

EE3220 System-on-Chip Design

Lecture Note 9

Introduction to Verilog HDL & FPGA

Review and Preview

- Lecture 1: Course Overview
- Lecture 2: Intro to Modern Microprocessor (CPU Architecture)
- Lecture 3: Embedded System Software (Software Architecture)
- Lecture 4: ARM Cortex-M4 (ARM Hardware)
- Lecture 5: From C to ARM Assembly (C Language, ARM Firmware)
- Lecture 6: Team Based Learning (ARM Firmware)
- Lecture 7: AAPCS (ARM Firmware)
- Lecture 8: Exceptions & Interrupts (ARM Firmware)
- Lecture 9: Intro to HDL & FPGA Design (HDL Design)
- Lecture 10: FPGA based SOC + Quiz 2 (FPGA based SOC)

TODAY'S TOPICS



```
graph TD; A[FPGA DESIGN FLOW] --> B[SYSTEM C]; A --> C[SYSTEM VERILOG]; A --> D[VERILOG]; B --> E[SYSTEM INPRIDEATION FLOW]; C --> F[SYSTEM VERILOG]; D --> G[VERILOG]; E --> H[SYSTEM INPRIDEATION FLOW]; F --> I[SYSTEM VERILOG]; G --> J[VERILOG];
```

- Introduction to Verilog HDL (Focus)
- Overview on SystemVerilog HDL/HVL
- Overview on System-Level Modeling with SystemC
- Introduction to FPGA Design Flow

TODAY'S TOPICS



- Introduction to Verilog HDL (Focus)
- Overview on SystemVerilog HDL/HVL
- Overview on System-Level Modeling with SystemC
- Introduction to FPGA Design Flow

Introduction to Verilog

- Background
- Coding Styles
- Verilog Fundamentals
- Some Examples

Verilog HDL Background

Background

- Verilog HDL History
- **WHY** study Verilog?
- **HOW** to learn Verilog?
- Software vs Hardware

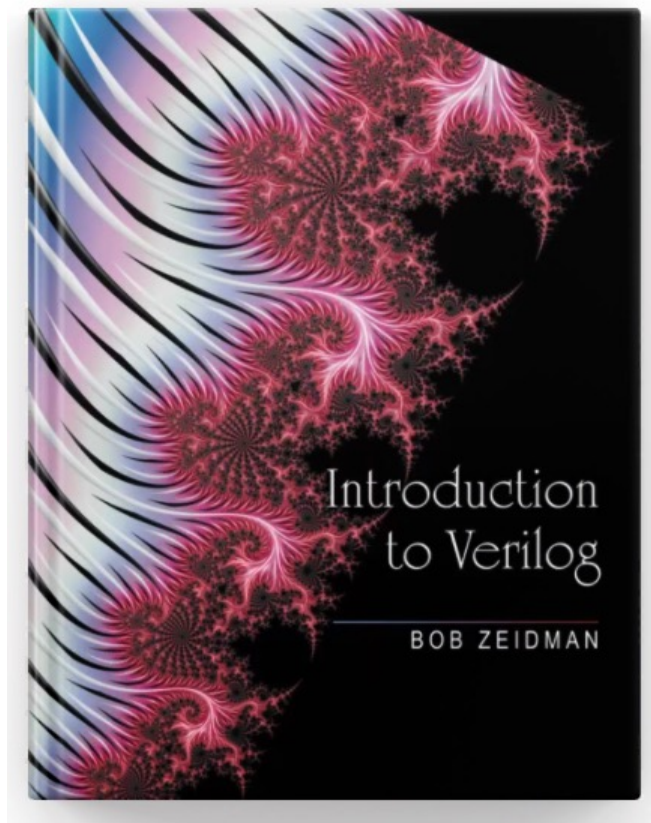
Verilog HDL History

- Verilog Hardware Description Language (HDL)
 - Gateway Design Automation developed Verilog as **LOGIC SIMULATION LANGUAGE** in 1984
 - HDL = Hardware Description Language (not Hard & Difficult Language)
 - Quite Easy: Similar to C
- Major Applications
 - **SIMULATION**: Verify the design is correct (functional and timing verification)
 - **SYNTHESIS**: Source code compiled into logic gates
 - It can also be used for formal verification.

WHY study Verilog?

- Hardware Engineers command **GOOD SALARIES** all over the world.
 - IC Logic Designers
 - IC Verification Engineers
 - IC Post-Silicon Validation Engineers
 - System Design Engineers
- Software Engineers also need to understand hardware design difficulties.
- System Architect must understand both **HARDWARE** and **SOFTWARE**.

HOW to learn Verilog?



https://www.tutorialspoint.com/compile_verilog_online.php
or Synopsys VCS, Cadence NCVerilog, Siemens Modelsim

```
1 module main;
2   wire [31:0] count;
3   wire rst, clk;
4   initial
5   begin
6     #5 $display("START COUNT = ", count);
7     #1000 $display("END COUNT = ", count);
8     $finish;
9   end
10  testbench tb(rst, clk);
11  counter cnt(rst, clk, count);
12 endmodule
13
14 module testbench(rst, clk);
15   output rst, clk;
16
17   reg rst2, clk2;
18   assign rst = rst2;
19   assign clk = clk2;
20
21   initial
22   begin
23     rst2 = 1;
24     #10 rst2 = 0;
25   end
26   initial
27   begin
28     clk2 = 0;
```

```
START COUNT =      0
END COUNT =      498
main.v:8: $finish called at 1005 (1s)
```

Software vs Hardware

- LOGIC
 - Software: 0 and 1
 - Hardware: 0, 1, Z, X
- CONCURRENCY
 - Software: Single Program Counter
 - Hardware: Millions of Logic Gates **CONCURRENTLY UPDATED**

Verilog Coding Styles

Verilog Coding Styles

- BEHAVIORAL LEVEL

- SIMULATION OK

- Often NOT SYNTHESIZABLE Design

```
initial begin
    ...
    #10 $display("Result = %0d", result);
end
```

- REGISTER TRANSFER LEVEL (RTL)

- SYNTHESIZABLE Design

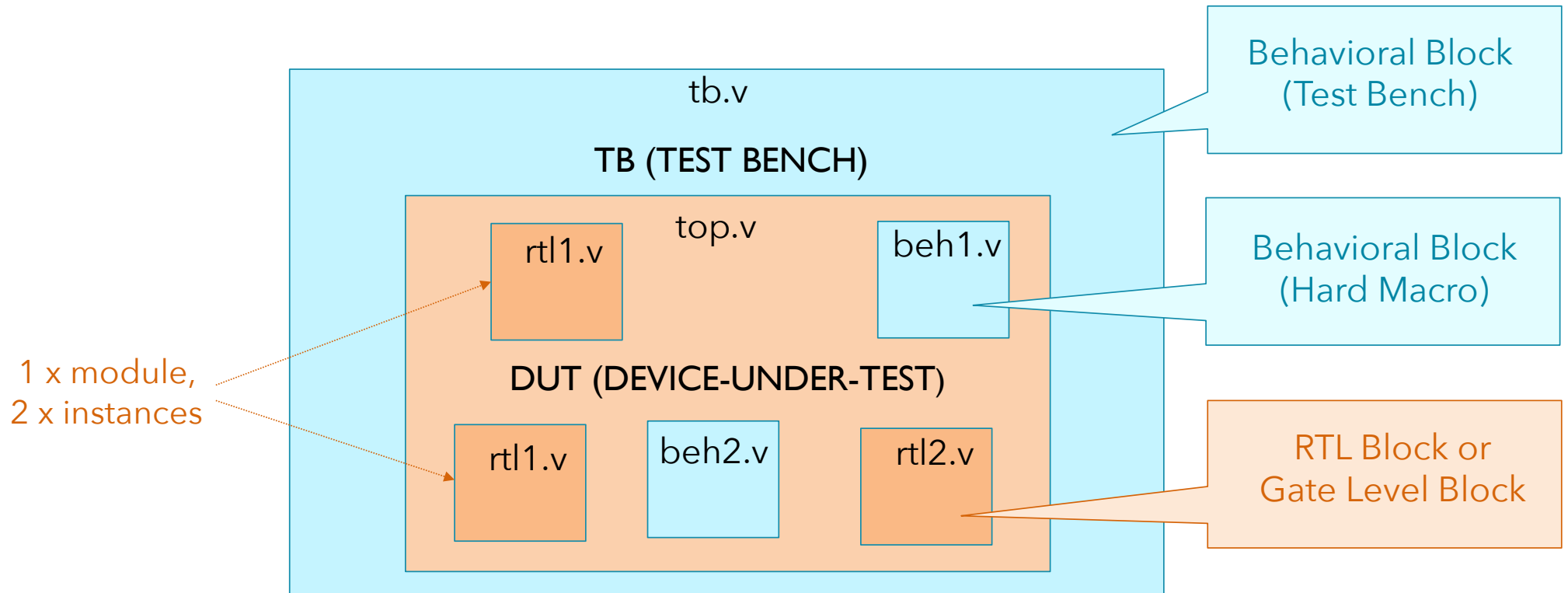
```
always @(posedge CLK) begin
    ...
    Q <= DATA;
end
```

