PSoC: zastosowania

- Rozwiązania oparte na idei mieszanego układu rekonfigurowalnego PSoC znalazły komercyjne zastosowania w wielu firmach (ich logotypy na kolejnych slajdach).
- Istnieją architektury PSoC zoptymalizowane pod kątem konkretnych zastosowań (np. motoryzacyjnych).

PSoC: rozwiązania motoryzacyjne

- Specjalizowana dla zastosowań motoryzacyjnych platforma traveo:
  - Przetwarzanie danych, połączenia sieciowe, elastyczność i spełnienie wymagań bezpieczeństwa dla systemów motoryzacyjnych
  - Zawiera AUTOSAR software
  - Optymalizacja procesów graficznych

PSoC: rozwiązania przemysłowe

- Przemysł 4.0 (sterowanie procesem produkcji z użyciem lokalnej chmury)
- Urządzenia medyczne
- Monitoring (systemy wizyjne)
- Systemy pomiarowe
- Sieci dystrybucji energii
- Inteligentne urządzenia przenośne (jako front-end IoT)

PSoC: rozwiązania konsumenckie

- Wearables (propozycje nazw polskich: urządzenia noszone, e-gadżety, ….)
- Inteligentne domy
- Elektronika osobista (np. rzeczywistość rozszerzona)
- Zdalne urządzenia sterująco-kontrolne

Weryfikacja i testowanie

- Gotowy kod opisujący działanie systemu cyfrowego musi być sprawdzony.
- Może to zachodzić na etapie skompilowanego kodu. Mówimy wtedy o **weryfikacji projektu**.
- Jeśli układ jest już wykonany (zaprogramowany układ rekonfigurowalny, wyprodukowany ASIC) mówimy o **testowaniu układu**.
- Ważne są oba etapy. Zadaniem weryfikacji jest znalezienie błędów merytorycznych w kodzie. Zadaniem testowania jest wyeliminowanie fizycznych błędów w układzie.

Testowanie

- Istnieje testowanie na etapie produkcji. Ma ono wyeliminować uszkodzone (nie spełniające wymagań) struktury scalone (chipy).
- Testowanie na etapie komercjalizacji to najczęściej tzw testowanie strukturalne, wykorzystujące wbudowane bloki testujące. Najczęściej są to:
  - Ścieżka krawędziowa
  - JTAG

Weryfikacja projektu

- Weryfikacja projektu (modelu) polega na wykazaniu, ze projekt jest zgodny ze specyfikacjami.
- Tradycyjnym sposobem formułowania specyfikacji jest tworzenie w języku naturalnym zbioru warunków opisujących pożądane zachowanie systemu.
- Wadą takiego opisu jest brak sformalizowanej reprezentacji, przystosowanej do komputerowego przetwarzania.
- Typową metodą weryfikacji jest wykonanie **symulacji** projektu. Sposób ten ma istotne wady. Najważniejsza to fakt, że do wyczerpującej weryfikacji wymagane jest dostarczenie wszystkich możliwych pobudzeń (kombinacji wektorów wejściowych).

Weryfikacja formalna

- Weryfikacja formalna to weryfikacja oparta na matematycznych metodach formalnych pozwalających dowieść zgodności systemu ze specyfikacją. Polega na wykonaniu formalnego dowodu na abstrakcyjnej reprezentacji systemu.
- Weryfikacja formalna nie wymaga tworzenia środowiska symulacyjnego (test bench) i generacji pobudzeń.
- Użytkownik definiuje w sformalizowany sposób pożądane zachowanie systemu (jego własności - assertion) i własności środowiska (constraint) w jakim pracuje system, w celu wyeliminowania niedozwolonych stanów wejść, itp.
- **SystemVerilog** należy do jednego z języków wykorzystywanych w weryfikacji formalnej.

Wykorzystanie weryfikacji formalnej

- Weryfikacja formalna powoduje:
  - wykrycie błędów na wcześniejszych etapach i z większą skutecznością niż podczas symulacji,
  - zmniejszenie liczby iteracji i czasu tworzenia projektu.
- Formalna definicja własności staje się integralną częścią projektu, co wpływa na lepsze możliwości powtórnego wykorzystania projektu (reuse).
- Wadą weryfikacji formalnej są koszty związane z poznaniem nowej metodologii, formalizmów logicznych oraz narzędzi EDA.
  - Z tego względu można zaobserwować w tej chwili pewien opór projektantów systemów cyfrowych.

