



Architecture Matérielle et Logicielle (LIF6)

Cahier de TP, printemps 2016

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	4
2	Circuits combinatoires et complément à deux	5
2.1	Circuits combinatoires	5
2.2	Quelques éléments du processeur LC-3	5
2.3	Dépassements en C	6
3	Circuits séquentiels, registres, mémoire	7
3.1	Bascules	7
3.2	Registres, compteurs et mémoire	7
3.3	Banc de registre	8
4	Test de circuit, circuits dédiés.	9
4.1	LOGISIM : test de circuit	9
4.2	Circuits dédiés	9
5	Introduction à l'archi LC-3, jouons avec le simulateur	11
5.1	Jouons avec le simulateur de LC-3	11
5.2	Écriture et simulation de programmes en assembleur LC-3	13
6	LC-3, Exercices de programmation	14
6.1	Chaînes de caractères	14
6.2	Un message codé (CC-TP 2015)	14
6.3	Saisie d'un entier au clavier	15
7	Construisons le LC-3 - partie 1	16
7.1	Le circuit LC-3	16
7.2	L'unité arithmétique et logique	17
7.3	Exécution des instructions arithmétiques	18
7.4	Pour la suite...	19
8	Construisons le LC-3 - partie 2	20
8.1	Décodage des instructions	20
8.2	Instructions d'accès mémoire	20
8.3	Instructions de branchement et saut	21
A	Documentation LOGISIM	22
A.1	Référence	22
A.2	Bibliothèque de composants	22
B	Documentation LC-3	24
B.1	Référence Bibliographique	24
B.2	Codage des instructions	24
B.3	ISA LC-3	24

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

EXERCICE 4 ► Additionneurs à retenue

Dans un nouveau fichier :

- Réalisez l'additionneur 1 bit à retenue du cours et testez-le.
- 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'avez pas le droit de dupliquer l'additionneur. Vous pouvez vous reporter à l'exercice correspondant du cahier de TD.

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

- (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.

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 $(s_1 s_1 s_0)_2 = i$. 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.

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_aluetu.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)		

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.

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 16 bits, sélectionner les 6 bits de poids faibles, et réaliser l'extension vers un entier 16 bits. On pourra utiliser le composant `BitExtender` de la librairie (dans `Wiring`).

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

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;
  }
```

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`.*
- 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.

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 ?*

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 ?

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.

“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)

- 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.

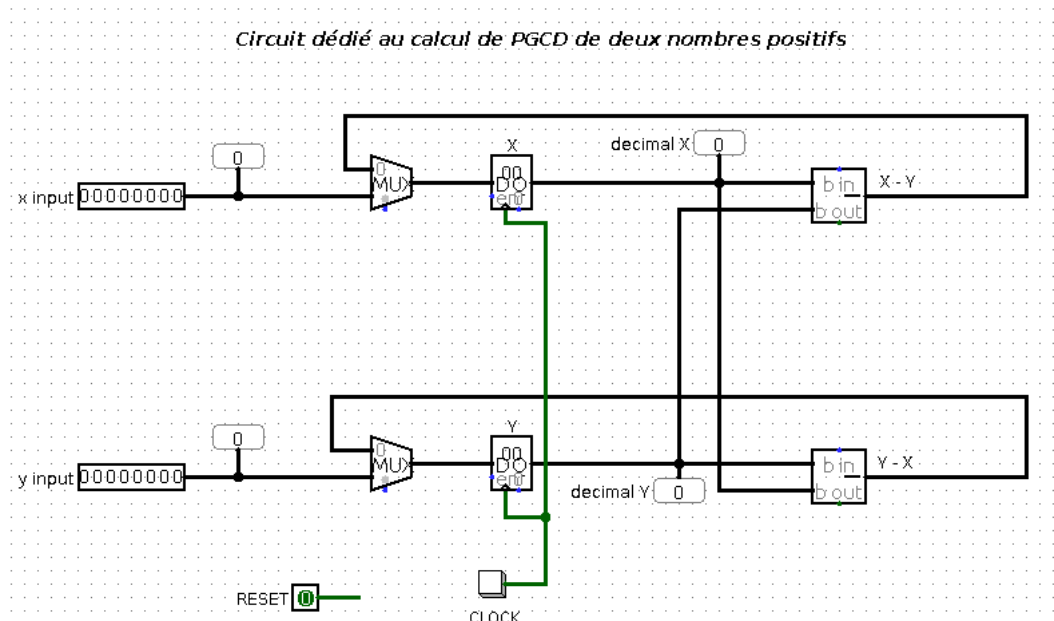


FIGURE 4.2 – Le circuit fourni

- 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!*
- Rajoutez les composants de contrôle : quand les entrées sont disponibles, presser le bouton `reset` doit initialiser les registres X et Y avec ces valeurs (au premier appui sur `Clock`). Ensuite, chaque appui le 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 ?

EXERCICE 4 ► Un automate reconnaisseur

Construire en LOGISIM un automate séquentiel reconnaissant le motif 111 (en séquence, c'est à dire que l'automate sort 1 dès qu'il a lu trois '1' de suite, et continue à sortir 1 tant que les 3 dernières entrées lues sont des '1', ...). 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 et les sorties sont calculées à partir de l'état courant et de l'entrée*
- 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.

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
```

