going to
                                                     -- memory
           Mem_Data_o : out STD_LOGIC_VECTOR (31 downto 0);-- the data that
                                                           -- goes to
                                                           -- memory
           WE_Mem : out STD_LOGIC; -- write enable signal for memory
           WE_IO : out STD_LOGIC;  -- write enable signal for I/O
                                   -- register (LEDs)
           IO_Data_o : out STD_LOGIC_VECTOR (15 downto 0)); -- The data
                                                            -- output to
                                                            -- IO
                                                            -- registers
end MMU;
```

```
architecture Behavioral of MMU is

signal Ext_IO_Data_i : STD_LOGIC_VECTOR(15 downto 0); -- The input data
                                                      -- coming from IO,
                                                      -- in this case
                                                      -- a pushbutton,
                                                      -- extended
                                                      -- to 16 bits

begin

-- Extending the input data from pushbutton to 16 bits, compatible with CPU
Ext_IO_Data_i(15 downto 1) <= (others => '0') ;
Ext_IO_Data_i(0) <= IO_Data_i;

-- First two MSBs of MIA_i determine the page number, based on design we know
-- we only are going to use page 0 and it is loaded in memory in the physical
-- base address of 0x00, so the Fisrt two MSBs of MIA_i is replaced by 0
MIA_o <= '0' & MIA_i(5 downto 0);

-- bits number 7 & 8 determine the page number for data, based on design we
-- know we only are going to use page 0 and it is loaded in memory
-- in the physical base address of 0x40,
-- first LSB bit of virtual address is going to select which half of data
-- returned by memory is our desired one and it doesn't affect the
-- the physical address generation
MDA_o <= '1' & MDA_i(6 downto 1);

-- since memory word differs from PU word, the first bit of virtual address
-- is going to select
-- which half of data returned by memory is our desired one
-- The x"01F0" is the address for Pushbutton IO register
PU_Data_o <= Ext_IO_Data_i           when (MDA_i = x"01F0") else
             Mem_Data_i(15 downto 0) when (MDA_i(0) = '0')  else
             Mem_Data_i(31 downto 16)when (MDA_i(0) = '1')  else
             (others => 'U');

-- The bit eight of the MDA determines if we are writing to memory or
-- peripheral and the write enable signals are assigned according to that
WE_Mem <= OEn when (MDA_i(8) = '0') else
          '0' when (MDA_i(8) = '1') else
          'U';

WE_IO <= OEn when (MDA_i(8) = '1') else
         '0' when (MDA_i(8) = '0') else
         'U';

IO_Data_o <= PU_Data_i;
```

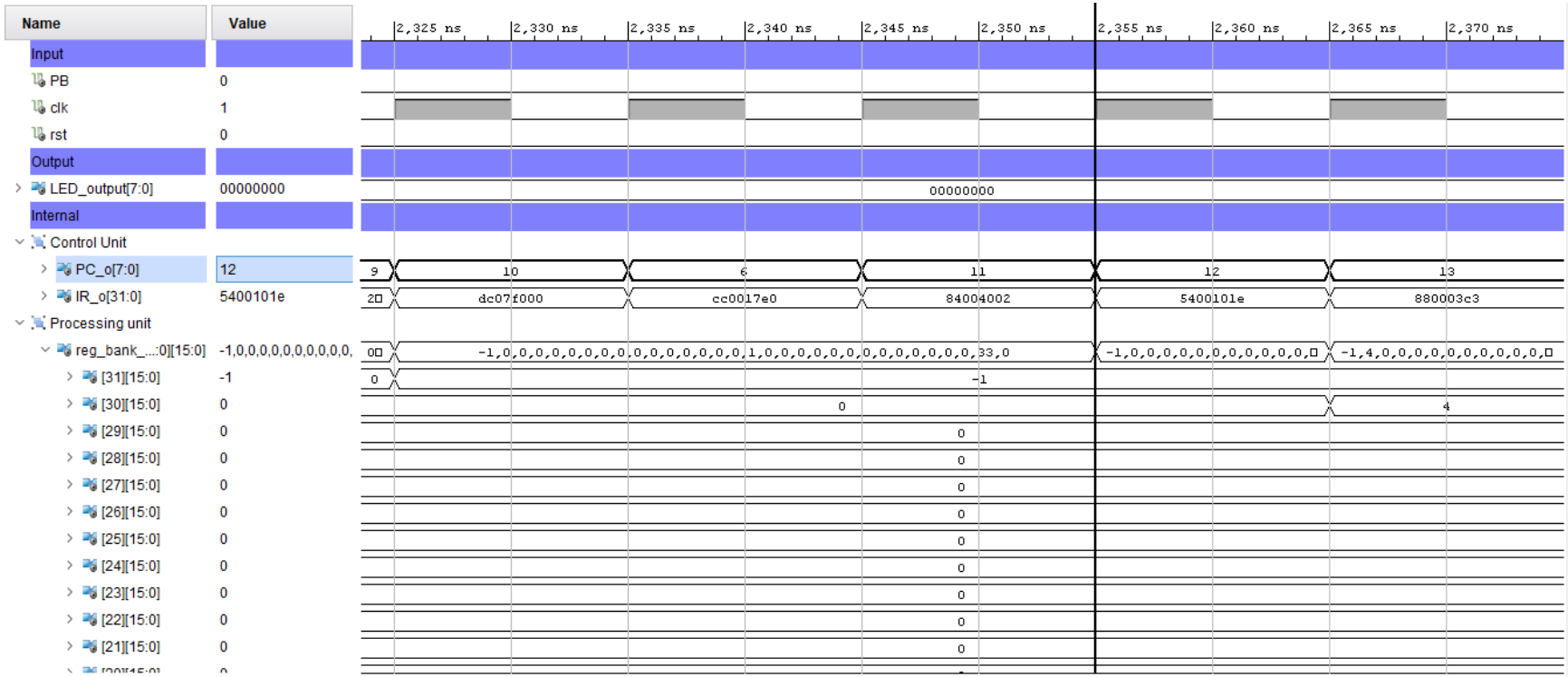
```
-- Whenever there is a write, the memory reads the current value of that
-- address, first LSB of virtual address is going to select
-- which half of data should be replaced and written to
Mem_data_o <= (Mem_data_i(31 downto 16) & PU_data_i) when (MDA_i(0) = '0')
               else
              (PU_data_i & Mem_data_i(15 downto 0)) when (MDA_i(0) = '1')
               else
              (others => 'U');

