



Architecture Matérielle et Logicielle (LIF6)

Cahier de TP, printemps 2016

Pistes pour les solutions...

Table des matières

1	Prise en main de LOGISIM, premiers circuits	3
1.1	Avant de démarrer	3
1.2	Circuits combinatoires de base	3
1.3	Exercice supplémentaire	7
2	Circuits combinatoires et complément à deux	9
2.1	Circuits combinatoires	9
2.2	Quelques éléments du processeur LC-3	10
2.3	Dépassements en C	12
3	Circuits séquentiels, registres, mémoire	14
3.1	Bascules	14
3.2	Registres, compteurs et mémoire	14
3.3	Banc de registre	16
4	Test de circuit, circuits dédiés.	18
4.1	LOGISIM : test de circuit	18
4.2	Circuits dédiés	19
5	Introduction à l'archi LC-3, jouons avec le simulateur	24
5.1	Jouons avec le simulateur de LC-3	24
5.2	Écriture et simulation de programmes en assembleur LC-3	26
6	LC-3, Exercices de programmation	28
6.1	Chaînes de caractères	28
6.2	Un message codé (CC-TP 2015)	29
6.3	Saisie d'un entier au clavier	30
A	Documentation LOGISIM	33
A.1	Référence	33
A.2	Bibliothèque de composants	33
B	Documentation LC-3	35
B.1	Référence Bibliographique	35
B.2	Codage des instructions	35
B.3	ISA LC-3	35

TP 1

Prise en main de LOGISIM, premiers circuits

Objectifs :

- Prise en main de LOGISIM
- Écriture de circuits combinatoires simples
- Découverte des fonctionnalités de LOGISIM.

1.1 Avant de démarrer

EXERCICE 1 ► Démarrage de cycle!

On part du principe ici que les TPs sont réalisés sous Linux, même s'il est possible de les faire sous Windows (il faut dans ce cas savoir se débrouiller avec la ligne de commande de Windows, et il suffira d'adapter les commandes utilisées sous Linux). Créez un répertoire pour les TP de LIF6, puis, dans votre navigateur, créez les favoris suivants :

- La page du cours.
- La page de l'outil LOGISIM <http://www.cburch.com/logisim/>.

Télécharger l'archive .jar de LOGISIM dans votre répertoire de TP : vous lancerez le logiciel en entrant la commande `java -jar archive.jar` dans un terminal (remplacer `archive` par le nom effectif de l'archive...).

EXERCICE 2 ► Tutoriel

Créez un répertoire pour le TP1, puis réalisez le tutoriel "Beginner's tutorial" disponible sur la page de LOGISIM¹. On n'oubliera pas *dès la création de la première porte logique* de sauvegarder son fichier, et de taper `Ctrl+s` régulièrement. Vous ne passerez pas trop de temps sur ce Tutoriel (une demi-heure maximum).

1.2 Circuits combinatoires de base

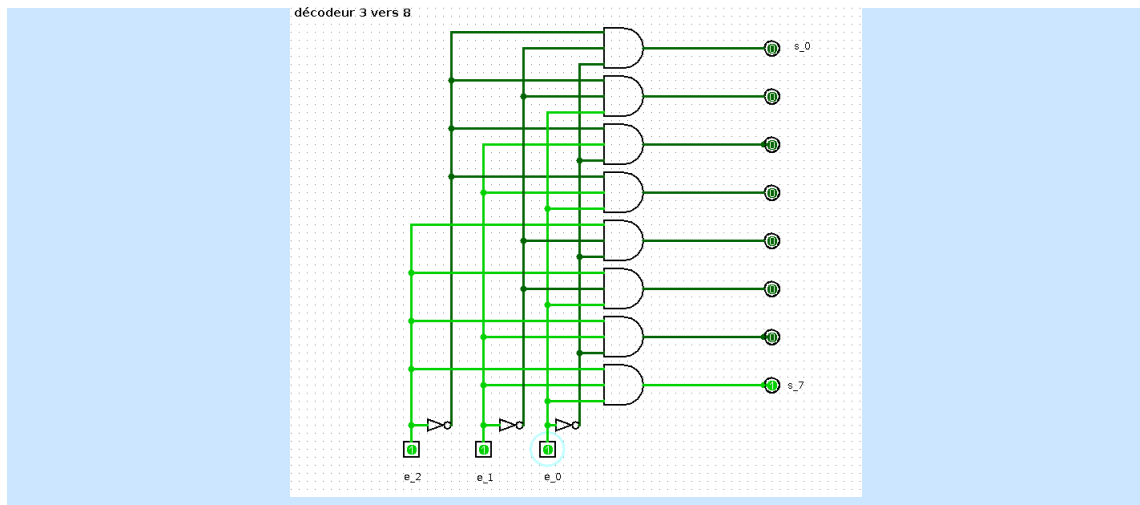
EXERCICE 3 ► Multiplexeurs/décodeurs

Dans un nouveau fichier :

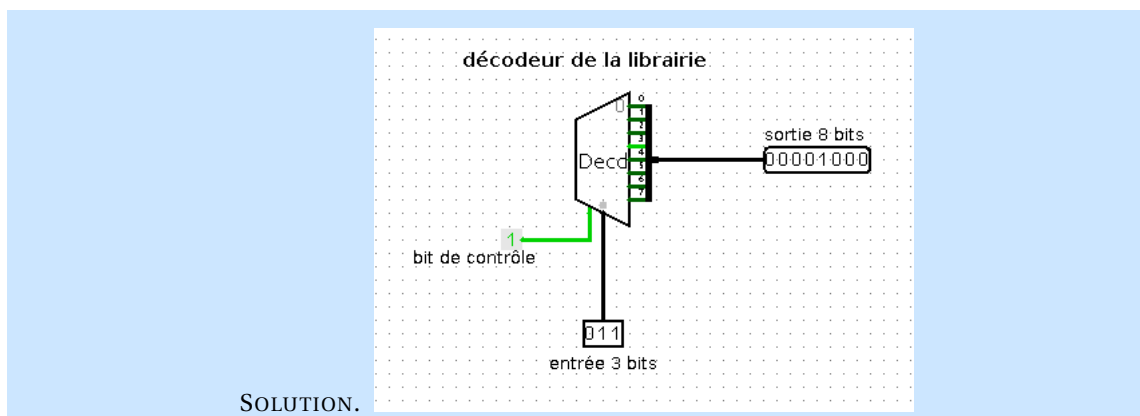
- Réalisez un décodeur 3 bits vers 8 bits. Testez. *On utilisera des portes (Gates) And à 3 entrées.*

SOLUTION. C'est du cours.

1. Oui, il est en anglais, so what?

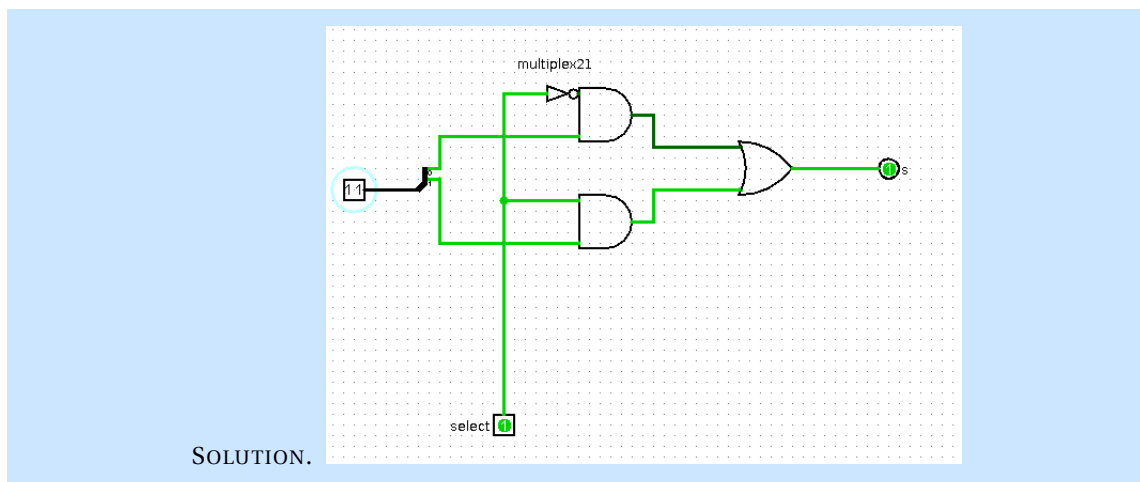


- Comparez le comportement avec un décodeur de la librairie. Testez avec une source (Pin carré) 3 bits, un afficheur (Pin rond 8 bits, et un Splitter (réglages : Fan Out:8, BitWidthIn:8) pour relier la sortie du décodeur à l'affichage 8 bits.

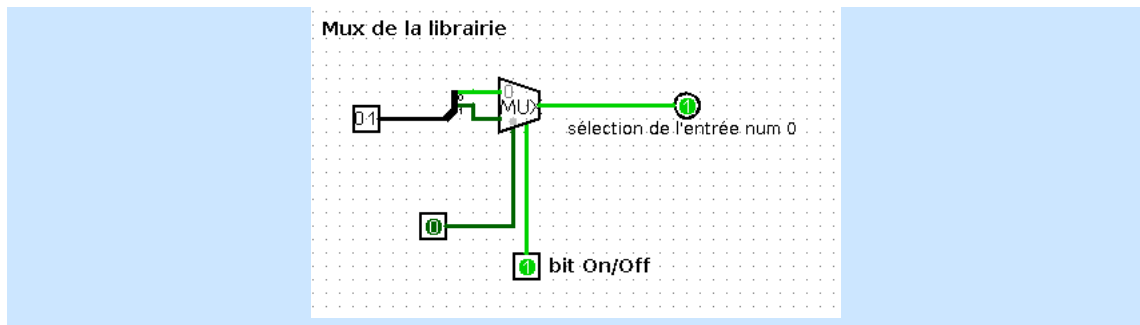


SOLUTION.

- Réalisez un multiplexeur 2 bits vers 1 bit. Comparez avec un multiplexeur de la librairie.



SOLUTION.

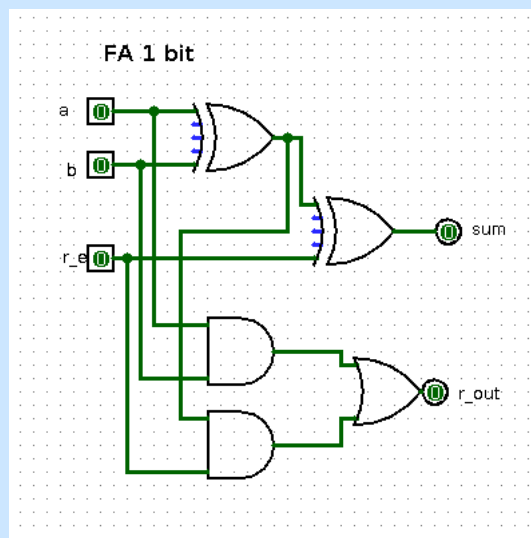


EXERCICE 4 ► Additionneurs à retenue

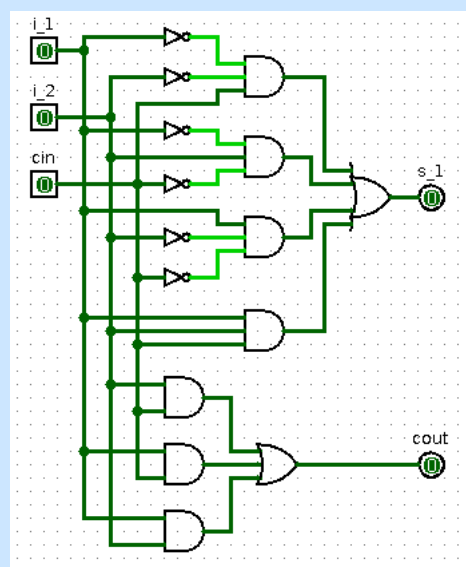
Dans un nouveau fichier :

- Réalisez l'additionneur 1 bit à retenue du cours et testez-le.

