HDU-ITMO Joint Institute
杭州电子科技大学 圣光机联合学院

Computer Systems Design

Laboratory work 1

Introduction to FPGA-based design flow

Author:

Alexander Antonov

Associate Professor

antonov@itmo.ru

2021

Contents

1. OBJECTIVES

- Students understand the design flow of FPGA-based design
- Students can use Xilinx Vivado Design Suite to implement hardware projects based on FPGA devices
- Students understand the project structure of FPGA project
- Students can manage design files, testbenches, and constraint files

2. OVERVIEW

Laboratory work 1 is aimed at understanding the design flow of FPGA-based project, its structure and role of its components. The students learn how to add custom logic using SystemVerilog Hardware Description Language, write testbenches, verify correctness of design in simulation environment, set up constraints, implement the design in FPGA device (if one is available) and collect metrics of obtained implementation. For now, the designed logic will be unoptimized, but optimization will be considered in the next Lab. This Lab sets the basis for further working with programmable logic devices.

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 combinational 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
9. Package your solution and submit to the teacher's email

5. GUIDANCE

Detailed guidance will be provided using the example of a 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 is a bus master module that "emulates" CPU host in System-on-chip (SoC) design. The block provides capability to initiate bus transactions both in simulation environment and on real board via serial port interface. This functionality is useful for initialization, communication and debug of custom cores in FPGA fabric. UDM block diagram is shown in Figure 1.

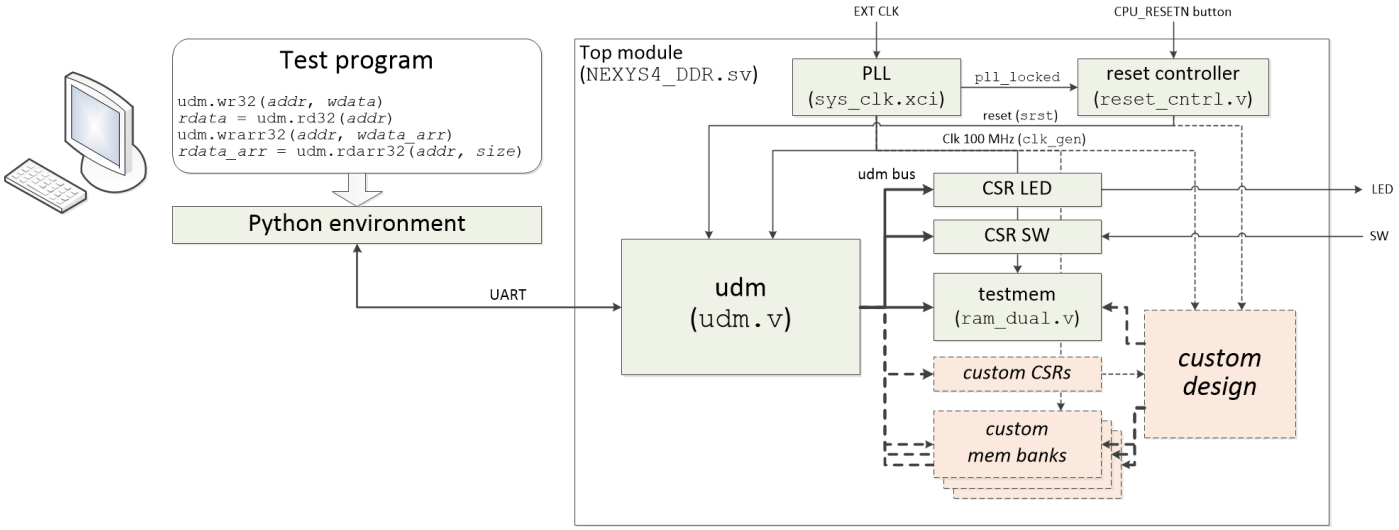


Figure 1 UDM block diagram

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 2019.2.

NOTE: avoid having non-English characters in project location path. Also, avoid very long 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 to generate bitstream. Upload the bitstream to FPGA device.