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

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 <i>loop</i>
fin:x3007	xF025	1111 0000 0010 0101	TRAP trapvect=x25	HALT
x3008	x0006	donnée	-	

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

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.

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

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 5.4 – `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    ;
5    PUTS           ;
    LEA R0,COURSE   ;
    PUTS           ;
    HALT           ;
HELLO: .STRINGZ "Hello, world!\n"
10 COURSE: .STRINGZ "LIF6\n"
    .END

```

Listing 5.5 – `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

```

2. 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.

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 complétera progressivement le programme ci-dessous.

Listing 6.1 – 'SaisieChaine.asm'

```
.ORIG x3000
; Programme principal
LEA R6,stackend ; initialisation du pointeur de pile
; *** A COMPLETER ***
5 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.

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 $k \oplus c$ (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 $R1 \wedge R3$, 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.

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 ?

TP 7

Construisons le LC-3 - partie 1

Objectifs : Le but de ce TP est de découvrir un circuit LOGISIM qui implémente une partie des instructions du LC-3 et que vous aurez à modifier dans le TP suivant. Le TP s'inspire fortement des TP et projet de l'équipe pédagogique d'architecture de Paris 7 : <http://www.liafa.jussieu.fr/~carton/Enseignement/Architecture/>, dont il reprend une partie du code source fourni aimablement par leurs auteurs.