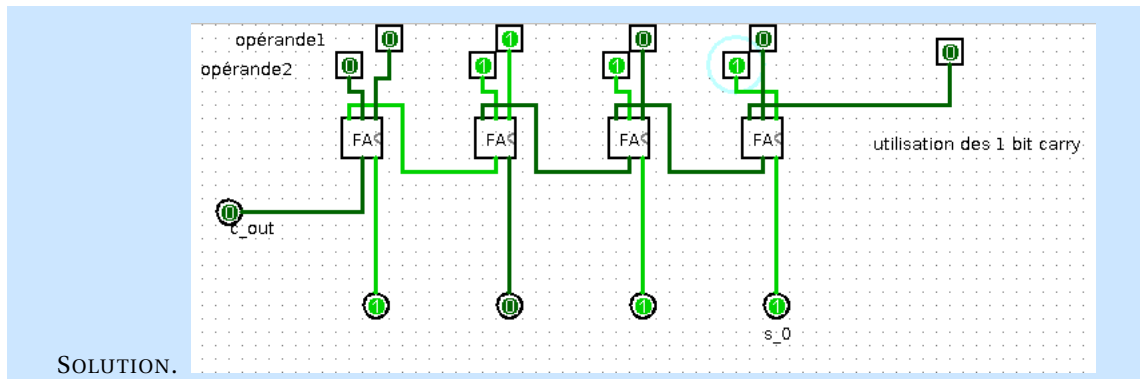
SOLUTION. Dans le cours l'additionneur utilise des xor :



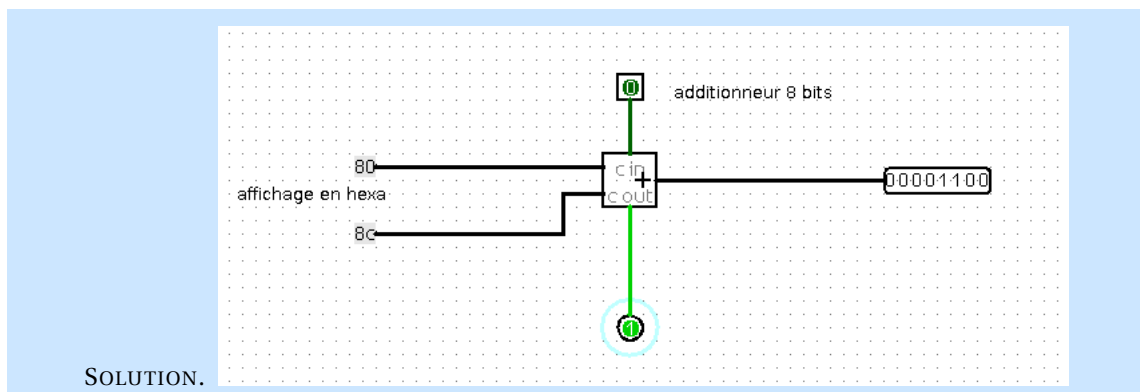
Si on utilise la fonctionnalité de LOGISIM pour créer les circuits à l'aide de la table de vérité (non demandé aux étudiants), on obtient :



- Regardez dans la documentation comment fonctionne l'encapsulation (Subcircuits). Nommez l'additionneur 1 bit "FA1" et utilisez le pour réaliser un additionneur 4 bits.



- Utilisez l'additionneur 8 bits de la librairie (Arithmetic->Adder) avec des “constantes” (Wiring->Constant) en entrée de l'addition et un afficheur (Probe) 8 bits en sortie. On vérifiera que $(80)_{16} + (8C)_{16} = (00001100)_2$ (et 1 de retenue).

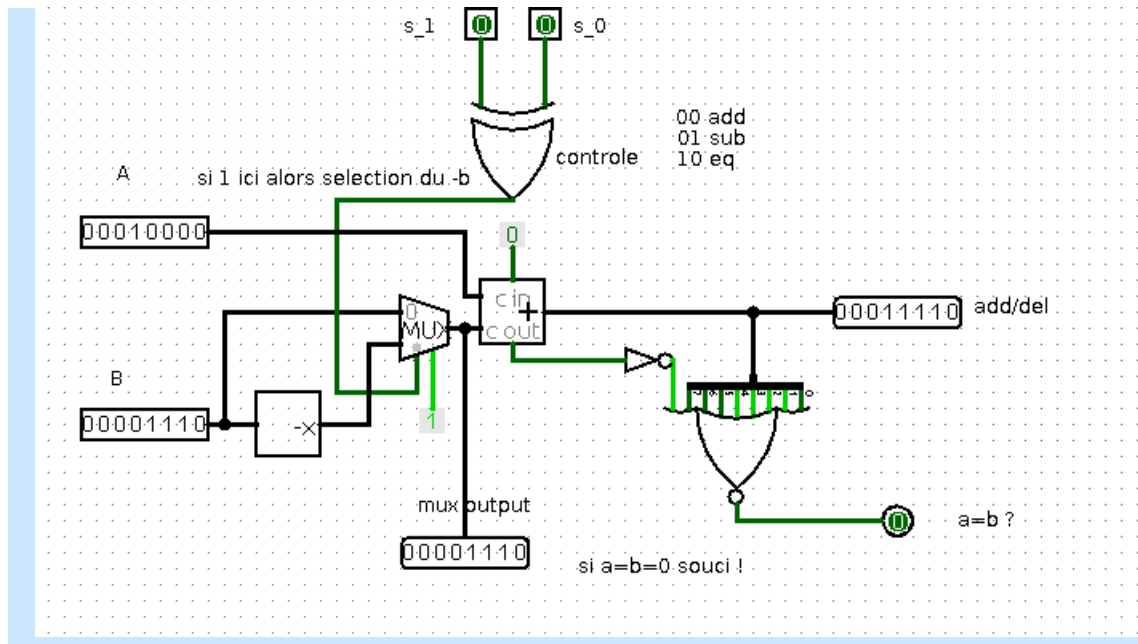


- En utilisant cet additionneur 8 bits (et les multiplexeurs de la librairie), réalisez un additionneur-soustracteur 8bits, qui calcule $a - b$ ou $a + b$ suivant la valeur d'un bit de contrôle c . Nous n'avons pas le droit de dupliquer l'additionneur. Vous pouvez vous reporter à l'exercice correspondant du cahier de TD.

SOLUTION. cf plus bas (ALU)

- (Si vous avez du temps) Réalisez une **ALU 8 bits** capable de faire une addition, une soustraction et un test d'égalité. L'opération sera choisie avec un signal qui vaut 00 pour une addition, 01 pour une soustraction et 10 pour un test d'égalité. On remarquera que c'est une modification mineure du circuit précédent.

SOLUTION.



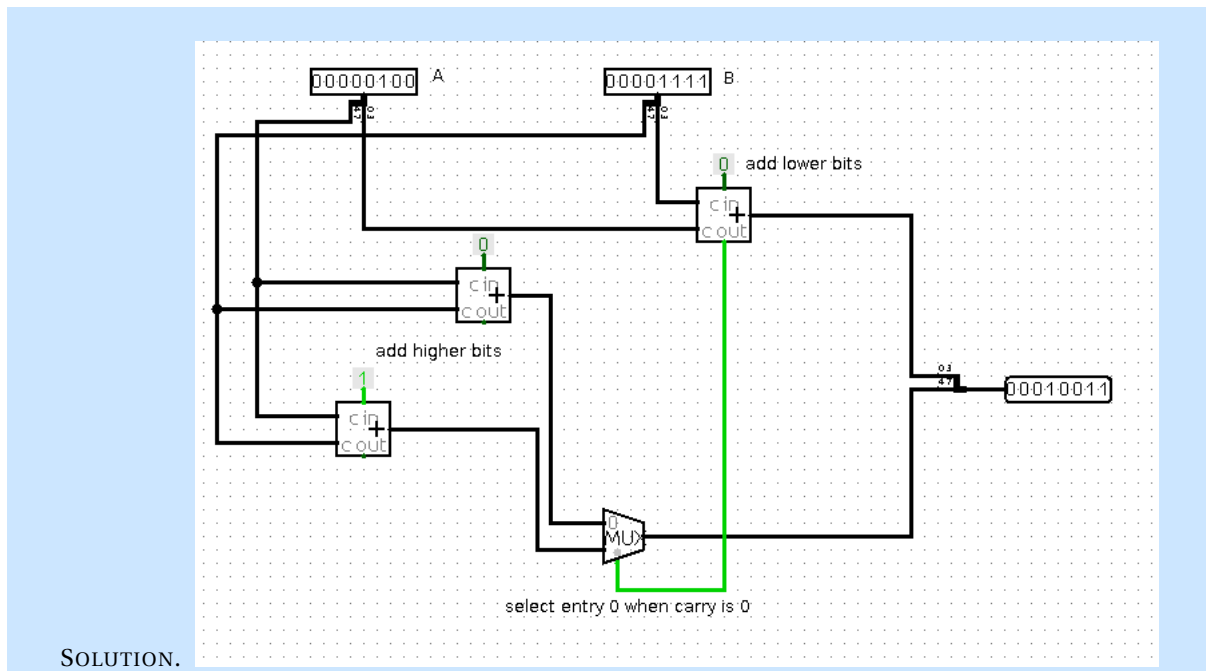
1.3 Exercice supplémentaire

D'après un TP de C. Alias, Inria & ÉNS Lyon.

EXERCICE 5 ► Retenue anticipée

L'inconvénient des additionneurs 8 bits en cascade est que chaque additionneur 1 bit doit attendre que sa retenue entrante soit disponible pour réaliser l'opération. Un additionneur 8 bits a donc un temps de traversée égal à 8 fois le temps de traversée d'un additionneur 1 bit. Un additionneur à retenue anticipée (*carry select*) peut être construit en utilisant le temps de traversée d'un additionneur 4 bits (utilisé pour additionner les 2×4 bits de poids faible) pour précalculer les deux résultats possibles de l'addition des 2×4 bits de poids forts (l'un avec une retenue entrante égale à 1, l'autre avec une retenue entrante nulle). Un multiplexeur est utilisé pour sélectionner le bon résultat lorsque la retenue entrante est finalement connue.

- Réalisez un tel additionneur en utilisant des additionneurs 4 bits de la librairie.
- Faites une démonstration à votre enseignant de TP.



TP 2

Circuits combinatoires et complément à deux

Objectifs :

- Implantation de circuits combinatoires.
- Pratiquer le complément à 2.

Fichiers fournis : tp2_aluetu.circ, tp2_comp2etu.c

2.1 Circuits combinatoires

EXERCICE 1 ► Circuits à construire

En commençant par écrire la table de vérité des fonctions booléennes désirées, construire les circuits suivants¹ :

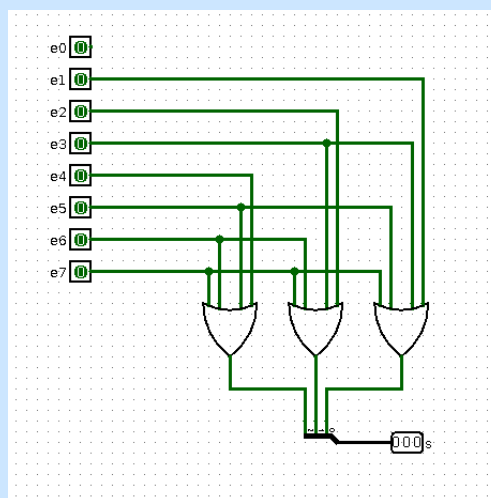
- **Encodeur octal** : c'est un circuit à 8 entrées e_7, \dots, e_0 et à trois sorties s_2, s_1, s_0 . Si e_i est à 1, on veut que $(s_1 s_0)_2 = i$. On suppose qu'un seul des e_i est à 1.
- **Parité impaire** sur 3 bits : c'est un circuit à 3 entrées et une sortie qui vaut 1 si et seulement si le nombre des entrées à 1 est impair.

Si il vous reste du temps à la fin de ce TP, vous pourrez également faire le dernier exercice du TP Précédent.

SOLUTION. Pour l'encodeur octal, c'est un circuit à 8 entrées e_7, \dots, e_0 et à trois sorties s_2, s_1, s_0 . Si e_i est à 1, on veut que $(s_1 s_0)_2 = i$ (cf le cours). La table de vérité tronquée :

e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0	s_2	s_1	s_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
1	0	0	0	0	0	0	0	1	1	1

et le dessin :



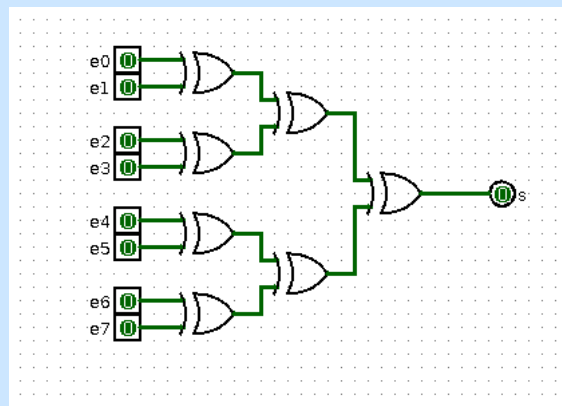
1. Ces deux exercices sont aussi dans le cahier de TD.

