

# Architecture de l'Ordinateur

**Responsable et Cours :** Robert Strandh

**Planning :** 13 semaines

1h20 de cours + 2h40 de TD + 4h de travail individuel  
par semaine

**Support de cours :** Robert Strandh et Irène Durand  
Architecture de l'ordinateur pour informaticiens  
Transparents

**Web:** <http://dept-info.labri.fr/~strandh/Teaching/Architecture/2004-2005/>

**Objectif :** Comprendre le fonctionnement interne des ordinateurs

**Contenu :** Circuits combinatoires et circuits séquentiels

Circuits pour l'arithmétique binaire

Logique à trois états et notion de bus

Mémoires

Instructions de base et microprogrammes

Interruptions

Protection et multiprogrammation

Mémoire cache

Mémoire virtuelle

# Circuits et Signaux

Chaque circuit a un certain nombre de "ports"

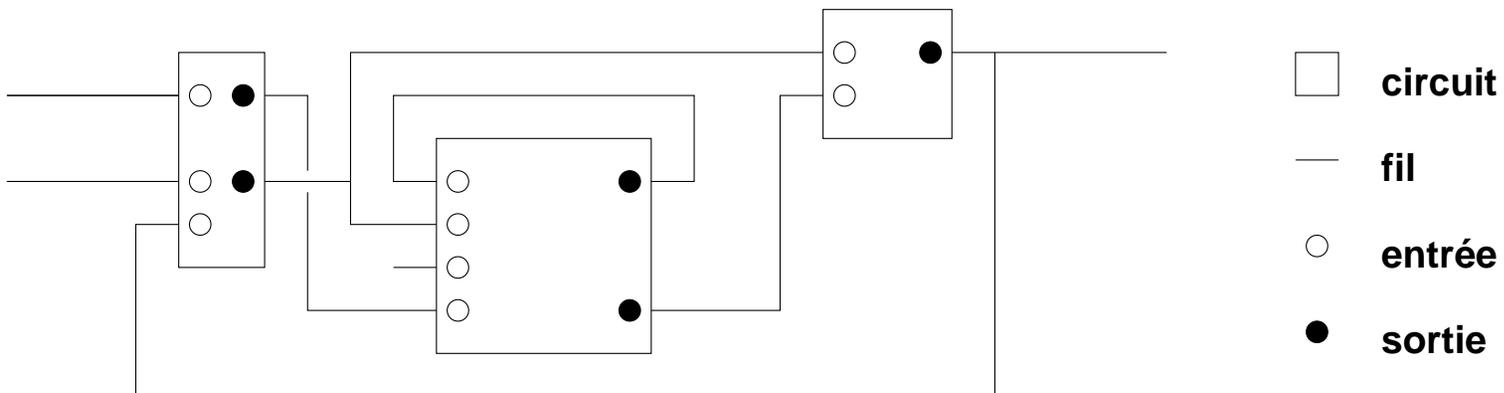
Un port est soit une entrée, soit une sortie  
(cette restriction sera supprimée plus tard)

Les ports sont interconnectés par des "fils"

Les fils transmettent des signaux entre les circuits

Un fil est connecté à au plus une sortie et un nombre arbitraire d'entrées

La valeur d'un signal est soit 0 (zéro, faux) ou 1 (un, vrai)



## **Circuits et Signaux (suite)**

**Les circuits sont souvent réalisés avec des transistors en silicium**

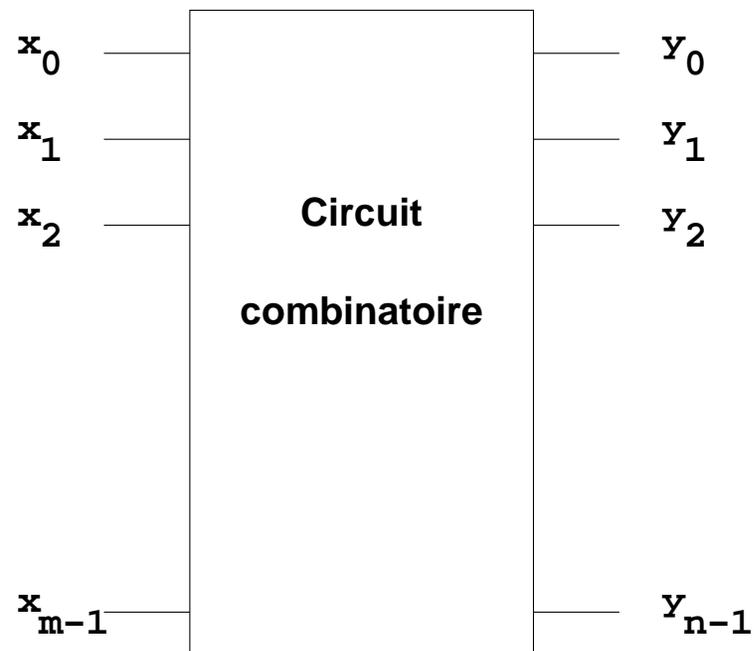
**Les fils sont réalisés en aluminium ou cuivre**

**La valeur 0 d'un signal est souvent représentée par une tension de 0 V (environ)**

**La valeur 1 d'un signal est souvent représentée par une tension de 3 ou 5 V (environ)**

## Circuits combinatoires

Un circuit combinatoire est un circuit à  $m$  entrées et  $n$  sorties tel que les valeurs des sorties dépendent uniquement de celles des entrées.



## Tables de vérité

Un circuit combinatoire peut être complètement décrit par une telle table

Elle montre la valeur de chaque sortie pour chaque combinaison des entrées

Exemple :

x	y	z	a	b
0	0	0	0	1
0	0	1	1	1
0	1	0	1	1
0	1	1	1	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	0	1

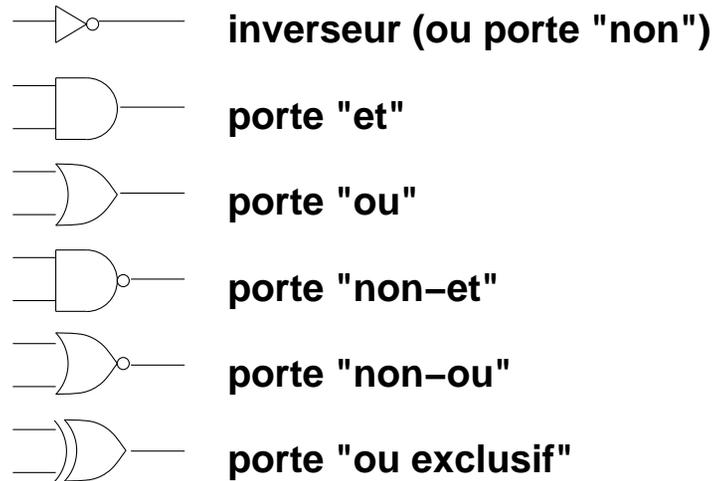
# Portes logiques

Ce sont des circuits combinatoires avec les caractéristiques suivantes :

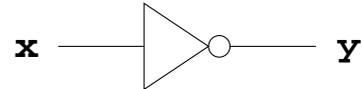
Une seule sortie

La fonction réalisée est régulière est simple

Nous allons traiter six type de portes logiques :



# Inverseur



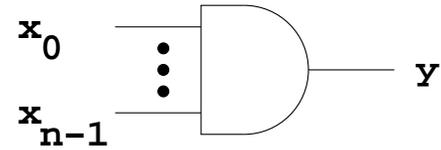
**Cette porte a exactement une entrée et exactement une sortie**

**La valeur de la sortie est toujours l'inverse de celle de l'entrée**

**Voici la table de vérité de l'inverseur :**

<b>x</b>	<b>y</b>
0	1
1	0

## Porte "et"



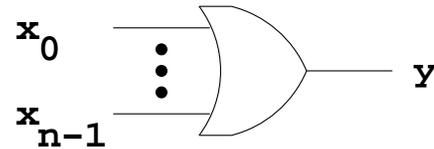
Cette porte a au moins deux entrées

La valeur de la sortie est 1 ssi la valeur de chaque entrée est 1

Voici la table de vérité (deux entrées) :

$x_1$	$x_0$	$y$
0	0	0
0	1	0
1	0	0
1	1	1

## Porte "ou"



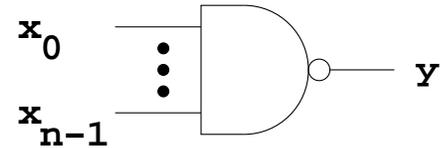
Cette porte a au moins deux entrées

La valeur de la sortie est 1 ssi la valeur d'au moins une entrée est 1

Voici la table de vérité (deux entrées) :

$x_1$	$x_0$	$y$
0	0	0
0	1	1
1	0	1
1	1	1

## Porte "non-et"



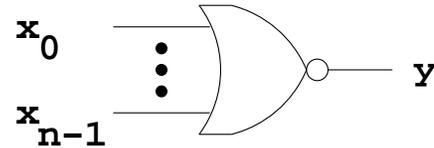
Cette porte a au moins deux entrées

La valeur de la sortie est 0 ssi la valeur de chaque entrée est 1

Voici la table de vérité (deux entrées) :

$x_1$	$x_0$	$Y$
0	0	1
0	1	1
1	0	1
1	1	0

## Porte "non-ou"



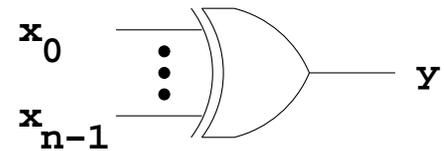
Cette porte a au moins deux entrées

La valeur de la sortie est 0 ssi la valeur d'au moins une entrée est 1

Voici la table de vérité (deux entrées) :

$x_1$	$x_0$	$Y$
0	0	1
0	1	0
1	0	0
1	1	0

## Porte "ou exclusif"



Cette porte a au moins deux entrées

La valeur de la sortie est 1 ssi la valeur d'exactly une entrée est 1

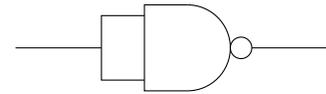
Voici la table de vérité (deux entrées) :

$x_1$	$x_0$	$y$
0	0	0
0	1	1
1	0	1
1	1	0

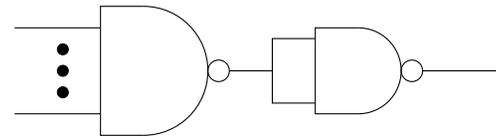
## Simulation de portes

Il suffit d'avoir des portes "non-et" pour simuler les autres portes

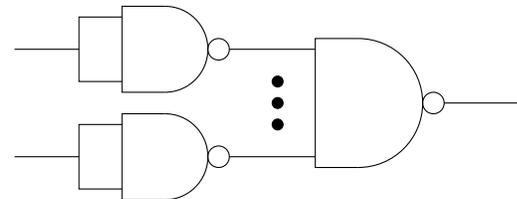
Pour simuler un inverseur, il suffit de brancher les deux entrées d'une porte "non-et" ensemble :



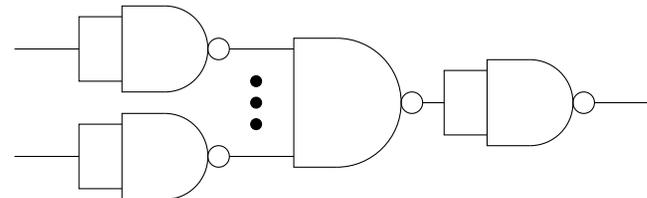
Pour simuler une porte "et" il suffit de brancher un inverseur à la sortie d'une porte "non-et" :



Pour simuler une porte "ou" il suffit de brancher un inverseur à chaque entrée d'une porte "non-et" :



Pour simuler une porte "non-ou" il suffit de brancher un inverseur à la sortie d'une porte "ou" :



La porte "ou exclusif" est un peu plus difficile, mais possible elle aussi.

# Construction de circuits combinatoires

**Plusieurs critères possibles :**

**Le moins de transistors possible**

**Le moins de consommation d'énergie possible**

**Le plus rapide possible**

**Spécification d'un circuit :**

**Table de vérité**

**Formule logique**

## Spécification: formule logique

<b>Constantes :</b> 0, 1	<b>loi</b>	<b>forme et</b>	<b>forme ou</b>
	<b>identité</b>	$1x = x$	$0+x = x$
<b>Variables :</b> x, y, z, ...	<b>nullité</b>	$0x = 0$	$1+x = 1$
<b>Opérateurs :</b> +, -, ·	<b>idempotence</b>	$xx = x$	$x+x = x$
	<b>inversion</b>	$x\bar{x} = 0$	$x+\bar{x} = 1$
	<b>commutativité</b>	$xy = yx$	$x+y = y+x$
	<b>associativité</b>	$(xy)z = x(yz)$	$(x+y)+z = x+(y+z)$
	<b>distributivité</b>	$x+yz = (x+y)(x+z)$	$x(y+z) = xy+xz$
	<b>absorption</b>	$x(x+y) = x$	$x+xy = x$
	<b>De Morgan</b>	$\overline{xy} = \bar{x}\bar{y}$	$\overline{x+y} = \bar{x}\bar{y}$

## Spécification : table de vérité

Pour toute combinaison des entrées, donner la valeur de la sortie

x	y	z	out
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Souvent, une spécification complète n'est pas nécessaire

x	y	z	out
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	-
1	0	0	-
1	0	1	1
1	1	0	0
1	1	1	1

## Abréviation de la table de vérité

Souvent la valeur de la sortie est identique pour plusieurs combinaisons d'entrées :

x1	x2	x3	x4	x5	y
0	0	-	-	-	1
0	1	-	-	-	0
1	0	-	-	0	0
1	0	-	-	1	1
1	1	0	-	-	1
1	1	1	-	-	0

Il est possible d'avoir des valeurs de sortie non précisées dans une table abrégée :

x1	x2	x3	x4	x5	y
0	0	-	-	-	1
0	1	-	-	-	0
1	0	-	-	0	-
1	0	-	-	1	1
1	1	0	-	-	-
1	1	1	-	-	0

## **Méthode générale de construction d'un circuit combinatoire**

**Si la table de vérité n'existe pas, la construire**

**Première couche :**

**Pour chaque ligne dont la valeur est 1, mettre une porte non-et, avec une entrée normale pour un 1, et une entrée inversée pour un 0**

**Deuxième couche :**

**Mettre une porte non-et avec en entrée les sorties de la première couche**

## Exemple de construction générale

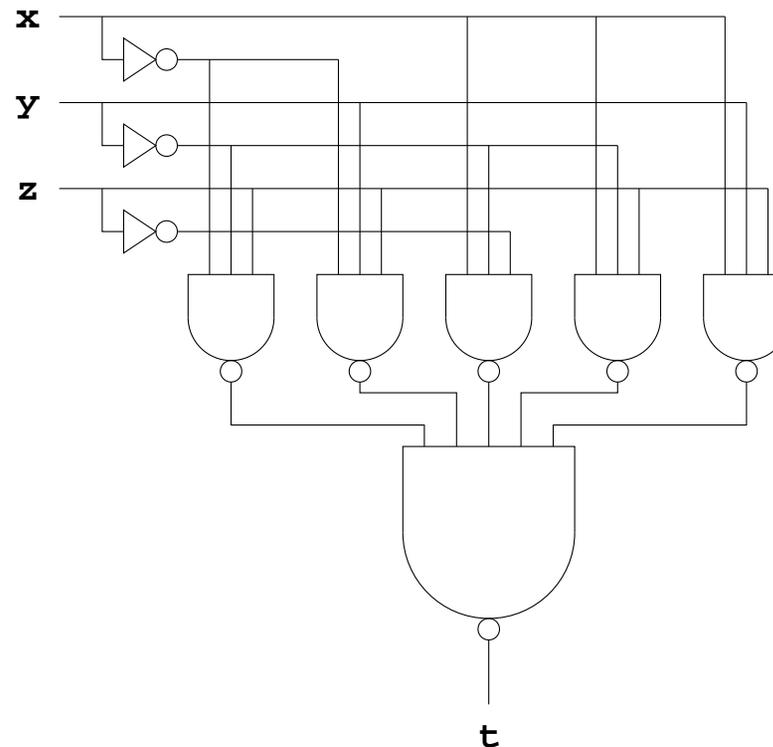
Réaliser un circuit combinatoire à partir de la formule :

$$t = x\bar{y} + z(\bar{x} + y)$$

Voici la table de vérité :

x	y	z	t
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Circuit :

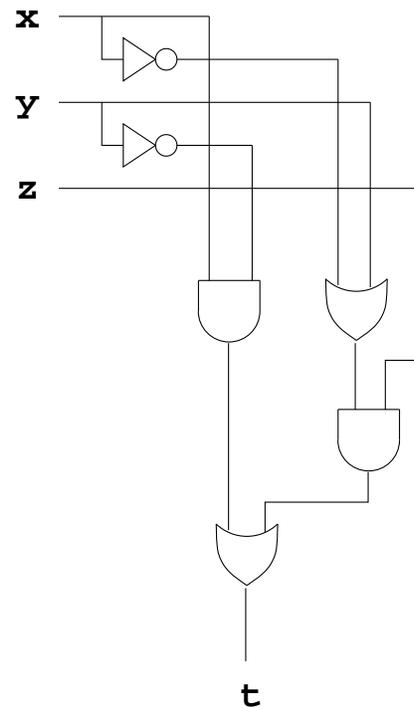


## Exemple de construction directe (à éviter)

Réaliser un circuit combinatoire à partir de la formule :

$$t = \overline{xy} + z(\overline{x} + y)$$

Circuit :



## Exemple où la méthode générale est inadaptée (Multiplexeur)

Circuit avec  $n$  entrées "adresses" et  $2^n$  entrées "données"

L'adresse permet de choisir l'une des entrées "données"

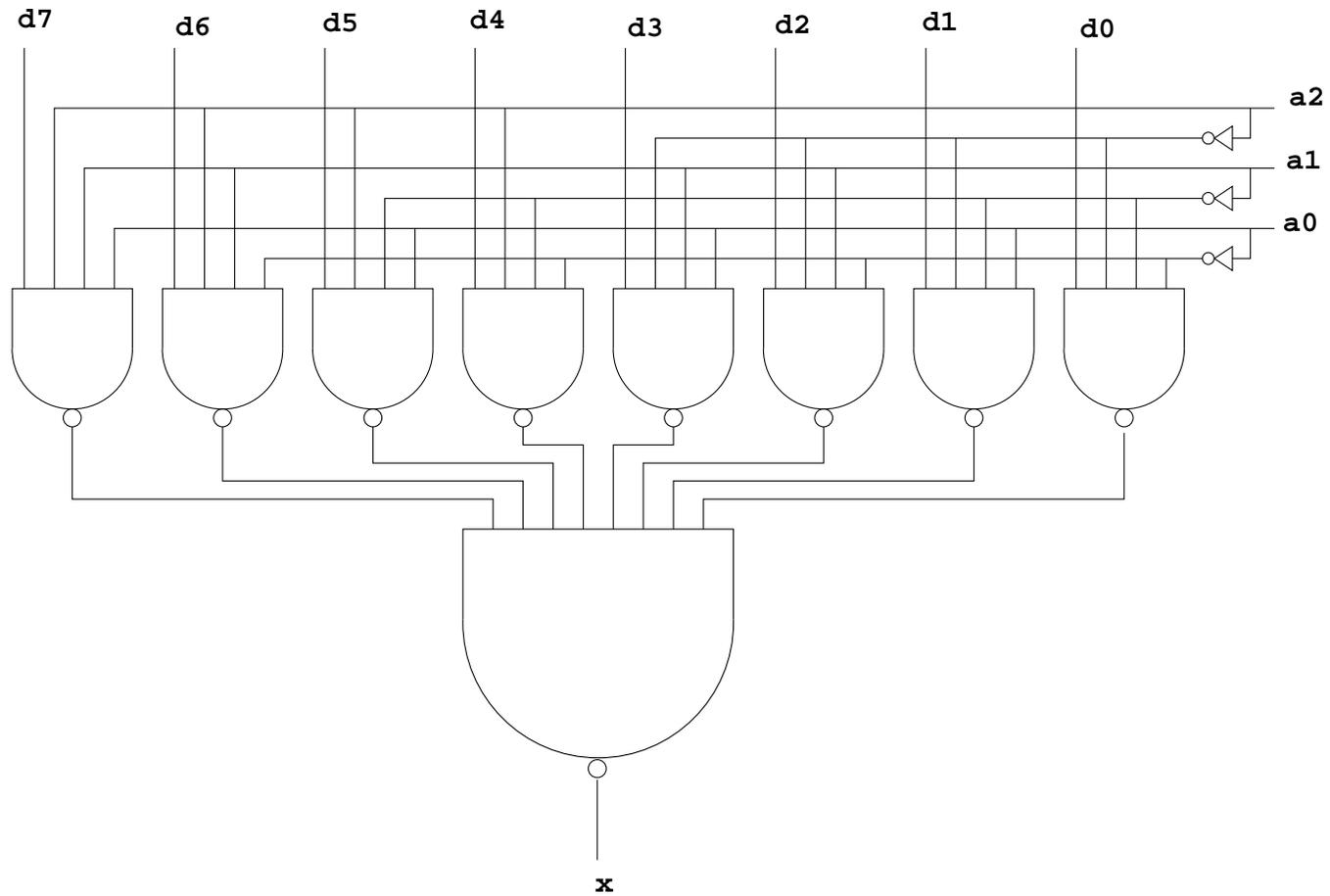
Table de vérité (abrégée) ( $n = 3$ )

a2	a1	a0	d7	d6	d5	d4	d3	d2	d1	d0	out
0	0	0	-	-	-	-	-	-	-	c	c
0	0	1	-	-	-	-	-	-	c	-	c
0	1	0	-	-	-	-	-	c	-	-	c
0	1	1	-	-	-	-	c	-	-	-	c
1	0	0	-	-	-	c	-	-	-	-	c
1	0	1	-	-	c	-	-	-	-	-	c
1	1	0	-	c	-	-	-	-	-	-	c
1	1	1	c	-	-	-	-	-	-	-	c

La table de vérité non abrégée a 2048 lignes dont 1024 avec out = 1

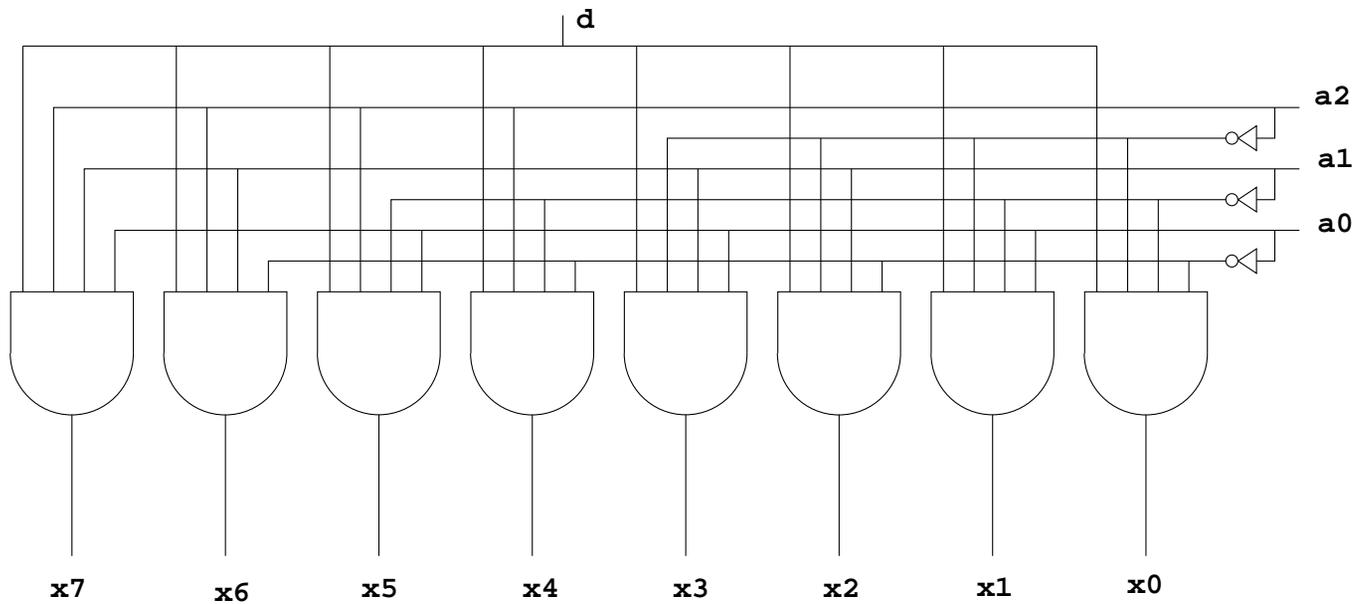
Donc le circuit construit avec la méthode générale a 1025 portes

