



FYSIKUM

# Digital System Construction - 1

Lecture 2: FPGA design flow and an introduction to VHDL

FPGA design flow

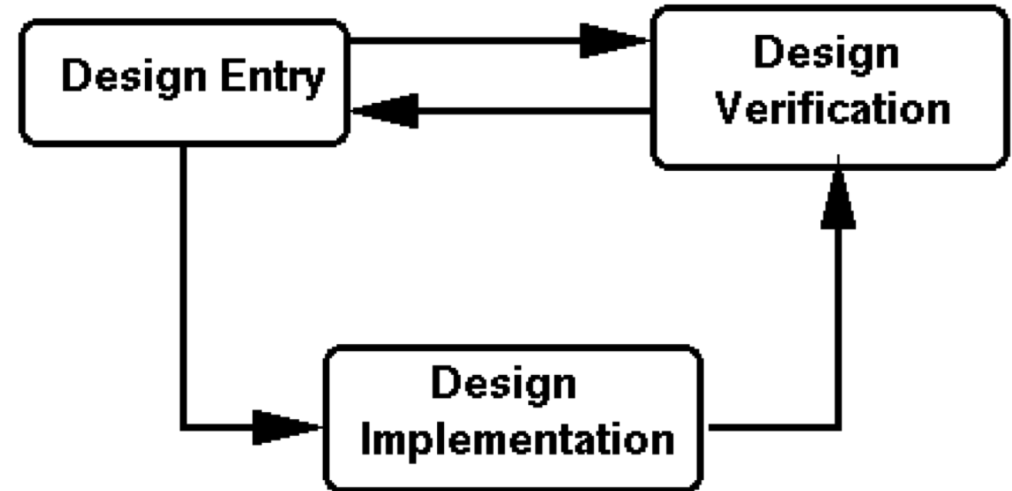
Introduction to VHDL

Generate and generics

Test benches

Vivado overview

# FPGA Design Flow



- **Design Entry**
  - ◆ Graphical tools or **HDL**
- **Simulation/verification:**
  - ◆ Verify functionality of the design
  - ◆ Calculate timing, resource usage, etc.
- **Implementation:**
  - ◆ Place and route design in the target device
  - ◆ Create a configuration bit-stream file
  - ◆ Download and verify configuration to hardware

# Design Entry

Two methods:

- **Text entry**

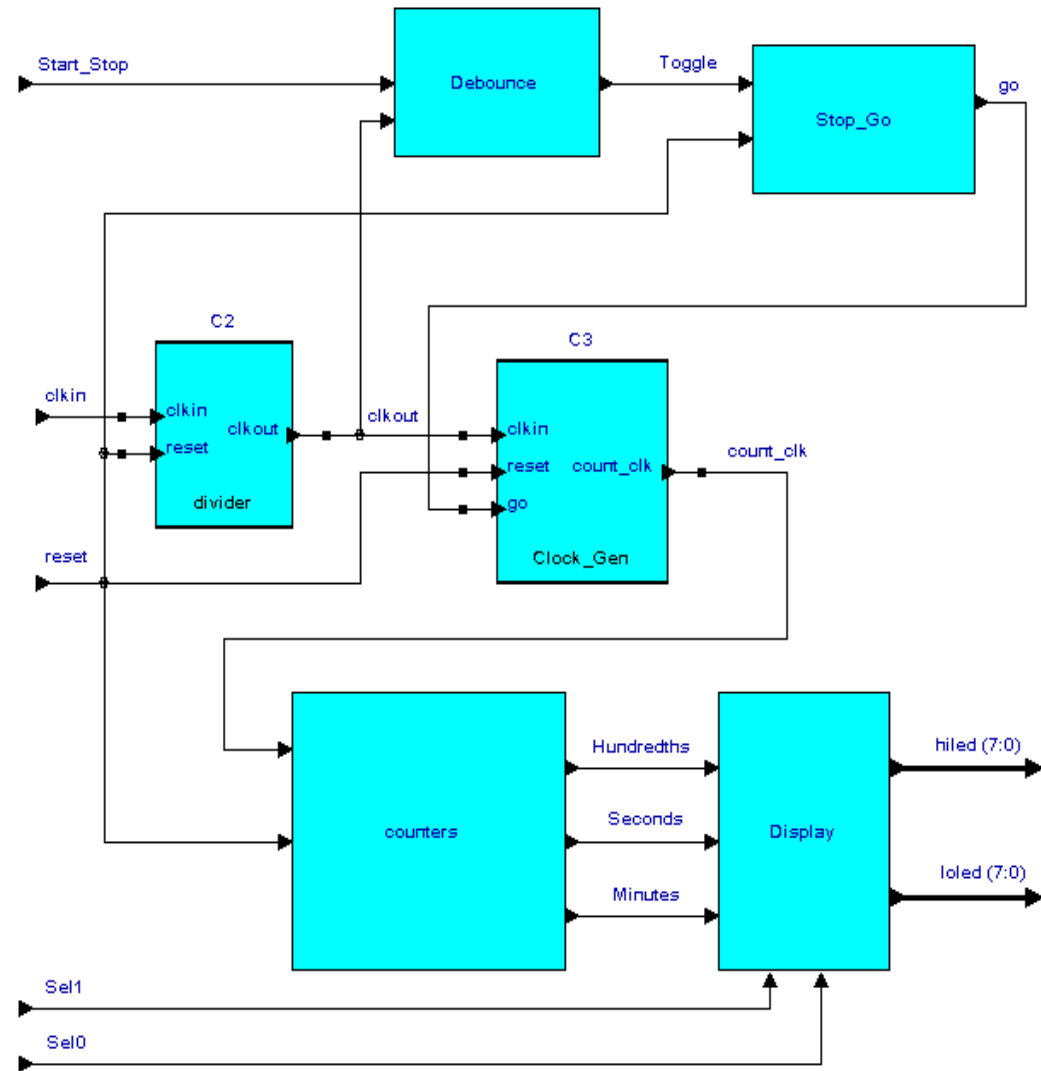
- ◆ Based on a HDL (**VHDL**, Verilog, System C...)
- ◆ Compact format, no special editing tools required
- ◆ Good for high-level designs and control logic

- **Graphical entry**

- ◆ Block diagrams, state diagrams, waveforms, etc
- ◆ Often combined with text entry
- ◆ Graphical diagrams are converted to HDL by the design software

# Graphical entry: block diagram

- Hierarchical design
  - ◆ Can navigate down to block contents
- Blocks can be designed with any method...
  - ◆ Block diagram
  - ◆ State diagram
  - ◆ HDL
- Graphical connections between blocks can be easy to follow
  - ◆ Which is a common reason for choosing graphical entry...



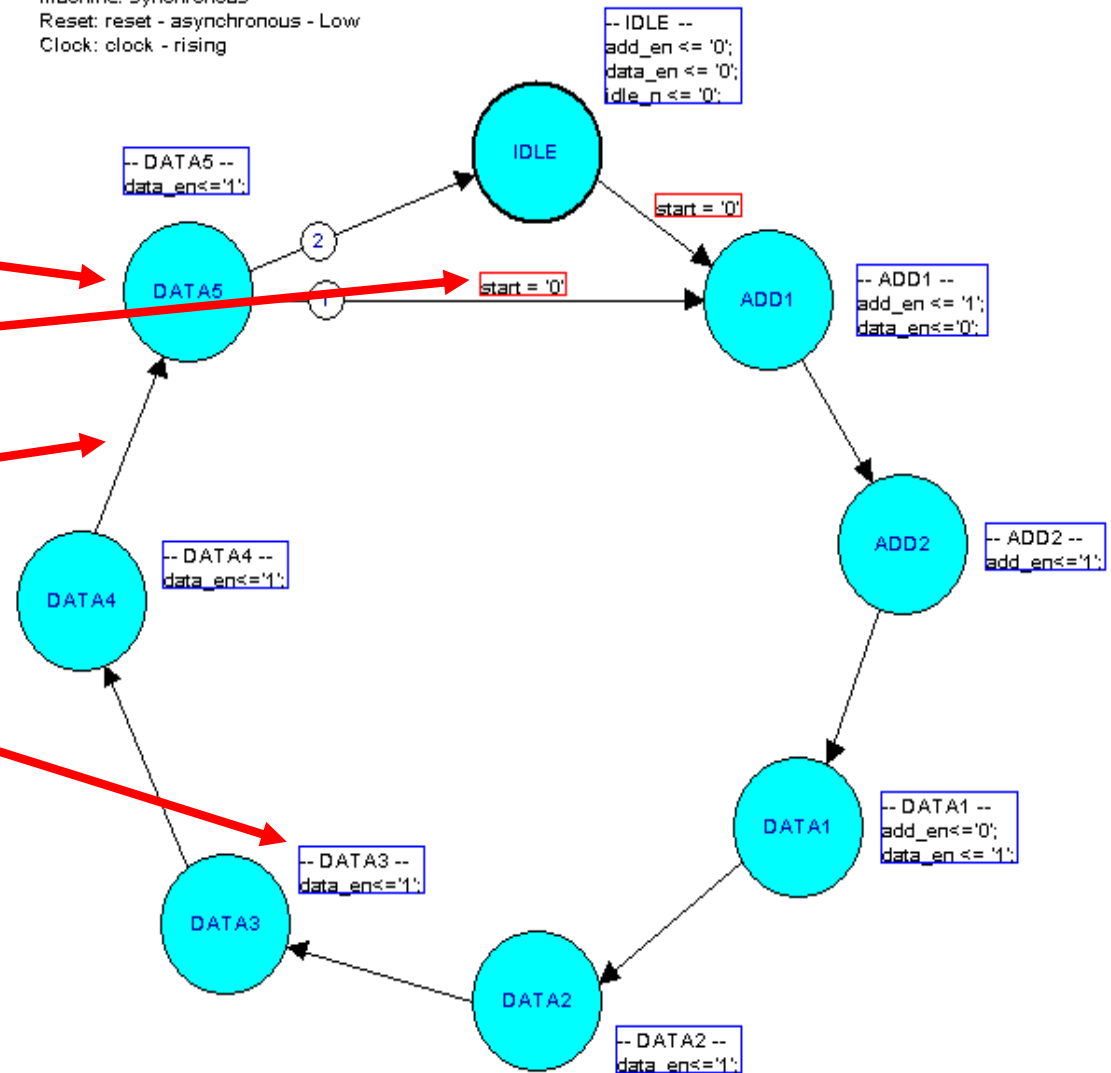
# Graphical entry: state diagram

Machine: synchronous  
Reset: reset - asynchronous - Low  
Clock: clock - rising

## ■ “Bubble” diagram

- ◆ States
- ◆ Conditions
- ◆ Transitions
- ◆ Outputs

## ■ Commonly used to specify control modules



# Simulation

Example of a simulation waveform display  
Test stimulus often defined in non-coding HDL (test bench)



