ActiveCore

Laboratory work manual

Using UDM bus transactor in FPGA designs

Author:

Alexander Antonov

antonov.alex.alex@gmail.com

2020

Contents

1. TARGET SKILLS

- Developing observable and controllable hardware designs using UDM bus transactor
- Verifying UDM-managed designs in simulation environment
- Using Xilinx FPGA and Vivado Design Suite for implementation of UDM-managed designs
- Testing UDM-managed designs from PC programming environment

2. OVERVIEW

This laboratory work is aimed at understanding the design flow of FPGA project managed by UDM bus transactor, its structure and role of its components. It is explored how to add custom RTL using SystemVerilog Hardware Description Language, write testbenches, verify correctness of design in simulation environment, implement the design in FPGA device and collect metrics of the obtained implementation.

3. PREREQUISITES

1. Xilinx Vivado 2019.2 HLx Edition (free for target board, available at https://www.xilinx.com/support/download.html).
2. ActiveCore baseline distribution (available at https://github.com/AntonovAlexander/activecore)
3. (for FPGA prototyping) Digilent Nexys 4 DDR FPGA board (https://store.digilentinc.com/nexys-4-ddr-artix-7-fpga-trainer-board-recommended-for-ece-curriculum/)
4. (for FPGA prototyping) working Python 3 installation with `pyserial` package

4. TASK

1. Examine UDM baseline project
2. (if FPGA board available) Implement UDM project in FPGA device and verify correctness of the baseline
3. Design RTL module in synthesizable SystemVerilog HDL according to your variant
4. Integrate your design with UDM bus master module
5. Write the testbench and simulate to verify correctness of your design
6. Implement your design, collect, and analyze metrics of the implementation
7. (if FPGA board available) Write HW test matching the testbench
8. (if FPGA board available) Program the design in FPGA board and make sure the design operates correctly

5. GUIDANCE

Detailed guidance will be provided using the example of a custom pipelined module that searches for the maximum value in 16-element array and returns this value and its index in the array.

1. Examine UDM baseline project

UDM (**U**ART-based **D**ebug **M**odule) is a bus master module that executes bus transactions controlled via serial port interface. This provides basic initialization, communication and debug capabilities for custom cores in FPGA fabric, allowing PC to “emulate” CPU host in System-on-Chip design. UDM block requires minimum setup, can be implemented in minutes, consumes minimum resources (< 1% of LUTs and FFs on target board) and requires no additional HW except for default serial port connectivity.

UDM block diagram is located at:

https://github.com/AntonovAlexander/activecore/blob/master/designs/rtl/udm/doc/udm_baseline_struct.png

NOTE: only 4-byte aligned accesses are allowed.

The project is located at: `activecore/designs/rtl/udm/syn/NEXYS4-DDR`. Open `NEXYS4_DDR.xpr` file using Xilinx Vivado.

NOTE: avoid having non-English characters in project location path.

2. (if FPGA board available) Implement UDM project in FPGA device and verify correctness of the baseline

Press “Generate Bitstream” button in Vivado software to generate bitstream (image that can be uploaded to FPGA).

Find out the name of COM port associated with the board (`COM<number>` on Windows hosts or `tty<number>` on Linux hosts).

Open test Python script (located at `activecore/designs/rtl/udm/sw/udm_test.py`) and fill the correct COM port name in line 7:

```
udm = udm("<correct COM port name>", 921600)
```

Run UDM test using `hw_test.py` Python script. The script will connect to the board and check response. The console output should be:

```
Connecting COM port...
COM port connected
Connection established, response:  0x55

SW read:  <value on switches>

---- memtest32 started, word size: 1024  ----
---- memtest32 PASSED ----
```

The script does the following:

1) Writing `0xaa55` value to CSR mapped on LEDs using `udm.wr32(addr, wdata)` function
2) Reading CSR mapped on switches and printing this value
3) Testing `testmem` memory block using `udm.memtest32(addr, wsize)` function

Type `help(udm)` in Python console for full API reference.

