HEIG VD | REDS Institut Reconfigurable and Embedded Digital Systems

Prof. Romuald Mosqueron

Prof. Olivier Auberson

**Laboratoire de systèmes logiques**
semestre automne 2024 - 2025

# Laboratoire ALU - Comparateur - Checksum

## Informations générales

Le rendu pour ce laboratoire se fera **par groupe de deux**, chaque groupe devra rendre son travail avant la date mentionnée sur Cyberlearn.

Ce laboratoire se déroule sur **une séance** et sera évalué de la façon suivante :

- Evaluation du circuit rendu
- Evaluation des réponses aux questions

**NOTE 1** : Afin de ne pas avoir de pénalité pensez à respecter les points suivants

- Toutes les entrées d'un composant doivent être connectées. (-0.1 sur la note par entrée non-connectée)
- Lors de l'ouverture de Logisim, bien préciser vos noms en tant que User
- Ne pas modifier (enlever/ajouter/renommer) les entrées/sorties déjà placées
- Contrairement à ce que vous avez pu voir en cours, merci de ne pas utiliser des portes XOR sur plus d'un bit.

**NOTE 2** : Lors de la création de votre circuit, tenez compte des points suivants afin d'éviter des erreurs pendant la programmation de la carte FPGA :

- Nom d'un circuit $\neq$ Label d'un circuit
- Nom d'un signal (Pin) $\neq$ Label et/ou Nom d'un circuit, toutes les entrées/sorties doivent être nommées
- Les composants doivent avoir des labels différents

**NOTE 3** : Nous vous rappelons que si vous utilisez les machines de laboratoire situées au niveau A, il ne faut pas considérer les données qui sont dessus comme sauvegardées. Si les machines ont un problème, nous les remettons dans leur état d'origine et toutes les données présentes sont effacées. Pensez à sauvegarder votre travail sur un autre support.

## Outils

Pour ce laboratoire, vous devez utiliser les outils disponibles sur les machines de laboratoire (A07/A09) ou votre ordinateur personnel avec Logisim installé.

⚠ **La partie programmation d'une FPGA ne peut se faire que sur les ordinateurs présents dans les salles (A07/A09).**

## Fichiers Logisim fourni

Vous devez télécharger à partir du site Cyberlearn le projet Logisim dédié à ce laboratoire.

Le projet contient certaines des entités que vous allez réaliser dans le cadre de ce laboratoire. Vous devrez compléter ces entités et en créer de nouvelles afin de réaliser les fonctions demandées.

**De plus, ne modifiez surtout pas les noms des entrées/sorties déjà placées dans ces entités et n'ajoutez pas d'entrée/sortie supplémentaires.**

## Conseil sur l'organisation du laboratoire

Pour permettre un suivi de ce laboratoire, vous devez remplir le fichier Excel mis à disposition sous Teams.

Ce laboratoire se déroule sur **4 séances**. vous pouvez suivre l'organisation suivante pour gérer votre travail sur ce laboratoire :

| **séance** | **Étapes à travailler** |
|---|---|
| 1 | Développement de l'ALU |
| 2 | Développement du comparateur |
| 3 | Développement du checksum |
| 4 | Finalisation du travail |

