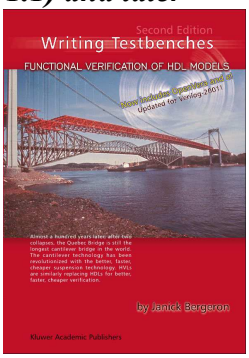
HDL projects verification

Agenda

- Verification definition
- Types of verification
- HDL Testbench template
- AHDL TB & ALINT CODE COVERAGE
- Simulation, emulation, prototyping
- Simulation acceleration

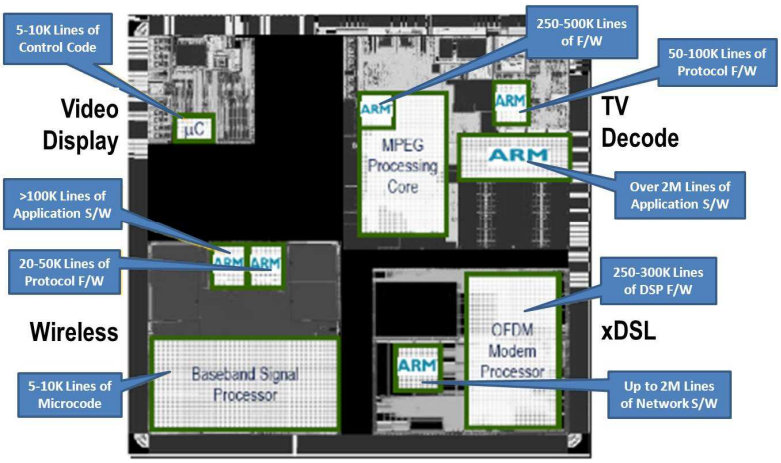
References

- *Writing Testbenches: Functional Verification of HDL Models, 2nd edition*
- *Writing Efficient Testbenches Xilinx XAPP199 (v1.1) and later editions*
- *Synthesis and Simulation Design Guide Xilinx*



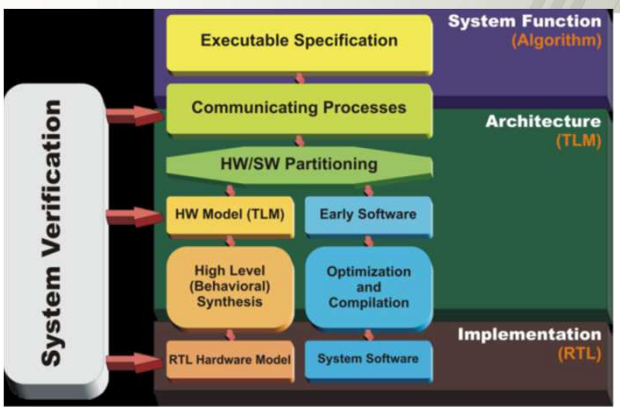
- *http://www.accellera.org/*
- *http://verificationacademy.com*
- *http://www.dvcon.org/*
- *http://www.deepchip.com/*

Jim Hogan of Vista Ventures LLC
z http://www.deepchip.com



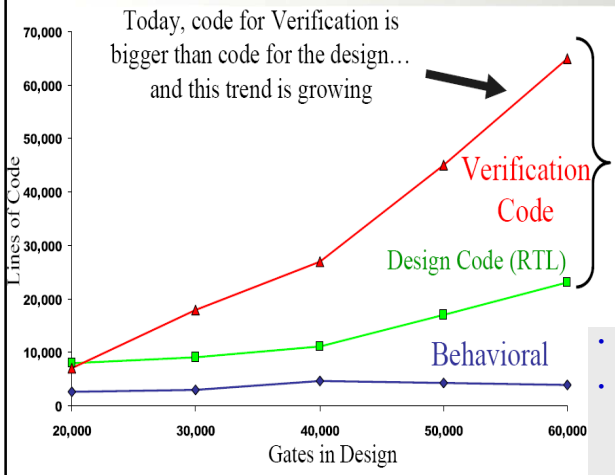
Typical SoC - dozens of HW blocks but millions of lines of code

Project verification



Free On-Line Dictionary Of Computing (foldoc.org):
Verification: The process of determining whether or not the products of a given phase in the life-cycle fulfill a set of established requirements. Let's agree on one definition suitable for this paper*: Verification (or Functional Verification) is the process of checking if the logic design at given stage of development conforms to the design specification.
***Meeting Growing Verification Demands ALDEC**

FPGA projects verification



A Widening Gap Caused by:

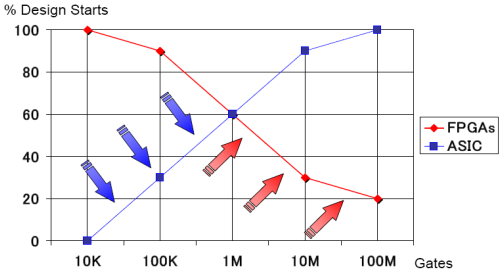
- Bigger designs
- More Interfaces
- More complexity
- Shorter schedules

- Verification takes up to 70% project time („critical path")
- Verification requires up to twice resources more then the source code development (separate teams usually)
- Up to 80% the final project code is the testbench

FPGA projects verification

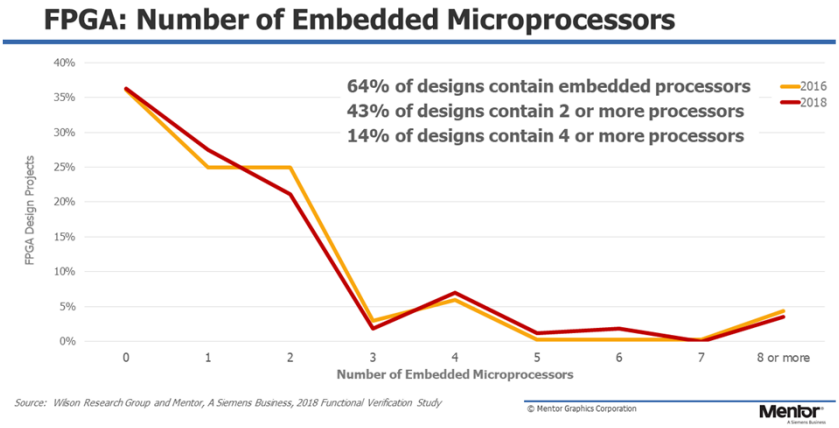
FGPA Complexity On The Rise

- As technology has advanced to sub 0.1 microns, FPGA's now feature more SoC like functionality previously targeted at ASICs…