3. Design RTL module in synthesizable SystemVerilog HDL according to your variant

Pipelined implementation requires the computation to be divided into **stages** (**control steps**, **c-steps**) that execute in **overlapped** fashion. The **schedule** for this implementation is shown in Table 1:

c-step number	operation
0	compare elements in pairs: 0-1; 2-3; 4-5; 6-7; 8-9; 10-11; 12-13; 14-15
1	compare pairing results from stage 0 in pairs: 0-1; 2-3; 4-5; 6-7
2	compare pairing results from stage 1 in pairs: 0-1; 2-3
3	compare pairing results from stage 2

Table 1 Schedule for pipelined implementation

Source code for the example module in shown in Listing 1:

```
module FindMaxVal_pipelined (
    input clk_i
    , input rst_i

    , input [31:0] elem_bi [15:0]

    , output logic [31:0] max_elem_bo
    , output logic [3:0] max_index_bo
```

```
);

// stage 0
logic [31:0] max_elem_stage0  [7:0];
logic [31:0] max_index_stage0 [7:0];
always @(posedge clk_i)
    begin
    for(integer i=0; i<8; i++)
        begin
        if (elem_bi[(i<<1)] > elem_bi[(i<<1)+1])
            begin
            max_elem_stage0[i] <= elem_bi[(i<<1)];
            max_index_stage0[i] <= i<<1;
            end
        else
            begin
            max_elem_stage0[i] <= elem_bi[(i<<1)+1];
            max_index_stage0[i] <= (i<<1)+1;
            end
        end
    end

// stage 1
logic [31:0] max_elem_stage1  [3:0];
logic [31:0] max_index_stage1 [3:0];
always @(posedge clk_i)
    begin
    for(integer i=0; i<4; i++)
        begin
        if (max_elem_stage0[(i<<1)] > max_elem_stage0[(i<<1)+1])
            begin
            max_elem_stage1[i] <= max_elem_stage0[(i<<1)];
            max_index_stage1[i] <= max_index_stage0[(i<<1)];
            end
        else
            begin
            max_elem_stage1[i] <= max_elem_stage0[(i<<1)+1];
            max_index_stage1[i] <= max_index_stage0[(i<<1)+1];
            end
        end
    end

// stage 2
logic [31:0] max_elem_stage2  [1:0];
logic [31:0] max_index_stage2 [1:0];
always @(posedge clk_i)
    begin
    for(integer i=0; i<2; i++)
        begin
        if (max_elem_stage1[(i<<1)] > max_elem_stage1[(i<<1)+1])
            begin
            max_elem_stage2[i] <= max_elem_stage1[(i<<1)];
            max_index_stage2[i] <= max_index_stage1[(i<<1)];
            end
        else
            begin
            max_elem_stage2[i] <= max_elem_stage1[(i<<1)+1];
            max_index_stage2[i] <= max_index_stage1[(i<<1)+1];
            end
        end
    end
```

```
// stage 3
always @(posedge clk_i)
    begin
    if (max_elem_stage2[0] > max_elem_stage2[1])
        begin
        max_elem_bo <= max_elem_stage2[0];
        max_index_bo <= max_index_stage2[0];
        end
    else
        begin
        max_elem_bo <= max_elem_stage2[1];
        max_index_bo <= max_index_stage2[1];
        end
    end

endmodule
```

Listing 1 Source code of the `FindMaxVal_pipelined` module in SystemVerilog HDL

4. Integrate your design with UDM bus master module

Add your created design file to the project using Vivado GUI.

UDM exposes a system bus into FPGA fabric for custom logic integration and testing. UDM bus has a simplistic, RAM-like protocol, supports pipelined transactions and can easily be converted to various standard protocols (AMBA AHB, Avalon, Wishbone, etc.).

UDM write transaction waveform is located at:

https://github.com/AntonovAlexander/activecore/blob/master/designs/rtl/udm/doc/udm_bus_wr_waveform.svg

UDM read transaction waveform is located at:

https://github.com/AntonovAlexander/activecore/blob/master/designs/rtl/udm/doc/udm_bus_rd_waveform.svg

UDM has several predefined addresses where LED and switches control and status registers (CSRs) are mapped, as well as test memory. Address map of UDM baseline is located at:

https://github.com/AntonovAlexander/activecore/blob/master/designs/rtl/udm/doc/udm_baseline_addr_map.md

Now we add custom CSRs to manage operation of the designed logic in top wrapper module. We need 16 CSRs for input data and 2 CSRs for output data. We should map these CSRs on free addresses, not overlapping with other CSRs and memories.

