Architecture et Système

Stefan Schwoon

Cours L3, 2023/2024, ENS Cachan

Circuits logiques

Composants pour construire un ordinateur, physiquement réalisés par des transistors, se comportant comme des portes logiques.

Questions étudiées dans la suite:

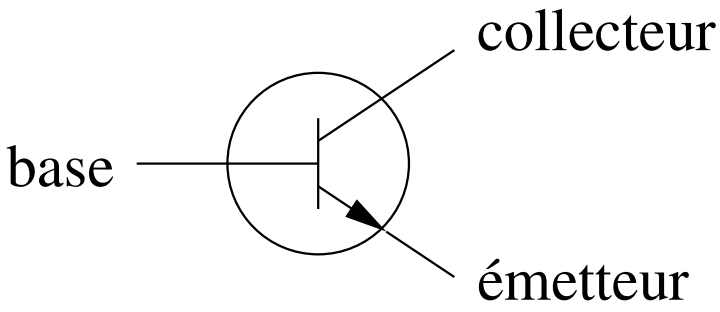
$\rightarrow$ Circuits statiques (entrée/sortie): fonctions logiques et arithmétiques

$\rightarrow$ Aspects dynamiques : stockage, mémoire, transfert de données, microprogrammation

Transistor

Découverte théorique 1925, utilisation dans les ordinateurs à partir des mi-1950s.

Il existe une grande variété de transitors. Exemple : transitor bipolaire NPN



Flux d'électrons possible entre émetteur et collecteur lorsque le voltage de base excède un certain seuil.

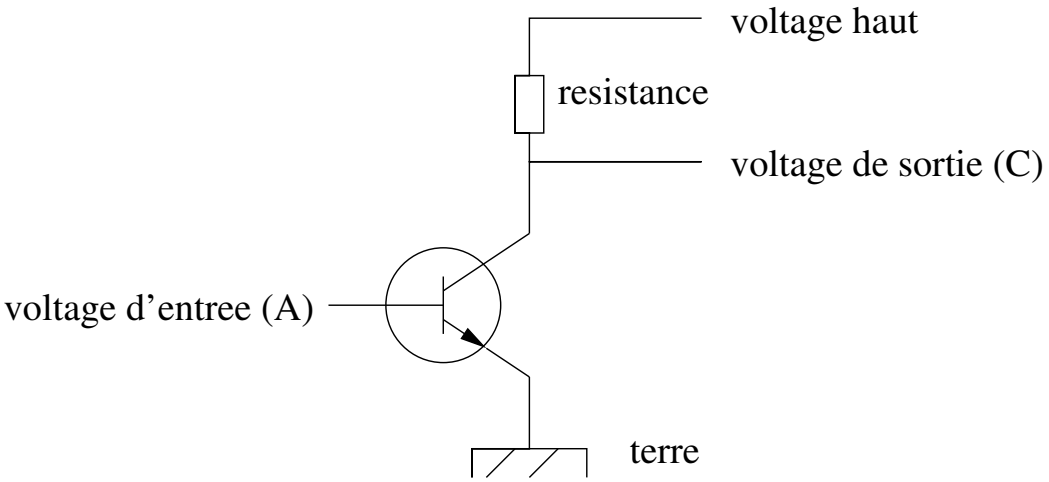
Transistor réalisant une manipulation logique

Le comportement du transistor permet de réaliser une logique binaire :

soit le voltage est en-dessous du seuil, alors le flux E$\rightarrow$C est interrompu;

