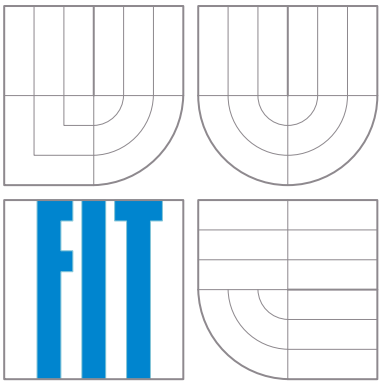
Vysoké učení technické v Brně
Fakulta informačních technologií



Pokročilé číslicové systémy

Verifikace číslicových systémů

cvičení

2012

Tento materiál vznikl za podpory Fondu rozvoje vysokých škol (projekt 1798/2012).

Obsah

1 Úvod

V procesu vývoje číslicových systémů je jejich (funkční) verifikace nedílnou a důležitou součástí. Jazyky pro popis číslicových systémů jako VHDL či Verilog obsahovaly (a obsahují) jednoduché konstrukce v samotné definici jazyka pro návrh testovacích obvodů či testovacích prostředí (testbench). Avšak v důsledku rostoucí složitosti vyvíjených obvodů vznikla potřeba vytvořit specializované jazyky obsahující prostředky pro návrh komplexních verifikačních prostředí (SystemVerilog, *e*, OpenVera). V souvislosti s tímto vývojem se rozrůstají i možnosti existujících nástrojů pro vývoj číslicových systémů a to zejména podporou těchto specializovaných jazyků a verifikačních metodik. Navíc vznikají i samostatné, specializované nástroje pro funkční a formální verifikaci číslicových obvodů.

V tomto cvičení se budeme zabývat jazykem **SystemVerilog** a verifikační metodikou **OVM** (angl. *Open Verification Methodology*). Na příkladu **verifikace aritmeticko-logické jednotky** (ALU) budou demonstrovány výhody specializovaného jazyka pro verifikaci, jednotného verifikačního prostředí definovaného metodikou OVM a samotného procesu funkční verifikace pomocí simulačního nástroje ModelSim od společnosti Mentor Graphics.

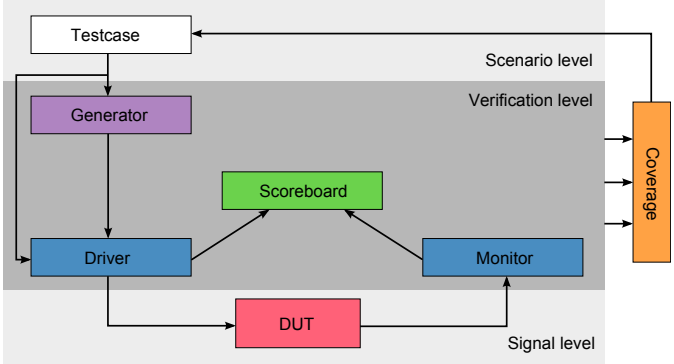
SystemVerilog je objektově orientovaný jazyk jenž vzniknul v roce 2005 jako rozšíření jazyka **Verilog** o vlastnosti orientované na **verifikaci**. Mezi hlavní doplněné vlastnosti patří: generování náhodných transakcí na základě omezujících podmínek (angl. *constrained-random stimulus generation*), podpora analýzy pokrytí funkcionality systému verifikačními běhy (angl. *coverage*) a také prostředky pro popis invariantů systému a kontrole jejich platnosti při verifikačních bězích (angl. *assertion-based verification*). Kromě původních datových typů převzatých z Verilogu byla přidána podpora dalších, určených ke zjednodušení verifikace číslicových systémů a práce s hodnotami na úrovni jednotlivých bitů či jejich skupin.

Jazyk SystemVerilog se během několika let vyvinul ve standardizovaný a všeobecně akceptovaný jazyk pro účely verifikace číslicových systémů. Díky tomu, že si osvojil některé prostředky z běžných vysokoúrovňových programovacích jazyků k zajištění znovupoužitelnosti kódu, začaly vznikat na něm postavené obecné metodiky pro tvorbu verifikačních prostředí. Nejúspěšnější z nich je v současnosti otevřená a volně dostupná metodika **Open Verification Methodology** (**OVM**), která poskytuje knihovnu základních a rozšířených tříd (komponent) pro tvorbu verifikačních prostředí v rámci balíku OVM 2.1.2. Tato metodika definuje, jakým způsobem implementovat lehce rozšiřitelné a znovu použitelné verifikační prostředí.