- GATE LEVEL

- STRUCTURAL Design (after Synthesis)

```
module half_add(input a, b, output s, c);
    xor(sum, a, b);
    and(carry, a, b);
endmodule
```

Verilog Coding Styles



Verilog Coding Styles: Behavioral Level

- Why do we build Behavioral Model?
 - Test Bench (Functional & Timing Verification)
 - Golden Model (Functional & Timing Verification)
 - Performance Evaluation
- Behavioral Level Syntax
 - Logical and arithmetic operators might not be synthesizable
 - For example, we cannot easily synthesize **FLOATING-POINT DIVIDER**
 - Syntax such as "initial" "while" "for" and delays (#10) are not synthesizable

Verilog Coding Styles: Register Transfer Level (RTL)

- RTL means **SYNTHESIZABLE**
- Exact Logic Synthesis TBD to meet:
 - Area Constraints
 - Timing Constraints
 - Power Constraints
- Logic Synthesis
 - Performed after Functional Verification.
 - Simulation delays (#10) may be included but are removed during the synthesis
 - “Unused” logic may be **OPTIMIZED** and **REMOVED**

```
module rtl_ex (y, a, b);  
    input a, b;  
    output y;  
    assign y = a && b;  
endmodule
```

Verilog Coding Styles: Gate Level

- How to Generate Gate Level?
 - Manual Design
 - LOGIC SYNTHESIS
- Next Steps
 - Clock Tree Synthesis (CTS)
 - Design For Testing (DFT)
 - Layout
 - Timing Verification (versus Functional Verification)

```
module gate_ex (y, a, b);  
    input a, b;  
    output y;  
    AND2 a1 (y, a, b);  
endmodule
```

Verilog Fundamentals

Verilog Fundamentals

- Syntax Basics
- Data Types
- Hierarchical Structures
- Module Components
- Operators
- Assignments
- Control Statements

Syntax Basics: Lexical Elements

- **CASE SENSITIVE** (like C)
- Variable names begin with alphanumeric character or underscore (`_`)
- Semicolons (`;`) are line terminators
- C-like Comments (`// ...` or `/* ... */`)
- **SYSTEM TASKS** begins with `$` character, such as `$display` or `$signed`
- **COMPILER DIRECTIVES** begins with ``` character, such as ``define` ``include` ``ifdef`

Syntax Basics: Reserved Keywords

RESERVED KEYWORDS							
and	always	assign	attribute	begin	buf	bufif0	bufif1
case	cmos	deassign	default	defparam	disable	else	endattribute
end	endcase	endfunction	endprimitive	endmodule	endtable	endtask	event
for	force	forever	fork	function	highz0	highz1	if
initial	inout	input	integer	join	large	medium	module
nand	negedge	nor	not	notif0	notif1	nmos	or
output	parameter	pmos	posedge	primitive	pulldown	pullup	pull0
pull1	rcmos	reg	release	repeat	rnmos	rpmos	rtran
rtranif0	rtranif1	scalared	small	specify	specparam	strong0	strong1
supply0	supply1	table	task	tran	tranif0	tranif1	time
tri	triand	trior	triereg	tri0	tri1	vectored	wait
wand	weak0	weak1	while	wire	wor		

Data Types: Logic Representation

- A bit may have the following value:
 - 0 represents a logic **LOW** (FALSE)
 - 1 represents a logic **HIGH** (TRUE)
 - Z represents **HIGH IMPEDANCE**
 - X represents either 0, 1, or Z

LOGIC OR				
	0	1	X	Z
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

LOGIC AND				
&	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

Data Types: Number Representation

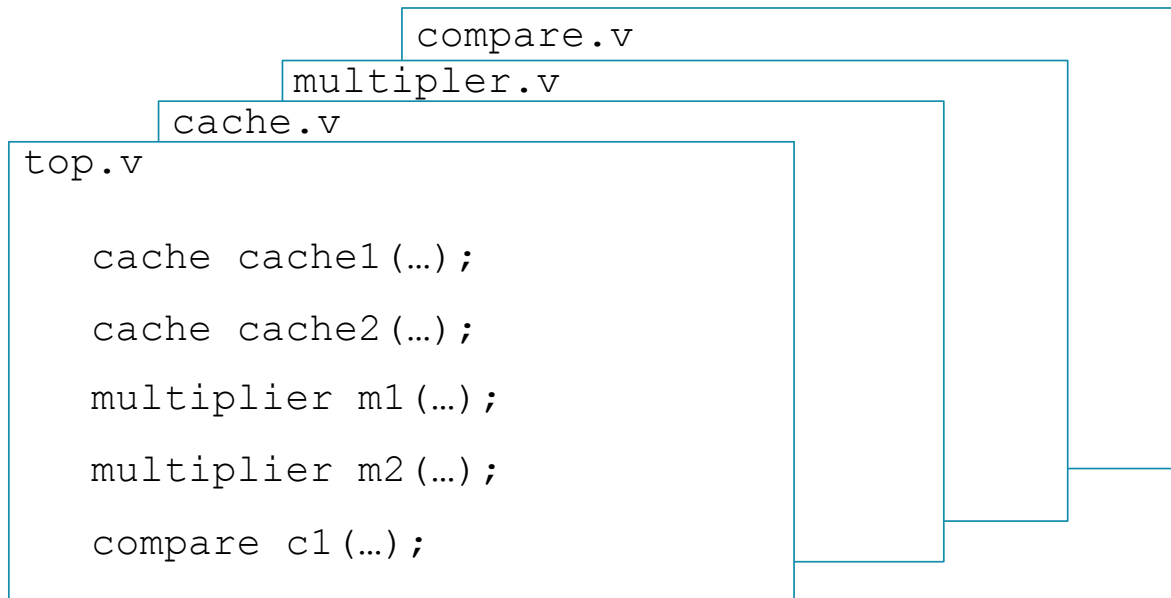
- `<size>'<base format> <number>`
 - `<size>`: number of bits
 - `<base format>`: b, d, o or h
 - `<number>`: legal digits with `'_'`
- Examples:
 - `3'b1, 3'b001`
 - `12'h8FF, 16'hFFFF_FFFF`

Data Types: Declaration (`reg` and `wire`)

- `reg`
 - A '`reg`' stores values from one assignment to another assignment in an `initial` or `always` block
 - BUT `reg` is **NOT NECESSARILY** implemented as a hardware register (memory). It is possible that a `reg` is implemented in combinational logic
 - Default value is `1'bX`
- `wire`
 - A '`wire`' corresponds to a physical wire
 - Less common net types include `wand`, `wor`, `tri`, `triand`, `trior`

Hierarchical Structures

- Hierarchical Structure implemented by **DEFINING** and **INSTANTIATING** modules (**DIVIDE & CONQUER**)

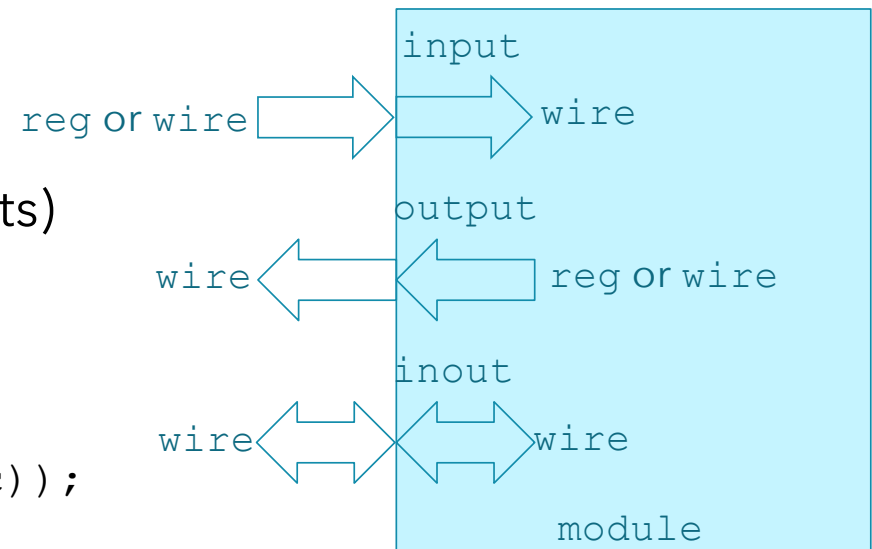


Module Components

```
module <module name> #(<param list>) (<port list>);  
  
    <DECLARATIONS>  
  
    <INSTANTIATIONS>  
  
    <DATA FLOW STATEMENTS>  
  
    <BEHAVIORAL BLOCKS>  
  
    <TASKS AND FUNCTIONS>  
  
endmodule
```

