

Wybrane zagadnienia weryfikacji. Testbench

- **Co to jest weryfikacja?**
- **Rodzaje weryfikacji**
- **HDL *Testbench***
- **AHDL TB & LINT & WAVES**

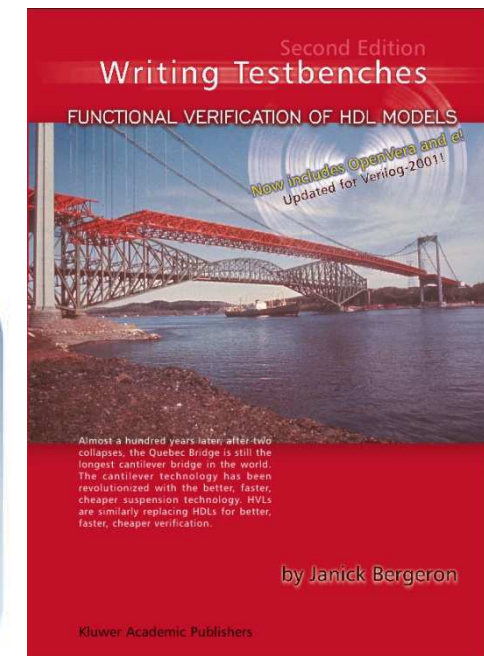
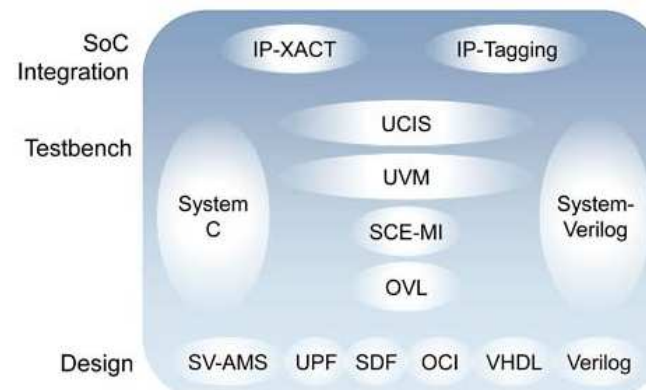
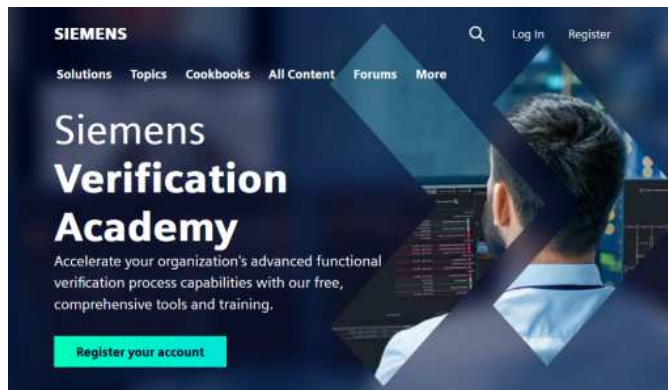


Accellera Systems Initiative to niezależna organizacja non-profit, której celem jest tworzenie, wspieranie, promowanie i rozwijanie standardów projektowania, modelowania i weryfikacji na poziomie systemowym do wykorzystania w światowym przemyśle elektronicznym.

Dzięki stałemu partnerstwu z IEEE, normy i rozwiązania techniczne opracowane przez Accellera Systems Initiative są przekazywane do IEEE w celu formalnej standaryzacji i bieżącego zarządzania.

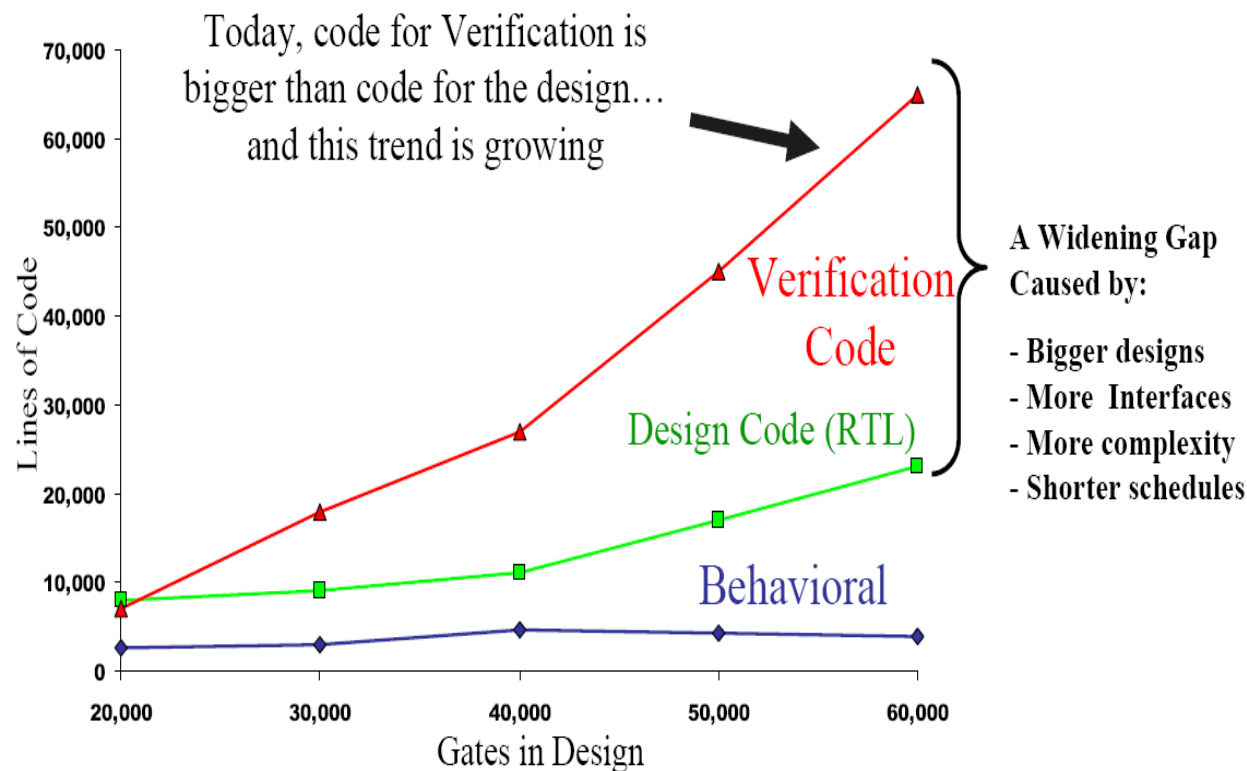
VHDL, SystemVerilog, SystemC, SDF, UVM, TLM, SCE-MI, OVL, PSL...

- ***Writing Testbenches: Functional Verification of HDL Models, 2nd edition***
- ***Writing Efficient Testbenches Xilinx XAPP199***
- ***VHDL Simulation Coding and Modeling Style Guide, Synopsis***
- <http://www.accellera.org/>
- <https://verificationacademy.com/>



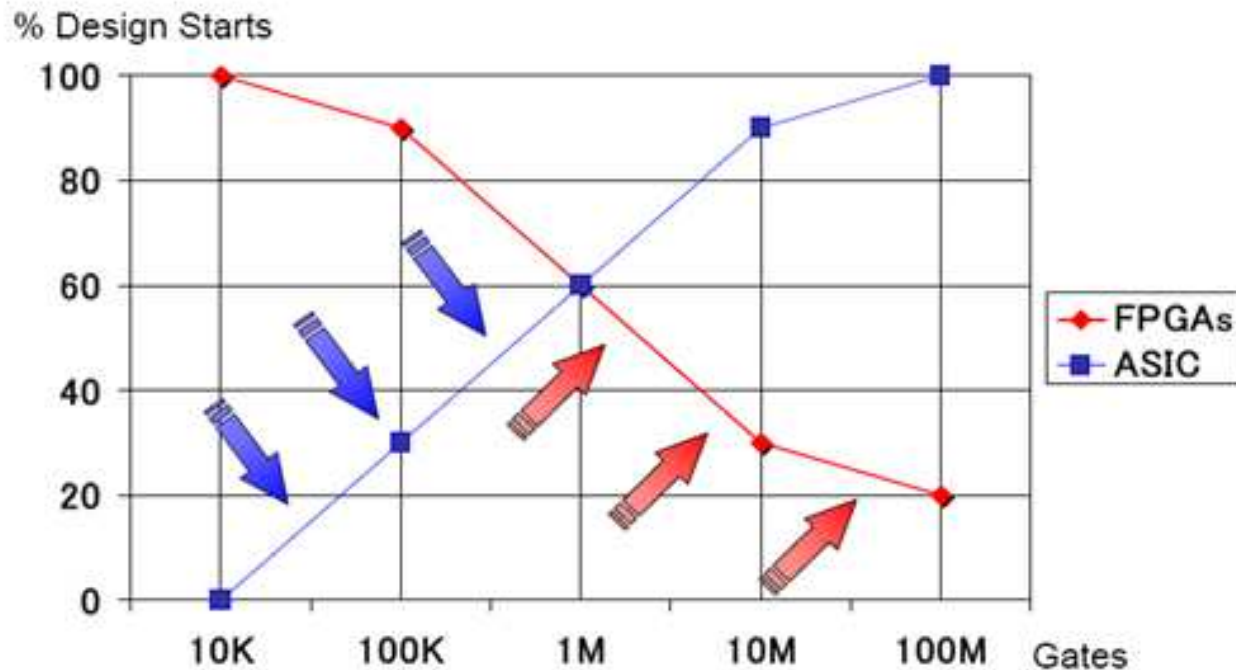
Weryfikacja projektów

- Weryfikacja zajmuje 70% czasu realizacji projektu („critical path”)
- Zagadnieniami weryfikacji zajmuje się dwa razy więcej inżynierów niż wprowadzaniem projektu (oddzielne ekipy !)
- Kod *testbencha* stanowi do 80% całego projektu



