# Systèmes digitaux

Cours 10

# Mémoire

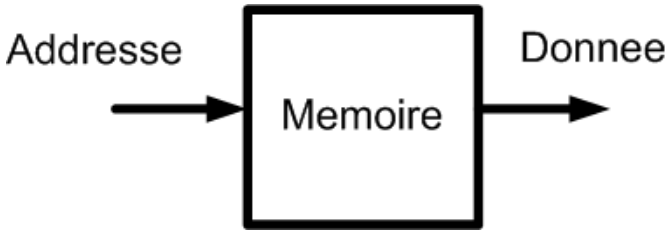
- Plusieurs applications demandent de la mémoire:
  - Il faut stocker des données: musique, image, etc.
- En logique, on utilise des flip flops:
  - Rapide
  - Mais gros
- Pas pratique lorsqu'on veut stocker beaucoup de données

# Mémoire

- On s'intéresse aux mémoires proprices pour une grande quantité de données
    - Plus petite taille
- On distingue 2 familles
    - La mémoire volatile (parfois appellee RAM): s'efface quand on éteint le systeme
    - La mémoire non-volatile (parfois appellée ROM): ne s'efface pas même quand on éteint le système

# Mémoire

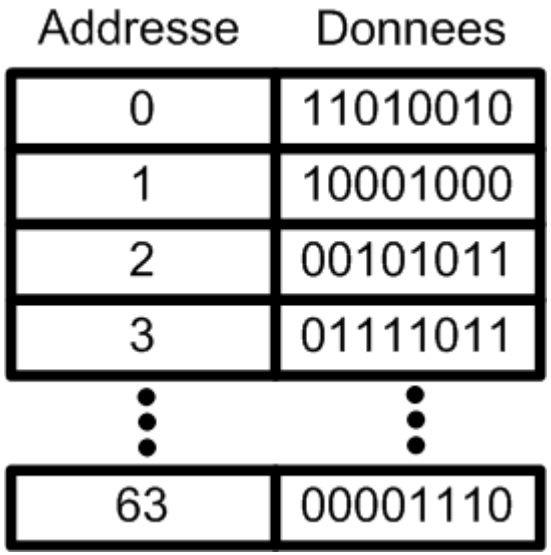
- Une mémoire est arrangée de la façon suivante:




- On spécifie l'addresse en entrée et la mémoire nous sort la donnée située à cet endroit

# Mémoire

- Le stockage dans une mémoire est vue de la façon suivante:




À l'addresse 0, il contient "11010010"

À l'addresse 1, il contient "10001000"

.

.

.

À l'addresse 63, il contient "00001110"

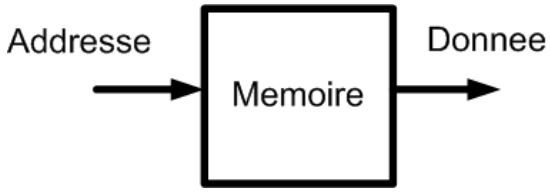
# Mémoire

- La mémoire peut contenir n'importe quelle information qu'on trouve pertinente
- Par exemple, chaque addresse peut contenir
  - Une série d'instructions à exécuter (pensez à un ordinateur)
  - Une séquence de notes musicales (converti en numérique) qu'on a prise
  - Des données pour nous aider à créer une fonction logique
  - Etc.

Voyons maintenant la structure d'une mémoire

# Mémoire

- Je présente une addresse en entrée et la mémoire me donne son contenu en sortie




- En mettant “000001” en entrée, j’obtiens “10001000” en sortie

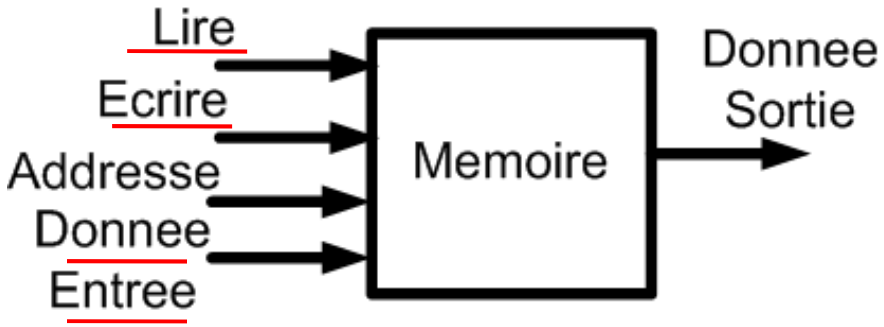
| Addresse | Donnees |
|---|---|
| 0 | 11010010 |
| 1 | 10001000 |
| 2 | 00101011 |
| 3 | 01111011 |
| ⋮ | ⋮ |
| 63 | 00001110 |

Dans cet exemple:

entrée est 6 bits d’addresse
sortie de 8 bits de données

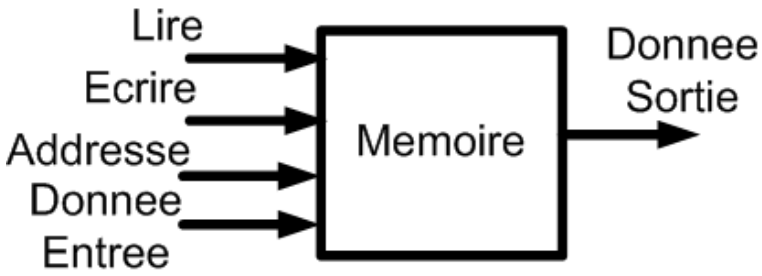
# Mémoire

- Une mémoire contient de l'information
  - Comment fait-on pour ajouter/modifier les données?
  - On aurait donc besoin **d'écrire**
- Pour écrire, j'ai besoin de 2 autres détails:
  - Je dois avoir un port pour entrer les données à écrire
  - Je dois avoir une manière d'indiquer si je veux lire ou écrire



# Mémoire

- Pour écrire, je dois
  - Indiquer l'addresse où je veux écrire
  - Indiquer la donnée à écrire
  - Mettre Ecrire = '1'



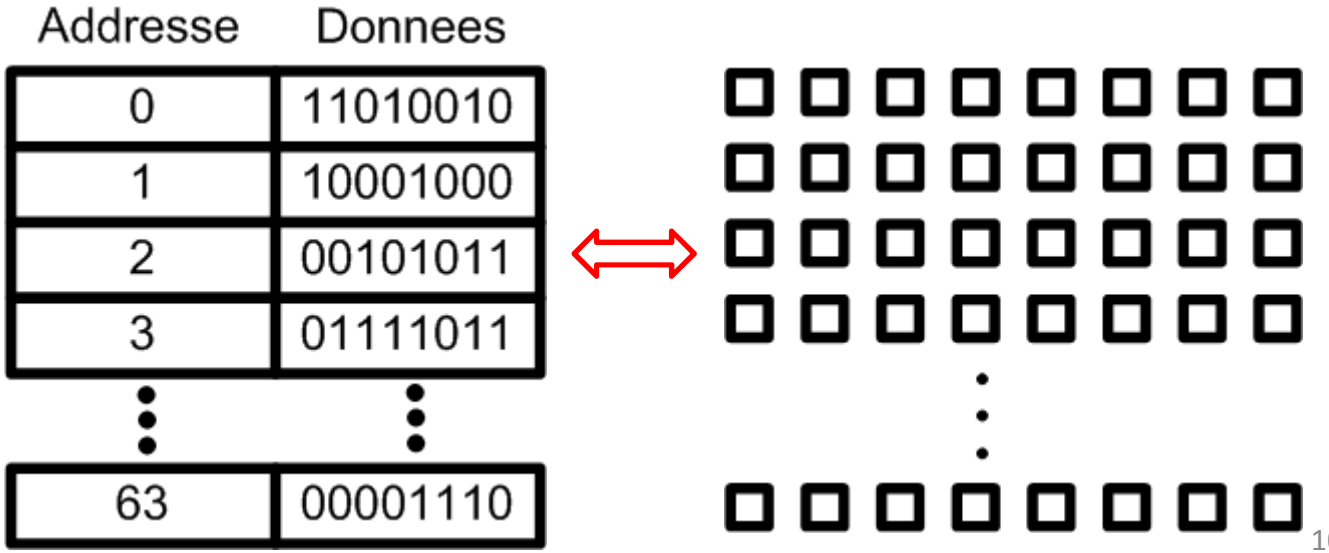

- Pour lire, je dois
  - Indiquer l'addresse où je veux lire
  - Mettre Lire='1'
  - La donnée en entrée peut être n'importe quelle valeur

En fait, Lire/Ecrire peut devenir un seul bit. 0: ecrire 1: lire, par exemple

# Mémoire

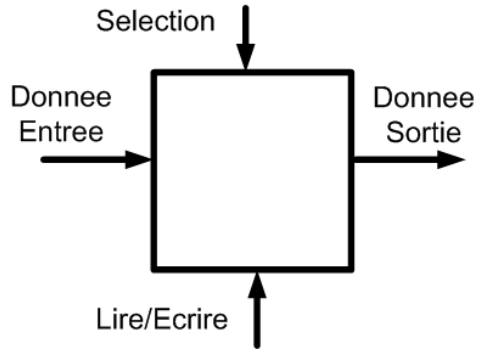
- On peut voir la mémoire comme une matrice
  - Chaque rangée représente une addresse
  - Et à chaque addresse, on retrouve un **MOT** d'un certain nombre de bits
  - Les mots sont typiquement 8 bits, 16 bits ou 32 bits

Dans cet exemple, j'ai une mémoire de 64 x 8bits



# Cellule mémoire

- Chaque bit de mémoire devrait ressembler à ceci

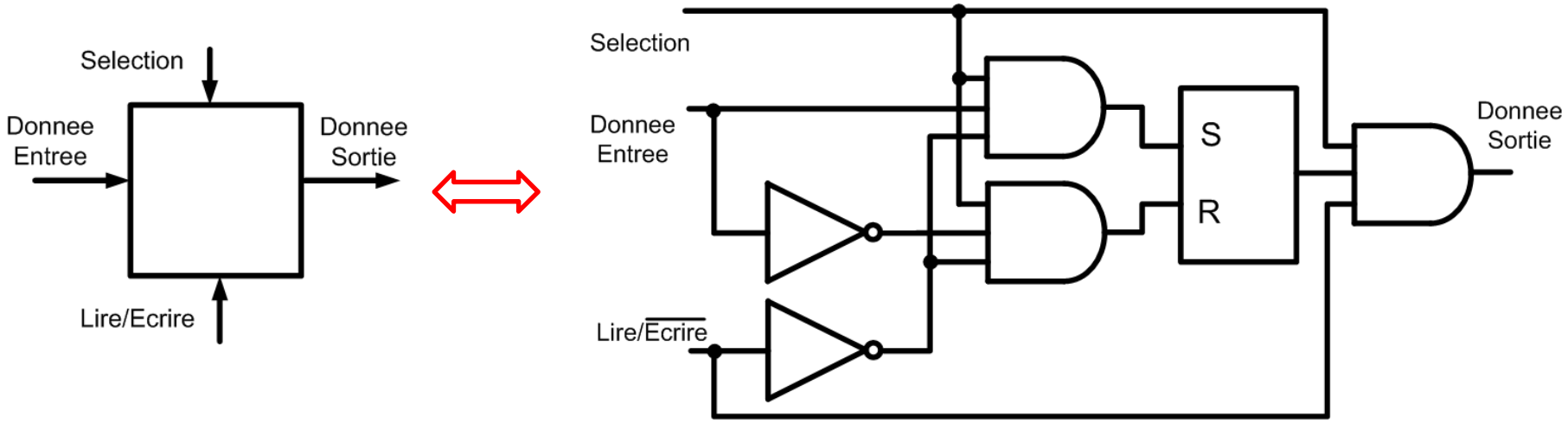



Cellule de 1 bit

- Il y a
  - Un port pour entrer la valeur
  - Un port pour sortir la valeur
  - Un port pour sélectionner que c'est à lui qu'on parle
  - Un port pour dire s'il faut lire ou écrire

