High-Level FPGA Design

Lab 1: Averaging Filter RTL Design

Audience: Digital designers familiar with HDL, simulation, synthesis, timing
Duration: 90–120 minutes
Purpose: Establish baseline RTL reference for later conversion/comparison to HLS

1. Learning Objectives

By the end of Lab 1, participants will be able to:

- Implement a **streaming averaging filter in RTL**
- Understand **latency, throughput, and resource usage** in a concrete design
- Express the design in **clock-by-clock behavior**, preparing for HLS comparison
- Generate **post-synthesis metrics** that will later be compared to HLS results

2. Problem Statement

Functional Specification

Design a **moving average filter**:

$$y[n] = \frac{1}{N} \sum_{i=0}^{N-1} x[n-i]$$

Where:

- N = 8 samples (power-of-two on purpose)
- Input: signed 16-bit samples, parameter DATAW = 16
- Output: signed 16-bit rounded result
- Basic streaming interface: in_valid, out_valid

Interface (explicit RTL contract)

```
module lab1_avg #(
    parameter int N = 8,
    parameter int DATAW = 16
) (
    input  logic          clk;
    input  logic          rst;
    input  logic          in_valid;
    input  logic signed [DATAW-1:0] in_data;

    output logic          out_valid;
    output logic signed [DATAW-1:0] out_data
);
```

3. Suggested Architecture (RTL-friendly)

Reference Architecture

- Shift register (N=8) for sample history
- Accumulator width: 16 + $\log_2(8)$ = 19 bits
- Division by 8 = right shift by 3 bits
- Two optimized versions: adder tree and running sum (sum_next = sum + new_sample - oldest_sample)

Key RTL Concepts

- Word growth
- Pipeline latency
- Explicit control of registers
- Clock enable via in_valid

4. Step-by-Step Instructions

Step 1: Create Vivado Project

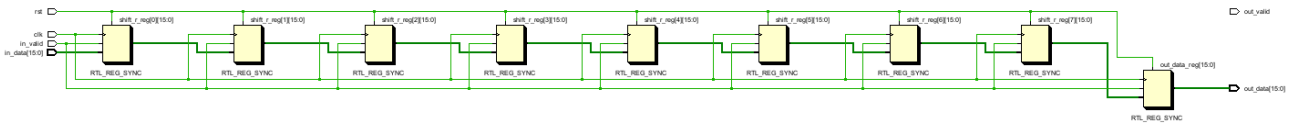
- Target Board: Red Pitaya or Device **xc7z010clg400-1**
- Add Sources, Add design sources, Add Files, lab1_avg.sv
- Add Sources, Add Constraints, constraints.xdc

Step 2: Declare Signals and Describe Shift Register

- Declare array for shift register `shift_r[N]`
- Describe data shifting in **`always_ff`** block, `@(posedge clk)` and when **in_valid**
  - Reset all registers and output when **rst** asserted
- For the first test let `out_data = shift_r(N-1)`

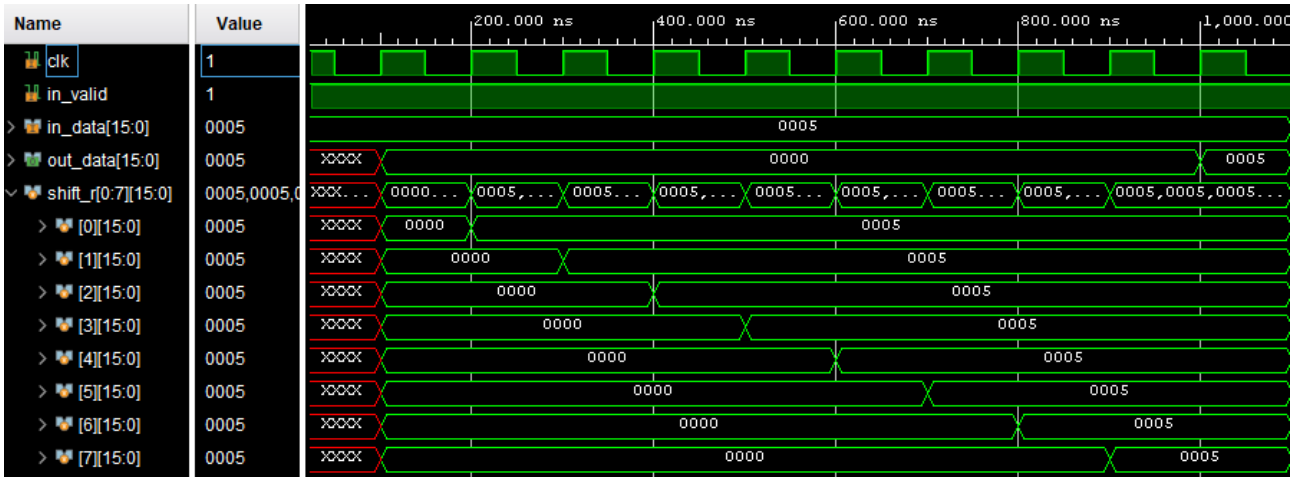
```
for (i = N-1; i > 0; i--) begin
     shift_r[i] <= shift_r[i-1];
end
shift_r[0] <= in_data;
out_data <= shift_r[N-1]; // for the first test
```