Fichiers fournis : LC3_etu.circ, AddSimple.mem

7.1 Le circuit LC-3

Récupérez le fichier LC3_etu.circ (Figure 7.1) et ouvrez-le avec LOGISIM. Ce fichier contient une implémentation partielle du LC-3, dans laquelle seules les instructions ADD, AND, NOT sont cablées, même si beaucoup d'infrastructure est déjà en place pour plus tard.

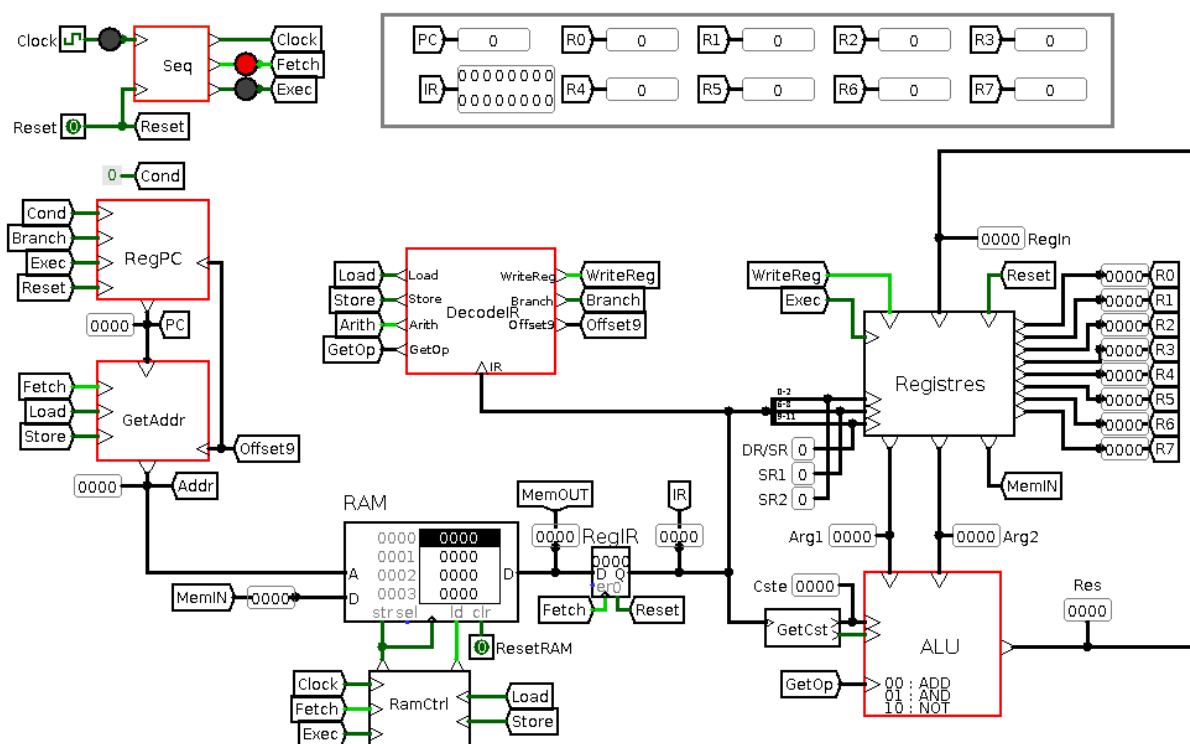


FIGURE 7.1 – Le circuit LC-3 fourni, onglet principal

Bref mode d'emploi. Pour simuler l'exécution de programmes LC-3 avec ce circuit :

- Entrez dans la RAM les octets d'un programme LC-3, d'une des trois manières suivantes :
 - En mode "Poke", on peut directement cliquer sur une case de la RAM et la remplir au clavier.
 - En faisant un clic-droit sur la RAM, on peut choisir "Edit Contents", ce qui ouvre un éditeur.
 - Le menu obtenu par un clic-droit sur la RAM propose aussi "Load Image" pour charger dans la RAM le contenu d'un fichier mémoire (voir ci-dessous).

- Pour faire avancer la simulation d'un programme, il faut changer l'état de l'horloge (Clock) du circuit (c'est-à-dire lui faire passer un front montant ou un front descendant) : vous pouvez soit cliquer sur le bouton d'horloge (en haut à gauche du circuit), soit envoyer un tic manuel *via* Ctrl-T.
- Si vous souhaitez relancer l'exécution du programme (remettre PC à 0, et vider le banc de registres), mettez l'horloge Clock à son niveau bas, puis faites passer l'entrée Reset à 1, puis de nouveau à 0.

Fichier mémoire. Un fichier mémoire chargeable dans une RAM LOGISIM est un simple fichier texte dont la première ligne est `v2.0 raw`. Viennent ensuite les différents octets de la mémoire en hexadécimal. Voir la section "Memory Components" de la documentation LOGISIM pour plus de détails. Voici par exemple le contenu du fichier `AddSimple.mem` :

