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

Computer Systems Design

Laboratory work 2

Hardware optimization to PPA constraints

Author:

Alexander Antonov

Associate Professor

antonov@itmo.ru

2021

Contents

1. OBJECTIVES

- Students can design hardware logic according to multi-cycle approach
- Students can design hardware logic according to pipelined approach
- Students can use Xilinx Vivado Design Suite to gather metrics from implemented design
- Students understand how to achieve PPA trade-off using multi-cycle and pipelined approaches

2. OVERVIEW

Laboratory work 2 covers designing optimized versions of hardware units according to various PPA (performance, power, area) constraints. Students have to re-design previous version of their hardware from Lab 1 according to multi-cycle and pipelined approaches. These approaches are ubiquitously used in real-world designs – in particular, almost all high-performance designs (CPUs, accelerators, interconnects, memory hierarchy components, etc.) have pipelined organization.

3. PREREQUISITES

Prerequisites are the same as for Lab 1:

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. Design multi-cycle implementation in synthesizable SystemVerilog HDL according to your variant
2. Integrate your design with UDM bus master module
3. Write the testbench and simulate to verify correctness of your design
4. Implement your design, gather and analyze metrics of the implementation
5. (if FPGA board available) Perform HW testing on FPGA board
6. Design pipelined implementation in synthesizable SystemVerilog HDL according to your variant
7. Integrate your design with UDM bus master module
8. Write the testbench and simulate to verify correctness of your design
9. Implement your design, gather and analyze metrics of the implementation
10. (if FPGA board available) Perform HW testing on FPGA board
11. (optional) Evaluate top achievable frequency for the designed implementations
12. (optional) Evaluate power consumption of your designs using Vivado power analysis tool
13. Package your solution and submit to the teacher's email

5. GUIDANCE

Detailed guidance will be provided using the example of a module with the same functionality as in Lab 1.

1. Design multi-cycle implementation in synthesizable SystemVerilog HDL according to your variant

Multi-cycle implementation requires the computation to be divided into **repeated** subcomputations that will **reuse** hardware resources.

Now we develop the **schedule** for our multi-cycle implementation:

c-step number	operation
0	check 1st element
1	check 2nd element
2	check 3rd element
…	…
15	check 15th element

Table 1 Schedule for multi-cycle implementation

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

NOTE: now the module has **state** (**registers**) inside, so we have added **clock signal** (`clk_i`) and **reset signal** (`rst_i`).

```
module FindMaxVal_multicycle (
    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
);

localparam CSTEP_NUM = 16;      // number of computational stages
logic [31:0] max_elem_next;
logic [3:0] max_index_next;
logic [3:0] cstep_counter, cstep_counter_next;
logic [31:0] current_max_elem, current_max_elem_next;
logic [3:0] current_max_index, current_max_index_next;

// subcomputation combinational logic
always @*
    begin

    // default values assignments
    cstep_counter_next = cstep_counter;
    current_max_elem_next = current_max_elem;
    current_max_index_next = current_max_index;
    max_elem_next = max_elem_bo;
    max_index_next = max_index_bo;

    // subcomputation on current cstep
    if (elem_bi[cstep_counter_next] > current_max_elem_next)
        begin
        current_max_elem_next = elem_bi[cstep_counter_next];
```

```
        current_max_index_next = cstep_counter_next;
        end

    // cstep processing
    if (cstep_counter_next == (CSTEP_NUM-1))
        begin
        max_elem_next = current_max_elem_next;
        max_index_next = current_max_index_next;
        current_max_elem_next = 0;
        current_max_index_next = 0;
        cstep_counter_next = 0;
        end
    else
        begin
        cstep_counter_next = cstep_counter_next + 1;
        end
    end

// sequential logic (registers assignments)
always @(posedge clk_i)
    begin
    if (rst_i)
        begin
        cstep_counter <= 0;
        current_max_elem <= 0;
        current_max_index <= 0;
        max_elem_bo <= 0;
        max_index_bo <= 0;
        end
    else
        begin
        cstep_counter <= cstep_counter_next;
        current_max_elem <= current_max_elem_next;
        current_max_index <= current_max_index_next;
        max_elem_bo <= max_elem_next;
        max_index_bo <= max_index_next;
        end
    end

endmodule
```

Listing 1 Source code of `FindMaxVal_multicycle` module in SystemVerilog HDL

2. Integrate the custom design with UDM bus master module

The CSR interface is the same as for combinational implementation.

