ARCHITECTURE MACHINE

Contenu :

Description des couches inférieures du fonctionnement logique d'un ordinateur :

Logique booléenne → circuits élémentaires → machine simple → assembleur

Partant de l'algèbre de Boole, qui donne les outils pour combiner les expressions vraies ou fausses, on décrit les circuits logiques élémentaires qui vont permettre de constituer des composants de base tels un comparateur de nombres, un additionneur/soustracteur, un registre, une RAM, une ROM. On décrira une machine simple (un microprocesseur sommaire) équipée de registres, de RAM, et d'un jeu d'instructions en assembleur dont on souhaite se doter. On réalisera la micro-programmation en ROM du jeu d'instructions.

Cette description peut utilement compléter la connaissance plus répandue des couches logiques supérieures, non abordées dans ce cours :

Assembleur → langages évolués → applications

Sommaire :

Algèbre booléenne

Soit a un événement ; a peut prendre les valeurs logiques « vrai » ou « faux », notées respectivement 1 et 0

a vrai $\Leftrightarrow$ a = 1
a faux $\Leftrightarrow$ a = 0

L'événement complémentaire de a, dit non-a, sera noté $\overline{a}$
exemple d'évènement : "je tire un 6 au lancer de dé". Cet évènement est faux si mon dé jeté montre une de ses faces de 1 à 5, vrai s'il montre le 6.
Son évènement complémentaire est "je tire une face de 1 à 5 au lancer de dé".

Le « ou inclusif » sera noté « + » : a+b (a ou b)
exemple de "ou inclusif" : grâce aux notes déjà acquises, je vais réussir mon examen si j'ai la moyenne en maths **ou** en physique. Dans l'hypothèse où j'ai la moyenne dans les deux matières, je suis bien sûr toujours dans la situation d'avoir réussi l'examen.

Le « et » sera noté « . » : a.b ou plus simplement ab (a et b)
exemple de "et" : pour faire 12 avec deux dés, je dois tirer six au premier jet **et** tirer six au second.

Tables de vérité (exemples)

a	$\overline{a}$
0	1
1	0

a	b	a+b
0	0	0
0	1	1
1	0	1
1	1	1

lire la table de vérité de "a ou b" (a+b) : on met en colonne toutes les valeurs que peut prendre l'évènement a, et on combine avec toutes les valeurs que peut prendre l'évènement b.
1ère ligne : on peut avoir a faux et b faux (valeurs 0), dans ce cas a ou b est faux. On renseigne donc zéro dans la colonne de droite, colonne de résultat a+b
2ème ligne : a peut encore être faux, mais b vrai cette fois (valeur 1). Dans ce cas, a ou b est vrai, on renseigne 1 dans la colonne résultat.
3ème ligne : on a épuisé tous les cas pour b dans l'hypothèse où a était faux, il reste à voir les cas où a est vrai. On renseigne donc la première colonne à 1, et on explore les deux cas (faux et vrai) pour b, etc...

Identités

elles sont simples, il faut juste faire l'effort de "décoder" la notation algébrique :

$\overline{\overline{a}}=a$ non(non-a), c'est a
$\overline{a+b}=\overline{a}\,\overline{b}$ non(a ou b) nécessite non-a et non-b
$\overline{ab}=\overline{a}+\overline{b}$
$a\overline{a}=0$
$a+\overline{a}=1$
$a+0=a$
$a.0=0$
$a+1=1$
$a.1=a$
$ab+ac=a(b+c)$
$(a+b)(a+c)=a+bc$

Si la "lecture" de l'identité n'est pas suffisamment simple, il reste la possibilité de la vérifier (=démontrer) en écrivant la table de vérité des deux expressions.

exemple de la seconde identité :

a	b	a+b	$\overline{a+b}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

a	b	$\overline{a}$	$\overline{b}$	$\overline{a}\,\overline{b}$
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0

les colonnes de droite de chaque tableau portent les mêmes valeurs, les deux expressions sont donc égales : $\overline{a+b}=\overline{a}\,\overline{b}$

Comme quoi, c'est utile de savoir écrire la table de vérité !

Compléments sur les notations

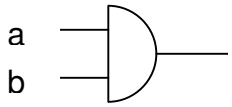
ou exclusif : $\oplus$
exemple de "ou exclusif" : j'ai un week-end de trois jours si le vendredi **ou** le lundi est férié. C'est vrai si vendredi est férié, vrai si lundi est férié, mais faux si vendredi et lundi sont fériés (le week-end est alors de quatre jours).

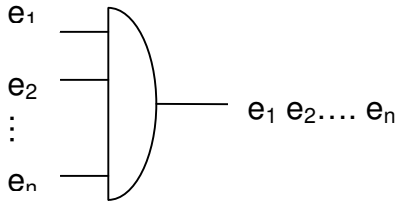
variantes de notation (non utilisées dans ce document) :

$\overline{a}=\neg a$
$+=\vee$
$.=\wedge$

Circuits logiques

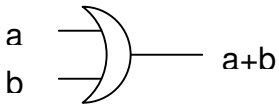
Porte et



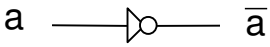


Ces dessins représentent des éléments (les circuits logiques) qui, en présence d'évènements en entrée, fournissent un évènement en sortie. L'évènement en sortie résulte d'opérations algébriques.

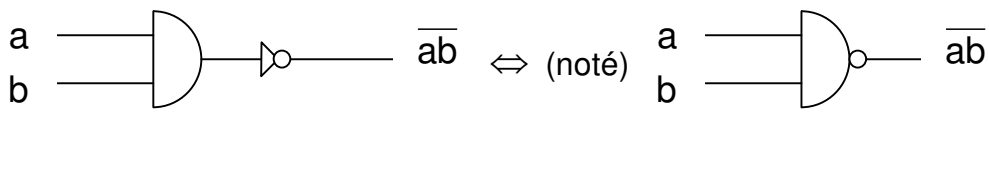
Porte ou



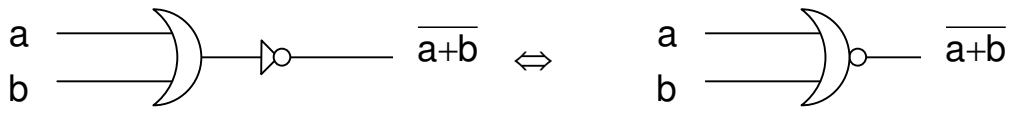
Inverseur



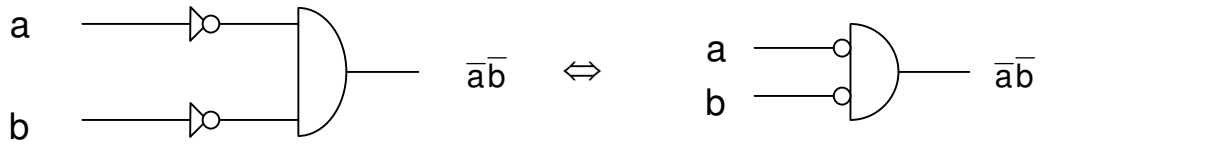
Exemples de combinaisons de portes



porte Nand
(not and, non-et)



porte Nor
(not or, non-ou)



2ème porte Nor car $\overline{a+b}=\overline{a}\overline{b}$

On aperçoit ici l'intérêt des identités algébriques, elle permettent d'identifier plusieurs solutions pour réaliser un circuit logique.

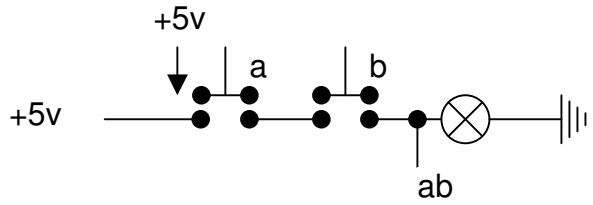
Réalisation, industrialisation

Dessiner des portes c'est bien, les réaliser concrètement c'est utile pour faire un ordinateur. S'attachant à la compréhension logique, nous n'entrerons pas dans le détail de ces aspects physiques, riches de nombreuses technologies et nécessitant des connaissances électroniques. Voici toutefois un aperçu simpliste :

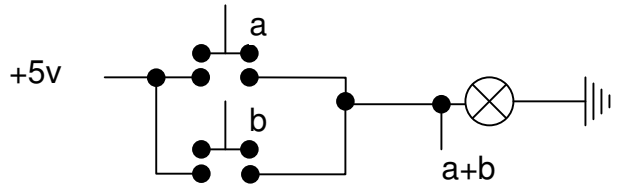
- à la notion d'événement vrai ou faux correspond une tension électrique pouvant prendre deux valeurs (en fait, deux plages de valeurs). La tension électrique apparaît entre le segment de circuit considéré (une entrée ou une sortie, par exemple) et la masse. On dira par exemple que a est vrai si la tension mesurée est autour de +5 volts, et que a est faux si la tension est nulle (ou proche de 0). Si on imagine interposer une ampoule électrique entre le point a et la masse, l'état vrai ou faux de a sera traduit par l'ampoule allumée ou éteinte.

- pour réaliser les portes logiques, il faut donc commander le passage ou non du courant vers la sortie, selon l'état des entrées : bref, un problème d'interrupteurs !

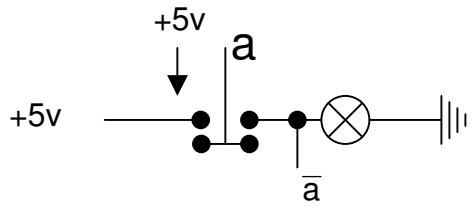
- porte et : les entrées a et b commandent chacune un interrupteur (le fermant sous tension), les deux interrupteurs sont montés en série ; pour que le courant passe en sortie les deux interrupteurs doivent être fermés :



- porte ou : les deux interrupteurs sont montés en parallèle ; le courant passe par l'un ou l'autre interrupteur (ou les deux).



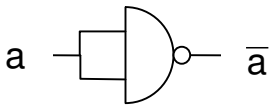
- Inverseur : l'entrée pilote un interrupteur qui reste ouvert sous tension, le courant passe vers la sortie si on le coupe en entrée.



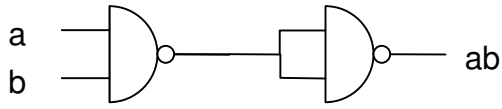
- réalisation : on fait jouer le rôle d'interrupteur à un transistor, composant que l'on sait miniaturiser.

- industrialisation : on optimise l'intégration, la miniaturisation et les performances (rapidité de réaction, consommation électrique, dissipation calorifique, durée de vie …) d'une porte de base, en général la porte Nand, dite universelle car toutes les autres portes peuvent être composées exclusivement à partir de portes Nand :

inverseur : il suffit de connecter ensemble les deux entrées :



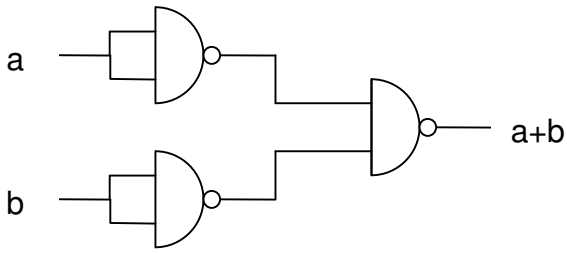
porte *et* : le *nand* étant un *et* inversé, il suffit d'ajouter un inverseur en sortie pour revenir au *et* initial :



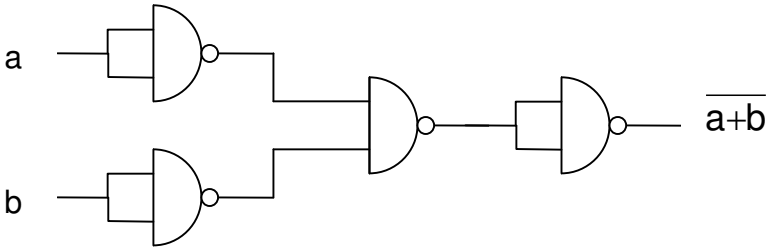
porte or : puisque $\overline{ab}=\overline{a}+\overline{b}$,

$\overline{\underbrace{\overline{a}\overline{b}}_{\text{nand}(\overline{a},\overline{b})}}=\overline{\overline{a}}+\overline{\overline{b}}=a+b$

on ajoute donc un inverseur à chaque entrée



porte nor : il suffit d'ajouter un inverseur à la porte précédente :



Il s'agit bien ici de rationalisation industrielle, et non d'optimisation logique !
Ce ne sont pas les circuits les plus simples à "lire" et comprendre, mais les plus simples à réaliser au moindre coût.

Ou exclusif (porte XOR)

Comment réaliser une nouvelle porte à base de celles dont on dispose déjà ?
La première étape consiste à faire la table de vérité de la fonction logique dont on veut disposer. Ecrivons la table de vérité du ou exclusif :

a	b	a⊕b
0	0	0
0	1	1
1	0	1
1	1	0

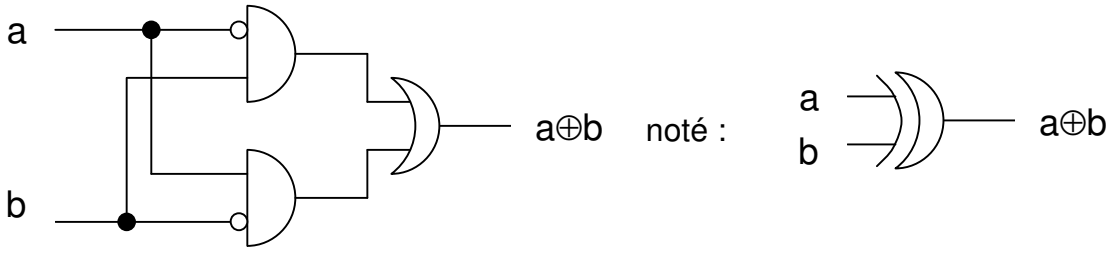
On remplit la table de vérité avec autant de lignes que de combinaisons possibles des entrées, puis on calcule la sortie pour chaque ligne.

a	b	a⊕b
0	0	0
0	1	1
1	0	1
1	1	0

$$\Rightarrow a \oplus b = \overline{a}\, b + a\, \overline{b}$$

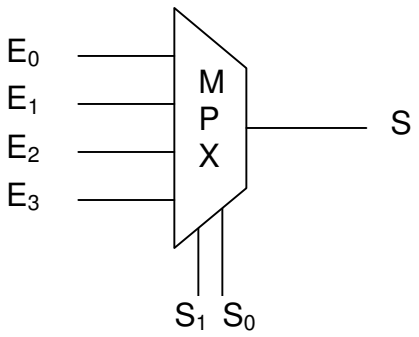
Ceci fait, on peut aisément écrire l'équation logique qui sera traduite à base de portes. a⊕b est vrai dans la configuration de la ligne 2, **ou** dans la configuration de la ligne 3. La configuration de la ligne 2, c'est **non-a et b** ; la configuration de la ligne 3 c'est **a et non-b**.