Jak uvidíme dále v cvičení, verifikační prostředí v OVM je uspořádáno hierarchicky, s nejnižší úrovní obsahující testovanou jednotku (angl. *Design Under Test* — DUT) se kterou probíhá komunikace přímo pomocí signálů na úrovni hradel, a vyšších vrstev, na nichž probíhá komunikace pomocí vysokoúrovňových transakcí. **Transakce** v OVM představují základní způsob komunikace mezi jednotlivými procesy verifikačního prostředí. Tento způsob práce s jednotlivými komponentami a jejich obousměrnou komunikací se nazývá modelování na úrovni transakcí (angl. *Transaction-level Modelling*).

Komponenty verifikačního prostředí zahrnují (Obrázek 1):

- nastavení vlastností a parametrů příslušného verifikačního testu (TESTCASE s parametry),
- řízení verifikačního běhu (SCENARIO LEVEL),
- vkládání automaticky generovaných vstupních transakcí (GENERATOR) na vstupní rozhraní verifikované jednotky (DRIVER),



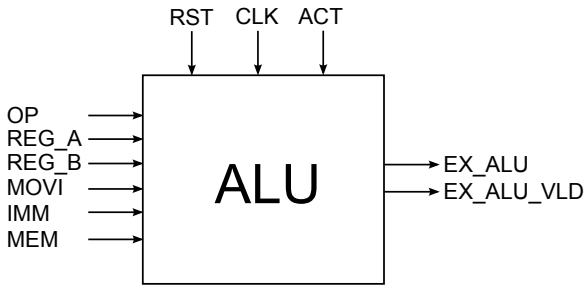
Obrázek 1: Schéma generického verifikačního prostředí

- sběr výstupních transakcí z výstupního rozhraní verifikované jednotky (MONITOR),
- sběr a analýzu statistik pro automaticky nebo ručně vytvořené body pokrytí ve vytvářeném modelu (COVERAGE),
- koncept sebekontrolního mechanismu (SCOREBOARD), v němž jsou na základě **specifikace** verifikovaného systému vytvářeny očekávané výstupní transakce (angl. *predicted responses*). Tyto jsou následně automaticky porovnány se skutečnými výstupními transakcemi systému (angl. *real responses*).

2 Popis verifikované jednotky ALU

Na Obrázku 2 je znázorněno rozhraní verifikované jednotky ALU. Toto rozhraní je tvořeno následujícími signály:

- řídicí:
  - `CLK [in]`: hodinový signal; všechny vstupy a výstupy ALU jsou aktivní na náběžné hraně tohoto signálu,
  - `RST [in]`: synchronní reset,
  - `ACT [in]`: aktivační vstup, hodnota `'1'` signalizuje požadavek na výpočet,



Obrázek 2: Rozhraní ALU jednotky

Tabulka 1: Význam hodnot vstupního signálu `OP` komponenty ALU

Hodnota	Význam
0x0	OUT = IN1 + IN2
0x1	OUT = IN1 - IN2
0x2	OUT = IN1 * IN2
0x3	OUT = IN2 >> 1
0x4	OUT = IN2 << 1
0x5	OUT = IN2 ROR 1
0x6	OUT = IN2 ROL 1
0x7	OUT = ~IN2
0x8	OUT = IN1 & IN2
0x9	OUT = IN1 | IN2
0xA	OUT = IN1 ^ IN2
0xB	OUT = ~(IN1 & IN2)
0xC	OUT = ~(IN1 | IN2)
0xD	OUT = ~(IN1 ^ IN2
0xE	OUT = IN2 + 1
0xF	OUT = IN2 - 1