Pour l'imparité, on peut faire une table de vérité :

x_2	x_1	x_0	$\text{imp}(x_2, x_1, x_0)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

On constate que $\text{imp}(x_2, x_1, x_0) = \overline{x_2}(x_1 \oplus x_0) + x_2(\overline{x_1} \oplus \overline{x_0})$. Comme $x \oplus y = \overline{x}y + x\overline{y}$, et que le \oplus est associatif, on obtient $\text{imp}(x_2, x_1, x_0) = x_2 \oplus (x_1 \oplus x_0) = x_2 \oplus x_1 \oplus x_0$.

Plus généralement, pour l'imparité à n bits : on note $x = (x_{n-1} \dots x_0)_2$. Notez que $\text{imp}(x) = 1$ ssi $\sum_{i=0}^{n-1} x_i \bmod 2 = 1$. Or, \oplus peut être vu comme l'addition modulo 2 ; on en déduit que $\text{imp}(x) = x_{n-1} \oplus \dots \oplus x_0$. Par exemple, pour l'imparité sur 8 bits, on peut utiliser le circuit suivant :



2.2 Quelques éléments du processeur LC-3

Dans la suite du cours, nous allons construire un processeur maison, le LC-3. Nous prenons de l'avance dans ce TP en construisant quelques sous-circuits que nous assemblerons ensemble dans un prochain TP (source : équipe pédagogique Archi, Univ P7).

EXERCICE 2 ► ALU LC-3

Récupérer sur la page web du cours le fichier `tp2_alu.tu.circ` et le tester pour savoir ce qu'il fait. On remplira les cases vides du tableau suivant avec des formules dépendant des entrées Input1, Input2 et Cst :

e_2/UseCst	0	1
(00)		
(01)		
(10)		
(11)		

SOLUTION. Le circuit ALU présente :

- 5 entrées : Input1, Input2, Cst (sur 16 bits), UseCst (1bit), e2 (2bits).
- 1 sortie Output sur 16 bits.

Il contient un additionneur 16 bits qui réalise l'addition de Input1 avec Input2 ou Cst ; un AND bit à bit, avec les mêmes opérandes, et un inverseur 16 bits, avec en entrée Input1.

Le multiplexeur 2*16bits evrs 16 bits commandé par UseCst permet de sélectionner la deuxième opé-

rande de l'additionneur et du AND bit à bit. Le multiplexeur 4*16 bits vers 16 bits commandé par e2 permet de choisir le résultat placé sur la sortie.

En résumé, cela donne le tableau :

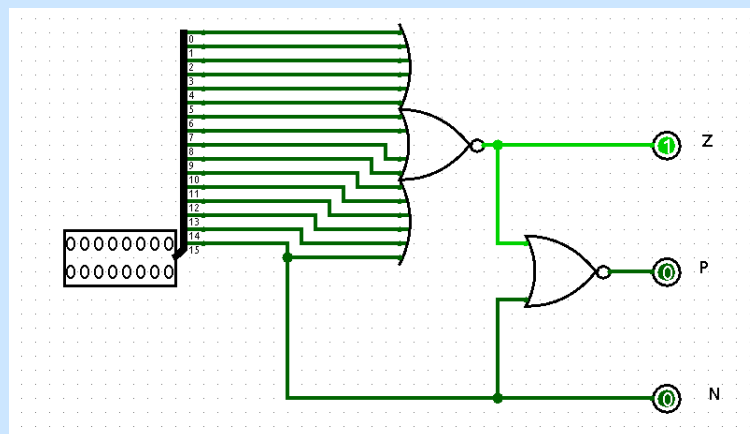
$e_2/UseCst$	0	1
(00)	Input1+Input2	Input1+Cst
(01)	Input1 AND Input2	Input1 AND Cst
(10)	NON(Input1)	NON(Input1)
(11)	-	-

EXERCICE 3 ► NZP LC-3

Dans un nouvel onglet du fichier précédent (nommé NZP), créer un circuit qui prend une entrée 16 bits nommée RES considérée en complément à 2 sur 16 bits, et qui en sortie a un "Pin" 3 bits nommé NZP. Le bit de poids faible (P) est égal à 1 ssi $RES > 0$, le bit du milieu (Z) est égal à 1 ssi $RES = 0$, et le bit de poids fort est à 1 ssi $RES < 0$. Bien tester.

SOLUTION. Si on note s le signe de RES, on a $s = 0 \Rightarrow RES \geq 0$, $s = 1 \Rightarrow RES < 0$ (prendre des exemples), donc :

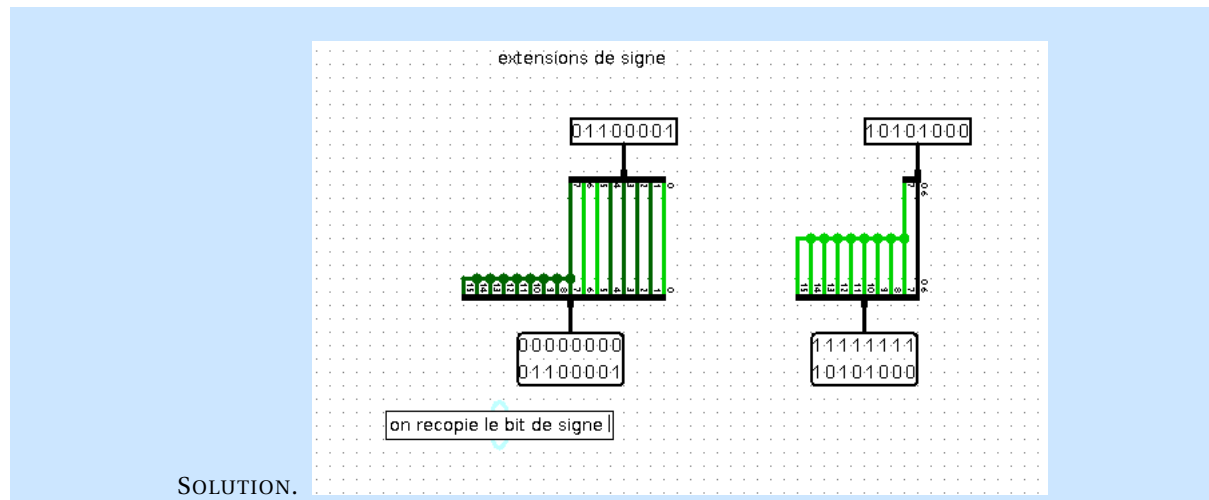
- $N = s$
- $P = \bar{s}\bar{Z}$.
- $Z = 0$ ssi tous les bits de RES sont nuls, donc $Z = \overline{r_{p-1} + r_{p-2} + \dots + r_0}$.



EXERCICE 4 ► Extensions de signe

D'après un des exercices de TD, l'extension de signe en complément à 2 se fait en dupliquant le bit de poids fort autant de fois que nécessaire. Créez dans un même fichier deux onglets différents :

- dans un onglet appelé Brouillon, construire l'extension de signe d'un entier codé en complément à 2 sur 8 bits vers 16 bits. Tester.
- dans un onglet appelé Offset6 prendre une entrée sur 8 bits, sélectionner les 7 bits de poids faibles, et réaliser l'extension vers un entier 16 bits. On pourra utiliser le composant BitExtender de la librairie (dans Wiring).



2.3 Dépassements en C

EXERCICE 5 ► Dépassement de capacité en complément à 2

Récupérer le fichier `tp2_comp2etu.c` sur la page web du cours. En supposant qu'un `char` prend un octet et un `short` 2 octets, prédire le comportement de ce programme à l'exécution. Vérifier.

```

1  #include <stdio.h>
   #include <stdlib.h>
   #include <fcntl.h>
   #include <unistd.h>

6
   int main()
   {
       unsigned char uc1, uc2, uc3; signed char sc1, sc2, sc3;
       unsigned short ui1, ui2, ui3; signed short si1, si2, si3;
11  printf("\n Taille de char : %lu octets \n\n", sizeof(char));
       uc1 = 200 ; uc2 = 60 ; uc3 = uc1 + uc2 ;
       printf("(unsigned char) uc1 = %d, uc2 = %d, uc1+uc2 = %d \n", uc1, uc2, uc3) ;
       sc1 = 100 ; sc2 = 60 ; sc3 = sc1+sc2 ;
       printf("(signed char) sc1 = %d, sc2 = %d, sc1+sc2 = %d \n", sc1, sc2, sc3) ;
16  sc1 = -100 ; sc2 = -60 ; sc3 = sc1+sc2 ;
       printf("(signed char) sc1 = %d, sc2 = %d, sc1+sc2 = %d \n", sc1, sc2, sc3) ;
       printf("\n Taille de short : %lu octets\n\n", sizeof(short));
       ui1 = 6000 ; ui2 = 60000 ; ui3 = ui1+ui2 ;
       printf("(unsigned short) ui1 = %d, ui2 = %d, ui1+ui2 = %d \n", ui1, ui2, ui3) ;
21  si1 = -10000 ; si2 = -30000 ; si3 = si1+si2 ;
       printf("(signed short) si1 = %d, si2 = %d, si1+si2 = %d \n", si1, si2, si3) ;
       return 0;
   }

```

SOLUTION. Directement dans le texte du source :

```

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

```

```
int main()
```

```

{
    unsigned char uc1, uc2, uc3; signed char sc1, sc2, sc3;
    unsigned short ui1, ui2, ui3; signed short si1, si2, si3;
    printf("\nTaille_de_char: %d octets\n\n", sizeof(char));

    uc1 = 200 ; uc2 = 60 ; uc3 = uc1 + uc2;
    printf("(unsigned_char)_uc1=%d, _uc2=%d, _uc1+uc2=%d\n", uc1, uc2, uc3) ;
    /* uc3 = 260 % 256 = 4. */

    sc1 = 100 ; sc2 = 60 ; sc3 = sc1+sc2 ;
    printf("(signed_char)_sc1=%d, _sc2=%d, _sc1+sc2=%d\n", sc1, sc2, sc3) ;
    /* sc1 + sc2 = 160 > 127 = 2^7-1, le plus grand entier en complement a 2 sur 8 bits.
       160 = 128 + 32 = 2^7 + 2^5. Le bits de rang 7 est interprete en complement a 2
       sur 8 bits comme un bit de poids negatif. Donc sc3 = -128 + 32 = -96.
    */

    sc1 = -100 ; sc2 = -60 ; sc3 = sc1+sc2 ;
    printf("(signed_char)_sc1=%d, _sc2=%d, _sc1+sc2=%d\n", sc1, sc2, sc3) ;
    /* sc1 + sc2 = -160 < -128, le plus petit entier en complement a 2 sur 8 bits.
       Codage de sc1 : -100 = -128 + 28 = -2^7 + 2^4 + 2^3 + 2^2 = (10011100)
       Codage de sc2 : -60 = -128 + 68 = -2^7 + 2^6 + 2^2 = (11000100)
       Codage de sc3 : (01100000) = 2^6 + 2^5 = 64 + 32 = 96.
       Remarquons que -160 % 256 = 96 (-160 = -1 * 256 + 96).
    */

    printf("\nTaille_de_short: %d octets\n\n", sizeof(short));

    ui1 = 6000 ; ui2 = 60000 ; ui3 = ui1+ui2 ;
    printf("(unsigned_short)_ui1=%d, _ui2=%d, _ui1+ui2=%d\n", ui1, ui2, ui3) ;
    /* ui3 = 66000 % 2^16 = 66000 % 65536 = 464. */

    si1 = -10000 ; si2 = -30000 ; si3 = si1+si2 ;
    printf("(signed_short)_si1=%d, _si2=%d, _si1+si2=%d\n", si1, si2, si3) ;
    /* si1 + si2 = -40000, donc si3 = -40000 % 65536 = 25536 (-40000 = -1*65536 + 25536). */

    return 0;
}

```

TP 3

Circuits séquentiels, registres, mémoire

Objectifs :

