Projet IN302 : 2020-2021

December 11, 2020

Le projet est à faire individuellement. Il sera à soumettre sur moodle avant le 09/01/2021, 23h59 (heure de moodle faisant foi). Le but du projet est d'écrire un programme C qui simule un additionneur entier 16 bits. Pour ce faire, vous écrirez un fichier nommé **additionneur.c** qui devra être compilable avec le Makefile fourni en annexe.

1 Principe du programme

Le programme se lancera de la façon suivante :

```
$ additionneur <val1> <val2>
```

<val1> et <val2> représentent des nombres entiers 16 bits codés en hexadécimale. Le programme devra afficher la valeur hexadécimale de la somme des deux paramètres, ainsi que la représentation binaire. Il retournera 0 en cas de succès, ou une valeur différente de 0 en cas de problème.

Exemple d'exécution :

```
$ additionneur 0x0001 0x0002
Le resultat vaut 0x0003 (0000 0000 0000 0011) overflow : 0
$ echo $? # Affiche la valeur de sortie
          # du dernier programme appele
0
```

2 Fonctionnement du programme

2.1 Lecture des paramètres

Pour commencer, le programme doit être capable de lire une valeur hexadécimal et la convertir en binaire. Pour rappel, les paramètres d'un programme sont accessibles via les variables argc (nombre de paramètres) et argv (tableau des paramètres) lorsque la fonction main est déclarée de la façon suivante :

```
int main (int argc, const char** argv)
```

La représentation binaire du nombre sera un tableau de **char** de taille 16, chaque **char** pouvant prendre comme valeur 1 ou 0. Vous devrez écrire une fonction qui prend en paramètre une chaîne de caractère représentant une valeur hexadécimale et qui remplit un tableau de **char** de taille 16 contenant chacun des bits de la valeur d'entrée. On considerera que le bit à l'indice 0 correspond au bit le plus à gauche (bit de poids fort) du champs de bits. La fonction doit renvoyer 0 si la conversion s'est bien passée, sinon 1. Exemple :

```
int convertir_entree (char* tab, char* strhexa);
//convertir_entree (
//      {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
//      "0x1234");
//      tab = {0,0,0,1,   0,0,1,0,   0,0,1,1,   0,1,0,0}
//      valeur de retour = 0
```

Pour ce faire, vous pouvez utiliser la fonction strtol ou bien parser chaque caractère de la chaîne d'entrée pour en extraire les différents bits. Pensez à vérifier que la chaîne représente bien un nombre hexadécimal et que sa représentation peut tenir sur 16 bits. Dans le cas contraire, considérez qu'il y a une erreur.

2.2 Additionneur

Le coeur du programme est constitué d'un simulateur d'additionneur 16 bits d'entiers positifs. Pour représenter ceci, vous devrez écrire des fonctions qui représentent les différentes portes logiques nécéssaires (AND, OR, NOT, XOR ...), puis appeler ces fonctions pour faire les différents élèments de votre additionneur. Vous pourrez vous aider des schémas vu en TDs / cours / internet pour reconstituer les composants.

2.2.1 Les portes logiques universelles

Pour commencer, vous devrez écrire les portes logiques universelles qui serviront de briques de base pour la suite des composants. Au sein de l'additionneur, ces fonctions seront les seules à pouvoir faire des comparaisons (if, expressions ternaires ...) / opérations sur les paramètres. Les portes / circuits de l'additionneur ne pourront faire que des appels de fonctions aux différents composants et des affectations de variables. Les portes logiques universelles sont les portes NAND et NOR, que vous devez implémenter dans par des fonctions ayant les prototypes respectifs suivants :

```
char pl_NAND (char A, char B);
char pl_NOR (char A, char B);
```

2.2.2 Les portes logiques avancées

Afin de simplifier l'écriture de l'additionneur, d'autres portes, créées en utilisant uniquements des portes NAND et NOR, doivent être implémentées. Écrivez les

fonctions qui simulent le comportement des portes XOR, AND, OR et NOT en utilisant uniquement des portes NAND ou NOR (au choix). Les fonctions auront les prototypes suivants :

```
char pl_XOR (char A, char B);
char pl_AND (char A, char B);
char pl_OR (char A, char B);
char pl_NOT (char A);
```

2.2.3 Additionneur complet 1 bit

En utilisant les différentes portes implémentées, écrivez la fonction qui simule un additionneur complet 1 bit. La fonction aura le prototype suivant :

```
char add_1b (char A, char B, char Cin, char* Cout)
```

La fonction renverra le résultat de l'addition des deux bits et utilisera le pointeur Cout pour renvoyer la retenue de sortie. Les variables A et B représentent les deux bits à sommer, et Cin représente la retenue d'entrée. Elle devra être uniquement formée d'appels aux fonctions des différentes portes déjà implémentées, et d'affections de variables.

2.2.4 Additionneur complet 16 bits

A l'aide des différents éléments déjà implémentés, écrivez la fonction

```
char add_16b (char* A, char* B, char* sum)
```

qui simule un additionneur complet 16 bits. La fonction remplit le tableau sum de taille 16 qui représente la somme des deux paramètres A et B. Elle renvoie 1 si la somme génère un overflow (c'est à dire s'il faut un 17e bit pour représenter le résultat), sinon 0. Pour simplifier le code, cette fonction pourra utiliser une boucle, mais elle devra faire appelle aux éléments déjà implémentés.

2.3 Affichage du résultat

Pour finir, le programme doit pouvoir afficher les résultats de l'addition. Vous devrez écrire une fonction qui convertit le tableau de bits résultant de l'addition en un entier afin de pouvoir l'afficher grâce au format %x de la fonction printf.

```
printf ("%x\n", 164); //affiche a4
```

Exemple :

```
uint16_t convertir_sortie (char* bits);
// convertir_sortie ({0,0,0,1, 0,0,1,0, 0,0,1,1, 0,1,0,0})
//    => 0x1234
```

3 Validation du programme

Afin de vérifier que le programme fonctionne correctement, voici un ensemble de commandes que vous pouvez tester pour vérifier différents cas d'utilisations.

```
./additionneur 0x1 0x1
```

Test de base, doit fonctionner en affichant 0x2 (0000 0000 0000 0010) overflow 0, et doit renvoyer la valeur 0.

```
./additionneur 0xffff 0x1
```

Test d'overflow, doit fonctionner en affichant 0x0 (0000 0000 0000 0000) overflow 1, et doit renvoyer la valeur 0.

```
./additionneur 1
```

Test du nombre de paramètres, doit afficher un message d'erreur pour manque de paramètre et doit renvoyer une valeur différente de 0.

```
./additionneur azerty 1
```

Test du format des paramètres, doit afficher un message d'erreur car un paramètre ne représente pas une valeur hexadécimale et renvoyer une valeur différente de 0.

```
./additionneur ffffffff 1
```

Test de la taille des paramètres, doit afficher un message d'erreur car un paramètre n'est pas représentable sur 16 bits et renvoyer une valeur différente de 0.

4 Remarques

La note dépendra du respect des consignes, de la quantité de fonctionnalités implémentées et de la lisibilité du code (indentation et propreté). Le code devra être commenté pour expliquer le fonctionnement des différentes fonctions. Ces commentaires feront office de mini rapport de projet et participent à la note finale.

Le travail devra être rendu sous la forme d'une archive contenant :

- le fichier additionneur.c,
- le Makefile founi en annexe qui permet de compiler le programme,
- d'un fichier AUTHOR qui contiendra le nom, prénom et numéro d'étudiant.

Pour rappel, ce projet est à faire seul. Toute triche pourra être sanctionnée d'une note de 0.