# Le circuit du multiplexeur



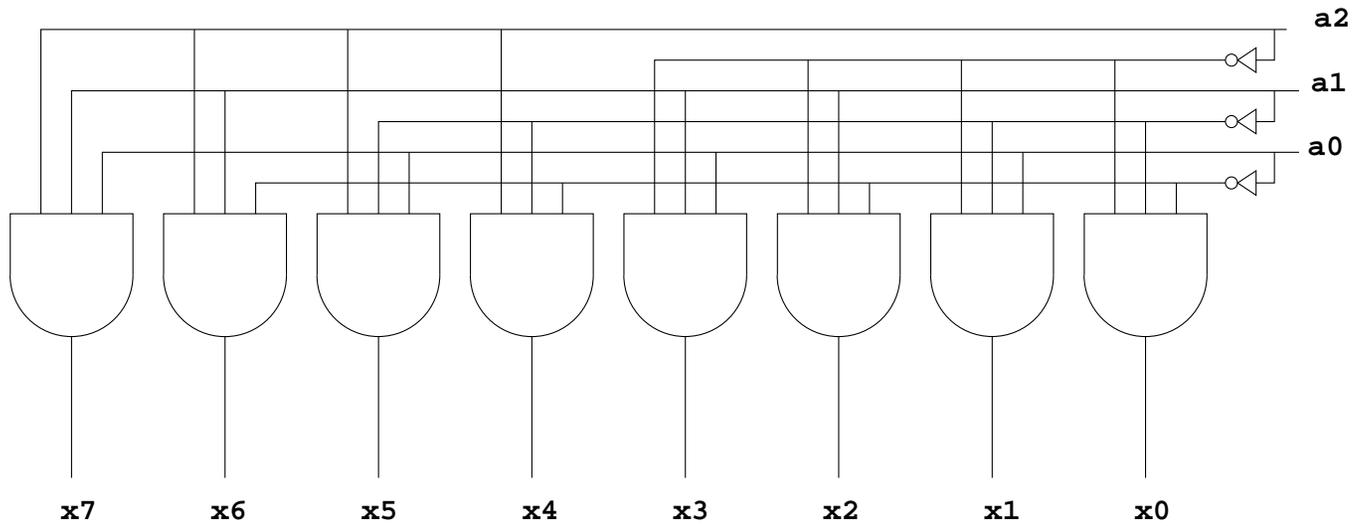
# Démultiplexeur

a2	a1	a0	d	x7	x6	x5	x4	x3	x2	x1	x0
0	0	0	c	0	0	0	0	0	0	0	c
0	0	1	c	0	0	0	0	0	0	c	0
0	1	0	c	0	0	0	0	0	c	0	0
0	1	1	c	0	0	0	0	c	0	0	0
1	0	0	c	0	0	0	c	0	0	0	0
1	0	1	c	0	0	c	0	0	0	0	0
1	1	0	c	0	c	0	0	0	0	0	0
1	1	1	c	c	0	0	0	0	0	0	0



# Décodeur

a2	a1	a0	x7	x6	x5	x4	x3	x2	x1	x0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0



# Arithmétique binaire

Représentation en base 10:

$$2034 = 2 * 10^3 + 0 * 10^2 + 3 * 10^1 + 4 * 10^0$$

Représentation en base 2:

$$11010 = 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$$



## **Représentation de nombres négatifs**

**Il est possible d'utiliser une représentation avec signe + valeur absolue**

**Pour l'arithmétique il faut alors un circuit d'addition et un autre de soustraction**

**Il y a une meilleure solution qui ne nécessite qu'un circuit d'addition**

**Cette solution s'appelle "complément à 2"**

**Pour la comprendre nous parlons de complément à 10 d'abord**

## **Représentation de nombres négatifs Complément à 10 avec précision infinie**

**Imaginons l'odomètre d'une voiture (ou d'un vélo) mais avec un nombre infini de roues.**

**Les nombres positifs sont représentés comme d'habitude**

**Les nombres négatifs avec un nombre infini de chiffres 9 à gauche**

$$-1 = \dots 9999999$$

$$-2 = \dots 9999998$$

$$-3 = \dots 9999997$$

**Supposons que nous avons un circuit d'addition de ce type de nombres**

**(Chaque nombre peut être représenté de façon finie)**

**C'est la représentation exacte utilisée par certains langage de programmation permettant la précision arbitraire des entiers (et des rationnels)**

## Représentation des nombres négatifs

Pour les nombres positifs de ce type, le circuit marche normalement

Pour additionner un nombre positif avec un nombre négatif, on fait comme si le nombre négatif était positif (et très grand)

...	0	0	0	3	4	(34)
...	9	9	9	9	3	(-7)
...	0	0	0	2	7	(27)

Pour additionner deux nombres négatifs, on fait la même chose

...	9	9	9	8	7	(-13)
...	9	9	9	9	3	(-7)
...	9	9	9	8	0	(-20)

Pour calculer la négation d'un nombre, il suffit de remplacer chaque chiffre  $c$  par  $(9 - c)$ , puis finalement additionner 1 au résultat.

## Représentation de nombres négatifs

Nous pouvons faire presque la même chose avec une précision finie

Mais il faut introduire la notion de débordement (overflow, underflow)

Si la roue la plus à gauche contient 0, 1, 2, 3 ou 4, alors un nombre positif

Si la roue la plus à gauche contient 5, 6, 7, 8 ou 9, alors un nombre négatif

Exemple d'addition:

2 3 3	2 3 3	2 3 3
1 0 5	5 2 1 (-479)	9 9 5 (-5)
3 3 8	7 5 4 (-246)	1 2 2 8 (???)

2 3 3	9 9 8 (-2)
3 2 1	8 8 1 (-119)
5 5 4 (-446!!!)	1 8 7 9 (???)

Si le resultat de l'addition de deux nombres positifs est un nombre positif, alors le resultat est bon

Si le resultat de l'addition de deux nombres négatifs est un nombre négatif, alors le resultat est bon

## **Représentation de nombres négatifs**

**Le résultat de l'addition d'un nombre positif et d'un nombre négatif est toujours bon (il faut éventuellement supprimer le chiffre supplémentaire)**

**Si le résultat de l'addition de deux nombres positifs est négatif, alors débordement**

**Si le résultat de l'addition de deux nombres négatifs contient un chiffre supplémentaire, alors débordement**

# Arithmétique binaire

**La même représentation marche en base 2 (complément à 2)**

**Les nombres positifs ont un 0 à la première position**

**Les nombres négatifs ont un 1 à la première position**

**Pour calculer la négation d'un nombre, il faut remplacer chaque chiffre par son inverse, puis finalement additionner 1**

**Le débordement se détecte de la même façon**

## **Représentation des nombres rationnels**

**Pas souvent en matériel, mais en logiciel, car nécessite la représentation des entiers avec une précision arbitraire**

**Représentation sous la forme de deux entiers, dont le premier éventuellement négatif**

**Les deux entiers n'ont pas de facteurs communs (représentation canonique)**

## **Représentation en virgule flottante (IEEE 754)**

**C'est une façon de représenter un sous ensemble des rationnels**

**Le nombre est divisé en mantisse est exposant les deux de taille fixe**

**La norme IEEE 754 définit 3 formats dont deux externes et un interne**

**Les formats internes sont : simple précision (32 bits) et double précision (64 bits)**

**Le format interne est utilisé à l'intérieur du processeur de calcul.**

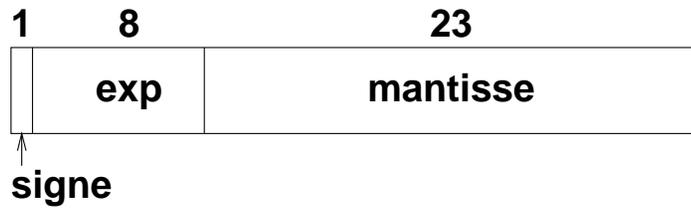
**La précision de ce format est de 80 bits (format étendu)**

**La multiplication étant plus fréquente que l'addition, on n'utilise pas la représentation de complément à 2, mais signe+valeur absolue pour la mantisse**

**Pour l'exposant on utilise une représentation similaire au complément à 2 appelée excédant 127 (simple précision) et excédant 1023 (double précision)**

# IEEE 754

Simple précision:



Double précision:



Le nombre représenté est  $s * m * 2^e$

## **Représentation de la mantisse (IEEE 754) (représentation normalisée)**

**La valeur représentée est toujours supérieure ou égale à zéro**

**De plus, la valeur est toujours normalisée ( $1 \leq v < 2$ ). C'est toujours possible, car on peut toujours ajuster l'exposant**

**La valeur représentée s'écrit donc : 1,...**

**Puisque le premier chiffre est toujours 1, on ne le représente pas (il est implicite dans la représentation)**

**En simple précision, la représentation 000000000000000000000000 indique donc la valeur 1, la représentation 100000000000000000000000 indique la valeur 1,5, etc.**



## **IEEE 754 (représentation non normalisée)**

**Il y a quatre représentations non normalisées:**

**Représentation dénormalisée**

**Représentation de zéro**

**Représentation de l'infini**

**NaN (Not a Number)**



## **IEEE 754 (représentation de zéro)**

**La valeur zéro est représentée par un champ de mantisse de 00000000  
et un champ exposant de 000000000000000000000000**

**C'est l'extension logique de la représentation dénormalisée**

**Il y a deux représentations pour zéro, dont une positive et une négative  
selon la valeur du champ signe**

## **IEEE 754 (représentation de l'infini)**

**Le champ exposant est 11111111 et le champ mantisse est 000000000000000000000000**

**Cette valeur est générée par des opérations arithmétiques dont le résultat dépasse ce qui est représentable de façon normalisée**

**Cette valeur est aussi acceptable en tant que opérande des opérations arithmétiques**

## **IEEE 754 (NaN, Not a Number)**

**Certaines opérations, par exemple la division de l'infini par l'infini, donne un résultat indéfini, dont la représentation contient le champ exposant 11111111 et n'importe quelle configuration de bits sauf 000000000000000000000000 du champs mantisse.**

## **Arithmétique en virgule flottante**

**Pour l'addition et la soustraction, il faut d'abord ajuster l'exposant du plus petit des deux opérandes pour que les deux exposants soit identiques**

**Il suffit alors de décaler la mantisse du plus petit nombre à droite (right shift) (y compris le bit implicite) et d'incrémenter son exposant**

**Si les exposants sont très différents, on risque de perdre des bits significatifs du plus petit nombre**

**Pour la soustraction, si les deux nombres diffèrent de peu, il peut y avoir une perte considérable de bits significatifs du résultat**

**Pour la multiplication, il suffit de multiplier les mantisses et d'additionner les exposants. Si l'on utilise un additionneur normal, il faut soustraire 127 du résultat pour obtenir la bonne représentation de l'exposant. Il faut aussi éventuellement normaliser le résultat, car la mantisse du résultat peut être supérieure ou égale à 2**

**Pour la division, il faut diviser les mantisses et soustraire les exposants**

## **Circuits pour l'arithmétique binaire (addition et soustraction)**

**L'utilisation de la méthode générale pour la construction d'un circuit combinatoire n'est pas possible**

**Même avec minimisation, un circuit à deux niveaux peut avoir trop de portes (selon le nombre de bits de la représentation)**

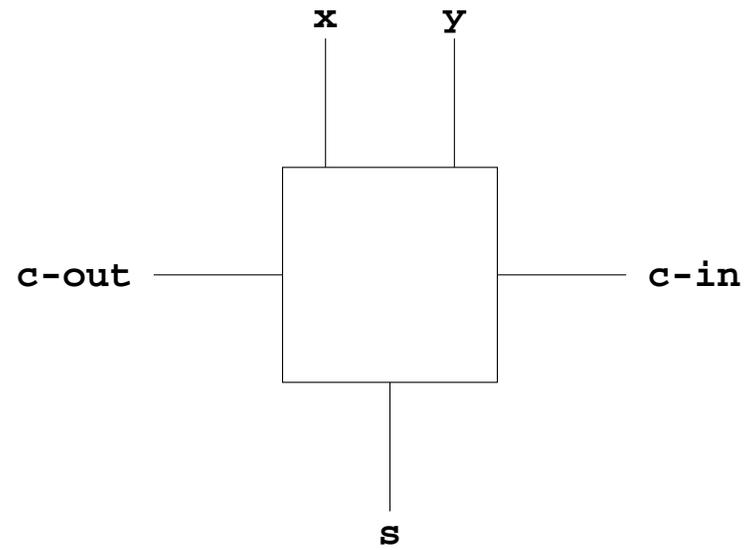
**On utilise un circuit combinatoire itératif**

**C'est la répétition d'un circuit combinatoire normal**

**Pour l'addition et soustraction, on peut commencer par un circuit combinatoire normal pour chaque bit significatif des opérandes**

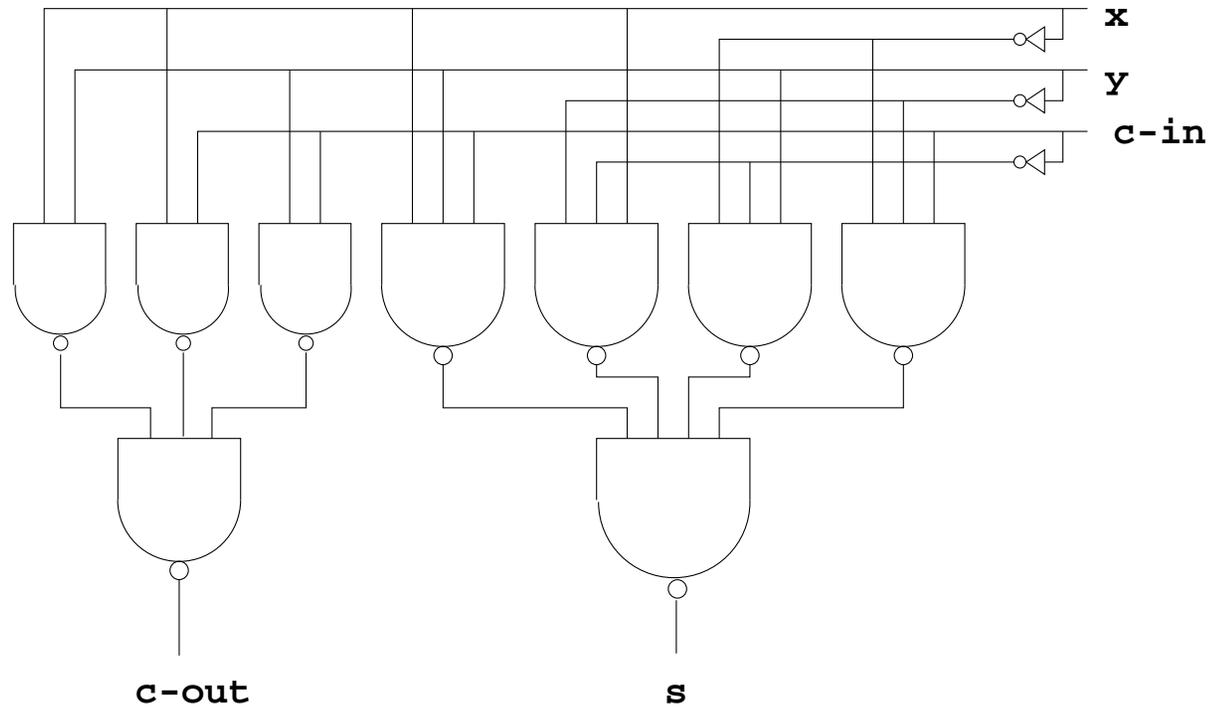
**Puis, on répète ce circuit autant de fois que le nombre de bits**

## Additionneur (1 bit)



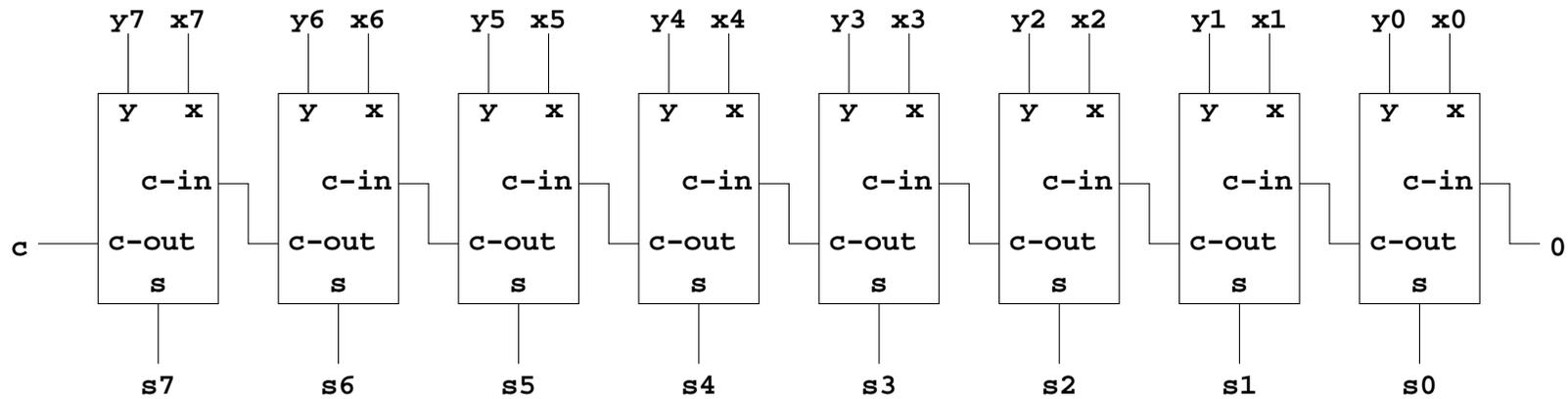
x	y	c-in	s	c-out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

# Additionneur, circuit



## Additionneur à n bits (exemple n = 8)

Il suffit maintenant de répéter le circuits n fois comme ceci :



## **Analyse du circuit d'addition**

**La profondeur (et donc le délai de calcul de la sortie) est proportionnelle au nombre de bits significatifs**

**Le nombre de portes est proportionnel au nombre de bits significatifs**

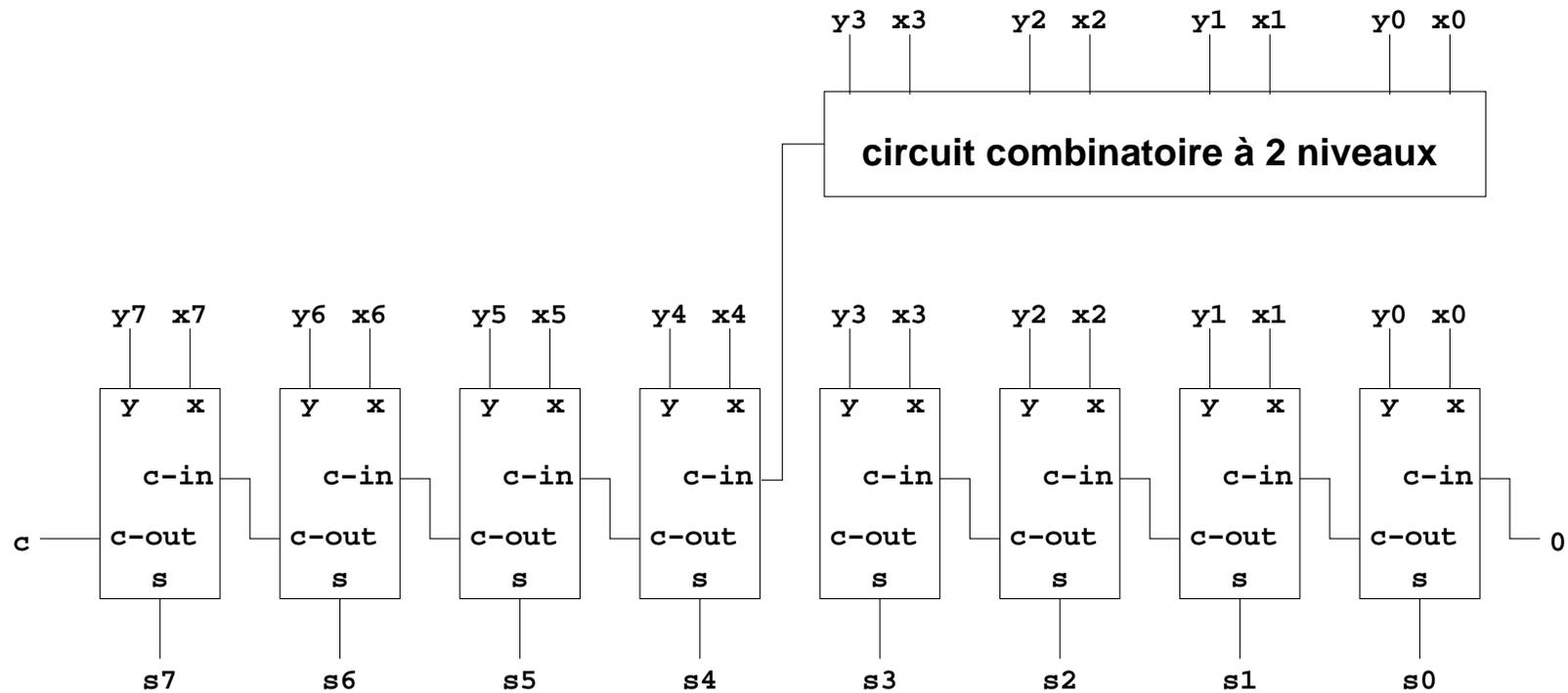
**Pour diminuer la profondeur, on peut imaginer plusieurs solutions intermédiaires :**

**Circuits pour l'accélération du calcul de la retenue**

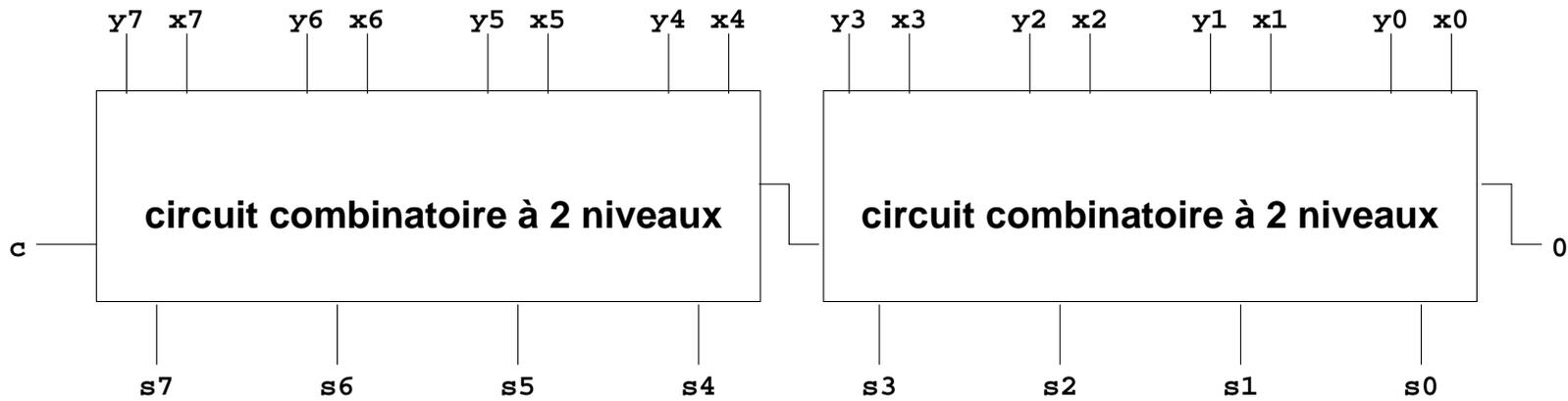
**Circuits d'addition de plusieurs bits à la fois (par exemple 4)**

# Accélération du calcul de la retenue

Idée de base :



## Addition de plusieurs bits à la fois



## Addition et soustraction

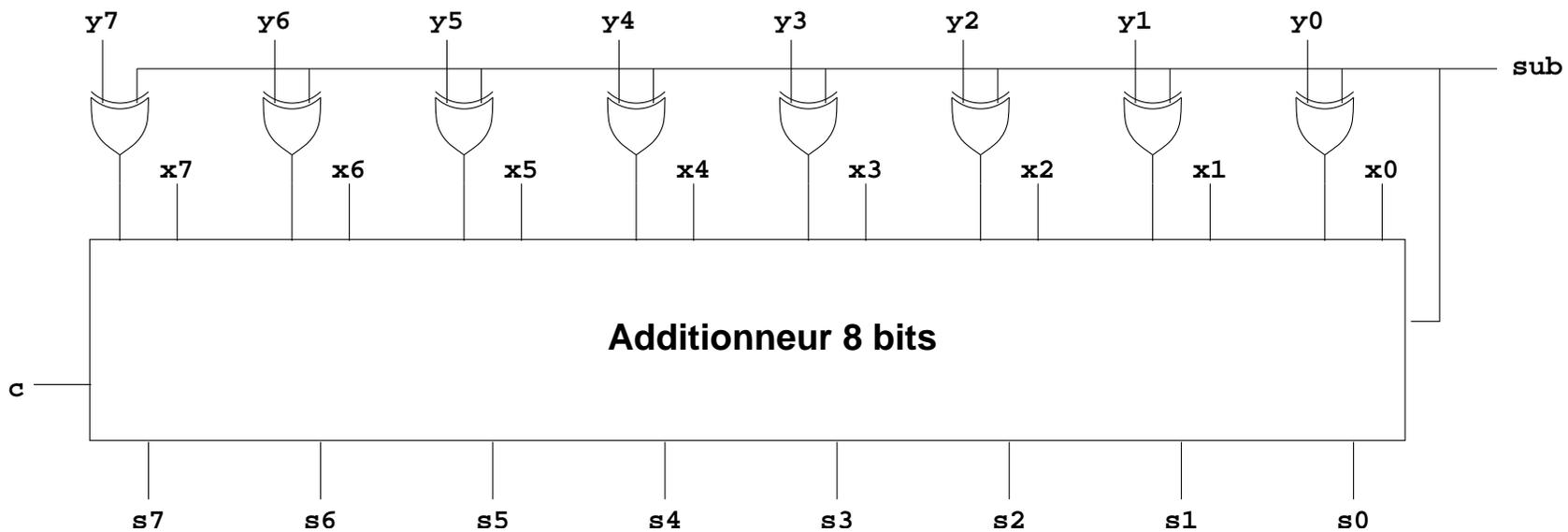
Pour calculer  $x - y$ , on calcule  $x + (-y)$ .

