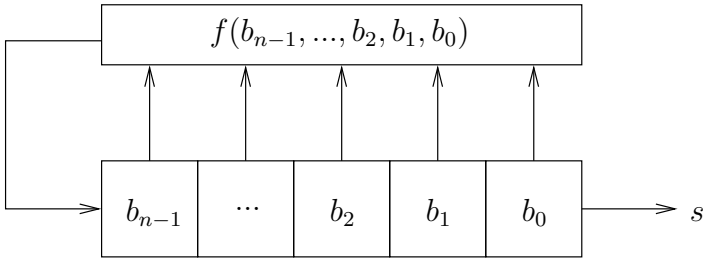
TD 3
Mémoires et macroblocs

Ex. 1 : Générateur de nombres aléatoires

On travaille dans cet exercice sur un composant appelé registre à décalage à rétroaction linéaire (*Linear Feedback Shift Register*) servant à générer des séquences « pseudo-aléatoires » de bits. Le principe de ce composant est détaillé dans la figure ci-dessous.



Chaque case correspond à une bascule D : il y a donc n bascules numérotées de 0 à $n-1$, n étant le nombre de bits du LFSR. La sortie s est le bit pseudo-aléatoire généré par le LFSR : elle affiche la valeur de la bascule numéro 0. A chaque cycle, on charge le contenu de la bascule i dans la bascule $i-1$: si on considère la séquence des valeurs contenues dans les bascules comme une valeur $B = b_{n-1}...b_1b_0$, alors B est décalée d'un bit vers la droite à chaque cycle. La valeur insérée dans la bascule $n-1$ est calculée en fonction des valeurs de chaque bascule via une fonction de rétroaction f définie par $f(b_{n-1}, ..., b_2, b_1, b_0) = c_{n-1}.b_{n-1} \oplus ... \oplus c_1.b_1 \oplus c_0.b_0$ où les c_i sont des constantes binaires données.

Dans la suite de l'exercice, on pose $n = 4$ et $f(b_3, b_2, b_1, b_0) = 0.b_3 \oplus 0.b_2 \oplus 1.b_1 \oplus 1.b_0 = b_1 \oplus b_0$.

Question 1 On suppose que les bascules sont initialisées avec les valeurs suivantes : $b_3 = 1$ et $b_2 = b_1 = b_0 = 0$. Remplir un tableau contenant les valeurs des bascules en fonction du temps jusqu'à ce qu'on retombe sur les valeurs initiales.

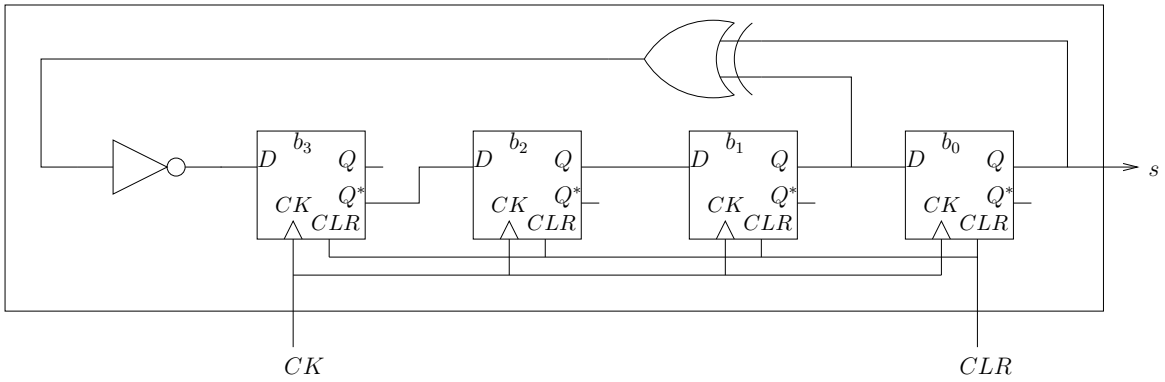
b_3	b_2	b_1	b_0	f
1	0	0	0	0
0	1	0	0	0
0	0	1	0	1
1	0	0	1	1
1	1	0	0	0
0	1	1	0	1
1	0	1	1	0
0	1	0	1	1

b_3	b_2	b_1	b_0	f
1	0	1	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0
0	1	1	1	0
0	0	1	1	0
0	0	0	1	1
1	0	0	0	0

Question 2 Que se passe-t-il si on se retrouve avec $b_3 = b_2 = b_1 = b_0 = 0$? Exprimer le nombre maximum de valeurs différentes possibles avant de retrouver les valeurs initiales en fonction de n.