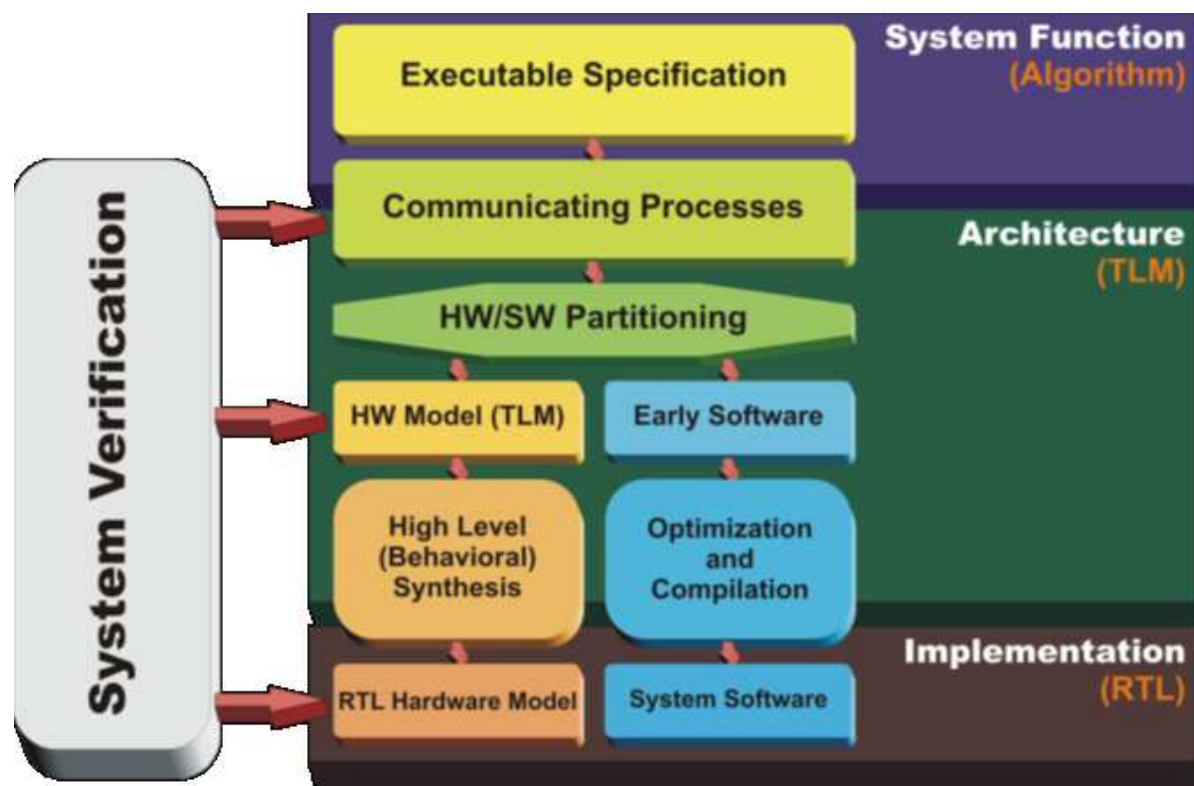
Rosnąca złożoność projektów FPGA

- 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

Weryfikacja projektów

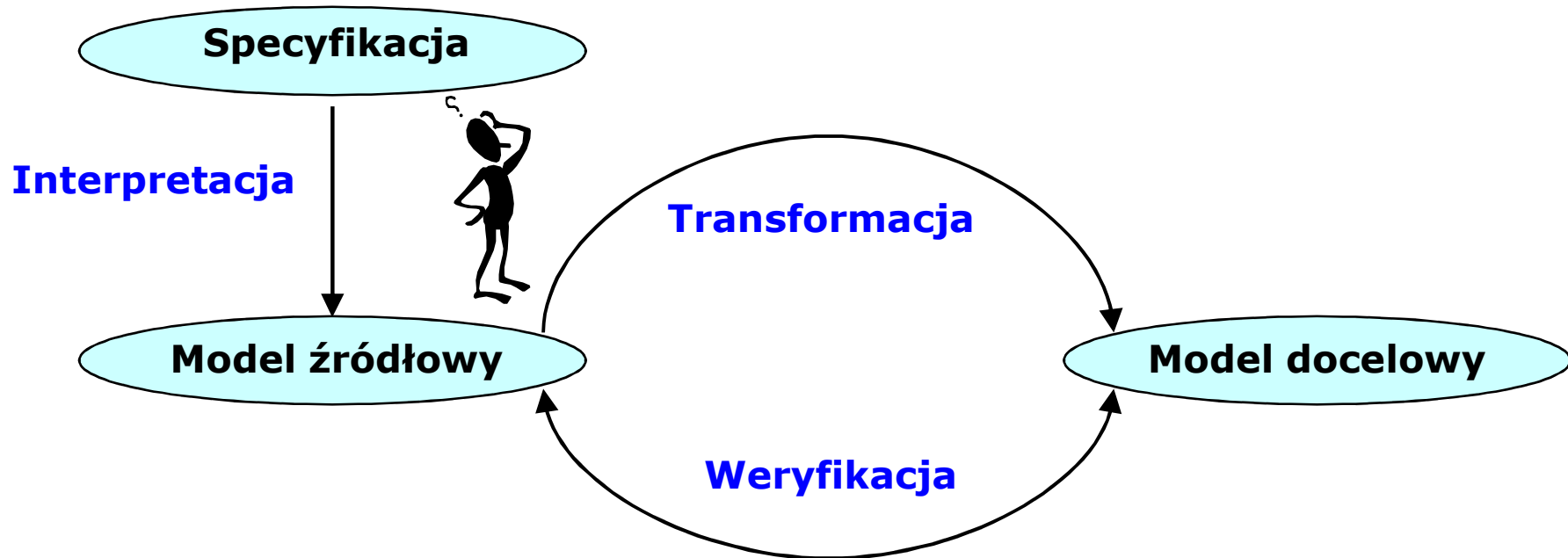


TLM to podejście w automatyzacji projektowania elektronicznego (EDA) do modelowania złożonych systemów cyfrowych, takich jak systemy SoC. Koncentruje się ono na transferach danych, czyli „transakcjach” między blokami funkcjonalnymi, a nie na szczegółowej implementacji samych bloków, co jest przydatne w analizie na poziomie systemu.

RTL to metoda projektowania wykorzystująca języki opisu sprzętu (HDL), takie jak VHDL lub Verilog, do określania sposobu, w jaki dane przemieszczają się pomiędzy rejestrami sprzętowymi w układzie cyfrowym.

Weryfikacja (ogólna): Proces mający na celu ustalenie, czy produkty danej fazy cyklu życia spełniają zbiór ustalonych wymagań.

Weryfikacja (weryfikacja funkcjonalna) to proces sprawdzania, czy projekt logiki na danym etapie rozwoju jest zgodny ze specyfikacją projektu.



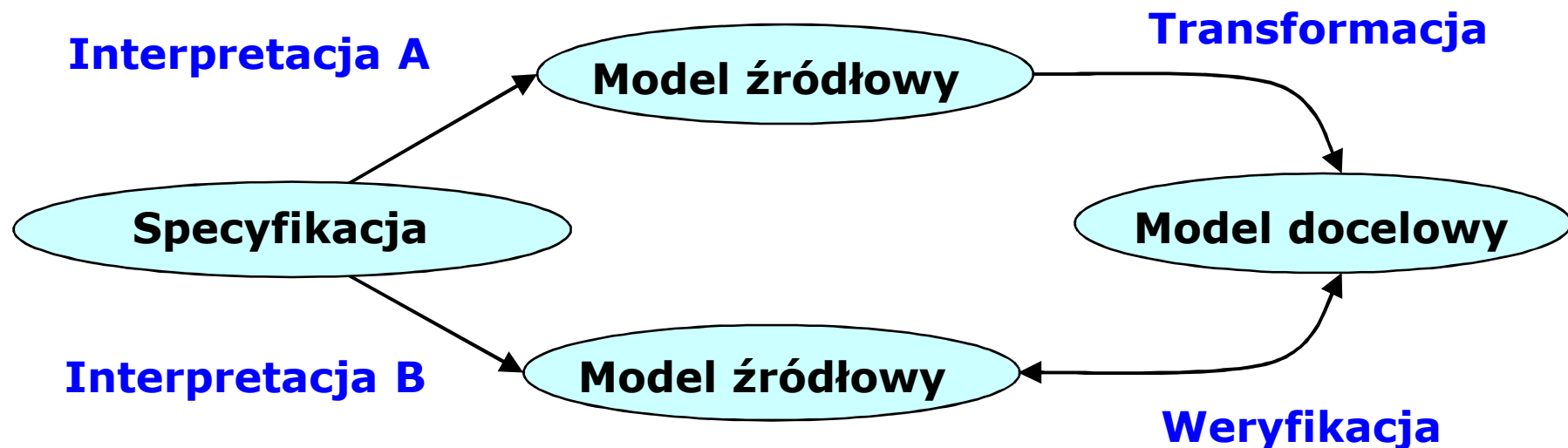
Transformacja to np.: kodowanie RTL, synteza, implementacja...

Weryfikacja wymaga wspólnego początku z transformacją.

Uwaga na czynnik ludzki, jeżeli ten sam projektant przeprowadza interpretację, kodowanie i weryfikację!

Weryfikacja – problem z interpretacją

- Automatyzacja (pomysł OK, ale jak go zrealizować ?)
- Strategia atomizacji problemu: sprowadzenie ludzkich decyzji do elementarnych problemów
- Wprowadzenie redundancji: dwa konkurencyjne zespoły

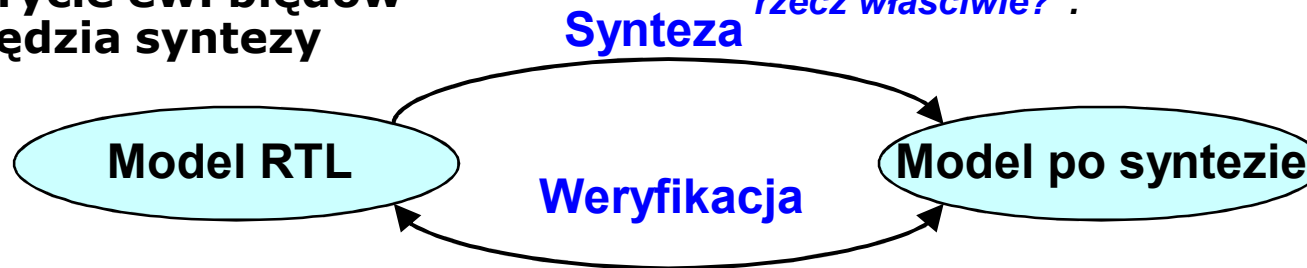


Weryfikacja – strategie...