En représentation en complément à 2, on peut calculer  $-y$  comme  $\bar{y} + 1$

On peut donc calculer  $x + (\bar{y} + 1) = (x + \bar{y}) + 1$

Le calcul de l'inverse de  $y$  se fait avec des portes de type "ou-exclusif"

L'addition de 1 se fait avec l'entrée  $c$ -in



## **Circuits séquentiels**

**Un circuit séquentiel est un circuit de  $m$  entrées et  $n$  sorties tel que les valeurs des sorties dépendent des valeurs des entrées et des anciennes valeurs des sorties**

**La notion d'ancienne valeur est définie par rapport à un signal appelé horloge**

**Notre définition est une simplification, car en général, les valeurs des sorties peuvent dépendre des valeurs encore plus anciennes**

**Mais il suffit de rajouter des sorties artificielles pour réduire le problème général en notre définition plus simple**

## Table d'état

Un circuit séquentiel (avec notre définition) peut être complètement décrit par une telle table

Elle montre la valeur de chaque sortie pour chaque combinaison des entrées et des sortie (ancienne valeur)

Exemple :

u/d	x	y	x'	y'
0	0	0	0	1
0	0	1	1	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	0	1
1	1	1	1	0

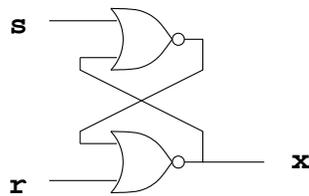
La nouvelle valeur (après un front d'horloge) d'une sortie  $s$  est indiquée par  $s'$

# Bascules

Une bascule est un circuit sans horloge capable de se souvenir d'un état précédent

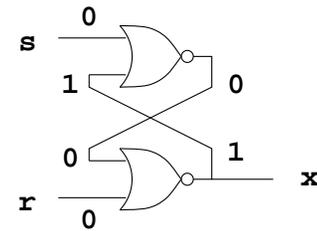
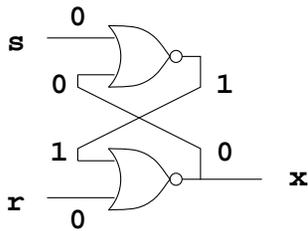
La bascule fondamentale s'appelle bascule SR (pour Set-Reset)

Voici sa réalisation :

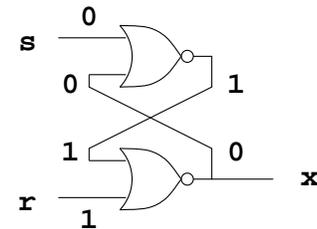
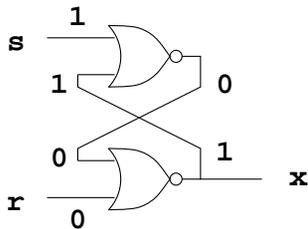


# Bascule SR

Une bascule SR a deux états stables possibles :



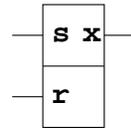
Si l'une des entrées est 1, alors voici le résultat :



Cet état est maintenu quand l'entrée passe de 1 à 0

## Symbole de la bascule SR

Les symboles pour les bascules ne sont pas aussi standardisés que ceux des portes logiques



## Bistables

Un bistable est un circuit séquentiel (donc avec horloge) similaire à une bascule, mais le changement d'état est synchronisé par une horloge

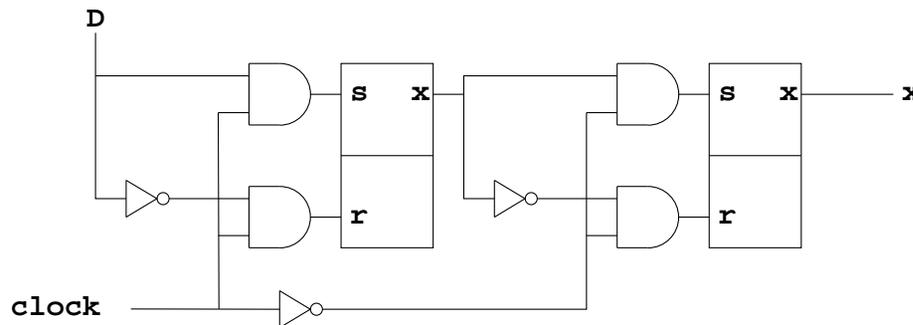
Il y a plusieurs types, mais ici nous traitons uniquement le bistable type D

Voici sa table d'état :

d	x	x'
0	0	0
0	1	0
1	0	1
1	1	1

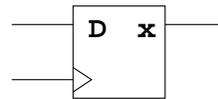
C'est donc un circuit séquentiel particulièrement simple, car l'ancienne valeur de la sortie x n'influence pas sa nouvelle valeur

Voici sa réalisation :



## Symbole du bistable D

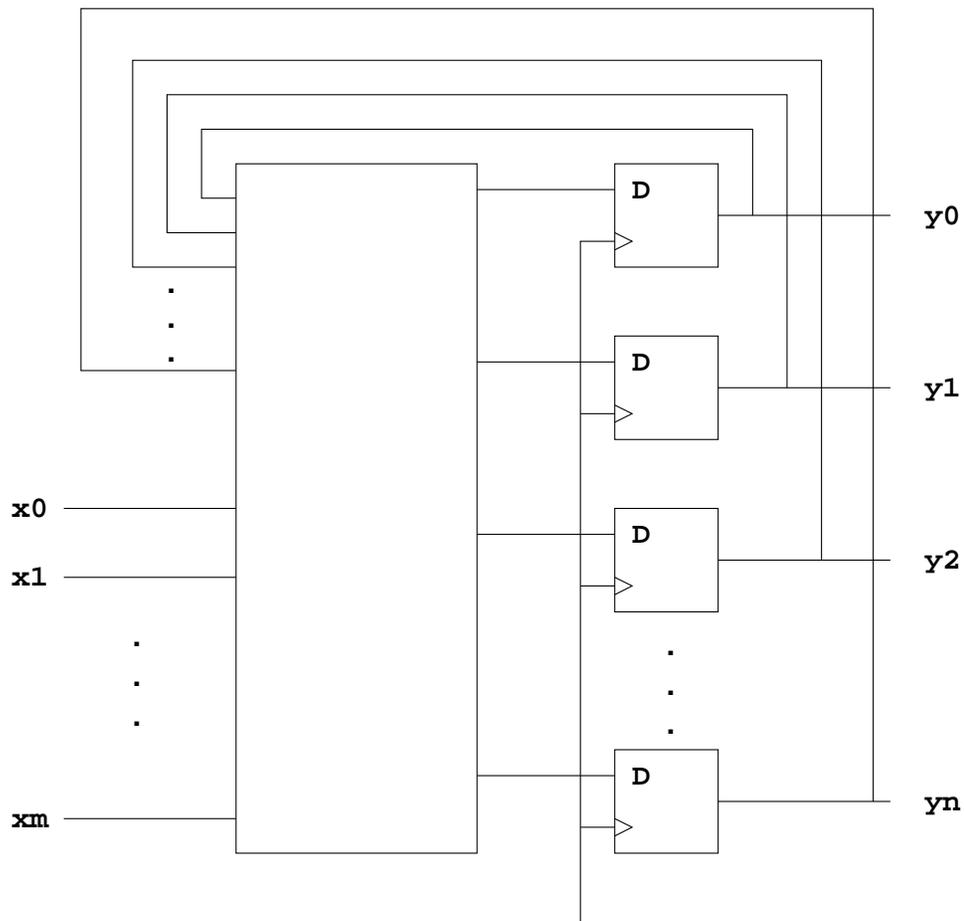
Voici le symbole du bistable D :



**Le petit triangle indique l'horloge et signifie que l'entrée est sensible aux transitions uniquement (front d'horloge)**

# Méthode générale de construction de circuits séquentiels

Notre méthode utilise  $n$  bistables D et un circuit combinatoire de  $m+n$  entrées et de  $n$  sorties :

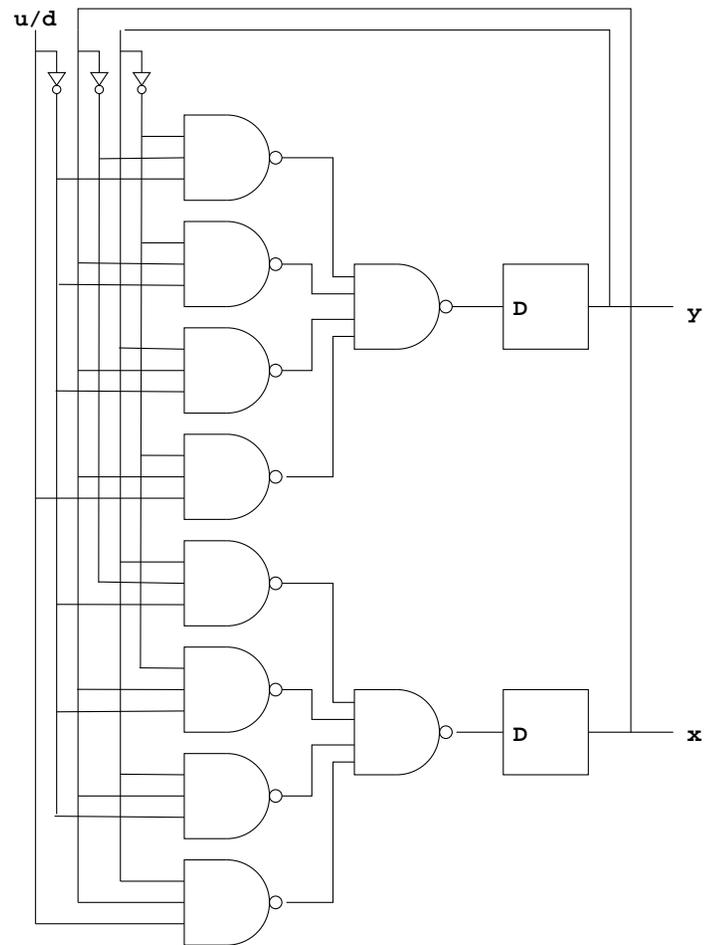


## Exemple de la méthode générale

Voici un exemple d'une table d'état :

u/d	x	y	x'	y'
0	0	0	0	1
0	0	1	1	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	0	1
1	1	1	1	0

Le circuit correspondant :



# Registres

Un registre est un circuit séquentiel avec  $n+1$  entrées (sans compter l'horloge) et  $n$  sorties

Voici la table d'état d'un registre à 4 bits :

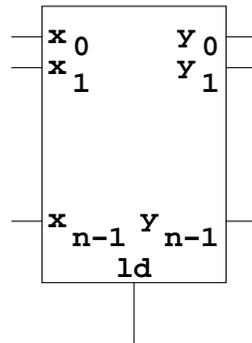
ld	x3	x2	x1	x0	y3	y2	y1	y0	y3'	y2'	y1'	y0'
0	-	-	-	-	c3	c2	c1	c0	c3	c2	c1	c0
1	c3	c2	c1	c0	-	-	-	-	c3	c2	c1	c0

Si ld (pour load) est 0, alors le registre préserve son contenu

Si ld est 1, alors le contenu est déterminé par les entrées

## Symbole du registre

Voici le symbole du registre :



# Compteurs

Un compieur est un circuit séquentiel de 0 entrées et de n sorties dont la valeur des sortie est l'ancienne valeur incrémentée de 1

Voici la table d'état d'un compteur de 3 bit (n = 3) :

y2	y1	y0	y2'	y1'	y0'
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

## Variations sur les compteurs

Plusieurs types de compteurs existent :

La possibilité d'incrémenter ou décrémenter selon la valeur d'une entrée

La possibilité de compter ou de maintenir la valeur selon la valeur d'une entrée

La possibilité de remettre la valeur à 0

La possibilité de charger une valeur particulière (compteur + registre)

L'utilisation d'un codage différent des nombres (gray, 7-segments, ...)

La possibilité de compter avec un incrément différent de 1

Toute combinaison des précédents

# Multiplication binaire

Trop difficile pour les circuits combinatoires

Il faut utiliser un circuit séquentiel

Pour chaque front d'horloge, une étape du calcul est effectuée

On utilise l'algorithme habituel, mais avec accumulateur :

$$\begin{array}{r} 1101 \\ 101 \\ \hline 1101 \\ 0000 \\ 1101 \\ \hline 1000001 \end{array}$$

$$\begin{array}{r} 1101 \\ 101 \\ \hline 1101 \leftarrow \text{accumulateur} \\ 0000 \\ \hline 1101 \leftarrow \text{accumulateur} \\ 1101 \\ \hline 1000001 \leftarrow \text{accumulateur} \end{array}$$

## Multiplication binaire (suite)

On note les facteurs  $x$  et  $y$ , et le resultat  $r$

Si  $x$  et  $y$  sont de  $n$  bits, alors  $r$  peut contenir  $2n$  bits

On écrit  $y$  de la façon suivante :

$$y = y_{n-1} 2^{n-1} + y_{n-2} 2^{n-2} + \dots + y_0 2^0$$

Après étape numéro  $i$ , l'accumulateur contient:

$$x * (y_{i-1} 2^{n-1} + y_{i-2} 2^{n-2} + \dots + y_0 2^{n-i})$$

Donc, quand  $i = n$ , alors, l'accumulateur contient le resultat,  $r$

## Multiplication binaire (suite)

Pour passer d'une étape à l'étape suivante, on fait la chose suivante :

$$r_{i+1} = r_i / 2 + x * y_i 2^{n-1}$$

Car :

$$r_i / 2 + x * y_i 2^{n-1} =$$

$$x * (y_{i-1} 2^{n-1} + y_{i-2} 2^{n-2} + \dots + y_0 2^{n-i}) / 2 + x * y_i 2^{n-1} =$$

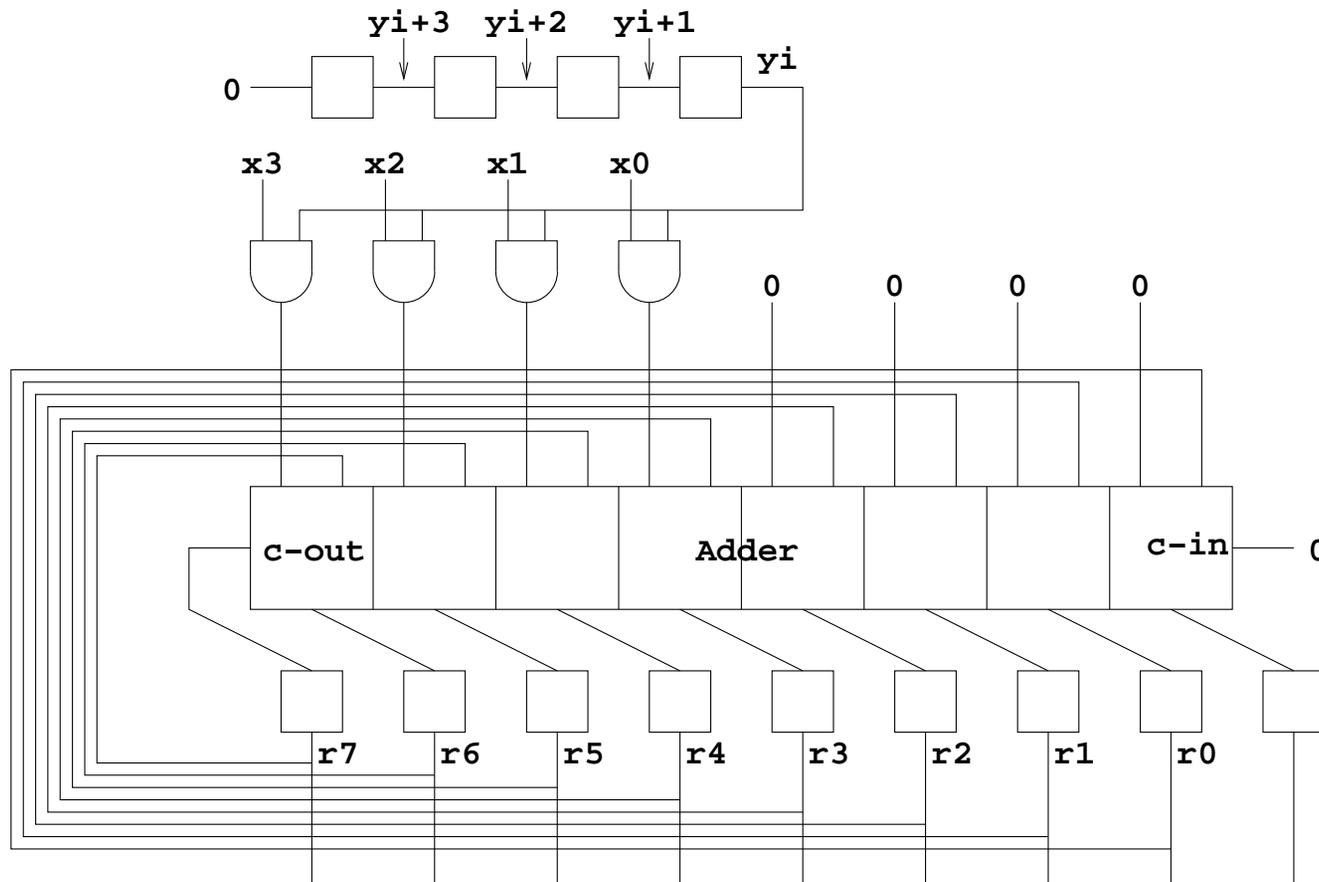
$$x * (y_{i-1} 2^{n-2} + y_{i-2} 2^{n-3} + \dots + y_0 2^{n-(i+1)}) + x * y_i 2^{n-1} =$$

$$x * (y_i 2^{n-1} + y_{i-1} 2^{n-2} + y_{i-2} 2^{n-3} + \dots + y_0 2^{n-(i+1)}) = r_{i+1}$$

## Multiplication binaire (suite)

Mais :  $r_{i+1} = r_i / 2 + x * y_i 2^{n-1} = (r_i + x * y_i 2^n) / 2$

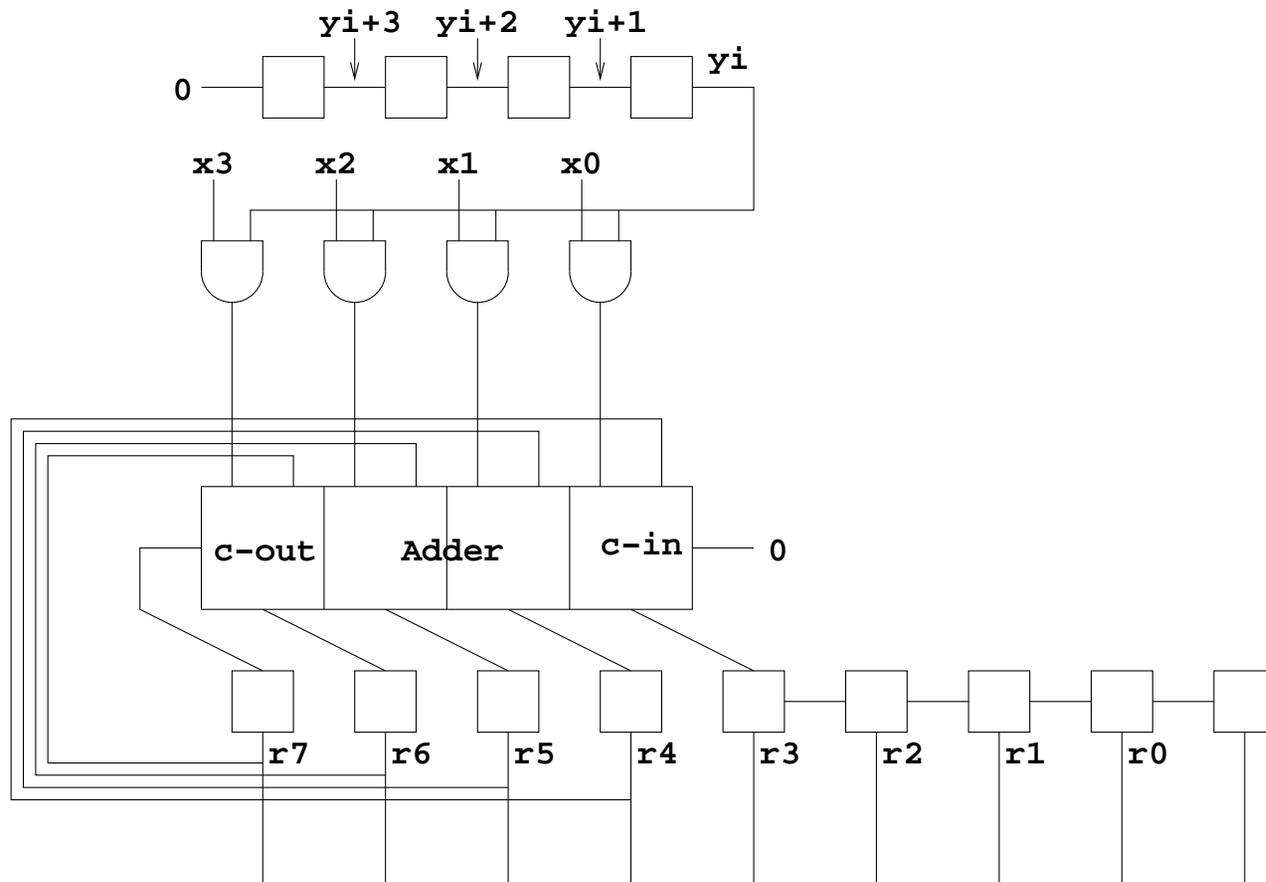
Voici un circuit séquentiel pour effectuer ce calcul :



## Multiplication binaire (suite)

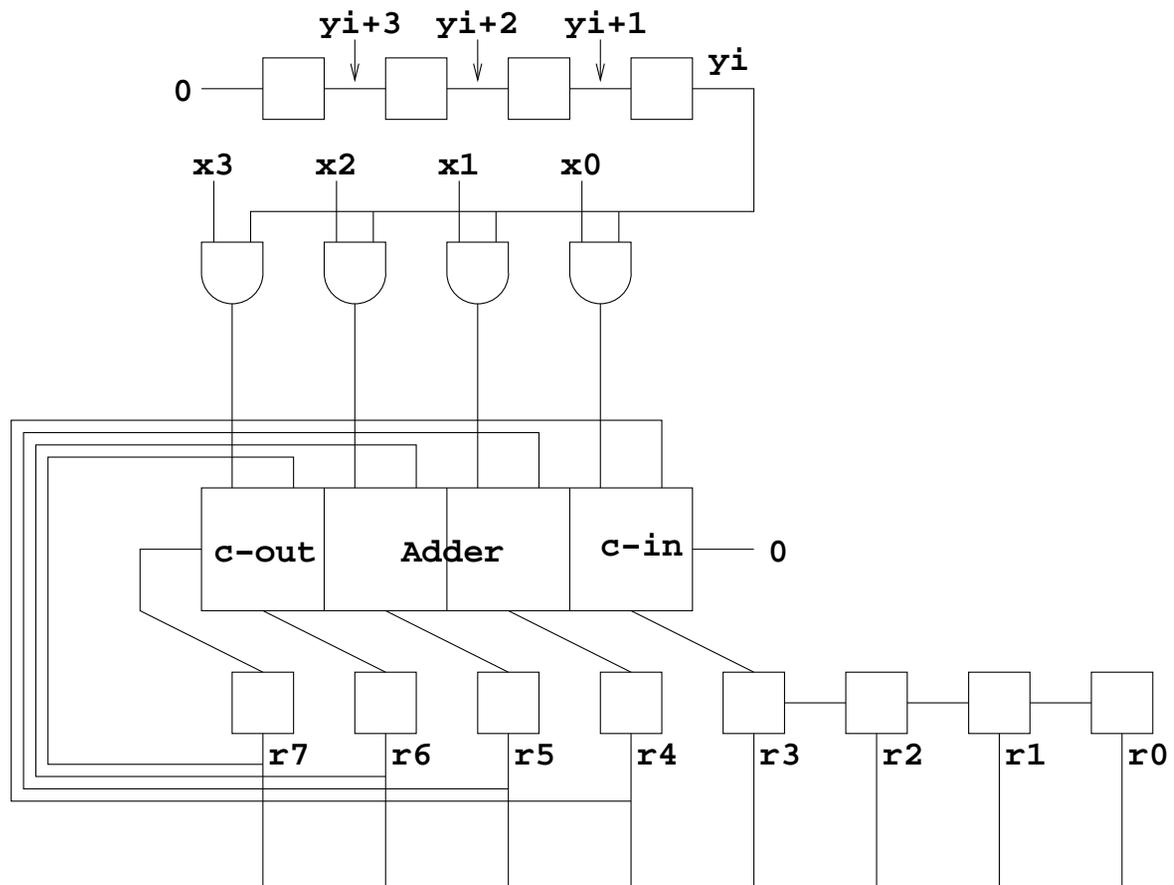
Le circuit précédent marche, mais contient des parties inutiles