SystemVerilog: geneza

- Ułomność Verilog w zakresie tworzenia testbench'y
- Powstanie komercyjnych języków weryfikacji (np. OpenVera, *e*) niedostępnych powszechnie.
- Powstanie stowarzyszenia Accellera firm związanych z narzędziami EDA.

„The most valuable benefit of SystemVerilog is that it allows the user to construct reliable, repeatable verification environments, in a consistent syntax, that can be used across multiple projects." (Chris Spear, SystemVerilog for verification)

SystemVerilog: podstawowe własności

- SV jest rozszerzeniem Verilog, a więc przejmuje wszystkie możliwości języka Verilog;
- W modelowaniu systemów SV dodaje do mechanizmów opisu sprzętu również możliwości charakterystyczne dla podejścia programistycznego (tzw. programmer view PV charakterystyczne np. dla SystemC)
  - włącza elementy programowania obiektowego,
  - dynamiczne wątki,
  - komunikacja pomiędzy procesami.
- Zawiera konstrukcje zorientowane na tworzenie testbench'ów. Przystosowany do stosowania metod weryfikacji funkcjonalnej tzw. Verification Methodology (OVM - Open VM, UVM – Universal VM).

OVM (Open Verification Methodology) w pigułce

- Stosuje się najpierw proste testy deterministyczne, aby określić poprawność działania podstawowych funkcji weryfikowanego systemu. Następnie przeprowadzane są testy statystyczne (losowo generowane wektory testowe).
- Ponieważ testowanie wyczerpujące (100% pokrycia testami) nie jest zwykle możliwe, kompromisem jest odpowiednie sterowanie generacją wektorów testowych, aby uzyskać maksymalne pokrycie przy minimalnej liczbie wektorów.
- OVM (UVM) oferuje biblioteki modułów wspomagających weryfikację funkcjonalną.

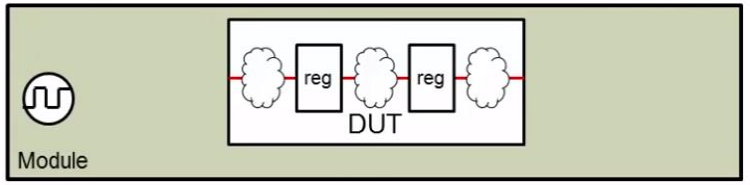
Weryfikacja z SystemVerilog

Konstrukcje wprowadzone w SystemVerilog:

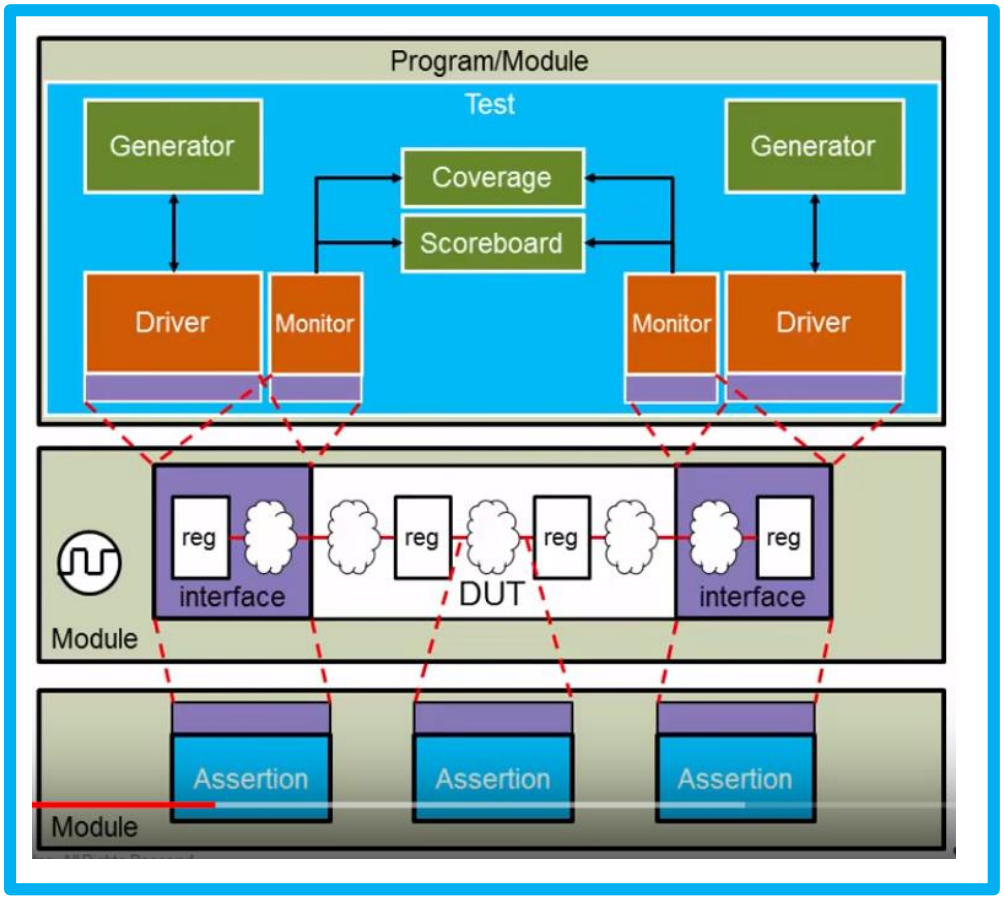
- class,
- randomization,
- coverage,
- interface,
- assertion,
- package