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 `udm_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 combinational module in synthesizable SystemVerilog HDL according to your variant

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

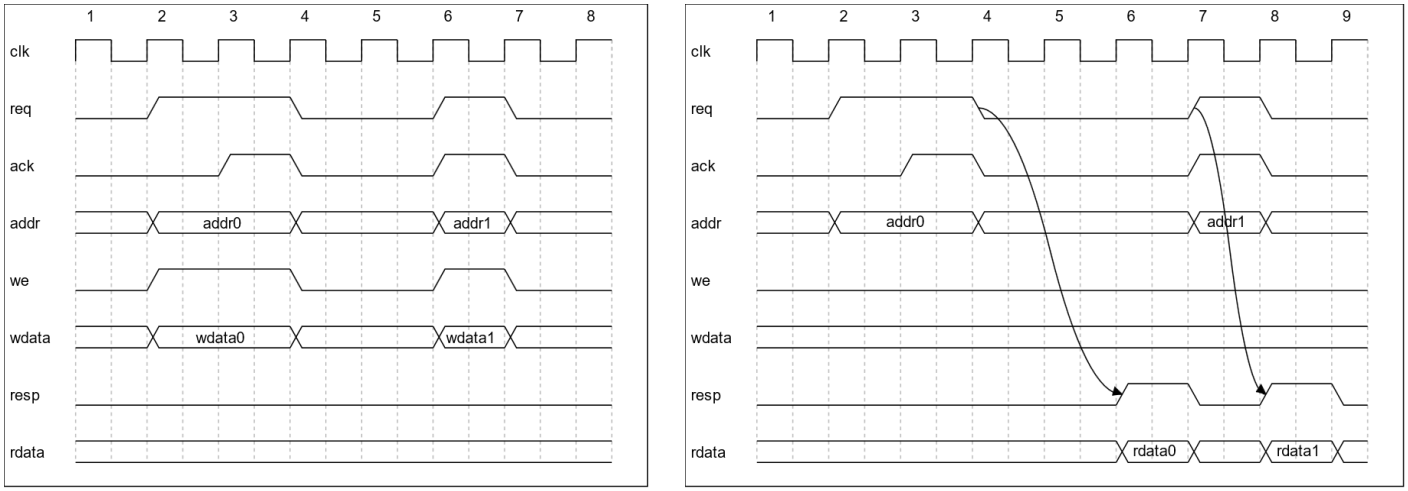
```
module FindMaxVal_comb (
    input [31:0] elem_bi [15:0]

    , output logic [31:0] max_elem_bo
    , output logic [3:0] max_index_bo
);

always @*
    begin
    max_elem_bo = elem_bi[0];
    max_index_bo = 3'd0;
    for(integer i=1; i<16; i++)
        begin
        if (elem_bi[i] > max_elem_bo)
            begin
            max_elem_bo = elem_bi[i];
            max_index_bo = i;
            end
        end
    end

endmodule
```

Listing 1 Source code of the `FindMaxVal_comb` 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 (see Figure 2), supports pipelined transactions and can easily be converted to various standard protocols (AMBA AHB, Avalon, Wishbone, etc.).



a b

Figure 2 UDM bus protocol: writing (a), reading (b)

UDM has several predefined addresses where LED and switches control and status registers (CSRs) are mapped, as well as test memory. Address map implemented in UDM baseline project is shown in Figure 3.

Address map

HW block	Start address	End address	Size
CSR_LED	0x00000000	0x00000000	4 B
CSR_SW	0x00000004	0x00000004	4 B
testmem	0x80000000	0x80000FFC	4 KB

Figure 3 Address map in UDM baseline project

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 in our example. 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_comb FindMaxVal_inst (
    .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 4).