# Cellule mémoire

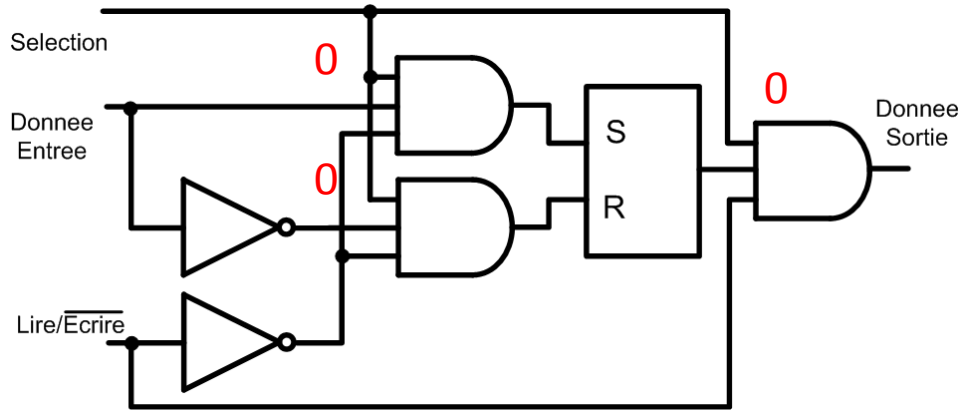
- De facon conceptuelle, on peut voir l'intérieur comme étant:




Allons voir comment ça fonctionne...

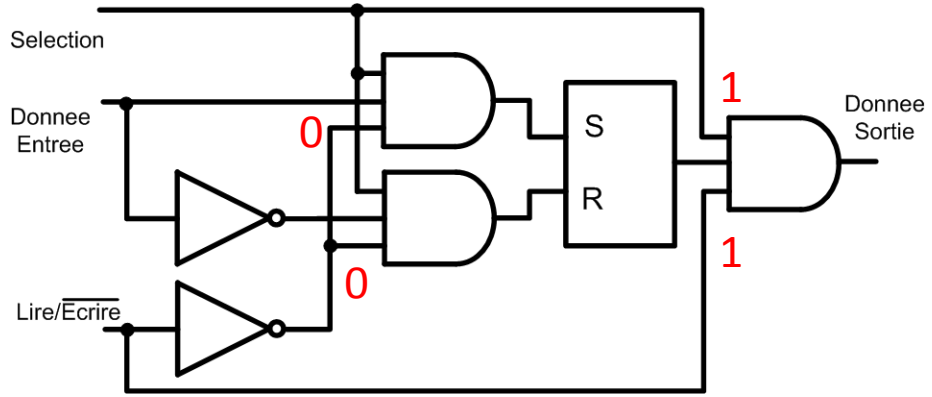
# Cellule mémoire

- Si on ne sélectionne pas (‘0’), la sortie est 0
  - Les entrées R et S sont 0: la bascule ne change pas
  - On ne change rien et la sortie est 0



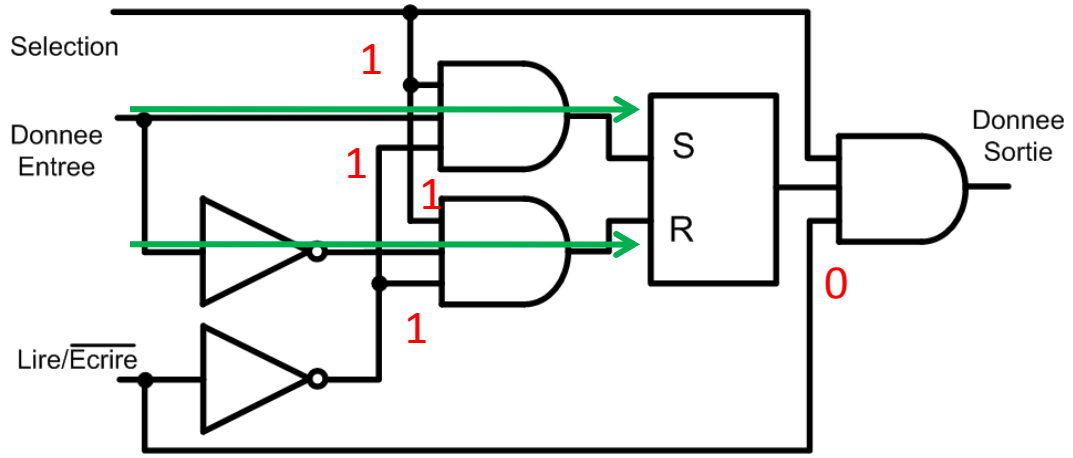
# Cellule mémoire

- Si on sélectionne (‘1’) pour lire (‘1’)
  - Les entrées R et S sont 0: la bascule ne change pas
  - La sortie sera égale au contenu de la bascule



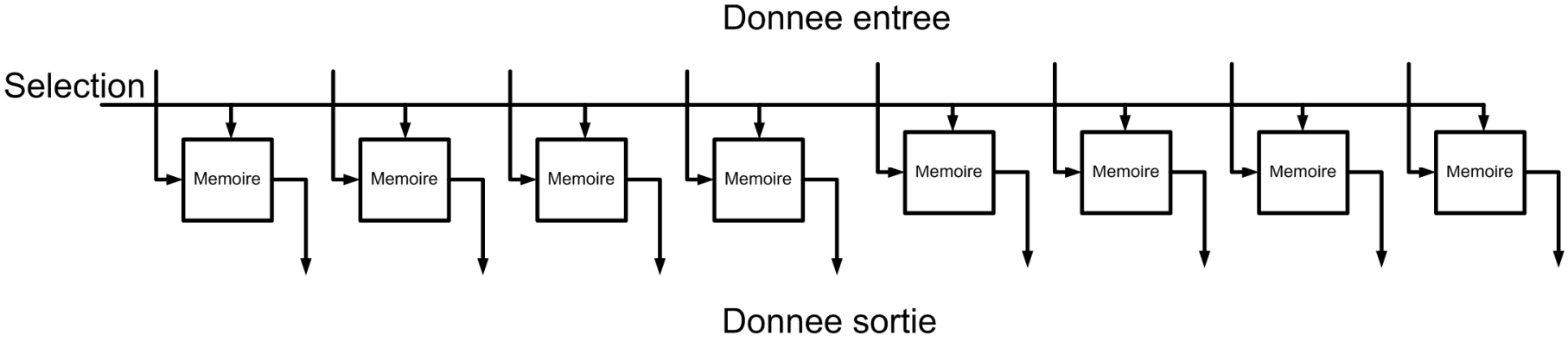
# Cellule mémoire

- Si on sélectionne (‘1’) pour écrire (‘0’):
  - Les entrées S et R dépendent de l’entrée
  - S sera ‘1’ si l’entrée est ‘1’
  - R sera ‘1’ si l’entrée est ‘0’
  - Notre bascule SR peut changer de valeur dans ce cas



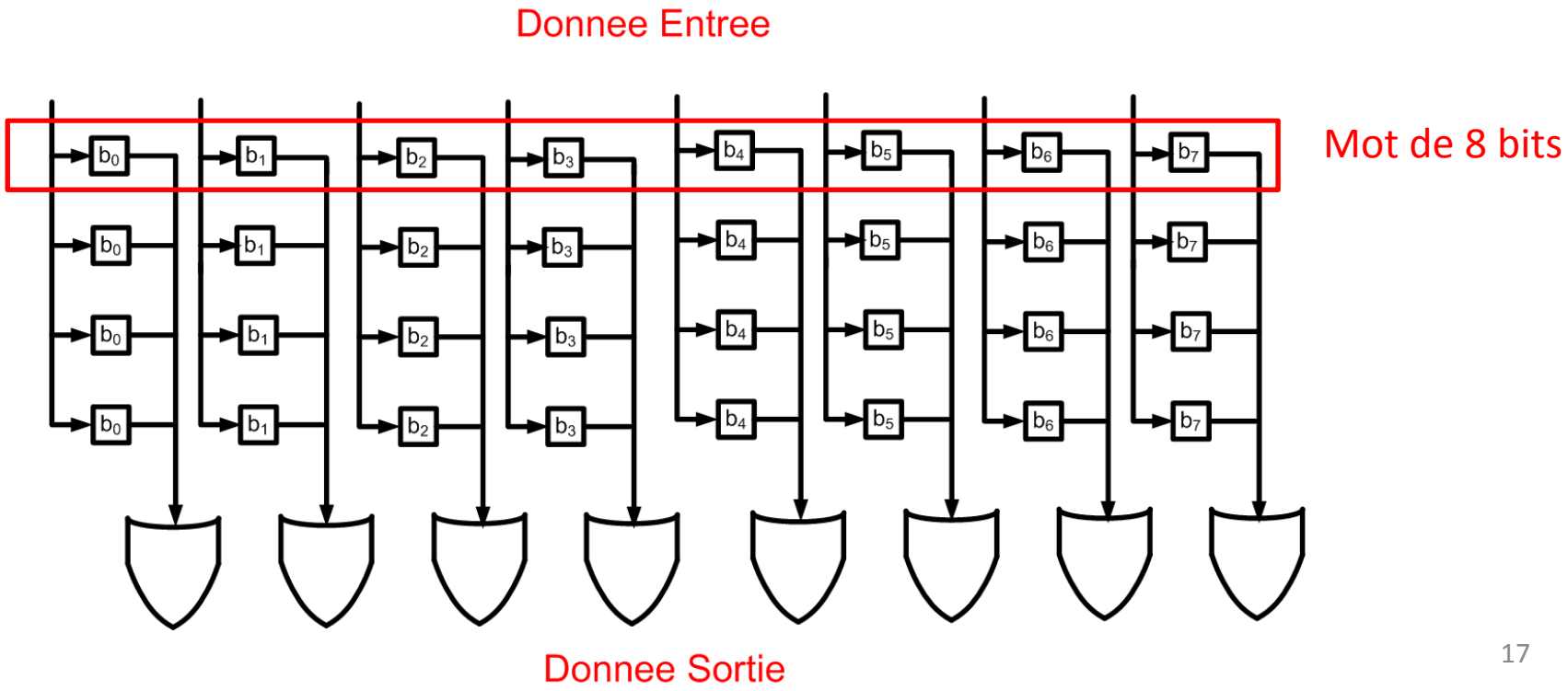
# Mot

- Chaque addresse contient un mot
- Dans un mot:
  - La ligne de sélection est la même pour tous les bits
  - Les entrées sont différents
  - Les sorties sont différents
  - Lire/Ecrire est partagé par tous (pas montré ici)