Module Components: Port Interface

- Ports
 - **MODULE INTERFACE** to communicate with external environment
- Types
 - input, output, inout (one or more bits)
- Unnamed vs Named Connections
 - `modA inst1 (a, b, c);`
 - `modA inst2 (.in1(a), .in2(b), .out(c));`



Module Components: Declaration of reg, wire

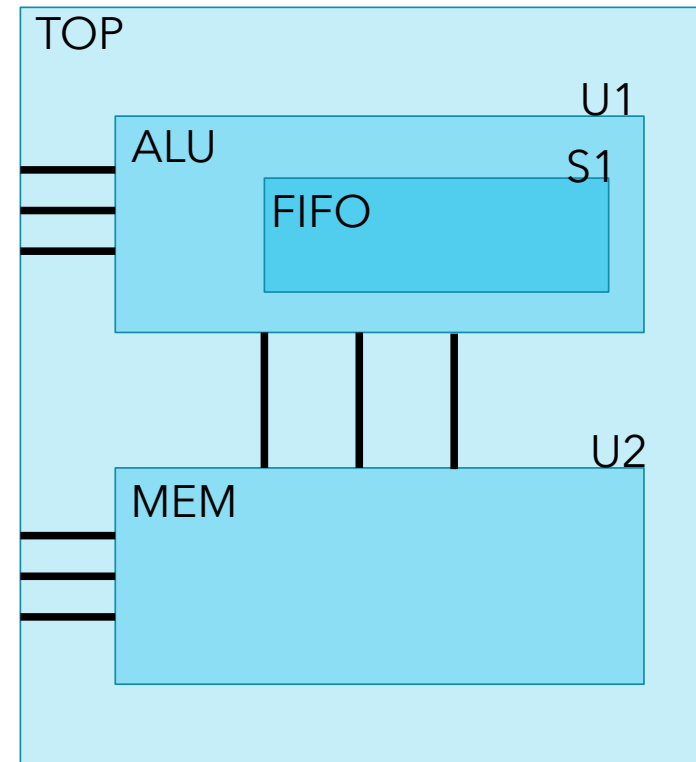
```
module RAM (output [7:0] Obus, input [7:0] Ibus, input [3:0] Addr,  
            input Clk, input Read, input Select);  
  
    reg [7:0] memory[15:0];  
    reg [7:0] obusreg;  
    wire      selectclk;  
  
    assign Obus = obusreg;  
    assign selectclk = Select & Clk;  
    always @(posedge selectclk)  
        if (Read==1'b0') memory[Addr] = Ibus;  
        else                obusreg = memory[Addr];  
endmodule
```

Module Components: Instantiation

```
module TOP (...);  
  ALU U1 (...);  
  MEM U2 (...);  
endmodule
```

```
module ALU(...);  
  FIFO S1 (...);  
endmodule
```

```
module FIFO(...);  
endmodule
```



Module Components: Behavioral Blocks

```
module TOP (...);  
...  
initial  
begin  
    $dumpvars;  
    $monitor(" %b| %b|", clk, a);  
end  
endmodule
```

Behavioral blocks are useful for debugging. They are also useful to represent third-party IP blocks (such as cache, multiplier) to speed up simulation.

Module Components: Tasks & Functions

- Tasks vs Functions
 - Calling conventions are different.
 - Tasks, not functions, may contain simulation delay.
- System Tasks
 - \$display, \$write,
\$monitor, \$time, \$finish,
\$stop, \$random

```
module T (input a1, a2, output reg [1:0] b1, b2);  
  always @(a1, a2)  
  begin  
    b1 = out (a1, a2);  
    out_task (a1, a2, b2);  
  end  
  function [1:0] out(input in1, in2);  
  begin  
    if (in1) out = {in2, in1}  
    else out = {in1, in2};  
  end  
endfunction  
task out_task(input in1, in2, output [1:0] out);  
begin  
  if (in1) out = {in2, in1};  
  else out = {in1, in2};  
end  
endtask  
endmodule
```

Operators: Handling Additional Bit Representations

- Operators to support X, Z
 - === if (a === b)
 - !== if (a !== b)
- Operators to support Unary Functions
 - &, |, ^
- Operators to support concatenation
 - {} {co, sum} = a + b + ci
 - {{{}} b = {3{a}}
- Operators to support Arithmetic Shift
 - <<< b <<< 1
 - >>> b >>> 1

```
if (data[7:0] == 8'bx)
begin
    // NEVER EXECUTED
end

if (data[7:0] === 8'bx)
begin
    // SOMETIMES EXECUTED
end

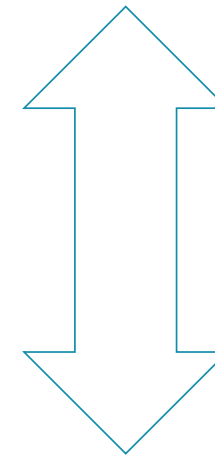
and_all_bits = &data;
or_all_bits  = |data;
xor_all_bits = ^data;
```

Operators: Precedence

Operator precedence

Unary, Multiply, Divide, Modulus	$+, -, !, \sim$ $*, / \%$
Add, subtract, shift	$+, -$ $\ll, \gg$
Relational Equality	$<, <=, >, >=$ $=, ==, !=$ $==, !=$
Reduction Logical	$\&, \sim\&$ $\wedge, \wedge\sim$ $, \sim $ $\&\&$ $ $
Conditional	$?:$

HIGHER PRECEDENCE



LOWER PRECEDENCE

What does "A && B || C && D" mean?

(A && B) || (C && D)? A && (B || C) && D?

Assignments

- Continuous Assignment (wire)

- `assign A = (B|C) & D;`
- `ALU U1 (A, B, C, D);`

- Procedural Assignment (reg)

- `initial begin rst = 1; #10 rst = 0; end`
- `always @(B or C) begin A = ~(B & C); end`

- Blocking and Non-Blocking Procedural Assignment

- `always @(posedge clk) begin C = C+1; A = C+1; end`
- `always @(negedge clk) begin D <= D+1; A <= D+1; end`

Sensitivity List Keywords:
posedge, negedge
or (same as comma ",")

Control Statements

- if / else / else if statements
 - Similar to C
- for statements
 - Similar to C
- case / casex / casez
 - case doesn't support DON'T CARE
 - casez considers 0, 1, z as DON'T CARE (?)
 - casex considers 0, 1, x, z as DON'T CARE (x)

```
always @(W or S)
begin
    if (!W) out = old_out;
    else if (!S) q = status;
end

always @(state)
begin
    casez (state)
        3'b1??: fsm = 0;
        3'b01?: fsm = 1;
        default: fsm = 1;
    endcase

    casex (state)
        3'b01x: fsm2 = 0;
        3'b0xx: fsm2 = 1;
        default: fsm2 = 1;
    endcase
end
```

Some Examples

D Flip Flops: Synchronous vs Asynchronous Reset

```
module DFF (input clk, reset, D, output reg Q);  
always @(posedge clk)  
begin  
    if (!reset)  
        Q <= 1'b0;  
    else  
        Q <= D;  
    end  
end  
endmodule
```

```
module DFF2 (input clk, reset, D, output reg Q);  
always @(posedge clk or negedge reset)  
begin  
    if (!reset)  
        Q <= 1'b0;  
    else  
        Q <= D;  
    end  
end  
endmodule
```

Asynchronous Reset is preferred
to avoid Race Conditions

8-bit Counter

```
module counter (input clk, reset, dyop, output reg [7:0] count);  
  
    always @(posedge clk or negedge reset)  
    begin  
        if (!reset)  
            count <= 8'b0;  
        else begin  
            if (stop)  
                count <= count;  
            else  
                count <= count + 1;  
        end  
    end  
  
endmodule
```

Parity Generator

```
module add_parity (DOUT, DIN);  
  
    output [32:0] DOUT;  
    input  [31:0] DIN;  
  
    wire parity;  
  
    assign parity = ^DIN;  
    assign DOUT = {DIN, parity};  
  
endmodule
```

TODAY'S TOPICS

- Introduction to Verilog HDL (Focus)
- Overview on SystemVerilog HDL/HVL
- Overview on System-Level Modeling with SystemC
- Introduction to FPGA Design Flow

SystemVerilog: Extension on Verilog

- Verilog
 - Hardware Description Language
- SystemVerilog
 - Hardware Description Language
 - **HARDWARE VERIFICATION LANGUAGE**
- Design versus Verification
 - Verification is much **TIME CONSUMING** than Design

SystemVerilog: How to **PROVE** design is correct?

