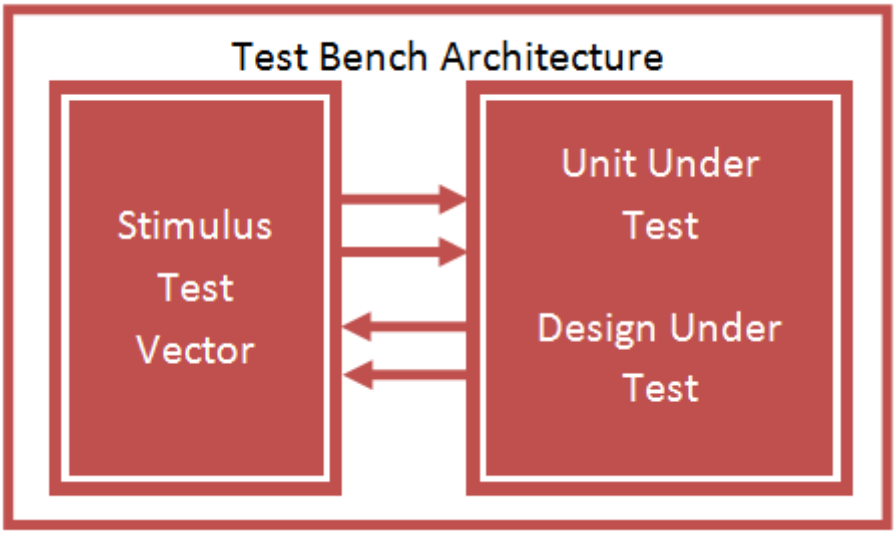
Rapport de vérification et tests matériels

DIMITRI PETITJEAN SEE



ENSEIRB-MATMECA

12/02/2021

ENSEIGNANT REFERENT : Bertrand LE GAL

Objectifs du module :

Etudier les différentes possibilités et méthodes de tests existantes pour du hardware et du software, comment les combiner pour optimiser ces tests. L'objectif est également de vérifier quelles sont les limites et contraintes des différents types de tests.

I. Software

A. Etape 1

La première étape consiste à implémenter l'algorithme du PGCD en langage C. Voici le code de l'algorithme :

```c
int PGCD(int A, int B)
{
    while(A != B){
        if(A > B){
            A = A-B;
        }else{
            B = B-A;
        }
    }
    return A;
}
```

Figure 1 : Fonction PGCD

Le « main » permettant de tester ce programme est le suivant :

```c
int main (int argc, char * argv []){
    printf("(II) Starting PGCD program\n");

    int A = atoi(argv[1]);
    int B = atoi(argv[2]);

    int pgcd = PGCD(A, B);

    printf("Le PGCD de %d et %d est %d\n", A, B, pgcd);

    printf("(II) End of PGCD program\n");
  return 0;
}
```

Figure 2 : Fonction main permettant de tester le programme

Je n'ai eu aucune difficulté à écrire ce programme, il a fonctionné du premier coup. Voici les tests réalisés avec quelques couples de valeurs :

```
dimitri@LAPTOP-KP4Q2AO4:/mnt/c/Users/dimit/Documents/Cours ENSEIRB/Vérification tests matériels/EN224-Test-et-verification/1_Software/Etape
$ ./src/main 14 15
Le PGCD de 14 et 15 est 1
dimitri@LAPTOP-KP4Q2AO4:/mnt/c/Users/dimit/Documents/Cours ENSEIRB/Vérification tests matériels/EN224-Test-et-verification/1_Software/Etape
$ ./src/main 65535 65534
Le PGCD de 65535 et 65534 est 1
dimitri@LAPTOP-KP4Q2AO4:/mnt/c/Users/dimit/Documents/Cours ENSEIRB/Vérification tests matériels/EN224-Test-et-verification/1_Software/Etape
$ ./src/main 65535 500
Le PGCD de 65535 et 500 est 5
dimitri@LAPTOP-KP4Q2AO4:/mnt/c/Users/dimit/Documents/Cours ENSEIRB/Vérification tests matériels/EN224-Test-et-verification/1_Software/Etape
$ ./src/main 7389 8279
Le PGCD de 7389 et 8279 est 1
dimitri@LAPTOP-KP4Q2AO4:/mnt/c/Users/dimit/Documents/Cours ENSEIRB/Vérification tests matériels/EN224-Test-et-verification/1_Software/Etape
$ ./src/main 2910 7367
Le PGCD de 2910 et 7367 est 1
dimitri@LAPTOP-KP4Q2AO4:/mnt/c/Users/dimit/Documents/Cours ENSEIRB/Vérification tests matériels/EN224-Test-et-verification/1_Software/Etape
$
```

Figure 3 : Test de l'algorithme de PGCD

Les valeurs sont cohérentes, l'algorithme semble être fonctionnel. Pour une valeur de A ou B négative, le programme rentre dans une boucle infinie. Pour A ou B qui vaut 0, il se passe la même chose.

B. Etape 2

A 2000 valeurs de tests, une boucle infinie se déclenche sur le programme. Pour pallier à cela, on va venir gérer le cas où une des valeurs vaut zéro. La seule difficulté que j'ai eu sur la rédaction du code ici est d'implémenter la gestion du zéro, je l'avais placé dans le « main » et pas directement dans la fonction PGCD.

```
if(A == 0){
    return B;
}else if(B == 0){
    return A;
}
```

Figure 4 : Gestion du zéro

C. Etape 3

On considère que la comparaison avec le résultat d'un autre algorithme de PGCD permettrait de vérifier que l'algorithme fonctionne quelque soit le nombre de tests. Le test de comparaison des valeurs de retour des deux algorithmes permet de vérifier si le PGCD obtenu est cohérent, sinon on affiche une erreur sur le test.

```
int PGCD_2(int A, int B){

    if(A == 0){
        return B;
    }else if(B == 0){
        return A;
    }

    while(B != 0){
        A = B;
        B = A % B;
    }

    return A;
}
```

Figure 5 : Implémentation du deuxième algorithme de PGCD

```c
int main (int argc, char * argv []){
    int nb_tests = atoi(argv[1]);

    while(nb_tests != 0){

        int32_t A = rand() % 65536;
        uint32_t B = rand() % 65536;

        int pgcd = PGCD(A, B);
        int pgcd_2 = PGCD_2(A, B);

        if(pgcd == pgcd_2){
            printf("Le PGCD de %d et %d est %d\n", A, B, pgcd);
        }else{
            break;
            printf("Pour A = %d, B = %d, PGCD trouve %d et PGCD_2 trouve %d\n", A, B, pgcd, pgcd_2);
        }

        nb_tests = nb_tests - 1;
    }
  return 0;
}
```

Figure 6 : Fonction main permettant le test des deux PGCD

L'avantage est que la probabilité que le test soit fiable est très élevée, en revanche, la probabilité qu'il soit fiable n'est pas de 100%, on peut donc avoir une erreur sur les deux algorithmes et se retrouver avec un résultat de test erroné.

D. Etape 4

J'ai paramétré des assertions dans la fonction PGCD qui vérifient bien que A et B soient compris entre 1 et 65535. L'intérêt de ces assertions est de tester le programme pour vérifier qu'il ne passe pas par des valeurs incohérentes ou problématiques sur l'algorithme. Il faut en revanche pour cela déterminer au préalable quelles sont ces valeurs.

```c
int PGCD(int A, int B)
{
    assert(A >= 0);
    assert(B >= 0);
    assert(A <= 65535);
    assert(B <= 65535);

    while(A != B){

        if(A == 0){
            return B;
        }else if(B == 0){
            return A;
        }

        if(A > B){
            A = A-B;
        }else{
            B = B-A;
        }
    }
    return A;
}
```

Figure 7 : Code de la fonction PGCD avec assertions

E. Etape 5

L'intérêt de mettre des postconditions dans le code est d'éviter une erreur en sortie de fonction si jamais on a oublié une précondition ou s'il y a eu un problème dans le fonctionnement. La limite des postconditions est qu'il est difficile de tester toutes les valeurs particulières après calcul pour vérifier que l'assertion soit bonne et/ou fonctionnelle.

```
assert(A > 0);
assert(A == B);
assert((t_A % A) == 0);
assert((t_B % B) == 0);
```

Figure 8 : Assertions postconditions

J'ai choisi de vérifier que A soit strictement positif pour commencer afin de vérifier que s'il vaut 0, on quitte bien la fonction par la partie de gestion du 0. L'assertion A == B permet de vérifier que l'algorithme se termine bien avec A et B égaux. Les deux dernières assertions permettent de vérifier que les A et B restants sont bien des diviseurs de la valeur stockée en entrée par t_A et t_B.

F. Etape 6

Une modification du Makefile est nécessaire afin d'intégrer les fichiers pgcd.h et pgcd.c qui ont été créés.

```
CC=gcc
LDFLAGS=-lm
EXEC=main
SRC=./src/main.c ./src/pgcd.c
DEPS=./src/pgcd.h
```

Figure 9 : Modifications apportées au Makefile pour tenir compte des nouveaux fichiers

J'ai également défini des tests unitaires dans la fonction main et ce afin de vérifier le bon fonctionnement de l'algorithme de PGCD.

```
assert(PGCD(0,5) == 5);
assert(PGCD(5,0) == 5);
assert(PGCD(0,0) == 0);
assert(PGCD(3,9) == 3);
assert(PGCD(9,3) == 3);
assert(PGCD(10,10) == 10);
```

Figure 10 : Tests unitaires réalisés pour vérifier le fonctionnement de l'algorithme

Les tests unitaires permettent de vérifier le fonctionnement global de notre programme, les cas généraux comme les cas particuliers. On ne peut en revanche pas tester toutes les possibilités.

G. Etape 7

Une modification du Makefile et des extensions des fichiers est nécessaires afin de basculer le programme en langage C++ afin de le rendre compatible avec le framework. On définit des objets appelés « require » et qui fonctionnent comme des tests unitaires. J'ai séparé comme demandé les test case afin de catégoriser les tests visant à vérifier le fonctionnement classique de l'algorithme, l'autre partie vise à tester les cas particuliers.

L'utilisation d'un framework permet de vérifier le fonctionnement du programme suivant les paramètres d'entrée que l'on passe, il permet en cas de non validation des tests unitaires de situer précisément le problème en catégorisant les tests que l'on souhaite réaliser. L'inconvénient est que l'on ne peut pas tester toutes les valeurs et que l'on peut en omettre, ce qui impacte la fiabilité du test.

```
CC=g++
LDFLAGS=-lm
EXEC=main
SRC=./src/main.cpp ./src/pgcd.cpp
DEPS=./src/pgcd.h
```

```
%.o: %.cpp
    $(CC) $(CFLAGS) -o $@ -c $< $(CFLAGS)
```

Figure 11 : Modifications apportées au Makefile

```
TEST_CASE ( "PGCD classique", "[pgcd]" ){

  SECTION(" A > B "){
    REQUIRE(PGCD(9,3) == 3);
    REQUIRE(PGCD(11,5) == 1);
    REQUIRE(PGCD(22,8) == 2);
  }

  SECTION(" A < B "){
    REQUIRE(PGCD(3,9) == 3);
    REQUIRE(PGCD(5,11) == 1);
    REQUIRE(PGCD(8, 22) == 2);
  }

  SECTION(" A = B "){
    REQUIRE(PGCD(9,9) == 9);
    REQUIRE(PGCD(11,11) == 11);
    REQUIRE(PGCD(22,22) == 22);
  }
}
```

Figure 12 : Tests pour le déroulement standard de l'algorithme

```
TEST_CASE ("PGCD particulier", "[pgcd]"){
  SECTION(" A = 0 "){
    REQUIRE(PGCD(0,3) == 3);
    REQUIRE(PGCD(0,5) == 5);
    REQUIRE(PGCD(0,8) == 8);
  }

  SECTION(" B = 0 "){
    REQUIRE(PGCD(3,0) == 3);
    REQUIRE(PGCD(5,0) == 5);
    REQUIRE(PGCD(8,0) == 8);
  }

  SECTION(" A = B = 0 "){
    REQUIRE(PGCD(0,0) == 0);
    REQUIRE(PGCD(0,1) == 2); // fausse assert pour vérifier le blocage en cas d'erreur
  }
}
```

Figure 13 : Tests réalisés pour les cas particuliers de l'algorithme

H. Etape 8

L'implémentation en C de cette étape relève de la manipulation de fichiers. J'ai généré les valeurs des 3 fichiers de référence sous Excel grâce à la fonction ALEA.ENTRE.BORNES(), le fichier de référence de résultats utilise la fonction PGCD préimplantée dans Excel.

L'intérêt de cette méthode est de pouvoir tester un grand nombre de données ce qui permet de rendre le test très fiable, de plus, on vient comparer le fichier de résultats avec le fichier de référence ce qui garantit la validité des résultats. L'inconvénient est que les valeurs particulières ne sont pas forcément testées, une utilisation de tests spécifiques peut à ce moment se révéler utile en complément.

```
int main (int argc, char * argv []){
    //int nb_tests = atoi(argv[1]);
    int buffer_A = 0;
    int buffer_B = 0;
    int buffer_Res = 0;

    FILE *ref_A = fopen("ref_A.txt", "r");
    FILE *ref_B = fopen("ref_B.txt", "r");
    FILE *resu_C = fopen("resu_C.txt", "w");

    printf("Fichier ouvert\n");

    for(int i = 0; i < 65536; i++){
      fscanf(ref_A, "%d", &buffer_A);
      fscanf(ref_B, "%d", &buffer_B);
      buffer_Res = PGCD(buffer_A, buffer_B);
      fprintf(resu_C, "%d\r\n", buffer_Res);
    }

    fclose(ref_A);
    fclose(ref_B);
    fclose(resu_C);
```

Figure 14 : Algorithme de manipulation de fichiers

II. Hardware

A. Etape 1

L'image qui suit représente la Machine à états que j'ai décrit en VHDL.

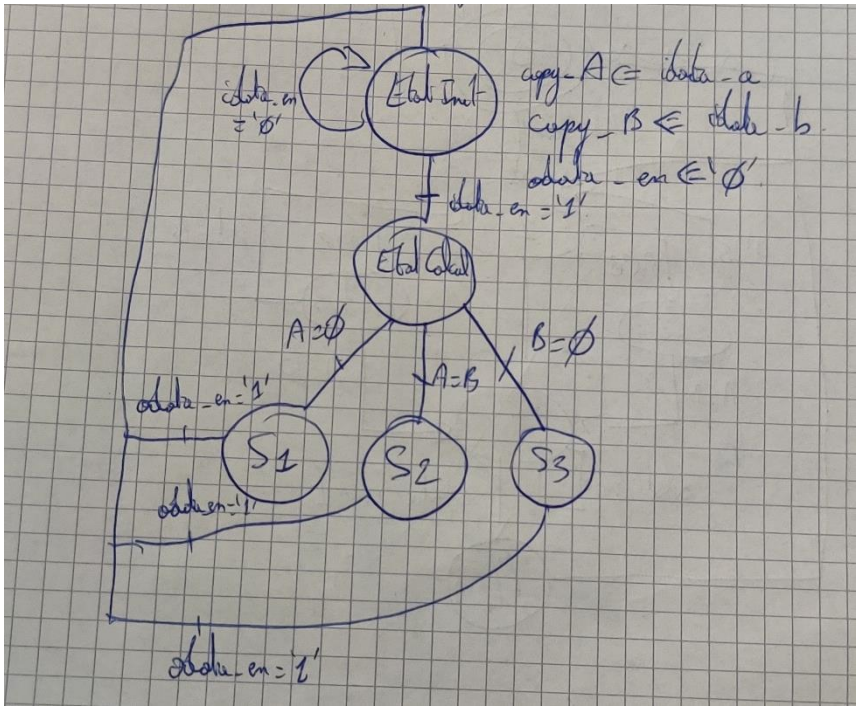


Figure 15 : Dessin de la machine à états

Exemple de valeurs testées pour vérifier le fonctionnement du PGCD.

```
RESET <= '1';
idata_a <= x"00000000";
idata_b <= x"00000000";
idata_en <= '0';
wait for 4*clock_period;

RESET <= '0';
wait for 4*clock_period;

idata_a <= x"00000018";
idata_b <= x"00000008";
idata_en <= '1';
wait for clock_period;
idata_a <= x"00000000";
idata_b <= x"00000000";
idata_en <= '0';
wait for 120*clock_period;
```

Figure 16 : Valeurs de test

On constate à l'aide de la simulation sur la figure qui suit que l'algorithme de PGCD fonctionne correctement, pour des valeurs de 24 et 8 passées en entrée, la valeur de sortie est 8. Le résultat met 3 cycles d'horloge à être déterminé, ce qui correspond au nombre d'occurrence de la réalisation de l'état calcul.

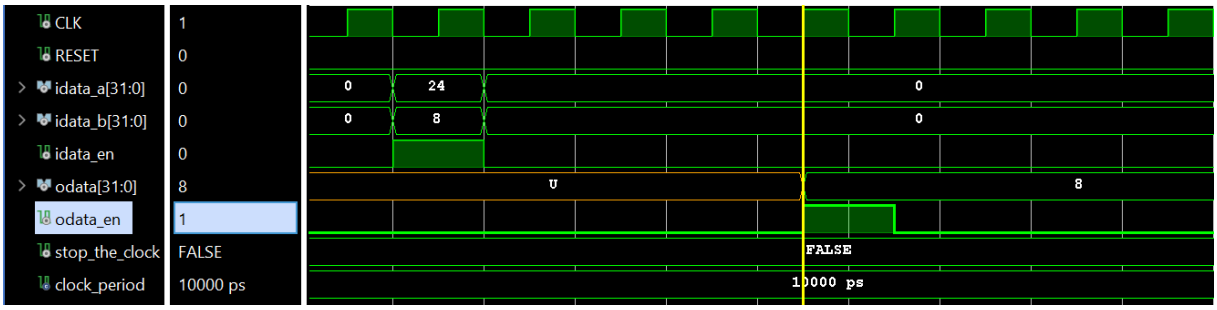


Figure 17 : Vérification par simulation

B. Etape 2

Pour vérifier le fonctionnement des assertions, j'ai intégré les même que celles définies à l'étape 5 software, avec en plus une assertion volontairement fausse pour vérifier le bon fonctionnement lorsque l'on tente de simuler. Voici la description des assertions :

```
assert copy_a >= 0 report "A négatif" severity error;
assert copy_b >= 0 report "B négatif" severity error;
assert copy_a <= 65535 report "A supérieur à 65535" severity error;
assert copy_b <= 65535 report "B supérieur à 65535" severity error;
```

```
assert copy_a >= 0 report "A doit être supérieur ou égal à 0" severity error;
assert copy_a = copy_b report "A n'est pas égal à B" severity error;
assert copy_i_a mod copy_a = 0 report "Reste différent de 0 entre A et Copy_A" severity error;
assert copy_i_b mod copy_b = 0 report "Reste différent de 0 entre B et Copy_B" severity error;
assert copy_a > 100 report "Verif assertions" severity failure;
```

Figure 18 : Assertions

```
# run 1000ns
Failure: Verif assertions
Time: 125 ns  Iteration: 0  Process: /PGCD_tb/uut/cal_output  File: C:/Users/dimit/Documents/Cours ENSEIRB/Verification_tests_mate
$finish called at time : 125 ns : File "C:/Users/dimit/Documents/Cours ENSEIRB/Verification_tests_materiels/EN224-Test-et-verifica
INFO: [USF-XSim-96] XSim completed. Design snapshot 'PGCD_tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
```

Figure 19 : Echec de la simulation grâce à l'assertion erroné

Ces tests unitaires permettent de vérifier le bon fonctionnement de l'algorithme mais est contraignant puisqu'il est obligatoire de vérifier manuellement que les résultats des tests correspondent aux attentes.

C. Etape 3

L'avantage de cette méthode est que l'on vient comparer à un programme en C qui est simple et a peu de risques d'être différent, il permet de s'assurer du fonctionnement de notre description VHDL rapidement et facilement. L'inconvénient est qu'il faut prendre le temps de développer le programme C et que sur un module complexe à tester, il se peut que le programme comporte des erreurs et on prend du temps.

D. Etape 4

Cette approche possède les même avantages et inconvénients que la précédente, elle assure en revanche la récupération sur des fichiers externes et évite de de voir copier à la main le résultat du programme C.