FPGA trainer board

Basys3

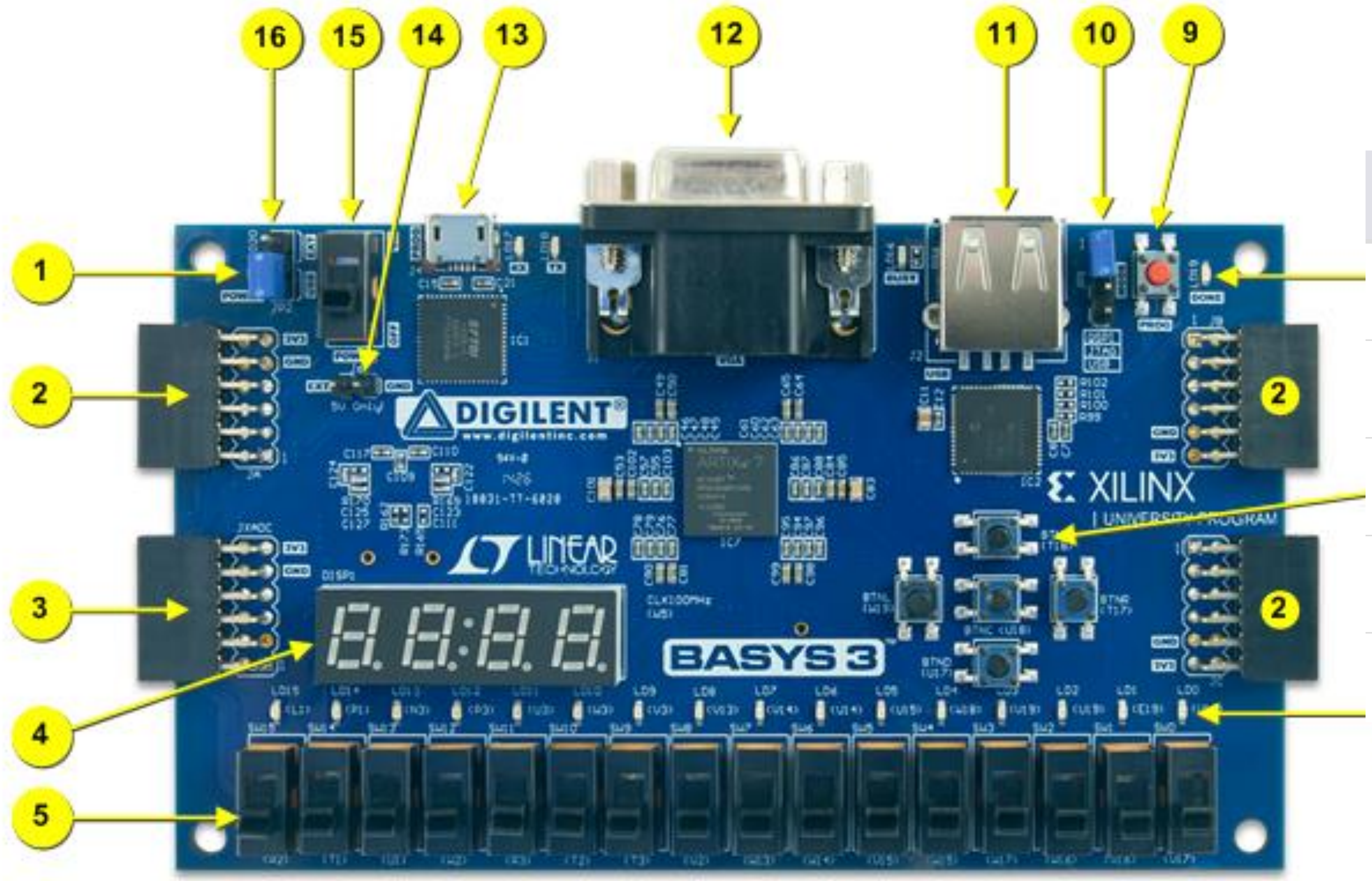
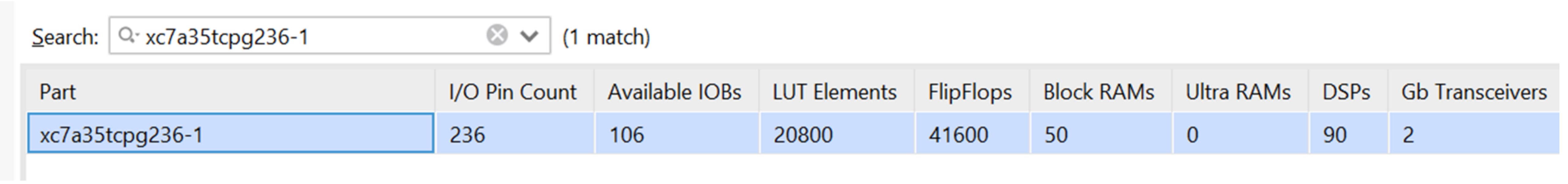


Figure 1. Basys3 board features

Basys3
Artix (XC7A35T)
33,280 logic cells
90 DSP Slices
1899Kb BRAM
Vivado

Callout	Component Description	Callout	Component Description
1	Power good LED	9	FPGA configuration reset button
2	Pmod connector(s)	10	Programming mode jumper
3	Analog signal Pmod connector (XADC)	11	USB host connector
4	Four digit 7-segment display	12	VGA connector
5	Slide switches (16)	13	Shared UART/ JTAG USB port
6	LEDs (16)	14	External power connector
7	Pushbuttons (5)	15	Power Switch
8	FPGA programming done LED	16	Power Select Jumper


Search: xc7a35tcpg236-1 (1 match)

Part	I/O Pin Count	Available IOBs	LUT Elements	FlipFlops	Block RAMs	Ultra RAMs	DSPs	Gb Transceivers
xc7a35tcpg236-1	236	106	20800	41600	50	0	90	2

Basys3

- Features the Xilinx Artix-7 FPGA: XC7A35T-1CPG236C
- 33,280 logic cells in 5200 slices (each slice contains four 6-input LUTs and 8 flip-flops)
- 1,800 Kbits of fast block RAM
- Five clock management tiles, each with a phase-locked loop (PLL)
- 90 DSP slices
- Internal clock speeds exceeding 450 MHz
- On-chip analogue-to-digital converter (XADC)
- Digilent USB-JTAG port for FPGA programming and communication
- Serial Flash
- USB-UART Bridge
- 12-bit VGA output
- USB HID Host for mice, keyboards and memory sticks
- 16 user switches
- 16 user LEDs
- 5 user pushbuttons
- 4-digit 7-segment display
- 4 Pmod ports: 3 Standard 12-pin Pmod ports, 1 dual-purpose XADC signal / standard Pmod port

Steps to Deploy on Basys 3

1. **Create a Vivado Project**:
   - Select the Basys 3 board during the project setup.
2. **Add Source Files**:
   - Include the Verilog file (`Calculator_FPGA.v`).
3. **Add Constraints**:
   - Import the XDC file to map inputs and outputs.
4. **Synthesize and Implement**:
   - Run synthesis, implementation, and bitstream generation.
5. **Program the FPGA**:
   - Load the bitstream onto the Basys 3 board using Vivado.
6. **Test the Design**:
   - Use switches to set `A`, `B`, and `mode`.
   - Observe the result on the LEDs.

Vivado is a **Xilinx** tool, which is now part of **AMD** after AMD acquired Xilinx in 2022. The Vivado Design Suite is used for designing and implementing projects on Xilinx FPGAs and SoCs.

An XDC file (Xilinx Design Constraints file) is a text-based file used in Xilinx FPGA design workflows to specify constraints for timing, clocks, I/O pin assignments, and physical design attributes. It is written in Tcl (Tool Command Language), similar to an SDC file, but tailored specifically for Xilinx tools like Vivado.

XDC file (Xilinx Design Constraints file)

```
# Switches
set_property PACKAGE_PIN V17 [get_ports {sw[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sw[0]}]
```

1. `set_property PACKAGE_PIN V17 [get_ports {sw[0]}]`

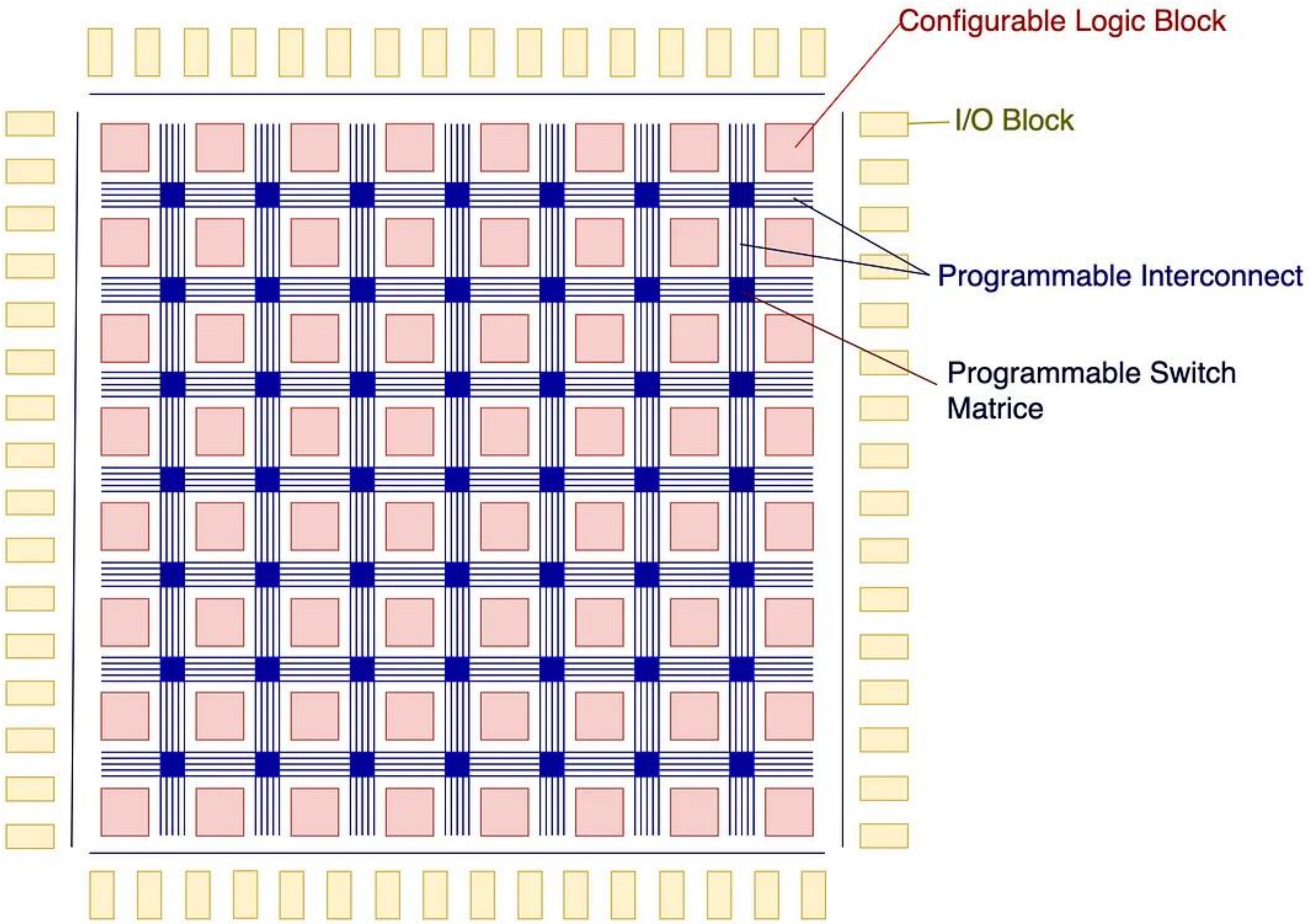
- `set_property`: This is a command in XDC files to assign specific properties to a pin or port.
- `PACKAGE_PIN V17`: This specifies the **pin** on the FPGA where the signal `sw[0]` (the 0th bit of the switch array) will be connected. `V17` is the physical pin number on the FPGA that is being used for this particular signal.
- `[get_ports {sw[0]}]`: This part refers to the **port** in your Verilog code. `sw[0]` is a port in your design, usually referring to the 0th bit of a switch (if you have an array like `sw[3:0]` for a 4-bit switch input). The `get_ports` command is used to identify that specific signal.

