HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Reconfigurable & embedded
Digital Systems

Unité de calcul ALU
« Arithmetic Logic Unit »

Départements : TIC

Unité d'enseignement CSN

Auteurs : **Emily Baquerizo**
Rafael Dousse

Professeur : **Etienne Messerli**
Assistant : **Anthony Jaccard**

Classe : **CSN**
Salle de labo : **A09**

Date : **27.03.2024**

Introduction

Ce laboratoire a pour but de concevoir une unité de calcul appelée ALU. Une ALU est une unité de calcul qui réalise des opérations arithmétiques et logiques et qui est présente dans les processeurs. Elle va donc s'occuper de faire différentes opérations dont le résultat va varier en fonction d'un opcode, qui est un code indiquant quelle opération réaliser. Ce laboratoire va donc nous permettre de réaliser une ALU avec 8 opérations différentes et cela doit être réalisé grâce à la description en VHDL. Ensuite, il faut pouvoir faire la modélisation des portes avec le synthétiseur et finalement tester notre implémentation grâce à un fichier de simulation puis sur la MAX V.

Les opérations que nous devons pouvoir réaliser sont des additions/soustractions entre deux nombres, l'incrémentation de 1, une multiplication par 2, les ET/OU logiques entre deux valeurs et passer une entrée sur la sortie, donc avoir le même résultat que le nombre entré. Les résultats d'addition et de soustraction doivent pouvoir se faire sur des nombres signés et non signés, et on doit aussi donner une indication dans les cas où il y a des dépassements signés et non signés, ainsi qu'indiquer si le résultat de l'opération vaut 0. Tout cela doit se faire en essayant d'optimiser au maximum la logique utilisée, et ce rapport doit nous permettre d'expliquer comment nous avons conceptualisé l'ALU et comment nous avons trouvé les fonctions qui nous permettent de modéliser notre ALU.

Conception

ADD
Bloc Add N bits repris de l'ancien laboratoire
Multiplication

- Décalage à gauche en prenant les bits na_i[N-2 … 0] et en rajoutant un bit '0' en bit de point faible : nouveau
- Passer par l'additionneur pour effectuer le calcul na_i + na_i

Soustractions na_i – nb_i et nb_i – na_i
Passer par l'additionneur en effectuant un complément à 2 pour les 2 possibles soustractions

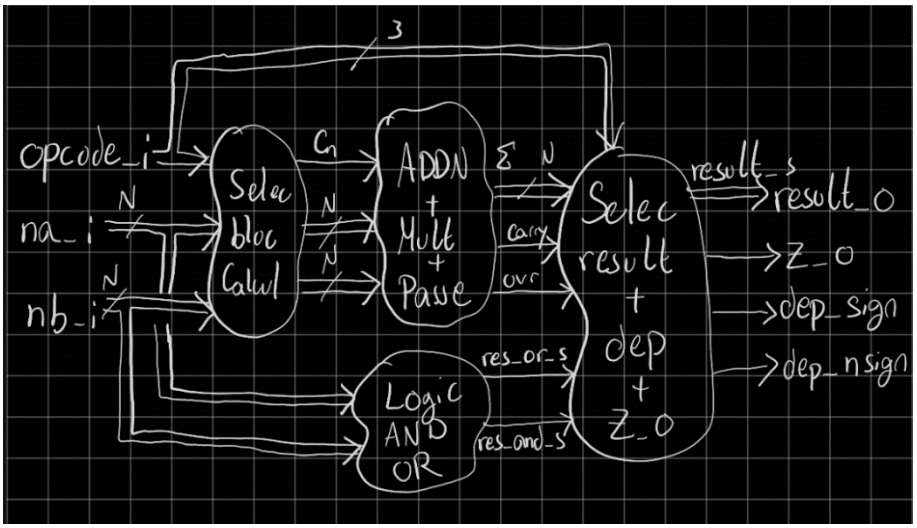
- na_i – nb_i = na_i + c2(nb_i) = na_i + not(nb_i) + 1
- nb_i – na_i = nb_i + c2(na_i) = nb_i + not(na_i) + 1