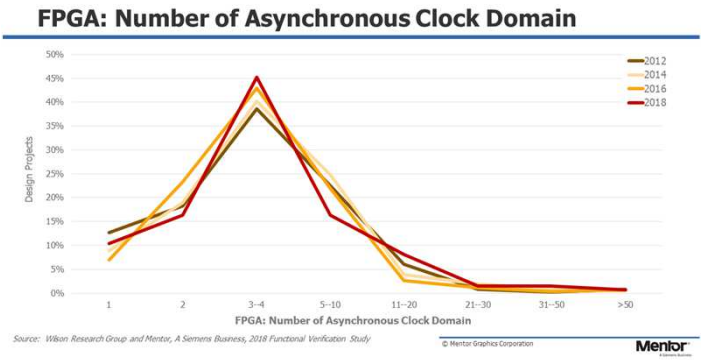
- Gate count capacity has increased
- More device features are available
- FPGA device complexity has increased

ZA:**Mentor Verifications Horizonts Blog The 2018 Wilson Research Group Functional Verification Study**



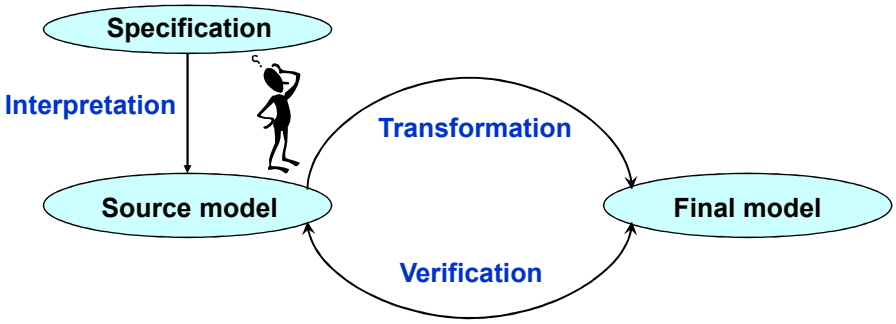
For example, our study found that 64% of all projects targeted their design at an FPGA containing one or more embedded processors, as shown above

ZA: Mentor Verifications Horizonts Blog
The 2018 Wilson Research Group Functional Verification Study



90% of designs being implemented as FPGAs contain two or more asynchronous clock domains. Verifying requirements associated with multiple asynchronous clock domains has increased both the verification workload and complexity. For example, a class of metastability bugs cannot be demonstrated on an RTL model using simulation. To simulate these issues requires a gate-level model with timing, which is often not available until later stages in the design flow. However, static clock-domain crossing (CDC) verification tools have emerged and are being adopted to help identify clock domain issues directly on an RTL model at earlier stages in the design flow.

Verification



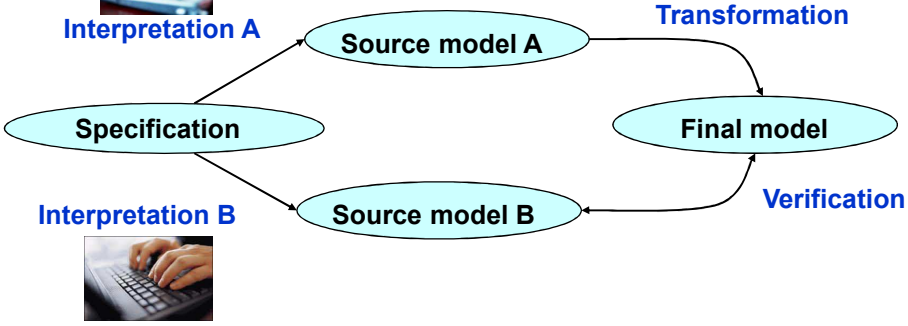
Transformation : RTL coding, synthesis, implementation

Verification requires reference !!!

Watch out ! *„human factor”* –
when the same developer performs both coding and verification

Verification concepts

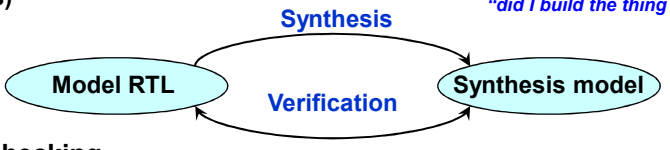
- Automatic code creation (golden templates...)
- Problem „atomisation”: diminish human factor
- Task redundation : two independent teams



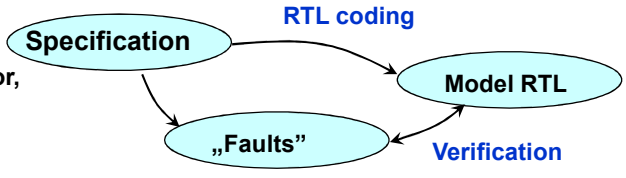
Verification strategies (functional/implementation)

Functional verification answers the question: “did I build the right thing?” as opposed to implementation verification, which answers the question “did I build the thing right?”

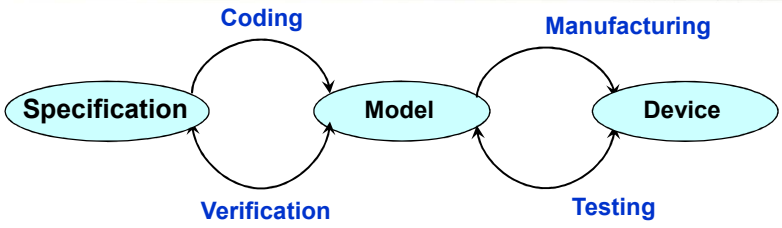
1. Comparison
   Netlist comparison (synthesis/implementation errors)



2. Model checking
   - Normal and abnormal situations behaviour
   - How to verify the abnormal situation? (FSM faults, bus error, protocol bugs...)



Verification and testing



Errors	Found	Not found
Wrong project		Typ II
Good project	Typ I	

Basic goals of any comprehensive verification process:
1. It must verify that the design does everything it is supposed to do.
2. It must verify that the design does not do anything it is not supposed to do.
3. It must show when the two goals have been met.

Verification plan

1. Proper and exact specification (when the final version fixed?)
2. First success plan (the most important functionality done)
3. Verification level definition:
   - *unit-level,*
   - *board-level,*
   - *system-level,*
4. Verification strategies (*black-box, white-box, grey-box, random)*
5. Traceability
6. Priorities (*must-have, should-have, nice-to-have*)
7. Easy verification (*design for verification)*
   (*loadable units, unit bypass, sample points, error injection mechanism ...*)

Functional verification - questions

What Documents?
- *Hardware Requirements Specification*
- *Commerical Bus/Component Specifications*
- *Hardware Design Specification*
- *Verification Environment Specification*
- *Test Plan*
- *ISO 9000 Process Flow*
- *Error Logs*
- *Status Reports*

What Lanaguage for Verification?
- *Specman (HVL)*
- *Vera (HVL)*
- *System Verilog*
- *SystemC with SCV (SystemC Verification Library)*
- *TestBuilder (C++)*
- *Custom C++*
- *Verilog*
- *VHDL*

