Reconfigurable Architecture (4)

osana@eee.u-ryukyu.ac.jp

2 weeks ago:

- Continuous assignment (assign) for combinational logic
  - **wire** type signals
  - **Blocking** assignment operator (=)
  - **Conditional** assignment by “?” operator
    - Pay attention for multiplexors

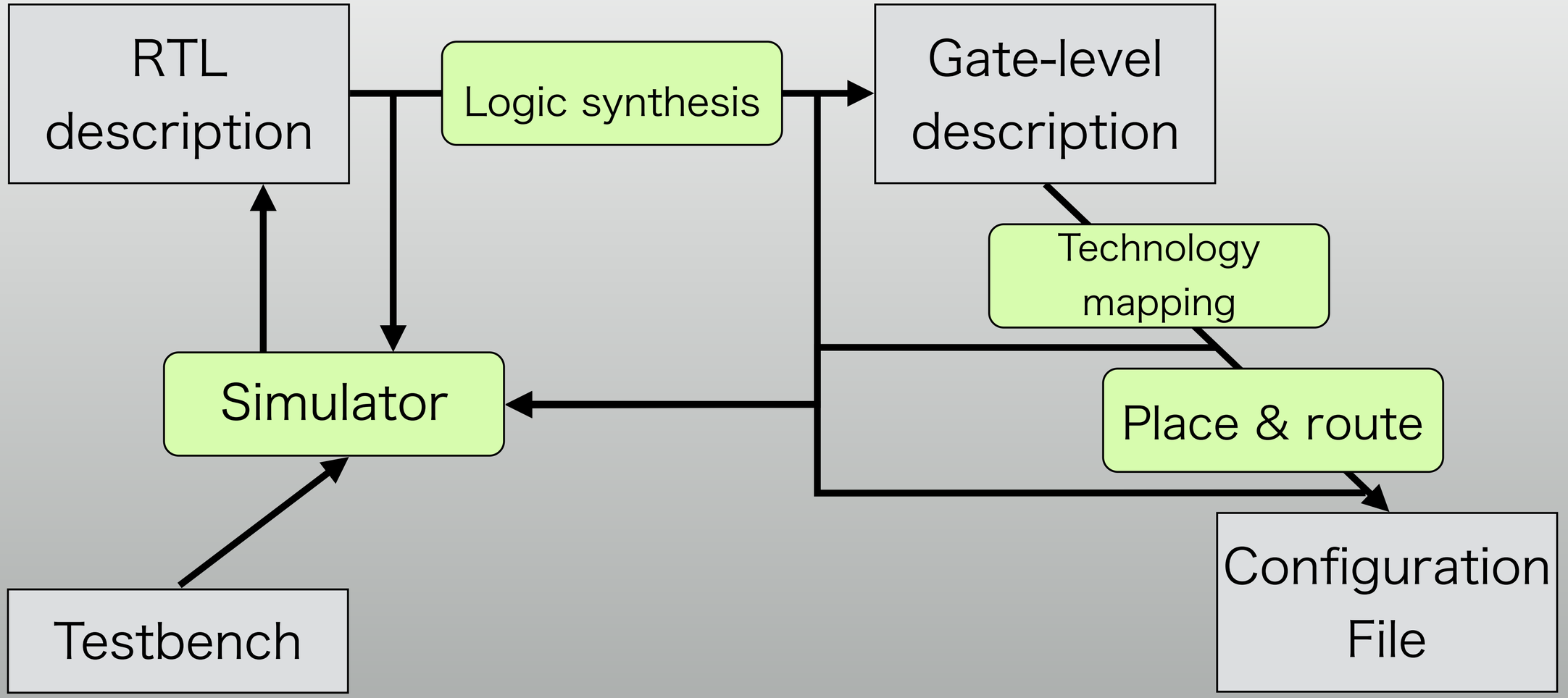
Last week:

- Procedural assignment (always) for sequential logic
  - **Clock** signal in “always @ (…) “ (usually posedge)
  - **reg** type signal (“output reg” is also OK)
  - **Non-blocking** assignment operator (<=)
  - **Control structures** such as “if” and “case” are available

This week:

- Behavior verification with HDL simulator
  - Things under test: last week's contents
  - See how to writing a testbench
    - Simulation flow with Vivado simulator is also shown

Basic design flow



Simulate always

- In all steps of design flow
  - RTL simulation, Post-synthesis simulation, Post-place & route…
  - RTL simulation is most important, and usually sufficient for FPGAs: because retry is possible if real-chip doesn’t work

Testbench is required

- Circuit never work alone itself
  - Some kind of external input is necessary
- Visual confirmation on waveform is not always perfect / possible
  - printf()-like debug is also powerful in HDL
    - Especially for event detection: then check waveform

Verilog simulators

- Xilinx Vivado simulator in this class
  - Cadence: NCsim (NC-Verilog)
  - Synopsys: VCS
  - Mentor Graphics: ModelSim
  - Stephen Williams: Icarus Verilog (open source)

Waveform viewers

- Integrated with simulator engine
  - Vivado Simulator, ModelSim
- Or provided separately
  - VCS (DVE), NCsim (simvision), Icarus Verilog (gtkwave)

Waveform files

- VCD (Value Change Dump): standard ASCII format
  - VCD is accessible by all simulators and viewers, but large
- Commercial simulators have their proprietary, compact formats
  - VPD (VCD Plus): Synopsys, SHM (Simulation History Manager): Cadence, WLF (Waveform Log File?): Mentor Graphics

Testbench vs RTL (1)

- Testbench has “flow of time”
  - **initial** statement, **$finish**, and **`timescale**
  - RTL has only events, but no beginning, history or end

Testbench vs RTL (2)

- Testbench has no port
  - Because testbench has everything outside RTL
- Testbench is not for synthesis: all syntax in Verilog is available
  - System tasks and many other syntax for simulation control

Writing flow of time (1)

- `timescale : specify unit time and time resolution
  - “`timescale 1ns/1ps” is commonly used
    - “#1” for 1ns delay
    - Delays < 1ps are rounded off
- #: delays evaluation or assignment (along `timescale)

Writing flow of time (2)

- initial: procedural assignments evaluated at t=0
  - Time course events with # operator
- always #: procedural assignments periodically evaluated
  - always # (10) to be evaluated every 10 unit time
  - Convenient for generating clock signals

Example testbench: Step 1

- Clock period of 10ns: 100MHz
  - 11ns to reset
  - 31ns to release reset
- No “unit under test” yet

```
`timescale 1ns/1ps

module testbench ();
  reg CLK, RST;

  initial CLK <= 1;
  always # (5) CLK <= ~CLK;

  initial begin
    RST <= 0;
  #11
    RST <= 1;
  #20
    RST <= 0;
  end
endmodule
```

Example testbench: Step 2

- With **parameter**
  - Better abstraction for clock period
  - **Real** type to prevent loss of digits
    (ex: Step=4 and 1.1*Step)

```
`timescale 1ns/1ps

module testbench ();
  reg CLK, RST;
  parameter real STEP = 10;

  initial CLK <= 1;
  always # (STEP/2) CLK <= ~CLK;

  initial begin
    RST <= 0;
  #(1.1*STEP)
    RST <= 1;
  #(2*STEP)
    RST <= 0;
  end
endmodule
```

RTL constructs in TB

- Ex: Clock counter
  - Good with waveform viewer
  - Also convenient with $display (shown later) and other system tasks

```
initial CLK <= 1;
always # (STEP/2) CLK <= ~CLK;

reg [31:0] CLK;
always @ (posedge CLK)
  CNT <= RST ? 0 : CNT+1;

initial begin
  RST <= 0;
#(1.1*STEP)
  RST <= 1;
#(2*STEP)
  RST <= 0;
end
```

System tasks

- Command-like constructs for simulators
  - Handling waveform files
  - Displaying messages or reading/writing files in simulation
  - Mathematical functions in real (FP) type: $sin, $cos…
    - Mostly ignored by synthesis tools (or causes an error)

$display, $write: stdio (1)

- $display (“format”, signal1, signal2…);
  - printf()-like function with newline at the end ($write w/o NL)
  - %b: binary, %d: decimal, %h: hexadecimal, %f: real
  - \t and \n for tab and newline
    - Called within initial / always block

$monitor: stdio (2)

* Similar to $display, called on its change
  * $monitoron / $monitoroff to suspend and resume

$f{display, write, monitor}

- File access, almost same with standard C library
  - mcd = $fopen(“filename”);
  - $fdisplay(mcd, “format”, signal, signal…);
  - $fclose (mcd);

Obtaining simulation time

- $realtime
  - Returns “real” time, in second
  - $display(“Time = %f ”, $realtime);
- $time
  - 64bit integer, unit is `timescale

Terminating simulation

- $finish: terminates simulation
  - Some simulators displays CPU time with $finish(1) or $finish(2)

    (while others don’t support this)
- In Vivado simulator, this is not mandatory because length of simulation can be specified in GUI
  - Important with command-line based simulators

Data conversion

- $bitstoreal, $realtobits: real <-> 64bit signal in IEEE-754 standard
  - Convenient in debugging scientific computing applications
- $itor, $rtoi: real <-> integer
- $random, $sin, $cos,…: many other (mathematical) functions

Saving waveform

- $dumpfile(“foo.vcd”); save waveform in “foo.vcd”
- $dumpvars(0); Record all signals in VCD file above
  - Vendor specific system tasks for vendor specific files
  - In Vivado simulator, no $dumpfile is required to see waveform