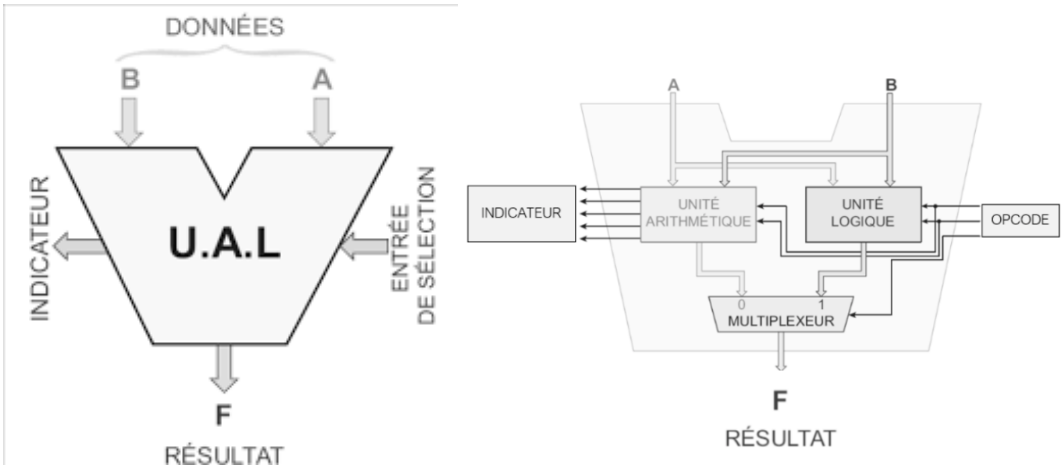
# 1 Contexte du laboratoire

L'objectif principal de ce laboratoire est la réalisation d'une unité arithmétique et logique (ALU) 4 bits. Cette ALU sera ensuite utilisée dans deux applications pratiques différentes : un comparateur et un système de calcul de checksum.

Vous trouverez ci-après quelques définitions utiles concernant l'ALU à réaliser durant ce laboratoire.

## 1.1 Définition de l'ALU

Il s'agit du composant électronique essentiel au fonctionnement d'un CPU, d'un GPU ou d'un micro-contrôleur, puisqu'il est chargé d'effectuer des opérations logiques et arithmétiques (soustraction, addition, multiplication ...). Il s'agit en réalité du composant qui exécute les calculs.




Le composant prend en entrée 2 valeurs A et B, effectue une opération choisie et en ressort un résultat F.

L'ALU renvoie également un indicateur. Cet indicateur permet de surveiller l'état du composant et les erreurs survenues : division par zéro, dépassement de capacité (overflow), Opcode inconnu ...

## 1.2 Les indicateurs NZCV

Dans un processeur, l'ALU met a disposition des indicateurs (aussi appelés flags). Ce sont un ensemble de bits permettant de statuer sur un résultat issu de l'ALU. Nous utiliserons ces indicateurs nommés NZCV permettant d'indiquer si le résultat est **N**égatif, égal à **Z**éro, le bit de **C**arry et le bit d'o**V**erflow).

## 1.3 Définition de l'Opcode

Pour connaitre l'opération à effectuer, un code opération (ou Opcode) est transmis à l'ALU. Ce code permet d'identifier l'unité à joindre et l'opération à effectuer au sein de ce composant.

*Par exemple : si l'opération à effectuer est une soustraction, il faut joindre l'unité arithmétique et sélectionner l'opération de soustraction.*

Les ALU sont présents partout, mais peuvent avoir des fonctionnalités differentes selon l'utilisation qu'on souhaite leur donner *ex : calcul de virgule flottante (FPU).*

## Réalisation du laboratoire

Pour réaliser votre laboratoire, vous devrez d'abord créer un certain nombre de composants puis les utiliser afin de concevoir l'ALU. Cette ALU sera ensuite instanciée et utilisée dans d'autres composants en fonction des besoins. L'idée est de développer un système de A à Z afin que vous puissiez faire chaque étape vous-même et ainsi bien comprendre les concepts vus dans la théorie du cours afin de les appliquer dans un cas pratique.

# 2 ALU

Dans le cadre de ce laboratoire, vous allez implémenter une ALU 4-bits capable de réaliser les opérations arithmétiques et logiques listées ci-dessous.

| Opération | Fonctionnalité | Opcode |
|---|---|---|
| A + B (signé) | Add/Sub | 000 |
| A - B (signé) | | 001 |
| A + B (non-signé) | | 010 |
| A - B (non-signé) | | 011 |
| A and B | Logique | 100 |
| A or B | | 101 |
| A nand B (non-signé) | | 110 |
| A xor B (non-signé) | | 111 |

**Architecture de l'ALU**

Cette ALU est composée de deux sous-blocs (voir figure ci-dessous), chacun réalisant une fonctionnalité particulière. Ces deux blocs contiennent :