Porównanie zgodności

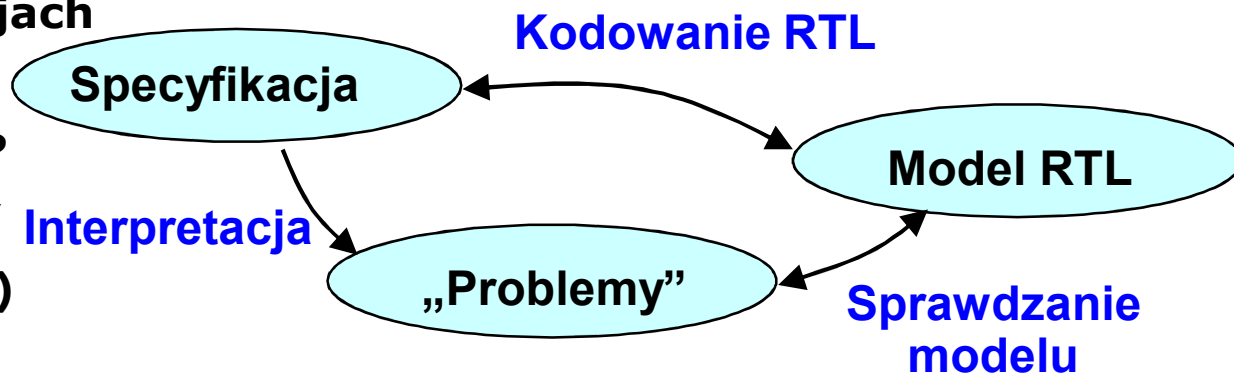
- Porównanie dwóch list połączeń (*netlist*)
- Wykrycie ew. błędów narzędzia syntezy

Weryfikacja funkcjonalna odpowiada na pytanie: „*czy zbudowałem właściwą rzecz?*”, w przeciwieństwie do weryfikacji implementacji, która odpowiada na pytanie: „*czy zbudowałem rzecz właściwie?*”.

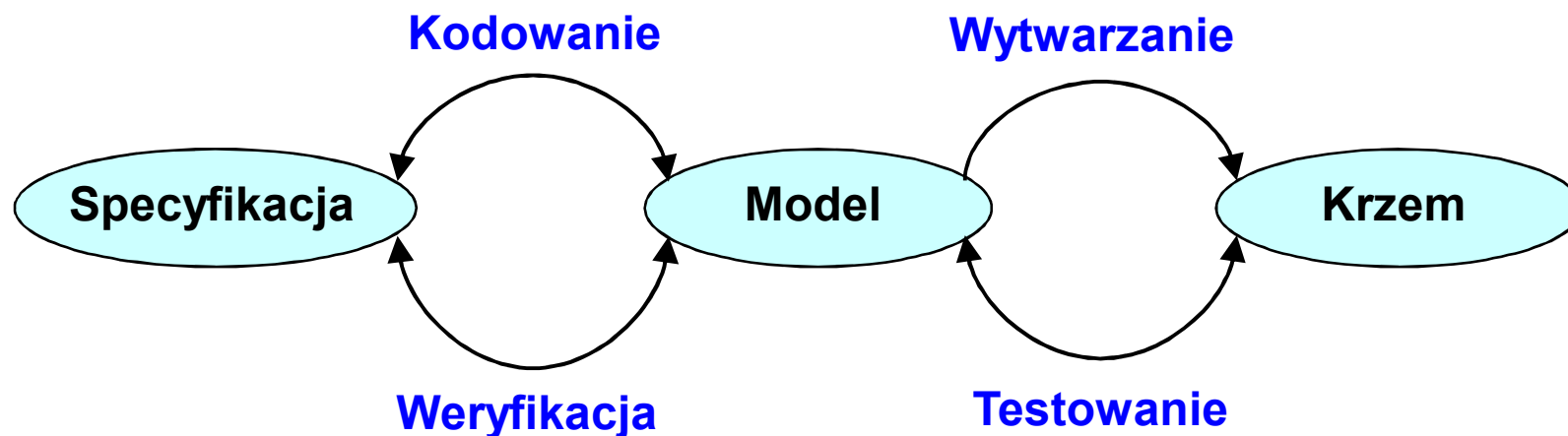


Sprawdzanie modelu

- Sprawdzenie zachowania się modelu w sytuacjach błędnych
- Problem: jak te sytuacje znaleźć? (automaty stanu, inne zachowanie się magistral itp.)



Weryfikacja a testowanie



Pomyłki	Błędy	Brak błędów
Zły Projekt		Typ II
Dobry Projekt	Typ I	



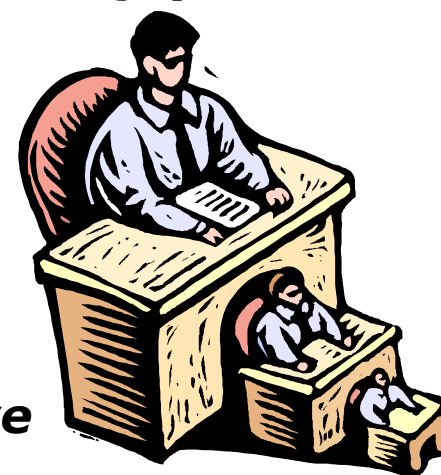
Basic goals of any comprehensive verification process:

- 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 first two goals have been met.**

Plan weryfikacji

Współczesne realia wymagają definiowania czasu realizacji poszczególnych etapów projektu (w tym czasu weryfikacji).

- 1. Odpowiednia i dokładna specyfikacja (kiedy koniec?)**
- 2. Plan pierwszego sukcesu (tylko najważniejsze funkcje)**
- 3. Definicja poziomu weryfikacji:**
unit-level, system-level, board-level
- 4. Strategie weryfikacji:**
black-box, white-box, grey-box, losowa
- 5. Identyfikacja weryfikowanych funkcji**
- 6. Priorytety: *must-have, should-have, nice-to-have***
- 7. Projektowanie „łatwo weryfikowalne” (*design for verification*), np.: elementy ładowalne, wprowadzanie modułów w tryb przeźroczysty (*unit bypass*), punkty weryfikacji (*sample points*), *error injection mechanism* itp.**



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

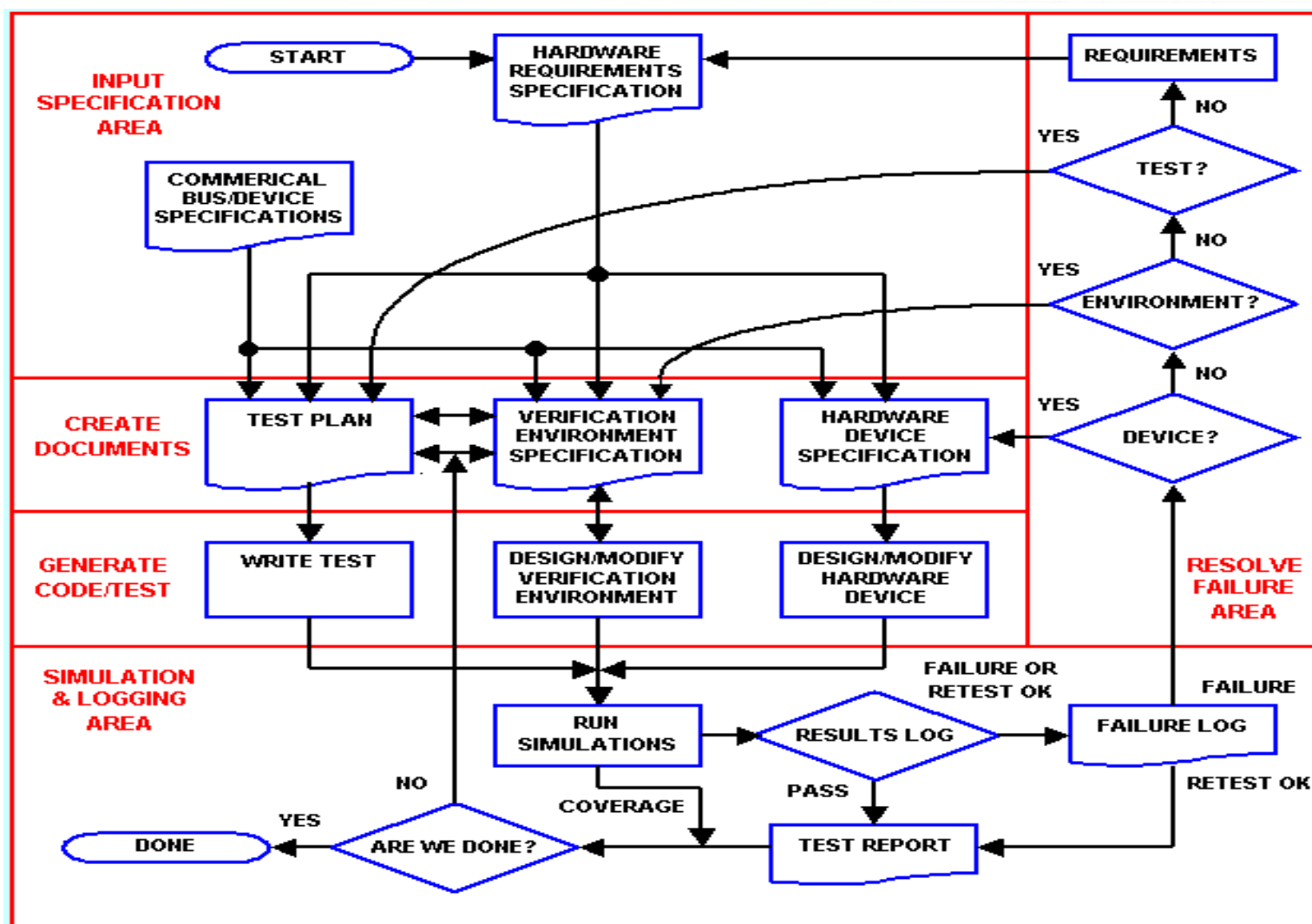
What Documents?

- 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 (UVM)*
- SystemC with SCV (SystemC Verification Library)*
- TestBuilder (C++)*
- Custom C++*
- Verilog*
- VHDL*

Weryfikacja funkcjonalna - proces

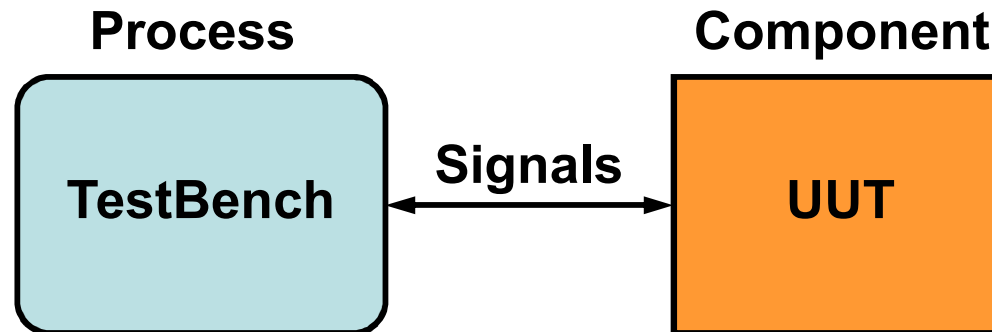


Zarządzanie procesem weryfikacji

- Konieczna kontrola kolejnych wersji/rewizji
- Obsługa zdarzeń weryfikacji:
 - wykryte błędy w projekcie
 - wykryte błędy lub niespójności w specyfikacji
 - zagadnienia optymalizacji *area/speed*
 - nowe pomysły na kolejne warianty weryfikacji
- Forma obsługi:
 - rozmowa z inżynierem/-ami (w wielu małych firmach działa ☺)
wady – brak jasnej odpowiedzialności, brak historii
 - karteczka na biurko inżyniera
wady – czasami się odkleja... i spada na podłogę
+ brak priorytetyzacji zadań, brak historii
 - procedury (*ticket system*)
 - bazy danych

W każdej z metod czas przeznaczony na zgłoszenie błędu nie powinien być dłuższy niż sama korekta błędu!