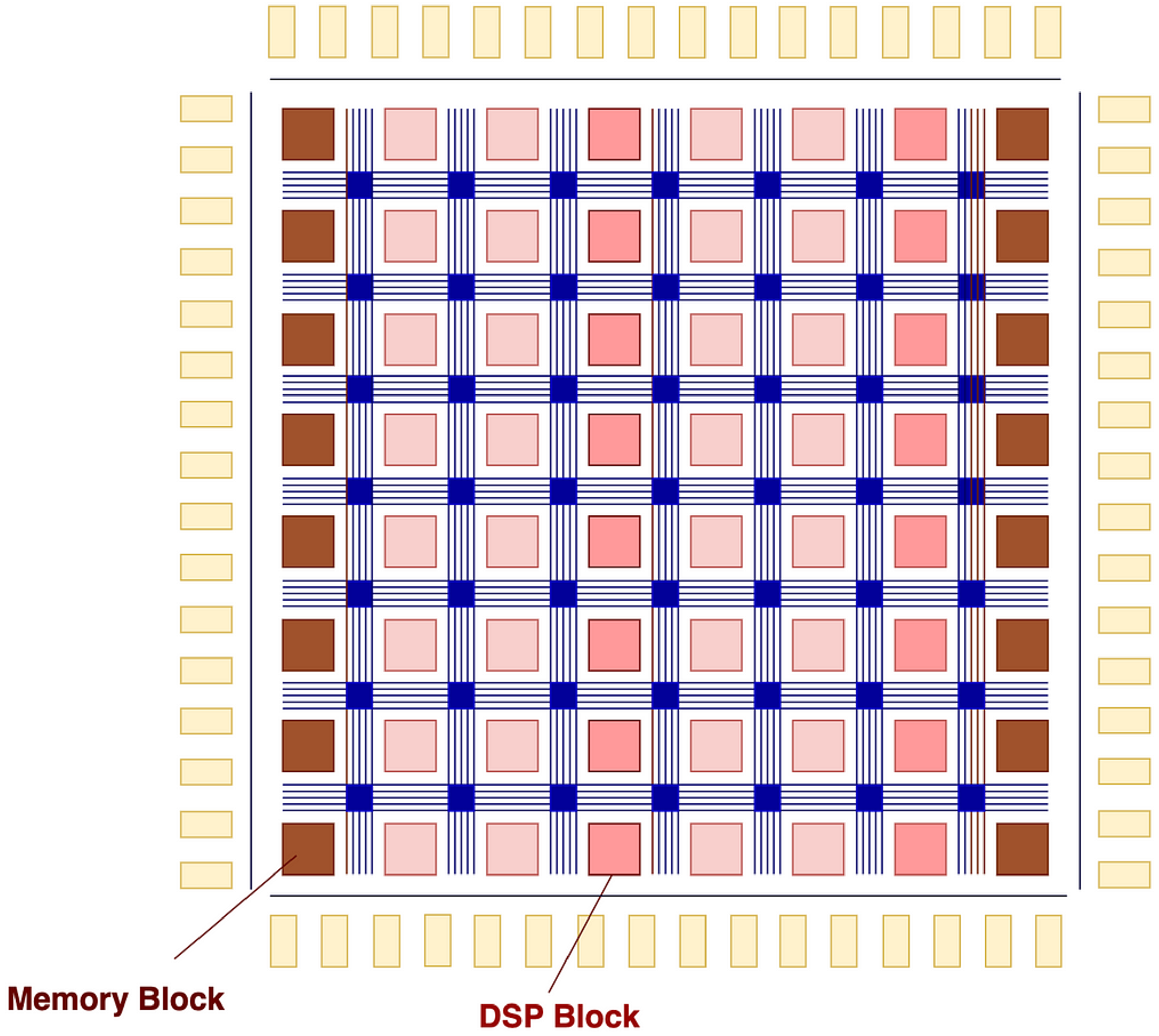
```
# Switches
set_property PACKAGE_PIN V17 [get_ports {sw[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {sw[0]}]
```

2. `set_property IOSTANDARD LVCMOS33 [get_ports {sw[0]}]`

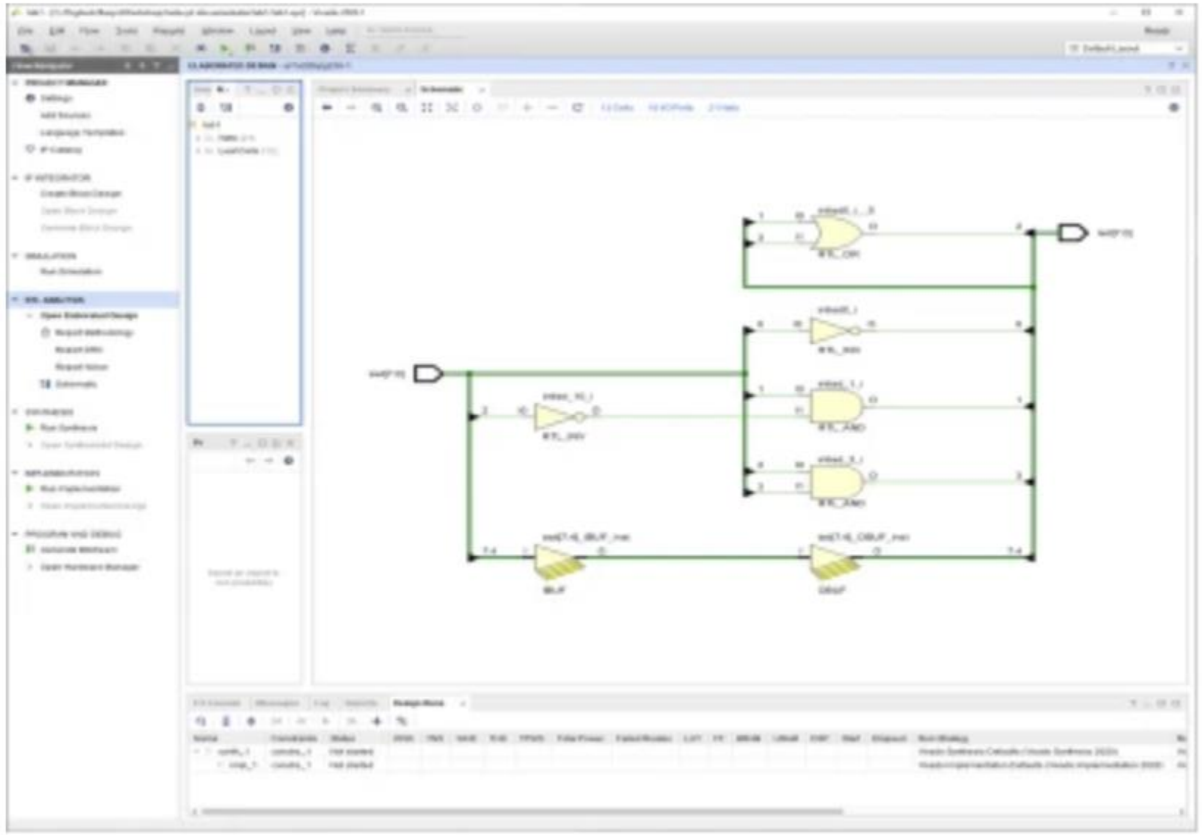
- `set_property`: This is again a command to assign a property to a pin or port.
- `IOSTANDARD LVCMOS33`: This specifies the **I/O standard** for the signal. In this case, the I/O standard is set to **LVCMOS33**, which is a standard for digital signals that operates at 3.3V logic levels (Low-Voltage CMOS).
  - **LVCMOS33** means that the logic high voltage level for this pin will be 3.3V, and the logic low level will be 0V.
- `[get_ports {sw[0]}]`: Refers to the specific signal (`sw[0]`), just like in the previous line. This part indicates that the I/O standard (LVCMOS33) is being applied to the `sw[0]` signal.

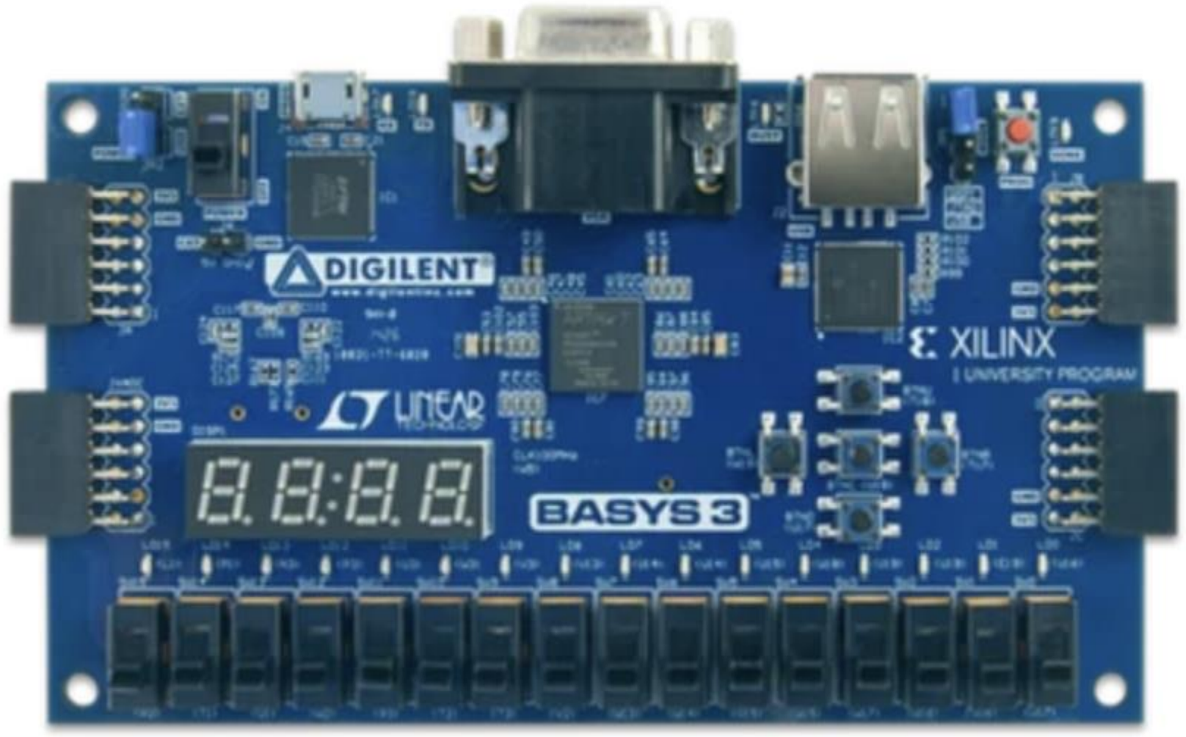
In most FPGA designs, LVCMOS33 is selected as the I/O standard because it ensures compatibility with external 3.3V components, is widely supported by modern FPGAs, and provides a good balance between power consumption, speed, and voltage tolerance.