# Implementation

Go from HDL design to a working circuit

- Two main phases:
  - ◆ Synthesis
    - ✦ Convert HDL description to match the internal FPGA (or ASIC) architecture
  - ◆ Implementation
    - ✦ Place components and interconnections in the target device
    - ✦ Optimize component placement and signal paths to meet timing/area constraints
    - ✦ Generate final design output files

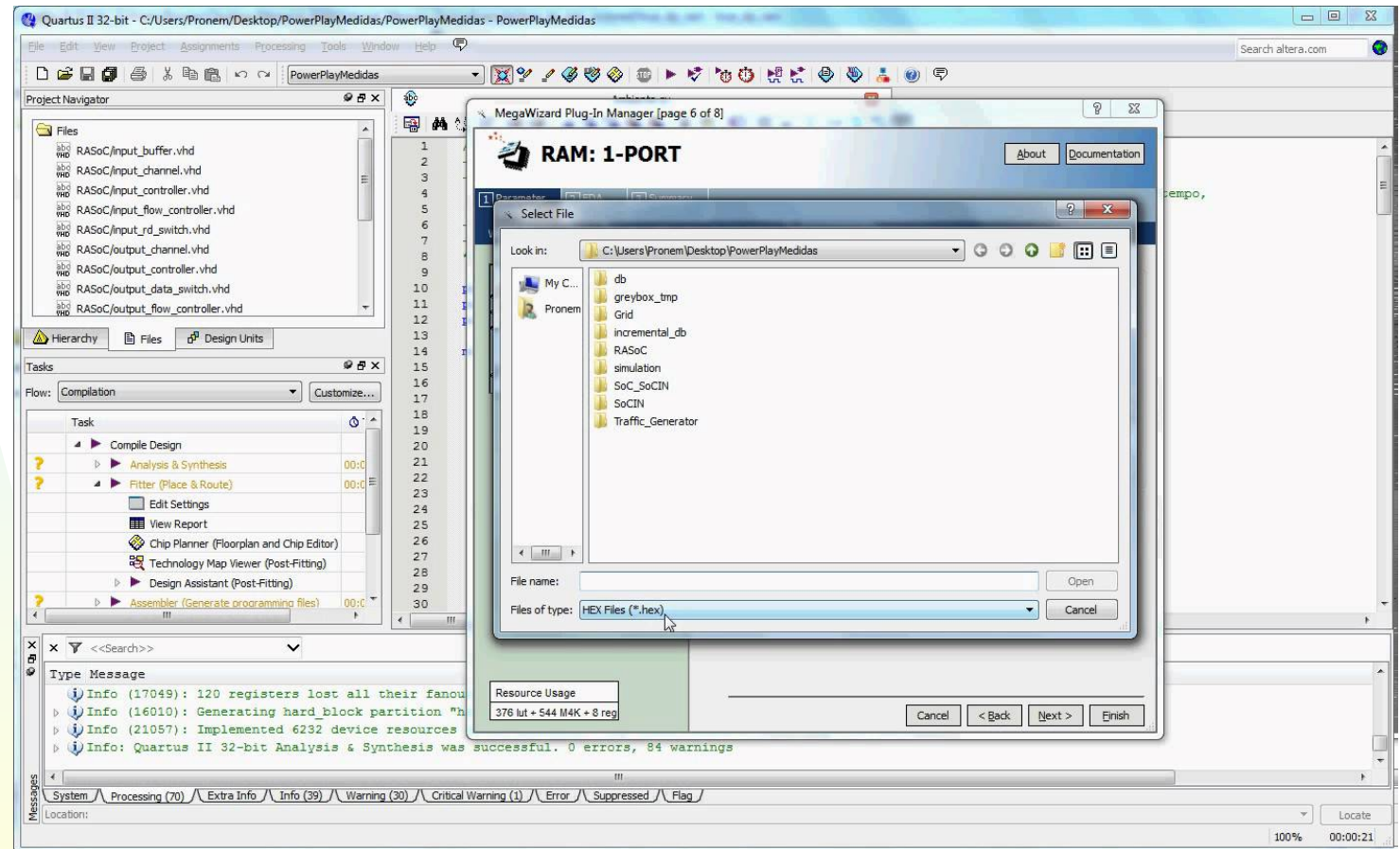
# FPGA design software

- FPGA companies provide/sell development software for their own products:
  - ◆ Vivado (Xilinx)
  - ◆ Quartus (Altera)
- Some third-party software (general-purpose)
  - ◆ Mentor Graphics HDL designer
  - ◆ Synopsis Synplify Premier
- Open-source tools:
  - ◆ Editing: Emacs (good VHDL and Verilog support)
  - ◆ Simulation: GHDL, Verilator, ...
- With third-party tools, proprietary software still needed for final implementation



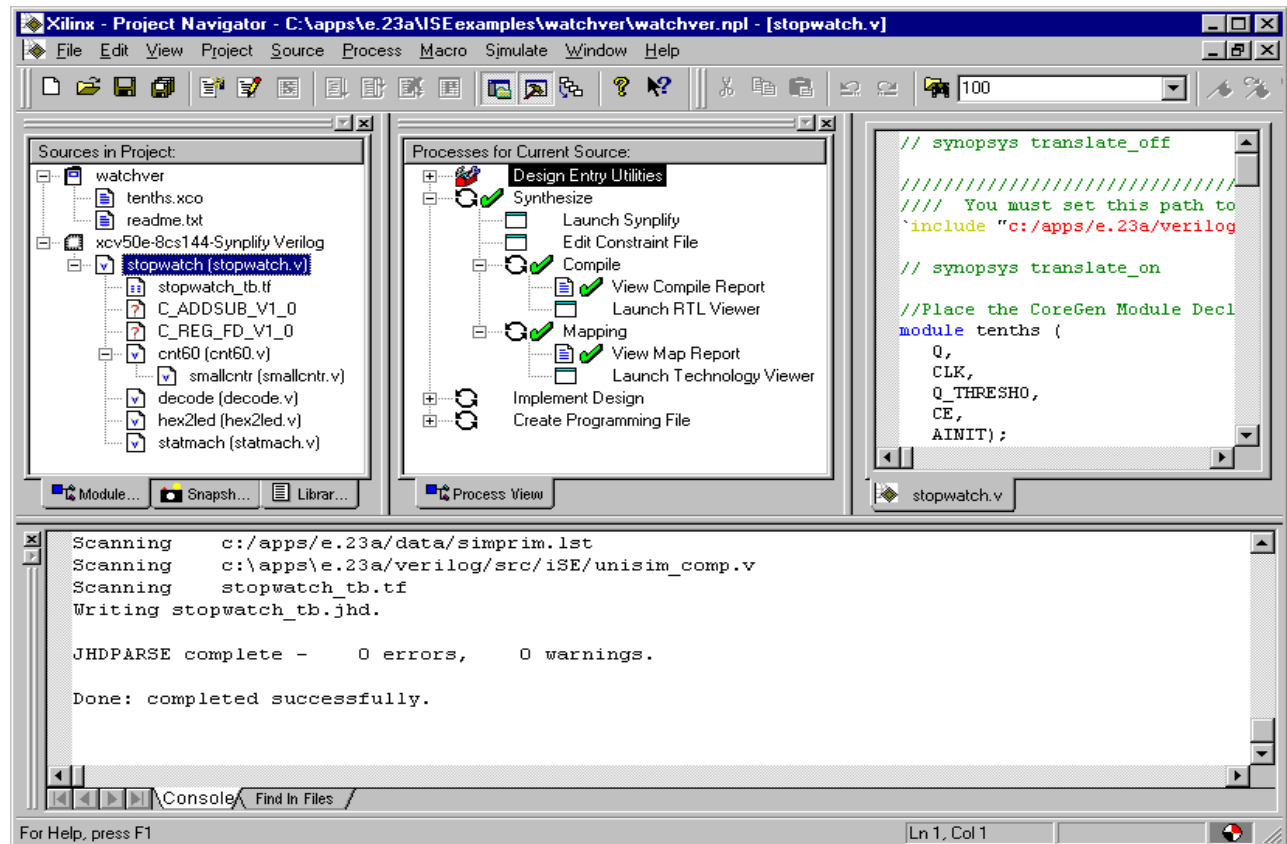
# Altera (Quartus II)

- Fully integrated design tool:
  - ◆ Multiple design entry methods
    - ✦ HDL
    - ✦ Built-in graphical editor
  - ◆ Logic synthesis
  - ◆ Implementation
  - ◆ Simulation
  - ◆ Timing & power analysis
  - ◆ FPGA configuration