— Un circuit combinatoire capable d'additionner et de soustraire deux valeurs,

— Un circuit combinatoire réalisant les fonctions logiques indiquées dans la liste d'opérations de l'ALU

**Schéma bloc de l'ALU**

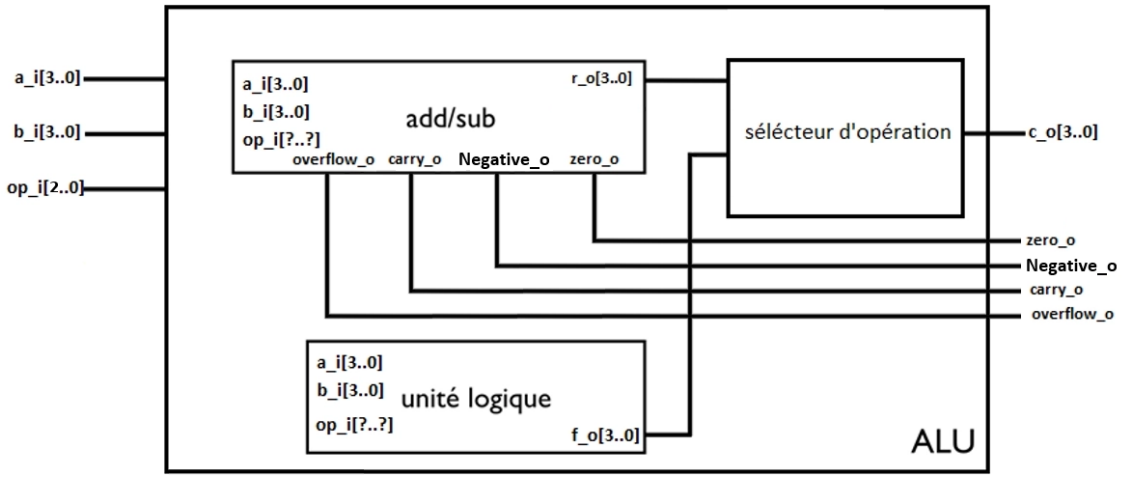



FIGURE 1 – Schéma bloc de l'ALU

| Nom I/O | Description | Nombre de bits |
|---|---|---|
| a_i | Données A | 4 bits |
| b_i | Données B | 4 bits |
| op_i | Opcode | 3 bits |
| c_o | Résultat de l'ALU | 4 bits |
| zero_o | Résultat de l'add/sub nul | 1 bit |
| negative_o | Résultat de l'add/sub négatif | 1 bit |
| carry_o | Signal de carry de l'add/sub | 1 bit |
| overflow_o | Signal d'overflow de l'add/sub | 1 bit |

**Etape 1 : Opcode**

**QUESTION 1** : Reprenez le schéma disponible en Figure 1 et, pour chaque bloc, déterminez et justifiez :
- les bits d'opcode permettant de joindre les différents blocs (l'idée est de répondre aux points d'interrogations dans le schéma bloc).
- les bits d'opcode permettant de sélectionner le bloc à activer.

**Vos réponses doivent être de la forme "les bits d'opcode du bloc add/sub sont les bits [X :Y]".**

**Etape 2-a : Add/Sub 4 bits : carry et overflow**

Dans le laboratoire Add/Sub, vous avez déjà eu l'occasion de créer un additionneur/soustracteur 4 bits. Afin de ne pas répeter trop de travail, nous vous avons mis à disposition les composants `Add1bit`, `Add4bits` et `AddSub4bits`.

Il est important de noter que le composant `AddSub4bits` ne traite ni l'overflow, ni la carry. Dans le composant `AddSub`, ajoutez la correction du carry que vous avez pu effectuer dans le laboratoire Add/Sub. Ajoutez également la gestion de l'overflow.

