ASR1 – TD6 : Circuits asynchrones

{ Jeremie.Detrey, Patrick.Loiseau, Nicolas.Veyrat-Charvillon } @ens-lyon.fr

http://perso.ens-lyon.fr/jeremie.detrey/06_asr1/

6 et 7 novembre 2006

1 Préliminaires

1.1 Sans horloge, la fête est plus folle

1. Rappelez quels sont les avantages des circuits asynchrones par rapport à leurs équivalents synchrones.

Réponse :

- *En synchrone, si l'on change la tension d'alimentation, ou bien la température de fonctionnement du circuit, les propriétés temporelles des transistors et des fils changent, et certains délais de propagation peuvent varier. En asynchrone, le circuit est insensible aux délais de transition des portes ou de propagation des fils, donc beaucoup plus robuste.*
- *En synchrone, tous les registres basculent en même temps (à chaque front montant de l'horloge), d'où interférences électromagnétiques dans le circuit. En asynchrone, ces transitions sont mieux réparties dans le temps.*
- *En synchrone, la distribution du signal d'horloge est très difficile pour pouvoir assurer la synchronisation de chacun des registres desservis par l'arbre de distribution. En asynchrone, plus d'horloge donc plus de soucis.*
- *En synchrone, lorsque l'on sandwiche de la logique combinatoire entre deux bancs de registres, il faut tenir compte du délai maximal de stabilisation de cette logique pour ne pas cadencer l'horloge trop vite. En asynchrone, les données sont disponibles dès qu'elles sont prêtes : on travaille en temps moyen et non plus en temps maximal.*
- *En synchrone, même si le résultat calculé par une partie du circuit ne nous intéresse pas, des changements de valeur sur les entrées de ce sous-circuit provoquent des transitions qui consomment du courant pour rien. En asynchrone, on peut facilement désactiver une partie complète d'un circuit et ainsi minimiser la consommation électrique.*
- *Enfin, en asynchrone, le protocole avec acquittement permet de composer facilement divers modules entre eux, sans avoir à se soucier de contraintes globales.*

On se place ici dans le cadre de la logique asynchrone avec protocole 4-phases et encodage double-rail.

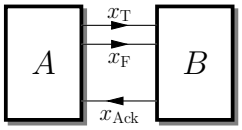
2. Malgré les apparences, la phrase précédente est en français. Que signifie-t-elle ?

Réponse :

L'encodage double-rail signifie que chaque bit de donnée x (possiblement valide ou invalide) va être encodé grâce à deux fils x_T et x_F de la manière suivante :

x_T	x_F	x
0	0	Invalide
0	1	Valide 0
1	0	Valide 1

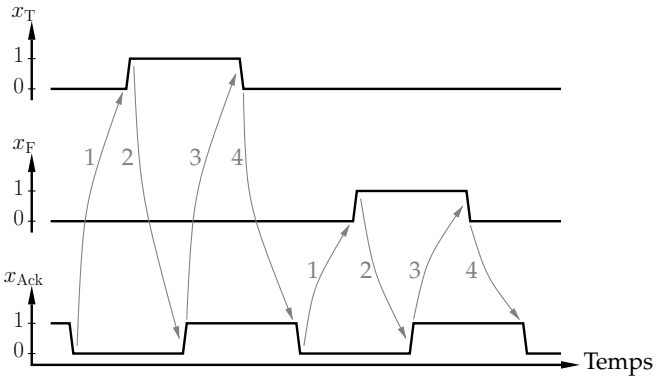
Le protocole 4-phases est un protocole d'acquittement des données, permettant au destinataire B de signaler à l'expéditeur A qu'il a bien reçu la donnée x que celui-ci lui a envoyé. Ceci est réalisé grâce à un signal x_{Ack}, comme présenté ci-dessous :



Le protocole 4-phases en lui-même est le suivant :

1. *lorsque x_{Ack} est à 0, A peut mettre une donnée valide sur la paire de fils $(x_{\mathrm{T}}, x_{\mathrm{F}})$;*
2. *lorsque B a bien reçu les données, il met son acquittement x_{Ack} à 1 ;*
3. *détectant l'acquittement à 1, A doit alors remettre sa donnée à l'état invalide ;*
4. *une fois que B a bien reçu la donnée invalide, il relâche son acquittement qui retourne à 0.*