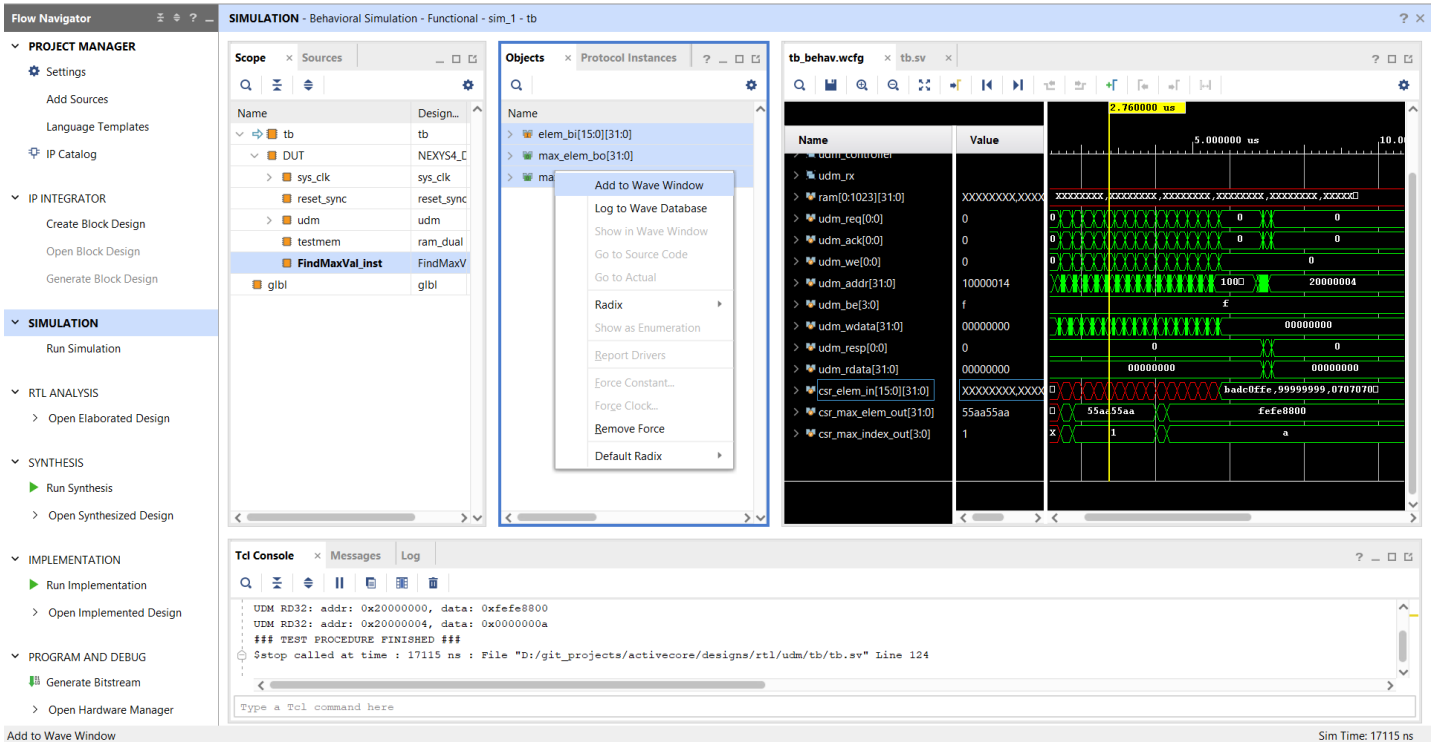


Figure 4 Adding signals to waveform

If needed, change waveform style for selected signals (digital/analog) and radix (binary, hexadecimal, decimal, etc), see Figure 5.

