Arithmétique binaire

10 novembre 2024

Objectif

Comprendre comment est effectuée
une multiplication entière entre deux codes binaires

Décalage, multiplication et division

$a = [a_{n-1}, \dots, a_0]$ sur n bits $A' = [a_{n-1}, \dots, a_0, 0]$ sur $n+1$ bits

- **Codage entiers naturels**

$$\text{valeur(A)}= \sum_{i=n-1}^{0} b_i \times 2^i$$

$$\text{valeur(A')}= \sum_{i=n}^{1} b_i \times 2^i = 2\times \text{ valeur(A)}$$

Décalage vers les poids forts avec 0 en a_0 $\rightarrow$ multiplication par 2

- **Codage signe+valeur absolue**

$$\text{valeur(A)}= (-1)^{a_{n-1}}. \sum_{i=n-2}^{0} b_i \times 2^i \text{ et}$$

$$\text{valeur(A')}= (-1)^{a_n}. \sum_{i=n-1}^{1} b_i \times 2^i = 2\times \text{ valeur(A)}$$

Décalage, multiplication et division

- **Codage avec biais** (biais)

$$\sum_{i=n-1}^{0} b_i \times 2^i = \text{valeur(A)+biais}$$

$$\sum_{i=n}^{1} b_i \times 2^i = 2\times\ (\text{valeur(A)+biais}) = 2\times\ \text{valeur(A)}+2.\text{biais}$$

avec biais multiplié par 2 (ok seulement pour biais $= 2^{n-1}$)

- **Codage en ca2**

$$\text{valeur(A)} = -2^n + \sum_{i=n-1}^{0} b_i \times 2^i$$

$$\text{valeur(A')} = -2^{n+1} + \sum_{i=n}^{1} b_i \times 2^i = 2\times\ \text{valeur(A)}$$

Synthèse pour une opération de décalage

- Décalage d'un code d'une position vers les poids forts
  $\rightarrow$ multiplier la valeur par 2 (biais compris)
- Résultat sur $n + 1$ bits
- Remarque : vers les poids faibles $\rightarrow$ division par 2

- Généralisé à p décalages
- Multiplication par 2^p résultat sur $n + p$ bits
- $p < 0 \rightarrow$ division entière par 2^p (en général sur n bits)

Quelques exemples

$A = 1101$, envisageons l'effet d'un décalage de 2 positions : $A' = 110100$

- **Entier naturel**

  valeur(A)$=2^3 + 2^2 + 2^0 = 8 + 4 + 1 = 13_{10}$
  valeur(A')$=2^5 + 2^4 + 2^2 = 32 + 16 + 4 = 52_{10}$

- **Entier négatif** (signe +valeur absolue)

  valeur(A)$=-(2^2 + 2^0) = -5_{10}$
  valeur(A')$=-(2^4 + 2^2) = -20_{10}$

- **Entier biaisé** (biais=8)

  valeur(A)$=2^3 + 2^2 + 2^0 - 8 = 8 + 4 + 1 - 8 = 5_{10}$
  valeur(A')$=2^5 + 2^4 + 2^2 - (8 \times 4) = 32 + 16 + 4 - 32 = 20_{10}$

- **Entier en ca2**

  valeur(A)$=-2^4 + 2^3 + 2^2 + 2^0 = -16 + 8 + 4 + 1 = -3_{10}$
  valeur(A')$=-2^6 + 2^5 + 2^4 + 2^2 = -64 + 32 + 16 + 4 = -12_{10}$

Algorithme de multiplication entière (base 10)

Multiplication faite "à la main" sur des nombres à 3 chiffres
$A = a_2 a_1 a_0 = 123_{10}$ à multiplier par $B = b_2 b_1 b_0 = 256_{10}$

$$
\begin{array}{cccccc|l}
 & & & 1 & 2 & 3 & \\
\times & & & 2 & 5 & 6 & \\
\hline
 & & & 7 & 3 & 8 & (123 \times 6 \times 10^0) \\
+ & & 6 & 1 & 5 & 0 & (123 \times 5 \times 10^1) \\
+ & 2 & 4 & 6 & 0 & 0 & (123 \times 2 \times 10^2) \\
\hline
= & 3 & 1 & 4 & 8 & 8 &
\end{array}
$$

Repose sur 2 opérations (en plus de l'addition)

1. la multiplication par b^p (p décalages d'un nombre)
2. multiplication digit par digit (table de multiplication)

