DIU "Enseigner l'informatique au lycée"
UE 3 : Architectures matérielles et robotique, systémes et réseaux
Circuits et Architecture des Ordinateurs
Cahier de TP

Responsable : H. LADJAL
hamid.ladjal@univ-lyon1.fr

https ://diu-eil.univ-lyon1.fr/bloc3/

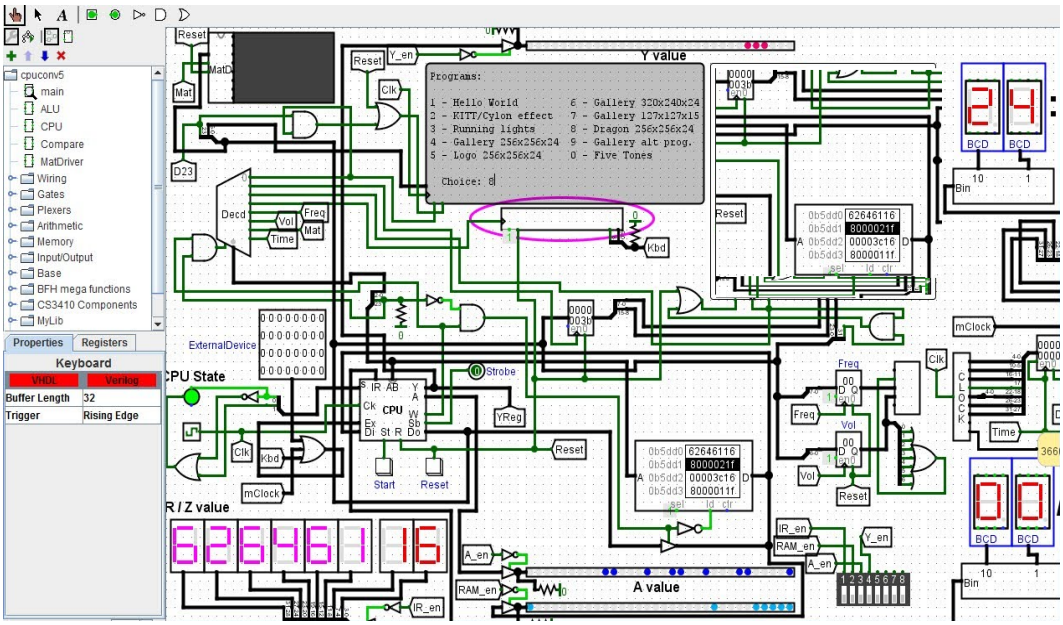


FIGURE 1 – Simulateur de circuits logiques Logisim.

Table des matières

TP1

Objectifs :

— Prise en main de l'outil de simulation LOGISIM
— Duvrir les diffntes fonctionnalite LOGISIM
— Ecrire et dlopper des circuits logiques (combinatoires et sentiels) de base.

Ce tp sinspire fortement des TP de l'UE INF1015L (LIFASR3) developpr H. Ladjal et N. Louvet.
Pour ces TP, on va utiliser le logiciel Logisim pour simuler facilement des circuits logiques : à défaut d'implanter un vrai processeur en Logisim, l'objectif sera d'implanter une petite « calculatrice programmable. » Cela donnera déjà un certain nombre d'idées sur les circuits requis pour implanter l'architecture d'un processeur.

Le but de ce premier TP est de se familiariser avec Logisim, et de réaliser certains circuits combinatoires que l'on utilisera par la suite.

1.1. Avant de commencer

Créez un répertoire pour les TP de LIFASR3, puis, dans votre navigateur web, créez un favori pour la page de Logisim (Google « Logisim ») : `http://www.cburch.com/logisim/`.

Télécharger l'archive `logisim-generic-2.7.1.jar` de Logisim dans votre répertoire de TP : vous lancerez le logiciel en entrant la commande `java -jar logisim-generic-2.7.1.jar` dans un terminal.

1.2. Premiers exercices

Une des particularités de Logisim est de pouvoir éditer et simuler son circuit en même temps.