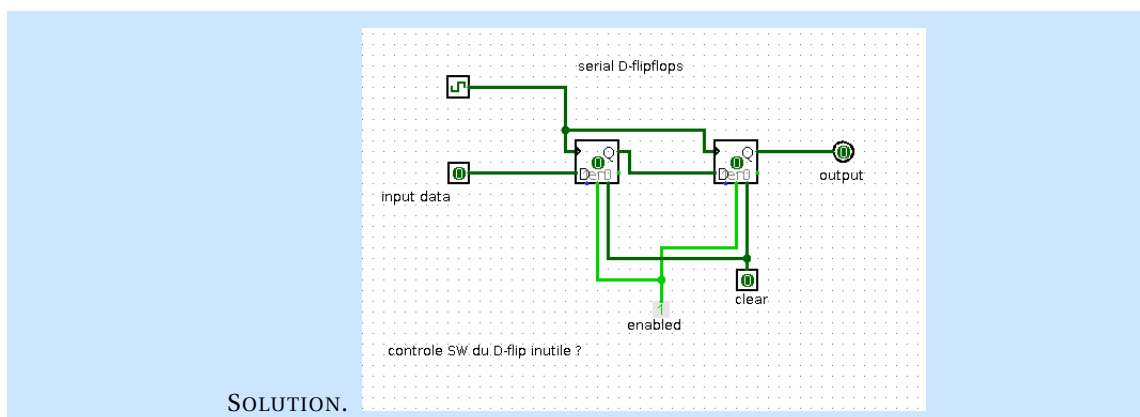
- Écrire des circuits séquentiels simples en LOGISIM.
- Utiliser la librairie LOGISIM pour la mémoire.
- Savoir réaliser un banc de registres

3.1 Bascules

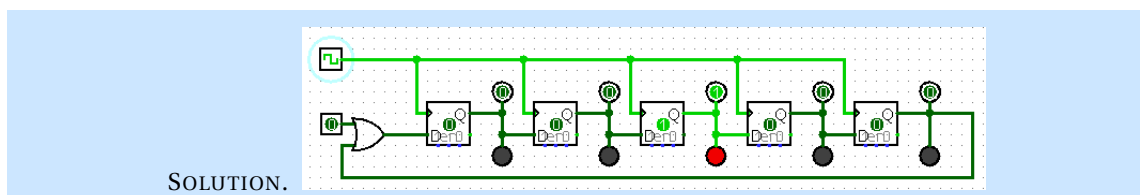
EXERCICE 1 ► Bascules

Le but est de bien comprendre le principe des **bascules D (flip-flop)**.

- Montez une **bascule D (flip-flop)** de la librairie (en mode front montant *Rising Edge* puis testez son comportement. Montez ensuite deux bascules D **en série** en les reliant au même signal d'horloge ; vérifiez le comportement sur deux cycles d'horloge successifs.



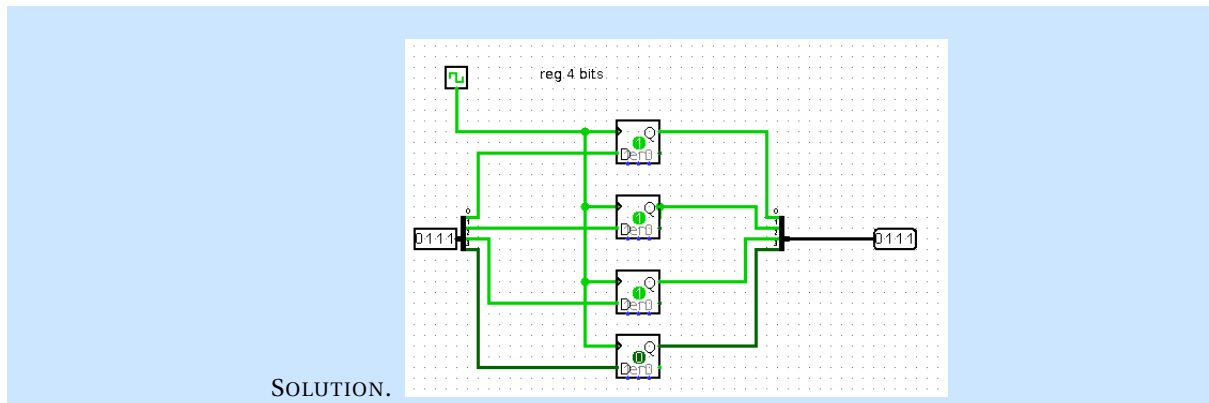
- Construisez un **chenillard à 5 leds** en utilisant 5 bascules D montées en série. Le principe est qu'à chaque cycle d'horloge, une seule led est allumée, et la led allumée est décalée d'un cran vers la droite à chaque cycle d'horloge. Le cycle suivant l'allumage de la led 5, c'est la led 1 qui doit de nouveau être allumée. Testez votre circuit.



3.2 Registres, compteurs et mémoire

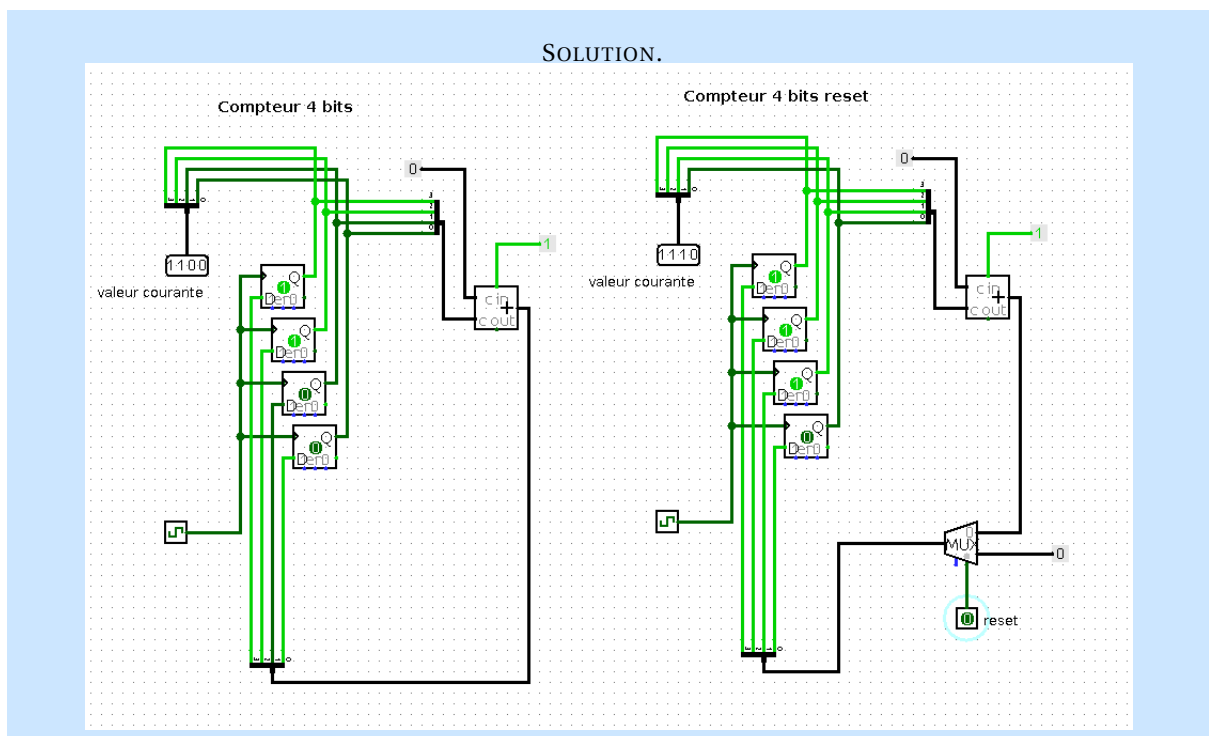
EXERCICE 2 ► Registre

Un registre n -bits est une mémoire constituée d'un assemblage en parallèle de bascules D. Construisez un **registre 4 bits**, que vous testerez, puis que vous comparerez avec celui de la bibliothèque (toujours sur front montant).



EXERCICE 3 ► Compteur

Réalisez un **compteur 4 bits** en utilisant les outils suivants de la bibliothèque : un registre 4 bits et un additionneur. Testez en mettant une horloge, avec l'option *Simulate->Ticks Enabled*, puis ajoutez un signal *reset* qui permet de remettre le compteur à 0.

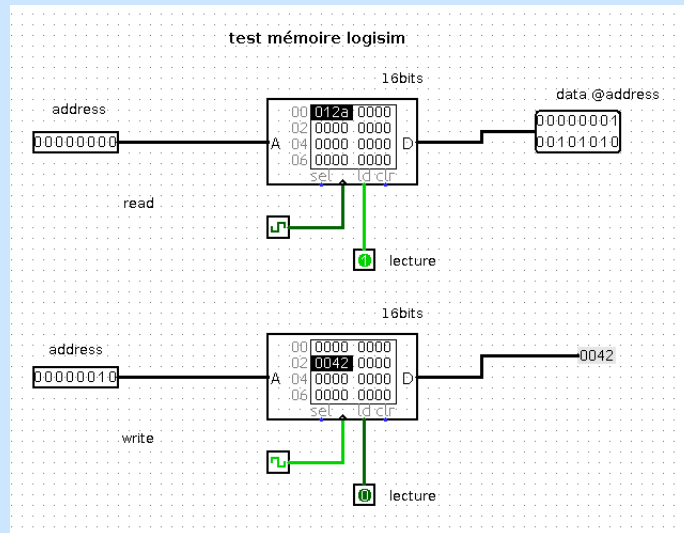


EXERCICE 4 ► Utilisation de la RAM

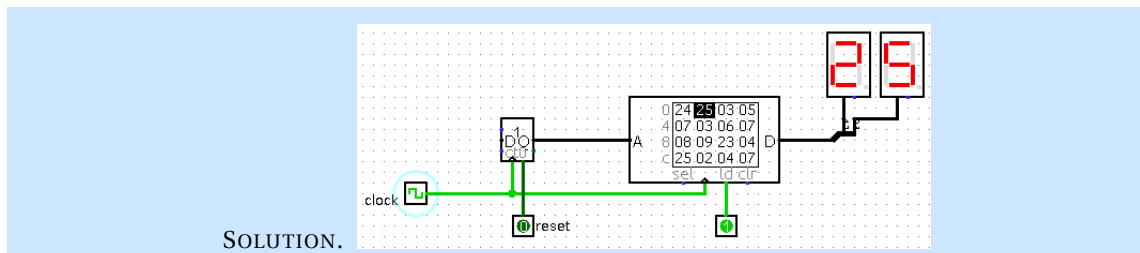
En vous aidant de la documentation :

- Instanciez une **RAM** avec adressage 4 bits et contenu 8 bits (en mode *One synchronous load/store port*, comme dans le cours).
- Faites la fonctionner en lecture. Pour remplir la mémoire avant de tester la lecture, on pourra faire un clic droit sur le composant de mémoire, puis *Edit Contents*. Il n'est pas nécessaire de sauvegarder le contenu de la mémoire dans un fichier.
- Dans un autre onglet, faire fonctionner une mémoire en écriture.
- Comment faire pour faire fonctionner la même mémoire en écriture et en lecture ? *On pourra judicieusement regarder le cours, la documentation de la RAM ainsi que le composant Buffer.*

SOLUTION. Dans la solution ci-dessous, la RAM testée a un adressage sur 8 bits, et stocke des données de 16 bits. Mais le principe est le même avec un adressage 4 bits et contenu 8 bits.



- Modifiez votre circuit de manière à ce qu'il affiche successivement (dans deux afficheurs hexadécimaux) le contenu de chaque case de la mémoire. Pour cela, vous utiliserez un compteur 4 bits, et à chaque cycle d'horloge votre circuit affichera le contenu de la case mémoire dont l'adresse est donnée par le compteur.



On peut aussi utiliser un composant Probe et afficher en Hexa directement.

3.3 Banc de registre

EXERCICE 5 ► Banc de registres

Construire un **banc de registres, avec 4 registres 4 bits** capable de lire deux registres et au besoin d'écrire un registre. On commencera par se poser la question du nombre d'entrées et de sorties d'un tel circuit.

