ICSC
Centro Nazionale di Ricerca in HPC,
Big Data and Quantum Computing

Verilog
Open Source and FPGA
Zynq, Pynq and Alveo

Mirko Mariotti

June 25, 2025

Contents

Modules

- Verilog modules are blocks of code (circuit) that can be reused within other blocks (similar to functions in computer programming languages)
- The keyword “module” starts a block that ends with “endmodule”.
- They could have input and output parameters.
-

```
module MyModule ([parameters]);
    inputs ...
    outputs ...
    internal variables ...
    ...
    Module Code ...
endmodule
```

Modules

- Verilog modules are blocks of code (circuit) that can be reused within other blocks (similar to functions in computer programming languages)
- The keyword “module” starts a block that ends with “endmodule”.
- They could have input and output parameters.

```
module MyModule ([parameters]);
    inputs ...
    outputs ...
    internal variables ...
    ...
    Module Code ...
endmodule
```

Modules

- Verilog modules are blocks of code (circuit) that can be reused within other blocks (similar to functions in computer programming languages)
- The keyword “module” starts a block that ends with “endmodule”.
- They could have input and output parameters.
-

```
module MyModule ([parameters]);
     inputs ...
     outputs ...
     internal variables ...
     ...
     Module Code ...
endmodule
```

Main Module

- A specific module is tagged as the main module. (similar to the main on a programming language)
- It is the “program” entry point.
- Usually it has inputs and outputs connected to FPGA physical IO.

Main Module

- A specific module is tagged as the main module. (similar to the main on a programming language)
- It is the "program" entry point.
- Usually it has inputs and outputs connected to FPGA physical IO.

Main Module

- A specific module is tagged as the main module. (similar to the main on a programming language)
- It is the “program” entry point.
- Usually it has inputs and outputs connected to FPGA physical IO.

Blocks

The verilog block are defined with begin - end.

There are two types of blocks:

Initial block
Is executed when the simulation start or as initial value.

Always block
It is always executed and is associated to a list that specify when execute the block

```
always  @ (a or b or sel)
begin
   ...
end
```

Blocks

The verilog block are defined with begin - end.

There are two types of blocks:

Initial block

Is executed when the simulation start or as initial value.

Always block

It is always executed and is associated to a list that specify when execute the block

```
always  @ (a or b or sel)
begin
   ...
end
```

Blocks

The verilog block are defined with begin - end.

There are two types of blocks:

Initial block
Is executed when the simulation start or as initial value.

Always block
It is always executed and is associated to a list that specify when execute the block

```
always  @ (a or b or sel)
begin
   ...
end
```

Blocks

The verilog block are defined with begin - end.

There are two types of blocks:

Initial block
Is executed when the simulation start or as initial value.

Always block
It is always executed and is associated to a list that specify when execute the block

```
always  @ (a or b or sel)
begin
   ...
end
```

Data

Wire e Reg

Wire:
They are used to connect different elements. They can be thought as physical wires. They can be read or assigned but they does not store information. Indeed they need to be continuously driven from an assignment or a module port.

Reg:
Store a value in Verilog. The value is kept until assigned again (similar to a variable)

Numbers

Verilog allows you to specify numbers with size and base:
`<size>'<base><value>`

Examples:

- `12` // decimal (default, base 10)
- `8'h5F` // 8-bit hexadecimal (0x5F)
- `6'b11_0010` // 6-bit binary (0b110010)
- `'o576` // octal (base 8), size omitted

The underscore (_) can be used as a digit separator for readability and is ignored by the compiler.

Scalars and Vectors

Variables in Verilog can be either scalars (single bit) or vectors (multiple bits).

Examples:

- `reg out;`
- `reg [7:0] databus;`
- `wire [1:0] select;`
- `wire enabled;`

Vectors are defined using the syntax: `[MSB:LSB]`. For example, `[7:0]` defines an 8-bit bus from bit 7 (most significant) to bit 0 (least significant).

Assignment

(reg) Blocking assignment:
A = 3;
The expression is evaluated in the execution flux and the variable assigned immediately.

(reg) NON blocking assigment:
A <= A + 1;
The expression is evaluated in the execution flux and the result stored in a temporary variable and assigned on the next step

(wire) Continuous assignment:
assign A = in1 & in2;
used to model combinatorial logic and rename signals.

Assignment

(reg) Blocking assignment:
A = 3;
The expression is evaluated in the execution flux and the variable assigned immediately.

(reg) NON blocking assigment:
A <= A + 1;
The expression is evaluated in the execution flux and the result stored in a temporary variable and assigned on the next step

(wire) Continuous assignment:
assign A = in1 & in2;
used to model combinatorial logic and rename signals.