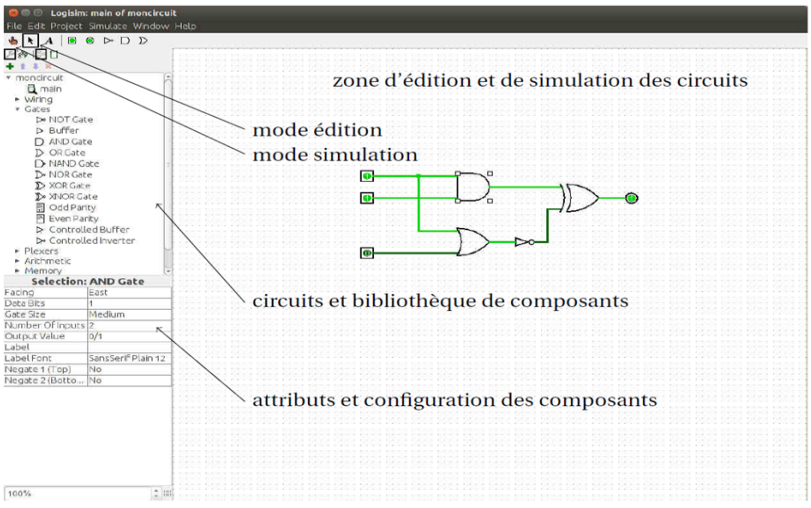


FIGURE 1.1 – Fenêtre de Logisim.

1.2.1 Mode édition

Pour utiliser le mode édition, il faut sélectionner la flèche comme indiqué en haut de la figure 1.1.

1. On peut alors choisir un composant dans la bibliothèque, sur la gauche. Pour l'ajouter dans son schéma, il suffit de cliquer sur le composant désiré, puis de le déposer dans le schéma.

2. Chaque composant que vous utiliserez aura des attributs modifiables dans la zone inférieur gauche de Logisim. Par exemple si l'on pose une porte AND, on peut modifier le nombre de signaux qu'elle prend en entrée (attribut `Number Of Inputs`), ou son orientation (attribut `Facing`).
3. Il est possible de faire des copier/coller d'un ou plusieurs composants (sélectionner, puis `Ctrl+C/V` ou bien `Ctrl+D`). Dans ce cas, les composants conserverons aussi tous les attributs préalablement définis.

Des éléments dont vous aurez besoin rapidement :
- Pour les entrées, l'élément `Pin` de `Wiring` dans la bibliothèque, avec l'attribut `output?=no` (voir aussi le « petit carré » dans la barre d'outils).
- Pour les sorties, l'élément `Pin` avec `output?=yes` ; (voir aussi le « petit rond » dans la barre d'outils).
- Les portes logiques sont présentes dans le répertoire `Gates` de la bibliothèque.

Notes pour plus tard :
- pour tous les composants qui ont cet attribut, vérifiez que **ThreeState=No**,
- méfiez vous « des fils qui se touchent » en particuliers entres les entrées rapprochées des portes logiques !

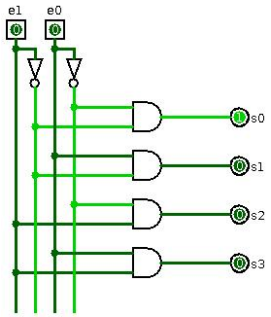
1.2.2 Mode simulation

Logisim est capable de simuler le circuit en affichant les valeurs des signaux directement sur le schéma : l'utilisateur peut définir les valeurs des bits en entrée et observer la réaction du circuit.

1. Pour utiliser le mode simulation, il faut sélectionner la main en haut à gauche de Logisim.
2. Il est possible de contrôler l'état des différentes entrées en cliquant directement dessus avec la « petite main » (forcément, uniquement en mode simulation).
3. En cliquant sur une entrée, la valeur doit alterner entre 1 en vert clair, et 0 en vert foncé.
4. Les sorties prennent les valeurs calculées par le circuit en fonction des entrées. Pour les fils qui ne transportent qu'un bit, la convention des couleurs est identique : 0 en vert clair, et 1 en vert foncé.

Exercice 1.2.1 : Un premier circuit pour apprendre à se servir de l'interface.

1) Ouvrez Logisim, et reproduisez le circuit ci-dessous aussi fidèlement que possible. Indications :
- Pour orienter les entrées (`Pin` avec `output?=no`), utilisez leur attribut `Facing`. Pour nommer les entrées, utilisez l'attribut `Label`, et pour placer le nom au bon endroit, l'attribut `Label Location`.
- Les mêmes remarquent valent pour les sorties (`Pin` avec `output?=yes`).
- Les portent AND doivent présenter deux entrées, donc utilisez `Number Of Inputs=2`.
- Faites des copiers-collers, quitte à modifier les attributs ensuite !
- Pour effacer un composant ou un fil : placez vous dessus, puis faites un click-droit et `Delete`.



2) Enregistrez votre travail dans le fichier `exo11.circ` (`File→Save`).

