Bordeaux INP – ENSEIRB-MATMECA

Filière Systèmes Electroniques Embarqués

Test et vérification matériels

Lucas MARTIN

Remis le : 25/02/2021

Partie SOFTWARE

Etape 1 :

Fonction C du PGCD:

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

    return A;
}
```

Retour console:

```
(II) Starting PGCD program

PGCD(9,6), expected : 3, result : 3
PGCD(13,6), expected : 1, result : 1
PGCD(6,6), expected : 6, result : 6
PGCD(6,9), expected : 3, result : 3
PGCD(65535,38965), expected : ???, result : 1
```

On remarque que malgré la trivialité du programme, il est très facile d'oublier quelques détails empêchant la compilation (tel que des points virgules par exemple), mais également que l'on ne traite pas tous les cas possibles, notamment le cas du 0 dans ce programme.

Etape 2

Si on prend des valeurs entre 0 et 65535, le programme sera bloqué dans une boucle infinie lorsque l'un ou l'autre serait égal à 0. Ainsi, si l'une des valeurs est 0, on renvoie la valeur de l'autre (PGCD(a ;0)=a). On rajoute donc une condition if pour tester si AU MOINS une des valeurs est 0. Si c'est le cas on renvoie l'autre valeur :

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

    while (A != B) {
```

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

    return A;
}
```

Pour 20 valeurs :

```
Starting PGCD program

PGCD(45298,17205); result : 1
PGCD(30353,65496); result : 1
PGCD(41136,1128); result : 24
PGCD(58950,42640); result : 10
PGCD(1064,16353); result : 1
PGCD(44307,48302); result : 1
PGCD(8540,2614); result : 2
PGCD(61259,253); result : 11
PGCD(21411,48019); result : 1
PGCD(17451,52715); result : 1
PGCD(55342,23844); result : 2
PGCD(58832,1530); result : 2
PGCD(48489,47140); result : 1
PGCD(9771,21169); result : 1
PGCD(14293,31994); result : 1
PGCD(30451,46642); result : 1
PGCD(65060,47614); result : 2
PGCD(18628,1808); result : 4
PGCD(17780,2008); result : 4
PGCD(60074,57483); result : 1
```

Fonction aléatoire :

```
int random_ret(void) {
    int ret = rand() % 65536; // between 1 and randmax, modulo 65535

    srand(ret); // Set a new seed for better random
    return ret;
}
```

Etape 3 :

On estime que si l'on implémente un autre algorithme de calcul de PGCD, on peut comparer les résultats. Il y a statistiquement (très) peu de chance qu'une erreur sur chaque algorithme donne la même valeur. Il est cependant possible qu'une erreur survienne sur l'un des deux algorithmes, et que l'on ne sache pas de quel algorithme le problème vient.

On obtient le code suivant avec le nouvel algorithme :

```
int PGCD_modulo(int A, int B) {
    if (A == 0)
```

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

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

    return A;
}
```

Ainsi que le main :

```
int main (int argc, char * argv []){
    int a = 0, b = 0;

    for (int i = 0; i < NUMBER_TEST; i++) {
        a = random_ret();
        b = random_ret();
        if (PGCD_modulo(a,b) == PGCD(a,b))
            printf("Ok \n");
        else {
            printf("Erreur: a = %d, b = %d\n", a, b);
            printf("PGCD_modulo : %d, PGCD : %d\n", PGCD_modulo(a,b)
, PGCD(a,b));
            return 0;
        }

    }

  return 0;
}
```

Le retour console affiche bien « ok » lors de tous les tests.

Etape 4 :

On reprend les mêmes conditions que lors de l'étape une, et cette fois-ci, on teste à l'entrée de la fonction afin de savoir si les valeurs d'entrée sont bonnes :

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

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

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

Pour faire disparaître les assertions, il faut ajouter dans la partie CFLAGS du Makefile, le flag : -DNDEBUG. Les assertions permettent de savoir dès l'appel si les paramètres entrés sont bons, mais dans ce cas, aucune vérification n'est réalisée sur la validité de la sortie.

Etape 5 :

En plus des préconditions que l'on a rajouté précédemment, il faut rajouter des assertions à la fin permettant de tester si la valeur retournée est bien supérieure à 0. Cela permet d'éviter de renvoyer une valeur erronée en fin de fonction, et de tester cette valeur. Il s'agit donc d'un test supplémentaire permettant de vérifier le fonctionnement de l'algorithme. Cependant, il est difficile de tester le fonctionnement de l'assertion et de savoir dans quels cas cette assertion a lieu. De plus, selon les cas, très peu de test sur l'exact validité de la valeur finale peuvent être testée.

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

    int tempo_a = A;
    int tempo_b = B;

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

```
        return A;

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

    assert(A > 0);
    assert(A == B);
    assert((tempo_a % A) == 0);
    assert((tempo_b % A) == 0);

    return A;
}
```

Etape 6

Les tests unitaires permettent d'avoir une idée de la valeur renvoyée selon certaines valeurs clés, mais encore faut-il trouver ces fameuses valeurs, les tester, tout en ayant, dans le cas de assert, une mauvaise indication d'où exactement le programme a quitté.

Etape 7

Les limitations liées à ce framework sont les mêmes que pour les tests unitaires, il est possible de choisir quelles sont les valeurs et les tests logiques à effectuer, il faudra tout de même trouver quelles sont les valeurs à implanter. Ce framework permet cependant une plus grande précision sur l'endroit exact de l'erreur au sein du TEST_CASE, et permet de voir qu'elle condition a été testé.

Etape 8

Boucle de calcul :

```
for (i = 0; i < 65536; i++) {
        fscanf(f_a, "%d", &value_a);
        fscanf(f_b, "%d", &value_b);
        value_c = PGCD(value_a, value_b);
        fprintf(f_c, "%d\r\n", value_c);
    }
```

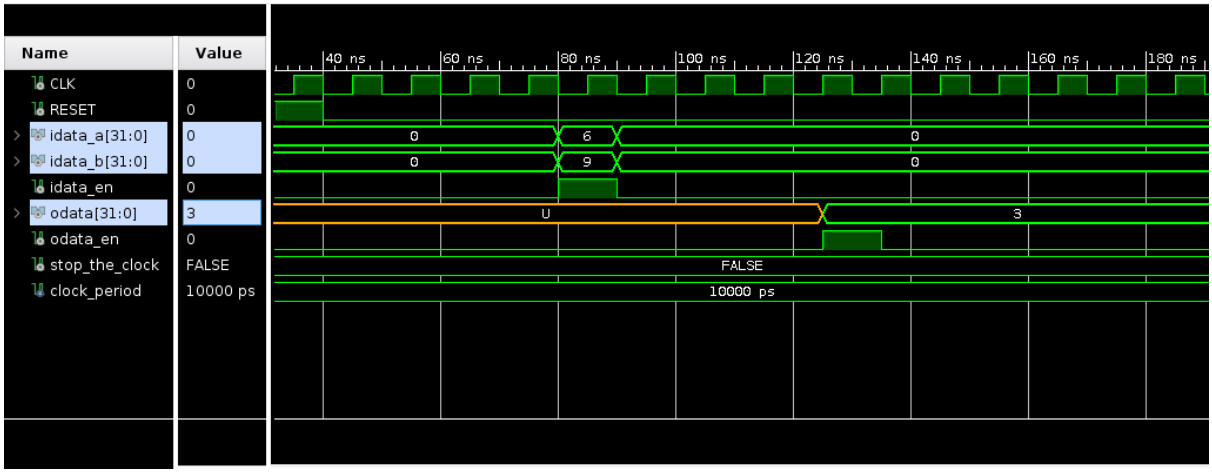
Cette méthode permet de tester notre algorithme sur un grand nombre de valeur que l'on sait déjà bonne, par le fait que ces résultats viennent d'une source sûre déjà validé. L'inconvénient c'est que les valeurs particulières ne sont pas forcément testées (peuvent dépendre de l'aléatoire). En fusionnant

cette méthode avec la méthode des frameworks, une solution peut permettre une première vérification globale du système dans son fonctionnement.

Partie HARDWARE

Etape 1

Choix des valeurs : 9 et 6



Ce test permet de vérifier les principales étapes de fonctionnement classique de notre PGCD. Des tests supplémentaires sont nécessaires pour vérifier les cas des valeurs particulières.

Etape 2

On a ajouté aux mêmes endroits, les mêmes conditions que dans la partie Soft (Etape 5). Nous avons cependant dû rajouter un signal de copie supplémentaire afin de pouvoir tester le modulo en assertion. Les assertions sont cependant plus précises en VHDL qu'en C / C++ lors de leurs utilisations car le paramètre « report » permet d'afficher dans le terminal un message d'erreur personnalisable.

Etape 3

A la fin de chaque affectation on rajoute la boucle suivante :

```
while odata_en = '0' loop
     idata_en <= '0';
     wait for 10 ns;
end loop;
```

Après modification du programme en C / C++ afin de n'avoir qu'à copier / coller le retour du terminal, on se rend compte que programmer ceci est très fastidieux et peut entraîner des erreurs si le programme C / C++ n'est pas bien programmer. Des tests de vérifications sont donc également nécessaire tout en ne permettant pas un grand nombre de tests possible.

Etape 4

Le traitement des fichiers sous VHDL se fera comme suit :

1. Ouverture des fichiers
2. Lecture de la ligne
3. Stockage de la ligne dans une variable
4. Transfert de la variable vers le signal

5. Envoi des signaux au composant
6. Fermeture des fichiers après avoir atteint la fin du fichier

L'avantage de cette étape est le fait que l'on n'aura pas à réaliser du travail manuel de copier / coller le retour terminal dans le testbench. Le testbench restera ainsi le même pour tous les tests tout en permettant de modifier les valeurs sur la partie programmée en C / C++.

Annexe :

Machine à état du PGCD