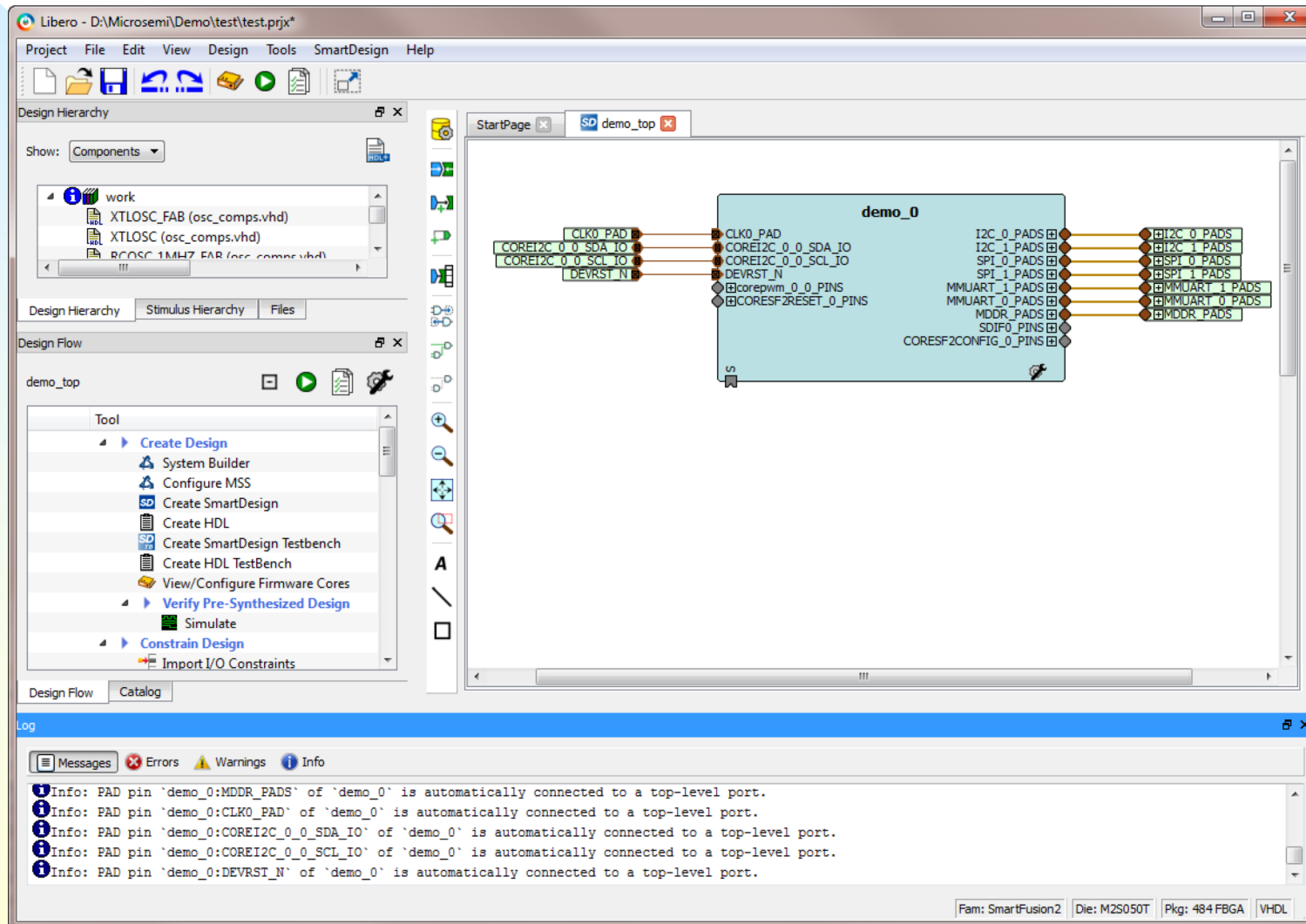


# Xilinx ISE (old)

- ◆ Complete design package
  - ◆ Design Entry (Graphical, HDL)
  - ◆ Synthesis (XST)
  - ◆ Implementation (Translate, Map, Place & Route)
  - ◆ Simulation (iSim)
  - ◆ iMPACT Programmer (Download cable, PROM generation)
- ◆ CORE Generator
  - Create customizable design units



# Actel Libero



# Xilinx Vivado (this course)

The screenshot displays the Xilinx Vivado IDE interface. The top menu bar includes File, Edit, Flow, Tools, Window, Layout, View, and Help. The main workspace is divided into several panels:

- Flow Navigator:** Shows project settings, simulation, RTL analysis, synthesis, implementation, and program and debug options.
- Project Manager:** Displays the project hierarchy, including Design Sources and Constraints.
- IP Properties:** Shows details for the Soft Error Mitigation IP, including Version (3.3), Part status (Pre-production), License (Included), Vendor (Xilinx, Inc.), and IP library (ip). The description states: "The Xilinx Soft Error Mitigation IP solution provides a pre-verified design which can detect and optionally correct and classify soft errors in Configuration Memory. A soft error is an unintended change to the state of memory bits caused by ionizing radiation. The solution does not..."
- IP Catalog:** A searchable list of IP blocks, with Soft Error Mitigation selected. The catalog shows various categories like Automotive & Industrial, AXI Infrastructure, BaseIP, Basic Elements, Communication & Networking, Debug & Verification, Digital Signal Processing, Embedded Processing, FPGA Features and Design, XADC, Math Functions, Memories & Storage Elements, Standard Bus Interfaces, and Video & Image Processing.
- Design Runs:** A table showing the status of synthesis and implementation runs.

Name	Part	Constraints	Strategy	Status	Progress	Start	Elapsed
synth_1	xc7k325tffg900-2	constrs_1	Vivado Synthesis Defaults (Vivado Synthesis 2012)	Not started	0%		
impl_1	xc7k325tffg900-2	constrs_1	Vivado Implementation Defaults (Vivado Implementation 2012)	Not started	0%		

# Many similarities between different design suites

- Overall layout similar to software development tools
- Same basic design flow
  - ◆ Design entry
  - ◆ Simulation
  - ◆ Implementation
  - ◆ Programming/debugging
- Companies have different design philosophies, and hardware/software are different “under the hood”
- Transitioning to different devices/software tools is not always trivial, but basic design skills and code are portable, especially if you design in HDL

# In this course we will use:

- Device: Xilinx FPGA (Artix 7)
- Design language: VHDL
- Development environment: Xilinx Vivado
  
- These are specific choices, but
  - ◆ Other environments are not that different
  - ◆ Not difficult to transfer skills

# Introduction to VHDL

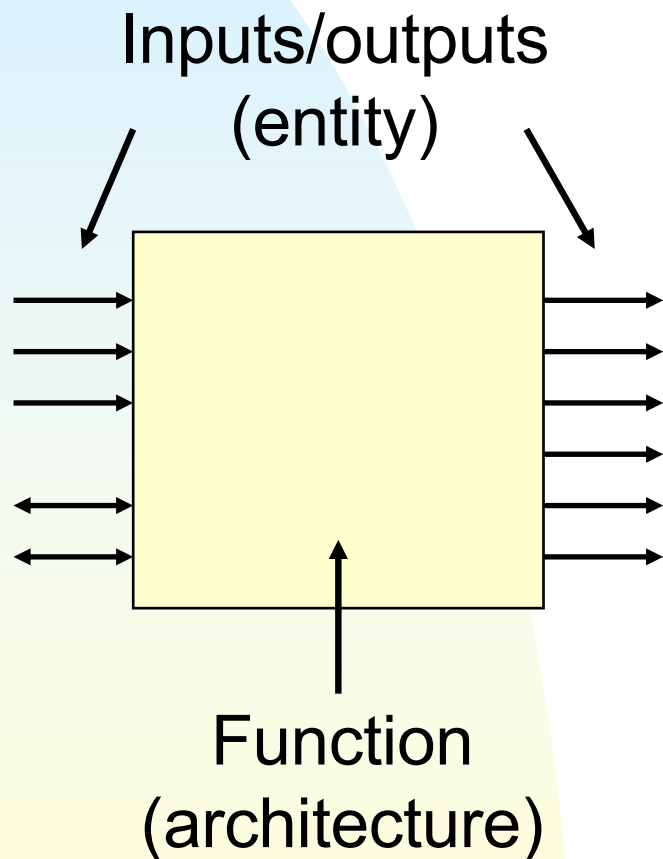
- **VHSIC Hardware Description Language**
  - ◆ (VHSIC: Very high speed integrated circuit)
- Both concurrent and sequential operations
- International standard
  - ◆ Revisions: 1987, 1993, 2002, 2008
  - ◆ Pure language definition
- Large standard, with multiple ways to code the same behavior
  - ◆ This course covers a “useful” subset of VHDL

# Describe circuits at different levels of abstraction

- Behavioral Level
  - ◆ Pure functional description
- Register-Transfer Level (RTL)
  - ◆ Structures and timing
- Logic (gate) level
- Switch level
  - ◆ Physical elements of FPGA/ASIC (LUTs, flip-flops, etc) and connections between them

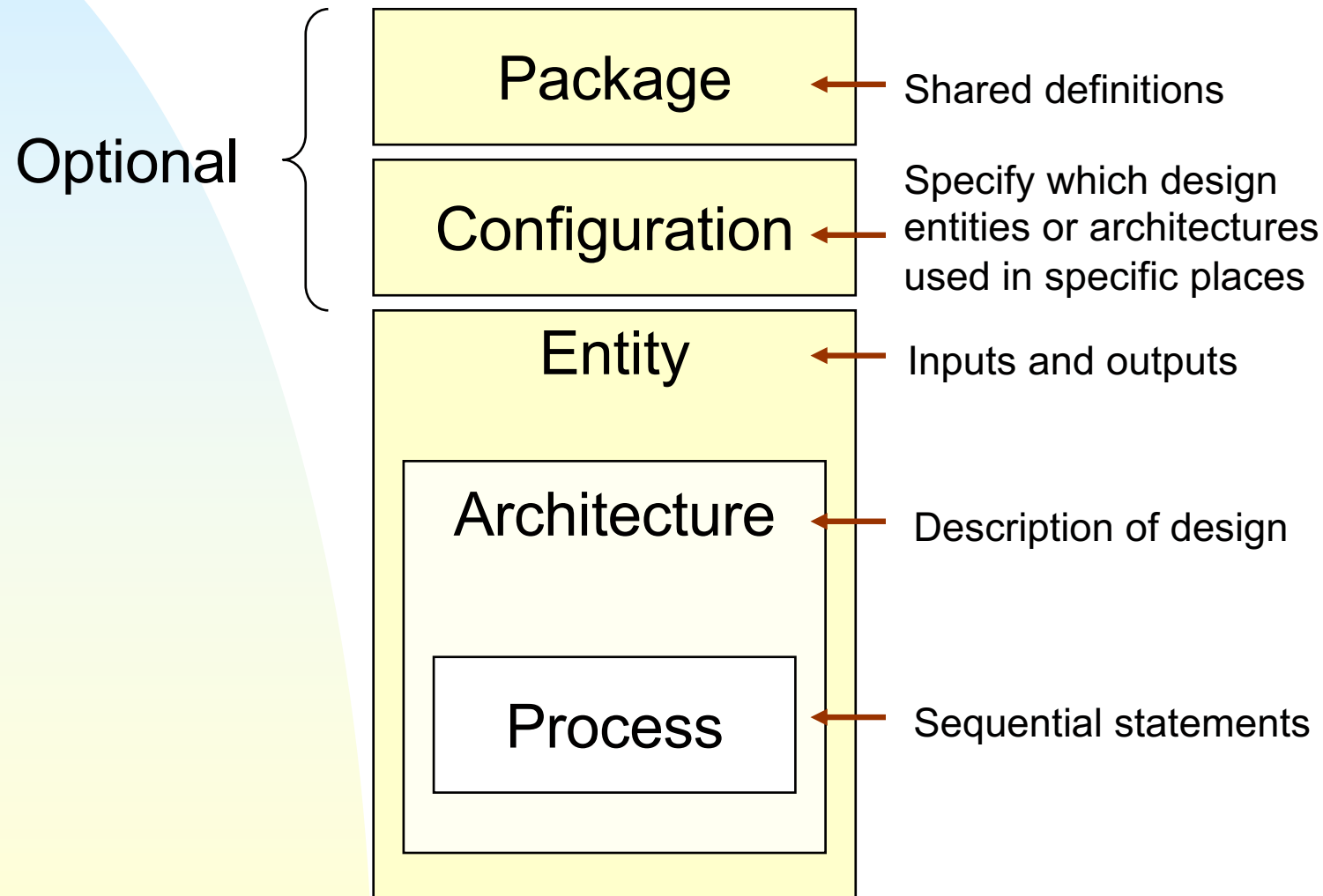


# Design unit (library unit)



- In most basic form, defines:
  - ◆ Inputs/outputs (entity)
  - ◆ Function of unit (architecture)
- Can be instantiated and connected together in higher level design units.

