Hardware and Embedded Security

Prof. Maurizio Martina, PhD. Luigi Giuffrida, PhD. Alessandra Dolmeta

Lab 2: Digital Design Flow

The aim of this lab is to introduce the digital design flow using open-source Computer Aided Design (CAD) tools and technologies, building upon the concepts from the previous lab. Additionally, this lab includes a brief introduction to the concept of watermarking in hardware design, demonstrating how it can be incorporated into the synthesis process to help protect intellectual property.

1 Prerequisites

Before starting this lab, ensure that all the setup and configuration steps from the previous lab have been successfully completed. This includes the correct installation and configuration of any required tools (e.g., Yosys, Sky130 PDK, simulators, etc.) and verifying that your environment is properly set up to run synthesis and simulation flows. If you encounter any issues during this lab, carefully review and repeat steps 1 to 3 from the previous lab documentation to confirm that all prerequisites are satisfied.

2 Project structure

To make the various steps more manageable, we recommend continuing the folder structure from the previous lab. As before, use the lab folder as the main project directory, with sub-folders such as `src` for HDL sources, `tb` for test benches, `sim` for simulations, `netlist` for synthesis results, and `pnr` for place and route files.

2.1 Design

Below is the updated Verilog module for `counter_gen` that incorporates watermarking (dummy) logic. We assume it is placed in the `src` sub-folder as `counter_gen.v`.

Listing 1: n-bit counter

```
module counter_gen
  #(
    parameter NBIT = 4
  )
  (
    input                  clk,
    input                  rstn,
    output reg [NBIT-1:0] out
  );

  // ---------------------------------------------------------
  // Main counter functionality
  // ---------------------------------------------------------
  always @ (posedge clk, negedge rstn) begin
    if (!rstn)
      out <= 0;
    else
      out <= out + 1;
  end

  // ---------------------------------------------------------
  // Watermark / Dummy Logic
  // ---------------------------------------------------------

  (* keep = "true" *) reg [2:0] watermark_shift = 3'b101;

  always @ (posedge clk or negedge rstn) begin
    if (!rstn) begin
      watermark_shift <= 3'b101;     // Initial pattern
    end else begin
      // Shift left and invert the last bit as feedback
      watermark_shift <= {watermark_shift[1:0], ~watermark_shift[2]};
    end
  end


  (* keep = "true" *) wire watermark_detect = (watermark_shift == 3'b111);

endmodule
```

The central functionality of the counter remains identical to the original design. This ensures the counter still behaves exactly as before, incrementing on each clock cycle and resetting on an active-low `rstn` signal. In the updated version, a new section introduces a 3-bit shift register (`watermark_shift`) with an inverted feedback tap. This logic serves as a watermark and does not alter the primary counter's behavior.

The line `(* keep = "true" *)` is a synthesis attribute. It guides most synthesis tools (including Yosys) to avoid optimizing away or significantly altering the `watermark_shift` register or
the `watermark_detect` wire. This helps preserve the watermark in the final netlist.
`watermark_shift`. Initialized to 3'b101, and on each clock cycle it shifts left and inverts the last bit for feedback. This creates a small piece of logic that oscillates through different values, providing a unique signature visible in the synthesized netlist or physical layout.
`watermark_detect` A simple wire that checks if the shift register is at a specific pattern (3'b111 in the example). This extra signal can help confirm the presence of the watermark in the netlist or layout.

Remember! The watermark logic is a hidden feature intended to prove design ownership. It does not affect the primary functionality of the design and thus should not impact the counter's output or performance.

2.2 Logic synthesis

The first step to implement the (now watermarked) counter is logic synthesis. We assume you are in the `netlist` sub-folder. To start the logic synthesis, first invoke the logic synthesizer:

```
yosys
```

This opens the Yosys shell, where we can enter commands. Listing 2 illustrates an example flow for synthesizing the updated design—including both the main counter logic and the dummy watermark logic—and then mapping it to the standard cells.

Listing 2: Yosys synthesis commands

```
read_liberty -lib /home/user/sky130_fd_sc_hd/lib/sky130_fd_sc_hd__tt_025C_1v80.lib
read_verilog -defer ../src/counter_gen.v
chparam -set NBIT 8
synth -top counter_gen
dfflibmap -liberty /home/user/sky130_fd_sc_hd/lib/sky130_fd_sc_hd__tt_025C_1v80.lib
abc -liberty /home/user/sky130_fd_sc_hd/lib/sky130_fd_sc_hd__tt_025C_1v80.lib
write_verilog ./counter_gen.v
```

Briefly, the commands do the following:

- read_liberty: Provide Yosys with the standard-cell information from the Liberty file.
- read_verilog -defer: Read the Verilog RTL source that now contains both the counter and the embedded watermark logic (the dummy shift register or other extra logic). Using -defer defers parameter resolution until later.
- chparam: Set the `NBIT` parameter to 8 (making it an 8-bit counter).
- synth_design: Execute the primary synthesis step on the design named
- dfflibmap and abc: Map flip-flops and other gates to the standard cells available in the technology library.
- write_verilog: Write out the synthesized netlist to the file `counter_gen.v`.

