

## TP : Circuits combinatoires et séquentiels

### Dans ce TP ...

#### ... vous aurez besoin des pré-requis suivants

- Codage des entiers.
- Éléments constitutifs d'un circuit combinatoire (portes logiques OU, ET, XOR, etc.).
- Calcul booléen.

#### ... vous allez apprendre les notions et compétences suivantes

- Construction d'un circuit combinatoire simple, à base de portes logiques de base.
- Construction d'un circuit combinatoire complexe, réutilisant des composants de votre création.
- Construction d'éléments de mémorisation (registres, mémoires adressables)
- Construction d'un petit circuit séquentiel
- Approches structurelle et comportementale de la description des composants et circuits.

### Outils

Dans ce TD, on va décrire et simuler des circuits en utilisant deux outils :

**Logisim** est un outil open-source à but pédagogique. C'est un programme graphique dans lequel on peut assembler des portes pour créer des circuits de plus en plus gros, puis observer leur comportement lorsqu'on applique des valeurs logiques aux entrées.

**VHDL** est un langage de description de circuit. C'est un standard industriel, moins facile à prendre en main mais plus puissant que Logisim. On utilisera le simulateur GHDL pour simuler des circuits décrits en VHDL, et le visualisateur de traces GTKWaves.

## 1 Prise en main de Logisim - Circuits combinatoires

Logisim est installé sur les machines du département IF, ici :

```
/opt/logisim-2.7
```

Il a été développé en java et se lance en tapant, depuis une ligne de commande shell :

```
java -jar /opt/logisim-2.7/logisim-generic-2.7.1.jar
```

Le logiciel est disponible gratuitement depuis <http://www.cburch.com/logisim/>.

Afin de vous permettre de prendre en main l'outil, suivez le tutoriel suivant, qui vous fait créer un circuit réalisant la fonction xor.

<http://www.cburch.com/logisim/docs/2.7/en/html/guide/tutorial/index.html>



**Validez avec un enseignant le bon fonctionnement de votre xor.**

## 1.1 Additionneur 1 bit

Dans Logisim, créez un nouveau projet. Vous y créerez tous les circuits demandés dans les différentes séances de TP.

Votre premier circuit dans ce projet réalisera l'addition 1 bit. Ce circuit possède 3 entrées *a*, *b* et *c\_in* représentant respectivement les deux valeurs à additionner et la retenue entrante de l'addition. Il possède 1 sortie *s* dénotant la valeur de la somme ainsi que 1 sortie *c\_out* dénotant la valeur de la retenue de sortie. On vous rappelle la définition de *s* et *c\_out* :

- $s = (a \text{ xor } b) \text{ xor } c_{in}$
- $c_{out} = (a \text{ and } b) \text{ or } (a \text{ and } c_{in}) \text{ or } (b \text{ and } c_{in})$

*Attention à ce que la simulation soit bien activée ! Dans le menu, Simulate->Simulation enabled !*

Vous aurez remarqué que, par défaut, les portes de votre circuit ont plus d'entrées que ce que vous utilisez. Logisim permet de changer ces paramètres dans le "cadre des attributs", en bas à gauche de la fenêtre. Ce cadre permet notamment de nommer les entrées (attribut "label") de changer l'orientation des portes (attribut "facing") et leur nombre d'entrées (attribut "number of inputs").

Au delà de l'exécution pas-à-pas, dirigée par le changement des valeurs d'entrées du circuit, que vous avez utilisée dans le tutoriel ci-dessus, Logisim offre un certain nombre d'outils pour valider le comportement de vos petites créations. Ceux-ci sont accessibles par<sup>1</sup> Project > Analyze Circuit.

Logisim vous propose notamment, pour le circuit en cours d'édition :

- La liste de ses entrées et sorties ;
- Sa table de vérité ;
- Une version textuelle de l'expression booléenne réalisée pour chacune des sorties.

 **Vérifiez avec un enseignant que le comportement observé à travers les informations fournies dans la fenêtre d'analyse est bien celui attendu.**

## 1.2 Additionneur 8 bits


Pour construire un additionneur 8 bits, vous n'allez pas (!) copier-coller 8 fois les portes que vous venez d'utiliser pour l'additionneur 1 bit. Plutôt, vous allez encapsuler votre additionneur 1 bit dans un composant que vous pourrez ensuite réutiliser 8 fois pour obtenir l'additionneur 8 bits.