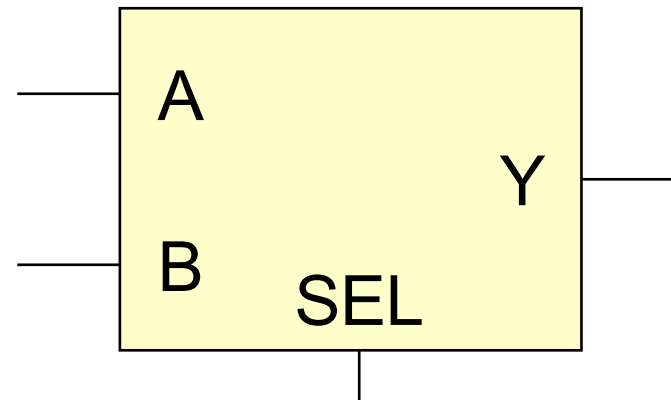
# VHDL design unit



# Example: Multiplexer

- Switches between inputs
  - ◆  $Y = A$  when  $SEL = 1$
  - ◆  $Y = B$  when  $SEL = 0$

A	B	SEL	Y
a	b	1	a
a	b	0	b

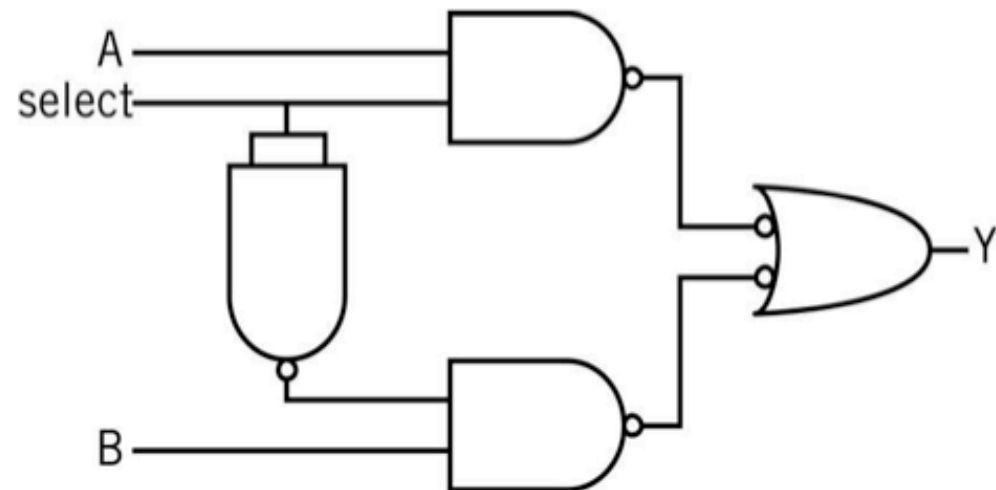


# Implementation in logic gates

(from previous lecture)

$$\begin{aligned} Y &= (A \cdot S) + (B \cdot \bar{S}) \\ &= \overline{\overline{(A \cdot S)}} + \overline{\overline{(B \cdot \bar{S})}} \\ &= \overline{(A \text{ nand } S)} + \overline{(B \text{ nand } \bar{S})} \end{aligned}$$

A	B	SEL	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



# Basic steps for coding a design unit in VHDL

- Define the ENTITY
  - ◆ Defines input and output ports
- Write the ARCHITECTURE
  - ◆ One (or more) for each entity
  - ◆ Describes the internal behavior
  - ◆ Can be written at different levels
    - ✦ Behavioral, RTL, etc.

# IEEE standard logic

Used in this course, and in most modern code

Start with this:

```
library ieee;  
use ieee.std_logic_1164.all;
```

- Standard data type introduced in late 1980s
- Data can have nine different values
  - ◆ '0' '1' : Driven logic 0 or 1
  - ◆ 'L' 'H' : Read logic (0 or 1)
  - ◆ 'Z' : High impedance
  - ◆ 'W' : Weak 1
  - ◆ 'U' : Uninitialized
  - ◆ 'X' : Unknown
  - ◆ '-' : Don't care

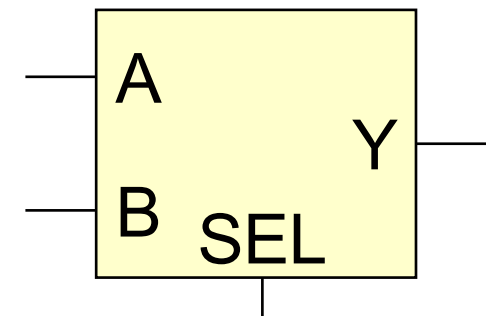
# Multiplexer entity

```
library ieee;  
use ieee.std_logic_1164.all;  
  
entity Mux is  
port (a:    in std_logic;  
      b:    in std_logic;  
      sel:  in std_logic;  
      y:    out std_logic  
      );  
end Mux;
```

Port name

Direction

Type



no semicolon  
after last port  
declaration!

# You can declare signals with one or many bits

- `std_logic`
  - ◆ Single bit
- `std_logic_vector (n downto 0)`
  - ◆ Vector with n+1 bits
  - ◆ Index range can be counted up or down:
    - ✦ Count down: (7 downto 4)
    - ✦ Count up: (0 to 9)



# Mux architecture (example):

Name of architecture

Associated entity

architecture arch1 of Mux is

```
signal c, d: std_logic; -- Internal signals
```

comment

```
begin
```

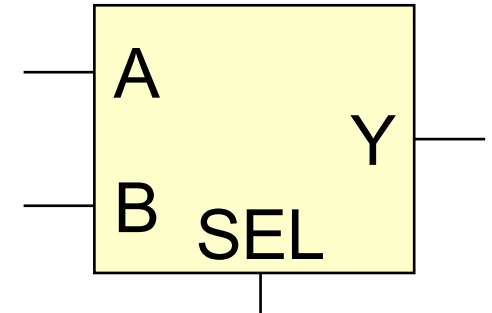
```
    c <= a and sel;
```

```
    d <= b and (not sel);
```

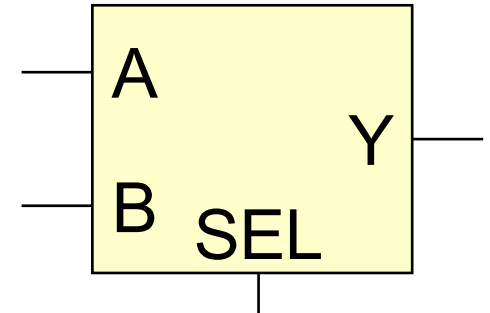
```
    y <= c or d;
```

Concurrent statements

```
end arch1;
```



# Can use longer expressions for more compact code



architecture arch2 of Mux is

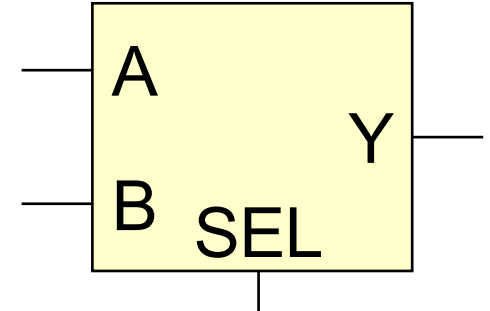
-- Note: no internal signals needed!

begin

    y <= (a and sel) or (b and (not sel));

end arch2;

# Behavioral example (when...else)



architecture arch3 of Mux is

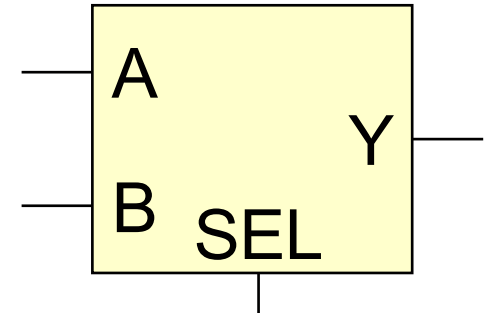
begin

```
y <= a when sel='0' else b;
```

end arch3;

"conditional signal assignment"

# Example: 'with..select':



```
architecture arch4 of Mux is
begin
    with sel select
        y <= a when '0',
            b when OTHERS;
end arch4;
```

“selected concurrent signal assignment”

# Note: single bit vs. vector representation

- Bits represented with single quotes
  - ◆ `a <= '0';`
  - ◆ `b <= 'Z';`
- Vectors represented with double quotes
  - ◆ `vector_1 <= "00110";`
  - ◆ `vector_2 <= "ZZZZZ";`

# Processes are containers for sequential code

- Example: `process (a, b, sel)`
  - ◆ The process is sensitive to the signals listed in the brackets.
- structures used in processes include:
  - ◆ if...then
  - ◆ case
  - ◆ loop
  - ◆ Etc.

# case statement:

(Sequential)

architecture arch5 of Mux is

begin

    process (a, b, sel)

    begin

        case sel is

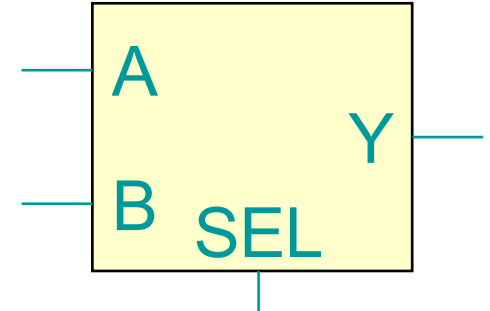
            when '1' => y <= a;

            when others => y <= b;

        end case;

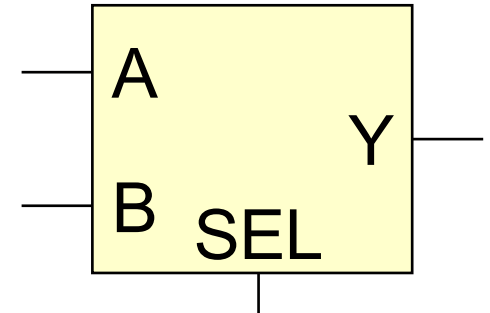
    end process;

end arch5;



# If...elsif...then

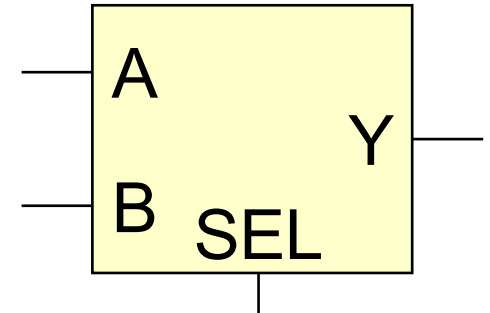
```
architecture arch6 of Mux is
begin
    process (a, b, sel)
    begin
        if (sel='1') then
            y <= a;
        elsif (sel='0') then
            y <= b;
        else
            y <= 'z';
        end if
    end process;
end arch6;
```