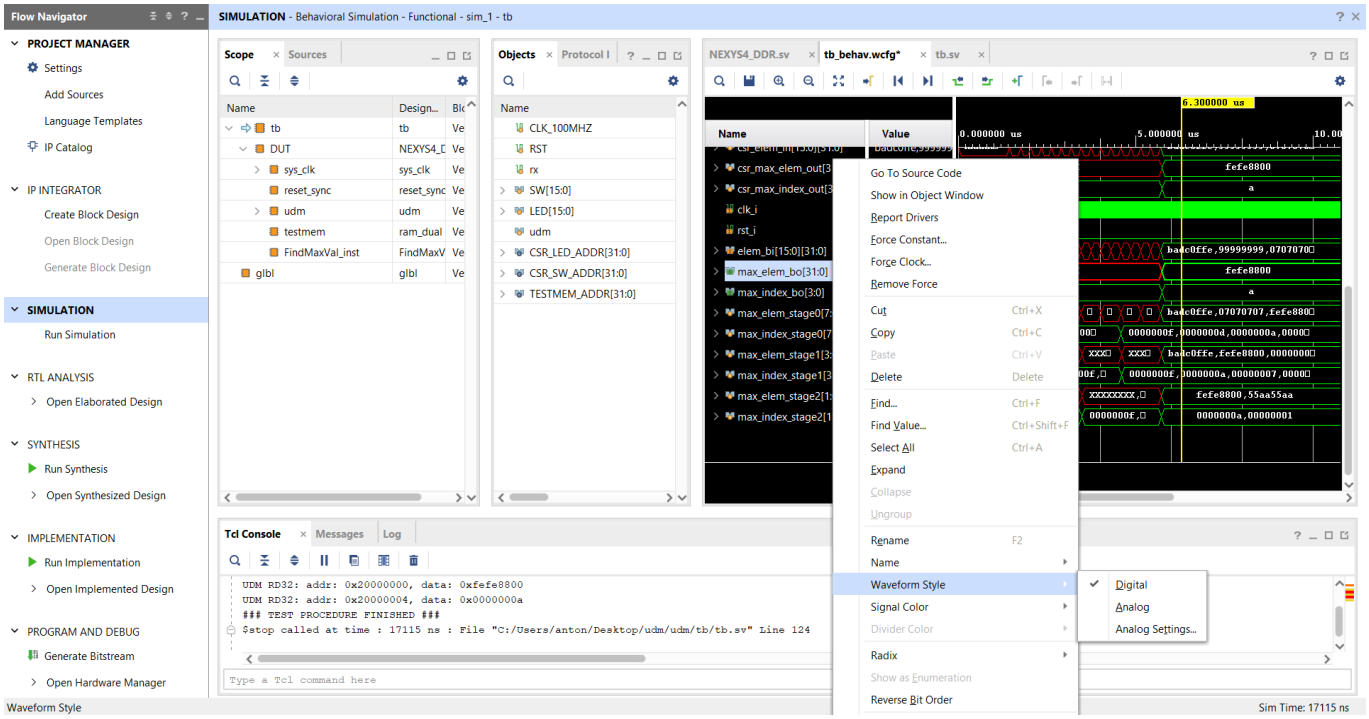


Figure 5 Waveform configuration

NOTE: To speed up UDM simulation, serial connection is bypassed. Keep in mind that UART is a low-speed interface, and transactions will take more time to complete in hardware than shown in simulation.

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 6.

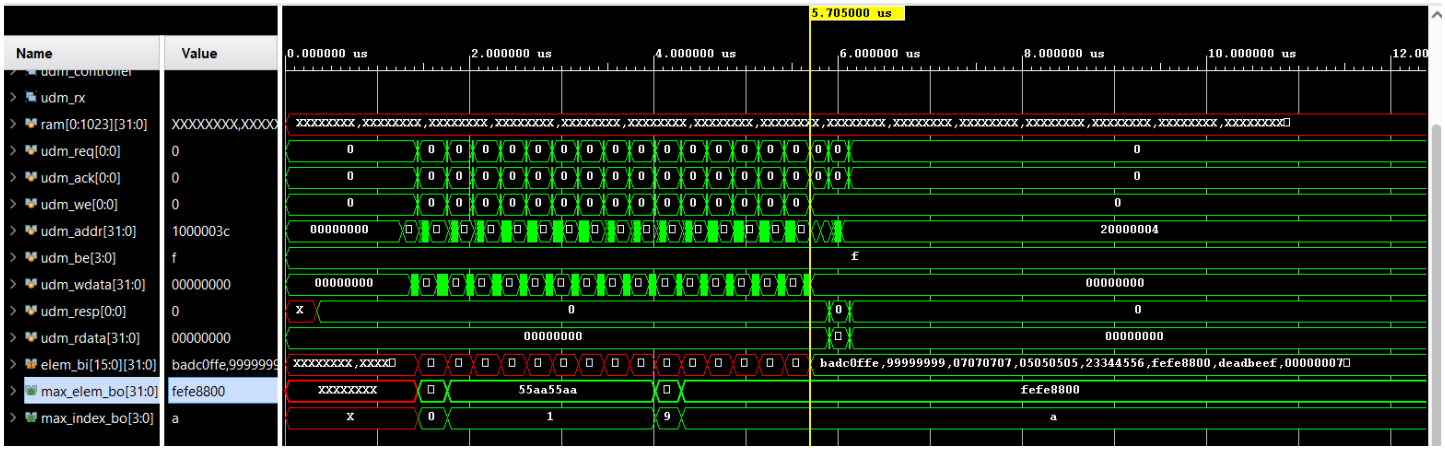


Figure 6 Waveform of simulation

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.

The following metrics are interesting:

- Timing:
  - WNS: -11.022 ns (not fine)
  - TNS: -331.079 ns (not fine)
- Performance:
  - Clock frequency: 10 ns (100 MHz clock) + 11 ns (WNS) = 21 ns (47 MHz)
  - Initiation Interval: 1 clock cycle; 21 ns
  - Bandwidth: 1 op/cycle; 47 Mop/second
  - Latency: 1 clock cycle; 21 ns
- HW resources (Implementation → Open Implemented Design → Report Utilization):
  - LUTs: may be empty for a trivial module
  - FFs (registers): may be empty for a trivial module

Though the designed hardware is functionally correct, timing closure has **failed** since we have designed a combinational circuit – a trivial, unoptimized implementation where the **entire** computation is executed in a **single** clock cycle.

In the next Lab, we will improve implementation via breaking the initial computation into subcomputations and scheduling them according to multi-cycle and pipelined approaches.

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) Program the design in FPGA board and make sure the design operates correctly

Output of Python program 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.