Assignment

(reg) Blocking assignment:
A = 3;
The expression is evaluated in the execution flux and the variable assigned immediately.

(reg) NON blocking assigment:
A <= A + 1;
The expression is evaluated in the execution flux and the result stored in a temporary variable and assigned on the next step

(wire) Continuous assignment:
assign A = in1 & in2;
used to model combinatorial logic and rename signals.

Operators

Operator	Name
[ ]	bit-select or part-select
( )	parenthesis
!	logical negation
~	negation
&	reduction AND
|	reduction OR
~&	reduction NAND
~|	reduction NOR
^	reduction XOR
~^ or ^~	reduction XNOR
+	unary (sign) plus
-	unary (sign) minus
{ }	concatenation
{{ }}	replication
*	multiply
/	divide
%	modulus

Operator	Name
+	binary plus
-	binary minus
<<	shift left
>>	shift right
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	case equality
!=	case inequality
&	bit-wise AND
^	bit-wise XOR
|	bit-wise OR
&&	logical AND
||	logical OR
?:	conditional

Flow control

If statement in Verilog

The **if** statement in Verilog is used to make decisions based on conditions, similar to other programming languages.

- Syntax:

```
if (condition) begin
   // statements if condition is true
end else begin
   // statements if condition is false
end
```

- The `else` part is optional.
- Used inside `always` or `initial` blocks.
- Conditions are typically expressions involving signals or variables.

Flow control

case statement in Verilog

The **case** statement in Verilog is used for multi-way branching, similar to `switch` in C/C++.

- Syntax:

```
case (expression)
   value1: statement1;
   value2: statement2;
   default: statement_default;
endcase
```

- Each `value` is compared to the `expression`; the matching branch is executed.
- The `default` branch is optional and is executed if no values match.
- Used inside `always` or `initial` blocks.

Flow control

for

The **for loop** in Verilog is used to repeat a block of code a fixed number of times, similar to C/C++. It is used in `initial` or `always` blocks, especially for initializing arrays or generating repetitive hardware structures.

- Syntax: `for (initialization; condition; increment) begin ... end`

```
integer i;
reg [7:0] array [0:7];

initial begin
   for (i = 0; i < 8; i = i + 1) begin
      array[i] = 0;
   end
end
```

- Note: The increment must be written as `i = i + 1` (no ++ operator).
- The for loop is unrolled during synthesis; it does not create a runtime loop in hardware.

Live examples

General informations 1

You can use either the local installation of Vivado or the cloud infrastructure.

Clone the day3 repository from GitHub

git clone https://github.com/FPGA-course-2025/day3

or, if you have already cloned the course repository

Pull the latest changes

git pull (from inside the day3 folder)

Live examples

General informations 2

Inside the verilog directory in day3 you will find the examples

A single example is one or more files with the extension .v or .vhdl and it's contained in a folder with the example name.

You can run the example with Vivado in the following way:

- create a new project
- add all the files in the project as design sources
- add the correct constrains file for the board you are using (also in the day3 verilog directory)
- create the bitstream, it should be generated without errors

Live examples

General informations 3: boards recap

Basys3:

- Part number: xc7a35tcpg236-1
- Constraints file: basys3.xdc

Arty 7:

- Constraints file: arty7.xdc

Counter

day3/verilog/counter

```
module counter(
     input clk,
     input [3:0] sw,
     input [3:0] btn,
     output reg [3:0] led
);
    reg [31:0] counter;

    always @ (posedge clk) begin
        led[0] <= counter[23];
        led[1] <= counter[24];
        led[2] <= counter[25];
        led[3] <= counter[26];
        counter <= counter + 1;
    end
endmodule
```

Let's try some improvements:

- Make the counter running faster (or slower)
- Add a reset button (with a switch or a button)

Led on and off

day3/verilog/ledonoff

```
module ledonoff(
     input clk,
     input [3:0] sw,
     input [3:0] btn,
     output reg [3:0] led
);
    reg oldbtn0;
    reg ledState;

    always @ (posedge clk) begin
        oldbtn0 <= btn[0];
        if (btn[0] && !oldbtn0) begin
            ledState <= ~ledState;
        end
        led[0] <= ledState;
    end
endmodule
```

It seems that sometimes the led does not turn off or on. What could be the problem?