Ceci est schématisé par le chronogramme suivant, dans lequel les flèches grises symbolisent la séquentialité chronologique des événements :



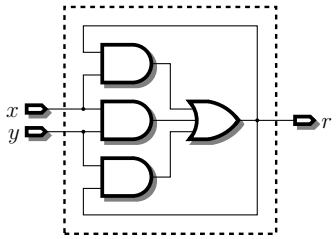
1.2 La porte de Muller

On rappelle ici la table de vérité et le symbole de la porte de Muller (aussi appelée *C-element*) :

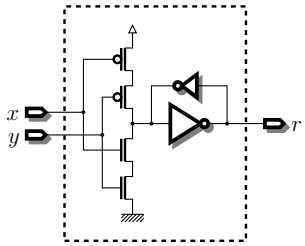
x	y	r
0	0	0
0	1	r
1	0	r
1	1	1

1. À quoi ce machin-là peut-il bien servir ?
Réponse :
La porte de Muller permet de synchroniser deux événements. Ainsi, elle ne passe à 0 que lorsque ses deux entrées sont passées à 0, et ne repasse à 1 que lorsque ses deux entrées sont elles aussi repassées à 1.
On parle aussi de ET événementiel*, ce qui explique le symbole de cette porte, inspiré du ET logique.*

2. Comment peut-on le réaliser à l'aide de portes de base ?
Réponse :
Il s'agit en fait d'une majorité dont la sortie reboucle sur une des entrées :

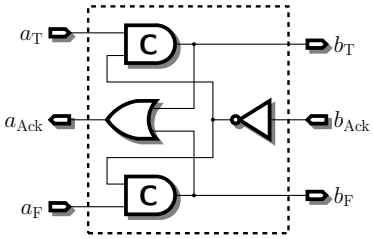


3. Et si l'on s'autorise des transistors en technologie CMOS, peut-on faire plus simple ?
Réponse :
Il faut une boucle d'inverseurs pour mémoriser la sortie lorsque x et y ont des valeurs différentes :



2 Registres asynchrones

On se donne le composant suivant, appelé demi-buffer (ou *half-buffer*) :



On suppose que les fils de données (a_T, a_F, b_T, b_F) sont préchargés à 0.

1. Essayez de comprendre le fonctionnement de ce demi-buffer, en supposant qu'il est connecté par son port d'entrée (a) et son port de sortie (b) à des circuit respectant le protocole 4-phases.
Réponse :
Le demi-buffer se contente de recopier son entrée sur sa sortie lorsque les entités qui se trouvent à sa droite et à sa gauche s'y prêtent.
Ainsi, lorsque la donnée en sortie $(b_\mathrm{T}, b_\mathrm{F})$ est invalide, l'acquittement a_Ack pour l'entrée est à 0. L'entité à gauche du demi-buffer peut donc lui donner une valeur valide sur l'entrée $(a_\mathrm{T}, a_\mathrm{F})$. Lorsque b_Ack passe à 0, cette donnée est mémorisée sur la sortie par les deux portes de Muller.
L'acquittement a_Ack de la donnée en entrée passe alors à 1, ce qui permet à l'entité sur la gauche d'indiquer un état invalide sur les fils d'entrée. Sur la droite du demi-buffer, une fois que l'entité

qui s'y trouve a lu la donnée $(b_\mathrm{T}, b_\mathrm{F})$, elle met à son tour son acquittement b_Ack à 1, ce qui permet aux portes de Muller de mémoriser l'état invalide.

L'acquittement a_Ack repasse alors à 0, ce qui signifie que le demi-buffer est à nouveau prêt à recevoir et retransmettre une nouvelle donnée.

2. Mettez six demi-buffers bout-à-bout et observez leur comportement en mettant en entrée tour-à-tour :
- un producteur, qui met une donnée sur $(a_\mathrm{T}, a_\mathrm{F})$ dès que l'acquittement a_Ack est à 0 puis l'enlève lorsque a_Ack passe à 1 ; ou
- rien du tout (donnée constante à *Invalide*) ;

et en sortie :
- rien du tout (acquittement b_Ack constant à 1) ; ou
- un consommateur, qui met son acquittement b_Ack dès qu'une donnée valide est présente sur $(b_\mathrm{T}, b_\mathrm{F})$, puis le repasse à 0 lorsque cette donnée redevient invalide.

