# Définitions

C’est un algorithme qui nous permet de vérifier l’intégrité d’une information. Afin de s’assurer qu’elle n’ait pas été modifiée durant son stockage ou sa transmission. Le but étant de refaire l’opération de checksum sur la donnée reçu et de le comparé à celui attendu.

Notre checksum est un contrôle de type **modular sum**, il a donc pour but de faire l’addition de toutes les données (dans notre cas ce sont 3 bytes), puis on doit faire le complément à 2 du résultat (inversion des bits et on addition un 1).

# Circuits

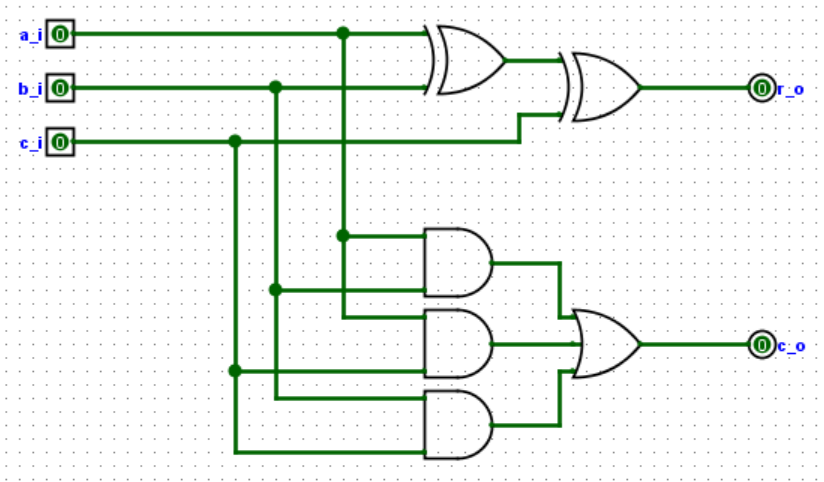
Partie décrivant les spécificités des circuits réalisés en classe.

## Additionneur 1 bit

C’est le circuit de base. Il va nous permettre de faire l’addition de 2 entrées et d’une retenue d’une longueur de 1 bit. On réalise donc un **OU-exclusif** entre les différentes entrées pour avoir le bit de sortie. Parfois dans l’addition il peut y avoir une retenu, on procède au traitement de celle-ci en appliquant un **ET** à chaque combinaison de 2 entrées et en transmettant ces trois résultats à un **OU**. Donc, nous avons les cas suivants :

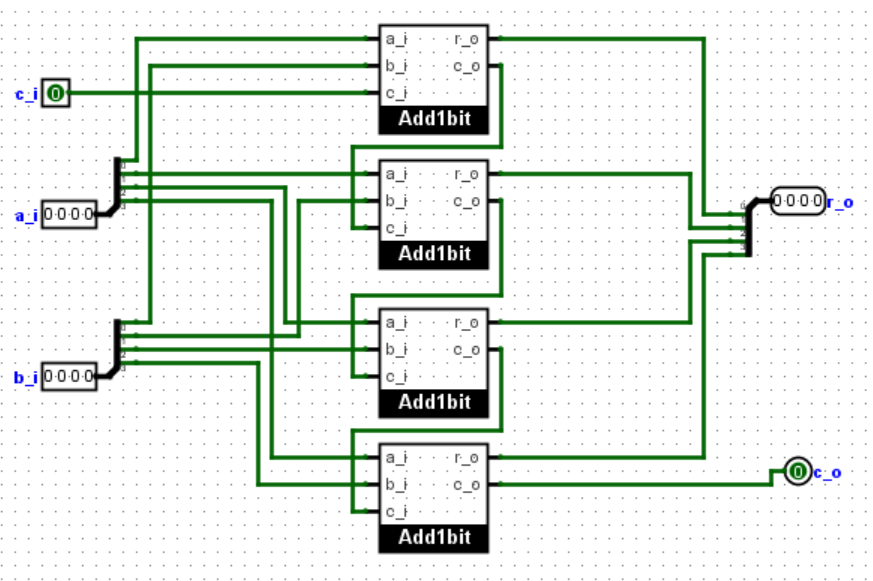
* 1 entrée à 1 🡪 1 en sortie et 0 en retenue
* 2 entrées à 1 🡪 0 en sortie et 1 en retenue
* 3 entrées à 1 🡪 1 en sortie et 1 en retenue

Il se passe le cas suivant, car on ne souhaite qu’avoir un bit en sortie. Donc si on fait 1 + 1, le résultat et 2 et il y a un dépassement pour afficher cela sur 1 bit.



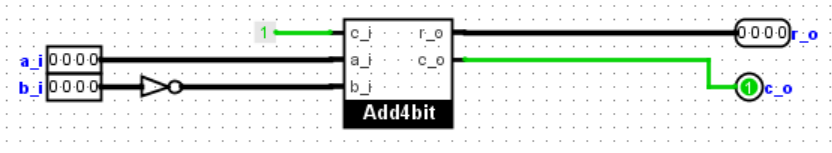
## Additionneur 4 bits

Ici, le système est assez simple. On possède 3 entrées, 2 entrées à 4 bits et une entrée qui serait la retenue d’une précédente opération. On va donc utiliser le système réaliser précédemment. On va séparer les entrées 4 bits en 4 entrées 1 bit, puis on utilisera 4 additionneurs 1 bit en cascade. En n’oubliant pas de relier les retenues obtenues dans les additionneurs en cascade. Ce système permet de représenter exactement ce que l’on ferait si l’addition devait être faite à la main (c-à-d que en colonne on ferait 1 + 1 et s’il y a une retenu on la met sur la colonne suivante, c’est pareil ici).



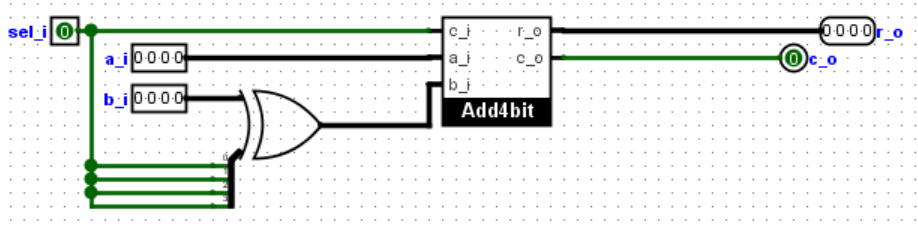
## Soustracteur 4 bits

Le but de ce circuit est de pouvoir soustraire deux entrées à 4 bits. Il faut pour pouvoir le réaliser comprendre comment fonctionne la soustraction binaire. Celle-ci est faite en faisant l’addition de la seconde entrée en complément à 2. On va donc utiliser un additionneur 4 bits et pour faire le complément à 2 de la seconde entrée, il va falloir inverser tous ses bits avec une porte **NON** et rajouter **1** à l’addition. Pour rajouter ce 1, on va donc utiliser le bit « de retenue » qui était présent sur le circuit.



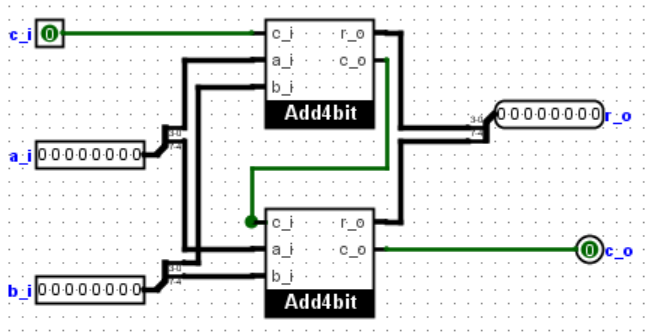
## Additionneur/soustracteur 4 bits

Le but est d’avoir un seul circuit permettant de faire les deux opérations vues précédemment. On a donc compris que l’addition et la soustraction sont assez similaires en termes de réalisation. Notre bit de retenu devient un bit de sélection d’opération. On retrouve donc notre première entrée directement dans l’additionneur. Notre seconde entrée passe par une porte **OU-Exclusif** avec notre bit de sélection, par ce système on peut donc soit laisser les entrées comme elles sont en cas d’addition (car on fait un OU-Exclusif avec une entrée toujours à 0). Et dans le cas d’une soustraction, cela nous permet d’inverser tous les bits de la seconde entrée pour ensuite faire son complément à 2. C’est aussi pour cela que le bit de sélection est relié à l’additionneur, car comme pour le soustracteur ça nous permet de faire une addition d’un 1 supplémentaire afin de réaliser le complément.



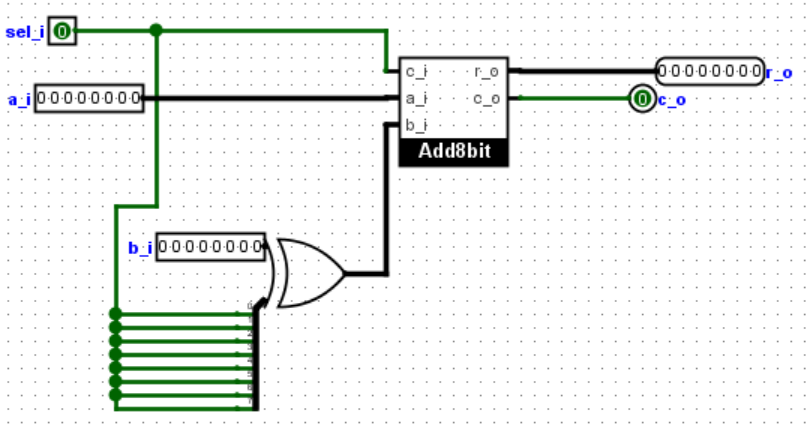
## Additionneur 8 bits

Comme lorsque l’on avait réalisé l’additionneur 4 bits, on va réutiliser un système précédent. On utilisera donc 2 additionneurs 4 bits et on refait un système en cascade. On va séparer les entrées en deux morceaux 4 bits, puis les réassembler à la fin pour avoir le résultat. Comme précédemment, on relie le bit de retenue pour que l’addition soit correcte s’il y a dépassement. Il y a aussi une troisième entrée de 1 bit qui pourra plus tard être utilisée comme bit de retenue ou de sélection dans des systèmes plus grands.



## Additionneur/soustracteur 8 bits

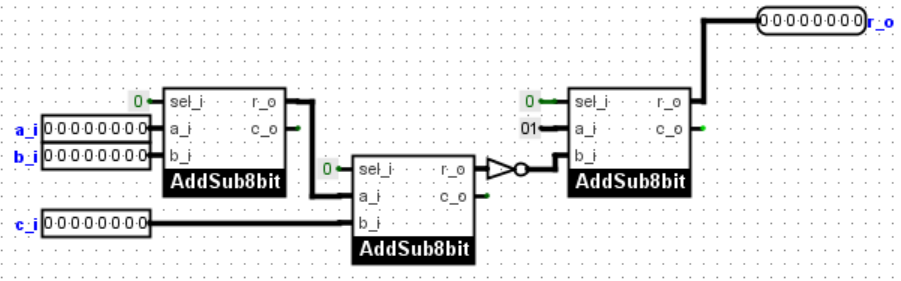
Le système est identique à la version 4 bits sauf que l’on va utiliser l’additionneur 8 bits. Il y a donc le même système de bit de sélection et de porte **OU-Exclusif**.



## Checksum

D’après la donnée, on doit réaliser un checksum de type modular sum. Il consiste à faire l’addition des 3 entrées disponibles et faire le complément à 2 du résultat. On va donc pour faire la première partie de cette opération utiliser un premier additionneur/soustracteur pour sommer les entrées A et B. Puis un second prenant le résultat obtenu précédemment et l’addition avec C. Les bits de retenus sont à laisser, il ne faut pas les impliquer dans le calcul.

Une fois cela fait, la seconde partie consiste à faire le complément à 2. On inverse avec une porte **NON** notre résultat et on utilise à nouveau un additionneur/soustracteur pour faire l’addition avec une entrée à 1. A partir de là, on a la somme de contrôle de nos 3 entrées.



## VerifyIntegrity

Ce dernier bloc a pour but de vérifier le bon fonctionnement de notre opération. Il faut avant de la réaliser bien comprendre les informations que nous avons en notre possession. On possède 4 entrées de 8 bits dans ces entrées il se trouve nos 3 entrées précédentes et notre checksum. Mais on ne sait pas quelle entrée correspond à quelle information.

Si on résume simplement l’opération faite par notre checksum, il additionne 3 nombres et fait leur complément à 2. On se retrouve donc avec l’inverse de l’addition. Prenons l’exemple suivant :

**A = 1 B = 2 C = 3**

Notre checksum additionne ces nombres, on obtient donc **6**. Puis le complément à 2 inverse ce nombre, on obtient donc **-6**. Pour vérifier que notre checksum à fonctionner, on doit donc s’assurer que le résultat est équivalent à 0. Il nous suffit alors de faire l’addition des 4 entrées, si tous les bits sont à 0, l’opération est donc correcte. Pour vérifier que tous les bits sont à 0, on va séparer chacun des bits et faire un **OU** avec ceux-ci. Car la porte OU, nous retourne 1 si une des entrées est à 1, donc si tout est à 0 nous aurons un 0.

La consigne nous indique aussi qu’il faut avoir un 1 comme résultat lorsque le checksum fonctionne. On va donc remplacer la porte **OU** par une **NON-OU**, comme ceci nous aurons 1 quand toutes les entrées sont à 0. Ce qui nous donne le système suivant.