- Perform RTL Analysis, Open Elaborated Design Schematic
  - You should see 8 registers and out_data register



- Run Simulation, Run Behavioral Simulation
- Use small TCL script for setting inputs (`source ./src_lab1/sim.tcl`)

```
add_force {/lab1_avg/clk} -radix bin {1 0ns} {0 50ns} -repeat_every 100ns
add_force {/lab1_avg/rst} -radix bin {1 0ns}
add_force {/lab1_avg/in_valid} -radix bin {1 0ns}
add_force {/lab1_avg/in_data} -radix dec {5 0ns}
run 100 ns
add_force {/lab1_avg/rst} -radix bin {0 0ns}
run 1000 ns
```



Step 3: Compute average

- Declare sum, use parameter to specify width

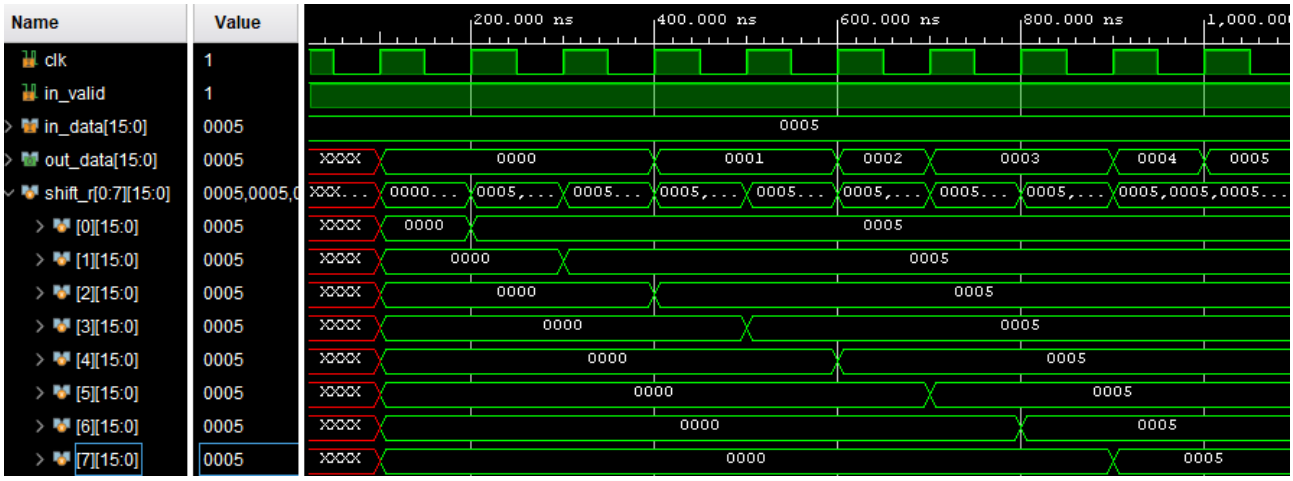
```
localparam int SUMW  = DATAW + $clog2(N);
```

- Add combinational block with for loop to describe adder

```
always_comb begin
   sum = '0;
   for (j = 0; j < N; j++) sum = sum + shift_r[j];
end
```

- Assign sum divided by 8 (right shift) to the **out_data**
- How to parametrize right shift amount?
- Check on simulation. What are expected results with this input sequence?

in_data	0	5	5	5	5	5	5	5	5
out_data	0	0	1						



Step 4: Testbench and interface

Testbench will be used for verification and automatic latency measurement.
Latency is number of clock cycles between the cycle where an input is *accepted* and the cycle where the *corresponding output* is asserted *valid*.