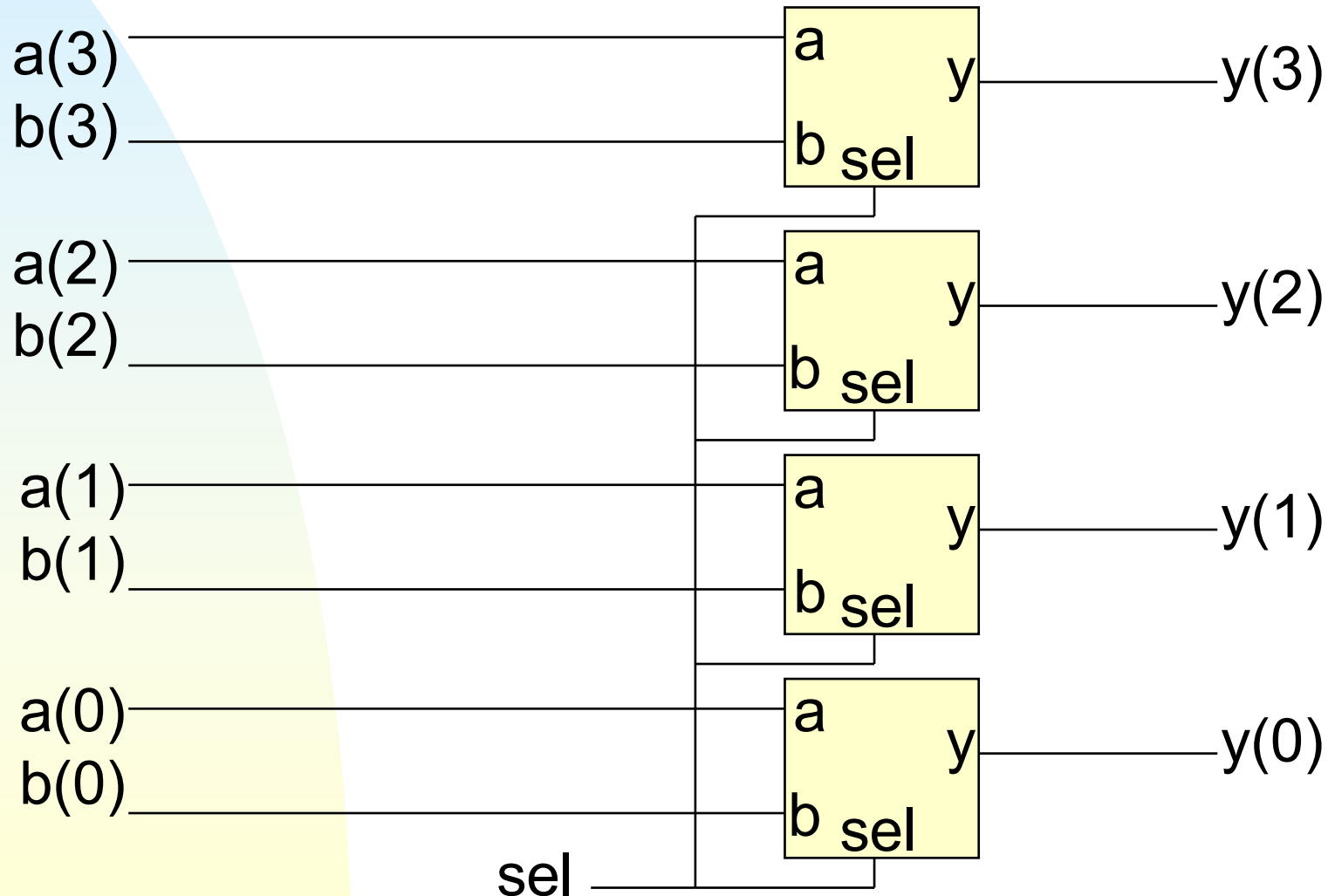


# Complete design unit:

```
--  
-- A 2 input multiplexer circuit  
--  
library ieee;  
use ieee.std_logic_1164.all;  
  
entity Mux is  
port(  a:      in std_logic;  
       b:      in std_logic;  
       sel:    in std_logic;  
       y:      out std_logic  
);  
end Mux;  
  
architecture arch2 of Mux is  
begin  
    y <= (a and sel) or (b and (not sel));  
  
end arch2;
```



# Entities can be instantiated as components



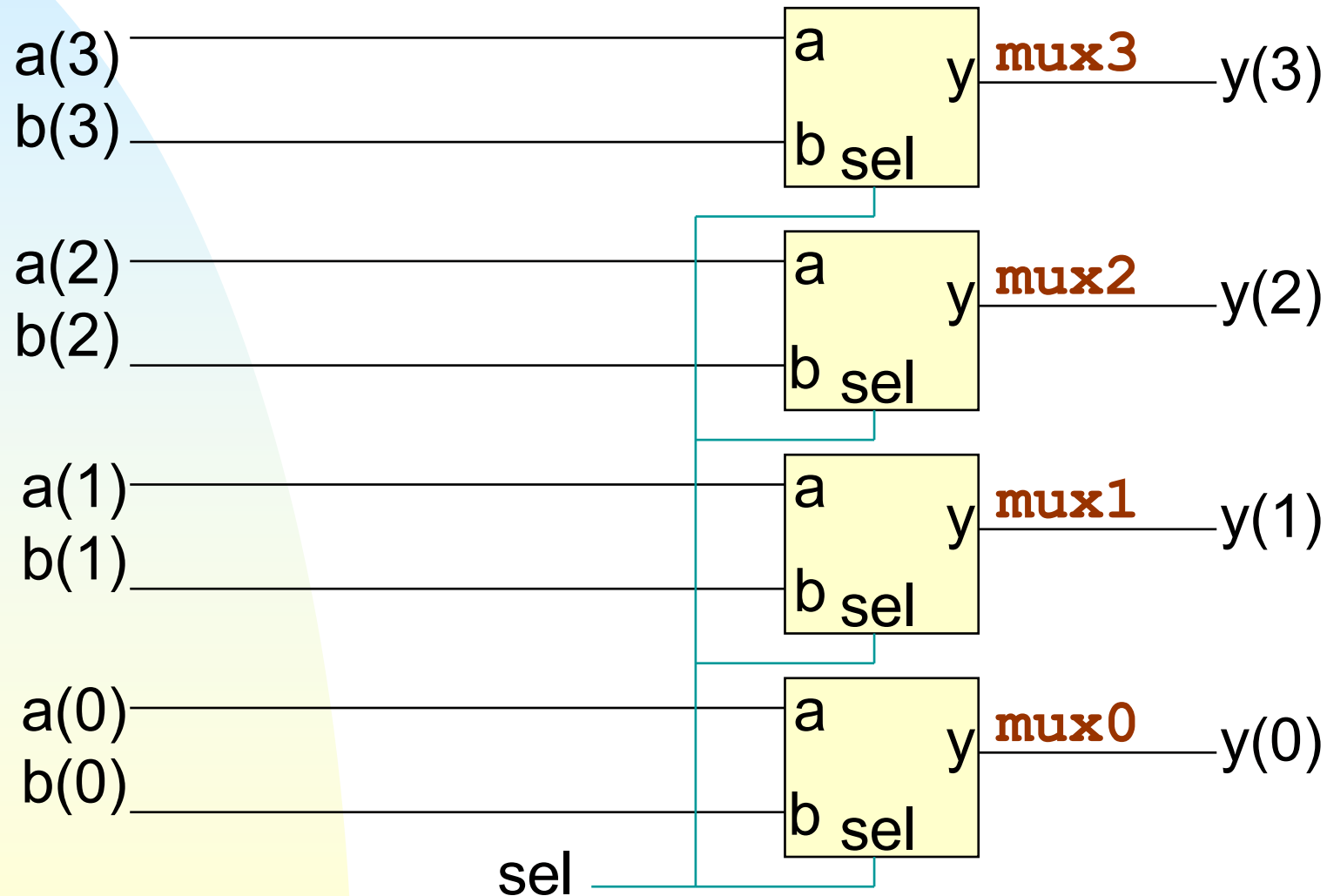
# Instantiating a component

- Declare the component in the higher-level architecture (before 'begin')
  - ◆ Not strictly required, but good practice
  - ◆ Syntax is almost identical to entity declaration, so you can usually cut-and-paste
- Instantiate the component in the architecture:
  - ◆ Give each instance a unique name
  - ◆ Follow with the entity name of the component
  - ◆ Map signals to the component's ports
    - ✦ Possibly generics too (will get to that...)

# Example: 4-bit MUX using four 1-bit MUXes

```
entity Mux42 is
port (a:    in std_logic_vector (3 downto 0);
      b:    in std_logic_vector (3 downto 0);
      sel:  in std_logic;
      y:    out std_logic_vector (3 downto 0)
);
end Mux42;
```

# Schematic of 4-bit MUX:



# Architecture (example)

architecture arch of Mux42 is

**component Mux** -- **Declare our 1-bit Mux component**

```
port( a:      in std_logic;
      b:      in std_logic;
      sel:    in std_logic;
      y:      out std_logic
    );
```

end component;

for all : Mux use entity work.Mux; -- **Configuration**

begin

**mux0:** Mux

```
port map (a=>a(0), b=>b(0), sel=>sel, y=>y(0));
```

**mux1:** Mux

```
port map (a=>a(1), b=>b(1), sel=>sel, y=>y(1));
```

**mux2:** Mux

```
port map (a=>a(2), b=>b(2), sel=>sel, y=>y(2));
```

**mux3:** Mux

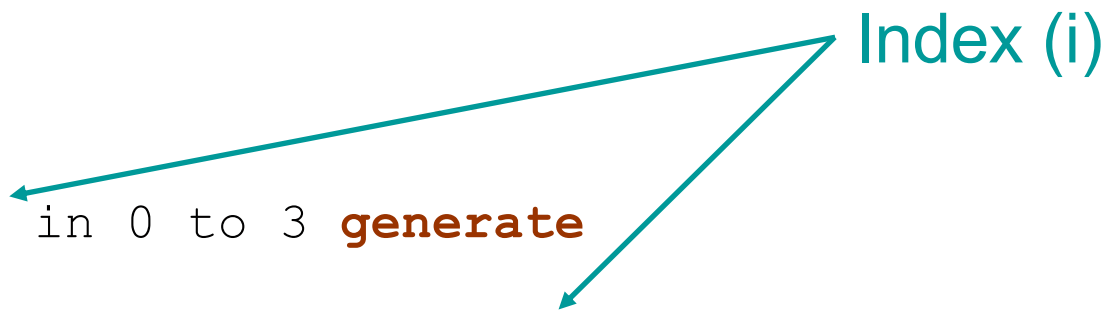
```
port map (a=>a(3), b=>b(3), sel=>sel, y=>y(3));
```

end arch;

# VHDL has a generate statement for repetitive structures

```
architecture arch1 of Mux42 is
  component Mux -- Use our Mux component
    port( a:      in std_logic;
          b:      in std_logic;
          sel:    in std_logic;
          y:      out std_logic
        );
  end component;

begin
  gen0: for i in 0 to 3 generate
    mux1: Mux
      port map (a=>a(i), b=>b(i), sel=>sel, y=>y(i));
  end generate gen0;
end arch1;
```



Index (i)

# Generics in VHDL

- Pass parameters to a component.
- Contain static information to define:
  - ◆ Structure
    - ✦ E.g. size/width of a generic component
  - ◆ Behavior
    - ✦ E.g. initialization values, whether to use rising/falling clock edge, etc.
- ◆ Only used to direct synthesis
- Must be declared with a default value



# Why use a generic?

- Example: add a time delay to the output
  - ◆ (Only useful in simulation!)

```
architecture arch2 of Mux is  
  
begin  
    y <= (a and sel) or (b and (not sel)) after 2 ns;  
  
end arch2;
```

- Problem: this is a fixed value!
- What if you want to be able to change it for different instantiations?