3) En vous mettant en mode simulation, complétez la table de vérité suivante :

e_1	e_0	s_3	s_2	s_1	s_0
0	0				
0	1				
1	0				
1	1				

4) Comment s'appelle le circuit que l'on vient de réaliser ? .....

.....

Exercice 1.2.2 : Un deuxième petit circuit

Commencez par créer un nouveau projet avec `File→New`. Vous enregistrerez votre travail dans le fichier `exo12.circ`. Dans cet exercice, on expérimente autour de multiplexeurs 4 bits vers 1 bit.

1) Notre multiplexeur doit comporter :
— 4 bits e_0, e_1, e_2, e_3 en entrée,
— 1 bit s en sortie,
— les bits de sélection c_1 et c_0 également en entrée.
L'entrée dont l'indice est donné par $(c_1 c_0)_2$ doit être envoyée sur la sortie s. Indiquez quelle valeur doit prendre s en fonction des entrées du circuit dans le tableau suivant.

c_1	c_0	s
0	0	
0	1	
1	0	
1	1	

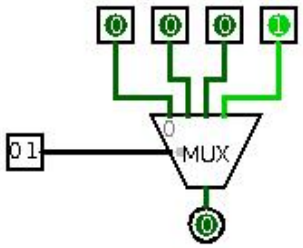
2) Donnez une formule pour s, en n'utilisant que les fonctions `AND`, `OR` et `NOT`. .....

.....

3) Implantez [1] votre multiplexeur dans Logisim, en n'utilisant que les fonctions `AND`, `OR` et `NOT`. Comment faire pour tester son bon fonctionnement ? .....

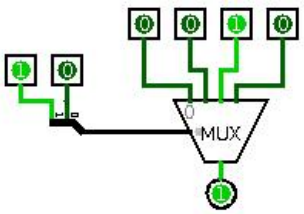
.....

4) Maintenant, trichons : il y a déjà un composant multiplexeur dans la bibliothèque de Logisim ! Instanciez [2] un composant **Multiplexer** du répertoire `Plexers`, en mettant bien dans les attributs **Include Enable ?=No** et **Disabled Output=Zero**. Implantez ensuite le circuit suivant, et testez le :



Indications :
— Pour le multiplexeur, il faut mettre **Include Enable ?=No** et **Disabled Output=Zero**, puis choisir judicieusement les attributs `Data Bits` et `Select Bits`.
— Notez que **l'entrée située sur la gauche dans le circuit regroupe 2 bits** : il s'agit d'une entrée `Pin` avec `Three-state?=No` (toujours vérifier !), `Output?=No` (logique) et **DataBits=2**.

5) On modifie le circuit précédent pour faire apparaître individuellement les deux entrées de sélection. Pour cela, on utilise un composant qui permet de séparer le paquet de deux fils de sélection du multiplexeur en deux fils : ce composant est le **Splitter** du répertoire `Wiring`. Implantez le circuit suivant, et testez le :



Un `Splitter` présente deux attributs entiers importants : **Bit Width In** et **Fan Out**. Un `Splitter` prend d'un côté un bus de (`Bit Width In`) bits, et les répartit en (`Fan Out`) petits paquets.

1. Le verbe est ici de la même façon que lorsqu'on dit « je vais implanter mon algorithme de tri en C. »
2. Verbe compliqué mais d'usage courant qui veut dire ici « placez dans votre circuit ».

1.3. Un demi-additionneur

1) Commencez par créer un circuit `DA` : pour cela vous pouvez faire `Project→Add circuit`, puis donner son nom au circuit. Assurez vous que vous éditez bien le circuit `DA`, en double-cliquant sur son nom : une loupe doit apparaître dessus. Réalisez le circuit représenté à gauche sur la Figure 1.2.

2) On crée une instance de `DA` dans le circuit principal : éditer le circuit `main` en double-cliquant sur son nom. Ensuite, déposez une instance de `DA` comme vous l'avez déjà fait pour des composants de la bibliothèque, et complétez le circuit comme indiqué à droite sur la Figure 1.2. Notez que l'on retrouve les entrées (a, b), et les sorties (r_s, s), « disposition » que lors de la création du circuit : vous pouvez faire s'afficher leurs noms en passant le pointeur dessus.

3) Complétez la table de vérité ci-dessous. Pourquoi appelle-t-on ce circuit un demi-additionneur ?

a	b	r_s	s
0	0		
0	1		
1	0		
1	1		

.....

.....

.....

.....