Follow
http://www.project-veripage.com/

Functional verification - questions

What HDL design Language?
- *Verilog*
- *VHDL*
- *Mixed Verilog/VHDL*

What HDL level?
- *Block (unit)*
- *Device*
- *System*

What test visibility?
- *Black box*
- *White box*
- *Gray box*

Type of test?
- *Directed*
- *Random*
- *Directed Random*

What data level?
- *Vector or bit*
- *Transaction*

Data creation?
- *Manual*
- *Random Generation (Pre-run)*
- *Random Generation (On-the-fly)*

Testbench checking?
- *Manual*
- *Golden model*
- *Self-checking (Post-run)*
- *Self-checking (On-the-fly)*
- *Assertion Checking*

Are we done?
- *Code coverage*
- *Functional coverage*

Other Considerations?
- *Source Control*
- *DUT Performance*
- *Code Reuse*
- *Regression Testing*
- *Load Sharing*
- *Simulation performance*
- *Acceleration*
- *Gate Level*

Rajda & Kasperek © 2018 Katedra Elektroniki AGH

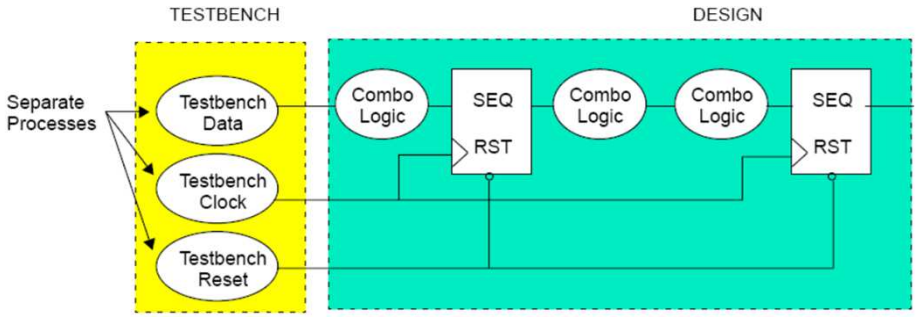
Functional verification management

System version control
Verification events handling:
- errors detected,
- specification gaps,
- code refactoring, code optimisation like *area/speed*,
- new ideas concepts,

Servicing:
- lets talk ... (in minor companies it is enough....☺)
  problems – no clear responsibility, no records ...
- procedures,
- data bases.

Any method chosen must be effective!!!
Active correction time must be shorter then correction management time

First things first...
Simple TESTBENCH – example



General Testbench Guidelines:
-Separate processes :
- -Data path,
- -System clock,
- -Asynchronous signals (like reset....)

TESTBENCH – example

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity counter_tb is
end counter_tb;
architecture TB_ARCHITECTURE of counter_tb is
        component counter -- Component declaration of the tested unit
        port(       CLK : in STD_LOGIC;
                    DATA : in STD_LOGIC_VECTOR(3 DOWNTO 0);
                    RESET : in STD_LOGIC;
                    LOAD : in STD_LOGIC;
                    Q : out STD_LOGIC_VECTOR(3 DOWNTO 0) );
    end component;
-- Stimulus signals - signals mapped to the input and inout ports of tested entity
    signal CLK : STD_LOGIC;
    signal DATA : STD_LOGIC_VECTOR(3 DOWNTO 0);
    signal RESET : STD_LOGIC;
    signal LOAD : STD_LOGIC;
-- Observed signals - signals mapped to the output ports of tested entity
    signal Q : STD_LOGIC_VECTOR(3 DOWNTO 0);
-- User can put declaration here
    shared variable ENDSIM: boolean:=false;
    constant CLK_PERIOD:time:= 30 ns;
    constant RESET_LENGTH:time:= 50 ns;
```

TB part 1

TESTBENCH – example

```
begin
-- Unit Under Test port map
    UUT : counter
        port map
                (CLK => CLK,
                DATA => DATA,
                RESET => RESET,
                LOAD => LOAD,
                Q => Q );
-- User can put stimulus here
    CLK_GEN: process
    begin
        if ENDSIM=false then
                CLK <= '0';
                wait for CLK_PERIOD/2;
                CLK <= '1';
                wait for CLK_PERIOD/2;
        else
                wait;
        end if;
    end process;
```

TB part 2

```
-- reset process
    RES: process
    begin   RESET<='0';
            wait for RESET_LENGTH;
            RESET<='1';
            wait;
    end process;
-- stimulus  process
STIM: process
    begin
            DATA<="0110";
            LOAD<='0';
            wait for 350 ns;
            LOAD<='1';
            wait for 50 ns;
            LOAD<='0';
            wait for 100 ns;
            ENDSIM:=true;
            wait;
    end process;

end TB_ARCHITECTURE;
```

TB part 3

TESTBENCH – WAIT instruction variants

```
junk : process
       begin
       CLK <= '0', '1' after 25 ns; -- not recomended
       wait for 50 ns;
end process;
```

Faster simulation when wait for is used

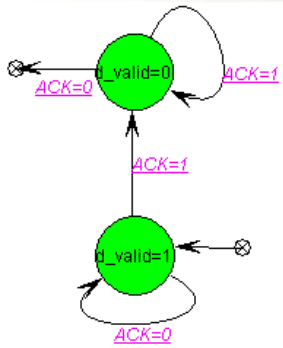
```
--Process with explicit "wait for time" statement
--This is a testbench process
process
begin
RESET <= '0';
wait for 50 ns;
RESET <= '1';
wait for 50 ns;
RESET <= '0';
wait;
end process;
```

```
--Process with explicit "wait on or sensitivity list"
--statement process
--Synthesizable (non-testbench) process style
begin
wait on WR;
DATA <= BUS_DATA;
end process;
```

```
--Process with explicit "wait until edge" statement
--This is a synthesizable (non-testbench) sequential
process
begin
wait until CLK = 1;
B <= A;
end process;
```

```
--Combination of "wait on, until, and or" statement
--Synthesizable (non-testbench) process style
wait on IN1 until CLK = '0';
```


TESTBENCH – behavioral coding advantage



TB is the right place for the effective behavioral VHDL coding

Example – „acknowledge process"

–When d_valid = 1 & ACK = 0
–When d_valid = 1 & ACK = 1
–When d_valid = 0 & ACK = 1
–When d_valid = 0 & ACK = 0

Behavioral VHDL coding advantages:
- no RTL coding limits (signal attribute 'event as many times as we want, wait for etc.)
- faster code development,
- shorter simulation.

TESTBENCH – behavioral coding advantage

```
comb: process (state, ACK)
begin
next_state <= state;
case state is
...
when MAKE_REQ=>
                d_valid <= '1';
                if ACK = '1' then
                next_state <= RELEASE;
                end if;
when RELEASE=>
                d_valid <= '0';
                if ACK = '0' then
                next_state <= ...;
                end if;
...
end;
SEQ: process (CLK)
begin
if CLK'event and CLK = '1' then
                if RESET = '1' then
                    state <= ....;
                else
                    state <=next_state;
                ......
end process SEQ;
```