end Behavioral;
```

VHDL code for testbench:

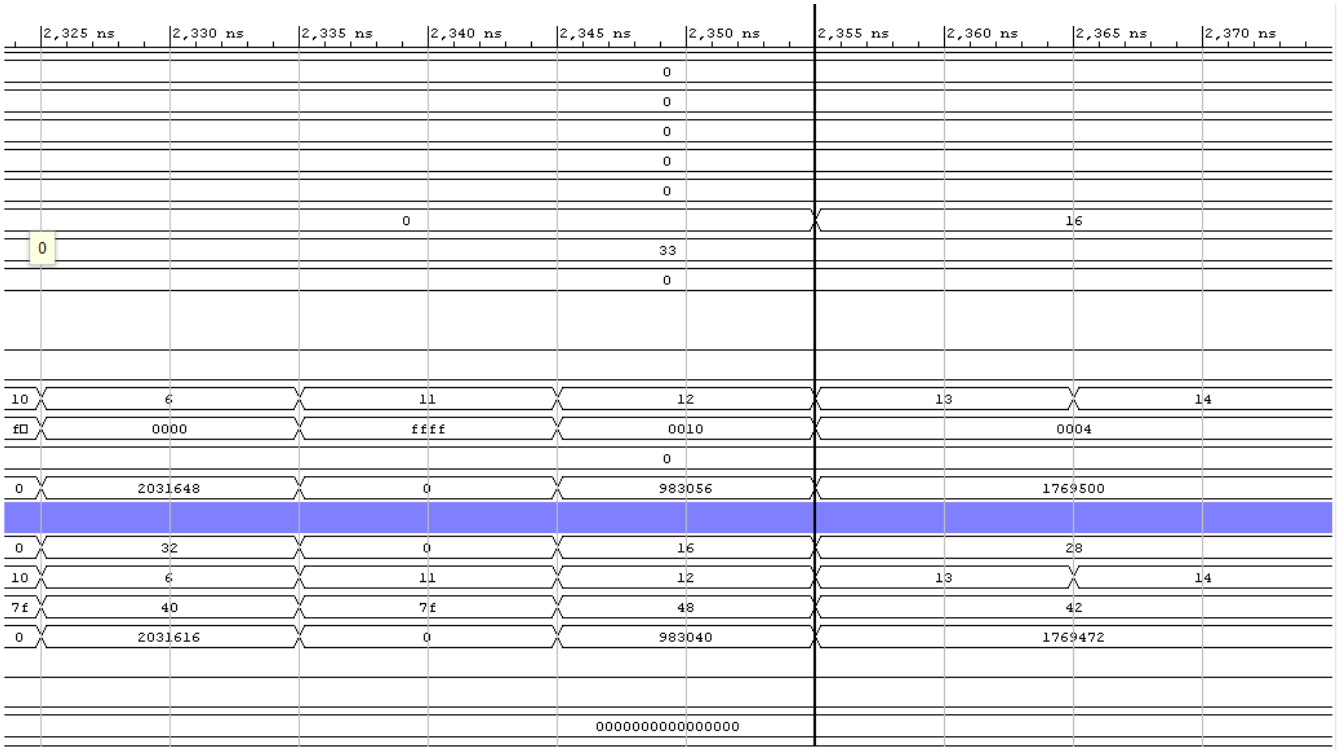
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Processor_tb is

end Processor_tb;

architecture Behavioral of Processor_tb is

constant clk_period : time := 10ns;
signal PB          :  STD_LOGIC;
signal clk         :  std_logic;
signal rst         :  std_logic;
signal LED_output  :  STD_LOGIC_VECTOR (7 downto 0);

begin
UUT : entity work.Processor
    Port map(PB => PB,
             clk => clk,
             rst => rst,
             LED_output => LED_output
             );

-- Clock process
clk_process :process
begin
  clk <= '0';
  wait for clk_period/2;
  clk <= '1';
  wait for clk_period/2;
end process;



-- The test strategy is to simply simulate pushing the pushbutton so the
-- execution of the instructions is continued in our program, other necessary
-- testing measures for synchronous circuits are also implemented
```