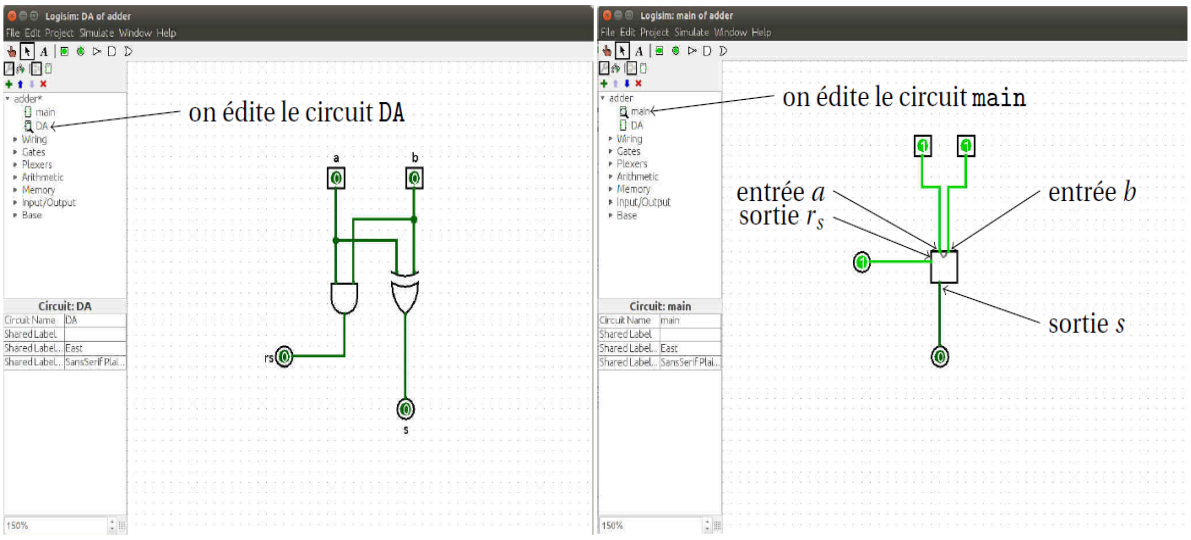


FIGURE 1.2 – Création (à gauche) et instanciation (à droite) du circuit DA.

1.4. Additionneur 1 bit complet

Un additionneur 1 bit complet prend en entrée trois bits a, b et r_e, et produit deux bits de sortie r_s et s tels que $2 \times r_s + s = a + b + r_e$ (au sens des entiers naturels).

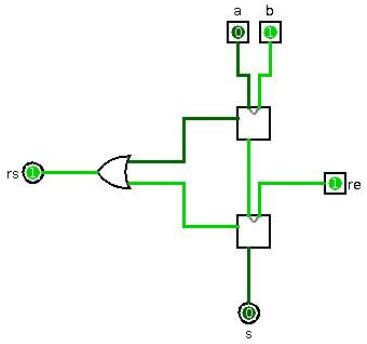
1) En faisant la table de vérité d'un additionneur 1 bit complet, trouver comment réaliser ce circuit à l'aide de deux demi-additionneur et d'une porte `OR`.

2) Implantez un circuit `AC` réalisant un additionneur 1 bit complet.

3) Veillez à bien tester votre circuit en vous basant sur sa table de vérité.

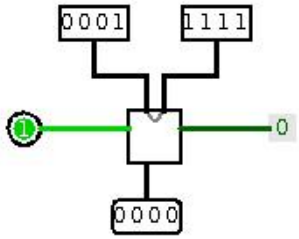
1.5. Additionneur 4 bits

Vous devez réaliser maintenant un additionneur 4 bits, dans un circuit `add4`, qui devra :
— prendre en deux entiers binaires a et b, chacun sur 4 bits, ainsi qu'une retenue entrante r_e sur 1 bit,
— produire en sortie l'entier binaire s sur 4 bits, et une retenue sortante r_s sur 1 bit.
Les entrées et sorties doivent vérifier $2^4 \times r_s + s = a + b + r_e$. Vous utiliserez :
— un splitter pour obtenir les 4 bits qui composent l'entrée a, un autre pour décomposer b,



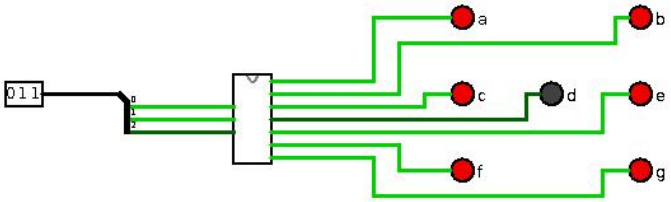
— un splitter regrouper les 4 bits qui composent la sortie s,
— 4 instances de votre circuit `AC`.