Commencez par suivre le tutoriel mis en place par les auteurs de Logisim, ici :

<http://www.cburch.com/logisim/docs/2.7/en/html/guide/subcirc/index.html>

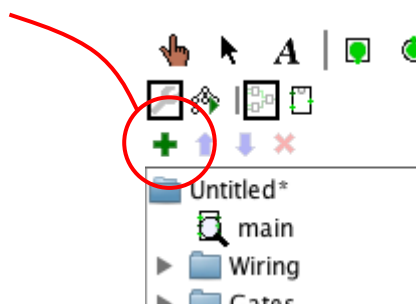
Concentrez-vous sur les parties intitulées **Creating circuits**, **Using subcircuits** ainsi que **Editing subcircuit appearance**. Dans cette dernière section, vous apprendrez à modifier la forme des composants que vous créez et *surtout* à nommer les ports d'entrée/sortie de vos composants.

Nommez les ports de votre additionneur, par exemple *a*, *b* et *c\_in* pour les entrées et *s* et *c\_out*.

 **Vérifiez avec un enseignant votre choix d'apparence pour votre additionneur 1 bit. Celui-ci aura une influence assez forte sur la facilité de construction de l'additionneur 8 bits.**

Changez ensuite le nom de votre circuit (jusqu'ici *main*) pour, par exemple, *add1bit*.

Enfin, créez un nouveau circuit dans le même projet en utilisant la petite croix verte ici :

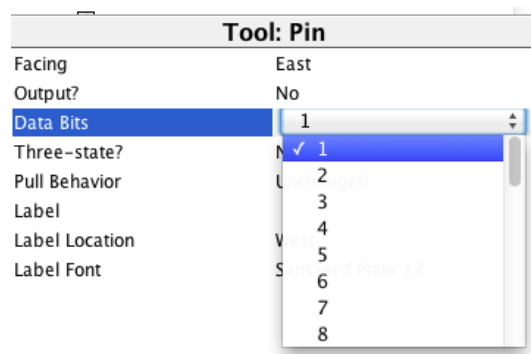


1. voir <http://www.cburch.com/logisim/docs/2.7/en/html/guide/analyze/index.html>

Changez son nom pour add8bits.

Créez maintenant votre additionneur 8 bits en utilisant l'additionneur 1 bit de la section précédente.

Pour le tester, vous pouvez vous servir de “**pin ports**” d'entrée/sortie d'une taille plus grande que celle par défaut (1 bit). Pour cela, sélectionnez l'outil “pin port” depuis la barre d'outils. Puis, dans l'encart en bas à gauche de la fenêtre “**Tool : Pin**” sélectionnez 8 pour le champ “**data bits**” (voir figure ci-dessous).



Pour pouvoir connecter ces ports à vos composants, vous utiliserez des “**Splitter**” qui vous permettront de regrouper des fils. Encore une fois, aidez-vous de la documentation en ligne :

<http://www.cburch.com/logisim/docs/2.7/en/html/guide/bundles/splitting.html>

Testez votre circuit à l'aide de la simulation pas à pas. Vous prendrez bien garde à l'initialisation de la retenue entrante de votre additionneur.

 **Validez avec un enseignant que le comportement observé est bien celui attendu.**

### 1.3 Additionneur / Soustracteur 8 bits

Comme brillamment décrit dans le poly du cours, on peut profiter de la circuiterie de l'additionneur pour créer un circuit additionneur/soustracteur. Les modifications nécessaires se basent sur le calcul du complément binaire d'un des chiffres paramètres de la soustraction. Pour deux entiers dont on veut calculer la différence  $A-B$ , on calcule le complément du second,  $\bar{B}$ , chaque bit de  $\bar{B}$  étant défini par la négation du bit correspondant dans  $B$ . Puis, on calcule l'addition  $A+\bar{B}+1$ .

Pour réaliser cela à partir de votre additionneur 8 bits, vous utiliserez la retenue entrante, qui permettra maintenant de décider si on souhaite une addition (elle vaut alors 0) ou une soustraction (elle vaut alors 1). Cette entrée conditionne la complémentation de la deuxième entrée de l'opération, ainsi que la valeur de sa retenue d'entrée. Le choix entre  $B$  et  $\bar{B}$  se fait à l'aide d'un xor pour chaque bit de  $B$ , prenant comme première entrée  $B$  et comme seconde entrée la retenue entrante de votre opération.