Module instantiation

day3/verilog/modulev

```
module modulev(
     input clk,
     input [3:0] sw,
     input [3:0] btn,
     output reg [3:0] led
);
    wire [3:0] pattern;

    patternMux mux (
        .sel(sw[0]),
        .pattern(pattern)
    );

    always @ (posedge clk) begin
        led <= pattern;
    end
endmodule
```

```
module patternMux(
    input sel,
    output [3:0] pattern
);

    assign pattern = sel ? 4'b1010 : 4'b0101;
endmodule
```

VHDL module instantiation

day3/verilog/modulevhdl

```
module modulev(
     input clk,
     input [3:0] sw,
     input [3:0] btn,
     output reg [3:0] led
);
    wire [3:0] pattern;

    patternMux mux (
        .sel(sw[0]),
        .pattern(pattern)
    );

    always @ (posedge clk) begin
        led <= pattern;
    end
endmodule
```

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

entity patternMux is
    port (
   sel : in std_logic;
   pattern: out std_logic_vector(3 downto 0)
    );
end entity patternMux;

architecture rtl of patternMux is
begin
   pattern <= "0101" when sel = '0' else
              "1010" when sel = '1' else
              "0000";  -- Default case if needed
end architecture rtl;
```

Open Source tools for simulation

- **iverilog**: a free Verilog simulator, it can be used to simulate the code and check the results.
- **gtkwave**: a free waveform viewer, it can be used to visualize the results of the simulation.
- **ghdl**: a free VHDL simulator, it can be used to simulate the VHDL code and check the results.

The day3/verilogExamples directory contains a set of Jupyter notebooks that can be used to run the examples with the open source tools. Similarly,

the day3/vhdlExamples directory contains a set of Jupyter notebooks that can be used to run the VHDL examples with the open source tools.

Open Source tools for synthesis and implementation

While the major FPGA vendors provide their own synthesis and implementation tools, there are also open source alternatives that can be used to synthesize and implement designs for FPGAs. These tools are particularly useful for educational purposes or for those who prefer to work with open source software.

Related to these tools, there is a growing community of open source FPGA tools that provide complete flows. Is some cases, these tools are better than the vendor tools, especially for smaller FPGAs or for specific applications.

Also the open hardware community is growing, with many projects that provide open source designs for FPGAs.

Open Source tools for synthesis and implementation