Testez ensuite votre circuit dans le `main`, en réalisant un circuit ressemblant à celui ci-dessous. Pour la retenue entrante, on utilise une constante égale à 0 (`Wiring`, `Constant`). Avec votre circuit, calculez $(0101)_2 + (0011)_2$, $(1111)_2 + (0001)_2$, $(1010)_2 + (1011)_2$. Dans chacun des cas, vérifiez à la main que vous obtenez bien le bon résultat. Si vous deviez tester tous les cas possibles, combien y en aurait-il ?



1.6. Le Dé électronique

On souhaite implanter dans Logisim l'afficheur « dé électronique » vu en TD. Le circuit principal aura l'allure suivante :



Entre l'entrée sur 3 bits placée sur la gauche, et les 7 LED permettant l'affichage, un transcodeur permet d'effectuer la conversion, de façon à ce que les bonnes LED soient allumées en fonction de l'entier indiqué. Pour fabriquer le transcodeur, nous allons utiliser un outil de Logisim permettant de créer automatiquement des circuits combinatoires d'après leur table de vérité.

1) Allez dans `Windows→Combinational Analysis` :
— dans l'onglet `Input`, indiquez les variables d'entrée du transcodeur (`x`, `x`, `z`) ;
— dans l'onglet `Output`, indiquez les variables de sortie (`a`, ..., `g`) ;
— dans `Table`, vous pouvez entrer la table de vérité de chaque sortie ;
— dans `Expression`, vous obtenez une expression booléenne pour chaque de sortie, et dans `Minimized` une expression simplifiée.
Une fois que vous avez entré toutes les tables de vérité, vous pouvez utiliser le bouton `Build circuit`, pour construire un circuit auquel vous donnerez le nom de `transcoder`. Pour tous les détails, voir dans l'aide de Logisim : `Guide to being a Logisim User → Combinational analysis`.

2) Utilisez le circuit `transcoder`, ainsi que des LED (`Input/Output → LED`) pour construire le circuit demandé. Testez le soigneusement !

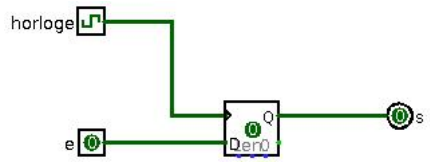
3) Vous pouvez ensuite expérimenter avec le composant `Memory→Random Genrator` pour simuler un lancer de dé !

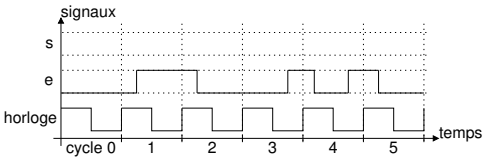
1.7. Bascules et registres

Exercice 1.7.1 : Bascules

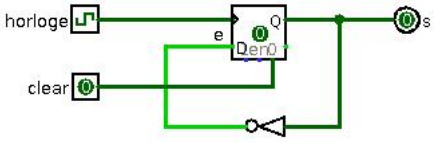
Une bascule (composant `D Flip-flop` dans le répertoire `Memory` de la bibliothèque) est une mémoire 1 bit. On ne travaille qu'avec des bascules régies par le front montant de l'horloge : attribut `Trigger=Rising Edge`. Une bascule reçoit un signal d'horloge, et présente une entrée et une sortie : *l'entrée est mémorisée à la fin d'un cycle sur le front montant de l'horloge, et le bit mémorisé est maintenu sur la sortie pendant tout le cycle suivant.*

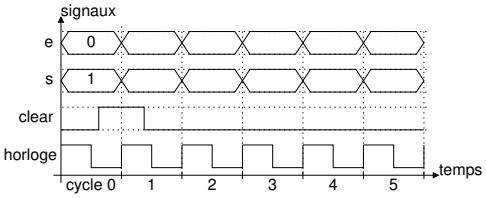
1) Commencez par brancher une bascule de Logisim comme indiquée ci-dessous. Servez-vous de ce circuit pour compléter le chronogramme fourni.





2) Expérimentez maintenant avec le circuit suivant. Notez que l'entrée `clear` de la bascule permet de la remettre à zéro à tout instan. Complétez le chronogramme.