- SOC as a State Machine
 - SOC may contain 1 billion flip flops, corresponding to $2^{1,000,000,000}$ states
 - Impossible to simulate all 2^{100} states (Earth contains $\sim 2^{167}$ Atoms)
- Solutions
 - **CONSTRAINED RANDOM SIMULATION**
 - Verification Coverage Analysis
 - Formal Verification

SystemVerilog: Key Features

- OOP (Similar to C++)
 - Data Encapsulation (no globals)
 - Inheritance (base class, sub-classes)
 - Polymorphism (function performs differently depending on the object type)
- Assertions (SVA)
 - `assert (state > 3);`
- Coverage
 - Functional Coverage (tape-out only when Coverage > 90% in all blocks)

SystemVerilog: UVM

- Universal Verification Methodology (Accellera)
 - Open Source SystemVerilog Library release supported by **ALL MAJOR EDA VENDORS**
 - **DE-FACTO VERIFICATION METHODOLOGY** (especially VIP)
- Key Features
 - Core Classes: `uvm_object`, `uvm_transaction`, `uvm_components`
 - Testbench Components: `testbench_top`, `test`, `transaction`, `generator`, `driver`, `monitor`, `agent`, `scoreboard`
 - UVM Phases: Build, Run-time, Clean-up (Callbacks)

TODAY'S TOPICS

- Introduction to Verilog HDL (Focus)
- Overview on SystemVerilog HDL/HVL
- Overview on System-Level Modeling with SystemC
- Introduction to FPGA Design Flow

SystemC: WHAT? WHY?

- What is SystemC?
 - SystemC Class Library can MODEL HARDWARE
 - Modules, Ports, Processes, Channels, Interfaces, Events
 - RTL (Register Transfer Level), TLM (Transaction Level Modeling)
 - SC_LOGIC_0, _1, _X, _Z
- Why SystemC?
 - Many standards are written in English with untimed C/C++ CODE
 - SystemC Design can be used as GOLDEN HARDWARE (PCIe)
 - High Level Synthesis

SystemC: HOW?

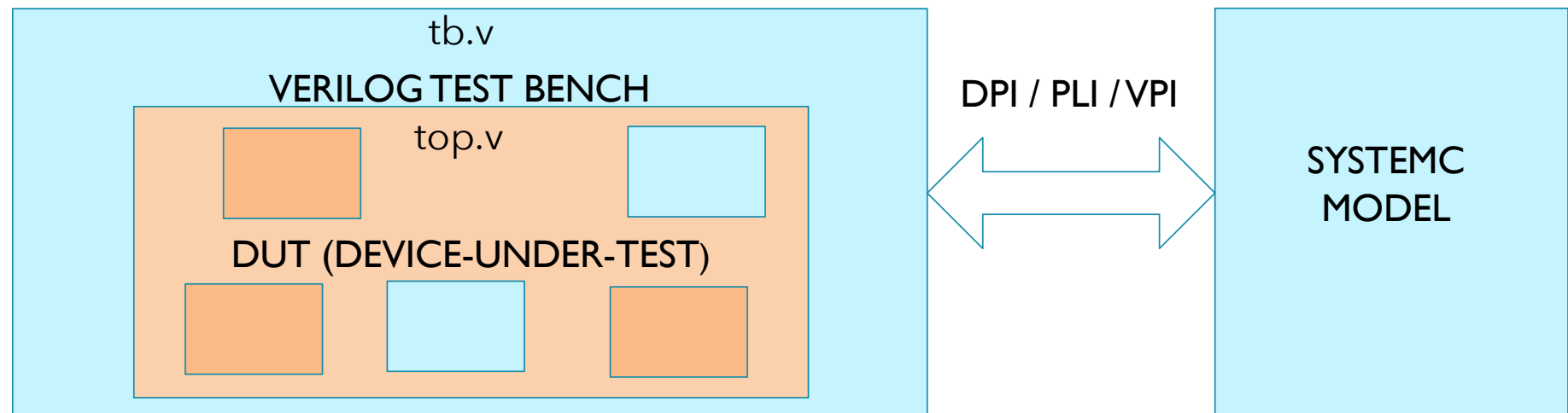
- How to implement SystemC Model?
 - Write C++ Code (like any C++ programs)
 - Supports Timing, Concurrency, Reactivity
 - Fast Simulation
- How to use SystemC Model?
 - High Level Synthesis (HLS)
 - SystemC Design can interface with Verilog and SystemVerilog testbenches using PLI and DPI

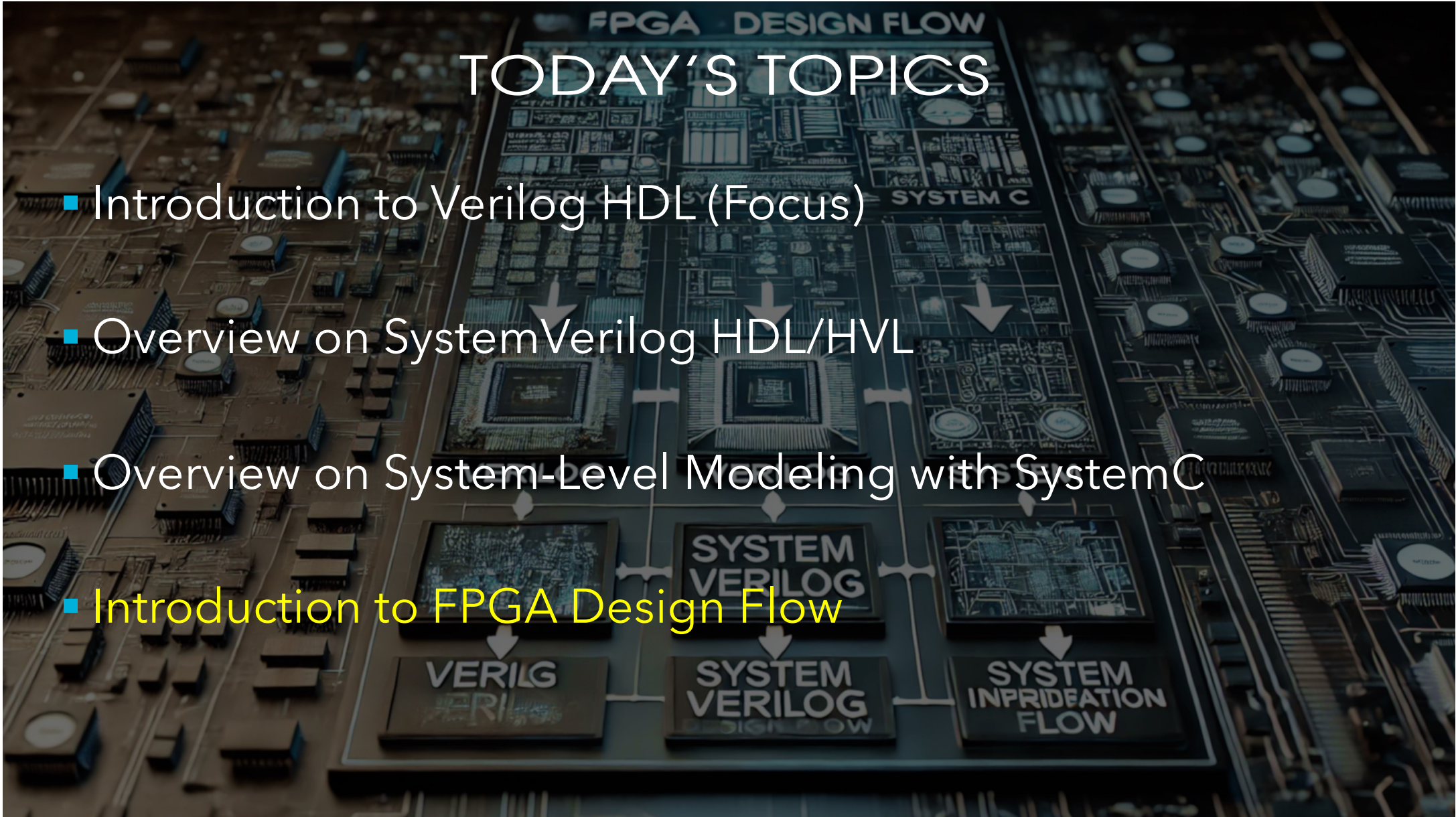
```
import "DPI" function real sin(real x);
real slope1, slope2;
always @(posedge clk)
begin
    slope1 <= $sine(angle); // PLI (OLD)
    slope2 <= sin(angle);   // DPI (NEW)
end

// USER DEFINED SYSTEM TASK (PLI)
int sine_calltf() {
    double a, result;
    a = tf_get(1);
    result = sin(a);
    tf_putrealp(0, result);
    return();
}
```