```
v2.0 raw
5020 1025 5260 1266 1440
```

Ce programme correspond au code assembleur contenu dans le fichier `AddSimple.asm` :

Listing 7.1 – 'AddSimple.asm'

```

1      .ORIG x0000                                ; code hexa
2      AND R0,R0,0                                ; 5020
3      ADD R0,R0,5      ; R0 <- 5                ; 1025
4      AND R1,R1,0                                ; 5260
5      ADD R1,R1,6      ; R1 <- 6                ; 1266
6      ADD R2,R1,R0      ; R2 <- R1 + R0        ; 1440
7      .END
    
```

EXERCICE 1 ► Simulation d'un programme

Ouvrez le circuit `LC3_etu.circ`, puis récupérez le programme `AddSimple.mem` sur la page du cours. Chargez-le dans la RAM et lancez sa simulation. Observez en particulier l'évolution des contenu des registres (PC, IR, R0, ..., R7).

EXERCICE 2 ► Cycle d'instruction

Placez vous dans le sous-circuit `Seq` (Figure 7.2), et expérimentez. Représenter les signaux `Clock`, `Fetch` et `Exec` sur un chronogramme, en supposant que `ClockOrig` est un signal créneau périodique. Que se passe-t'il quand `Reset` est activé, puis relâché un peu plus tard ? Quel est le rôle joué par ce circuit dans cette implantation du LC-3 ?

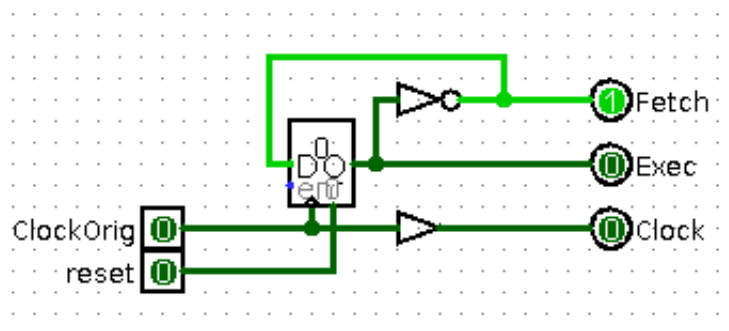


FIGURE 7.2 – Composant Seq du LC-3

7.2 L'unité arithmétique et logique

Placez-vous dans le module ALU (Figure 7.3), qui contient l'unité arithmétique et logique. Notez l'usage d'un multiplexeur, qui sélectionne parmi quatre entrées (dont une actuellement non affectée) en fonction d'un fil de contrôle sur 2 bits.

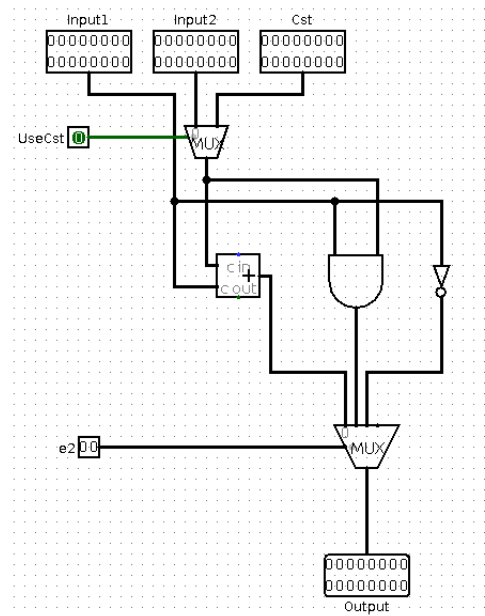


FIGURE 7.3 – Composant ALU du LC-3

EXERCICE 3 ► Valeurs de contrôle

Expérimentez différentes valeurs pour e_1 et e_2 . Quelles valeurs donner à e_1 et e_2 pour :

- obtenir en sortie le ET bit à bit de Input1 et Input2?
- obtenir l'addition de Input1 et Cst
- obtenir le NON bit à bit de Input1 ?

EXERCICE 4 ► Contrôle de l'ALU

Reprenez le jeu d'instructions du LC-3 (cf cours) Les opérandes des instructions arithmétiques sont soit deux registres, soit un registre et une constante littérale.

- Comment sont différenciés les deux cas ?
- Quels bits différencient les opcodes des instructions arithmétiques ADD, AND et NOT? Déduisez-en à quoi devront être branchés e_1 et les deux bits de e_2 dans le circuit complet.

7.3 Exécution des instructions arithmétiques