On réalisera a⊕b par des portes logiques de base : (*non*-a **et** b) **ou** (a **et** *non*-b).
On a besoin de deux portes **et**, d' une porte **ou**, et de deux *inverseurs* :



Multiplexeur (MPX)

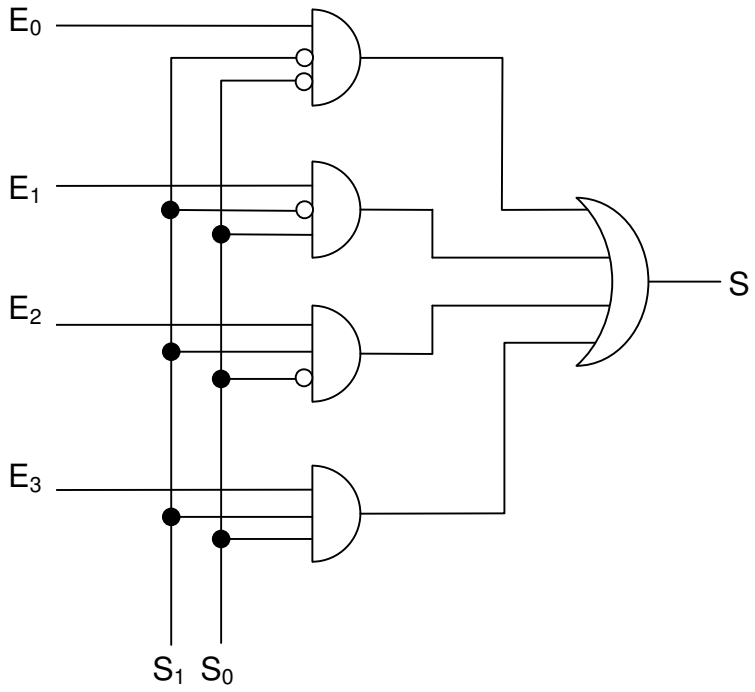
Exemple à 4 entrées, 1 sortie (S_1 S_0 constituant le sélecteur de commande) :



Le multiplexeur sélectionne un des signaux en entrée, et le véhicule en sortie, en obéissant à la commande du sélecteur. Les quatre cas de la table de vérité des deux évènements de sélection permettent la sélection parmi les quatre entrées :

S_1	S_0	S
0	0	e_0
0	1	e_1
1	0	e_2
1	1	e_3

d'où :



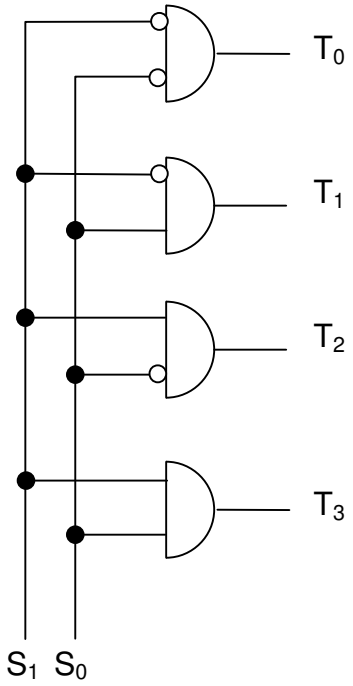
Chaque porte "et" a deux entrées contrôlées par les valeurs de S_1 et S_0, une seule porte "et" peut avoir ces deux entrées à vrai. Seule cette porte "laisse passer" le signal d'évènement, vrai si E est vrai, faux sinon. Toutes les autres portes "et" ont au moins un signal faux en entrée, peu importe l'état de leur entrée E, elles produisent faux en sortie. Le "ou" final est donc entre l'entrée sélectionnée ou trois entrées à faux, le résultat est donc l'entrée sélectionnée.

Décodeur

Les 4 valeurs possibles codées sur 2 bits S_0, S_1 sont décodées sur 4 voies

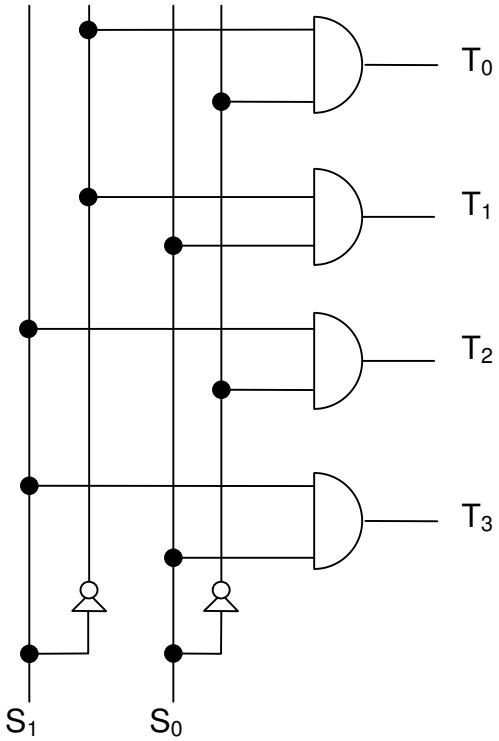
S_1	S_0	T_0	T_1	T_2	T_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

D'où :



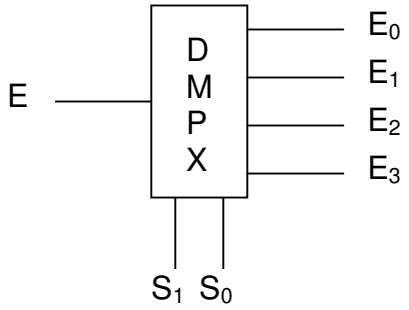
Le signal S en entrée est considéré "codé" puisqu'il peut prendre quatre valeurs en s'appuyant sur seulement deux évènements (en les combinant).
On peut se représenter S comme un nombre de 0 à 3 codé en binaire sur deux bits.
T est décodé, puisque chacune des quatre valeurs possibles est identifiable par son signal (évènement T_i) dédié.

Ou mieux (2 inverseurs au lieu de 4) :



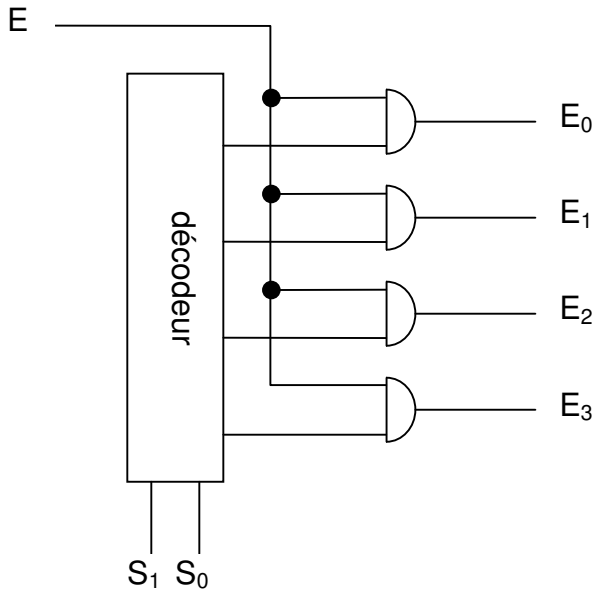
Puisqu'on a plusieurs fois besoin du signal inversé issu de S_1 et S_0, autant se doter de l'inverse comme d'une pseudo-entrée supplémentaire, et l'utiliser ensuite directement ; plutôt que d'inverser à chaque occasion le signal d'origine.

Démultiplexeur (DMPX)



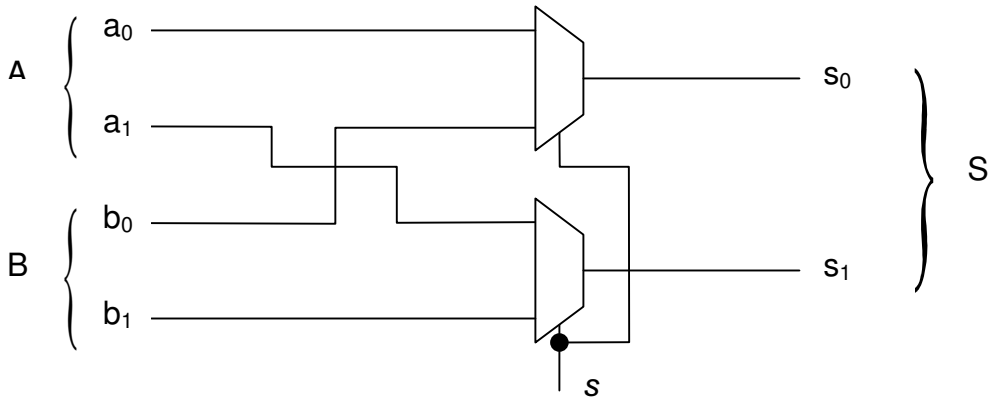
S_1 et S_0 sont les fils de sélection qui vont permettre d'aiguiller E vers l'un des E_i , les autres étant à zéro. Le démultiplexeur fait le travail inverse du multiplexeur.

Réalisation :

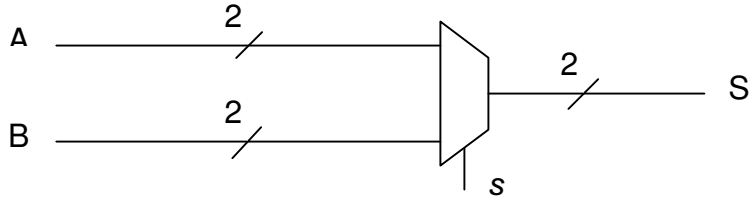


Puisqu'on s'est précédemment doté d'un décodeur, la réalisation du démultiplexeur devient aisée.

Exemple de multiplexeur 2 entrées 2bits, sortie 2 bits

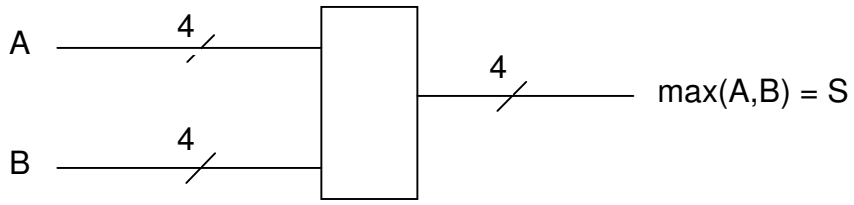


Que l'on peut noter :



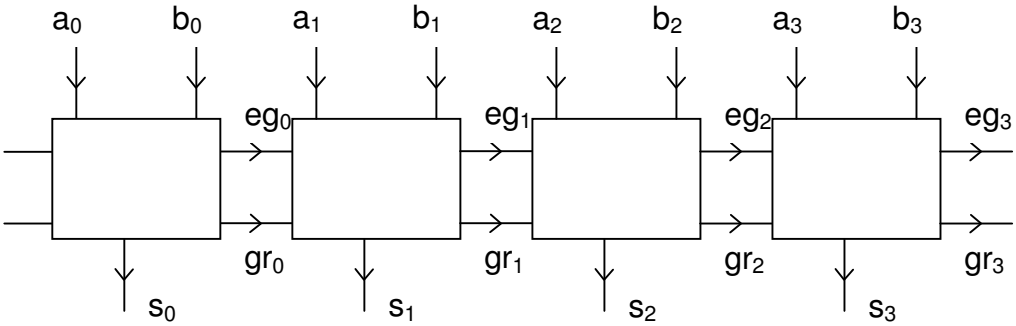
Comparateur de 2 nombres de 4 bits

Voici le schéma de ce qu'on veut réaliser :



En sortie, on veut donc le plus grand des deux nombres en entrée.

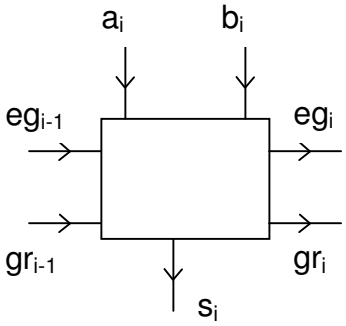
On compare bit à bit à partir du bit de poids fort (noté ici bit 0) :



avec gr_i (plus grand) = 0 si $a_i > b_i$
gr_i (plus grand) = 1 si $a_i < b_i$
eg_i (égal) = 1 si $a_i = b_i$
"gr", c'est "b strictement plus grand que a"

Pour chaque bit comparé, soit c'est celui de a le plus grand (gr faux), soit c'est celui de b (gr vrai), soit il y a égalité. L'information doit être propagée de cellule en cellule : soit jusqu'alors il y a égalité, donc il faudra regarder en détail les bits suivants, soit l'un des nombres a déjà fait la différence, il faut savoir lequel pour sélectionner ses bits suivants en sortie.

Chaque cellule a donc la forme suivante :



Comment doit-elle fonctionner ?

Algorithme de remplissage de la table de vérité pour une cellule :

```
si eg_{i-1} = 1                          (jusqu'à ce stade, tous les bits étaient identiques)
    si a_i = b_i alors                   (toujours égalité)
        eg_i ← 1                         (on informe la cellule suivante que l'égalité perdure)
        gr_i ← x                         (n'importe quoi, 0 ou 1, on ne s'en servira plus)
        s_i ← a_i                        (ou b_i, peu importe ils sont égaux, il faut produire l'info en sortie)
    sinon
        eg_i ← 0                         (il n'y a plus égalité)
        s_i ← max(a_i,b_i)               (=1 puisque ai≠bi⇒ai=bi=0 ; le + grand vaut 1 et l'autre 0)
        si a_i > b_i alors               (on identifie le plus grand)
            gr_i ← 0                     (on a convenu gr vrai si a < b, ici gr est donc faux)
        sinon
            gr_i ← 1                     (on a convenu gr vrai si a < b, ici gr est donc vrai)
        fsi
    fsi
sinon                                    (l'inégalité est déjà établie)
    si gr_{i-1} = 0 alors                (on reporte le plus grand nombre en sortie, est-ce a ?)
        s_i ← a_i                        (oui, gr étant faux, c'est a le plus grand)
    sinon
        s_i ← b_i                        (gr étant vrai, c'est b le plus grand)
    fsi
    eg_i ← eg_{i-1}    (=0)              (on propage l'information "inégalité déjà établie")
    gr_i ← gr_{i-1}                      (on propage l'information "qui est le plus grand")
fsi
```

cet algorithme aide à remplir les 3 colonnes résultats de la table de vérité suivante :

a_i	b_i	eg_{i-1}	gr_{i-1}	s_i	eg_i	gr_i
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	x
0	0	1	1	0	1	x
0	1	0	0	0	0	0
0	1	0	1	1	0	1
0	1	1	0	1	0	1
0	1	1	1	1	0	1
1	0	0	0	1	0	0
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	1	0	0
1	1	0	0	1	0	0
1	1	0	1	1	0	1
1	1	1	0	1	1	x
1	1	1	1	1	1	x

D'où (on exprimera $\overline{Si}$ plutôt que S_i par économie : 6 cas vrais (= lignes à 0) contre 10) :

$$\overline{si} = \overline{ai}\,\overline{bi}\,\overline{e}\,\overline{g_{i-1}}\,\overline{gr_{i-1}} + \overline{ai}\,\overline{bi}\,e\,\overline{g_{i-1}}\,gr_{i-1} + \overline{ai}\,\overline{bi}\,e\,g_{i-1}\,\overline{gr_{i-1}}$$
$$+ \overline{ai}\,bi\,\overline{e}\,\overline{g_{i-1}}\,gr_{i-1} + \overline{ai}\,bi\,e\,g_{i-1}\,gr_{i-1} + ai\,\overline{bi}\,\overline{e}\,\overline{g_{i-1}}\,\overline{gr_{i-1}} + ai\,\overline{bi}\,\overline{e}\,g_{i-1}\,gr_{i-1}$$