9. Package your solution and submit to the teacher's email

Now submit your solution to the teacher. Contents of the package:

1. Report on laboratory work
    a. Title page: course, student's name (Chinese and English), lab work number, date, variant number
    b. Screenshots of obtained simulation waveforms
    c. Report on module characteristics:
        i. timing: TNS, WNS
        ii. module's performance: clock frequency, bandwidth (initiation interval, operations/second), latency (cycles, seconds)
        iii. HW resources: LUTs, FFs
    d. Comments on achieved characteristics
2. ActiveCore distribution including your Vivado project

Please submit to: antonov@itmo.ru

6. VARIANTS

1. Higher-threshold counter

Description: count the number of elements that are larger than threshold

Input data: array of values (8x 32-bit integers), threshold (32-bit integer)

Output data: number elements that are bigger than threshold value

Python model:

```
# hw model - synthesizable operations only
def higher_threshold(datain, threshold):
    higher_counter = 0
    higher_mask = [0] * len(datain)

    # stage 0
    for datain_index in range(0, len(datain)):
        if datain[datain_index] > threshold:
            higher_mask[datain_index] = 1

    # stage 1
    for higher_index in range(0, len(datain)):
        if higher_mask[higher_index]:
            higher_counter = higher_counter + 1

    return higher_counter

# generating test stimulus
x = []
for i in range(0, 8):
    xnew = i*500000000
    x.append(xnew)
    print("x: " + '0x{:08x}'.format(xnew))

threshold = 0x80000000
print("higher than " + '0x{:08x}'.format(threshold) + ": " + str(higher_threshold(x,
threshold)))
```

Python model output:

```
x: 0x00000000
x: 0x1dcd6500
x: 0x3b9aca00
x: 0x59682f00
x: 0x77359400
x: 0x9502f900
x: 0xb2d05e00
x: 0xd09dc300
higher than 0x80000000: 3
```

2. Lower-threshold counter

Description: count the number of elements that are lower than threshold

Input data: array of values (8x 32-bit integers), threshold (32-bit integer)

Output data: number elements that are smaller than threshold value

Python model:

```
# hw model - synthesizable operations only
def lower_threshold(datain, threshold):
    lower_counter = 0
    lower_mask = [0] * len(datain)

    # stage 0
    for datain_index in range(0, len(datain)):
        if datain[datain_index] < threshold:
            lower_mask[datain_index] = 1

    # stage 1
    for lower_index in range(0, len(datain)):
        if lower_mask[lower_index]:
            lower_counter = lower_counter + 1

    return lower_counter

# generating test stimulus
x = []
for i in range(0, 8):
    xnew = i*500000000
    x.append(xnew)
    print("x: " + '0x{:08x}'.format(xnew))

threshold = 0x80000000
print("lower than " + '0x{:08x}'.format(threshold) + ": " + str(lower_threshold(x, threshold)))
```

Python model output:

```
x: 0x00000000
x: 0x1dcd6500
x: 0x3b9aca00
x: 0x59682f00
x: 0x77359400
x: 0x9502f900
x: 0xb2d05e00
x: 0xd09dc300
lower than 0x80000000: 5
```

3. Most significant one

Description: identify the position of most significant one in an integer

Input data: argument value (32-bit integer)

Output data: position of most significant one (32-bit integer)

Python model:

```
# hw model - synthesizable operations only
def anlz_byte(src_byte):
    wrk_byte = src_byte
    msb_detected = False
    msb = 0
    for i in range(0, 8):
        if (wrk_byte & 0x1) != 0:
            msb_detected = True
            msb = i
        wrk_byte = wrk_byte >> 1
    return msb_detected, msb

def msb(x):
    byte_msb = [0, 0, 0, 0]
    byte_msb_detected = [False, False, False, False]

    # stage 0: analyze most significant ones in individual bytes
    byte_msb_detected[0], byte_msb[0] = anlz_byte((x >>  0) & 0xff)
    byte_msb_detected[1], byte_msb[1] = anlz_byte((x >>  8) & 0xff)
    byte_msb_detected[2], byte_msb[2] = anlz_byte((x >> 16) & 0xff)
    byte_msb_detected[3], byte_msb[3] = anlz_byte((x >> 24) & 0xff)

    # stage 1: produce final result of analysis
    msb = 0
    for i in range(0, 4):
        if byte_msb_detected[i]:
            msb = byte_msb[i] + (i << 3)

    return msb

# generating test stimulus
for i in range(0, 16):
    x = pow(2, (i*2)+1) + i*333
    print("x: " + '0x{:08x}'.format(x) + ", msb: " + str(msb(x)))
```