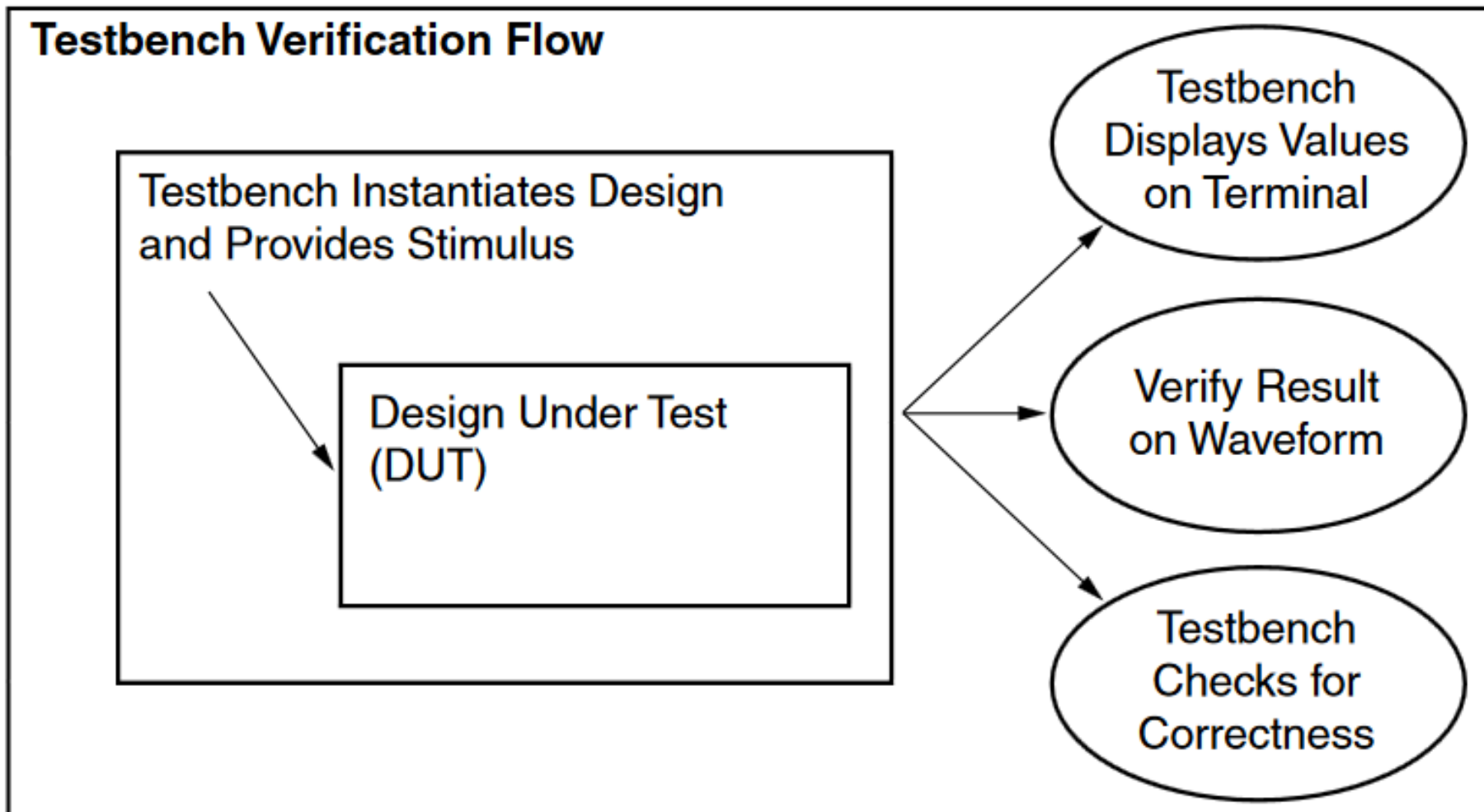
Testbench – połączenie Process/Component



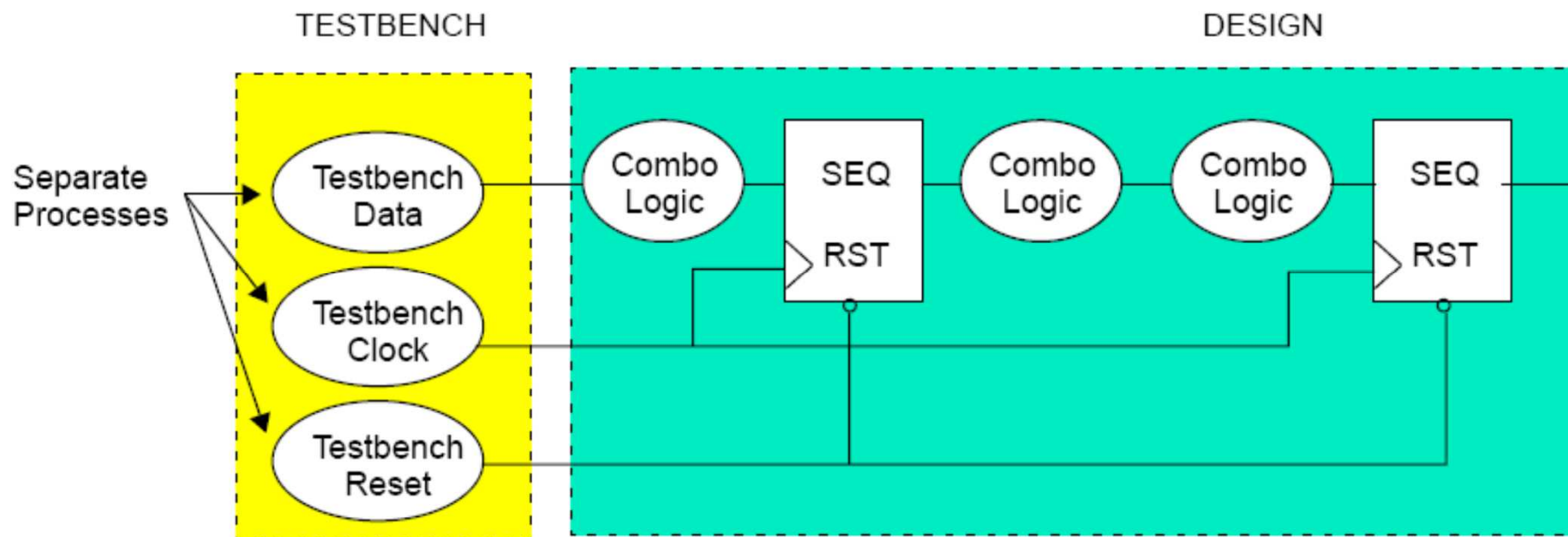
Rodzaje procesów testujących

- **Ad hoc**
Zbiór wektorów testowych do podstawowych testów funkcjonalnych.
- **Algorytmiczny**
Prosty algorytm generujący wymuszenia, np. pętla zwiększająca zmienną przez cały jej zakres do testowania dekodera lub ALU.
- **Plik wektorów**
Rozwiązanie strukturalne: proces czytający plik z wektorami do testowania.

NAJLEPIEJ: **Testbench automatycznie sprawdzający działanie i generujący wynik: Error albo OK !!!**



Testbench – przykłady elementarne



Zalecenie generalne – osobne procesy:

- obsługa danych
- zegar systemowy
- sterowanie asynchroniczne (reset)

Testbench – przykład

TB część 1

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity counter_tb is
end counter_tb;

architecture TB_ARCHITECTURE of counter_tb is
-- Component declaration of the tested unit
component counter
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 END_SIM: boolean:=false;
constant CLK_PERIOD: time:= 30 ns;
constant RESET_LENGTH: time:= 50 ns;
```

TB część 2

```
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
-- clock process
CLK_GEN: process
begin
  if END_SIM = false then
    CLK <= '0';
    wait for CLK_PERIOD/2;
    CLK <= '1';
    wait for CLK_PERIOD/2;
  else
    wait;
  end if;
end process;
```

TB część 3

```
-- 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;
  END_SIM := true;
  wait;
end process;

end TB_ARCHITECTURE;
```

Testbench – warianty **wait**

```

-- 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;
  
```

Szybciej symulowane są TB oparte
jedynie na behawioralnych
 instrukcjach **wait for**

```

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

```

-- Process with explicit "wait on or sensitivity list"
-- statement process
-- Synthesizable (non-testbench) process style
process
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;
  
```

```

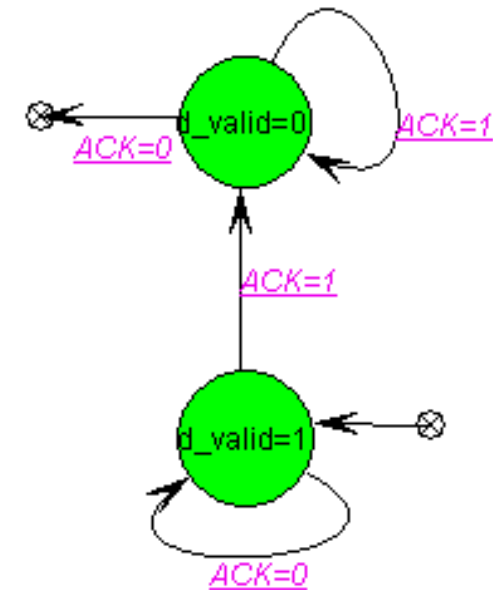
-- Combination of "wait on, until" statement
-- Synthesizable (non-testbench) process style
wait on IN1 until CLK = '0';
  
```

Testbench – pole dla instrukcji behawioralnych

Kod TB jest bardzo wdzięcznym miejscem na modelowanie systemu instrukcjami behawioralnymi.

Przykład – „akcja potwierdzenia” (*handshake*):

- *when* $d_valid = 1$ & $ACK = 0$
- *when* $d_valid = 1$ & $ACK = 1$
- *when* $d_valid = 0$ & $ACK = 1$
- *when* $d_valid = 0$ & $ACK = 0$



Zalety podejścia behawioralnego:

- możliwość wykorzystania bogatego zbioru instrukcji HDL (brak ograniczeń kodowania zgodnego z RTL, np.: używanie arcybutu sygnału **'event** dla wielu sygnałów, instrukcje **wait for** itp.)
- szybkość pisania kodu
- krótszy czas symulacji

Testbench – pole dla instrukcji behawioralnych

```
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;
    ...
  endcase;
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
```



Łatwo, szybko i przyjemnie...