SystemC: Integrating SystemC with Verilog TB

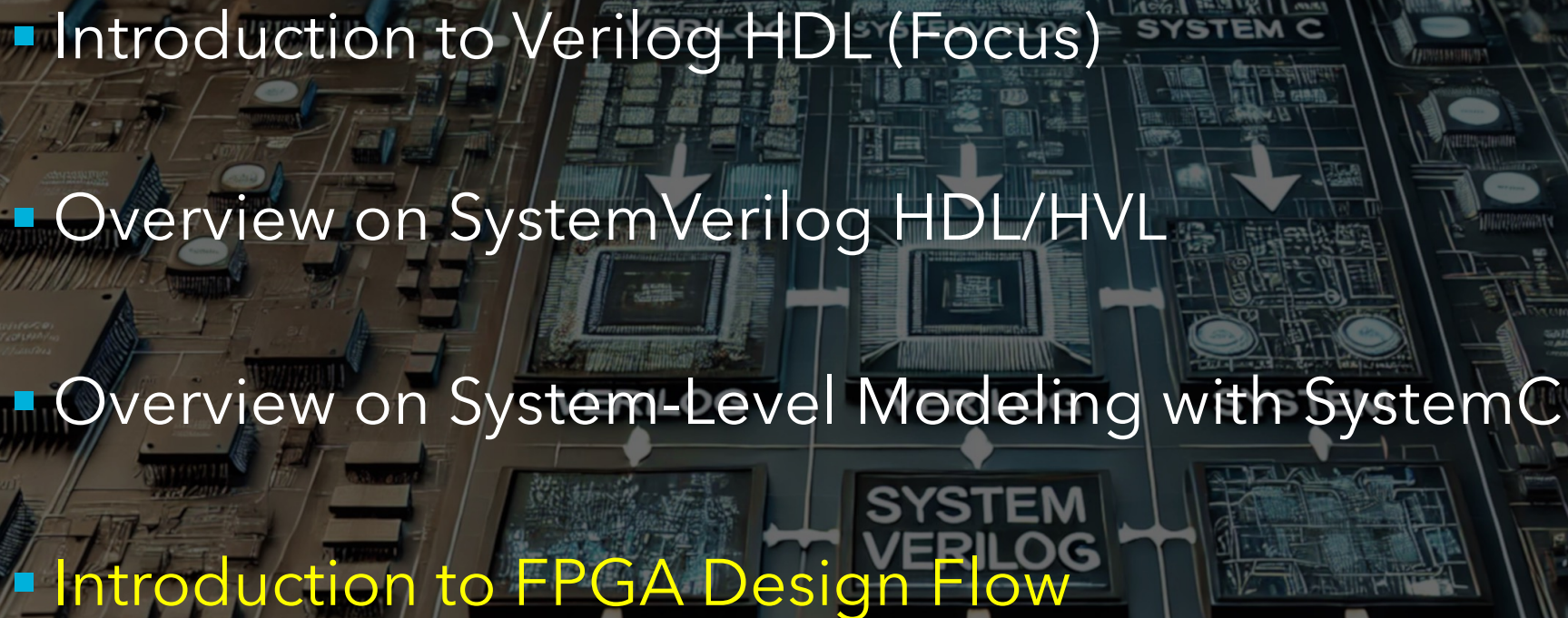
- How to run co-simulate SystemC with Verilog / SystemVerilog?
 - Use co-simulation interface, like SystemC Verification Library (SCV) or third-party tool, such as Cadence Incisive, Synopsys VCS
 - Use Verilog DPI (Direct Programming Interface) or PLI (Programming Language Interface) or SystemVerilog VPI (Verilog Procedural Interface)





TODAY'S TOPICS

- Introduction to Verilog HDL (Focus)
- Overview on SystemVerilog HDL/HVL
- Overview on System-Level Modeling with SystemC
- Introduction to FPGA Design Flow

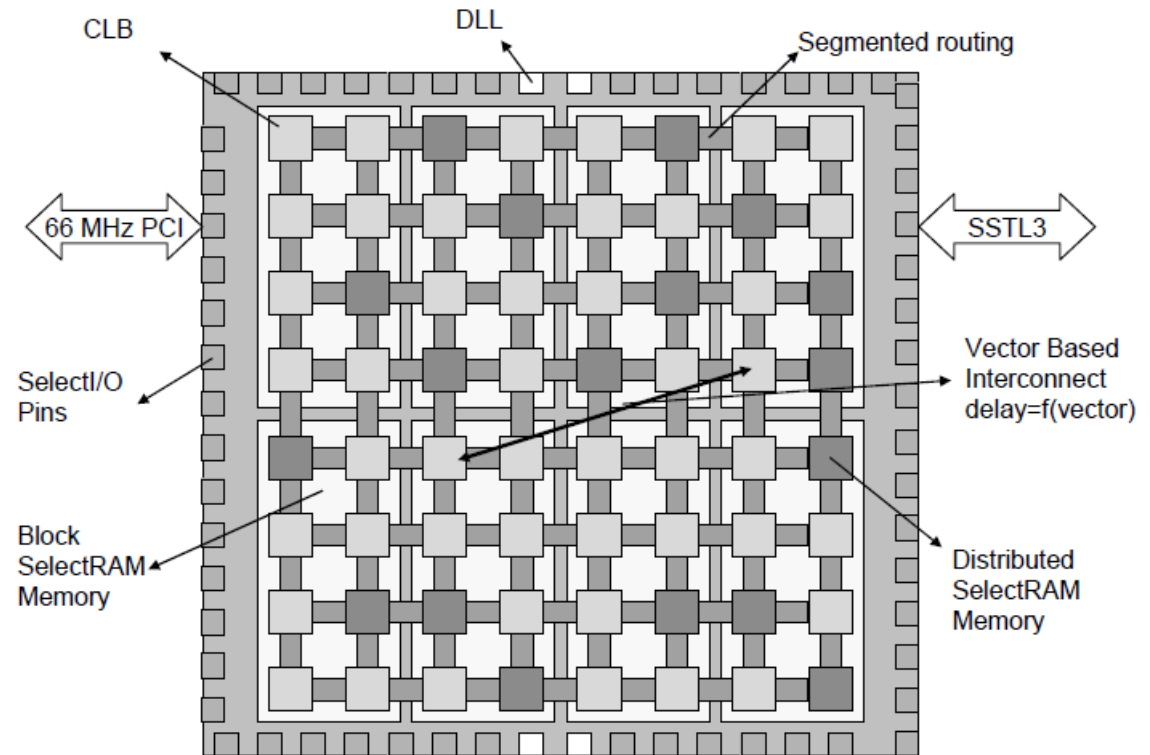
- 
- Introduction to Verilog HDL (Focus)
 - Overview on SystemVerilog HDL/HVL
 - Overview on System-Level Modeling with SystemC
 - Introduction to FPGA Design Flow