Dans les deux cas de soustraction, il faut interpréter correctement le carry et l'overflow sortant du bloc ADDN.
Incrémentation
Passer par l'additionneur avec 2 options : soit envoyer comme entrée à Q un vecteur de type « 0…1 » soit envoyer comme entrée à Q une entrée à 0 tout en envoyant comme entrée de cin '1'.
Passe
On renvoie tout simplement na_i à la sortie result_o. Aucun dépassement à gérer.
AND et OR
Dans ces deux cas, on peut utiliser directement des portes AND et OR :

na_i AND nb_i et na_i OR nb_i

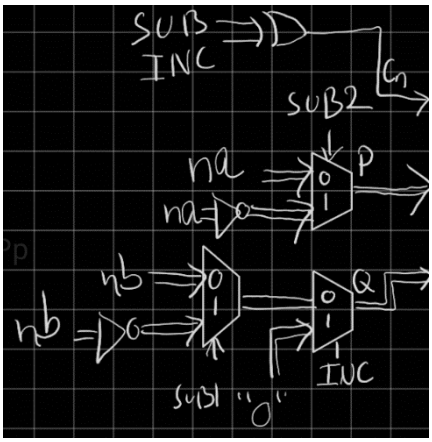
1.1 Première Idée

Une première façon de faire serait de séparer l'ALU en 4 blocs :

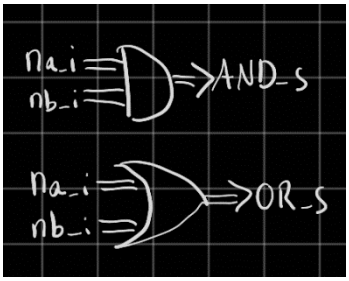


1.1.1 Bloc Select des entrées pour le bloc ADDN, « Selec bloc calcul »

Création de 3 signaux SUB1, SUB2, INC et SUB. SUB1 correspond à l'opcode « 010 », SUB2 à l'opcode « 011 », INC à l'opcode « 100 » et SUB au résultat de SUB1 XOR SUB 2.