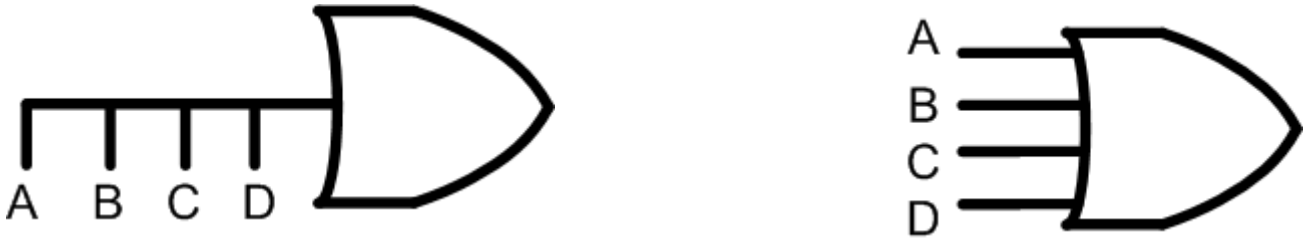
# Mémoire

- On connecte les mots pour former la matrice
  - Tous les bit0 partagent la même ligne de données
  - Toutes les sorties de bit0 entrent dans une porte OU



# Mémoire

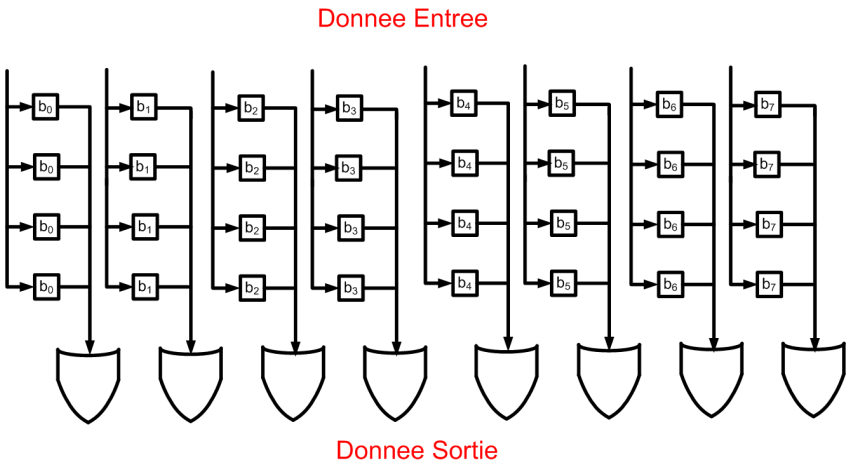
- Selon notre notation, les 2 diagrammes suivants sont équivalents:




- Il manque plusieurs détails dans le diagramme
  - On n'a pas dessiné les lignes de sélection
  - On n'a pas dessiné les signaux lire/écrire

# Mémoire

- Les données en entrée sont connectés à toutes les cellules en meme temps:
  - On utilise les lignes sélection pour distinguer
- Les sorties passent toutes par le même OU
  - Il faut forcer les autres signaux à 0
  - Il faut donc que leurs sélections tombent à 0

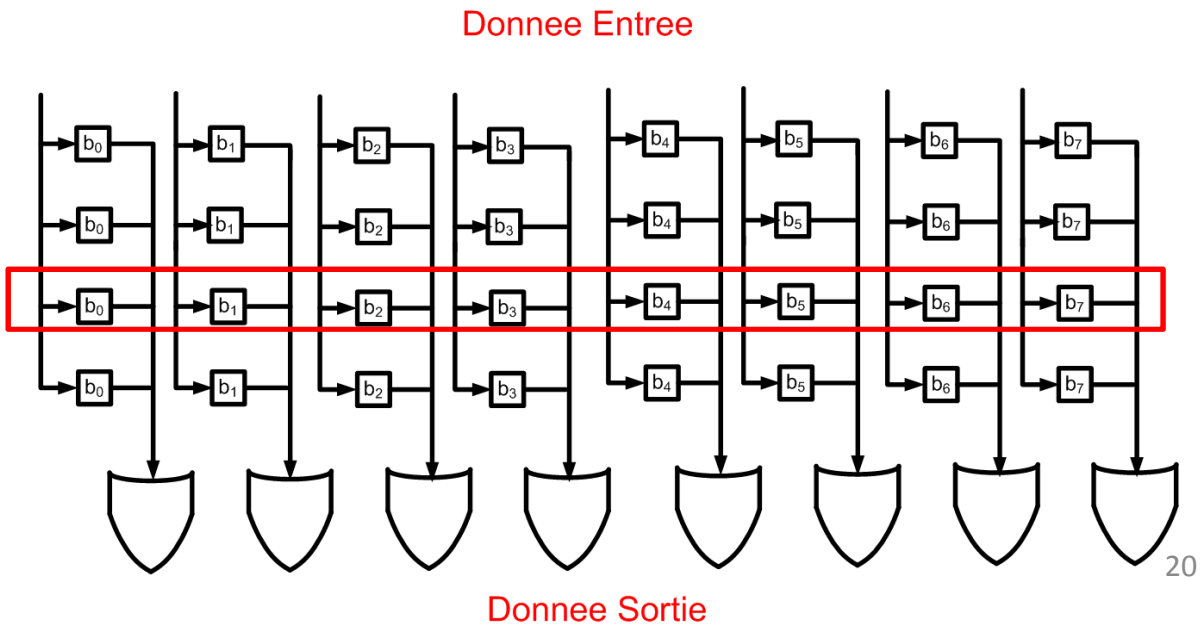


# Mémoire

- La ligne lire/ecrire est connectée à toutes les cellules
- La ligne de sélection est seulement active pour le mot choisi

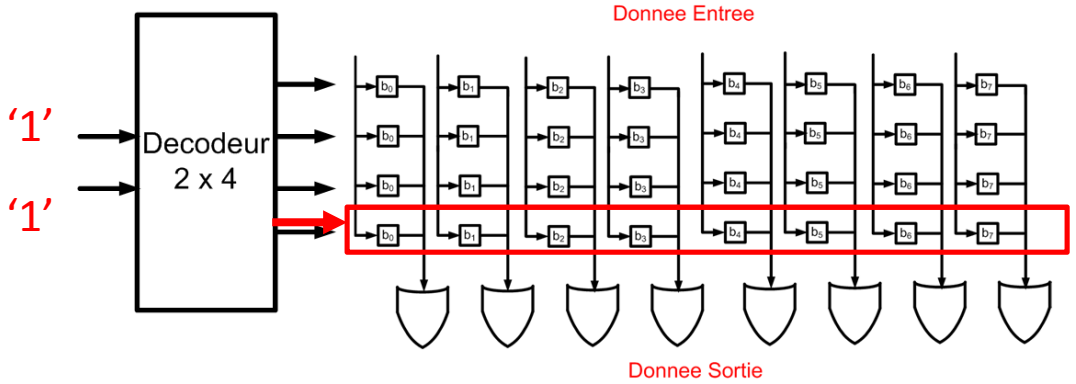
Si je voulais l'addresse 2, toutes ces cellules sont selectionnees...

Les autres ne le sont pas



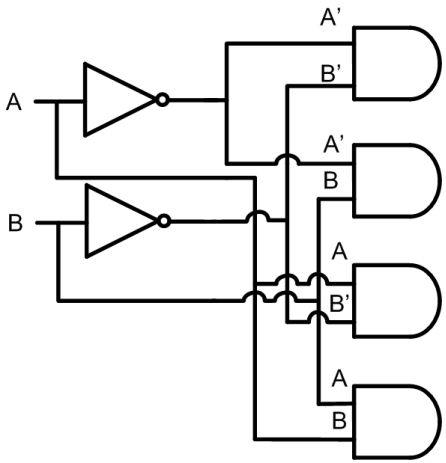
# Mémoire

- En entrant une addresse, j'aimerais que:
  - Le mot de cette addresse soit sélectionné ('1')
  - Tous les autres mots ne le soient pas ('0')
- C'est un encodage de type "one-hot"
  - Quel genre de circuit prend une addresse binaire et génère une sortie "one-hot"?
  - Un décodeur!



# Décodeur mémoire

- Examinons le décodeur en détail...
- On retourne en arrière et on se rend compte qu'un décodeur est composé de:
  - N portes **ET** et $LOG_2N$ entrées
  - Si on avait 4 addresses, on aurait besoin de 4 portes **ET** et de 2 entrées

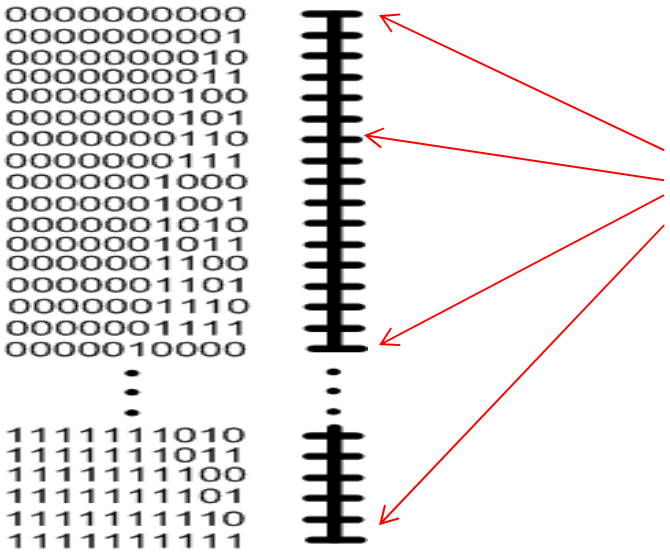


# Décodeur mémoire

- Augmentons la taille...
- Pour 1024 addresses, on aurait besoin de 1024 portes ET et 10 entrées
    - Pour les mémoires de 1MB, on aurait besoin d'un million portes ET et 20 entrées
    - Ce n'est que pour le décodage d'addresse.
    - La mémoire prend aussi de l'espace
- Sommes-nous capables de réduire cette taille?

# Décodeur mémoire

- Construisons une mémoire de 1K addresses
  - La facon “naive” c’est d’utiliser une approche linéaire
  - On énumère toutes les $2^{10}$ possibilités et chaque barre correspond à une donnée de 8 bits (par exemple)

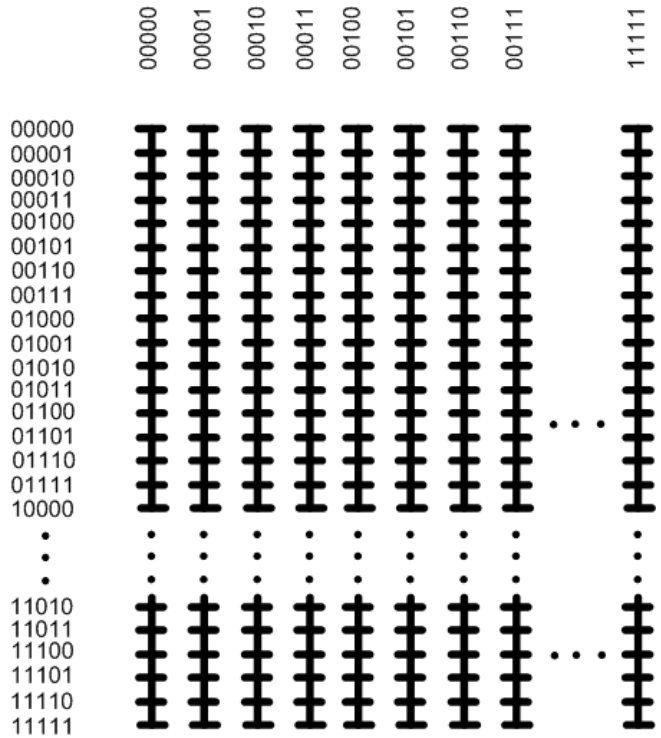



Il y a 1024 barres

On a 10 bits d’entrée et le décodeur a 1024 portes ET

# Décodeur mémoire

- À la place de 1 colonne de 1024, on peut avoir une matrice
  - Dans ce cas-ci, une matrice de 32x32 (=1024)