```
process
begin
...
                d_valid <= '1';
                wait until ACK = '1';
                d_valid <= '0';
                wait until ACK = '0';
...
end process
```

Easy & fast ...

TESTBENCH – hints

```
--one clock  domain signals generation
divider: process
Begin
  clk50 <= '0';
  clk100 <= '0';
  clk200 <= '0';
  loop -- forever
      for j in 1 to 2 loop
            for k in 1 to 2 loop
               wait on clk;
               clk200 <= not clk200;
            end loop;
      clk100 <= not clk100;
     end loop;
     clk50 <= not clk50;
  end loop;
end process divider;
```

```
-- different clock domain signals generation
-- separate process
Clock_A : process
       begin
       CLK_A <= '0';
       wait for 200 ns;
       CLK <= '1';
       wait for 200 ns;
end process;

Clock_B : process
       begin
       CLK_B <= '0';
       wait for 33 ns;
       CLK_B <= '1';
       wait for 33 ns;
end process;
```

TESTBENCH – hints

```
decode: process
procedure do_instr(instr:t, data:d) is
begin
....
end do_instr;

begin
case I1 is
      when "0000" => do_instr(STOP, data);
      when "0001" => do_instr(JMP, data);
      when "0010" => do_instr(CALL, data);
      ...
end case;
```

Faster when external calls limited

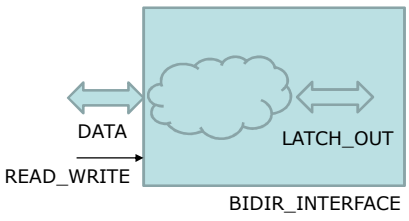
```
decode: process
procedure do_instr(instr:t, data:d) is
begin
....
end do_instr;

begin
case I1 is
      when "0000" => instr := STOP;
      when "0001" => instr := JMP;
      when "0010" => instr := CALL;
      ...
end case;
do_instr(instr, data);
...
```


TESTBENCH – bidirectional signals R/W

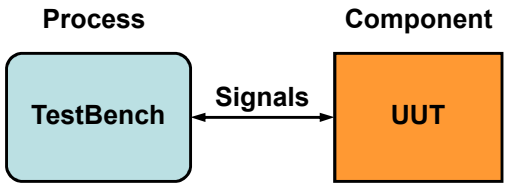
```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
entity bidir_interface is
port (          DATA : inout STD_LOGIC_VECTOR(1 downto 0);
      READ_WRITE : in STD_LOGIC);
end bidir_ interface ;

architecture XILINX of bidir_ interface is
signal LATCH_OUT : STD_LOGIC_VECTOR(1 downto 0);
begin
process(READ_WRITE, DATA)
      begin
      if (READ_WRITE = '1') then
            LATCH_OUT <= DATA;
      end if;
end process;
process(READ_WRITE, LATCH_OUT)
      begin
      if (READ_WRITE = '0') then  – write active
            DATA(0) <= LATCH_OUT(0) and LATCH_OUT(1);
            DATA(1) <= LATCH_OUT(0) or LATCH_OUT(1);
      else                    – read active
            DATA(0) <= 'Z';
            DATA(1) <= 'Z';
      end if;
end process;
end XILINX;
```



Bidirectional signals MUST be set to 'Z' when are read from TB side

TESTBENCH
– Process/Component connection



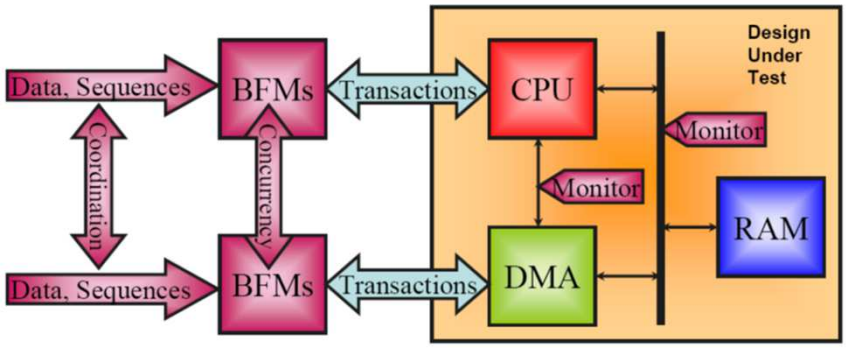
Verification stages

- ***Ad Hoc***
  First success plan
- ***Algorythm based***
  Simple or advanced but repeatable
- ***File system***
  Final solution : proces with stimulus files and results

Final solution: self checking Testbench : PASS or FAIL!!!

TESTBENCH – structural approach

- *BFMs to control & monitor I/O transactions*
- *Externally generated data and expected results*
- *Concurrent, coordinated system process modeling*
- *Internal transaction and bus monitors*



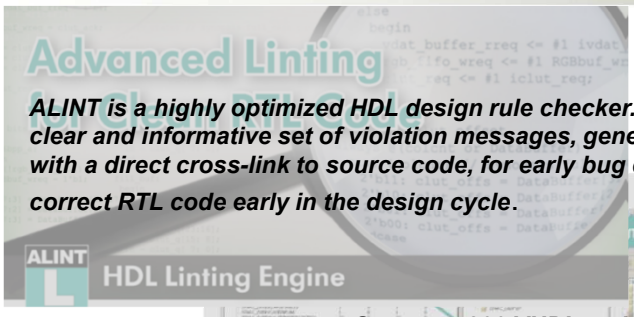
Active HDL –TB support

- **TB templates**

- **clock generation,**
- **signals assigments:**
  - ***Concurrent mode,***
  - ***Sequenced mode,***
- **automatic waveform comparison,**
- **logging,**
- **random stimuli,**

- **Standard Waveform and Vector Exchang WAVES (IEEE-STD-1029.1)**
- ***Code Coverage***
- ***Linting***

Formal verification - *Linting*



ALINT is a highly optimized HDL design rule checker. ALINT includes a clear and informative set of violation messages, generated during linting with a direct cross-link to source code, for early bug detection, ensuring correct RTL code early in the design cycle.

Key Features

- ***Supports 200 VHDL and Verilog Design Rules.***
- ***Clock Domain Crossing (CDC) support.***
- ***Source code checks, design elaboration and synthesis emulation.***
- ***User Modified Design Rules.***
- ***Cross-Probing of error messages, Violation Viewer.***
- ***Configuration Management***

Formal verification with ALINT - rules

LINT_3003: Memory '%s' is read and written at the same time
Sample Code:

```
process (clk, address, data, rw)
  type ramtype is array(natural range <>) of std_logic_vector(7 downto 0);
  variable ram : ramtype(15 downto 0);
begin
  if (rising_edge(clk)) then
    if (rw = '1') then
      ram(address) := data;
    end if;
    data <= ram(address);
  end if;
end process;
```

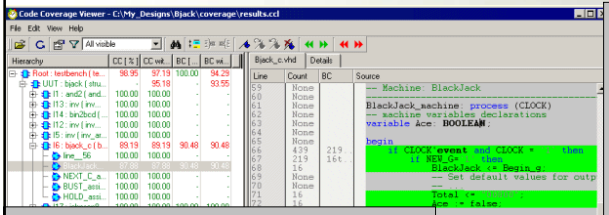
LINT_3001
Incomplete sensitivity list
Sample Code:

```
process (clk)
begin
  if (reset = '1') then
    sig <= '0';
  elsif (rising_edge(clk)) then
    sig <= not clk;
  end if;
end process;
```

LINT_5009 Reset signal '%s' is active high and low
Sample Code:

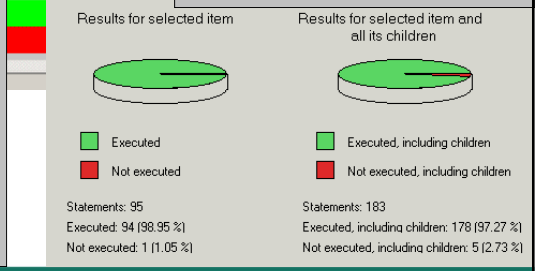
```
architecture tb of tb is
  signal a, b : std_logic;
begin
  process (reset)
  begin
    if (reset = '1') then
      a <= '0';
    end if;
  end process;
  process (reset)
  begin
    if (reset = '0') then
      b <= '0';
    end if;
  end process;
end architecture tb;
```


AHDL – Code Coverage



•Code Coverage is integrated into the Active-HDL simulation kernel.
•Code Coverage can measure the effectiveness of testbenches so the most effective test can be run first. This helps to uncover bugs in the design verification process immediately during long regression tests.

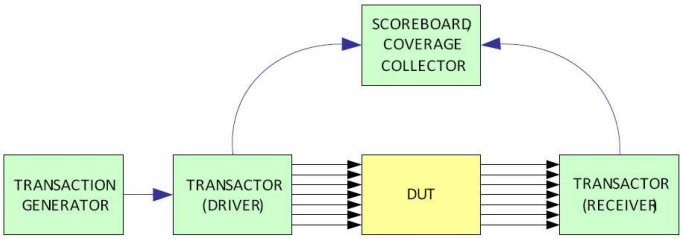
Code Coverage provides the following benefits to the designer:
•The users can easily find sections of a model that have not been exercised by a testbench. It allows a modification of the testbench to cover all untested parts of the design.
•Code Coverage helps to identify sections of the model executed very frequently. This allows the user to optimize the execution of the model during the simulation.



Verification methodologies
*** ALDEC**

Verification methodologies

- UVM, OVM, VMM, AVM, RVM...



UVM, OVM - Universal/Open Verification Methodologies
(VMM) Verification Methodology Manual (VMM)
(AVM) Advanced Verification Methodology
System Verilog SystemC itp.
(OSVVM) Open Source VHDL Verification Methodology http://osvvm.org/

Open Source VHDL Verification Methodology (OSVVM) http://osvvm.org/

Links

This page contains links to other sources of information that may be helpful to OS-VVM and VHDL users.
We can place link to your website here if it has ties to VHDL and you are willing to reciprocate (place link to us on your website).
Contact Us form is located at the bottom of the page.

Founders of OS-VVM

Aldec, Inc.