On suppose que vous avez ouvert le circuit `LC3_etu.circ`, puis chargé le programme `AddSimple.mem` dans la RAM : servez vous de la simulation de ce programme pour répondre aux questions suivantes.

EXERCICE 5 ► Phase Fetch

1. En observant le module `RegPC`, expliquez comment est calculé $PC + 1$. A quel instant $PC + 1$ remplace PC ?
2. Lors du cycle du chargement, comment une instruction est-elle chargée dans le registre `RegIR` ?
3. Pour l'instant, on n'exécute que des instructions arithmétiques : expliquez les valeurs de sorties produites par `DecodeIR`.

EXERCICE 6 ► Phase Exec

1. Observez le fonctionnement du module `Registres` : pourquoi utiliser trois ports de lecture ? Dans quel cas sera utilisé le port de lecture `OUT (SR)` ?

- En vous plaçant dans la phase exec de l’instruction `ADD R0,R0,5` (de code `x1025` en mémoire), expliquez comment est exécutée cette instruction. Comment les opérandes sources sont amenées à l’ALU ? Comment le résultat de l’opération est rangé dans le registre destination ? A quel instant le résultat est définitivement stocké ?

EXERCICE 7 ► Chronogramme

En supposant que l’horloge générale `Clock` du circuit est un signal créneau périodique, représenter l’évolution de l’état du circuit sur un chronogramme au cours des cycles des deux premières instructions du programme `AddSimple.mem`. Vous représenterez les signaux suivants : `Clock`, `Fetch`, `Exec` ; les valeurs suivantes : `PC`, `IR`, `GetOp`, `R0`, `R1` sur la Figure 7.4.

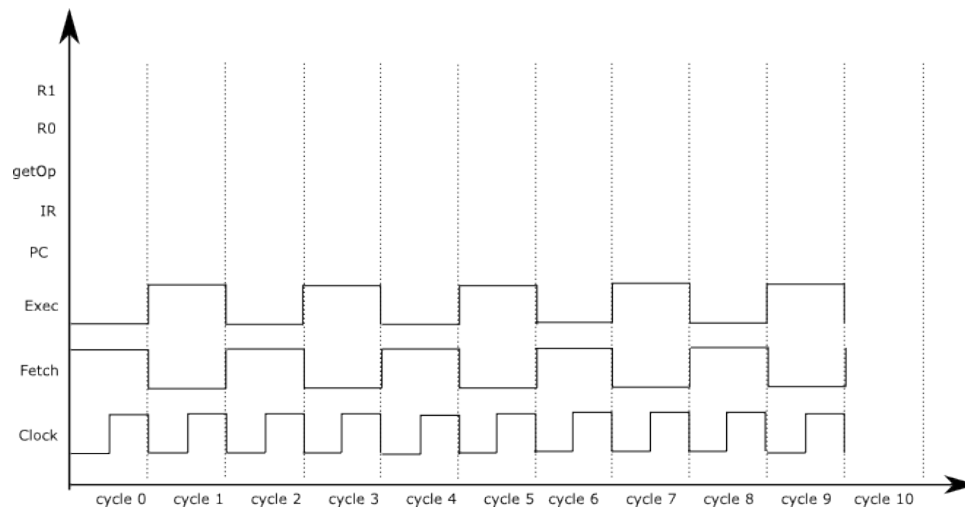


FIGURE 7.4 – Chronogramme pour `AddSimple`

7.4 Pour la suite...

EXERCICE 8 ► Un programme

Préparez un fichier `Cst2007.mem` contenant les instructions en langage machine pour charger 2007 dans le registre `R1`. Pour cela utilisez soit un éditeur de texte séparé, soit l’éditeur hexadécimal intégré puis l’entrée de menu “Save Image”. Simulez ce programme : que constatez-vous ?

EXERCICE 9 ► Opcode LC-3

Construisez la table de vérité de toutes les instructions du LC-3 en fonction des 4 bits de l’opcode (cette table sera utile plus tard). Que pouvez-vous observer ? Comment caractériser les instructions qui peuvent modifier des registres `R0`, ..., `R7` ?

TP 8

Construisons le LC-3 - partie 2

Objectifs : Le but de ce TP est de terminer le circuit LOGISIM qui implémente une partie des instructions du LC-3. Le circuit fourni au TP précédent implante déjà les instructions NOT, ADD, AND du LC-3. Il reste à lui ajouter des composants pour qu'il implante d'autres instructions :