System tasks: summary

- $display, $monitor, $write, $fdisplay, $fmonitor, …
- $realtime, $time
- $finish
- $bitstoreal, $realtobits, $itor, $rtoi, $sin, $cos, …
- $dumpfile, $dumpvars

Module under test (UUT)

- Becomes a submodule of testbench
  - Input signals generated in testbench, usually as reg variable
  - Other system model (i.e, DRAMs) may also included as submodules, and connected to UUT the by wires
  - Output signal may be connected wires, or left unconnected

Simple example

```
`timescale 1ns/1ps

module sw_led_tb ();
  reg [3:0]  SW;
  reg        PUSH;
  wire [3:0] LED;

  sw_led uut (.SW(SW), .LED(LED),
              .PUSH(PUSH));

  initial begin
          SW <= 4'b0001; PUSH <= 0;
    #(10) SW <= { SW[2:0], SW[3] };
    #(10) SW <= { SW[2:0], SW[3] };
  end
endmodule
```

```
module sw_led
  (
   input  wire [3:0] SW,
   input  wire       PUSH,
   output wire [3:0] LED
  );

  wire  SW1_ = ~SW[1];
  assign LED[0] = PUSH & SW[0];
  assign LED[1] = SW1_;
  assign LED[2] = SW1_ & SW[2];
  assign LED[3] = |SW;

endmodule
```

Observing signals

- Waveform viewer: walking down module hierarchy
  - Can see any signal in the design,
  - But usually not easy

Test logic in HDL

- Writing test assistance logic in testbench
  - Adding signals such as “OK” with simple combinational logic
  - Using $display in some specific conditions
- Testbench can access signals inside UUT

Accessing signals inside

- For simulation only:
  - i.e) uut.SW1_
  - Can go even deeper by: inst1.inst2.inst3.signal
- Not good for synthesis
  - Not possible in VHDL

```
module sw_led
  (
   input  [3:0] SW,
   input        PUSH,
   output [3:0] LED
  );

  wire  SW1_ = ~SW[1];
  assign LED[0] = PUSH & SW[0];
  assign LED[1] = SW1_;
  assign LED[2] = SW1_ & SW[2];
  assign LED[3] = |SW;

endmodule
```

Try it

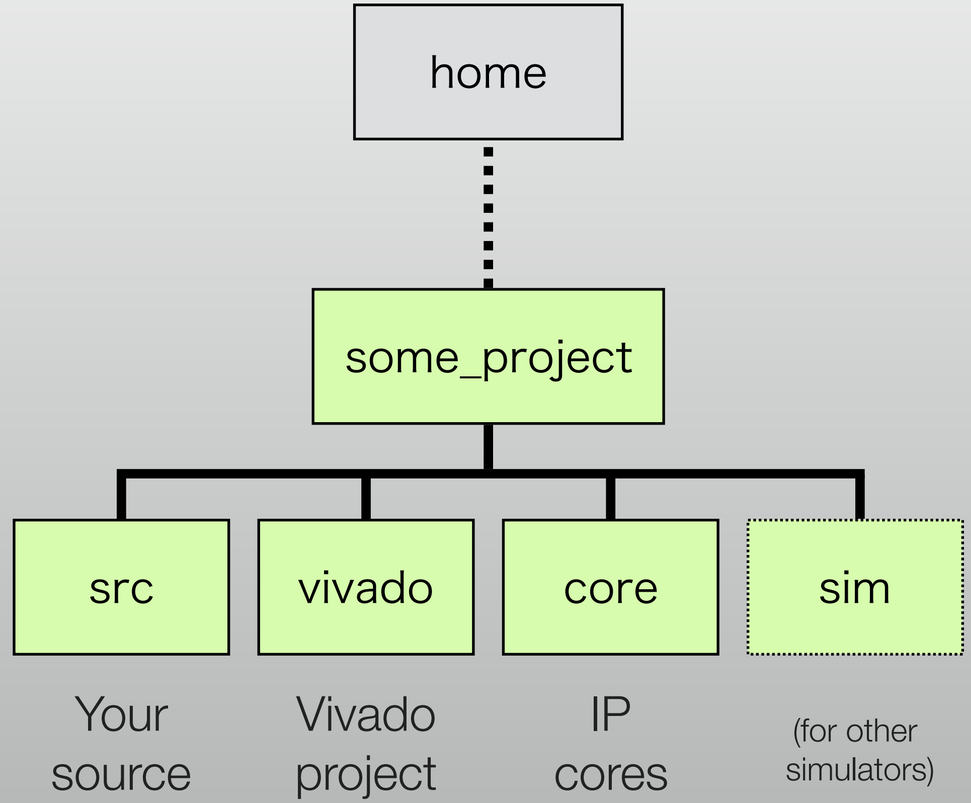
- Simulate (and implement next week) in Vivado
  - Write a testbench and RTL
  - A “project” must be generated first, with target device
- For simulator, the target device is not essential

Where to place source files

- Inside project folder, in Vivado's default
  - CAD generates a lot of files,
    separating your own source code is important
    - Project is sometime broken, or becomes a problem in reuse

Typical directory organization

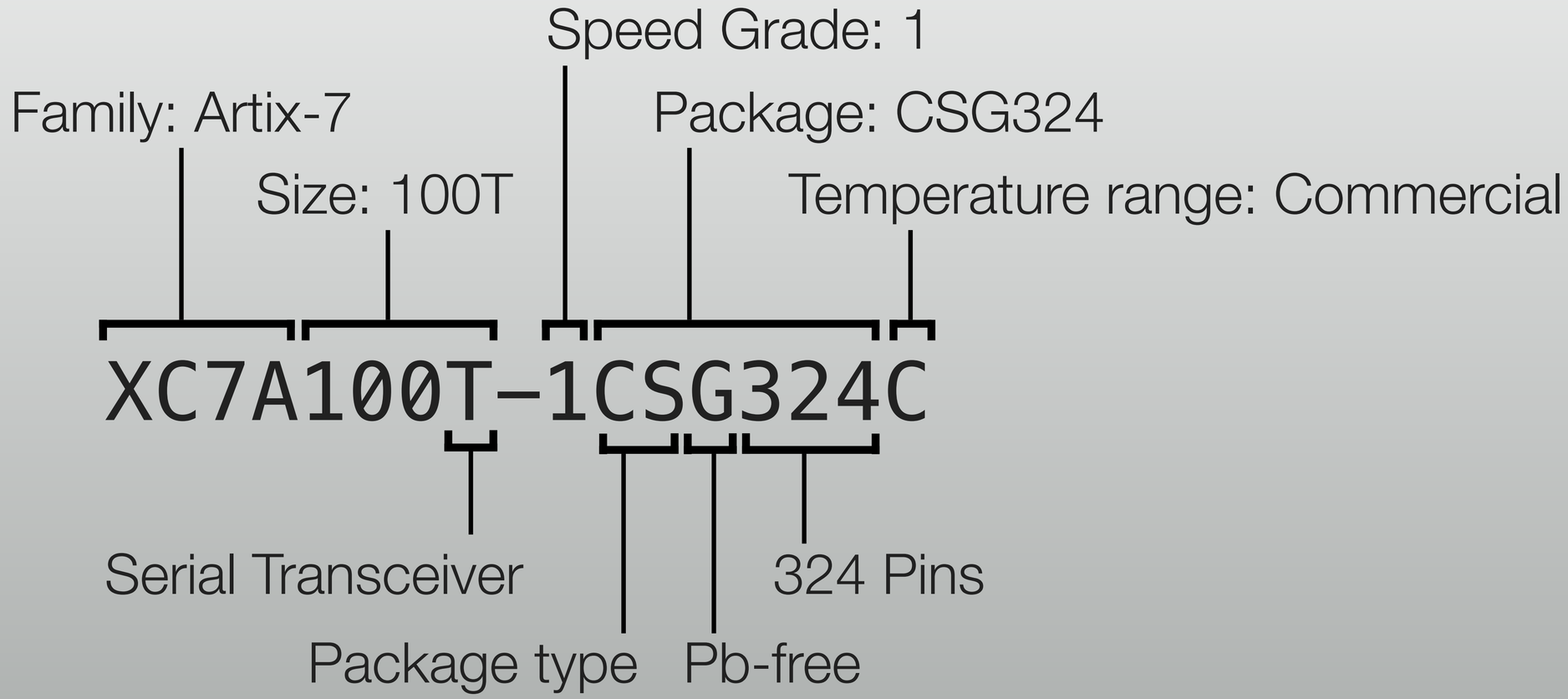
- Separate source and project
  - Source code is managed manually
  - Project only refers the source files



FPGA ordering # (or model #)

- Device family
  - Xilinx: Virtex, Kintex, Artix, Zynq, Spartan, …
  - Altera: Stratix, Arria, Cyclone, MAX, …
- Device size and additional features
- Package and speed grade