SOLUTION. Le banc de registre (cf le cours), possède 4 registres 4 bits, 5 entrées :

- L'indice du premier registre à lire (sur 2 bits).
- L'indice du deuxième registre à lire (sur 2 bits).
- L'indice du registre à écrire (sur 2 bits).
- La donnée à écrire (sur 4 bits).
- Le signal d'écriture (1 = écriture) (1 bit).

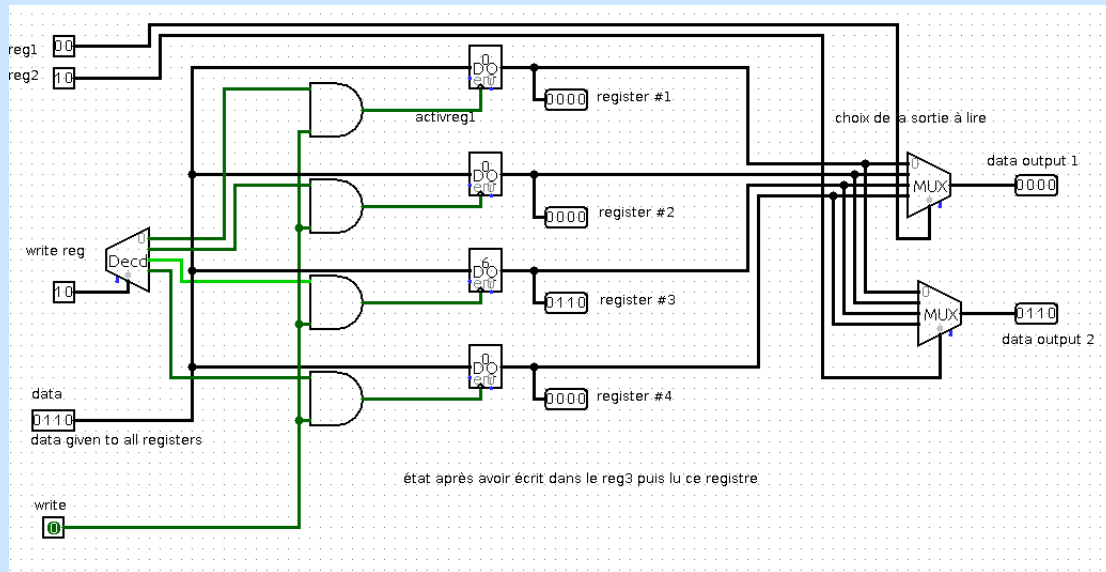
et 2 sorties :

- La valeur du premier registre (sur 4 bits)
- La valeur du deuxième registre (sur 4 bits)

La partie lecture est la plus simple à faire, il suffit de brancher la sortie des registres à un multiplexeur qui permet de choisir le registre à lire à partir de son numéro (reg1 ou reg2). On peut tester cette partie en modifiant le contenu des registres (clic droit dessus).

La partie écriture est un peu plus complexe, la donnée est donnée à l'ensemble des registres. Un registre i écrit cette donnée si et seulement si l'entrée write est 1, et son numéro est donné en entrée (donc il faut un décodeur).

On obtient finalement :



TP 4

Test de circuit, circuits dédiés.

Objectifs :

- Utiliser le simulateur de LOGISIM pour tester ses circuits.
- Concevoir un “circuit dédié”.

Fichiers fournis : tp4_cpt4.circ, tp4_pgcdetu.circ

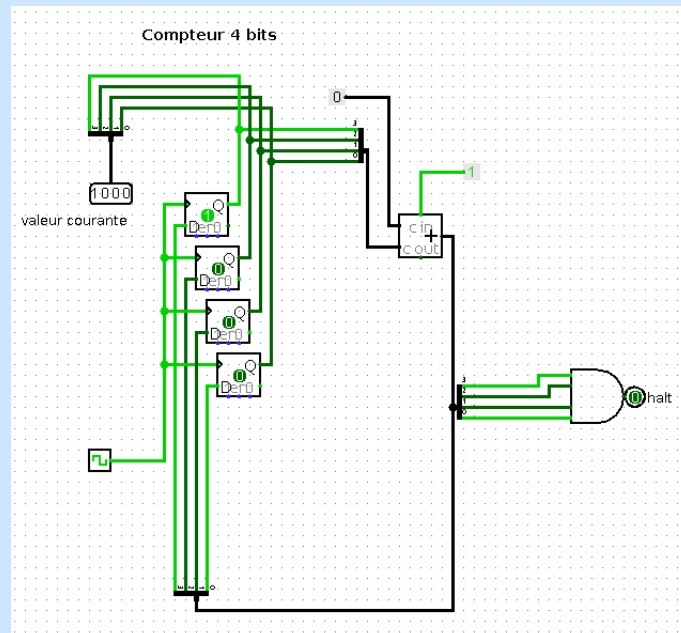
4.1 LOGISIM : test de circuit

EXERCICE 1 ► Simulation à la ligne de commande

Récupérez le compteur 4 bits du tp précédent, fichier tp4_cpt4.circ.

- Regardez la documentation (Command Line Verification) et lancez une simulation à la ligne de commande.
- En rajoutant une sortie nommée halt (voir la doc), faire en sorte que la simulation termine après un cycle du compteur. *Indication : après quelle valeur du compteur la simulation doit-elle s'arrêter ?*

SOLUTION. Voici le compteur modifié :



La commande :

```
java -jar logisim-generic-2.7.1.jar tp3_testcompteur4.circ -tty table
```

fournit :

```
0000
0001
```

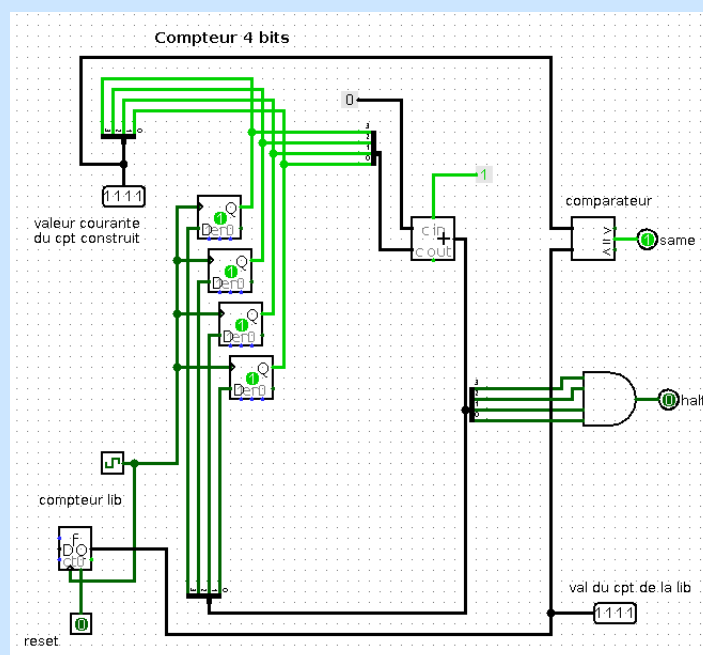
...
1110

EXERCICE 2 ► Comparaison fonctionnelle de circuits

On va maintenant comparer ce compteur avec le compteur de la bibliothèque LOGISIM :

- Récupérer le compteur 4 bits de la bibliothèque, et lire sa documentation.
- En ajoutant un comparateur 4 bits, construire une unique sortie `same` qui est vraie ssi à chaque tic d'horloge la sortie de notre compteur est la même que celle de celui de la bibliothèque. Quelle est la valeur attendue pour la simulation ?

SOLUTION. Sans trop de difficulté, on obtient :



Pour bien faire, il faudrait aussi supprimer les valeurs des compteurs. La simulation donne alors une unique colonne de 1.

4.2 Circuits dédiés

EXERCICE 3 ► Une calculette à PGCD

D'après <http://dept.cs.williams.edu/~tom/courses/237/>. Nous allons construire un circuit qui réalise le calcul du PGCD pour les entiers positifs (8 bits). La figure 4.1 vous fournit une définition et un programme C pour ce calcul.

- Quel est le pgcd de 12 et 8 ?
- Expliquer la différence entre cet algorithme et celui que vous connaissez pour calculer le pgcd ?

Pour vous simplifier la vie, nous vous fournissons un circuit (figure 4.2) qui possède déjà les composants de données et de calcul. Nous n'aurez plus qu'à ajouter les composants de contrôle.

- Sur la page web du cours, téléchargez le circuit `tp4_pgcdetu.circ`.
- Observez le circuit : les entrées de gauche seront utilisées pour rentrer les valeurs initiales pour `x` et `y` (en binaire). Des sondes (Probe) décimales ont été ajoutées pour pouvoir lire ces valeurs en base 10 (ainsi que les valeurs en sortie des registres).
- Remarquez bien qu'un "tick" d'horloge effectue 1 étape dans le calcul, *le calcul n'est pas instantané!*

“En arithmétique élémentaire, le plus grand commun diviseur, abrégé en général PGCD, de deux nombres entiers naturels non nuls est le plus grand entier qui divise simultanément ces deux entiers. Par exemple le PGCD de 20 et 30 est 10. En effet, leurs diviseurs communs sont 1, 2, 5 et 10.”

Listing 4.1 – 'Afficherec.asm'

```

1  int gcd(int x, int y){
    while (x != y){
        if (x<y) y=y-x;
        else   x=x-y;
    }
    return x;
6 }
```

FIGURE 4.1 – Documentation pour le PGCD (Wikipédia)

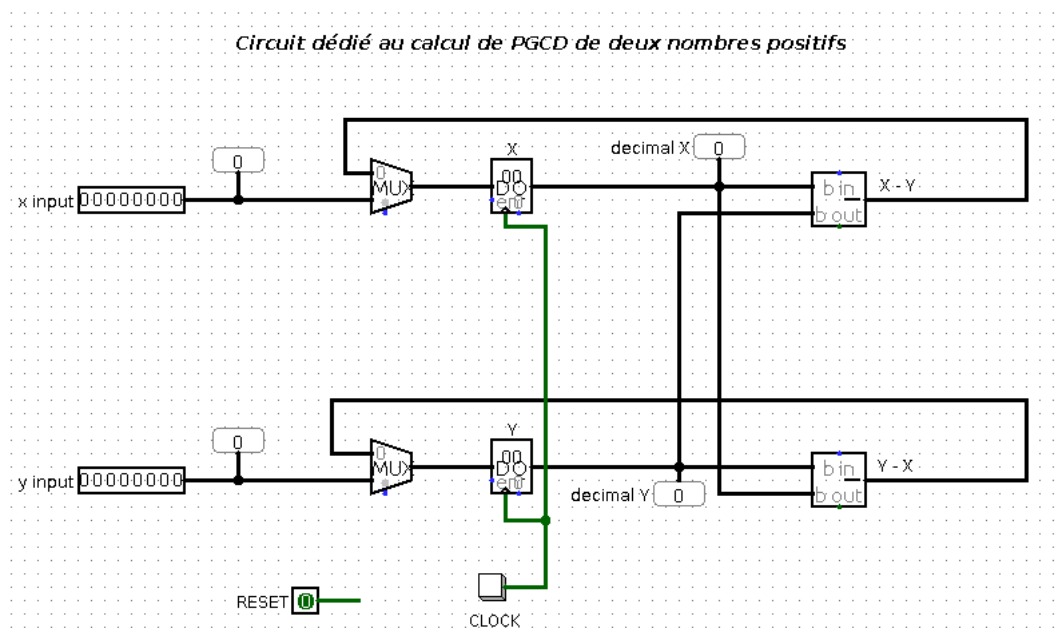
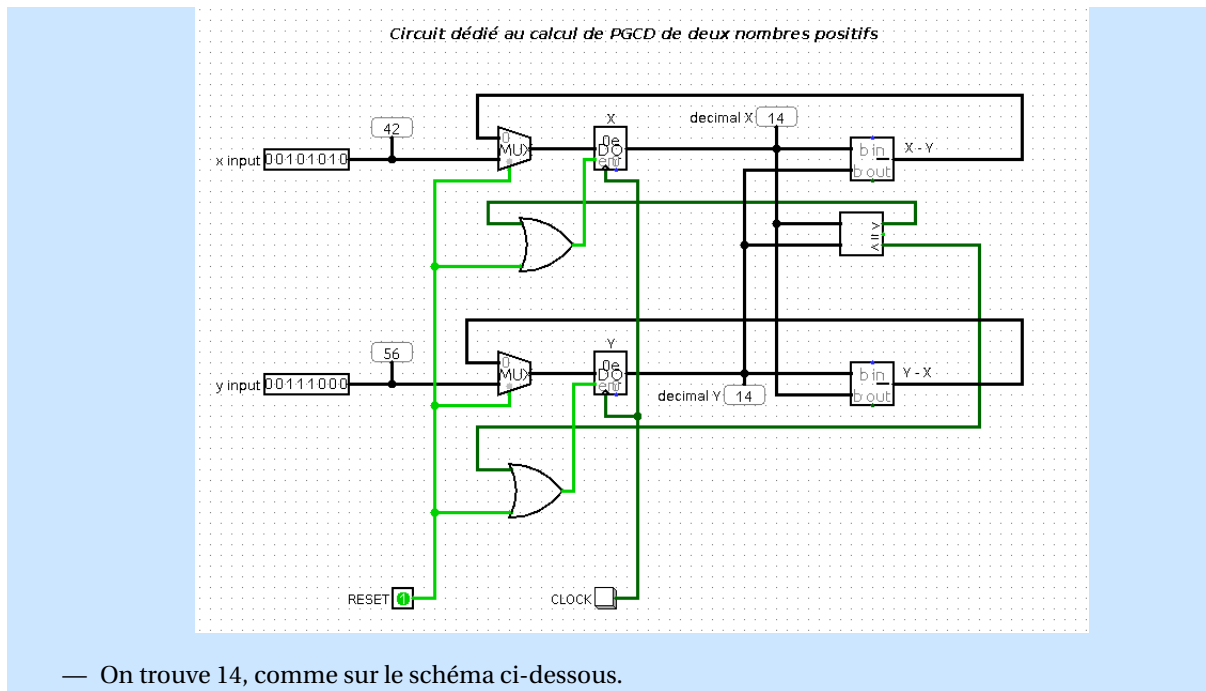