**NOTE** : Pour rappel, avec $S_A$ le signe de $A$, $S_B$ le signe de $B$, $S_R$ le signe du résultat et $sel$ le sélecteur addition / soustraction, l'équation de l'overflow est :

$$\overline{sel} \cdot \overline{S_A} \cdot \overline{S_B} \cdot S_R + \overline{sel} \cdot S_A \cdot S_B \cdot \overline{S_R} + sel \cdot S_A \cdot \overline{S_B} \cdot \overline{S_R} + sel \cdot \overline{S_A} \cdot S_B \cdot S_R.$$

**Etape 2-a : Add/Sub 4 bits : zero et negative**

Vous allez maintenant implémenter la gestion de les sorties `negative_o` et `zero_o` dans votre composant `AddSub`.

La sortie `negative_o` doit être activée si le résultat de l'addition/soustraction est négatif.

La sortie `zero_o` doit être activée si le résultat de l'addition/soustraction est nul.

**QUESTION 2** : Lorsqu'un overflow se produit, le signe n'est plus géré correctement et la sortie `negative_o` n'a donc pas la bonne valeur. Corrigez ce problème dans votre circuit et expliquez votre raisonnement dans le fichier pdf.

**Etape 3 : Unité logique**

Concevoir le bloc « unité logique » réalisant les opérations logiques de l'ALU. Prévoir un bus de sortie F à 4 bits comme indiqué ci-dessous.

L'entrée op[ ? :?] de l'unité logique sélectionne l'opération logique bit à bit des entrées A et B à réaliser. L'unité logique génère une sortie à 4 bits F[3 :0].

Par exemple, si op = 00, la sortie de l'unité logique doit avoir le comportement suivant :

| Sortie | Equation |
|---|---|
| F[3] | A[3] and B[3] |
| F[2] | A[2] and B[2] |
| F[1] | A[1] and B[1] |
| F[0] | A[0] and B[0] |

**Etape 4 : ALU**

Concevoir le bloc `ALU` en intégrant les deux précédents blocs comme indiqué sur le schéma-bloc.

# 3 Comparateur

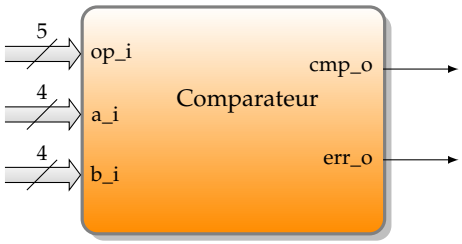
Dans un processeur, l'ALU met à disposition, en plus du résultat de calcul, les flags `NZCV` (**N**egative, **Z**ero, **C**arry, O**v**erflow). Ces flags peuvent typiquement être utilisés pour effectuer des comparaisons entre deux valeurs.

Maintenant que vous avez réalisé une ALU simple, vous allez l'utiliser pour la réalisation du comparateur.

Les opérations à réaliser par le comparateur sont les suivantes :

| **Opération** | **Opcode** |
|---|---|
| A $\geq$ B (signé) | 00001 |
| A < B (signé) | 01001 |
| A $\neq$ B | 10001 |
| A = B | 00011 |
| A $\geq$ B (non signé) | 01011 |
| A < B (non signé) | 10011 |

**Entité du bloc Comparateur**




| **Nom I/O** | **Description** |
|---|---|
| op_i | Opcode pour sélection de l'opération |
| a_i | Données A |
| b_i | Données B |
| cmp_o | Résultat de l'opération |
| err_o | Flag d'erreur |

**Etape 1 : Comparaisons**

Vous allez devoir mettre en place les comparaisons demandées dans le composant `CompareHandling` afin qu'il puisse fonctionner correctement. Pour cela, vous allez synthétiser les fonctions logiques indiquées dans le tableau ci-dessous.

| Opération | Fonction logique |
|---|---|
| A $\geq$ B (signé) | |
| A < B (signé) | |
| A $\neq$ B | |
| A = B | |
| A $\geq$ B (non-signé) | |
| A < B (non-signé) | |

**QUESTION 3** : Malheureusement, nous avons oublié de vous fournir les fonctions logiques du comparateur... C'est donc à vous de faire des essais successifs en observant les flags `NZCV` avec différentes valeurs en entrée. Les fonctions logiques ne dépendent que de ces flags. Remplissez le tableau avec ce que vous pouvez observer.

Implémentez ensuite ces fonctions logiques dans le circuit `CompHandling`.