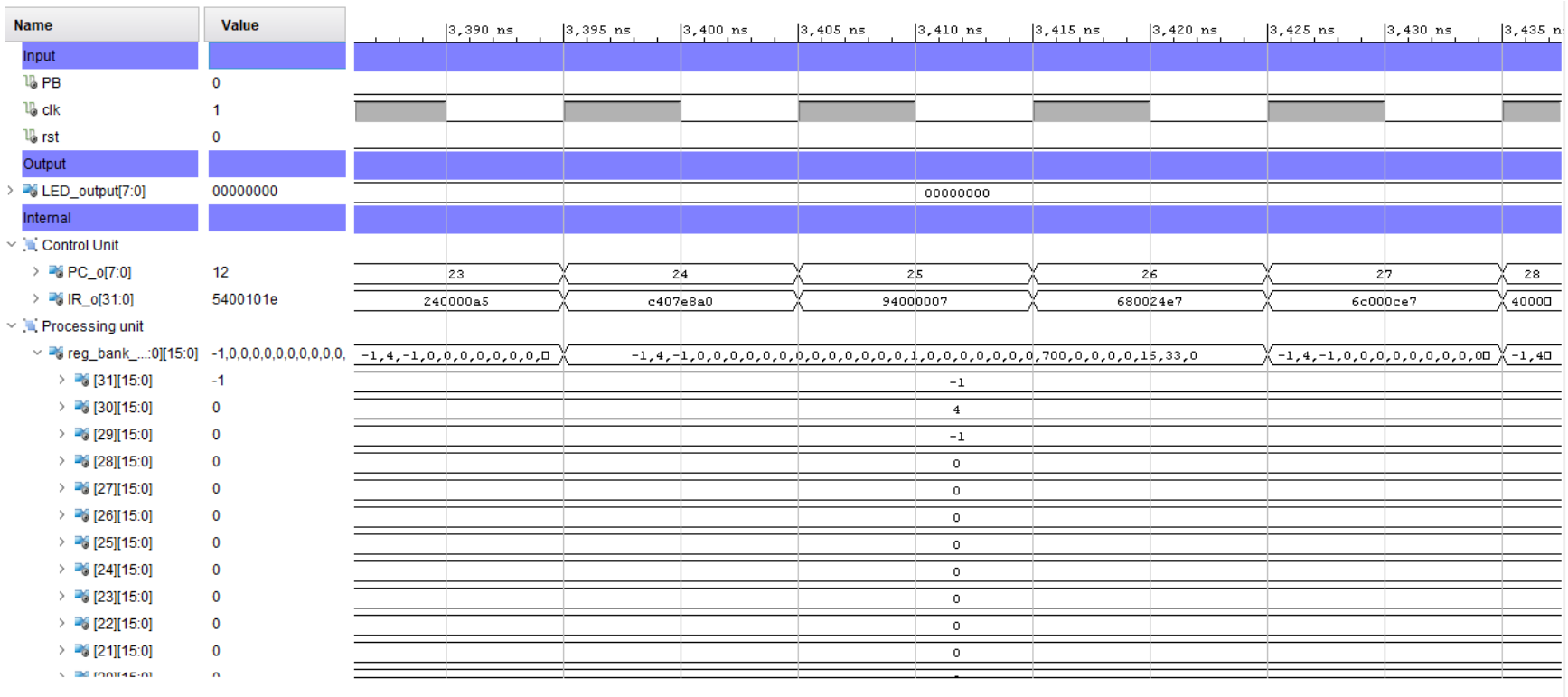
```
test_process : process
begin
    -- wait 100 ns for global reset to finish
    wait for 100ns;

    wait until falling_edge(clk); -- Clock Synchronization

    -- initializing inputs
    PB <= '0';
    rst <= '0';
    wait for clk_period*1;

    -- resetting the module
    rst <= '1';
    wait for clk_period*1;

    rst <= '0';
    wait for clk_period*5;
    -- Simulating pushing the button
    PB <= '1';
    wait for clk_period*10;

    PB <= '0';
    wait for clk_period*40;

    -- Middle reset
    rst <= '1';
    wait for clk_period*1;

    rst <= '0';
    wait for clk_period*1;
    -- going through instructions again
    PB <= '1';
    wait for clk_period*10;

    PB <= '0';

    wait;
end process;

end Behavioral;
```

Simulation screenshots:

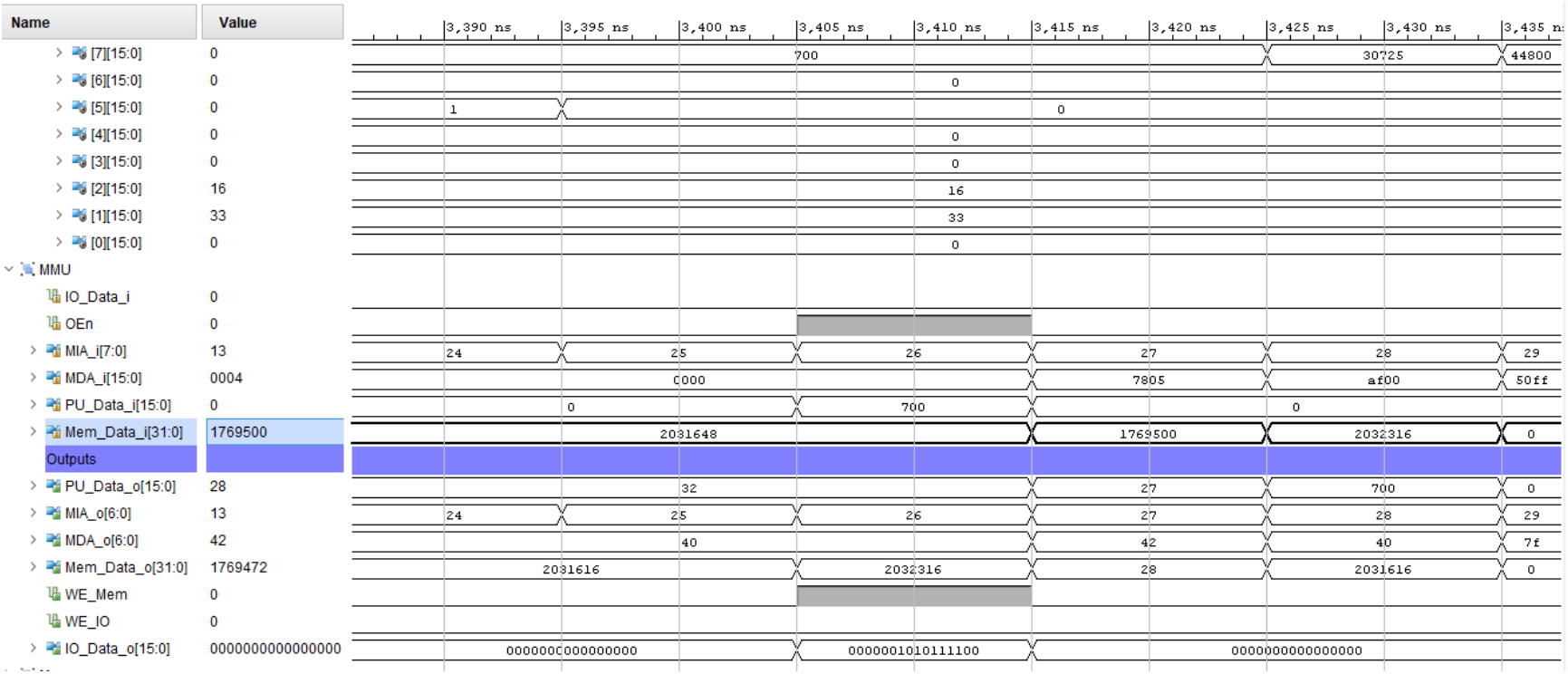


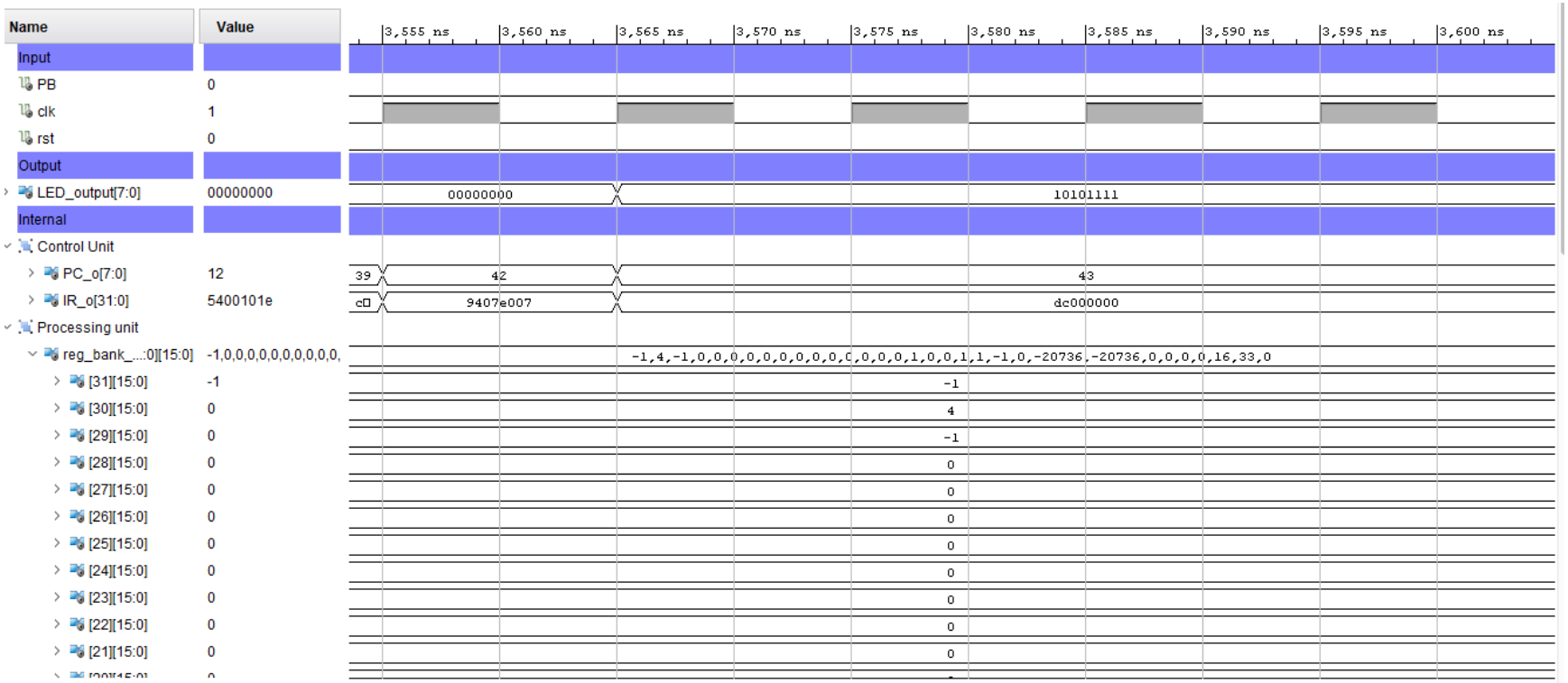
Name	Value
[7][15:0]	0
[6][15:0]	0
[5][15:0]	0
[4][15:0]	0
[3][15:0]	0
[2][15:0]	16
[1][15:0]	33
[0][15:0]	0
MMU	
IO_Data_i	0
OEn	0
MIA_i[7:0]	13
MDA_i[15:0]	0004
PU_Data_i[15:0]	0
Mem_Data_i[31:0]	1769500
Outputs	
PU_Data_o[15:0]	28
MIA_o[6:0]	13
MDA_o[6:0]	42
Mem_Data_o[31:0]	1769472
WE_Mem	0
WE_IO	0
IO_Data_o[15:0]	0000000000000000