Here we map input CSRs on the following addresses:

Input data CSRs:

- `csr_elem_in`: 0x10000000-0x1000003C (16x elements with 4-byte stride)

Output data CSRs:

- `csr_max_elem_out`: 0x20000000
- `csr_max_index_out`: 0x20000004

Instantiate the CSRs and the designed module in top wrapper module (`NEXYS4_DDR.sv`) and connect it to custom CSRs. Resulting code is shown in Listing 2 (modified parts are highlighted in cyan).

```
module NEXYS4_DDR
#( parameter SIM = "NO" )
(
  input     CLK100MHZ
    , input   CPU_RESETN

    , input   [15:0] SW
```

```
    , output logic  [15:0] LED

    , input   UART_TXD_IN
    , output  UART_RXD_OUT
);

localparam UDM_BUS_TIMEOUT = (SIM == "YES") ? 100 : (1024*1024*100);
localparam UDM_RTX_EXTERNAL_OVERRIDE = (SIM == "YES") ? "YES" : "NO";

logic clk_gen;
logic pll_locked;

sys_clk sys_clk
(
    .clk_in1(CLK100MHZ)
    , .reset(!CPU_RESETN)
    , .clk_out1(clk_gen)
    , .locked(pll_locked)
);

logic arst;
assign arst = !(CPU_RESETN & pll_locked);

logic srst;
reset_cntrl reset_cntrl
(
   .clk_i(clk_gen),
   .arst_i(arst),
   .srst_o(srst)
);

logic udm_reset;

logic [0:0] udm_req;
logic [0:0] udm_we;
logic [31:0] udm_addr;
logic [3:0] udm_be;
logic [31:0] udm_wdata;
logic [0:0] udm_ack;
logic [0:0] udm_resp;
logic [31:0] udm_rdata;

udm
#(
    .BUS_TIMEOUT(UDM_BUS_TIMEOUT)
    , .RTX_EXTERNAL_OVERRIDE(UDM_RTX_EXTERNAL_OVERRIDE)
) udm (
   .clk_i(clk_gen)
   , .rst_i(srst)

   , .rx_i(UART_TXD_IN)
   , .tx_o(UART_RXD_OUT)

   , .rst_o(udm_reset)

   , .bus_req_o(udm_req)
   , .bus_we_o(udm_we)
   , .bus_addr_bo(udm_addr)
   , .bus_be_bo(udm_be)
   , .bus_wdata_bo(udm_wdata)
   , .bus_ack_i(udm_ack)
```

```
  , .bus_resp_i(udm_resp)
  , .bus_rdata_bi(udm_rdata)
);

localparam CSR_LED_ADDR         = 32'h00000000;
localparam CSR_SW_ADDR          = 32'h00000004;
localparam TESTMEM_ADDR         = 32'h80000000;

localparam TESTMEM_WSIZE_POW    = 10;
localparam TESTMEM_WSIZE        = 2**TESTMEM_WSIZE_POW;

logic testmem_udm_enb;
assign testmem_udm_enb = (!(udm_addr < TESTMEM_ADDR) && (udm_addr < (TESTMEM_ADDR +
(TESTMEM_WSIZE*4))));

logic testmem_udm_we;
logic [TESTMEM_WSIZE_POW-1:0] testmem_udm_addr;
logic [31:0] testmem_udm_wdata;
logic [31:0] testmem_udm_rdata;

logic testmem_p1_we;
logic [TESTMEM_WSIZE_POW-1:0] testmem_p1_addr;
logic [31:0] testmem_p1_wdata;
logic [31:0] testmem_p1_rdata;

// testmem's port1 is inactive
assign testmem_p1_we = 1'b0;
assign testmem_p1_addr = 0;
assign testmem_p1_wdata = 0;

ram_dual #(
    .mem_init("NO")
    , .mem_data("nodata.hex")
    , .dat_width(32)
    , .adr_width(TESTMEM_WSIZE_POW)
    , .mem_size(TESTMEM_WSIZE)
) testmem (
    .clk(clk_gen)

    , .dat0_i(testmem_udm_wdata)
    , .adr0_i(testmem_udm_addr)
    , .we0_i(testmem_udm_we)
    , .dat0_o(testmem_udm_rdata)

    , .dat1_i(testmem_p1_wdata)
    , .adr1_i(testmem_p1_addr)
    , .we1_i(testmem_p1_we)
    , .dat1_o(testmem_p1_rdata)
);

assign udm_ack = udm_req;   // bus always ready to accept request
logic csr_resp, testmem_resp, testmem_resp_dly;
logic [31:0] csr_rdata;

// CSR instantiation
logic [31:0] csr_elem_in [15:0];
logic [31:0] csr_max_elem_out;
logic [3:0] csr_max_index_out;

// module instantiation
FindMaxVal_pipelined FindMaxVal_inst (
```

```
    .clk_i(clk_gen)
    , .rst_i(srst)
    , .elem_bi(csr_elem_in)
    , .max_elem_bo(csr_max_elem_out)
    , .max_index_bo(csr_max_index_out)
);

// bus request
always @(posedge clk_gen)
    begin

    testmem_udm_we <= 1'b0;
    testmem_udm_addr <= 0;
    testmem_udm_wdata <= 0;

    csr_resp <= 1'b0;
    testmem_resp_dly <= 1'b0;
    testmem_resp <= testmem_resp_dly;

    if (udm_req && udm_ack)
        begin

        if (udm_we)      // writing
            begin
            if (udm_addr == CSR_LED_ADDR) LED <= udm_wdata;
            if (udm_addr[31:28] == 4'h1) csr_elem_in[udm_addr[5:2]] <= udm_wdata;
            if (testmem_udm_enb)
                begin
                testmem_udm_we <= 1'b1;
                testmem_udm_addr <= udm_addr[31:2];     // 4-byte aligned access only
                testmem_udm_wdata <= udm_wdata;
                end
            end

        else            // reading
            begin
            if (udm_addr == CSR_LED_ADDR)
                begin
                csr_resp <= 1'b1;
                csr_rdata <= LED;
                end
            if (udm_addr == CSR_SW_ADDR)
                begin
                csr_resp <= 1'b1;
                csr_rdata <= SW;
                end
            if (udm_addr == 32'h20000000)
                begin
                csr_resp <= 1'b1;
                csr_rdata <= csr_max_elem_out;
                end
            if (udm_addr == 32'h20000004)
                begin
                csr_resp <= 1'b1;
                csr_rdata <= csr_max_index_out;
                end
            if (testmem_udm_enb)
                begin
                testmem_udm_we <= 1'b0;
                testmem_udm_addr <= udm_addr[31:2];     // 4-byte aligned access only
                testmem_udm_wdata <= udm_wdata;
```

```
                testmem_resp_dly <= 1'b1;
                end
            end
        end
    end

// bus response
always @*
    begin
    udm_resp = csr_resp | testmem_resp;
    udm_rdata = 0;
    if (csr_resp) udm_rdata = csr_rdata;
    if (testmem_resp) udm_rdata = testmem_udm_rdata;
    end

endmodule
```