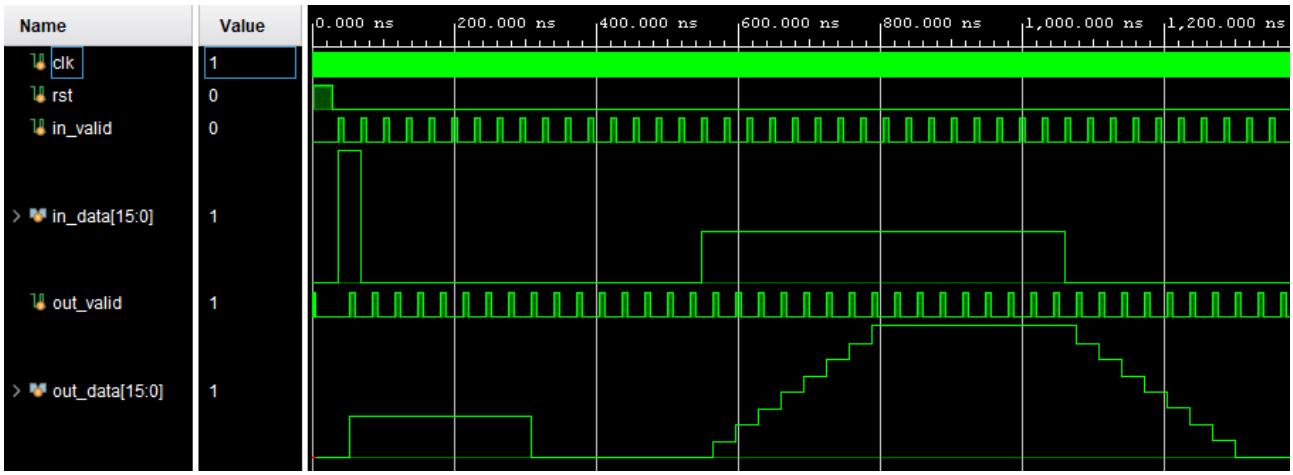
- Add Sources, Add Simulation Sources, tb_avg.sv
- The testbench performs 3 tests: impulse input, constant input and ramp
- Inital reset and toggling in_valid
- Now you should upgrade control interface to provide out_valid and uncomment atomatic checkng of out_data

- option 1: **out_valid** is asserted following **in_valid** (1 cycle latency). The average filtering result shoud be computed with combinational logic.

```
end else if (in_valid) begin
    ...
    out_valid <= 1;
end
```

- option 2: The result is stored in register (better circuit performance) and **out_valid** has additional clock cycle delay. Consider this option for your design.

Simulation with **out_valid**, 2 cycle latency



Adding one register increases latency to 2 cycles, which is not a problem in typical signal processing design. But multy-cycle computation can add requirement to send the data slowly (pause between in_valid cycles), unless the circuit is implemented with pipeline.

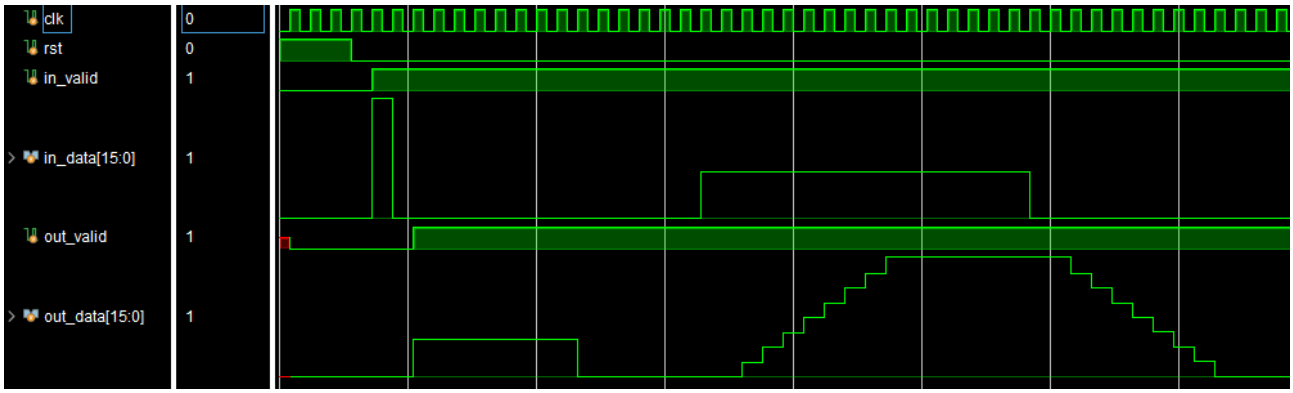
Fully pipelined circuits can accept new input at each clock cycle and produce valid results with fixed delay corresponding to the latency.

Initiation Interval (II) is minimum number of cycles between accepting consecutive inputs. Fully pipelined circuits have initiation interval II = 1.

Throughput is the *rate of completed results* produced by the design in *steady state*, given in results per second or per cycle. For optimal case, after pipeline fill, throughput = 1 / II.

- Comment automatic checking in the testbench, which will reduce pause beween input valid to 0 cycles and check operation of your circuit.
- How to create averaging filter with one cycle pipeline?