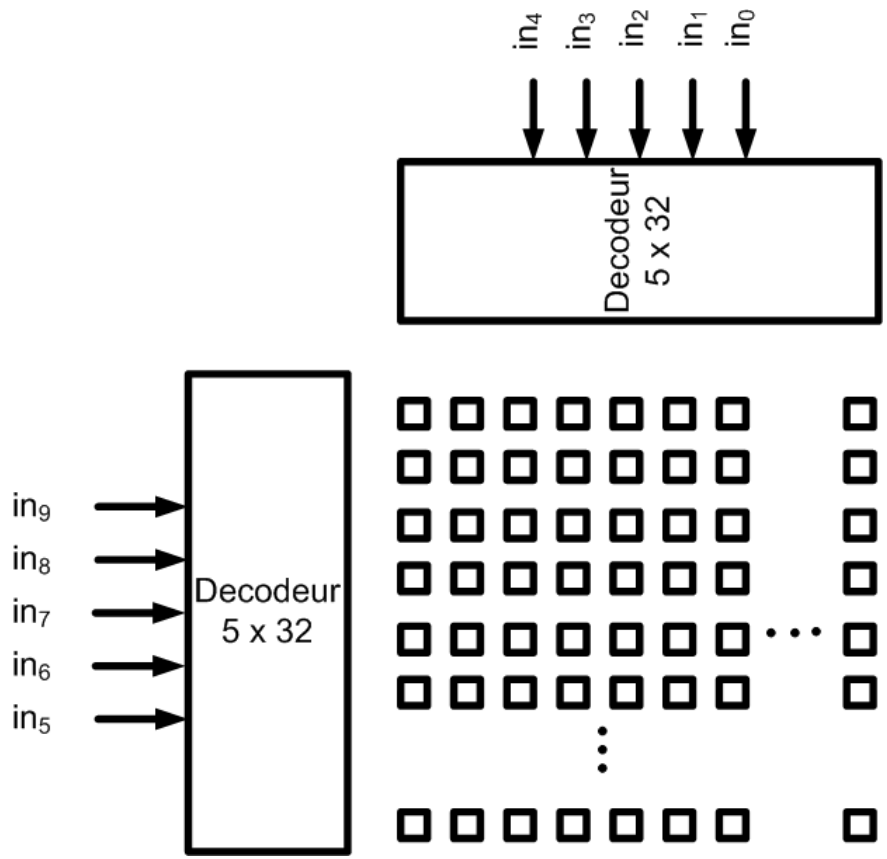

Chaque barre correspond à 8 bits de données

Il y a la même quantité de bits que l'autre (1024)

On remplace le décodeur 10 x 1024 par 2 décodeurs de 5-a-32.. donc, 64 portes ET

# Décodeur mémoire

- On se retrouve maintenant avec 2 décodeurs:
  - Une pour les rangées et une pour les colonnes




Chaque carré représente 8 bits

Ça remplace les barres des diagrammes précédents

# Décodeur mémoire

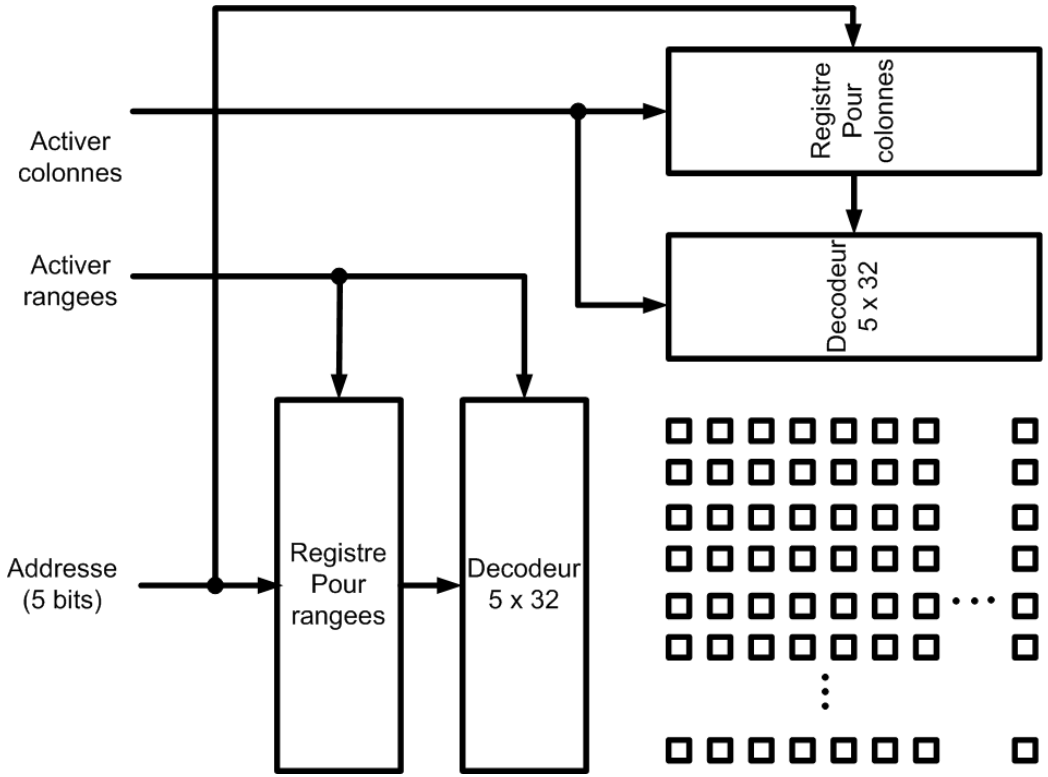
- On est passé de 1024 portes ET a 64 portes ET en utilisant une approche 2D:
  - À la place d'avoir 1 colonne et 1024 rangées, on a 32 colonnes et 32 rangées.
- Il est possible d'avoir d'autres configurations:
  - 16 colonnes et 64 rangées
  - 8 colonnes et 128 rangées
  - …

# Décodeur mémoire

- Mathématiquement, le plus efficace c'est quand c'est carré:
  - Dans notre cas, 32x32 (2 décodeurs de 32 portes ET)
  - Essayez vous-mêmes de faire mieux: pas possible (ex: 16x64 donne 80 portes ET)
- Rangées et de colonnes doit etre $2^N$
  - Ceci fera en sorte que nos décodeurs seront utilisés à leurs capacites maximales
  - Sinon, pas optimal

# Structure de mémoire

- Voici une implantation possible:
  - On économise les broches en spécifiant 5 bits d'addresse à la fois...



# Exemple concret

- Le HM5117800 est une mémoire de Hitachi
  - Il a une capacité de 2M ($2^{21}$) addresses de 8 bits chaque
- Combien de portes ET a-t-on besoin pour les décodeurs
  - S'il utilisait la structure à 1 colonne?
  - S'il utilisait la structure par matrice de facon optimale?
- Combien a-t-il de bits d'addresse dans chaque cas?

# Exemple concret

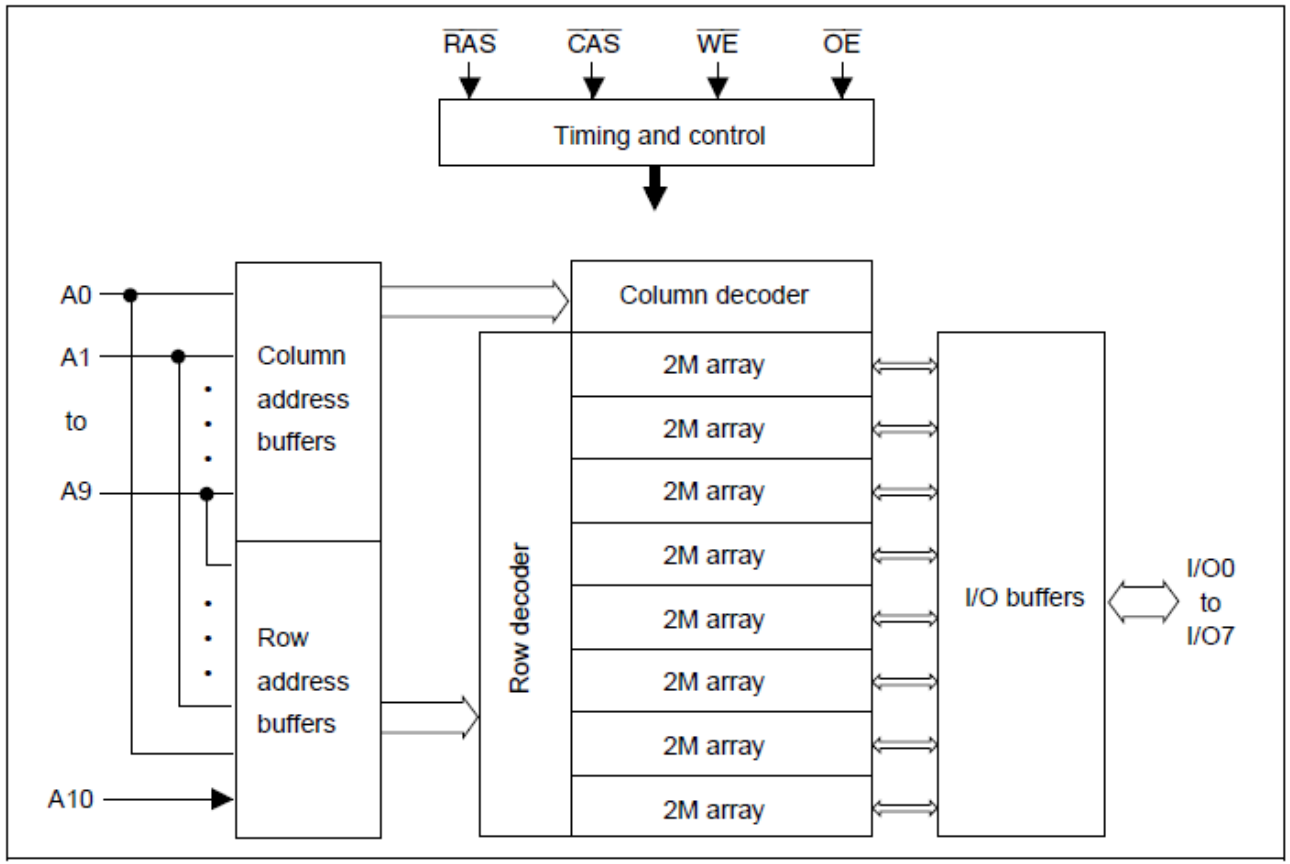
- 1M est égal à $2^{20}$
- Avec 2M ($2^{21}$), on pourrait avoir une implantation à 1 colonne:
  - On aurait 1 colonne de $2^{21}$ addresses
  - On aurait donc $2^{21}$ portes ET
  - Pour l'addresse, on a $LOG_2(2^{21})$=21 bits
- On aurait donc 21 bits d'addresse

# Exemple concret

- Le mieux c'est d'avoir une approche 2D:
  - On veut être le plus carré possible
  - Avec 21 bits, ce n'est pas possible
- Le mieux c'est un côté de 10bits et un côté de 11bits
  - On aurait $2^{11}+2^{10}$ portes ET (3072)
  - C'est une réduction importante quand on compare à $2^{21}$ (2097152)

# Exemple concret

- La structure HM5117800 utilise l'approche optimale



# Code VHDL pour la memoire

- Il faut definir un TYPE pour la matrice:

```
TYPE matrix IS ARRAY(0 to 15) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
```