Réponse :

On obtient un pipeline : les données se déplacent de la gauche vers la droite le plus loin possible, jusqu'à ce que le pipeline soit éventuellement rempli.

3. Combien de valeurs valides peut stocker ce circuit de six demi-buffers ? Justifiez l'appellation de demi-buffer.

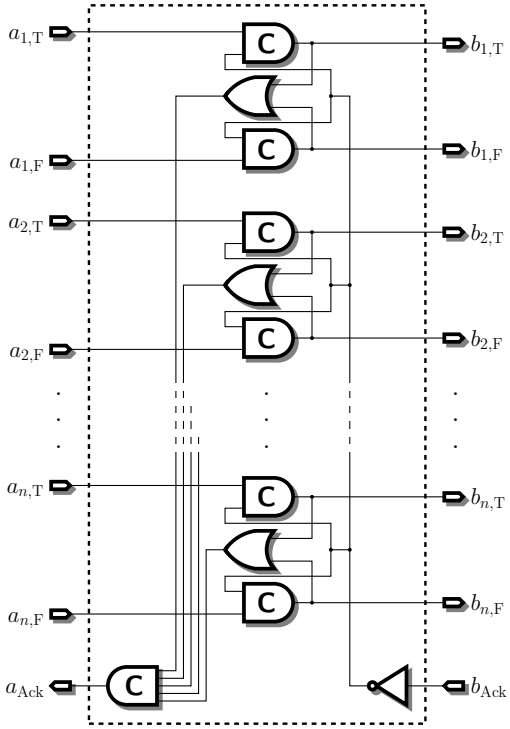
Réponse :

Le pipeline construit à l'aide de six demi-buffers ne peut contenir que trois valeurs : il faut donc deux demi-buffers pour stocker une valeur.

4. Comment construire un demi-buffer pour des valeurs sur plus d'un bit (plusieurs paires de rails de données, mais un seul acquittement) ?

Réponse :

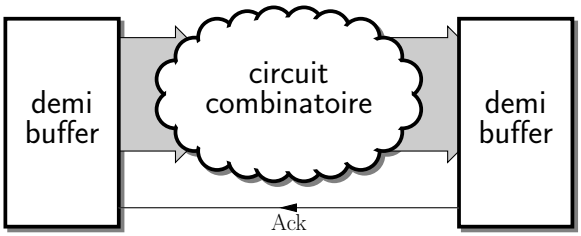
Il suffit de mettre plusieurs demi-buffers en parallèle. On distribue l'acquittement des sorties b_Ack et on synchronise l'acquittement des entrées grâce à une porte de Muller à n entrées :



3 Calculs asynchrones

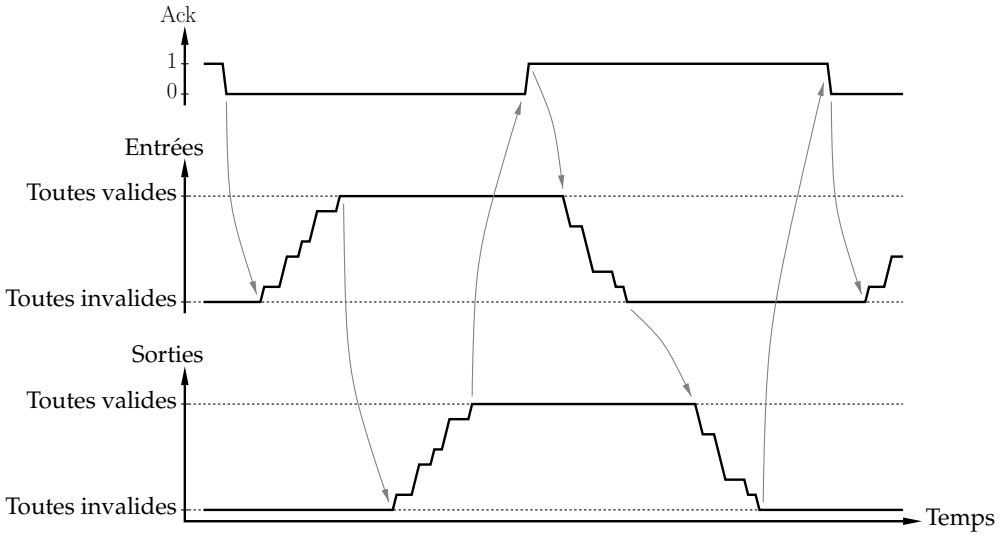
3.1 Concepts de synchronisation

On souhaite maintenant effectuer des calculs sur les données. On insère donc pour cela de la logique combinatoire entre nos étages de demi-buffers :