Python model output:

```
x: 0x00000002, msb: 1
x: 0x00000155, msb: 8
x: 0x000002ba, msb: 9
x: 0x00000467, msb: 10
x: 0x00000734, msb: 10
x: 0x00000e81, msb: 11
x: 0x000027ce, msb: 13
x: 0x0000891b, msb: 15
x: 0x00020a68, msb: 17
x: 0x00080bb5, msb: 19
x: 0x00200d02, msb: 21
x: 0x00800e4f, msb: 23
x: 0x02000f9c, msb: 25
x: 0x080010e9, msb: 27
x: 0x20001236, msb: 29
x: 0x80001383, msb: 31
```

4. Least significant zero

Description: identify the position of least significant zero in an integer

Input data: argument value (32-bit integer)

Output data: position of least significant zero (32-bit integer)

Python model:

```
# hw model - synthesizable operations only
def anlz_byte(src_byte):
    wrk_byte = src_byte
    lsb_detected = False
    lsb = 0
    for i in range(7, -1, -1):
        if (wrk_byte & 0x80) == 0:
            lsb_detected = True
            lsb = i
        wrk_byte = wrk_byte << 1
    return lsb_detected, lsb

def lsb(x):
    byte_lsb = [0, 0, 0, 0]
    byte_lsb_detected = [False, False, False, False]

    # stage 0: analyze least significant zeroes in individual bytes
    byte_lsb_detected[0], byte_lsb[0] = anlz_byte((x >>  0) & 0xff)
    byte_lsb_detected[1], byte_lsb[1] = anlz_byte((x >>  8) & 0xff)
    byte_lsb_detected[2], byte_lsb[2] = anlz_byte((x >> 16) & 0xff)
    byte_lsb_detected[3], byte_lsb[3] = anlz_byte((x >> 24) & 0xff)

    # stage 1: produce final result of analysis
    lsb = 0
    for i in range(3, -1, -1):
        if byte_lsb_detected[i]:
            lsb = byte_lsb[i] + (i << 3)

    return lsb

# generating test stimulus
for i in range(0, 16):
    x = pow(2, (i*2)+1) + i*333
    x = x | (x >> 1) | (x >> 2)
    print("x: " + '0x{:08x}'.format(x) + ", lsb: " + str(lsb(x)))
```

Python model output:

```
x: 0x00000003, lsb: 2
x: 0x000001ff, lsb: 9
x: 0x000003ff, lsb: 10
x: 0x0000077f, lsb: 7
x: 0x000007ff, lsb: 11
x: 0x00000fe1, lsb: 1
x: 0x00003fff, lsb: 14
x: 0x0000efdf, lsb: 5
x: 0x00038ffe, lsb: 0
x: 0x000e0fff, lsb: 12
x: 0x00380fc3, lsb: 2
x: 0x00e00fff, lsb: 12
x: 0x03800fff, lsb: 12
x: 0x0e001cff, lsb: 8
x: 0x38001fbf, lsb: 6
x: 0xe0001fe3, lsb: 2
```

5. Square root

Description: calculate square root of the value

Input data: argument value (32-bit integer)

Output data: square root value (32-bit integer)

Python model:

```
import math

# alg model - python math can be used here
def sqrt_alg(x):
    return math.floor(math.sqrt(x))

# hw model - synthesizable operations only
def sqrt_hw(x):
    m = 0x40000000;
    y = 0;
    while (m != 0):  # Do 16 times (16 stages)
        b = y | m;
        y >>= 1;

        if (x >= b):
            x -= b;
            y |= m;
        m >>= 2;
    return y;
# generating test stimulus
for x in range(0, 100, 10):
    alg_val = sqrt_alg(x)
    hw_val = sqrt_hw(x)
    if (alg_val == hw_val):
        print("Correct! x: ", hex(x).ljust(6), "; y: ", hex(hw_val).ljust(6))
    else:
        print("ERROR! x: ", hex(x).ljust(6), "; y(model): ", hex(alg_val).ljust(6), "
y(hw): ", hex(hw_val).ljust(6))
```

Python model output:

```
Correct! x:  0x0    ; y:  0x0
Correct! x:  0xa    ; y:  0x3
Correct! x:  0x14   ; y:  0x4
Correct! x:  0x1e   ; y:  0x5
Correct! x:  0x28   ; y:  0x6
Correct! x:  0x32   ; y:  0x7
Correct! x:  0x3c   ; y:  0x7
Correct! x:  0x46   ; y:  0x8
Correct! x:  0x50   ; y:  0x8
Correct! x:  0x5a   ; y:  0x9
```