- deux instructions d'accès mémoire, LD et ST ;
- une instruction de branchement, BR.

Ce TP s'inspire fortement des TP et projet de l'équipe pédagogique d'architecture de Paris 7 : <http://www.liafa.jussieu.fr/~carton/Enseignement/Architecture/>.

8.1 Décodage des instructions

À l'issue de ce TP, la version simplifiée du processeur que nous allons construire permettra d'exécuter les instructions arithmétiques (ADD, AND, NOT), deux instructions d'accès direct à la mémoire (LD, ST) et une instruction de branchement (BR).

EXERCICE 1 ► **Circuit DecodeIR**

Les signaux de sortie du sous-circuit DecodeIR (Load, Store, Arith, GetOp, WriteEnable, Branch et Offset9) permettent d'activer correctement le chemin de données en fonction de la classe d'instruction à exécuter. Déterminez en fonction de IR[13, 12] (bits 13 et 12 du registre d'instruction) comment doivent être activés ces signaux. Complétez le sous-circuit DecodeIR. *N'oubliez pas Offset9!*

8.2 Instructions d'accès mémoire

Dans cette section on va ajouter les composants du circuit pour les instructions LD et ST.

EXERCICE 2 ► **Programme de test**

Écrire en langage machine LC-3 un programme *de taille minimale* pour tester les instructions LD et ST : ce programme chargera deux entiers depuis les adresses (5)₁₀ et (6)₁₀ dans deux registres, puis rangera leur somme à l'adresse (6)₁₀. Vous utiliserez judicieusement PennSim pour traduire votre programme en langage machine chargeable dans la RAM du LC-3 de LOGISIM¹

EXERCICE 3 ► **Implantation de LD**

Dans cet exercice, on se concentre sur l'implantation de l'instruction LD.

1. Quelle action doit être effectuée dans le chemin de données du processeur lors de la phase d'exécution d'une instruction LD DR, label ?
2. Lors de l'exécution d'une instruction LD, quel sous-circuit est chargé de placer la mémoire RAM en mode lecture ?
3. L'unité GetAddr se charge de calculer l'adresse de la mémoire qui doit être accédée Addr dans la RAM. Au cours de la phase de Fetch, que doit valoir Addr ? Même question au cours de la phase Exec d'une instruction LD.
4. Complétez à l'aide des signaux Offset9, PC, Load, Store, Fetch la formule valable pour l'exécution de toutes les instructions :

1. Au passage, pourquoi peut-on utiliser ici le codage fourni par PennSim, qui stocke le programme à partir de l'adresse x3000, alors que notre circuit stocke le programme à partir de l'adresse x0000 ?