Listing 2 Source code of the updated `NEXYS4_DDR.sv` module

5. Write the testbench and simulate to verify correctness of your design

The basic testbench functionality consists in the following operations:

- write the input data (stimulus) to the target synthesizable module (Design Under Test, DUT);
- start the computation (not needed here);
- read and verify the result.

Go to the testbench file (`tb.sv`) and find the main test procedure (`initial` block in the end of the file). Fill the input data with some random values and retrieve the result. Resulting initial block is shown in Listing 3 (modified parts are highlighted in cyan).

```
initial
    begin
    logic [31:0] wrdata [];
    integer ARRSIZE=10;

  $display ("### SIMULATION STARTED ###");

  udm.cfg(`DIVIDER_115200, 2'b00);

  SW = 8'h30;
  RESET_ALL();
  WAIT(100);
  udm.check();
  udm.hreset();

  // test data initialization
  udm.wr32(32'h10000000, 32'h112233cc);
  udm.wr32(32'h10000004, 32'h55aa55aa);
  udm.wr32(32'h10000008, 32'h01010202);
  udm.wr32(32'h1000000C, 32'h44556677);
  udm.wr32(32'h10000010, 32'h00000003);
  udm.wr32(32'h10000014, 32'h00000004);
  udm.wr32(32'h10000018, 32'h00000005);
  udm.wr32(32'h1000001C, 32'h00000006);
  udm.wr32(32'h10000020, 32'h00000007);
  udm.wr32(32'h10000024, 32'hdeadbeef);
  udm.wr32(32'h10000028, 32'hfefe8800);
  udm.wr32(32'h1000002C, 32'h23344556);
  udm.wr32(32'h10000030, 32'h05050505);
  udm.wr32(32'h10000034, 32'h07070707);
  udm.wr32(32'h10000038, 32'h99999999);
```

```
   udm.wr32(32'h1000003C, 32'hbadc0ffe);

   // fetching results
   udm.rd32(32'h20000000);
   udm.rd32(32'h20000004);

   WAIT(1000);

   $display ("### TEST PROCEDURE FINISHED ###");
   $stop;
end
```

Listing 3 Test procedure for the designed module

Note that maximum value is `0xfefe8800` at index 10 (`0xa`).