Aldec is an industry-leading Electronic Design Automation (EDA) company delivering innovative design creation, simulation and verification solutions to assist in the development of complex FPGA, ASIC, SoC and embedded system designs.

With an active user community of over 35,000, 50+ global partners, offices worldwide and a global sales distribution network in over 43 countries, the company has established itself as a proven leader within the verification design community.

SynthWorks Design, Inc.

SynthWorks provides training in leading edge VHDL verification techniques, including transaction based testing, bus functional

Open Source VHDL Verification Methodology (OSVVM)
http://osvvm.org/

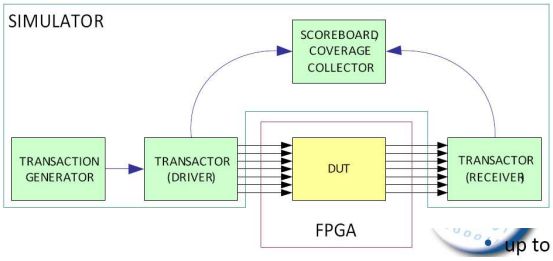
SynthWorks

OSVVM Benefits

- OSVVM provides advanced verification capabilities
  - Functional Coverage
  - Random: Intelligent and Constrained
  - Error Reporting: Alerts & Logs
  - Memory Models
  - Transaction Level Modeling
  - Scoreboards
- Intelligent Coverage = Simple, Powerful, Concise Methodology
  - Define Functional Coverage
  - Randomize across coverage holes
  - Refine with directed, algorithmic, file-based or CR methods
- OSVVM is for the VHDL Community
  - Works with your existing VHDL Testbench
  - Re-use or upgrade your current VHDL testbench models
  - Simple, Concise, Readable

Verification methodologies
*** ALDEC**

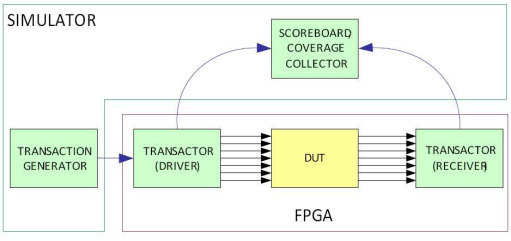
- 10x faster: **100 kHz**



Simulation acceleration
DUT in FPGA

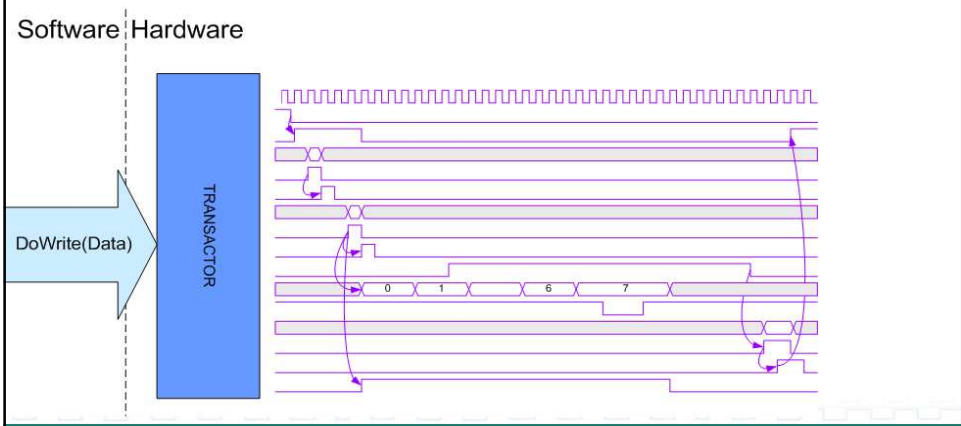
- up to 1000x faster than simulation: **10 Mhz**

Simulation acceleration
*DUT **and TB** in FPGA*

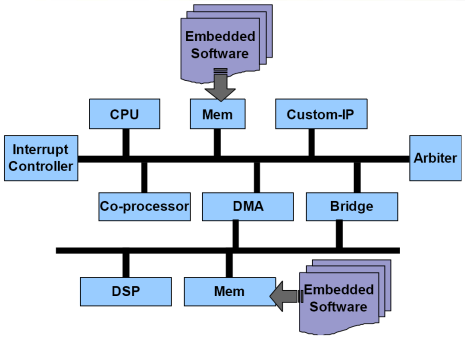


Verification methodologies
*** ALDEC**

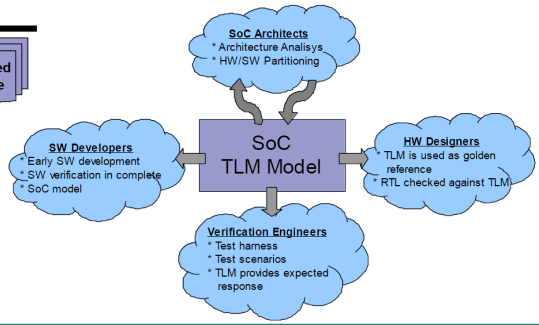
- **Synthesizable transactor** translates function calls into sequence of bits
- **Eliminates** HW-SW communication **bottleneck**



SOC Verification
ALDEC

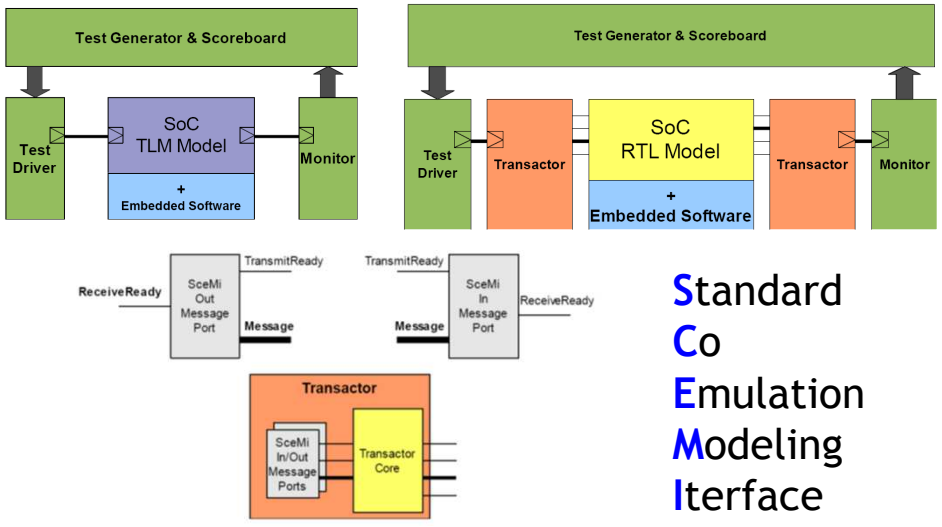


Transaction level system modeling



SOC Verification

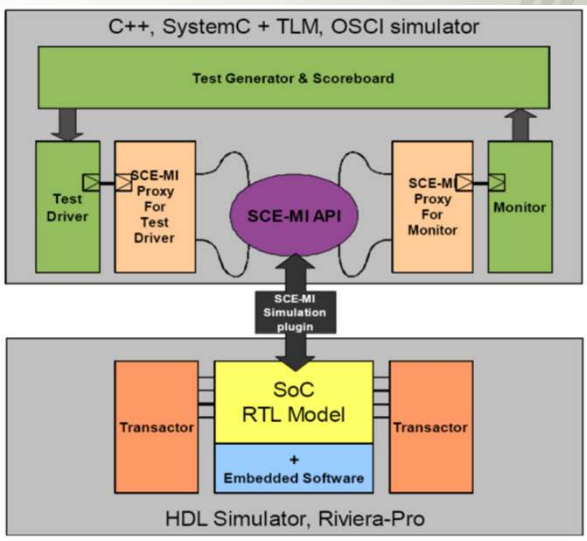
ALDEC



Standard
Co
Emulation
Modeling
Iterface

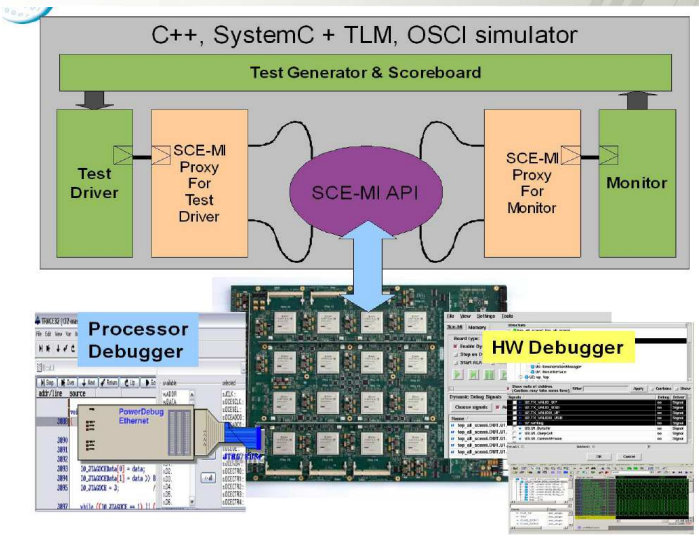
SOC Verification

ALDEC



SOC Verification

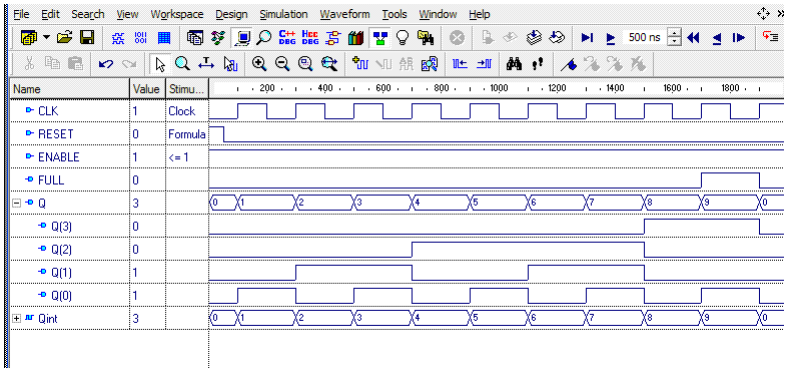
ALDEC



Hardware design and verification

* SIMULATION ACCELERATION AND EMULATION SUCCESS JASON ANDREWS, CADENCE DESIGN SYSTEMS

Logic simulation



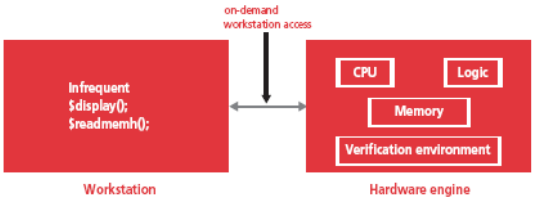