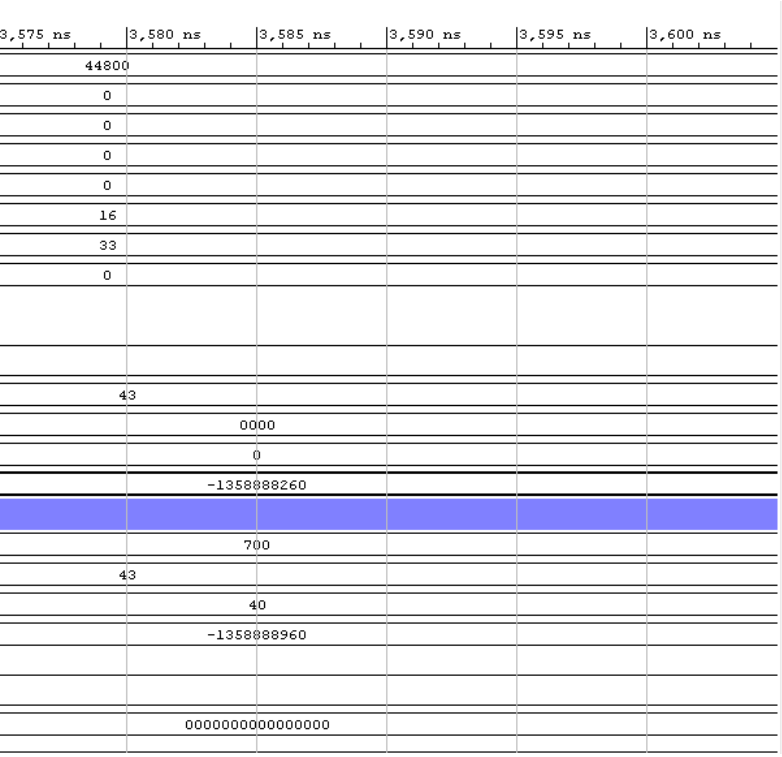


Name	Value
Input	
PB	0
clk	1
rst	0
Output	
LED_output[7:0]	00000000
Internal	
Control Unit	
PC_o[7:0]	12
IR_o[31:0]	5400101e
Processing unit	
reg_bank_...:0][15:0]	-1,0,0,0,0,0,0,0,0,0,0,
[31][15:0]	-1
[30][15:0]	0
[29][15:0]	0
[28][15:0]	0
[27][15:0]	0
[26][15:0]	0
[25][15:0]	0
[24][15:0]	0
[23][15:0]	0
[22][15:0]	0
[21][15:0]	0


Name
Value
3,390 ns
3,395 ns
3,400 ns
3,405 ns
3,410 ns
3,415 ns
3,420 ns
3,425 ns
3,430 ns
[7][15:0] 0
[6][15:0] 0
[5][15:0] 0
[4][15:0] 0
[3][15:0] 0
[2][15:0] 16
[1][15:0] 33
[0][15:0] 0
MMU
IO_Data_i 0
OEn 0
MIA_i[7:0] 13
MDA_i[15:0] 0004
PU_Data_i[15:0] 0
Mem_Data_i[31:0] 1769500
Outputs
PU_Data_o[15:0] 28
MIA_o[6:0] 13
MDA_o[6:0] 42
Mem_Data_o[31:0] 1769472
WE_Mem 0
WE_IO 0
IO_Data_o[15:0] 0000000000000000
700
30725
44800
0
1
0
0
0
16
33
0
24
25
26
27
28
29
0000
7805
af00
50ff
0
700
0
2031648
1769500
2032316
0
32
27
700
0
24
25
26
27
28
29
40
42
40
7f
2031616
2032316
28
2031616
0
0000000000000000
0000001010111100
0000000000000000


Name
Value
3,555 ns
3,560 ns
3,565 ns
3,570 ns
3,575 ns
3,580 ns
3,585 ns
3,590 ns
3,595 ns
3,600 ns
Input
PB 0
clk 1
rst 0
Output
LED_output[7:0] 00000000
00000000
10101111
Internal
Control Unit
PC_o[7:0] 12
39
42
43
IR_o[31:0] 5400101e
c0
9407e007
dc000000
Processing unit
reg_bank_...:0][15:0] -1,0,0,0,0,0,0,0,0,0,0,
-1,4,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,1,-1,0,-20736,-20736,0,0,0,0,16,33,0
[31][15:0] -1
-1
[30][15:0] 0
4
[29][15:0] 0
-1
[28][15:0] 0
0
[27][15:0] 0
0
[26][15:0] 0
0
[25][15:0] 0
0
[24][15:0] 0
0
[23][15:0] 0
0
[22][15:0] 0
0
[21][15:0] 0
0

Name	Value
[7][15:0]	0
[6][15:0]	0
[5][15:0]	0
[4][15:0]	0
[3][15:0]	0
[2][15:0]	16
[1][15:0]	33
[0][15:0]	0
MMU	
IO_Data_i	0
OEn	0
MIA_i[7:0]	13
MDA_i[15:0]	0004
PU_Data_i[15:0]	0
Mem_Data_i[31:0]	1769500
Outputs	
PU_Data_o[15:0]	28
MIA_o[6:0]	13
MDA_o[6:0]	42
Mem_Data_o[31:0]	1769472
WE_Mem	0
WE_IO	0
IO_Data_o[15:0]	0000000000000000