Eléments de la multiplication entière en base 2

- Addition de deux nombres : déjà vu
- Décalage de p rangs pour multiplication par 2^p : déjà vu
- Table de multiplication :

a	b	$a \times b$
0	0	0
0	1	0
1	0	0
1	1	1

D'où la forme booléenne de la multiplication de deux bits

$$a \times b = a.b$$

Algorithme de multiplication entière en base 2

Calcul de $C = A \times B$ sur nbits (résultat sur $2 \times \ n$ bits)

$C \leftarrow 0$
while $(B \neq 0 \ldots 0)$ **do**
  if $b_0 = 1$ **then**
    $C = C + A$
  end if
  $A = 2 * A$
  $B = B\%2$ (division entière : décalage vers poids faibles)
end while

Exemple de multiplication binaire

$A = 1010\ (10_{10})$ et $B = 1001\ (9_{10})$ D'où $C = A \times B = 90_{10}$

1. $B = 1001 \neq 0\ \ b_0 = 1 \Rightarrow C \leftarrow 0 + A = [00001010]$
   $B \leftarrow [100]$ et $A \leftarrow [10100]$
2. $B = 100 \neq 0\ \ b_0 = 0 \Rightarrow C$ inchangé
   $B \leftarrow [10]$ et $A \leftarrow [101000]$
3. $B = 10 \neq 0\ \ b_0 = 0 \Rightarrow C$ inchangé
   $B \leftarrow [1]$ et $A \leftarrow [1010000]$
4. $B = 1 \neq 0\ \ b_0 = 1 \Rightarrow$
   $C \leftarrow [00001010] + [01010000] = [01011010]$
   $B \leftarrow [0]$ et $A \leftarrow [10100000]$
5. $B = 0$ sortie de la boucle **while**

On a bien : $[01011010] = 2^6 + 2^4 + 2^3 + 2^1 = 64 + 16 + 8 + 2 = 90$

Synthèse de cette technique algorithmique

- Simple à concevoir car principe utilisé en base 10
- Utilise des opérations tout à fait réalisables en binaire
- Boucle **while** non combinatoire (séquentielle : vu en S6 DES)
- Temps de calcul =

$$n \times \big(2 \times \text{durée décalage} : 2 \times \delta_t$$
$$+\text{durée addition } n \text{ bits} : 3 \times \delta_t$$
$$+\text{durée test } (B = 0) : \delta_t \big)$$

Soit une durée de $n \times 6 \times \delta_t$ pour des mots de n bits

Exemple : pour $n = 64$ bits : durée multiplication $= 384 \times \delta_t$

D'où des approches pour calculer les expressions des bits du résultat (câblage de l'opération)

Approche combinatoire pure

Faire la multiplication de mots de n bits revient à construire une table de vérité à $2 \times n$ entrées soit à $2^{2 \times n}$ lignes.

n	nb lignes
1	4
2	16
3	64
4	256
...	...
64	$\approx 3 \times 10^{38}$

Totalement irréaliste au delà de n=2 ou 3

Faisons l'exercice pour $n = 2$

Câblage d'un multiplieur 2 bits x 2 bits

$$A = [a_1 a_0] \text{ et } B = [b_1 b_0],\ C = A \times B = [c_3 c_2 c_1 c_0]$$

a_1	a_0	b_1	b_0	Produit $_{10}$	c_3	c_2	c_1	c_0
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0
0	1	0	0	0	0	0	0	0
0	1	0	1	1	0	0	0	1
0	1	1	0	2	0	0	1	0
0	1	1	1	3	0	0	1	1
1	0	0	0	0	0	0	0	0
1	0	0	1	2	0	0	1	0
1	0	1	0	4	0	1	0	0
1	0	1	1	6	0	1	1	0
1	1	0	0	0	0	0	0	0
1	1	0	1	3	0	0	1	1
1	1	1	0	6	0	1	1	0
1	1	1	1	9	1	0	0	1

$a_1 a_0 \backslash b_1 b_0$	00	01	11	10
00	0	0	0	0
01	0	1	1	0
11	0	1	1	0
10	0	0	0	0

$c_0 = a_0 . b_0$

$a_1 a_0 \backslash b_1 b_0$	00	01	11	10
00	0	0	0	0
01	0	0	1	1
11	0	1	0	1
10	0	1	1	0

$c_1 = a_1 . \overline{b_1} . b_0 + a_0 . b_1 . \overline{b_0} + a_1 . \overline{a_0} . b_0 + \overline{a_1} . a_0 . b_1$