NOTE: don't forget to connect clock and reset signal to `clk_gen` and `srst` signals respectively (changed lines 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_multicycle 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

3. Write the testbench and simulate to verify correctness of the design

The test procedure is the same as for combinational implementation. Waveform for the simulation is shown in Figure 1.

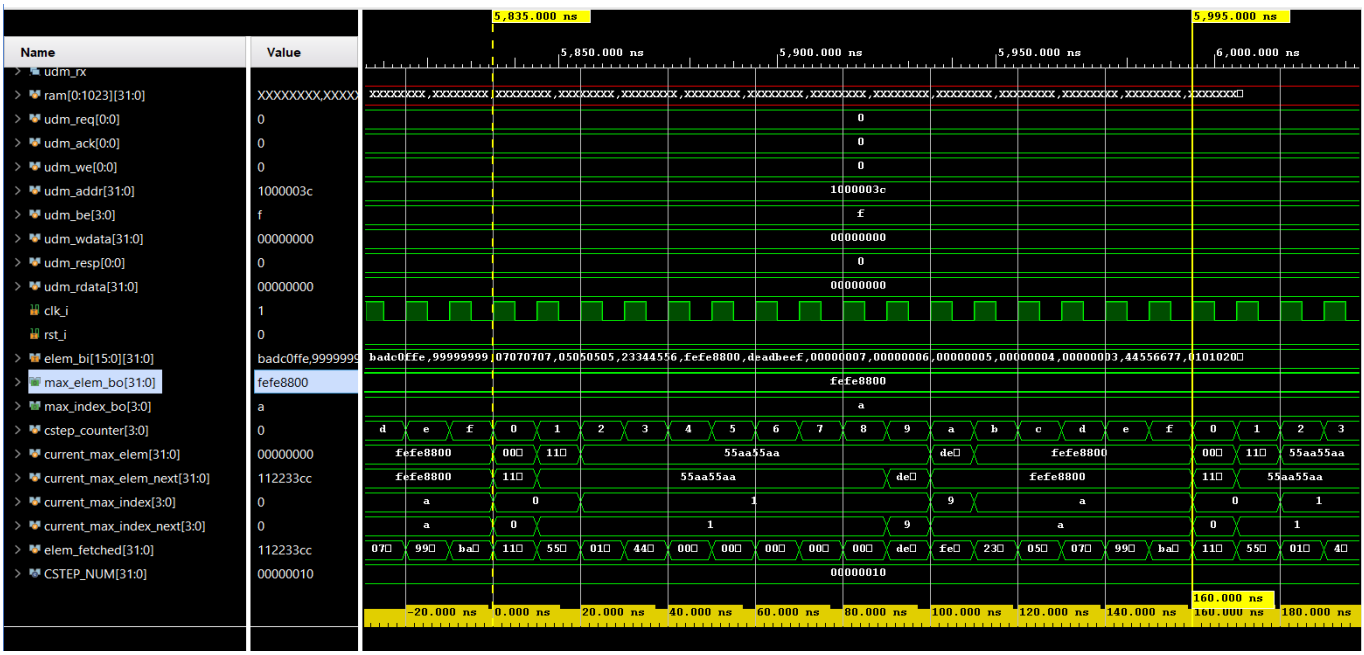

Figure 1 Simulation waveform for multi-cycle implementation

In the waveform, cstep-by-cstep progression of computation managed by `cstep_counter` can be observed. The simulation is correct, the module works as intended.

NOTE: Consider stage-by-stage debugging of multi-cycle implementation in case output result is incorrect.

4. Implement the design, gather 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: 3.526 ns (fine)
  - TNS: 0 ns (fine)
- Performance:
  - Clock frequency: 10 ns (100 MHz)
  - Initiation Interval: 16 clock cycles (equal to schedule length); 160 ns
  - Throughput: 0,0625 op/cycle; 6,25 Mop/second
  - Latency: 16 clock cycles (equal to schedule length); 160 ns
- HW resources (Implementation → Open Implemented Design → Report Utilization):
  - LUTs: 149
  - FFs (registers): 76

The timing closure is **successful** since our multi-cycle circuit analyses only a **single** element within a **single** clock cycle. However, it requires **16 clock cycles** to finish the computation (instead of one clock cycle for combinational implementation).

5. (if FPGA board available) Perform HW testing on FPGA board

Python tests and HW testing procedure are the same as in Lab 1.

6. Design pipelined implementation in synthesizable SystemVerilog HDL according to your variant

Pipelined implementation requires the computation to be divided into **stages** that execute in **overlapped** fashion.