Software simulation refers to an event-driven logic simulator that operates by propagating input changes through a design until a steady-state condition is reached. Software simulators run on workstations and use languages such as Verilog®,VHDL, SystemC, SystemVerilog, and to describe the design and verification enviroment

Hardware design and verification

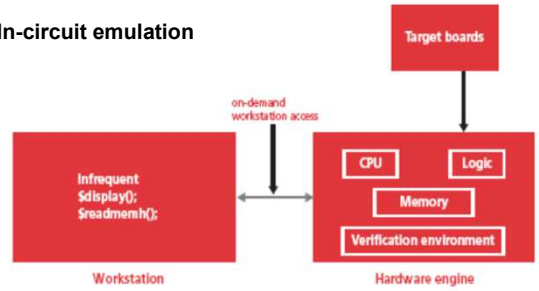
** SIMULATION ACCELERATION AND EMULATION SUCCESS JASON ANDREWS, CADENCE DESIGN SYSTEMS*

• Emulation

Emulation refers to the process of mapping an entire design into a hardware platform designed to further increase performance. There is no constant connection to the workstation during execution, and the hardware platform receives no input from the workstation



In-circuit emulation



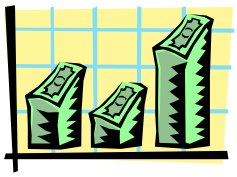
Hardware design and verification

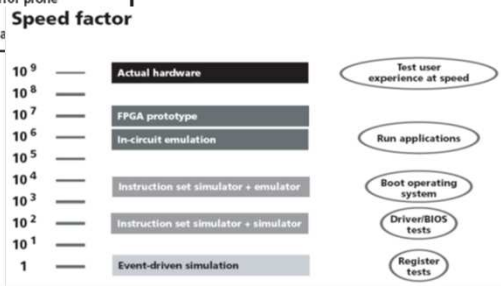
• Prototyping

refers to the construction of custom hardware or the use of reusable hardware (breadboard) to construct a hardware representation of the system. A prototype is a representation of the final system that can be constructed faster and made available sooner than the actual product. This speed is achieved by making tradeoffs in product requirements such as performance and packaging. A common path to a prototype is to save time by substituting programmable logic for ASICs. Since prototypes are usually built using FPGAs, they are often confused with and compared to emulation systems that also use FPGA technology.

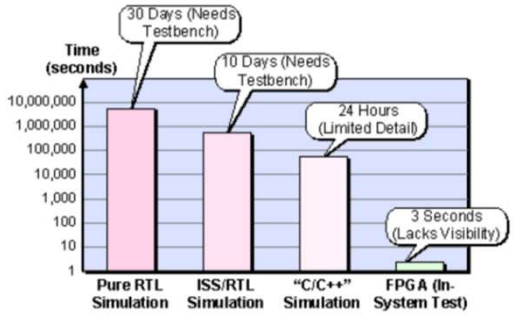
Hardware design and verification

Logic simulation	Acceleration	In-circuit emulation	FPGA prototype
Accurate	Faster	Fast enough for embedded software	Inexpensive
Flexible	Handles large designs	Real world stimulus	Fast enough for embedded software
Not fast enough for large designs	Debug via simulator user interface	Short implementation time from RT-level	Long implementation time
Not fast enough to test embedded software	Easier to get working	Slower than FPGA prototype	Little, if any, debug capability
	Not fast enough for embedded software	Higher cost than FPGA prototype	Error prone





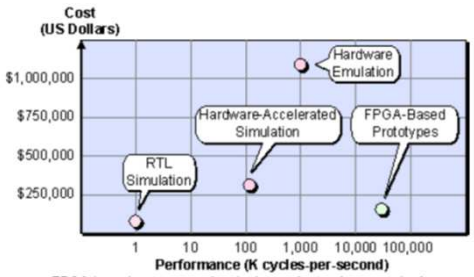
Hardware design and verification



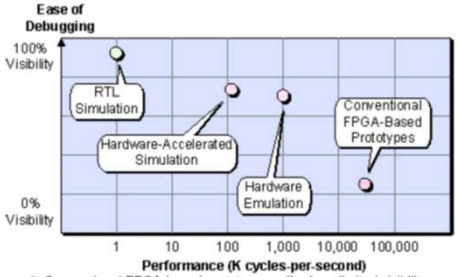
1. FPGA-based prototypes offer an extreme performance advantage over various software simulation techniques.