La sortie `cmp_o` doit être activée si le résultat de la comparaison est vrai.

**Etape 2 : Gestion d'erreurs**

Comme vous l'aurez probablement remarqué, le comparateur utilise un opcode encodé sur 4 bits, mais il n'utilise pas toutes les valeurs encodables sur 4 bits. Un opcode invalide doit donc être détecté et une erreur doit être renvoyée.

**QUESTION 4** : Dressez la table de vérité du signal err_o en fonction de l'entrée op_i et donnez l'équation simplifiée résultante de cette table. Attention : en réfléchissant à la problématique et en vous basant sur les bits qui n'ont qu'une seule valeur possible, vous pouvez grandement simplifier la table de vérité et son nombre d'entrées !

Implémentez ensuite cette équation dans le circuit `CompError`.

**Etape 3 : Implémentation du comparateur**

Dans le circuit `Comp`, instanciez vos trois composants `ALU`, `CompHandling` et `Comp Error` et connectez-les différents signaux pour assurer le bon fonctionnement du comparateur.

**Etape 4 : Intégration sur carte**

Pour l'intégration du comparateur, un circuit `CompMAXV` vous a été fourni.

Intégrez donc ce circuit sur la carte MAXV en suivant le mappage suivant :

| Nom I/O | Description |
|---|---|
| Nom I/O | Position sur carte |
| opcode_i | DS1 |
| a_i[3 :0] | S7 :S4 |
| b_i[3 :0] | S3 :S0 |
| r_o | L0 |
| err_o | L7 |

**Une fois votre intégration terminée, vérifiez le bon fonctionnement du système et faites valider votre travail par l'assistant.**

# 4 Checksum

## 4.1 Qu'est-ce qu'un checksum ?

Un checksum (ou *somme de contrôle*) est une séquence de chiffres et de lettres permettant de vérifier l'intégrité d'un fichier après un transfert entre deux machines. Cela permet de vérifier si le fichier a été altéré lors de son envoi et s'il contient des erreurs.

Pour créer un checksum, on exécute un programme qui contient un algorithme de hashage. il s'agit d'un algorithme qui prend un fichier en entrée de taille quelconque (de quelques Mo à plusieurs Go) et qui exécute un calcul pour en déduire une somme de contrôle. Cette somme de contrôle est toujours de longueur fixe, quelle que soit la taille du fichier.

Une fois générée, cette séquence est transférée indépendemment du fichier. Le destinataire peu ensuite vérifier l'intégrité de son fichier (en générant à nouveau une somme de contrôle ou via un utilitaire de hashage).

Par exemple, les distributions Linux, qui font plusieurs Go fournissent une somme de contrôle pour vérifier que l'image ISO téléchargé soit fiable avant de créer un USB bootable.

## 4.2 Checksum modular sum

Le checksum peut être réalisé de plusieurs façons avec des algorithmes différents. Dans le cadre de ce laboratoire, vous allez réaliser un checksum de type **modular sum**.

Le principe du checksum de ce type est de calculer un checksum sur un ensemble de données afin de tout vérifier en même temps.

Par exemple, avec trois données à transférer, un checksum serait calculé sur ces trois données, puis celui-ci serait envoyé à la suite des données.

**Comment calculer un modular sum :**

Un modular sum se calcule en additionnant les données entre elles sans carry, puis en effectuant le complément à deux du résultat.

# Cadre du checksum pour ce laboratoire

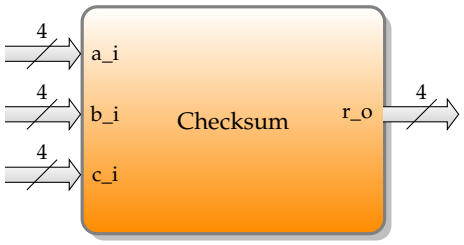
Afin de faciliter la compréhension du processus, nous n'utiliserons pas des gros fichiers avec beaucoup de données, mais simplement des données de 4 bits.