Example of Ordering

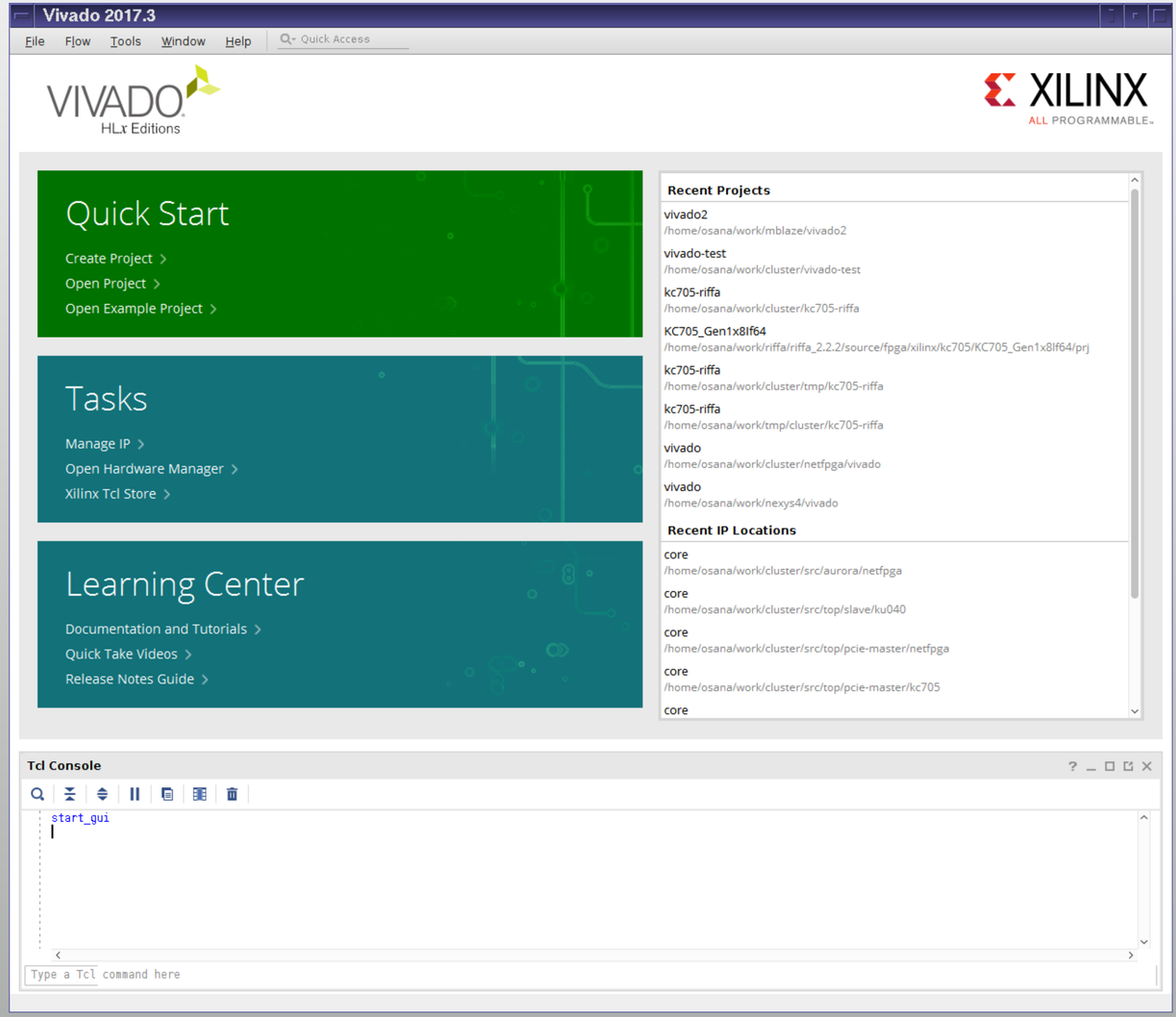


Hands-on

- Create a Vivado project
  - Write a simple RTL and testbench
  - Then run simulation
- For simpleness, source files in project directory

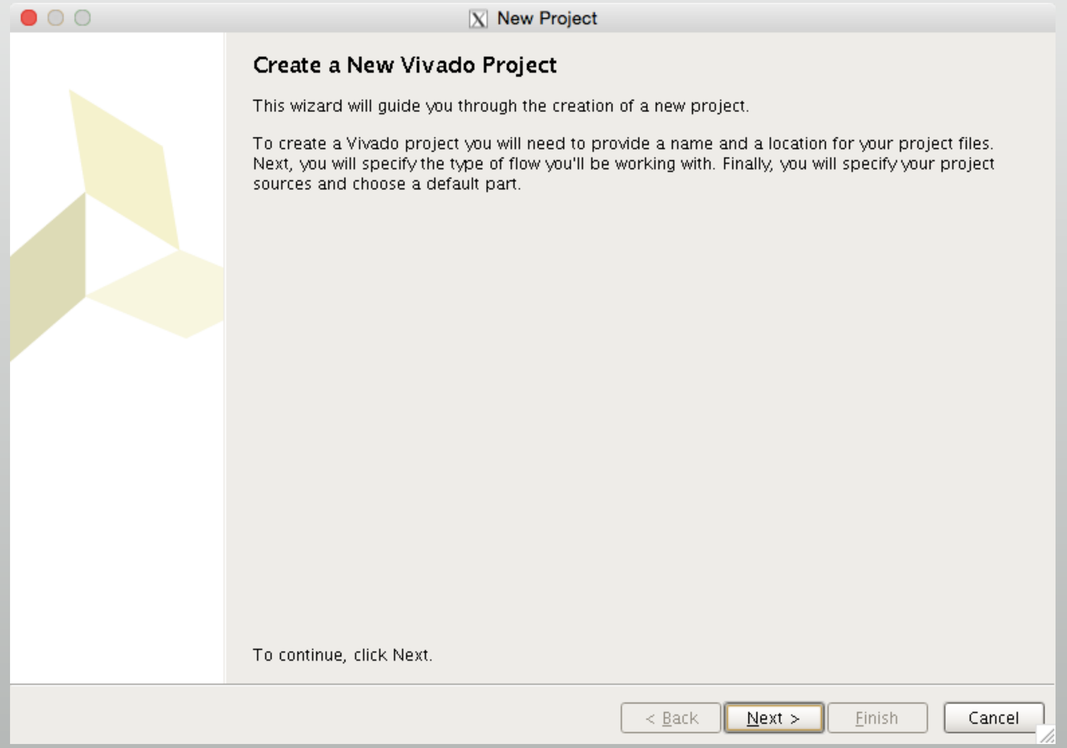
Launch Vivado

* and “Create Project”



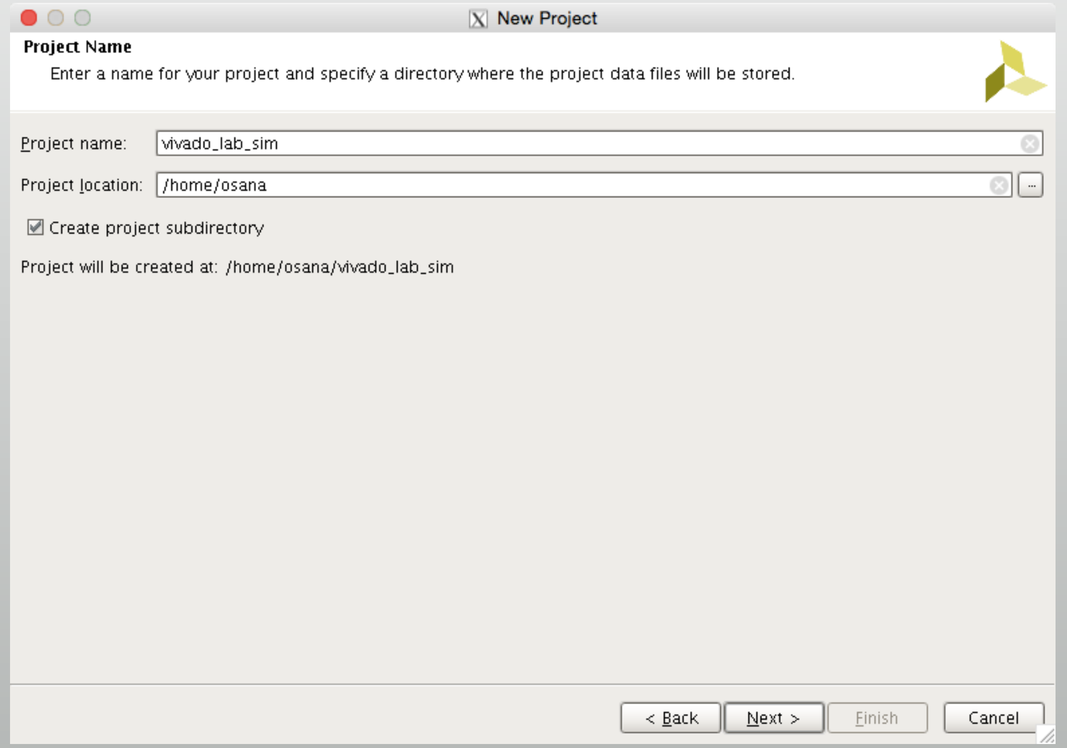
Create project (1/5)