Travaillons (un peu d'algèbre !) à simplifier autant que possible cette expression ; par exemple en remarquant que $\overline{ai}\,\overline{bi}$ est en "et" commun des quatre premières expressions reliées par "ou" ; et que $\overline{e}\,g_{i-1}$ est dans la même situation pour les deux derniers "ou" :

$$= \overline{ai}\,\overline{bi}\,(\underbrace{\overline{e}\,\overline{g_{i-1}}\,\overline{gr_{i-1}} + \overline{e}\,\overline{g_{i-1}}\,gr_{i-1} + e\,\overline{g_{i-1}}\,\overline{gr_{i-1}} + e\,g_{i-1}\,gr_{i-1}}_{x})$$
$$+ \overline{e}\,g_{i-1}\,(\overline{ai}\,bi\,\overline{gr_{i-1}} + ai\,\overline{bi}\,gr_{i-1})$$

avec (on continue à simplifier)

$$x = \overline{e}\,g_{i-1}\,(\underbrace{\overline{gr_{i-1}} + gr_{i-1}}_{1}) + e\,g_{i-1}\,(\underbrace{\overline{gr_{i-1}} + gr_{i-1}}_{1}) = \underbrace{\overline{e}\,g_{i-1}}_{1} + \underbrace{e\,g_{i-1}}$$

x est donc toujours vrai, soit

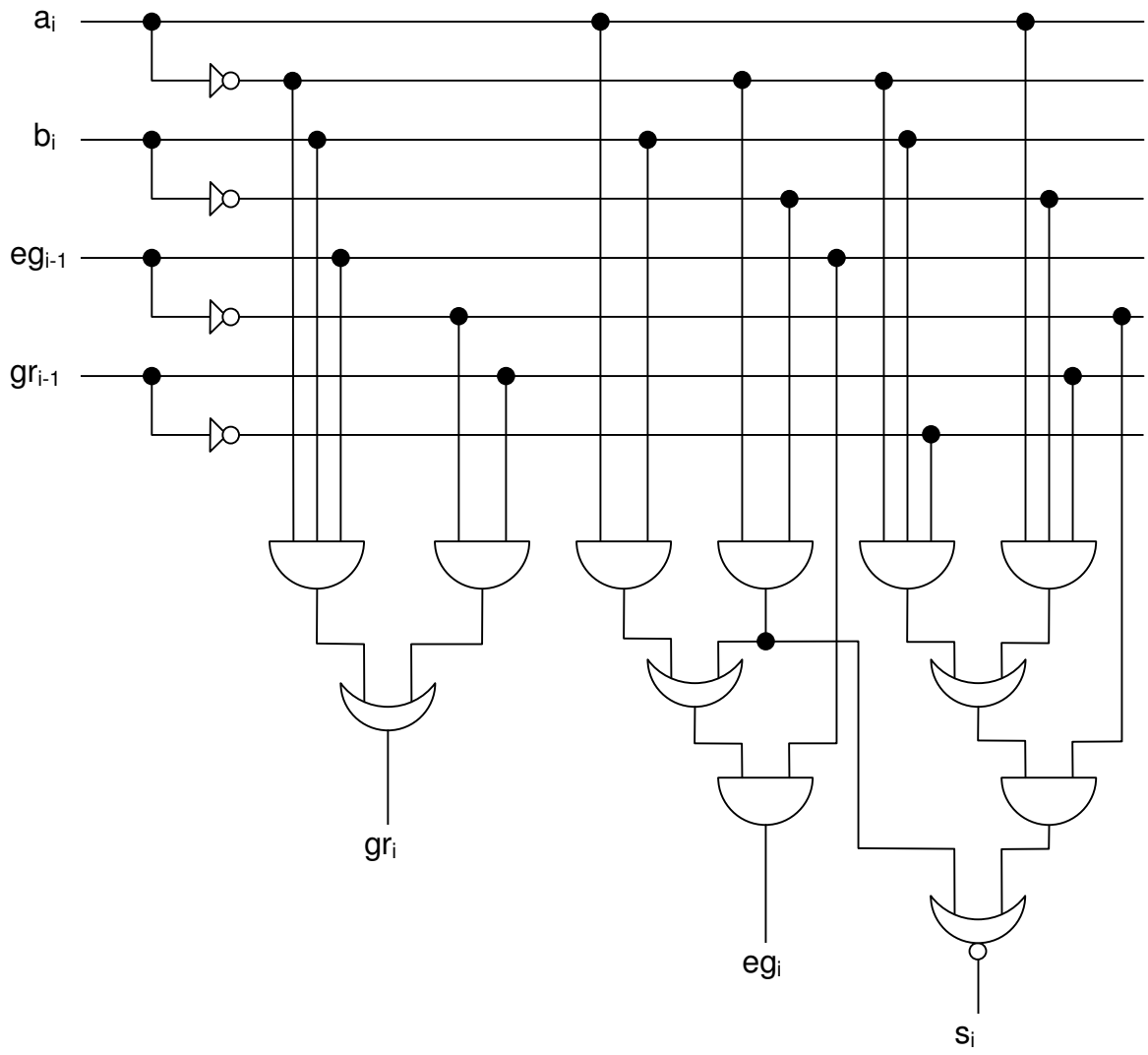
$$\overline{si} = \overline{ai}\,\overline{bi} + \overline{e}\,g_{i-1}\,(\overline{ai}\,bi\,\overline{gr_{i-1}} + ai\,\overline{bi}\,gr_{i-1})$$

Il faut faire le même exercice pour les sorties eg_i et gr_i : écrire l'équation algébrique "brute" en lisant la table de vérité, puis la simplifier. On établit ainsi :

$$gr_i = \overline{ai}\,bi\,e\,g_{i-1} + \overline{e}\,g_{i-1}\,gr_{i-1}$$
$$e\,g_i = e\,g_{i-1}\,(\overline{ai}\,\overline{bi} + ai\,bi)$$

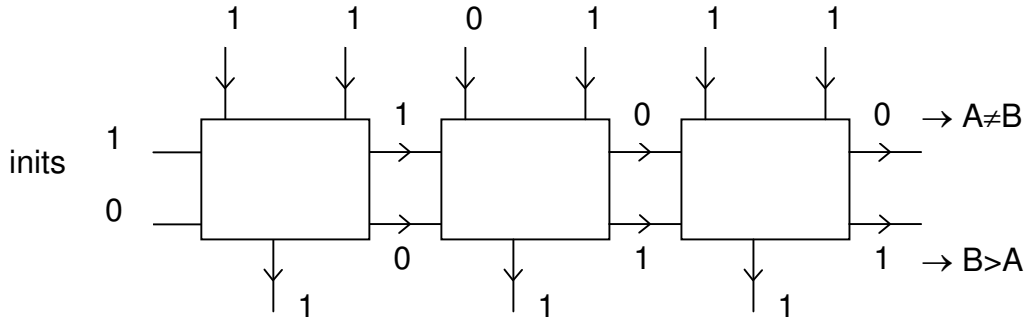
Ces trois équations simplifiées permettent de déduire le circuit pour une cellule :

$$\overline{s_i} = \overline{a_i}\,\overline{b_i} + \overline{e_{g_{i-1}}}(\overline{a_i}\,b_i\,\overline{gr_{i-1}} + a_i\,\overline{b_i}\,gr_{i-1})$$
$$gr_i = \overline{a_i}\,b_i\,e_{g_{i-1}} + \overline{e_{g_{i-1}}}\,gr_{i-1}$$
$$e_{g_i} = e_{g_{i-1}}(\overline{a_i}\,\overline{b_i} + a_i\,b_i)$$



Exemple sur 3 bits :

A = 101 (5) , B = 111 (7)



Additionneur

Représentation des nombres signés

. On utilise un bit de signe : 0 = positif ou nul ; 1 = négatif
. Pour inverser le signe d'un nombre, on effectue son complément à 2 :

- on effectue d'abord le complément à 1, obtenu par $\oplus 1$ bit à bit
- on ajoute 1 pour obtenir le complément à 2

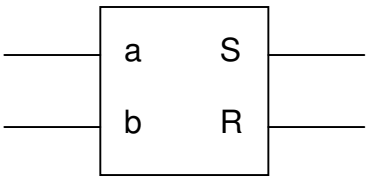
. Exemple sur 4 bits :

	0 0 1 1	= +3
	1 1 0 0	= complément à 1
	1 1 0 1	= complément à 2 = -3
+	0 0 1 1	vérifions….
=	0 0 0 0	-3 + 3 = 0

On peut ainsi coder les nombres de –8 à +7 sur 4 bits ; de –128 à +127 sur 8 bits ; de –32768 à +32767 sur 16 bits ….

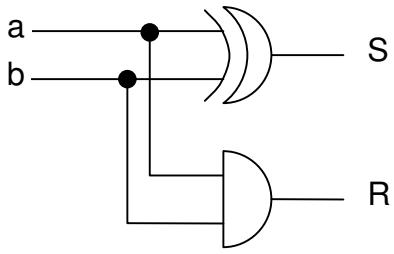
Demi-additionneur

Sur 1 bit (S=Somme, R=Retenue) :

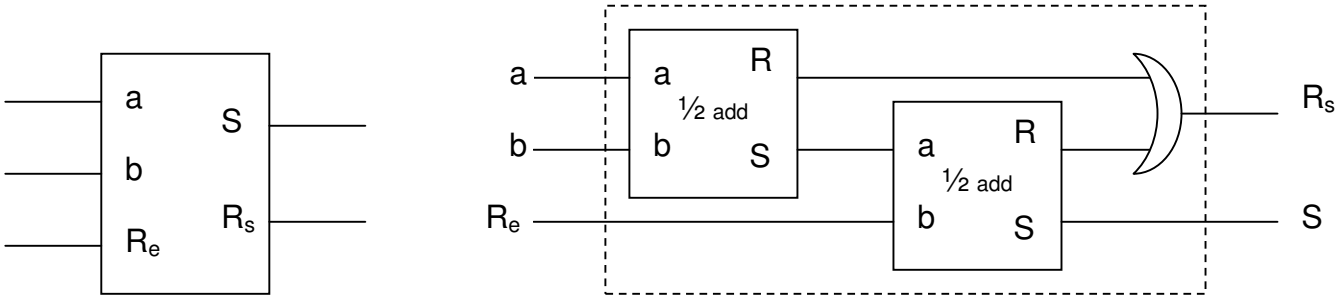


a	b	R	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

D'où le circuit :la somme est une porte Xor, la retenue est une porte et

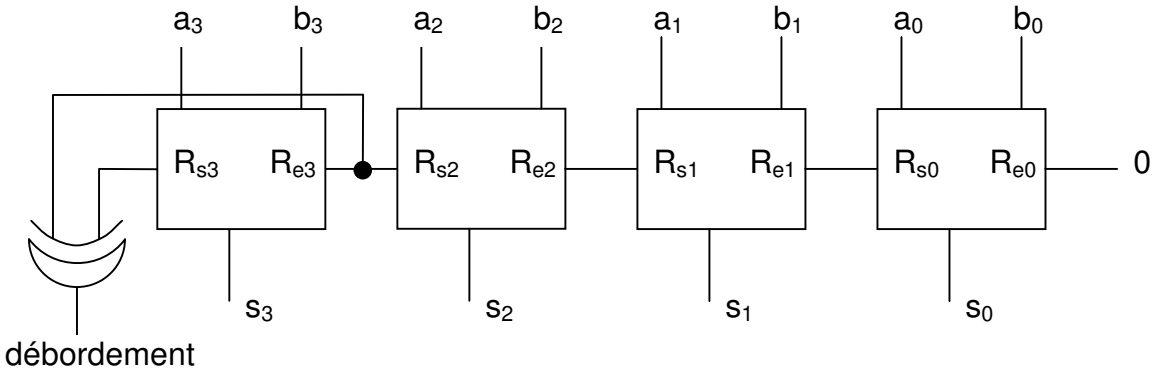


Additionneur complet (prend en compte une retenue antérieure, se réalise avec deux 1/2 additionneurs)



Avec Re = Retenue en entrée ; Rs = Retenue en sortie

Sur 4 bits :



Explication du circuit du débordement :
. il ne peut pas y avoir débordement si A et B sont de signes contraires
. si A et B sont de même signe, S doit être de même signe ; sinon il y a débordement

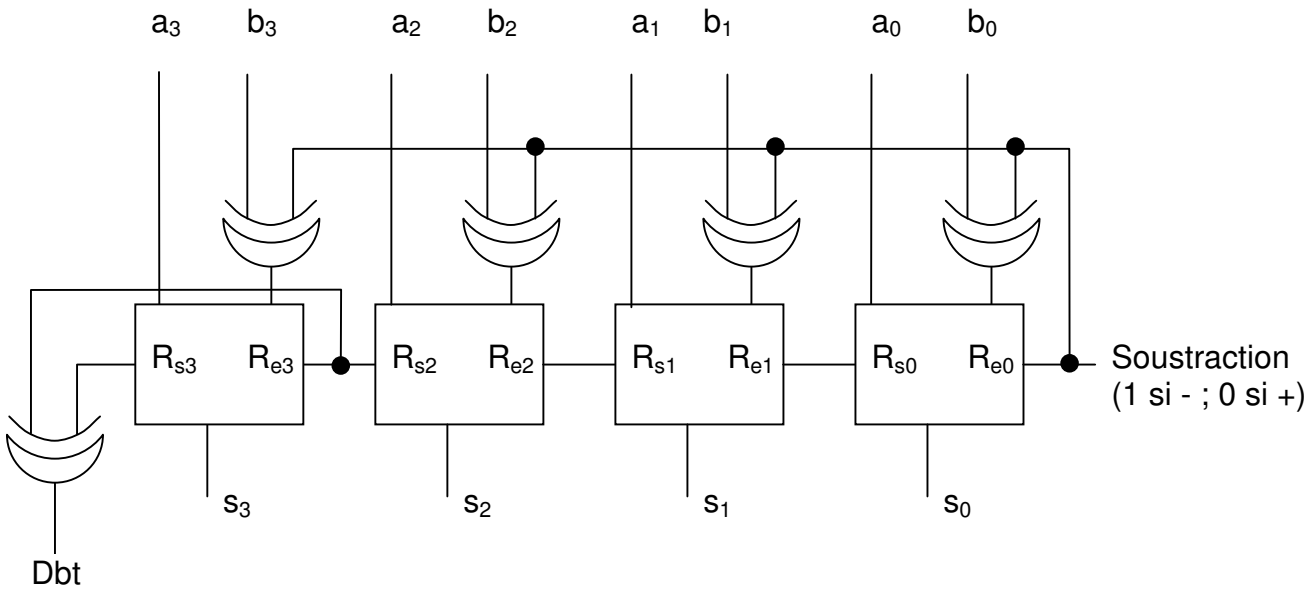
d'où la table :

a_3	b_3	R_{e3}	S_3	R_{s3}	Dbt	
0	0	0	0	0	0	A, B et S de même signe
0	0	1	1	0	1	A, B de même signe et non S
0	1	0	1	0	0	A, B de signes contraires
0	1	1	0	1	0	A, B de signes contraires
1	0	0	1	0	0	A, B de signes contraires
1	0	1	0	1	0	A, B de signes contraires
1	1	0	0	1	1	A, B de même signe et non S
1	1	1	1	1	0	A, B et S de même signe

d'où $Dbt=\overline{a}3\overline{b}3Re3+a3b3\overline{R}e3$

ou mieux : $Dbt=Re3\overline{R}s3+\overline{R}e3Rs3=Re3\oplus Rs3$ (on constate dans la table que $Dbt=1 \Leftrightarrow R_{e3}\neq R_{s3}$; on peut donc exprimer Dbt en fonction des seules valeurs de R_{e3} et R_{s3})

Additionneur-soustracteur

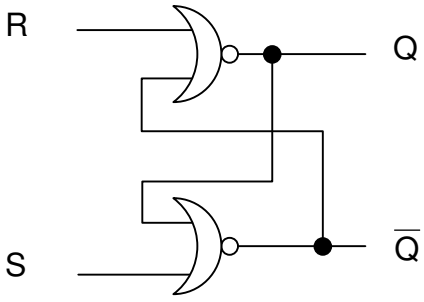


Dans le cas de la soustraction, le 1 en entrée assure à la fois le complément à 1 de B (Xor bit à bit) et le complément à 2 (retenue en entrée réalisant le +1)