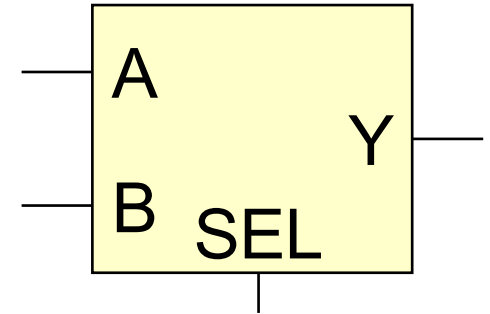
# Declaring and using a generic

```
library ieee;
use ieee.std_logic_1164.all;

entity Mux is
  generic (mux_delay: time := 2 ns);
  port(
    a:    in std_logic;
    b:    in std_logic;
    sel:  in std_logic;
    y:    out std_logic
  );
end Mux;

architecture arch2 of Mux is
begin
  y <= (a and sel) or (b and (not sel)) after mux_delay;

end arch2;
```



Declare generic  
in entity

Use in architecture

# Instantiating the component

```
architecture arch1 of my_entity is
  component Mux
    generic (mux_delay: time := 2 ns);
    port( a:      in std_logic;
          b:      in std_logic;
          sel:    in std_logic;
          y:      out std_logic
        );
  end component;
```

```
begin
  mux1: Mux
    generic map (mux_delay => 4 ns) -- no semicolon yet
    port map (a => A, b => B, sel => SEL, y => Y);
end arch1;
```

Instantiate with different delay



# More useful generic example

```
architecture arch1 of my_entity is
  component inverter
    generic (width: integer := 4); -- default parameter
    port( in:    in std_logic_vector (width-1 downto 0);
          out:   in std_logic_vector (width-1 downto 0);
    end component;

  signal input, output: std_logic_vector (7 downto 0);

begin
  inv1: inverter
    generic map (width => 8) -- no semicolon yet
    port map (in=>input, out=>output);
end arch1;
```

Instantiate with a different width



# Test benches in VHDL

- Used to simulate and test design at multiple stages:
  - ◆ VHDL code
  - ◆ Post-synthesis translation
  - ◆ Detailed post-implementation timing model
- Usually written in non-synthesizable VHDL
- Main parts:
  - ◆ Unit under test (UUT) instantiated as a component
  - ◆ Stimulus (input signals) to UUT
  - ◆ Optionally, compare UUT outputs to expected results
- Usually instantiate UUT in TB architecture, implement stimulus/testing using processes

# Test bench: entity

```
--  
-- VHDL template for creating test benches  
--  
-- Library declarations: Add/change as needed  
LIBRARY ieee;  
USE ieee.std_logic_1164.ALL;  
USE ieee.std_logic_unsigned.all;  
  
-- entity declaration for your testbench (blank)  
  
ENTITY test_tb IS  
END test_tb;
```

# UUT and signal declaration

```
ARCHITECTURE behavioral OF test_tb IS
  -- Component Declaration: Unit Under Test (UUT)
  COMPONENT Mux
    PORT (
      a: in std_logic;
      b: in std_logic;
      sel: in std_logic;
      y : out std_logic);
  END COMPONENT;

  --Signal definitions:

  signal a, b, sel, y: std_logic := '0';

  -- Clock definitions (for clocked designs)
  signal clk: std_logic := '0';
  constant clk_period : time := 10 ns;
```

# UUT instantiation and clock generation

```
BEGIN

-- Clock process (toggle clock after each half period)
clk_process : process
begin
    clk <= not(clk);
    wait for clk_period/2;
end process;

-- Instantiate the Unit Under Test (UUT).
    uut: Mux
        port map (a => a, b => b, sel => sel, y => y);
```



# Simple stimulus

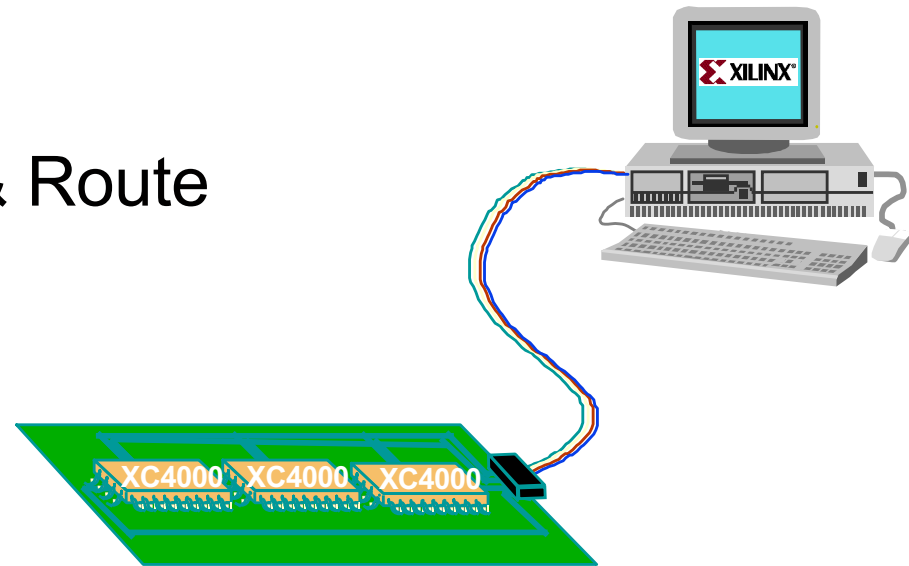
```
-- Stimulus process
```

```
stim_proc: process  
begin  
    wait for clk_period;  
    a <= '1';  
    b <= '0';  
    y <= '0';  
    wait for clk_period  
    y <= '1';  
    wait_for_clk_period;  
    a <= '0';  
  
    wait;  
end process;
```

```
END;
```

# FPGA Design Flow

- Design entry
- Simulation
  - ◆ Create stimulus/test bench
  - ◆ Compile and run in simulator
- Implementation
  - ◆ Synthesis,
  - ◆ Translation, Place & Route
  - ◆ Generate bitstream
  - ◆ Analyze timing, etc
- Download to hardware



# Vivado design environment

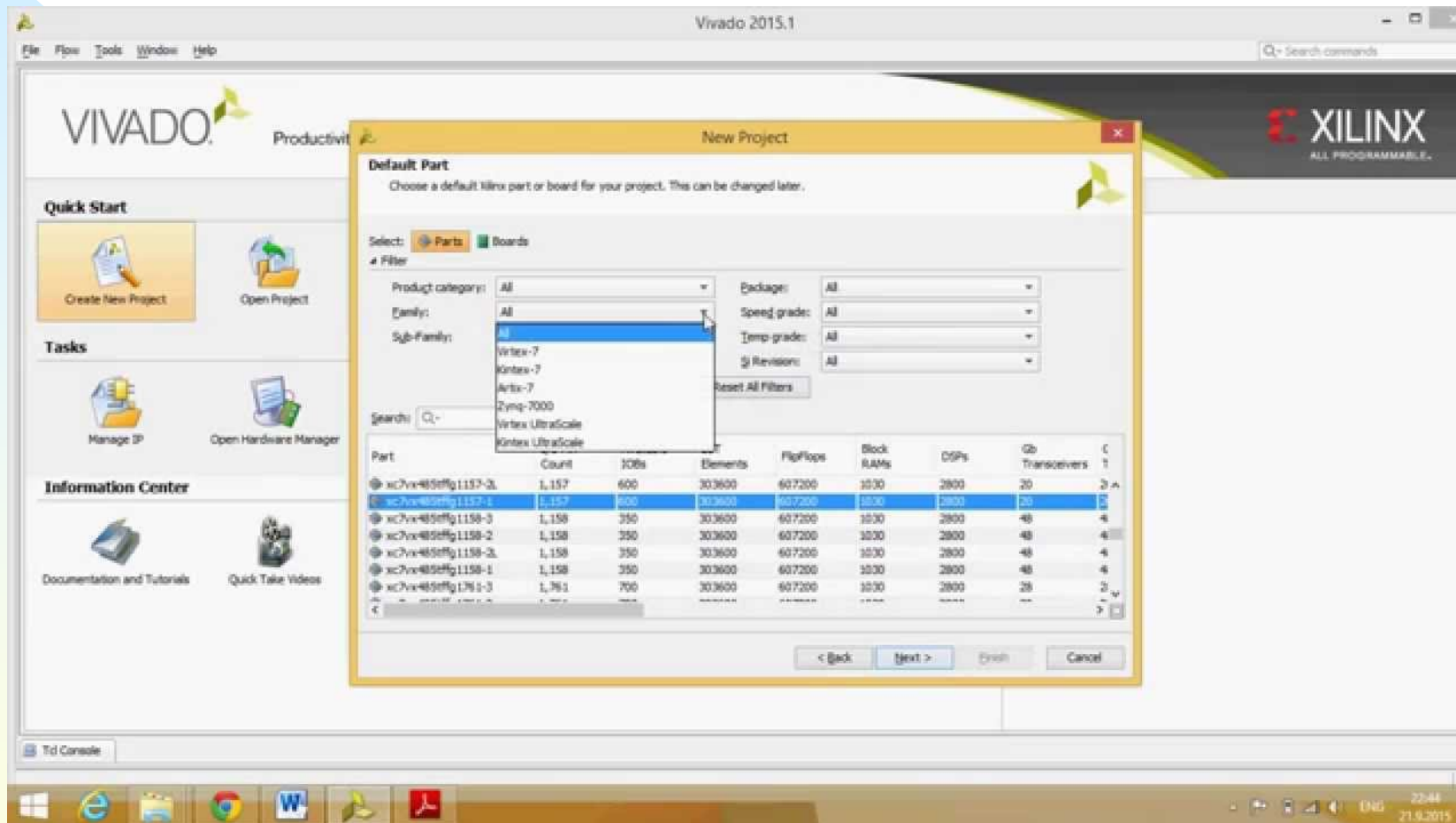
The screenshot displays the Vivado design environment interface. The main window is titled "Project Manager - project\_1". The left sidebar contains a "Flow Navigator" with various design steps like "Project Manager", "Simulation", "RTL Analysis", "Synthesis", "Implementation", and "Program and Debug". The central area is divided into several panes:

- Project Manager:** Shows "Sources" (Design Sources, Constraints (1)) and "Hierarchy" (Libraries, Compile Order).
- IP Properties:** Displays details for the "Soft Error Mitigation" IP, including Version (3.3), Part status (Pre-production), License (Included), Vendor (Xilinx, Inc.), and IP library (ip). The description states: "The Xilinx Soft Error Mitigation IP solution provides a pre-verified design which can detect and optionally correct and classify soft errors in Configuration Memory. A soft error is an unintended change to the state of memory bits caused by ionizing radiation. The solution does not..."
- IP Catalog:** A searchable list of IP blocks. The "Soft Error Mitigation" IP is highlighted, showing its version (3.3) and status (Pre-production, Included).
- Design Runs:** A table showing the status of design runs.