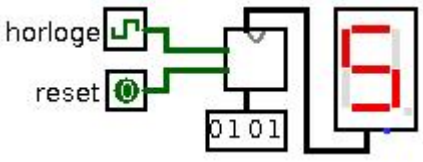


Exercice 1.7.2 : Registres

Un registre permet de mémoriser un mot binaire : par exemple, un registre 4 bits mémorise un mot de 4 bits. Les registres sont fabriqués à l'aide de bascules. Ainsi, *le mot placé en entrée du registre est mémorisée à la fin d'un cycle, et le mot mémorisé est maintenu sur la sortie pendant tout le cycle suivant.*

1) A l'aide de 4 bascules, implantez un circuit `reg4`, réalisant un registre 4 bits. Votre circuit présentera en entrée : un mot de 4 bits à mémoriser, le signal d'horloge, un signal `reset` permettant de remettre le contenu du registre à zéro. En sortie, votre registre présentera simplement le mot mémorisé sur 4 bits.

2) Dans le circuit principal, instanciez `reg4`, et testez-le dans le circuit représenté ci-dessous (le composant utilisé pour afficher le contenu du registre est un `Hex Digit Display` dans le répertoire `Input/Output`).



Exercice 1.7.3 : Compteurs

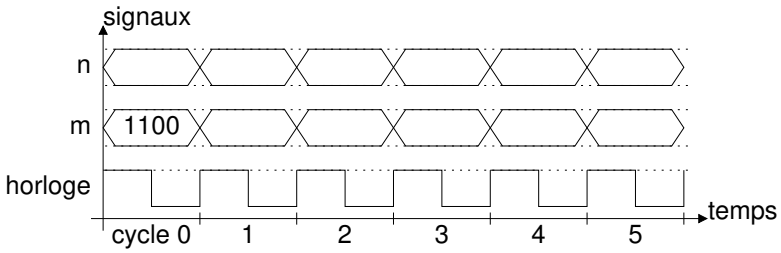
Un compteur est un circuit séquentiel qui reçoit un signal d'horloge, et un signal `reset` permettant de le remettre à zéro ; il présente une seule sortie sur k bits, formant un entier naturel n, qui est la valeur du compteur :

— le passage de `reset` à 1 remet la valeur n du compteur à 0,
— quand `reset` = 0, la valeur du compteur est incrémentée à chaque fin de cycle d'horloge [3].

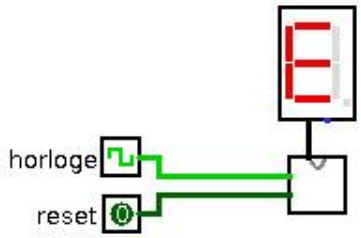
1) A l'aide de votre registre `reg4`, et d'un additionneur de la bibliothèque de Logisim, implantez le circuit d'un compteur 4 bits appelé `cmpt4`.

2) Dans le circuit `cmpt4`, on appelle m la valeur de l'entier en entrée de `reg4`, et n la valeur en sortie. Complétez le chronogramme suivant.

3. Il y a pleins d'autres subtilités, notamment ce qui arrive au compteur lorsqu'il atteint sa valeur maximale ; pour l'instant, on suppose que le compteur fonctionne modulo 2^k : si sur un cycle il est à $2^k - 1$, au cycle suivant le compteur retombe à 0.



3) Testez votre circuit `cmpt4`, en l'utilisant dans un circuit ressemblant à celui ci-dessous.



1.8. Banc de registres et mémoire

Un processeur doit lire et écrire des registres, et il est pratique de les regrouper dans un composant appelé *banc de registres* : ces registres recevront les résultats intermédiaires produits lors de exécution d'un programme.

Les registres sont rapides, mais assez coûteux[4] : les processeurs ne contiennent qu'un nombre réduit de registres. Par contre, ils peuvent utiliser une mémoire RAM (*Random Access Memory*) beaucoup plus grande et moins coûteuse, pour lire des programmes et des données, et écrire des résultats. Dans la suite, on s'intéresse par commodité à un autre type de mémoire, la mémoire ROM (*Read Only Memory*) qui ne peut qu'être lue.

1.9. Une unité arithmétique et logique

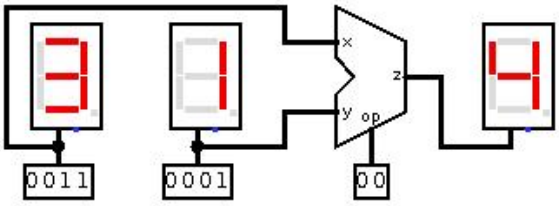
On veut mettre au point une unité arithmétique et logique (ALU) pour notre calculatrice : il s'agit d'un circuit qui prend en entrée deux entiers `x` et `y` codés sur n bits, et produit en sortie un résultat `z`, également sur n bits. Un port d'entrée supplémentaire `op` sur k bits permet de sélectionner l'opération dont on veut obtenir le résultat : k bits permettent de choisir parmi 2^k opérations.