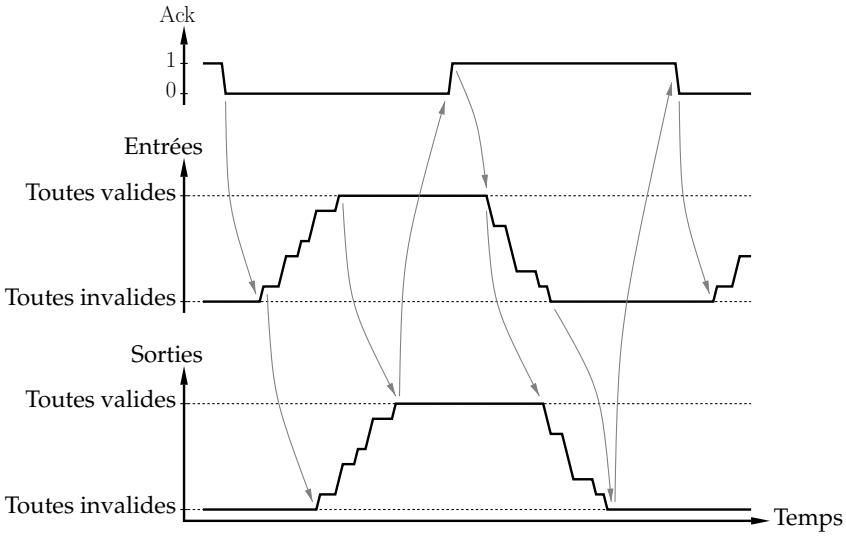


On distingue cependant deux types de comportement pour ces circuits combinatoires :

Circuits fortement indicatifs Dans ce type de circuit, les sorties ne commencent à être valides que lorsque toutes les entrées le sont. De même, lors du retour des entrées à l'état invalide, les sorties ne redeviennent invalides que lorsque toutes les entrées le sont déjà. Cela peut se voir sur le chronogramme suivant, dans lequel les flèches grises indiquent les contraintes de séquentialité temporelle :

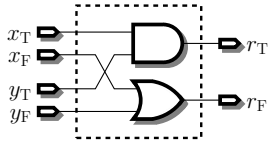


Circuits faiblement indicatifs Ce type de circuit est sensiblement moins contraint : des sorties peuvent devenir valides dès que certaines entrées le sont, et de même, elles peuvent redevenir invalides dès que certaines entrées le sont aussi. On a alors le chronogramme type suivant :



3.2 Portes de base

1. Que dire de la porte suivante ?



Réponse :
Elle ressemble fichtrement à un ET double-rail au vu de sa table de vérité :

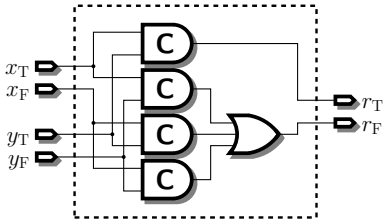
$(x_T, x_F) = x$	$(y_T, y_F) = y$	$(r_T, r_F) = r$
$(0,0) = I$	$(0,0) = I$	$(0,0) = I$
$(0,0) = I$	$(0,1) = 0$	$(0,1) = 0$
$(0,0) = I$	$(1,0) = 1$	$(0,0) = I$
$(0,1) = 0$	$(0,0) = I$	$(0,1) = 0$
$(0,1) = 0$	$(0,1) = 0$	$(0,1) = 0$
$(0,1) = 0$	$(1,0) = 1$	$(0,1) = 0$
$(1,0) = 1$	$(0,0) = I$	$(0,0) = I$
$(1,0) = 1$	$(0,1) = 0$	$(0,1) = 0$
$(1,0) = 1$	$(1,0) = 1$	$(1,0) = 1$

2. Est-elle fortement indicative ? faiblement indicative ?
Réponse :
Elle n'est ni fortement ni faiblement indicative. Elle n'est carrément pas indicative du tout en fait.