Checking pipeline performance, latency = 2, II = 1



Step 5: Evaluate implemented design performance

- Run Synthesis, Open Synthesized Design
- Check Utilization
- Check Data Path Delay with TCL command

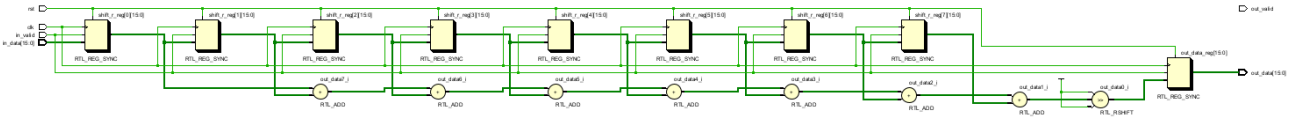
```
report_timing -delay_type max -max_paths 1
```

- Run Implementation
- Report timing and read Worst Negative Slack (WNS)
    - Note: min clock period = 8 ns – WNS

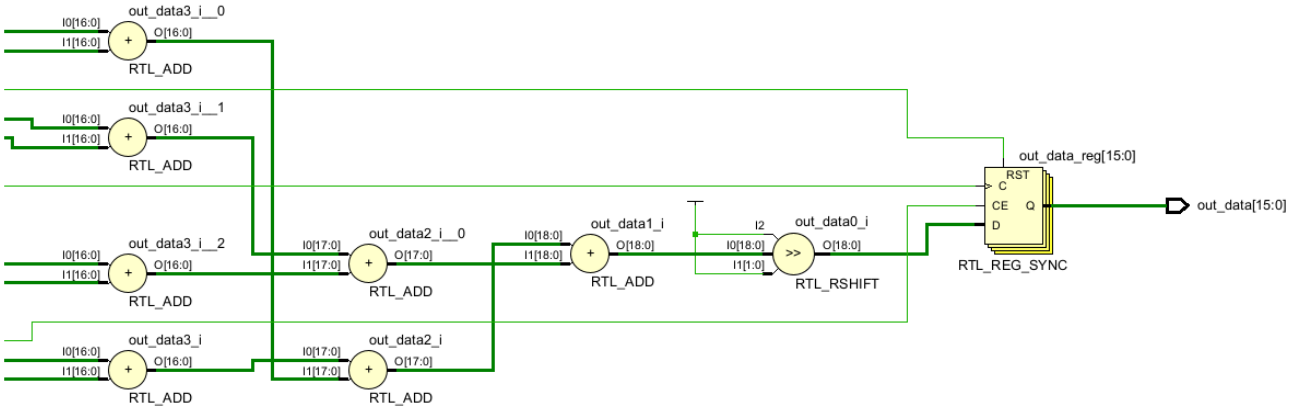
- Compute max clock frequency

FPGA Utilization	LUT	FF
Timing	Delay	Freq

- What is a problem of long sequence of adders considering performance?



- Adder tree would be better solution…



Step 6: Optimize circuit

- Optimal implementation of the averaging filter utilizes **running sum**
- Value of sum should be stored in a register

```
sum <= sum + in_data - shift_r[N-1];
```

- Simulate circuit to check operation
- Run Synthesis and Implementation
- Check utilization and timing performance

8 adders: LUT 75, FF 144, 6.3 ns, 139 MHz
tree: LUT 116, FF 144, 5.822ns, 159 MHz
Running: LUT 44, FF 74, Data Path Delay 3.402ns, 276 MHz

Step 7: Check FPGA Primitive Usage

- Compare syntesis Utilization for N=8 and N=1024
- Synthesis mapps shift register to FFs or SRL cells

Primitive	N=8	N=1024	Category
FDRE	76	1099	Flop & Latch
SRLC32E	20	512	Distributed Memory
LUT2	19	27	LUT
IBUF	17	19	IO
OBUF	16	17	IO
LUT4	16	16	LUT
LUT3	16	16	LUT
CARRY4	5	7	CarryLogic
BUFG	1	1	Clock

- Consider implementation of data storage with RAM and pointers

```
// 1. Subtract the oldest sample and add the newest
// Update running sum: NewSum = OldSum + NewSample - OutgoingSample
sum <= sum + in_data - ram[ptr];

// 2. Overwrite the oldest sample in the circular buffer (BRAM)
ram[ptr] <= in_data;

// 3. Update pointer
ptr <= ptr + 1;
```

5. *Optional Stretch Tasks

- Implement rounded division by power of 2
- Add integer divider if N is not power of 2
    - Check circuit timing
- Switch to saturation arithmetic
    - Use smaller sum register and saturation instead of overflow
- Force DSP usage and observe mapping