Exercice 1.9.1 : ALU 4 bits

Dans cet exercice, les opérandes en entrée de l'ALU sont sur 4 bits ($n = 4$), et on souhaite qu'elle puisse effectuer les 4 opérations usuelles sur les entiers naturels : addition, soustraction, multiplication et division. L'opération à effectuer est déterminée par le codage suivant :

op	z
00	x + y
01	x − y
10	x × y
11	x ÷ y

1) Implantez un circuit `alu4`, réalisant l'ALU demandée. Pour les opérations, utilisez les composants proposés dans le répertoire `Arithmetic` de Logsim. En outre, vous utiliserez un composant `Multiplexer` judicieusement configuré pour sélectionner le bon résultat en sortie de l'ALU



2) Testez votre composant `alu4` dans un circuit ayant l'allure de celui ci-dessous[5].

4. On parle ici du coût monétaire par bit d'information
5. Votre ALU aura la forme d'un rectangle, ce qui est très bien ! Dans le circuit représenté ici, l'auteur s'est « amusé » à donner une forme

Le but de ce TP est de faire un peu d'architecture des ordinateurs : on se donne un langage minimaliste (le jeu d'instructions), et on tâche d'implanter un circuit permettant d'exécuter les programmes écrits dans ce langage. Pour faire au plus simple, on se contente de réaliser une petite calculatrice programmable.

1.10. Une calculatrice programmable

On va utiliser les composants suivants :
— une mémoire ROM qui va contenir les instructions du programme à exécuter,
— une ALU pour effectuer les opérations arithmétiques (vu au TP précédent),
— un banc de 8 registres 8 bits (vu au TP précédent).
La calculatrice comporte donc 8 registres pour stocker les résultats intermédiaires d'un calcul, nommés `R0`,...,`R7`.

Exercice 1.10.1 : Codage des programmes

Le jeu d'instructions comporte les instructions données dans le tableau suivant. Dans ce tableau :
— `DR` est le nom d'un registre de destination (l'instruction écrit dedans),
— `SR1` et `SR2` sont des registres sources (l'instruction lit dedans),
— `Imm6` est un entier naturel codé sur 6 bits.
Pour stocker un programme dans la mémoire ROM, on choisit de coder chaque instruction sur 12 bits comme précisé dans le tableau (les noms de registres sont codés sur 3 bits ; par exemple, `R5` est codé par 101).

instruction	**action**	**codage en binaire**											
		code			opérandes								
		11	10	9	8	7	6	5	4	3	2	1	0
`SET DR, Imm6`	`DR <- Imm6`	0	0	0	DR			Imm6					
`ADD DR, SR1, SR2`	`DR <- SR1 + SR2`	1	0	0	DR			SR1			SR2		
`SUB DR, SR1, SR2`	`DR <- SR1 - SR2`	1	0	1	DR			SR1			SR2		
`MUL DR, SR1, SR2`	`DR <- SR1 × SR2`	1	1	0	DR			SR1			SR2		
`DIV DR, SR1, SR2`	`DR <- SR1 ÷ SR2`	1	1	1	DR			SR1			SR2		

Voici par exemple un programme qui évalue 10×2, en laissant le résultat dans le registre `R3` :

```
SET R1, 10          // effectue R1 <- 10
SET R2, 2           // effectue R2 <- 2
MUL R3, R1, R2      // effectue R3 <- R1 * R2
```

Les instructions du programme vont être codées de la façon suivante :

instruction	**en binaire**												**en hexadécimal**
	11	10	9	8	7	6	5	4	3	2	1	0	
`SET R1, 10`	0	0	0	0	0	1	0	0	1	0	1	0	04A
`SET R2, 2`	0	0	0	0	1	0	0	0	0	0	1	0	082
`MUL R3, R1, R2`	1	1	0	0	1	1	0	0	1	0	1	0	CCA

Dans le mémoire ROM, on doit donc avoir (tout en hexadécimal) :

adresse	00	01	02	...
contenu	04A	082	CCA	...

1) Poursuivez le programme précédent, de façon à ce qu'il évalue $(10 \times 2) \times (9 + 6)$.

2) Donnez le codage (en binaire, puis en hexadécimal) de votre programme.

3) Comment se distingue l'instruction `SET` des instructions arithmétiques (`ADD`, `SUB`, ...) ?

4) Pour une instructions arithmétique, quels bits permettent de déterminer l'opération à effectuer ?