Pour rendre votre circuit plus lisible, on vous encourage à encapsuler les 8 xor dans un composant dédié.

 **Validez avec un enseignant le comportement de votre additionneur/soustracteur.**

## 2 Prise en main de VHDL

Cette partie du TP a besoin que vous lisiez l'annexe B du poly. Arrêtez-vous avant B-3, et enchaînez sur le TP.

Avant tout, décompressez le fichier `tp1-vhdl.tgz`.

### 2.1 Passer un circuit combinatoire au banc de test

#### 2.1.1 Entités et instances

Ouvrez dans un éditeur de texte `fulladder.vhdl` et `testbench_fulladder.vhdl`.

Le premier contient une *entité* `fulladder` qui est un full-adder (légèrement saboté). L'entité décrit une boîte noire, une interface. L'*architecture* d'une entité décrit une implémentation de cette boîte noire. En Logisim, on a d'abord décrit l'architecture de `add1bit`, puis on l'a emballée dans une entité pour pouvoir la réutiliser. On peut avoir plusieurs architectures par entité, et c'est utile en pratique, d'où cette syntaxe un peu lourde. Dans ce TP, on n'aura toujours qu'une architecture par entité.

Le second fichier, `testbench_fulladder.vhdl`, contient une entité `testbench_fulladder`<sup>2</sup> qui est un banc de test. Ce banc de test décrit une *instance* de `fulladder`, et un processus séquentiel `process` qui applique des entrées à ce composant.

Vérifiez que vous comprenez la différence entre une entité et une instance. Indice : c'est la même différence qu'entre une fonction et un appel de fonction, mais pour le matériel. En Logisim, on a utilisé plusieurs instances de l'entité `add1bit`.

Faites un dessin correspondant à l'entité `testbench_fulladder` : un gros rectangle qui représente la boîte noire, et des sous-composants dedans, comme en Logisim.

 **Validez ce dessin avec un enseignant.**

#### 2.1.2 Simulation du VHDL

Ajoutez les 6 cas de test qui manquent au banc de test, par copier-coller.

Pour simuler, nous utiliserons le simulateur VHDL libre `ghdl`. La compilation nécessite deux étapes :

- `ghdl -a fulladder.vhdl`  
"analyse" le code source, vérifie la syntaxe, et compile le fichier `vhdl` en un fichier objet `fulladder.o` et un fichier de configuration `work-obj93.cf`.  
Essayez cette commande. Constatez que nous avons saboté `fulladder.vhdl` pour que cette commande produise des erreurs. Réparez-le.  
Quand cette commande passe, jetez un œil à `work-obj93.cf`. Ensuite, même traitement pour l'autre fichier :  
`ghdl -a testbench_fulladder.vhdl`  
Re-jetez un œil à `work-obj93.cf`.
- `ghdl -e testbench_fulladder`  
"élabore". Ici l'argument `testbench_fulladder` est un nom d'entité VHDL (pas de fichier). Cette commande crée un exécutable `testbench_fulladder` qui est un simulateur de l'entité correspondante. Essayez.  
Techniquement, cette commande construit l'exécutable en liant (*link*) les `.o` précédents. Pour savoir quoi lier, elle lit `work-obj93.cf`.

Le simulateur obtenu, ici `testbench_fulladder`, peut s'exécuter dans un terminal : tapez

```
./testbench_fulladder --help
```

qui liste les options de votre simulateur. Les deux qui nous intéressent sont

- `--stop-time` sans quoi la simulation tourne indéfiniment (comme votre téléphone portable).
- `--vcd` qui produit un fichier de trace (un chronogramme) que nous pourrions observer avec `gtkwave`.

Essayez donc :

```
./testbench_fulladder --stop-time=20ns --vcd=testbench_fulladder.vcd
```

---

<sup>2</sup>. Ce n'est pas obligatoire que le fichier ait le même nom que l'entité mais c'est en général une bonne idée. Comme pour les classes en C++, en fait.

```
gtkwave testbench_fulladder.vcd
```

Dans la fenêtre GTKWave, la colonne du milieu contient les signaux que vous voulez voir apparaître dans le chronogramme : tirez-y à la souris votre signal `testcin` depuis la colonne de gauche. Il faut aussi cliquer sur Time->Zoom->Zoom Best Fit pour mettre à une échelle correcte l'axe des temps. Observez les deux sorties du *full adder*. Constatez qu'il ne marche pas du tout. C'est qu'on aussi a saboté ses équations logiques. Réparez-les.