- **Yosys**: a free synthesis tool, it can be used to synthesize the Verilog code and generate a netlist.
- **nextpnr**: a free place and route tool, it can be used to place and route the netlist generated by Yosys.
- **icepack**: a free tool to generate the bitstream for Lattice iCE40 FPGAs.

A common way to install these tools is to use the **oss-cad-suite** project, which provides a comprehensive set of open source tools for FPGA design.

The day3/ossExamples directory contains a set of Jupyter notebooks that can be used to run the examples with the open source tools and some small FPGAs.

Zynq

Zynq is a family of FPGAs that combines a dual-core ARM Cortex-A9 processor with an FPGA fabric. This allows for a high degree of integration and flexibility, as the processor can be used to run software applications while the FPGA fabric can be used to implement custom hardware accelerators or other logic.

PS-PL

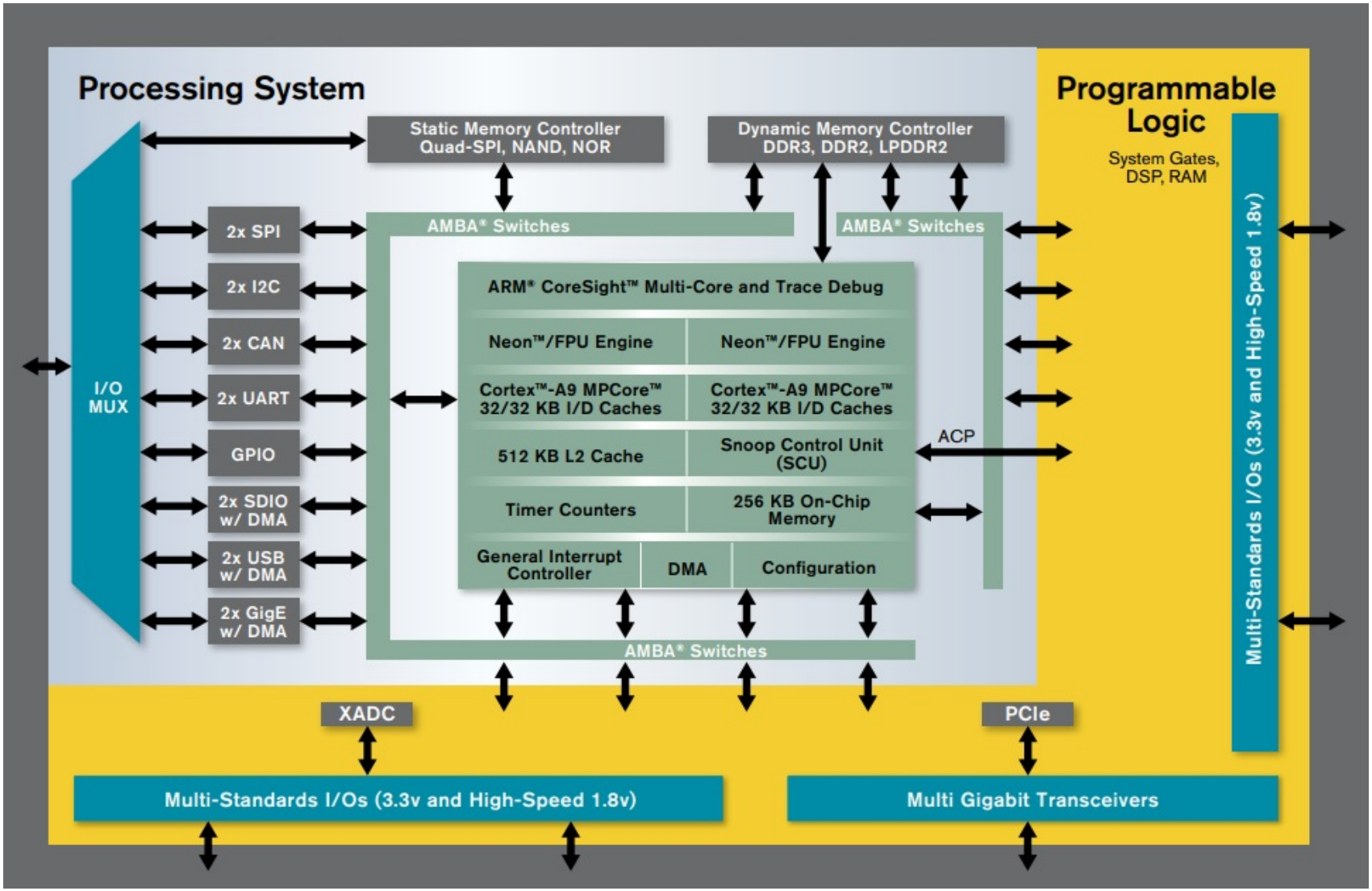
The Zynq architecture is divided into two main parts: the Processing System (PS) and the Programmable Logic (PL).

The PS contains the ARM processor, memory controllers, and other peripherals, while the PL contains the FPGA fabric.

The PS can run a standard operating system, such as Linux, or a real-time operating system like FreeRTOS.

The PS and PL can communicate with each other through a set of interfaces, such as AXI, which allows for high-speed data transfer between the two parts.

PS-PL


Processing System
Programmable Logic
System Gates, DSP, RAM
Static Memory Controller Quad-SPI, NAND, NOR
Dynamic Memory Controller DDR3, DDR2, LPDDR2
I/O MUX
2x SPI
2x I2C
2x CAN
2x UART
GPIO
2x SDIO w/ DMA
2x USB w/ DMA
2x GigE w/ DMA
AMBA® Switches
AMBA® Switches
ARM® CoreSight™ Multi-Core and Trace Debug
Neon™/FPU Engine
Neon™/FPU Engine
Cortex™-A9 MPCore™ 32/32 KB I/D Caches
Cortex™-A9 MPCore™ 32/32 KB I/D Caches
512 KB L2 Cache
Snoop Control Unit (SCU)
ACP
Timer Counters
256 KB On-Chip Memory
General Interrupt Controller
DMA
Configuration
AMBA® Switches
Multi-Standards I/Os (3.3v and High-Speed 1.8v)
XADC
PCIe
Multi-Standards I/Os (3.3v and High-Speed 1.8v)
Multi Gigabit Transceivers

Zynq VHDL demo

Let's try to run a simple VHDL example on the Zynq board.

Pynq

Pynq is a project that provides a Python-based framework for developing applications on Zynq-based FPGAs.

It allows users to write Python code that can interact with the FPGA fabric, enabling the development of hardware accelerators and other custom logic directly usable in Python.