- datové:
  - `OP(3:0) [in]`: volba operace ALU (viz Tabulka 1),
  - `REG_A(DATA_WIDTH-1:0) [in]`: vstup z registru jenž obsahuje první operand,
  - `MOVI(1:0) [in]`: vybírá druhý operand (viz Tabulka 2),
  - `REG_B(DATA_WIDTH-1:0) [in]`: vstup z registru jenž obsahuje druhý operand (`MOVI = "00"`),
  - `MEM(DATA_WIDTH-1:0) [in]`: vstup z paměti jenž obsahuje druhý operand (`MOVI = "01"`),
  - `IMM(DATA_WIDTH-1:0) [in]`: přímo zadaný druhý operand (`MOVI = "10"`),
- výstupy:
  - `EX_ALU_VLD [out]`: signál značící platnost výsledku (může být již tentýž takt jako je `ACT = '1'`),
  - `EX_ALU(15:0) [out]`: výsledek operace (hodnota je platná jen když `EX_ALU_VLD = '1'`); pro operace mimo násobení je výsledek předán v jednom taktu, u násobení je výsledek předán ve dvou taktech (v obou je signál `EX_ALU_VLD` nastaven do `'1'`, první je předána méně významná část výsledku).

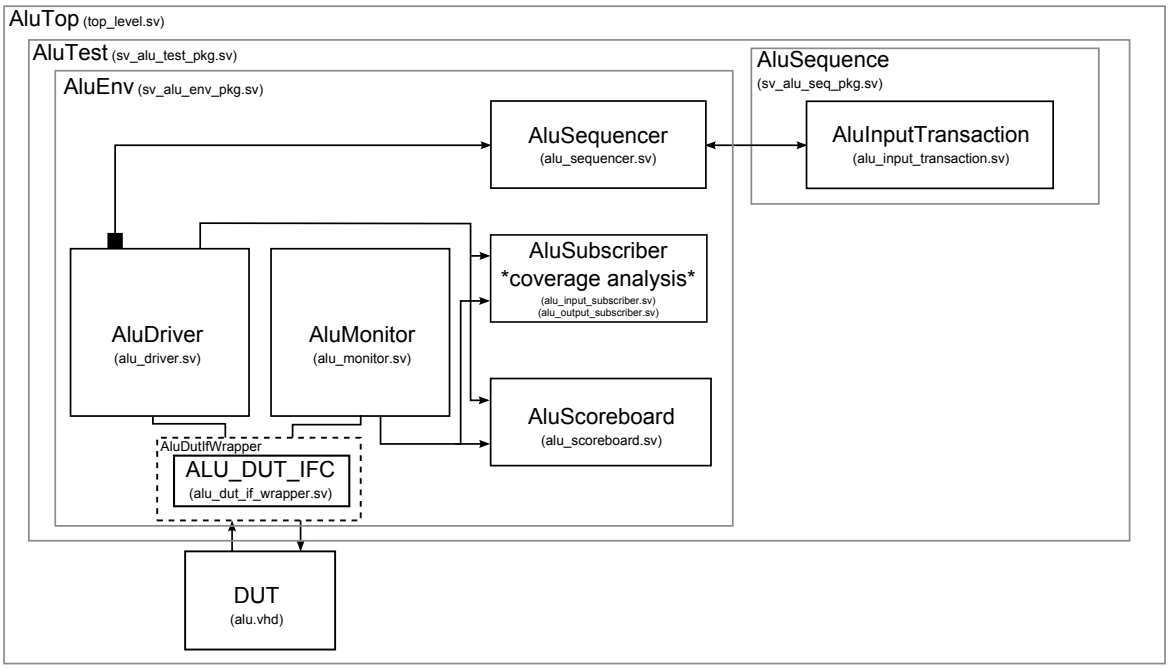
Tabulka 2: Význam hodnot vstupního signálu `MOVI` komponenty ALU

Hodnota	Význam
0x0	REG_B
0x1	MEM
0x2	IMM
0x3	<RESERVED>

3 Popis verifikačního prostředí ALU v OVM

Na Obrázku 3 je znázorněno verifikační prostředí pro ALU, které je rozděleno na tyto části:

`AluTop:` Na nejvyšší úrovni verifikačního prostředí dochází k instancování základních komponent, především testované jednotky (`DUT`), rozhraní pro komunikaci s ní (`ALU_DUT_IFC`), generování hodinového a resetovacího signálu a spuštění testu dle zadaných parametrů (délka generování resetu `RESET_TIME`, perioda hodinového signálu `CLK_PERIOD`, počet transakcí `TRANSACTION_COUNT` a nastavení počáteční hodnoty generátoru náhodných čísel `SEED`).



Obrázek 3: Verifikační prostředí pro ALU v OVM

`AluTest:` V této vrstvě se instancuje verifikační prostředí `AluEnv` a spouští se generování posloupnosti transakcí, jejichž formát je definován pomocí `AluSequence`. Komponenta `AluDriver` získává generované transakce z komponenty `AluSequencer`.

`AluSequence:` V rámci generování vstupních transakcí pro ALU je nejprve objektu třídy `AluSequence` předán prototyp (blueprint) transakce typu `AluInputTransaction`. Následně probíhá vytváření transakcí tím způsobem, že je vytvořena kopie prototypu transakce, jejíž datová

část je na základě omezujících podmínek vyplněna náhodnými daty či konstantami. Je vytvořena celková sekvence o maximálním počtu transakcí `TRANSACTION_COUNT` zadaném v parametrech verifikačního běhu.

`AluEnv:` Komponenta `AluDriver` přijímá sekvence transakcí z `AluSequencer`. S testovanou jednotkou (`DUT`) nekomunikuje přímo, ale skrze wrapper daného vstupního rozhraní, protože transakci je nutno transformovat na podobu bitových signálů. Transakce jsou také zaslány do komponenty `AluScoreboard`. Úlohou komponenty `AluMonitor` je příjem transakcí od wrapperu výstupního rozhraní ALU a jejich zasílání pro zpracování do komponenty `AluScoreboard`.

V komponentě `AluScoreboard` dochází k transformaci transakcí přijatých od komponenty `AluDriver` podle specifikace ALU a vytvoření očekávané výstupní transakce, jejíž formát je definován ve třídě `AluOutputTransaction`. Takto vytvořené transakce jsou automaticky porovnávány s reálnými transakcemi, které jsou přijaty z komponenty `AluMonitor`. V případě neshody je oznámena chyba, kterou je možné dále analyzovat. Hodnocení průběhu verifikace se provádí v komponentě `AluSubscriber`. Ta obsahuje tzv. body pokrytí, kterými verifikátor definuje, které vlastnosti, funkce či hraniční stavy systému chce v rámci verifikace ověřit. Tento proces se provádí s podporou samotného simulačního nástroje.

Jednotlivé komponenty jsou uspořádány hierarchicky v adresářích (viz také Příloha):

src	test_lib	seq_lib	env_lib

4 Úlohy k řešení

Úloha 1 — oprava chyby v testované komponentě.

Následujícím příkazem v příkazovém řádku spustíte verifikační proces v nástroji ModelSim:

```
# vsim -do simulation.fdo
```

V této fázi není potřebné měnit nastavení verifikačního běhu ani další parametry. Po spuštění se zobrazí textový výpis s průběhem verifikace, který skončí chybou. Na základě vypsaných informací odhalte zdroj chyby, opravte ji v příslušném zdrojovém souboru a ověřte opravu chyby opakovaným spuštěním verifikačního procesu.

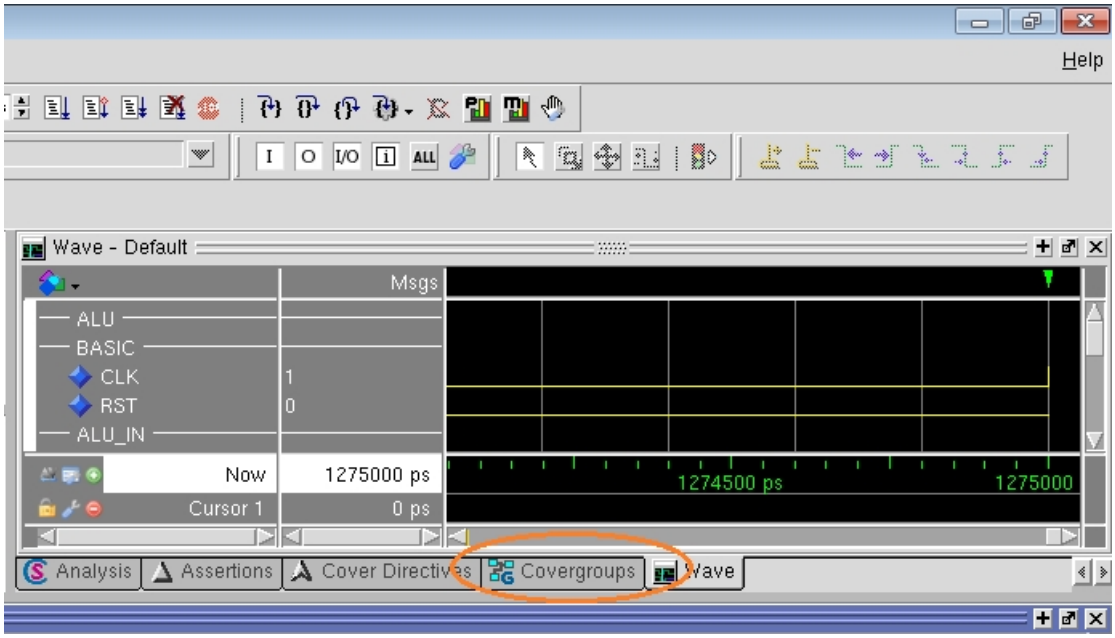
Nápověda: Všimněte si vstupů a řídicích signálů ALU. Proč je očekávaný a získaný výsledek odlišný?

Úloha 2 — oprava chyby ve verifikačním prostředí.

Po opravení chyby spusťte verifikační proces znovu příkazem v ModelSimu:

```
VSIM(paused)> do simulation.fdo
```

Zobrazí se chyba související s neznámým nastavením operandu. V tomto kroku bude potřeba zasáhnout do verifikačního prostředí.



Obrázek 4: Skupiny pokrytí

Nápověda: Opět pozorně sledujte výpis z verifikace. Pozornost směřujte i na příslušný řádek v souboru `alu_scoreboard.sv` a jeho analýzu. Proč dochází k chybě s neznámým operandem?

Úloha 3 — doplnění bodu pokrytí.

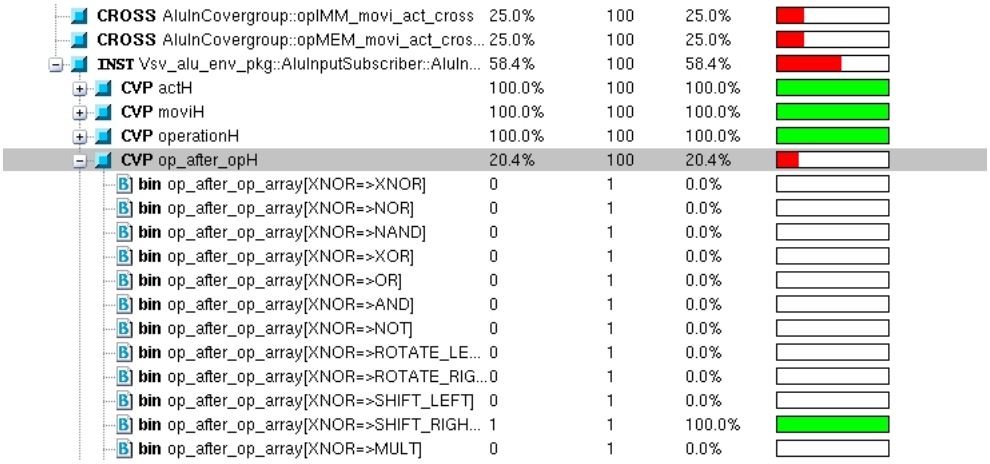
Po nalezení zdroje chyby spusťte verifikaci jako v předchozím případě. V tomto případě už proběhne celý verifikační běh s aktuálními nastaveními v pořádku. Zobrazte si záložku Covergroups v okně Wave v prostředí ModelSimu, Obrázek 4.

V zobrazeném okně je seznam bodů pokrytí definovaných v souboru `alu_input_subscriber.sv`. Rozšiřte tento seznam o další bod pokrytí, ve kterém bude sledováno pokrytí posloupnosti operandů (pouze sledování vytváření párů posloupností, tj. přechodů z jednoho operandu do druhého). Bod pokrytí nazvěte `op_after_op` a spusťte verifikaci znovu.

Nápověda: Jazyk SystemVerilog umožňuje definovat skupiny bodů pokrytí s jednotlivými body pokrytí, které jsou sledovány v simulačním nástroji. Bod pokrytí umožňuje definovat hodnotu (nebo množinu hodnot), které je potřeba během verifikace sledovat. Body pokrytí se definují jako tzv. „koše“ (angl. *bins*), pro které se počítá počet shod.

Příklad:

```
int i;
covergroup range_cover;
  coverpoint i {
  bins zero = {0};                  // počet shod s hodnotou 0
} endgroup
```



Obrázek 5: Sledování posloupnosti operací

Také je možno vytvořit bod pokrytí, kterým se sledují přechody mezi hodnotami, počet přechodů není omezen.

Příklad:

```
covergroup CoverPort;
coverpoint port {
  bins t1 = (0 => 1), (0 => 2), (0 => 3);
} endgroup
```

V předchozím příkladu se pomocí koše `t1` dá sledovat počet přechodů signálu `port` z hodnoty 0 do hodnot 1, 2, nebo 3.

Snažte se zjednodušit práci tak, aby nebylo nutné ručně definovat dvojice posloupností. Vytváříte konečnou množinu možných přechodů, použijte vhodný datový typ. Výsledný zápis je velmi jednoduchý, inspirujte se již vytvořenými body pokrytí.

Úloha 4 — zvýšení úrovně pokrytí.

Opět si zobrazte okno bodů pokrytí a všimněte si, že do seznamu přibyl nově definovaný bod, viz Obrázek 5.

Také si všimněte seznamu vytvořených dvojic posloupností. Verifikace proběhla v pořádku, ale celkové pokrytí je velmi malé. Pokuste se zvýšit pokrytí na 100 % u všech bodů.

Nápověda: Zvýšení pokrytí se typicky dosahuje delším verifikačním během. Experimentálně zkuste (orientačně) najít nejmenší hodnotu počtu transakcí nutných k dosažení 100 % pokrytí změnou nastavení parametrů verifikačního testu.

Úloha 5 — modifikace ALU a verifikačního prostředí.

V posledním kroku rozšíříme možnosti jednotky ALU o další dvě operace a provedeme potřebné změny i ve verifikačním prostředí. Operace ALU je možno řídit 4-bitovým signálem, využívá se však pouze 14 z celkových 16 možných. ALU rozšíříme o operace inkrementu a dekrementu.

1. Doplňte zdrojový VHDL soubor ALU jednotky o operace inkrementu a dekrementu.
2. Protože v procesu verifikace je potřeba ve verifikačním prostředí vytvářet model verifikované jednotky, je potřeba doplnit generování nových dvou operací i do příslušného bloku verifikačního prostředí.
3. Upravte omezení (constraint) týkající se počtu generovaných operací a zvyšte tuto hodnotu o další dvě operace. *Nápověda*: úprava se týká souboru, ve kterém jste již opravovali chybu související s generováním neznámého operandu (úloha 2).
4. V souboru `alu_input_subscriber.sv` upravte seznam definující výčtový typ pro generované operace. Nezapomeňte upravit vytvořený bod pokrytí pro sledování posloupnosti dvojic generovaných operací.
5. Opět analyzujte dosažené pokrytí a najděte experimentálně počet transakcí potřebných k dosažení úrovně 100 % pokrytí.

5 Příloha

Hierarchie nejdůležitějších souborů

```
|-- env_lib
    |-- alu_driver.sv
    |-- alu_dut_if_wrapper.sv
    |-- alu_env.sv
    |-- alu_input.sv
    |-- alu_input_subscriber.sv    // coverage points
    |-- alu_monitor.sv
    |-- alu_output_subscriber.sv
    |-- alu_scoreboard.sv          // scoreboarding
    |-- alu_sender.sv
    `-- sv_alu_env_pkg.sv
```

```
|-- seq_lib
    |-- alu_input_transaction.sv   // constraints
    |-- alu_output_transaction.sv
    |-- alu_sequence.sv
    |-- alu_sequencer.sv
    |-- haven_input_transaction.sv
    |-- haven_output_transaction.sv
    |-- haven_sequence_item.sv
    `-- sv_alu_seq_pkg.sv
```

```
 |-- src
     |-- alu
     |   |-- Modules.tcl
     |   |-- alu.vhd    // zdrojový kód ALU
     |
     `-- mult
         |-- mult.vhd
         |-- simulation.fdo
         `-- testbench.vhd
```

```
|-- test_lib
    |-- alu_test.sv
    `-- sv_alu_test_pkg.sv
```

```
.
|-- simulation.fdo      // spuštění simulace
|-- test_parameters.sv  // nastavení parametrů testu
`-- top_level.sv
```