Architecture des Circuits : Cahier d'Exercices – TP 1 (4h)

Table des matières

Chapitre 1

Circuits combinatoires

Préambule

Au cours de cette séance de travaux pratiques, vous allez utiliser un simulateur (“Logisim”) pour “réaliser” vos premiers circuits numériques. De façon à éviter de perdre trop de temps à “tirer des fils” (ce qui est rapidement un brin rébarbatif), nous limiterons la taille des mots manipulés par nos circuits à 4 bits [1]. Avant d’aller plus loin, vérifiez que vous êtes bien conscients qu’il n’y a pas de différence conceptuelle entre une circuit 4 bits et un circuit 8, 16 ou 32 bits. Si vous n’en êtes pas sûr, n’hésitez pas à demander ... Notez que la largeur du mot manipulé physiquement par votre circuit ne limite pas forcément la taille des mots qu’il peut manipuler ; mais c’est une autre histoire et nous l’aborderons dans une prochaine séance ...

Cette séance se compose de trois parties a peu près équivalentes (attention : le sujet du TP couvre donc trois chapitres) : dans un premier temps, vous allez vous intéresser à des circuits combinatoires simples (nous prendrons comme exemple l’additionneur 4 bits) et vous en profiterez pour vous familiariser avec Logisim. Vous passerez ensuite aux circuits séquentiels (registres 4 bits et mémoire adressable). Enfin, dans la troisième partie, vous réaliserez une ALU (Arithmetic and Logic Unit) 4 bits.

1 Prise en main de Logisim

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

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

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

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

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

QUESTION 1 ► Afin de vous permettre de prendre en main l’outil, suivez le tutoriel suivant, qui vous fait créer un circuit réalisant la fonction `xor` :

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

2 Additionneur 1 bit

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

QUESTION 2 ► Votre premier circuit dans ce projet réalisera l’addition 1 bit. Ce circuit possède 3 entrées `a`, `b` et `c_in` représentant respectivement les deux valeurs à additionner et la retenue entrante de l’addition. Il possède 1 sortie `s` dénotant la valeur de la somme ainsi que 1 sortie `c_out` dénotant la valeur de la retenue de sortie.
On vous rappelle la définition de `s` et `c_out` :

1. Ce qui ne nous fait pas remonter à un passé si lointain : les processeurs 4 bits étaient encore utilisés il n’y a “pas si longtemps” (le 4004 de Intel a été fabriqué jusqu’au début des années 80).

```
s = (a xor b) xor c_in
c_out = (a and b) or (a and c_in) or (b and c_in)
```

Attention à ce que la simulation soit bien activée ! Dans le menu "Simulate", choisissez "Simulation enabled" !
Vous aurez remarqué que, par défaut, les portes de votre circuit ont plus d'entrées que ce que vous utilisez. Logisim permet de changer ces paramètres dans le "cadre des attributs", en bas à gauche de la fenêtre. Ce cadre permet notamment de nommer les entrées (attribut "label") de changer l'orientation des portes (attribut "facing") et leur nombre d'entrées (attribut "number of inputs").
Au delà de l'exécution pas-à-pas, dirigée par le changement des valeurs d'entrées du circuit, que vous avez utilisée dans le tutoriel ci-dessus, Logisim offre un certain nombre d'outils pour valider le comportement de vos petites créations. Ceux-ci sont accessibles par[2] `Project > Analyze Circuit`.
Logisim vous propose notamment, pour le circuit en cours d'édition :

— La liste de ses entrées et sorties ;

— Sa table de vérité ;

— Une version textuelle de l'expression booléenne réalisée pour chacune des sorties.

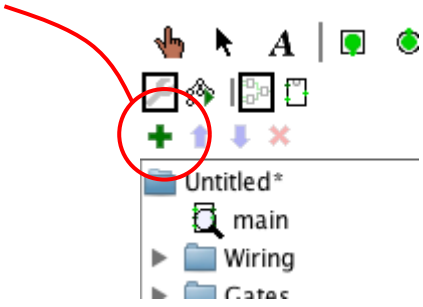
3 Additionneur 4 bits

Pour construire un additionneur 4 bits, vous n'allez pas (!) copier-coller 4 fois les portes que vous venez d'utiliser pour l'additionneur 1 bit. Plutôt, vous allez encapsuler votre additionneur 1 bit dans un composant que vous pourrez ensuite réutiliser 4 fois pour obtenir l'additionneur 4 bits.