This particular example involves the booting of a real-world cell phone designIn addition to requiring a testbench, even a high-capacity, high-performance RTL simulator took 30 days to boot the system. Similarly, a traditional hardware/software co-verification environment using an instruction set simulator (ISS) – which also required a testbench – took 10 days to boot the system. Meanwhile, a C/C++ simulation of the system brought the boot time down to 24 hours, but this form of verification provided only limited visibility into the internal workings of the system.

Hardware design and verification



2. FPGA-based prototypes give the best price/performance by far.

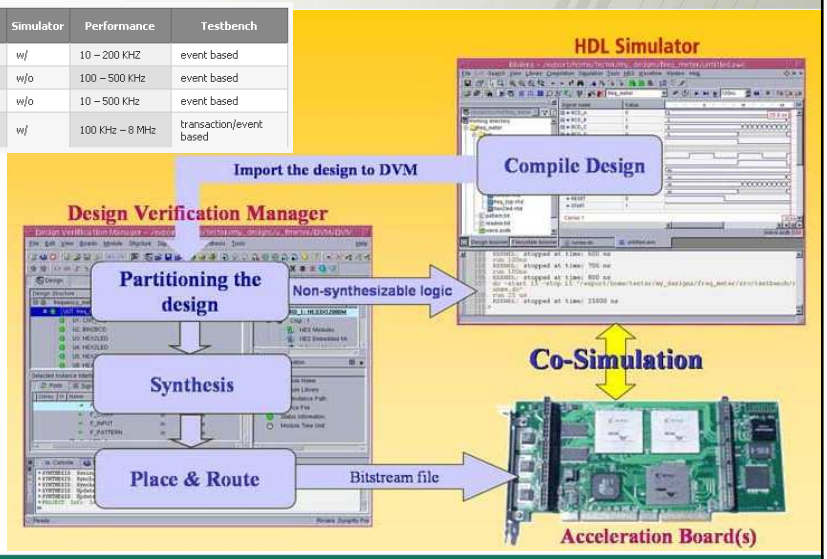


3. Conventional FPGA-based prototypes suffer from limited visibility, which makes them difficult and time-consuming to debug.

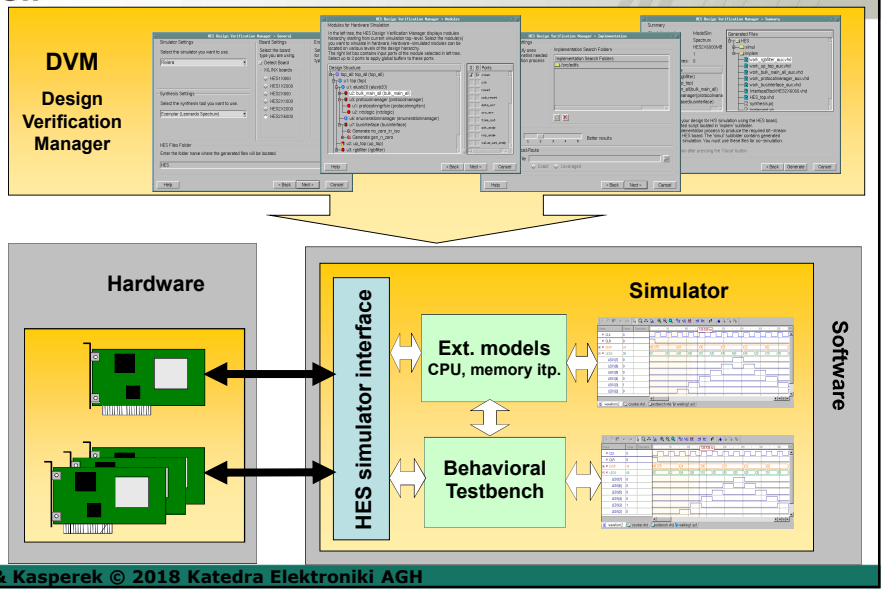
By comparison, an in-system FPGA booted the system in only three seconds. This means that the FPGA-based environment can be used to verify the system running under real-time workloads; also that this environment can be used as a platform for embedded and application software developers to integrate and verify their code in the context of the real system. The main problem with the FPGA – when used in a traditional verification environment – is lack of visibility with regard to its internal signals and state, including the contents of any memories.

Aldec - HES

Verification mode	Simulator	Performance	Testbench
Hardware acceleration	w/	10 – 200 KHZ	event based
Vector Based Testbench	w/o	100 – 500 KHz	event based
SystemC co-simulation	w/o	10 – 500 KHz	event based
HW/SW co-verification of ARM systems	w/	100 KHz – 8 MHz	transaction/event based

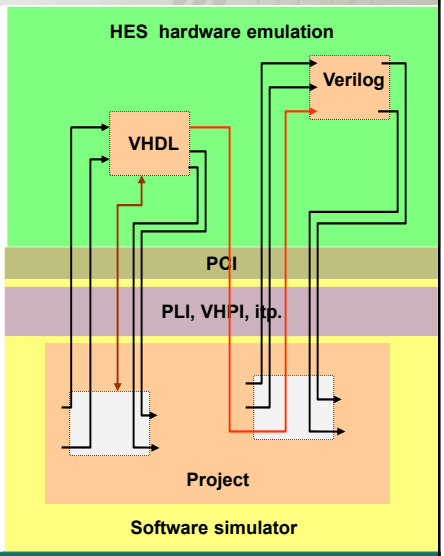


HES simulation environment

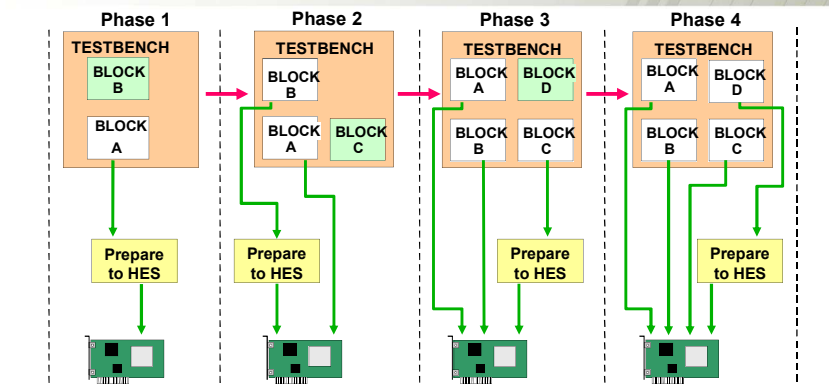


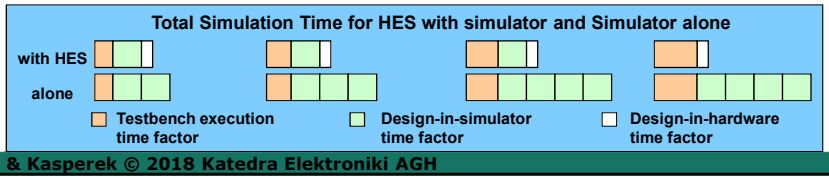
HES – hardware – software cosimulation

- ✓ Hardware modules on HES board are connected with simulator via PCI interface.
- ✓ Logic modules connections are made by software links (black lines on a picture).
- ✓ Logic inter modules connections are made also by software links (red lines on a picture).
- ✓ All kind connection are simulated: inputs, outputs, 3-state and bidirectional.

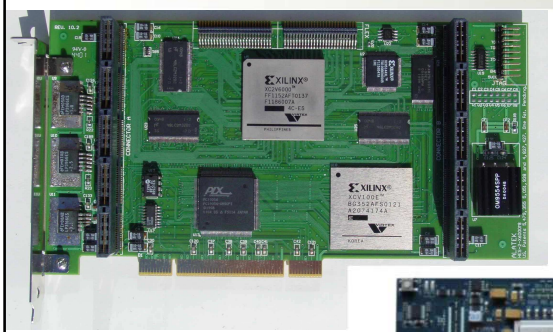


HES *Incremental development*



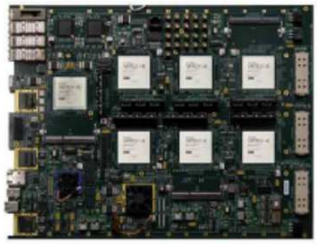


HES



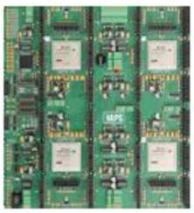
- Xilinx
  - Virtex V800
  - Virtex 2xV2000
- Altera
  - Apex 1000
- Inne układy FPGA

HES



DINI Group
DN Boards
capacity up to:
37M ASIC gates

- Third party boards



Synopsys
HAPS-54
capacity up to:
8M ASIC gates

ALDEC
HES5
capacity up to:
5M ASIC gates