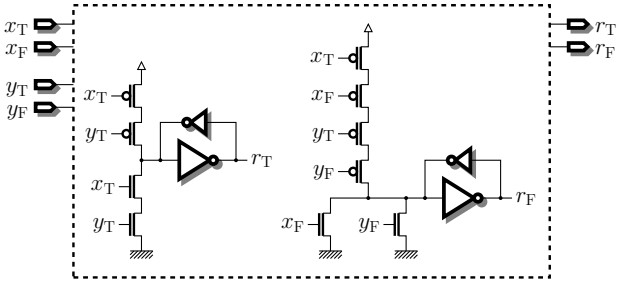
3. Quel est le problème que pose cette porte ?
Réponse :
On n'est plus capable de déterminer l'état (validité / invalidité) des entrées de cette porte en ne regardant que ses sorties : elle ne donne plus aucune indication.
Cela pose un problème car la construction du signal d'acquittement Ack *est effectuée par le demi-buffer en sortie cette porte. Ce demi-buffer n'étant conscient que de l'état de la sortie de la porte,*

il se peut qu'il change l'état de l'acquittement avant même que certaines des entrées n'aient eu le temps d'être valides / invalides.

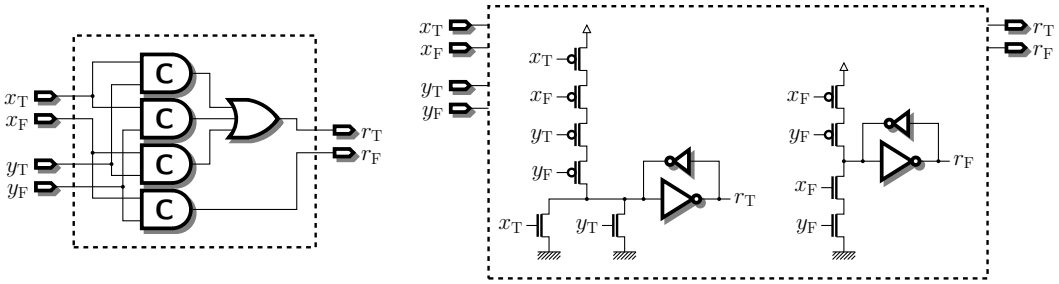
4. Comment faire un ET fortement indicatif ?
Réponse :
*On utilise la méthode DIMS (*Delay-Insensitive Minterm Synthesis*) qui consiste à synchroniser toutes les valeurs possibles des entrées grâce à des portes de Muller :*



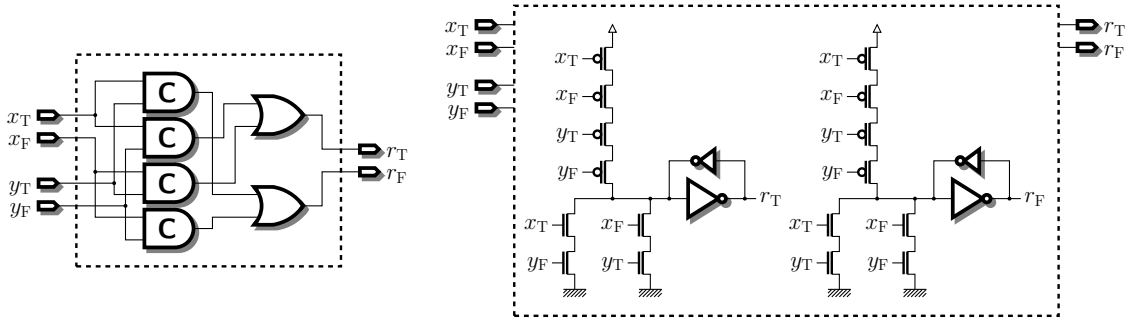
5. Quel est le coût (en transistors CMOS) de votre porte ? Peut-on faire mieux ?
Réponse :
Chaque porte de Muller coûte 8 *transistors (*4 *pour la porte proprement dite, et* 2 *par inverseur du bistable de mémorisation). La porte OU à trois entrées (réalisée comme un NON-OU suivi d'un inverseur) coûte elle aussi* 8 *transistors. Cela donne au total* 40 *transistors.*
On peut bien entendu faire mieux, en 18 *transistors :*