- On définit ensuite une variable de ce type:

```
SIGNAL memoire : matrix;
```

- La variable mémoire est: 16 x 8 bits

# Code VHDL pour la mémoire

- Pour accéder à la mémoire on utilise des entiers: integer (pas std_logic_vector)
  - Exemple: dataout <= memoire(3);
- Comment utiliser l'addresse?
  - La commande CONV_INTEGER
  - dataout <= memoire(CONV_INTEGER(address));

# Code VHDL pour la mémoire

- L'intérieur du process **sequentiel** ressemble à:

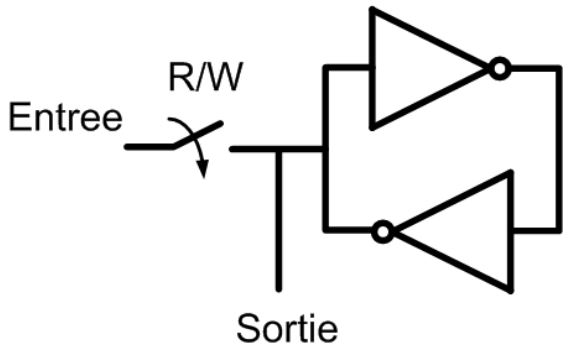
```
IF clk'EVENT AND clk = '1' THEN
  IF rwn = '1' THEN
      dataout <= memoire(CONV_INTEGER(address));
  ELSE
      memoire(CONV_INTEGER(address)) <= datain;
  END IF;
END IF;
```

# Mémoire volatile

- Il existe 2 grandes classes de mémoire volatile:
  - SRAM: Static RAM
  - DRAM: Dynamic RAM
- RAM: Random access memory
  - Les vieilles technologies sont séquentielles
- Ces 2 types de mémoire perdent leurs valeurs quand on éteint le système
  - La différence est dans leurs structures et dans le comportement

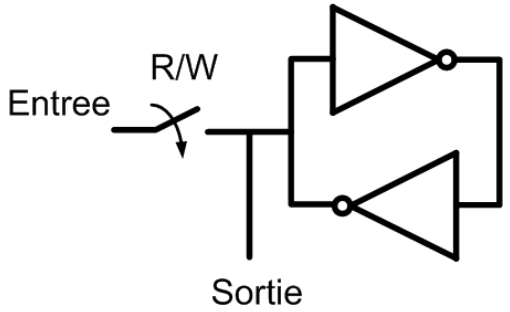
# SRAM

- Une SRAM est l'équivalent d'une bascule
  - Il y a un signal qui permet l'écriture
  - Il y a un élément mémoire qui fonctionne en rétroaction positive (feedback positif)
- La structure classique ressemble à ceci:



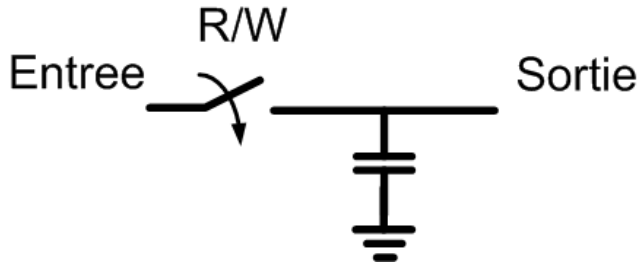
# SRAM

- La combinaison d'inverseurs conserve la valeur
  - Le commutateur permet l'écriture d'une valeur dans la mémoire
  - La vitesse d'opération est relativement rapide
  - La valeur est conservée tant que l'alimentation est gardée
  - Problème: relativement gros (6 transistors)



# DRAM

- La DRAM est beaucoup plus petite:
  - Elle ne demande qu'un commutateur et 1 capacité
  - La capacite peut etre un transistor et le commutateur est souvent reuit a 1 transistor
  - Donc c'est 3 fois plus efficace que la SRAM
  - Pour des millards de bits, ca vaut la peine



# DRAM

- L'avantage est donc la taille
- Les problèmes sont:
  - La valeur dans la capacité est perdue avec le temps (courant de fuite). Il faut donc la rafraîchir.
  - Plus lent: il y a peu de charges stockée et il faut des amplificateurs pour lire la valeur
  - La lecture est destructrice: la petite capacité de stockage est plus petite que les lignes de lecture

# Mémoire non-volatile

- La mémoire non-volatile a la caractéristique de ne pas s'effacer quand on l'éteint:
  - Idéalement, il devrait garder sa valeur éternellement
- Il existe 4 grandes classes de mémoires non-volatiles:
  - ROM: Read-only memory
  - PROM: Programmable ROM
  - EPROM: Erasable PROM
  - EEPROM: Electrically Erasable PROM

# Mémoire non-volatile

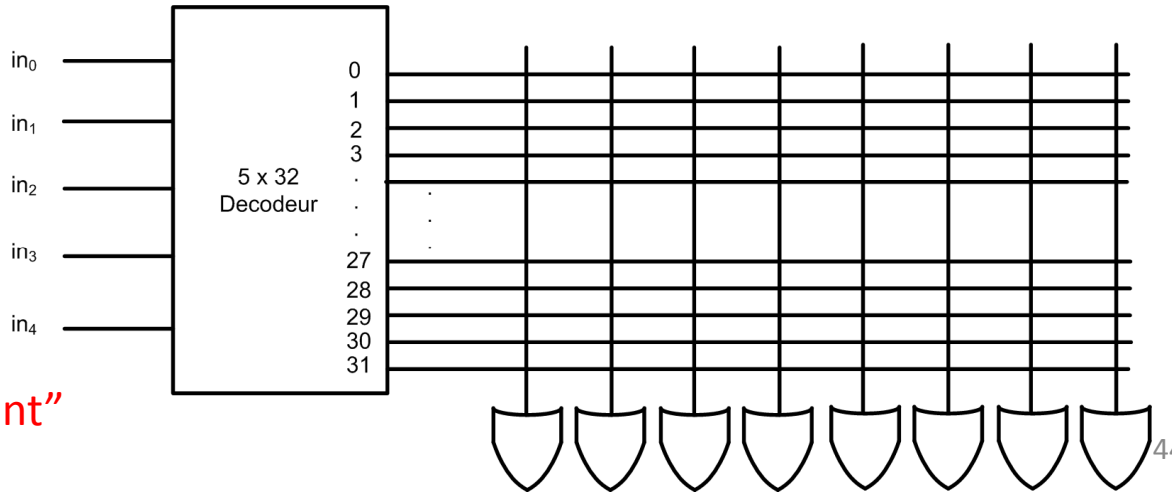
- **ROM**: Mémoire programmée à l'usine
- **PROM**: Mémoire programmée par l'usager avec un outil qui brûle des fusibles du PROM
- **EPROM**: Mémoire programmée par l'usager et qui est effaçable avec la lumière ultra-violet
- **EEPROM**: Mémoire programmée par l'usager et qui est effacable avec des tensions élevées

# Mémoire non-volatile

- Les mémoires non-volatiles ont souvent cette structure classique:
  - Comme il est là, il n'y a pas de connexions entre le décodeur et les portes OU
  - La sortie sera toujours 0
  - En le programmant, ça fait les connexions

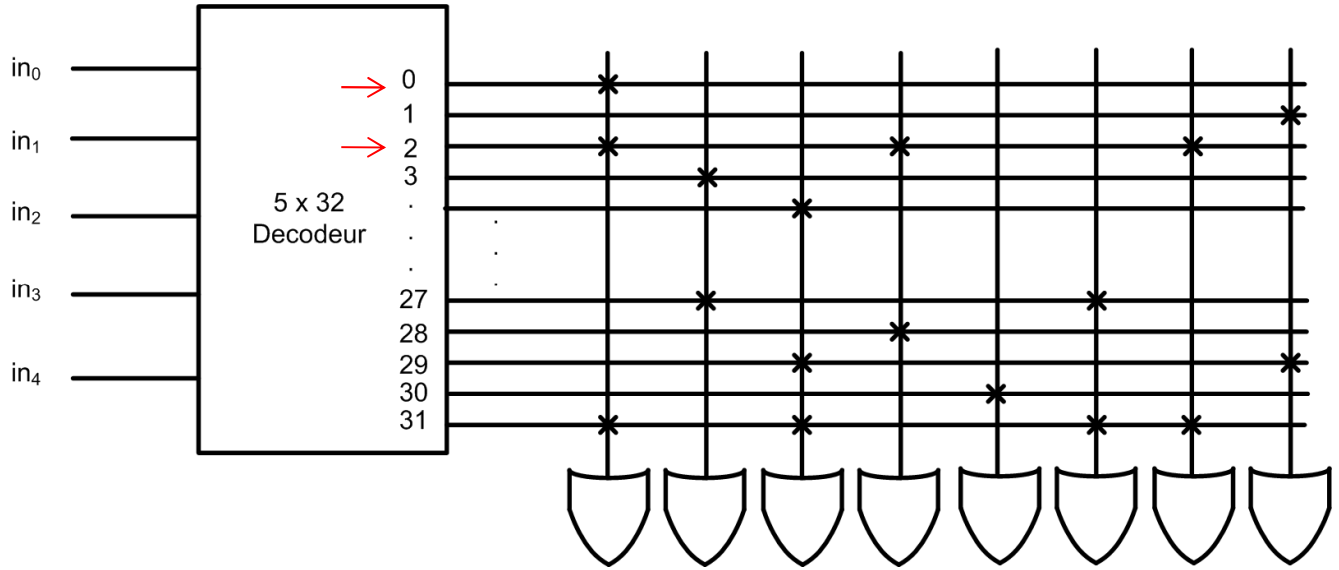
En réalite, c'est le contraire qui se produit

"les connexions sont faits et on les brise en programmant"



# Mémoire non-volatile

- On représente la programmation avec un X
  - Ça indique le la sortie du décodeur est connecté à l'entrée des portes OU
  - Si l'entrée etait "00000", la sortie devrait être "10000000"
  - Avec "00010", on aurait "10010010"



# Mémoire non-volatile

- Les mémoires non-volatiles peuvent servir à stocker des données:
  - Ex: Musique enregistrée. Chaque addresse correspond à la valeur à une µs donnée
  - Si on lisait une donnée à chaque µs et on l'envoyait à un speaker, on entendrait de la musique.