Il y a $2^n - 1$ valeurs différentes, puisqu'on peut coder 2^n valeurs différentes sur n bits et que le cas $B = 0$ n'est pas possible : si $b_3 = b_2 = b_1 = b_0 = 0$, f telle qu'on l'a choisie dans cet exercice est une fonction constante valant toujours 0.

Question 3 Une réalisation d'un LFSR 4 bits (avec la fonction de rétroaction : $f(b_3, b_2, b_1, b_0) = b_1 \oplus b_0$) à l'aide de 4 bascules D est proposée ci-dessous.

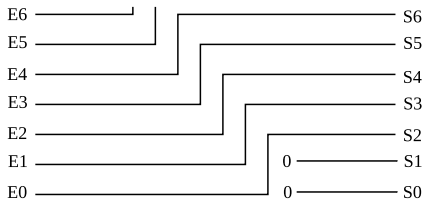


Justifier l'utilisation de l'inverseur et de la connexion à $\overline{Q}$ sur la bascule $b3$.

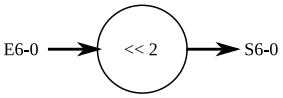
Pour la séquence initiale, il suffit de raisonner sur les valeurs en sortie des bascules : initialement, toutes les bascules sont à 0 après le *CLR*. Or, comme on prend la sortie $\overline{Q}$ de b_3, la séquence obtenue initialement en sortie des bascules est bien 1000. Pour les cycles suivants, on doit inverser l'entrée de b_3 pour annuler le fait qu'on prend $\overline{Q}$ en sortie de la bascule.

Ex. 2 : Décaleur à barillet

Un décalage d'un nombre constant de bits ne nécessite aucune porte logique : il s'agit de connecter les fils de façon adéquate (voir schéma ci-dessous).



Décalage à gauche de 2 positions



Symbole

Un décaleur à barillet (ou *barrel shifter*) est un opérateur combinatoire à deux entrées x et p qui permet de décaler un nombre x, codé sur n bits, d'un certain nombre de bits $p < n$ vers la gauche ou la droite. Notez qu'au contraire d'un décalage constant, un décaleur à barillet fonctionne pour toutes les valeurs de p. On note typiquement :

— $x << p$ le décalage de x de p bits vers la gauche en remplissant les cases libérées à droite par des 0 ;

— $x >> p$ le décalage de x de p bits vers la droite en remplissant les cases libérées à gauche par le bit de signe du nombre x initial (on parle de décalage arithmétique) ;

— $x >>> p$ le décalage de x de p bits vers la droite en remplissant les cases libérées à gauche par des 0 (on parle de décalage logique).

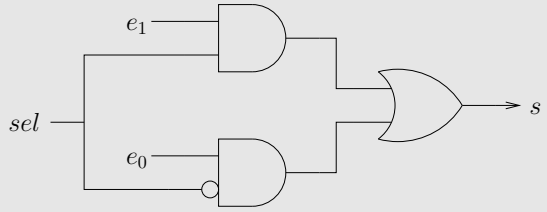
Question 1 Sur combien de bits doit être codé le décalage p si on suppose que x est codé sur 32 bits ? En déduire la relation générale entre le nombre de bits de x (n) et de p (m). Mathématiquement parlant, à quelles opérations élémentaires correspondent les différents décalages ?