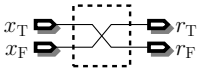
6. Faites de même pour les portes OU et XOR.
Réponse :
Pour la porte OU, on obtient, en DIMS et en CMOS :



De même, pour le XOR :

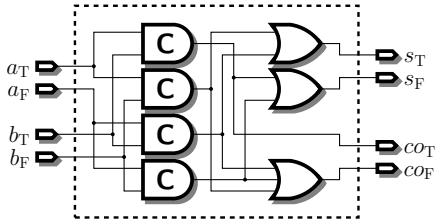


7. Quel est le coût de la porte NON ?
Réponse :
L'inverseur asynchrone est gratuit : il suffit d'intervertir les fils x_T et x_F.

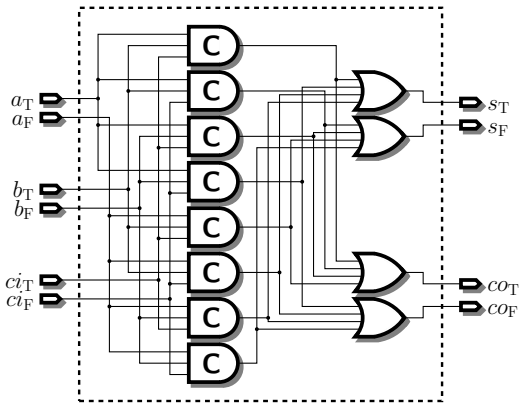


3.3 Additions

1. Utilisez la méthode DIMS (*Delay-Insensitive Minterm Synthesis*) pour construire un *half-adder* fortement indicatif.
Réponse :



2. De même pour un *full-adder*.
Réponse :



3. Assemblez tout ça pour faire un additionneur 8 bits avec propagation de retenue.
Réponse :
On met le HA sur les poids faibles, en série avec 7 FA pour les bits suivants. La retenue se propage

des poids faibles vers les poids forts et la retenue sortante du dernier FA devient le $9^{ème}$ bit de la somme.

4. Quel est le délai (en nombre de portes traversées) de cet additionneur ?
Réponse :
Chaque FA étant fortement indicatif, sa retenue sortante co n'est disponible qu'une fois sa retenue entrante ci arrivée. Le délai de l'additionneur est donc celui de la propagation de cette retenue sur toute la largeur de l'additionneur. Cela donne donc la traversée de 1 HA et de 7 FA, soit 8 portes de Muller et 8 portes OU.

L'état invalide se propage de la même manière, avec donc le même délai.

5. Voyez-vous un moyen d'aller plus vite ? À quel prix ?
Astuce : étudiez les propriétés de propagation de la retenue dans un *full-adder* en fonction des opérandes.
Réponse :
On se rend bien compte que dans l'additionneur précédent c'est la propagation de la retenue qui prend du temps. Sans elle, tous les bits de somme pourraient être calculés en parallèle.

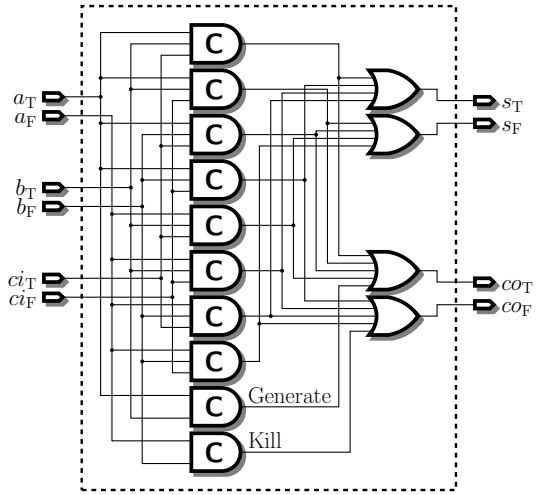
Cependant, on peut voir que la valeur de la retenue sortante co d'un FA ne dépend pas toujours de la retenue entrante ci. Ainsi, si l'on dresse la table de vérité de co en fonction des bits des opérandes a et b, on trouve :

a	b	co
0	0	0
0	1	ci
1	0	ci
1	1	1

*Ainsi, lorsque a et b valent 0, la retenue sortante sera toujours 0, quelle que soit la retenue entrante (*Kill*). De même, lorsque a et b valent 1, la retenue sortante sera toujours 1 (*Generate*). Enfin, dans les cas restants, la retenue entrante est directement propagée sur la retenue sortante (*Propagate*). On a ainsi :*

$$\begin{cases} \text{Kill} = \overline{a} \cdot \overline{b}, \text{ et} \\ \text{Generate} = a \cdot b. \end{cases}$$