Testbench – zegary

*-- Generacja podzielonego zegara
-- głównego*

```
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;
```

*-- W przypadku stosowania osobnych
-- domen czasowych – osobne
-- procesy generujące zegary*

```
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;
```

Redukcja wielokrotnych punktów wywołań

```
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;
```

```
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 – sterowanie sygnałów R/W

***Sygnały dwukierunkowe
MUSZĄ być wprowadzone
w stan 'Z' w trakcie odczytu
od „strony” TB***

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

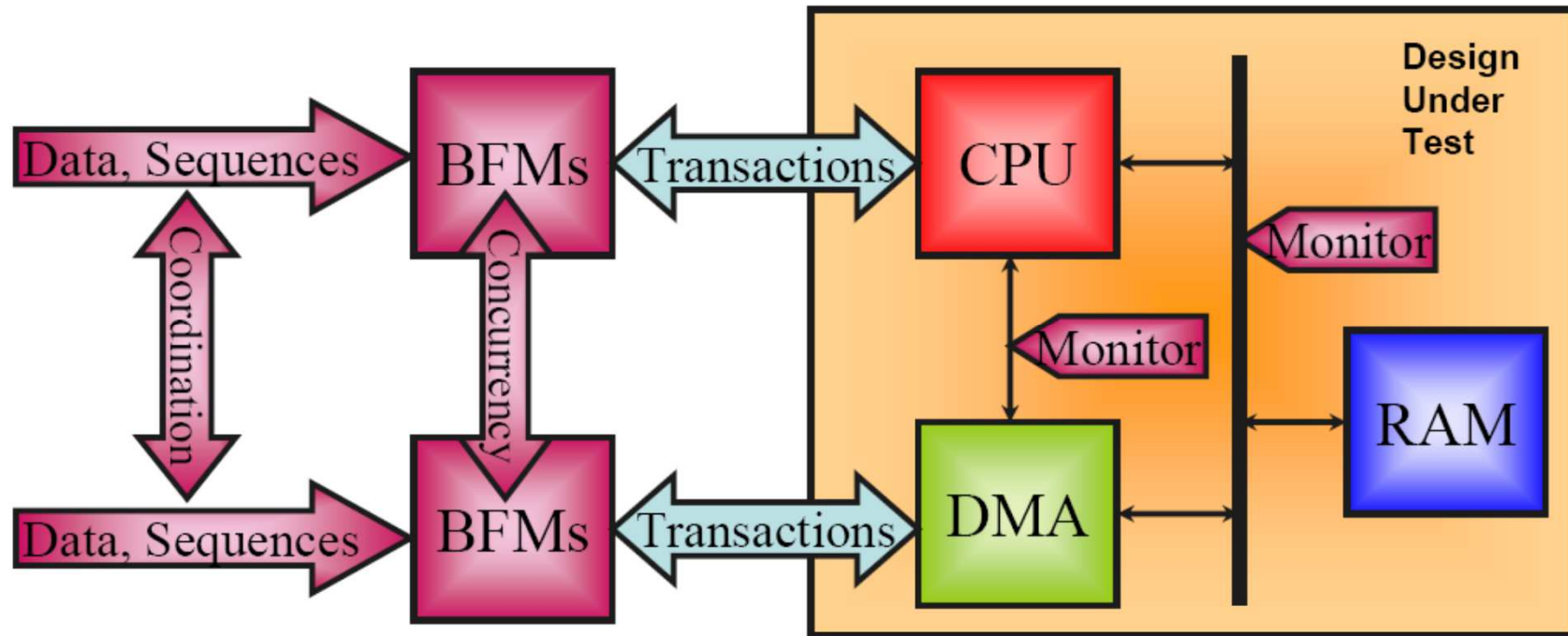
entity bidir_infer is
port (
    DATA: inout STD_LOGIC_VECTOR(7 downto 0);
    READ_nWRITE : in STD_LOGIC);
end bidir_infer ;

architecture XILINX of bidir_infer is
signal LATCH_IN : STD_LOGIC_VECTOR(7 downto 0);
signal LATCH_OUT : STD_LOGIC_VECTOR(7 downto 0);

begin
input: process(READ_nWRITE, DATA)
begin
    if (READ_nWRITE = '1') then
        LATCH_IN <= DATA;
    end if;
end process;

output: process(READ_nWRITE, LATCH_OUT)
begin
    if (READ_nWRITE = '0') then
        DATA <= LATCH_OUT;
    else
        DATA <= (others => 'Z');
    end if;
end process;
end XILINX;
```

Testbench – podejście strukturalne

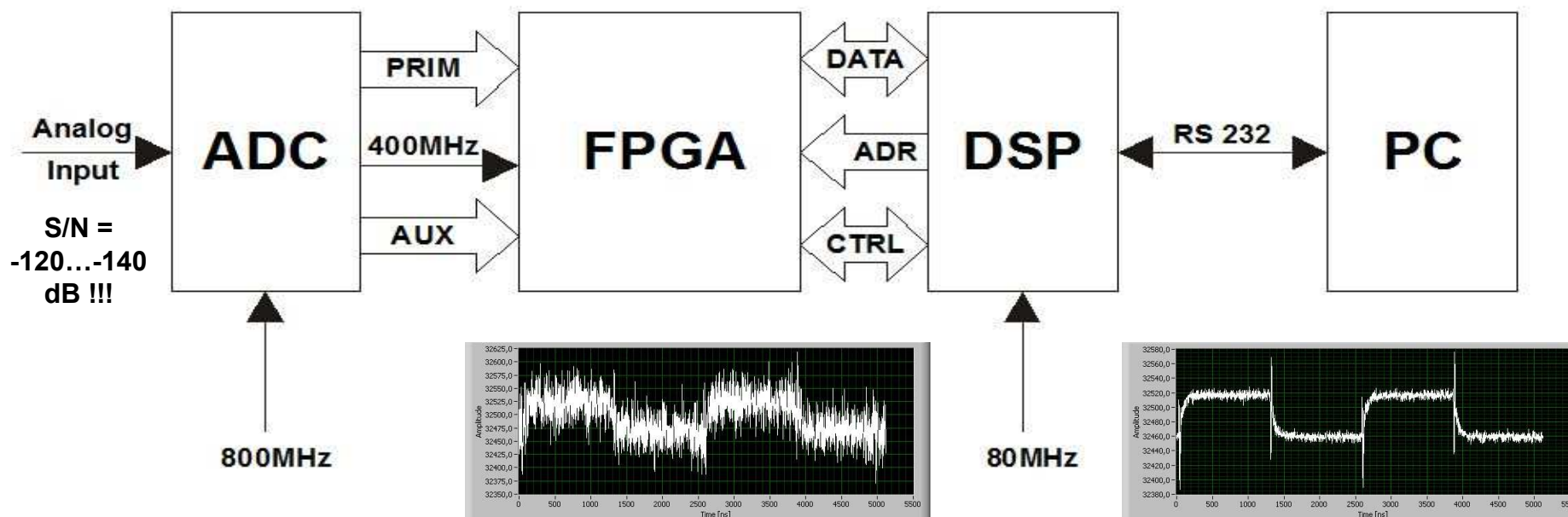


- *Bus Functional Models (BFMs) to control & monitor I/O transactions*
- *Externally generated data and expected results*
- *Concurrent, coordinated system process modeling*
- *Internal transaction and bus monitors*

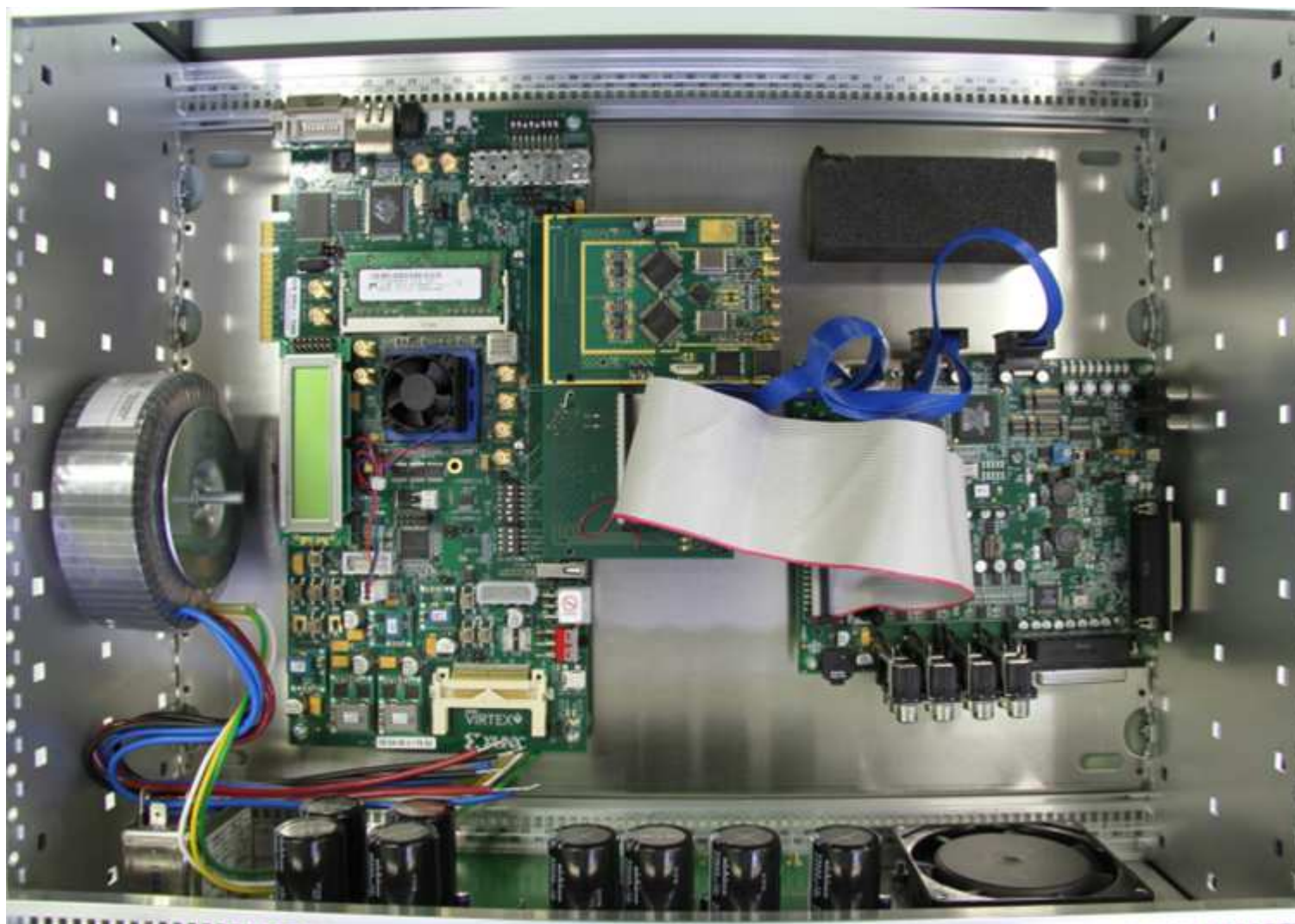
Digital Receiver for EPR Spectrometer 3 versions