* Just click next



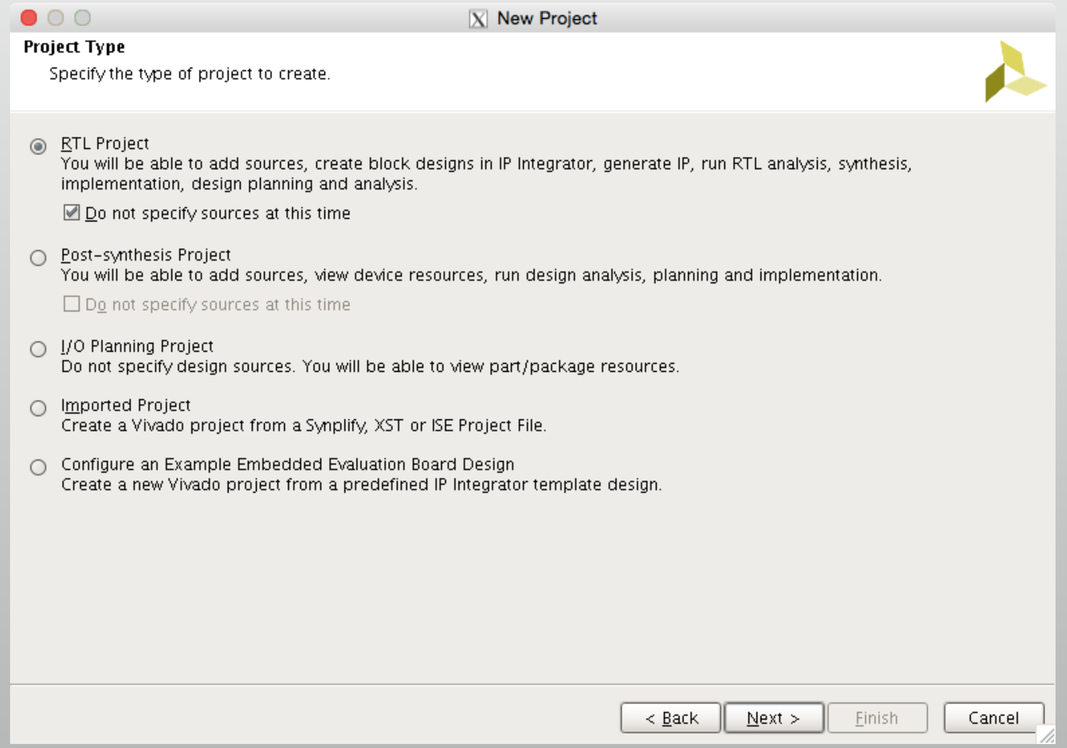
Create project (2/5)

- Name and location
  - Name “vivado_lab_sim”
  - Folder with the project name is created



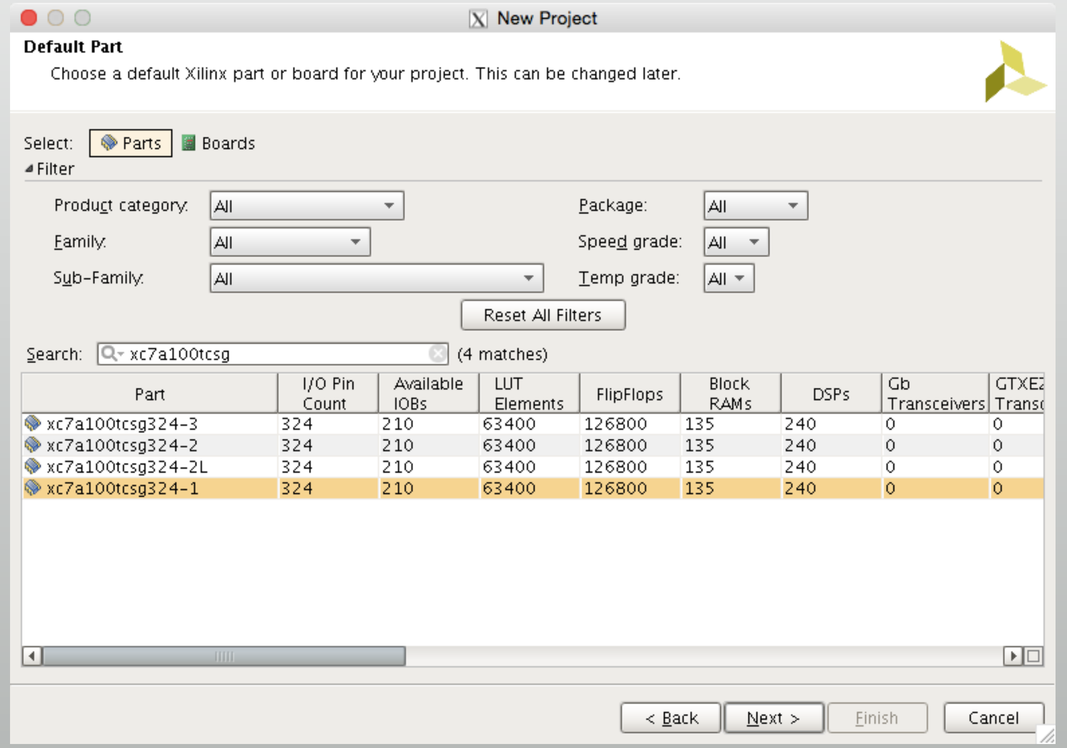
Create project (3/5)

- ”RTL Project” is the basic
  - Still have no source code:
    “Do not specify sources…”



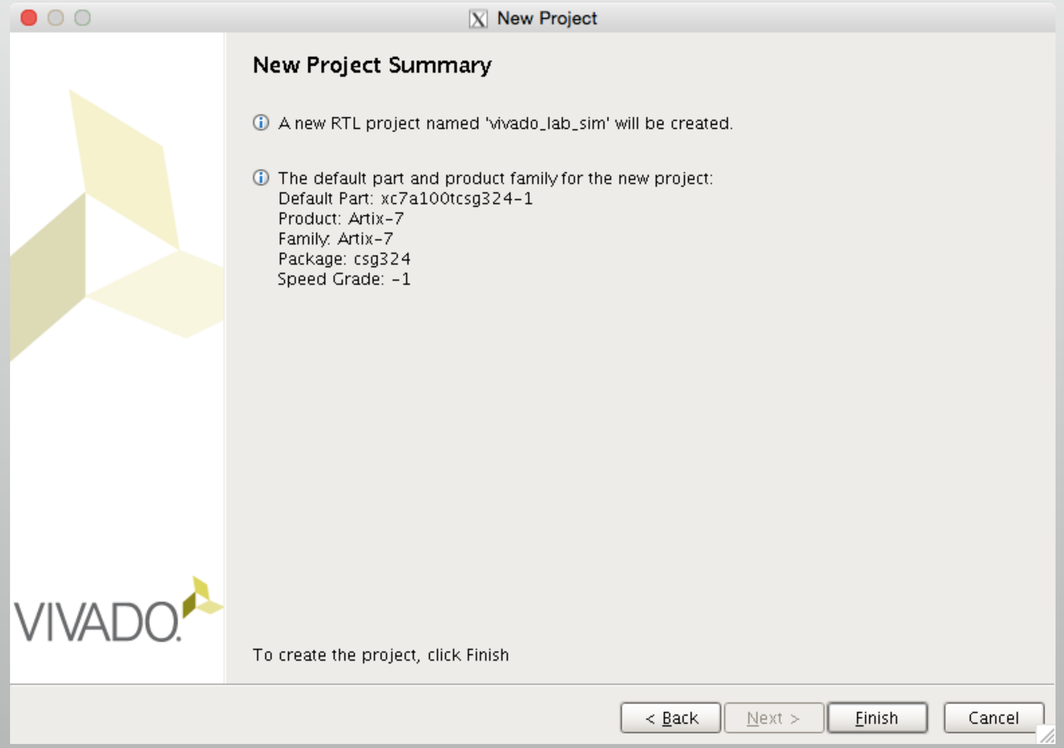
Create project (4/5)

- Choose device
  - XC7A100TCSG324-1
  - Enter in “Search” field, or search by category / family (Artix-7)