On construit alors la cellule FA faiblement indicative suivante :

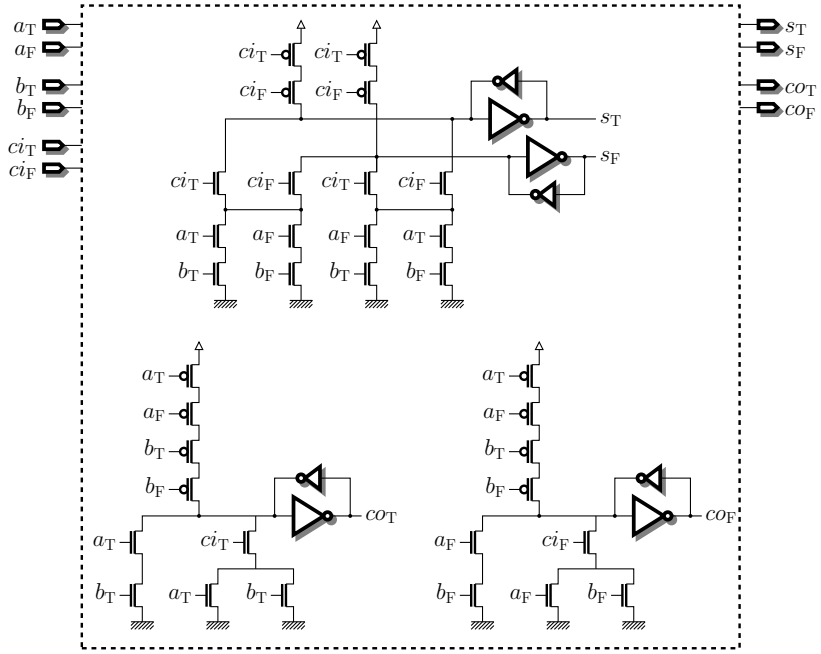


6. Allez-vous vraiment plus vite ?
Réponse :
Lorsque les entrées sont valides, la chaîne de propagation de la retenue est divisée en plusieurs petites propagations qui s'effectuent en parallèle. Dans certains cas, bien sûr, la retenue doit se propager sur toute la largeur de l'additionneur, mais ces cas restent exceptionnels. En moyenne, la longueur de la plus grande chaîne de propagation est en $\log_2 n$ *pour un additionneur* n *bits. On accélère donc bien le temps de calcul moyen.*

Par contre, lors du retour à invalide, on a le même problème qu'avant : il faut toujours propager l'état invalide sur toute la chaîne de retenue. L'additionneur est donc plus rapide pour faire les calculs, mais est toujours long à se réinitialiser.

7. Comment faire mieux ?
Réponse :
L'additionneur de Martin résoud le problème en gardant le même principe pour éviter de propager la retenue lors des calculs, mais en changeant aussi la dépendance des données pour le retour à l'état invalide. En effet, dans cet additionneur, l'invalidité du bit de somme s *ne dépend que de l'invalidité de la retenue entrante* ci*, et l'invalidité de la retenue sortante ne dépend que de l'invalidité des opérandes* a *et* b*. Ainsi, la chaîne de retenue est découpée lors du retour à l'invalide.*

Voici la cellule FA de l'additionneur de Martin, sous forme de transistors CMOS :



Pour en savoir plus, n'hésitez pas à aller à la bibliothèque emprunter le bouquin de Jens Sparsø intitulé *Principles of asynchronous circuit design – A systems perspective*. Les quelques 200 premières pages sont d'ailleurs un très bon tutoriel sur la logique asynchrone et sont de surcroît disponibles librement sur le net à l'adresse suivante :
`http://www2.imm.dtu.dk/pubdb/views/publication_details.php?id=855`