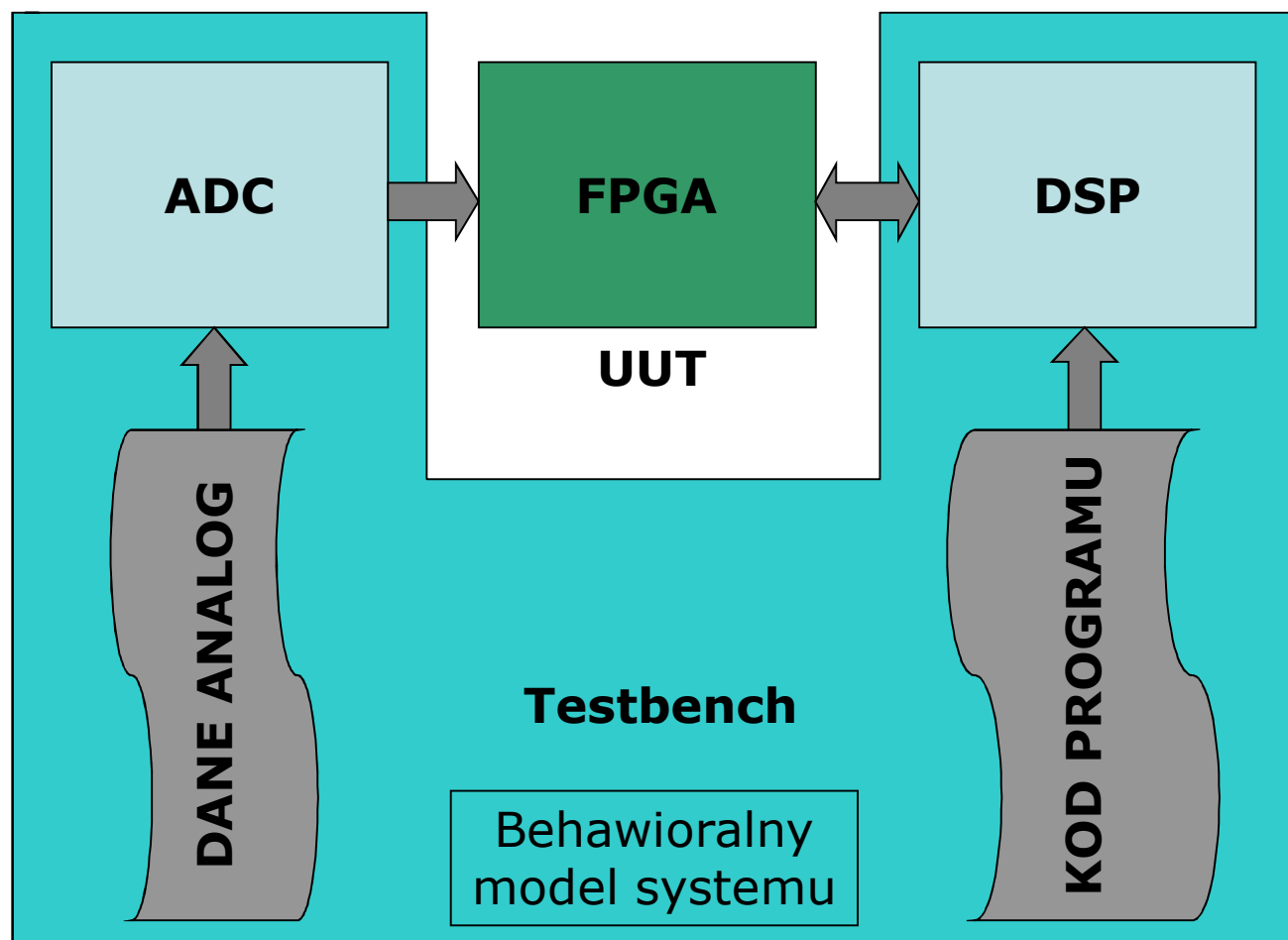
- **2005 version**
ADC: 800Msps/8b FPGA: Virtex-II DSP: ADSP-2191
- **2011 version (DARPA)**
ADC: 2×500Msps/12b FPGA: Virtex-6 DSP: ADSP-21469
- **2018 version**
ADC: 2×400Msps/14b FPGA & DSP: Zynq 7Z020



Digital Receiver for EPR Spectrometer 2011 version



Testbench – przykład systemu Model odbiornika spektrometru EPR



Predefiniowane typy tekstowe

Typy `text` i `line`

Należą do nich typy `text` oraz `line`. Używane są do operacji wejścia i wyjścia podczas symulacji. Występują w deklaracji `file` wraz z funkcjami odczytu, zapisu i pomocniczymi.

STD library - TEXTIO Package:

`readline`, `read`, `writeline`, `write`

Przykład:

```
readline (F: in text; L: out line);  
read (L: inout line; ITEM: integer);
```

IEEE library – STD_LOGIC_TEXTIO Package:

`read`, `write`
`oread`, `owrite`
`hread`, `hwrite`

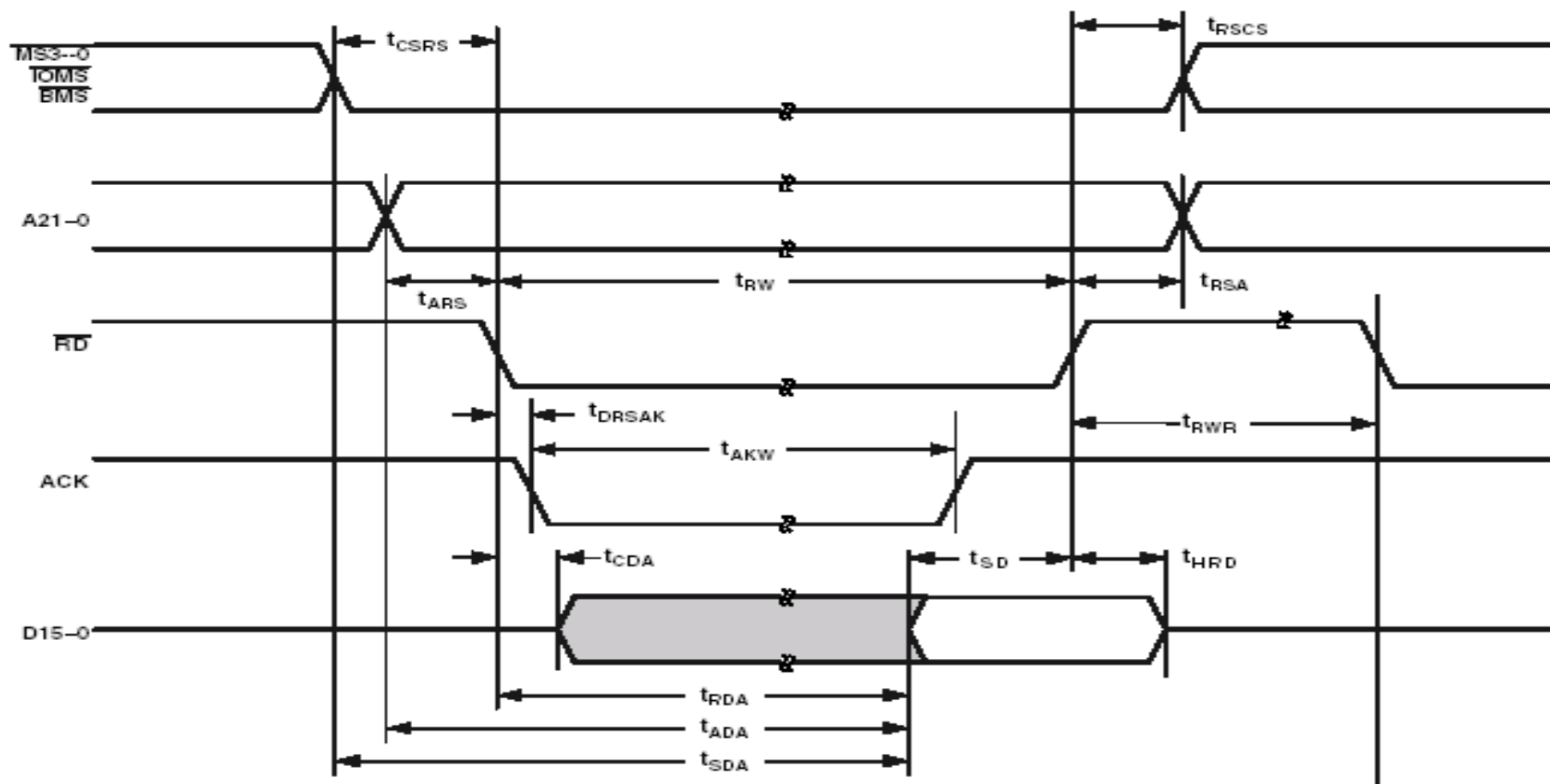
Wbudowane:

`endfile` (*filename*) , `endline` (*linename*)



Testbench – przykład systemu

Procedura modelująca transfer na magistrali



Testbench – przykład systemu

Procedura modelująca transfer na magistrali

Parameter ^{1, 2}	Min	Max	Unit
<i>Switching Characteristics</i>			
t_{CSRS} Chip Select Asserted to $\overline{RD}$ Asserted Delay	$0.5t_{HCLK}-3$		ns
t_{ARS} Address Valid to $\overline{RD}$ Setup and Delay	$0.5t_{HCLK}-3$		ns
t_{RSCS} $\overline{RD}$ Deasserted to Chip Select Deasserted Setup	$0.5t_{HCLK}-2$		ns
t_{RW} $\overline{RD}$ Strobe Pulsewidth	$t_{HCLK}-2+W^3$		ns
t_{RSA} $\overline{RD}$ Deasserted to Address Invalid Setup	$0.5t_{HCLK}-2$		ns
t_{RWR} $\overline{RD}$ Deasserted to $\overline{WR}$, $\overline{RD}$ Asserted	t_{HCLK}		
<i>Timing Requirements</i>			
t_{AKW} ACK Strobe Pulsewidth	t_{HCLK}		ns
t_{RDA} $\overline{RD}$ Asserted to Data Access Setup		$t_{HCLK}-4+W^3$	ns
t_{ADA} Address Valid to Data Access Setup		$t_{HCLK}+W^3$	ns
t_{SDA} Chip Select Asserted to Data Access Setup		$t_{HCLK}+W^3$	ns
t_{SD} Data Valid to $\overline{RD}$ Deasserted Setup	7		ns
t_{HRD} $\overline{RD}$ Deasserted to Data Invalid Hold	0		ns
t_{DRSAK} ACK Delay from $\overline{RD}$ Low	0		ns