pozwalają na utworzenie w jednolitym narzędziu i języku rozbudowanego środowiska weryfikacyjnego.

tak było w Verilog



tak może być w SV



SystemVerilog typy

- SystemVerilog wprowadza wiele nowych struktur danych w stosunku do Verilog. Część z nich to udoskonalenia w zakresie opisu działania sprzętu, część zaś użyteczne dla testbenchy struktury charakterystyczne dla języków programowania. Najważniejsze to:
  - typy dwuwartościowe (w Verilog'u tylko 4-wartościowe: x, z, 0, 1) – pozwalają na przyspieszenie symulacji i mniejsze wykorzystanie pamięci,
  - kolejki, tablice dynamiczne i automatycznie zapisywane – wbudowane mechanizmy przeszukiwania i sortowania
  - unie i struktury spakowane (packed)
  - klasy i struktury
  - łańcuchy z wbudowaną obsługą
  - typy wyliczeniowe (enumerated)

SystemVerilog typy: przykłady

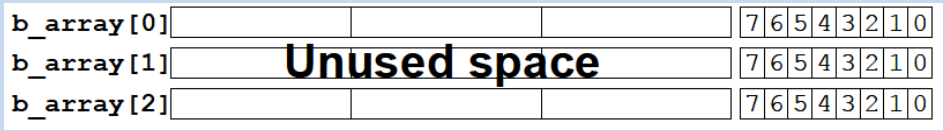
Typ logic, logic[]:
zastępuje reg,
występuje w przypisaniach proceduralnych i ciągłych

Typy dwustanowe:
bit, bit[] – zawsze unsigned
int - 32-bit signed integer
byte - 8-bit signed
shortint - 16-bit signed integer
longint - 64-bit signed integer

Tablice:

- int asc[4] = '{0,1,2,3}; - literał '{} inicjuje elementy tablicy asc, skrócona deklaracja (zamiast int asc[0:3])
- asc = '{4{8}}; - ustawienie 4 wartości na 8
- SV przechowuje każdy element tablicy w obszarze 32 bitów (przez co różnice w obsłudze tablic są bardzo duże w stosunku do Verilog), np.

bit[7:0] b_array[3]



Instrukcje proceduralne

SystemVerilog wprowadza wiele mechanizmów typowych dla C, np. deklarowanie indeksu i zakresu wewnątrz pętli, operatory ++ i --, instrukcje kontroli pętli break i continue, etykietowanie początku i końca modułów i podprogramów. Przykłady (z SystemVerilog for verification, Chris Spear)

```
initial
    begin : example
    integer array[10], sum, j;
    // Declare i in for statement
    for (int i=0; i<10; i++)        // Increment i
        array[i] = i;
    // Add up values in the array
    sum = array[9];
    j=8;
    do // do...while loop
        sum += array[j];       // Accumulate
    while (j--);                    // Test if j=0
    $display("Sum=%4d", sum); // %4d - specify width
    end : example                   // End label
```

```
initial begin
    logic [127:0] cmd;
    integer file, c;
    file = $fopen("commands.txt", "r");
        while (!$feof(file)) begin
            c = $fscanf(file, "%s", cmd);
            case (cmd)
            "": continue; // Blank line - skip to loop end
            "done": break; // Done - leave loop
            // Process other commands here
                ...
            endcase // case(cmd)
        end
    $fclose(file);
    end
```