FIGURE 4.2 – Le circuit fourni

- Rajoutez les composants de contrôle : quand les entrées sont disponibles, l'entrée (Pin) reset=1 doit permettre d'initialiser les registres X et Y avec ces valeurs (au premier appui sur Clock). Ensuite, on remettra l'entrée reset à 0. Puis chaque appui du bouton Clock cause l'exécution d'une étape de l'algorithme. Une fois que le PGCD est trouvé, plus aucun changement ne doit arriver. *On pourra judicieusement se poser les deux questions suivantes : quelles sont les conditions de sélection de l'entrée 1 du multiplexeur du haut (resp. du bas) ? Quelles sont les conditions d'écriture de chacun des registres (entrée Enable) ?*
- Vérifiez avec $x = 42$ et $y = 56$. Le PGCD trouvé est ?

SOLUTION. — Le pgcd de 12 et 8 est 4.

- En général, l'algorithme que l'on connaît utilise la propriété suivante : si $a > b$ alors $pgcd(a, b) = pgcd(a \bmod b, b)$. Ici on n'effectue pas de division, mais des soustractions successives.
- Composants de contrôle :
 Sans surprise, le signal reset permet de sélectionner l'entrée du Mux. On sélectionne l'entrée x (resp. y) seulement si reset est à 1.



EXERCICE 4 ► Un automate reconnaisseur

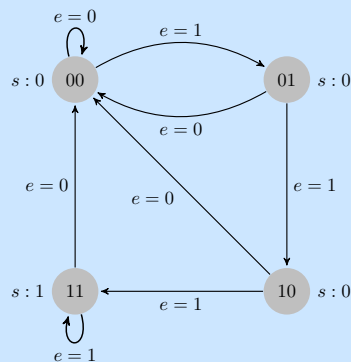
Construire en LOGISIM un automate séquentiel reconnaissant le motif 111 : la sortie doit être à 1 sur un cycle si, lors des trois cycles précédents, l'entrée était à 1 (en tout cas sur le front montant de l'horloge, en fin de cycle). On utilisera **obligatoirement** la méthodologie suivante :

- Décrire la machine voulue en langage courant. *En particulier, que veut dire “reconnaître” ?*
- Quelles sont les entrées et les sorties de l'automate à construire ?
- Dessiner un automate équivalent au circuit à construire.
- Construire la table de vérité du circuit. *L'état suivant est calculé à partir de l'état courant et de l'entrée, et la sortie est calculée à partir de l'état courant uniquement.*
- Dessiner sur papier le circuit.
- Dessiner avec LOGISIM et tester (ce n'est pas si simple !)
- Dans un autre onglet, construire un circuit qui utilise des flaps-flops pour se rappeler de 3 valeurs successives de l'entrée.
- (optionnel) Tester l'égalité fonctionnelle de ces deux circuits.

SOLUTION. La fonctionnalité du circuit est :

```
001011011101011010011101111110 <- INPUT
?00000000010000000000100011110 <- OUTPUT
```

Il y a donc une unique entrée sur 1 bit et une unique sortie sur 1 bits. Sans surprise, les états de notre automate vont coder “j’ai lu N” fois “1” de suite avec $0 \leq N \leq 3$, donc 4 états (donc 2 bits).



La table de vérité calcule la sortie s en fonction de l'état courant (current state, codé sur 2 bits $q_1 q_0$) et de l'entrée (e , 1 bit). Elle calcule aussi le nouvel état ($q_1^+ q_0^+$). Sans suspense on obtient immédiatement :

q_1	q_0	e	q_1^+	q_0^+	s
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	1	1	1

La fonction de sortie est facile à calculer et implémenter :

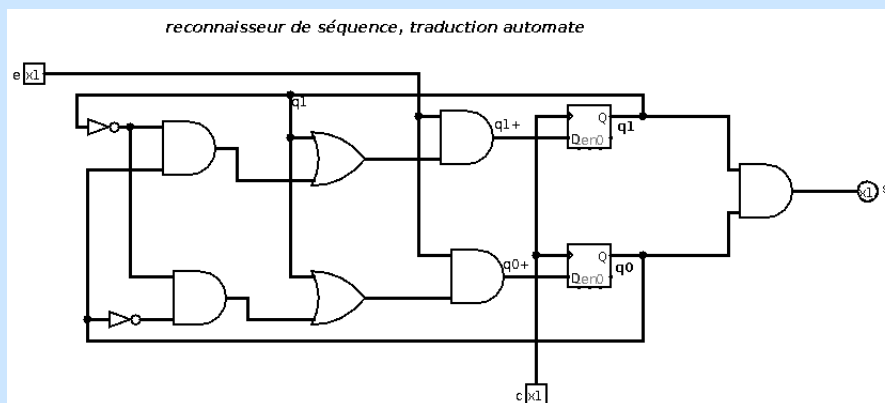
$$s = q_1 \cdot q_0$$

Pour les fonctions de calcul de nouvel état, les formules sont un peu plus compliquées :

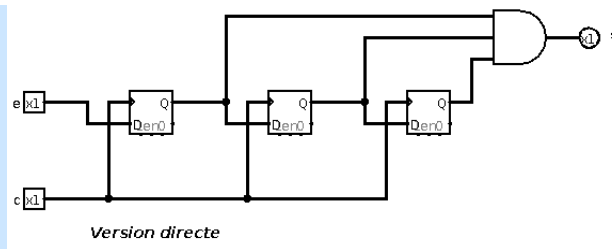
$$q_0^+ = e \cdot (\bar{q}_1 \bar{q}_0 + \bar{q}_0 q_1 + q_0 q_1) = e \cdot (\bar{q}_0 \bar{q}_1 + q_1)$$

$$q_1^+ = e \cdot (\bar{q}_1 q_0 + q_1 \bar{q}_0 + q_1 q_0) = e \cdot (q_1 + q_0 \bar{q}_1)$$

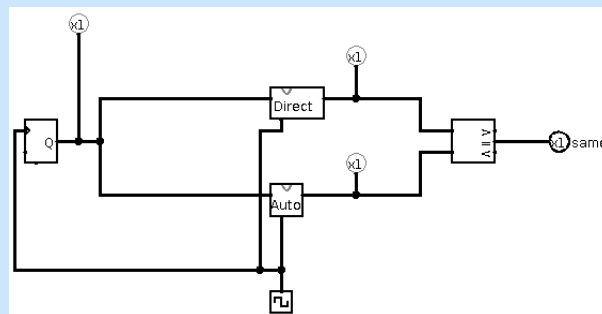
Ce qui donne, sous logisim :



La reconnaissance directe est moins fastidieuse :



et enfin, la comparaison bonus :



TP 5

Introduction à l'archi LC-3, jouons avec le simulateur

Objectifs :

- Utiliser un simulateur de l'architecture LC-3 pour comprendre le jeu d'instructions.
- Écrire des programmes simples en langage machine LC-3, puis en assembleur LC-3.

Ce tp s'inspire fortement des "lab 6" et "lab 7" <http://castle.eiu.edu/~mathcs/mat3670/index/index.html>, avec l'aimable permission de leurs auteurs. Nous allons utiliser le simulateur LC-3 nommé PennSim dont la documentation est disponible à l'adresse (ou à partir de la page du cours, ou google PennSim) :

<http://castle.eiu.edu/~mathcs/mat3670/index/Webview/pennsim-guide.html>

Fichiers fournis : tp51a.asm, tp51b.asm, tp52.asm, tp54a.asm, tp54b.asm

5.1 Jouons avec le simulateur de LC-3

EXERCICE 1 ► Installation, documentation

Commencez par récupérer tp51a.asm et tp51b.asm.

1. Lire attentivement la documentation en ligne du logiciel.
2. Assembler, charger et exécuter pas-à-pas le programme de test tp51a.asm rappelé ci-dessous. Comme nous n'avons pas chargé l'OS, il faut amener PC à la première adresse du programme à l'aide de la commande `set PC x3000`. Observer l'évolution de l'état des registres et de la mémoire au cours de l'exécution du programme. Pourquoi est-il possible d'exécuter ainsi le programme sans charger l'OS?

Listing 5.1 – tp51a.asm

```
.ORIG X3000      ; spécifie l'adresse de chargement du programme
LD R1,a
LD R2,b
ADD R0,R1,R2
5  ADD R0,R0,-1
    ST R0, r
stop: BR stop      ; juste une astuce pour bloquer l'exécution ici
r:    .BLKW 1
a:    .FILL 10
10  b:    .FILL 6
    .END
```

SOLUTION. Faire attention que le programme as lancé le soit à partir de l'interface graphique, et non à partir d'un terminal utilisateur (auquel cas ce serait l'assembleur x86 qui serait invoqué). On n'a pas besoin de charger l'OS car le programme ne fait appel à aucune de ses primitives.

3. Remettre le simulateur à zéro avec la commande `reset`. Assembler et charger l'OS, puis le programme tp51b.asm. Exécuter pas-à-pas le programme, et observer bien l'exécution de l'appel système `PUTS` : à quel programme appartiennent les instructions exécutées?

Listing 5.2 – tp51b.asm

```
.ORIG X3000      ; spécifie l'adresse de chargement du programme
LEA R0,chaîne
PUTS
```



```

        HALT                ; rend la main à l'OS
5  chaine: .STRINGZ "hello\n"
        .END

```

SOLUTION. PUTS est une macro, qui provoque une interruption logicielle (TRAP x22), et lance d'exécution de code de l'OS (la routine TRAP_PUTS pour être précis; d'ailleurs, cette routine fait aussi appel à TRAP_OUT).

EXERCICE 2 ► Exécution d'un programme en langage machine

Vous avez en TD réalisé le décodage d'un programme LC-3 écrit en langage machine. Une correction se trouve Figure 5.1. Parcourir rapidement cette correction, et répondre aux questions suivantes¹ :

- À l'aide de quelles instructions récupère-t-on une donnée en mémoire dans ce programme ? Pouvait-on faire autrement ?
- Comment est réalisé le saut de compteur de programme pour réaliser la boucle ? Que devient le *label* dans le programme assemblé ?

Ensuite, assembler et lancer la simulation pas à pas sur le fichier tp52.asm :

Listing 5.3 – tp52.asm

```

;; Author: Bill Slough for MAT 3670
;; Adapted for LIF6/Univ Lyon 1 by Laure Gonnord, oct 2014.
        .ORIG X3000          ; where to load in memory instructions and data
        .FILL x5020
5       .FILL x1221
        .FILL xE404
        .FILL x6681
        .FILL x1262
        .FILL x16FF
10      .FILL x03FD
        .FILL xF025
        .FILL x0006          ; data word
        .END

```

Bien que l'on ait "assemblé" à la main, il faut quand-même effectuer avec la commande `as` la transformation en un fichier objet `.obj`. Bien comprendre toutes les étapes lors d'une exécution pas-à-pas avant de passer à l'exercice suivant. On remarquera que le simulateur LC-3 donne l'équivalent en langage d'assemblage des instructions machine considérées.