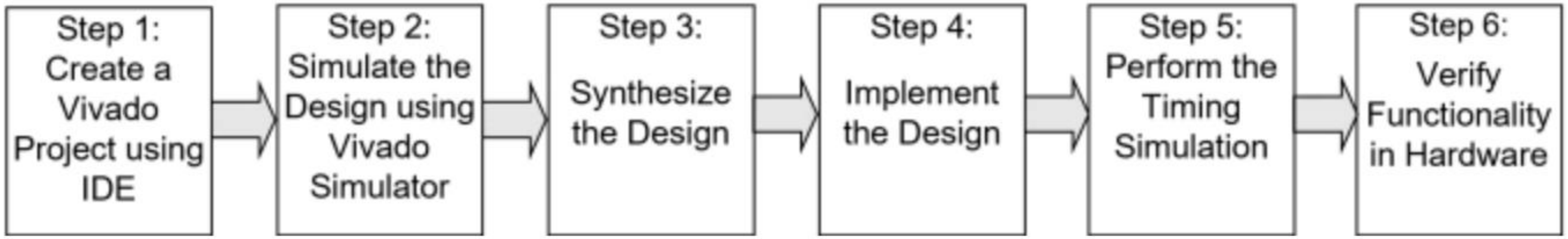

Configurable Logic Block
I/O Block
Programmable Interconnect
Programmable Switch Matrice


Memory Block
DSP Block

Process of Digital System Design Using Xilinx Vivado IDE and implement that in an FPGA board.






Step 1: Create a Vivado Project using IDE
Step 2: Simulate the Design using Vivado Simulator
Step 3: Synthesize the Design
Step 4: Implement the Design
Step 5: Perform the Timing Simulation
Step 6: Verify Functionality in Hardware

General Flow

Step 1: Create a Vivado Project using IDE
Step 2: Simulate the Design using Vivado Simulator
Step 3: Synthesize the Design
Step 4: Implement the Design
Step 5: Perform the Timing Simulation
Step 6: Verify Functionality in Hardware

Create a Vivado Project using IDE (Step 1)

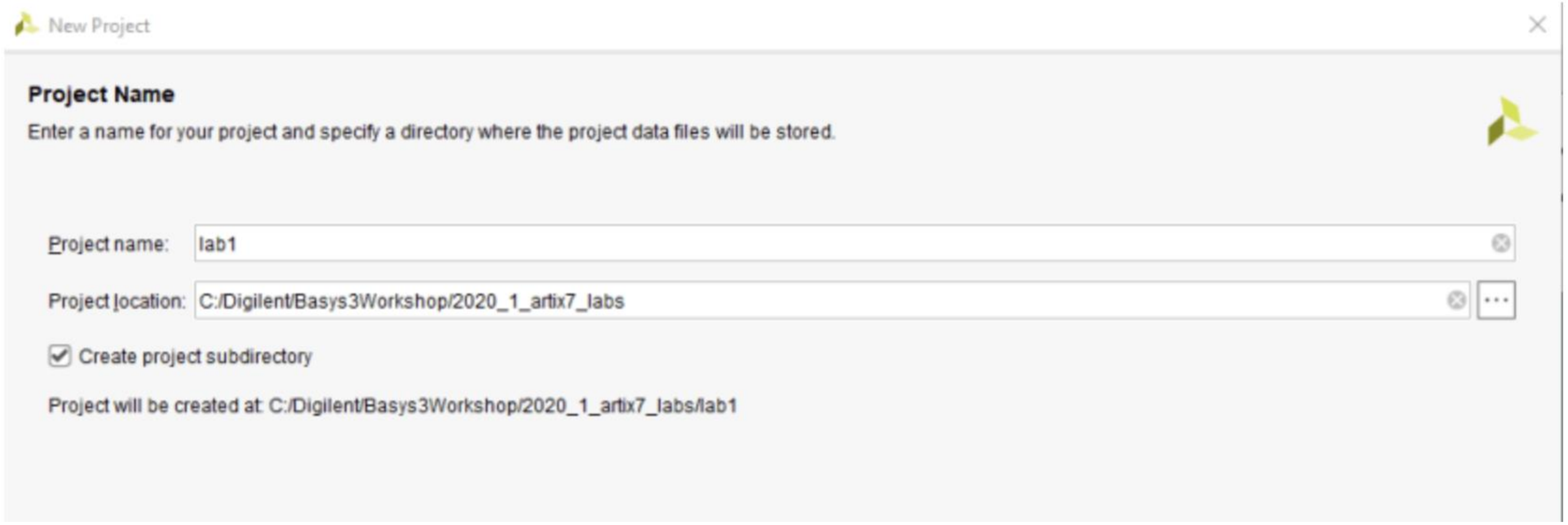
Launch Vivado and create a project targeting the XC7A35TCPG236-1 (Basys3) and using the VHDL HDL. Use the provided lab1.vhd and lab1.xdc files from the *2020_1_artix7_sources\lab1*

Open Vivado by selecting **Start > All Programs > Xilinx Design Tools > Vivado 2020.1> Vivado 2020.1**

Click **Create New Project** to start the wizard. You will see *Create A New Vivado Project* dialog box. Click **Next**.

Click the Browse button of the *Project location* field of the **New Project** form, browse to **C:/Digilent/Basys3Workshop/2020_1_artix7_labs**, and click **Select**.

Enter **lab1** in the *Project name* Make sure that the *Create Project Subdirectory* box is checked. Click **Next**.


New Project

Project Name

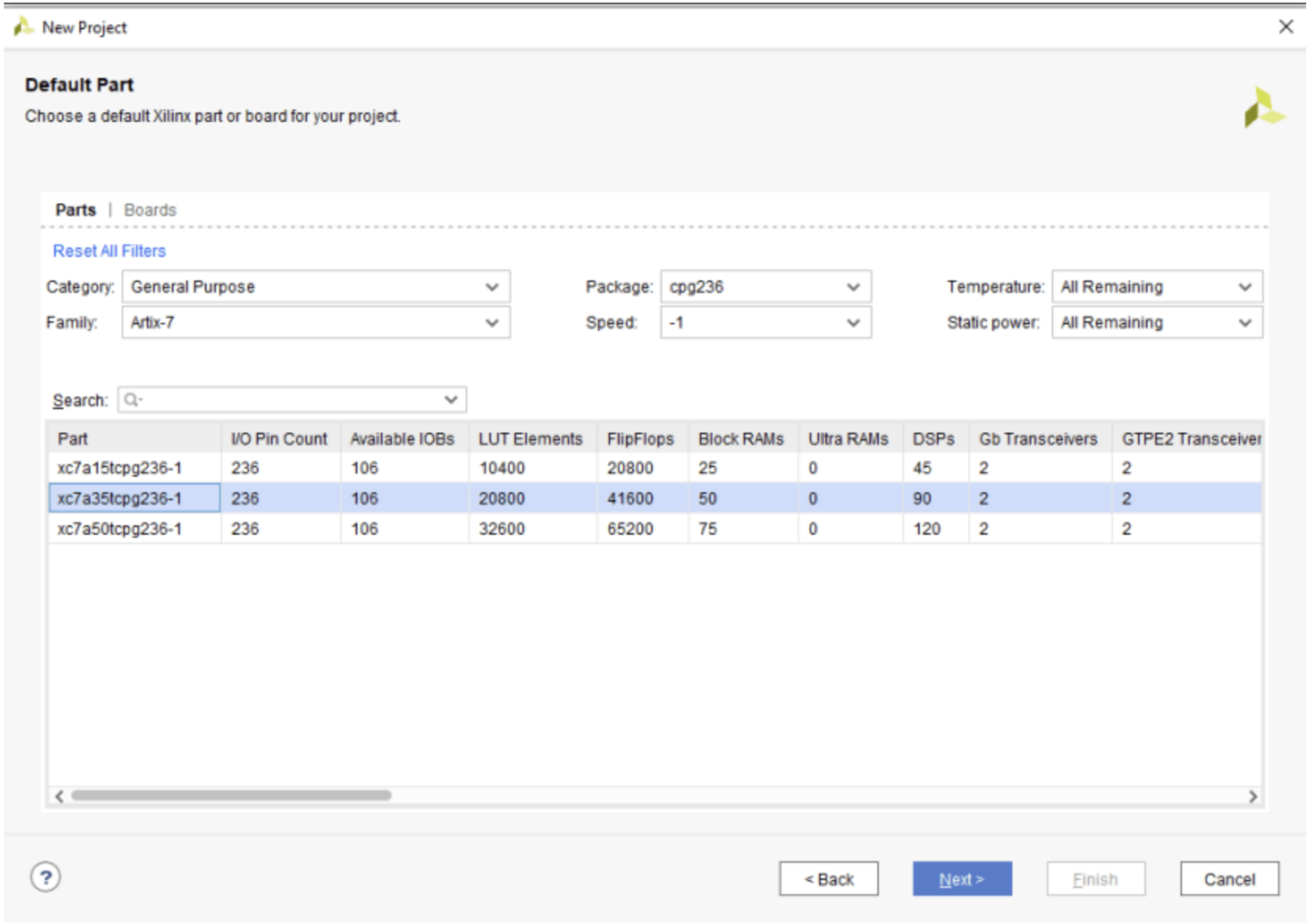
Enter a name for your project and specify a directory where the project data files will be stored.

Project name: lab1

Project location: C:/Digilent/Basys3Workshop/2020_1_artix7_labs

☑ Create project subdirectory

Project will be created at: C:/Digilent/Basys3Workshop/2020_1_artix7_labs/lab1



New Project

Default Part

Choose a default Xilinx part or board for your project.

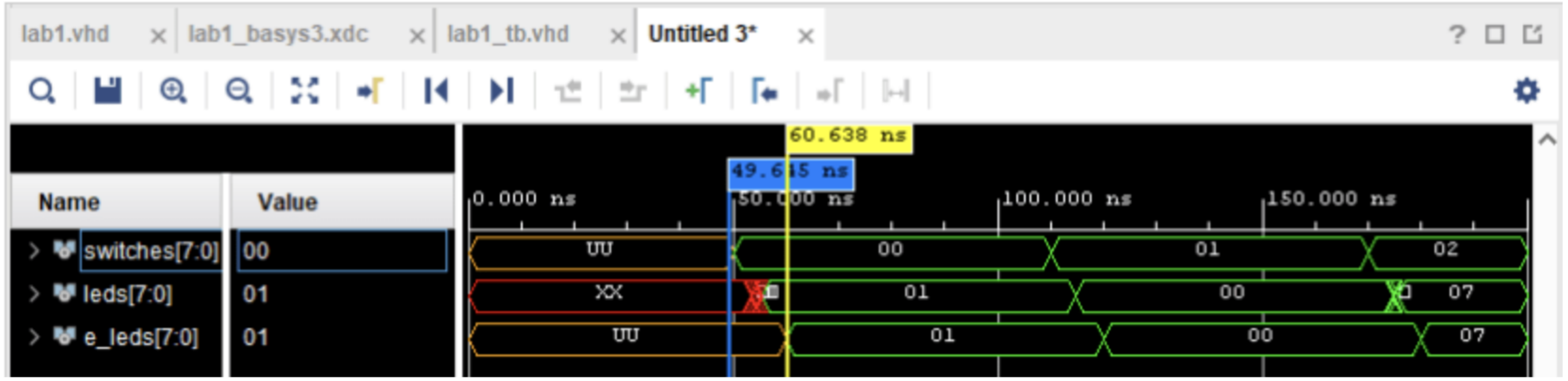
Parts | Boards

Reset All Filters

Category: General Purpose
Family: Artix-7
Package: cpg236
Speed: -1
Temperature: All Remaining
Static power: All Remaining

Search:

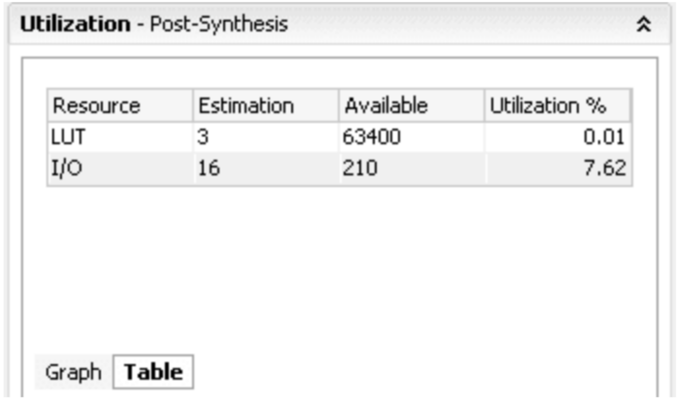
Part	I/O Pin Count	Available IOBs	LUT Elements	FlipFlops	Block RAMs	Ultra RAMs	DSPs	Gb Transceivers	GTPE2 Transceiver
xc7a15tcpg236-1	236	106	10400	20800	25	0	45	2	2
xc7a35tcpg236-1	236	106	20800	41600	50	0	90	2	2
xc7a50tcpg236-1	236	106	32600	65200	75	0	120	2	2



Simulation (step 2

Click on the **Table** tab in the **Project Summary** tab.