6. Cubic root

Description: calculate cubic root of the value

Input data: argument value (32-bit integer)

Output data: cubic root value (32-bit integer)

Python model:

```
import math
import numpy

# alg model - python math can be used here
def cube_alg(x):
    return math.floor(numpy.cbrt(x))

# hw model - synthesizable operations only
def cube_hw(x):
    y = 0;
    for s in range(30, -3, -3):  # Do 10 times (10 stages)
        y = 2*y;
        b = (3*y*(y + 1) + 1) << s;
        if (x >= b):
            x = x - b;
            y = y + 1;
    return y;

# generating test stimulus
for x in range(0, 100, 10):
    alg_val = cube_alg(x)
    hw_val = cube_hw(x)
    if (alg_val == hw_val):
        print("Correct! x: ", hex(x).ljust(6), "; y: ", hex(hw_val).ljust(6))
    else:
        print("ERROR! x: ", hex(x).ljust(6), "; y(model): ", hex(alg_val).ljust(6), "; y(hw): ", hex(hw_val).ljust(6))
```

Python model output:

```
Correct! x:  0x0    ; y:  0x0
Correct! x:  0xa    ; y:  0x2
Correct! x:  0x14   ; y:  0x2
Correct! x:  0x1e   ; y:  0x3
Correct! x:  0x28   ; y:  0x3
Correct! x:  0x32   ; y:  0x3
Correct! x:  0x3c   ; y:  0x3
Correct! x:  0x46   ; y:  0x4
Correct! x:  0x50   ; y:  0x4
Correct! x:  0x5a   ; y:  0x4
```

7. BCD adder

Description: add two binary-coded decimal values

Input data: two 16-bit BCD integers (4 BCD digits)

Output data: one 16-bit BCD integer (4 BCD digits)

Python model:

```
def hex_to_bcd(value):
    digits = []
    for i in range(0, 4):
        digits.append((value >> (i << 2)) & 0xf)
    return digits

def bcd_to_hex(digits):
    value = 0
    for i in range(0, 4):
        value = value + (digits[i] << (i << 2))
    return value

# hw model - synthesizable operations only
def add_bcd_digits(digit0, digit1, carry_in):
    carry = 0
    result = digit0 + digit1 + carry_in
    if result > 9:
        carry = 1
        result = result - 10
    return carry, result

def add_bcd(val0, val1):

    val0_digits = hex_to_bcd(val0)
    val1_digits = hex_to_bcd(val1)
    res_digits = []
    carry = 0

     # 4 stages: one for each byte
    for i in range(0, 4):
        carry, result = add_bcd_digits(val0_digits[i], val1_digits[i], carry)
        res_digits.append(result)

    return bcd_to_hex(res_digits)

# generating test stimulus
for i in range(0, 3):
    x0 = i + ((i+1) << 4) + ((i+2) << 8) + ((i+3) << 12)
    x1 = (i+6) + ((i+4) << 4) + ((i+3) << 8) + (i << 12)
    print("x0: " + '{:04x}'.format(x0) + ", x1: " + '{:04x}'.format(x1) + ", sum: " +
str(hex(add_bcd(x0, x1))))
```

Python model output:

```
x0: 3210, x1: 0346, sum: 0x3556
x0: 4321, x1: 1457, sum: 0x5778
x0: 5432, x1: 2568, sum: 0x8000
```

8. Bitonic sort

Description: sort array of values

Input data: array of values (8x 32-bit integers)

Output data: sorted array of values (8x 32-bit integers)

Python model:

```
import math
import numpy

# alg model - python math can be used here
def sort_alg(x):
    y = x[:];
    y.sort();
    return y;

# hw model - synthesizable operations only
def sort_asc(x, index0, index1):
    if (x[index0] > x[index1]):
        x[index0], x[index1] = x[index1], x[index0]

def bitonic_sort_8_hw(x):
    y = x[:]

    # stage 1
    sort_asc(y, 0, 1)
    sort_asc(y, 3, 2)
    sort_asc(y, 4, 5)
    sort_asc(y, 7, 6)

    # stage 2
    sort_asc(y, 0, 2)
    sort_asc(y, 1, 3)
    sort_asc(y, 6, 4)
    sort_asc(y, 7, 5)

    sort_asc(y, 0, 1)
    sort_asc(y, 2, 3)
    sort_asc(y, 5, 4)
    sort_asc(y, 7, 6)

    # stage 3
    sort_asc(y, 0, 4)
    sort_asc(y, 1, 5)
    sort_asc(y, 2, 6)
    sort_asc(y, 3, 7)

    sort_asc(y, 0, 2)
    sort_asc(y, 1, 3)
    sort_asc(y, 4, 6)
    sort_asc(y, 5, 7)

    sort_asc(y, 0, 1)
    sort_asc(y, 2, 3)
```

```
    sort_asc(y, 4, 5)
    sort_asc(y, 6, 7)
    return y;

# generating test stimulus
x = [51, 160, 4, 77, 194, 223, 13, 84]

y_alg = sort_alg(x)
y_hw = bitonic_sort_8_hw(x)

for index in range(0, 8):
    if (y_alg[index] == y_hw[index]):
        print("Correct!     index:     ",     hex(index).ljust(6),     ";     y:     ",
hex(y_hw[index]).ljust(6))
    else:
        print("ERROR!    index:    ",    hex(index).ljust(6),    ";    y(model):    ",
hex(y_alg[index]).ljust(6), "; y(hw): ", hex(y_hw[index]).ljust(6))
```