Si x est codé sur 32 bits, p est compris entre 0 et 31 (on suppose qu'on n'autorise pas le décalage de 32 bits qui a pour effet de mettre tous les bits de x à 0 ou à 1), on a donc besoin de 5 bits pour coder p. De façon générale, on a besoin de $m = \lceil \log_2(n) \rceil$ (*i.e.* le plus petit entier supérieur ou égal à $\log_2(n)$) bits pour coder p.

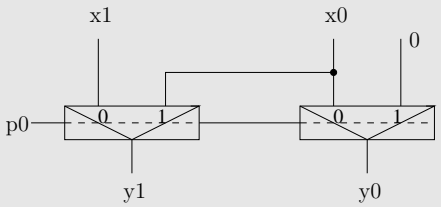
Décaler un entier relatif x de p bits vers la gauche revient à le multiplier par 2^p. Réciproquement, décaler arithmétiquement de p bits vers la droite revient à diviser par 2^p. Le décalage logique n'a de sens mathématique que pour les entiers naturels, mais est par ailleurs utile pour la manipulation de bits en programmation bas niveau (pilotes de périphériques, etc).

Question 2 Proposer l'implantation d'un décaleur à gauche d'une entrée x pour $n = 2$. Quelle porte élémentaire est idéale pour cette implantation ?

La porte de base pour cette opération est le mux 1 bit $2 \to 1$.



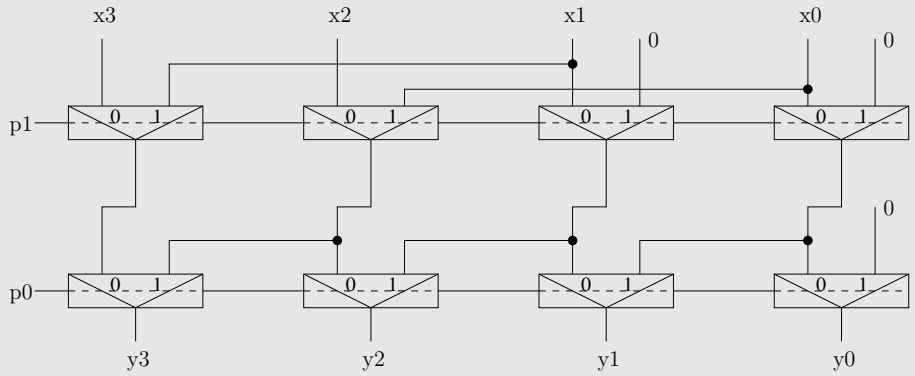
Il suffit de connecter les mux élémentaires comme précisé ci-dessous pour réaliser la fonction.



Question 3 En vous basant sur le résultat précédent, proposer une implantation pour $n = 4$ et $n = 8$. Évaluer le circuit : nombre de couches logiques à traverser et complexité en surface, l'unité étant le mux 1 bit $2 \to 1$.

L'astuce ici consiste à remarquer que les étages successifs peuvent chacun décaler d'une puissance entière de 2, et ainsi former un décalage de x avec n'importe quel entier naturel. Ainsi, p_0 décale de 0 ou 1 position, p_1 décale de 0 ou 2 positions, p_2 décale de 0 ou 4 positions, etc.

Le résultat pour un décaleur sur 4 bits est présenté ci-dessous.



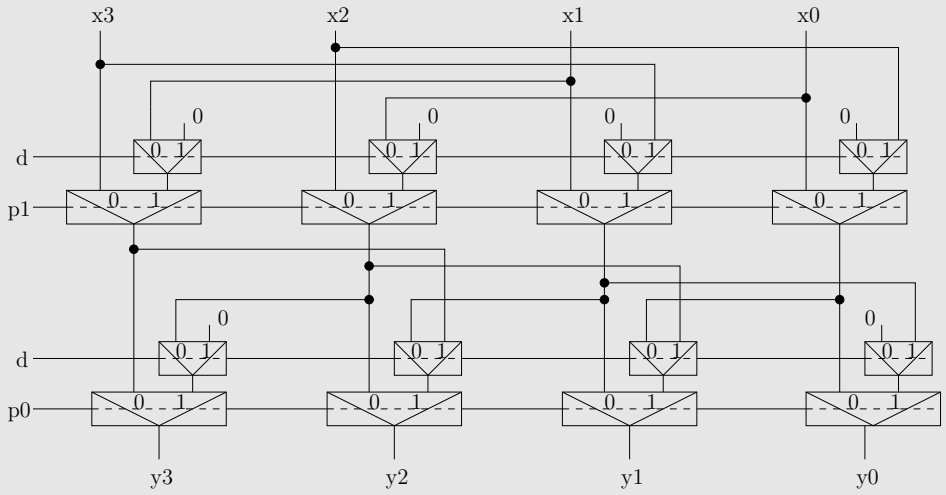
Faire la version 8 bits permet de bien ancrer l'idée, mais coûte du temps, ... On fait le calcul en $\lceil \log_2(n) \rceil$ couches logiques avec n mux élémentaires par couche logique, ce qui fait tout de même une complexité en surface en $O(n\lceil \log_2(n) \rceil)$.

Pour aller plus loin...
Question 4 On veut maintenant gérer le décalage logique vers la droite. On ajoute pour cela une entrée d telle que $d = 1$ si on effectue un décalage à droite. Modifier le décaleur 4 bits pour gérer le décalage logique à droite en plus du décalage à gauche. Attention, il y a deux solutions :

- la solution la plus évidente est de choisir à chaque étage un bit de l'étage précédent en fonction du sens du décalage à effectuer,
- la solution avec "miroirs" : on utilise le décaleur à gauche vu à la question 3. Si on doit décaler à droite, on permute les bits de l'entrée ("miroir" : on met en entrée du décaleur les bits 0 à 3 au lieu des bits 3 à 0), puis on effectue sur cette nouvelle valeur un décalage à gauche ; les bits du résultat obtenu sont permutés à nouveau en sortie.

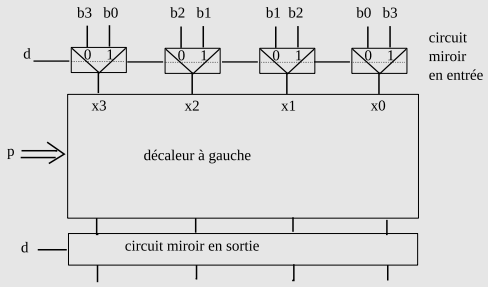
Evaluer les deux solutions.

Solution 1 :



On peut aussi le faire avec des mux $3 \to 1$, ce qui simplifie un peu le schéma.

Solution 2 :



Solution 1 : couches logiques : $\lceil \log_2(n) \rceil$ avec $2 * n$ mux par couche logique, complexité en surface en $O(2 * n \lceil \log_2(n) \rceil)$.

Solution 2 : couches logiques : $\lceil log_2(n) \rceil + 2$ avec n mux par couche logique, complexité en surface en $O(n(\lceil \log_2(n) \rceil + 2))$.

Pour aller plus loin...

Question 5 On veut enfin ajouter la gestion du décalage arithmétique à droite. On dispose d'une nouvelle entrée a valant 1 si on veut effectuer un décalage arithmétique lors d'un décalage à droite. On peut remarquer que le décalage à gauche est toujours arithmétiquement valable, indépendamment de la valeur de a. Modifier le schéma du décaleur gauche/droite 4 bits pour y ajouter la gestion du bit de signe dans les décalages à droite.

Ex. 3 : Étude d'une mémoire vidéo

On cherche à afficher une image bitmap sur un écran via le port VGA. L'image qui nous intéresse possède une résolution de 320×240, avec 8 bits par pixel. Cette image est stockée dans une mémoire contenant des mots de 32 bits. On peut donc coder 4 pixels par mot mémoire.

Question 1 Combien de mots mémoire sont nécessaires pour enregistrer l'image ? Sur combien de bits doit-on coder les adresses ?

Il y a $320 \times 240 = 76800$ pixels, chacun codé sur 8 bits, donc 19200 mots de 32 bits en mémoire.