Quick Overview on FPGA

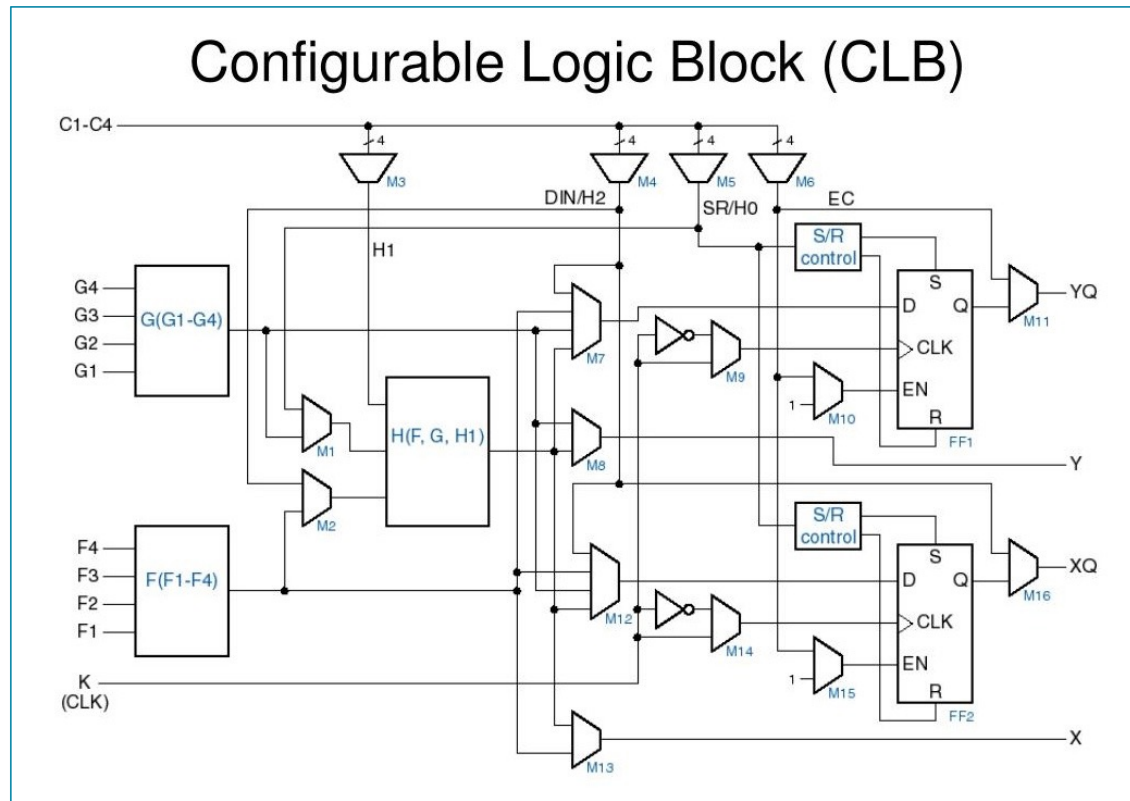
- Background
- FPGA + CPU Core Architectures
- FPGA Design Tools

FPGA Overview: Block Diagram

- Configurable Logic Blocks
 - RECONFIGURABLE LOGIC
- Delay Locked Loops
 - RECONFIGURABLE TIMING
- Segmented Routing
 - RECONFIGURABLE INTERCONNECT
- COMMON IP BLOCKS
 - CPU Cores
 - Ethernet/USB/PCIe/SATA
 - ADC/DAC



FPGA Overview: Configurable Logic Block (CLB)



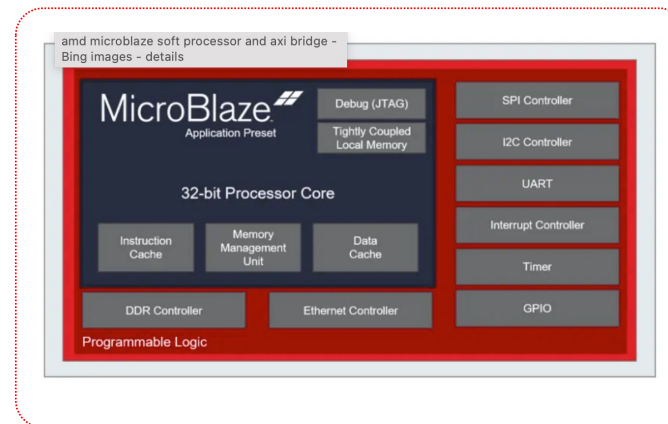
Non-volatile memory becomes configurable logic

Commercial FPGA supports both Hard & Soft IPs

Zynq® UltraScale+™ MPSoCs: CG Devices

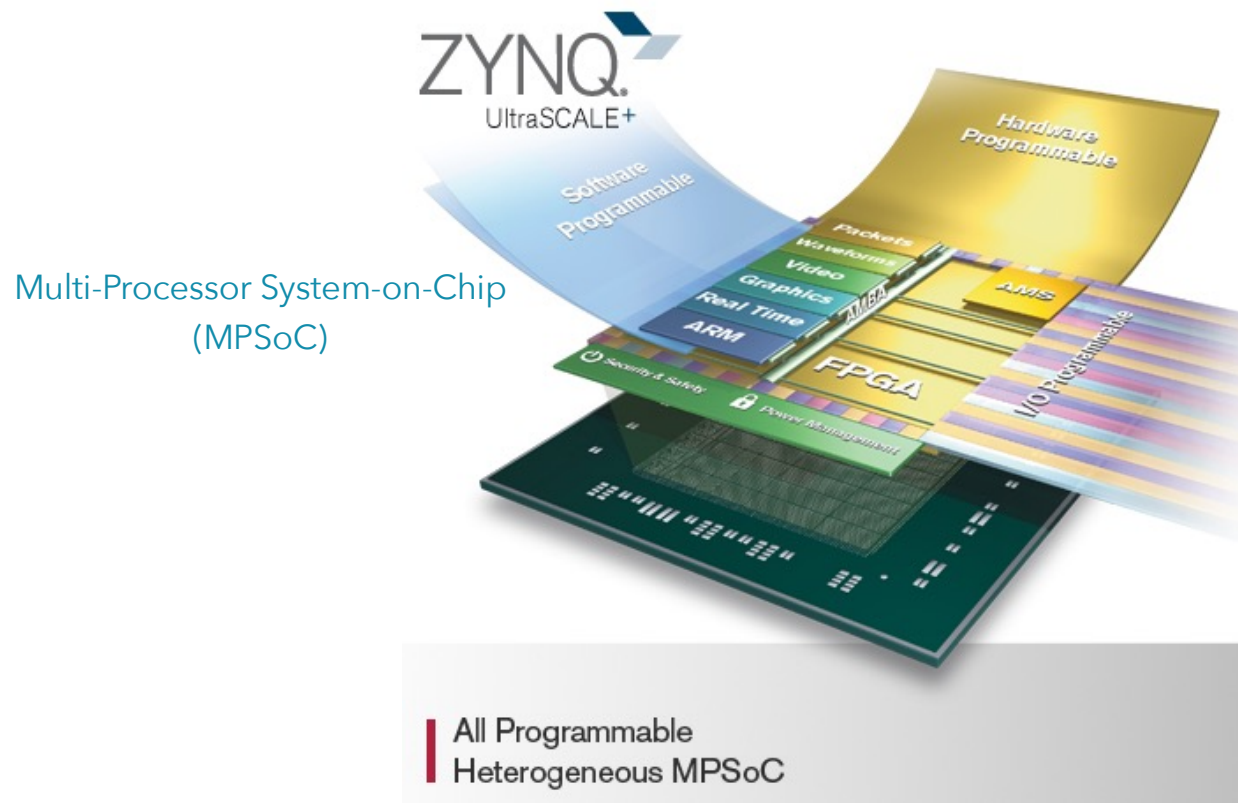
	Device Name ⁽¹⁾	ZU2CG	ZU3CG	ZU4CG	ZU5CG	ZU6CG	ZU7CG	ZU9CG
Processing System (PS)	Application Processor Core	Dual-core ARM® Cortex™-A53 MPCore™ up to 1.3GHz						
	Processor Unit Memory w/ECC	L1 Cache 32KB I / D per core, L2 Cache 1MB, on-chip Memory 256KB						
	Real-Time Processor Core	Dual-core ARM Cortex-R5 MPCore up to 533MHz						
	Processor Unit Memory w/ECC	L1 Cache 32KB I / D per core, Tightly Coupled Memory 128KB per core						
	Dynamic Memory Interface	x32/x64: DDR4, LPDDR4, DDR3, DDR3L, LPDDR3 with ECC						
	Static Memory Interfaces	NAND, 2x Quad-SPI						
	High-Speed Connectivity	PCIe® Gen2 x4, 2x USB3.0, SATA 3.1, DisplayPort, 4x Tri-mode Gigabit Ethernet						
	General Connectivity	2xUSB 2.0, 2x SD/SDIO, 2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO						
	Power Management	Full / Low / PL / Battery Power Domains						
	Security	RSA, AES, and SHA						
Integrated Block Functionality	AMS - System Monitor	10-bit, 1MSPS - Temperature and Voltage Monitor						
	PS to PL Interface	12 x 32/64/128b AXI Ports						
Programmable Logic (PL)	System Logic Cells (K)	103	154	192	256	469	504	600
	CLB Flip-Flops (K)	94	141	176	234	429	461	548
	CLB LUTs (K)	47	71	88	117	215	230	274
	Max. Distributed RAM (Mb)	1.2	1.8	2.6	3.5	6.9	6.2	8.8
	Total Block RAM (Mb)	5.3	7.6	4.5	5.1	25.1	11.0	32.1
	UltraRAM (Mb)	-	-	13.5	18.0	-	27.0	-
	Clock Management Tiles (CMTs)	3	3	4	4	4	8	4
	DSP Slices	240	360	728	1,248	1,973	1,728	2,520
	PCI Express® Gen 3x16 / Gen4x8	-	-	2	2	-	2	-
	150G Interlaken	-	-	-	-	-	-	-
	100G Ethernet MAC/PCS w/RS-FEC	-	-	-	-	-	-	-
	AMS - System Monitor	1	1	1	1	1	1	1
	GTH 16.3Gb/s Transceivers	-	-	16	16	24	24	24
	GTY 32.75Gb/s Transceivers	-	-	-	-	-	-	-
	Extended ⁽²⁾	-1 -2 -2L						
	Industrial	-1 -1L -2						