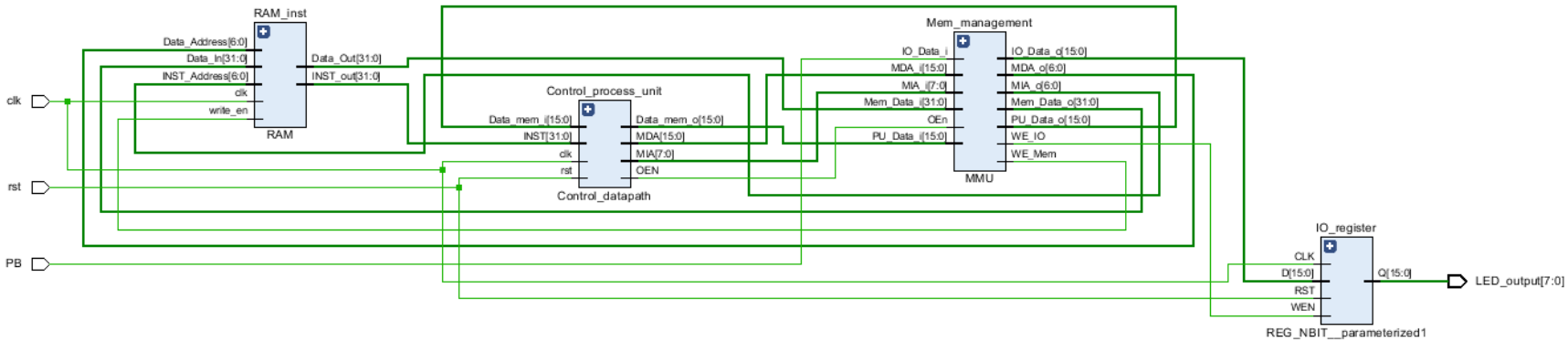


```
---------------------------------------------------------------------------------
Start RTL Component Statistics
---------------------------------------------------------------------------------
Detailed RTL Component Info :
+---Adders :
          2 Input     16 Bit       Adders := 1
          3 Input     16 Bit       Adders := 1
          2 Input      8 Bit       Adders := 1
+---XORs :
          2 Input     16 Bit         XORs := 1
+---Registers :
                      32 Bit    Registers := 1
                      16 Bit    Registers := 32
                       8 Bit    Registers := 1
+---Muxes :
          2 Input     32 Bit        Muxes := 1
         14 Input     16 Bit        Muxes := 2
          2 Input     16 Bit        Muxes := 69
          2 Input      8 Bit        Muxes := 1
          2 Input      5 Bit        Muxes := 1
         21 Input      4 Bit        Muxes := 1
          2 Input      1 Bit        Muxes := 3
---------------------------------------------------------------------------------
Finished RTL Component Statistics
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
Start RTL Hierarchical Component Statistics
---------------------------------------------------------------------------------
Hierarchical RTL Component report
Module REG_NBIT
Detailed RTL Component Info :
+---Registers :
                      32 Bit    Registers := 1
Module REG_NBIT__parameterized0
Detailed RTL Component Info :
+---Registers :
                       8 Bit    Registers := 1
Module control_logic
Detailed RTL Component Info :
+---Adders :
          2 Input      8 Bit       Adders := 1
+---Muxes :
          2 Input      8 Bit        Muxes := 1
          2 Input      5 Bit        Muxes := 1
         21 Input      4 Bit        Muxes := 1
Module ALU
Detailed RTL Component Info :
+---Adders :
          2 Input     16 Bit       Adders := 1
          3 Input     16 Bit       Adders := 1
+---XORs :
          2 Input     16 Bit         XORs := 1
+---Muxes :
         14 Input     16 Bit        Muxes := 2
          2 Input      1 Bit        Muxes := 2
Module REG_NBIT__parameterized1
Detailed RTL Component Info :
+---Registers :
                      16 Bit    Registers := 1
Module reg_bank
Detailed RTL Component Info :
+---Muxes :
          2 Input     16 Bit        Muxes := 64
```

```
Module data_path
Detailed RTL Component Info :
+---Muxes :
          2 Input     16 Bit        Muxes := 3
Module MMU
Detailed RTL Component Info :
+---Muxes :
          2 Input     32 Bit        Muxes := 1
          2 Input     16 Bit        Muxes := 2
          2 Input      1 Bit        Muxes := 1
--------------------------------------------------------------------------------
Finished RTL Hierarchical Component Statistics
--------------------------------------------------------------------------------
```