Les bascules (flip-flop)

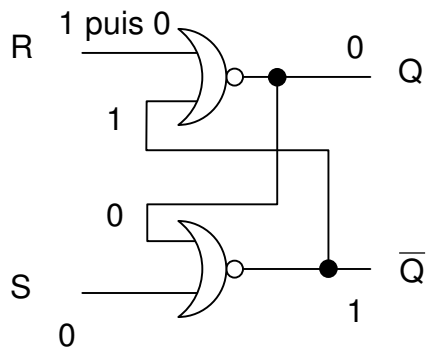
La bascule RS

(R=Reset, Set=Set)

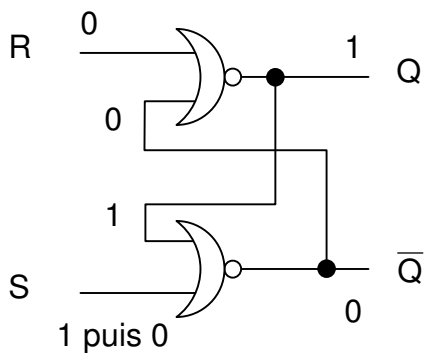


Comment ça marche ? Voyons ce qu'il se passe lorsqu'on applique un bref signal vrai, soit en entrée Set, soit en entrée Reset :

Reset :

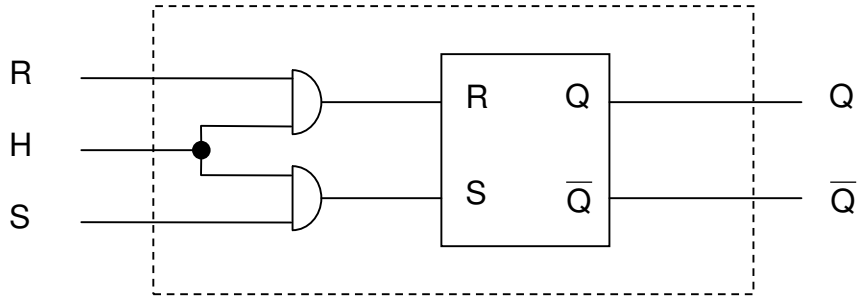


Set :



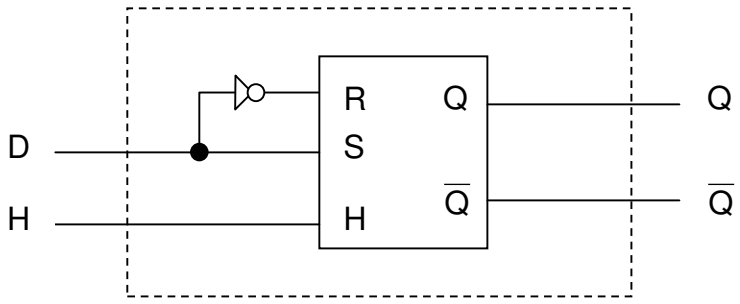
Q est donc positionné par un signal set ou reset, après quoi la bascule reste stable : il y a mémorisation d'un bit d'information.

La bascule RSH



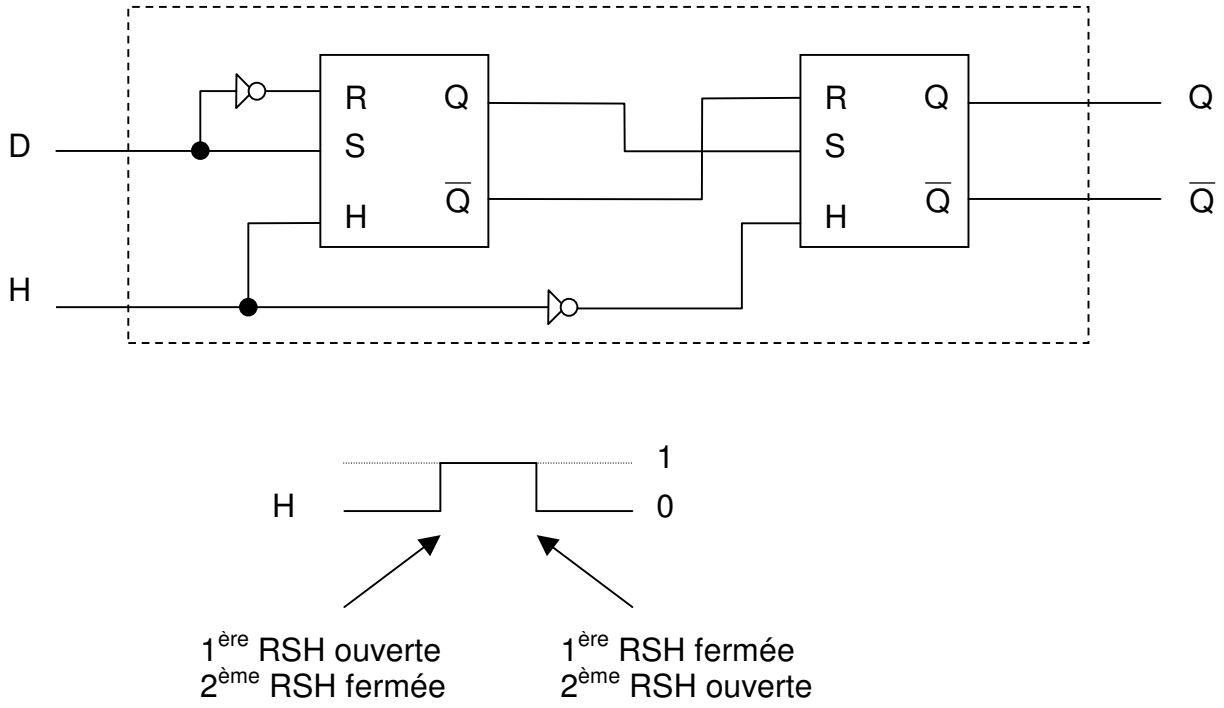
H est une autorisation d'écriture : un set ou un reset ne peut être validé que si H=1.
Application : en général, H=signal horloge, pour maîtriser les synchronisations.

La bascule D (bascule latche)



La donnée D en entrée est validée par H=1. La bascule mémorise D jusqu'au prochain top horloge et présente sa valeur en Q.

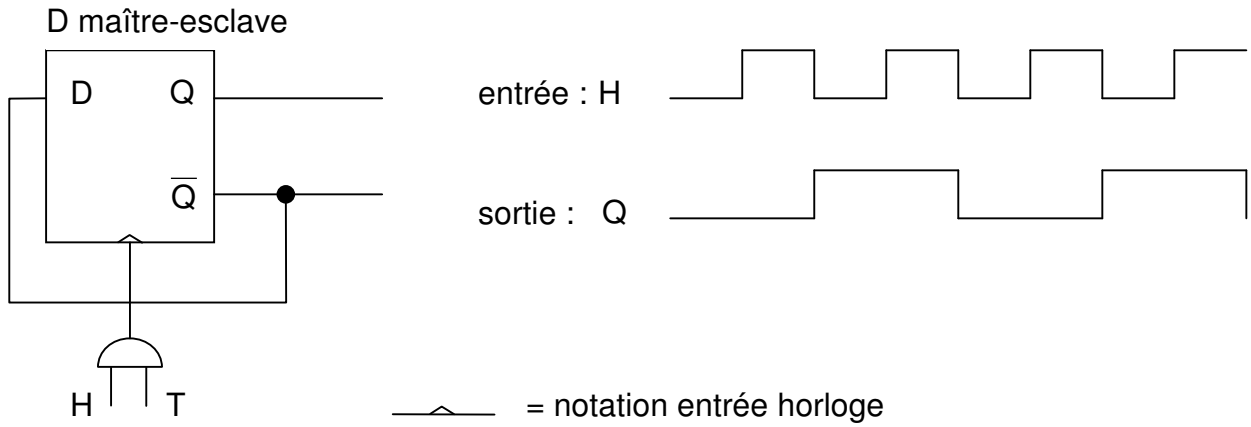
RSH maître-esclave, D maître-esclave



D est donc validée et mémorisée en Q sur le front descendant de H. On obtient un contrôle temporel précis.

Compteurs

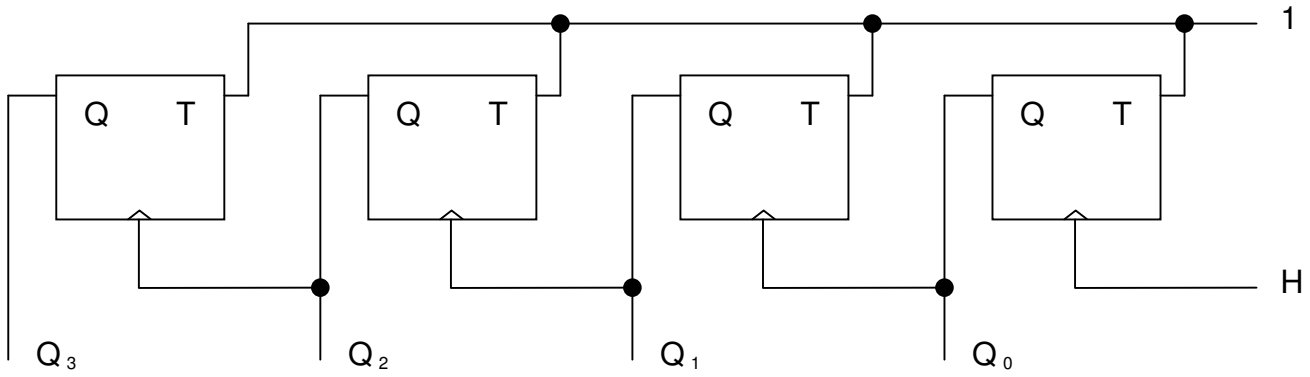
Bascule T (toggle)



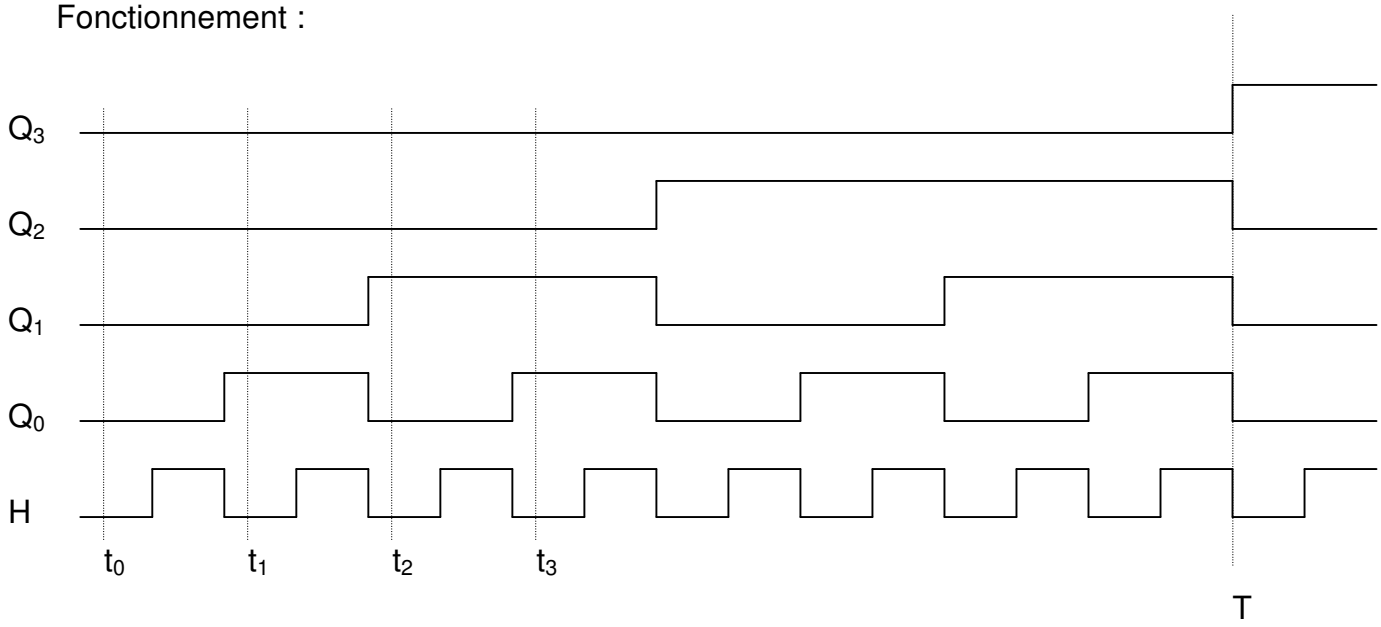
Q est initialement à zéro, donc $\overline{Q}$ est vrai ; D est donc vrai en entrée. D est validé en Q sur le front descendant de H, Q monte donc à 1. D passe désormais à faux ($\overline{Q}$), ce qui sera validé par la redescente de Q au second front descendant de H. T est une commande : vrai, la bascule fonctionne ; faux son état devient stable.

Compteur asynchrone

A base de bascules T (exemple sur 4 bits) :



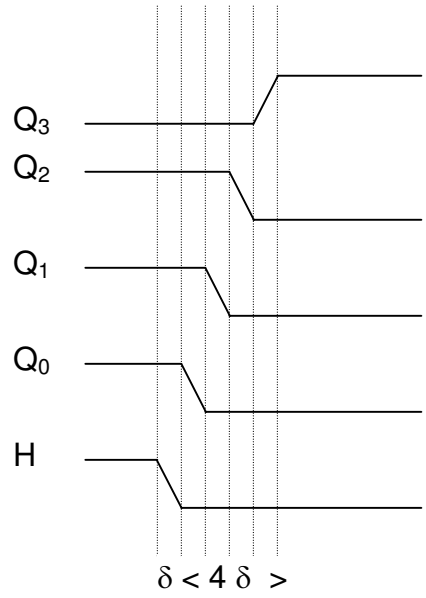
Fonctionnement :



	Q_3	Q_2	Q_1	Q_0	Q
t_0	0	0	0	0	0
t_1	0	0	0	1	1
t_2	0	0	1	0	2
t_3	0	0	1	1	3
…					…

Chaque bascule T fournit à sa suivante un signal d'horloge de fréquence divisée par deux. Les quatre bits de sortie Q_i composent un nombre Q qui "compte" les périodes d'horloge.

Inconvénient : le front d'horloge a une durée δ non nulle, si on agrandit la situation à l'instant T du schéma précédent :



entre le moment où H est passée à 0 et celui où les basculements sont validés jusqu'au bit de poids fort, il peut s'écouler jusqu'à N δ pour un compteur à N bits.