Notice that there are an estimated three LUTs and 16 IOs (8 input and 8 output) that are used.


Utilization - Post-Synthesis

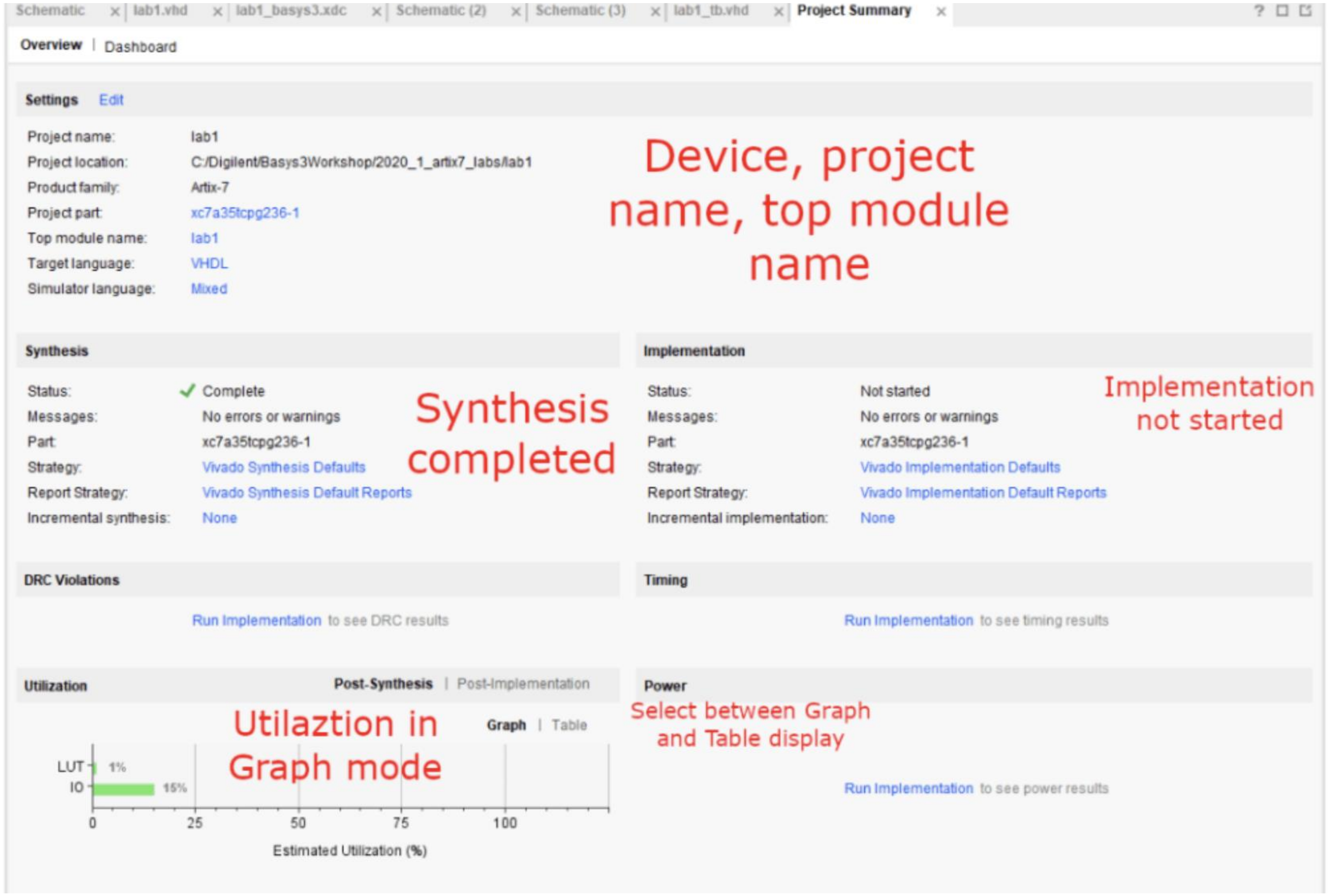
Resource	Estimation	Available	Utilization %
LUT	3	63400	0.01
I/O	16	210	7.62

Figure 22. Resource utilization estimation summary for the Basys3

Synthesize the Design (Step 3)

Synthesize the design with the Vivado synthesis tool and analyze the Project Summary output.

Click on **Run Synthesis** under the *Synthesis* tasks of the *Flow Navigator* pane.



Settings Edit

Project name:	lab1
Project location:	C:/Digilent/Basys3Workshop/2020_1_artix7_labs/lab1
Product family:	Artix-7
Project part:	xc7a35tcpg236-1
Top module name:	lab1
Target language:	VHDL
Simulator language:	Mixed

Synthesis

Status:	✓ Complete
Messages:	No errors or warnings
Part:	xc7a35tcpg236-1
Strategy:	Vivado Synthesis Defaults
Report Strategy:	Vivado Synthesis Default Reports
Incremental synthesis:	None

Implementation

Status:	Not started
Messages:	No errors or warnings
Part:	xc7a35tcpg236-1
Strategy:	Vivado Implementation Defaults
Report Strategy:	Vivado Implementation Default Reports
Incremental implementation:	None

DRC Violations

Run Implementation to see DRC results

Timing

Run Implementation to see timing results

Utilization

Power

Run Implementation to see power results

Implement the Design (Step 4)

Implement the design with the Vivado Implementation Defaults (Vivado Implementation 2020) settings and analyze the Project Summary output.

Click on **Run Implementation** under the *Implementation* tasks of the *Flow Navigator* pane.

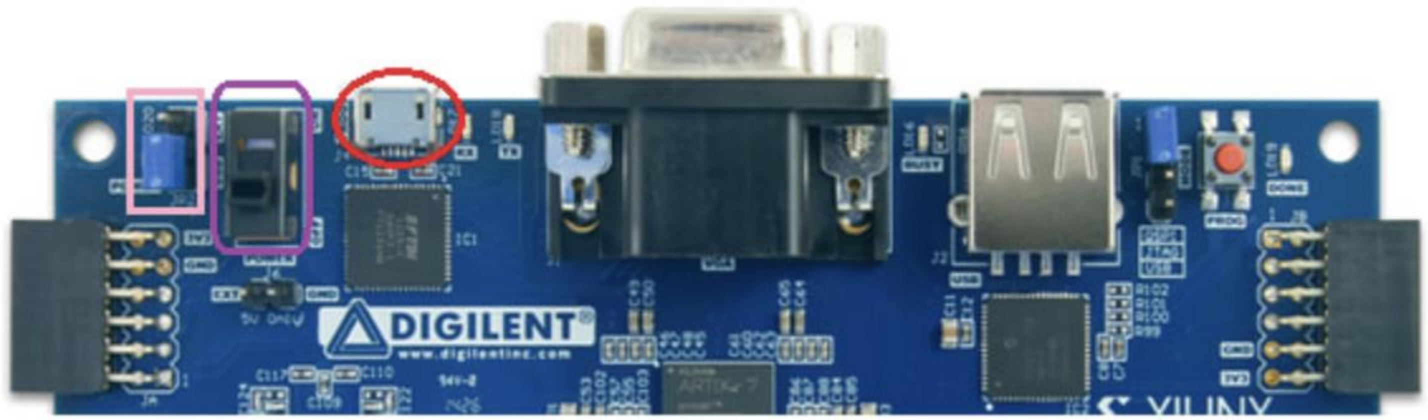
The implementation process will be run on the synthesized design. When the process is completed an *Implementation Completed* dialog box with three options will be displayed.

Generate the Bitstream and Verify Functionality (Step 6)

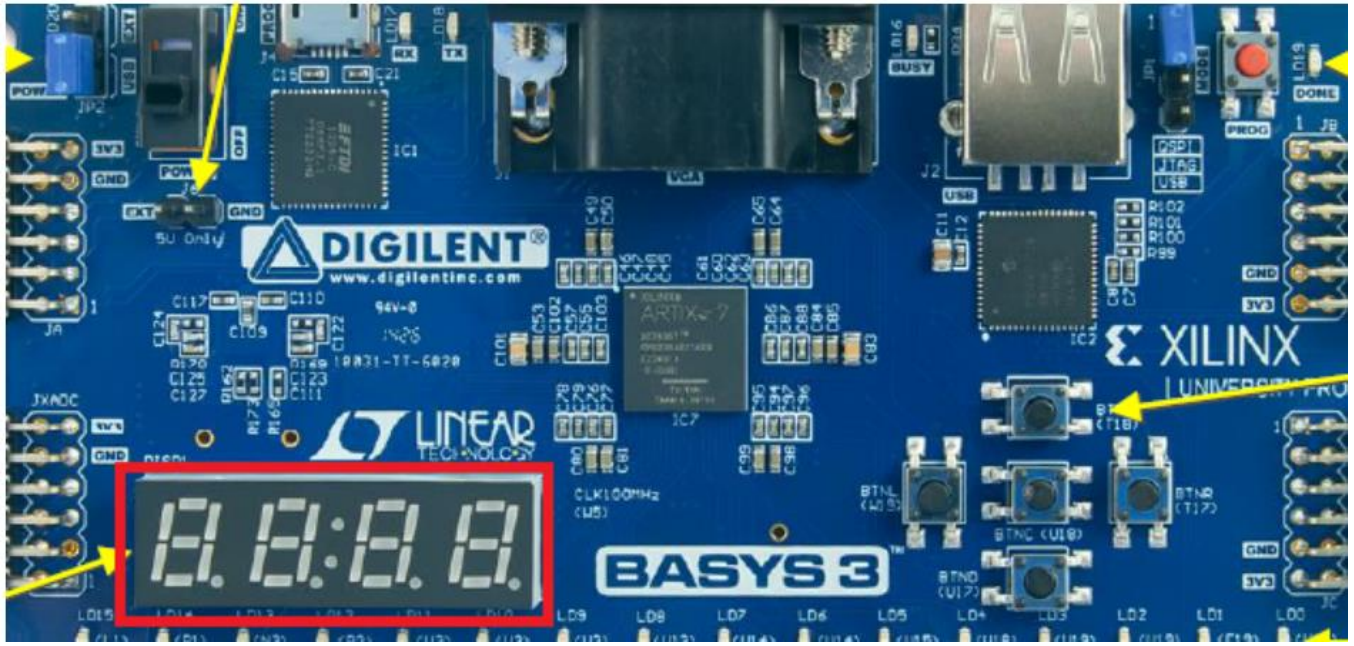
Connect the board and power it ON. Generate the bitstream, open a hardware session, and program the FPGA.

Make sure that the Micro-USB cable is connected to the JTAG PROG connector (next to the power supply connector).