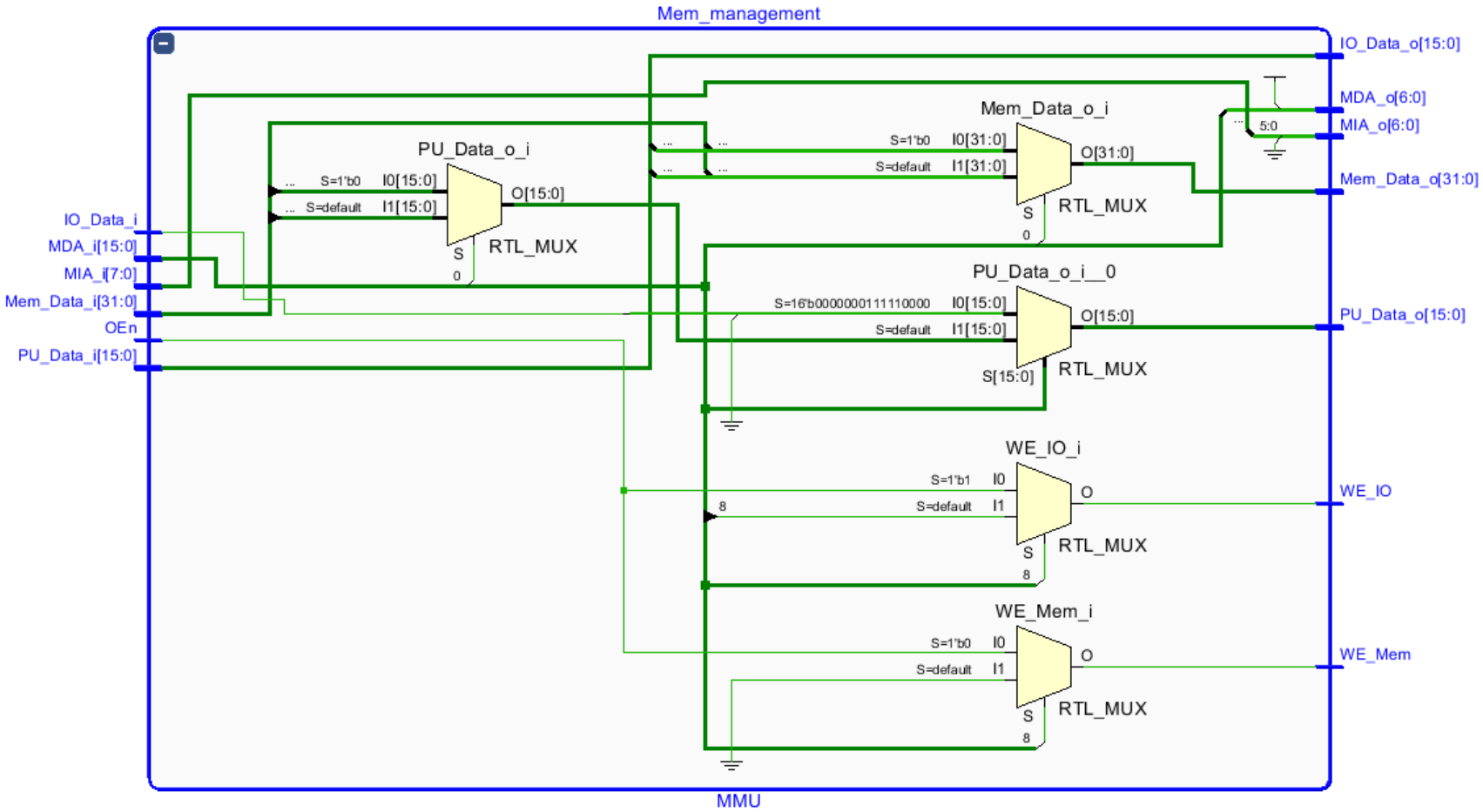
Schematics:

Top Module:



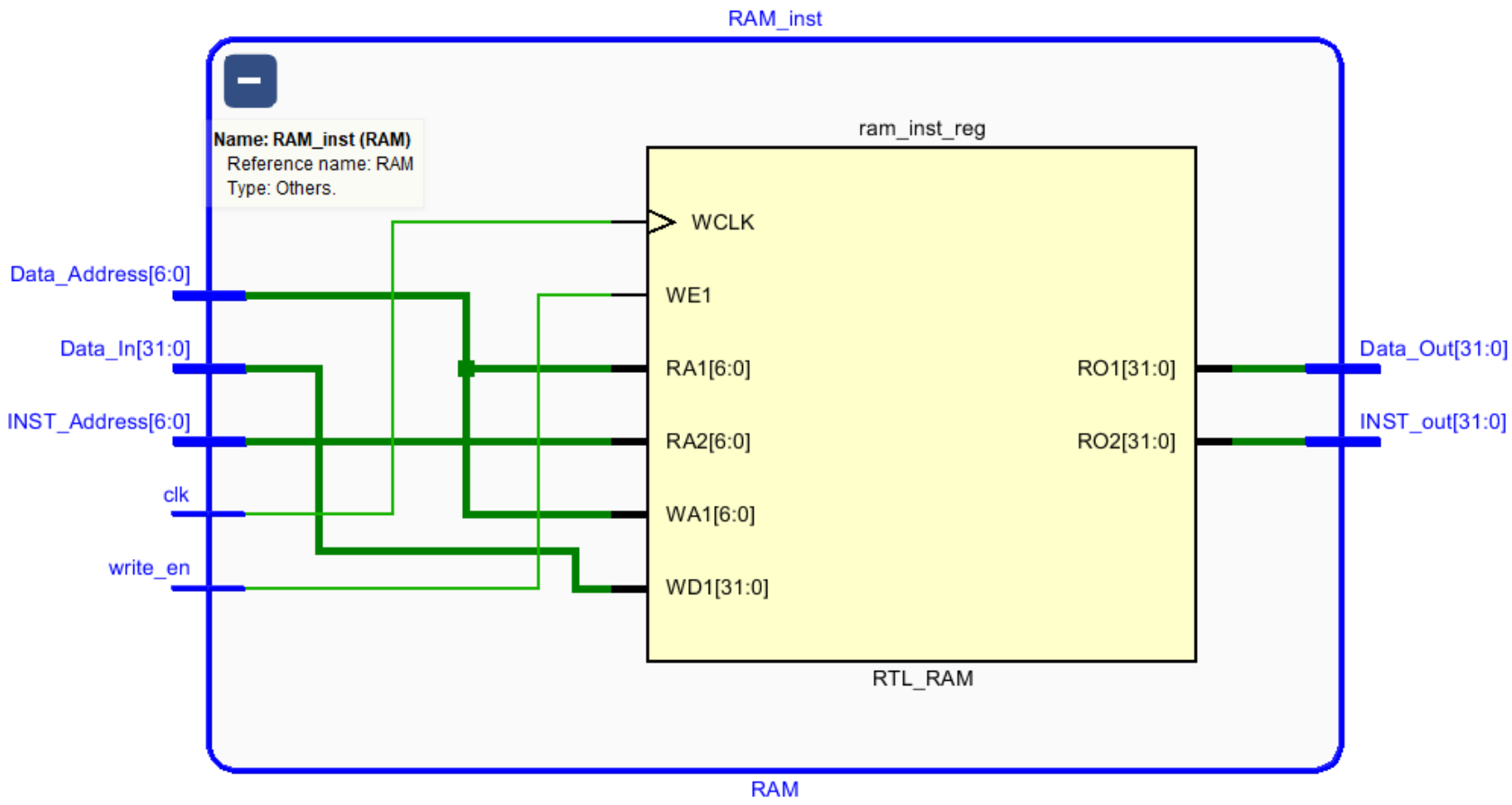
The modules are connected together as we intended.