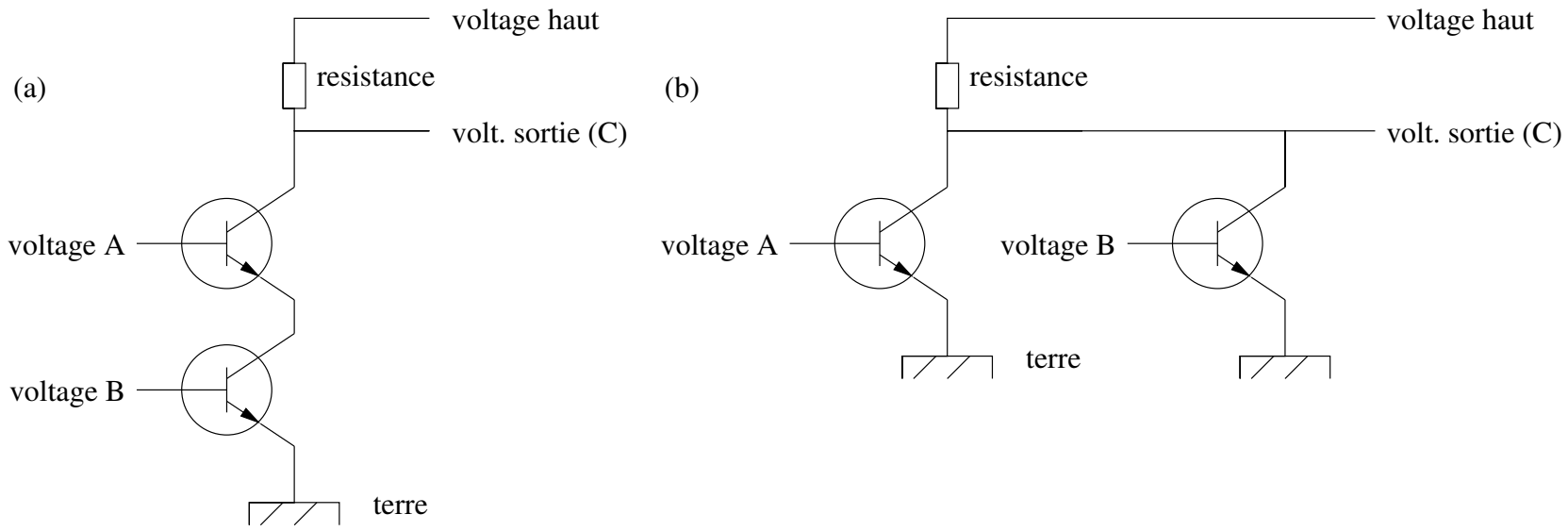
soit le voltage est en-dessus du seuil, alors il y a un flux.

Utilisation d'un transistor pour la négation logique :



Opérations binaires

Portes NON-ET (a) et NON-OU (b) réaliser avec des transistors:



Interprétation : voltage haut $\widehat{=}$ 1, voltage bas $\widehat{=}$ 0

Importance de NON-ET et NON-OU

Les opérateurs NON-ET et NON-OU (NAND/NOR en anglais) sont importants pour deux raisons :

Ils sont facilement réalisable avec des transistors.

Toute autre fonction logique peut être exprimé avec soit NON-ET, soit NON-OU :

$\neg A \equiv A \barwedge A$;

$A \wedge B \equiv (A \barwedge B) \barwedge (A \barwedge B)$;

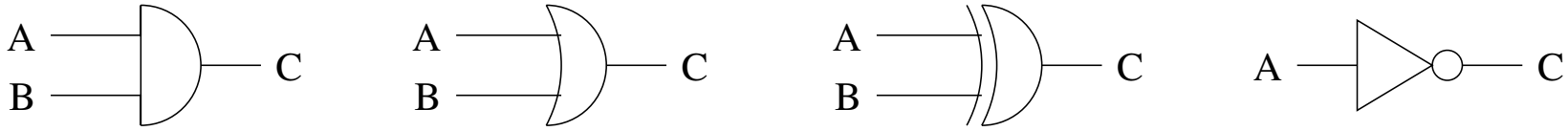
$A \vee B \equiv (A \barwedge A) \barwedge (B \barwedge B)$.

Désormais, pour faire abstraction des détails physiques, on représentera les circuits en forme de diagrammes avec des portes logiques qui traitent des bits avec valeurs 0 et 1.

Portes logiques

Diagrammes pour NON-ET (à gauche) et NON-OU (à droit):

Diagrammes derivés, réalise p.ex. par la combinaison de plusieurs portes NON-ET/OU.



ET OU XOR NON

Complexité des circuits

Dans un ordinateur, on traite des circuits logiques qui réalisent des fonctions assez complexes et avec beaucoup de bits en entrée (p.ex., l'addition sur des entiers de 32 ou 64 bits).

On s'intéresse donc à optimiser les circuits par rapport à :

leur taille – minimiser le nombre de transistors utilisés, c'est à dire le coût du circuit ;

leur profondeur – le chemin le plus long (en nombre de transistors) qu'un signal doit traverser détermine le délai pour calculer le résultat, compte tenu du fait que chaque transistor dispose d'un certain délai pour réagir aux changements du voltage en entrée.