Make sure that the board is set to use USB power (via the Power Select jumper JP2 on the Basys3)

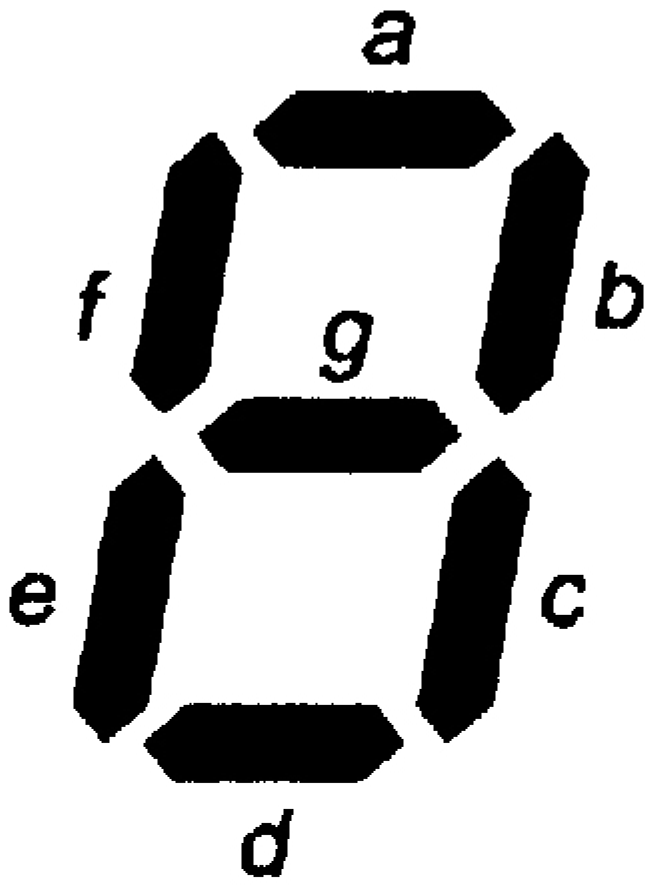


Introduction to the Basys3 Seven-Segment Display



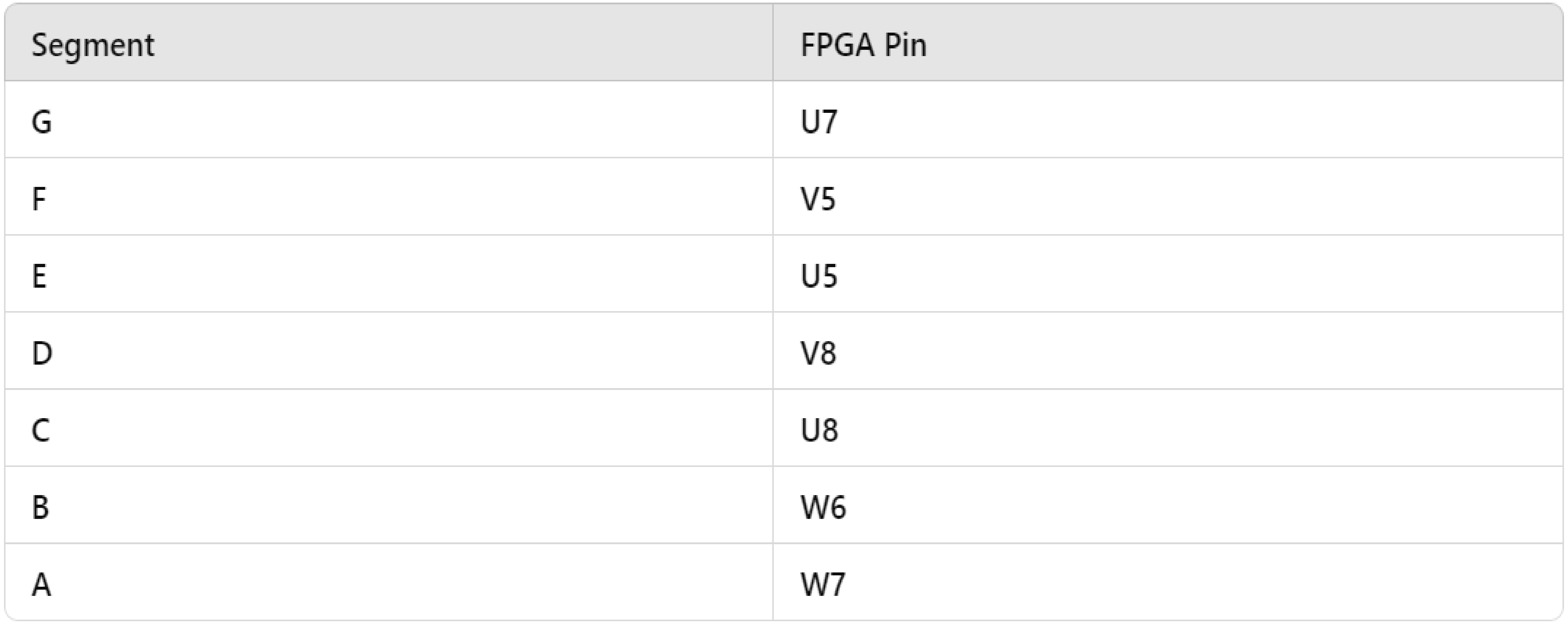
common-anode 4-digit 7-segment LED display

- The **Basys3 board** has **four 7-segment displays.**
- Each **digit** is controlled using **anode signals.**
- Each **segment** (a–g, dp) is **active-low** (0 = ON, 1 = OFF).



Segments (✓ = ON)							Display	Segments (✓ = ON)							Display
a	b	c	d	e	f	g		a	b	c	d	e	f	g	
✓	✓	✓	✓	✓	✓		0	✓	✓	✓	✓	✓	✓	✓	8
	✓	✓					1	✓	✓	✓			✓	✓	9
✓	✓		✓	✓		✓	2	✓	✓	✓		✓	✓	✓	A
✓	✓	✓	✓			✓	3			✓	✓	✓	✓	✓	b
	✓	✓			✓	✓	4	✓			✓	✓	✓		C
✓		✓	✓		✓	✓	5		✓	✓	✓	✓		✓	d
✓		✓	✓	✓	✓	✓	6	✓			✓	✓	✓	✓	E
✓	✓	✓					7	✓				✓	✓	✓	F

To correctly map the seven-segment display pins on the Basys3 board, you must assign each segment (a-g) to the correct FPGA pin in the .xdc constraints file. Below is the correct pin mapping based on the Basys3 reference manual:

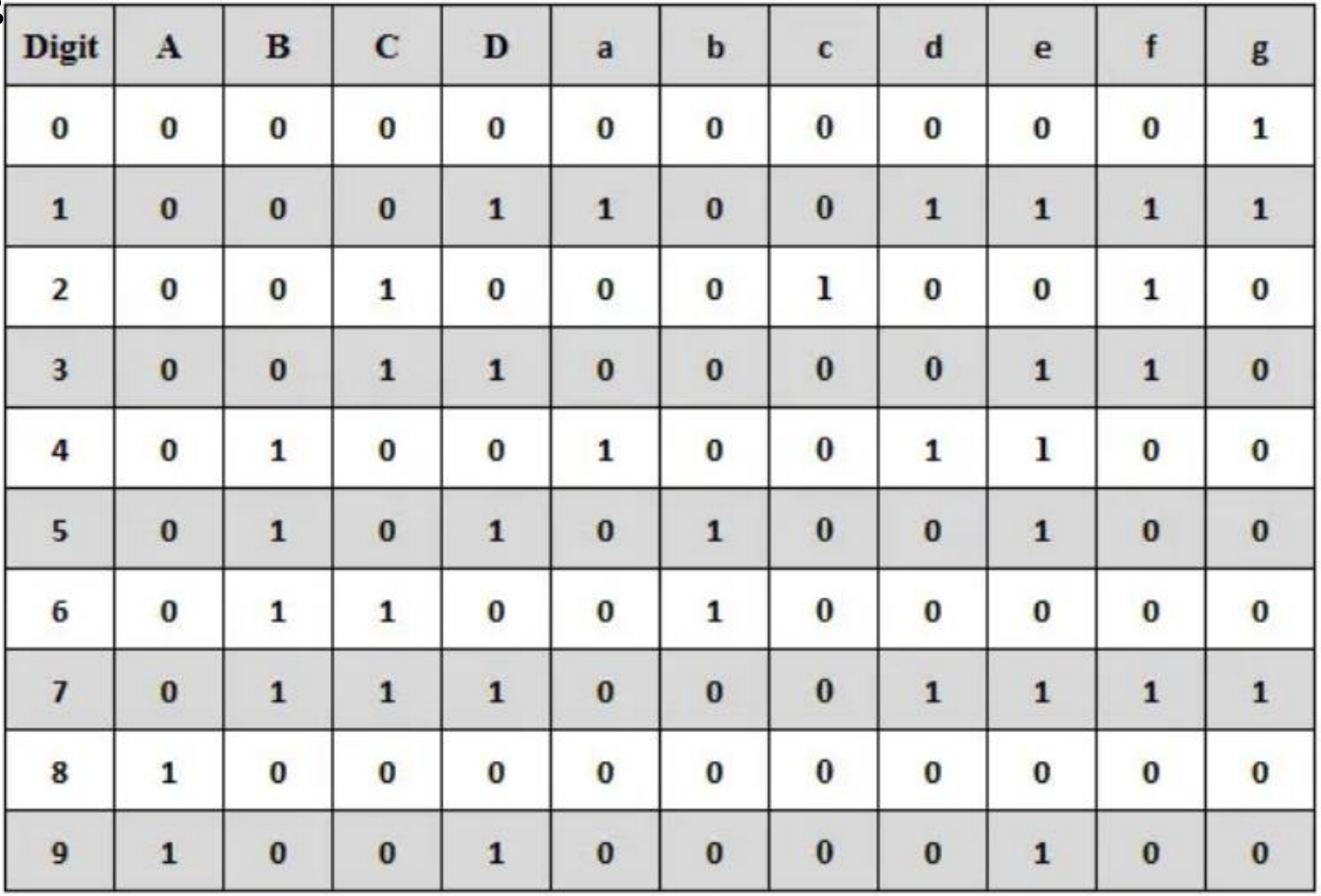


Segment	FPGA Pin
G	U7
F	V5
E	U5
D	V8
C	U8
B	W6
A	W7

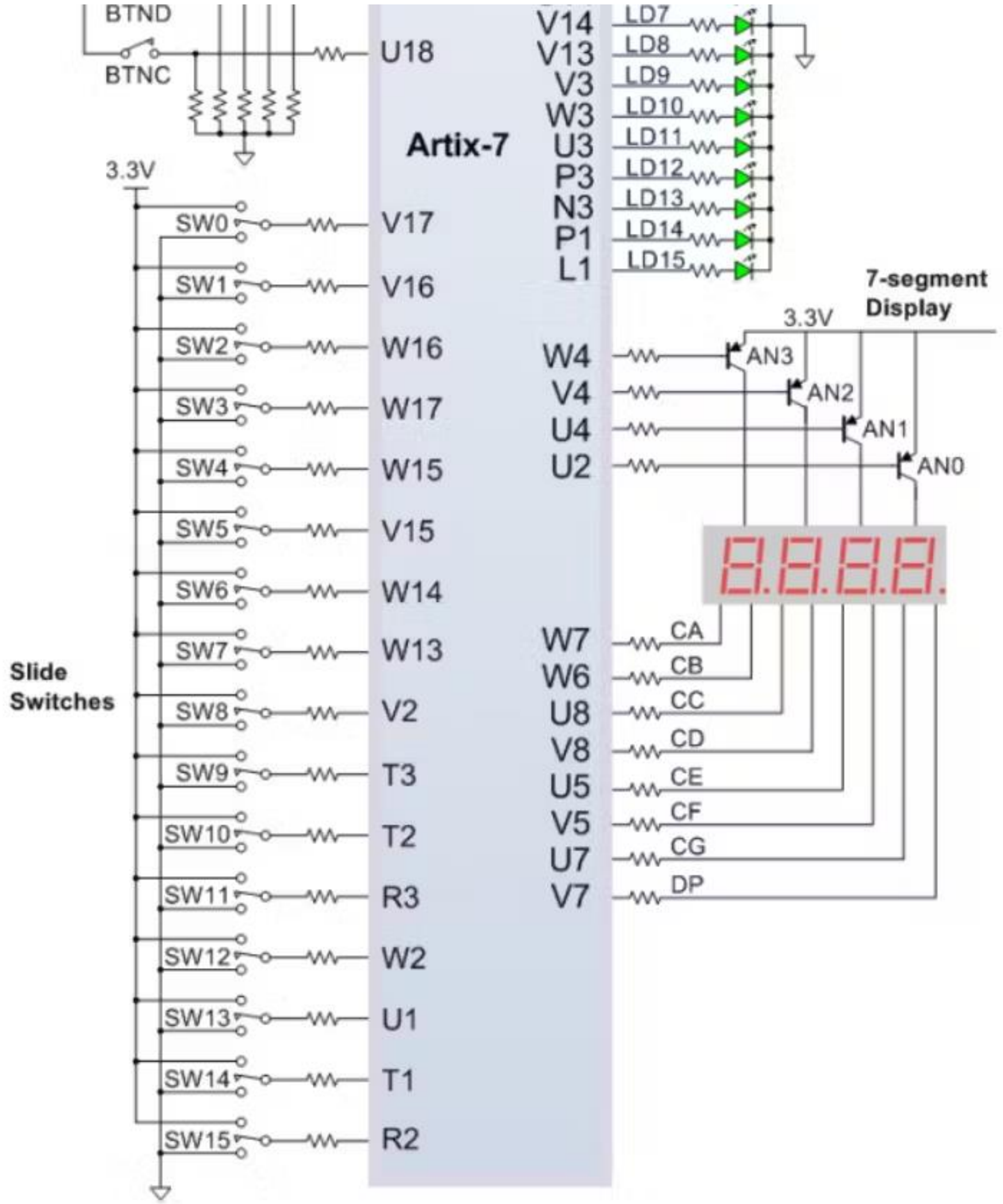
Anode Control (an[3:0])

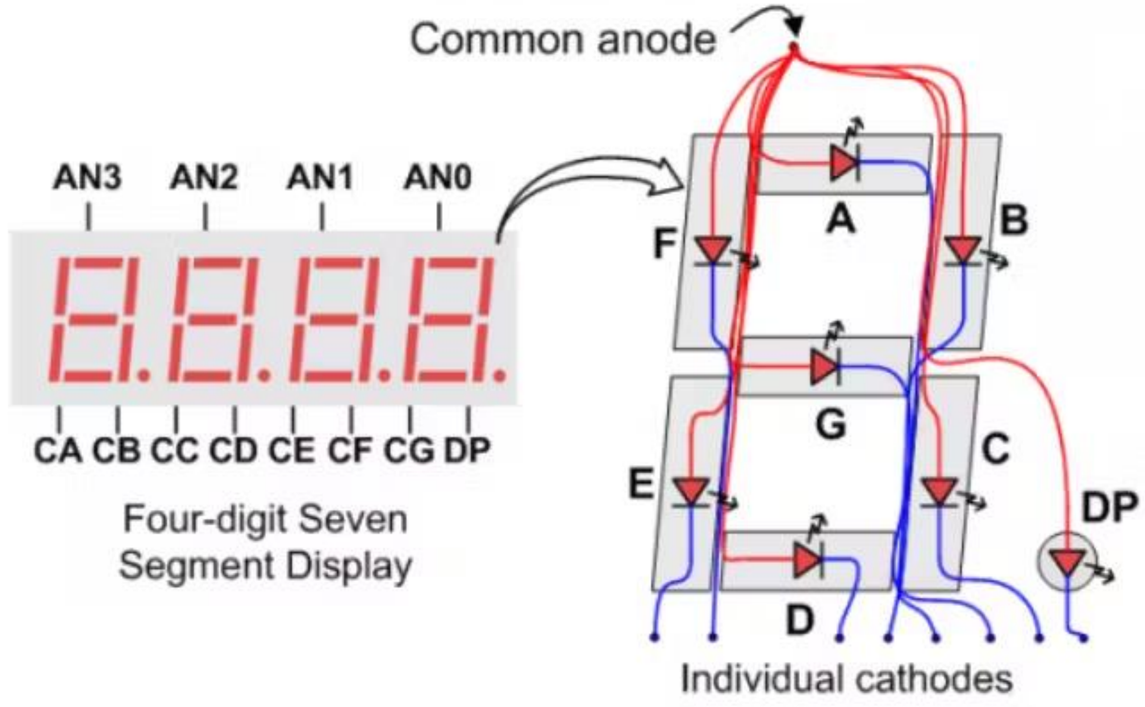
Anode	FPGA Pin
AN3	W4
AN2	V4
AN1	U4
AN0	U2

Decoder logic for converting a binary code into a 7-segment code conversion for displaying



Digit	A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	1	0	0	1	1	1	1
2	0	0	1	0	0	0	1	0	0	1	0
3	0	0	1	1	0	0	0	0	1	1	0
4	0	1	0	0	1	0	0	1	1	0	0
5	0	1	0	1	0	1	0	0	1	0	0
6	0	1	1	0	0	1	0	0	0	0	0
7	0	1	1	1	0	0	0	1	1	1	1
8	1	0	0	0	0	0	0	0	0	0	0
9	1	0	0	1	0	0	0	0	1	0	0


BTND
BTNC
U18
Artix-7
3.3V
Slide Switches
SW0 V17
SW1 V16
SW2 W16
SW3 W17
SW4 W15
SW5 V15
SW6 W14
SW7 W13
SW8 V2
SW9 T3
SW10 T2
SW11 R3
SW12 W2
SW13 U1
SW14 T1
SW15 R2
V14 LD7
V13 LD8
V3 LD9
W3 LD10
U3 LD11
P3 LD12
N3 LD13
P1 LD14
L1 LD15
7-segment Display
3.3V
W4 AN3
V4 AN2
U4 AN1
U2 AN0
W7 CA
W6 CB
U8 CC
V8 CD
U5 CE
V5 CF
U7 CG
V7 DP


Common anode
AN3
AN2
AN1
AN0
CA CB CC CD CE CF CG DP
Four-digit Seven Segment Display
A
B
C
D
E
F
G
DP
Individual cathodes

Setting Up Vivado for Basys3

Steps to Cover:

1. Create a new Vivado Project
   - Select **Basys3 (xc7a35tcpg236-1)** as the FPGA.
   - Use **Verilog** as the design language.
2. **Add the Constraints File (`.xdc`)**
   - Download the **Basys3 Master XDC** from Digilent.
   - Modify it to enable **7-segment pins** and **anodes**.

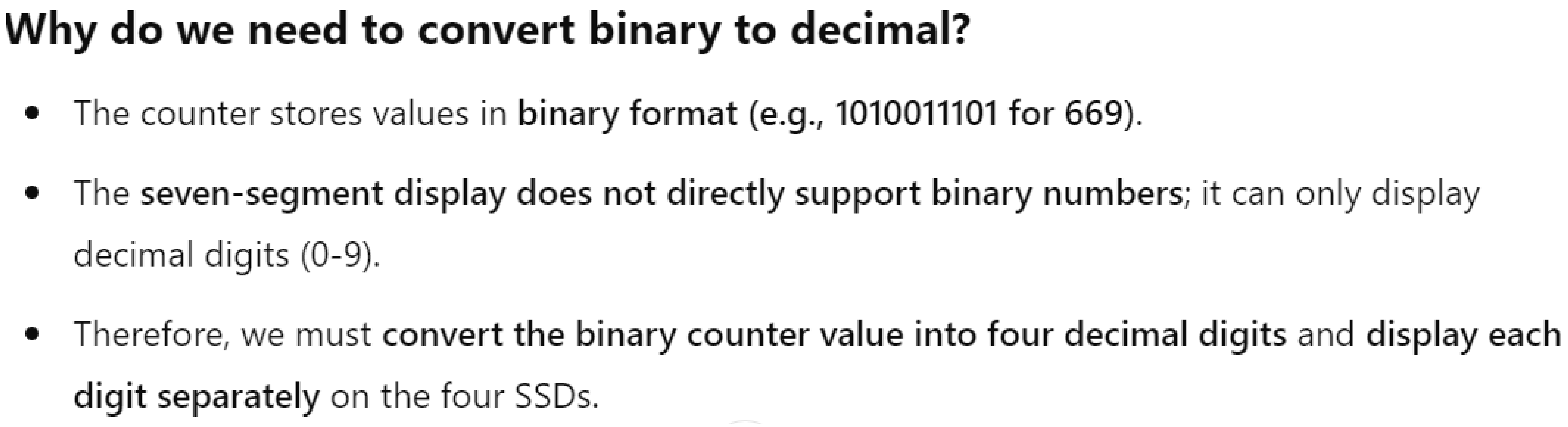
The Basys3 board has a 100 MHz clock (on pin W5) that can be used for counter design. However, since 100 MHz is very fast, it is common to use a clock divider to generate a slower clock signal. Here’s how you can use the Basys3 clock for a counter design in Verilog.

```
## Clock Signal
set_property PACKAGE_PIN W5 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]

## LED Outputs
set_property PACKAGE_PIN U16 [get_ports {led[0]}]
set_property PACKAGE_PIN E19 [get_ports {led[1]}]
set_property PACKAGE_PIN U19 [get_ports {led[2]}]
set_property PACKAGE_PIN V19 [get_ports {led[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[*]}]

## Reset Button
set_property PACKAGE_PIN W19 [get_ports rst]
set_property IOSTANDARD LVCMOS33 [get_ports rst]
```

Displaying a 10-bit Counter Output on Basys3 Seven-Segment Displa

Step 1: Design a 10-bit counter

A 10-bit counter can count from 0 to 1023 (2^10 - 1). Since Basys3 has four seven-segment displays, we need to display a 4-digit decimal number (0000–1023).



Why do we need to convert binary to decimal?

- The counter stores values in **binary format (e.g., 1010011101 for 669)**.
- The **seven-segment display does not directly support binary numbers**; it can only display decimal digits (0-9).
- Therefore, we must **convert the binary counter value into four decimal digits** and **display each digit separately** on the four SSDs.

Step 2: Binary to Decimal Conversion

Since the maximum counter value is **1023**, we extract the four decimal digits using **division and modulus** operations:

- Thousands digit: `(count / 1000) % 10`
- Hundreds digit: `(count / 100) % 10`
- Tens digit: `(count / 10) % 10`
- Ones digit: `(count % 10)`

Example Calculation (Binary to Decimal)

Example 1: Counter Value = 669 (Binary: 1010011101)

```makefile
Thousands  = (669 / 1000) % 10  = 0
Hundreds   = (669 / 100) % 10   = 6
Tens       = (669 / 10) % 10    = 6
Ones       = 669 % 10           = 9
```

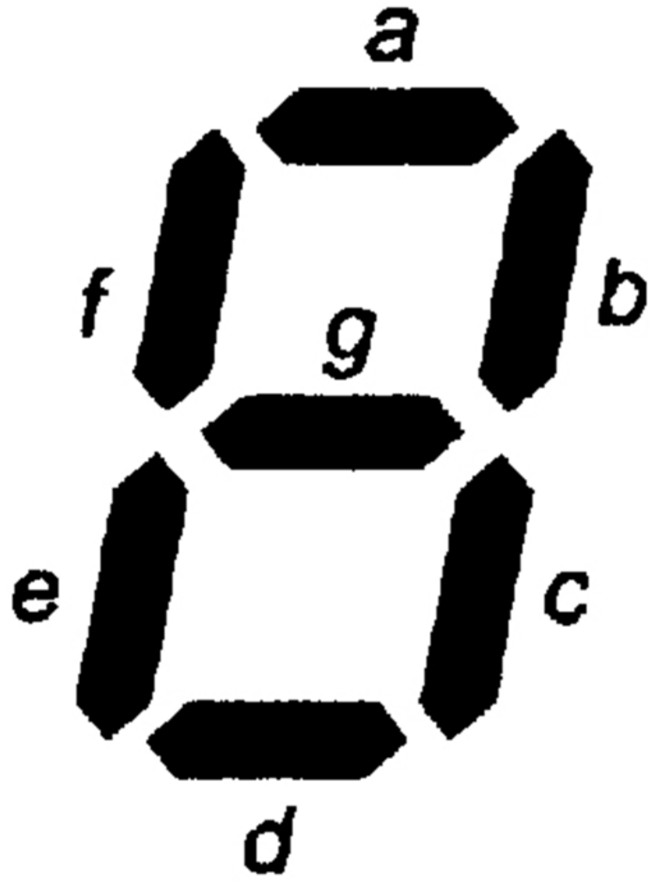
Final Digits: "0669“

Each decimal digit will be displayed on a separate seven-segment display.

Step 3: Seven-Segment Display Encoding

A **seven-segment display (SSD)** has **seven LEDs** arranged in a figure-eight pattern. Each digit (0-9) has a unique LED pattern.

Digit	A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	1	0	0	1	1	1	1
2	0	0	1	0	0	0	1	0	0	1	0
3	0	0	1	1	0	0	0	0	1	1	0
4	0	1	0	0	1	0	0	1	1	0	0
5	0	1	0	1	0	1	0	0	1	0	0
6	0	1	1	0	0	1	0	0	0	0	0
7	0	1	1	1	0	0	0	1	1	1	1
8	1	0	0	0	0	0	0	0	0	0	0
9	1	0	0	1	0	0	0	0	1	0	0



Step 4: Multiplexing the Seven-Segment Display

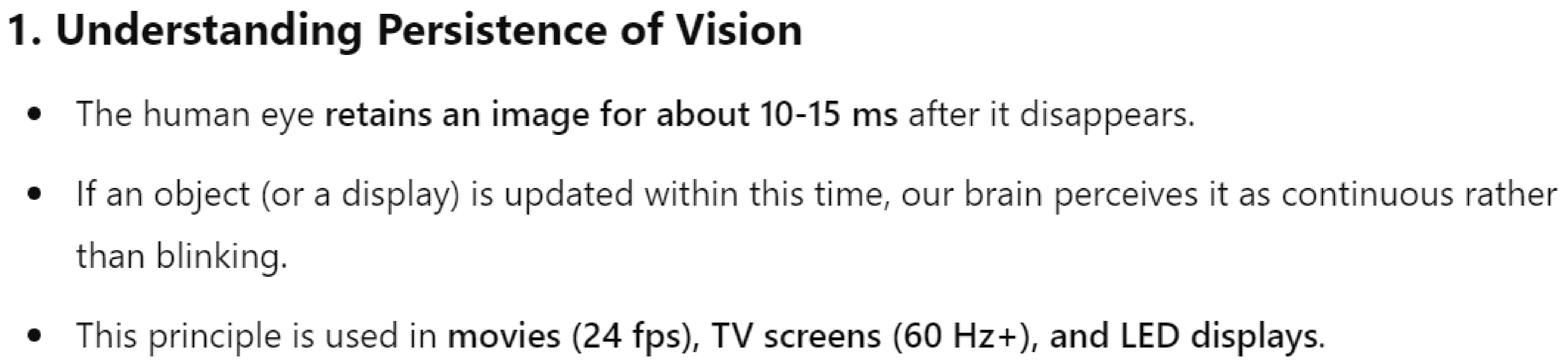
Basys3 has **four seven-segment displays**, but only **one display can be activated at a time.**

If we drive all SSDs simultaneously, they will **all display the same number**.

- We need multiplexing.....**but we need continuous display on each SSD without any flickering**
- **Digit time (time per digit during time multiplexing) must be set carefully so that our brain perceives the display as continuous rather than blinking.**

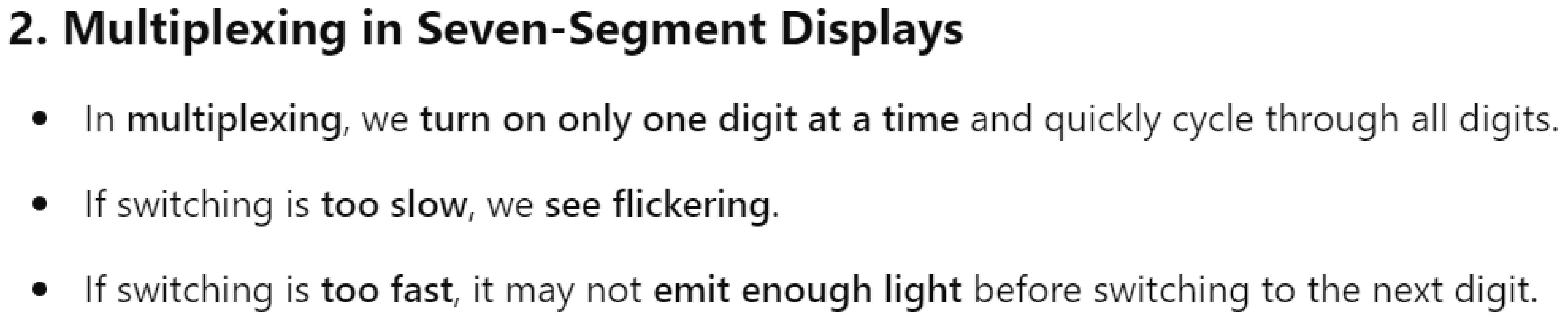
How to choose Time per Digit in Seven-Segment Multiplexing?

- We exploit … **Human persistence of vision (POV)** **and** **the refresh rate** required for stable display perception



1. Understanding Persistence of Vision

- The human eye **retains an image for about 10-15 ms** after it disappears.
- If an object (or a display) is updated within this time, our brain perceives it as continuous rather than blinking.
- This principle is used in **movies (24 fps), TV screens (60 Hz+), and LED displays.**



2. Multiplexing in Seven-Segment Displays

- In **multiplexing**, we **turn on only one digit at a time** and quickly cycle through all digits.
- If switching is **too slow**, we **see flickering**.
- If switching is **too fast**, it may not **emit enough light** before switching to the next digit.

To avoid the displaying discontinuity perceived by the human eye, the four seven-segment LEDs should be continuously refreshed at about 1KHz to 60Hz or it should be refreshed at every 1ms to 16ms.

Cycle Repetition in Seven-Segment Multiplexing

In a **multiplexed seven-segment display**, the cycle repeats **after all digits have been refreshed once**.

If we choose **1 ms per digit** and have **N digits**, the **full cycle repeats every**:

$$T_{\text{cycle}} = N \times T_{\text{digit}}$$

For example:

- **4-digit display** (Basys3 has 4 seven-segment digits)
- **1 ms per digit**

$$T_{\text{cycle}} = 4 \times 1 \text{ ms} = 4 \text{ ms}$$

Thus, **every 4 ms**, the display completes one full refresh cycle. This means the **entire display refreshes 250 times per second (250 Hz)**.

- Each digit is ON for **1 ms** before switching.
- The full display is **refreshed 250 times per second (250 Hz)**, well above the **60 Hz flicker threshold**.

Multiplexing Strategy

1. A **refresh counter** cycles through the four SSDs.
2. At each step, it **activates one display** and **sends the corresponding digit's 7-segment pattern**.
3. The switching happens **so fast (1 kHz refresh rate)** that the human eye perceives all four digits at once.

Digit Refreshing Timing

- **Digit 1 (Ones place)** → ON for **1 ms**
- **Digit 2 (Tens place)** → ON for **1 ms**
- **Digit 3 (Hundreds place)** → ON for **1 ms**
- **Digit 4 (Thousands place)** → ON for **1 ms**

Why Not Faster or Slower?

- **Slower than 1 ms (e.g., 5 ms per digit)** → **Flickering** may be visible.
- **Faster than 1 ms (e.g., 100 µs per digit)** → Each digit gets very little time to emit light, causing the display to **appear dim**.

Thus, **1 ms per digit (1 kHz refresh rate)** is a balanced choice:

✓ No flickering
✓ Sufficient brightness
✓ Easy to implement in FPGA

Step 5: Implementing Verilog Modules

We divide the design into **four modules**:

1. Counter Module (10-bit Counter)

This module increments the counter every second.

2. Binary-to-Decimal Converter

This module converts the **10-bit counter** into **four decimal digits**.

3. Seven-Segment Decoder

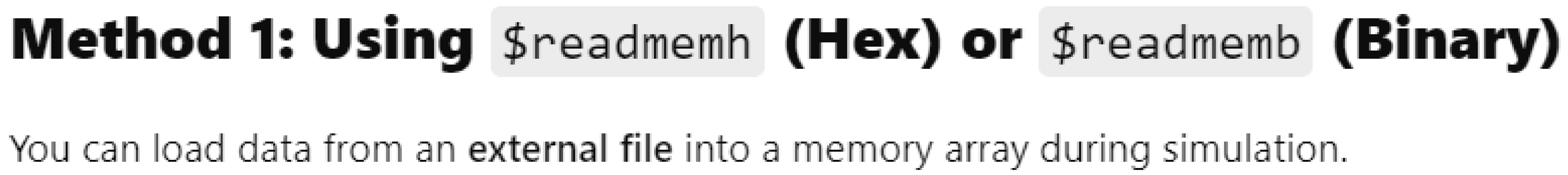
This module converts **each decimal digit** (0-9) **into a 7-bit SSD pattern**.

4. Refresh Counter (Multiplexing Controller)

This module activates **one display at a time** and switches between them at **1 kHz**.

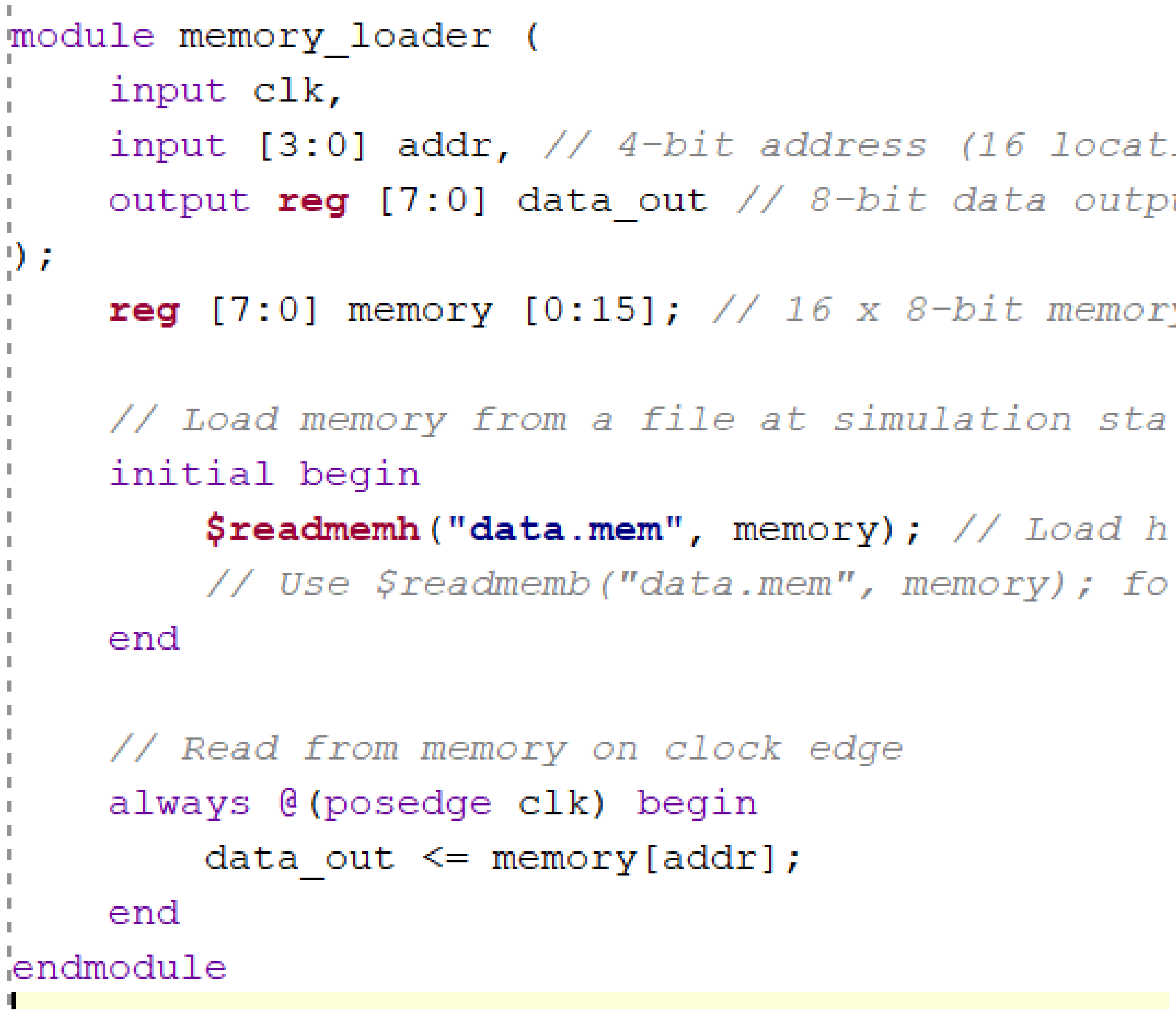
Loading Data into Memory in Verilog

- There are multiple ways to load data into memory in Verilog, depending on whether you want to initialize it at startup or load it dynamically.