Task i function

- SystemVerilog wprowadza kilka udoskonaleń powodujących, że podprogramy stają się bardziej podobne do C.
  - Skrócone deklaracje (jak w C) argumentów, domyślny kierunek input, np. task mytask (output logic [31:0] x, logic y);
  - Dodatkowy typ (kierunek) **ref** pozwalający modułowi wywołującemu na ciągły dostęp do aktualnej wartości obiektu (nie dopiero po skończniu działania podprogramu).
  - Możliwość wyspecyfikowania domyślnych wartości argumentów (jeśli pominięte w wywołaniu).
  - Dodanie return jako instrukcji zakończenia wykonywania podprogramu.

Elementy programowania obiektowego: klasa

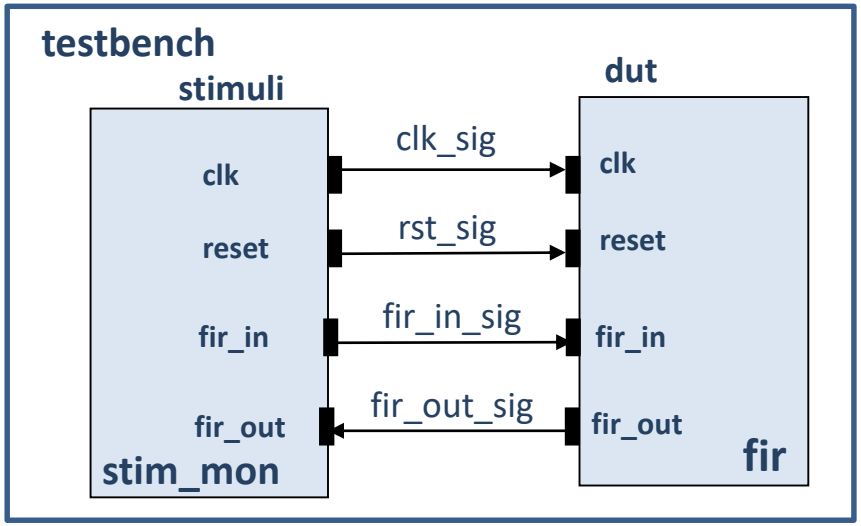
- Klasa definiowana może być w module, program, package lub poza nimi. Jest wykorzystywana przede wszystkim w programowaniu testbenchy.
- Tworzenie obiektu klasy, w odróżnieniu od instancji modułu tworzonej na etapie elaboracji (kompilacji), następuje w momencie wywołania.

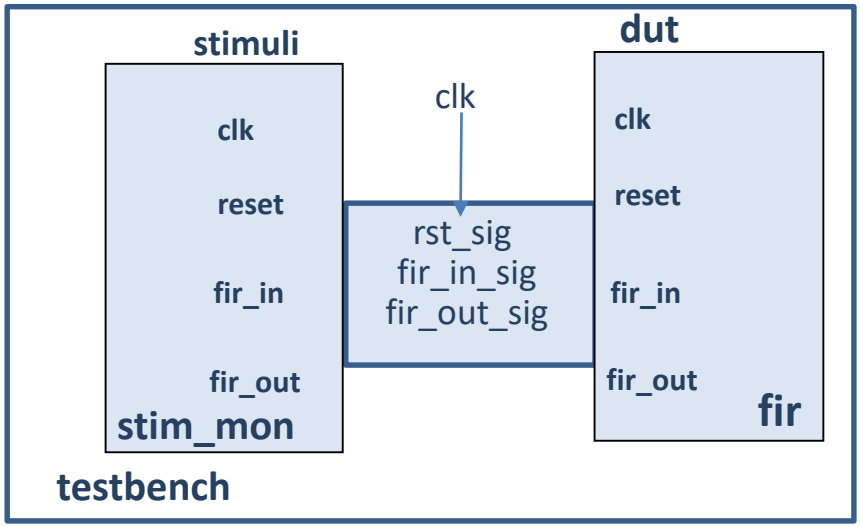
```
// deklaracja klasy
class BusTran;
      bit [31:0] addr, crc, data[8];
      function void display;
            $display("BusTran: %h", addr);
      endfunction : display
      function void calc_crc;
            crc = addr ^ data.xor;
      endfunction : calc_crc
endclass : BusTran
```

```
// konstrukcja obiektu klasy z wartością null
BusTran b; // Declare a handle
b = new; // Allocate a BusTran object
// definicja klasy wraz z wartościami new
class BusTran;
 logic [31:0] addr, crc, data[8];
                function new;
                addr = 3;
                foreach (data[i])
                data[i] = 5;
                endfunction
endclass
```