Now we develop the **schedule** for our pipelined implementation:

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 2 Schedule for pipelined implementation

Source code for the example module in shown in Listing 3.

```
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 ////

// intermediate signals declaration
logic [31:0] max_elem_stage0  [7:0];
logic [31:0] max_index_stage0 [7:0];
logic [31:0] max_elem_stage0_next  [7:0];
logic [31:0] max_index_stage0_next [7:0];

// combinational logic
always @*
    begin
    for(integer i=0; i<8; i++)
        begin
        max_elem_stage0_next[i] = 0;
        max_index_stage0_next[i] = 0;
        if (elem_bi[(i<<1)] > elem_bi[(i<<1)+1])
            begin
            max_elem_stage0_next[i] = elem_bi[(i<<1)];
            max_index_stage0_next[i] = i<<1;
            end
        else
            begin
            max_elem_stage0_next[i] = elem_bi[(i<<1)+1];
            max_index_stage0_next[i] = (i<<1)+1;
            end
       end
    end

// writing to registers
always @(posedge clk_i)
    begin
    if (rst_i)
        begin
        for (integer i=0; i<8; i++) max_elem_stage0[i] <= 0;
        for (integer i=0; i<8; i++) max_index_stage0[i] <= 0;
        end
    else
        begin
        for (integer i=0; i<8; i++) max_elem_stage0[i] <= max_elem_stage0_next[i];
        for (integer i=0; i<8; i++) max_index_stage0[i] <= max_index_stage0_next[i];
        end
    end

//// stage 1 ////

// intermediate signals declaration
logic [31:0] max_elem_stage1  [3:0];
```

```
logic [31:0] max_index_stage1 [3:0];
logic [31:0] max_elem_stage1_next  [3:0];
logic [31:0] max_index_stage1_next [3:0];

// combinational logic
always @*
    begin
    for(integer i=0; i<4; i++)
        begin
        max_elem_stage1_next[i] = 0;
        max_index_stage1_next[i] = 0;
        if (max_elem_stage0[(i<<1)] > max_elem_stage0[(i<<1)+1])
            begin
            max_elem_stage1_next[i] = max_elem_stage0[(i<<1)];
            max_index_stage1_next[i] = max_index_stage0[(i<<1)];
            end
        else
            begin
            max_elem_stage1_next[i] = max_elem_stage0[(i<<1)+1];
            max_index_stage1_next[i] = max_index_stage0[(i<<1)+1];
            end
        end
    end

// writing to registers
always @(posedge clk_i)
    begin
    if (rst_i)
        begin
        for (integer i=0; i<4; i++) max_elem_stage1[i] <= 0;
        for (integer i=0; i<4; i++) max_index_stage1[i] <= 0;
        end
    else
        begin
        for (integer i=0; i<4; i++) max_elem_stage1[i] <= max_elem_stage1_next[i];
        for (integer i=0; i<4; i++) max_index_stage1[i] <= max_index_stage1_next[i];
        end
    end

//// stage 2 ////

// intermediate signals declaration
logic [31:0] max_elem_stage2  [1:0];
logic [31:0] max_index_stage2 [1:0];
logic [31:0] max_elem_stage2_next  [1:0];
logic [31:0] max_index_stage2_next [1:0];

// combinational logic
always @*
    begin
    for(integer i=0; i<2; i++)
        begin
        max_elem_stage2_next[i] = 0;
        max_index_stage2_next[i] = 0;
        if (max_elem_stage1[(i<<1)] > max_elem_stage1[(i<<1)+1])
            begin
            max_elem_stage2_next[i] = max_elem_stage1[(i<<1)];
            max_index_stage2_next[i] = max_index_stage1[(i<<1)];
```

```
            end
        else
            begin
            max_elem_stage2_next[i] = max_elem_stage1[(i<<1)+1];
            max_index_stage2_next[i] = max_index_stage1[(i<<1)+1];
            end
        end
    end

// writing to registers
always @(posedge clk_i)
    begin
    if (rst_i)
        begin
        for (integer i=0; i<2; i++) max_elem_stage2[i] <= 0;
        for (integer i=0; i<2; i++) max_index_stage2[i] <= 0;
        end
    else
        begin
        for (integer i=0; i<2; i++) max_elem_stage2[i] <= max_elem_stage2_next[i];
        for (integer i=0; i<2; i++) max_index_stage2[i] <= max_index_stage2_next[i];
        end
    end

//// stage 3 ////

// intermediate signals declaration
logic [31:0] max_elem_next;
logic [3:0] max_index_next;

// combinational logic
always @*
    begin
    max_elem_next = 0;
    max_index_next = 0;
    if (max_elem_stage2[0] > max_elem_stage2[1])
        begin
        max_elem_next = max_elem_stage2[0];
        max_index_next = max_index_stage2[0];
        end
    else
        begin
        max_elem_next = max_elem_stage2[1];
        max_index_next = max_index_stage2[1];
        end
    end

// writing to registers
always @(posedge clk_i)
    begin
    if (rst_i)
        begin
        max_elem_bo <= 0;
        max_index_bo <= 0;
        end
    else
        begin
        max_elem_bo <= max_elem_next;
```

```
        max_index_bo <= max_index_next;
        end
    end

endmodule
```