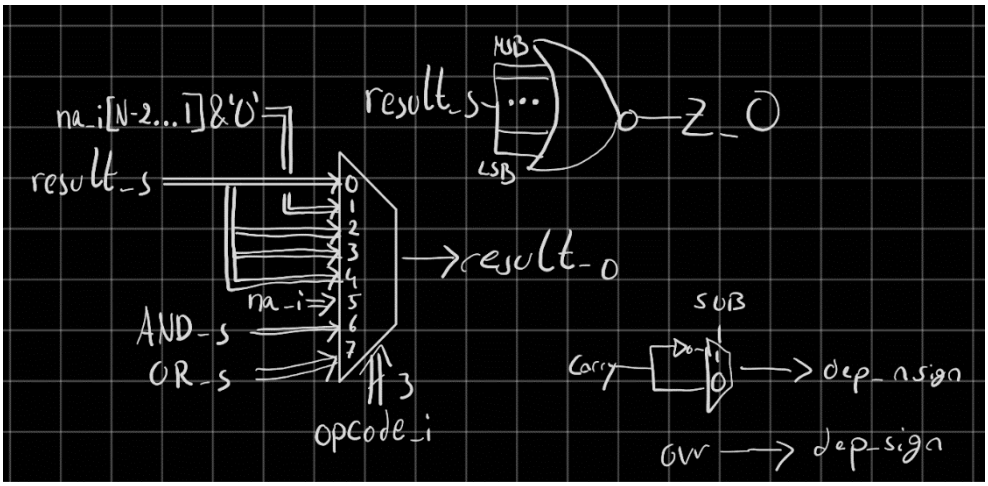


1.1.2 Bloc Logic AND OR



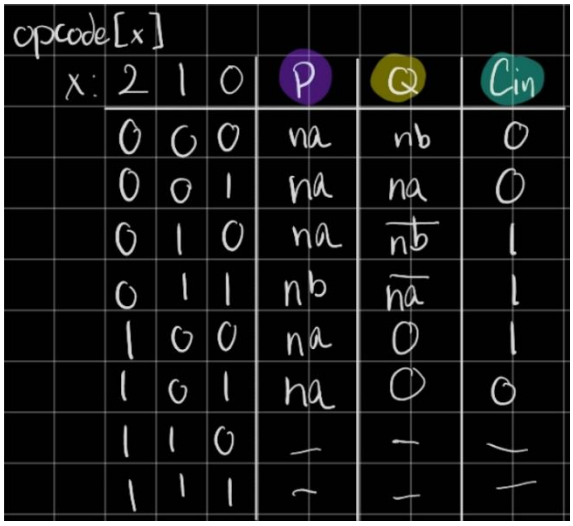
1.1.3 Bloc Select pour result_o

Utiliser un MUX 8 à 1 pour choisir quel résultat envoyer. Pour l'entrée 0 à 4 (sauf l'entrée 1), envoyer le résultat sortant du bloc ADDN. Pour l'entrée 4, effectuer un décalage à gauche et renvoyer ce résultat. Pour l'entrée 5, envoyer directement na_i. Pour l'entrée 5, envoyer le résultat de l'opération na_i AND nb_i. Pour l'entrée 6, envoyer le résultat de l'opération nb_i OR nb_i.



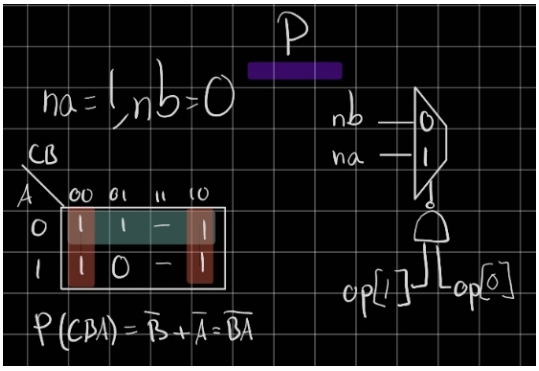
1.2 Idée finale

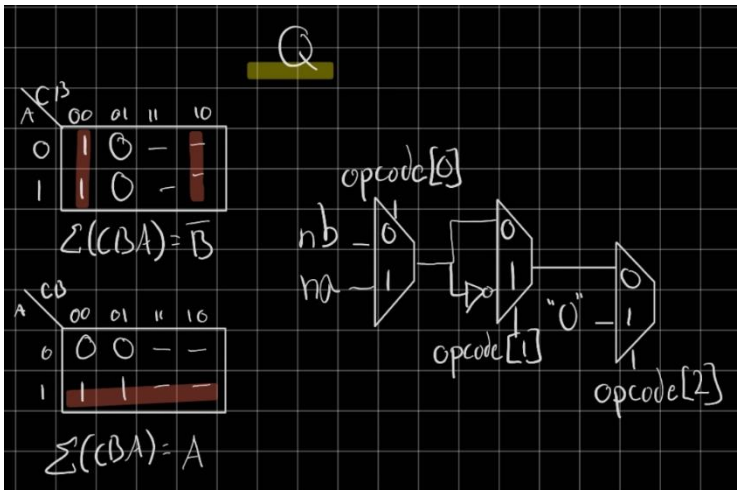
Pour l'idée finale, nous nous sommes basés sur l'idée initiale mais au lieu de choisir quel résultat renvoyer en sortie après le bloc ADDN, nous avons réduit le choix final à 2 possibilités (result_s : sortie du bloc ADDN et porte_s : sortie du bloc Logic). Ainsi, nous passons d'un MUX 8 à 1 à un MUX 2 à 1. Ainsi, nous séparerons l'ALU en 4 parties : un bloc pour gérer les entrées à envoyer au bloc ADDN, un bloc ADDN qui reprend l'additionneur effectué lors du lab2, un bloc pour les portes logic AND et OR et un dernier bloc qui s'occupera des dépassements, de la sortie z_o et du result_o.