 **Faites valider par un enseignant une fenêtre GTKWave qui montre un fulladder qui marche. En attendant qu'il arrive, écrivez un Makefile.**

## 2.2 Construire un additionneur paramétré

### 2.2.1 Description structurelle de l'additionneur

Dans le fichier `adder.vhdl` vous avez plusieurs nouveautés.

- Le type `std_logic_vector` décrit un vecteur de bits. On décrit sa taille par les indices min et max, inclus tous deux. Si `x` est un `std_logic_vector`, alors `x(3)` est le bit numéro 3.
- Des paramètres génériques de l'architecture peuvent être donnés avec le mot-clé `generic`. On l'utilise ici pour définir la taille `n` de l'additionneur. Ici c'est un entier, mais on peut avoir des `generic` de n'importe quel type. Une remarque importante : pour simuler une architecture, on aura besoin que tous ses paramètres soient instanciés.
- La boucle `generate for` itère dans l'espace, pas dans le temps.

Ajoutez quelques tests d'addition. Vérifiez en simulant que vous calculez bien la somme. L'exercice est surtout de trouver les bonnes lignes de commande.

Déplacez la ligne `c(0) <= cin` ; à la fin, et constatez que cela ne change rien à la simulation.

 **Expliquez ce miracle à un enseignant.**

### 2.2.2 Une description plus précise des aspects temporels

Le seul endroit dans lequel un calcul est réalisé est le `fulladder`. Pour observer un chronogramme plus réaliste, on va spécifier le temps de ce calcul. VHDL permet d'écrire des choses comme

```
a <= b and c after 5 ps;
```

Ici `ps` désigne la picoseconde (qui vaut combien de secondes ?).

Jeu numéro 1 : le délai typique d'une porte XOR est de 15ps et le délai typique d'un inverseur, d'un AND ou d'un OR (à deux entrées) est de 10ps.

Raffinez votre entité `fulladder`. Observez le chronogramme de l'addition `01111111 + 00000001`. Remarque : dans GTKWave, double-cliquer sur un nom de signal `std_logic_vector` dans la fenêtre du milieu montre tous ses bits séparément.

 **Faites valider par un enseignant.**

Jeu numéro 2 (si vous êtes en avance) : En CMOS, on n'a droit qu'au NAND, NOR et inverseur, avec un délai de 5ps chacun. Réécrivez les équations logiques du full adder pour n'utiliser que ces portes. Cherchez des équations logiques qui minimisent le temps de propagation de retenue.

Jeu numéro 3 (si vous êtes très en avance) : En fait on contrôle le délai des portes à 20% près. Pour modéliser ceci, mettez (au hasard) certains délais à 4 ps et certains délais à 6ps. Observez l'apparition de transitoires (*glitches*) dans la simulation.

### 3 Registres et mémoires

Jusqu'à présent on s'est contenté de décrire des circuits combinatoires : la valeur des sorties de votre additionneur à un instant donné  $t$  ne dépend pas du passé, mais uniquement de la valeur de ses entrées à cet instant précis.

C'est très bien, mais on ne va pas aller très loin avec ça. Très vite, on va vouloir faire des calculs qui vont nécessiter plusieurs opérations *successives* : à chaque étape, une opération combinatoire produira un résultat temporaire qui sera une opérande d'une opération à l'étape suivante. Ces étapes sont les fameux instants successifs de l'exécution de notre programme et de tels circuits sont appelés *circuits séquentiels*.

Pour pouvoir construire de tels circuits, il va nous falloir des petits circuits pour mémoriser des valeurs, *d'un instant sur l'autre*. Vous allez maintenant découvrir comment on construit de tels composants qu'on nomme *registres*, en partant d'éléments simples appelés *bascules* (en anglais *latch*), eux-mêmes construits à partir de portes logiques.

#### 3.1 Description comportementale des registres en VHDL

##### 3.1.1 Mais d'abord, un générateur d'horloge

Pour commencer nous allons simuler le plus simple des circuits séquentiels : un générateur d'horloge.