QUESTION 3 ► Commencez par suivre le tutoriel mis en place par les auteurs de Logisim, ici :

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

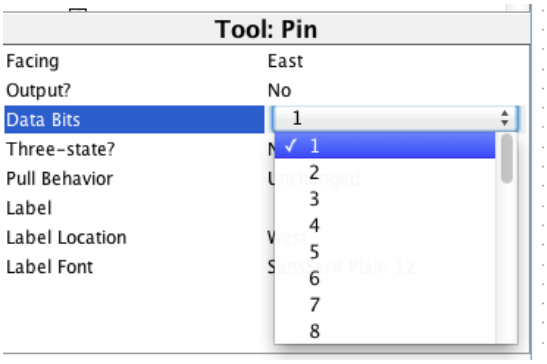
Concentrez-vous sur les parties intitulées **Creating circuits**, **Using subcircuits** ainsi que **Editing subcircuit appearance**. Dans cette dernière section, vous apprendrez à modifier la forme des composants que vous créez et *surtout* à nommer les ports d'entrée/sortie de vos composants. Nommez les ports de votre additionneur, par exemple `a`, `b` et `c_in` pour les entrées et `s` et `c_out`. Changez ensuite le nom de votre circuit (jusqu'ici `main`) pour, par exemple, `add1bit`. Enfin, créez un nouveau circuit dans le même projet en utilisant la petite croix verte ici :



Changez son nom pour `add4bits`.

QUESTION 4 ►
Créez maintenant votre additionneur 4 bits en utilisant l'additionneur 1 bit de la section précédente.
Pour le tester, vous pouvez vous servir de "**pin ports**" d'entrée/sortie d'une taille plus grande que celle par défaut (1 bit). Pour cela, sélectionnez l'outil "pin port" depuis la barre d'outils. Puis, dans l'encart en bas à gauche de la fenêtre "**Tool : Pin**" sélectionnez 4 pour le champ "**data bits**" (voir figure ci-dessous).

2. voir `http://www.cburch.com/logisim/docs/2.7/en/html/guide/analyze/index.html`



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

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

Testez votre circuit en réalisant plusieurs additions. Vous prendrez bien garde à l’initialisation de la retenue entrante de votre additionneur.

Chapitre 2

Registres et mémoires

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

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

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

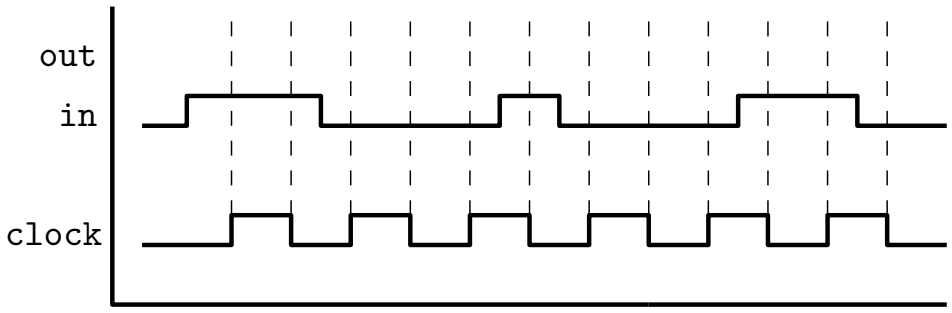
1 Registres

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

1.1 Latch et flip-flop

QUESTION 5 ► Construisez un verrou (latch ainsi qu'un registre 1 bit (flip-flop) dans Logisim.
Dans le poly du cours (chapitre 4.2) ainsi que dans les transparents, vous trouverez une description de ces deux composants. Vous trouverez le multiplexeur (aka "mux") dans la bibliothèque de Logisim (section "Plexers").

QUESTION 6 ► Remplissez le chronogramme suivant pour vous assurer que vous comprenez le comportement du flip-flop.



1.2 Flip-flop avec reset

QUESTION 7 ► Complétez votre flip-flop afin de permettre sa remise à zéro.