Il faut 15 bits pour coder les adresses vu que $2^{14} = 16384 < 19200 \leq 2^{15} = 32768$.

On suppose que la couleur d'un pixel est codée suivant ses trois composantes R(rouge), V(vert) et B(bleu) (dans cet ordre là). Le codage choisi ici permet de désigner 4 nuances de rouge, 8 nuances de bleu et 8 nuances de vert.

Question 2 Comment coder le blanc, le noir, un vert vif ?

Un pixel sera codé sur 2+3+3 bits = 8 bits (2 bits pour le rouge, 3 bits pour le vert et le bleu). Le blanc sera codé $0xFF$ (toutes les composantes chromatiques à leur maximum), le noir sera codé $0x00$(toutes les couleurs à leur minimum chromatique), le vert vif sera codé $0b00111000 = 0x38$(la composante verte au maxi, les autres au mini).

On suppose que les pixels sont rangés dans la mémoire à la suite des uns des autres, les quatre premiers pixels (de coordonnées $0 \leq X \leq 3$, $Y=0$) à l'adresse 0, les quatre suivants (de coordonnées $4 \leq X \leq 7$, $Y=0$) à l'adresse 1, et ainsi de suite. Le premier pixel de la ligne suivante étant stocké à la suite du dernier pixel de la ligne courante.

Question 3 À quelle adresse en mémoire est stockée la couleur du pixel de coordonnées (x, y) ? Dans quel octet du mot mémoire à cette adresse ? Comment calculer cette adresse et cette position d'octet efficacement par un circuit ?

Le pixel (x, y) est le $x * 320 + y^{e}$pixels en mémoire. Comme 4 pixels sont rangés à chaque adresse mémoire, cela correspond à l'adresse $\left\lfloor \frac{x+320*y}{4} \right\rfloor$. Le reste de cette division donne la position de l'octet qui nous intéresse. Pour le calcul en circuit, on remplace la multiplication par des addition en décomposant 320 en base 2 : $320 = 256 + 64 = 2^8 + 2^6$. D'où $x + 320 * y = x + (y << 8) + (y << 6)$; les deux bits de poids faibles nous donnent la position de l'octet dans le mot mémoire, le reste l'adresse en mémoire.

Question 4 On souhaite positionner le pixel de coordonnées $(100, 3)$ avec la couleur blanche. Quels sont les signaux (Adresse, données, commandes) à fournir à la mémoire pour obtenir ce résultat ? Faites attention à ne pas modifier les pixels voisins !

L'adresse du pixel est (100+3*320)/4=265 (pas de reste), à mettre dans AD. La donnée permettant de fixer la couleur à blanche est 0xFF (cf. question précédente), à mettre dans DATA. Pour éviter de modifier les trois pixels suivants, il faut d'abord récupérer le mot mémoire correspondant, modifier l'octet de poids faible puis écrire. Les signaux de commande sont WE=inactif et CE=actif pour lire, WE=actif et CE=actif pour écrire.

L'écran vers lequel est envoyée l'image supporte le système de couleur *High color* : 32 nuances de rouge et de bleu, 64 nuances de vert. Le vert est favorisé car l'oeil humain y est plus sensible.

Question 5 Comment convertir les composantes de couleurs de notre image en format *High Color* ? On veillera à ce que les nuances soit réparties régulièrement sur le spectre des nuances possibles pour chaque couleur.

Une solution simple (mais peu adaptée) serait de transformer les 2 ou 3 bits en 5 ou 6 (suivant la couleur) en rajoutant des 0 en tête. Ce faisant, les valeurs iraient de 0 à 7 (pour les couleurs sur 3 bits) alors que le maximum est $2^5 - 1 = 31$ ou $2^6 - 1 = 63$. Nous aurions donc une image très sombre.

Afin d'avoir un bon contraste, il est préférable de rajouter les zéros dans les bits de poids faibles. Par exemple pour le rouge, si la valeur de départ sur 2 bits est $r_1 r_0$, on prendra $r_1 r_0 000$.