Pynq also provides a set of pre-built overlays, which are pre-configured FPGA designs that can be used to accelerate specific applications, such as image processing or machine learning.

Pynq

With this kind of technology, FPGAs are not anymore just a hardware platform, but they can be used as computing accelerators in a high-level programming language like Python.

With the term co-design, we mean the process of developing both hardware and software components together, taking advantage of the strengths of both domains.

In the pynq website, (**https://www.pynq.io/**) you can find a lot of examples and tutorials to get started with Pynq and Zynq-based FPGAs.

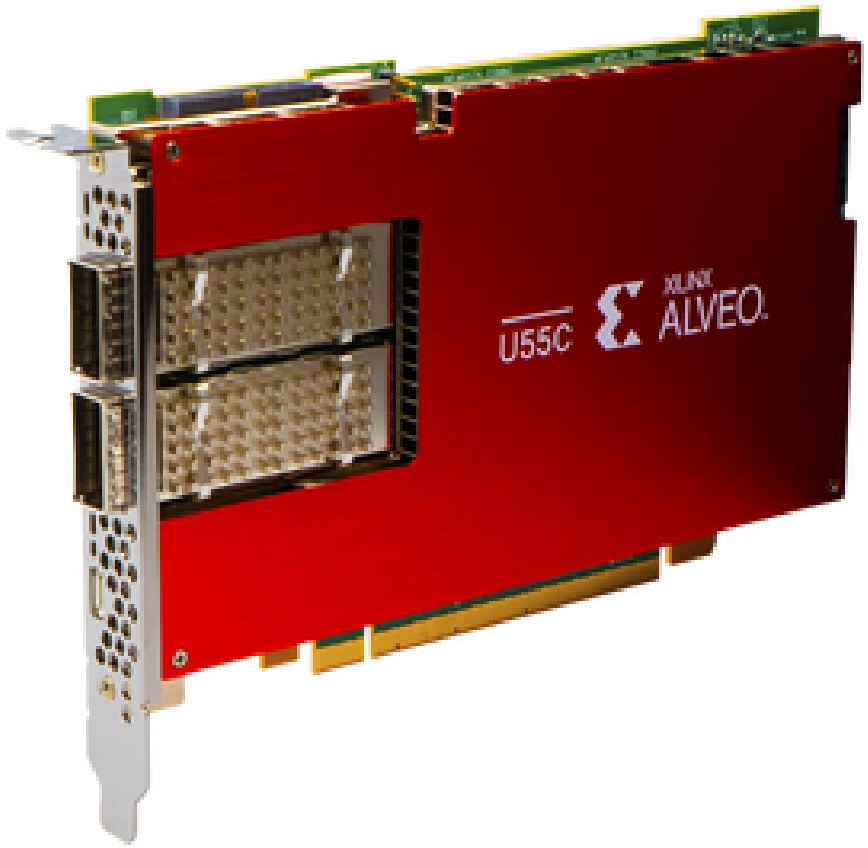
Pynq

Pynq is also available on the cloud environment, let's see an example of how to use it.

in the day3/pynqExamples directory you can find the Jupyter notebook and an example of co-design in c++ as well.

Alveo

Alveo is a family of high-performance FPGAs designed for data center applications.



Alveo U55C

Feature	Xilinx Alveo U55C	Digilent Arty A7
FPGA Family	Xilinx Virtex UltraScale+ HBM	Xilinx Artix-7
FPGA Model	XCU55CHBVAU58P	XC7A35T or XC7A100T
Process Technology	16nm FinFET	28nm planar CMOS
Logic Resources (LUTs)	˜1.3 million	33,650 (A7-35T) / 101,440 (A7-100T)
Block RAM	70.6 Mb + 8 GB HBM	1.8 Mb
DSP Slices	3,744	90 (A7-35T) / 240 (A7-100T)
HBM (High Bandwidth Memory)	8 GB HBM2	None
Memory Bandwidth	Up to 460 GB/s	Limited to external SRAM/DDR
Connectivity	PCIe Gen4 x16	USB-UART, PMOD, Arduino headers
Power Consumption	High (server-grade, >75W typical)	Low (˜1–5W)
Form Factor	Full-height, full-length PCIe card	Standalone board (Arduino-style)
Typical Applications	Data center, HPC, AI, hardware acceleration	Education, prototyping, hobbyist projects
Approx. Price (2025)	>€10,000	˜€150–250
Software Support	Vitis, Vivado, XRT, OpenCL	Vivado, Vitis (for optional MicroBlaze)
Host Requirements	Server/workstation with PCIe x16 slot	None (USB/UART for debugging)
Availability	Specialized enterprise resellers	Widely available online

Table: Comparison between Xilinx Alveo U55C and Digilent Arty A7

HLS

Board like the Alveo U55C are used in data centers, so FPGAs are not only used for prototyping or for electronics projects, but start to be used in high-performance computing and in data centers.

Concurrently to this shift in the use of FPGAs, there is also a shift in the way to program them. The development of High-Level Synthesis (HLS) tools allows developers to write code in high-level programming languages like C, C++, or OpenCL, which is then translated into hardware description languages (HDL) like Verilog or VHDL.