Pour cela, créez un nouveau circuit qui utilise votre flip-flop précédent et possède une entrée supplémentaire *reset*[1]

1.3 Registre à commande de chargement

QUESTION 8 ► Ajoutez une commande de chargement à votre registre.

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

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

1.4 Registre à 4 bits

On ne sait pour l'instant que mémoriser 1 bit. Pour aller plus loin, il va nous falloir de quoi mémoriser des paquets de bits. En pratique, la taille des registres est très liée à d'autres éléments de l'architecture concernée : taille des bus, taille de la mémoire. Pour ce TP, vous implémenterez donc des registres 4 bits. Contrairement aux différents verrous et flip-flop que nous avons construits jusque-là, la complexité ne se situe pas dans le comportement temporel mais dans le nombre des boîtes et des connexions constituant le circuit (d'où la limite à 4 bits ;).

QUESTION 9 ► Construisez un registre 4 bits en collant côte-à-côte 4 registres 1 bit. Veillez à la synchronisation de l'ensemble. Vérifiez que vous avez bien compris que votre circuit aura 7 entrées et 4 sorties ...

2 Mémoires adressables

Lorsqu'on veut stocker beaucoup de données en même temps, utiliser des registres indépendants implique une trop grande complexité d'interconnexion (i.e. il y a trop de filasse). On invente alors (tadam) des mémoires adressables.
Dans cette partie, vous allez implémenter une telle mémoire adressable. Elle sera petite (4 mots de 4 bits), mais vous constaterez également qu'étendre ce composant à des capacités plus grandes est simple.

ATTENTION : Si à ce stade du TP vous êtes en retard, sautez les sections 2.1 et 2.2 pour passer directement à la section 2.3. Dans ce cas, utilisez les multiplexeurs et démultiplexeurs de la bibliothèque Logisim mais prenez tout de même le temps, hors séance, de faire ces deux sections ...

2.1 Réalisation d'un démux 1 vers 4 de 1 bit

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

QUESTION 10 ► Réalisez un démultiplexeur 1 vers 4 à 1 bits.

2.2 Réalisation d'un mux 4 vers 1 de 4 bits

Cet aiguillage se fait aussi dans l'autre sens, pour pouvoir envoyer une des *n* entrées vers l'unique sortie.

QUESTION 11 ► Réalisez ce composant *multiplexeur n vers 1*, avec $n = 4$, pour des entrées booléennes (de 1 bit chacune) et une sortie booléenne.

1. Nota : il existe deux types de reset : le reset *synchone*, qui met le contenu du registre à zéro sur un front d'horloge, et le reset *asynchrone* qui met le registre à zéro au moment de l'activation du reset (quel que soit l'état de l'horloge à ce moment là). Ici on se contentera du reset synchrone mais en pratique, la plupart des registres (y compris le registre de la bibliothèque de Logisim) sont des registres asynchrones.

Cet aiguillage manipule des fils (entrées et sortie) de taille 1, on parle de multiplexeur 4 vers 1 à 1 bit. On peut le généraliser pour que les informations à sélectionner soient de taille quelconque. Ça se fait de manière hiérarchique, par exemple en encapsulant des multiplexeurs de m bits pour faire un multiplexeur de $2m$ bits.

QUESTION 12 ► Réalisez un multiplexeur 4 vers 1 à 2 bits. Puis à 4 bits.

2.3 Construction d'une mémoire de 4 mots de 4 bits

QUESTION 13 ► Concevez maintenant une mémoire de 4 mots de 4 bits.

Vous avez tous les composants nécessaires. Elle sera d'une taille considérable : 16 bits en tout (2 octets) ! Pour vous aider un peu, elle aura les entrées suivantes :

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

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

Chapitre 3

Une ALU 4 bits

Rappel : Si vous ne savez pas ce qu'est une ALU (ou une UAL) commencez par vous poser la question ...

Notre ALU fera deux opérations arithmétiques à deux opérandes (ADD, SUB) et cinq opérations logiques à un ou deux opérandes (AND, OR, XOR, NOT, LSR). Elle écrira 3 flags en sortie : Z, C, N.

QUESTION 14 ► C'est quoi un flag ?

QUESTION 15 ► À votre avis, pourquoi notre ALU implémente-t-elle le LSR ("Logical Shift Right") mais pas le LSL ("Logical Shift Left"). Quelle est la différence entre un LSR et un ASR ("Arithmetic Shift Right") ?

1 Opérations arithmétiques : Additionneur / Soustracteur 4 bits

Comme brillamment décrit dans le poly du cours, on peut réutiliser l'additionneur pour créer un circuit additionneur/soustracteur 4 bits. Les modifications nécessaires se basent sur le calcul du complément binaire d'un des deux paramètres de la soustraction. Pour deux entiers dont on veut calculer la différence $A - B$, on calcule le complément du second, $\overline{B}$, chaque bit de $\overline{B}$ étant défini par la négation du bit correspondant dans $\overline{B}$. Puis, on calcule l'addition $A + \overline{B} + 1$.

Pour réaliser cela à partir de votre additionneur 4 bits, vous utiliserez la retenue entrante, qui permettra maintenant de décider si on souhaite une addition (elle vaut alors 0) ou une soustraction (elle vaut alors 1). Cette entrée conditionne la complémentation de la deuxième entrée de l'opération, ainsi que la valeur de sa retenue d'entrée. Le choix entre B et $\overline{B}$ se fait à l'aide d'un `XOR` pour chaque bit de B, prenant comme première entrée B et comme seconde entrée la retenue entrante de votre opération.

Vérifiez que vous avez bien compris les deux paragraphes précédents ... Sinon demandez !

QUESTION 16 ► Réalisez un additionneur / soustracteur 4 bits en suivant les indications précédentes. Attention : n'oubliez pas la sortie `C_OUT`.

2 Opérations logiques

ATTENTION : Si à ce stade du TP vous êtes en retard, ne réalisez qu'une partie des opérations logiques (a minima une opération unaire et une opération binaire) puis passez directement à l'intégration de l'ALU (section 3). Prenez tout de même le temps, hors séance, de terminer votre ALU ...

2.1 Operations logiques unaires

QUESTION 17 ► implémentez et intégrez deux modules réalisant respectivement un NOT et un LSR sur un mot de 4 bits.

2.2 Opérations logiques binaires

QUESTION 18 ► implémentez et intégrez trois modules réalisant respectivement les opérations logique bit à bit AND, OR et XOR sur des mots de 4 bits.

3 Implémentation d'une ALU 4 bits

Maintenant que vous disposez de modules capables de calculer toutes les opérations arithmétiques et logiques vous allez pouvoir passer à la réalisation de votre ALU. Celle-ci recevra en entrée deux mots de 4 bits (les opérandes A et B) ainsi que 4 bits de commande (permettant de sélectionner l'opération à effectuer). Elle fournira en sortie S, un mot de 4 bit (résultat de l'opération), ainsi que trois drapeaux ("flags") de 1 bits indiquant si le résultat est nul (flag Z), négatif (flag N) ou si le calcul a généré une retenue (flag C). Attention, si les flags Z et N ont un sens pour toutes les opérations de l'ALU, ce n'est pas le cas du flag C. Vous mettrez donc C à zéro lorsque l'opération en cours ne peut lui donner de valeur. Dans le cas particulier de l'opération LSR, le flag C contiendra le bit sorti à droite.

TABLE 3.1 – Encodage des différentes opérations possibles

CodeOp	Opération
0000	`S = A + B`
0001	`S = A - B`
0010	`S = A AND B`
0011	`S = A OR B`
0100	`S = A XOR B`
0101	`S = LSR A`
1000	`S = NOT A`
1001	`S = B`

QUESTION 19 ► Intégrer vos sept (+1 ;) opérations dans une ALU ; testez-là. Vérifiez en particulier que vos flags sont bien positionnés ...

QUESTION 20 ► (facultative) À ce stade, ce n'est pas beaucoup plus compliqué de passer sur 8 bits ... il suffit, en gros de doubler tous les modules ...

QUESTION 21 ► (facultative) Là c'est un peu plus compliqué ... en doublant la plupart des modules mais en en triplant quelque-uns, réaliser une ALU 8 bits plus rapide que la précédente ...