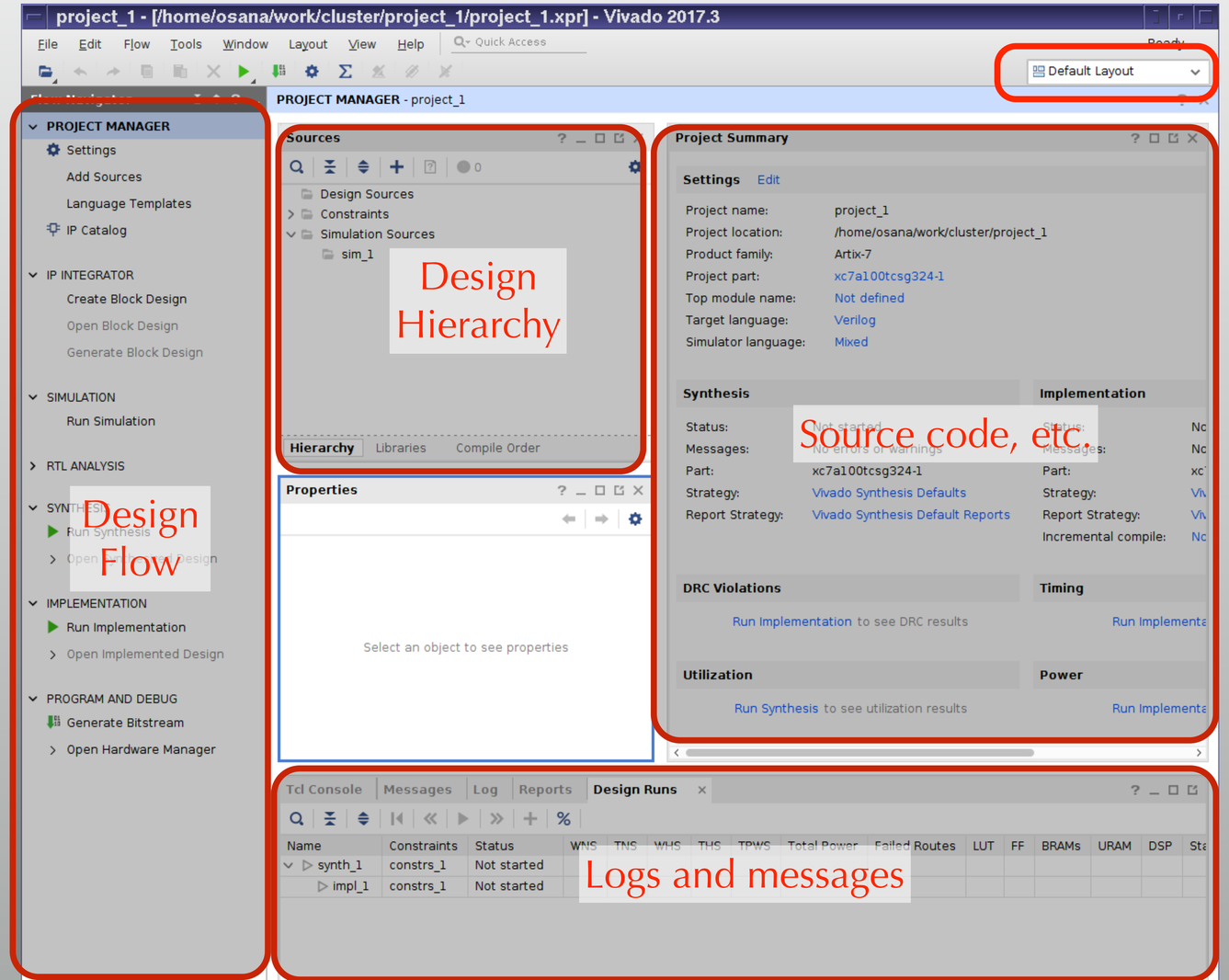


Create project (5/5)

* Check the settings



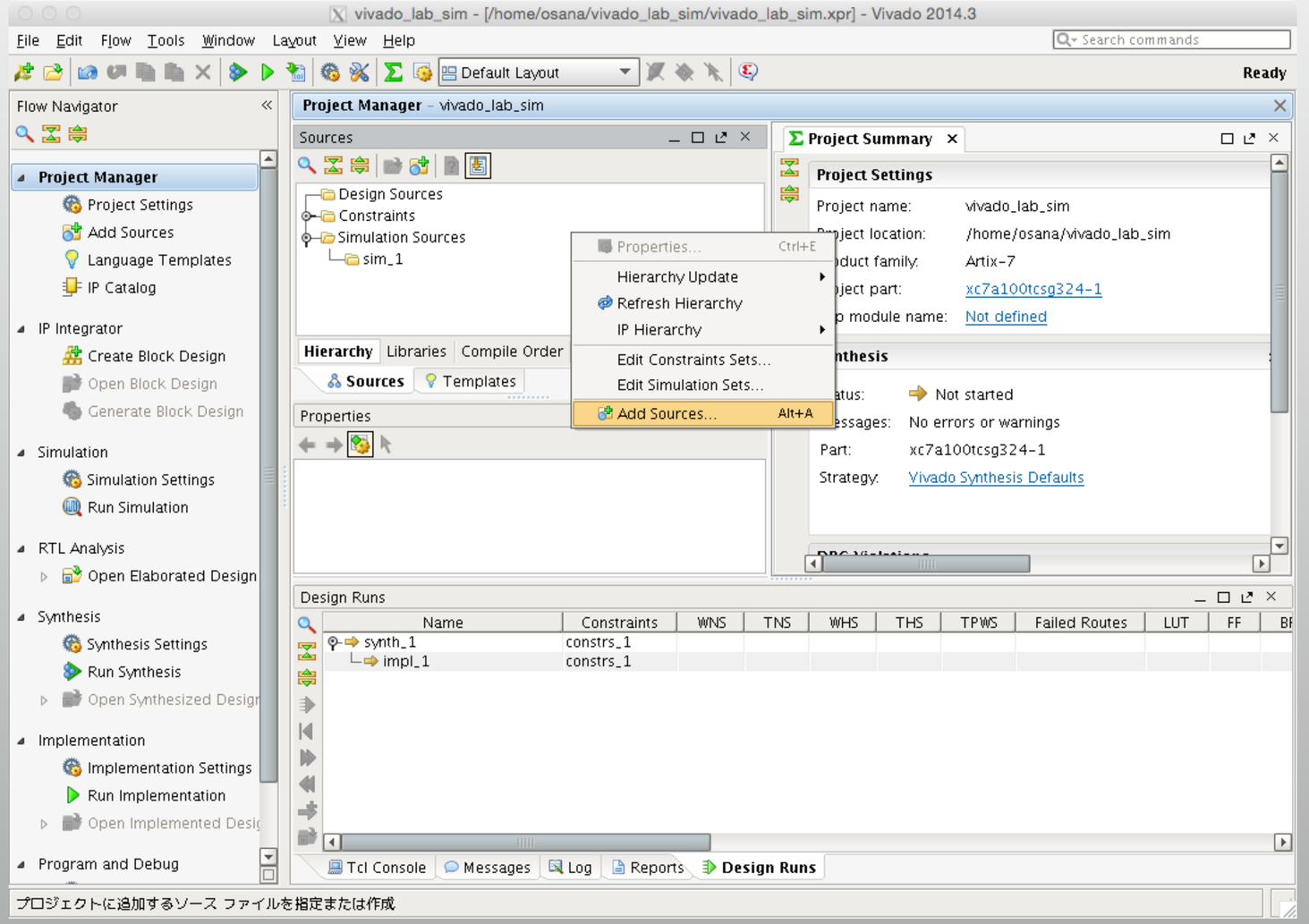
Vivado screen



"Default Layout" makes reset the screen

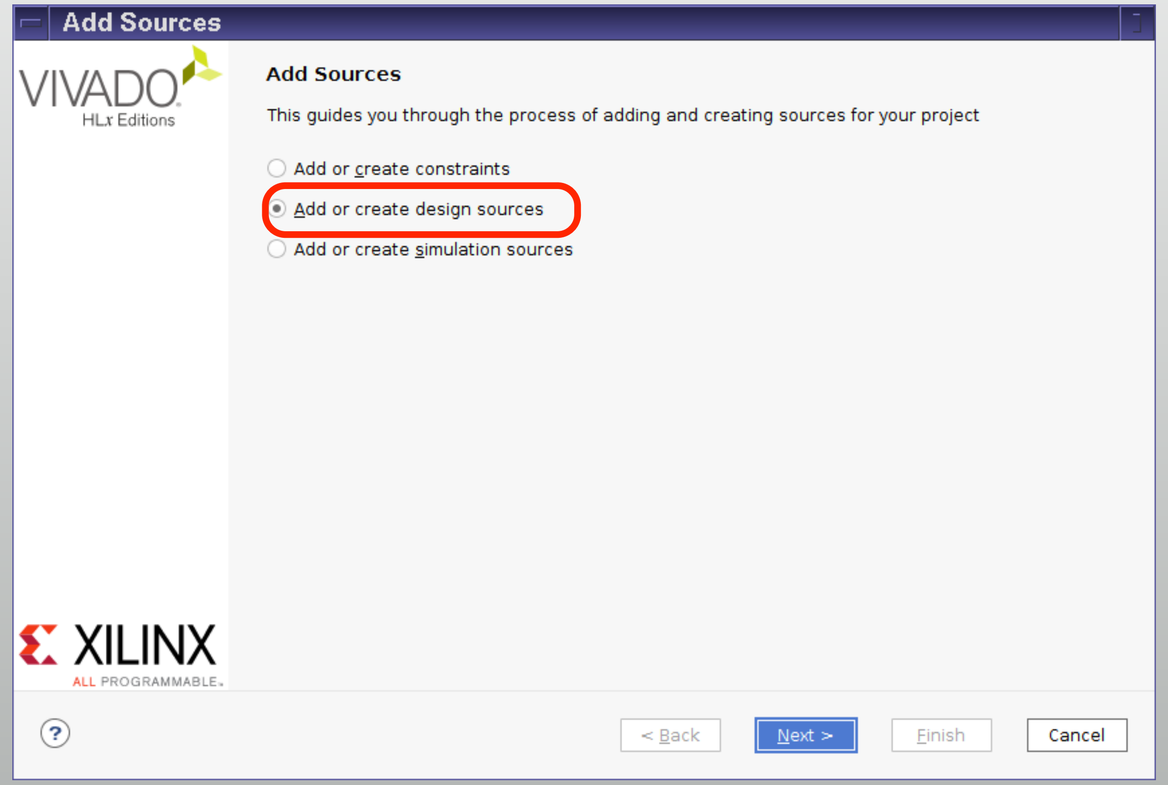
Add source code (1/4)

* “Add Sources” to add one



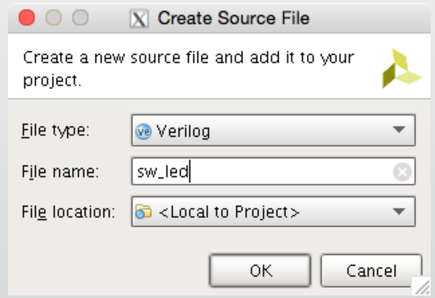
Add source code (2/4)