Now run the simulation. Add all interesting signals (including system bus interface and your module internals) to the waveform. The signals can be added using context menu on signals listed in Vivado GUI (see Figure 1).

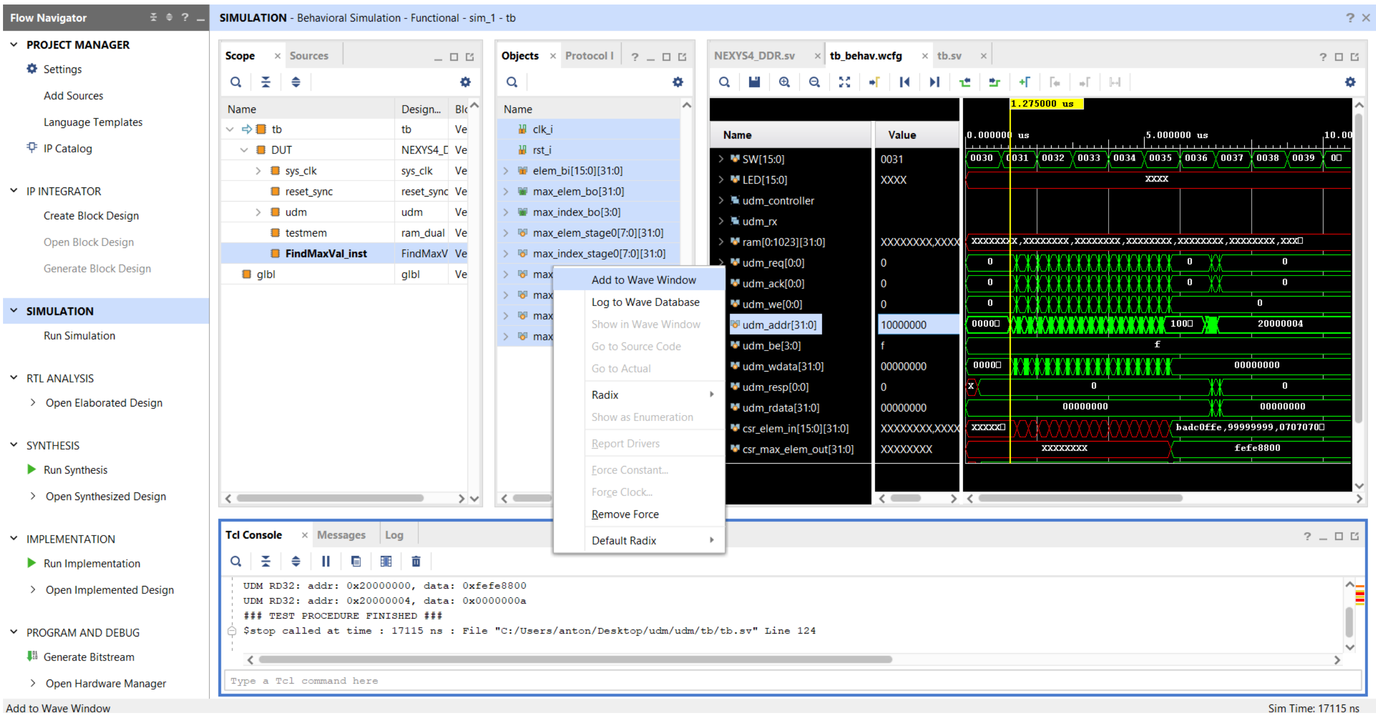

Figure 1 Adding signals to waveform

If needed, change waveform style for selected signals (digital/analog) and radix (binary, hexadecimal, decimal, etc).