Memory Management Unit:



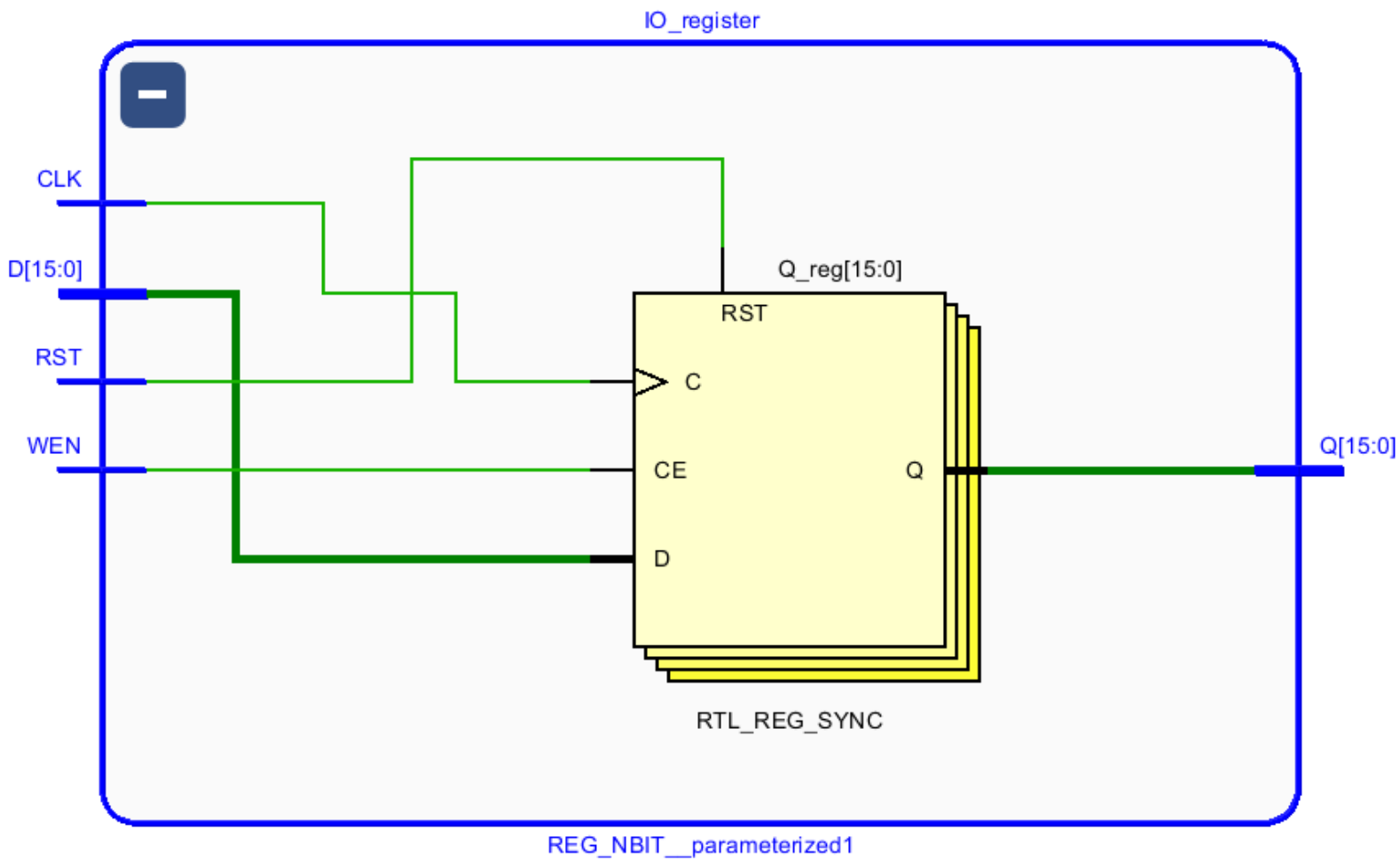
The conditional assignments are translated to muxes as we desired.

Memory Schematics:



A RAM is inferred by synthesis tools from our behavioral code, as we intended.

IO Register:


IO_register
CLK
D[15:0]
RST
WEN
Q_reg[15:0]
RST
C
CE
D
Q
Q[15:0]
RTL_REG_SYNC
REG_NBIT__parameterized1