Première simplification (éviter l'addition avec 0) :



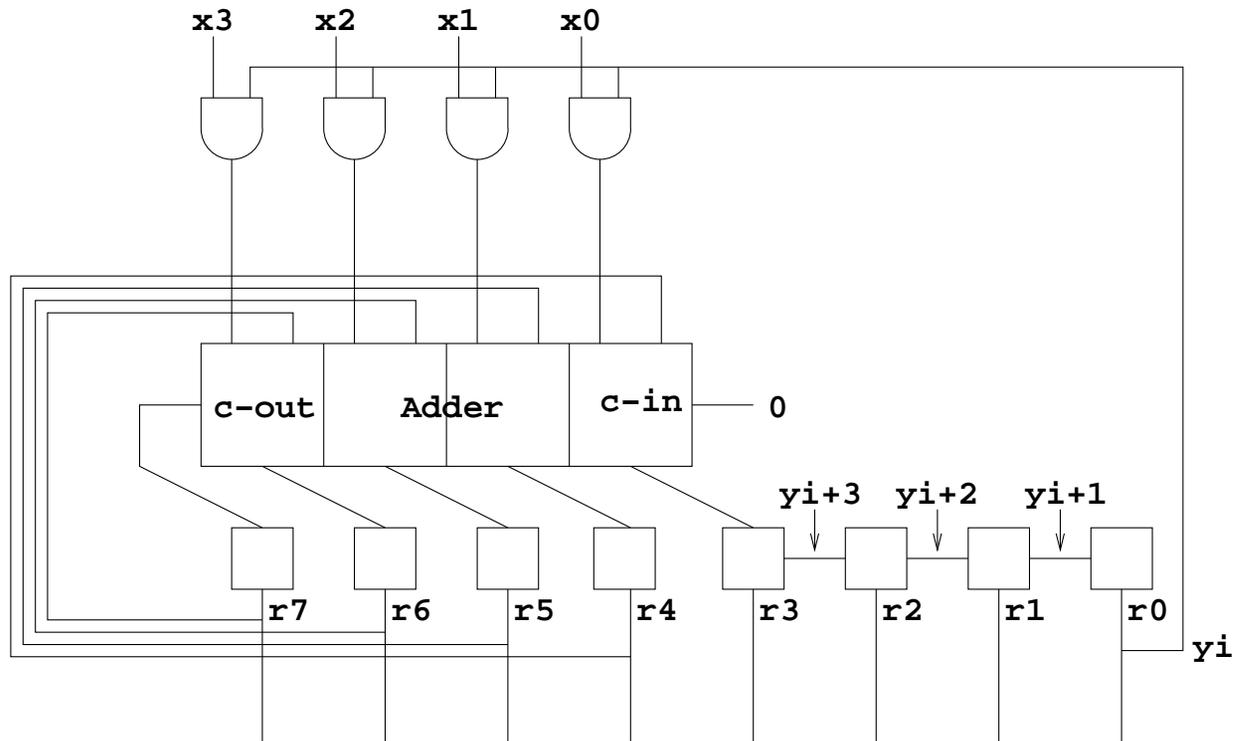
## Multiplication binaire (suite)

Deuxième simplification (le dernier bit contient toujours 0) :



## Multiplication binaire (suite)

Troisième simplification (stocker y dans r) :

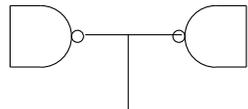


## Logique à trois états

Normalement, un circuit peut avoir l'une des deux valeurs 0 ou 1

Avec la logique à trois états, on introduit une troisième possibilité : non définie

Si l'on branche deux sorties de deux circuits différents ensemble, alors on risque de détruire au moins l'un des deux :



Avec des circuits à trois états, on peut le faire, à condition qu'au plus l'un des circuits branchés ait une valeur définie (0 ou 1)

Les circuits de ce type ont une entrée supplémentaire que l'on appelle "enable"

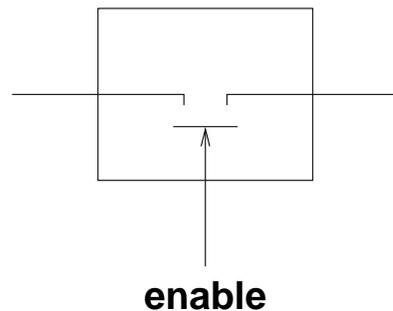
Si cette entrée est 1, alors le circuit se comporte comme un circuit normal

Si cette entrée est 0, alors la sortie du circuit est non définie

## Logique à trois états (suite)

Il est toujours possible de convertir un circuit normal en un circuit à trois états avec un circuit que l'on appelle "bus driver" ou "pilote de bus".

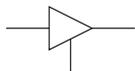
Voici comment on peut imaginer son fonctionnement :



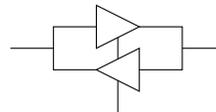
Quand "enable" est 1, alors il y a libre passage entre les deux autres ports

Quand "enable" est 0, alors il n'y a pas de connexion entre les deux autres ports

Symboles :



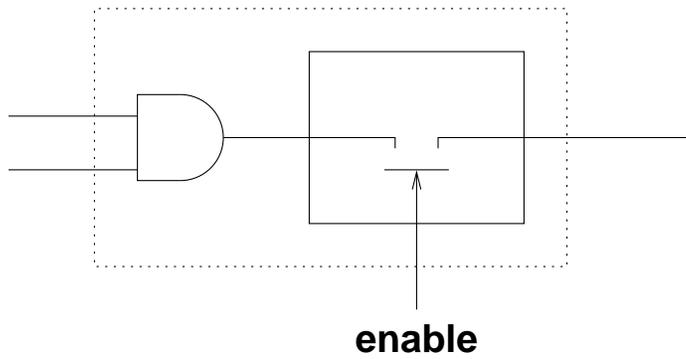
unidirectionnel



bidirectionnel

## Logique à trois états (suite)

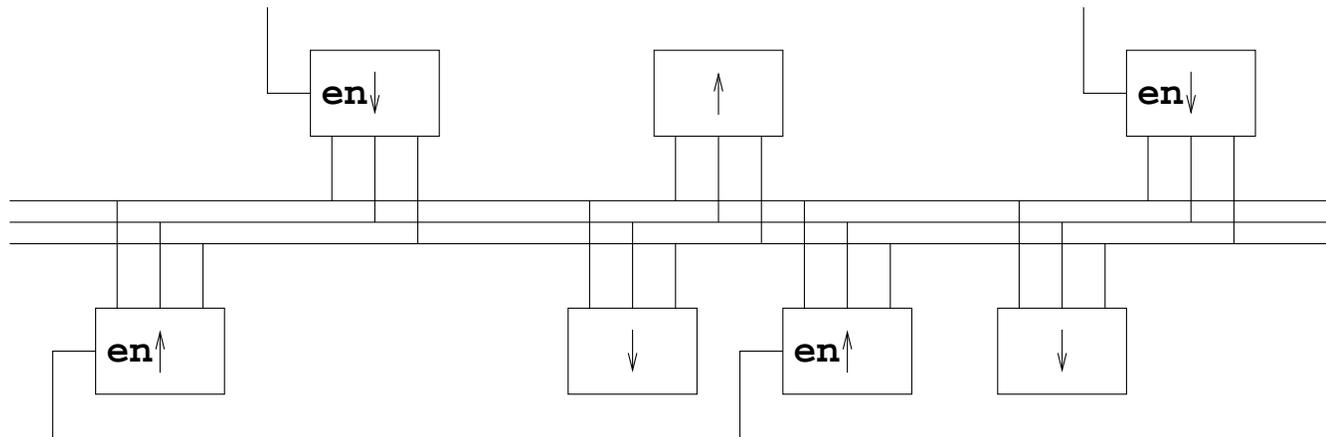
Pour convertir un circuit normal (par exemple une porte "et") en circuit à trois états, il suffit de faire passer la sortie par un "bus driver" :



## La notion de bus

**Un bus est une collection de fils sur lesquels on peut connecter les sorties de plusieurs circuits à trois états, ainsi que les entrées de circuits arbitraires**

**Un seul des circuits à trois états peut avoir ses sorties à 0 ou 1. Les autres doivent avoir la valeur non définie**



# Mémoires

**Une mémoire est un circuit similaire à une bascule**

**Ce n'est ni un circuit séquentiel (car sans horloge),  
ni un circuit combinatoire (car son état dépend du passé)**

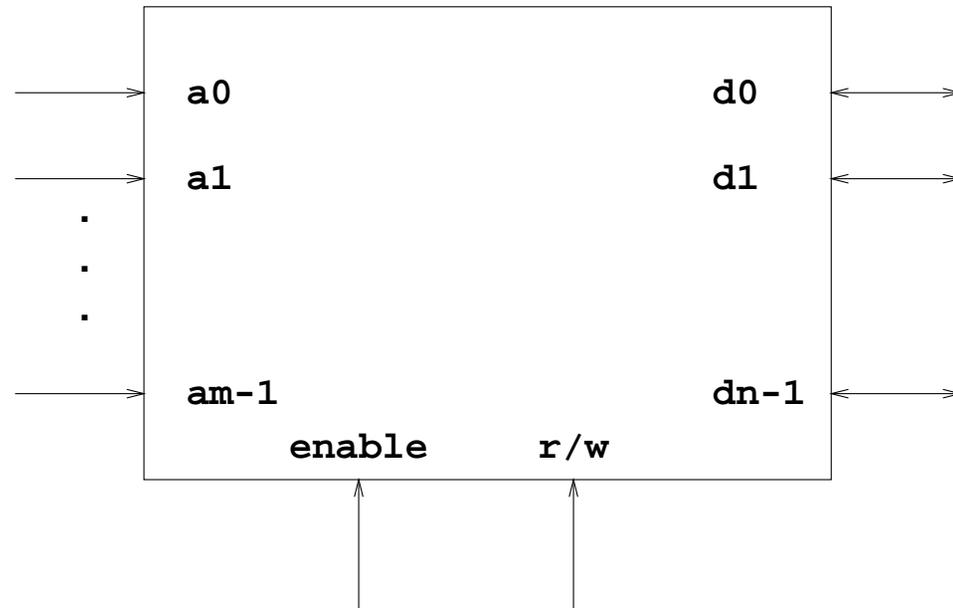
**De plus, pour diminuer le nombre de ports, les entrées et  
les sorties sont connectées sur les même fils**

**Une mémoire peut stocker un certain nombre de bits d'information**

**Les bits sont organisés en un certain nombre de mots de taille fixe  
(par exemple 8, 16, 32, 64, 128, ou 256 bits), mais pas forcément  
puissance de 2**

**Le nombre de mots est souvent une puissance de 2**

## Mémoires (suite)



Les entrées  $a_0 \dots a_{m-1}$  sont utilisées pour sélectionner un mot

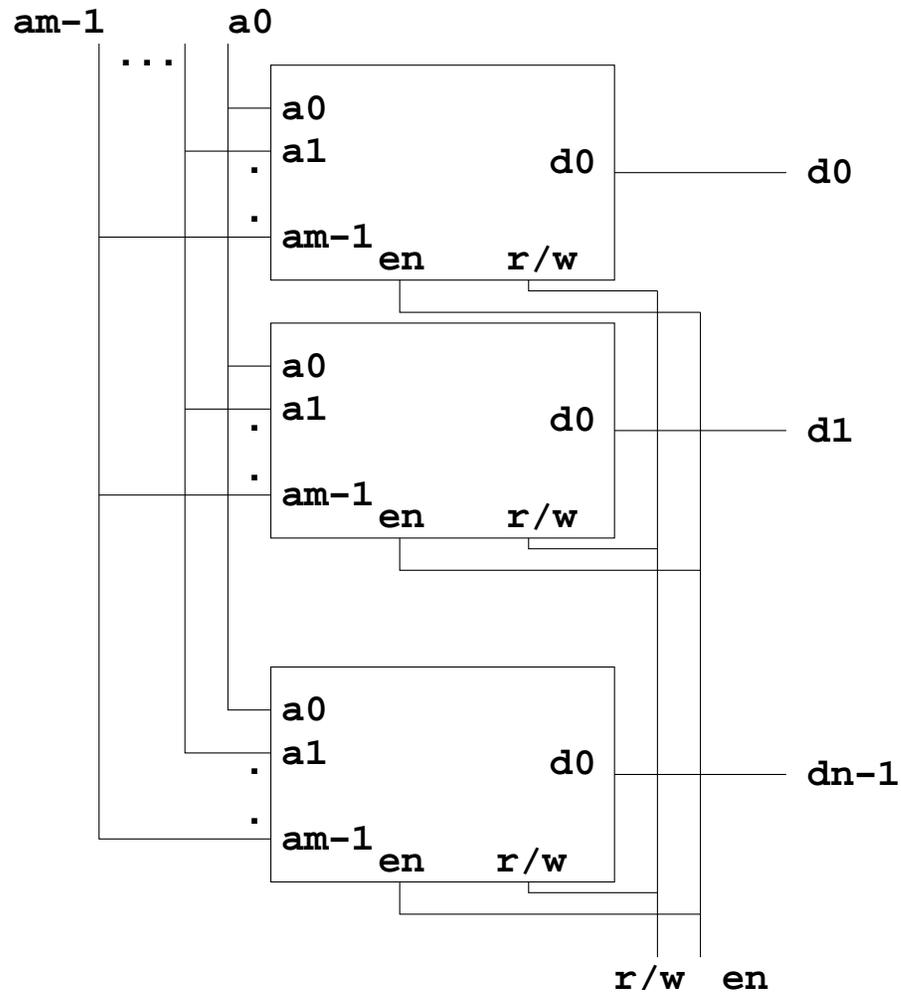
L'entrée  $enable$  indique si les sorties  $d_0 \dots d_{n-1}$  sont dans un état défini

L'entrée  $r/w$  (read/write) détermine la direction (entrée ou sortie) des ports  $d_0 \dots d_{n-1}$



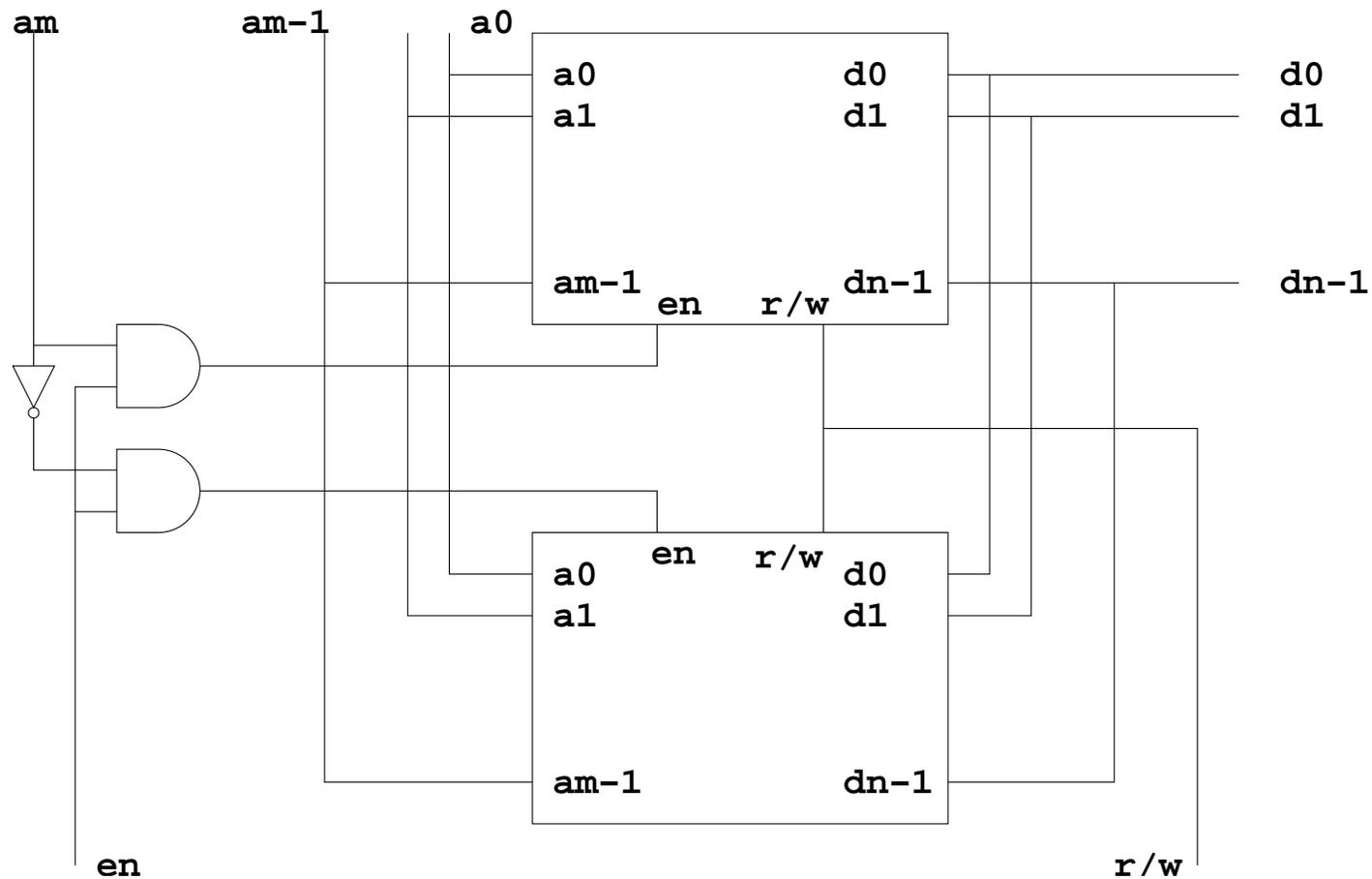
## Mémoires (suite)

Étant données  $n$  mémoires avec  $2^m$  mots de 1 bit, il est facile de construire une mémoire avec  $2^m$  mots de  $n$  bits :

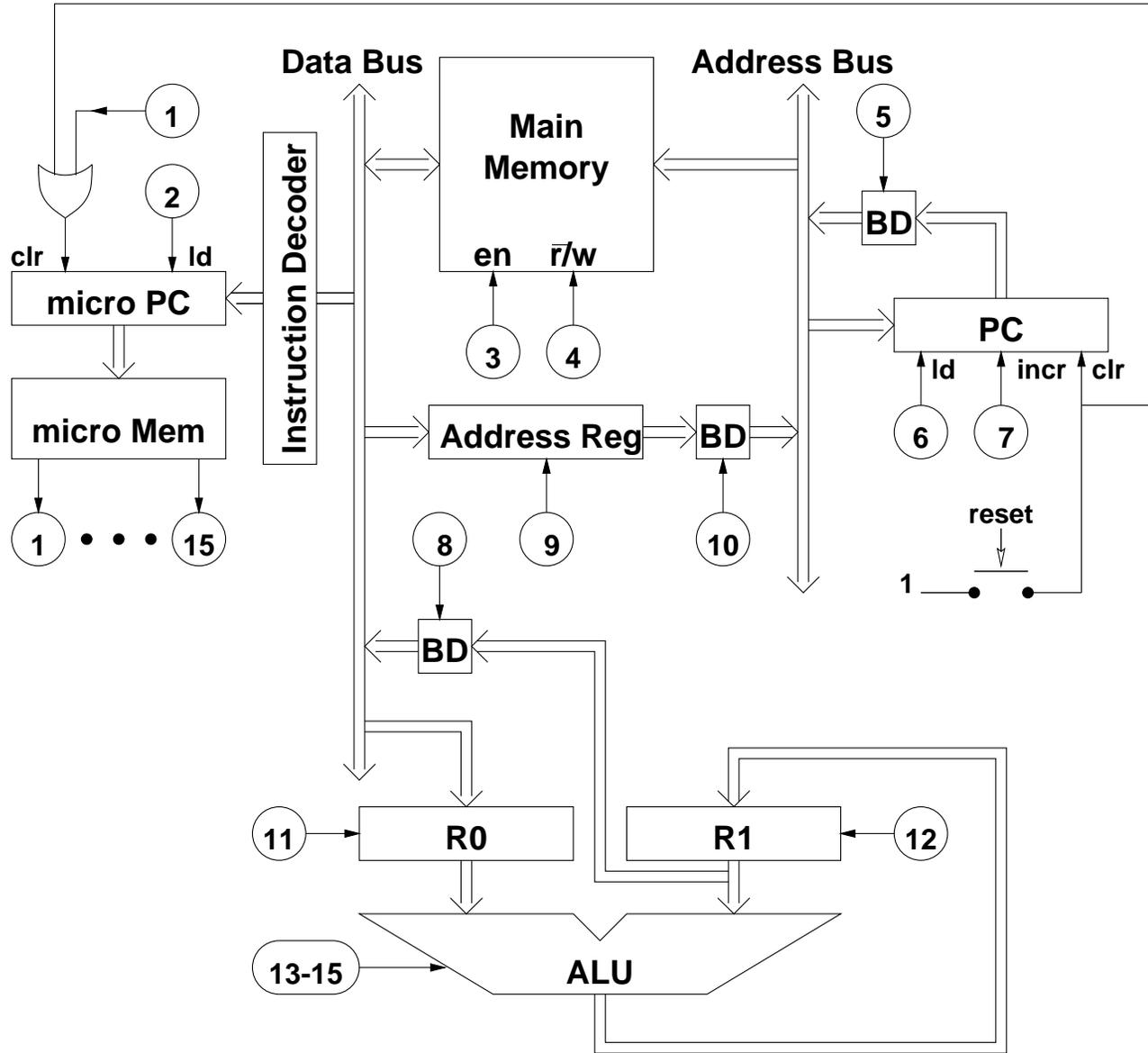


## Mémoires (suite)

De la même manière, étant données deux mémoires de  $2^m$  mots de  $n$  bits, il est facile de créer une mémoire de  $2^{(m+1)}$  mots de  $n$  bits:



# Le premier ordinateur



## **La micro mémoire**

**Un circuit combinatoire avec 6 entrées et 15 sorties**

**Les sorties sont les MOPs (micro-opérations)**

**C'est comme une mémoire mais sans écriture (read-only memory, ROM)**

**Le contenu de la micro mémoire détermine le comportement de l'ordinateur**

**Notre travail sera donc de remplir la micro mémoire**

**Ce contenu est appelé le micro programme**

## **Le micro PC**

**C'est le compteur ordinal (program counter) du micro programme**

**C'est un circuit séquentiel avec 6 entrées et 6 sorties**

**Le signal clr permet de remettre le contenu à 0**

**Le signal ld permet de charger un contenu à partir du décodeur d'instructions**

**Si clr et ld sont 0, alors, micro PC est un compteur normal**

**Le signal clr est prioritaire par rapport à ld**

## **Le décodeur d'instructions**

**C'est un circuit combinatoire avec 5 entrées et 6 sorties**

**Il traduit un codes d'instruction en adresse en micro mémoire du début du micro programme pour l'instruction**

**C'est comme une mémoire à lecture uniquement (ROM)**

## L'unité arithmétique et logique (ALU)

C'est un circuit combinatoire de 19 entrées et 8 (pour le moment) sorties

Selon les MOPs 13–15, capable d'effectuer une opération arithmétique ou logique

Les codes sont les suivant:

<b>000</b>	<b>sortie = première entrée</b>
<b>001</b>	<b>sortie = deuxième entrée décalée une position à gauche</b>
<b>010</b>	<b>sortie = deuxième entrée décalée une position à droite</b>
<b>011</b>	<b>sortie = somme des deux entrées</b>
<b>100</b>	<b>sortie = difference entre la deuxième et la première entrée</b>
<b>101</b>	<b>sortie = le ET (bit à bit) des deux entrées</b>
<b>110</b>	<b>sortie = le OU (bit à bit) des deux entrées</b>
<b>111</b>	<b>sortie = la négation (bit à bit) de la deuxième entrée</b>