```

Si .....
Alors Addr = PC
Sinon Addr = .....
    
```

5. Complétez GetAddr de façon à ce que le circuit permette l'exécution de LD.
6. Au niveau du banc de registres, comment doit-êtré effectuée l'écriture dans DR de la donnée ? *On vérifiera qu'il n'y a rien à mettre à jour à l'intérieur du composant Registers, mais par contre des modifications sont à faire dans le circuit principal pour bien amener la bonne donnée en entrée.*
7. Expérimentez votre circuit en exécutant les trois premières instructions de votre programme de test.

EXERCICE 4 ► **Implantation de ST**

On se concentre maintenant sur l'implantation de l'instruction ST.

1. Quelle action doit-êtré effectuée dans le chemin de données du processeur lors de la phase d'exécution d'une instruction ST `SR, label` ?
2. Comment la RAM est-elle placée en mode écriture lors de l'exécution d'un ST ?
3. Comment est calculée l'adresse de destination dans la RAM ?
4. Mettre à jour votre circuit au voisinage du banc de registres, si nécessaire.
5. Expérimentez votre circuit en exécutant l'instruction ST de votre programme de test.

8.3 Instructions de branchement et saut

Dans cette section on cherche à rajouter les composants du circuit pour l'instruction BR.

EXERCICE 5 ► **Programme de test**

Écrire en langage machine LC-3 un programme qui permettra de tester BR.

EXERCICE 6 ► **Conditions d'activation - NZP**

On veut implanter le sous-circuit NZP. On rappelle que l'architecture du LC-3 spécifie trois drapeaux N, Z et P, qui indiquent respectivement si la dernière valeur chargée dans le banc de registres était strictement négative, nulle, ou strictement positive. Le circuit NZP contient un registre 3 bits (*falling edge triggered*) qui stocke l'état des drapeaux.

1. Ecrire, en fonction des 16 bits formant l'entrée IN du circuit (valeur à tester), les fonctions logiques donnant les valeurs que doivent prendre N, Z et P.
2. Quand doit-êtré mis à jour le registre 3 bits contenant l'état des drapeaux N, Z et P ?
3. La sortie Cond du sous-circuit prend la valeur 1 si, d'après les drapeaux N, Z et P, le BR en cours d'exécution doit provoquer un saut : donnez une formule pour Cond.
4. Complétez le sous-circuit NZP.
5. Modifiez le chemin de donnée du LC-3, de façon à ce que les drapeaux N, Z et P soient mis à jour : soit d'après MemOUT dans le cas de l'exécution d'une instruction de chargement mémoire (signal Load à 1), soit d'après la sortie Res de l'ALU dans tous les autres cas. *On fera attention au choix signal commandant l'écriture dans le registre (entrée Clock du composant NZP), qui ne peut êtré l'horloge du circuit, pourquoi ?*

EXERCICE 7 ► **Calcul de l'adresse de saut**

Il ne nous reste plus qu'à mettre à jour le calcul de la prochaine valeur du registre PC. Cela se passe dans le sous-circuit RegPC.

1. Quelle action doit effectuer une instruction BR `[n] [z] [p], label` durant la phase Exec ?
2. Complétez le sous-circuit RegPC, puis testez à l'aide de votre programme.
3. Que faudrait-il faire pour prendre en compte l'instruction JSR ?

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



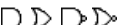
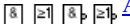
A Logisim library holds a set of *tools* that allow you to interact with a circuit via clicking and dragging the mouse in the canvas area. Most often, a tool is intended for adding components of a particular type into a circuit; but some of the most important tools, such as the Poke Tool and the Select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

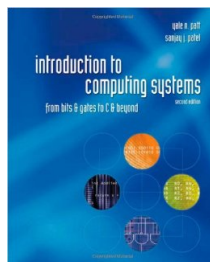
-  [Poke Tool](#)
-  [Edit Tool](#)
-  [Select Tool](#)
-  [Wiring Tool](#)
-  [Text Tool](#)
-  [Menu Tool](#)
-  [Label](#)

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	N/ZP	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>