Adresse	Contenu	Contenu binaire	Détails des instructions	pseudo-code
x3000	x5020	0101 000 000 1 00000	AND, DR=SR=R0, Imm5=x00	$R_0 \leftarrow R_0 \& 0 = 0$
x3001	x1221	0001 001 000 1 00001	ADD, DR=R1, SR=R0, Imm5=x01	$R_1 \leftarrow R_0 + 1 = 1$
x3002	xE404	1110 010 0 0000 0100	LEA, DR=R2, Offset9=x04	$R_2 \leftarrow 3007(@fin)$
x3003	x6681	010 011 010 00 0001	LDR, DR=R3, baseR=R2, Offset6=x01	$R_3 \leftarrow mem[3008] = 6$
loop:x3004	x1262	0001 001 001 1 00010	ADD, DR=R1, SR=R1, Imm5=x02	$R_1 \leftarrow R_1 + 2$
x3005	x16FF	0001 011 011 1 11111	ADD, DR=R3, SR=R3, Imm5=-1!	$R_3 \leftarrow R_3 - 1$
x3006	x03FD	0000 001 1 1111 1101	BRp offset=-3	if $R_3 > 0$ goto loop
fin:x3007	xF025	1111 0000 0010 0101	TRAP trapvect=x25	HALT
x3008	x0006	donnée	-	

FIGURE 5.1 – Un programme en binaire/hexadécimal (tp52.asm)

1. On peut se reporter à l'antisèche de l'annexe B.2

SOLUTION. Ce programme effectue donc 6 tours de boucles (le nombre contenu à l'adresse 0x3008), à la fin R_1 contient donc $1 + 6 * 2 = 13$. L'accès à une case en mémoire s'effectue ici en chargeant l'adresse dans un registre (LEA) puis en accédant à cette adresse avec LDR. On aurait pu directement utiliser LD. Le saut de compteur pour réaliser la boucle est codé dans l'instruction. C'est le programme assembleur qui réalise la conversion *label* vers "décalage de PC".

EXERCICE 3 ► Assemblage à la main

Sur papier d'abord :

1. Écrire un programme en assembleur LC-3 qui écrit 10 fois le caractère 'Z' sur l'écran.
2. Assembler ce programme à la main, puis sur le modèle du Listing 5.3, créer un programme "pré-assemblé".
3. Utiliser le simulateur pour tester votre programme.

SOLUTION. C'est une simple boucle for, mais il faut tout assembler à la main, ce qui est assez long :

Listing 5.4 – tp53cor.asm

```
;; Author: Laure Gonnord
.ORIG X3000
.FILL xE006      ; r0 <- char Z à pc + 1 + 6
.FILL x5260      ; r1 <- 0
.FILL x126A      ; r1 <- r1 + 10
.FILL xF022      ; trap x22 imprime 'Z'
.FILL x127F      ; r1 <- r1 - 1
.FILL x03FD      ; si >0 alors pc + 1 - 3
.FILL xF025      ; halt
.FILL x005A      ; 'Z'
.END
```

5.2 Écriture et simulation de programmes en assembleur LC-3

Jusqu'à présent nous avons écrit des programmes en remplissant la mémoire directement avec les codages des instructions. Nous allons maintenant écrire des programmes de manière plus simple, en écrivant les instructions en *assembleur LC-3*.

EXERCICE 4 ► Exécution, modification

1. Prévoir le comportement des fichiers tp54a.asm et tp54b.asm. Vérifier avec le simulateur. Quelle est la différence entre les instructions PUTS et OUT?

Listing 5.5 – tp54a.asm

```
;; Author: Bill Slough MAT 3670
;; Adapted for LIF6/Univ Lyon 1 by Laure Gonnord, oct 2014.
.ORIG x3000      ; specify the "origin"; i.e., where to load in memory
LEA R0,HELLO     ;
PUTS             ;
LEA R0,COURSE    ;
PUTS             ;
HALT            ;
HELLO: .STRINGZ "Hello, world!\n"
COURSE: .STRINGZ "LIF6\n"
.END
```

Listing 5.6 – tp54b.asm

```

;; Author: Bill Slough for MAT 3670
;; Adapted for LIF6/Univ Lyon 1 by Laure Gonnord, oct 2014.
        .ORIG x3000      ; specify the "origin"; i.e., where to load in memory
        LD R1,N          ;
5        NOT R1,R1        ;
        ADD R1,R1,#1      ; R1 = -N
        AND R2,R2,#0      ;
LOOP:    ADD R3,R2,R1      ;
        BRzp ELOOP        ;
10       LD R0,STAR       ;
        OUT               ;
        ADD R2,R2,#1      ;
        BRnzp LOOP        ;
ELOOP:   LEA R0,NEWLN      ;
15       PUTS             ;
STOP:    HALT             ;
N:        .FILL 6          ;
STAR:    .FILL x2A         ; the character to display
NEWLN:   .STRINGZ "\n"    ;
20       .END

```

SOLUTION. Le premier programme écrit les trois chaînes à la suite. Bien remarquer que l'on récupère l'adresse des débuts de chaîne avec LEA. Le second programme imprime une ligne de N astériques, avec N une constante donnée en mémoire.

- Écrire un programme assembleur LC-3 qui calcule le min et le max de deux entiers, et stocke le résultat à un endroit précis en mémoire, de label min. Tester avec différentes valeurs.

SOLUTION. Pour le minimum, voici une solution (non vérifiée) :

Listing 5.7 – minmax

```

;; Author: Bill Slough for MAT 3670
;; Modified by Laure Gonnord for LIF6 univ lyon 1.
;; Determines the minimum value of two integers
        .ORIG x3000      ; specify the "origin"; i.e., where to load in memory
5  IF:    LD R0,X          ; if (x-y <0)
        LD R1,Y
        NOT R2,R1
        ADD R2,R2,#1
        ADD R3,R0,R2
10       BRzp ELSE        ; x-y>=0 --> goto else
THEN:    ST R0,MIN         ; x is the min
        BR STOP           ; goto STOP
ELSE:    ST R1,MIN
STOP     HALT
15  X:     .FILL 6
Y:        .FILL 8
MIN:     .BLZW 1          ; reserved for result
        .END

```

TP 6

LC-3, Exercices de programmation

Objectifs :

- Utiliser le simulateur de l'architecture LC-3 pour bien comprendre le jeu d'instructions.
- Écrire des programmes en assembleur LC-3.

Les exercices ci-dessous seront conçus sur papier puis testés à l'aide du logiciel PENNSIM.

Fichiers fournis : tp6_codage.asm, tp6_saisie.asm

6.1 Chaînes de caractères

EXERCICE 1 ► Saisie d'une chaîne de caractères

Le système d'exploitation du LC-3 fournit une interruption permettant d'afficher une chaîne de caractères (PUTS \equiv TRAP x22), mais on n'a pas la possibilité de saisir une chaîne de caractères. Le but est d'écrire une routine permettant cela. On complètera progressivement le programme ci-dessous.

Listing 6.1 – 'SaisieChaine.asm'

```
5 .ORIG x3000
; Programme principal
  LEA R6,stackend ; initialisation du pointeur de pile
  ; *** A COMPLETER ***
  HALT
; Pile
stack: .BLKW #32
stackend: .FILL #0
.END
```

1. Avant de déclencher une interruption vers le système d'exploitation dans une routine, il est important de sauvegarder l'adresse de retour contenue dans R7 : pourquoi ?
2. Écrire une routine saisie permettant de saisir une chaîne de caractères au clavier, en rangeant les caractères lus à partir de l'adresse contenue dans R1. La saisie se termine lorsqu'un retour chariot (code ASCII 13) est rencontré, et la chaîne de caractères doit être terminée par un caractère '\0' (de code ASCII 0).
3. Tester la routine en écrivant un programme qui affiche "Entrez une chaîne : ", effectue la saisie d'une chaîne en la rangeant à une adresse désignée par une étiquette ch, puis affiche "Vous avez tapé : " suivi de la chaîne qui a été saisie.

SOLUTION. 1. Parce que ce registre peut être écrasé lors de l'interruption.

2. Routine saisie :

```
5 ; Sous-routine pour saisir une chaîne de caractères
; paramètre d'entrée : l'adresse R1 du début de chaîne
saisie: ADD R6,R6,#-1
        STR R7,R6,#0 ; on empile l'adresse de retour (R7)
        ADD R2,R1,#0 ; R2 <- R1
loop:   GETC ; lit un caractère au clavier, résultat dans R0
        OUT ; affiche le caractère lu
        ADD R4,R0,#-10 ; calcule la différence entre NL et le caractère lu
```

```

10      BRz endloop      ; si la différence est nulle, on sort de la boucle
      STR R0,R2,#0      ; sinon, on stocke le car. lu à l'ad. pointée par R2
      ADD R2,R2,#1      ; on incrémente le pointeur en vu de l'itération suivante
      BR loop
endloop: AND R0,R0,#0
      STR R0,R2,#0      ; on s'assure de placer un 0 en fin de chaîne
15      LDR R7,R6,#0
      ADD R6,R6,#1      ; on restaure l'adresse de retour (R7)
      RET

```

3. Le reste du programme principal, tout d'abord les appels aux sous-routines :

```

; Programme principal
      LEA R6,stackend ; initialisation du pointeur de pile

      LEA R0,msg1      ;
5      PUTS             ; affiche la chaîne à l'adresse msg1

      LEA R1,ch1        ; l'adresse où doit être rangée la chaîne à saisir
      JSR saisie        ; saisie de la chaîne de caractères

10     LEA R0,msg2      ;
      PUTS             ; affiche la chaîne à l'adresse msg2

      LEA R0,ch1        ;
15     PUTS             ; affiche la chaîne à l'adresse ch1

      AND R0,R0,#0
      ADD R0,R0,#10
      OUT              ; affiche un retour à la ligne

```

et ensuite, la partie données :

```

; partie dédiée aux données
msg1:  .STRINGZ "Entrez une chaîne   : "
msg2:  .STRINGZ "Vous avez tapé     : "
ch1:   .BLKW #8

```

6.2 Un message codé (CC-TP 2015)

Récupérez le fichier `tp6_codage.asm` sur la page web du cours. Il s'agit de compléter la routine `dechiffre` pour qu'elle permette de déchiffrer le message qui se trouve rangé à partir de l'adresse `msg` sous la forme d'une chaîne de caractères se terminant par 0. La routine prend comme paramètres l'adresse du début du message dans `R0`, et la clé du chiffrement `k` dans `R1`. Pour décoder, la routine remplacera chacun des caractères `c` du message par $k \hat{=} c$ (ou-exclusif bit-à-bit entre `k` et `c`), sauf le caractère 0 final.

1. Assemblez et exécutez une première fois le programme : quel est le message affiché ?
2. Si a et b sont deux variables booléennes, on rappelle que $a \oplus b$ désigne le ou-exclusif entre a et b . En utilisant les lois de Morgan, vérifiez que

$$a \oplus b = \overline{\overline{a \cdot b} \cdot \overline{a \cdot b}}$$

3. On suppose que `R1` contient la clé et `R3` un caractère du message. Donnez un morceau de code en assembleur pour remplacer `R3` par $R1 \hat{=} R3$, en utilisant `R4` et `R5` comme variables intermédiaires du calcul.
4. En utilisant `R2` comme un pointeur pour parcourir le message, donnez sur papier un pseudo-code pour la routine `dechiffre`. Vous référencerez simplement le code de la question précédente par (*).

5. Complétez la routine `dechiffre` dans le fichier `tp6_codage.asm`, d'après le pseudo-code de la question précédente. Testez votre programme, en l'assemblant et en l'exécutant : quelle est la chaîne de caractères affichée ?
6. Comment faire pour coder un message avec la clé fournie ? Complétez le programme (`tp6_codage.asm`) pour tester votre proposition.