Komunikacja DUT - testbench

- W klasycznym Verilog (VHDL) komunikacja pomiędzy modułem generującym testy i monitorującym a weryfikowanym modułem (DUT Design Under Test) zachodzi przy użyciu portów i ukierunkowanych sygnałów (typ net). Moduł testbench zawiera instancje modułów testującego i testowanego i odwzorowanie połączeń.
- SystemVerilog wprowadza interfejsy. W najprostszym zastosowaniu interfejs jest wiązką bezkierunkowych połączeń (typ logic). W module głównym (testbench) tworzymy instancje modułów stimuli i dut oraz **instancję interfejsu**.





Komunikacja poprzez interfejs: przykład fir

```
// opis interfejsu

interface fir_if (input bit clk);
logic [7:0] fir_in, fir_out;
logic reset;
endinterface
```

```
// łączenie w module głównym

module testbench;
    bit clk;
    always #5 clk = ~clk; // generacja zegara
// instancja firif interfejsu fir_if
    fir_if firif(clk);
// instancje modułów
    fir dut(firif);
    stim_mon stimuli(firif);
endmodule : testbench
```

SystemVerilog pozwala na łączenie modułów zarówno poprzez porty jak i interfejsy, np:
fir dut (.clk (firif.clk), .fir_in (firif.fir_in), .reset (firif.reset), .fir_out (firif.fir_out));

Program i module w SystemVerilog

- W SystemVerilog odróżniamy model działania układu opisany w modułach od części testującej (testbench) opisanej jako program. Te dwie części muszą być odseparowane logicznie i **czasowo**.
- Program nie ma hierarchii i nie można w nim używać współbieżnych bloków (np. always), wykorzystuje się wiele (również systemowych) zadań i funkcji oraz wprowadzoną w SV obsługę własności (definicja i potwierdzenie), np.

```
property request_2state;
@(posedge clk) disable iff (reset);
$isunknown(request) == 0;
endproperty
assert_request_2state: assert property request_2state
```

- Niebagatelnym problemem jest zapewnienie takiego harmonogramowania procesów, aby części te prawidłowo ze sobą współpracowały nie powodując błędów synchronizacji (aby sygnały testowe nie były wystawiane w złych momentach czasowych w stosunku do zbocza zegarowego). Jednym z „łączników” pomiędzy programem a modułem jest interfejs – generacja zegara w interfejsie znacznie zmniejsza ryzyko błędów synchronizacji.
- Ale to już całkiem inna, złożona historia. Na szczęście dla Państwa nie na ten wykład …….

Testy pseudolosowe

- SystemVerilog posiada mechanizmy ułatwiające testowanie pseudolosowe CRT (Constraint Random Test).
- CRT składa się z dwu części:
  - kodu testującego używającego strumienia losowych wartości i podłączającego go do DUT,
  - ziarna dla generatora liczb pseudolosowych.
- Zmiana ziarna powoduje generację innego zestawu wektorów testowych.
- Testowanie z użyciem generatora liczb pseudolosowych jest znacznie bardziej wydajne niż tworzenie własnych testów (*directed testing*).
- CRT nie jest testowaniem wyczerpującym, dlatego konieczne są mechanizmy dające najlepsze pokrycie funkcjonalne (testy wykrywające najwięcej błędów).
- Ale to też już inna historia…….