Exercice 1.10.2 : Réalisation de la calculatrice

Téléchargez le fichier `calcetu.circ` ; il va vous permettre de réaliser la calculatrice. Évidemment, un chargé de TP malveillant a tout saboté, donc vous allez devoir refaire pleins de choses...

Le circuit (voir Figure 1.3) comporte une `horloge` pour synchroniser les circuits séquentiels, et un bouton `reset` pour remettre à zéro les registres du banc (circuit `regfile`) et l'adresse `PC` produite par un compteur.

L'idée est que les instructions qui se trouvent dans la ROM soient exécutées les unes à la suite des autres. A chaque cycle d'horloge, `PC` donne l'adresse de l'instruction qui doit être exécutée, et `PC` est incrémenté en fin de cycle de façon à passer à l'exécution de l'instruction suivante, au cycle suivant. Ainsi, on veut qu'à chaque cycle,
— l'instruction en sortie de la mémoire ROM soit exécutée,
— le résultat de cette instruction soit stocké dans le banc de registres en fin de cycle.

un peu standard à son ALU : ne perdez pas de temps avec ça !

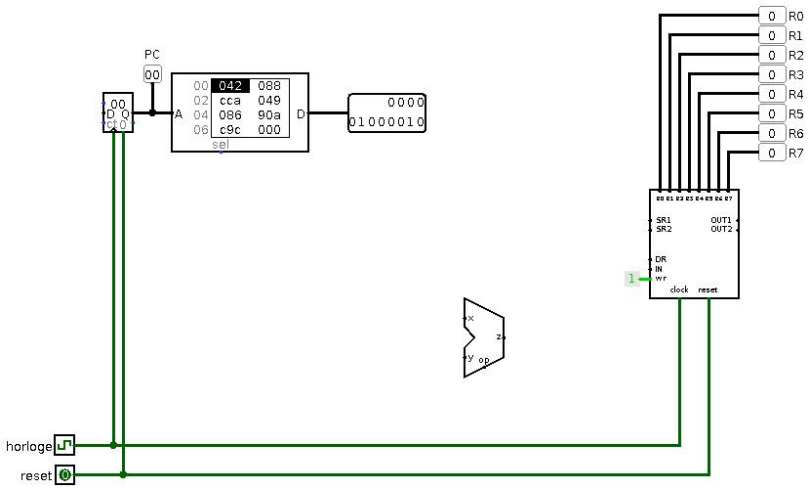


FIGURE 1.3 – Le circuit initial.

Notez que l'on ne se préoccuper des d'entrées-sorties ici : suivre les calculs, il faut observer les composants `Probes` branchés sur le banc de registres.

1) Commencez par vous assurez que la ROM contient bien, aux adresses $(01)_{16}$ à $(02)_{16}$, les trois premières instructions du programme de l'exercice précédent.

2) Supposez que l'instruction à exécuter soit une instruction `SET DR, Imm6`. A l'aide de `splitters`, connectez `DR` sur l'entrée de même nom du banc de registres, et `Imm6` sur l'entrée `IN` du banc (attention, il faut passer de 6 à 8 bits). Comme les deux premières instructions du programme sont des instructions `SET`, vous pouvez facilement tester votre circuit.

3) Supposez maintenant que l'instruction en cours d'exécution soit une instruction arithmétique de la forme `ARITH DR, SR1, SR2`. Avec des `splitters`, extrayez du codage de l'instruction : les deux bits `op` qui indiquent quelle est l'opération arithmétique à effectuer, et les opérandes `SR1` et `SR2`.

4) Connectez les champs `SR1` et `SR2` venant de l'instruction aux entrées de même nom du banc de registres : vous obtenez ainsi les contenus des registres correspondants sur les sorties `OUT1` et `OUT2` du banc, que vous pouvez brancher sur l'`ALU`.

5) Ensuite, connectez les deux bits `op` de façon à ce que l'`ALU` effectue la bonne opération arithmétique. En profitant du fait que la troisième opération du programme est un `MUL`, testez votre travail.

6) Dans le cas d'une instruction arithmétique, comment le numéro de registre de destination `DR` doit-il être connecté au banc de registre ?

7) A ce stade, il ne vous reste normalement plus qu'à brancher judicieusement l'entrée `IN` du banc de registre, de façon à ce qu'elle reçoive :

— dans le cas d'une instruction `SET`, la valeur donnée par `Imm6` étendue de 6 à 8 bits,
— dans le cas d'une instruction arithmétique, la valeur obtenue en sortie de l'`ALU`.

8) Entrez dans la ROM le programme de l'exercice précédent, et utilisez-le pour tester votre calculatrice.