Minimiser les deux au même temps - parfois contradictoire !

Mésures de complexité

Dans le suivant on s'intéresse à des fonctions avec n bits en entrée où n est variable (mais typiquement une puissance de 2).

On s'intéresse particulièrement au comportement des circuits quand n grandit, c'est à dire la *complexité asymptotique*.

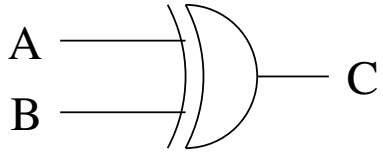
Objectif typique: taille $\mathcal{O}(n)$, profondeur $\mathcal{O}(\log n)$.

Remarques : l'analyse asymptotique nous permet certaines libertés en construisant les circuits.

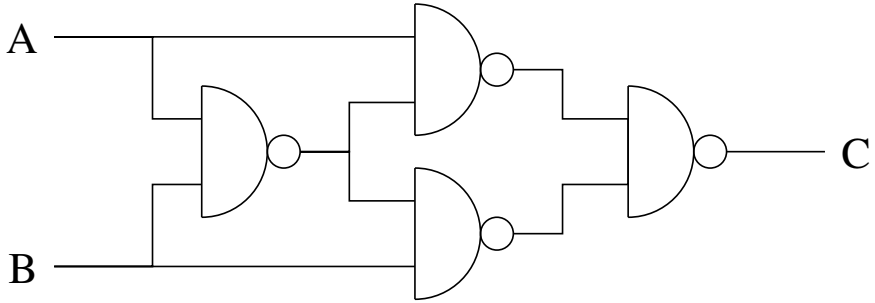
On se permettra des portes OU, ET, NON etc ; la complexité mésurée en nombre de transistors n'augmente que d'un facteur constant.

On peut même se permettre des portes ET/OU avec plus que deux valeurs en entrée, dans la mésure où le nombre d'entrées reste indépendant de n.

Exemple : réalisation de OU exclusif (XOR)



Réalisable par plusieurs portes NON-ET :



Premier diagramme: taille/profondeur 1

Deuxième diagramme: taille 4, profondeur 3

Nombre de transistors: 8

Fonctions arithmétiques: demi-additionneur

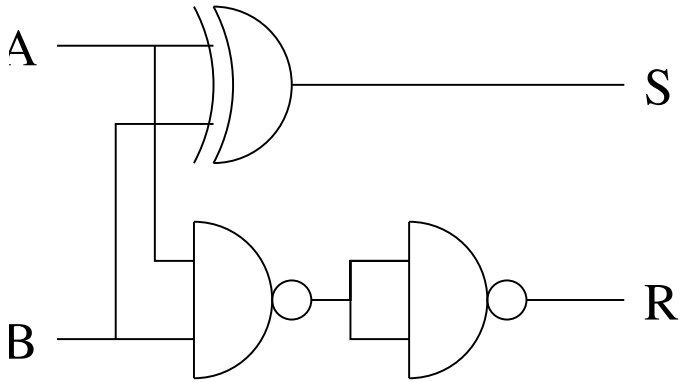
Fonction réalisée par un demi-additionneur:

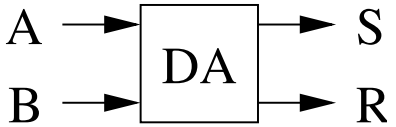
deux bits en entrée, A et B;

deux bits en sortie, R (la *retenue*) et S (la *somme*);

résultat souhaité : $(R.S)_2 = A + B$ (ou . dénote la concaténation)

Réalisation potentielle :



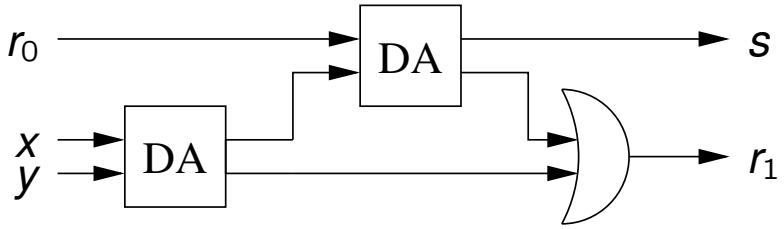


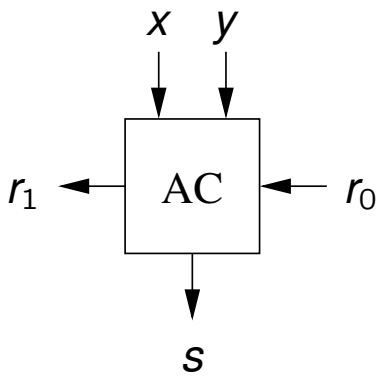
Additionneur complet

Un additionneur complet réalise l'addition de trois bits x, y et r_0, ou r_0 représente la retenue d'une autre addition.

Résultat souhaité : $(r_1.s)_2 = x + y + r_0$

Réalisation à l'aide des DA :



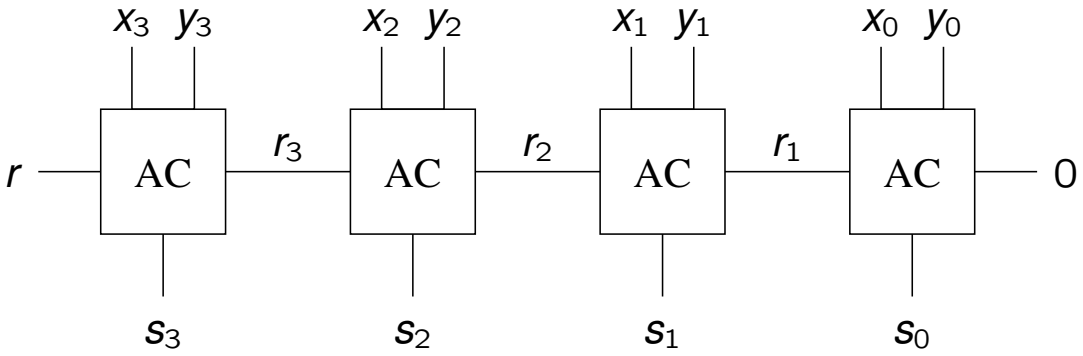


Addition de deux entiers

Supposons que nous avons deux entiers (non-négatifs) à deux bits :
$x = (x_3.x_2.x_1.x_0)_2$ and $y = (y_3.y_2.y_1.y_0)_2$.

On souhaite calculer $s = x + y$ sous la forme $s = (r.s_3.s_2.s_1.s_0)_2$.

Réalisation avec enchaînement de quatre AC :



La géneralisation du principe d'enchaînement à des vecteurs de n bits donnerait un circuit avec taille et profondeur $\mathcal{O}(n)$.

La profondeur est mauvaise car un ordinateur deviendrait deux fois plus lent si on augmente la taille des registres, p.ex. en passant de 32 à 64.

On va étudier une solution avec profondeur *logarithmique*.

Additioneur basé sur propagation/génération

Soit $0 \leq i \leq j < n$. On considère les bits aux positions i à j dans les vecteurs x et y en entrée :

> le bloc $i..j$ génère une retenue si $r_{j+1} = 1$ *indépendemment* des bits sur d'autres positions (on note $g_{i,j} = 1$);
>
> le bloc $i..j$ propage la retenue si $r_i = 1$ implique $r_{j+1} = 1$ (on note $p_{i,j} = 1$).

Du coup, $r_{j+1} = g_{i,j} \vee (r_i \wedge p_{i,j})$.

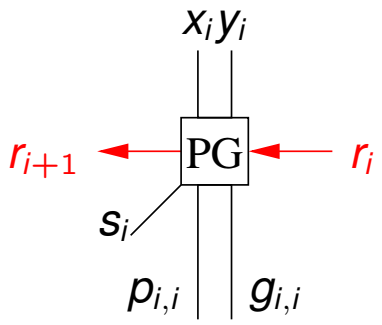
On peut calculer ces deux effets rien qu'avec $(x_i, y_i) \cdots (x_j, y_j)$!

Le cas $i = j$:

$g_{i,i} = 1$ ssi $x_i = y_i = 1$

$p_{i,i} = 1$ ssi $x_i + y_i \geq 1$.

Symbole pour un AC avec calcul des bits p/g en plus :



Le cas $i < j$:

Supposons que le bloc $i..j$ est découpé en deux blocs plus petits, $i..k$ et $k+1..j$, pour $i \leq k < j$.

$$g_{i,j} = g_{k+1,j} \vee (g_{i,k} \wedge p_{k+1,j})$$

$$p_{i,j} = p_{i,k} \wedge p_{k+1,j}$$

Symbole pour calculer la combinaison des bits p/g :

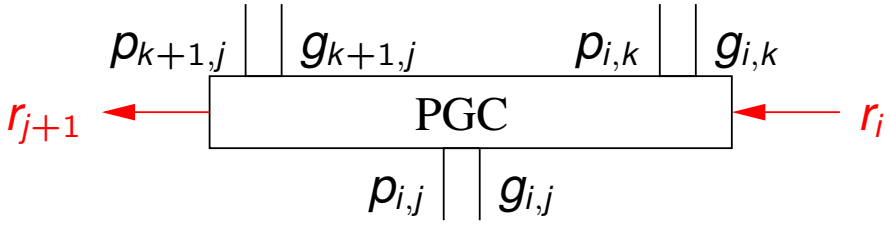
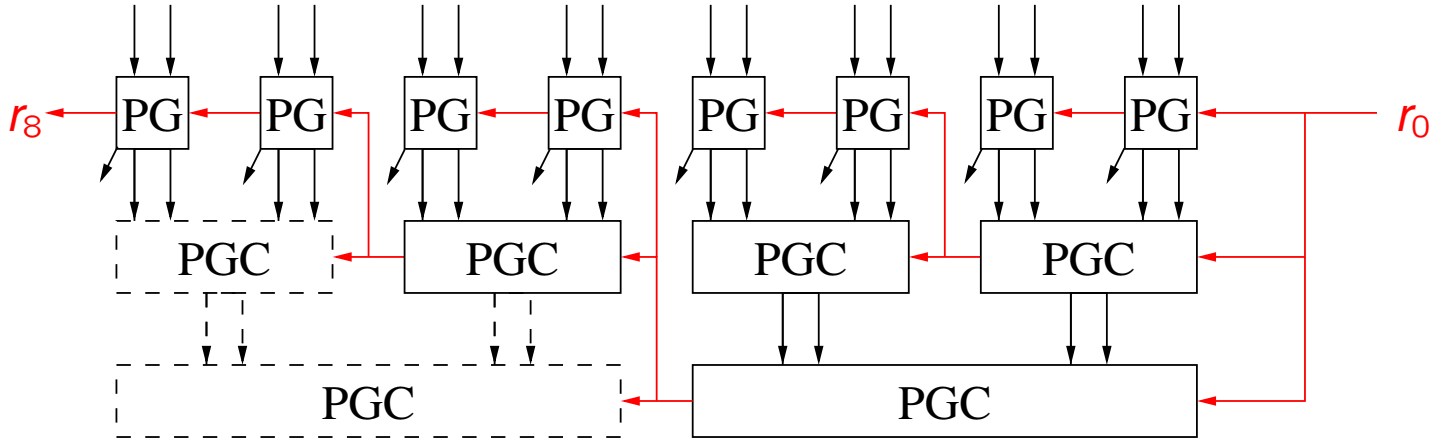


Diagramme simplifié pour $n = 8$, seule la propagation des retenue est détaillée :



Pour comprendre la profondeur du circuit, notons que

- le calcul des p/g est indépendant des valeurs de retenues ;
- la retenue d'un bloc PGC devient stable dès qu'on connaît les p/g et la retenue en entrée.

Du coup, le plus long chemin a une longueur d'environ $2 \log_2 n$.

Circuit pour multiplication

Soient $x = (x_{n-1} \cdots x_0)_2$ et $y = (y_{n-1} \cdots y_0)_2$ deux entiers naturels.

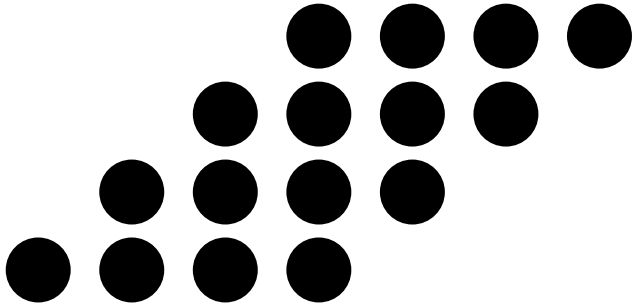
Construisons un circuit efficace pour $z = (z_{2n-1} \cdots z_0)_2$ tel que $z = x \cdot y$.

