Anatole JOURNET SEE 2A

Rapport

EN224

Table des matières

Aucune entrée de table d'illustration n'a été trouvée.

Introduction

Dans ce rapport nous allons prendre en main différents moyen de test permettant la vérification de la conception de software ou de hardware pour un programme visant à calculer le PGCD de deux nombres.

Pour rappel, PGCD correspond à Plus Grand Dénominateur Commun.

Software

Etape 1

Dans un premier temps, on implémente en C l'algorithme suivant qui permet le calcul du PGCD de deux nombres.

```
FONCTION PGCD(A, B )
  TANT QUE A /= B ALORS
    SI A > B ALORS
      A = A - B
    SINON
      B = B - A
    FIN ALORS
  FIN TANT QUE
  RENVOYER A
FIN FONCTION
```

```c
int PGCD(int A, int B)
{
    if (A==0){
        return B;
    }
    else if (B==0){
        return A;
    }
```

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

Afin de vérifier le bon fonctionnement de cette fonction on exécute le programme pour les 10 valeurs de référence ci-dessous :

A	B	PGCD attendu	PGCD obtenu
10	5	5	5
95	36	1	1
88	72	8	8
219	28	1	1
333	416	1	1
79	501	1	1
63	1	1	1
700	77	7	7
24	34	2	2
65535	505	5	5

A l'issu de ce test on constate que notre fonction semble fonctionner correctement. Nous pouvons même voir qu'il fonctionne avec A égal à la borne maximum des nombres valides. En revanche, nous ne l'avions pas tester mais ce programme ne fonctionne pas si A ou B = 0.

Etape 2

Afin de tester un maximum de valeurs, on test notre programme avec n couples de valeurs aléatoires.

On génère 20 couples, puis 2000, puis 20 000

On constate que lorsque que l'on passe le nombre d'exécution à 20 000, le programme bloque à la ligne 11496. En commentant l'exécution de la fonction PGCD on constate qu'à l'itération 11497 le nombre A passe à 0.

```
      11496 -> A = 27854 et B = 14 le PGCD est de 2

      11497 -> A = 27854 et B = 14
```

Si le programme bloque à ce niveau, c'est parce que si A ou B vaut 0, le programme rentre dans une boucle infinie. En effet si l'un des deux nombres vaut 0 les lignes A = A-B ou B = B-A n'opèrent aucun changement et le programme tourne en boucle.

Pour régler ce problème, il suffit de modifier la fonction PGCD pour gérer le cas du A ou B = 0

```
int PGCD(int A, int B)
{
    if (A==0){
        return B;
    }
    else if (B==0){
        return A;
    }
    else {
        while (A != B){
            if (A>B){
                A = A-B;
            }
            else B = B - A;
        }
        return A;
    }
}
            }

            else{

                B = B-A;

            }
```

```
        }
    }
    return A;
}
```

Etape 3

Dans l'étape précédente il fallait vérifier le résultat manuellement. Cette vérification devient très rapidement fastidieuse et chronophage lorsque le nombre de combinaison devient important.

Afin de valider de manière automatique nos résultats, on réalise une deuxième fonction permettant de calculer le PGCD.

```
nt PGCD_methode2(int A, int B){
    int reste = 1;

    while (reste !=0){
        int resultat = A-B;
        if (resultat > 0){
            A=resultat;
        }
        else if (resultat<0){
            B=resultat*-1;
        }
        else if (resultat == 0){
            return A;
        }
    }
    return 0;
}
```

On estimera par la suite, que si les deux méthodes renvoient le même résultat pour le même couple de valeurs, alors le résultat sera juste.

Pour chaque couple de valeur on réalise le calcul avec notre fonction et la méthode n°2. Puis on compare les deux valeurs

```
int test_PGCD(int PGCD, int PGCD_methode2){
    if (PGCD_methode2 == PGCD){
        return 1;
    }
    else{
```

```
        return 0;
    }
}
```

Grace à cette méthode on peut grandement améliorer notre couverture de test. Cependant elle représente un défaut, si notre deuxième fonction n'est pas fonctionnelle, notre .

Etape 4

Afin de renforcer les tests de notre programme, on ajoute des assertions sur les valeurs d'entrée et de sortie de notre fonction.

```
assert(A<=65535);
assert(B<=65535);
assert(resultat<=65535);
assert(resultat>0);
```

Ces lignes permettront de vérifier que les valeurs de A, B et du résultat ne dépassent pas la borne MAX = 65535. Elles permettront également de vérifier que le résultat est positif.

En lançant ce programme avec des valeurs dépassant les bornes [0 ; 65535] on obtient l'erreur suivante sur le terminal :

```
/main 25 150000
(II) Starting PGCD program
main: src/main.c:28: main: Assertion `B<=65535' failed.
Aborted
```

Si on modifie le makeFile en ajoutant la ligne suivante, on désactive les asserts.

```
CFLAGS=-O2 -Wall -DNDEBUG
```

Les assertions peuvent s'avérer très utiles pour débugger son code. Cependant elles restent toujours très dépendantes du développeur et il est facile d'oublier une condition à tester.

Etape 5

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

    for (int i=0; i<65536; i++){
        int A = rand() %65534 + 1;
        int B = rand() %65535 + 1;
        assert(A<=65535);
        assert(B<=65535);
        int resultat = PGCD(A, B);
        int resultat2 = PGCD_methode2(A,B);
        assert(resultat<=65535);
        assert(resultat>0);
        assert(resultat2<=65535);
        assert(resultat2>0);
        int test = test_PGCD(A, B, resultat);
        assert(test==1);
    }

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

Les post-conditions permettent de vérifier les valeurs de sortie après l'appelle de la fonction que l'on souhaite tester. Ils permettront d'ajouter une vérification afin de s'assurer un peu plus de la viabilité de notre programme.

En revanche, nous sommes dans l'impossibilité de tout tester, il faut donc faire des compromis entre les tests et l'intérêt de ceux-ci.

Etape 6

Maintenant que nous savons faire des assertions nous allons nous en servir pour faire des tests unitaires.

Pour cela, on réalise des asserts sur des appels à notre fonction PGCD et nous vérifierons que les valeurs renvoyées sont bien les bonnes.

On réalise ces tests pour différentes valeurs choisit de manière aléatoire ou posant potentiellement problème.

```
//test avec un des deux arguments égale au PGCD
    assert (PGCD(3500,14)==14); printf("test passed \n");
    assert (PGCD(25,250)==25); printf("test passed \n");

    //test pour des valeurs banales
    assert (PGCD(10,5)==5); printf("test passed \n");
    assert (PGCD(3650,24)==2); printf("test passed \n");
    assert (PGCD(95,36)==1); printf("test passed \n");
    assert (PGCD(88,72)==8); printf("test passed \n");
    assert (PGCD(700,77)==7); printf("test passed \n");

    //test aux limites
    assert (PGCD(65535,1)==1); printf("test passed \n");
    assert (PGCD(1,65535)==1); printf("test passed \n");

    //test pour 0
    assert (PGCD(0,65)==65); printf("test passed \n");
    assert (PGCD(72,0)==72); printf("test passed \n");
    assert (PGCD(0,0)==0); printf("test passed \n");
```

Grace à ces tests unitaires on peut tester la fonction PGCD indépendamment de sa réalisation. Cela signifie que notre fichier de test fonctionnera même si on venait à apporter des modifications à la fonction PGCD.

```
(II) Starting PGCD program
test passed
test passed
test passed
test passed
test passed
test passed
test passed
test passed
test passed
test passed
test passed
(II) End of PGCD program
```

Ci-contre, le résultat de l'exécution du programme sur le terminal.

L'inconvénient majeur de ces tests unitaires est qu'il faut renseigner manuellement les résultats attendus de chaque test.

Un autre inconvénient est qu'un assert termine le programme. Ainsi si on à une grosse batterie de test à réaliser et que l'un des premiers est faux le reste des tests ne se fera pas. Cela peut être vraiment handicapant si l'on lance un test pendant une longue durée et que l'un des tests n'est pas validé sans que l'utilisateur s'en rende compte. Cela peut être source de perte de temps.

Etape 7

Une méthode de test unitaire plus performante consiste à utiliser des Framework.

Les Framework permettent de catégoriser les tests selon des « test case » et des « sections »

```
TEST_CASE( "cas normaux", "[pgcd]" ) {


    SECTION( "A>B") {
        REQUIRE( PGCD(135, 55) == 2 );
        REQUIRE( PGCD(133, 97) == 1 );
    }
```

Dans l'exemple ci-dessus, on a un test case de fonctionnement normal dans lequel on retrouvera une section testant uniquement des valeurs avec A>B

Nous avons volontairement placé une erreur, le PGCD de 135 et 55 est 5 et non 2. Ainsi, nous obtenons cela :

```
src/main.cpp:15: FAILED:
  REQUIRE( PGCD(135, 55) == 2 )
with expansion:
  5 == 2

===============================
test cases: 1 | 1 failed
assertions: 1 | 1 failed
```

On constate que le Framework nous renvoie le détail des tests cases passés et échoués avec pour les tests échoués, un détail de la valeur attendu et obtenu.

On remarque également que lorsqu'un test échoue, les autres tests de la section ne sont pas exécutés. En revanche, ceux des autres sections le sont.

Si l'on modifie cette erreur et que l'on réexécute on obtient l'affichage suivant indiquant que tout s'est bien passé.

```
===============================================
All tests passed (2 assertions in 1 test case)
```

Les Framework sont des tests unitaires plus puissant que les asserts car ils offrent une catégorisation des tests ainsi qu'un détail en cas d'erreur bien plus important ce qui facilite grandement le débogage. Un autre avantage est qu'un test case qui échoue ne termine pas le programme ce qui palie au problème de perte de temps identifié avec les asserts. Cependant on a toujours un renseignement manuel des valeurs de retour des tests.

Etape 8

La meilleure solution sera l'utilisation de modèles de références contenant des valeurs de A, de B et le résultat attendu pour 65535 valeurs aléatoires. Nous avons utilisé Excel pour générer un grand nombre de valeurs d'entrée avec leurs valeurs de retour associer et stocker le tout dans des fichiers.

Nous ajouterons donc une fonction de lecture de ces fichiers dans notre programme. Le but sera de comparer les résultats obtenus avec notre programme avec les résultats d'Excel :

```
FILE* fichier_A = fopen("src/ref_A.txt", "r");
FILE* fichier_B = fopen("src/ref_B.txt", "r");
FILE* fichier_C = fopen("src/res_C.txt", "w");
int A = 0;
int B = 0;

while ((feof(fichier_A) == 0) && (feof(fichier_B) == 0)){
    fscanf(fichier_A, "%d", &A);
    fscanf(fichier_B, "%d", &B);

    int resultat = PGCD(A, B);

    fprintf(fichier_C, "%d\n", resultat);
}

fclose(fichier_A);
```

```
fclose(fichier_B);
fclose(fichier_C);
system("diff -b -s src/ref_C.txt src/res_C.txt");
printf("(II) End of PGCD program\n");
```

Dans un premier temps nous lisons les valeurs de A et B que nous allons utiliser comme paramètre de notre appel à fonction.

Nous stockerons les résultats dans un fichier.

Pour finir, nous utilisons la fonction système « diff » qui permet de comparer deux fichiers.

Si les tests sont tous validé les deux fichiers devrait été identique

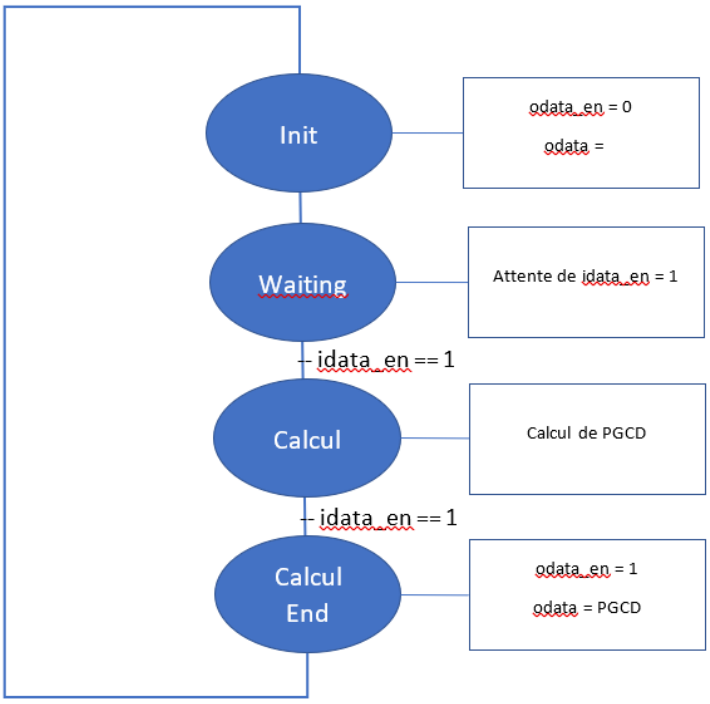
```
Files src/ref_C.txt and src/res_C.txt are identical
```

Ce système est très performant est permet de tester un nombre important de cas en très peu de temps. Cependant, cela nécessite un temps de développement du test plus long que les précédentes méthodes car il demande la création de fichiers de références.

Hardware

Etape 1

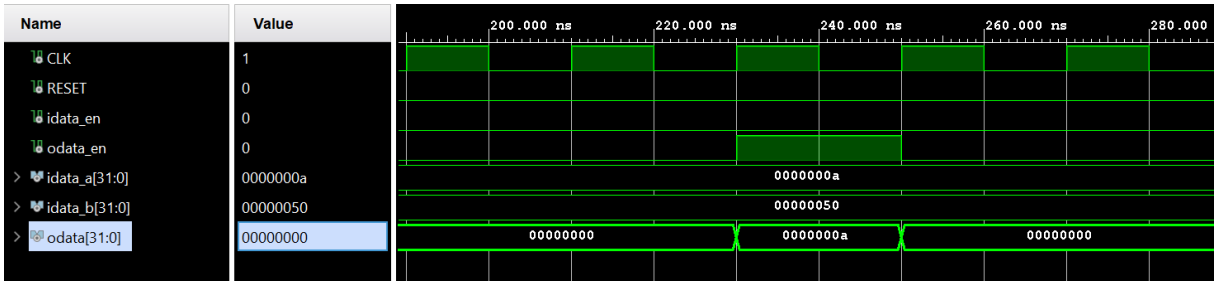
La machine d'état décrivant le calcul du PGCD de deux nombres est la suivante :



Pour vérifier le bon fonctionnement de notre programme, nous écrirons un test bench dont la partie test sera décrite de la façon suivante :

```
idata_a  <= STD_LOGIC_VECTOR( TO_UNSIGNED( 10, 32) );
idata_b  <= STD_LOGIC_VECTOR( TO_UNSIGNED( 80, 32) );
idata_en <= '1';
wait for 50 ns;
idata_en <= '0';
wait for 1000 ns;
```

Ainsi, nous obtiendrons la simulation suivante :



A l'aide de cette simulation nous pouvons constater que notre programme fonctionne pour ce couple de valeurs.

Etape 2

Pour vérifier le bon fonctionnement de notre programme, nous utiliserons des assertions comme dans le programme en C. Dans un premier temps nous vérifierons que les valeurs en entrée sont bien inférieures à notre borne max = 65535. Puis nous allons réaliser des tests sur le résultat du calcul, d'abord qu'il est valide, donc supérieur à 0. Puis qu'il est inférieur à la borne max.

```
                        ASSERT(UNSIGNED(idata_a) <= 65535)
                            REPORT "Erreur A depasse les bornes "
                        SEVERITY ERROR;

                        ASSERT(UNSIGNED(idata_b) <= 65535)
                            REPORT "Erreur B depasse les bornes "
                        SEVERITY ERROR;
...
                         ASSERT(UNSIGNED(interne_idata_a) > 0)
                            REPORT "Erreur PGD invalide "
                         SEVERITY ERROR;

                         ASSERT(UNSIGNED(interne_idata_a) <= 65535)
                            REPORT "Erreur PGD depasse les bornes "
                         SEVERITY ERROR;
```

Pour verifier le fonctionnement des assert, on intègre volontairement une erreur dans notre test bench :

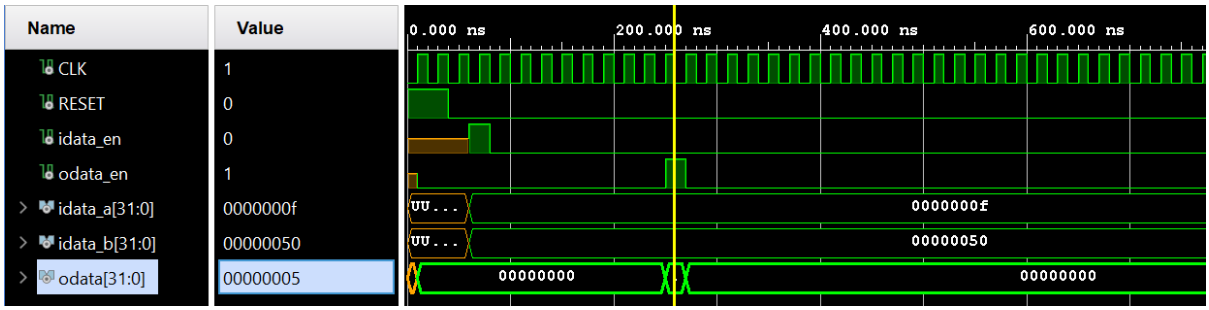
```
idata_a <= STD_LOGIC_VECTOR (to_unsigned (65536,32));
idata_b <= STD_LOGIC_VECTOR (to_unsigned (0,32));
```

C'est le 1er assert qui va se déclencher et renvoyer une erreur comme ceci :

```
Error: Erreur A depasse les bornes
Time: 50 ns  Iteration: 1  Process: /tb
```

Etape 3

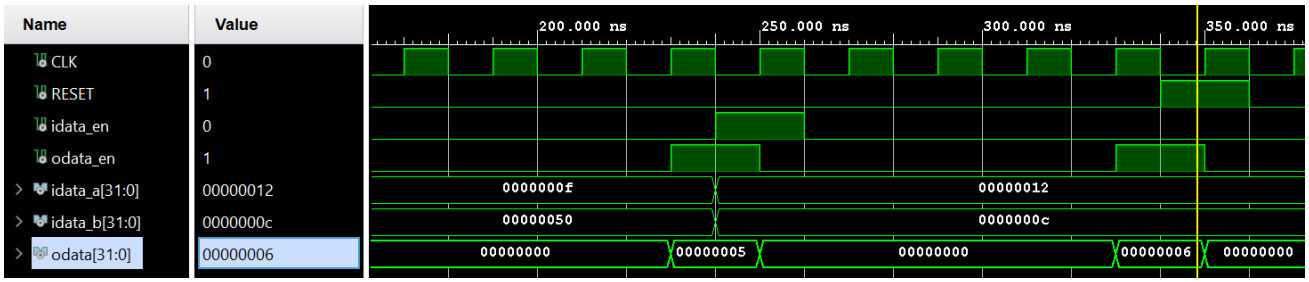
```
idata_a  <= STD_LOGIC_VECTOR( TO_UNSIGNED( 10, 32) );
idata_b  <= STD_LOGIC_VECTOR( TO_UNSIGNED( 80, 32) );
idata_en <= '1';
wait for 10 ns;
idata_en <= '0';
wait for 1000 ns;
```



Ceci est la solution que nous utilisions jusqu'à présent. L'inconvénient principal est que nous devions exécuter plusieurs fois le codes pour avoir plusieurs tests. Nous allons donc modifier notre programme pour que plusieurs tests s'enchaînent.

```
idata_a  <= STD_LOGIC_VECTOR( TO_UNSIGNED( 15, 32) );
idata_b  <= STD_LOGIC_VECTOR( TO_UNSIGNED( 80, 32) );
idata_en <= '1';
wait for 20 ns;
while odata_en = '0' loop
    idata_en <= '0';
    wait for 10 ns;
end loop;

idata_a  <= STD_LOGIC_VECTOR( TO_UNSIGNED( 18, 32) );
idata_b  <= STD_LOGIC_VECTOR( TO_UNSIGNED( 12, 32) );
idata_en <= '1';
wait for 20 ns;
while odata_en = '0' loop
    idata_en <= '0';
    wait for 10 ns;
end loop;
```



Ici, deux tests s'enchainent, nous testons dans un premier temps le calcul pour les valeurs 15 et 80, puis 18 et 12. En regardant les simulations, nous remarquons que le programme fonctionne et nous renvoie bien les bonnes valeurs.

Le problème est que nous devons lire les simulations pour vérifier le bon fonctionnement du programme. Pour régler cela, nous utiliserons des assertions, elles permettront de vérifier automatiquement le résultat du calcul.

Elles s'utilisent de la façon suivante :

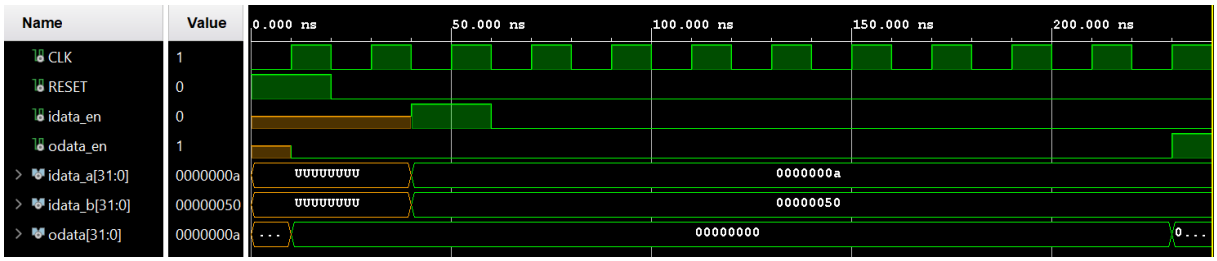
```
ASSERT UNSIGNED(odata) = TO_UNSIGNED( 11, 32)
    REPORT "PGCD incorrect"
    SEVERITY FAILURE;
```

Nous avons ajouté un "REPORT" qui permet l'affichage d'un message dans la console tcl.

Nous avons également mis un niveau FAILURE pour que la simulation se termine si la condition n'est pas validée.

Dans cet exemple, nous avons volontairement introduit une erreur, (le PGCD doit être égal à 10 et non à 11) pour vérifier le comportement des assertions.

Ci-dessous, une simulation du programme comprenant les assertions.

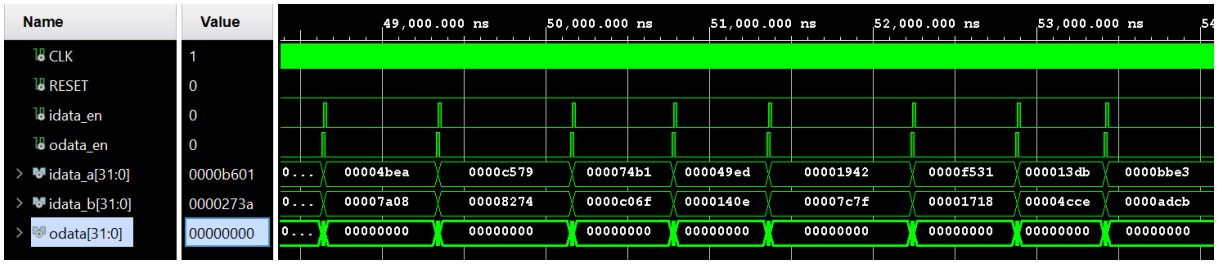


Nous voyons la simulation s'est interrompue. Le message de console correspondant à cette interruption est le suivant :

```
Failure: PGCD incorrect
Time: 240 ns  Iteration: 0
```

Maintenant que nous avons vérifier le fonctionnement des assertions, nous voulons augmenter le nombre de couples de valeurs à tester. Pour cela, nous créerons un programme en C nous permettra de générer rapidement des milliers de tests différents. Il utilisera les fichiers de références que nous avions réalisés pour la partie SOFTWARE.

Ainsi, nous obtiendrons une simulation qui ressemble à cela :



Nous avons zoomé sur une partie au hasard de la simulation.

Le nombre très élevé de tests nous empêche de lire aisément la simulation, c'est pour cela que les assertions sont très utiles. En effet, elles nous permettront de vérifier, en un simple coup d'œil dans la console tcl, le bon fonctionnement de notre programme.

Dans notre cas, aucune erreur n'a été détectée.