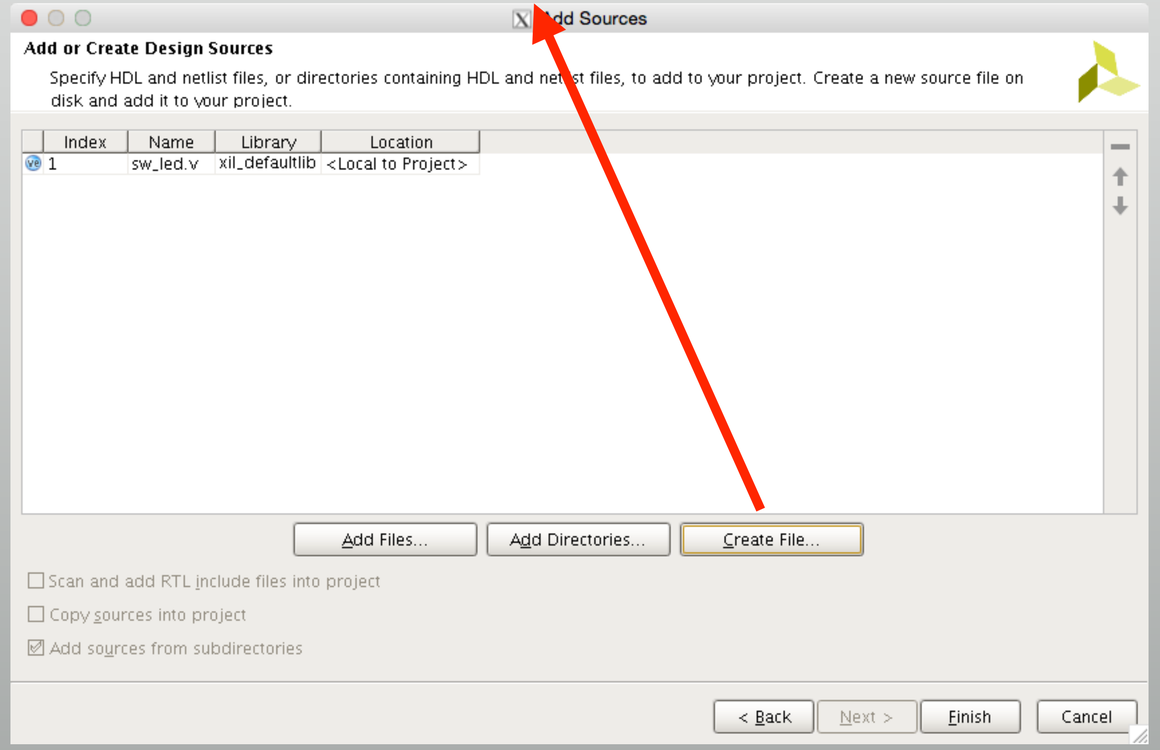
- RTL corresponds to “Design Source”



Add source code (3/4)

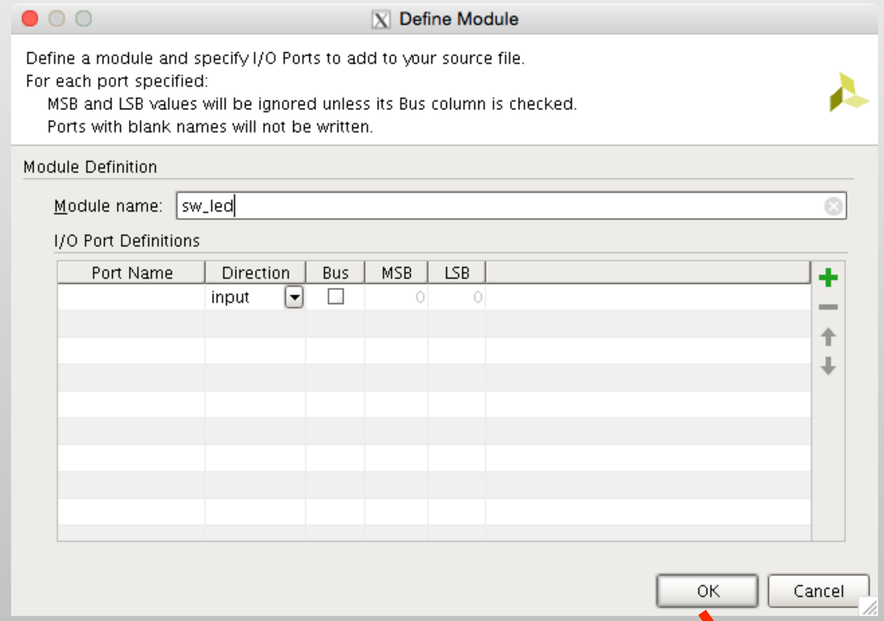
- “Create File” because there’s no file yet
  - Name “sw_led"
- If there’s already source file, do “Add Files”

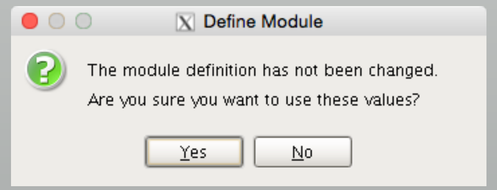




Add source code (4/4)

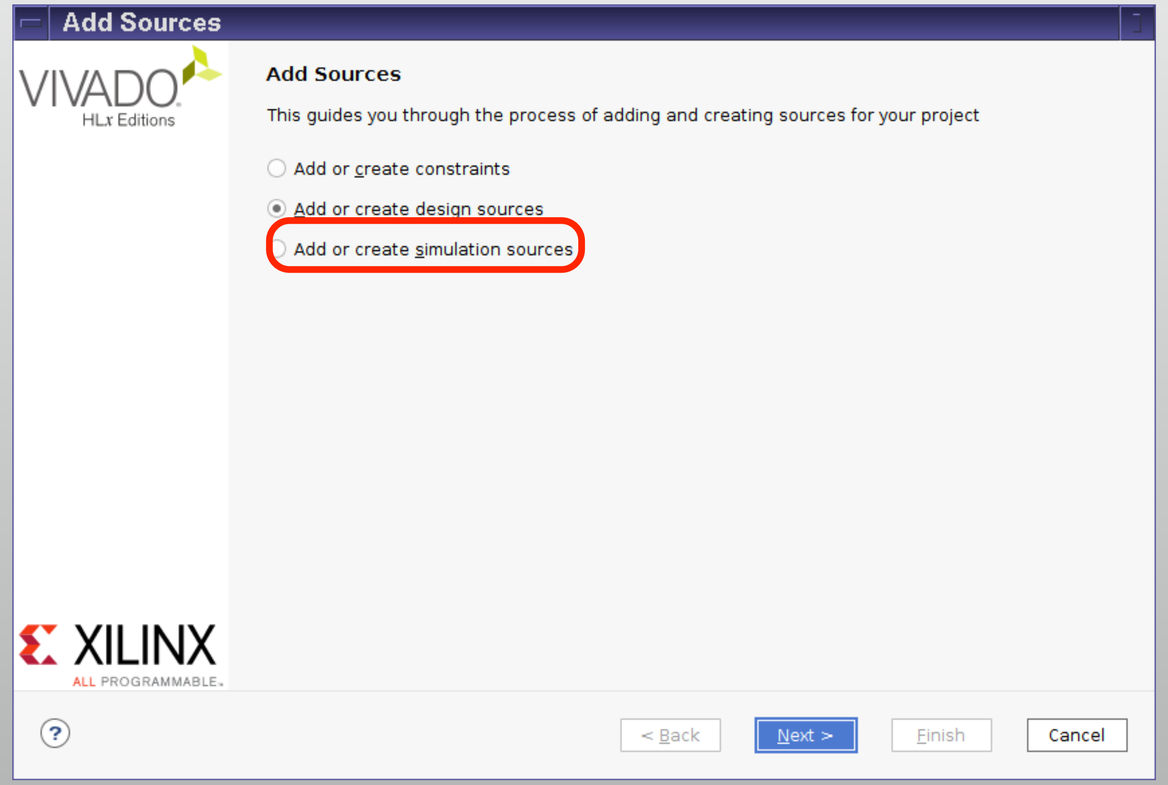
- Vivado will ask the module's port organization
  - Just ignore it