Python model output:

```
Correct! index:  0x0    ; y:  0x4
Correct! index:  0x1    ; y:  0xd
Correct! index:  0x2    ; y:  0x33
Correct! index:  0x3    ; y:  0x4d
Correct! index:  0x4    ; y:  0x54
Correct! index:  0x5    ; y:  0xa0
Correct! index:  0x6    ; y:  0xc2
Correct! index:  0x7    ; y:  0xdf
```

9. Sine wave

Description: generate sine value using first three terms of Taylor series

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots.$$

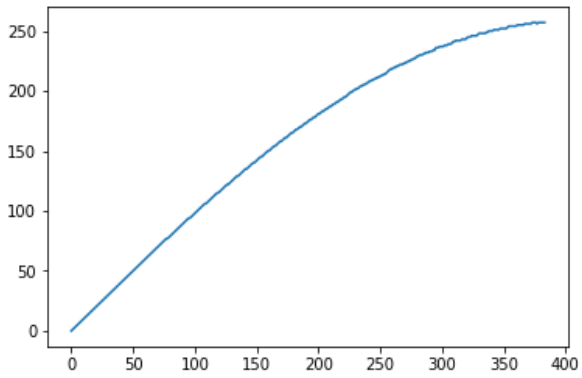
Input data: argument value (16-bit, fixed point: 0x00.00)

Output data: sine value (16-bit, fixed point: 0x00.00)

Python model:

```
import matplotlib.pyplot as plt

# hw model - synthesizable operations only
def sin_hw(x):

    div6 = 0x2a
    div120 = 0x02
    term0 = x

    # stage 0
    pow2 = (x * x) >> 8

    # stage 1
    pow3 = (pow2 * x) >> 8

    # stage 2
    term1 = (pow3 * div6) >> 8
    pow5 = (pow2 * pow3) >> 8

    # stage 3
    term2 = (pow5 * div120) >> 8
    term0_minus_term1 = term0 - term1

    # stage 4
    y = term0_minus_term1 + term2
    return y;

# generating test stimulus
x = []
y = []
for index in range(0, 384, 1):
    x.append(index)
    y.append(sin_hw(index))

plt.plot(x, y)
plt.show()
```

Python model output:



10. Cosine wave

Description: generate cosine value using CORDIC method

Input data: argument value (32-bit integer, interpreted as fixed point: 0x0000.0000)

Output data: cosine value (32-bit integer, interpreted as fixed point: 0x0000.0000)

Python model:

```
import matplotlib.pyplot as plt

# hw model - synthesizable operations only
def shift_right_arith(val, n):        # right arithmetic shift - use ">>>" operation in SystemVerilog
    return val * 2.0**(-n)

def cordic_iteration(angle):

    atan_table = [0xc910, 0x76b2, 0x3eb7, 0x1fd6,
                  0x0ffb, 0x07ff, 0x0400, 0x0200,
                  0x0100, 0x0080, 0x0040, 0x0020,
                  0x0010, 0x0008, 0x0004, 0x0002]

    x = 65536
    y = 0
    z = angle

    for i in range(0, 16):
        if z <= 0:
            d = -1
        else:
            d = 1

        nextx = x - shift_right_arith((y * d), i)
        nexty = y + shift_right_arith((x * d), i)

        x = nextx
        y = nexty
        z = z - (d * atan_table[i])
    return x

# generating test stimulus
x = []
y = []
for index in range(0, 98304, 128):
    x.append(index)
    y.append(cordic_iteration(index))

plt.plot(x, y)
plt.show()
```

Python model output: