

# EN224 - Test et vérification

ALBERTY Maxime

9 février 2021

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Software</b>	<b>2</b>
2.1	Etape 1 . . . . .	2
2.2	Etape 2 . . . . .	2
2.3	Etape 3 . . . . .	3
2.4	Etape 4 . . . . .	4
2.5	Etape 5 . . . . .	4
2.6	Etape 6 . . . . .	5
2.7	Etape 7 . . . . .	6
<b>3</b>	<b>Hardware</b>	<b>7</b>
3.1	Etape 1 . . . . .	7
3.2	Etape 2 . . . . .	8
3.3	Etape 3 . . . . .	8
<b>4</b>	<b>Conclusion</b>	<b>8</b>

# 1 Introduction

Ce TP a pour but de découvrir différentes méthodes de test et vérification dans le développement de composants logiciels et matériels.

## 2 Software

### 2.1 Etape 1

L'implémentation de la fonction PGCD fût une simple traduction de l'algorithme présenté en langage C.

```
1 int PGCD(int A, int B)
2 {
3     while(A != B){
4         if (A > B){
5             A = A - B;
6         } else {
7             B = B - A;
8         }
9     }
10    return A;
11 }
12
```

Cependant il est important de resté concentré, car même si l'algorithme est trivial, une étourderie est vite arrivé et peut faire perdre du temps inutilement.

Le test de la fonction est effectué au moyen de couples de valeurs d'on aura préalablement calculé le PGCD.

```
1 printf("PGCD(1024,800) = %d\n",PGCD(1024,800));
2 printf("PGCD(800,1024) = %d\n",PGCD(800,1024));
3 printf("PGCD(32767,65535) = %d\n",PGCD(32767,65535));
4 printf("PGCD(65535,32767) = %d\n",PGCD(65535,32767));
5 printf("PGCD(512,2048) = %d\n",PGCD(512,2048));
6 printf("PGCD(2048,512) = %d\n",PGCD(2048,512));
7 printf("PGCD(458,6272) = %d\n",PGCD(458,6272));
8 printf("PGCD(6272,458) = %d\n",PGCD(6272,458));
9 printf("PGCD(783,125) = %d\n",PGCD(783,125));
10 printf("PGCD(125,783) = %d\n",PGCD(125,783));
```

Il faut ensuite comparé manuellement les résultats aux calculs.

### 2.2 Etape 2

Afin de pouvoir test plus de valeurs sans avoir à dupliquer les lignes de tests, nous ajoutons une génération aléatoires des valeurs de A et B comprises entre 0 et 65535.

```
1 #define MAXRAND 65535
2 #define MINRAND 0
3
4 int RandA(void){
5     int A = (rand() % (MAXRAND + 1 - MINRAND)) + MINRAND;
6     return A;
7 }
8
9 int RandB(void){
10    int B = (rand() % (MAXRAND + 1 - MINRAND)) + MINRAND;
11    return B;
12 }
```

En ajoutant une boucle FOR dans le main, on peut ainsi test beaucoup plus de valeurs.

```
1 for(int i = 0 ; i < 200000 ; i++){
2     A = RandA();
3     B = RandB();
4     printf("%d\t%d\t%d\t%d\n", i, A, B,PGCD(A, B));
5 }
```

Avec l'ajout de cette fonctionnalité, j'ai pu remarqué que la fonction PGCD ne prenait pas en compte les cas où  $A = 0$  ou  $B = 0$ . Conformément à l'annexe du sujet, la fonction PGCD devient alors :

```

1 int PGCD(int A, int B)
2 {
3     while(A != B){
4         if (A==0) return B;
5         if (B==0) return A;
6         if (A > B){
7             A = A - B;
8         } else {
9             B = B - A;
10        }
11    }
12    return A;
13 }

```

La vérification des résultats de la fonction PGCD est possible mais beaucoup trop long car il est nécessaire de comparer manuellement les résultats avec les calculs préalablement effectués.

## 2.3 Etape 3

Afin de tester plus de couple d'entrée, il peut être intéressant de comparer ma fonction PGCD avec une autre déjà éprouver. La nouvelle approche est la suivante :

- Assignez à  $N_1$  la valeur de  $N_2$  et à  $N_2$  la valeur du reste de la division de  $N_1$  par  $N_2$ ;
- Recommencez jusqu'à ce que le reste de la division soit nul;
- A ce moment,  $N_1$  contient alors le PGCD de  $N_1$  et  $N_2$ .

Cette approche correspond à cette fonction C :

```

1 int PGCD2(int A, int B){
2     int reste;
3
4     while(B != 0){
5         reste = A % B;
6         A = B;
7         B = reste;
8     }
9     return A;
10 }

```

En modifiant la boucle de l'étape 2, il est possible de tester plus de valeur en comparant le résultat des fonctions PGCD1 et PGCD2.

```

1 for(i = 0 ; i < 65536 ; i++){
2     A = RandA();
3     B = RandB();
4     pgcd_1 = PGCD1(A,B);
5     pgcd_2 = PGCD2(A,B);
6     test = (pgcd_1==pgcd_2)?true:false;
7     printf("%d\t%d\t%d\t%d\t%d\n", i, A, B,PGCD1(A, B), test);
8 }

```

Cette méthode nous permet de comparer les résultats de deux fonction effectuant la même opération mais avec des approches différentes. Seulement si les deux fonctions ont des résultats identiquement faux pour des couples de valeurs, le test sera positif au lieu d'indiquer une erreur de calcul.

## 2.4 Etape 4

La vérification des résultats peut commencer en assurant une cohérence entre les valeurs d'entrée et de sortie de la fonction PGCD. Cela est réalisé par des assertions.

```
1 int PGCD(int A, int B)
2 {
3     //Pre-condition
4     assert(A>=0);
5     assert(B>=0);
6     assert(A<=65535);
7     assert(B<=65535);
8
9     while(A != B){
10         if (A > B){
11             A = A - B;
12         } else {
13             B = B - A;
14         }
15     }
16     return A;
17 }
```

Ces pré-condition vont permettre d'assurer que les valeurs de  $A$  et  $B$  font partie de la plage des valeurs pour laquelle la fonction est conçue.

Ajouter limitation des pré-conditions

Il faut également noter que les assertions ne servent que pour le développement. Lors de la compilation de la version finale du programme, il faut désactiver les assertions.

```
1 gcc mon_prog.c -o mon_prog //Compilation avec les assertions
2 gcc mon_prog.c -o mon_prog -NDEBUG //Compilation sans les assertions
```

## 2.5 Etape 5

En plus de vérifier si les données fournis à la fonction sont conforme, on va vérifier que le résultat est cohérent.

- La valeur de sortie ne peut pas être plus grande qu'une des valeurs d'entrée,
- Si  $A \neq 0$  et  $B \neq 0$ , la valeur à la sortie de la boucle WHILE ne peut pas être également à 0.

```
1 int PGCD(int A, int B)
2 {
3     int firstA = A;
4     //pre-condition
5     assert(A>=0);
6     assert(B>=0);
7     assert(A<=65535);
8     assert(B<=65535);
9     while(A != B){
10         if(A == 0) return B;
11         if(B == 0) return A;
12         if (A > B){
13             A = A - B;
14         } else {
15             B = B - A;
16         }
17     }
18     //Post-condition
19     assert(A > 0);
20     assert(A <= firstA);
21     return A;
22 }
```

Ajouter limitation des post-conditions

## 2.6 Etape 6

Dans cette partie, nous séparons la fonction PGCD dans un fichier spécifique. Le nouveau fichier PGCD.c contient donc le code suivant :

```
1 #include "assert.h"
2 #include "pgcd.h"
3
4 int PGCD(int A, int B)
5 {
6     assert((A>=0) & (B>=0));
7     while(A != B){
8         if(A == 0) return B;
9         if(B == 0) return A;
10        if (A > B){
11            A = A - B;
12        } else {
13            B = B - A;
14        }
15    }
16    assert(A > 0);
17    return A;
18 }
```

Afin d'utiliser notre fonction dans le fichier main.c, il faut créer un fichier PGCD.h.

```
1 #ifndef _PGCD_H
2 #define _PGCD_H
3
4 int PGCD(int A, int B);
5
6 #endif
```

Le fichier main.c est alors comme suit:

```
1 #include "stdio.h"
2 #include "stdlib.h"
3 #include "math.h"
4 #include "assert.h"
5
6 #include "pgcd.h"
7
8 int main (int argc, char * argv []){
9     printf("(II) Starting PGCD program\n");
10
11     assert(PGCD(1024,800)==32);
12
13     assert(PGCD(32767,65535)==1);
14
15     assert(PGCD(512,2048)==512);
16
17     assert(PGCD(458,6272)==2);
18
19     printf("(II) End of PGCD program\n");
20     return 0;
21 }
```

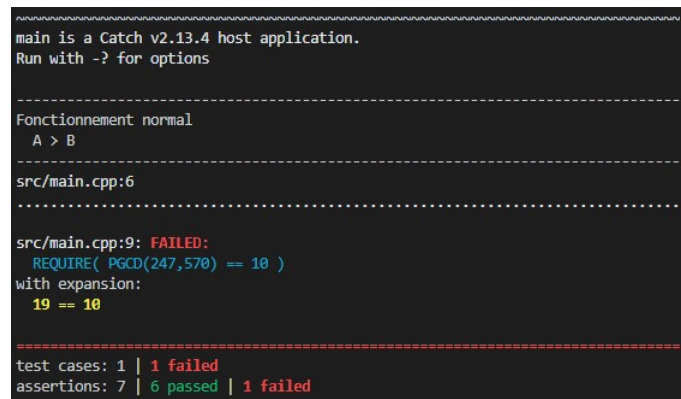
Les tests unitaires permettent de vérifier le fonctionnement d'une fonctionnalité pendant son développement. Ainsi on vérifie si les modifications que l'on apporte nous pas créer de bug. On parlera alors de test de non-régression.

## 2.7 Etape 7

Il existe de framework permettant de simplifier la rédaction des procédure de test et l'analyse des résultats. Dans cette exercice, le framework Catch2 va permettre d'exprimer des séquences de tests.

```
1 #include "pgcd.hpp"
2 #define CATCH_CONFIG_MAIN
3 #include "catch.hpp"
4
5 TEST_CASE ( "Fonctionnement normal", "[PGCD]" ){
6     SECTION("A > B"){
7         REQUIRE( PGCD(1250,570) == 10 );
8         REQUIRE( PGCD(5615,1248) == 1 );
9         REQUIRE( PGCD(247,570) == 19 );
10        REQUIRE( PGCD(14796,570) == 6 );
11    }
12    SECTION("A < B"){
13        REQUIRE( PGCD(6580,9896) == 4 );
14        REQUIRE( PGCD(1250,3245) == 5 );
15        REQUIRE( PGCD(1250,2000) == 250 );
16        REQUIRE( PGCD(1250,1251) == 1 );
17    }
18    SECTION("A = B"){
19        REQUIRE( PGCD(25,25) == 25 );
20        REQUIRE( PGCD(1250,1250) == 1250 );
21        REQUIRE( PGCD(100,100) == 100 );
22        REQUIRE( PGCD(8000,8000) == 8000 );
23    }
24 }
25
26 TEST_CASE ( "Fonctionnement autre", "[PGCD]" ){
27     SECTION("A > B"){
28         REQUIRE( PGCD(65535,570) == 15 );
29         REQUIRE( PGCD(1248,0) == 1248 );
30         REQUIRE( PGCD(570,0) == 570 );
31         REQUIRE( PGCD(42,0) == 42 );
32     }
33     SECTION("A < B"){
34         REQUIRE( PGCD(42,65535) == 3 );
35         REQUIRE( PGCD(1,65535) == 1 );
36         REQUIRE( PGCD(0,480) == 480 );
37         REQUIRE( PGCD(0,42) == 42 );
38     }
39     SECTION("A = B"){
40         REQUIRE( PGCD(65535,65535) == 65535 );
41         REQUIRE( PGCD(0,0) == 0 );
42     }
43 }
```

Dans le cas où des erreurs sont détectées, l'affichage est le suivant :



```
main is a Catch v2.13.4 host application.
Run with -? for options

-----
Fonctionnement normal
  A > B
-----
src/main.cpp:6
.....

src/main.cpp:9: FAILED:
  REQUIRE( PGCD(247,570) == 10 )
with expansion:
  19 == 10

-----
test cases: 1 | 1 failed
assertions: 7 | 6 passed | 1 failed
```

Figure 1: Exemple d'affichage en cas de test échoué

Dans le cas contraire, on retrouvera l'affichage suivant:

```
All tests passed (8 assertions in 1 test case)
```

Figure 2: Exemple d’affichage en cas de test réussi

Cependant, si une erreur se produit dans une section, cette dernière est abandonnée pour passer à la suivante. Si d’autres erreurs sont présentes sur d’autres cas de la section, elles ne seront pas visibles. Nous n’obtenons pas un rapport total des tests.

## 3 Hardware

### 3.1 Etape 1

De même que pour l’étape 1 de la partie software, il faut décrire un module VHDL permettant d’implanter le calcul du PGCD de deux nombres. Le module aura le prototype suivant :

```

1 ENTITY PGCD IS
2 PORT (
3     CLK      : in  STD_LOGIC;
4     RESET    : in  STD_LOGIC;
5
6     idata_a   : in  STD_LOGIC_VECTOR (31 downto 0);
7     idata_b   : in  STD_LOGIC_VECTOR (31 downto 0);
8     idata_en  : in  STD_LOGIC;
9
10    odata     : out STD_LOGIC_VECTOR (31 downto 0);
11    odata_en  : out STD_LOGIC
12 );
13 END PGCD;
```

Le module VHDL fonctionne grâce à la machine à état ci-dessous :

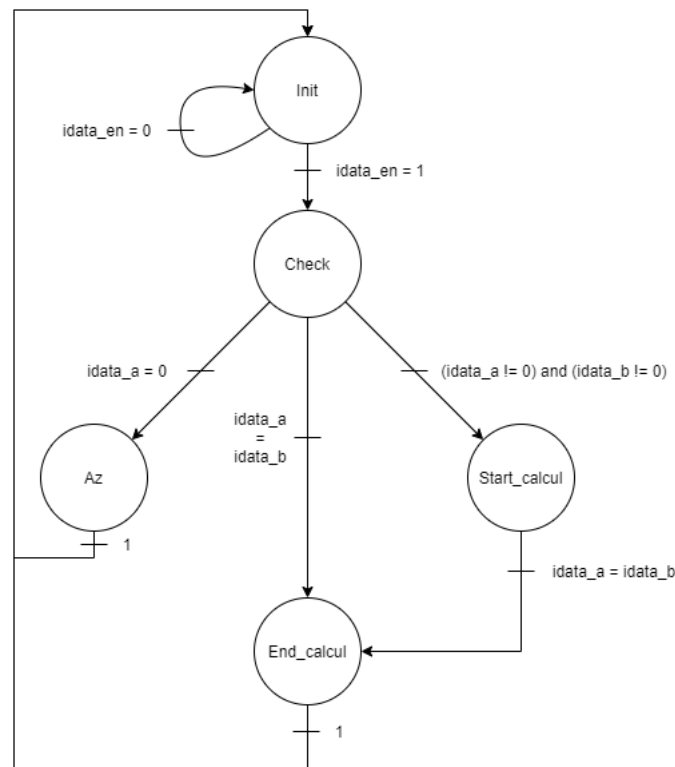


Figure 3: Machine état module PGCD

La description de cette machine nécessite 4 process:

- Mise à jour synchrone de l’état

- Calcul de nouvelle état en fonction des entrées
- Calcul des sorties en fonction de l'état en cours
- Calcul du PGCD

```

1  maj_etat : process ( CLK , RESET )
2  begin
3      if (RESET ='1') then
4          pr_state <= Etat_init ;
5          elsif ( CLK ' event and CLK ='1') then
6              pr_state <= nx_state ;
7          end if;
8  end process maj_etat;

```

## 3.2 Etape 2

## 3.3 Etape 3

## 4 Conclusion



## Todo list

<input type="checkbox"/> Ajouter limitation des pré-conditions . . . . .	4
<input type="checkbox"/> Ajouter limitation des post-conditions . . . . .	4