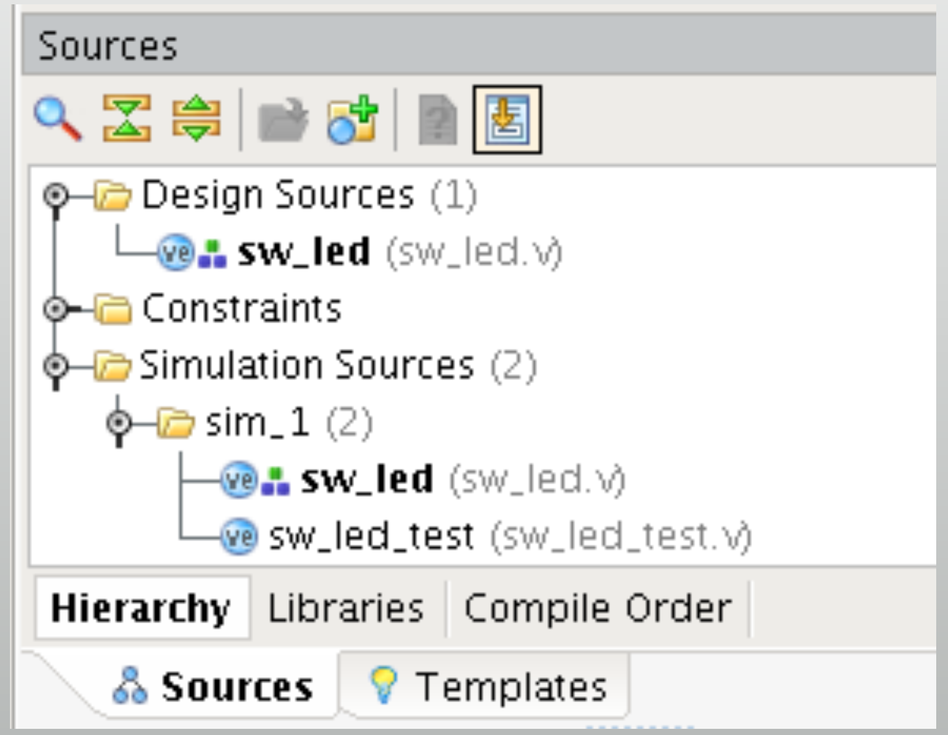
Add testbench

- “Simulation Source” is that
  - Name “sw_led_test”
  - Rest is same to RTL



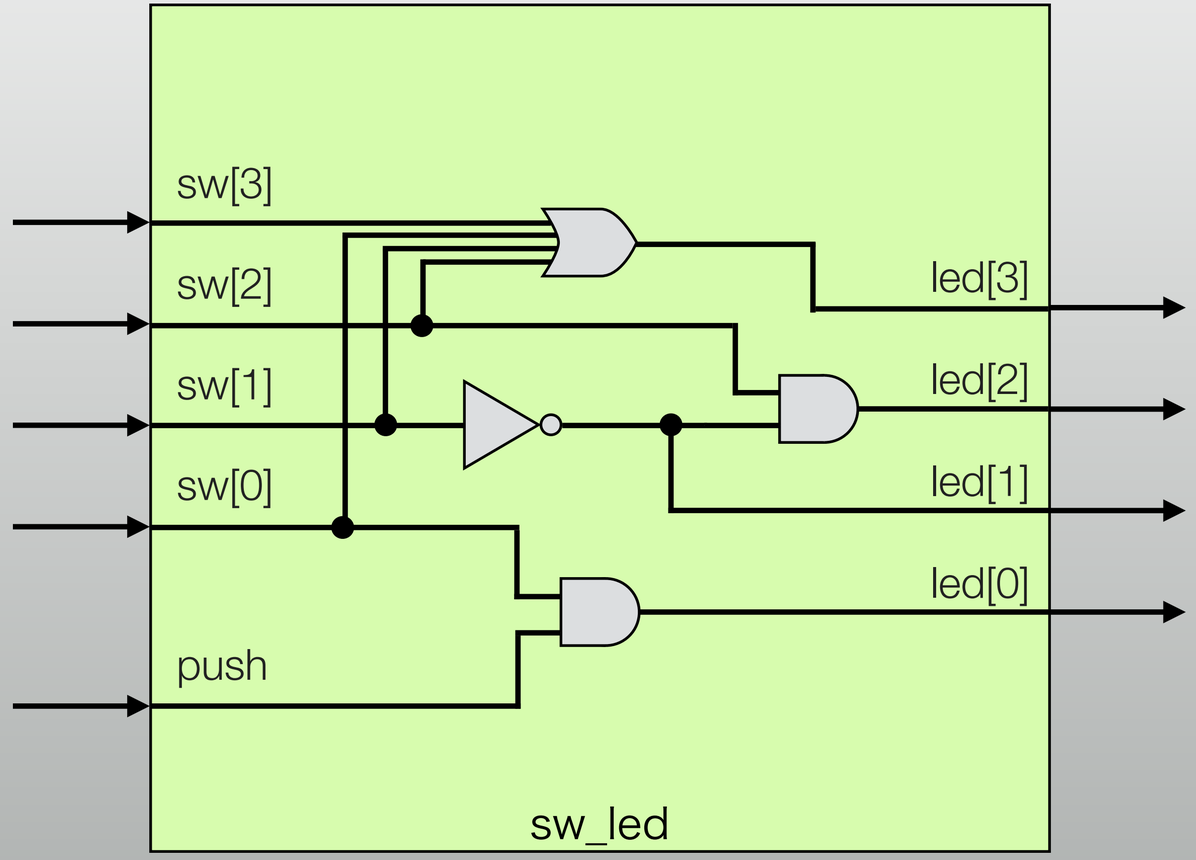
Design hierarchy

- Design Sources = RTL
- Simulation Sources = Testbench + RTL
  - Multiple testbenches are possible (create other simulation set than sim_1)



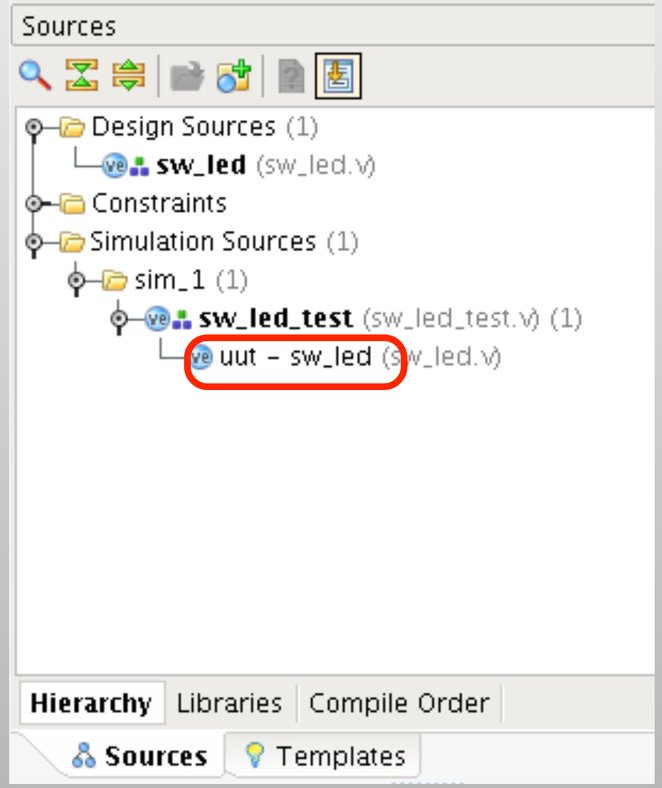
Example

* Source code in the last slide today



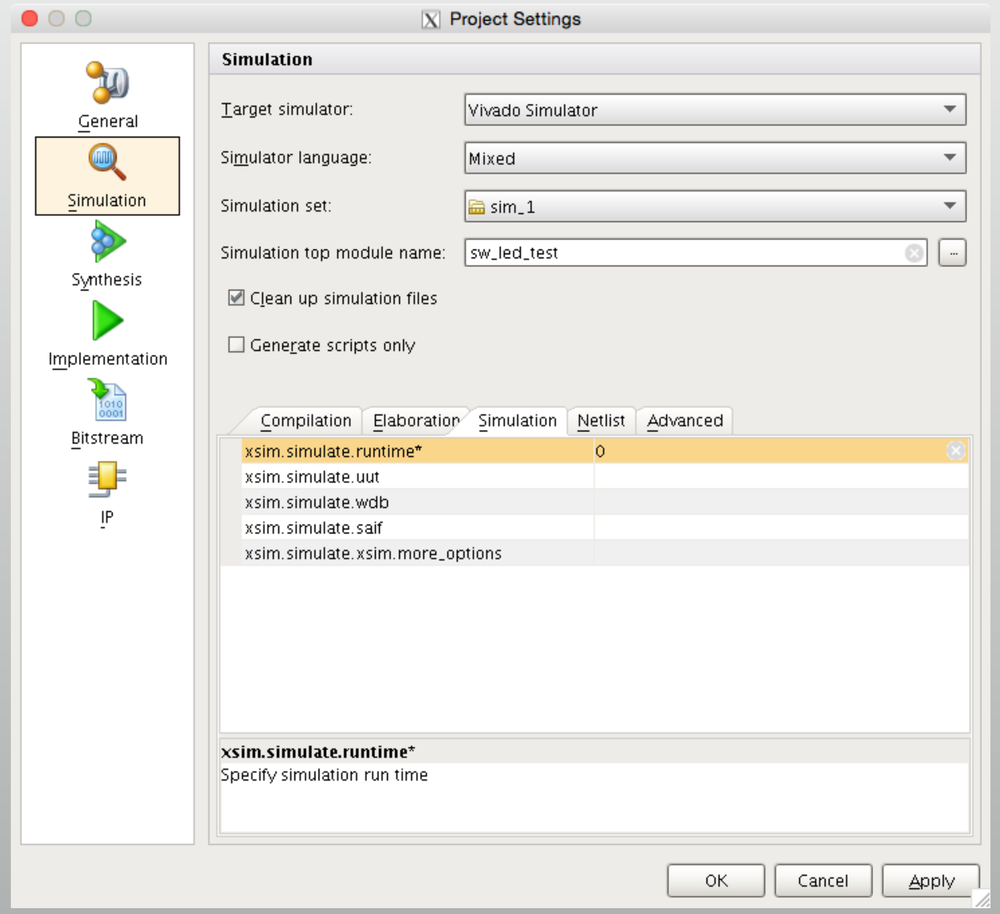
Design hierarchy review

- RTL module under testbench
  - “Instance name - module name” is shown



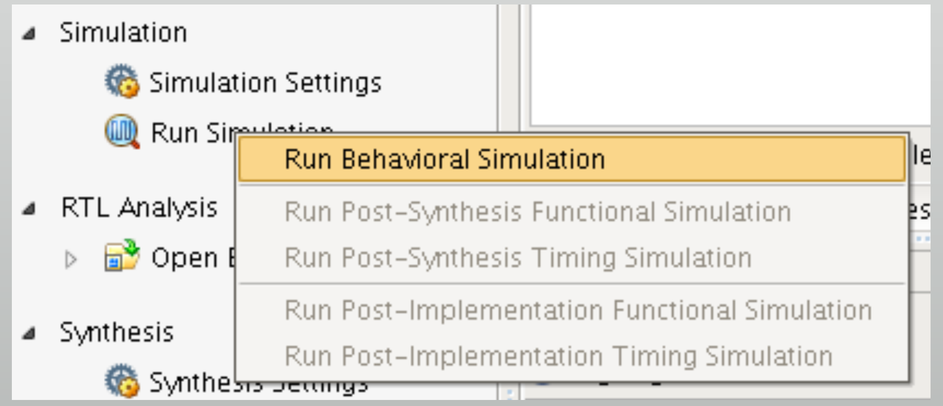
Simulation settings

- “Settings” in Flow navigator
  - Simulation settings → Simulation → runtime to 0
  - Default is 1000ns

