Vérification fonctionnelle des systèmes numériques

Scripts

Yann Thoma

Reconfigurable and Embedded Digital Systems Institute
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

Février 2017

Vérification automatique

- But: vérification la plus complète possible du système
- Résultat Go/No Go (OK, KO)
- Indication de l'instant d'une erreur
- Possibilité de stopper la simulation sur une erreur
- Indication du nombre total d'erreurs
- Permet une re-vérification complète (Go/no Go) après une modification ou une correction
- Les interactions utilisateur doivent être minimisées
  - Nécessaire pour garantir la similarité des tests effectués
  - $\Rightarrow$ Automatisation via des scripts

La suite est spécifique au logiciel QuestaSim

Simulation automatique: scripts

- Une simulation doit pouvoir être lancée via des scripts
- Evite des erreurs de manipulation
- Simplifie la vie

Exemple: sim.do

```
# définition de la bibliothèque de travail
vlib work
# compilation du design à tester
vcom counter.vhd
# compilation du banc de test
vcom counter_tb.vhd
# lancement de la simulation
vsim work.counter_tb
# ajout de tous les signaux au chronogramme
add wave -r *
# exécution de la simulation
run -all
```

Type d'exécution

- Via le terminal de l'interface graphique:

```
do sim.do
```

Type d'exécution

- Via le terminal de l'interface graphique:

```
do sim.do
```

- En ligne de commande, en lançant QuestaSim:

```
>>vsim -do sim.do
```

Type d'exécution

- Via le terminal de l'interface graphique:

```
do sim.do
```

- En ligne de commande, en lançant QuestaSim:

```
>>vsim -do sim.do
```

- En ligne de commande, en lançant QuestaSim en mode ligne de commande:

```
>>vsim -c -do sim.do
```

Type d'exécution

- Via le terminal de l'interface graphique:
```
do sim.do
```
- En ligne de commande, en lançant QuestaSim:
```
>>vsim -do sim.do
```
- En ligne de commande, en lançant QuestaSim en mode ligne de commande:
```
>>vsim -c -do sim.do
```
- En ligne de commande, en lançant QuestaSim en mode ligne de commande, en passant un argument:
```
>>vsim -c -do "do sim.do unargument"
```

Commandes standards (1)

Commande do

- Syntaxe:

```
do <filename> [<parametres>]
```

- Description:
  - La commande do exécute le script contenu dans le fichier passé en paramètre
- Exemple:
  - Lancement du script de compilation

```
do vhdl_compile.do
```

  - Chargement du format du chronogramme

```
do wave.do
```

Commandes standards (2)

Commande `vlib`

- Syntaxe:

```
vlib <libname>
```

- Description:
  - La commande `vlib` crée une nouvelle bibliothèque de design (*design library*). Elle crée notamment le répertoire de la librairie
- Exemple:
  - Création d'un répertoire pour la bibliothèque par défaut *work*

```
vlib work
```

  - Création d'un répertoire pour la bibliothèque d'un paquetage

```
vlib monPaquetage
```

Commandes standards (3)

Commande `vmap`

- Syntaxe:

```
vmap <libname> <dirname>
```

- Description:
  - La commande `vmap` définit un mapping entre le nom de bibliothèque `<libname>` et son emplacement sur le disque `<dirname>`.
- Exemple:
  - Mapper une bibliothèque générale pour un projet particulier

```
vmap monPaquetage /proj/monPaquetage
```

- Remarque:
  - Il n'est pas nécessaire d'effecuer `vmap work work` (normalement)

Commandes standards (4)

Commande vcom

- Syntaxe:

```
vcom
[-87] [-93] [-2002] [-2008]      Choix de la norme
[-work <libname>]                Nom de bibliothèque
                                 à utiliser
                                 Par défaut: work

<filename>
```

- Description:
  - La commande vcom compile les sources VHDL passées en paramètre dans le répertoire indiqué par la bibliothèque <libname>
- Exemple:
  - Compilation de deux fichiers dans *work*

```
vcom -2008 -work work add.vhd add_tb.vhd
```

Commandes standards (5)

Commande `vlog`

- Syntaxe:

```
vlog  [-work <libname>] [+incdir+<dirname>]
      <filename>
```

- Description:
  - La commande `vlog` compile les sources SystemVerilog (ou Verilog) passées en paramètre dans le répertoire indiqué par la bibliothèque `<libname>`. Il est possible d'indiquer des répertoires contenant des dépendances du projet
- Exemple:
  - Compilation d'un ficher dans *work*

```
vlog -work work top.sv
```

  - Compilation avec inclusion de répertoire

```
vlog -work work +incdir+unvip/ testcase.sv
```

Commandes standards (6)

Commande `vsim`

- Syntaxe:

```
vsim  [-t [<multiplier>] <timeunit>]
      <libraryname>.<designunit>
      [.<architecture>]
```

- Description:
  - La commande `vsim` lance le simulateur et charge le design spécifié.
  - Il est possible de spécifier le pas de simulation, qui est par défaut de 1ns.
- Exemple:
  - Chargement d'un design

```
vsim work.add_tb
```

  - Chargement d'un design avec pas de simulation de 10ps

```
vsim -t 10 ps work.add_tb
```

Commandes standards (6b)

Commande `vsim`

- Possibilité de passer des paramètres génériques au banc de test

```
vsim  -Gparam1=value1 -Gparam2=value2 ...
```

- Description:
  - Les paramètre génériques sont passés en préfixant l'affectation avec `-G`
  - Permet d'avoir un banc de test pour l'exécution de plusieurs tests

Commandes standards (6c)

Commande `vsim`

- Possibilité de passer des paramètres génériques au banc de test
- Exemple pour un banc de test ayant un paramètre générique `SIZE`:

```
entity add_tb is
generic ( SIZE : integer := 8);
port ( ... );
end add_tb;
```

  - Chargement d'un design pour un additionneur de 16 bits

```
vsim -GSIZE=16 work.add_tb
```

  - Chargement d'un design pour un additionneur de 32 bits

```
vsim -GSIZE=32 work.add_tb
```

Commandes standards (7)

Commande `view`

- Syntaxe:

```
view <window_type>
```

- Description:
  - La commande `view` permet d'ouvrir une fenêtre dans QuestaSim.
  - `<window_type>` correspond au nom de la fenêtre:
    - memory, process, signals, source, structure, variables, wave
- Exemple:
  - Ouverture de la fenêtre des signaux

```
view signals
```

  - Ouverture du chronogramme

```
view wave
```

Commandes standards (8)

Commande `add wave`

- Syntaxe:

```
add wave ...
```

- Description:
  - La commande `add wave` permet d'ajouter des signaux ou variables au chronogramme
- Exemple:
  - Ajout de tous les signaux

```
add wave *
```

  - Ajout d'un séparateur

```
add wave -divider Stimuli
```

  - Ajout de signaux à un groupe

```
add wave -expand -group Stimuli *_sti
```

Commandes standards (9)

Commande `run`

- Syntaxe:

```
run [<timesteps>[<time_units>]] | [-all]
```

- Description:
  - La commande `run` lance la simulation proprement dite. Il est possible de spécifier le temps total de simulation ou de la lancer *ad eternum*
- Exemple:
  - Lancement pour 1000 ns

```
run 1000 ns
```

  - Lancement pour détection automatique de fin

```
run -all
```

Scripts paramétrables

- Toutes les étapes ne sont pas forcément nécessaires
  - Par exemple: pas besoin de recompiler si on ne change qu'un fichier d'input
- Décomposition du fichier de script, grâce à Tcl/Tk
- Possibilité d'analyser les arguments de l'appel pour automatisation complète

Scripts paramétrables

- Nombre d'arguments: variable `$argc`
- Les différents arguments: `$1`, `$2` `$3`, ...

Exemple

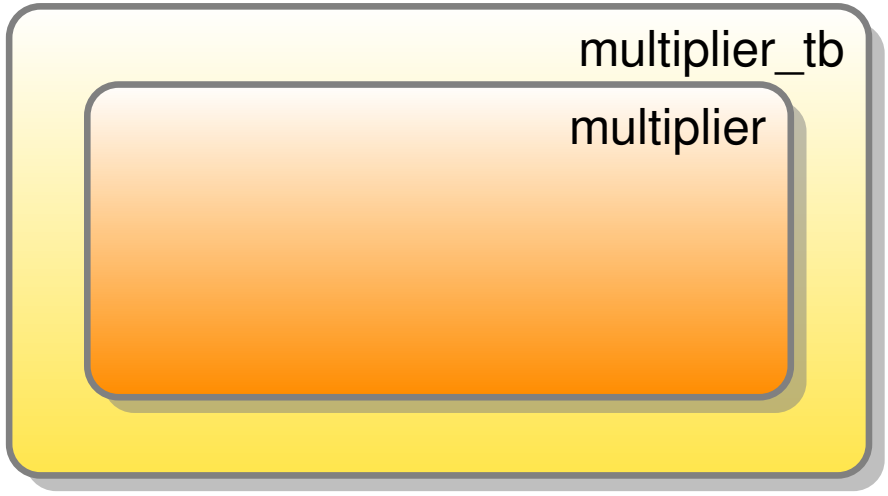
```
if {$argc > 0} {          ← S'il y a un argument
    if {[string compare $1 "action1"] == 0} {
        # do action 1
    } elseif {[string compare $1 "run"] == 0} {
        if ($argc == 2)
            vsim -GSIZE=$2    ← On passe l'argument 2 comme générique
        else
            puts -nonewline "Missing an argument"
    }
else
    puts -nonewline "Missing an argument"
```

Scripts: exemple (1)

- Décomposition du fichier de script en fonctions:
  - Chargement de bibliothèques
  - Compilation VHDL
  - Compilation SystemVerilog
  - Simulation
  - Exécution totale
  - Programme principal
- Exemple: Multiplicateur

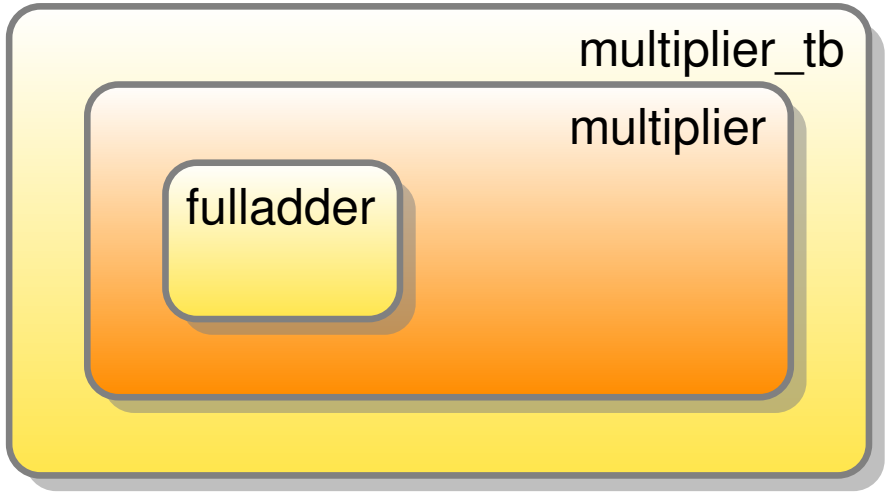
Scripts: exemple (1)

- Décomposition du fichier de script en fonctions:
  - Chargement de bibliothèques
  - Compilation VHDL
  - Compilation SystemVerilog
  - Simulation
  - Exécution totale
  - Programme principal
- Exemple: Multiplicateur



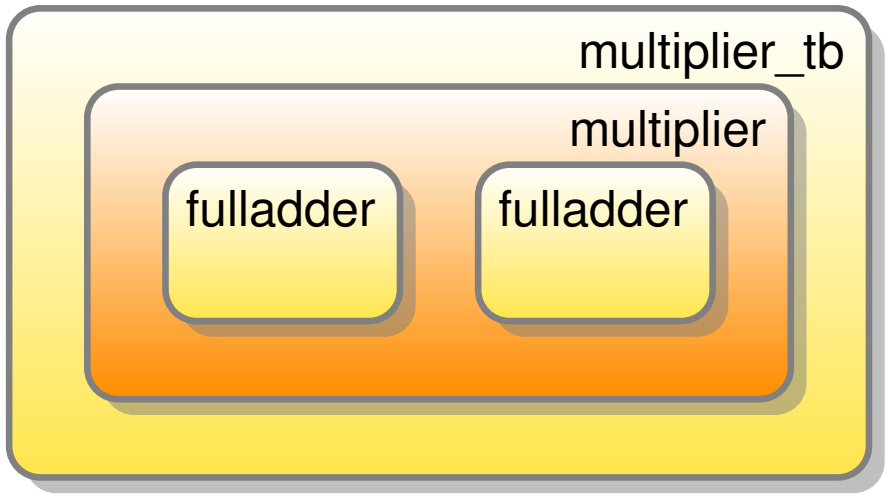
Scripts: exemple (1)

- Décomposition du fichier de script en fonctions:
  - Chargement de bibliothèques
  - Compilation VHDL
  - Compilation SystemVerilog
  - Simulation
  - Exécution totale
  - Programme principal
- Exemple: Multiplicateur



Scripts: exemple (1)

- Décomposition du fichier de script en fonctions:
  - Chargement de bibliothèques
  - Compilation VHDL
  - Compilation SystemVerilog
  - Simulation
  - Exécution totale
  - Programme principal
- Exemple: Multiplicateur



Scripts: exemple (2)

Programme principal (1)

```
if {[file exists work] == 0} {
    vlib work
}

puts -nonewline "  Path_VHDL => "
set Path_VHDL     "./src_vhdl"
global Path_VHDL

puts -nonewline "  Path_TB => "
set Path_TB     "./src_tb"
global Path_TB
```

Crée la bibliothèque si nécessaire

Affecte une variable globale

Affecte une variable globale

Scripts: exemple (3)

Programme principal (2)

```
if {$argc==1} {
    if {[string compare $1 "all"] == 0} {
        do_all
    } elseif {[string compare $1 "comp_vhdl"] == 0} {
        vhdl_compil
    } elseif {[string compare $1 "comp_sv"] == 0} {
        sv_compil
    } elseif {[string compare $1 "sim"] == 0} {
        sim_start
    } else { puts "Unsupported argument" }
}
else {
    do_all
}
```

S'il y a un argument

Scripts: exemple (4)

Compilation VHDL

```
proc vhdl_compil { } {
  global Path_VHDL
  global Path_TB

  puts "\nVHDL compilation :"

  vcom $Path_VHDL/fulladder.vhd
  vcom $Path_VHDL/multiplier.vhd
  vcom $Path_TB/multiplier_tb.vhd
}
```

Variable globale

Variable globale

Scripts: exemple (5)

Simulation

```
proc sim_start { } {

  vsim -t 1ns -novopt work.multiplier_tb
#  do wave.do
  wave refresh
  run -all
}
```

Typiquement généré par QuestaSim

Do All

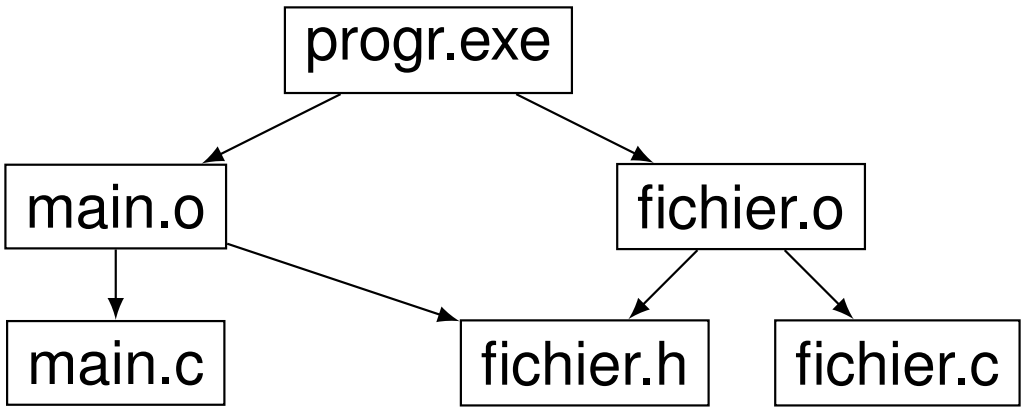
```
proc do_all { } {
  vhdl_compil
  sv_compil
  sim_start
}
```

Scripts interactifs

- Il est possible de créer des scripts interactifs
  - L'utilisateur peut rentrer des données et interagir
- Pas utile pour l'automatisation
- Problème de terminal sous Linux
  - Terminal vs. console QuestaSim
- Passer des arguments semble être une meilleure option

Makefile

- Le monde logiciel exploite des Makefile
  - Permettent de réduire le processus de compilation au minimum
  - Expriment des dépendances entre fichiers
  - Ne sont exécutées que les étapes de compilation nécessaires à l'ensemble des fichiers modifiés
  - Les outils IDE génèrent automatiquement des Makefile pour les projets logiciels
- Pourquoi ne pas en profiter dans le monde matériel?
  - Un peu délicats à gérer à la main

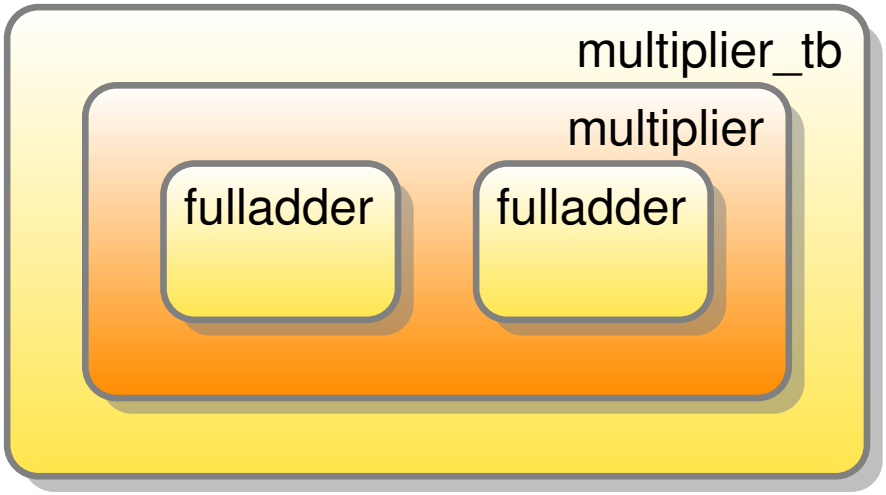


Makefile: QuestaSim

- Commande disponible: `vmake`
- A exécuter dans le répertoire de travail
- Après une compilation complète du projet
- Scanne le répertoire `work`
- Génère un fichier `Makefile`
- Exécuter ensuite la commande `make` lance une compilation conditionnelle
- Problème: Doit être réexécutée lors d'ajouts de fichiers

Makefile: Exemple (1)

- Une hiérarchie:




```
# Generated by vmake version 2.2

# Define path to each library
LIB_STD = /opt/questasim/linux/../std
LIB_IEEE = /opt/questasim/linux/../ieee
LIB_WORK = work
```

Makefile: Exemple (2)

```
# Define path to each design unit
IEEE__std_logic_1164 = $(LIB_IEEE)/std_logic_1164/_primary.dat
IEEE__numeric_std = $(LIB_IEEE)/numeric_std/_primary.dat
IEEE__std_logic_textio = $(LIB_IEEE)/std_logic_textio/_primary.dat
STD__textio = $(LIB_STD)/textio/_primary.dat
IEEE__std_logic_arith = $(LIB_IEEE)/std_logic_arith/_primary.dat
IEEE__std_logic_unsigned = $(LIB_IEEE)/std_logic_unsigned/_primary.dat
WORK__multiplier_tb__comp = $(LIB_WORK)/multiplier_tb/comp.dat
WORK__multiplier_tb = $(LIB_WORK)/multiplier_tb/_primary.dat
WORK__multiplier__struct = $(LIB_WORK)/multiplier/struct.dat
WORK__multiplier = $(LIB_WORK)/multiplier/_primary.dat
WORK__fulladder__comp = $(LIB_WORK)/fulladder/comp.dat
WORK__fulladder = $(LIB_WORK)/fulladder/_primary.dat
VCOM = vcom
VLOG = vlog
VOPT = vopt
SCCOM = sccom
```

Makefile: Exemple (3)

```
whole_library :     $(WORK__multiplier_tb__comp) \
    $(WORK__multiplier_tb) \
    $(WORK__multiplier__struct) \
    $(WORK__multiplier) \
    $(WORK__fulladder__comp) \
    $(WORK__fulladder)

$(WORK__fulladder) \
$(WORK__fulladder__comp) : fulladder.vhd \
        $(IEEE__std_logic_unsigned) \
        $(IEEE__std_logic_arith) \
        $(IEEE__std_logic_1164)
    $(VCOM) -2002 -explicit fulladder.vhd
```

Makefile: Exemple (4)

```
$(WORK__multiplier) \
$(WORK__multiplier__struct) : multiplier.vhd \
        $(IEEE__std_logic_unsigned) \
        $(IEEE__std_logic_arith) \
        $(IEEE__std_logic_1164)
    $(VCOM) -2002 -explicit multiplier.vhd

$(WORK__multiplier_tb) \
$(WORK__multiplier_tb__comp) : multiplier_tb.vhd \
        $(IEEE__std_logic_unsigned) \
        $(IEEE__std_logic_arith) \
        $(WORK__multiplier) \
        $(STD__textio) \
        $(IEEE__std_logic_textio) \
        $(IEEE__numeric_std) \
        $(IEEE__std_logic_1164)
    $(VCOM) -2002 -explicit multiplier_tb.vhd
```

Dépend de multiplier

Makefile: Conclusion

- Les Makefile peuvent être automatiquement générés
- Ou écrits à la main
- Peuvent être utiles pour de gros projets
- Sinon le temps de compilation est négligeable en comparaison de celui de simulation
- Les fichiers de script sont plus facilement éditables

Conclusion

- Les scripts permettent d'automatiser la simulation
- Ils font gagner du temps
- Ils permettent d'éviter des erreurs
- $\Rightarrow$ A utiliser en toute circonstance
- Attention à les rendre les plus portables possibles
  - $\Rightarrow$ Chemins de fichiers relatifs
- Un collègue doit pouvoir l'exécuter tel quel
  - Nécessaire pour l'exploitation d'un gestionnaire de version par une équipe