Un exemple de code VHDL est donné dans le fichier `clock.vhdl`.

Allez le lire et comprenez-le bien. Si vous avez l'impression qu'il y a des fautes de frappe dedans, c'est normal. Simulez-le.

##### 3.1.2 Description du comportement du registre

Un registre est défini en VHDL, dans une architecture, par un process. Ce process doit réagir à un évènement sur l'horloge, ce qu'on écrit `process(clk)`. Voici une manière d'écrire un registre à un bit :

```
process(clk)
begin
    if rising_edge(clk) then
        q <= d;
    end if;
end process;
```

Voici un registre avec clock enable :

```
process(clk)
begin
    if rising_edge(clk) then
        if enable = '1' then
            q <= d;
        end if;
    end if;
end process;
```

Remarquez qu'on n'a pas écrit `process(clk,enable)`. Ce serait syntaxiquement correct : la parenthèse après le process décrit la *liste de sensibilité* du processus, c'est à dire la liste des signaux sur lesquels une transition réveille le processus. Mais on veut construire un registre avec *enable synchrone* : il doit prendre en compte `enable` uniquement au front montant de l'horloge. Inutile donc de réagir à un évènement sur `enable`.

**Question 1** Emballez ces deux morceaux de code dans des entités "registres n bits".

### 3.1.3 Un compteur

**Question 2** Construisez un compteur. Vous avez besoin de registres avec reset, écrivez-les en vous inspirant de ce qui est donné.

Construisez un banc de test qui alimente votre compteur par un générateur d'horloge, et regardez-le compter dans gtkwave.

**Question 3** Un peu d'*overclocking* ! Augmentez la fréquence d'horloge jusqu'à ce que votre compteur ne marche plus. Pour cette question on pourra modifier le générateur d'horloge pour qu'il prenne un paramètre générique qui donne sa fréquence.

 **Faites valider par un enseignant.**

Nous avons décrit nos registres uniquement de manière comportementale : pour la simulation c'est suffisant. On décrit ce qu'ils font, mais on ne sait pas comment les construire.

La suite va répondre à cette question, et pour cela nous revenons à Logisim.

*Si vous êtes en avance ou si vous voulez faire des heures sup, vous pouvez en parallèle implémenter les mêmes choses en LogiSim et en VHDL. Dans ce cas, le but est d'arriver à une seconde architecture de votre entité registre, qui sera cette fois structurelle. Vous devrez aussi chercher comment on choisit quelle architecture est associée à quelle instance de quelle entité. C'est relativement bien expliqué ici :*

[http://www.pldworld.com/\\_hdl/1/www.ireste.fr/fdl/vcl/lesd/les\\_6.htm](http://www.pldworld.com/_hdl/1/www.ireste.fr/fdl/vcl/lesd/les_6.htm)

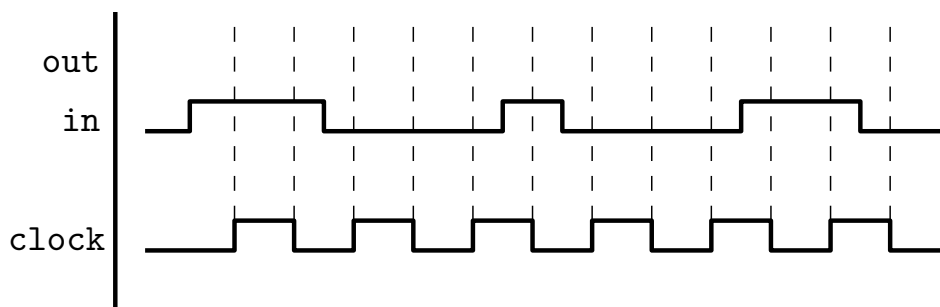
## 3.2 Description structurelle des registres en Logisim

NB : Créez chacun des circuits demandés séparément dans Logisim, notamment pour pouvoir facilement y revenir plus tard.

### 3.2.1 Latch et flip-flop

Dans le poly du cours (chapitre 4.2), vous trouverez une description de la bascule (latch) et du registre 1 bit (flip-flop). Construisez ces deux circuits dans Logisim.

Remplissez le chronogramme suivant pour vous assurer que vous comprenez le comportement du flip-flop.