Un implémentation possible : un circuit par opération plus un multiplexeur

## **Les registres R0 et R1**

**Circuits séquentiels chacun avec 9 entrées (8 + Id) et 8 sorties**

**L'entrée Id est piloté par MOP 11 (R0) et MOP 12 (R1)**

**L'entrée de R0 vient du bus de données**

**La sortie de R1 peut (via MOP 8 et un pilote de bus) sortir sur le bus de données**

**Le pilote de bus (bus driver, BD) est un circuit à trois états**

## **La mémoire principale**

**Une mémoire avec 8 lignes de données et 8 lignes d'adresse**

**Le signal enable est piloté par MOP 3 et read/write par MOP 4**

**Si r/w est 0, alors lecture, si r/w est 1 alors écriture**

**Si enable est 0, alors pas de connexion entre la mémoire et le bus de données**

## **Le registre d'adresses**

**C'est un registre ordinaire**

**Le signal Id est piloté par le MOP 9**

**Le contenu peut sortir sur le bus d'adresses via un pilote de bus piloté par le MOP 10**

**Ce registre permet la communication entre le bus de données et le bus d'adresses**

**Ce sera donc possible d'utiliser des données stockées en mémoire pour adresser la mémoire**

## **Le compteur ordinal**

**C'est un circuit séquentiel un peu compliqué avec 11 entrées et 8 sorties**

**Si clr est 1 alors le contenu sera toujours 0 après le front d'horloge**

**Sinon (clr = 0) si ld est 1 alors le contenu sera la valeur du bus d'adresses**

**Sinon (clr = 0, ld = 0) si incr est 1 alors le contenu sera incrémenté**

**Sinon (clr = 0, ld = 0, incr = 0) le contenu ne change pas**

**Le contenu peut sortir sur le bus d'adresses via un pilote de bus et MOP 5**

## Contenu de la micro mémoire et du décodeur d'instructions

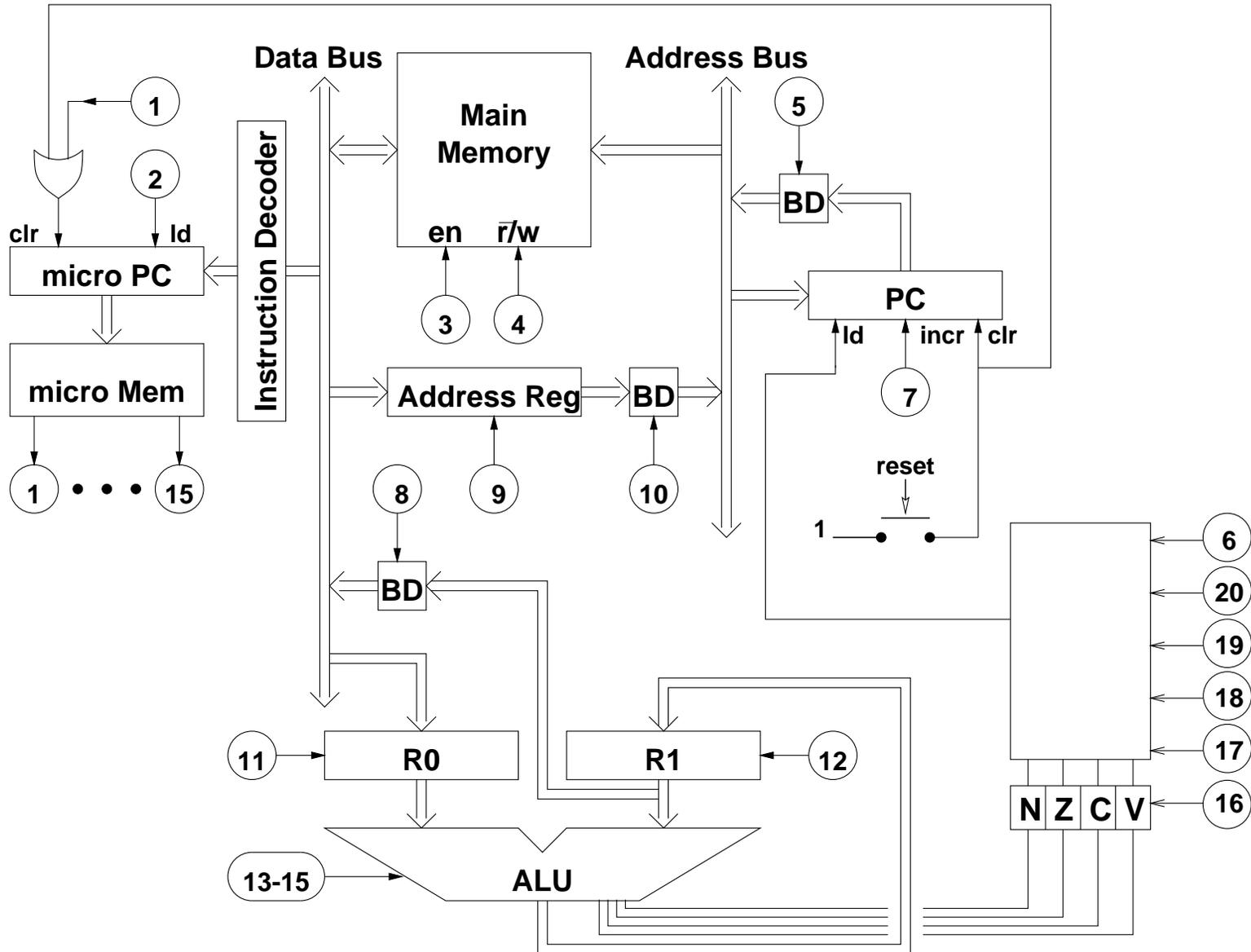
micro mémoire

000000:	0 1 1 0 1 0 1 0 0 0 0 0 0 0 0
000001:	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
000010:	1 0 1 0 1 0 1 0 0 0 1 0 0 0 0
000011:	0 0 1 0 1 0 1 0 1 0 0 0 0 0 0
000100:	1 0 1 0 0 0 0 0 0 1 1 0 0 0 0
000101:	0 0 1 0 1 0 1 0 1 0 0 0 0 0 0
000110:	0 0 0 0 0 0 0 1 0 1 0 0 0 0 0
000111:	0 0 0 1 0 0 0 1 0 1 0 0 0 0 0
001000:	0 0 1 1 0 0 0 1 0 1 0 0 0 0 0
001001:	0 0 0 1 0 0 0 1 0 1 0 0 0 0 0
001010:	1 0 0 0 0 0 0 1 0 1 0 0 0 0 0
001011:	1 0 0 0 0 0 0 0 0 0 0 1 0 0 0
001100:	1 0 0 0 0 0 0 0 0 0 0 1 0 0 1
001101:	1 0 0 0 0 0 0 0 0 0 0 1 0 1 0
001110:	1 0 0 0 0 0 0 0 0 0 0 1 0 1 1
001111:	1 0 0 0 0 0 0 0 0 0 0 1 1 0 0
010000:	1 0 0 0 0 0 0 0 0 0 0 1 1 0 1
010001:	1 0 0 0 0 0 0 0 0 0 0 1 1 1 0
010010:	1 0 0 0 0 0 0 0 0 0 0 1 1 1 1
010011:	0 0 1 0 1 0 1 0 1 0 0 0 0 0 0
010100:	1 0 0 0 0 1 0 0 0 1 0 0 0 0 0
010101:	
...	

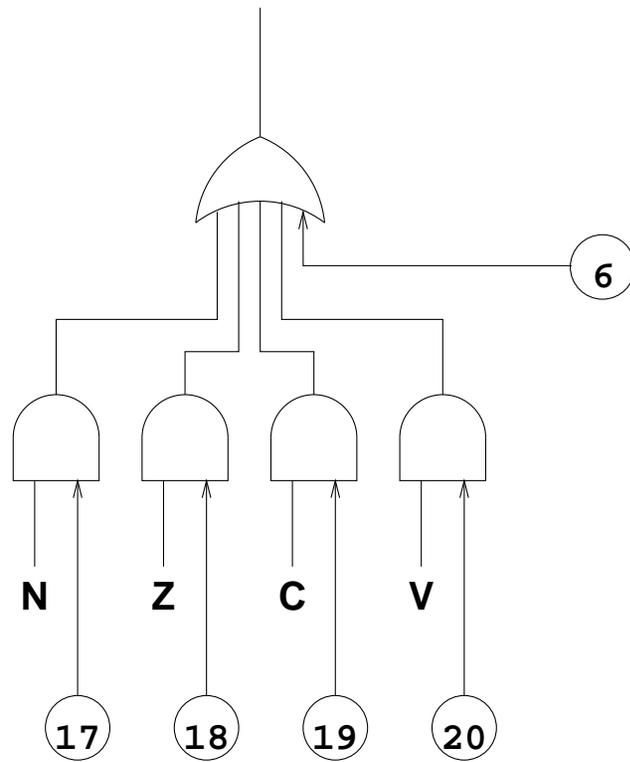
décodeur d'instructions

00000:	000001	NOP
00001:	000010	LDIMM
00010:	000011	LD
00011:	000101	ST
00100:	001011	COPY
00101:	001100	SHL
00110:	001101	SHR
00111:	001110	ADD
01000:	001111	SUB
01001:	010000	AND
01010:	010001	OR
01011:	010010	NOT
01100:	010011	JAL
01101:		
01110:		
01111:		
10000:		
10001:		
10010:		
10011:		
10100:		
10101:		
...		

# Sauts conditionnels



## Contenu du circuit



## Conteu de la micro mémoire et du décodeur d'instructions

### micro mémoire

000000:	0 1 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0
000001:	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
000010:	1 0 1 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0
000011:	0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0
000100:	1 0 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0
000101:	0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0
000110:	0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0
000111:	0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 0 0 0
001000:	0 0 1 1 0 0 0 1 0 1 0 0 0 0 0 0 0 0
001001:	0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 0 0 0
001010:	1 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0
001011:	1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0
001100:	1 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0 0
001101:	1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0
001110:	1 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 0 0 0 0
001111:	1 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0
010000:	1 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 0 0
010001:	1 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 0 0 0 0
010010:	1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0
010011:	0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0
010100:	1 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0
010101:	
...	

### décodeur d'instructions

00000:	000001	NOP
00001:	000010	LDIMM
00010:	000011	LD
00011:	000101	ST
00100:	001011	COPY
00101:	001100	SHL
00110:	001101	SHR
00111:	001110	ADD
01000:	001111	SUB
01001:	010000	AND
01010:	010001	OR
01011:	010010	NOT
01100:	010011	JAL
01101:		
01110:		
01111:		
10000:		
10001:		
10010:		
10011:		
10100:		
10101:		
...		

## Conteu de la micro mémoire et du décodeur d'instructions

### micro mémoire

001010:	1 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0
001011:	1 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0
001100:	1 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0
001101:	1 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0
001110:	1 0 0 0 0 0 0 0 0 0 1 0 1 1 1 0 0 0
001111:	1 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0
010000:	1 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 0
010001:	1 0 0 0 0 0 0 0 0 0 1 1 1 0 1 0 0 0
010010:	1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0
010011:	0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0
010100:	1 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0
010101:	0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0
010110:	1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0
010111:	0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0
011000:	1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0
011001:	0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0
011010:	1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1
011011:	0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0
011100:	1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1
...	

### décodeur d'instructions

00000:	000001	NOP
00001:	000010	LDIMM
00010:	000011	LD
00011:	000101	ST
00100:	001011	COPY
00101:	001100	SHL
00110:	001101	SHR
00111:	001110	ADD
01000:	001111	SUB
01001:	010000	AND
01010:	010001	OR
01011:	010010	NOT
01100:	010011	JAL
01101:	010101	JN
01110:	010111	JZ
01111:	011001	JV
10000:	011011	JC
10001:		
10010:		
10011:		
10100:		
...		

## Sous-programmes

En C:

```
f()
{
    h();
}
...
g()
{
    h();
}
```

```
h()
{
    ...
    return;
}
```

**Problème principal :**

**Le sous-programme doit connaître l'adresse de l'appelant**

## Sous-programmes (suite)

Solution utilisée par Fortran (il y a longtemps)

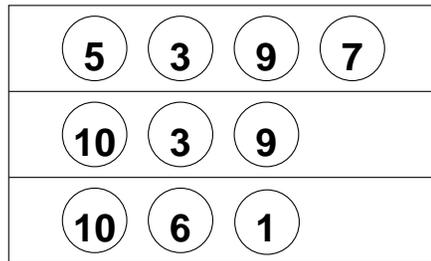
```
f:          ...          hret:      0
           ldimm fhret   h:          ...
           copy
           st hret
           ...
           jal h
fhret:     ...

g:          ...
           ldimm ghret
           copy
           st hret
           ...
           jal h
ghret:     ...
```

Cette solution dépend de l'existence d'une instruction jin (jump indirect).

## Sous-programmes (suite)

L'instruction jin est réalisable avec l'architecture actuelle :



**Problème :**

**On utilise les registres pour stocker l'adresse hret**

**Solution :**

**Introduire une instruction jsr (jump to subroutine)**

## Sous-programmes (suite)

Utilisation de jsr :

```
f:      ...
        jsr h-1
        ...

g:      ...
        jsr h-1
        ...

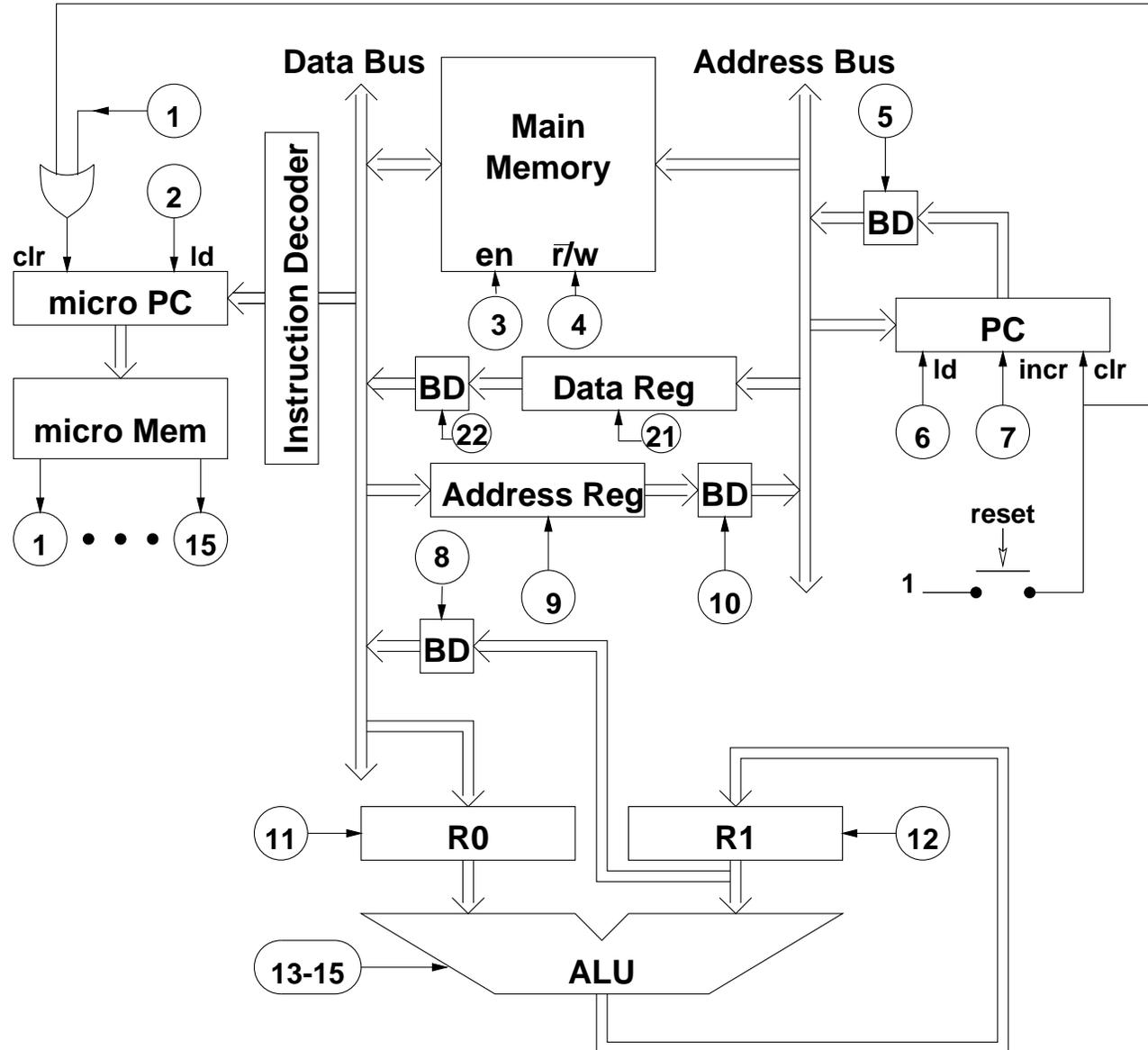
h:      0
        ...
        jin h-1
```

L'instruction jsr n'est pas réalisable avec l'architecture actuelle

Description de jsr :

1. Stocker la valeur de PC dans l'adresse donnée (ici h-1)
2. Charger l'adresse donnée + 1 (donc ici h) dans PC

# Modifications pour jsr



## Micro programme pour jsr

5	3	9	7
5	21		
10	22		
10	22	4	
10	22	4	3
10	22	4	
10	6		
7	1		

# Récurtivité

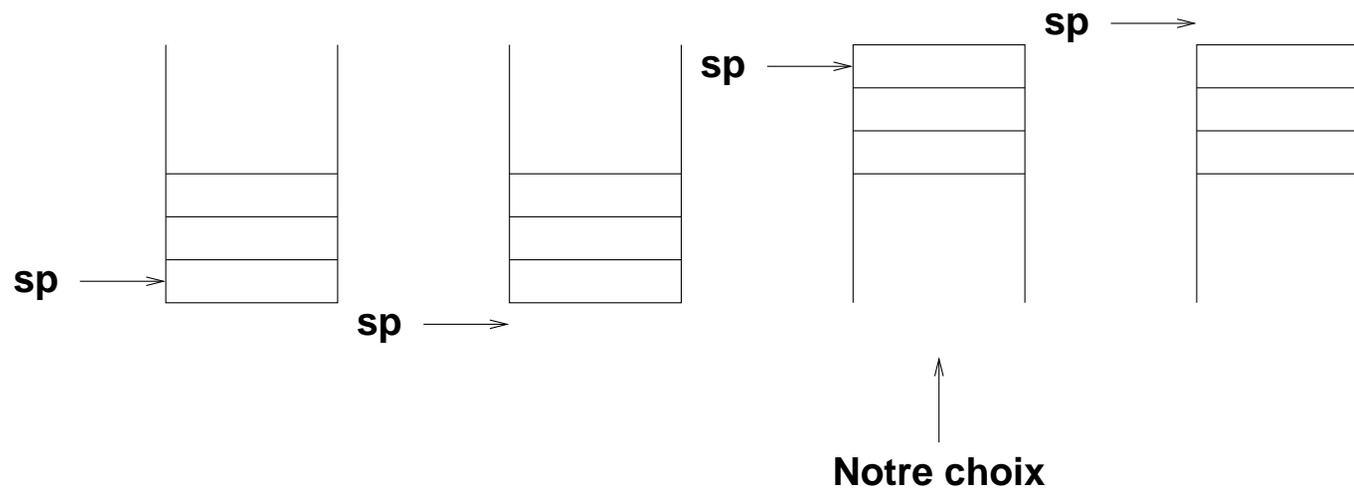
Le jsr actuel ne permet pas la récurtivité

Pour le faire, il faut une pile

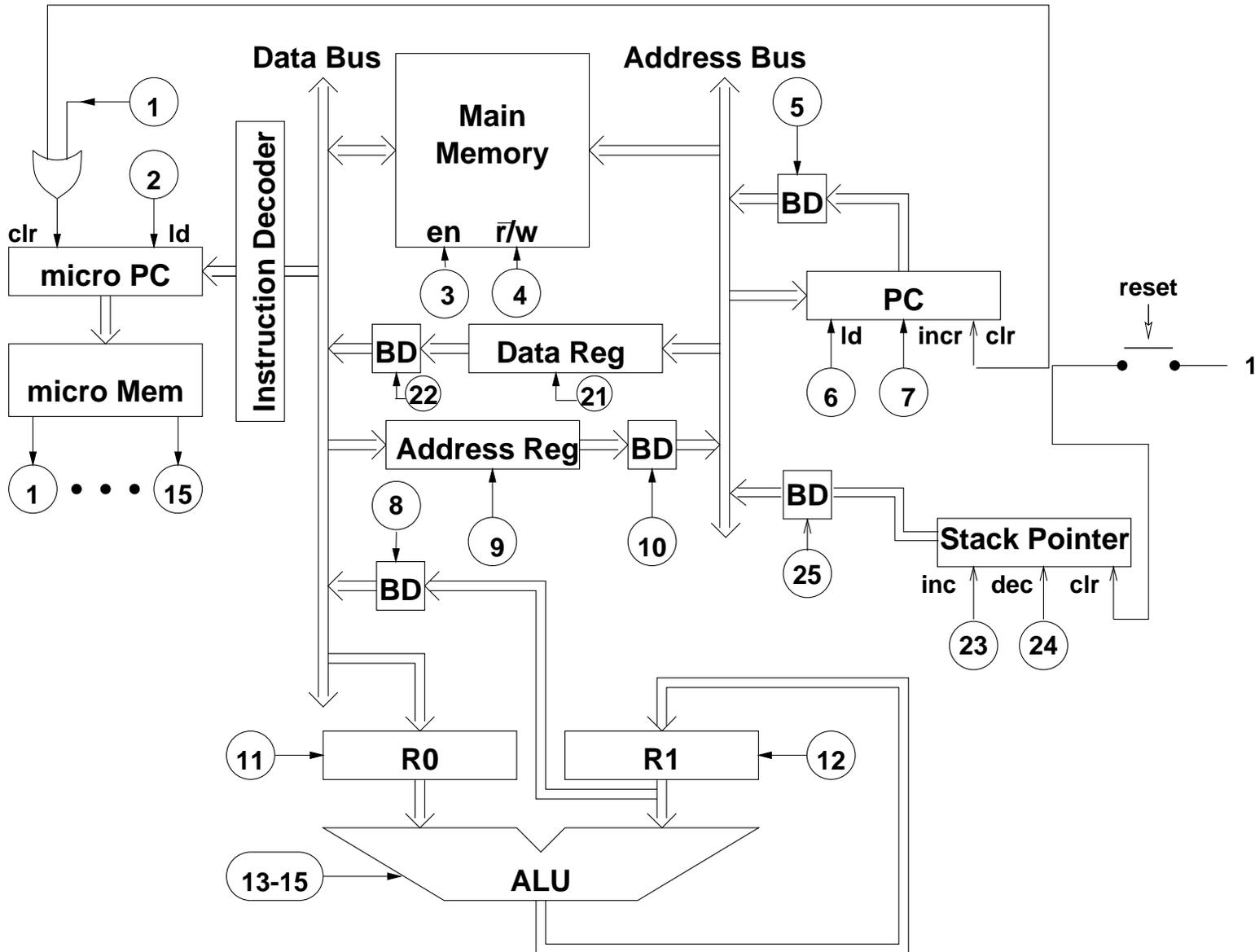
La pile est définie par un registre : pointeur de pile (sp, stack pointer)

Ce registre doit permettre l'incrémentatation et la décrémentatation

Plusieurs conventions de la pile sont possibles :



# Modifications pour la pile



# Instructions jsr et ret avec pile

## Utilisation :

```
f:    ...           h:    ...
      jsr h         ret
      ...

g:    ...
      jsr h
      ...
```

## Description jsr :

1. empiler PC
2. chercher l'adresse donnée dans PC

## Description ret :

1. mettre le sommet de la pile dans PC
2. dépiler

## Micro programme pour jsr et ret avec pile

jsr

5 3 9 7

5 21 24

25 22

25 22 4

25 22 4 3

25 22 4

10 6 1

ret

25 3 9

10 6 23 1

## **Utilisation de la pile pour passage de paramètres**

**Avant de faire un jsr, empiler les arguments à fournir au sous programme**

**Le sous programme peut dépiler**

**Nous avons donc besoin de deux instructions de plus :**

**push**

**empiler le contenu de R1**

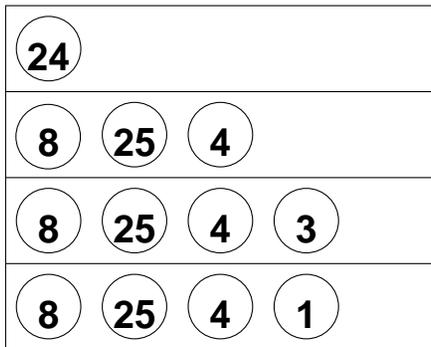
**pop**

**mettre le sommet de la pile dans R0 et dépiler**

**Ces deux instructions sont réalisables avec l'architecture actuelle**

## Implémentation de push et de pop

push :



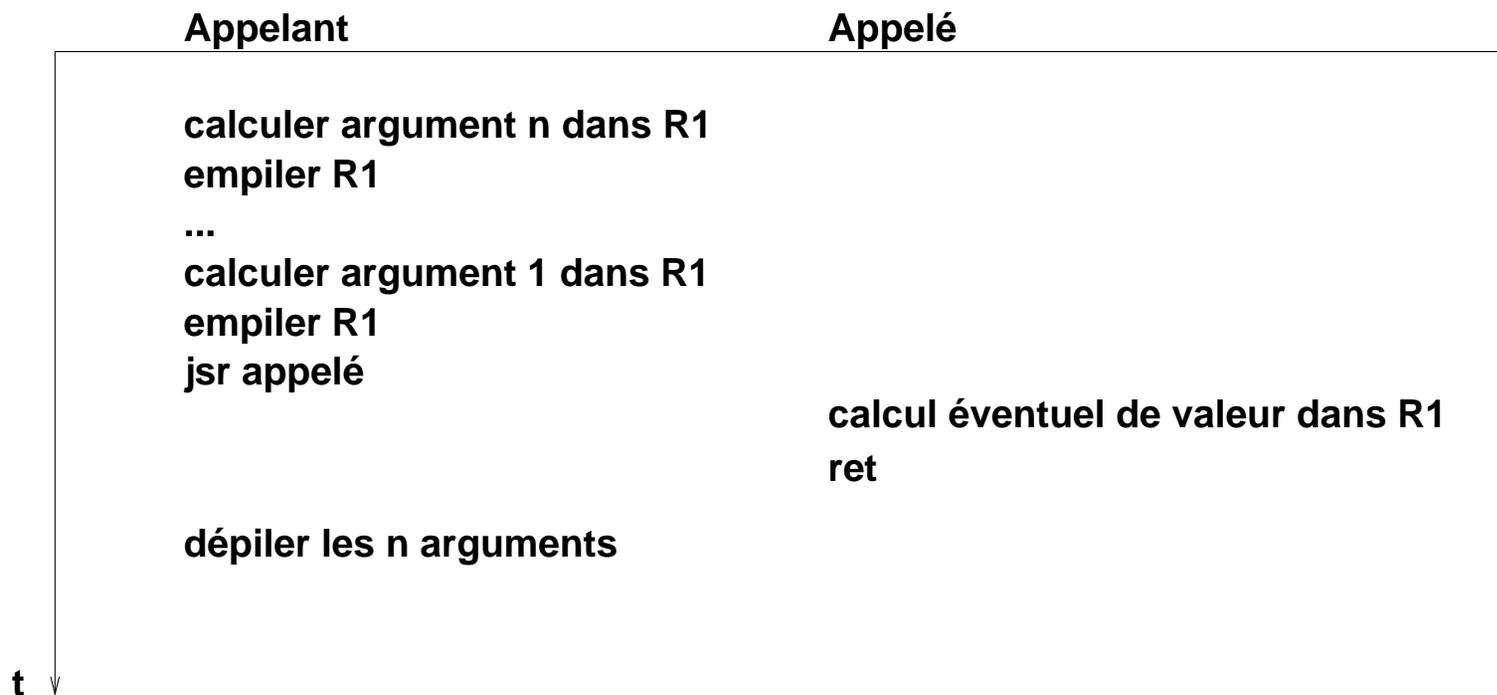
pop :



## Protocole d'appel de sous programmes

Un protocole d'appel est un ensemble de règles précises pour déterminer le partage de responsabilités entre un sous programme appelant et un sous programme appelé.