Listing 3 Source code of `FindMaxVal_pipelined` module in SystemVerilog HDL

7. Integrate the custom design with UDM bus master module

Integration is the same as for multi-cycle implementation with only one difference: instantiate `FindMaxVal_pipelined` module instead of `FindMaxVal_multicycle` in `NEXYS4_DDR` top module.

8. Write the testbench and simulate to verify correctness of the design

The test procedure is the same as for multi-cycle implementation. Waveform for the simulation is shown in Figure 2.

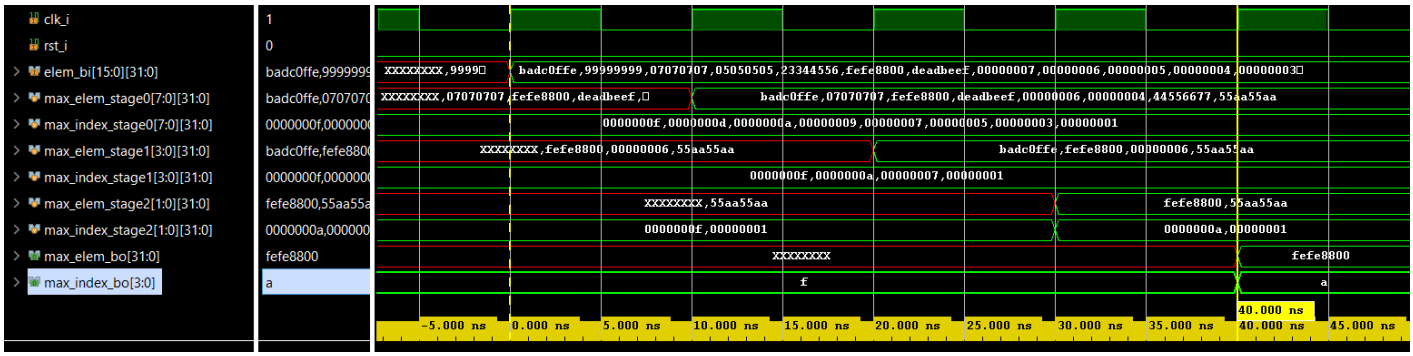

Figure 2 Simulation waveform for pipelined implementation

In the waveform, stage-by-stage propagation of data through the pipelined datapath can be observed. The simulation is correct, the module works as intended.

NOTE: Consider stage-by-stage debugging of pipelined implementation in case output result is incorrect.

9. Implement the design, gather 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: 4 clock cycles (equal to schedule length); 160 ns
- HW resources (Implementation → Open Implemented Design → Report Utilization):

- LUTs: 498
- FFs (registers): 506

The timing closure is **successful** since our pipelined circuit analyses only a **several element pairs in parallel** within a **single** clock cycle. It requires **4 clock cycles** to finish the computation (instead of one clock cycle for combinational implementation). However, since these subcomputations go in overlapped fashion, we can pass new computation each cycle and achieve top throughput of 100 Mop/second.

10. (if FPGA board available) Perform HW testing on FPGA board

Python tests and HW testing procedure are the same as in Lab 1.

11. (optional) Evaluate top achievable frequency for the designed implementations

Increase output frequency of `sys_clk` PLL until timing starts failing for each designed implementation. Evaluate top performance of each implementation based on top frequency value when timing closure is achieved.

12. (optional) Evaluate power consumption of your designs using Vivado power analysis tool

Vivado power analysis tool is launched from “Implementation” → “Open implemented design” → “Report power”.

13. Package your solution and submit to the teacher’s email

The package content is equal to Lab 1, but should also include schedules for designed implementations.

6. VARIANTS

Same as for Lab 1.