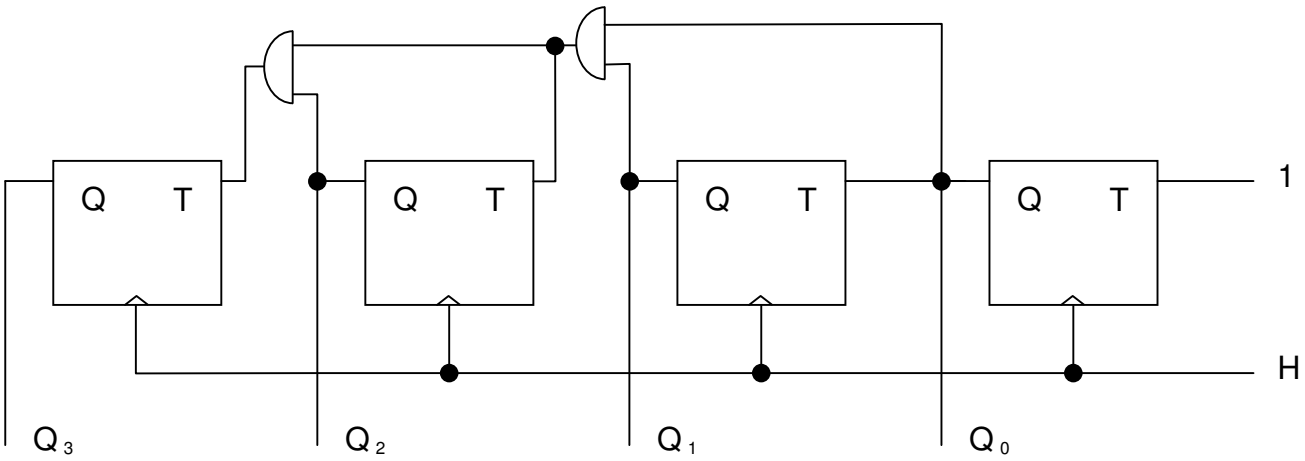
On doit donc tenir compte de la contrainte : $\tau > N\,\delta$

où τ est la période de l'horloge.

Ce qu'on peut traduire : si l'horloge va trop vite pour le compteur, celui-ci ne fonctionne plus !

Compteur synchrone

(exemple sur 4 bits) :

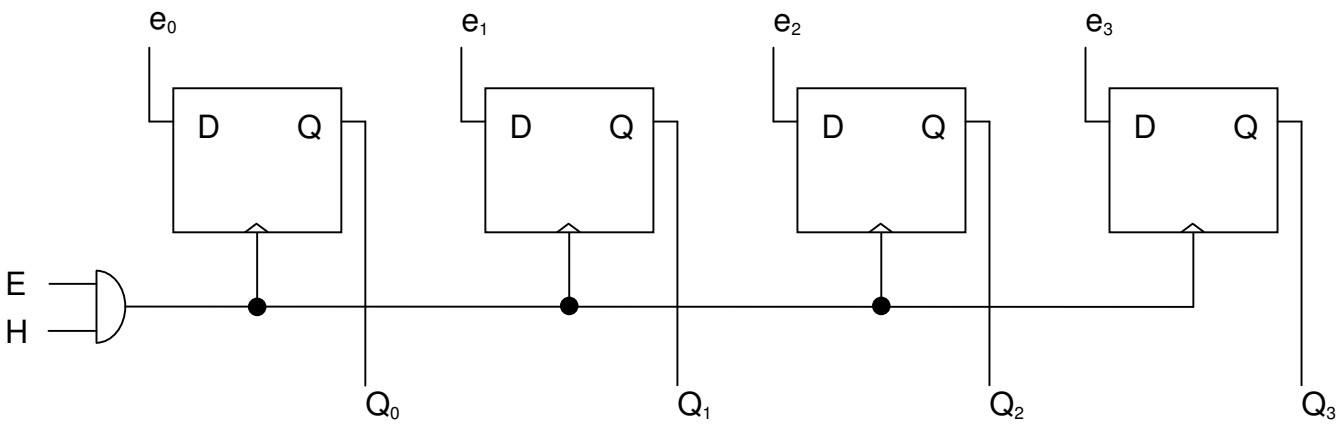


Il y a changement d'état d'une bascule seulement si toutes les bascules de poids inférieur sont à 1. On se sert des entrées T, contrairement au schéma précédent (tous les T étaient forcés à 1). Les bascules ont l'horloge en commun, les basculements sont donc synchrones.

Registres

On utilise des bascules D maître-esclave pour mémoriser un nombre.
Exemples sur 4 bits :

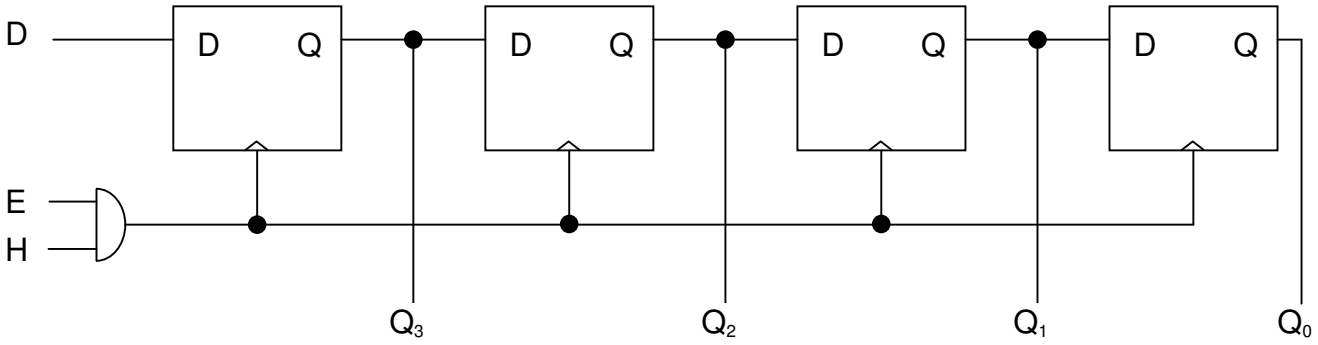
Chargement parallèle, lecture parallèle



E (enable) = autorisation d'écriture. Lorsqu'on valide E, l'entrée e est inscrite dans le registre. Lorsqu'on "coupe" E, le registre reste stable et donc mémorise la valeur précédente et la présente en Q.

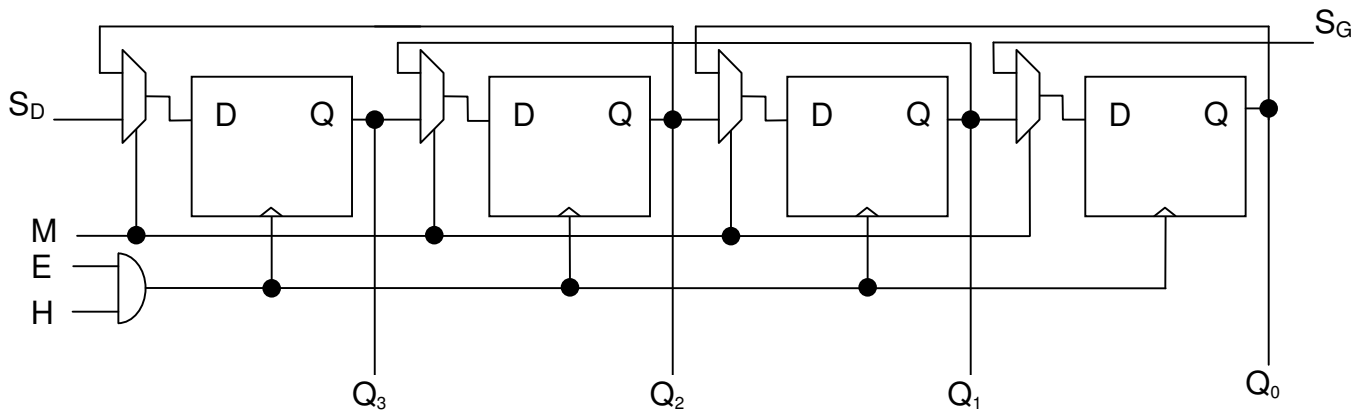
Registre à décalage

A droite :



Dans ce registre, la donnée D est entrée bit à bit (en série) depuis le registre de gauche, les bits sont décalés à droite à chaque front descendant d'horloge, tant que Enable est vrai.

A droite ou à gauche selon un fil de commande :

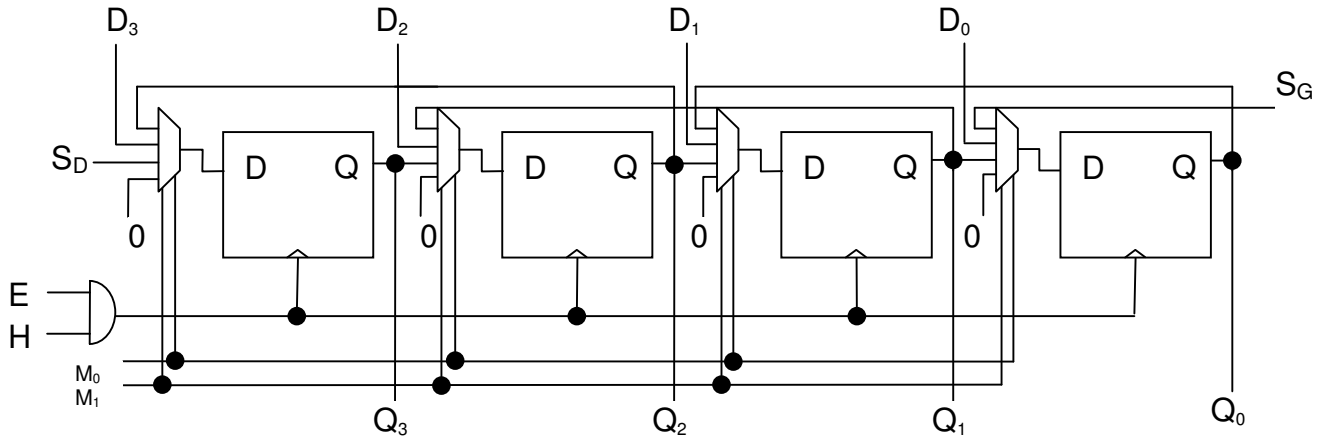


M : fil de commande des MPX
S_D : entrée Série décalage à Droite
S_G : entrée Série décalage à Gauche

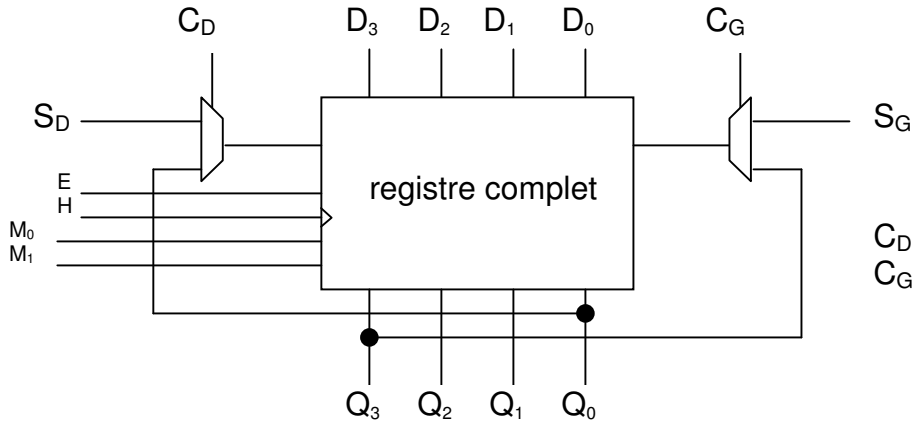
Nota : outre l'entrée de données en série, le décalage est utile en arithmétique. Par exemple, décaler d'un bit à gauche le nombre contenu dans le registre, avec un zéro en entrée, c'est le multiplier par deux.

Registre complet

décalage à gauche, décalage à droite, chargement parallèle, lecture parallèle, remise à zéro :



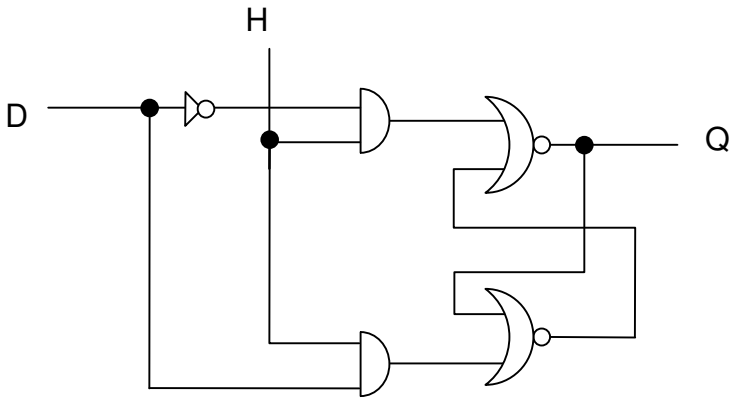
Registre avec décalage circulaire droite / gauche



C_D : circulaire droite
C_G : circulaire gauche

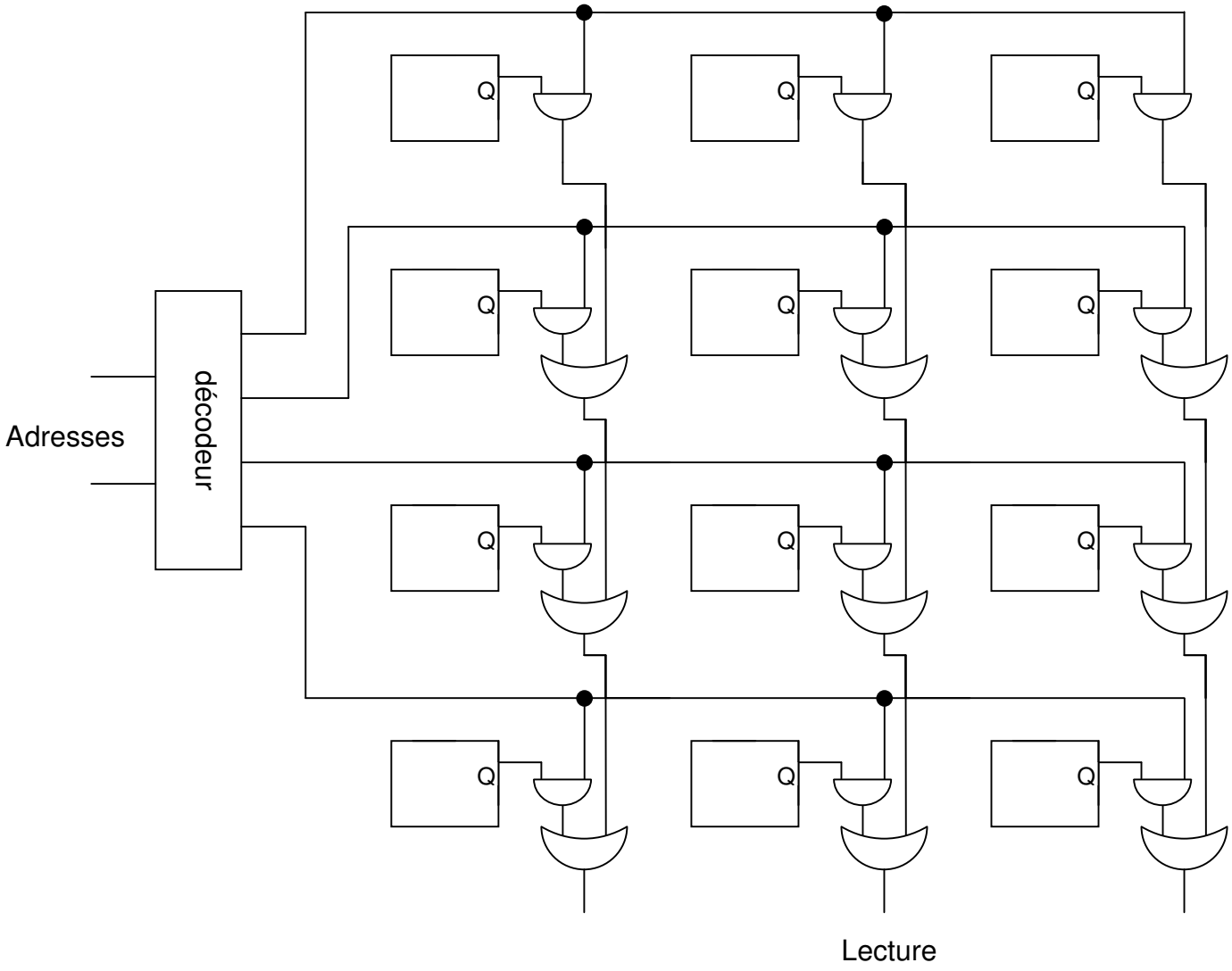
RAM (Random access memory)