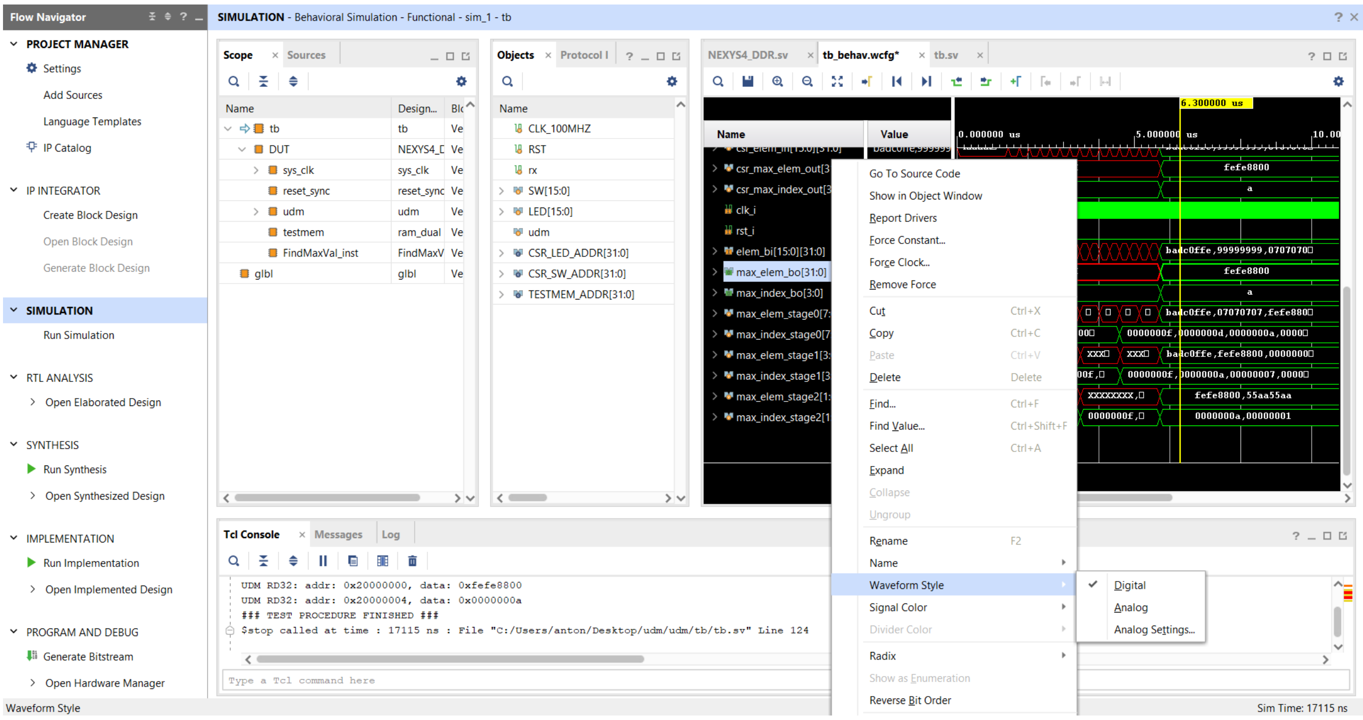

Figure 2 Waveform configuration

Console output for simulation is shown in Listing 4.

```
UDM WR32: addr: 0x10000000, data: 0x112233cc
UDM WR32: addr: 0x10000004, data: 0x55aa55aa
UDM WR32: addr: 0x10000008, data: 0x01010202
UDM WR32: addr: 0x1000000c, data: 0x44556677
UDM WR32: addr: 0x10000010, data: 0x00000003
UDM WR32: addr: 0x10000014, data: 0x00000004
UDM WR32: addr: 0x10000018, data: 0x00000005
UDM WR32: addr: 0x1000001c, data: 0x00000006
UDM WR32: addr: 0x10000020, data: 0x00000007
UDM WR32: addr: 0x10000024, data: 0xdeadbeef
UDM WR32: addr: 0x10000028, data: 0xfefe8800
UDM WR32: addr: 0x1000002c, data: 0x23344556
UDM WR32: addr: 0x10000030, data: 0x05050505
UDM WR32: addr: 0x10000034, data: 0x07070707
UDM WR32: addr: 0x10000038, data: 0x99999999
UDM WR32: addr: 0x1000003c, data: 0xbadc0ffe
UDM RD32: addr: 0x20000000, data: 0xfefe8800
UDM RD32: addr: 0x20000004, data: 0x0000000a
### TEST PROCEDURE FINISHED ###
```

Listing 4 Console output of simulation

Note that max element and its index have been read correctly at addresses 0x20000000 and 0x20000004 respectively (highlighted in cyan). Waveform for the simulation is shown in Figure 3.