Testbench – przykład systemu

Procedura modelująca transfer na magistrali

```
procedure read_cycle (...) is
```

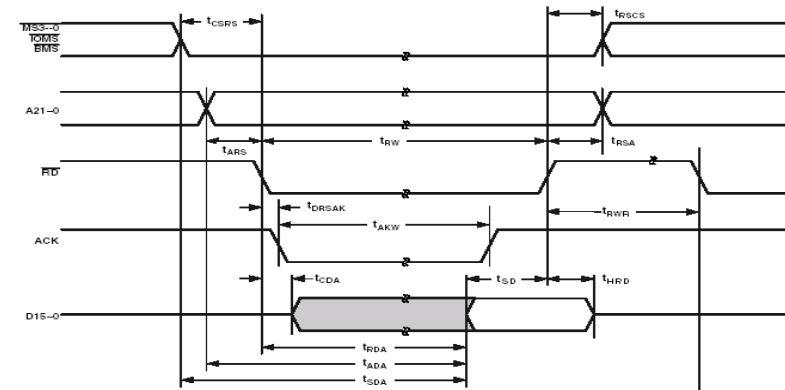
```

constant T_CSRS: TIME := H_clk_per/2 - 3ns;
constant T_ARS:  TIME := H_clk_per/2 - 3ns;
constant T_RW:   TIME := H_clk_per - 2ns + (WS * H_clk_per);
constant T_RSA:  TIME := H_clk_per/2 - 2ns;
```

```
begin
```

```

    ms3 <= '0';
    addr <= address;
    wait for T_ARS;
    rd <= '0';
    wait for T_RW;
    rd <= '1';
    data_read := data;
    wait for T_RSA;
    addr <= (others => 'Z');
    ms3 <= '1';
end procedure read_cycle;
```



Parameter ^{1,2}	Min	Max	Unit
<i>Switching Characteristics</i>			
t_CSRS	0.5t _{HCLK} -3		ns
t_ARS	0.5t _{HCLK} -3		ns
t_RW	0.5t _{HCLK} -2		ns
t_RSA	t _{HCLK} -2+W ³		ns
t_RWR	0.5t _{HCLK} -2		ns
<i>Timing Requirements</i>			
t_AKW	t _{HCLK}		ns
t_SDA		t _{HCLK} -4+W ³	ns
t_ADA		t _{HCLK} +W ³	ns
t_SDA		t _{HCLK} +W ³	ns
t_SD	7		ns
t_HRD	0		ns
t_DRSK	0		ns

Testbench – przykład systemu Metajęzyk do modelowania transakcji na magistrali

adsp2191.txt ->

begin

...

READ_CODE: process
begin

```
while not (endfile(transfers)) loop -- end file checking
  readline(transfers,IN_LINE);      -- read line of a file
  read(IN_LINE,code);               -- read in operation code
  hread(IN_LINE,address);           -- read in address
  hread(IN_LINE,data);              -- read in data
  code_array(i) := code;            -- put code in code table
  address_array(i) := address;      -- put address in address table
  data_wr_array(i) := data;         -- put data in data table
  i := i + 1;
end loop;
```

```
N 0028 0000 #czekaj na inicjalizację
W 4011 0001 #uruchom DCM
R 4011 0001 #sprawdź DCM
W 4001 01FF #ustaw długość akwizycji
W 4002 00FF #ustaw liczbę akumulacji
W 4000 0001 #ustaw 'Run' @ OneShot
N 0035 0000 #czekaj n * 12.5ns
W 4003 0000 #kasuj 'Int'
W 4000 0000 #kasuj 'Run'
R 0000 0200 #czytaj bufor danych
```

Testbench – przykład systemu

Wyniki symulacji w pliku raportów

```
....  
for index in 0 to max loop  
  decode: case code_array(index) is  
    when 'W' =>  
      ...  
      write_cycle(...); -- write cycle timing model  
      write(OUT_LINE, NOW, right, 12, ns); -- current simulation time  
      write(OUT_LINE, code_array(index), right); -- current code  
      write(OUT_LINE, address_array(index), right); -- current address  
      write(OUT_LINE, data_wr_array(index), right); -- current data  
      writeline(results, OUT_LINE);  
    when 'R' => ...  
....
```

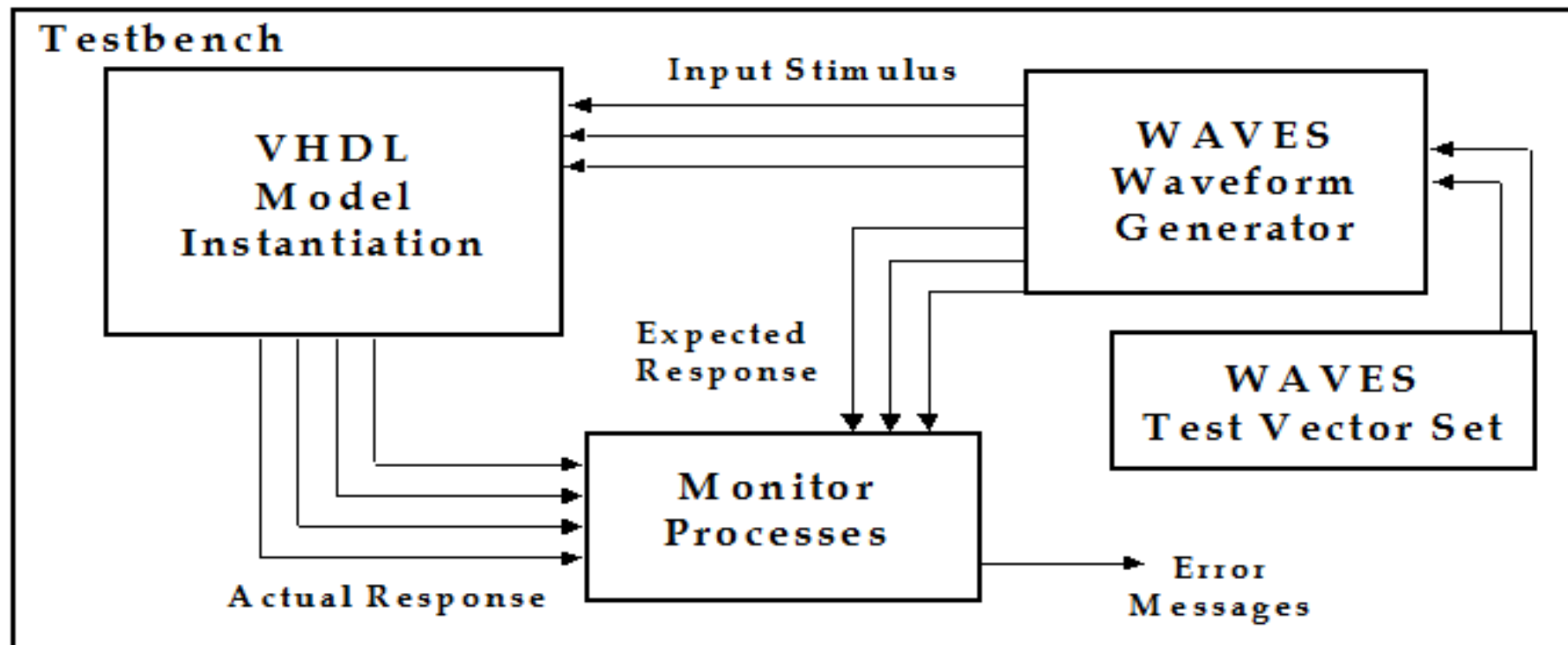
simulation.txt ->

```
...  
6939 ns W 0100000000000000 0000000000000000  
6964 ns N  
6994.5 ns R 0000000000000000 0000000100000000  
7025 ns R 00000000000000001 0000000100100000  
7055.5 ns R 00000000000000010 0000000101000000  
7086 ns R 00000000000000011 0000000101100000  
...
```

ActiveHDL – wsparcie dla TB

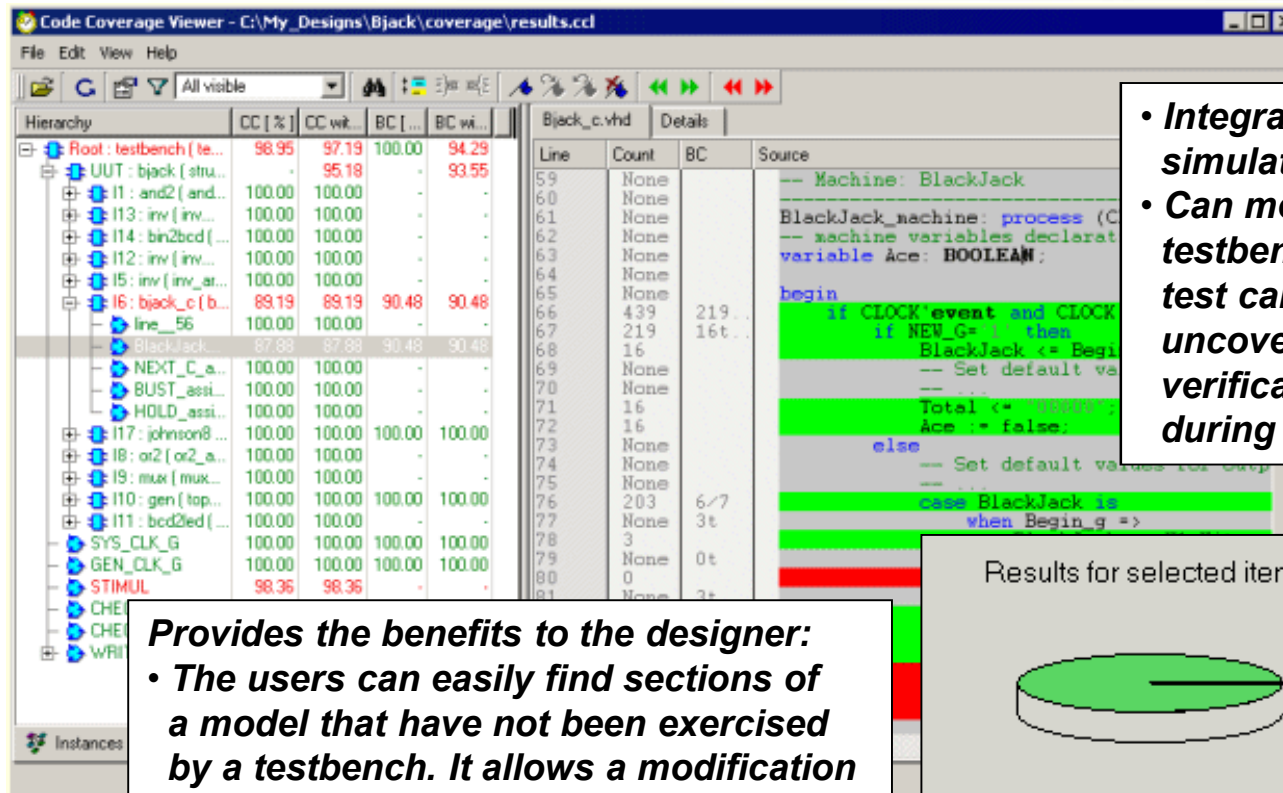
- Szablony TB
 - generacja zegara
 - wymuszenia wartości sygnałów:
 - *Concurrent mode*
 - *Sequenced mode*
 - porównanie wyników
 - zapisywanie wyników
 - funkcje randomizacji
- Waveform and Vector Exchange Specification
WAVES (IEEE-STD-1029.1)
- *Code Coverage*
- *Linting*