... ou mémoire vive.
Exemple avec 4 mots de 3 bits. Les bascules utilisées sont des « D » simples :



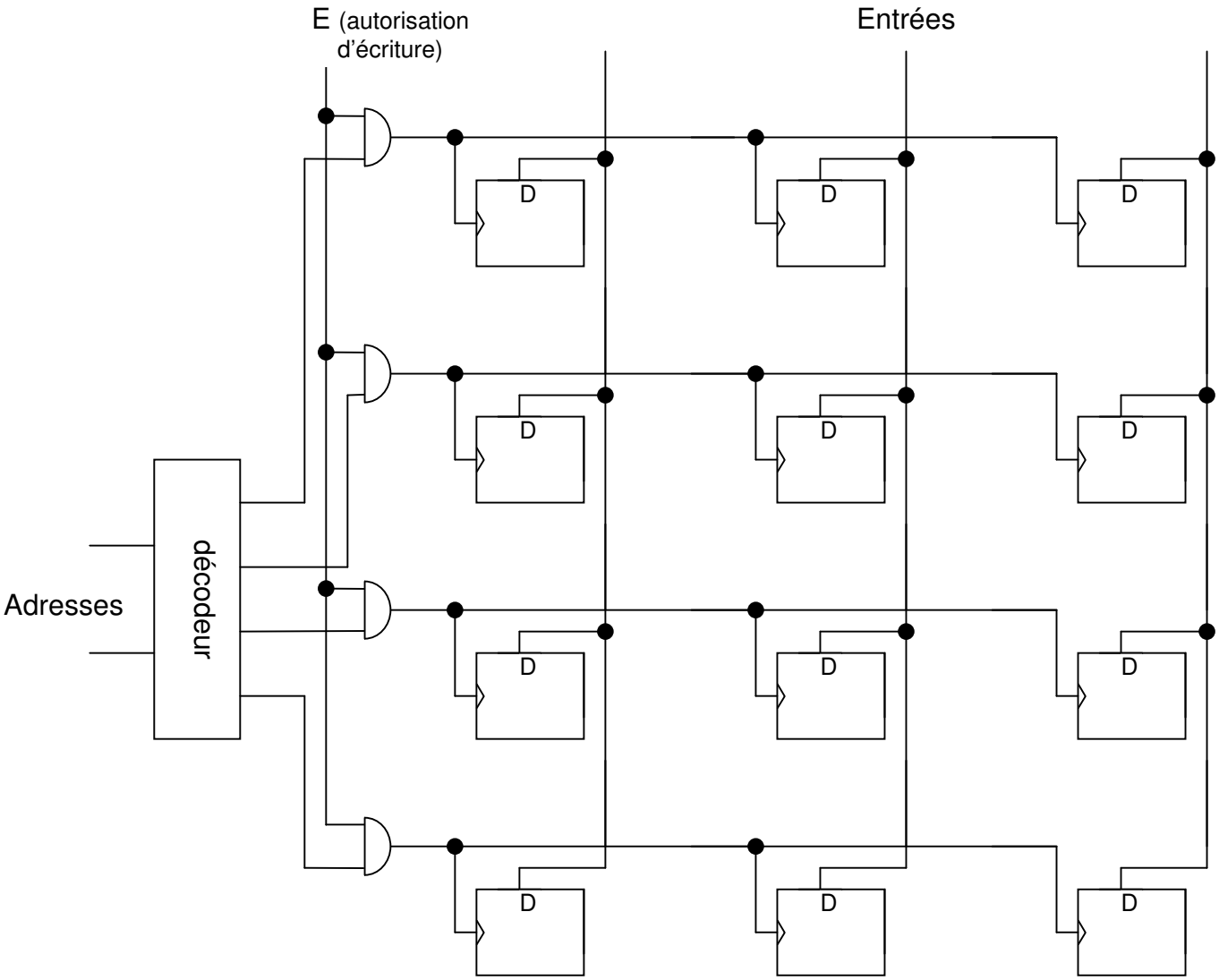
Circuit permettant la lecture

La donnée (en ligne) est sélectionnée par l'adresse décodée. Les adresses non lues sont inhibées par les portes et. Le résultat est propagé par les portes ou jusqu'en sortie.



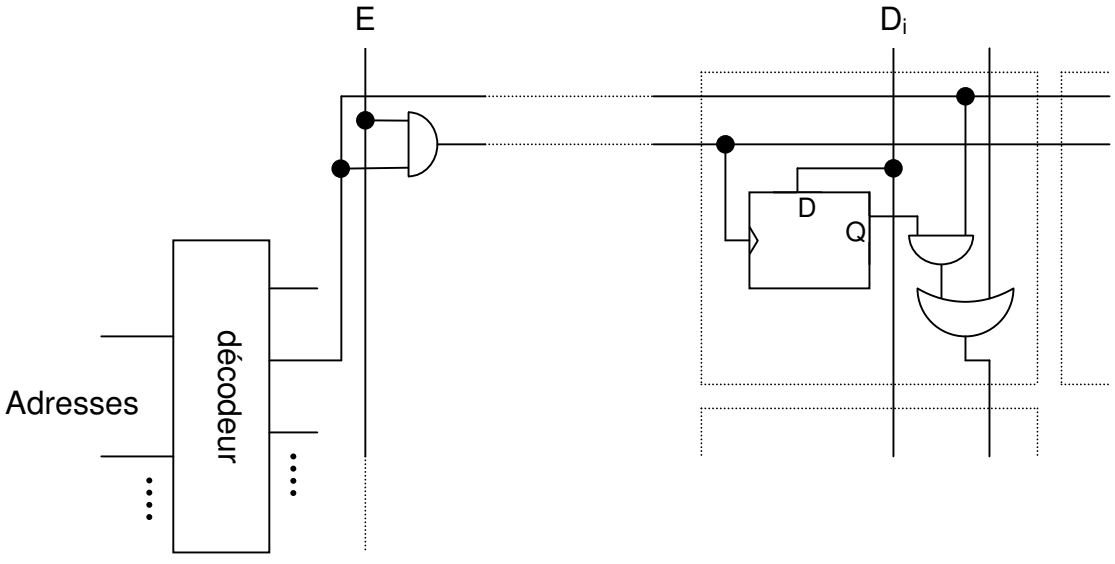
Circuit permettant l'écriture

Le décodeur d'adresse autorise l'écriture seulement sur la ligne choisie.



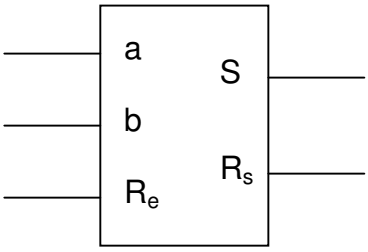
Elément complet de la RAM

c'est la superposition des deux schémas précédents, permettant lecture et écriture d'une donnée à une adresse :



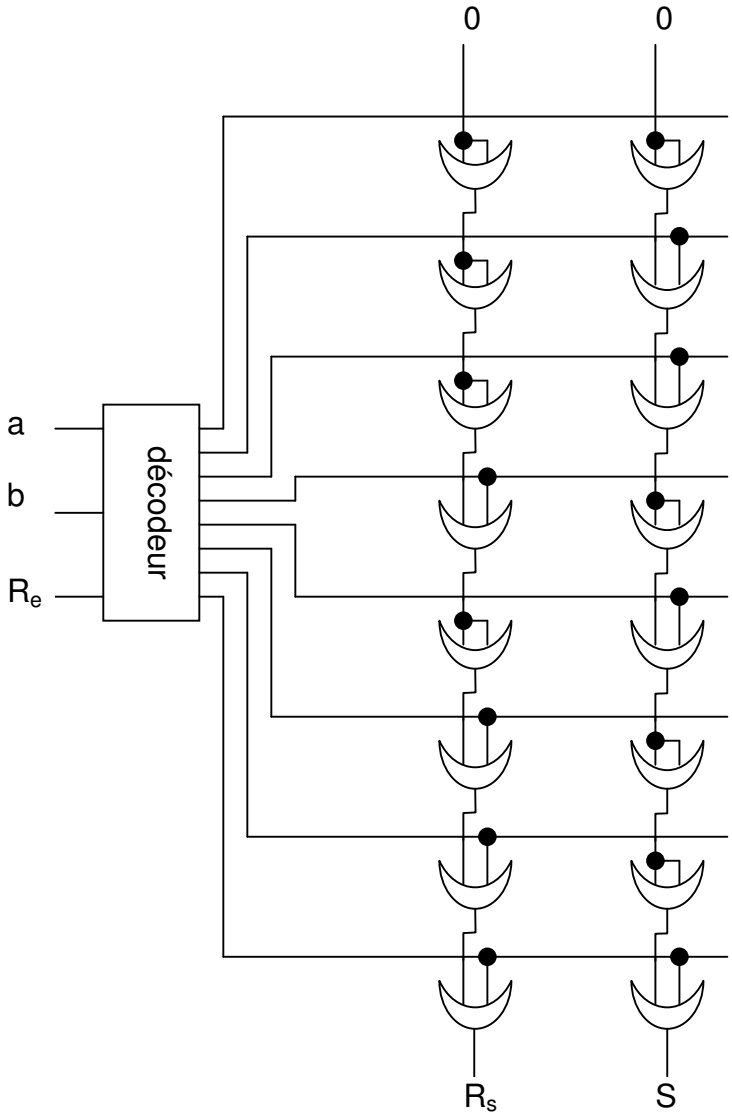
ROM (Read Only Memory)

Exemple de l'additionneur complet : comment le réaliser en ROM ?



a	b	R_e	R_s	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

On traduit les 0 ou 1 des colonnes résultat de la table de vérité par l'une ou l'autre configuration de porte "ou" comme suit :



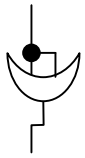
Chaque ligne du circuit traduit une ligne de la table de vérité, chaque colonne correspond à une colonne de résultat de la table.

Le signal en sortie du décodeur est à 1 sur une et une seule ligne, correspondant à la configuration des entrées.

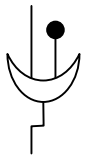
Le câblage des deux portes « ou » correspondant à cette ligne « active » exploite ou non ce signal à 1, selon la valeur attendue en sortie, lue dans la table.

Le résultat n'est pas affecté par les portes « ou » précédentes, toutes à 0, et est propagé jusqu'à la sortie du circuit par les portes « ou » successives, sans altération puisque les autres entrées des portes « ou » successives sont à 0.

Configuration d'une porte "ou" pour un 0 en sortie :
cette porte est neutre. Le signal étant initialisé à 0
(cf en haut du schéma) reste à cette valeur.



Configuration d'une porte "ou" pour un 1 en sortie :
le signal 1 est pris sur la sortie d'adresse décodée.
Si cette entrée "adresse" est à zéro, la porte est neutre
et laisse "descendre" l'information précédente.



Un circuit logique à n_e entrées et n_s sorties peut donc être réalisé avec une ROM de taille $2^{n_e} \times n_s$.

La réalisation logique étant simple et répétitive, la réalisation physique le sera aussi (logique de production industrielle).

L'exemple montré est un circuit ROM ayant une fonction de composant logique, mais n'importe quel type d'information peut être stocké en ROM (des instructions de code machine, des données figées ....).

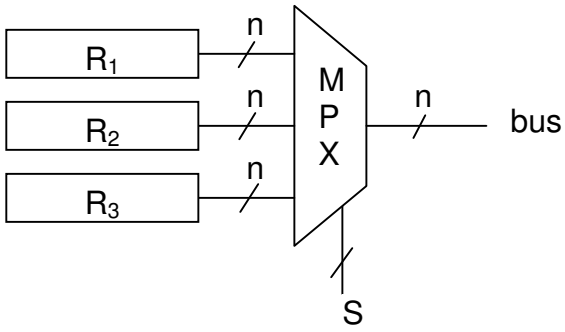
Dans tous les cas, il s'agit de fournir en sortie une donnée figée, selon l'adresse à laquelle on vient lire.

Cela se traduit aussi par une table de vérité : tous les bits (= toutes les colonnes) constituant la donnée souhaitée en sortie, selon l'adresse lue (= la ligne).

Bus, porte tri-state

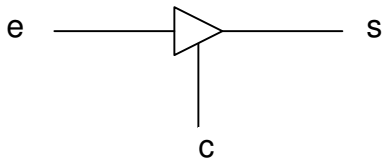
Nous avons dessiné de nombreux circuits, il s'agit maintenant de les faire communiquer : le bus véhicule les données entre les divers circuits.

Accès par MPX



Exemple de partage d'un bus à n bits par trois registres (n bits également).
Cette solution est coûteuse.

Porte tri-state

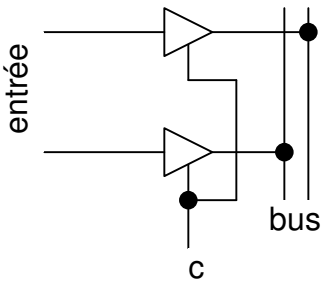


e = entrée
s = sortie
c = contrôle

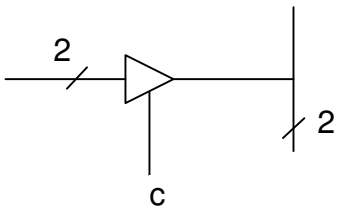
Fonctionnement logique de cette porte :
si c = 1 alors s = e (2 états possibles, 0 ou 1)
si c = 0 alors s = 3ème état neutre, ne forçant pas la sortie à une valeur quelconque

c	e	s
0	0	3ème état
0	1	3ème état
1	0	0
1	1	1

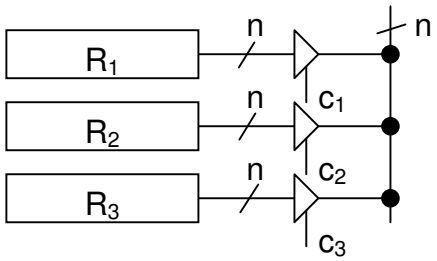
Exemple d'accès à un bus 2 bits :



noté :



Exemple précédent :



La machine simple

On y vient, nous allons dessiner un micro-processeur simple. On l'équipe avec :

Les registres : ce sont des 16 bits à entrée/sortie parallèle, avec un signal d'autorisation d'écriture.
On utilisera quatre registres notés R0, R1, R2, R3.
R0 servira d'accumulateur (registre « privilégié » sur lequel s'appuieront certaines opérations), et disposera en sortie d'un signal « test » vrai si R0 est à zéro, faux sinon.
R1 et R2 seront des registres de travail.
R3 sera le compteur ordinal, c'est à dire qu'il contiendra systématiquement l'adresse en RAM de la prochaine instruction à exécuter. Il est doté d'un signal en entrée d'incrémentation noté i_{R3} (R3 $\leftarrow$ R3 + 1 si i_{R3} vrai)

Unité arithmétique : on dispose d'un additioneur / soustracteur

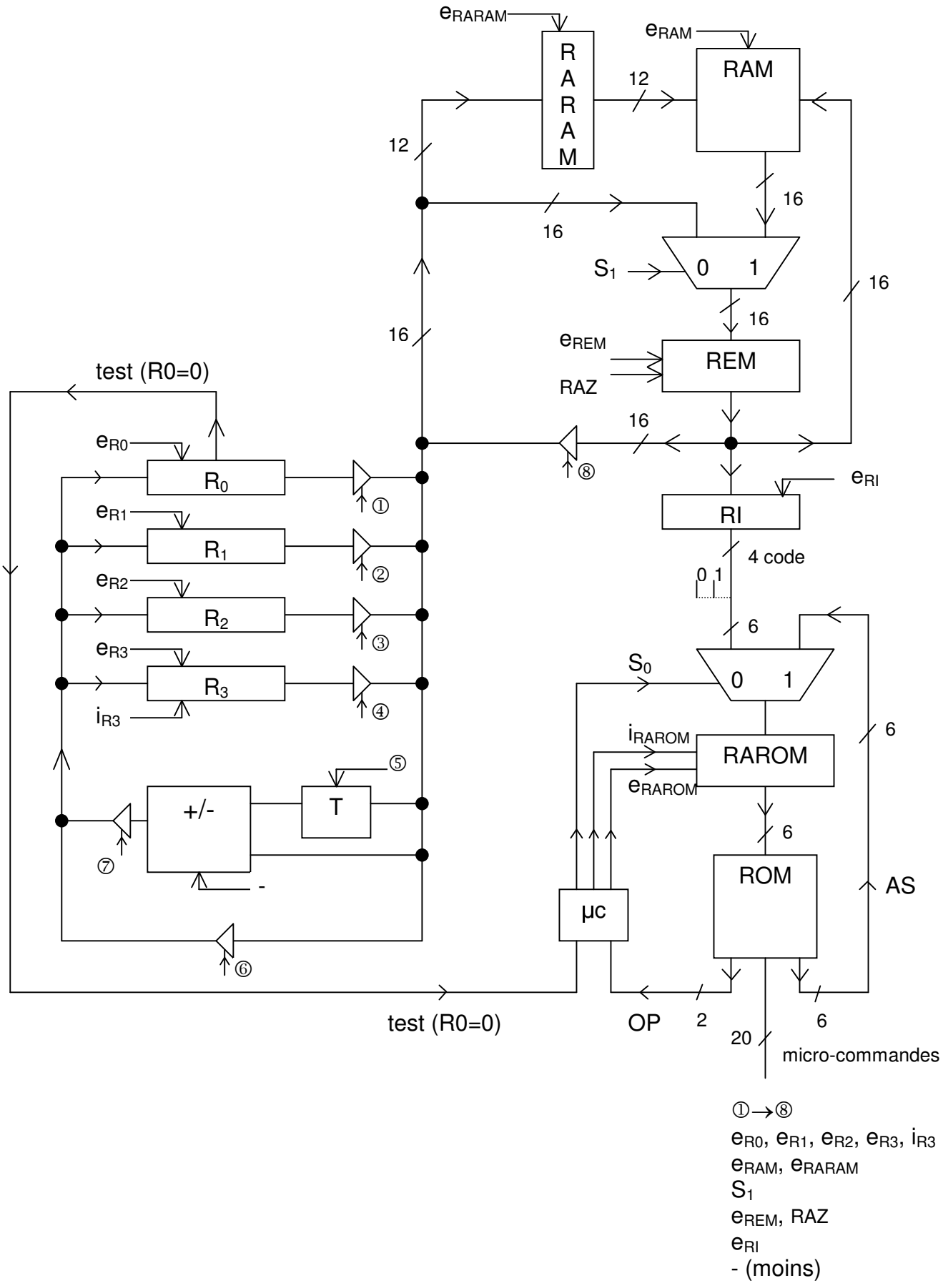
RAM : adressée sur 12 bits (2^{12} = 4096 adresses = 4 K mots adressables) ; elle stocke des mots de 16 bits. Elle contiendra les instructions machine codées par l'utilisateur (assembleur) et des données utilisateur variables.

ROM : c'est le concepteur de la machine simple (nous !) qui la micro-programmera pour réaliser les instructions assembleur dont on voudra se doter. Elle sera adressée sur 6 bits (2^6 = 64 adresses maxi).

Micro-contrôleur : c'est le circuit combinatoire qui nous permettra de contrôler le déroulement du micro-code en ROM.

En résumé : le micro-contrôleur (μc) permet de gérer la ROM, la ROM permet de gérer la RAM et les registres.

Schéma complet de la machine simple



Notations :

$e_{registre}$	= autorisation d'écriture sur le registre
$i_{registre}$	= incrémentation du registre
T	= toggle (sert de tampon : les données en entrées sont reproduites en sortie, un cycle d'horloge après le signal de toggle)
RARAM	= registre d'adressage de la RAM (sur 12 bits dans notre cas)
RAROM	= registre d'adressage de la ROM (sur 6 bits dans notre cas)
S	= sélection multiplexeur
REM	= registre d'échange mémoire (sur 16 bits, comme la RAM)
RI	= registre instruction (sert à adresser en ROM une instruction assembleur)
µc	= micro-contrôleur
OP	= opération confiée au µc (micro-contrôleur)
AS	= adresse suivante en ROM
RAZ	= remise à zéro (portant sur les 4 bits de poids fort du REM)

Le micro-contrôleur

Son rôle est de contrôler la séquence des exécutions d'instructions en ROM. On a besoin de pouvoir :

. exécuter des instructions en séquence,
. ou sauter à des adresses précodées en ROM (branchement inconditionnel),
. ou sauter selon l'état du signal test fourni par l'accumulateur R_0 (branchement conditionnel)
. ou enfin sauter à des adresses spécifiées par l'utilisateur, codées par lui en RAM (l'utilisateur a codé une mnémonique assembleur, traduite en code machine par l'adresse qu'on évoque). On appelle cette fonction du µc le branchement externe.

Ses entrées sont donc :

- un code OP sur 2 bits, identifiant l'une des 4 opérations de branchement à réaliser en ROM
- le signal test fourni par l'accumulateur R_0, qui servira à gérer le branchement conditionnel

Ses sorties sont : S_0, i_{RAROM}, e_{RAROM}

Table de vérité du µc :

OP	Mnémonique	Sorties
00	BS (branchement en séquence)	$i_{RAROM} \leftarrow 1$
01	BI (branchement inconditionnel)	$e_{RAROM} \leftarrow 1$, $S_0 \leftarrow 1$
10	BC (branchement conditionnel)	si test = 1 alors $i_{RAROM} \leftarrow 1$ sinon $e_{RAROM} \leftarrow 1$, $S_0 \leftarrow 1$
11	BE (branchement externe)	$S_0 \leftarrow 0$, $e_{RAROM} \leftarrow 1$

Nota : dans cette table ainsi que dans les suivantes, les sorties non précisées explicitement (sortie ← valeur) sont à 0.

BS : si le code OP (fourni par l'instruction en cours de la ROM) en entrée vaut 00, un branchement en séquence est demandé. Il suffit d'incrémenter le registre d'adresse en ROM, ce qui sera effectif au prochain cycle d'horloge, et la ROM présente désormais en sortie le contenu de son adresse suivante. On obtient l'exécution en séquence du micro-code en ROM.

BI : si le code OP vaut 01, un branchement inconditionnel est demandé. La ROM ne fournit pas que le code OP à 01, elle fournit également les 6 bits d'information de l'adresse en ROM à laquelle on veut sauter. On "aiguille" donc le sélecteur S_0 pour que cette adresse vienne alimenter le registre d'adresse ROM, sur lequel il reste à autoriser l'écriture. Au prochain cycle d'horloge validant l'écriture, la nouvelle adresse ROM est effective. AS transite donc via le MPX ($S_0 \leftarrow 1$) et est chargée en RAROM ($e_{RAROM} \leftarrow 1$). On obtient l'exécution d'un « goto AS » où AS est une adresse en ROM.

BC : si le code OP vaut 10 et que test est vrai (il y a zéro dans l'accumulateur), on décide d'exécuter l'instruction ROM suivante, si test est faux on veut gérer cet autre cas différemment, donc on se branche à une autre adresse spécifiée par l'instruction en ROM. En résumé, suivant la valeur de « test », on réalise un BS ou un BI.

BE : enfin, si le code OP est 11, on laisse venir l'adresse voulue spécifiée dans l'instruction assembleur écrite par l'utilisateur. Une instruction assembleur sera codée en RAM sur un mot (16 bits) avec 4 bits identifiant l'instruction (notre assembleur sera donc limité à au plus 16 instructions différentes) et 12 bits de données (une adresse en RAM, par exemple). Le code instruction permettra un accès direct, en ROM, au micro-code qui réalisera l'instruction. Nous utiliserons les premières adresses en ROM pour du micro-code d'intérêt commun ou un peu

de place libre réservée, le micro-code d'exécution des instructions assembleur sera placé à partir de l'adresse 16, soit :

010000 = adresse 16, instruction assembleur 0000
010001 = adresse 17, instruction assembleur 0001
etc ……

Les 4 bits de poids faible sont donc lus dans le registre d'instruction RI, les 2 bits de poids fort sont forcés à 01.

Dans le cas du branchement externe, l'adresse en ROM ainsi constituée transite via le MPX (S0 ← 0) et est chargée en RAROM (e_{RAROM} ← 1) moyennant un cycle d'horloge.

Après cette opération BE, le micro-code en ROM de l'instruction chargée dans RI est prêt à s'exécuter (il est présenté en sortie de la ROM).

Micro-programmation

Le jeu d'instructions

On veut se doter des instructions assembleur suivantes :

Code	Mnémonique		Signification	adresse en ROM
0000	MOV	R0, R1	R1 ← R0	16
0001	MOV	R0, R2	R2 ← R0	17
0010	MOV	R1, R2	R2 ← R1	18
0011	MOV	R1, R0	R0 ← R1	19
0100	MOV	R2, R0	R0 ← R2	20
0101	MOV	R2, R1	R1 ← R2	21
0110	LD	R0, adr	R0 ← MEM(adr)	22
0111	ST	R0, adr	MEM(adr) ← R0	23
1000	LDI	R0, val	R0 ← val	24
1001	B	adr	R3 ← adr	25
1010	BZ	R0, adr	R3 ← adr si R0=0	26
1011	ADD	R0, R1	R1 ← R1 + R0	27
1100	SUB	R0, R1	R1 ← R1 – R0	28

Le codage en ROM

Nous avions réservé les premières adresses. Aux adresses 0 à 2, on place le micro-code permettant de passer à l'instruction assembleur suivante, dont l'adresse en RAM se trouve dans le compteur ordinal R3. Aide à la lecture :

- adresse 0 :
. on publie l'adresse contenue dans R3 sur le bus en ouvrant la porte 4, et on autorise l'écriture dans le registre d'adresses RAM. La RAM "pointe" désormais sur l'instruction suivante, mais on n'a pas fini notre travail. On passe à l'instruction suivante en codant le signal BS à l'attention du µc.
. on exécute maintenant le micro-code d'adresse 1, donc on aiguille le multiplexeur S1 pour positionner le contenu lu en RAM (instruction assembleur suivante) dans le registre d'échange mémoire, en autorisant son écriture. On n'a toujours pas fini, donc on continue (BS).
. on exécute maintenant le micro-code d'adresse 2, qui copie le code de l'instruction assembleur lue dans le registre instruction (en autorisant l'écriture). La commande BE envoyée au µc l'instruit d'exécuter au prochain cycle horloge cette instruction. Par ailleurs, dans le même cycle d'exécution (ligne d'adresse 2), on incrémente le compteur ordinal R_3, ce qui prépare la machine simple pour l'exécution de l'instruction assembleur suivante (à aller chercher en RAM).

On remarquera que le déroulement du micro-code en ROM étant rythmé par l'horloge de la machine simple, on doit passer à une ligne d'instruction suivante à chaque fois que les données transitent par un registre ou la RAM.

Aux adresse 16 à 28, on trouve le début des codes réalisant les instructions assembleur, ou l'instruction complète selon sa complexité.

On a déjà vu que la ROM serait limitée à 64 entrées (adresses sur 6 bits), ce sera suffisant puisque la plus haute adresse utilisée est 28.

La ROM aura par ailleurs 28 sorties : l'adresse suivante en ROM (AS) soit 6 bits, le code opération (OP) destiné au µc soit 2 bits, et les 20 micro-commandes (signaux de contrôle des différents éléments).

La réalisation matérielle de la ROM ne pose aucune difficulté dès lors qu'on a établi sa table de vérité : 6 bits d'adresse en entrée d'un décodeur, 64 lignes (pas toutes utilisées), 28 colonnes en sortie.

Table de vérité de la ROM :

Adresse	AS (6bits)	OP (2bits)	Micro-commandes (20 bits)
0		BS	④ ← 1, e_{RARAM} ← 1
1		BS	S_1 ← 1, e_{REM} ← 1
2		BE	e_{RI} ← 1, i_{R3} ← 1
3		BS	S_1 ← 1, e_{REM} ← 1
4	0	BI	⑧ ← 1, ⑥ ← 1, e_{R0} ← 1
5		BS	① ← 1, S_1 ← 0, e_{REM} ← 1
6	0	BI	e_{RAM} ← 1
7	0	BI	⑧ ← 1, ⑥ ← 1, e_{R0} ← 1
8	0	BC	
9	25	BI	
10	0	BI	① ← 1, ⑦ ← 1, e_{R1} ← 1
11	0	BI	① ← 1, - ← 1, ⑦ ← 1, e_{R1} ← 1
12			
13			
14			
15			
16	0	BI	① ← 1, ⑥ ← 1, e_{R1} ← 1
17	0	BI	① ← 1, ⑥ ← 1, e_{R2} ← 1
18	0	BI	② ← 1, ⑥ ← 1, e_{R2} ← 1
19	0	BI	② ← 1, ⑥ ← 1, e_{R0} ← 1
20	0	BI	③ ← 1, ⑥ ← 1, e_{R0} ← 1
21	0	BI	③ ← 1, ⑥ ← 1, e_{R1} ← 1
22	3	BI	⑧ ← 1, e_{RARAM} ← 1
23	5	BI	⑧ ← 1, e_{RARAM} ← 1
24	7	BI	RAZ ← 1
25	0	BI	⑧ ← 1, ⑥ ← 1, e_{R3} ← 1
26	8	BI	
27	10	BI	② ← 1, ⑤ ← 1
28	11	BI	② ← 1, ⑤ ← 1

Lisons ensemble cette table de vérité, pour les principales instructions :

- adresse 16, instruction assembleur 0000 : on veut écrire dans R_1 le contenu de l'accumulateur R_0. On ouvre donc la porte 1, ce qui publie le contenu de R_0 sur le bus, on ouvre la porte 6 pour que la donnée soit présentée en entrée des registres, et on autorise l'écriture uniquement sur le registre cible, R_1. Le travail est fini, on peut passer à l'instruction suivante, ce qu'on a pré-codé en adresse 0 : on effectue donc un branchement inconditionnel (BI) vers cette adresse.

- les cinq instructions suivantes, adresses 17 à 21, sont similaires.

- adresse 22 : on veut charger l'accumulateur avec une donnée contenue à une adresse RAM. On a vu, en détaillant précédemment le µcode servant à passer à l'instruction suivante, que (en étape 2, cf ligne d'adresse 1 de la ROM) l'instruction assembleur était stockée dans le registre d'échange mémoire. 4 bits sur 16 identifient l'opération (0110 pour notre instruction "LD R0, adr"), les 12 bits restants sont à disposition de l'utilisateur qui programme en assembleur, c'est là qu'est codée l'adresse qu'on veut lire. On ouvre donc la porte 8 pour publier cette adresse vers le registre d'adresse RAM, qu'on autorise à écriture. La RAM est désormais prête pour, au prochain cycle d'horloge, founir la donnée voulue. Le travail n'est pas fini, il faut passer à une autre instruction, mais pas en séquence : l'adresse suivante est réservée pour l'instruction assembleur 23 ("ST R0, adr"). On réalise donc un branchement à la première adresse libre, soit 3, et on continue à coder :
. adresse 3 : on ouvre la porte S_1, pour écrire la donnée contenue à l'adresse souhaitée dans le registre d'échange mémoire, qu'on autorise en écriture. L'instruction assembleur qui y était précédemment contenue est écrasée, mais on en a déjà tiré toutes les informations utiles. On passe au cycle d'horloge et à l'instruction suivants (BS).
. adresse 4 : on ouvre la porte 8 pour publier la donnée sur le bus, la porte 6 pour la présenter aux registres, et on autorise l'écriture sur l'accumulateur R_0. Le travail est fini, on peut passer à l'instruction suivante (BI 0).

- adresse 23 : c'est l'instruction symétrique de la précédente, on commence de la même façon en positionnant la RAM sur l'adresse qui nous intéresse, et on passe à la suite :
. adresse 5 : on publie la donnée à stocker (venant de l'accumulateur) en ouvrant la porte 1, on la stocke dans le registre d'échange mémoire en aiguillant le signal de contrôle MPX S_1 à 0, et en autorisant l'écriture sur le REM. On n'a pas fini, branchement en séquence donc.
. adresse 6 : on autorise l'écriture en RAM (précédemment positionnée à la bonne adresse), c'est fini on passe à l'instruction assembleur suivante (BI 0).

- adresse 24 : on veut charger une valeur dans l'accumulateur. La valeur a été codée par l'utilisateur, dans les 12 bits disponibles de son instruction assembleur. Elle est donc disponible dans le registre d'échange mémoire (positionné au chargement de l'instruction, comme déjà vu). Cependant, les 4 bits de poids fort contiennent le code de l'instruction ; pour que les 16 bits de données soient à la valeur souhaitée il faut remettre à zéro ces 4 bits. On n'a plus besoin de leur valeur (c'était une adresse de départ en ROM, or le µcode est déjà en train de s'exécuter depuis cette adresse). Le travail n'est pas fini, on passe à la ligne 7
. adresse 7 : on publie la donnée vers les registres en ouvrant les portes 8 et 6, et on autorise l'écriture sur l'accumulateur R_0. Le travail est fini, on peut passer à l'instruction suivante (BI 0).

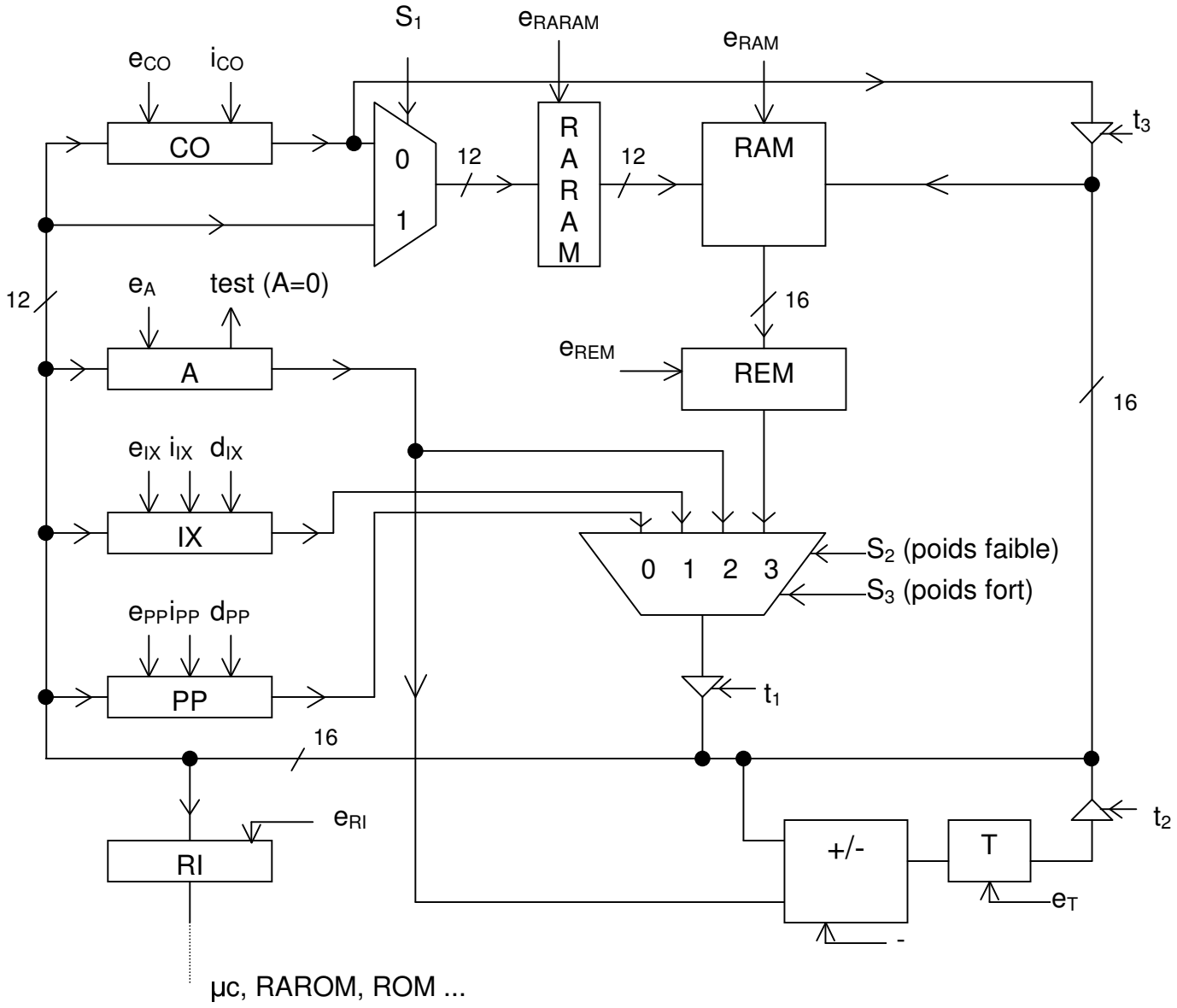
- adresse 25 : c'est une instruction "Branch" à une adresse (mnémonique "B adr"), il suffit de charger le registre compteur ordinal R_3 avec l'adresse contenue dans le registre d'échange mémoire. Il est inutile de remettre à zéro les 4 bits de code instruction, puisque de toutes façons R_3 ne publie ses données que vers le registre d'adresse RARAM, alimenté par un bus "tronqué" à 12 bits.

- adresse 26 : pour coder le test (susceptible de générer un branchement en séquence), une ligne ne suffit pas, et la ligne 27 correspond à l'adresse d'une autre instruction. On "part" donc de suite là où on a plus de place (BI 8).
. adresse 8 donc : Si R0 est nul (mnémonique BZ = Branch on Zero), on veut sauter à l'adresse fournie.
Dans ce cas, puisque R_0 est à zéro, le signal "test" est à 1. On décide donc d'exécuter l'instruction ROM suivante (le µc a été configuré ainsi), donc la ligne d'adresse 9, qui branche inconditionnellement à l'adresse 25 : on réutilise le code de l'instruction déjà écrite, qui effectue le branchement à l'adresse.
Revenons à la ligne 8 : dans le cas contraire, R_0 non nul, on ne veut pas sauter à l'adresse inscrite dans l'instruction assembleur, mais passer en séquence à l'instruction suivante : le travail est fini, l'adresse cible du branchement conditionnel BC en ligne 8 est donc 0.

- adresse 27 : pour additionner le contenu de l'accumulateur au contenu de R_1 (et stocker le résultat dans R_1), on ouvre la porte 2 pour publier le contenu de R_1, et la 5 pour stocker la valeur dans le registre Toggle. C'est pas fini, on continue en ligne 10.
. adresse 10 : on publie le contenu de l'accumulateur R_0 en ouvrant la porte 1, l'additionneur/soustracteur a désormais les deux valeurs en entrées ; on ouvre la porte 7 pour publier le résultat vers les registres, et on autorise l'écriture sur R_1. C'est fini, branchement en 0 pour passer à l'instruction suivante.

- adresse 28 : pour soustraire, c'est le même principe que l'instruction précédente, avec en sus le positionnement du signal de soustraction (moins) en entrée de l'additionneur/soustracteur.

Machine simple n°2



RAM = 4 K mots de 16 bits, adressés sur 12 bits
Format d'instruction : code OP sur 4 bits, adresse sur 12 bits
Registres : CO et RARAM sur 12 bits, les autres sur 16 bits

CO = compteur ordinal
A = accumulateur
IX = registre index
PP = pointeur de pile
RI = registre instruction

Le micro-contrôleur (non représenté sur le schéma, ainsi que la ROM et son registre d'adresse) est identique à celui de la machine précédente.

Le jeu d'instructions :

Code	Mnémonique		Signification	adr en ROM
0000	LD	A, adr	A ← MEM(adr)	16
0001	ST	A, adr	MEM(adr) ← A	17
0010	LD	IX, A	IX ← A	18
0011	LD	PP, A	PP ← A	19
0100	LDX	A	A ← MEM(IX)	20
0101	STX	A	MEM(IX) ← A	21
0110	ADD	A, adr	A ← A+MEM(adr)	22
0111	SUB	A, adr	A ← A-MEM(adr)	23
1000	CALL	adr	PP ← PP-1 ; MEM(PP) ← CO ; CO ← adr	24
1001	RET		CO ← MEM(PP) ; PP ← PP+1	25
1010	PUSH	A	PP ← PP-1 ; MEM(PP) ← A	26
1011	POP	A	A ← MEM(PP) ; PP ← PP+1	27
1100	INC	IX	IX ← IX+1	28
1101	DEC	IX	IX ← IX-1	29
1110	BZ	adr	CO ← adr si A=0	30
1111	B	adr	CO ← adr	31

Micro-programmation : à vous de jouer ! (solution page suivante)

Précision : le multiplexeur central a deux signaux de contrôle, S_2 et S_3. S_3 est le bit de poids fort, S_2 celui de poids faible. Ainsi, l'entrée sélectionnée est :

0 exprimé en binaire = 00 : entrée 0 si S_3=0 et S_2=0
1 exprimé en binaire = 01 : entrée 0 si S_3=0 et S_2=1
2 exprimé en binaire = 10 : entrée 0 si S_3=1 et S_2=0
3 exprimé en binaire = 11 : entrée 0 si S_3=1 et S_2=1

Table de vérité de la ROM :

Adresse	AS (6bits)	OP (2bits)	Micro-commandes (21 bits)
0		BS	$S_1 \leftarrow 0$, $e_{RARAM} \leftarrow 1$
1		BS	$e_{REM} \leftarrow 1$
2		BE	$S_2 \leftarrow 1$, $S_3 \leftarrow 1$, $t_1 \leftarrow 1$, $e_{RI} \leftarrow 1$, $i_{CO} \leftarrow 1$
3		BS	$e_{REM} \leftarrow 1$
4	0	BI	$S_2 \leftarrow 1$, $S_3 \leftarrow 1$, $t_1 \leftarrow 1$, $e_A \leftarrow 1$
5	0	BI	$S_2 \leftarrow 0$, $S_3 \leftarrow 1$, $t_1 \leftarrow 1$, $e_{RAM} \leftarrow 1$
6		BS	$e_{REM} \leftarrow 1$
7		BS	$S_2 \leftarrow 1$, $S_3 \leftarrow 1$, $t_1 \leftarrow 1$, $e_T \leftarrow 1$
8	0	BI	$t_2 \leftarrow 1$, $e_A \leftarrow 1$
9		BS	$e_{REM} \leftarrow 1$
10	8	BI	$S_2 \leftarrow 1$, $S_3 \leftarrow 1$, $t_1 \leftarrow 1$, - $\leftarrow 1$, $e_T \leftarrow 1$
11		BS	$S_2 \leftarrow 0$, $S_3 \leftarrow 0$, $t_1 \leftarrow 1$, $S_1 \leftarrow 1$, $e_{RARAM} \leftarrow 1$
12		BS	t3 $\leftarrow 1$, $e_{RAM} \leftarrow 1$
13	0	BI	$S_2 \leftarrow 1$, $S_3 \leftarrow 1$, $t_1 \leftarrow 1$, $e_{CO} \leftarrow 1$
14		BS	$e_{REM} \leftarrow 1$
15	0	BI	$S_2 \leftarrow 1$, $S_3 \leftarrow 1$, $t_1 \leftarrow 1$, $e_{CO} \leftarrow 1$, $i_{PP} \leftarrow 1$
16	3	BI	$S_2 \leftarrow 1$, $S_3 \leftarrow 1$, $t_1 \leftarrow 1$, $S_1 \leftarrow 1$, $e_{RARAM} \leftarrow 1$
17	5	BI	$S_2 \leftarrow 1$, $S_3 \leftarrow 1$, $t_1 \leftarrow 1$, $S_1 \leftarrow 1$, $e_{RARAM} \leftarrow 1$
18	0	BI	$S_2 \leftarrow 0$, $S_3 \leftarrow 1$, $t_1 \leftarrow 1$, $e_{IX} \leftarrow 1$
19	0	BI	$S_2 \leftarrow 0$, $S_3 \leftarrow 1$, $t_1 \leftarrow 1$, $e_{PP} \leftarrow 1$
20	3	BI	$S_2 \leftarrow 1$, $S_3 \leftarrow 0$, $t_1 \leftarrow 1$, $S_1 \leftarrow 1$, $e_{RARAM} \leftarrow 1$
21	5	BI	$S_2 \leftarrow 1$, $S_3 \leftarrow 0$, $t_1 \leftarrow 1$, $S_1 \leftarrow 1$, $e_{RARAM} \leftarrow 1$
22	6	BI	$S_2 \leftarrow 1$, $S_3 \leftarrow 1$, $t_1 \leftarrow 1$, $S_1 \leftarrow 1$, $e_{RARAM} \leftarrow 1$
23	9	BI	$S_2 \leftarrow 1$, $S_3 \leftarrow 1$, $t_1 \leftarrow 1$, $S_1 \leftarrow 1$, $e_{RARAM} \leftarrow 1$
24	11	BI	$d_{PP} \leftarrow 1$
25	14	BI	$S_2 \leftarrow 0$, $S_3 \leftarrow 0$, $t_1 \leftarrow 1$, $S_1 \leftarrow 1$, $e_{RARAM} \leftarrow 1$
26	32	BI	$d_{PP} \leftarrow 1$
27	34	BI	$S_2 \leftarrow 0$, $S_3 \leftarrow 0$, $t_1 \leftarrow 1$, $S_1 \leftarrow 1$, $e_{RARAM} \leftarrow 1$
28	0	BI	$i_{IX} \leftarrow 1$
29	0	BI	$d_{IX} \leftarrow 1$
30	0	BC	
31	0	BI	$S_2 \leftarrow 1$, $S_3 \leftarrow 1$, $t_1 \leftarrow 1$, $e_{CO} \leftarrow 1$
32		BS	$S_2 \leftarrow 0$, $S_3 \leftarrow 0$, $t_1 \leftarrow 1$, $S_1 \leftarrow 1$, $e_{RARAM} \leftarrow 1$
33	0	BI	$S_2 \leftarrow 0$, $S_3 \leftarrow 1$, $t_1 \leftarrow 1$, $e_{RARAM} \leftarrow 1$
34	4	BI	$e_{REM} \leftarrow 1$, $i_{PP} \leftarrow 1$