SOLUTION. 1. Rowu: {}{st;

2. Appelons F la quantité de droite. En appliquant $\overline{(A \cdot B)} = \overline{A} + \overline{B}$ avec $A = \overline{(a \cdot b)}$ et $B = \overline{(a \cdot b)}$, on a $F = \overline{a} \cdot b + a \overline{b}$ immédiatement.

```

3.
    ; calcul de R3 <- xor(R1, R3)
    NOT R4, R1
    AND R4, R4, R3
    NOT R4, R4
5    ; ici, R4 = not(and(not(R1), R3))
    NOT R5, R3
    AND R5, R1, R5
    NOT R5, R5
    ; ici, R5 = not(and(R1, not(R3)))
10   AND R3, R4, R5
    NOT R3, R3
    ; ici R3 = xor(R1, R3)

```

4. On met tout par facilité :

dechiffre:

```

    ADD R2,R0,0      ; R2 <- R0
loop:
    LDR R3,R2,0      ; R3 <- mem[R2]
5    BRz endloop     ; si R3 = 0, on a atteint la fin de chaîne
    ; calcul de R3 <- xor(R1, R3)
    NOT R4, R1
    AND R4, R4, R3
    NOT R4, R4
10   ; ici, R4 = not(and(not(R1), R3))
    NOT R5, R3
    AND R5, R1, R5
    NOT R5, R5
    ; ici, R5 = not(and(R1, not(R3)))
15   AND R3, R4, R5
    NOT R3, R3
    ; ici R3 = xor(R1, R3)
    STR R3,R2,0      ; mem[R2] <- R3
    ADD R2,R2,1      ; R2 <- R2+1
20   BR loop
endloop:
    RET

```

5. Hello again!

6. Pour coder un message il suffit de mettre un message en clair à la place du message codé.

6.3 Saisie d'un entier au clavier

Vous modifierez et complétez progressivement le fichier `tp6_saisie.asm`.

1. La routine `saisie` permet de lire un entier naturel en base 10 au clavier, et place l'entier lu dans `R1`. Modifiez-la pour qu'elle affiche « Entrez un entier naturel : » avant d'effectuer la saisie.
2. Complétez la routine `aff` de façon à ce qu'elle affiche autant d'étoiles « * » que l'entier naturel contenu dans `R1` lors de son appel. Après avoir affiché `R1` étoiles, la routine doit aussi afficher un retour à la ligne. Suivez les consignes données en commentaire dans le fichier.
3. Modifiez le programme principal `main` pour qu'il effectue la lecture d'un entier au clavier avec `saisie`, puis son affichage avec `aff`.
4. La routine `mul10` fournie permet de multiplier par 10 le contenu de `R1`. Mais, telle qu'elle vous est fournie, elle exécute 10 instructions : modifiez cette routine de façon à ce qu'elle exécute au plus 6 instructions, tout en calculant toujours le même résultat.
5. L'adresse de retour contenue dans `R7` est sauvegardée et restaurée à l'aide de la pile dans `saisie` et `aff`, mais pas dans `mul10` : pourquoi ?

SOLUTION. Le listing suivant fournit des éléments de correction :

```

    .ORIG x3000
; Programme principal
main:  LD R6,spinit ; on initialise le pointeur de pile
      JSR saisie   ; R1 <- lecture d'un entier naturel au clavier
      JSR aff      ; affichage de R1 étoiles à l'écran
      HALT

; routine mul10 : effectue R1 <- R1 * 10
; paramètre donnée-résultat : R1 (on note n la valeur d'entrée de R1)
; registre temporaire (écrasé par l'appel) : R2
mul10: ADD R1,R1,R1 ; R1 <- 2*n
      ADD R2,R1,0   ; R2 <- R1 = 2*n
      ADD R1,R1,R1 ; R1 <- 4*n
      ADD R1,R1,R1 ; R1 <- 8*n
      ADD R1,R1,R2 ; R1 <- 8*n + 2*n = 10*n
      RET

; routine saisie : pour saisir un entier naturel décimal.
; paramètre résultat : R1
; registres temporaires (écrasés par l'appel) : R0, R2
; routine saisie(paramètre résultat : R1) {
;   R1 <- 0;
;   R0 <- readcar(); // lecture d'un caractère au clavier
;   while(R0 - '\n' != 0) {
;       R2 <- '0';
;       R0 <- R0 - '0'; // conversion du caractère en un chiffre
;       R1 <- R1 * 10 + R0;
;       R0 <- readcar();
;   }
; }

saisie: ADD R6,R6,#-1 ; on réserve un emplacement sur la pile
      STR R7,R6,0   ; on sauvegarde l'adresse de retour R7
      LEA R0,prompt
      PUTS          ; affichage du message de prompt
      AND R1,R1,0   ; R1 <- 0
      GETC          ; R0 <- lecture d'un caractère
      OUT           ; affiche le caractère tapé
loops:  ADD R2,R0,-10 ; R2 <- R0 - '\n'
      BRz endloops ; si R0 = '\n', on sort de la boucle

```

```

40      LD R2,carzero    ; R2 <- '0'
      NOT R2,R2
      ADD R2,R2,1      ; R2 = - '0'
      ADD R0,R0,R2      ; R0 <- R0 - '0'
      JSR mul10         ; R1 <- R1 * 10 (R2 est écrasé)
45      ADD R1,R1,R0     ; R1 <- R1 * 10 + R0
      GETC              ; R0 <- lecture d'un caractère
      OUT               ; affiche le caractère tapé
      BR loops         ; retour au début de boucle
endloops: LDR R7,R6,0    ; on restaure le registre R7
50      ADD R6,R6,#1     ; on restaure le pointeur de pile
      RET

; Constantes pour la routine saisie :
carzero: .FILL 48        ; code ascii du caractère '0'
prompt:  .STRINGZ "Entrer un entier: "

55      ; routine aff : pour afficher n étoiles.
      ; paramètre donnée : R1
      ; registre temporaire (écrasé par l'appel) : R0, R2
      ; routine aff(paramètre donnée : R0) {
60      ;   R2 <- R1;
      ;   while(R2 >= 1) {
      ;       printchar('*');
      ;       R2 <- R2-1;
      ;   }
65      ;   R0 <- R1;
      ; }
aff:     ADD R6,R6,#-1    ; on réserve un emplacement sur la pile
      STR R7,R6,0        ; on sauvegarde l'adresse de retour R7
      ADD R2,R1,0        ; R2 <- R1
70      loopa: BRnz endloopa ; si R2 <= 0, on sort de la boucle
      LD R0,carretoile   ; R0 <- '*'
      OUT               ; affiche à l'écran '*'
      ADD R2,R2,-1       ; R2 <- R2 - 1
      BR loopa          ; retour au début de boucle
75      endloopa: LD R0,carcr
      OUT
      LDR R7,R6,0        ; on restaure le registre R7
      ADD R6,R6,#1       ; on restaure le pointeur de pile
      RET

80      ; Constantes pour la routine aff :
      caretoile: .FILL 42 ; code ascii du caractère '*'
      carcr:     .FILL 10 ; code ascii du retour chariot

      ; Gestion de la pile
85      spinit:  .FILL stackend
      .BLKW #256
      stackend: .BLKW #1 ; adresse du fond de la pile
      .END

```


Annexe A

Documentation LOGISIM

A.1 Référence

Le logiciel que nous utilisons en TP pour les circuits est disponible sur la page :

<http://www.cburch.com/logisim/index.html>

C'est un outil à vocation pédagogique, qui permet de dessiner et de simuler des circuits logiques simples.

A.2 Bibliothèque de composants

Page suivante, on fournit une copie d'écran des composants disponibles dans la bibliothèque LOGISIM.

Library Reference












Wiring library
Gates library
Plexers library
Arithmetic library
Memory library
Input/Output library
Base library

Library Reference





A Logisim library holds a set of *tools* that allow you to interact with a circuit via clicking and dragging the mouse in the canvas area. Most often, a tool is intended for adding components of a particular type into a circuit; but some of the most important tools, such as the Poke Tool and the Select Tool, allow you to interact with components in other ways.

All of the tools included in Logisim's built-in libraries are documented in this reference material.






Wiring library

-  [Splitter](#)
-  [Pin](#)
-  [Probe](#)
-  [Tunnel](#)
-  [Pull Resistor](#)
-  [Clock](#)
-  [Constant](#)
-  [Power/Ground](#)
-  [Transistor](#)
-  [Transmission Gate](#)
-  [Bit Extender](#)

Gates library









-  [NOT Gate](#)
-  [Buffer](#)
-  [AND/OR/NAND/NOR Gate](#)
-  [XOR/XNOR/Odd Parity/Even Parity Gate](#)
-  [Controlled Buffer/Inverter](#)

Plexers library








-  [Multiplexer](#)
-  [Demultiplexer](#)
-  [Decoder](#)
-  [Priority Encoder](#)
-  [Bit Selector](#)

Arithmetic library








-  [Adder](#)

-  [Subtractor](#)
-  [Multiplier](#)
-  [Divider](#)
-  [Negator](#)
-  [Comparator](#)
-  [Shifter](#)
-  [Bit Adder](#)
-  [Bit Finder](#)






Memory library

-  [D/T/J-K/S-R Flip-Flop](#)
-  [Register](#)
-  [Counter](#)
-  [Shift Register](#)
-  [Random](#)
-  [RAM](#)
-  [ROM](#)

Input/Output library

-  [Button](#)
-  [Joystick](#)
-  [Keyboard](#)
-  [LED](#)
-  [7-Segment Display](#)
-  [Hex Digit Display](#)
-  [LED Matrix](#)
-  [TTY](#)

Base library

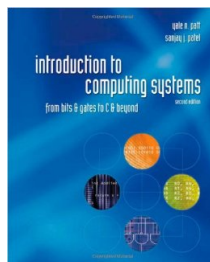
-  [Poke Tool](#)
-  [Edit Tool](#)
-  [Select Tool](#)
-  [Wiring Tool](#)
-  [Text Tool](#)
-  [Menu Tool](#)
-  [Label](#)

Annexe B

Documentation LC-3

B.1 Référence Bibliographique

Le LC-3 est un processeur développé dans un but pédagogique par Yale N. Patt et J. Patel dans [Introduction to Computing Systems : From Bits and Gates to C and Beyond, McGraw-Hill, 2004].



Des sources et exécutables sont disponibles à l'adresse : <http://highered.mcgraw-hill.com/sites/0072467509/>

B.2 Codage des instructions

Tableau récapitulatif des instructions LC-3 utilisées en TP :

syntaxe	action	NZP	codage																
			opcode				arguments												
			F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	
NOT DR,SR	DR ← not SR	*	1	0	0	1	DR		SR		1 1 1 1 1 1								
ADD DR,SR1,SR2	DR ← SR1 + SR2	*	0 0 0 1				DR		SR1		0	0 0		SR2					
ADD DR,SR1,Imm5	DR ← SR1 + SEXT(Imm5)	*	0 0 0 1				DR		SR1		1	Imm5							
AND DR,SR1,SR2	DR ← SR1 and SR2	*	0 1 0 1				DR		SR1		0	0 0		SR2					
AND DR,SR1,Imm5	DR ← SR1 and SEXT(Imm5)	*	0 1 0 1				DR		SR1		1	Imm5							
LEA DR,label	DR ← PC + SEXT(PCOffset9)	*	1 1 1 0				DR		PCOffset9										
LD DR,label	DR ← mem[PC + SEXT(PCOffset9)]	*	0 0 1 0				DR		PCOffset9										
ST SR,label	mem[PC + SEXT(PCOffset9)] ← SR		0 0 1 1				SR		PCOffset9										
LDR DR,BaseR,Offset6	DR ← mem[BaseR + SEXT(Offset6)]	*	0 1 1 0				DR		BaseR		Offset6								
STR SR,BaseR,Offset6	mem[BaseR + SEXT(Offset6)] ← SR		0 1 1 1				SR		BaseR		Offset6								
BR[n][z][p] label	Si (cond) PC ← PC + SEXT(PCOffset9)		0 0 0 0				n	z	p	PCOffset9									
NOP	No Operation		0 0 0 0				0	0	0	0 0 0 0 0 0 0 0									
RET (JMP R7)	PC ← R7		1 1 0 0				0 0 0		1 1 1		0 0 0 0 0 0								
JSR label	R7 ← PC; PC ← PC + SEXT(PCOffset11)		0 1 0 0				1	PCOffset11											

B.3 ISA LC-3

La documentation "officielle" est accessible via la page du cours, ou encore à l'adresse :

<http://highered.mheducation.com/sites/dl/free/0072467509/104653/PattPatelAppA.pdf>