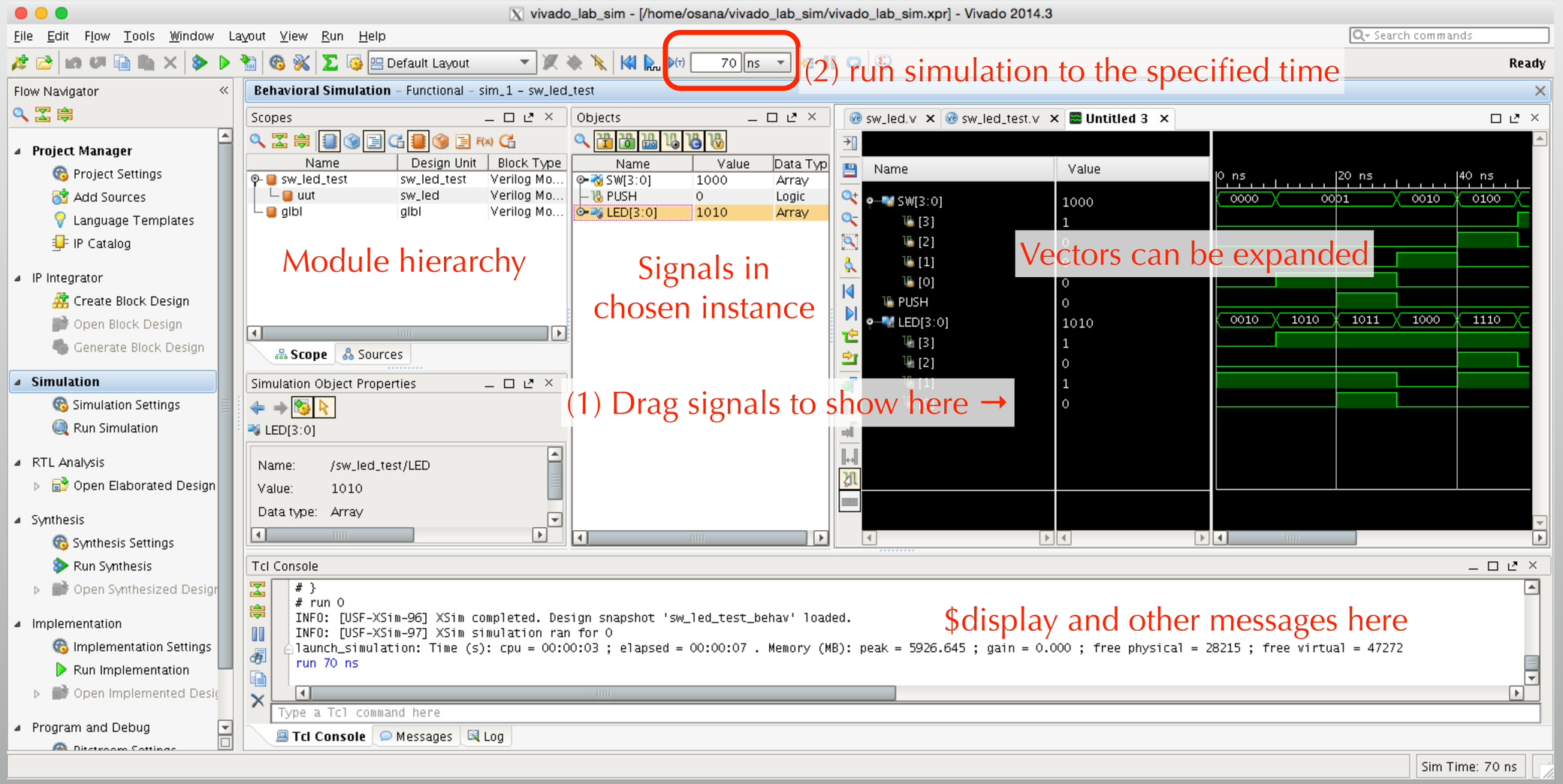


Run compilation

- Run Simulation →
  Run Behavioral Simulation
  - Compiler will be launched
  - Post-synthesis and other simulation is also possible

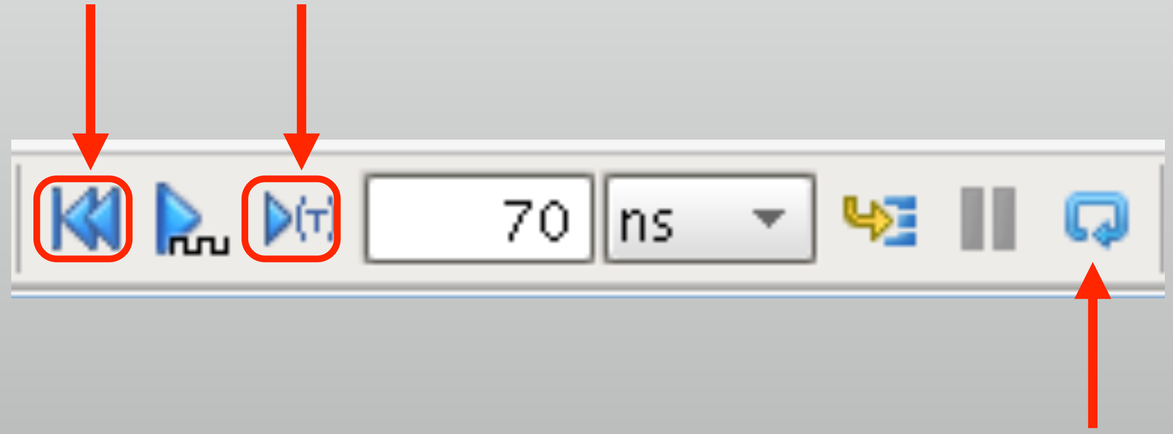


Simulation


vivado_lab_sim - [/home/osana/vivado_lab_sim/vivado_lab_sim.xpr] - Vivado 2014.3
File Edit Flow Tools Window Layout View Run Help
Default Layout
70 ns
Ready
(2) run simulation to the specified time
Flow Navigator
Project Manager
Project Settings
Add Sources
Language Templates
IP Catalog
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Open Elaborated Design
Synthesis
Synthesis Settings
Run Synthesis
Open Synthesized Design
Implementation
Implementation Settings
Run Implementation
Open Implemented Design
Program and Debug
Behavioral Simulation - Functional - sim_1 - sw_led_test
Scopes
Name Design Unit Block Type
sw_led_test sw_led_test Verilog Mo...
uut sw_led Verilog Mo...
glbl glbl Verilog Mo...
Module hierarchy
Objects
Name Value Data Typ
SW[3:0] 1000 Array
PUSH 0 Logic
LED[3:0] 1010 Array
Signals in chosen instance
Scope Sources
Simulation Object Properties
LED[3:0]
Name: /sw_led_test/LED
Value: 1010
Data type: Array
(1) Drag signals to show here →
sw_led.v sw_led_test.v Untitled 3
Name Value
SW[3:0] 1000
[3] 1
[2] 0
[1] 0
[0] 0
PUSH 0
LED[3:0] 1010
[3] 1
[2] 0
[1] 1
[0] 0
Vectors can be expanded
0 ns 20 ns 40 ns
0000 0001 0010 0100
0010 1010 1011 1000 1110
Tcl Console
}
run 0
INFO: [USF-XSim-96] XSim completed. Design snapshot 'sw_led_test_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 0
launch_simulation: Time (s): cpu = 00:00:03 ; elapsed = 00:00:07 . Memory (MB): peak = 5926.645 ; gain = 0.000 ; free physical = 28215 ; free virtual = 47272
run 70 ns
$display and other messages here
Type a Tcl command here
Tcl Console Messages Log
Sim Time: 70 ns

To run again…

To add signal(s) to waveform, reset and run again



Recompile when source code is modified

Today’s source code

```
`timescale 1ns/1ps

module sw_led_tb ();
  reg [3:0]  SW;
  reg        PUSH;
  wire [3:0] LED;

  sw_led uut (.SW(SW), .LED(LED),
              .PUSH(PUSH));

  initial begin
    $monitor(“%t SW: %b, PUSH: %b”, $time, SW, PUSH);

        SW <= 4’b0000; PUSH <= 0;
    #10 SW <= 4’b0001;
    #10 PUSH <= 1;
    #10 SW <= { SW[2:0], SW[3] };  PUSH <= 0;
    #10 SW <= { SW[2:0], SW[3] };
    #10 SW <= { SW[2:0], SW[3] };
  end
endmodule
```

```
module sw_led
  (
   input  wire [3:0] SW,
   input  wire       PUSH,
   output wire [3:0] LED
  );

  wire  SW1_ = ~SW[1];
  assign LED[0] = PUSH & SW[0];
  assign LED[1] = SW1_;
  assign LED[2] = SW1_ & SW[2];
  assign LED[3] = |SW;

endmodule
```

Just copy and paste to TB + RTL file, then it’ll work