• WAVES Testbench Diagram



The WAVES-Based testbenches are based on concurrent procedures and processes. One procedure reads an external test vector file (*.vec) and produces both the input stimulus and output pattern vectors. The test vector file is based on a text format specified by the WAVES standard. In addition to the mentioned procedure, some stimuli signals can be generated by additional processes without reading the VEC file. Other procedures compare output signals of the UUT entity with the pattern vectors. After the testbench is executed, all detected discrepancies are reported to the console window and to a log file.

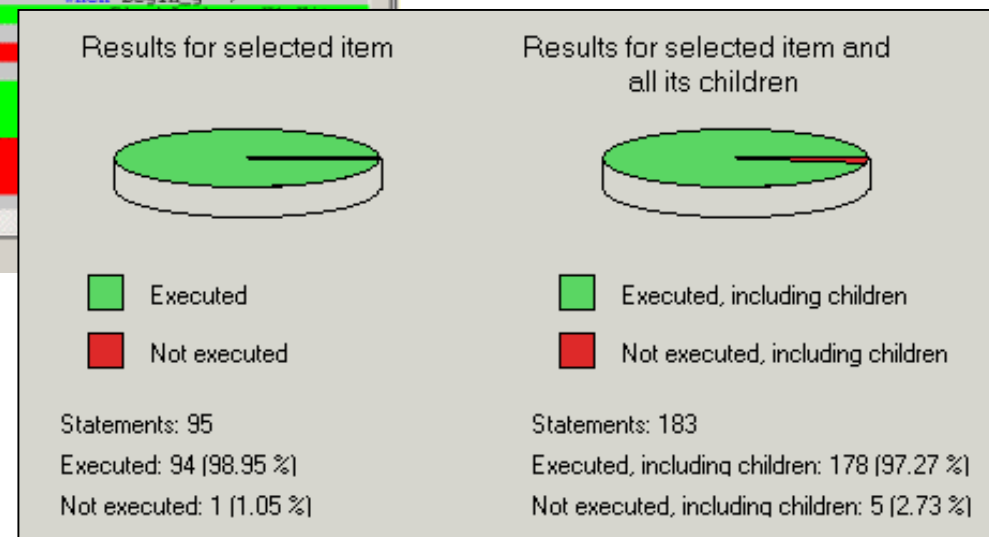
ActiveHDL: Code Coverage



- **Integrated into the Active-HDL simulation kernel.**
- **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.**

Provides the 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.**
- **Helps to identify sections of the model executed very frequently. This allows the user to optimize the execution of the model during the simulation.**



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.



HDL Linting Engine

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

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;
```

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;
```

Safety-Critical Design Assurance Guidelines

- RTCA/DO-254:
 - ♦ Design assurance guidelines for Airborne Electronic Hardware (AEH)
 - ♦ Accepted by the Federal Aviation Administration (FAA) in 2005
 - ♦ The goal of the standard is to ensure that AEH works reliably
 - ♦ Design Assurance Levels (DAL) A—E determine hardware design objectives
 - ♦ DO-254 projects have special requirements for tools and design flows



DO-254
When Safety is Critical

<http://www.do254.com>

Aldec DO-254 HDL Coding Standards

- Aldec DO-254 standard:
 - ♦ Based on the feedback from real customers *(DO-254 programs)*
 - ♦ Good foundation of guidelines for any design *(Not DAL A/B only)*
- Basic rule groups:
 1. Structure and portability *(Data types, naming, coding, statements...)*
 2. Downstream checks *(Racing, sensitivity lists, clocks & resets, bit widths...)*
 3. Error-prone patterns *(Subprograms, registers, interconnections, hazardous blocks...)*

Group #1: Structure & Portability

- Code structure and readability for efficient reuse, examples:

- ♦ Declare one object per line and always add comments
- ♦ Port description order should follow a pattern
- ♦ Avoid using hard-coded numbers for characteristics that may change
- ♦ Do not use similar identifiers even in different namespaces
- ♦ Use VHDL data types properly
- ♦ All objects declared in the code must be used
- ♦ Global design parameters must be defined in a package
- ♦ ...

Avoid using hard-coded numbers for characteristics that may change

Main message –

Hard-coded number(s) used in the process.

Detail message –

"00000000" should be parameterized.

```

1 process ( CLOCK, RESET, ENABLE )
  begin

    if RESET='0' then
      FIBOUT <= "00000001";
      PREV_FIB <= "00000000";

    elsif falling_edge(CLOCK) and ENABLE='1' then
      PREV_FIB <= FIBOUT;

2 case (FIBOUT="00000000" and (PREV_FIB="00000000")) is
  when true => FIBOUT <= "00000001";
  when others => FIBOUT <= FIBADD;
end case;
ise

```

Group #2: Downstream Stages

- Problems that normally surface later in design flow, examples:
 - ♦ Do not use non-synthesizable subset of the HDL language
 - ♦ Do not use combinatorial feedbacks (*racing conditions*)
 - ♦ Avoid unreachable conditions (*code that will never be executed*)
 - ♦ Define all the necessary signals in sensitivity lists (*combinatorial process*)
 - ♦ Avoid internally generated clocks unless they are properly isolated
 - ♦ Gated clocks can be used only at a top level (*ASIC-specific*)
 - ♦ ...

Group #3: Error-Prone Patterns

- Design patterns that are prone to problems and errors, examples:
 - ♦ Empty blocks should not be used
 - ♦ Do not describe multiple independent conditions in a process
 - ♦ Avoid unconnected and misused ports
 - ♦ All referenced signals should have drivers
 - ♦ Avoid using hazardous synchronization schemes (*process-level*)
 - ♦ Do not locate logic between asynchronous clock domains (*metastable conditions*)
 - ♦ ...

DO-254/CTS (Compliance Tool Set)

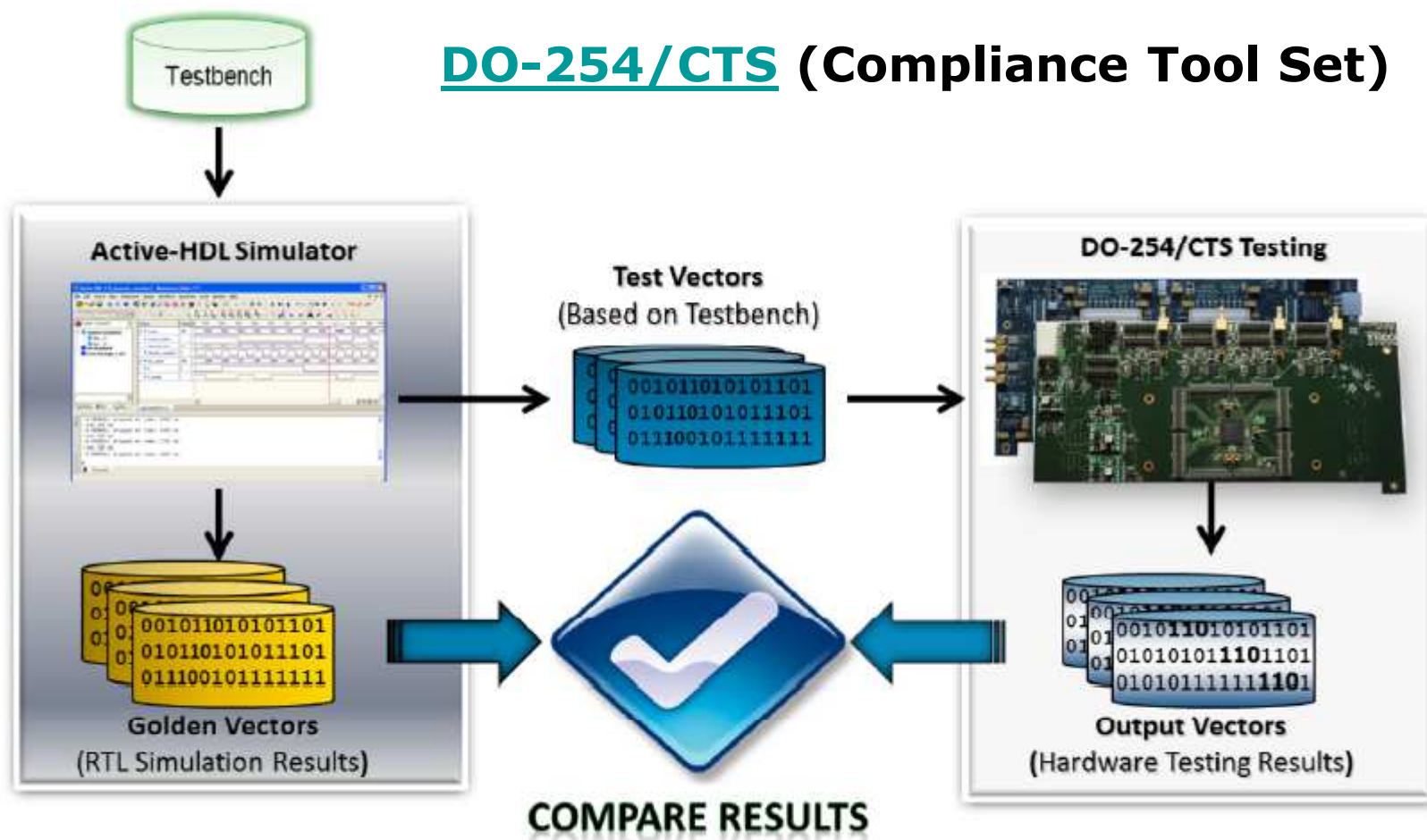


Figure 4: DO-254/CTS Methodology

Ciąg dalszy
nastąpi...