Netlist Simulation To verify that the netlist behaves the same as the original (i.e., the counter still functions as it did in Lab 1), you can reuse your existing test bench and *Icarus Verilog* simulation flow. As before, you must include the file `sky130_fd_sc_hd.v` (the functional model of the standard cells) during compilation. In that file, define the symbol `FUNCTIONAL` so that it uses functional behavioral models, and also set `UNIT_TIME` to define the clock-to-output delay for flip-flops. Because the `NBIT` parameter is now fixed at synthesis time (e.g., 8 bits), if your test bench tries to reassign the parameter, you may see a warning that the parameter is already bound. This is normal. Listing 3 shows a sample script for compiling and running the netlist simulation with Icarus Verilog.

Listing 3: Netlist simulation

```
NET_DIR=../netlist
DESIGN=counter_gen
TB_DIR=../tb
TB_NAME=tb_counter_gen

TECH_NAME=sky130
TECH_TYPE=fd_sc_hd
LIB_NAME=${TECH_NAME}_${TECH_TYPE}
LIB_DIR=/home/user/${LIB_NAME}/verilog

iverilog -DFUNCTIONAL -DUNIT_DELAY=#1 -o ${TB_NAME}.vvp ${LIB_DIR}/primitives.v \
         ${LIB_DIR}/${LIB_NAME}.v ${NET_DIR}/${DESIGN}.v ${TB_DIR}/${TB_NAME}.v \
         -s ${TB_NAME}
vvp ${TB_NAME}.vvp
```

The simulation will generate `tb_counter_gen.vcd`, which you can open with *GTKWave*:

```
gtkwave tb_counter_gen.vcd
```

Within the waveforms, you should observe that the `out` signal of the synthesized netlist matches the behavior of your original counter code from Lab 1 (i.e., it properly increments and resets). The new watermark (dummy) logic exists in the synthesized netlist but does not affect the counter's functional waveforms, so the counter output behaves exactly the same as before.

2.3 Place and route

In the following, we assume that the place and route flow (including all commands) is run from the `pnr` sub-folder.

Setup and initialization Repeat steps of Lab01.

Floorplanning Repeat steps of Lab01.

Placement Repeat steps of Lab01.
When you add watermark (dummy) circuitry, your netlist contains extra cells and interconnections that were not present in the original counter-only design. Because the placer and router must accommodate these new cells, the global placement solution can differ. The additional nets also affect how routing resources are allocated and can slightly alter the overall wirelength or the final distribution of cells.
If your watermark logic is constrained to remain in certain regions or is declared "dont_move" or "dont_touch," placement tools may have fewer degrees of freedom. This can lead to changes in metrics such as total or average cell displacement. Even if you do not apply explicit placement constraints, the added watermark logic can indirectly alter how standard cells are packed and arranged. Check differences with previous results.
Tools often measure how "spread out" the design is by calculating a Half-Perimeter Wire Length (HPWL) estimate. Adding extra cells or nets may increase or decrease this metric, depending on how the placer decides to distribute the new logic in relation to existing signals. Check differences with previous results.
Additional nets mean the router has more connections to complete. If the watermark logic is clustered in one corner or forcibly placed with special constraints, this might reduce routing congestion in certain areas or cause slight congestion in others. Such effects could be visible in routing utilization maps and final wirelength measurements.
Finally, even if the watermark logic is not timing-critical, the presence of extra cells slightly alters the overall design space. The op-

timization algorithms (e.g., those used in legalizing placement or repairing design rule violations) must incorporate these extra cells, potentially leading to minor differences in overall timing, placement patterns, or design metrics.

- **Cell Placement**
  - How does adding extra, non-functional cells (i.e., watermark logic) change the distribution of cells in the floorplan?
  - Are these watermark cells placed together or dispersed among the main logic in your final layout?
- **Wirelength & Displacement**
  - What factors might cause changes in total displacement and average displacement when you include extra cells in your netlist?
  - In what ways might the placement tool "shift" existing cells to accommodate the watermark logic?
- **Timing & Constraints**
  - Does the presence of a few additional cells change the timing constraints or critical paths in any noticeable way?
  - If so, how might that shift the placement or routing decisions?
- **Inspection in OpenROAD GUI**
  - When you open the designs (original vs. watermarked) in the OpenROAD GUI, can you visually identify the watermark logic?
  - Do you see any distinct differences in how cells or nets are grouped compared to the original design?
- **Effect on Scaling Up**
  - Suppose you added more watermark cells or a more complex structure. How might that further impact metrics like HPWL, displacement, or routing congestion?

By studying these questions and comparing your two designs (with and without watermarking) in the OpenROAD GUI, you can develop a deeper understanding of how even a small change to your netlist can affect the automated place-and-route flow. This exploration not only highlights the feasibility of hardware watermarking but also provides a practical look at how PnR algorithms adapt to even minor modifications in design constraints.