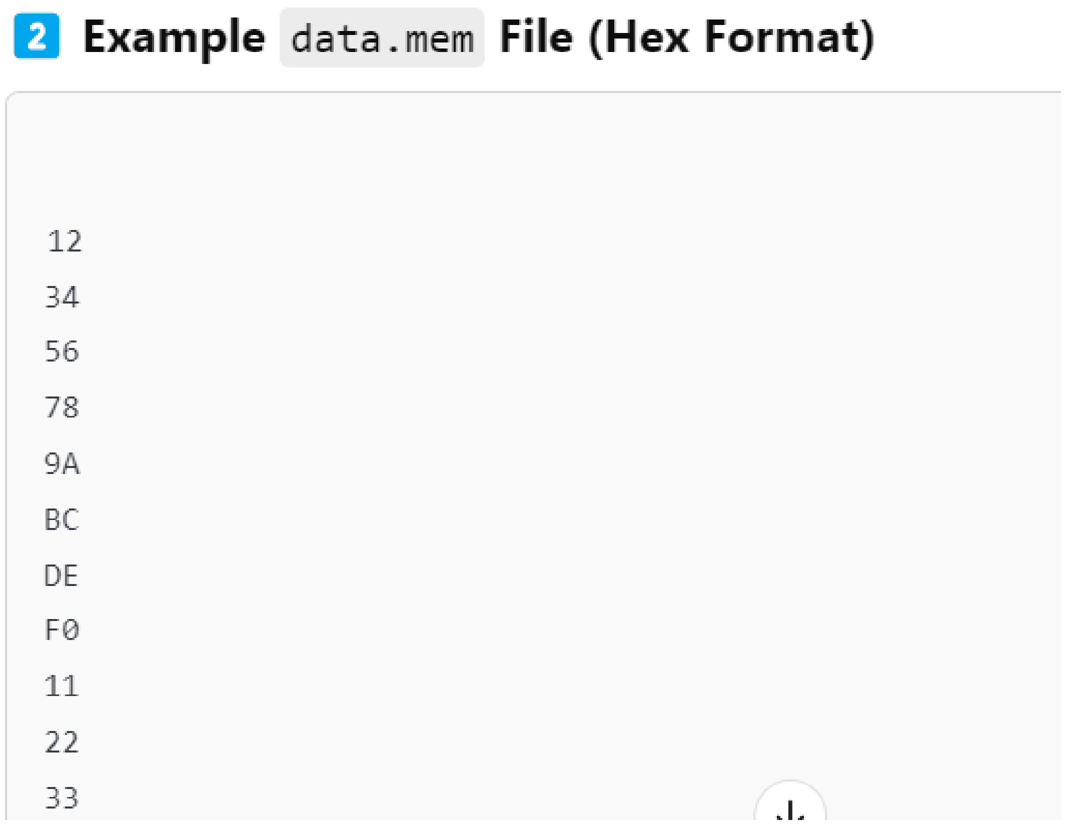


Method 1: Using `$readmemh` (Hex) or `$readmemb` (Binary)

You can load data from an **external file** into a memory array during simulation.



```
module memory_loader (
    input clk,
    input [3:0] addr, // 4-bit address (16 locations)
    output reg [7:0] data_out // 8-bit data output
);
    reg [7:0] memory [0:15]; // 16 x 8-bit memory

    // Load memory from a file at simulation sta
    initial begin
        $readmemh("data.mem", memory); // Load h
        // Use $readmemb("data.mem", memory); fo
    end

    // Read from memory on clock edge
    always @(posedge clk) begin
        data_out <= memory[addr];
    end
endmodule
```



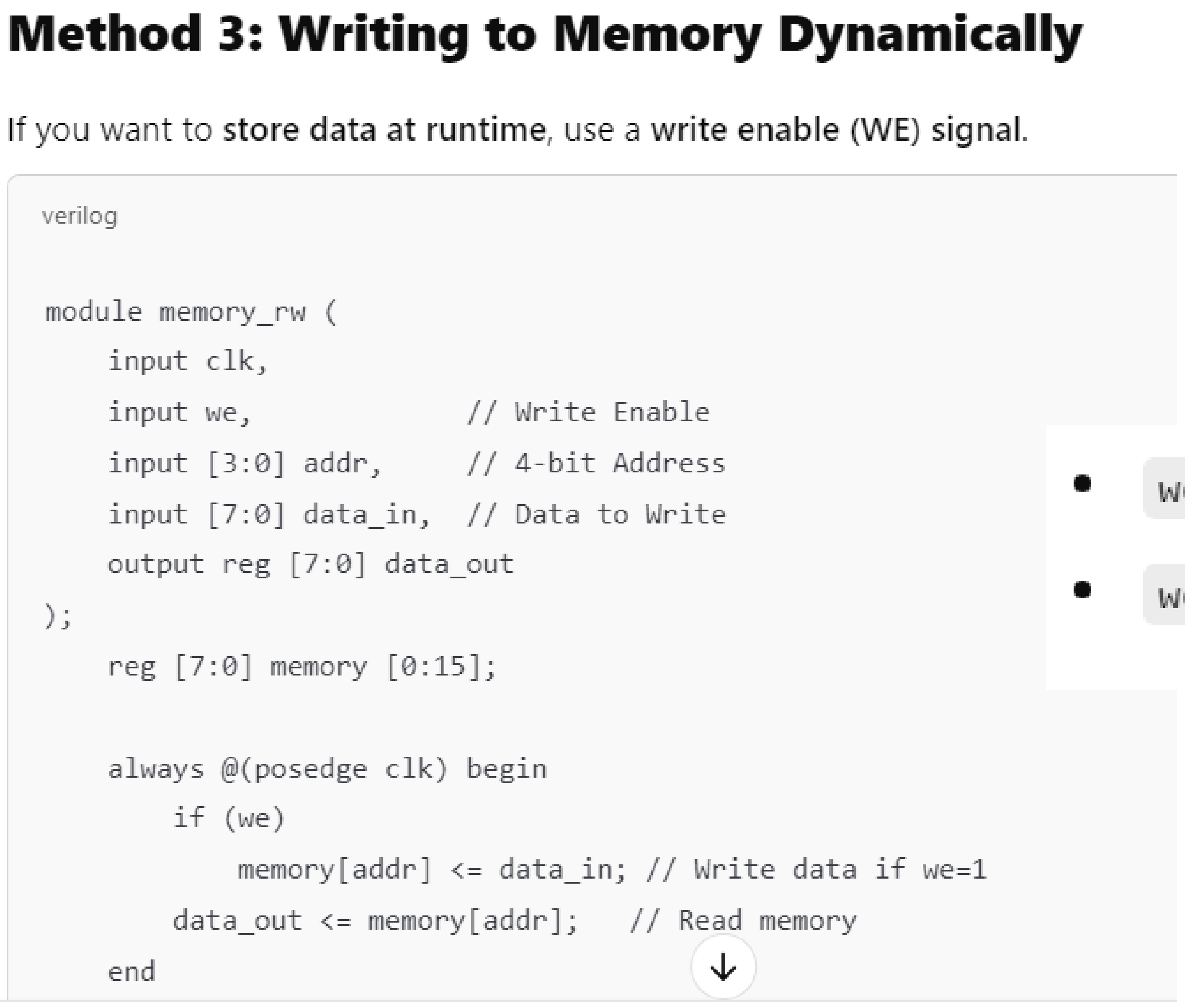
2 Example `data.mem` File (Hex Format)

```
12
34
56
78
9A
BC
DE
F0
11
22
33
```

Method 2: Initializing Memory in Code

If you don't want to use an external file, you can **initialize memory directly** in Verilog.

```
// Initialize memory values directly
initial begin
    memory_array[0] = 4'b0001; // 1
    memory_array[1] = 4'b0010; // 2
    memory_array[2] = 4'b0011; // 3
    memory_array[3] = 4'b0100; // 4
    memory_array[4] = 4'b0101; // 5
    memory_array[5] = 4'b0110; // 6
    memory_array[6] = 4'b0111; // 7
    memory_array[7] = 4'b1000; // 8
    memory_array[8] = 4'b1001; // 9
    memory_array[9] = 4'b1010; // 10
end

always @(posedge clk) begin
    if (rst) begin
        data_out <= 4'b0000;
    end else if (wr_en) begin
        memory_array[addr] <= data_in; // Write operation
    end else begin
        data_out <= memory_array[addr]; // Read operation
    end
```

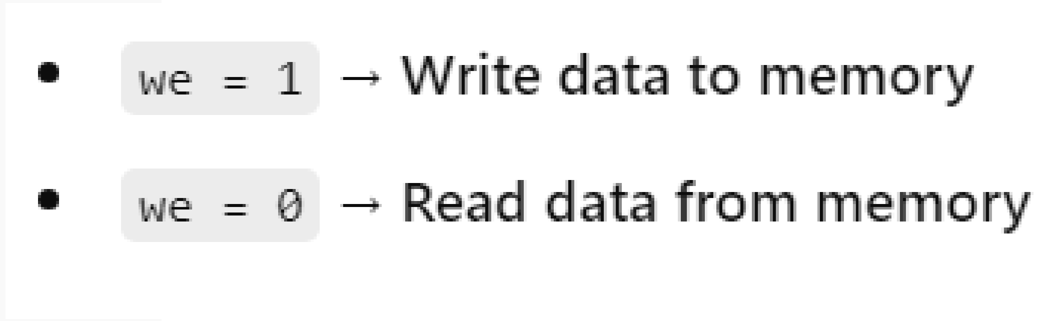


Method 3: Writing to Memory Dynamically

If you want to **store data at runtime**, use a **write enable (WE) signal**.

verilog

```
module memory_rw (
    input clk,
    input we,             // Write Enable
    input [3:0] addr,     // 4-bit Address
    input [7:0] data_in,  // Data to Write
    output reg [7:0] data_out
);
    reg [7:0] memory [0:15];

    always @(posedge clk) begin
        if (we)
            memory[addr] <= data_in; // Write data if we=1
        data_out <= memory[addr];   // Read memory
    end
```



- `we = 1` → Write data to memory
- `we = 0` → Read data from memory