Name	Part	Constraints	Strategy	Status	Progress	Start	Elapsed
synth_1	xc7k325tffg900-2	constrs_1	Vivado Synthesis Defaults (Vivado Synthesis 2012)	Not started	0%		
impl_1	xc7k325tffg900-2	constrs_1	Vivado Implementation Defaults (Vivado Implementation 2012)	Not started	0%		

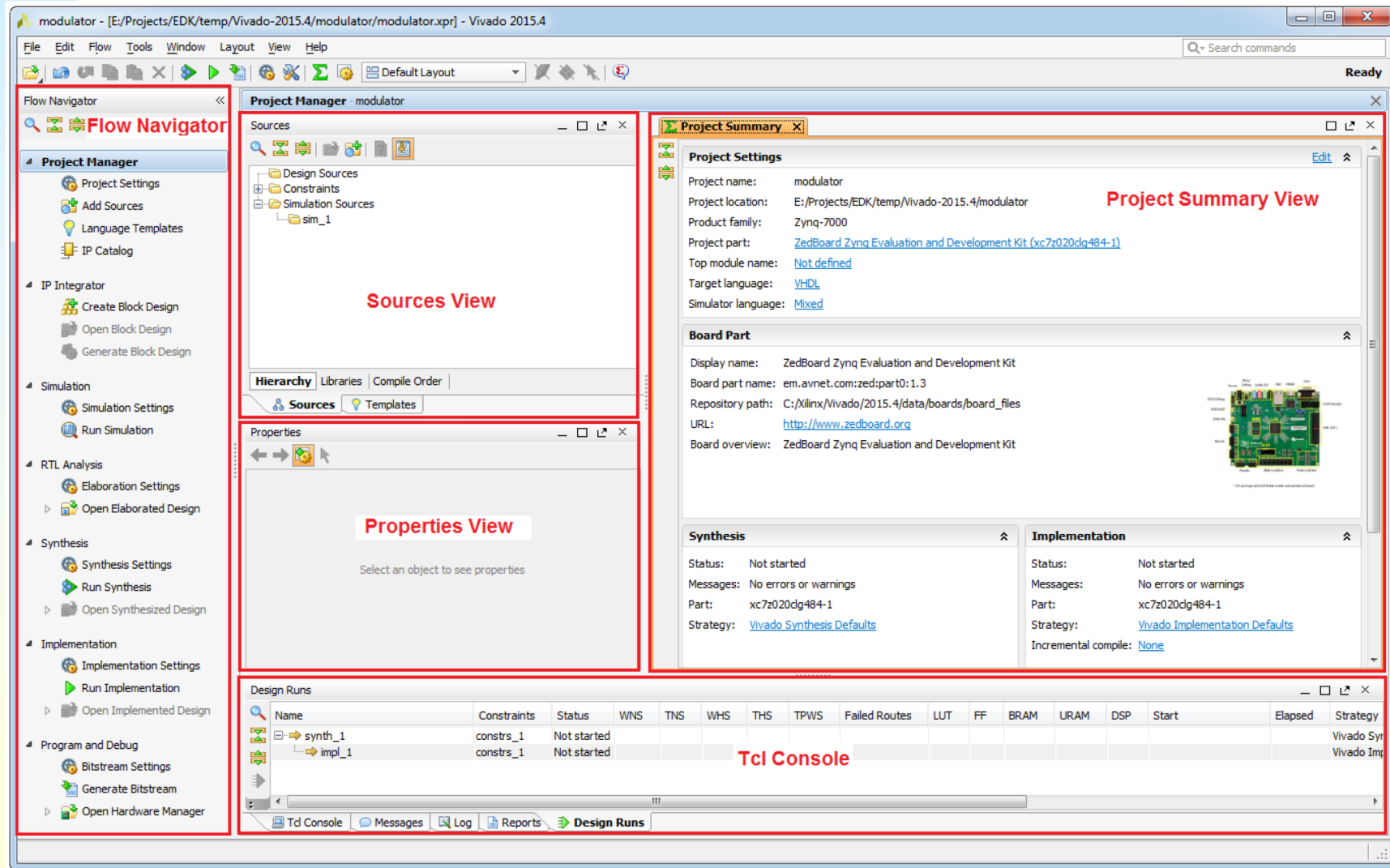
At the bottom right, a yellow box contains the text: "So, how do you use it?"

# Create a new project



Define HDL project, target device, source files (if desired)

# Vivado IDE layout



# Flow navigator

The image shows a screenshot of the Vivado 2015.1 software interface. On the left, the 'Flow Navigator' pane is visible, containing a tree view of project tasks such as 'Project Manager', 'IP Integrator', 'Simulation', 'RTL Analysis', 'Synthesis', 'Implementation', and 'Program and Debug'. A red oval highlights this pane. In the center, the 'Block Design' window shows the project 'modulator\_ipi'. A red arrow points from the 'Properties' tab in this window to the 'Behavioral Simulation' step in the flow navigator diagram. The flow navigator diagram itself is a vertical sequence of eight blue rounded rectangular boxes connected by downward-pointing arrows: 'RTL Design', 'Behavioral Simulation', 'Synthesize', 'Post Synthesis Simulation', 'Implement (Place&Route)', 'Post Implementation Simulation', and 'Debug'. The 'Behavioral Simulation' step is highlighted with a red border. The bottom of the screenshot shows the 'Tcl Console' with various commands and status messages.

# User I/O planning

The screenshot shows the Vivado 2013.2 I/O Planning tool. The main window displays a grid of I/O pins with various constraints and settings. The 'I/O Planning' menu item is circled in red. The 'I/O Ports' table at the bottom is also circled in red.

Name	Direction	Neg Diff Pair	Site	Fixed	Bank	I/O Std	Vcco	Vref	Drive Stre...	Slew
All ports (4)										
Scalar ports (4)										
CLK	Input		AB2	<input checked="" type="checkbox"/>	34	LVC MOS 18		1.800		
DIN	Input		AD1	<input checked="" type="checkbox"/>	34	LVC MOS 18		1.800		
QN	Output		AE1	<input checked="" type="checkbox"/>	34	LVC MOS 18		1.800	12	SLO
QP	Output		AE3	<input checked="" type="checkbox"/>	34	LVC MOS 18		1.800	12	SLO

# Simulation

The screenshot displays the Vivado 2015.1 simulation environment. The main window shows a timing diagram for a simulation run. The diagram has a time axis from 999,995 ps to 1,000,000 ps. A vertical yellow line marks the 1,000,000 ps point. The diagram shows various signals, including logic signals (U, 0, 1) and arrays (e.g., I\_Fif\_Data[31:0], SR\_T1[31:0]). A red arrow points from the 'Objects' panel to the 'o\_Fif\_Data[31:0]' signal in the timing diagram.

The 'Objects' panel shows a list of objects with their names, values, and data types. The 'o\_Fif\_Data[31:0]' object is highlighted in blue.

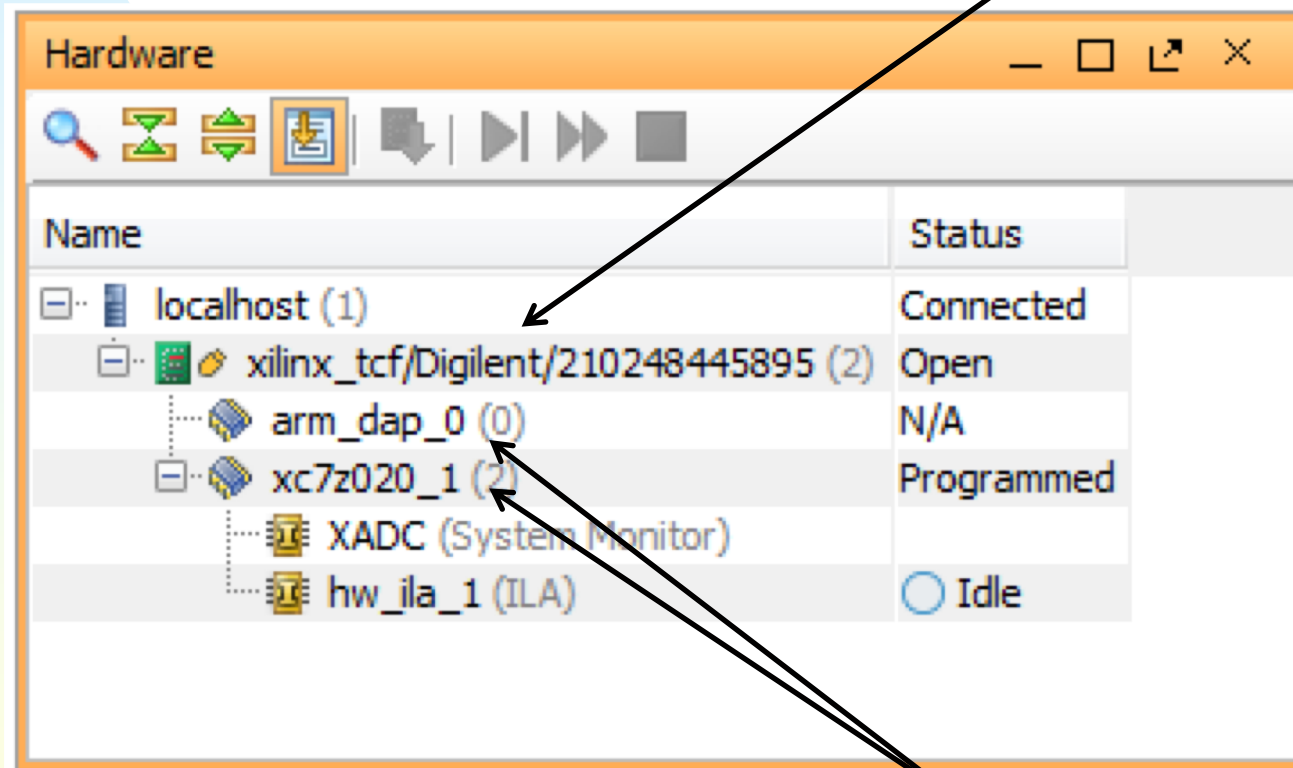
The 'Tcl Console' at the bottom shows the following output:

```
# }
# run 1000ns
INFO: [USF-XSim-96] XSim completed. Design snapshot 'Sybox_func_synth' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:05 ; elapsed = 00:00:13 . Memory (MB): peak = 1386.340 ; gain = 12.883
```