Exemple de protocole d'appel (plusieurs implémentations du langage C) :



## **Accès aux arguments dans un sous programme**

**Actuellement, la seule possibilité est d'utiliser pop, ce qui n'est pas pratique**

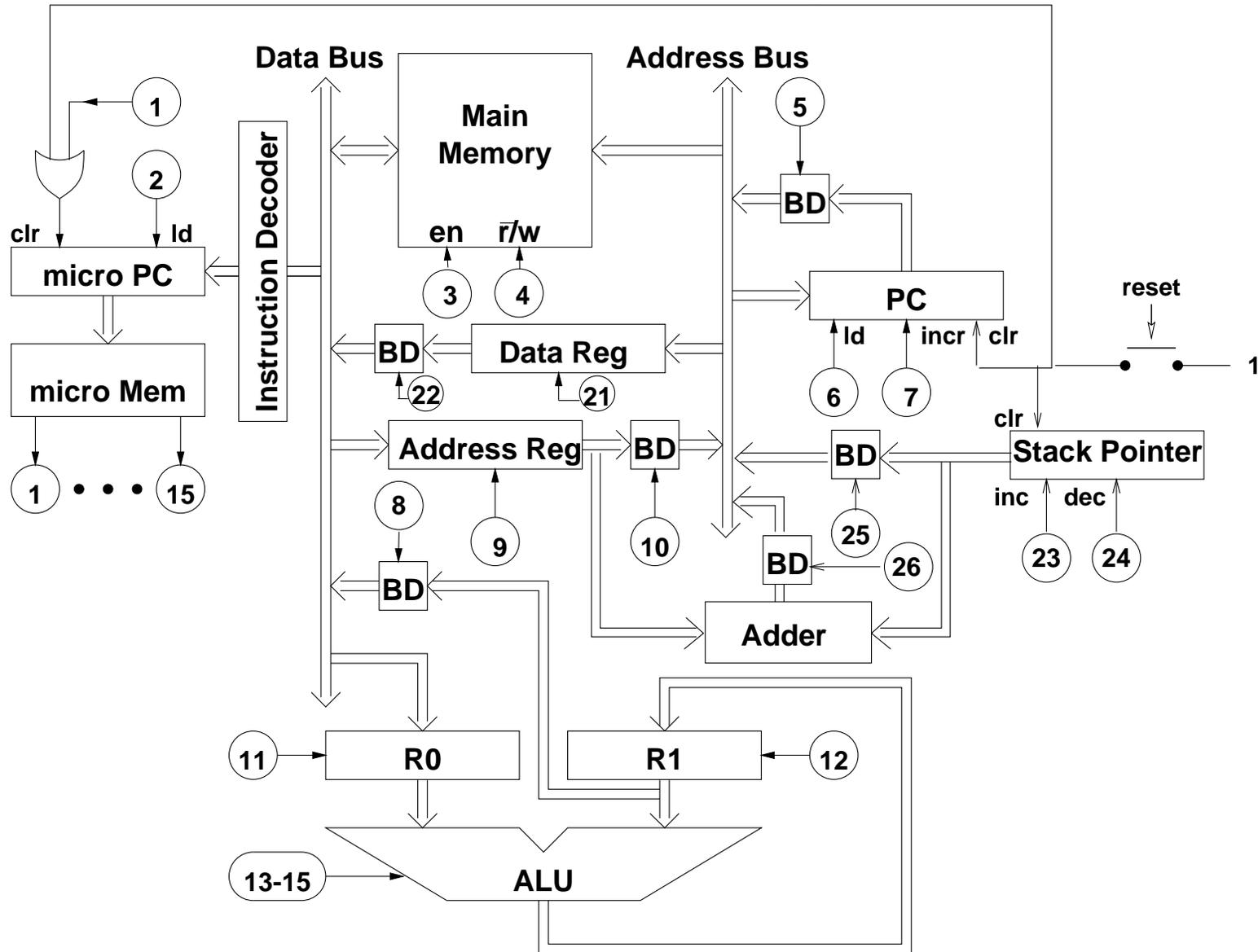
**Il nous faut une possibilité d'accéder à une valeur en mémoire dont l'adresse est  $sp+constante$**

**De cette manière, le sous programme peut récupérer argument  $i$  avec l'adresse  $sp+(i+1)$**

**Nous avons donc besoin d'additionner le contenu de  $sp$  avec une constante**

**Cette constante sera stockée après le code d'instruction, et copiée dans le registre d'adresses**

# Modifications pour l'accès aux arguments



## Nouvelle instruction pour l'accès aux argument

Il est maintenant possible d'écrire une instruction lds (load from stack).

5	3	9	7
26	3	11	1

## **Dépiler les arguments**

**Après le retour de l'appelé, l'appelant doit dépiler les arguments**

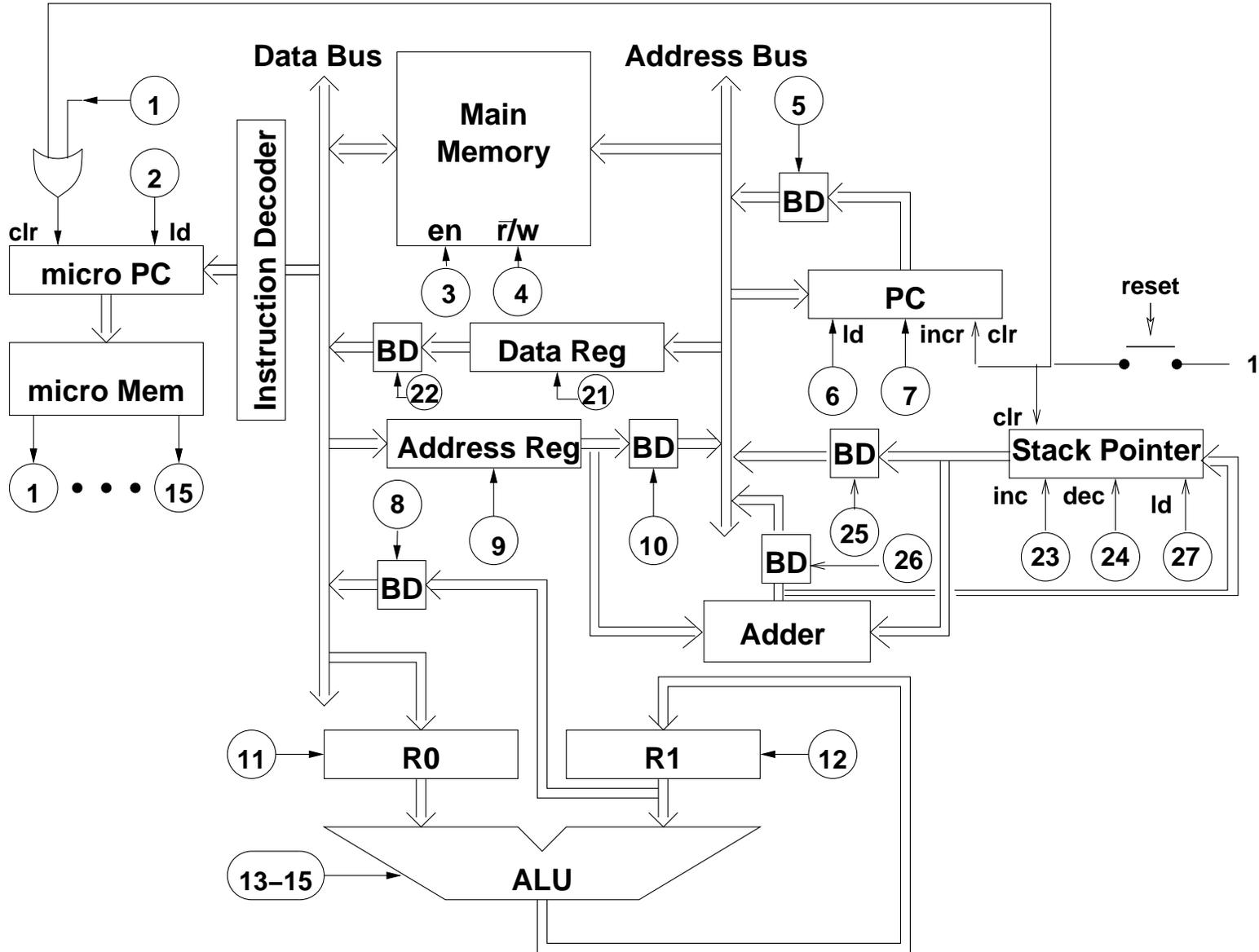
**Actuellement, la seule possibilité est d'utiliser pop**

**Une amélioration possible est de pouvoir additionner une constante à sp**

**Il est possible d'utiliser l'additionneur existant, mais il faut pouvoir stocker le résultat dans sp**

**Ceci nécessite des fils supplémentaires, et une modification de sp pour permettre un signale ld**

# Modifications pour l'arithmétique de sp

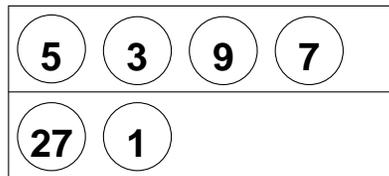


## Nouvelle instruction pour l'arithmétique de sp

Il est maintenant possible d'écrire une instruction addsp

L'addition est toujours modulo 256, donc pas besoin de subsp

Voici l'implémentation



## Variables locales à un sous programme

Avec les instructions maintenant à notre disposition, il est possible d'avoir des variables locales allouées dans la pile

Voici le format de la pile pour un sous programme avec n arguments et m variables locales :

<b>var loc m</b>
<b>⋮</b>
<b>var loc 1</b>
<b>adresse de retour</b>
<b>arg 1</b>
<b>⋮</b>
<b>arg n</b>

## Nouveau protocole d'appel

**Appelant**

**calculer argument n  
empiler R1**

**...**

**calculer argument 1  
empiler  
jsr appelé**

**addsp n**

**Appelé**

**addsp -m  
calculer en utilisant lds, sts  
addsp m  
ret**

## **Entrées/Sorties**

**L'ordinateur doit pouvoir communiquer avec des unités externes  
(disque, clavier, écran, . . .)**

**Les unités sont banchées sur le bus d'adresses et le bus de données**

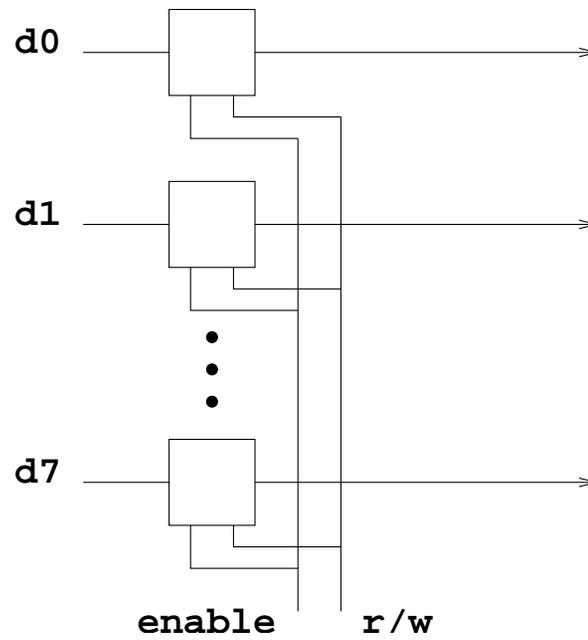
**Elles se comportent de la même façon que la mémoire principale**

**De cette manière, on peut utiliser l'instruction st pour la sortie, et  
l'instruction ld pour l'entrée de données**



## Sortie (suite)

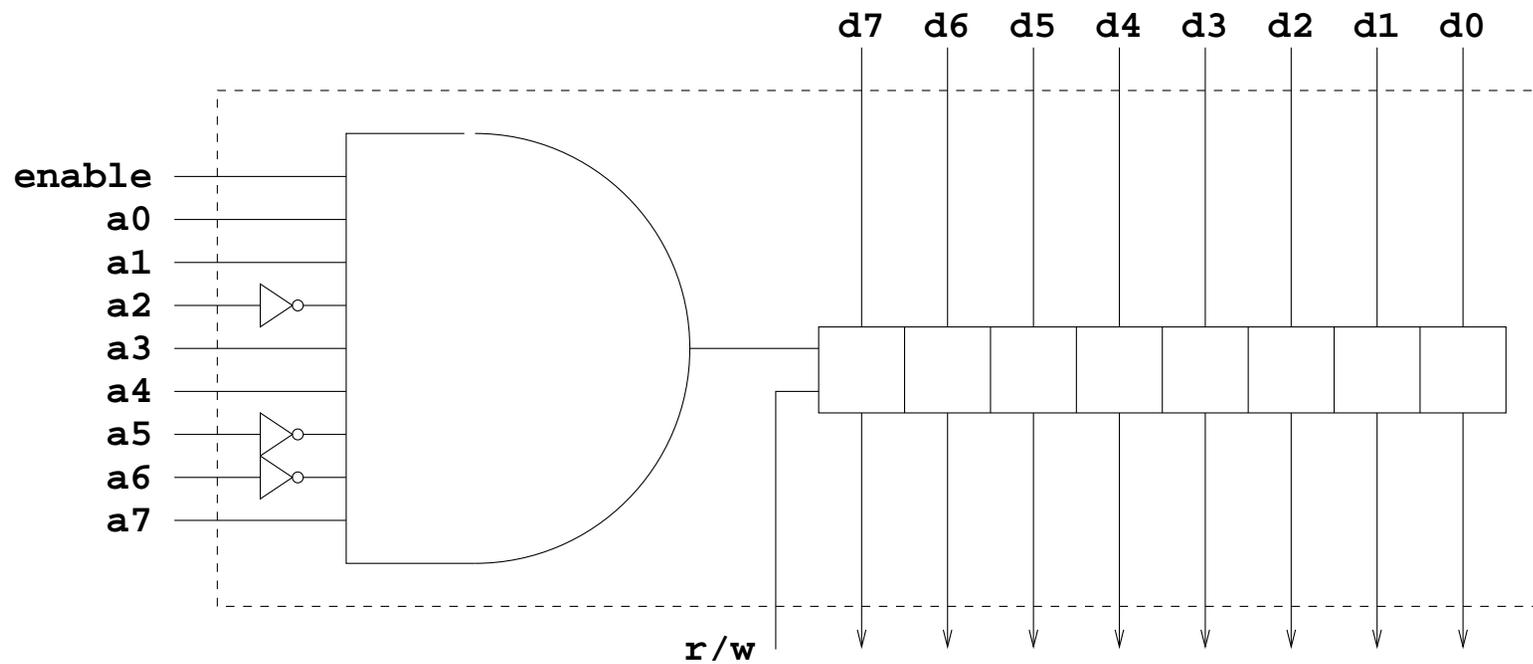
Pour avoir un registre à (par exemple) 8 bit, il suffit d'en mettre 8 en parallèle :



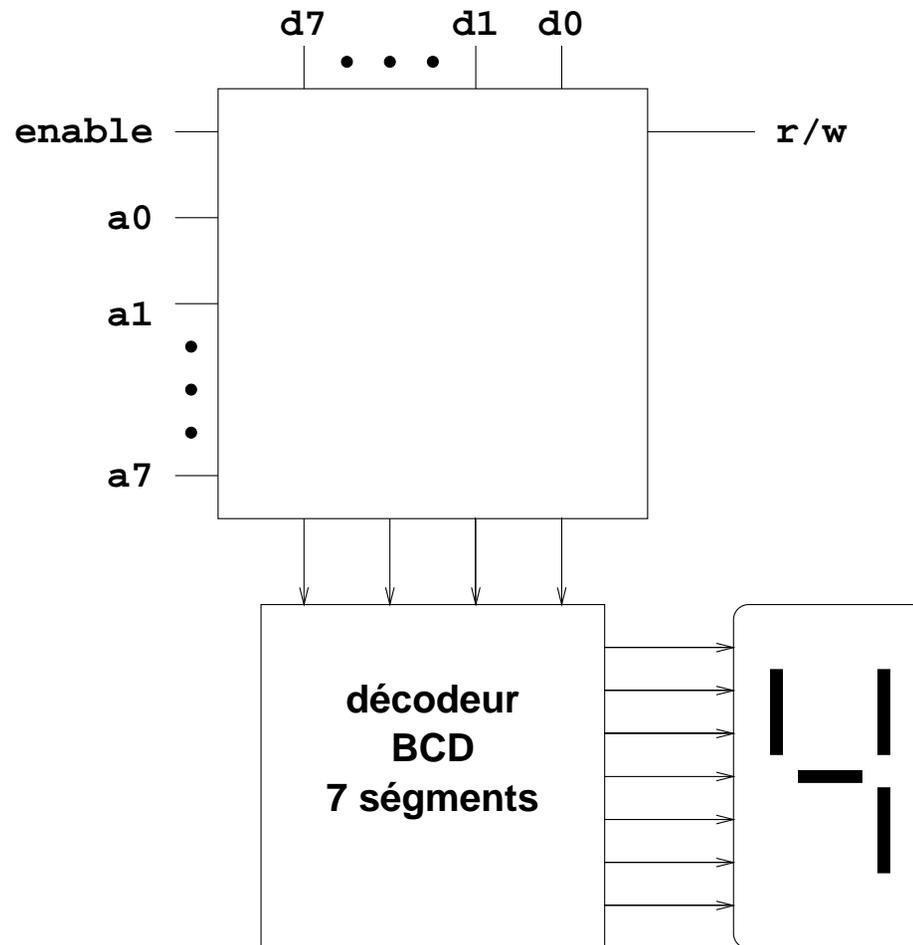
## Sortie (suite) Décodage d'adresses

Pour le décodage d'adresses, il faut une porte et des inverseurs.

Exemple : nous souhaitons utiliser l'adresse 10011011



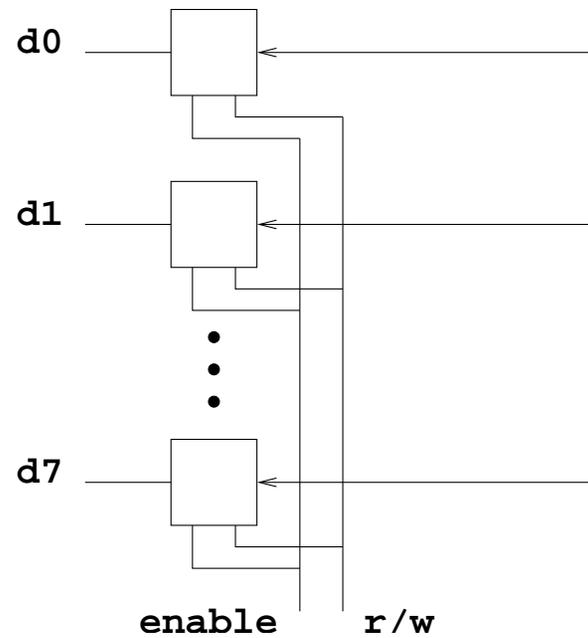
## Sortie (suite) Exemple





## Entrée (suite)

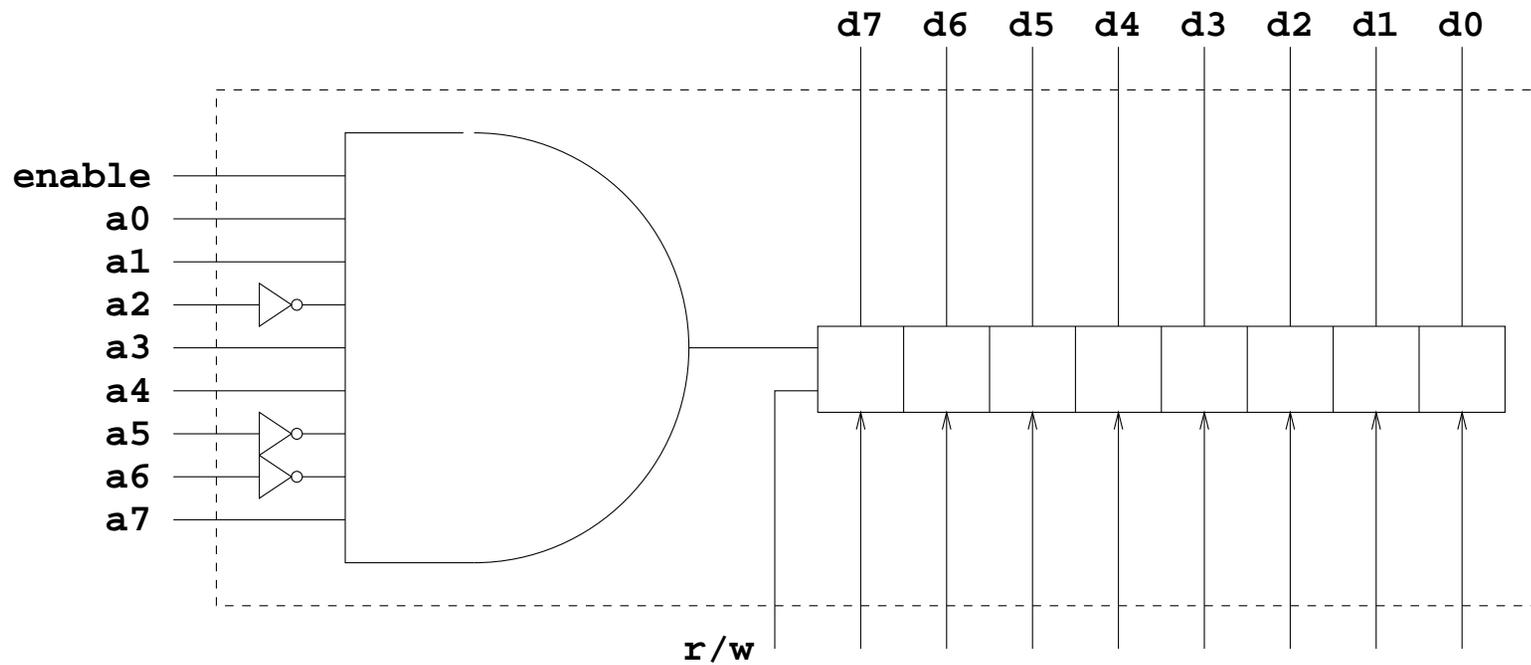
Pour avoir (par exemple) 8 bit, il suffit d'en mettre 8 en parallèle :



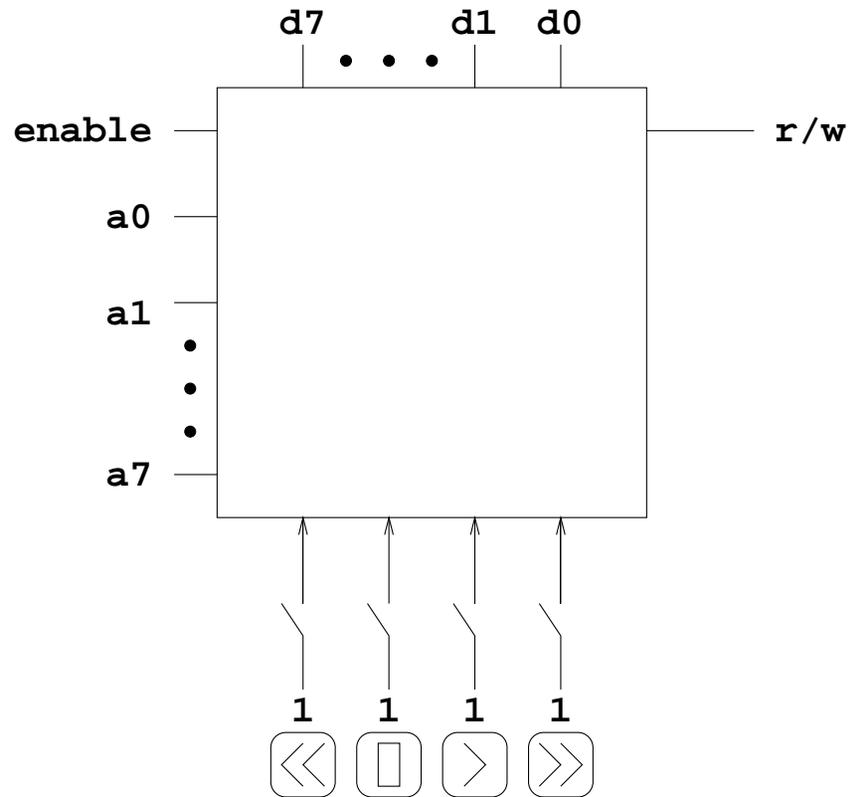
## Entrée (suite) Décodage d'adresses

Pour le décodage d'adresses, il faut une porte et des inverseurs.

Exemple: nous souhaitons utiliser l'adresse 10011011



# Entrée (suite) Exemple



## **Entrée (suite)**

### **Problème :**

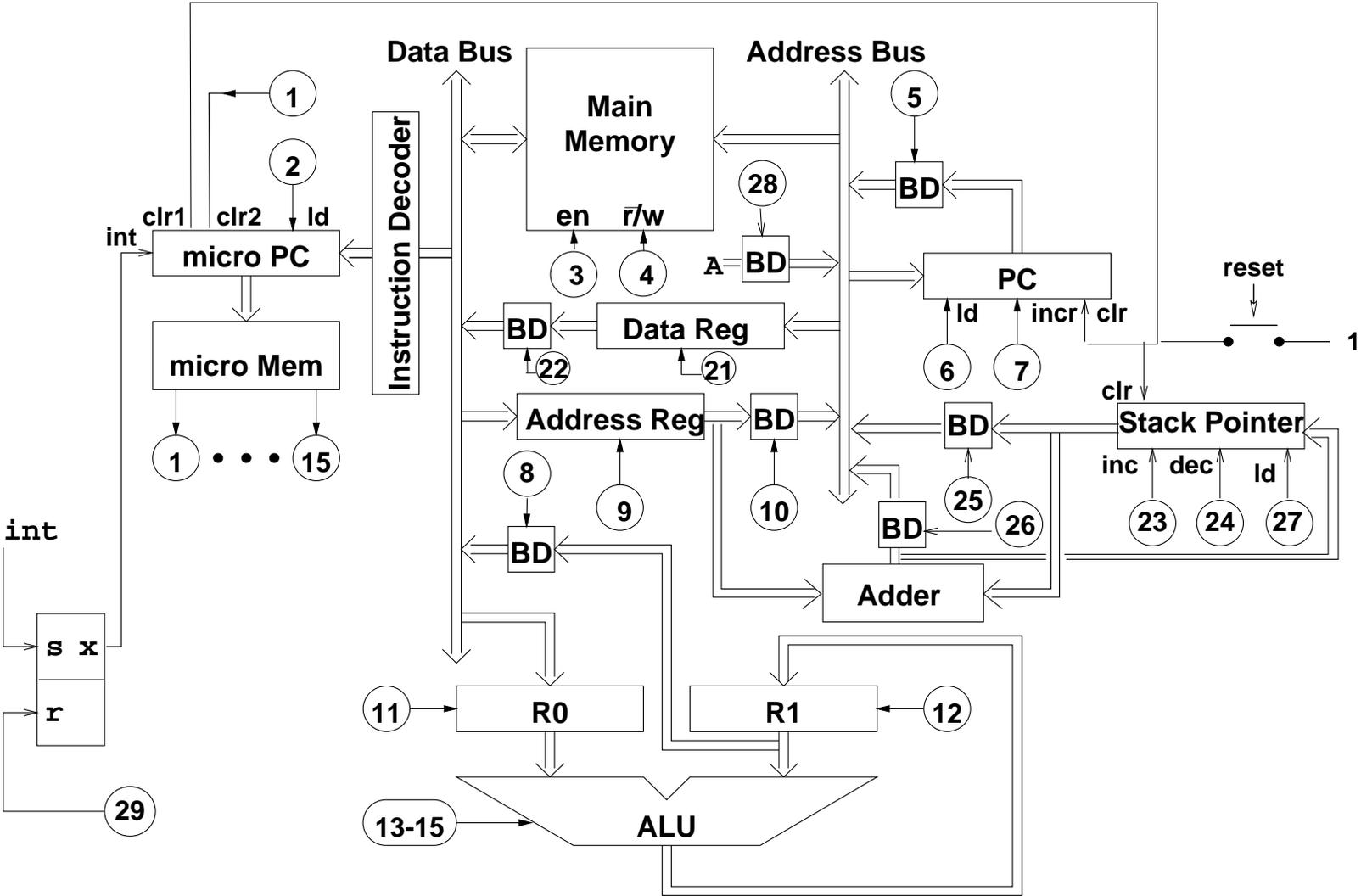
**L'unité centrale doit tester périodiquement la disponibilité d'entrée**

### **Solution :**

**Interruptions. L'unité externe sera capable d'interrompre le traitement normal de l'unité centrale**

**C'est comme une instruction jsr provoquée par un événement externe**

# Modifications pour les interruptions



## Modification de micro PC

<u>clr1</u>	<u>clr2</u>	<u>int</u>	<u>ld</u>	
1	-	-	-	<b>zéro</b>
0	0	0	0	<b>incrémentation</b>
0	0	0	1	<b>chargement</b>
0	0	1	0	<b>incrémentation</b>
0	0	1	1	<b>chargement</b>
0	1	0	0	<b>zéro</b>
0	1	0	1	<b>ne peut pas arriver</b>
0	1	1	0	<b>chargement d'une constante c</b>
0	1	1	1	<b>ne peut pas arriver</b>

**La constante c correspond à la première adresse en micro mémoire de l'implementation du micro programme pour traiter les interruptions**

## Micro programme pour les interruptions

29	5	21	24
25	22	4	
25	22	4	3
25	22	4	
28	6	1	

## **Différences entre notre architecture et une architecture réelle**

**Principalement deux types d'architecture :**

**CISC (Complex Instruction Set Computer)**

**Pentium, Vax, 68000, . . .**

**RISC (Reduced Instruction Set Computer)**

**MIPS, SPARC, PowerPC, Alpha, . . .**

## CISC/RISC

### CISC

Relativement peu de registres

Plusieurs modes d'adressage  
(immediate, direct, indirect, offset,  
indexed, indexed indirect, . . .)

Format des instructions irrégulier

Instructions souvent très complexes

Passage d'arguments dans la pile

JSR empile l'adresse

Registres souvent asymétriques

### RISC

Beaucoup de registres

Peu de mode d'adressage  
utilisation de load/store et  
load immediate

Format des instructions régulier

Instructions simples

Passage d'arguments dans des registres

JSR copie PC dans un registre

Registres souvent symétriques

# Optimisations

**L'exécution d'une instruction peut commencer avant que l'exécution de l'instruction précédente ne soit terminée (pipeline)**

**Exécution d'une instruction simple en un cycle d'horloge**

**Plusieurs unités d'exécution d'instructions permettant à plusieurs instructions d'exécuter en parallèle**

**Unité flottante séparée permettant l'exécution d'une opération flottante et d'une autre opération en même temps**

**Opérations sur les flottants très rapides  
(addition en < 3 cycles, multiplication en < 5 cycles)**

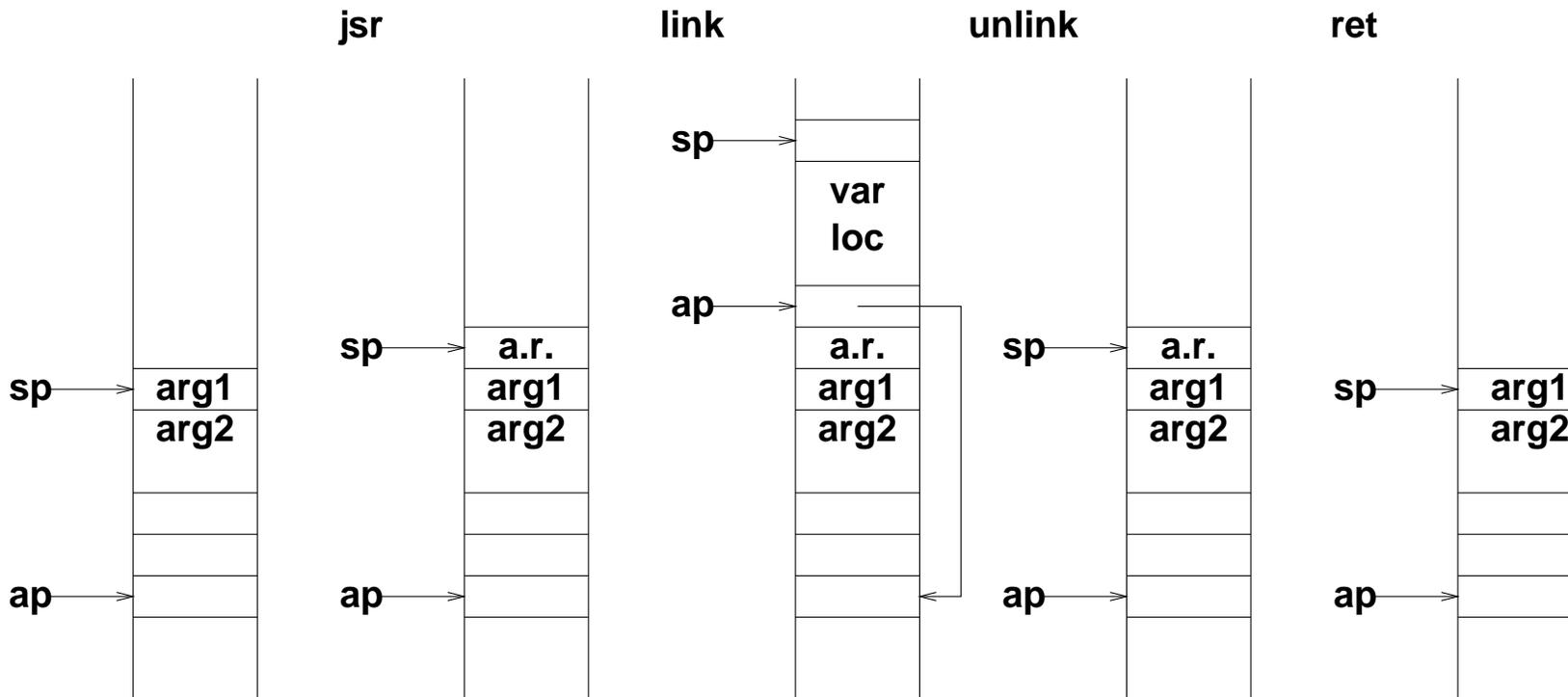
**Prédiction de branchements (= sauts) permettant l'exécution d'une instruction jump en un cycle d'horloge**

**Prédiction de l'adresse de retour de l'instruction ret par une pile interne**

# Instructions pour les langages haut niveau

Registre supplémentaire (bp ou ap) pour les arguments d'une fonction

Instructions link/unlik pour allouer/desallouer les variables locales

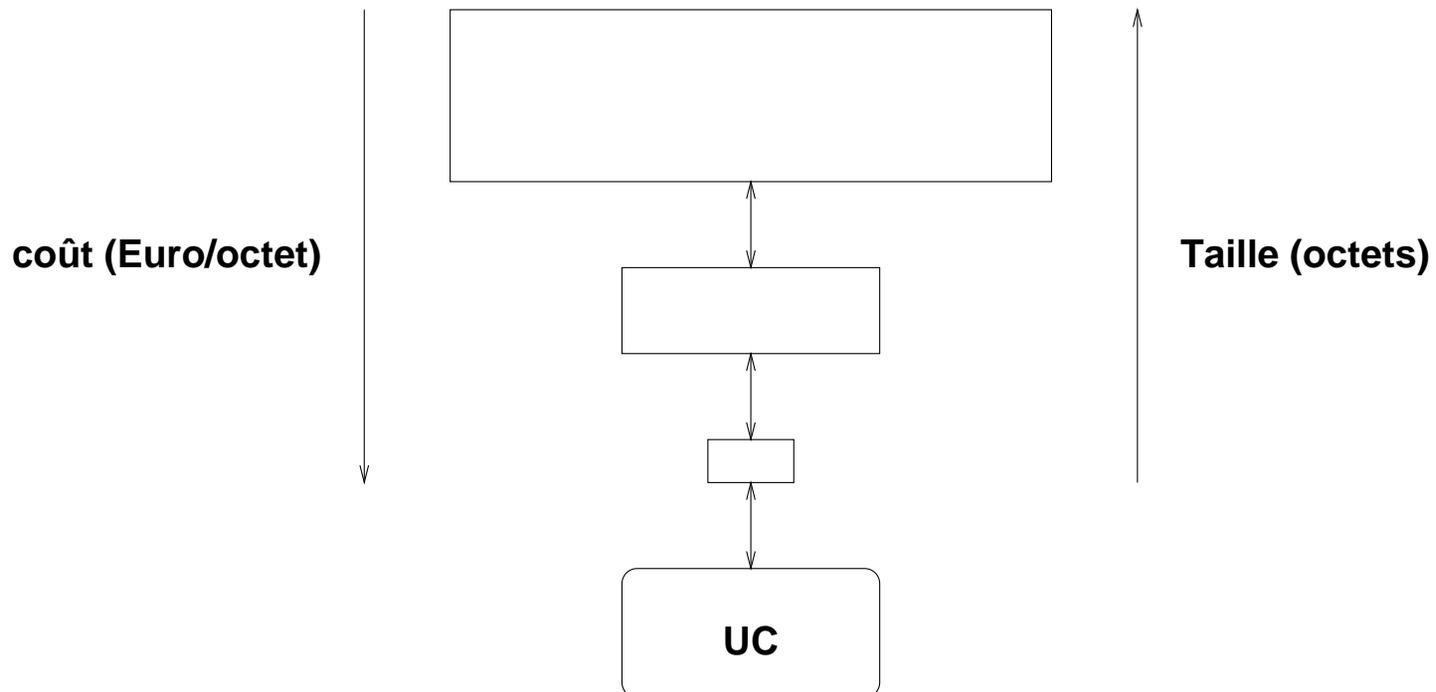


# Mémoire Cache

**Problème : la mémoire est beaucoup plus lente que le processeur (facteur 10)**

**Les mémoires aussi rapides que le processeur coûtent trop chères**

**La solution : une mémoire à plusieurs niveaux :**



# Mémoire Cache

**Il y a souvent 3 couches : disque, mémoire principale, et mémoire cache**

**L'objectif est de faire croire que la mémoire entière est de la taille de celle que est la plus grande, et du coût de celle qui est la moins chère**

**La raison que cela marche souvent est le principe de localité :**

**Une suite d'accès à la mémoire n'est pas distribuée de façon aléatoire. Les adresses des accès sont regroupées en un petit nombre de groupes. Les adresses d'un groupes font référence à un petit intervalle en mémoire**

**Raison du principe :**

**Code séquentiel plus boucles de petite taille**

**Données organisées en tableaux et structures**

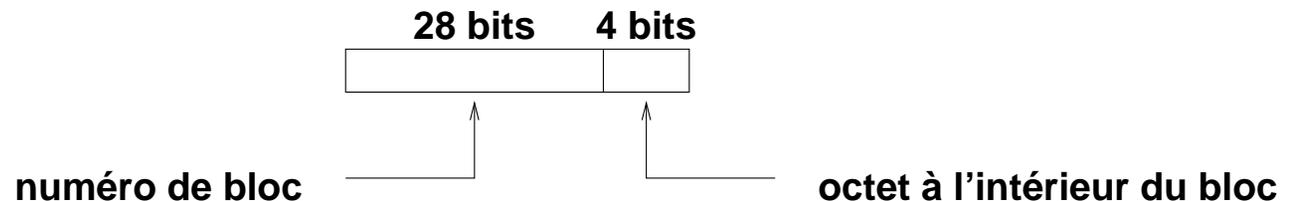
## Cache, idée de base

L'idée de base est donc de copier un intervalle de code ou de données de la mémoire lente à la mémoire rapide pour un accès plus rapide plus tard

Cet intervalle est souvent de taille 16 octets (actuellement). Nous appelons un tel intervalle un "bloc" de mémoire

Une adresse est donc conceptuellement constituée par deux parties, le numéro de bloc, et l'octet à l'intérieur du bloc (si les adresses de la machine correspondent à des octets).

Exemple (adresses de 32 bits):



## **Cache, chargement de blocs**

**Quand un bloc est référencé pour la première fois, le microprogramme le charge en mémoire cache**

**Si le bloc est référencé de nouveau, le cache est capable de satisfaire la référence sans accès à la mémoire principale**

**Quand le cache est plein, il faut choisir un bloc à supprimer**

**Le choix du bloc à supprimer est déterminé par le type de cache et par l'algorithme utilisé**

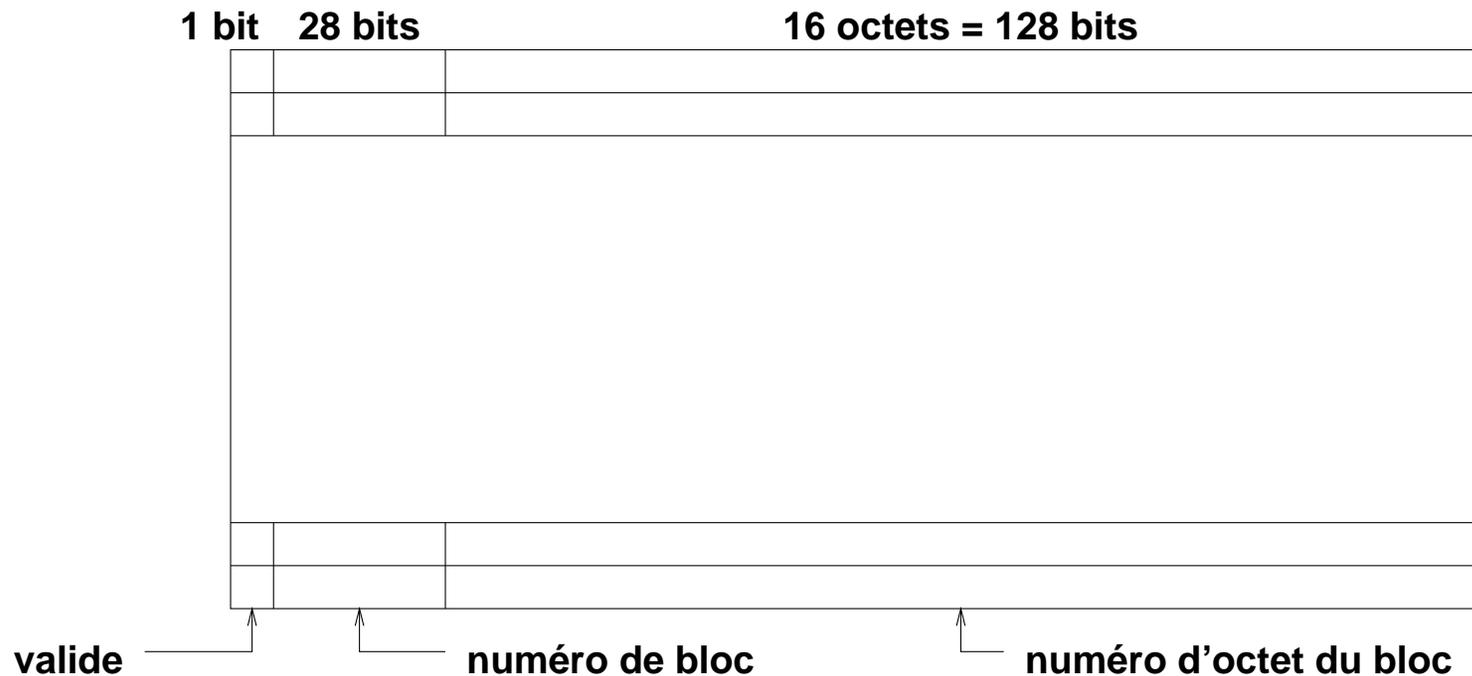
**L'algorithme utilisé est souvent une version de LRU (least recently used, moins récemment utilisé), mais il y en a d'autres (random, ...)**

**Il reste donc la différence entre différents types de cache. On utilise le mot "associativité" pour décrire cette différence**

## Cache associatif

Le cache est composé d'un nombre de "lignes de cache"  
(ou simplement "ligne", anglais : cache lines), égal au nombre de blocs

Chaque ligne contient un bit pour indiquer si le contenu de la ligne est valide,  
un nombre de bits assez large pour déterminer un numéro de bloc de façon  
unique (dans notre exemple précédent : 28 bits), et le bloc (16 octets dans notre  
exemple, soit 128 bits)

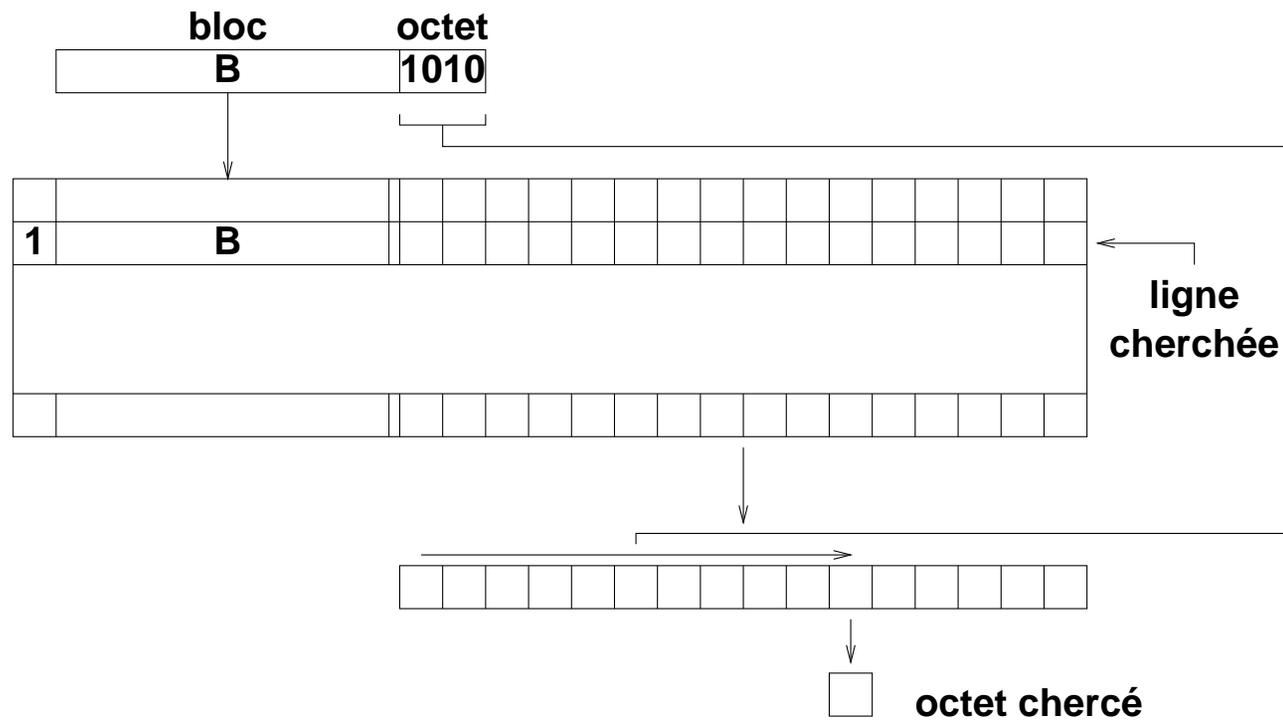


# Cache associatif

Initialement, le bit "valide" est 0 pour toutes les lignes

Quand un bloc est chargé, le bit correspondant est mis à 1

Quand un accès à l'adresse A est tenté, le microprogramme vérifie si le bloc est dans le cache:



## **Cache associatif**

**Pour que l'accès soit rapide, le numéro de bloc cherché doit être comparé en parallèle avec toutes les lignes simultanément**

**Ceci nécessite un circuit spécial, très coûteux**

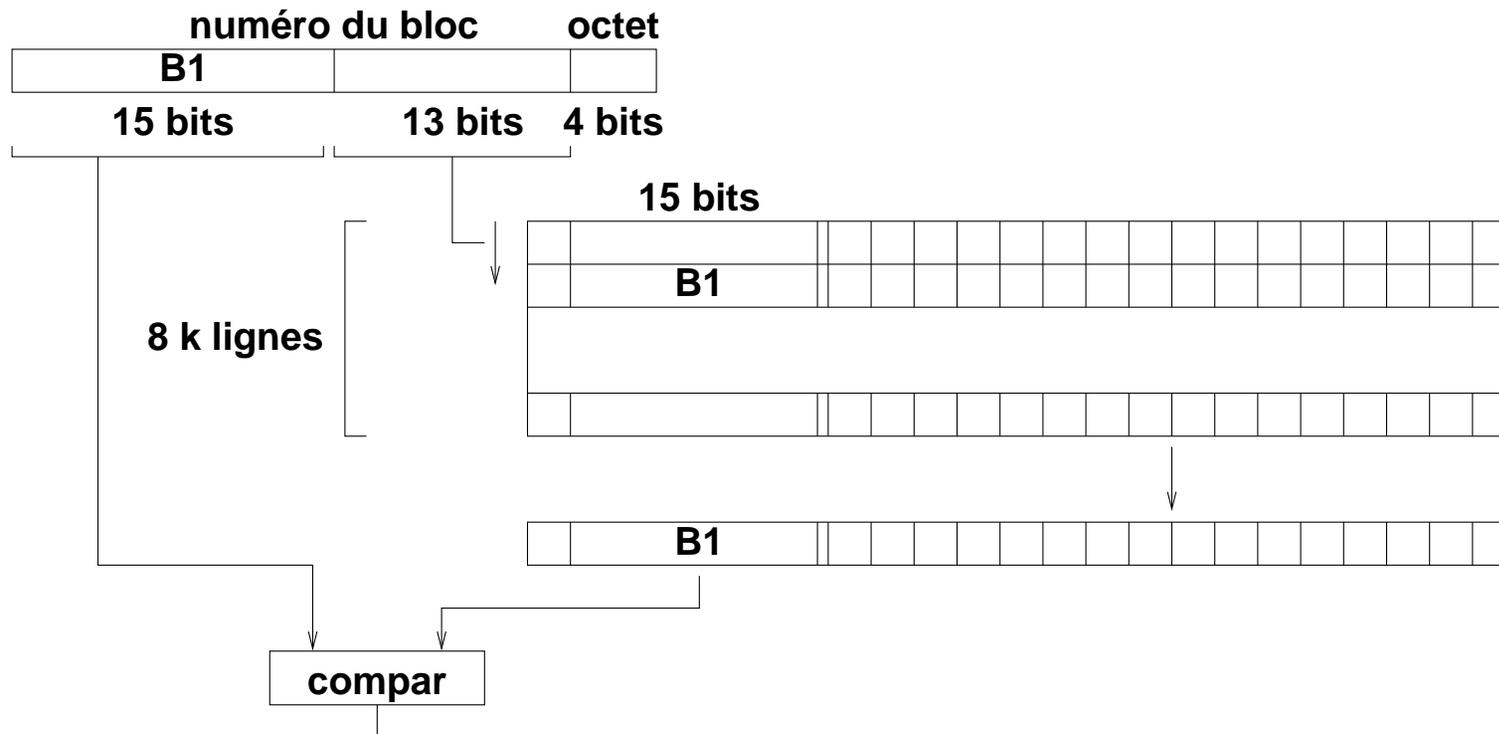
**Pour cette raison, on n'utilise pas souvent un cache associatif**

# Cache direct

Un bloc ne peut pas être placé n'importe où dans le cache

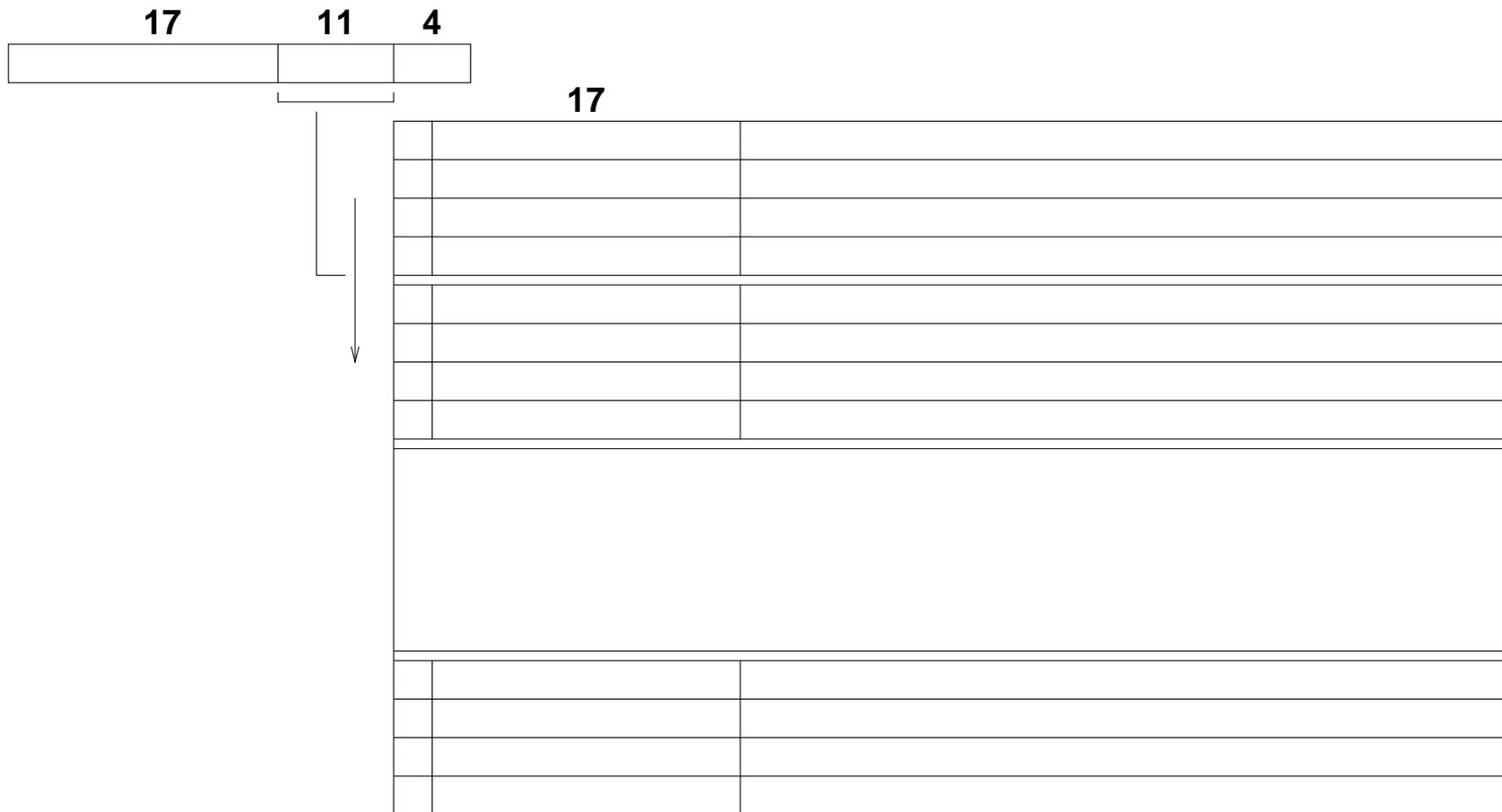
Chaque bloc en mémoire principale a sa place prédéterminée par les bits les moins significatifs de son numéro.

Exemple (taille du cache 128 ko, adresses à 32 bits, taille d'un bloc 16 octets) :



## Cache associatif à N groupes

C'est une solution intermédiaire entre associatif et direct (ex N=4, taille 128 ko)



## **Cache associatif à N groupes**

**Ce type de cache est moins sensible à certains motifs d'accès**

**Le cache associatif et le cache direct sont des cas spéciaux de ce cache (N = 1 pour le cache direct, N = M pour le cache associatif, où M est le nombre de lignes)**

**Une bonne valeur de N est 2-4.**

**N est parfois appelé l'"associativité du cache"**

## **Choix des paramètres**

### **Taille d'un bloc**

**Le surcoût dû au circuits supplémentaires dépend du nombre de blocs**

**Pour une taille fixe de cache, il est donc préférable d'avoir des blocs de taille importante**

**Mais cela augmente la probabilité qu'une partie du bloc sera inutile**

**Une bonne valeur est 4 ou 8 mots (de 32 ou 64 bits selon la machine)**

### **Associativité**

**Le coût augmente de manière considérable avec l'associativité**

**Il est rare pour un programme d'avoir besoin d'une associativité supérieure à 4**

**Une associativité de 2 est le minimum**

**Une bonne valeur est 2 ou 4**

## **Influence du cache sur la programmation**

**C'est une bonne idée d'aligner le code et les données sur les lignes de cache**

**Un sous programme et un tableau devrait donc commencer sur une adresse multiple de la taille d'un bloc**

**Le compilateur s'occupe du code et malloc s'occupe des données**

**Le compilateur s'occupe également des données statiques**

**Il faut éviter des boucles et des tableaux dont la taille dépasse celle du cache (pour un cache associatif, c'est la seule considération)**

**Pour un cache direct, il faut éviter une distance entre appelant et appelé multiple de la taille du cache, ainsi que la manipulation simultanée de données à distance multiple de la taille du cache**

**Ces considération influencent rarement la programmation d'applications, mais souvent la programmation système (compilateurs, gestionnaires de la mémoire, ...)**

## **Stratégie d'écriture du cache**

### **Écriture immédiate**

**Chaque écriture provoque une écriture en mémoire principale**

**Si le nombre d'écriture est considérable, ça peut diminuer l'efficacité du cache**

**C'est facile à implémenter**

### **Écriture différée**

**Écriture en mémoire principale uniquement en cas de besoin (changement de processus, écriture sur disque, synchronisation)**

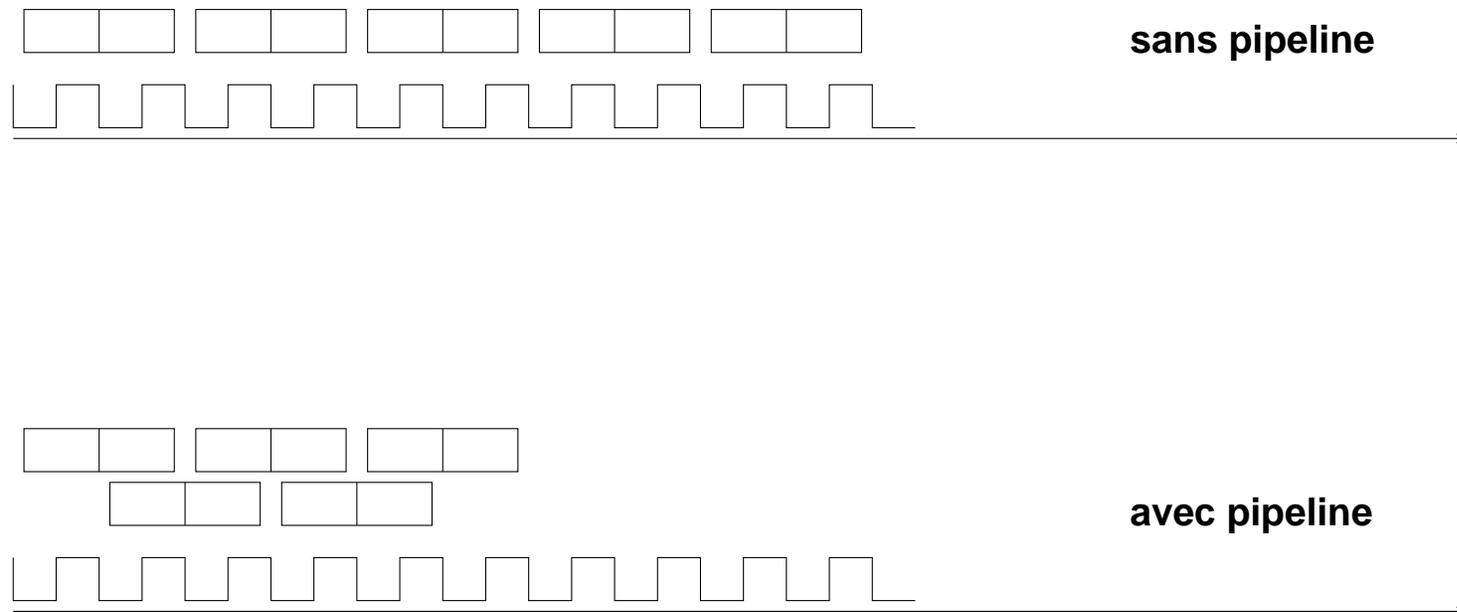
**Plus difficile à implémenter**

# Pipeline

**C'est une façon d'augmenter le nombre d'instructions exécutées par unité de temps sans diminuer le temps d'exécution des instructions**

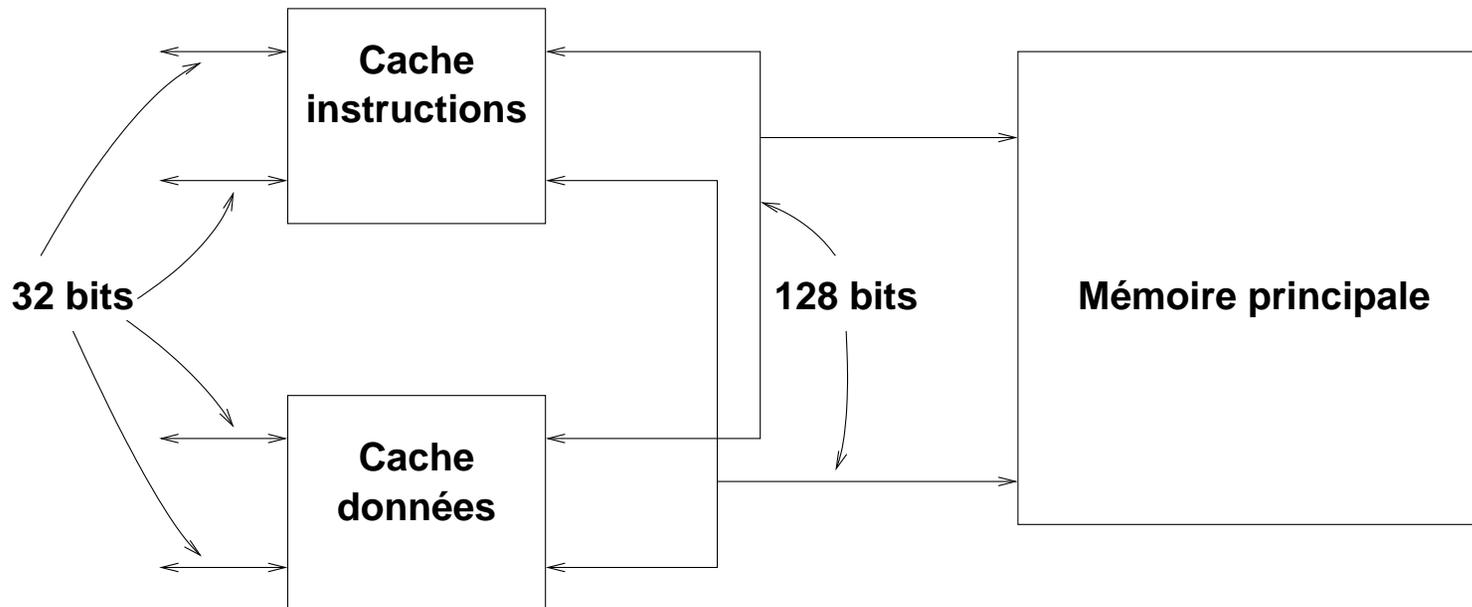
**Il faut donc exécuter plusieurs instructions en parallèle**

**Exemple, dans notre architecture, les instructions arithmétiques n'utilisent pas les bus. On peut alors charger l'instruction suivante en même temps**



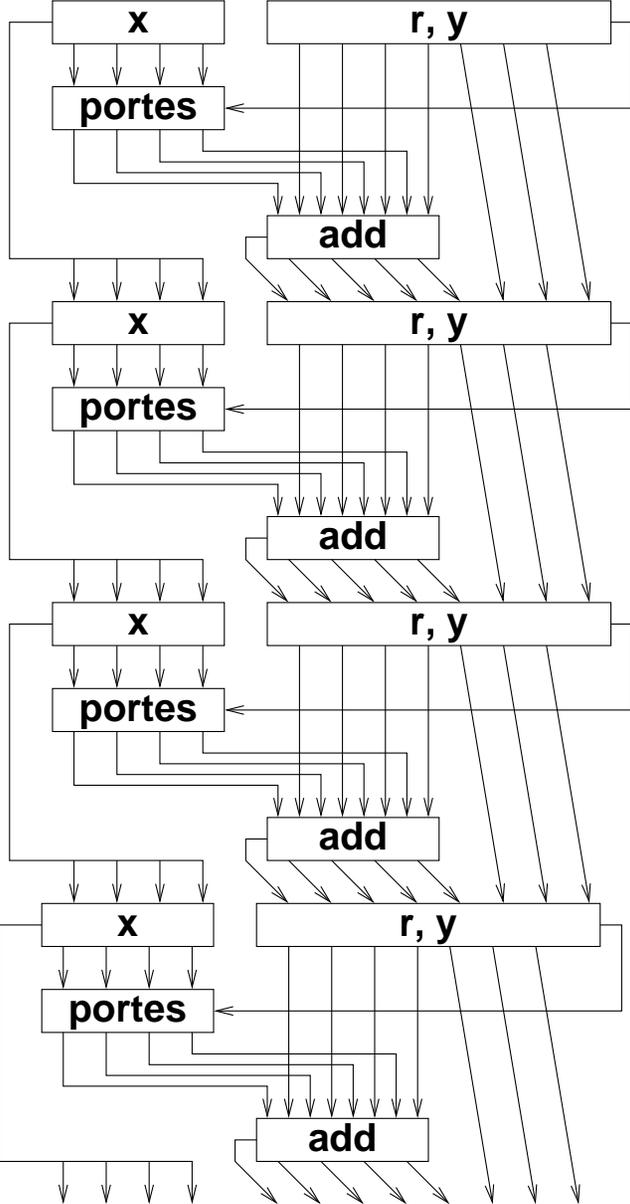
# Pipeline

On peut souvent éviter l'utilisation simultanée du bus par le chargement et l'exécution d'une instruction même si son exécution a besoin de données en mémoire. Pour cela il faut un cache pour les instructions et un cache pour les données :

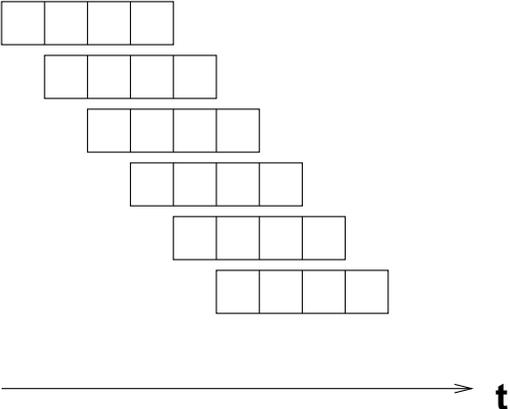




# Pipeline



Exécution :



# **Multiprocesseurs**

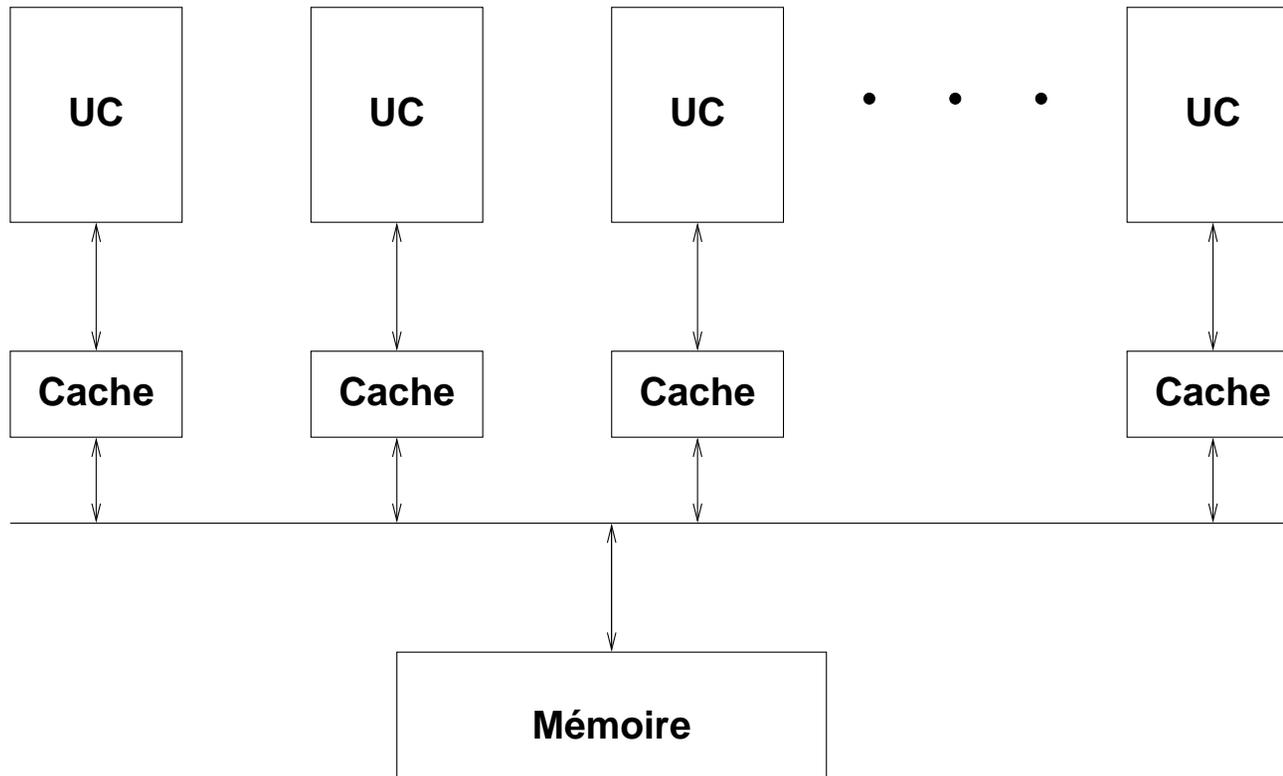
**Plusieurs types selon si la mémoire est partagée ou distribuée et selon la méthode d'interconnexion**

**Nous allons parler du type SMP (Symmetric Multi Processor)**

**Il s'agit d'un système à mémoire partagée avec un nombre relativement faible de processeurs (2–16) pour l'instant**

# SMP

Pour éviter que le bus devienne un embouteillage, on préfère mettre un cache pour chaque processeur :



## **SMP**

**Plusieurs caches peuvent contenir une copie du même bloc**

**Si c'est pour lecture uniquement, il n'y a pas de problème**

**Pour l'écriture, il y a un problème**

**Il faut éviter l'écriture simultanée par plusieurs processeurs**

**Il faut informer les autres processeurs des modifications**

**Pour résoudre ce problème on utilise un mécanisme de furetage (snooping)**

## **Furetage**

**La lecture se passe comme avant**

**Un processeur P qui souhaite écrire, émet un message d'invalidation sur le bus**

**Les autres processeurs Q1, Q2, ... vérifient s'ils ont une copie du bloc**

**Si oui, alors le bloc est invalidé, et le processeur Qi correspondant est forcé de le charger à nouveau s'il en a besoin**

**Les écritures suivantes du bloc par P n'ont pas besoin de message d'invalidation**

**Quand un autre processeur Qi demande le bloc, le contenu du bloc est envoyé sur le bus par P. Puis le bloc est invalidé par P**

**Si deux processeurs P1 et P2 souhaitent écrire simultanément, l'ordre d'exécution est déterminé arbitrairement**

# Synchronisation

**Nécessaire avec un système multiprogrammé**

**Encore plus nécessaire avec un multiprocesseur**

**Exemple : accès au disque par un seul processus à la fois**

**Les processeurs modernes ont des instructions spécialisées pour la synchronisation**

**Une telle instruction est "swap" (échanger). Le contenu d'un registre et d'un mot en mémoire sont échangés de façon atomique**

## **Synchronisation avec l'instruction swap**

**Une adresse A en mémoire sera 1 si une ressource est utilisée, 0 sinon**

**Pour réserver la ressource, un processeur P fait :**

- 1. Mettre 1 dans un registre R**
- 2. SWAP A et R**
- 3. Si R contient 1 alors répéter à partir de 2**
- 4. Quand on arrive ici, on peut utiliser la ressource en exclusivité**

**Pour libérer la ressource, il suffit de mettre 0 dans A**

## **Synchronisation (suite)**

**Avec un seul processeur, chaque instruction est atomique (c'est pour ça que nous avons décidé de tenir compte des interruptions après terminaison de l'exécution de l'instruction)**

**Avec plusieurs processeurs, le problème est plus compliqué**

**Chaque processeur doit s'assurer du bus en exclusivité avant de s'en servir**

**C'est un problème résolu par l'arbitrage du bus (que nous n'avons pas le temps d'en parler)**