Hard IPs

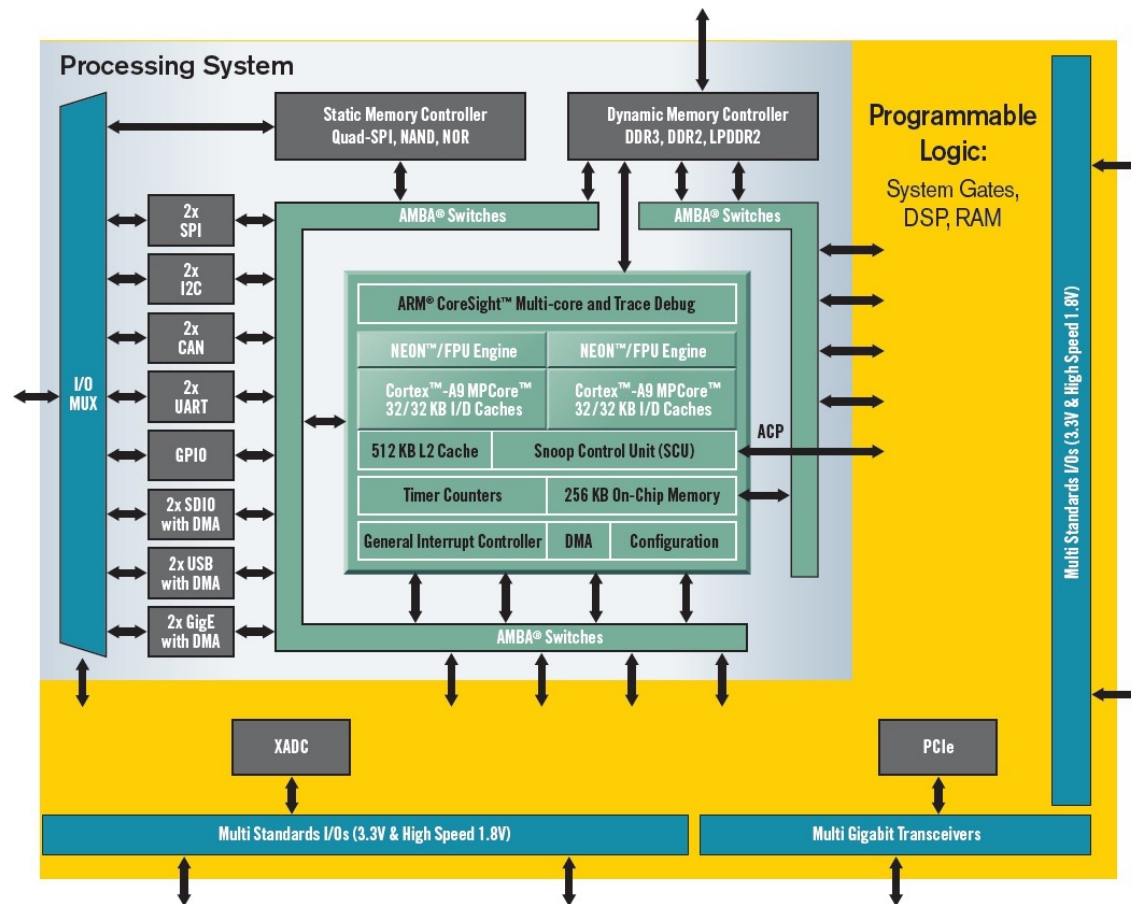


Soft IPs

AMD FPGA + CPU Core Architectures: ZYNQ Family



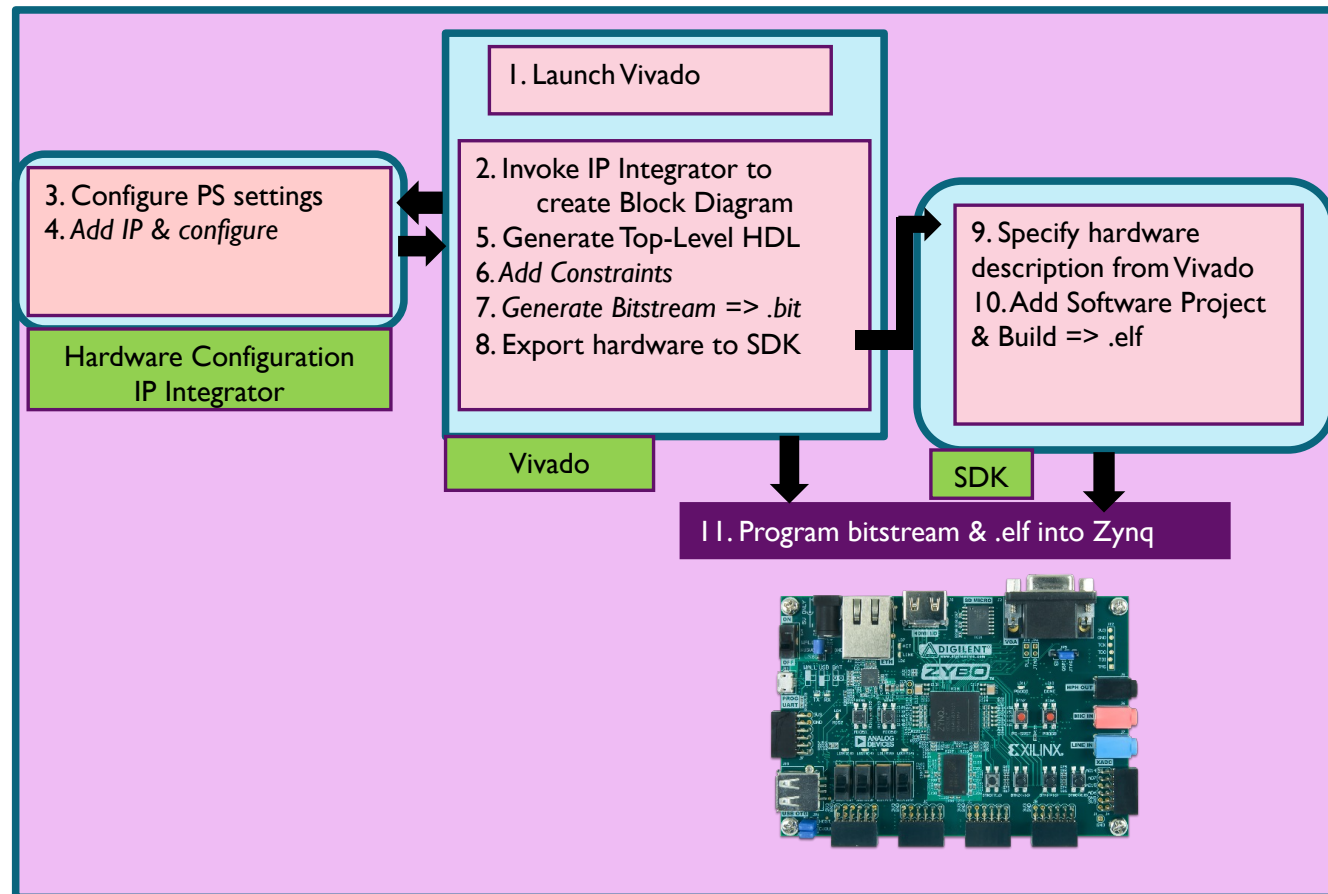
AMD FPGA + CPU Core Architectures



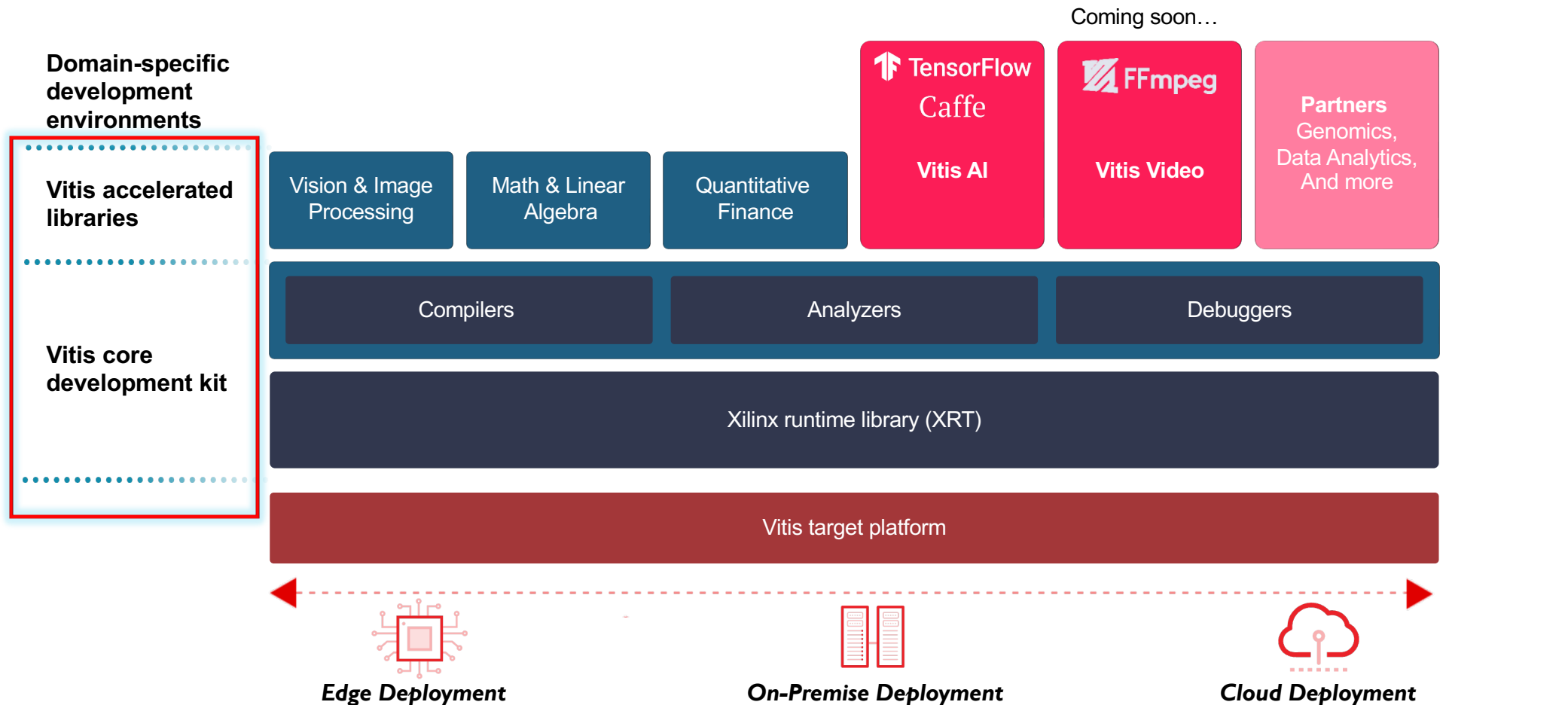
AMD FPGA + CPU Core Architectures

- Embedded design with AMD / Xilinx Zynq is based on:
 - Processor and peripherals
 - Dual ARM® Cortex™ -A9 processors of Zynq-7000 SoC
 - AXI interconnect
 - AXI component peripherals
 - Reset, clocking, debug ports
 - Software platform for processing system
 - Bare Metal Applications or OS's (e.g. Linux, FreeRTOS)
 - C language support
 - Processor services
 - C drivers for hardware
 - User application
 - Interrupt service routines (optional)

AMD FPGA Design Tools: Vivado

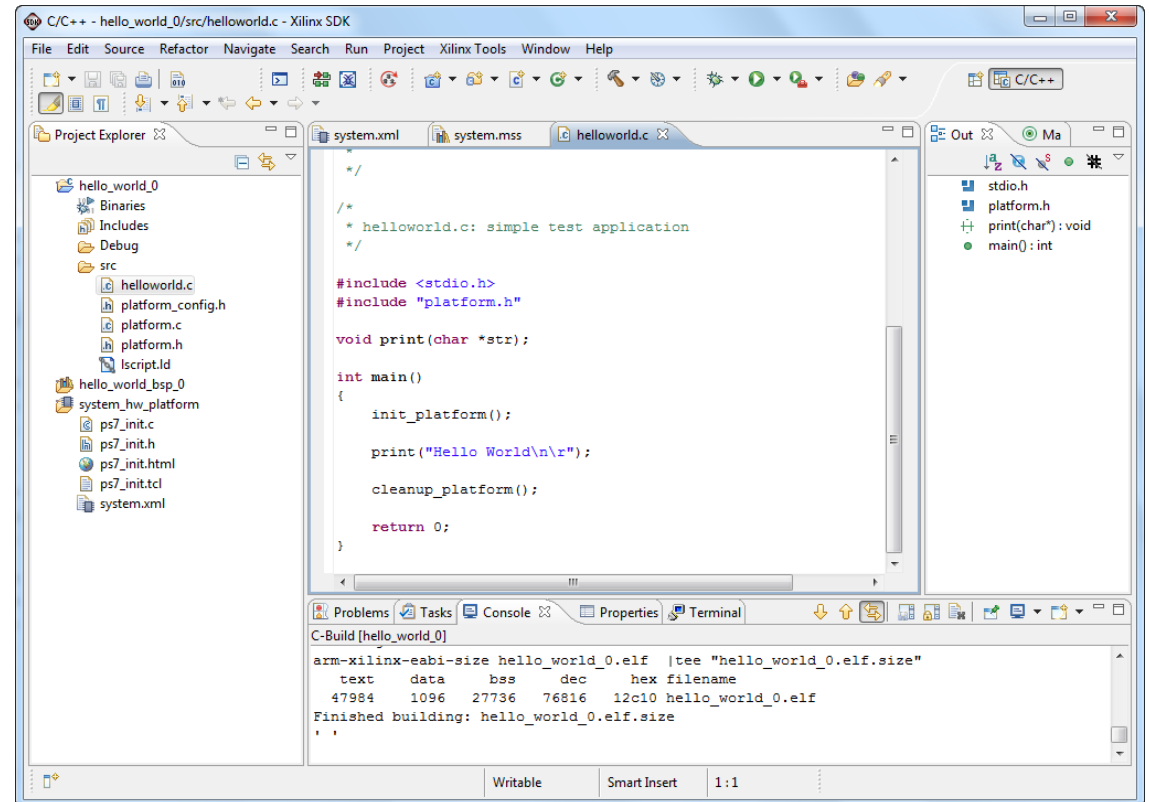


AMD FPGA Design Tools: Vitis



AMD FPGA Design Tools: Vitis

- Create hardware platform project
 - Automatically performed when Vitis is launched from Vivado
- Create BSP
 - Board support package
- Create software application
- Create linker script
- Build project
 - compile, assemble, link output file `<app_project>.elf`



AMD FPGA Design Flow: Vivado + Vitis

- Create a new **VIVADO** project
 - Invoke IP Integrator
 - Construct the hardware portion of the embedded design
 - Create top level HDL wrapper
 - Synthesize any non-embedded components
 - Export the hardware description
- Launch **VITIS** unified development environment
 - Create a new software board support package and application projects
 - Compile the software with the GNU cross-compiler
 - Download the programmable logic's completed bitstream to FPGA
 - Download the program (ELF) to FPGA

RE-CAP OF TODAY'S CLASS

- Introduction to Verilog HDL (Focus)
- Overview on SystemVerilog HDL/HVL
- Overview on System-Level Modeling with SystemC
- Introduction to FPGA Design Flow