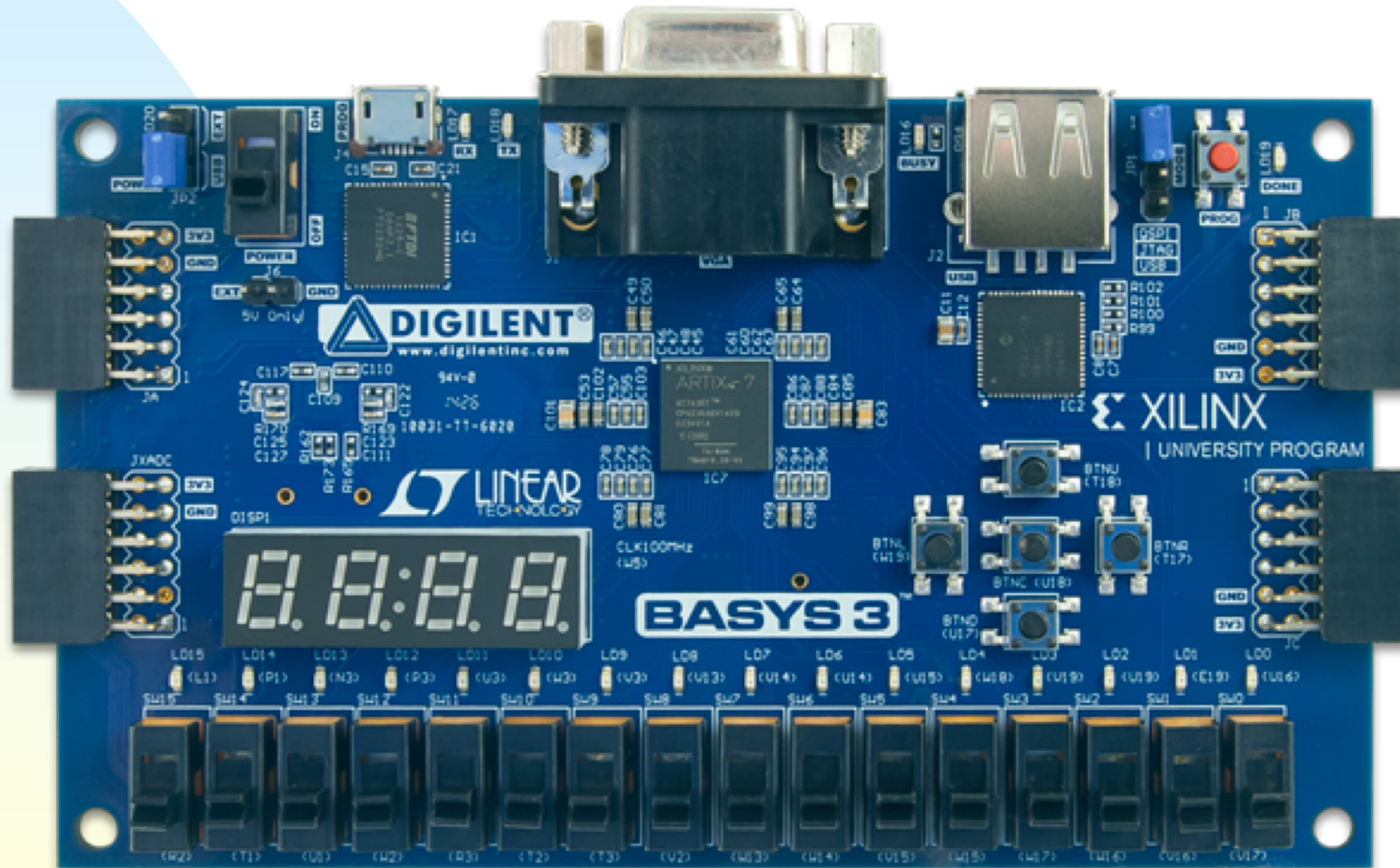
# Hardware manager

Programmer



Device(s) in chain

# Target to your hardware

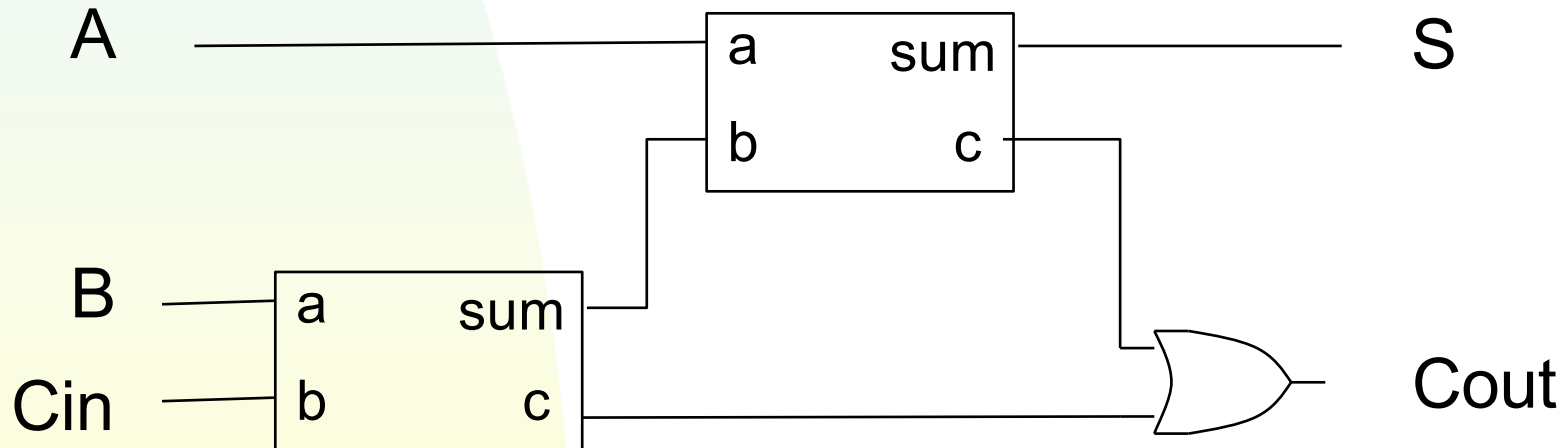


# Lab 1: Combinatorial logic

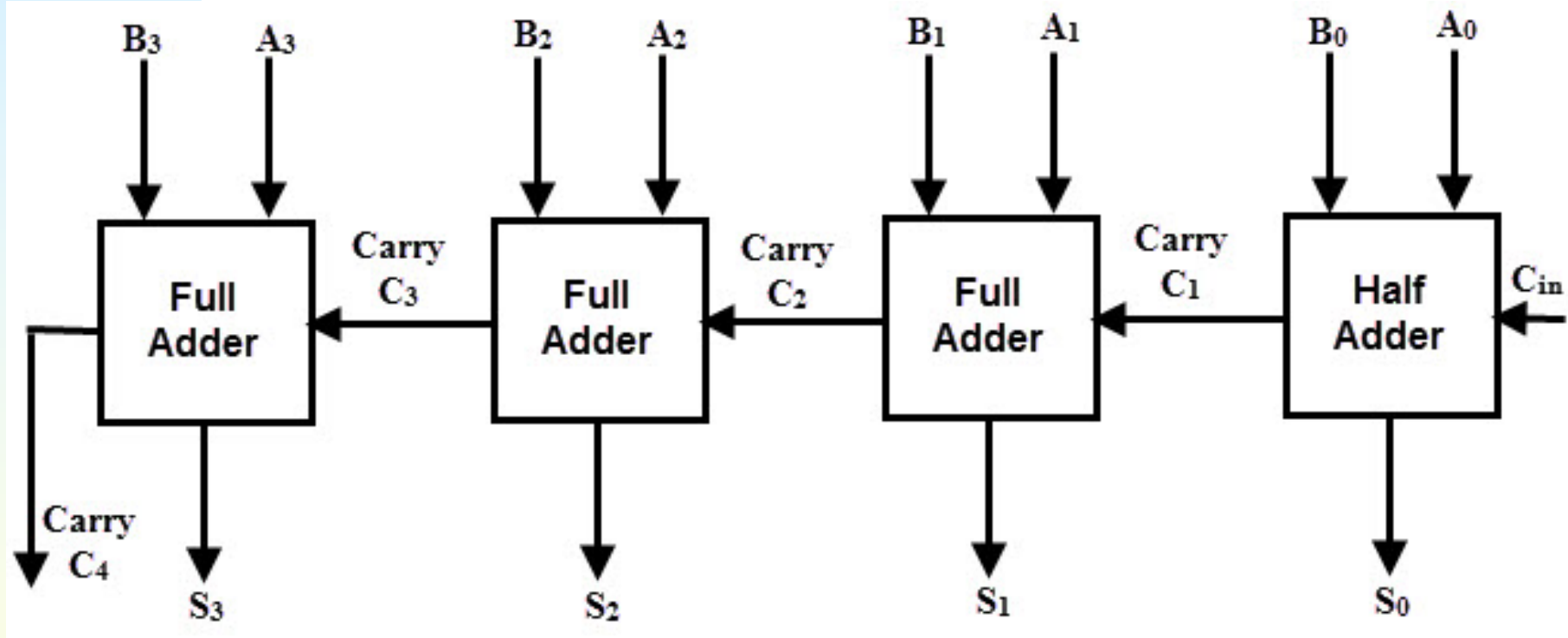
- Introduction to Xilinx Vivado:
- Implement simple logic gates (AND, OR, XOR)
  - ◆ VHDL entry
  - ◆ Simulation
  - ◆ Implement/test in FPGA
- Half-adder
- Full-adder
- Four-bit parallel adder

# Full adder (review)

- Includes carry bit from previous summation



# 4-bit parallel adder



$$A(3 : 0) + B(3 : 0) + C_{in} = S(3 : 0) + C_{out}$$