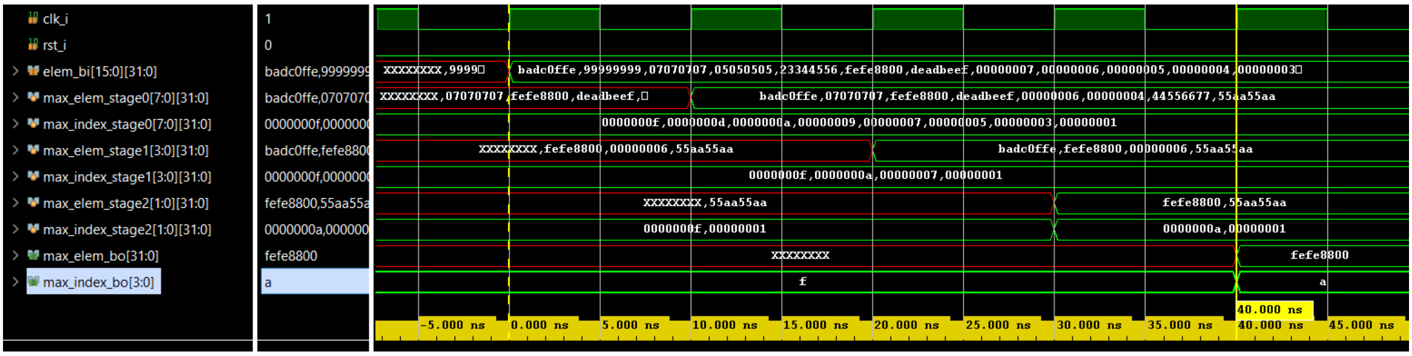

Figure 3 Waveform of simulation

The simulation is correct, DUT works as intended.

6. Implement your design, collect, and analyze metrics of the implementation

Press “Generate Bitstream” to run implementation and obtain the image for FPGA device.

Metric values are the following:

- Timing:
  - WNS: 4.883 ns (fine)
  - TNS: 0 ns (fine)
- Performance:
  - Clock frequency: 10 ns (100 MHz)
  - Initiation Interval: 1 clock cycle; 10 ns
  - Throughput: 1 op/cycle; 100 Mop/second
  - Latency: 16 clock cycles; 160 ns
- HW resources (Implementation → Open Implemented Design → Report Utilization):
  - LUTs: 498
  - FFs (registers): 506

The timing closure is **successful**.

7. (if FPGA board available) Write HW test matching the testbench

Open test Python script (located at `activecore/designs/rtl/udm/sw/udm_test.py`) and write the test program matching SystemVerilog testbench. This program is needed for HW testing in FPGA board. Source code for the program is shown in Listing 5.

```
from __future__ import division

import udm
from udm import *

udm = udm('<your COM port name>', 921600)

# test data initialization
udm.wr32(0x10000000, 0x112233cc);
udm.wr32(0x10000004, 0x55aa55aa);
udm.wr32(0x10000008, 0x01010202);
udm.wr32(0x1000000C, 0x44556677);
```

```
udm.wr32(0x10000010, 0x00000003);
udm.wr32(0x10000014, 0x00000004);
udm.wr32(0x10000018, 0x00000005);
udm.wr32(0x1000001C, 0x00000006);
udm.wr32(0x10000020, 0x00000007);
udm.wr32(0x10000024, 0xdeadbeef);
udm.wr32(0x10000028, 0xfefe8800);
udm.wr32(0x1000002C, 0x23344556);
udm.wr32(0x10000030, 0x05050505);
udm.wr32(0x10000034, 0x07070707);
udm.wr32(0x10000038, 0x99999999);
udm.wr32(0x1000003C, 0xbadc0ffe);

# fetching results
print("csr_max_elem_out:  ", hex(udm.rd32(0x20000000)))
print("csr_max_index_out: ", hex(udm.rd32(0x20000004)))
```

Listing 5 HW test program in Python

8. (if FPGA board available) Verify the design in FPGA

Program the design in FPGA board and make sure the design operates correctly. Output of Python program for our example is shown in Listing 6.

```
Connecting COM port...
COM port connected
Connection established, response:  0x55

csr_max_elem_out:   0xfefe8800
csr_max_index_out:  0xa
```

Listing 6 Output of HW test program in Python

Ensure that output of the HW test program matches simulation results. In our case, HW appears to work as intended.