### 3.2.2 Flip-flop avec reset

Compléter votre flip-flop afin de permettre sa remise à zéro. Pour cela, créez un nouveau circuit qui utilise votre flip-flop précédent et possède une entrée supplémentaire *reset*.

### 3.2.3 Registre à commande de chargement

Enfin, ajoutez une commande de chargement à votre registre. Pour cela, encapsulez votre flip-flop à reset, et étendez-le une entrée supplémentaire *enable* : la valeur du registre n'est modifiée pour prendre la valeur d'entrée que si *enable* est vrai. Sinon, la sortie conserve sa valeur actuelle.

Note : Vous venez de construire un registre complet à 1 bit, qui vous permet de conserver une information sur plusieurs cycles d'exécution et de changer cette information à la demande, grâce à la commande *enable*.

 **Faites valider par un enseignant.**

### 3.2.4 Registre à n bits

On ne sait pour l'instant que mémoriser 1 bit. Pour aller plus loin, il va nous falloir de quoi mémoriser des paquets de bits. En pratique, la taille des registres est très liée à d'autres éléments de l'architecture concernée : taille des bus, taille de la mémoire. Pour ce TP, nous nous arrêterons à des registres 8 bits. Contrairement aux différentes bascules et au flip-flop que nous avons construits jusque-là, la complexité ne se situe pas dans le comportement temporel mais dans le nombre des boîtes et des connexions constituant le circuit. Il va s'agir de coller côte-à-côte 8 registres 1 bit et de les synchroniser avec la même horloge.

### 3.2.5 Un petit compteur

Nous allons maintenant concevoir un circuit qui va nous permettre de compter. La valeur du compteur sera enregistrée dans un registre 8 bits. Nous compterons donc de la valeur 0 à la valeur  $2^8 - 1$ .

Pour cela, vous aurez besoin de votre additionneur 8 bits réalisé plus tôt (voir section 1.2).

 **Faites valider par un enseignant.**

Parce que c'est trop beau, vous pouvez maintenant lancer une simulation en choisissant *click enabled* dans le menu *Simulate*. Vous pouvez même régler la fréquence assez haut pour vérifier rapidement que votre compteur se comporte "bien comme il faut" lorsqu'il atteint la borne sup.

## 3.3 Mémoires adressables

Lorsqu'on veut stocker beaucoup de données en même temps, utiliser des registres indépendants implique une trop grande complexité d'interconnexion (i.e. il y a trop de filasse). On invente alors (tadam) des mémoires adressables.

Dans cette partie, vous allez implémenter une telle mémoire adressable. Elle sera petite (8 mots de 8 bits), mais vous constaterez également qu'étendre ce composant à des capacités plus grandes est simple.

### 3.3.1 Observation du comportement d'une mémoire

A la fin de la séance, vous aurez construit une mémoire adressable de 8 mots de 8 bits. Ce composant est par ailleurs déjà implémenté dans Logisim. Testez une RAM de cette taille, en suivant le tutorial sur le site de Logisim :

<http://www.cburch.com/logisim/docs/2.3.0/libs/mem/ram.html>

Choisissez une RAM de type *Separate load and store ports* : c'est celle-là que vous allez construire dans la suite du TP.

### 3.3.2 Réalisation d'un démux 1 vers 8 de 1 bit

Un dé-multiplexeur est un composant qui réalise un aiguillage. On se donne 1 entrée de *données*,  $k = \lceil \log_2 n \rceil$  entrées de sélection et  $n$  sorties de *données*. À l'intérieur, il faut trouver la bonne combinaison de portes logiques .... À vous de jouer.

#### Réalisation d'un mux 8 vers 1 de 8 bits

Cet aiguillage se fait aussi dans l'autre sens, pour pouvoir envoyer une des  $n$  entrées vers l'unique sortie. Réalisez ce composant *multiplexeur n vers 1*, avec  $n=8$ , pour des entrées booléennes (1 bit chacune) et une sortie booléenne.

Cet aiguillage manipule des fils (entrées et sortie) de taille 1, on parle de multiplexeur 8 vers 1 à 1 bit. On peut le généraliser pour que les informations à sélectionner soient de taille quelconque. Ça se fait de manière hiérarchique, par exemple en encapsulant des multiplexeurs de  $m$  bits pour faire un multiplexeur de  $2m$  bits.

