ARC TD2

Logique séquentiel, mémoire
(1 séance sur machine)

Construit à partir du poly de TD-TP des cours IF-AC et IF-AO

1 Registres et mémoires

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

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

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

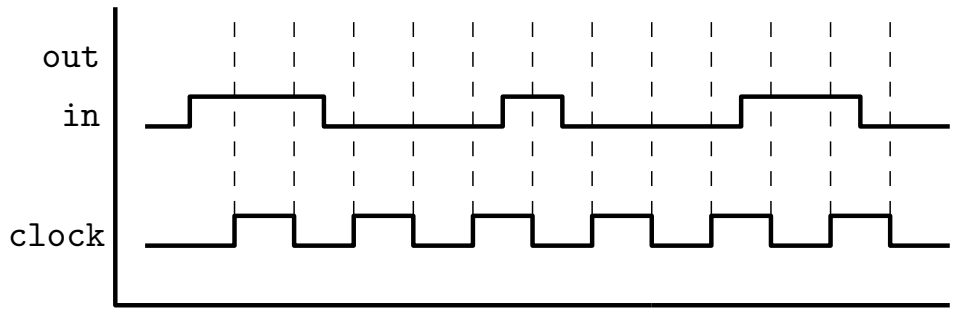
1.1 Registres

NB : Créez chacun des circuits demandés séparément dans Logisim, notamment pour pouvoir facilement y revenir plus tard. Chaque cicuit logisim (appelé *library* peut être utilisé dans un autre circuit en l'important (*Project->load library->logisim library*)

1.1.1 Latch et flip-flop

QUESTION 1 ▶ Construisez un verrou (latch ainsi qu'un registre 1 bit (flip-flop) dans Logisim.
Dans le poly du cours (chapitre 4.2, p41), vous trouverez une description de ces deux composants. Vous trouverez le mux dans la bibliothèque de portes de Logisim, attention, il faut configurer le mux *sans* signal enable.

QUESTION 2 ▶ Remplissez le chronogramme suivant pour vous assurer que vous comprenez le comportement du flip-flop.



1.1.2 Flip-flop avec reset

QUESTION 3 ▶ Complétez votre flip-flop afin de permettre sa remise à zéro.

Pour cela, créez un composante *flipflop* avec votre circuit, et créez un nouveau circuit qui utilise votre flip-flop et possède une entrée supplémentaire *reset*. Vous trouverez dans les composants *wiring* un composant *constant* utile pour la constante 0 du reset.

1.1.3 Registre à commande de chargement

QUESTION 4 ▶ 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*.

Autre Remarque Importante : A partir de maintenant nous allons utiliser les registres 1 bits disponibles dans l'onglet 'Memory→Register' de logisim (il bug aleatoire de l'outil se produit lorsqu'on utilise les registre fait à la main... mais normalement ce sont les mêmes !). Donc **utilisez les registre logisim pour la suite du TP**, ils ont les mêmes ports d'entrée : `Data` (entrée), `enable`, `Clock`, `Clear` (reset) et `Output`.

1.1.4 Registre à n bits

On ne sait pour l'instant que mémoriser 1 bit. Pour aller plus loin, il va nous falloir de quoi mémoriser des paquets de bits. En pratique, la taille des registres est très liée à d'autres éléments de l'architecture concernée : taille des bus, taille de la mémoire. Pour ce TP, nous nous arrêterons à des registres 8 bits. Contrairement aux 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.

QUESTION 5 ▶ Construisez un registre 8 bits en collant côte-à-côte 8 registres 1 bit (à commande de chargement). Veillez à la synchronisation de l'ensemble.

1.1.5 Un petit compteur

Nous allons maintenant concevoir un circuit qui va nous permettre de compter. La valeur du compteur sera enregistrée dans un registre 8 bits. À chaque coup d'horloge, les compteur sera incrémenté de 1. Nous compterons donc de la valeur 0 à la valeur $2^8 - 1$.

QUESTION 6 ▶ Construisez ce compteur en utilisant le registre que vous venez de construire ainsi que l'additionneur 8 bits réalisé en TD1. Utilisé le composant *Clock* du groupe *wiring* pour l'entrée de l'horloge.

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

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

1.3 Observation du comportement d'une mémoire

A la fin de la séance, vous aurez construit une mémoire adressable de 8 mots de 8 bits. Ce composant est par ailleurs déjà implémenté dans Logisim.

QUESTION 7 ▶ Testez une RAM de cette taille, en suivant le tutorial sur le site de Logisim :

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

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

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

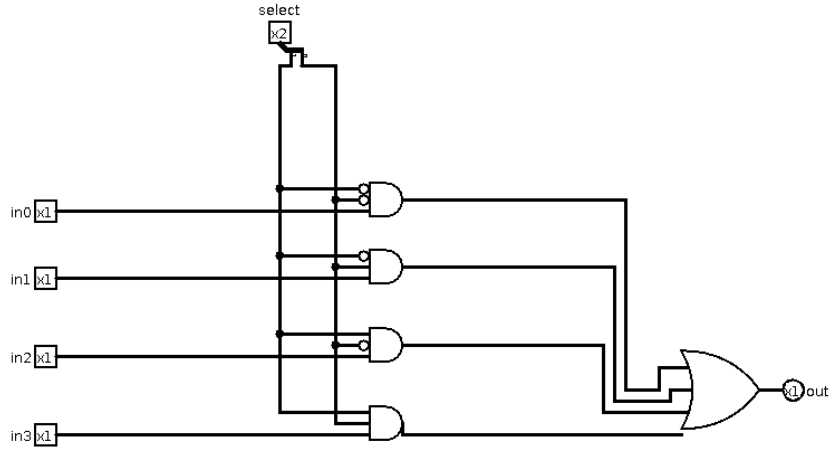
Un dé-multiplexeur est un composant qui réalise un aiguillage. On se donne :

- 1 entrée de *données* composée de x bits, pour l'instant on va supposer une donnée de $x = 1$ bit.
- une *addresse* de k bits.
- n sorties de *données* de x bit (encore une fois ici $x = 1$).

L'adresse indique le numéro de la sortie vers laquelle va être aiguillée la donnée d'entrée, on a donc la relation : $k = \lceil log_2(n) \rceil$. il faut trouver la bonne combinaison de portes logiques ....

QUESTION 8 ► Réalisez un aiguillage d'une donnée de 1 bit vers 8 sortie possible (donc avec une adresse de $2 = log_2(4)$ bits. On peut voir ce circuit comme un circuit d'écriture d'une valeur de 1 bit dans une mémoire de 4 mots de 1 bits. On remarquera que pour une porte logique à plusieurs entrée, on peut rajoute un NOT individuellement sur chacune des entrées si besoin (en éditant les attributs de la porte).

QUESTION 9 ► On peut aussi vouloir réaliser un aiguillage dans l'autre sens, pour pouvoir envoyer une des n entrées vers l'unique sortie. Par exemple avec le cicuit suivant, mux de 4 vers 1 de 1 bit :



1.3.2 Réalisation d'un mux 4 vers 1 de 3 bits

L'aiguillage que l'on a réalisé manipule des fils (entrées et sortie) de taille 1, on parle de multiplexeur 1 vers 4 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 10 ► Réalisez un multiplexeur 1 vers 4 à 2 bits. Puis à 4 bits.
À partir de maintenant, vous utiliserez les composants logosim `Multiplexer` et `Demultiplexer` de la famille `Plexer`, attention, en général on désactivera le `enable` pour ce type de composant (`Include Enable` → `No`. Les signaux enable sont plutôt utilisés pour déclancher le chargement de registre ou de mémoires.

1.3.3 Etude d'une mémoire de 4 mots de 4 bits

Ouvrez le circuit logisim `memory4x4` fournit sur Moodle, étudiez son comportement. QUESTION 11 ► quelle est la taille de cette mémoire ? Ecrivez 1,3,7 et F dans les quatres premières cases de cette mémoire.
QUESTION 12 ► Instanciez un composant `RAM` de la famille `memory` de Logisim, parametrez le pour qu'il ait le même comportement que la mémoire qui vous est fournie.
Faites valider par un enseignant.

2 Conclusion

Les circuits logiques comprenant des registres ou de la mémoire constituent ce que l'on appelle la *logique séquentielle* (par oposition à la logique combinatoire). Le comportement de ces circuit peut vite devenir incompréhensible si ils ne sont pas conçus avec des règles précises.
Dans la suite du cours on va voir des principes fondamentaux de la conceptions de circuits séquentiels : les automates, la séparation du contrôle et du calcul et l'utilisation pour la conception de circuits dédiés et d'un automate particullier qui est le CPU.