De plus, vous n'allez pas réellement envoyer de données entre deux machines, mais réaliser deux composants : un composant "émetteur" qui va générer un checksum à partir de trois données et un composant "récepteur" qui va simplement vérifier si les trois données correspondent au checksum donné.

Pour le premier composant, il vous suffira de donner trois valeurs 4 bits en entrée pour voir à la sortie le checksum modular sum correspondant. Vous pourrez ensuite prendre note de plusieurs combinaisons de données / checksum.

Pour le second composant, vous allez introduire en entrée les différentes combinaisons de données et checksum dans un ordre aléatoire et vérifier si la combinaison est correcte.

**Entité du calculateur de checksum**



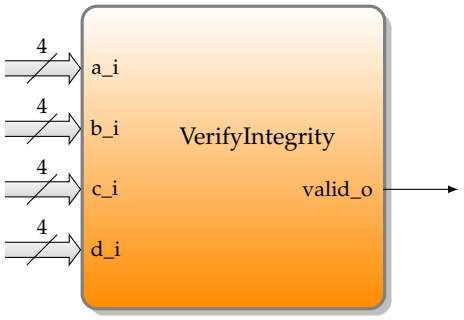

| Nom I/O | Description |
|---|---|
| a_i | Données A |
| b_i | Données B |
| c_i | Données C |
| r_o | Résultat du checksum |

**Etape 1 : Implémentation du calculateur de checksum**

Pour implémenter le calculateur de checksum, utilisez plusieurs blocs ALU developpés précedemment et d'autres composants si nécessaire. Votre circuit devra donc pouvoir calculer un checksum 4 bits sur trois données 4 bits. Votre logique sera à implémenter dans le circuit `checksum`.

Pour rappel : Un modular sum se calcule en additionnant les données entre elles sans carry, puis en effectuant le complément à deux du résultat.

**Entité du vérificateur de checksum**




| Nom I/O | Description |
|---|---|
| a_i | Données A |
| b_i | Données B |
| c_i | Données C |
| d_i | Données D |
| valid_o | Résultat de la vérification |

**Etape 2 : Implémentation du vérificateur de checksum**

Le circuit `VerifyIntegrity` prend quatre entrées (trois données et leur checksum) et permet de calculer un bit de sortie valide égal à 1 lorsqu'aucune altération des données n'est détectée.

Ce circuit reçoit donc quatre données et ne peut pas savoir lequel est le checksum.

Une fois de plus, vous pouvez utiliser plusieurs blocs ALU pour réaliser votre circuit.

**Attention : pour le développement de votre circuit, vous ne devez ni utiliser le composant checksum, ni reproduire le calcul du checksum.**

**Etape 3 : Validation du composant verfiy_integrity par chronogramme**

Afin de valider le fonctionnement global du système, créez un nouveau circuit et nommez le `chronogramme`.

Dans celui-ci, instanciez le composant `VerifyIntegrity` et connectez à ses entrées les éléments suivants :

- une constante 4 bits avec la valeur `0xA`
- une constante 4 bits avec la valeur `0xB`
- une constante 4 bits avec la valeur `0xE`
- un compteur avec 4 bits de données

L'ordre de connexion de ces éléments sur les entrées du composant `VerifyIntegrity` ne doit pas influencer le résultat.

Générez un chronogramme montrant que la sortie du composant `VerifyIntegrity` est activée à un certain moment.

Une fois votre chronogramme généré, répondez aux questions suivantes :

**QUESTION 5** : Quelle est la valeur de checksum attendue pour les constantes définies ?
(utiliser les composants créés précédemment).

**QUESTION 6** : Sauvegarder le chronogramme (capture d'écran). Quelles observations peut-on faire ?
Selon quelles conditions la sortie *VerifyIntegrity* passe-t-elle à 1 ?
Expliquer votre raisonnement dans le rapport au format *.pdf*.

# 5 Rendu

Pour ce laboratoire, chaque binôme devra rendre :

- votre fichier *.circ*
- un fichier au format *.pdf* contenant les réponses aux différentes questions théoriques.

Vous devez déposer les rendus sur Cyberlearn jusqu'à la date indiquée dans l'espace de rendu consacré à votre classe.