Réalisez un multiplexeur 8 vers 1 à 2 bits. Puis à 4 bits, puis à 8 bits.



## Construction d'une mémoire de 8 mots de 8 bits

Ça y est ! Vous avez tout ce qu'il vous faut pour réaliser une mémoire. Elle sera d'une taille considérable : 64 bits en tout (8 octets) ! Pour vous aider un peu, elle aura les entrées suivantes :

- une entrée d'adresse A.
- une entrée de données DI (pour *Data In*).
- une entrée de contrôle WE (pour *Write Enable*) qui permet de décider quand on veut écrire la donnée présente sur le bus DI à l'adresse A.
- une entrée `reset`, pour réinitialiser la mémoire.
- une entrée `clk`, pour piloter la mise à jour de la mémoire.

Votre mémoire aura aussi une sortie de données DO (pour *Data Out*) sur laquelle on peut lire la valeur contenue à l'adresse A.

 **Faites valider par un enseignant.**

## 4 Construire une ALU en VHDL

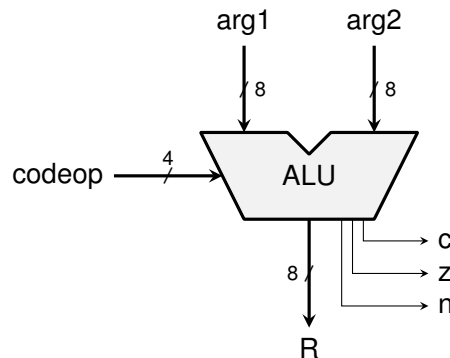
Deux constructions syntaxiques utiles pour les vecteurs de bits :

- `(n-1 downto 0 => '0')` crée un vecteur constant "000000" de taille  $n$ .
- l'opérateur `&` fait la concaténation de deux vecteurs de bits.

Par exemple la constante 1 sur  $n$  bits s'écrit `one <= (n-1 downto 1 => '0') & '1'`;

Par ailleurs le fichier `mux.vhd` contient un exemple de syntaxe pour un multiplexeur à 4 entrées. Il vous faudra ci-dessous un multiplexeur à 16 entrées. Vous pouvez en faire une entité séparée, mais vous pouvez aussi utiliser la syntaxe à base de `select` directement dans votre architecture.

On vous demande de réaliser une ALU (*arithmetic and logic unit*) dont la boîte noire est la suivante :



Elle ne calcule qu'en complément à 2 sur 8 bits. Les sorties *c*, *z*, *n* sont des *drapeaux* qui indiquent respectivement s'il y a un *carry out*, si le résultat est nul (*zero*), si le résultat est négatif.

Le codage de l'opération est donné par la table suivante :

TABLE 1 – Encodage des différentes opérations possibles

codeop	mnémonique	MàJ drapeaux	remarques
0000	<code>arg1 + arg2 -&gt; dest</code>	oui	addition
0001	<code>arg1 - arg2 -&gt; dest</code>	oui	soustraction
0010	<code>arg1 and arg2 -&gt; dest</code>	oui	et logique bit à bit
0011	<code>arg1 or arg2 -&gt; dest</code>	oui	ou logique bit à bit
0100	<code>arg1 xor arg2 -&gt; dest</code>	oui	ou exclusif bit à bit
0101	<code>LSR arg1 -&gt; dest</code>	oui	logical shift right ; bit sorti dans C ; arg2 inutilisé
1000	<code>(not) arg1 -&gt; dest</code>	non	not si arg2S=1, sinon simple copie
1001	<code>arg2 -&gt; dest</code>	non	arg1 inutilisé

Remarque : les codeop non mentionnés dans cette table sont inutilisés pour le moment (réservés pour une extension future...). En attendant, votre ALU peut bien sortir ce qu'elle veut dans ces cas-là : *don't care* ! Écrivez un banc de test qui teste différentes opérations.

### Et s'il reste du temps

On s'use un peu les yeux sur ces chronogrammes.

VHDL vous permet donc d'écrire un banc de test qui compare la sortie de votre ALU avec la sortie attendue, et produit un message d'erreur en cas de désaccord. Vous trouverez tout sur le Ternet, par exemple en l'interrogeant sur les mots-clé *assert* et *report*.