Technique de multiplication classique :

$$z = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (x_i \cdot y_j \cdot 2^{i+j})$$

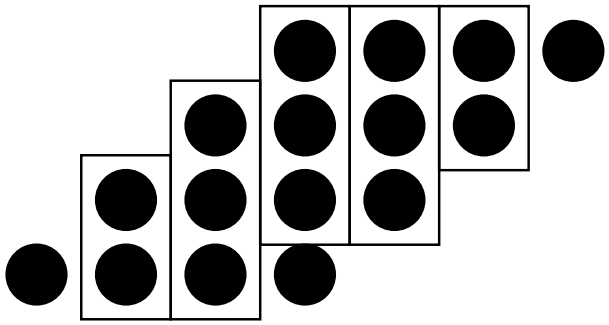
Notons que $x_i \cdot y_j \in \{0, 1\}$, du coup on obtient n^2 bits avec des “poids” différents.

Visualisation (pour $n = 4$), chaque boule représent un produit $x_i y_j$:

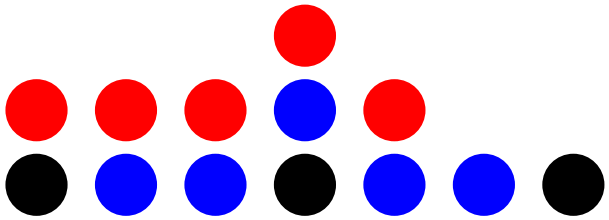


Ayant obtenu les n^2 bits, il faut donc faire l'addition dans les $2n$ colonnes (tout en propagant les retenues). Le nombre maximal de "boules" dans une colonne est de n.

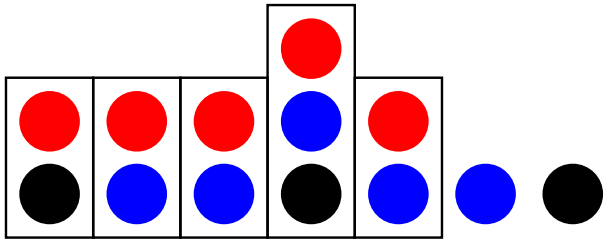
Principe : Successivement réduire les colonnes. On trouve des groupes de deux ou trois “boules” et on fait passer tous les groupes (en parallèle) par des DA ou AC.



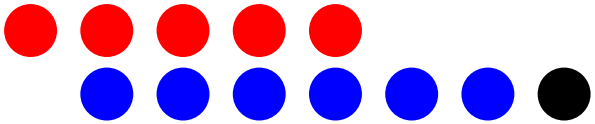
Ensuite on distribue les deux bits résultants (somme et retenue) aux bonnes colonnes :



Itération : identifier des groupes de deux ou trois . . .



. . . ce qui donne :



Ayant réduit le résultat tout les colonnes à deux, on fait l'addition comme avant.

Analyse du multiplicateur

Il faut n^2 portes ET pour obtenir les produits de départ.

Ensuite, on réduit la hauteur des colonnes par un facteur d'environ $2/3$ par itération.

> Une analyse précise montre qu'après $\log_{3/2} n$ itérations on obtient une hauteur d'au plus 4.
>
> Une fois la hauteur réduite à 4, trois itérations supplémentaires sont suffisantes.
>
> Du coup, la réduction se fait en profondeur $\mathcal{O}(\log n)$.

Le circuit pour l'addition finale est de taille $\mathcal{O}(n)$ et de profondeur $\mathcal{O}(\log n)$

Du coup, la multiplication entière se fait en taille $\mathcal{O}(n^2)$ et profondeur $\mathcal{O}(\log n)$.

Fonctions logiques sur les mots

Les ordinateurs permettent typiquement d'appliquer un opérateur logique sur tous les bits de deux mots en parallèle :

$$x \circ y = (x_{n-1} \circ y_{n-1}, \cdots, x_0 \circ y_0)_2$$

Exemples :

cas $\circ = \wedge$: souvent utilisé avec des constants, p.ex. $x \wedge 16$ sert à isoler le 4ème bit de x.

cas $\circ = \vee$: fait l'union des bits qui sont 1, p.ex. $x := x \vee 16$ met le quatrième bit de x à 1.