    (il faut que ce soit converti en analogique avant)
- Ça peut aussi servir à implanter des fonctions

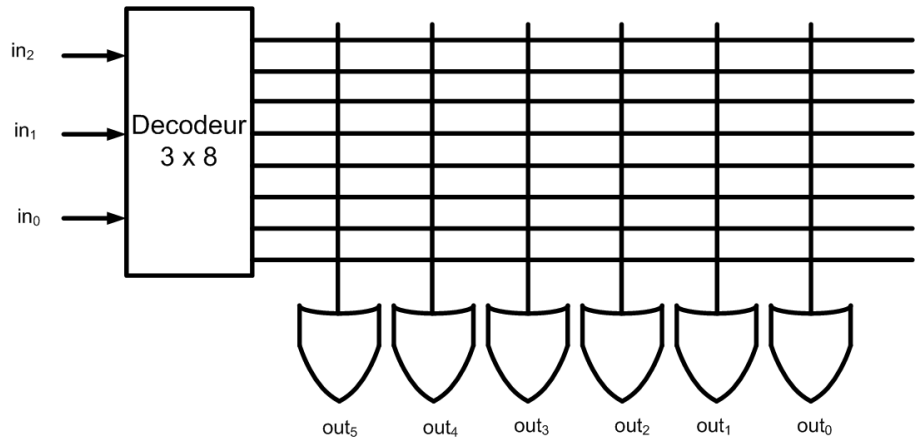
# Mémoire non-volatile

- Retournez voir les décodeurs:
  - En connectant les bonnes sorties a des portes OU, on peut implanter n'importe quelle fonction
- Avec les mémoires, on peut faire la même chose:
  - En programmant la mémoire, on fait les connexions voulues entre le décodeur et les portes OU
  - On peut donc implanter n'importe quelle fonction
  - On appelle ce concept "Look up table" (LUT)

# Exemple

- À l'aide d'une ROM 8x6bits, faites un circuit qui prend 3 bits et qui le met au carré
- But: Remplir le tableau et mettre les connexions dans le diagramme:

| Addresse (entree) | Donnee (sortie) |
|---|---|
| 000 | |
| 001 | |
| 010 | |
| 011 | |
| 100 | |
| 101 | |
| 110 | |
| 111 | |



# Exemple

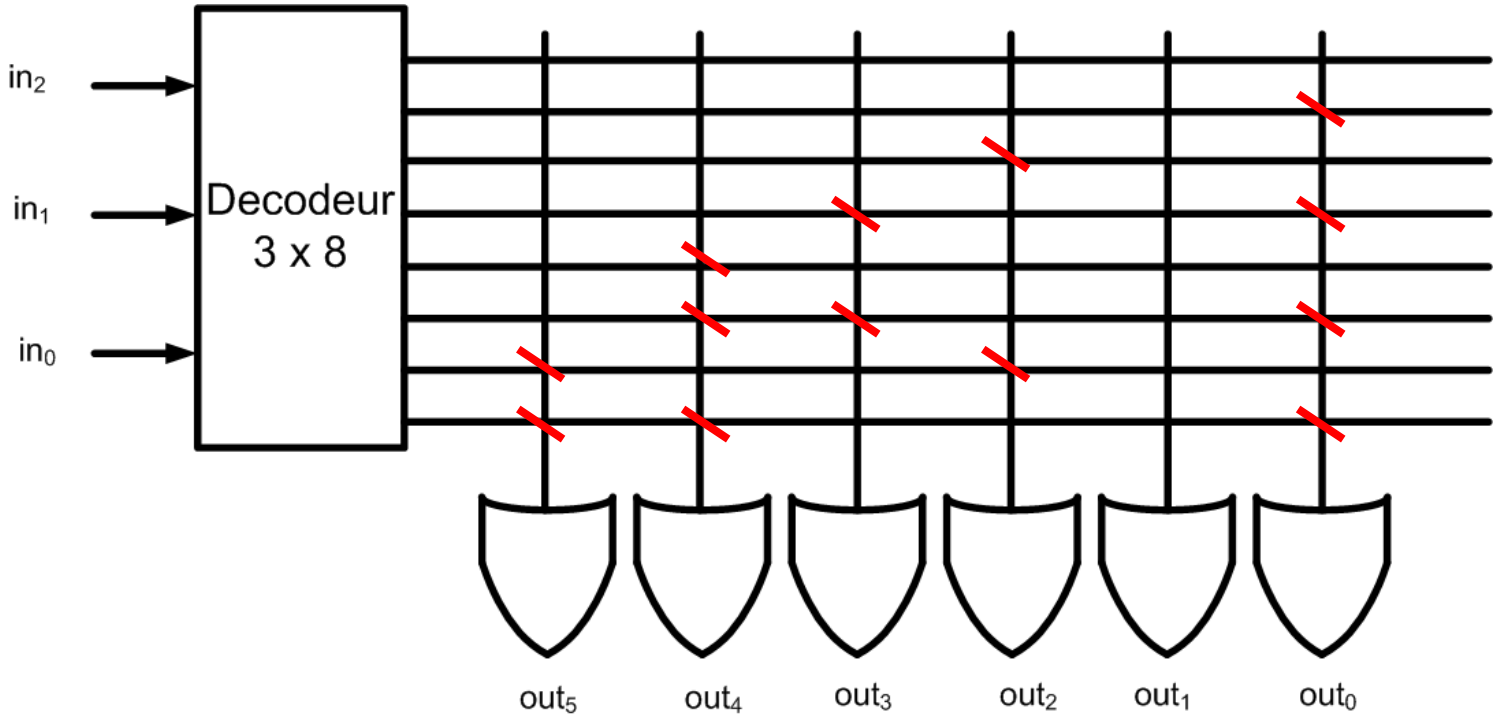
- Le tableau devrait ressembler à ceci:

| Addresse | Donnee |
|---|---|
| 000 | 000000 |
| 001 | 000001 |
| 010 | 000100 |
| 011 | 001001 |
| 100 | 010000 |
| 101 | 011001 |
| 110 | 100100 |
| 111 | 110001 |

# Exemple

- Le circuit ressemblerait à ceci:

| Addr | Donnee |
|---|---|
| 000 | 000000 |
| 001 | 000001 |
| 010 | 000100 |
| 011 | 001001 |
| 100 | 010000 |
| 101 | 011001 |
| 110 | 100100 |
| 111 | 110001 |



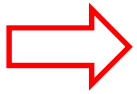
# Exemple (seul)

- À l'aide d'une ROM 8x2bits, faites un circuit qui prend 3 bits et qui le divise par 3
  - Le circuit arrondit TOUJOURS au nombre entier le plus élevé
  - Par exemple: 7/3=3

# Exemple (seul)

- On fait le tableau qui montre les entrées et sorties
  - On sait maintenant quelles valeurs mettre en mémoire

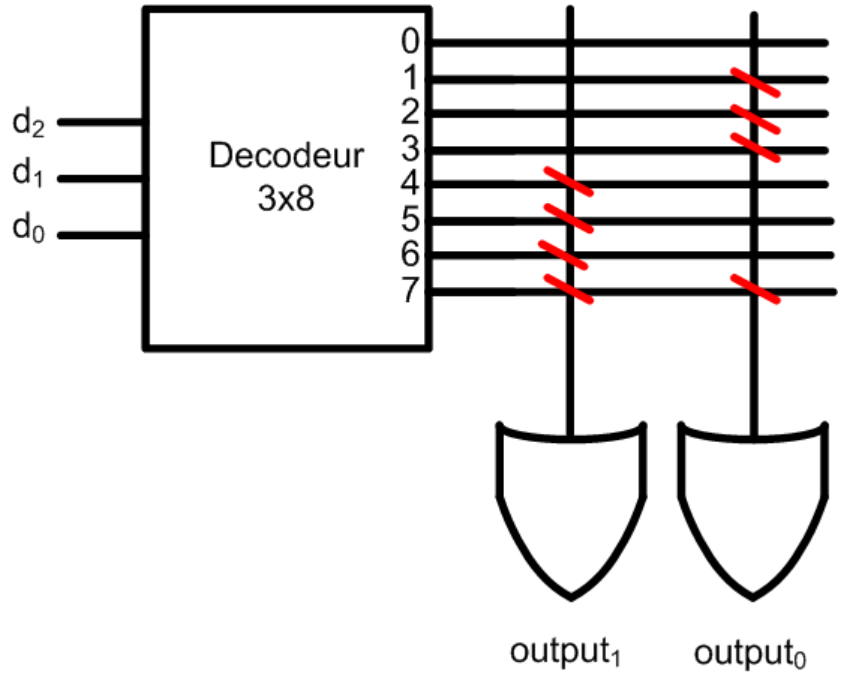
| Entree | Sortie |
|---|---|
| 000 | 00 |
| 001 | 01 |
| 010 | 01 |
| 011 | 01 |
| 100 | 10 |
| 101 | 10 |
| 110 | 10 |
| 111 | 11 |



| Addresse | Donnees |
|---|---|
| 000 | 00 |
| 001 | 01 |
| 010 | 01 |
| 011 | 01 |
| 100 | 10 |
| 101 | 10 |
| 110 | 10 |
| 111 | 11 |

# Exemple (seul)

- Une ROM peut être vue en 3 parties:
  - Le décodeur en entrée
  - La grille de mémoire
  - Les OU en sortie

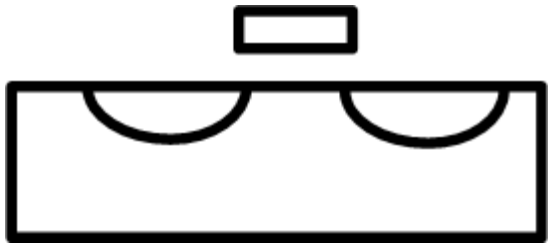


# Mémoire non-volatile

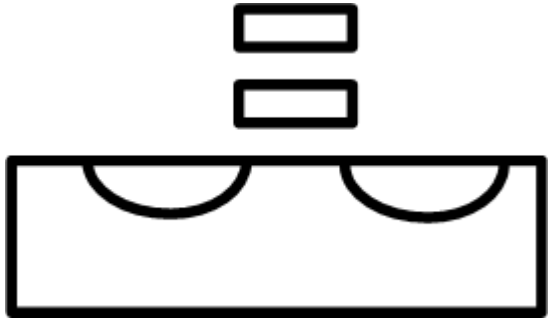
- PROM: Utilise une technologie de fusibles pour programmer
  - Pensez à un fil mince sur puce où on envoie un gros courant
  - Le **gros** courant chauffe le fil et le brise
- Une fois programme, il n'est pas possible d'effacer:
  - Le fil est brisé

# Mémoire non-volatile

- Les EPROMs et les EEPROMs ont une structure semblable
  - Ils sont faits à peu près comme des transistors CMOS
  - En plus, ils ont une grille flottante pour stocker les charges



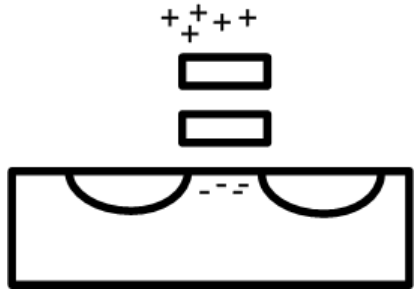
Transistor CMOS



EPROM/EEPROM

# Mémoire non-volatile

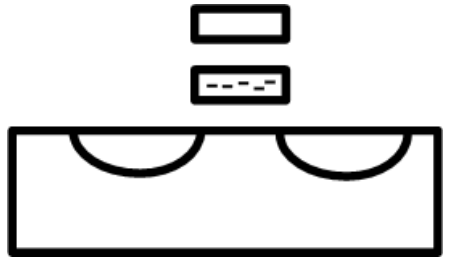
- En appliquant une tension à la grille du haut (grille de contrôle), on peut former un canal



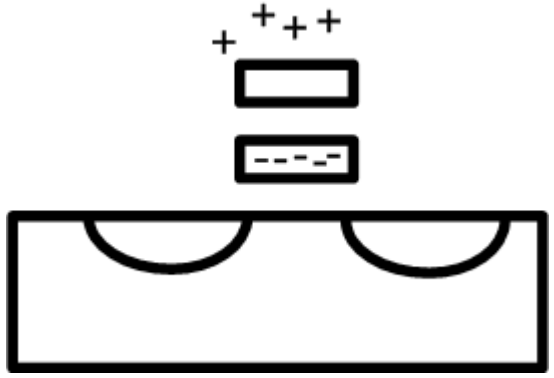
- Le courant pourrait donc circuler

# Mémoire non-volatile

- Si on décidait de mettre une TRÈS grande tension, les électrons passeraient l'isolant