$a_1 a_0 \backslash b_1 b_0$	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	1
10	0	0	1	1

$c_2 = a_1 . \overline{a_0} . b_1 + a_1 . b_1 . \overline{b_0}$

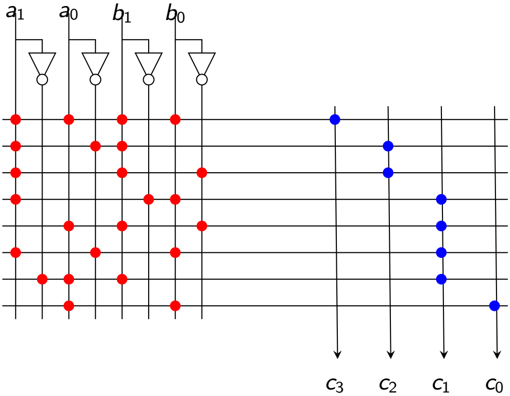
$a_1 a_0 \backslash b_1 b_0$	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	1	0
10	0	0	0	0

$c_3 = a_0 . a_1 . b_1 . b_0$

pas de difficulté conceptuelle, mais laborieux

Temps de calcul $3 \times \delta_t$ (comme l'addition)

Mise en œuvre du multiplieur 2 bits



Multiplieurs n bits

- Conception d'un multiplieur **très calculatoire** avec TV et TK
- Autre approche : sommes partielles (multiplieurs 2 bits)

$$
\begin{array}{cccc}
 & & a_1 & a_0 \\
 & \times & b_1 & b_0 \\
\hline
 & & a_1.b_0 & a_0.b_0 \\
+ & a_1.b_1 & a_0.b_1 & \\
\hline
= & & & sp_{0.0} \\
= & sp_{1.1} & sp_{1.0} & \\
= & sp_{2.0} & &
\end{array}
$$

avec (revoir additionneur 2 bits)

$$
\begin{cases}
sp_{0.0} & = a_0.b_0 \\
sp_{1.0} & = (a_1.b_0) \oplus (a_0.b_1) \\
sp_{1.1} & = a_1.b_0.a_0.b_1 \\
sp_{2.0} & = a_1.b_1
\end{cases}
$$

- Ramène le problème de multiplication au calcul de sommes

- Calcul des sommes de sommes partielles

				$sp_{0.0}$
$+$		$sp_{1.1}$	$sp_{1.0}$	
$+$		$sp_{2.0}$		
$=$	c_3	c_2	c_1	c_0

- Il ne reste qu'à écrire, puis simplifier :

$$\begin{cases} c_0 & = sp_{0.0} = a_0.b_0 \\ c_1 & = sp_{1.0} = (a_1.b_0) \oplus (a_0.b_1) = a_1.\overline{b_1}.b_0 + a_0.b_1.\overline{b_0} + a_1.\overline{a_0}.b_0 + \overline{a_1}.a_0.b_1 \\ c_2 & = sp_{1.1} \oplus sp_{2.0} = (a_1.b_1) \oplus (a_1.b_0.a_0.b_1) = a_1.b_1.\overline{b_0} + a_1.b_1.\overline{a_0} \\ c_3 & = sp_{1.1}.sp_{2.0} = a_1.b_1.a_1.b_0.a_0.b_1 = a_1.b_0.a_0.b_1 \end{cases}$$

- Bien plus accessible qu'avec table de vérité et tables de Karnaugh
- Simplification algébrique $\Rightarrow$ outil de conception !
  (Intel® Quartus® Prime par exemple)

Solution câblée

Temps de calcul $= 3 \times \delta_t$ **quel que soit le nombre de bits** 🙂

Conclusions et extension aux nombres entiers relatifs

- **Signe + valeur absolue**
  - calcul sur $n-1$ bits
  - bit signe $= b_{n-1} \oplus a_{n-1}$
- **Biais**
  -$(A + biais) \times (B + biais) = A \times B + (A + B + biais) \times biais$
  - ne se prête pas à la multiplication
- **ca2**
  - A et B positifs : pas de problème
  - A négatif : $A \times B = (2^n - |A|) \times B \neq 2^{2n} - (|A| \times B)$
    1. calcul sur valeurs absolues
    2. bit de signe $= b_{n-1} \oplus a_{n-1}$
    3. si bit de signe =1 , ca2 (résultat)