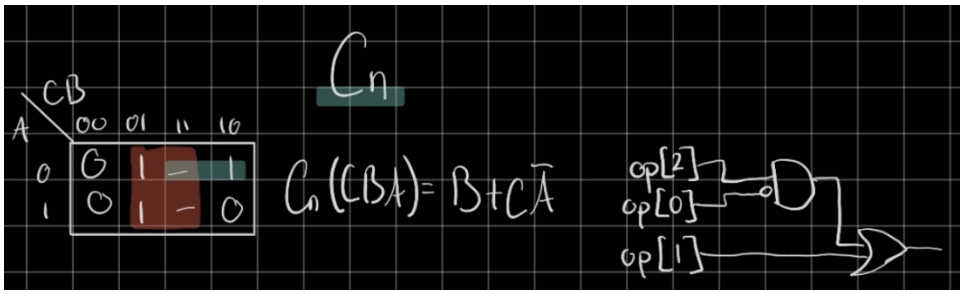


1.2.1 Bloc Select pour les entrées du bloc ADDN

Il est possible de séparer ce bloc en trois parties : un premier décidant de l'entrée P, un deuxième choisissant l'entrée Q et le troisième s'occupant de l'entrée des carry-in. Un des éléments importants de ce bloc est la décomposition de l'opcode. En effet, dans l'idée initiale, nous avions utilisé l'opcode complet pour chaque MUX décidant des entrées de notre bloc ADDN. Or, après analyse, il se trouve qu'il est possible d'utiliser chaque bit de l'opcode de manière indépendante, permettant ainsi de réduire le nombre d'unité logique en supprimant des comparateurs.
(*Pour des raisons de lisibilité, les vecteurs de bits sont dessinés avec une ligne simple au lieu de deux lignes*)

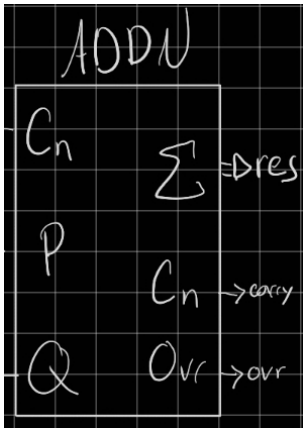






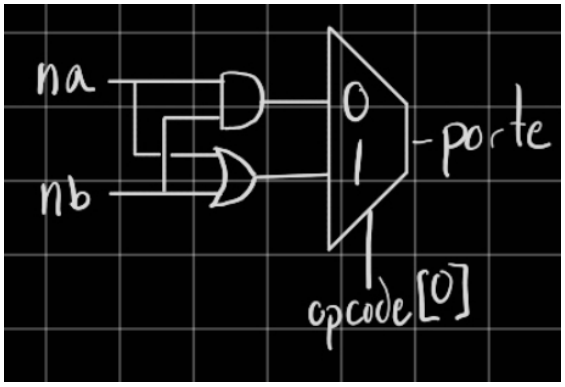
1.2.2 Bloc ADDN

Pour ce bloc, nous avons repris l'additionneur effectué lors de précédent laboratoire et apporté quelques modifications au niveau du VHDL pour pouvoir l'appliquer dans notre ALU.



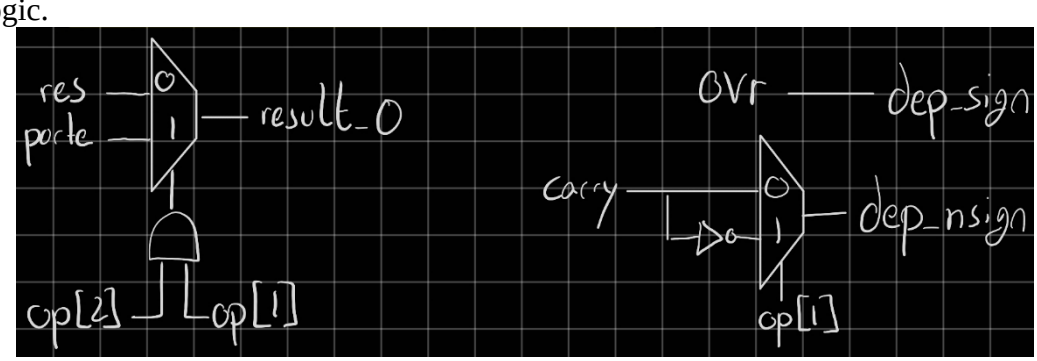
1.2.3 Bloc Logic AND et OR

Ce bloc calcule en interne l'opération AND ou OR et renvoie en sortie le résultat à l'aide d'un MUX 2 à 1 et d'un sel correspondant au LSB de l'opcode.



1.2.4 Bloc dep_sign, dep_nsign, z_o et result_o

Ce bloc interprète les dépassements venant du bloc ADDN, vérifie si le result_s est nul ou non à l'aide d'une comparaison. De plus, result_o est désormais choisis à l'aide d'un MUX 2 à 1 qui décide à l'aide de l'opcode[2..1] s'il le résultat demandé est un calcul issu du bloc ADDN ou du bloc Logic.



Simulation automatique

Log de la simulation automatique de l'ALU pour 6 bits :

```
# vsim -voptargs=""+acc"" work.alu_top_tb -GVAL_N=6
# Start time: 22:55:39 on Mar 27,2024
# ** Note: (vsim-3812) Design is being optimized...
# Loading std.standard
# Loading std.textio(body)
# Loading ieee.std_logic_1164(body)
# Loading ieee.numeric_std(body)
# Loading ieee.math_real(body)
# Loading work.logger_pkg(body)
# Loading work.project_logger_pkg
# Loading work.common_pkg(body)
```

```
# Loading work.alu_top_tb(test_bench)#1
# Loading work.alu_nbits_top(struct)#1
# Loading work.addn_full(struct)#1
# Loading work.addn(flot_don)#1
# Loading work.addn(flot_don)#2
# 1
# ** Note: >> [NOTE]    :  @ 1 ns Start of simulation
#    Time: 1 ns  Iteration: 0  Instance: /alu_top_tb
# ** Note: >> [NOTE]    :  @ 500516 ns
# +--------------------+
# | FINAL REPORT       |
# |--------------------+
# | Nb warnings = 0
# | Nb errors   = 0
# |
# | Verbosity level is : note
# |
# | *** VOUS ETES LES MEILLEURS ***
# | *** Bravo, pas d'erreurs     ***
# |
# | END OF SIMULATION
#    Time: 500516 ns  Iteration: 0  Instance: /alu_top_tb
```

Tests de validation

Opcode	na_i	nb_i	z_o	dep_nsign	dep_sign	result_o
000	127	1	0	0	1	128
000	15	10	0	0	0	25
000	255	1	0	1	1	0
001	8	0	0	0	0	16
001	127	0	0	0	1	254
001	128	0	1	1	1	0
010	20	10	0	0	0	10
010	10	20	0	1	0	246
010	127	127	1	0	0	0
011	10	20	0	0	0	10
011	128	10	0	1	1	242
011	20	10	0	1	0	246
100	20	0	0	0	0	21
100	127	0	0	0	1	128
100	255	0	1	1	1	0
101	255	0	0	0	0	255
101	128	0	0	0	0	128
110	127	64	0	-	-	64
110	32	255	0	-	-	32
111	127	64	0	-	-	192
111	32	255	0	-	-	255

2 Question

2.1 Comparaison

Expliquer comment vous pouvez obtenir à l'aide de votre ALU, le résultat des tests :
na_i < nb_i ou na_i > nb_i

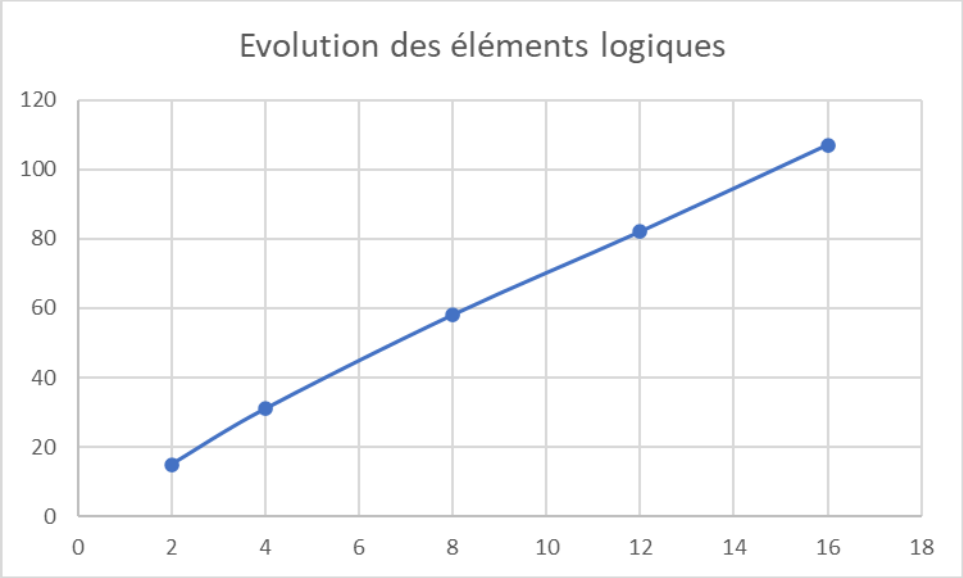
On peut trouver le résultat de ces 2 tests à l'aide des deux fonctions sub. En effet, on peut tout simplement effectuer na_i – nb_i et selon le résultat de ce calcul, on pourra en déduire les réponses à ces deux comparaisons. Dans le cas des nombres signés, il suffit de prendre le bit de signe (MSB), si ce dernier vaut 1 alors cela indique que nb_i est plus grand que na_i. Dans le cas des nombres non-signés, il va falloir se tourner plutôt vers dep_nsign car si nb_i est plus grand que na_i, on va avoir un dépassement de présent car il n'est pas possible d'avoir un nombre négatif dans le monde des non-signés.
A noter, qu'il est également possible d'utiliser l'opération nb_i – na_i à la place de na_i – nb_i. Il faudra juste faire attention à ajuster les résultats pour obtenir les bons résultats.

Nombre d'éléments logique

Lors de la synthèse avec Quartus, on nous demande d'observer la quantité d'éléments logiques utilisée par l'application pour composer l'entièreté de notre ALU. Pour ce faire, nous devons utiliser trois tailles de bits différentes pour notre analyse. Les tailles sont N = 4, 8 et 12. Voici nos résultats :

- 2bits : 15 éléments logiques
- 4 bits : 31 éléments logiques
- 8 bits : 58 éléments logiques
- 12 bits : 82 éléments logiques
- 16 bits : 107 éléments logiques

Pour avoir une meilleure vision de ce que cela représente, nous avons ajouté quelques tests supplémentaires avec 2 bits et 16 bits, et nous avons représenté les résultats sur un nuage de points.
Au début, nous pensions que le nombre d'éléments logiques pourrait croître de manière exponentielle, comme nous l'avons vu en classe avec les portes de comparaison. Mais, en observant la tendance sur le nuage de points, nous constatons que notre fonction est plutôt linéaire (bien qu'elle ne le soit pas complètement). En effet, nous remarquons que de 2 bits à 4 bits, nous avons une augmentation d'un facteur d'environ 2 du nombre d'éléments logiques. C'est la même chose quand on passe de 4 bits à 8 bits ou de 8 bits à 16 bits. Nos portes logiques augmentent de manière significative, ce qui est normal puisque nous avons besoin de plus de portes, que ce soit des AND/OR, des comparateurs ou des multiplexeurs. Cependant, cela est moins significatif que lorsque nous avions un code différent de celui que nous avons maintenant, où nous ne considérions pas seulement un bit de l'opcode, mais l'opcode dans son ensemble



Conclusion

Ce laboratoire nous a permis d'approfondir nos connaissances sur les principes de fonctionnement et la conception d'une ALU, tant au niveau du schéma de blocs que de la description en VHDL. Il nous a également fait réaliser l'importance, dans la conception d'un projet tel que celui-ci, de prendre le temps de bien analyser comment développer des fonctions efficaces. Celles-ci doivent nous permettre d'optimiser la quantité de logique nécessaire, plutôt que de se lancer précipitamment, comme nous avons pu le faire initialement.

Date : 27.03.2024

Noms des étudiants : Emily Baquerizo et Rafael Dousse