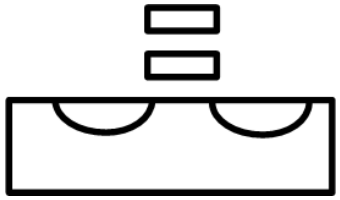


- Maintenant, même en appliquant une tension, on n'aurait pas de canal

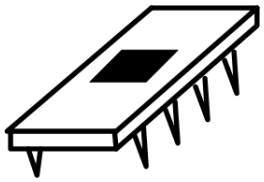


# Mémoire non-volatile

- Alors, chaque point de connexion dans la matrice d'éléments mémoire sera ceci:

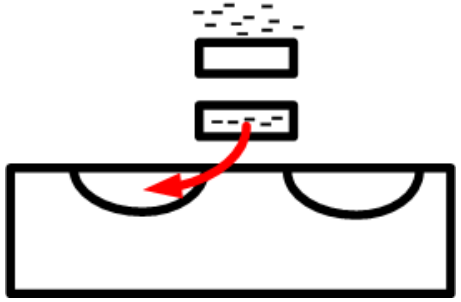


- Pour effacer la mémoire, on utilise de la lumière ultra violet (EPROM):
  - Ça fournit assez d'énergie pour permettre aux électrons de se libérer
  - Pour ça, il faut que la puce ait une "fenêtre"



# Mémoire non-volatile

- La technologie EEPROM fait en sorte qu'il est possible d'effacer électriquement:
  - Les matériaux sont différents et permettent des tensions positives et négatives élevées



  - NOTE: Les mémoires FLASH sont des EEPROM avec une structure pour effacer des blocs complets d'un coup

# PLD

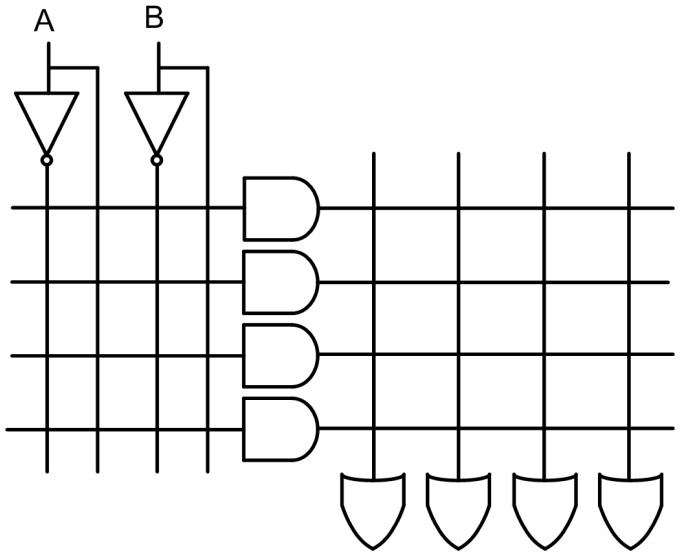
- On peut utiliser les ROMs pour implanter des fonctions:
  - On décide quelles connexions sont brûlées et quelles connexions doivent rester intactes
  - Ça donne un circuit combinatoire
- Il existe aussi d'autres produits qui sont utilisés pour faire le même travail
  - Ces produits font tous parti de la famille des Programmable Logic Devices (PLD)

# PLD

- Dans la famille des PLD, il existe 3 types:
  - ROM: on les connait deja
  - PAL: Programmable Array Logic
  - PLA: Programmable Logic Array
- Meme si PAL et PLA sont les mêmes mots, ils représentent des structures différentes
  - On va voir les différences à la prochaine diapo
  - La similitude est qu'ils sont tous composés d'une section ET suivi d'une section OU

# PLD

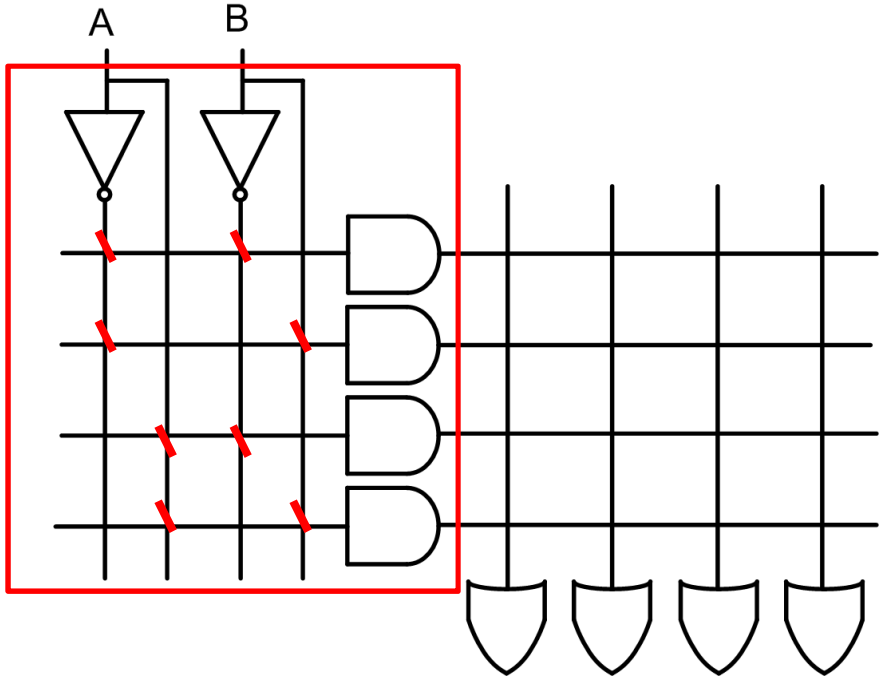
- L'architecture générale d'un PLD ressemble à ceci:
  - Les entrées peuvent être connectées aux portes ET
  - Les sorties des portes ET peuvent être connectées aux portes OU




Les connexions sont mises en mémoire et contrôlent un commutateur

# ROM

- Dans une ROM, la partie avec les ET n'est pas modifiable:
  - Toutes les connexions sont présentes
  - Cette structure fait le circuit d'un décodeur



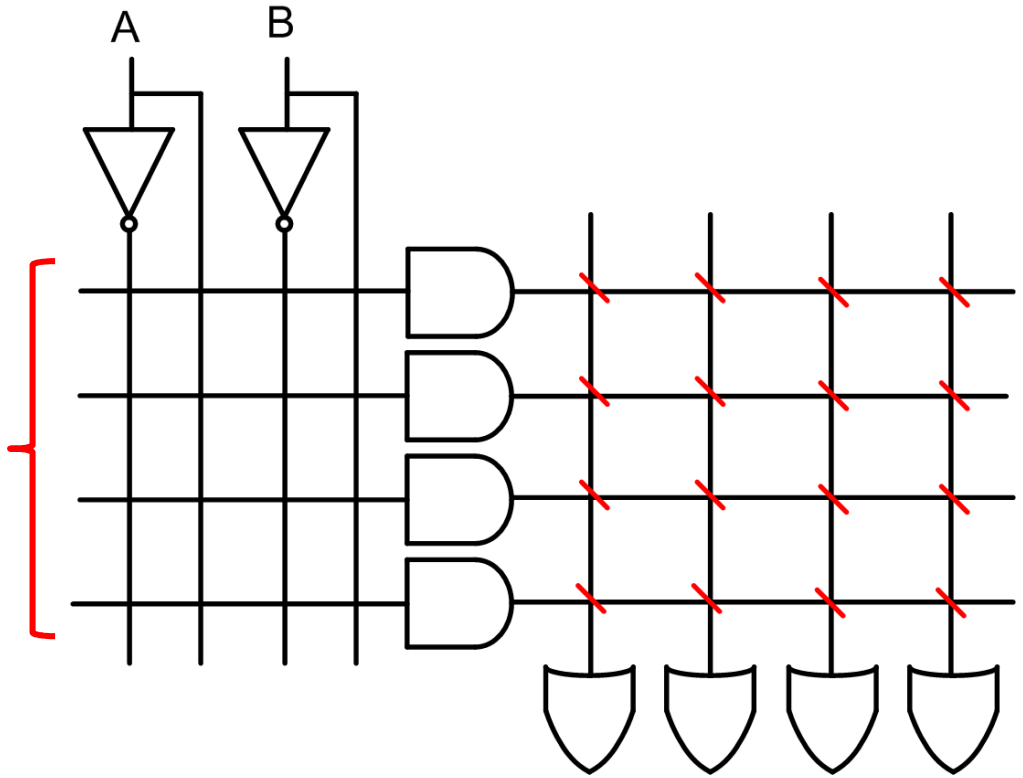

Décodeur 2 x 4

On configure cette partie seulement

# PAL

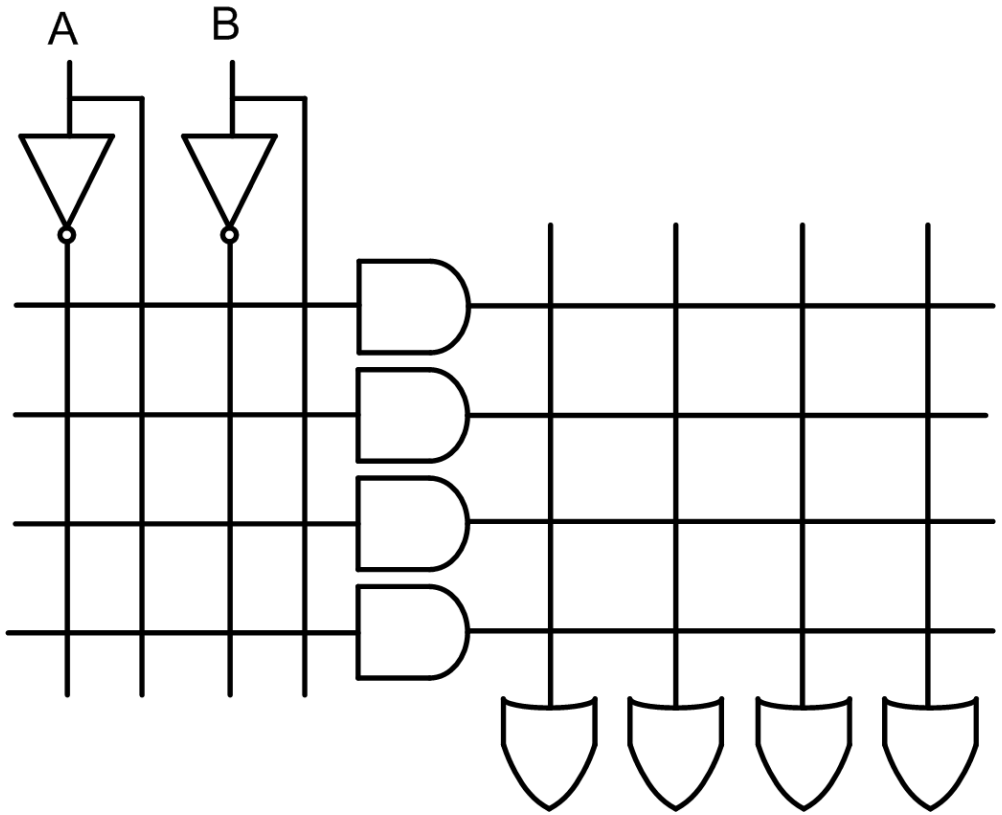
- Dans une PAL, la partie avec les OU n'est pas modifiable:

On configure cette partie seulement



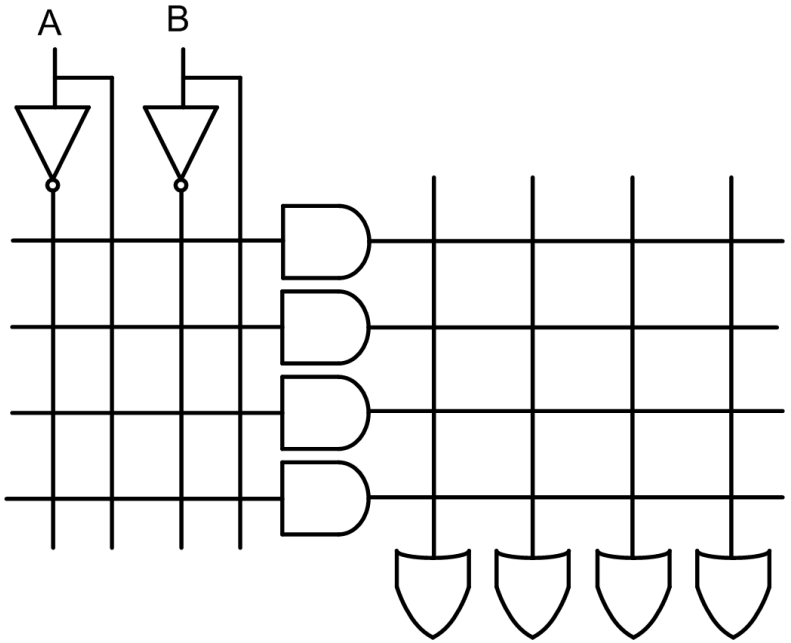
# PLA

- Dans une PLA, les 2 parties (ET et OU) sont programmables



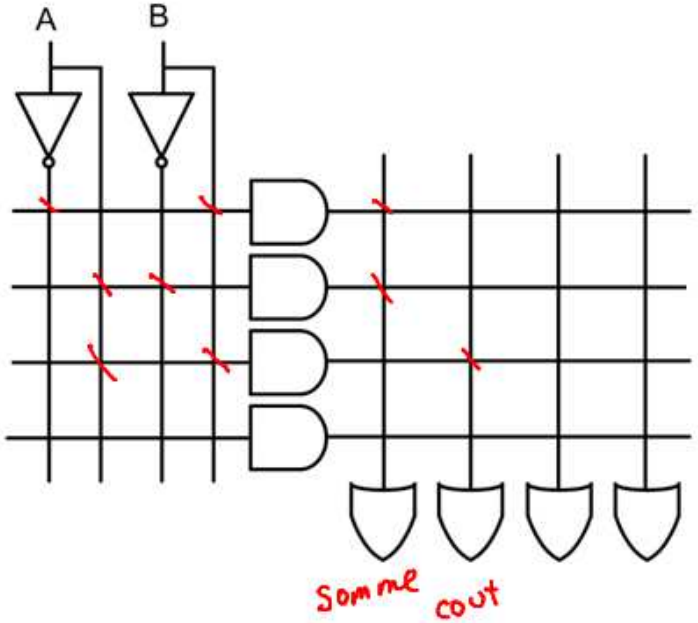
# Exemple (seul)

- Configurez le PLA pour faire un demi-additionneur de 1 bit



# Exemple (seul)

- La somme est 1 quand AB=01 ou AB=10
  - On le montre avec les 2 premieres lignes
- Le $C_{OUT}$ est 1 quand AB=11
  - On le montre sur la 3e ligne



# PLD

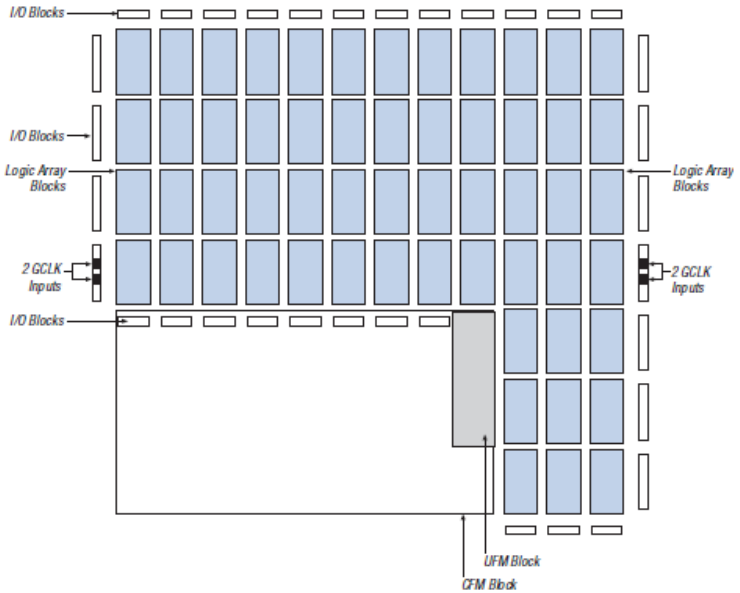
- Les PLDs sont normalement beaucoup plus gros que ce qu'on vient de voir
  - Ils ont plus d'entrées
  - Ils ont plus de sorties
  - Ils ont aussi des flip flops pour permettre de faire des circuits séquentiels simples
- Certains appellent ca des SPLD (des PLDs simples)

# CPLD

- Parfois, on va vouloir faire des systèmes plus complexes
  - On va utiliser des puces qui combinent plusieurs PLDs ensembles
  - On appelle ca des CPLD (PLD complexes)
- Quand on parle de CPLD, on se rapproche de plus en plus des FPGAs
  - La distinction entre CPLD et FPGA est un peu floue
  - Il n'est pas facile de les distinguer

# CPLD

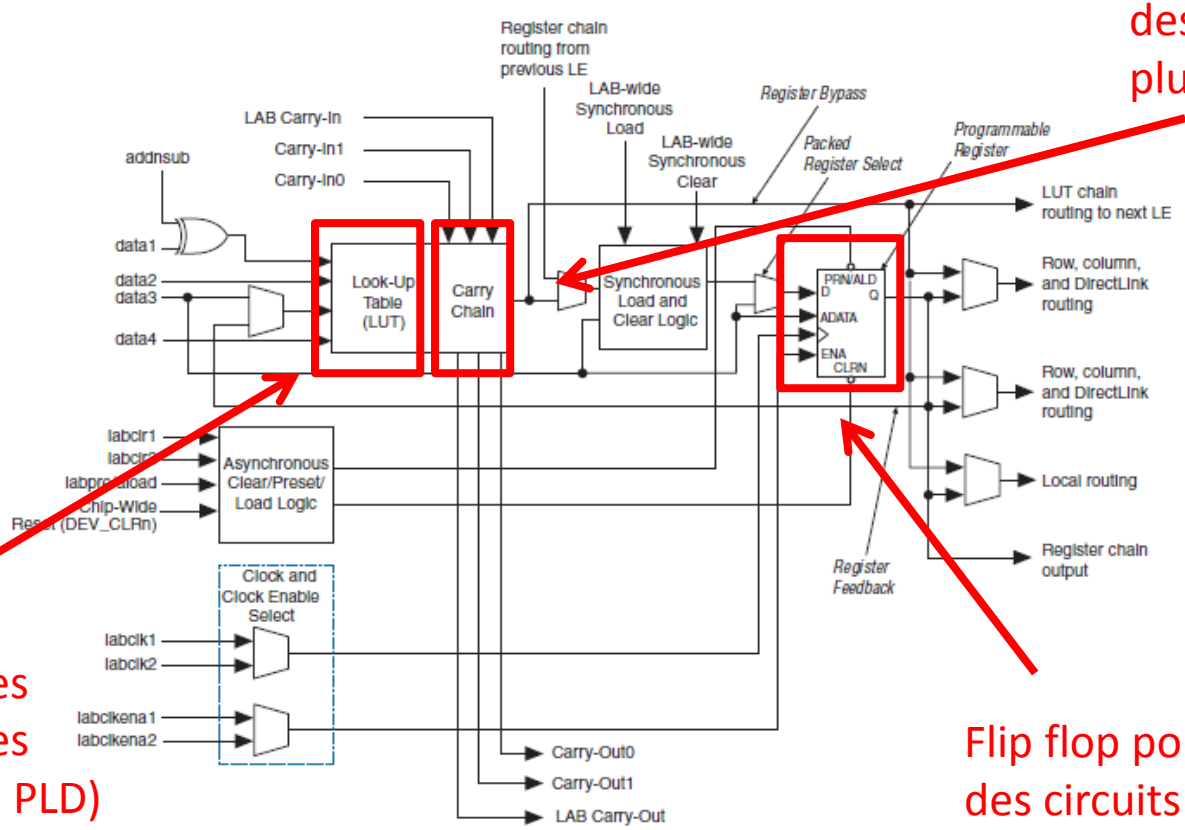
- Les CPLDs
  - Ont une structure plus distribuée et très régulière
  - Sont moins denses et moins chers
  - Utilisent une mémoire non-volatile pour la programmation




Architecture d'une MAXII d'Altera

# CPLD

- Chaque bloc ressemble a ceci:




Logique pour faire des additionneurs plus performants

LUT pour faire des fonctions logiques (C'est comme un PLD)

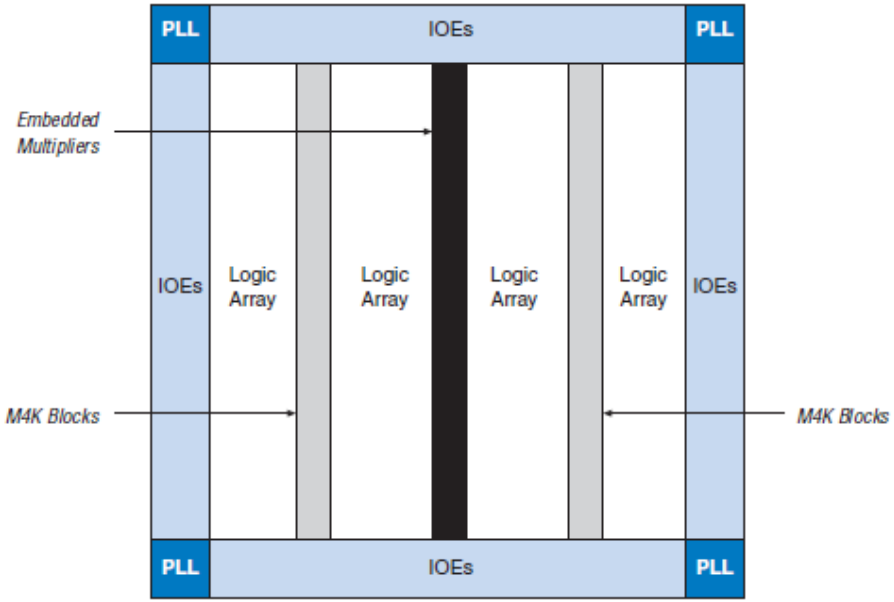
Flip flop pour faire des circuits séquentiels

# CPLD

- En interconnectant plusieurs de ces blocs, on peut faire des systèmes plus complexes
- Cependant, ce n'est pas le plus efficace
    - On reprogramme de la mémoire non-volatile: c'est plus long et ça réduit la durée de vie de la mémoire
- Pour des circuits plus complexes, on utilise des FPGAs
    - Ils sont plus efficaces mais aussi plus coûteux

# FPGA

- L'architecture d'un Cyclone II ressemble à ceci:
  - On remarque la logique, des multiplicateurs dédiés et de la mémoire dédiée
  - Les autres choses sont les entrées/sorties et les synchroniseurs d'horloge



# FPGA

- Chaque LOGIC ARRAY (diapo précedente) contient 16 éléments logiques comme celui-ci: